Developing Object-Oriented Software

An Experience-Based Approach

IBM Object-Oriented Technology Center

To join a Prentice Hall PTR internet mailing list, point to

http://www.prenhall.com/register

Prentice Hall PTR
Upper Saddle River, New Jersey 07458
http://www.prenhall.com

chitecture

Apple Events
CodeCheck
Freelance Graphics
Hypercard
Macintosh
Microsoft Project
Novell
ODMG
OMT
OpenDoc
Paradigm Plus
ProLint
Rational Rose
StateChart
SuperProject
VisualBasic
VisualSmalltalk
Windows
Word Pro

1.0 Introduction

There are numerous critical success factors associated with the use of object technology for software development. These include, but are not limited to:

- Management support for the use of object technology in a particular project
- Assessing team needs for training in object technology and supporting tools and making the needed investments in training and tools
- Ensuring that the development team has access to the knowledge of experienced practitioners of object-oriented software development either as team members or as mentors
- Understanding of the object-oriented software development life cycle by all members of the project's management, technical, and support teams
    - Understanding the differences between object-oriented development and whatever approach to software development is currently being used in the organization
    - Understanding the impacts that those differences have on the steps used in building software
- Understanding the artifacts that need to be produced at each phase of the development life cycle in order to support successful use of object technology
    - Understanding the various choices of methods and techniques that exist for producing these artifacts and how to select the approach best suited to a particular project

This book focuses on the critical areas of "understanding the object-oriented software development life cycle" and "understanding the artifacts that need to be produced." It presents the approach to object-oriented software development that the IBM Object-Oriented Technology Center (OOTC)[1] uses and advocates in its day-to-day activities supporting object technology use in IBM.

The OOTC has been in operation since 1992 with a mission to provide information and support on the use of object technology within many of IBM's software development laboratories. The three primary means of providing this support are:

1. Mentoring IBM software development projects on how best to apply object technology to their projects
2. Providing short-term assistance such as technical presentations and seminars, pointers to technical information, and design and code reviews

[1] The IBM OOTC and object technology centers are described in [Korson96].

3. Developing documents that provide information on various aspects of object technology and its usage.

The OOTC is staffed by people with expertise and experience in the use of object technology and has supported the use of object technology in 70 projects at 20 IBM locations worldwide. It has produced 15 internal documents on various aspects of object technology and its usage and more than 22,000 copies of those documents are in circulation. The OOTC has also provided short-term assistance to over 2,000 people.

This book is a direct product of that body of work and of the collective experience of the OOTC's members. It is reflective of our current understanding of the approaches to object-oriented software development that have been most successful for us and our clients.

The approach presented in this book was developed over a period of five years and has evolved to its present state on two paths:

1. Evolution: This book presents an approach to object-oriented software development at which OOTC members have arrived at philosophical agreement despite some differing preferences on specific object-oriented methods and techniques. The approach has been documented and enhanced through numerous cycles of use in OOTC mentoring engagements and solicitation of feedback from readers.
2. Genealogy: This approach is based on the aspects of many other methods and approaches to object-oriented software development that we have found to be of the most practical value in our work with object-oriented development projects.

These paths are discussed in the following three sections.

1.1 THE EVOLUTION OF THIS APPROACH

The mentors who comprise the OOTC, typically about 14 in number, are recruited from throughout the IBM software community. The staff is diverse in terms of professional background and areas of object technology expertise and has included members with backgrounds in research, marketing and services, tools development, product development, and education and training with expertise in a variety of domains including operating systems, engineering software, class library development, networking software, and databases.

The staff's areas of object technology expertise have covered areas such as analysis, design, C++, Smalltalk, Java, metrics, testing, reuse, tools, databases, and object request brokers.

The customer set that the OOTC exists to support, IBM's Software Development Laboratories, is also widely diverse in terms of problem domains, approaches to software development, development tools and platforms, target platforms, favorite object methodologies and notations, and object technology skills.

Further, in the early days of the OOTC, object technology was evolving rapidly and there were few standard approaches, processes, methods, or tools.

When the OOTC began providing mentoring services, not much was thought was given to the potential problems of diverse mentors and clients working with an evolving technology. The advice that the OOTC's mentors gave their clients during engagements was based on "what they knew" and was not formed into a consistent, repeatable documented approach.

This was because while each OOTC mentor had developed a high degree of expertise and experience relative to their clients, it was typical that this expertise and experience was different. Typically, two mentors would be assigned to an engagement and it was common that they had not worked with each other previously. This meant that either there was no common ground from which to work, or that any common ground that might exist was not readily apparent.

This approach worked for the OOTC and its clients during the OOTC's first year, but it was clear that this approach would not be adequate for long. There were several reasons for this:

1. The OOTC's clients began to demand documentation of the advice and guidance that they were receiving in the mentoring sessions. There was a need for documents that the clients could refer to in the absence of the OOTC's mentors.
2. The client base was becoming more sophisticated and experienced. Object technology was growing in usage and importance inside IBM and the knowledge and expectations of the OOTC clientele were growing with it. The clients were beginning to expect well-thought out answers to increasingly complex questions. Additionally, many of the projects had made preliminary choices of methods and notations that the OOTC needed to work with. So, the OOTC's advice and guidance on best practices for object-oriented software development needed to fit well with these differing notations and methods in order to be effective.
3. Consistency was a problem. Sometimes a project follow-up visit might have to be performed by different mentors because of scheduling conflicts. A mentor new to a project would often be puzzled by what they found their predecessors had advised.
4. There was no basis for quality control of the OOTC mentoring offering. Because each mentor used their own particular approach, and the mentoring process was not documented, it was difficult to judge what approaches were meeting with the most success.

The OOTC team agreed that to continue to be successful it was important to reach some consensus on the approach to be used in mentoring engagements and on the important advice and guidance that the OOTC was giving its clients. The benefits expected from this were:

1. A consistent and repeatable approach. This would provide a baseline, agreed approach from which change could be discussed and negotiated.

2. A documented approach with documented advice and guidance. This would allow mentoring clients to have a reference guide to use when OOTC mentors were not at hand.

3. A handbook for OOTC mentors. It would provide a reference document for new OOTC mentors to master, support, and help evolve.

4. A vehicle for exchanging approaches, experiences, successes, and failures amongst OOTC members. This would be critical for creating a cohesive team of mentors and for ensuring that the overall process could continue to evolve in a positive direction.

Reaching consensus was not easy. This was largely due to the fact that each OOTC mentor had strongly held opinions about the relative merits of their favorite methods, approaches, and techniques.

The starting point for attempting to develop consensus was for each OOTC mentor to present the team with an overview of their approach to object-oriented development including examples of the artifacts that they normally produced throughout the development cycle. At the end of this exercise it was clear that consensus was a long way off.

It was soon observed, however, that, while the flow through the development process and the specific techniques used to create artifacts were different, the artifacts (or work products) themselves were quite similar. For example, everyone was using a Rumbaugh-like Object Model and producing something similar to Jacobson's Object Interaction Diagrams. This insight caused the team to begin to focus on what they produced rather than how they produced it. The "what" turned out to be fairly standard.

The team also considered two particular OOTC mentoring engagements that were very successful in their use of object technology. These completely different projects had some strong similarities in approaches: They produced similar work products at the same phases of the development life cycle; they were dedicated to maintaining these work products; and they consolidated the work products into a project workbook.

This led the us to concentrate on work products in the context of a project workbook. It was decided that an approach that standardized on work products but allowed for individual choices on particular techniques used to produce work products was the best fit for the OOTC. It would help us to gain the benefits of a standardized approach while not sacrificing the strengths that each mentor possessed in applying individual techniques.

Thus this book began to take the form of work products described in the context of a workbook, accompanied by a tool kit of techniques producing those work products.

It soon became apparent, though, that what began as a necessity had turned into a virtue. Providing mentoring within the context of a well-defined project workbook structure turned out to be a good decision independent of OOTC history. Focusing on work products instead of on process permits standardization and structure without sacrificing the flexibility demanded by projects that typically differ very widely and many of which have already chosen particular methods, notations, and techniques.

Since this approach was first documented in early 1995, it has been enhanced regularly based on feedback derived from its usage:

- In OOTC mentoring engagements
- By other object-oriented projects inside IBM
- As the basis for material presented in IBM education courses on object-oriented software development
- By IBM marketing and services personnel in engagements with IBM customers
- By more than 2,000 IBM employees as a general reference guide on object-oriented development

These enhancements include:

- Improvement of work product descriptions and advice and guidance
- The addition of new work products which were found to be useful
- The addition of techniques that had proven useful in the OOTC mentoring practice
- Crisper definition of the object-oriented development life cycle that works best with this approach
- The addition of a complete case study that demonstrates how the work products described in this approach flow from phase to phase in the development life cycle.

So, the evolution of this approach did not end with the first attempt at defining, documenting, and using it successfully. Rather, it has been continuously improved since then through a process of use, feedback, and enhancement.

1.2 THE GENEALOGY OF THIS APPROACH

This approach is not identical to any published method such as those described in books by James Rumbaugh, Grady Booch, or Ivar Jacobson, although it borrows concepts and notations from these and others. It has been our experience and observation that mature practitioners of object-oriented software development do not follow any standard method exactly as it is published. Most seasoned veterans of object-oriented development wind up hybridizing their approach—they use what they find works from a base method and extend it by borrowing from other approaches and experiences that they find useful. This was the case with the OOTC, as is shown in Figure 1-1. This figure shows some of the influences of the approach described in this book.

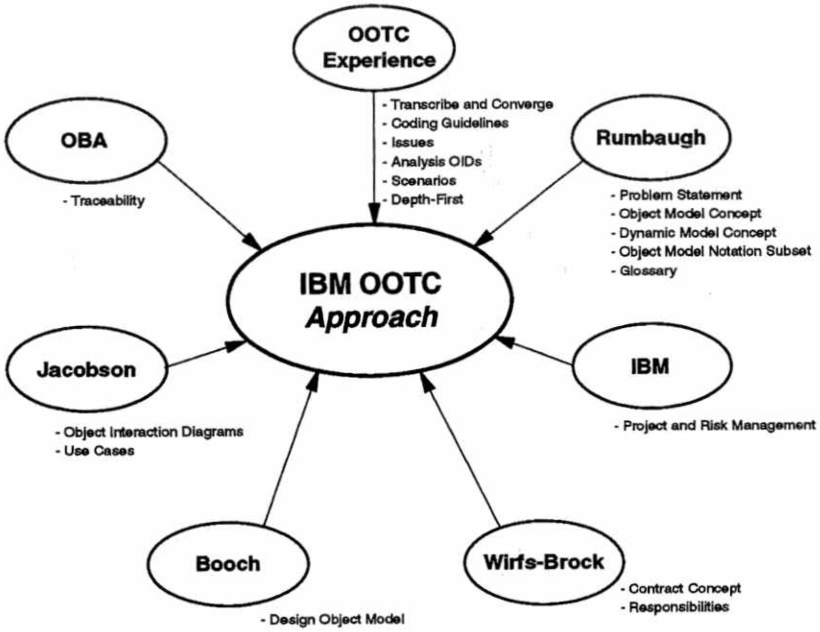


Figure 1-1. *Sources of the IBM OOTC Approach.*

The methods referenced above are considered to be "established methods." (Sometimes called First Generation Methods.) Hybrids, such as Fusion [Coleman94], [Malan96] or SOMA [Graham95] (sometimes referred to as "Second Generation Methods" in the literature) have combined ideas from more than one of the established methods. There are a whole range of emerging methods evolving at this time in the object-oriented community (for example, KISS [Kristen94], BON [Walden95], OORAM [Reenskaug96], ROOM [Selic94], et cetera). As they become better known and more experience is gained, they will undoubtedly influence the evolution of our approach.

We have incorporated various aspects, such as techniques and notations, from long established and second generation methods. As Figure 1-1 shows, our approach has not been developed from scratch, but incorporates several aspects from other methods, such as notations and various work products for requirements, analysis, and design.

1.2.1 Notation

The Object Modeling Technique (OMT) [Rumbaugh91a] is one of the more popular methods in the object technology field. It is well adapted within the community; it has extensive tool support; and there are many training courses available. Several books and articles ([Derrer95], [D'Souza95], [D'Souza94], [Rumbaugh91b], et cetera) discuss modeling with OMT. For these reasons we decided to adapt the OMT notation style as a base for our notation.

In our experience OMT strengths are:

- Flexibility and extensibility of the method
- Easy to learn and somewhat intuitive
- Strong treatment of relationships.

Having said that, the key messages of this book are entirely independent of notation, and the ideas that we present in this book can be used with other notations, for example Grady Booch, James Rumbaugh, and Ivar Jacobson's "Unified Modeling Language" notation [Booch96].[2] This is also confirmed by our own experience where approximately 40 percent of the projects we have been involved in have used other notations such as Booch, and Shlaer and Mellor).

1.2.2 Work Products

Many of our requirement work products are taken straight from standard software development practices (such as *Business Case* and *Acceptance Plan*). The same is true of our project management work products (*Resource Plan*, *Schedule*, et cetera). These have been included in this book for completeness, and not because they are novel.

The *Glossary* work product is maintained through all development phases beginning with requirements gathering. It is based on OMT's Data Dictionary ([Rumbaugh91a], pages 156-157).

The Object Behavior Analysis (OBA) approach ([Gibson90], [Rubin92], [Wirfs-Brock92]) has as its primary emphasis the modeling, representation, and communication of the requirements of a system through the use of a consistent vocabulary and by maintaining full traceability of the resultant artifacts to the stated business goals and objectives [Rubin94]. OBA includes an interesting feature called a *traceability model.* This model shows the interdependencies of the various work products. The traceability concept of our approach is based on this model.

[2] The Unified Modeling Language is a merging of the models and notation from the Booch, OMT, and Objectory methods by Grady Booch, James Rumbaugh, and Ivar Jacobson.

In common with many other emerging object-oriented development approaches, we make use of the great contribution to the field made by Ivar Jacobson, namely Use Cases. A Use Case is defined as "... a special sequence of transactions in a dialogue between a user and the system. Each use case is thus a specific way of using the system. A Use Case may have one basic course and several alternative courses." ([Jacobson92], page 510). Our use of Use Cases differs from the above definition. Our Use Cases are not sequences of transactions but statements of externally visible system behavior. We refine Use Cases into *Scenarios* each of which elaborates a Use Case with a set of assumptions and a set of outcomes. Each of our Scenarios may have an Object Interaction Diagram (OID) that shows graphically in terms of interactions between objects how a particular Object Model can support its Scenario. Ivar Jacobson incorporates the function of all three of these work products: Use Case, Scenario, and OID into his concept of Use Cases. There is scope, of course, for both sets of ideas to be used in parallel, but we believe that our approach is frequently helpful, as it encourages developers to focus on one concern at a time. The idea of a Scenario as we have defined it, in terms of assumptions and outcomes, turns out to be exactly what is required when system behavior has to be defined rigorously. The idea of defining behavior in terms of assumptions (preconditions) and outcomes (postconditions) comes from software engineering. This view has also been used by Syntropy [Cook94].

We have found state modeling to be a useful means of gaining insight into the lifecycle of objects (as in [Rumbaugh91a] and [Shlaer92]), although applied on a very selective basis.

The static aspects of our approach are specified using an entity centric object model (such as OMT or Booch), with modeling constructs such as classes, attributes, and associations. The *Analysis Object Model* and *Design Object Model* work products are based on OMT's object model concept [Rumbaugh91a], [Rumbaugh95a]. The *Design Object Model* is augmented with directionality and many of Grady Booch's adornments [Booch94].

In the Unified Modeling Language, a notation framework is introduced to present the design information "for an object-oriented system under construction" [Booch96]. The notation covers the modeling aspects for static class relationships, use cases, message tracing, state modeling, and system organization. Our approach is an enhancement of OMT (and by the way these aspects have not been resolved in the Unified Modeling Language) by (1) simplifying the notation for the practical usage, (2) adding development processes and emphasizing how to utilize the notation and modeling in each development phase, and (3) separating the notation for the business modeling (object-oriented analysis) that is independent of technology from the one for solution modeling (object-oriented design) that is dependent of underlying technology.

OMT's Dynamic Model Concept [Rumbaugh91a], [Rumbaugh95b], and Jacobson's Interaction Diagrams [Jacobson92] have leveraged our dynamic modeling work products: *Object Interaction Diagrams* and *State Models* at both the analysis and design levels of abstraction.

From the Responsibility-Driven Design (RDD) method by Rebecca Wirfs-Brock ([Wirfs-Brock89]) we have incorporated the concept of responsibilities during analysis and RDD's Contract concept in our *Subsystem Model*. In RDD the examination of collaborations and responsibilities is the central part of the method. RDD regards an application very much in a client-server fashion where objects collaborate and provide services for each other. These services are described in a contract between the service provider and a client requesting the service.

Our *Subject Areas* work product is inspired by Sally Shlaer and Steve Mellor's Domains [Shlaer92] and Peter Coad's Subjects [Coad90]. Both Domains and Subject Areas are mechanisms for permitting developers to focus on one logically independent topic at a time. Subject Areas differ from Domains (and Subsystems), however, because they are identified at analysis time. Subject Areas partition the *problem domain* that relates directly to the business problem and that is the subject matter of analysis.

Much of our project management work products derive from the project management techniques, processes, and approaches that have been long employed by software development groups inside IBM. Some have been modified for use with an object-oriented approach. Some are used in much the same way as they have always been. This is one of the strengths of the approach as far as our ability to have it accepted by the managers and technical leads of project development groups. They don't have to throw away everything they have done in the past and, in particular, in the important area of project management; much of what they have done in the past can still be effectively employed in the object-oriented world.

And finally, many of the key components of our approach such as coding guidelines, our use of issues, and our approach to analysis object interaction diagrams, as well as techniques such as Depth-First (see Section 17.1) and Transcribe and Converge (see Section 18.3) have evolved from our work with software development projects in our mentoring practice.

1.3 ROLES IN AN OBJECT-ORIENTED PROJECT

The authors hope that this book has something of interest to almost everyone involved with object-oriented software development whatever their specific role. Throughout this book (particularly in Part 3 and Part 4) there are references to who does certain things in the process. When using this book it is important to understand what we mean by a certain role. The major roles of interest for purposes of this book follow. You may also see elsewhere in this book some off-shoot of a role described below or some slightly different terminology used. We hope that with an understanding of the following, that those slight variations will not be confusing or misleading, and we will continue to strive toward consistency in future releases.

When trying to relate these roles to those you may see on your team, it is important to note that in some cases one person may fill more than one role. For example, an architect might also be the team leader.

Customer

The real-life external or internal customer for whom the product is intended, or some representative of the external customer (such as a marketing representative or a "typical" end user), or any other receiver of the project deliverables.

Planner

Acquires and coordinates project requirements and also develops and tracks the Project Schedule.

Project Manager

Owns overall responsibility for the development of a project. Ensures that proper personnel and resources are available and tracks tasks, schedules, and deliverables.

Team Leader

Provides technical direction for the project. Leads team through the development process. Of course in large projects, you may have an overall team leader and leaders of smaller teams.

Architect

Responsible for overall design/architecture of the project. Manages the interfaces to other development activities related to this project.

Analyst

Takes user requirements and generates project specifications. Interprets user intention and defines the problems that need to be solved. Responsible for developing domain analysis model with users and other team members.

Designer

Responsible for the design of a subsystem or category of classes. Directs implementation and manages interfaces to other subsystems.

Domain Expert

Understands a particular business area. Keeps project focused on solving problems with relationship to the domain.

Developer

Implements the overall design; owns and designs specific classes and methods, codes and unit-tests them, and builds the product. Developer is a broad term and specialization is possible, such as a Class Developer.

Information Developer

Creates the documentation that will accompany the product when it is released. Includes installation material as well as reference, tutorial, and product help information in both paper and machine readable form.

Human Factors Engineer

Responsible for the usability of the product. Works with customers to ensure that user interface requirements are met.

Tester

Validates the function, quality, and performance of the product. Develops and executes Test Cases for each development phase.

Librarian

Responsible for creating and maintaining the integrity of the project library, that is comprised of the project work products. Is also responsible for enforcing work product standards.

1.4 METHOD AND LANGUAGE INDEPENDENCE

An object-oriented application development method should be implementation language-independent. That is to say that it should not make a significant difference in your development approach whether you are going to be building the application in Smalltalk, C++, or Java. The approach described in this book is language-independent and has been used in C++, Smalltalk, and Java projects.

The work products described in this book are language-independent. For example, an Analysis Object Model (see Section 11.3) has no language-specific requirements.

This is not to say there are no differences between individual work products depending on the language of implementation. Naturally, there are differences, particularly in low-level design and implementation work products. However, the same overall process is followed and the same work products are produced.

1.5 ADAPTING THIS APPROACH FOR A SPECIFIC PROJECT

This approach has been used in many software development projects both inside and outside of IBM. In some cases it has been used more or less as described in this book and in other cases has been used as a baseline and adapted to the particular needs of projects.

Many organizations have an existing methodological approach to systems development and wish to preserve as much of that approach as possible while defining a standard methodology to be used on object-oriented projects in their organizations. Rather than introduce a completely new approach, there is a desire to take an evolutionary approach and preserve as much as possible of their current development process.

Our experience has shown that many elements of standard development approaches are applicable to the object-oriented development world. Elements such as user documentation requirements, deployment, user training, and system testing tend to survive relatively unchanged in the move to the object-oriented world.

The areas that are impacted the most are planning, requirements, analysis, and design.

Planning is impacted due to the iterative and incremental nature of object-oriented development. Many standard approaches to development are based on a waterfall approach. Planning incremental and iterative development is much different from traditional styles.

The key differences in the area of requirements, analysis, and design are that when using object technology they should be expressed in terms of collaborating objects. Thus, existing techniques that express the world in terms of processes or entities must be changed. These are the areas that will be most impacted by a transition to objects.

It is possible to integrate the approach in this book with an existing development process. To do this, each work product in this book should be evaluated in light of an existing process. Where you find equivalency, feel free to substitute existing work products that you are more comfortable with. You may also wish to extend this approach in areas that you prefer to cover in more detail. You may also choose to cut back in areas of coverage that you find overly detailed. However, be careful as it isn't as simple as you might think. Two of the more common problems we have seen are:

- *Eliminating an object-oriented work product in favor of a nonobject equivalent* - We have seen projects eliminate Object Interaction Diagrams in favor of some form of process modeling. Unfortunately, process models do not express the world in terms of objects and therefore do not meet the requirements of object analysis or design.

- *Assuming an object is an entity* - Many organizations have some form of entity modeling in their existing processes and try to adapt an entity-based work product into an object-based work product. This should not be done. An entity is not an object and the manner in which entity-based work products are developed is not the same as object-based work products.

A standard approach should be defined in light of real systems development needs. Once defined, any process should be modified in the light of experiences. Therefore, it is essential that a "continuous improvement process" be defined. Getting feedback on how the approach is working or not working on projects must be part of the process.

In addition, object technology methodology is still an evolving field so it is necessary to keep an eye on developments within the industry in order to take advantages of important advancements.

Part 1. *Key Messages*

The key messages we wish to convey in this book are that we believe that the development of object-oriented software should be:

1. Work Product[3] Oriented and Workbook-Centered

2. Iterative and Incremental

3. Scenario-Driven

These key messages are derived from insights gained by the authors during object-oriented project mentoring engagements and are founded on an evolved philosophy rooted in two fundamental principles:

- Separation of Concerns

- Management of Risks

Together, these messages and principles form a framework for object-oriented software development that is largely independent of the specific notations and languages in that you might choose to develop, and independent of the specific techniques that you might use for development. We consider that this separation of process, notation, and techniques is very important, because it allows you to tailor each independently to the needs of your problem and development context. We don't think that one shape of development method should or could fit all development projects.

[3] A *work product* is a concrete result of a planned project-related activity such as analysis, design, or project management. Work products include items delivered to customers and items used purely internally within a project. Examples of work products are Project Schedules, Object Models, Source Code, and even executable software products.

In the following chapters we describe what we mean by each of the key messages, why we think that these ideas are so important, and how the remainder of this book is related to them. But first, let's look at the fundamental principles and how they affect our approach.

Separation of Concerns

Modern software development is a complex, expensive, and risky endeavor. Adding an object technology transition to it does not immediately ease the endeavor, but it does add confusion. The confusion comes from many sources, including competing and overlapping: technologies, middleware products, experts and advice, CASE tools, analysis and design methods, notation and even "war stories." All of these ingredients are blended together into an assortment of enticing soups and given exotic, popular, or misleading names. Not until you recursively analyze the ingredients of the different soups can you see the familiar, the common, and the unique elements and how they interact when called for in the different "recipes."

From our own experience, as well as from studying object-oriented analysis and design methods from renown experts, we have come to the understanding that certain products of object-oriented analysis and design are essential and that their necessity and value have little to do with the notation, tool, method, technique, process, or technology used to develop them. This led us to a guiding principle of *Separation of Concerns*. Just as software systems need to be analyzed[4] before they are designed[5], so too do we need to analyze the software development problem before we design development approaches (strategies).

After our analysis of this problem, and through our work with object-oriented software development projects, we designed an approach that maintained separation of certain concerns in order to maximize applicability of the approach, allowing it to support different methods, processes, and tool environments. We chose to separate:

- Tools and Notation from Work Products
- Work Products from Development Process and Techniques
- Analysis from Design from User Interface Design from Implementation
- Project and Risk Management from Work Product Development

Although we have separated these topics, we have also described how they relate to each other. So, you will find work product descriptions providing possible notations, advice on tool usage, references to techniques, and hints of traceability (method). Multiple techniques and methods are related to the work products that they yield. Project and Risk Management are planning activities that result in their own work products.

[4] Analysis is the act of exploring the problem domain to gain a better understanding of the problem.

[5] Design is the process of planning construction of a solution to a problem.

More importantly, the work product orientation itself allows us to focus on one concern at a time. For example, our Use Case work product focuses on basic functionality, whereas our Scenario work product expands on functionality by enumerating the different outcomes implied by different starting conditions.

By separating the concerns here, we hope you get to understand them more easily, as individual topics. By relating them in multiple ways, we hope you learn to combine them in your own situations to solve your unique problems most appropriately.

Management of Risks

In his *Mythical Man-month*, Fred Brooks quipped,

> How does a project get to be a year late?
> ...One day at a time.

This should not only remind us of the importance of maintaining and following a plan but also remind us of how lightly we consider risks. It is as if "risk" had the connotation of imminent and complete failure only. In fact, there are many small risks that arise, accumulate, and feed on each other on a daily basis that are equally pernicious. Nearly everything we profess in our approach is based on managing risk:

Separation of concerns
Addresses the risks of mental overloading, confusion, losing touch with fundamentals, and unnecessarily tying the fate of one concept, facility, or work product to another less suitable one.

Separating analysis from design
Addresses the risk of designing a solution for the wrong problem by allowing us to understand the users problem and its domain before we set out to create a solution.

Separation of user interface design from system design
Addresses the risks of creating a developer oriented (vs. user oriented) system and of letting the user interface get locked into the design by allowing us to involve experts (users and human factors) in an asynchronous cycle of user interface design and prototyping while the development team addresses design model and system environment issues.

Work product orientation
Addresses the risk of losing ground when tools, notations, techniques, method, or process need to be adjusted by allowing us to vary them while maintaining the essence and value of completed work.

Workbook-centered

Addresses the risk of clumping too much and too varied work products into hard to manage phase review documents by allowing us to work in parallel teams, incrementally, and iteratively adjusting our tools and methods while continuing to track the completeness and consistency of our tangible development artifacts.

Prototyping

Addresses the risk of waiting too long before knowing essential information by allowing us to minimize schedule and effort investments while resolving risks that require knowledge only achievable through direct experience.

Incremental development

Addresses the risk of "putting all our eggs in one basket" by allowing us to "learn as we go and apply what we learned," thereby minimizing the impact of misconceptions and suboptimal decisions.

Iterative development

Addresses the same risk as incremental development and of excessive schedule dependency by allowing us the freedom to try different alternatives and not get hung up on getting it right the first time.

Scenario-driven development

Addresses the risk of designing a system that does not satisfy the requirements by allowing us to focus on the traceability of functional requirements while casting our designs into objects.

If you look at the techniques sections you should be able to recognize how each of them addresses some risk too.

Perhaps (if we're successful), you will be able to quip:

How did this project manage to complete on schedule?
...By addressing each and every risk, one at a time.

2.0 Work Product Oriented and Workbook-Centered Development

We refer to our approach as "work product oriented" and "workbook-centered," since a prime focus of the way in which we do object-oriented development is to focus on the development of work products and carefully manage them in a logical entity called the "Project Workbook" that spans the development life cycle. We have observed that our more successful mentoring engagements have been with those projects that have taken a serious, rigorous approach to the development and maintenance of such a project workbook.

This is not a conceptual or theoretical approach to object-oriented development. The approach has evolved out of the mentoring engagements led by the authors since 1992 and is reflective of our experiences during those engagements.

During the evolution of our approach, there was general agreement on appropriate work product content and workbook structure, but we found differing opinions when it came to techniques for producing the work products. This is one reason why we have separated the presentation of the work products from a presentation of the techniques that might be used to build them. You could think of the techniques we list as a toolkit to be applied (and extended) as appropriate. They are offered for the reader to review and consider.

2.1 WHAT DOES WORK PRODUCT ORIENTED MEAN?

The object-oriented paradigm shift allows software developers to view the problem domain and their projected solutions to problems as a set of collaborating autonomous objects, each with their own attributes, relations, and behaviors. This view creates a natural framework, based on real-world and conceptual objects, in which analysis, design, and implementation take place. Though the characteristics of objects are normally hidden, they form the basis for classification (via commonalities and differences) within the framework. This allows for the conceptual and constructive efficiencies of inheritance and reuse, the flexibility of substitution, and the robustness of limiting the scope of changes. Unlike the procedural paradigm, which views software problems and solutions as a hierarchy of procedure invocations, an object-oriented approach yields a network of connected active building blocks.

This type of paradigm shift is not limited to the domain of software structure (i.e., programming). It can also be applied to the domain of software development (i.e., development process). In the same way that we shifted from the procedural view of software having phases, actions, tasks, and processes to thinking of software in terms of objects with encapsulated and, possibly inherited, attributes, relations, and behaviors, so too can we make the shift in thinking about *how* we develop software. Instead of thinking of the development *process* having phases, activities, tasks, and processes, we can think of software development as a network of collaborating objects (called work products or deliverables) which have unique and common attributes, relationships with other work products, and methods for developing, verifying, and presenting themselves.

Just as the object-oriented paradigm shift allows flexibility and robustness, so too does the work product oriented paradigm shift. Rather than specifying a prescriptive, step-by-step, procedure for creating working software from incomplete requirements, we can, instead, think of the facets and views of the problem we need to understand and the types of specifications we need to design that software. In essence, we need to identify the real world objects *from the software development domain* and develop them in an object-oriented way. We call these *work products*.

Which of these work products you need to develop and in which order they should be developed depends on the characteristics of your problem domain and your project. Factors that influence which work products you produce and the order in which you develop them depends on several factors:

- The size of the project

  Projects with more requirements, more people, or longer schedules tend to need more project management work products, a greater emphasis on subsystems, reuse, portability, testing, and packaging.

- The complexity of the problem

  Complex projects tend to have more risks that need to be managed via creative project management, which includes prototyping and use of iterative and incremental processes, and a more detailed treatment of system architecture.

- The degree of technical uncertainty involved

  Concerns such as performance in a networked or client-server environment, performance of large or heavily queried databases, intuitiveness and usability of systems oriented towards casual users, et cetera, require more emphasis on risk management, prototyping, and creative project management (iterative and incremental), as well as more emphasis on domain and requirements analysis.

- The nature of the givens and requirements

  The existence of legacy systems, data, and conceptual models as well as open versus closed sets of functional requirements and reuse intentions affect the precedence (relative importance and traceability) among work products.

The general and specific characteristics of work products are discussed in Part 3 and sample work products can be found in the workbook in Part 5.

2.2 WHAT DOES WORKBOOK-CENTERED MEAN?

Focusing on the production of work products is not enough. Since they comprise the primary means of project communication, they must be available and organized for easy access by the entire project.

To manage the independent development of numerous work products of varying types by many teams or individuals, successful projects create and manage a central depository, the *Project Workbook*, for all work products developed in support of the project. The "care and feeding" of this project workbook becomes the unifying characteristic of the project.

At its lowest level the workbook is organized by work product types. Depending on the project characteristics, higher levels of organization might be based on the subsystems that comprise the product, the releases that incrementally satisfy the requirements, development phases, or some other prime characteristic. In practice, what this requires is early agreement on what the structure of the project workbook will be, the specific work products it will contain, and the format of these work products.

This approach results from the observation that projects that have been successful have produced and used good project workbooks. We have also seen projects fail, due to poor requirements gathering, an almost total lack of analysis, ad hoc design, or no documentation. These failures could have been avoided had the team followed a workbook-driven approach.

2.3 WHAT IS A PROJECT WORKBOOK?

A project workbook is a *logical* book containing all project *work products*. The workbook is *logical* in the sense that its physical medium is not relevant, and it may refer to some of its constituent work products instead of containing them directly. Work products are the result of planned project activities such as project management, analysis, design, et cetera. Work products include items delivered to customers and items used purely internally within the project. Project Schedules, Analysis Object Models, Source Code files, and even the executable software product are all work products.

An important facet of a project workbook is that all the work products that contribute to the workbook have a common structure, at least conceptually. The common work product structure that we suggest is described in Section 8.1. See Part 3 for descriptions of each of the work products that we recommend for inclusion in a project workbook, and Part 4 for a collection of techniques that may be used to build these work products.

Note: Do not confuse the template that we have used in this book to describe each kind of work product with the structure that we suggest you use when constructing real work products for a project.

Ideas for the structure of your Project Workbook are presented below.

2.4 WORKBOOK STRUCTURE

The following lists the major chapters of a project workbook. Each chapter can be viewed as a different perspective on the project and acts as a "logical" container of the work products associated with that perspective.[6]

1. **Requirements** represent the application requirements from the customer's view.
2. **Project Management** is information required for the successful management of the software development effort.
3. **Analysis** work products are a formal representation of the problem domain from the customer's point of view.
4. The **User Interface Model** documents the design of the application's user interface.
5. **Design** describes the structure of the software to be built.
6. **Implementation** is the working application and all work products required to build it.
7. The **Test** chapter contains work products associated with the validation of the software.
8. Typically the **Appendix** is the repository for historical (old) work products.

We have chosen to organize the workbook by these perspectives, rather than chronologically, in order to support iterative and incremental development as well as other life cycle variations, which will be discussed later in this book. The purpose of the workbook is to emphasize that all work be documented as work products and be organized for ready access by the whole project team.

While we discuss the workbook as if it were a single, unified artifact, in reality, it may consist of a number of separate things: for example, current analysis and design products may be kept in a single document; historical analysis and design work may be kept in a file folder in a drawer somewhere; a project plan may be maintained as a separate document; and the code may reside on the hard drive of a team server.

[6] By "logical," we mean that each chapter might actually contain the work products, might just refer to the work products residing elsewhere, or might refer to other workbooks.

2.4.1 Composite Structure Workbook

Workbooks are a logical structure to organize and hold work products. The typical high-level structure of a workbook is organized by development phases or perspectives as shown earlier. This looks very simple and is an intuitive structure for a single development cycle of a monolithic system.

Physical Workbook Structure for Complex Systems

Large systems are often divided into subsystems to afford parallel development, application of specialized skills, and planned reuse. Also, large subsystems are often subdivided further into smaller subsystems for the same reasons. Since we recommend using the same the same workbook outline structure for each system and subsystem, we envision the complete system being described by a set of workbooks which reference each other. Logically, the system contains subsystems and the system workbook contains subsystem workbooks. Physically, though, each subsystem could be documented in a whole separate workbook or in whole major sections of a single physical workbook as shown in 2-1.

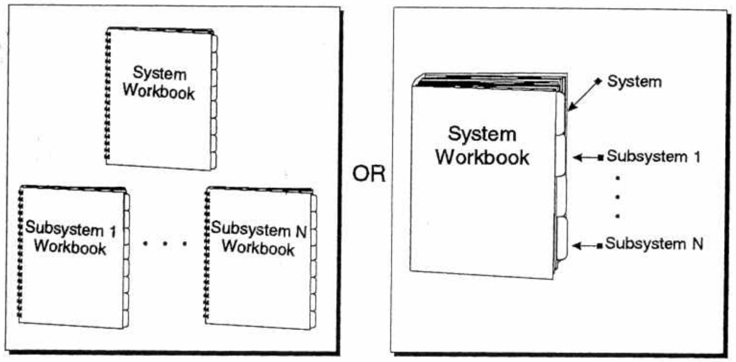


Figure 2-1. Physical Workbook Schemes for Subsystems.

System Phases or work products shared by subsystems can simply be referred to by the subsystems, for example:[7]

[7] We use the HyperText Markup Language (HTML) for general familiarity. You should be able to map this concept to GML, SGML, et cetera.

```
<h1>Subsystem D
<h2>Project Management
<p>Subsystem D will comply with the same Project Management
decisions as System X (see "System X, Project Management...").
```

For systems or subsystems with significant releases and/or internal integration checkpoints, we advise organizing the Workbook around those releases. It is probably best to keep the subsystem organization foremost, but keep all the phase/work-product material together in document parts. For a single physical workbook, a subsystem organization would be:

```
<TITLE>System X: Project Workbook>
<! Introductory preface, ToC, ...>
<hr><!----------------------->
<h1> System Release 1
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<h1> Subsystem A Release 1
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<h1> Subsystem A Release 2
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<! Appendices, Glossary, et cetera.>
```

If each subsystem has its own workbook, then simply use document parts (for example, HTML's <hr><h1> tags) to separate the releases:

```
<TITLE>System X: Project Workbook>
<! Introductory preface, ToC, ...>
<hr><!----------------------->
<h1> System Release 1
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<h1> Subsystem A Release 1
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<h1> Subsystem A Release 2
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<! Appendices, Glossary, et cetera>
```

```
<TITLE>Subsystem A: Project Workbook>
<! Introductory preface, ToC, ...>
<hr><!----------------------->
<h1> System Release 1
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<h1> Subsystem A Release 1
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<h1> Subsystem A Release 2
<h2> Requirements
...
<h2> Testing
<hr><!----------------------->
<! Appendices, Glossary, et cetera>
```

Internal releases (integration checkpoints) can be kept as dèltas over the previous release. External releases should be organized as complete workbooks (show all the work products that make up that release). This is not too difficult when using a word/document processor that supports imbedding files. For these, the common work products can be kept in common files and imbedded unchanged into all releases that use them. When a work product changes, future workbooks can refer to the correct version of the imbed file. A configuration management and version control system should be used to manage the versions of the source files (just as you would for source code files).

Composite Structure of Workbook

When a system is decomposed into subsystems, there is a "composite pattern" [Gamma95] among the system and subsystems, and also between their corresponding workbooks. In this pattern, a "component" can be a "leaf component" (complete) or a "composite" (defined by one or more subcomponents). See Figure 2-2.

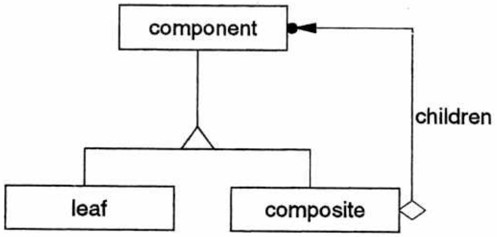


Figure 2-2. Composite Pattern.

Translating to system terminology, a system or subsystem is either complete or is composed of other subsystems. Likewise, a system or subsystem workbook is either complete (self-contained) or refers (defers) to lower level subsystem workbooks. The subsystems' run-time dependencies might be represented as a directed graph; however, the subsystems' definition is a simple containment hierarchy (see Figure 2-3.)

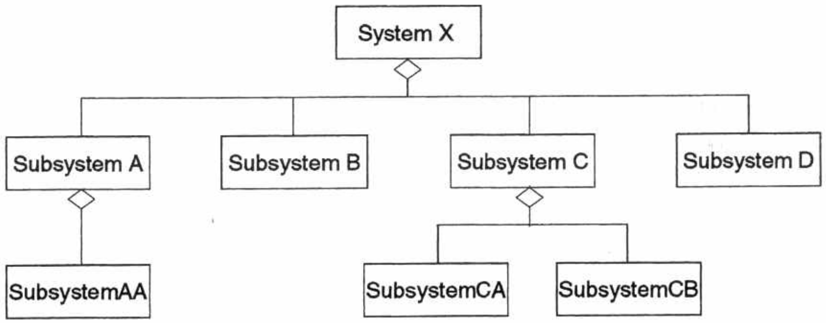


Figure 2-3. System/Subsystem Workbook Structure.

The relationship between a composite workbook and its component workbooks is defined by the Subsystem Model, see Section 13.5. The enclosing workbook can simply define and coordinate the relationships among its subsystem workbooks. In this case, some of enclosing workbook sections will be thin (or empty) since the content has been divided among (and deferred to) its subsystems. This will be more likely when the subsystem boundaries are sufficiently clear and stable or when the subsystems are not dependent on each other for definition (a reuse potential indication). It is less likely when the subsystems are more dependent on each other.

In the independent subsystem case, it is best to defer as much work (analysis, design, et cetera) as possible as early as possible. That is to say, identify the subsystems and their responsibilities and divide the work among parallel development teams as soon as you can.

In the dependent subsystem case, spend more time in the composite workbook, coming to a common analysis, high-level design, et cetera, so that there are fewer decisions (and potential for inconsistency) when the subsystem teams are formed and turned loose.

Workbook Strategy

The work products described in this book are those that we have found useful for medium and large development projects. Small projects might be expected to use only a subset of the work products. The goal of a project workbook is to act as a vehicle for coordinating, directing, and communicating development effort. While there are many ways in which project size may be measured, the characteristic that is of interest to us in this book is the amount of effort that must be put into coordination, direction, and communication between members of the development group.

One simple measure of gauging project size is the number of parallel development teams (where a team typically consists of four to six people).

Pick a strategy that matches the characteristics of your project:

- Small project (one or two teams)
  - Use single monolithic workbook
  - Separate logical workbooks for each major release
- Medium (three or four teams) and large (five or more teams) projects
  - Separate logical workbooks for each subsystem and release
  - If subsystems are independent/reusable
    - Defer as much work to subsystems workbooks as soon as possible
  - If subsystems are interdependent
    - Limit the work in subsystem workbooks by addressing it in common (composite-level) workbooks.

Spending the effort to keep information in the workbook current is essential to a project's success. Things can't just exist in someone's head—they must be written down somewhere so they can be reviewed and understood by others.

2.5 TERMINOLOGY

Workbook A *logical document* containing all the *work products* of a project.

Logical document A collection of machine-readable and other material that is considered to be a single, conceptual whole, even though its physical representation may be distributed across media, tools, and location.

Work product A concrete result of a planned project-related activity such as analysis or project management. Work products include items delivered to customers and items used purely internally within a project. Examples of work products include Project Schedules, Object Models, Source Code, and even the executable software product.

For additional terms see the "Glossary" on page 617.

3.0 Iterative and Incremental Development

The approach that we describe in this book is iterative and incremental in the sense that we believe that an iterative and incremental development process is beneficial for most (not all) projects. What that means will be described in this chapter.

A variety of development process models have been proposed. These include waterfall, spiral, iterative, incremental models, and others. This chapter will discuss some of these, including the advantages and disadvantages of each. The chapter will then relate these to the needs of real projects, in particular project risks, and present a process model that we call the "iterative and incremental" model.

Note that many overlapping and conflicting definitions exist in the vocabulary of process models. This chapter does not attempt to be definitive but instead it should be read partly as a clarification of terminology for the purpose of later discussions in this book, and partly as an attempt to explain our understanding and interpretation of the terms. The particular terms used here are not necessarily standard in the literature. They are used here because they are considered the most appropriate.

3.1 DEVELOPMENT PROCESS MODELS

3.1.1 The Waterfall Process Model

The waterfall model of development was the first attempt to discuss the software development process in semiformal terms. The waterfall model consists of a sequence of "phases" whereby in each a particular (and unique) kind of development work is done.

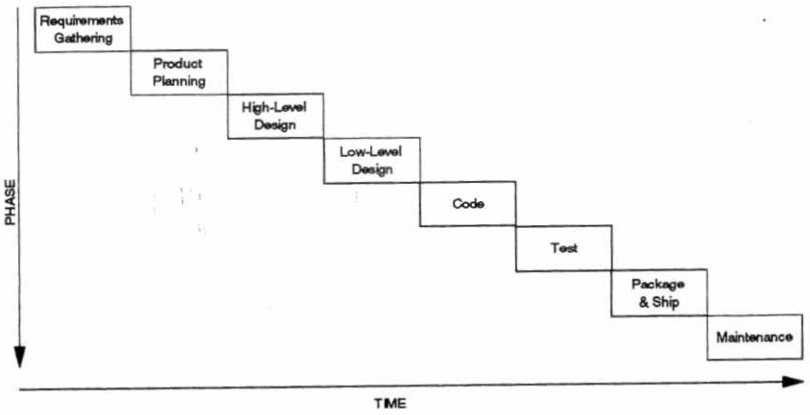


Figure 3-1. *Waterfall Process Model.*

The specific phases identified by Boehm [Boehm88], in Figure 3-1, are not relevant here; what is important is the sequential nature of the activities: All requirements are gathered before the next phase begins, et cetera. The essence of the model is that project phases are identified, ordered, and then carried out in this order without ever revisiting previous phases. That at least is the theory.

In reality, the model is usually contaminated by schedule pressures and downstream decisions:

- Successor phases often start before their predecessors end due to the pressure to shorten development schedules and sometimes due simply to early staffing of people with the special skills to carry out the successor activity. One might think of these as "leaks" in the waterfall.
- Earlier phases are often revisited, creating "gravity defying eddies" in the waterfall metaphor, in order to reconcile details with discoveries and decisions made in later phases.

The early appearance of "leaks and eddies" signaled the naivete of the model.

Advantages:

- *Simplicity.* The main advantage of the waterfall model is that it is the simplest possible model. Everyone can understand it, provided of course that everyone agrees on the definition of each project phase and its relationship to all the other phases.
- *Ease of management.* A corollary of the simplicity of the model is the ease of managing a project run along these lines, provided of course that the model is appropriate to the project. When managing a waterfall project, tracking progress against a plan is straightforward, as there is only one planned pass over each work product that is, therefore, either complete or incomplete.

Disadvantages

- *Incomplete or unknown requirements.* The model assumes that it is possible to gather all project requirements before analyzing the problem and designing a solution. In practice, requirements are very rarely all known in advance in this way. Even if complete requirements are written as an initial step, requirements frequently change as the project proceeds as both developers and potential users or customers of the software understand the problem and the solution better. If requirements do change during a waterfall project, the process model has no way of taking this into account. The result is frequently a crisis.
- *Incomplete design experience.* Assuming that complete, correct, and unchanging requirements can be gathered in one sweep, it may still not be possible to design a solution to the problem in one go. Design occurs at many levels from architecture down to low-level design and coding. It frequently happens in practice that lower-level design activities shed light on the implications of higher-level design statements. The issues raised in this way are usually foreseeable in advance, if only sufficient thought and experience had gone into the design. The fact remains, however, that complete design attention to detail and complete design experience are rarely available in practice, for whatever reason. In such a case it is difficult or impossible to design a complete solution and expect it to be successful without any feedback from developers, testers, users, et cetera. A waterfall project has no way of addressing a need for redesign if one arises. The result, once again, is predictable.
- *Scheduling.* Scheduling a waterfall project requires a great deal of confidence in the relative resource required for each project phase. A serious mistake in project planning is very hard to recover from, because the waterfall process makes it difficult to deliver partial solutions: It is all or nothing. This is partly a result of the fact that the waterfall process does not partition work in time except by phase, and partly because the planning of a waterfall project is itself one of the initial project phases that (in waterfall style) is carried out once and not expected to change.

Applicability

All published process models are appropriate to some kinds of project. The waterfall model is appropriate to projects that are very similar to projects that have already been completed by the same development team, or projects that involve relatively minor extensions or modifications to existing software components. In such cases the requirements, design, and resources can probably all be anticipated and a sequential planning and stepping through the project phases is probably the simplest and best solution.

3.1.2 The Incremental Component Process Model

By "incremental component process" we mean the building of components one by one. This process is simply the composition of a "Divide and Conquer" approach and the waterfall process. The process requires the system to be decomposed ("Divide") into components that are each developed using the Waterfall Process ("Conquer"). Conceptually, the waterfall process is repeated sequentially by the development team, but in practice parallel development teams would realize some concurrency. The components might be as fine grained as objects, but they are more likely to be coarser grained.

Advantages

- *Simplicity.* This process is only slightly more complicated than the waterfall process. The fewer the components, the simpler it is.
- *Ease of management.* The incremental process has the same ease of management as waterfall. The only difference is that there are a series of waterfalls.
- *Scheduling.* Since each increment is smaller than the whole project, management sees finer grained scheduling, checkpoints, and reporting of progress. It is much more informative to say "75 percent of the components are complete" than to say "We're two-thirds through high-level design."
- *Benefit of Experience.* Later increments benefit from the knowledge, skills, and resources developed during earlier increments. Confidence and quality levels normally increase for later increments.

Disadvantages

- *Incomplete or unknown requirements.* The incremental process has the same disadvantage as the waterfall process in that it only addresses each phase (of each component) once. If requirements change, or are not understood in the first place, the process cannot deal with it.
- *Incomplete design experience.* The incremental process is slightly better than the waterfall process here in that experience gained from developing one component can sometimes be transferred to later ones. But again, this process does not account for insight gained in later phases of a component's development affecting decisions made in earlier phases.

Applicability

The incremental component process model is appropriate for projects that have readily identifiable components that are relatively independent of each other or build upon each other. It is especially useful where there are many similar components, since later components are more likely to benefit from the knowledge and experience gained during earlier development. It is also appropriate when fine grained schedule tracking is indicated to manage development cycle time risks.

3.1.3 The Iterative Process Model

The iterative model describes a process where an initial set of work products evolves into the production system by reworking the work products iteratively. In a sense it is similar to the incremental process in that it employs the waterfall process repetitively, but it differs in that it reworks the same (monolithic) component each time. It is often characterized by having monthly or weekly internal "drivers," "code drops," and alpha, beta, and early-ship releases.

Advantages

- *Simplicity.* Conceptually, the iterative process is quite simple: Just run through the waterfall process repeatedly on the same scope until it is correct. If, based on previous experience, you know how many passes it will take to "get it right," then it is not hard to set up and explain.
- *Incomplete or unknown requirements.* With this process you have several chances to deal with late discovery of requirements. When you discover a requirement that

affects phases previous to the one you're working on, just schedule it to be handled on the next pass.

- *Incomplete design experience.* Similarly, as design, implementation, testing, and packaging experience is gained during early iterations, they can be factored into earlier phases (e.g., nonfunctional requirements, scenarios, state models, subsystem models, et cetera) of later iterations. This provides a "second chance to do it right."

Disadvantages

- *Ease of management.* Determining the number and duration of iterations can be difficult. Since requirements and architecture are allowed to be open-ended, tracking progress against a plan is difficult.
- *Scheduling and resource planning.* In addition to the confidence required in understanding the resource requirements for each phase of waterfall development, the iterative process also requires confidence in understanding the relative resource requirements of each iteration. Statements such as "We've just completed the second of six planned iterations" only have meaning as a measure of progress toward project completion if the schedule duration and resource requirements for all iterations are understood since one iteration is likely to be different from the next.

Applicability

Despite the disadvantages, the iterative model is widely used and is appropriate for projects working in new domains or environments where requirements and technical difficulties cannot be foreseen. It works best with established development groups where the project manager has some experience with the resource and scheduling needs and characteristics of the group.

3.2 PROJECT RISKS

When discussing different development process models, two things become clear. Firstly, no one process is the right one for all projects. Radically different projects require different development processes. Whether or not this is desirable is not the point; it is inevitable. Secondly, the decision of which process to use is risk-driven. If there are no risks associated with your development project, then there is no need to adopt anything other than a waterfall development process. Anything else would be unnecessarily complicated. For good or for bad, it is almost never the case that a project has no risks.

The presence of risks implies a need to test and to answer technical questions sufficiently early in the life cycle of a project so that there will be time to act on the information obtained. The nature of your project risks will determine the process model that is most appropriate to you. By understanding the relationship between risks and process it is possible, and profitable, to tailor your development process to your needs, which might involve using a process that is different from any of the published variations. This should not be a cause for undue concern. Greater concern should be reserved for the unthinking application of development processes.

The relationship between risk and process is precisely the one mentioned above: *The scheduling of any project activity that carries risk must take that risk into account.* This is not, of course, to say that all bets must be hedged; many, for example, the collapse of the building in which development is being carried out, might be too expensive for the probability of the risked event occurring. A decision must be made of which risks to take into account.

There are many kinds of project risk. Section 10.7 provides examples and a discussion of risk management in general. Risks associated with development are frequently managed appropriately by arranging the project timetable so that risky development activities are scheduled with adequate time for review, testing, rework, re-review, et cetera. When this principle is applied to whole categories of development risks, it leads to some form of iterative and/or incremental schedule, but one which is tailored to the needs of a particular project and not just uniformly iterative or uniformly incremental. This topic will be explored further in the following section, and again in Section 5.0.

3.3 THE ITERATIVE AND INCREMENTAL PROCESS MODEL

Having said that no one process is appropriate for all projects, we cannot now claim to present the ultimate process, and it is not our intention to do so. It has, however, been our experience that almost all projects fall into the category of those requiring at least some incremental aspect and at least some iterative aspect. Furthermore, it has also been our experience that projects typically benefit from a development process that is strongly incremental.

At this point some clarification of terminology is required. By "incremental" we refer, here and subsequently, to *incremental by requirements.* That is, an incremental process builds executable drivers or products each of which satisfy successively more end-user requirements. The portion of a project that aims at developing a particular driver or product is termed an "increment." By "iterative" we refer, here and subsequently, to a process in which *iterative rework* is performed within increments.

Thus, when we say that most projects would benefit from a development process that is strongly incremental, we are observing that most projects:

- have requirements that are uncertain and incomplete,
- employ technologies with which the development team are not entirely familiar,
- are reasonably large and complex.

The first two points imply obvious risks that are most directly addressed by some form of iterative and/or incremental schedule, as mentioned in Section 3.2. The last point, however, rules out a development process that is predominantly iterative, for the reasons discussed in Section 3.1.3, principally design instability. The answer would seem to be some form of hybrid process.

Development may be thought of in terms of a matrix of requirements vs. components as shown in Table 3-1.

Table 3-1. Requirements Versus Components Matrix.

Components	**Requirements**									
	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**
A	A1	A2								
B		B2			B5		B7		B9	
C	C1		C3			C6		C8		C10
D			D3	D4						
E		E2			E5	E6			E9	
F				F4			F7			F10
G	G1					G6		G8		
H					H5				H9	
I				I4			I7		I9	
J						J6		J8		J10

Development may proceed either horizontally or vertically in terms of this matrix. Vertical development, constructing the application a row at a time, is essentially an *incremental by component* process. Of course, to test (vertical) end-user functionality, all the rows that contribute function will by then have to be in place. Horizontal development, constructing the application a column at a time, is essentially an *incremental by requirement* process that implements only that piece of the design at each stage to support the requirements in question. Both vertical and horizontal development processes are incremental in some sense, but in the former it is components that are delivered incrementally while in the latter it is end-user requirements that are being satisfied incrementally.

The advantage of an incremental by requirements process is that it permits the system to be built slowly and with focus (as in the incremental by component process), and hence with attention to quality, while still permitting tests of end-to-end functionality to be performed early in the project (like the iterative process).

Envisaged, then, is a cyclic process in each cycle of which particular end-to-end requirements are designed and implemented (see Figure 3-2)

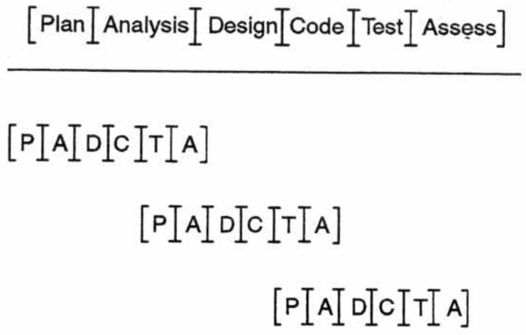


Figure 3-2. Overlapping Iterative Development Process.

To address project risks each development cycle must not only satisfy the new requirements on that cycle but it must also budget resources to rework existing functionality in the light of reviews, tests, user responses, et cetera, stemming from the previous cycle. The element of rework in each cycle will vary from cycle to cycle, but it should dominate development. To remind us of the fact that each development cycle is driven by the need to satisfy the particular requirements that define that cycle, and not by rework, each development cycle is called an "increment." Each increment includes a certain proportion of iterative rework. When planning the increments it will not be possible to anticipate which particular parts of the design will require the rework, but a certain proportion of each increment should nevertheless be set aside to anticipate the inevitable fact that rework of some kind will be necessary. Iterative rework might be necessary to correct errors, increase performance, increase modularity, or increase extensibility. Some of these issues will be anticipated; others will arise only during reviews and tests.

Increments can overlap in time, but functional dependencies between increments and the need to plan the rework content of an increment imposes constraints on the degree of overlapping that is possible in practice.

We call this development process an "iterative and incremental" process. Note that while the iterative and incremental process defines the large-scale structure of a project, it does not prescribe the way in which the development of each increment is to be carried out. That is, it does not prescribe the internal structure of an increment. The discussion of the iterative and incremental development process is continued in Section 5.0.

Advantages

- Focus on high risk project activities
- Early feedback on key design points
- Tangible deliverables at every stage of development
- Establishment of an end-to-end software development framework
- Overlap of development activities
- Incremental product release
- Rapid operational capability
- Efficient use of resources
- Reduced risk of product disaster
- Flexible in the face of changing requirements and environments
- Permits rework of architecture and design
- Developers can learn as they go
- Reduction of risk by reducing uncertainty
- Improvement of quality by continuous testing during development
- More visible reuse opportunities

Disadvantages

- Additional project management and process complexity
- Difficulties in identifying the requirements for each iteration
- Difficulties in prioritizing risks
- Subsequent increments may affect the work done in the earlier ones
- Coordination of teams working in different stages
- Additional change-control management required
- Possible ignoring of long-term architectural and usage considerations
- Lack of an initial complete specification to aid early estimation

Applicability

The iterative and incremental process is not universally applicable. It is appropriate to medium and large projects with end-to-end requirements that can be partitioned conveniently and that face uncertainties. These uncertainties might take the form of:

- Unknown, uncertain, or incomplete requirements
- Design risks
- Unfamiliar application domains
- Unfamiliar technologies (such as object technology)

Because the iterative and incremental process combines both iterative and incremental aspects, it can be used as a generic template for a project. Projects that lend themselves more to a waterfall or iterative approach simply have few increments or a greater proportion of iterative rework in each increment respectively. Thus waterfall and iterative processes can, if one wishes, be considered as "degenerate" forms of the iterative and incremental process model. That might be a useful point of view because the iterative and incremental process forces the planner to think about risks and about scheduling. Adoption of a purely waterfall or iterative approach preempts or constrains such considerations.

The iterative and incremental process model is further discussed in Section 17.2.

3.4 TERMINOLOGY

Waterfall development
A process model for software development in which development is partitioned into unique phases, each characterized by a particular kind of activity such as planning or analysis. The phases are strictly ordered and carried out in that order with each phase being performed exactly once.

Iterative development A process model for software development in which the system evolves by iterating over its work products during a sequence of iterations. Each iteration results in executable system code.

Incremental by requirements development
A process model for software development in which the project is divided into a sequence of increments. The work products of each increment satisfy successively more end-user requirements. Each increment results in executable system code.

Incremental by components development
A process model for software development in which the project is divided into a sequence of increments. During each increment a particular set of components of the final product is constructed. Each increment results in executable code for one or more subsystems.

Iterative and incremental development
A process model for software development in which the project is divided into a sequence of increments. During each increment a combination of incremental by requirements development and iterative rework is performed. Each increment is principally driven by the end-user requirements that define that increment, but a certain proportion of the increment is set aside for iterative rework of existing work products to take reviews, tests, experience, and other feedback into account.

For additional terms, see the "Glossary" on page 617.

4.0 Scenario-Driven Development

The approach that we describe in this book is scenario-driven in the sense that there is very strong traceability from requirements right through to Source Code and Test Cases, and that a key work product in this chain of traceability is the Scenario.

4.1 THE TRACEABILITY GAP

The overly simplistic object technology hype is that you take a set of requirements; build an object model that identifies classes relevant to the problem and the relationships between the classes; add design classes relevant to the solution; code the classes; and that's about it. Unfortunately, this hype is believed by many people.

There are a number of flaws in this argument. It ignores the fact that requirements are rarely complete or correct, and the fact that design involves much more than the adding of classes. The former is discussed in Section 3.0; the latter is addressed in Sections 7.5 and 13.2. A more subtle problem is that the hype does not indicate how the classes of the object model are to be identified, and how the object model is to be validated against the requirements. In other words, there is a traceability gap between requirements and the object model. This applies equally to the analysis object model and to the design object model.

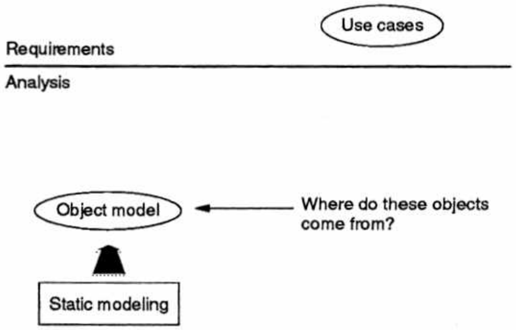


Figure 4-1. The Traceability Gap.

Using purely *static* modeling, the construction of an object model, it is impossible to bridge the gap between requirements and either analysis or design (Figure 4-1). What is missing is any reference to the *dynamic* modeling: That is, the construction of models that describe how the system works to satisfy its requirements. Dynamic modeling is not relevant solely to real-time systems; it is a vital part of the mainstream development process. Dynamic modeling fills the traceability gap, whether or not the system is real-time (Figure 4-2).

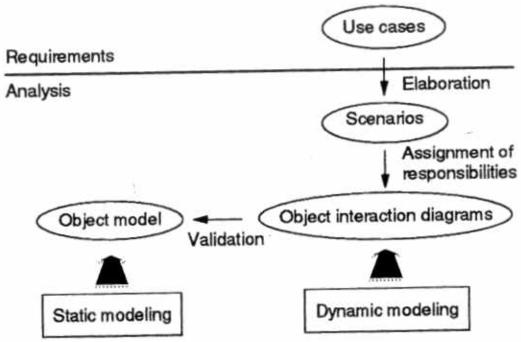


***Figure** 4-2. Static and Dynamic Modeling.*

4.2 USE CASES

In order to understand requirements traceability, we should first look at the work product designed to capture the functional requirements. It is called the *Use Case Model* and you can read all about it in Section 9.2. For now, though, it is sufficient to describe it as the set of:

Actors People and other external objects that use the system or interface with it.

Use Cases What the actors do with the system.

In this book we treat Use Cases as the definition of all externally visible, end-to-end behavior of the system. To keep things manageable and focused, we don't concern ourselves with all the variations of outcome due to the possible permutations of assumptions. For example, "Fax machine Receives Fax" does not concern itself with "out of paper," "incomplete transmission," "power disruption," et cetera.

You might think of this as outlining the Quick Reference Card for the system.

4.3 SCENARIOS

Scenarios are the work products intended to elaborate Use Cases by enumerating all the variations of outcome due to the possible sets of assumptions. If it helps, think of the set of assumptions as the possible starting states the system can be in when a Use Case is enacted and the outcomes as the resultant ending states. The Scenario need not concern itself with *how* it is to be implemented in the target system or which other objects it needs to collaborate with to accomplish the task, unless it is part of the required outcome. For details, see Sections 11.4 and 13.7.

By following this *separation of concerns* and by using a workbook organization (with naming conventions), we have taken the first step toward requirements traceability. There is a thread now from each actor to its set of Use Cases and from each Use Case to its set of Scenarios. The Scenarios can be checked for completeness and consistency and also used in early project estimation metrics.

You might think of this as outlining the Principles of Operation manual for the system.

4.4 OBJECT INTERACTION DIAGRAMS (OIDS)

The Object Interaction Diagram is the next work product of interest in scenario-driven development, see Sections 11.5 and 13.8. Its purpose is to divide the responsibility of attaining the outcome specified by a Scenario amongst the objects indicated by the:

Use Case - the actors (especially when acted on or prompted)

Scenario - assumptions and outcomes are really states of participating objects

Domain (from an evolving Object Model) - real or conceptual objects that help in this environment

During analysis, identification and division of responsibilities among the objects are the prime goals. During design, the responsibilities take a more detailed form indicating method names, parameter types, data flow, et cetera, for all the participating objects.

At this point you should be able to see how the requirements can be traced from Use Cases down to services (methods) that need to be implemented in the set of collaborating objects. You might think of this as tracing the internal events resulting from the enactment of the Use Case within a particular Scenario.

4.5 A CHAIN OF TRACEABILITY

We have seen how Use Cases are a convenient way of defining the system boundary and expressing the top-level functional requirements of the system. We have also seen how Scenarios can be used to expand the Use Cases into detailed functional requirements. It has then been shown how OIDs can be used to demonstrate how objects obtained from an object model can interact in order to perform a scenario. Put together, this is a recipe for closing the traceability gap. A continuous chain of traceability now runs as follows.

- Use Cases
- Scenarios
- Object Interaction Diagrams (OIDs)
- State Models
- Class Descriptions
- Object Model

Thus far, the list holds for both analysis and design sets of work products separately, although there is obviously analysis/design traceability too.

- Source Code
- Test Cases

Each of these work products, and their interrelationships, is explained in Part 3. A complete table of traceability between work products is presented in 8.6. In a scenario-driven approach, all development activity is derived from the need to satisfy functional requirements expressed as scenarios. Furthermore, the work products on this main line are linked by a strong concept of validation.

- Fundamental functional requirements are recorded as Use Cases.
- Each Use Case is expanded into multiple Scenarios by listing possible combinations of assumptions and outcomes.
- Scenarios are elaborated into Object Interaction Diagrams (OIDs) by depicting object collaborations and the assignment of responsibilities to classes.
- The state-dependent behaviors of certain objects is explored through State Models that show the possible states and state transitions of an object.
- The Object Model characterizes the classes identified by the OIDs in terms of their key attributes, their operations, their states, and their relationships.
- The Class Descriptions collect all the information associated with a class from the Object Model, OIDs, and State Models.
- The Source Code for each class is developed from the specifications found in the Class Descriptions.

- The Scenarios are used as specifications for Test Cases that validate the classes, subsystems, and systems implemented via their Source Code, which closes the development loop.

Many variations on this theme are possible and desirable depending on circumstances. What is important is that each of these work products are produced, and that they have certain relationships with each other. It is primarily the order in which the work products are produced, updated, and validated with respect to each other that can and should vary depending on the development context. This is the link between the scenario-driven and work product oriented concepts.

The connection between the iterative and incremental and scenario-driven ideas is that the requirements of each increment should, we believe, be defined in terms of scenarios. This enables Test Cases derived from requirements to close the development loop of each increment. This idea is explored further in Section 6.0.

An important aspect of defining increment requirements in terms of scenarios, and using the traceability between work products described above, is that we do not advocate designing complete classes and then building them. Instead, only those methods that a class requires in order to carry out a scenario of the current increment are designed and built in that increment. Thus while classes are used to structure analysis, design, and implementation work products, it is scenarios that scope this work and are used to decide which classes and which methods are to be included.

The links between Use Cases, Scenarios, OIDs, and Object Models in particular can be exploited in many ways. For example, during development many Issues will arise. These Issues frequently take the form of a question: how should we implement a particular Scenario bearing in mind certain factors? The obvious way to express the various design options for the problematic Scenario is by means of OIDs, each OID demonstrating a different way of assigning responsibilities to objects of the Object Model in order to carry out the Scenario. Resolution of the Issue consists of selecting one of these design options. Scenarios and OIDs can therefore be used as a basis not only for documenting completed designs but also as a vehicle for expressing design problems and options.

For more details on this technique, see Section 18.7.

4.6 SCENARIO-DRIVEN DEVELOPMENT VARIATIONS

Although we recommend the Scenario-driven approach for most projects since it validates the purpose and need for each object and its features, there are situations when alternate approaches are more appropriate. We present them here so that you will understand the nature of the exceptions that justify these variations.

We should also note that these variations should be consciously considered for each subject area, subsystem, and even each increment of development, so that the best approach is used in every situation. All of these approaches are valid and interoperable—you can "mix and match."

Before we dive into the variations, let's present the core sequence of work products to be developed for the Scenario-driven approach (see Figure 4-3.

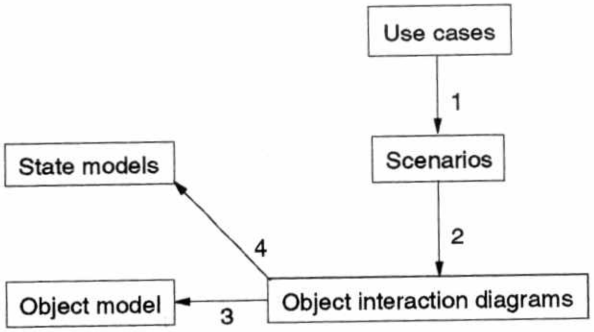


***Figure** 4-3. The Scenario-Driven Sequence of Work Product Development.*

In this sequence, functional requirements, in the form of Use Cases, are elaborated into Scenarios, which enumerate the assumption and outcome pairs associated with each Use Case. Then each Scenario is elaborated into an Object Interaction Diagram which distributes the system function to the objects by assigning class responsibilities and identifying object collaborations. The classes and their OID-implied attributes, associations, and operations are next organized by class and transferred to an Object Model. When a significant amount of class behavior varies due to an object's state, the operation sequences identified in the OIDs referring to that class are collected and reorganized into a State Model (e.g. a State Transition Diagram). This will ease the culling out of method descriptions that are part of the Class Description work product.

The Scenario-driven approach is most appropriate for the following situation:

Given Rich set of functional requirements (use cases)
Not Given Legacy data domain structure
Goals Functionality, conformance to agreed functional requirements, strong traceability through development process

4.6.1 Data-driven approach

The data-driven approach is a natural one for designers who have been involved in data modeling or database oriented programming. It should not be used as the standard approach, but is appropriate when working with a rich pre-existing data domain *and* an *open-ended* set of functional requirements. That is, data-driven is often appropriate when striving for evolutionary stability, longevity, and extensibility, since the data and their organization are more important than the functional requirements for each increment of development.

The sequence of work product development (modeling) starts with the Object Model (data model) that has prime significance. Other work products (models) are subordinate to it and their development follows the same sequence as the scenario-driven approach (that's why we call this a variation of scenario-driven). Traceability of functional requirements therefore is secondary to development of a strong data model.

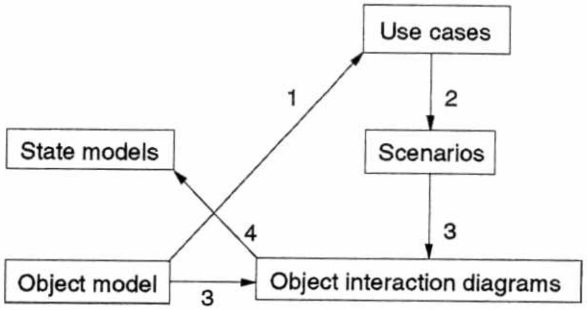


***Figure** 4-4. The Data-Driven Sequence of Work Product Development.*

In this sequence, the Object Model is developed first. It is usually derived from a pre-existing entity-relationship (ER) diagram, database schema, or legacy data structures. Although classes, attributes and most associations (relationships) can be harvested this way, it is not so easy to identify the responsibilities and operations for each class. So, after the data portion of the Object Model is developed, the normal scenario-driven process is used to identify Use Cases, Scenarios, Object Interactions, and State Models. The Object Model acts like a design constraint. It is updated from the OID with the assignment of operations and possibly new attributes and associations that became evident during the dynamic modeling.

The data-driven approach (see Figure 4-4) is most appropriate for the following situation:

Given Rich pre-existing data domain (e.g., Object Model)

Not Given	*Closed set* of functional requirements
Goals	Extensibility, longevity, stability

4.6.2 State-driven approach

The state model-driven approach is a natural one for designers who have been involved in "real-time" system design and those involved in business process re-engineering. It should not be used as the standard approach, but is appropriate when working with a rich pre-existing behavioral domain *and* an *open-ended* set of functional requirements. That is, a state model-driven process is often appropriate when striving for emulation of real-world behavior in distributed control applications, since object state transitions, actions, and collaborations are more important than simply satisfying the functional requirements for each increment of development.

Don't think of state modeling as something reserved for real-time manufacturing process control systems and traffic control systems. *Orders, Order items, Invoice, Customer Account, Catalog Item,* et cetera, all have significant real-world state models which define the very meaning of what we call business.

The sequence of work product development (modeling) starts with the state models for stateful objects. These have prime significance. Other work products are subordinate to it, and their development follows the same sequence as the data and scenario-driven approaches (we still call this a variation of scenario-driven). Traceability of functional requirements and the structure of the data model are secondary to development of a state models that closely model the real-world.

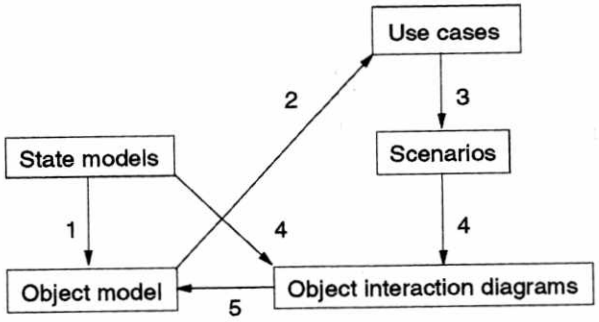


***Figure** 4-5. The State Model-Driven Sequence of Work Product Development.*

In this sequence, the State Models are developed first. They are usually derived from pre-existing State Models from legacy systems. Although state attributes, transition operations (actions), and some associations (relationships) can be harvested this way, only a limited portion of a system's classes and their characteristics can be identified like this. So, after the few State Models are developed and transferred to the Object Model, the Data-driven variation of the Scenario-driven process is used to identify Use Cases, Scenarios, and Object Interactions. The State Model acts like a design constraint when developing the Object Interaction Diagrams. Finally, the Object Model is updated with all the additional classes (and their attributes, relations and operations) that became evident during the dynamic modeling.

The state model driven approach (as shown in Figure 4-5) is most appropriate for the following situation:

Given	Rich behavioral model (e.g., State Models)
Not Given	*Closed set* of functional requirements
Goals	Distributed control, emulation of real-world behavior

4.7 TRACEABILITY FOR SCENARIO-DRIVEN DEVELOPMENT

An extremely important concept to developing object-oriented software is the concept of traceability. This is recognition of the fact that there exist interdependencies between various work products and should the content of one work product change, there may be a ripple effect impacting other work products in the project workbook.

For any work product we need to understand what other work products it can be "impacted by," and what other work products it "impacts."

The benefit of traceability is that it can dramatically lessen the impact and risk associated with changes to the project. Of course to gain the benefits of traceability does not come for free. It is vital that documentation be kept current, and this involves effort and rigor on the part of the team. Documentation cannot be an afterthought.

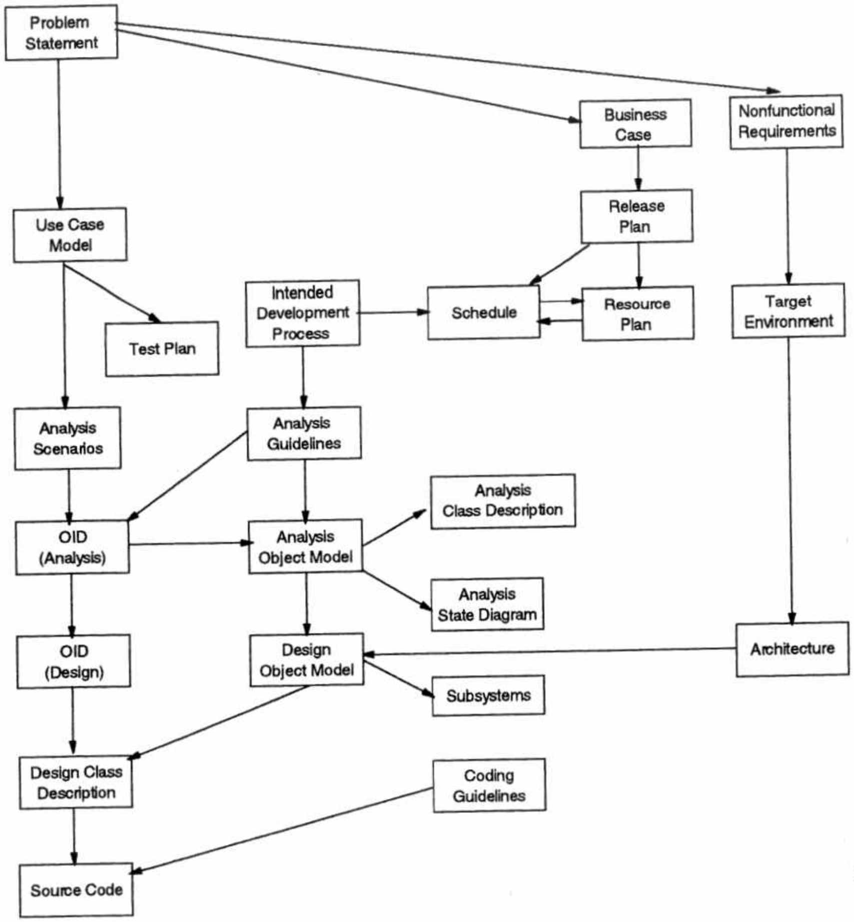


Figure 4-6. Partial Work Product Traceability Using Scenario-Driven Approach.

Figure 4-6 shows some of the traceability dependencies among some of the work products described in this book. Note that this is only a partial traceability model intended to illustrate the traceability and scenario-driven development. In this figure, we can see that the Analysis OIDs are impacted by scenarios and in turn impact Analysis Object Model and Design OIDs.

Traceability Example

Imagine you have developed an application to support a Life Insurance company's sales agents. The application is in production and involves 200 classes and about 40,000 lines of code. Now a change occurs to the way a policy is processed. With traceability the changes required to the application are easier to manage. Traceability allows us to:

- Identify affected Use Cases (and if required generate a new Use Case)
- Identify impacted scenarios and their Object Interaction Diagrams (analysis)
- Revise the Analysis Object Model (adding any new classes, attributes, behaviors, and relationships)
- Review and alter the associated Design Object Interaction Diagrams (OIDs) as required
- Identify classes and methods impacted by relationship between the Design OIDs and code files
- Update code

Object-oriented analysis, design, and implementation support traceability, because of the manner that Use Cases and Scenarios can be traced through the entire project life cycle. Rather than having to go through thousands of lines of code and determine which of it is impacted by changes, traceability presents a structured, manageable, methodical approach to maintenance and change control. This is one of the reasons that so many object-oriented applications report improved productivity on object-oriented application maintenance.

4.8 TERMINOLOGY

Use Case A statement of top-level functional system requirements. A Use Case is usually defined textually, but for convenience it may also be represented in a *Use Case Model.* A Use Case may represent a way that the system is to be used by external *Actors*, or a way that the system is to use its *Actors*. See Section 9.2.

Actor An agent external to the system, with which the system interacts. See Section 9.2.

Use Case Model A representation, often graphical, showing the boundary of a system, the *Actors* external to the system, and the *Use Cases* internal to the system. See Section 9.2.

Scenario An externally-visible system behavior. A Scenario is a *Use Case* plus a set of assumptions plus a set of outcomes. Assumptions and outcomes can be specified formally or informally as appropriate to the project and the development status. A Scenario can be thought of as a detailed functional system requirement. See Sections 11.4 and 13.7.

Object Interaction Diagram (OID)

An OID is a graphical depiction of the way that objects interact to carry out a *Scenario*. An OID is frequently depicted as vertical time lines representing the participating objects, and horizontal arrows representing object interactions. See Section 11.5 and 13.8.

Scenario-driven development

A developmental focus on a continuous chain of traceability from *Use Cases* through *Scenarios*, *OIDs*, to *Object Models* and the other analysis and design work products.

For additional terms, see the "Glossary" on page 617.

Part 2. *Development Process*

We describe the work products that we have found to be effective in many object-oriented software development projects in Part 3, Work Products. This might be thought of as the "what" of a project. Part 4, Work Product Construction Techniques, describes a toolkit of techniques to help build these work products. This might be thought of as the "how" of a project. A vital and (relatively) independent dimension to project development is the development process, which might be characterized as the "when" of a project. Process is about "when" in the sense that it is concerned with the sequencing of the various project activities.

Although in a work product oriented approach there is no concept of a fixed development sequence, work products are still developed in a nonrandom order. Work products don't just happen, they are the result of planned activities. The order in which they are developed is determined by their increment and phase associations as well as by their supportive relationships with each other within a phase.

It is appropriate to discuss the sequencing of activities at a number of different levels of granularity. In Section 3.3, The Iterative and Incremental Process Model, we considered the usefulness of an iterative and incremental process in which a project is divided into a sequence of increments, each of which adds functionality to the system, and also incorporates an element of iterative rework. This is the level at which process is usually discussed. Each increment, however, constitutes a miniprocess of its own, and the sequencing (and definition) of the phases that make up each increment must also be addressed by a development process. Lastly, each phase focuses attention on a particular set of work products. How is effort on these work products to be sequenced?

There are, therefore, three levels of granularity at which it is appropriate to discuss process. These might be characterized by the following questions.

- What is a project (in terms of increments)?
- What is an increment (in terms of phases)?
- What is a phase (in terms of work products)?

This part of the book sets out to answer each of these questions in turn.

It was mentioned in Section 3.0, Iterative and Incremental Development, that no one process is applicable to all projects. Although there is no such thing as a "typical" project, it is useful to describe a model project that can serve as a reference point when attempting to discuss the constraints that one can encounter when trying to develop in the real world. Most projects will not employ this process precisely, but will bear some resemblance to what is described. They will all be "variations on a theme."

Variation itself is actually a theme of this book. The principal reason for separating the presentation of work products in Part 3, from the presentation of development techniques in Part 4, is to emphasize the prime importance of focusing on work products when deciding upon a development strategy. One should fit development techniques around the work products, and mix and match the techniques to fit the project and application context. The same applies to process. The process that is concentrated upon has been found to be appropriate to many projects. Every project is different, however, and deserves thought in its own right at the level of process. You should treat what is written here in the spirit of a reference model from which variations will inevitably have to be made to tune the model to the needs of your project. Even within the process presented here, there are many possible minor variations.

The lowest level of development activity that is described in this part of the book is the *phase*. In each phase, focus is placed on a particular set of work products. For example, during the analysis phase we are most concerned with constructing the analysis group of work products such the Analysis Object Model. The particular work products that we recommend are produced in each phase are described in Part 3, Work Products. Think of *phase* not as a single portion of time within the development process but rather as a recurring period of focus on a particular facet of development. For instance, in an iterative and incremental process, the developer "visits" the design phase several times.

Project increments and the project as a whole can be thought of as containers or packages of phases. In that sense it seems logical to start the discussion of process with phases and then to proceed to package the phases into increments and then projects. We have not opted to do that. Instead, we have chosen a top-down approach to describing process. The reason for this is that the top-level process of a project defines the relationship between increments, and hence provides context for the discussion of increments. An increment, similarly, has a certain structure, and it is within the context of this structure that it is appropriate to discuss the phases that constitute each increment.

5.0 Overall Project Structure

The highest level of process granularity is the entire project. This chapter considers how this top level of process concerns can be addressed. The two following chapters address themselves to the lower-level concerns of how increments are put together and how an increment is composed of phases that the "real" development work is done.

5.1 OVERALL PROCESS

As mentioned in Section 3.3, The Iterative and Incremental Process Model, most projects that we have seen are of the following form:

- Ill-defined, incomplete, or uncertain requirements
- Use technologies (such as object technology) or components with which the development team is not completely familiar
- Large and complex

As discussed in Section 3.0, Iterative and Incremental Development, such requirements do not lend themselves to waterfall, pure iterative, or incremental component development processes. The problem with a waterfall process is that the project risks are far too high for all development strands only to be brought together near the end of the project. There is no time budgeted for the inevitable rework. The trouble with a purely iterative approach is that while time is allowed for considerable rework, in fact this is the structure of the process. The rework required for a project of significant size is likely to result in an unacceptable amount of *design churn.* That is, a process based solely on iteration will tend to be unstable if the project is reasonably large. An incremental component process will not suffer from the problem of instability because of size, but it permits component integration, and hence testing of the overall design, too late for test results and other kinds of feedback to be incorporated. One answer is to combine the end-to-end early testing advantages of an iterative approach with the stability of an incremental component process. The resulting iterative and incremental process is the one that is described in Section 3.3, The Iterative and Incremental Process Model.

It is this rationale that leads us to recommend the iterative and incremental process as a "default" process. It will not fit all projects but it does very well for most. We suggest that it is used unless it can be demonstrated that it is inappropriate for a particular project. Section 5.4, Variations on Project Shape considers factors that might influence a project and that might indicate that a top-level process variation is required.

An iterative and incremental project consists of a sequence of increments whereby in each increment time is allowed for reworking existing software in the light of test results, feedback from end users, design reviews, et cetera. The effect of an iterative and incremental process is to bring forward in time all the testing phases of the project. It is based on the assumptions that we probably don't know all the system requirements in advance, that the requirements will change, and that we will both understand the problem domain better and discover better ways of solving problems as we go along.

5.2 PROJECT SHAPE

In practice, the number of iterations and increments to be performed on a particular set of work products will vary depending on the risk associated with that work product. For example, a project may do requirements gathering, analysis, and architectural design in a single increment, but the remaining design, implementation, and testing work in several increments and iterations. This would be appropriate for a large project where architectural design churn is considered a much greater risk than misunderstanding requirements. Many other combinations are possible. Each combination is appropriate to a particular development context; in addition, each combination determines a particular shape that the project will have.

Activities that are scheduled for the initial increment and for which no subsequent iterative or incremental effort is budgeted may be considered to belong to an initial, waterfall part of the project. A purely waterfall project is a special case since *all* activities are of this low-risk nature. (Having said that, waterfall projects usually end up with iterative and incremental development cycles for product maintenance and enhancement, even if they were not initially conceived in this way.)

As mentioned above, many variations are possible, but a project shape that is appropriate to many projects follows (see Table 5-1).

Table* *5-1*. *Typical Project Shape.

Increment	**Purpose**	**Activities**
1	Get project started	Develop all planned work products for the Requirements and Project Management phases. Review Analysis work products with the customer. Concurrently, analysts, designers, and implementers develop guidelines for their phases. Designers can also start the System Architecture, Target Environment, and Subsystem Model work products.
2	Familiarize team with process and environment	Using the Depth-First technique [8] with a few Use Cases, develop all planned work products for the Analysis, User Interface, Design, and Implementation phases.
3..n-1	Complete project development	Taking a few Use Cases at a time, develop all planned work products for each phase through Implementation. As new work products are developed, old ones may need to be extended or amended.
n	Package, test, and deliver the product	Implement the Physical Packaging Plan and conduct the Installation, System, and Acceptance test plans.

5.3 WORK PRODUCTS RELATED TO PLANNING PROJECT SHAPE

The work products that are most relevant to planning the overall development process and the project shape are discussed in:

- Section 10.1, Intended Development Process
- Section 10.3, Resource Plan
- Section 10.4, Schedule
- Section 10.5, Release Plan
- Section 10.7, Risk Management Plan
- Section 10.11, Project Dependencies

Those sections include further information, including examples and suggestions for how to construct the work products.

5.4 VARIATIONS ON PROJECT SHAPE

Table 5-1 suggests a shape for a "normal" project. Real-life risks and situations usually force variations to be made. The following list identifies some common conditions that impact real projects and usually cause a project to deviate from the typical project shape described above.[9] After each condition, we have listed some project shape variations to consider when addressing these conditions.

Weak Requirements

- Iterate on Problem Statement and Use Case Model with Customer before doing any Project Management work products.
- Jump ahead into Analysis Scenarios or Screen Flows or even a User Interface Prototype to ferret out unforeseen requirements and complexities.
- If this is a competitive situation, try developing a Use Case Model based on the competition's capabilities. Then reduce the model to what your customer needs and what you are capable of delivering.

[8] See Section 17.1, "A Depth-First Approach to Software Development" on page 363 for a description of the technique.

[9] To learn more about this topic, we suggest reading Chapter 5 in [Goldberg95].

Complex/Unfamiliar Domain

- Engage domain experts.
- During project management phase schedule more iterations and increments for the analysis phase.
- Use the scenario-driven process to define the domain from the functional requirements.
- Seek customer "buy in" with the analysis phase work products.

Complex/Unfamiliar Target Environment

- Engage technical area experts and consultants.
- Schedule education in key areas for developers immediately before design or implementation phases.
- During project management phase, identify technical risks, assign to key developer, and track resolution.
- Schedule, track, and implement as many technical prototypes as are needed to resolve the risks before it is too late.
- Schedule more iterations and increments in design or implementation phase (whichever has risks).

Complex/Unfamiliar Development Environment

- Prioritize the Development Environment work product and schedule its review and iterations to start long before the rest of the development team depends on it.
- Schedule education in key areas for analysts and developers immediately before attempting object-oriented analysis, object-oriented design, or object-oriented programming.
- Schedule more Depth-First[8] approach increments so that team gets more experience using the techniques and tools of the development environment with smaller increments of function (less risk).
- If there are many early concerns, perform a Depth-First increment for a trivial problem (perhaps unrelated to the planned product).

Parallel Development

- During project management phase, focus more attention on Resource Plan and Schedule. Iterate on these when Subsystem Model is available so that contracts with earliest or riskiest dependencies are prioritized.
- During analysis phase, focus early attention on Subject Areas work product to identify potential parallel analysis and development of subsystems.

- Start System Architecture work product early to start capturing intent to structure system for parallel development (nonfunctional requirement).
- Start Subsystem Model work product early to focus on intersubsystem (intergroup) contracts.
- During the design phase, emphasize the Subsystem Model and API work products.
- Other than these coordinating activities, treat the development of each subsystem as if it were a complete system. That is, develop a separate project workbook for each subsystem and use the complete process on each subsystem allowing tailoring as needed.
- Use an iterative and incremental approach that requires periodic integration testing to validate subsystem APIs and contracts.
- Be especially vigilant if subsystems are to be subcontracted out.

Short Schedules

- Don't plan too many increments. Limit it to two or three.
- Favor prototyping over formal increments to address risks.
- When developing the Project Workbook Outline, focus on those work products that are most effective in attaining your goals.
- When developing Analysis, Design, and Coding Guidelines, consider the cost and benefit of each given your schedule.

Long Schedules

- Long schedules don't mean lax schedules. Long schedules require more details and attention to risks than shorter ones.
- Pay special attention to scheduling risk-resolving prototypes.
- Take advantage of the opportunity to fit in more meaningful increments and iterations.
- Allow time for injecting additional increments especially midway and near the end of the development.
- Early and periodically in the project management phase, look for parallel development opportunities. If you were offered another team or ten more developers, how would you use them?
- If you lost a team, how would you adjust? Plan for contingencies before they are needed.

Reverse Engineering and Re-engineering Projects

- Here's a case when the Waterfall approach, a single increment of development, may work the best.
- If risks are low, plan to use the simple, efficient Waterfall approach.
- Be selective when considering work products for Project Workbook Outline and when considering guidelines for each phase.
- Produce all the work products that you planned to do.

6.0 Project Increments

A project increment is a miniproject, so it should be no surprise that planning one is like planning a miniproject. You should consider all the possible work products that can be developed in light of the constraints implied by the increment. Here are some things to consider:

Purpose Each increment is planned to accomplish a specific set of goals. They should be documented so that the participants can share a common vision. The reasons for this are the same as those that require the whole project to have a Problem Statement. But here, inside an increment, the purpose should be simpler since its goal is to focus a small part of the project for a shorter time.

Scope Each increment should be limited by the selection of a specific set of:

- Use Cases and Scenarios
- Subject Areas or Subsystems
- Development phases that will be carried out
- Work products that will be produced
- Completion dates (Schedule)

Guidelines A project should decide whether the work products and guidelines that govern them should evolve throughout the product's development, or whether they should address the final criteria from the start.

Process Besides the criteria implied by guidelines, each increment may employ a variation on the process used for its duration. For example:

- Will there be iterations within the increment?
- In which order will the work products be developed? (Which are the prime and which are the subordinate models?)
- How often will work products be reviewed or verified? (as soon as they are completed, at end of phase within the increment, or at the end of the increment, et cetera.)
- Besides work product verification, how will any product deliverables be tested? Will there be an independent test team to integrate and test new and extended components (exercise the scenarios specified in the scope)?

6.1 TYPICAL PROJECT INCREMENTS

We address the preceding considerations for the typical project increments identified in Table 5-1 in the following four tables.

Table 6-1. *Project Initiation Increment.*

Purpose	Get the project started, fully understand the requirements and the problem domain, lay the foundation for design and implementation to begin.
Scope	Develop all intended Requirements, Project Management, and Analysis work products. Concurrently develop (or select) User Interface, Design, and Coding Guidelines. Develop first pass of System Architecture, Target Environment, and Subsystem models. Allocate 25 percent of schedule for this increment.
Guidelines	All Requirements, Project Management, and Analysis work products should comply with predetermined guidelines right from the start.
Process	All work products will be verified as they are completed. Their owners should allow for a couple of iterations within the scheduled completion dates. The phase review should simply verify that all critical Issues have been closed and others assigned to owners with appropriate due dates. The Acceptance Plan and Use Case Model will be verified with the customer at the end of the phase.

Table 6-2. *Developer Familiarization Increment.*

Purpose	Familiarize the development team with the new processes, techniques, and tools that they will need to use in this project. Shake down the process and development environment.
Scope	Using the Depth-First technique with a small number of simple Use Cases, develop all the planned work products for the Analysis, User Interface, Design, and Implementation phases. Excluding education and training, only allocate a couple of weeks for this..
Guidelines	Try to comply with the guidelines developed for each phase, but don't get stuck trying. If the guidelines need adjustment, this is when that should be determined.
Process	All work products will be verified as they are completed. Their owners should allow for a couple of iterations within the scheduled completion dates. There will be no phase reviews, but the increment review will focus on identification of Issues associated with the development process, environment, and tools.

Table 6-3. *Normal Development Increment.*

Purpose	Complete the development of the product.
Scope	Selecting a significant number of related Use Cases, develop all the planned work products for the Analysis, User Interface, Design, and Implementation phases. Allocate 50 percent of the schedule for several increments of this type.
Guidelines	All Analysis, Design, and Implementation work products should comply with predetermined guidelines right from the start.
Process	All work products will be verified as they are completed. Their owners should allow for a couple of iterations within the scheduled completion dates. The phase review should simply verify that all critical Issues have been closed and others assigned to owners with appropriate due dates. The User Interface Prototype will be verified with the customer at the end of User Interface design phase.

Table 6-4. *Release Increment.*

Purpose	Package, test, and deliver the product.
Scope	Implement the Physical Packaging Plan and conduct the Installation, System, and Acceptance Test Plans. Allocate 20 percent of the schedule for several iterations of these activities.
Guidelines	All Implementation and Test work products should comply with predetermined guidelines right from the start.
Process	All work products will be verified as they are completed. Their owners should allow for a couple of iterations within the scheduled completion dates. The phase review should simply verify that all critical Issues have been closed and others assigned to owners with appropriate due dates.

6.2 WORK PRODUCTS RELATED TO PLANNING INCREMENTS

The work products that are most relevant to planning increments are:

- Section 10.1, "Intended Development Process" on page 127
- Section 10.3, "Resource Plan" on page 135
- Section 10.4, "Schedule" on page 139
- Section 10.5, "Release Plan" on page 144
- Section 10.6, "Quality Assurance Plan" on page 147
- Section 10.7, "Risk Management Plan" on page 152
- Section 10.9, "Test Plan" on page 164

Those sections include further information, including examples and suggestions for how to construct the work products.

6.3 VARIATIONS ON INCREMENTS

6.3.1 A Set-Up Increment

Set-up increments are similar to Depth-First increments except that the *purpose* of the former focuses more on shaking down and learning the development process, techniques, environment, and tools. The *scope* is limited to only those phases, work products, tools, and techniques that are new to the development team. The Use Cases and Scenarios don't have to come from real requirements. The Schedule is usually very tight unless this is being run concurrently with the Project Initiation phase. The *guidelines* are usually slackened and the *process* is aimed at finding development environment and process problems.

6.3.2 A Depth-First Increment

A Depth-First increment differs from a Set-up increment in that its *purpose* is to do a small amount of real development (it counts) while learning the development process, techniques, environment, and tools. The *scope* can go from analysis scenarios through implementation or it can start from design scenarios. It should use real Use Cases and Scenarios. The Schedule is usually short but not as short as a Set-up increment. The phase *guidelines* are respected but can be questioned. The *process* should be as close to normal development as the initiates can bear.

6.3.3 Release 1.0 Increment

It may sound strange to have a *Release 1.0 Increment*, but considering that it usually follows a *Beta* release of the product, it is a special type of increment.

Its *purpose* is to finish the development that didn't make it into the Beta release and to clean up the many problems often discovered with the Beta release. The *scope* is usually limited to implementation and test work products, but can occasionally go back as far as design scenarios. There are often Target Environment assumptions and outcomes that were not considered before the product entered the real world. The Schedule is usually very short since a published *ship date* is usually at risk. The phase *guidelines* are respected but are occasionally deferred when "missing the date" is a possibility. The *process* should be as close to normal development as the weary developers can bear.

6.3.4 Release 1.1 Increment

Sometimes software products "ship by definition"; that is, before they are ready. Release 1.1 is what the product developers usually intended to ship and what the customer expected. The Schedule for Release 1.1 is usually chosen so that the customers will automatically receive it before they put Release 1.0 into production (hopefully, before the customer dared to install it).

Release 1.1's *purpose* is usually to fix "known bugs" that could not be addressed in Release 1.0 without impacting the ship date. Therefore, the *scope* is limited to implementation and test work products. No new requirements or "features" are allowed into this increment. The Schedule is always very short. The phase *guidelines* are respected but are occasionally deferred when "missing the date" twice is a possibility. The *process* should be as close to normal development as the ready-to-quit developers will tolerate.

6.3.5 Release 2.0 Increment

For Release 2.0, development should be back to normal. The only difference is that there is a base legacy to protect and improve upon. All work products from previous releases are now assets to work from (though some may seem like liabilities). Release 2.0 usually incorporates significant new function or changes in design to better address Nonfunctional Requirements (e.g., performance or usability).

Release 2.0's *purpose* is to significantly extend or change the design of Release 1.x. If the risks are small and the process, Development Environment, Target Environment, and Architecture are well understood, this can be developed in a single, waterfall, increment. Otherwise use the increments defined for the typical project shape described earlier.

In any case, these increments should clearly enumerate all goals it is expected to attain. Therefore, the *scope* will include all phases, but will concentrate on Requirements and Analysis work products if the main purpose is to *extend* function of the product, and will concentrate on Prototyping and Design work products if the main purpose is to *change* the design of the product. The Schedule is usually shorter than the original release. The phase *guidelines* are respected and often improved as the veterans gained religion since their first project. The *process* is an experience-based improvement of what was normal development during the first release.

7.0 Development Phases

A development phase is a state of product development that focuses on making progress on a particular aspect or facet. In this book we have chosen to focus on the following development phases:

- Requirements Gathering
- Project Management
- Problem Analysis
- User Interface Design
- System Design
- Implementation
- Testing

We realize that there are other phases that may precede or follow these such as market analysis, project initiation, marketing, deployment, maintenance, withdrawal, and harvesting reusable components, but we have chosen to limit our scope to those phases of interest to *software developers*.

Although the list of phases may remind some of the waterfall approach to development, we strongly recommend that you view them as periodic phases. They may be entered repeatedly to add new functionality to the system (incremental development) or to rework previous work products to correct or improve them (iterative development). In general, the phases discussed below are presented in the order that you will encounter them in a normal project, but there will always be reasons to delay or accelerate development of certain work products normally associated with a particular phase.

The following diagram provides a pictorial overview of the development phases and their relationships with each other and with key development repositories. It is intended to show that:

- Requirements Gathering and System Test are pre and post-iterative phases, respectively.
- Analysis, Design and Implementation phases occur in an Iterative and Incremental process.
- Analysis needs to be performed before Design.
- User Interface and System Design can be done concurrently.
- Design precedes Implementation.
- Project Management is an on-going activity.

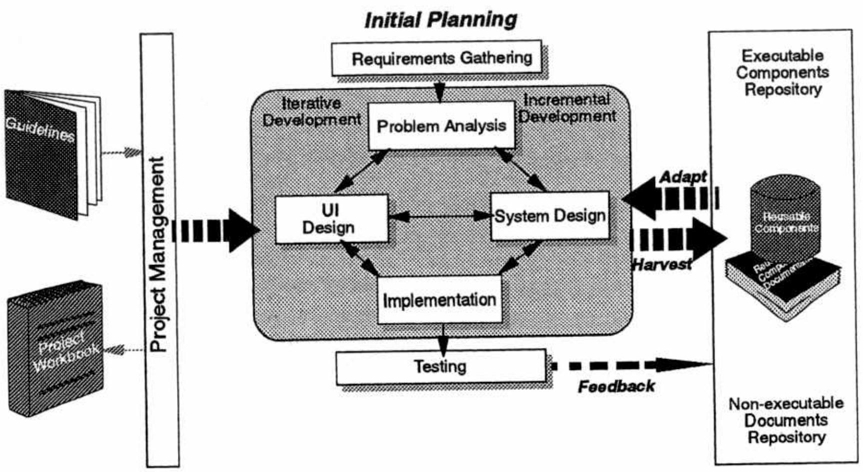


Figure 7-1. Development Life Cycle Overview.

The following sections provide a description of the development phases discussed in this book.

7.1 REQUIREMENTS GATHERING

Requirements are crucial because they provide a scope and a boundary for later activities, especially planning, analysis, and design.

Poorly understood requirements can severely impact the development efforts of the project team. There are two major kinds of requirements that are important to the object-oriented development process:

- ***User Requirements*** describe what the needs of the intended users are. User requirements drive analysis. Our approach to object-oriented development strongly recommends that these requirements be expressed by Use Cases.
- ***Nonfunctional Requirements*** that address concerns such as:
  - *Performance:* requirements that clarify the space, time and other system resource constraints the customers require of the application. The performance requirements drive activity during the design phase.

  - *Platform:* requirements that detail the desired "delivery vehicles" for the system, such as hardware and target language, et cetera. Platform requirements drive both the interface specification and implementation activities.

Nonfunctional Requirements drive design. If Nonfunctional Requirements are not known during the early phases of a project, they can be deferred provided they are known at design time. Design time commences when design decisions need to be taken as a precursor to development. At this time, the Nonfunctional Requirements must be nailed down in order to permit the development team to get started.

7.2 PROJECT MANAGEMENT

Project Management is ongoing throughout the life of a project. As a practice, its concerns are not much different from a traditional effort. Some of the differences are:

- Managing iterative development that is new to some managers.
- Understanding the object-oriented work products across analysis, design and implementation.
- Coming up with initial sizing of a software development effort.
- Bringing people up the object-oriented learning curve.

These will all improve as our experience deepens with object-oriented technology.

7.2.1 Initial Project Planning

This aspect of the project management phase usually occurs before the requirements gathering phase. Its purpose is to identify the need and objectives for the project. The development organization must decide whether or not to proceed with the project and a decision must be made as to whether or not to use object-oriented technology.

Initial sizings will usually be done at this time and used for purposes of beginning the pursuit of funding.

The process of staffing will get under way as well. In a project using object technology, particularly for the first time, a critical activity will be to do a skills assessment (or technology maturity assessment) of the likely time. This assessment should determine the skills needed to do the project (including object skills, domain skills, development environment skills et cetera). You should then begin to make plans to fill any gaps through staffing, education, and consideration of the use of mentors or consultants. If outside expert help will be used in the form of consultants, this is the best time to bring them in.

The nature of initiating a project can vary widely depending on the culture and processes of the organization doing the development. During initial planning, most of the work products developed are business as usual and are not specific to object-oriented technology. The biggest challenge most organizations face at this time is overcoming the challenge of estimating effort. Because object-oriented projects are built iteratively and incrementally, sizing the total effort required often proves a serious challenge. The best advice to follow on a first-time project is to size it using your business as usual approach (i.e., as if you were using your traditional, non-object-oriented development approach). Use this as a strawman and revise it as you proceed.

7.2.2 Organize Project Plan

To some, the question "Which comes first, the requirement or the plan?" is much like the chicken and the egg problem. Since gathering requirements is an activity that requires project resources, it should be planned. However, the project plan is driven by the requirements, because its main purpose is to:

- Group the requirements into releases that will be delivered as a unit;
- Define the development processes and activities; and,
- Allocate project resources to the activities.

A lot more goes into a project plan, the gory details of which are described in the associated workbook (Section 10.0, Project Management Work Products). The challenge is to complete it quickly without getting in the way of the real work to be done! Unfortunately, this pressure to get things moving to avoid "plan paralysis" (similar to analysis paralysis) can cause "plan churn" to occur. Plan churn can cause everything to come to a screeching halt while the details change again and again, or worse, create the need to throw away work that has been done and start over.

Solving these two problems is where the iterative and incremental development process comes in. Just as "time prevents everything from happening at once," an iterative and incremental development process eliminates the need to plan everything at once.

The use of planned iterations, where discoveries made during later activities cause changes to the work products of previous ones (with a subsequent "ripple" effect), can relieve most of the pressure of having to create a perfect plan (or analysis/design model, et cetera) up front.

7.2.3 Maintain Project Workbook

Quality documentation is a critical success factor for large-scale object-oriented development. Although it may be arguable that a small one-person software development effort can be done in a person's head, any nontrivial project requires accurate and timely documentation at every stage. Documentation is the key to technical project communication. Without it there is no project.

The project workbook communicates to the project its plans, its decisions, and its progress. Timely and accurate work products consolidated in the project workbook are vital. An up-to-date project workbook allows for movement of personnel in, out, and around the project. A project workbook is tailorable to any size and type of project, but it all starts out by defining a Project Workbook Outline.

Good project workbooks make for successful projects.

7.3 PROBLEM ANALYSIS

Analysis is about domain understanding and is essential to good object-oriented development. Object-oriented analysis tells us what objects are part of a domain as well as their attributes and behaviors. Object-oriented analysis consists of techniques and work products that identify objects that are relevant to the problem being solved. The process includes classifying the objects and finding relationships among them.

During object-oriented analysis we apply techniques to understand, develop, and communicate user requirements for an application. The analysis phase focuses on clarifying and representing requirements in a concise manner at a more detailed level than during requirements gathering. Many object-oriented analysis techniques are graphical in nature and involve work products that contain diagrams.

To summarize then, object-oriented analysis is concerned with understanding the problem domain. This involves:

- Identifying objects and their attributes
- Learning about how objects behave and their responsibilities
- Understanding how objects interact with each other

Analysis must be done with persons who know the domain: for example, if building an application for Insurance Underwriters, then an Insurance Underwriter should be involved in the analysis process.

Analysis should be done in a constraint-free fashion. Avoid letting implementation constraints impact analysis.

1. Constraints can and do impact domain understanding.

   Knowing that a database will be implemented in a relational database like DB2 should not affect analysis, but often developers will build relational data models during analysis as a short-cut to implementation. They then come to think about the world in terms of tables that they will be building eventually. This is a mistake. Do not let the final design or implementation influence the way you think about, discuss, or model the domain.

2. Design detail clutters and obscures analysis.

3. Adding design detail to analysis would make it difficult for users to understand and validate.
4. Adding design or implementation detail into an analysis model makes it less generic and thus less reusable.
5. Implementation constraints can change (for example, platform or database).

These changes should not invalidate your analysis.

Analysis must be independent of the delivery environment. If the planned deployment changes from MVS to OS/2 in a client-server environment, the analysis work products should not change. The objects that are meaningful to users in their domain will not change if the implementation environment changes. This is important if there is any hope of reusing the analysis work products. The benefits are great if this approach is adopted. Deployment decisions such as hardware platform or development software can change over time. By keeping the analysis independent of these considerations, the analysis work products can be reused, regardless of the production system. There is a risk that deployment decisions can affect the application analysis. It is tempting during analysis to say, "it is a given that we are going to represent our data in DB2, so let's just represent things in tables." Also, there is a danger that this may impact on the analysis by leading to the attitude that "we can't do that in DB2, so let's ignore it." These considerations mitigate against achieving a true understanding of the application domain, which is the purpose of analysis. Analysis should be constraint-free.

7.4 USER INTERFACE DESIGN

User Interface design is concerned with planning the construction of software that has an intuitive and standardized human-machine interface. Today, this usually means the layout, appearance, and flow of control involving a graphical video display with "windows."

While it is possible to consider the User Interface design as being part of system design, it is treated in isolation to allow for a separation of concerns that is frequently exercised in real-world software development. User interface work is usually done before and independent of the system design phase, often by a team of user interface specialists within the project.

Often, User Interface design is done in parallel with the analysis phase. In such cases, coordination between the user interface and analysis teams is important.

When the "friendliness" (easiness, intuitiveness, and lack of surprises) of a user interface is critical to the success of the products, a User Interface Prototype is built to get user feedback on that friendliness and to better understand the user's conceptual model of how it should work.

7.5 SYSTEM DESIGN

While during analysis we focused on problem-domain objects, during design we focus on solution domain objects. Some of the classes that existed in the analysis model will disappear (as they won't be implemented) and new ones will appear specific to the System Architecture and Target Environment (e.g., a DB2 interface class).

Many teams have difficulty discerning between analysis and design and want to skip the analysis phase and go straight to design. In an object-oriented project, it is essential that analysis not be bypassed.

Design is concerned with how an application will be built and involves factoring in Nonfunctional Requirements such as:

- Platforms
- Languages
- Performance
- Interoperability
- Persistence
- Maintainability
- Use of standard components and subsystems
- Reuse
- Cost
- Time

Much of design is involved with addressing these constraints. It often involves weighing trade-offs, as many of the design considerations are mutually exclusive (for example, high functionality with small memory).

By separating analysis and design we are able to make the analysis more generic and reusable, so that when design decisions change, the analysis model will still hold. For example, in the banking industry, the way that a Bank Loan is decided upon should be the same regardless of the software tools used to implement a lending application.

If we construct hybrid analysis/design work products, it is difficult to identify what represents business requirements and which are introduced at design time for implementation reasons.

7.6 IMPLEMENTATION

Implementation involves the transformation of design work products (detailed plans for solution) into compilable software (Source Code) and other product deliverables. In object-oriented development, the bulk of implementation effort is concerned with creating class implementations in an object-oriented programming language that supports the specifications recorded in the Design Class Descriptions. Although the Design Class Descriptions focus on external characteristics, they often include information concerning the internal states, operations (method descriptions), and representation. When they don't, the implementer need only access other design work products, especially the Design Object Model and the Design Object Interaction Diagrams.

In a scenario-driven approach, the implementer will benefit from following the traceability of functional requirements as Use Cases lead into Design Scenarios and Design OIDs. The results are organized into classes that are described by the Design Object Model and the Design Class Descriptions.

In an iterative and incremental process these work products evolve, so the implementer's task is to match that evolution in the class's Source Code. The incremental implementations can be tested by applying the Design Scenarios (with their assumptions) and verifying the outcomes. This completes a typical development increment.

As a scenario takes a thin slice of the problem and represents an end-to-end solution, at the end of an iteration a subset of the application has been developed. After a number of iterations, the functionality builds up gradually to encompass more and more of the requirements.

A benefit of this approach is that the developers can work from the Object Interaction Diagrams (OIDs) that directly convey requirements in a form that is usable during implementation. This still gives a developer a fair amount of latitude—the OID represents a form of contract between the collaborating classes. The messages being passed back and forth tell the implementer of a class what its responsibilities are, but not how to fulfill them—the class remains a black box.

Another interesting aspect of this approach is that it results in "just-in-time programming." Classes and their methods are implemented to fulfill the requirements of a set of scenarios, but no more.

7.7 TESTING

Testing within object-oriented projects is a bit different from testing traditional software. Unlike traditional development, testing is not a phase that occurs only after the completion of development.

In object-oriented projects discrete testing phases, similar to traditional software development, still occur. Depending on what sort of testing you are used to doing, you may still perform System Test, Integration Test, User Acceptance Test, Stress Test, and Performance Test. The difference is that you may perform some of these more frequently on subsets of the system. For example, at the end of each development increment, the executable code produced by the increment must be unit and function tested by running Test Cases that test the function specified by the scenarios designated to be developed in that increment. After *implementing the entire application*, traditional function and system tests can be run using executable "white" and "black" box Test Cases developed from Scenarios.

Thus testing is much more of an ongoing activity. This is one of the strengths of object-oriented application development and greatly lessens the risk of building the wrong application (by virtue of the ongoing feedback coming back from the testing).

In addition to the normal (but more frequent) testing of executable code, there are other activities that must occur within each phase of the development process that might be considered to be testing activities. We refer to these activities as *validation* and *verification* of work products. Validation consists of establishing that the right work product has been built. Verification consists of checking that the work product has been built correctly. Validation and Verification are so important to the object-oriented development process that they are a fundamental part of each work product's development in our approach.

For purposes of this book, we will consider testing to mean the traditional testing of product executable code, and we will treat validation and verification as an integral part of developing a work product. This is discussed in Section 8.3, Validation and Verification of Work Products. You will also find advice on verification of work products in each work product description.

7.8 SUMMARY OF DEVELOPMENT PHASES

The following table shows a summary of the life-cycle phases.

Table 7-1 (Page 1 of 2). Development Phase Summary.

Phase	Activity	Work Products
Requirements Gathering	Group requirements into Use Cases and prioritize by importance, window of opportunity, and technical complexity.	Problem Statement, Use Case Model, Nonfunctional Requirements, Prioritized Requirements, Business Case, Acceptance Plan
Project Management	Allocate requirements to releases and/or increments and plan activities that manage resource availability and other project constraints.	Intended Development Process, Project Workbook Outline, Resource Plan, Schedule, Release Plan, Quality Assurance Plan, Risk Management Plan, Reuse Plan, Test Plan, Metrics, Project Dependencies, Issues
Problem Analysis	Develop solutions to scenarios in terms of active objects that group related tasks and communicate with other objects in order to complete them.	Analysis Guidelines, Subject Areas, Object Model, Scenarios, Object Interaction Diagrams, State Models, Class Descriptions
User Interface Design	Document how users will interact with the application.	UI Guidelines, Screen Flows, Screen Layouts, UI Prototype
System Design	Plan a solution to the problem examined during analysis in terms of interacting objects, within the constraints specified by the Nonfunctional Requirements.	Design Guidelines, System Architecture, APIs, Target Environment, Subsystem Model, Object Model, Scenarios, Object Interaction Diagrams, State Models, Class Descriptions, Rejected Alternatives

Table 7-1 (Page 2 of 2). Development Phase Summary.

Phase	Activity	Work Products
Implementation	Systematically code the classes specified as a result of design in a programming language according to documented public/private interfaces so that they can be built and installed on the target platforms.	Coding Guidelines, Physical Packaging Plan, Development Environment, Source Code, User Support Materials
Testing	Insure that the application meets the requirements set forth in the Problem Statement and Requirements.	Test Cases

Part 3. *Work Products*

Project workbooks contain work products. This section of the book describes work products that we recommend be built during the course of an object-oriented software development project. They are grouped according to the sections described in Section 2.4, "Workbook Structure" on page 20.

Table 8-2 (Page 1 of 2). Work Product Traceability.

IMPACTED BY	IMPACTS																																															
IMPACTS IMPACTED BY	Problem Statement	Use Case Model	Nonfunctional Requirements	Prioritized Requirements	Business Case	Acceptance Plan	Intended Development Process	Project Workbook Outline	Resource Plan	Schedule	Release Plan	Quality Assurance Plan	Risk Management Plan	Reuse Plan	Test Plan	Metrics	Project Dependencies	Issues	Analysis Guidelines	Subject Areas	Analysis Object Model	Analysis Scenarios	Analysis OIDs	Analysis State Models	Analysis Class Descriptions	User Interface Guidelines	Screen Flows	Screen Layouts	UI Prototype	Design Guidelines	System Architecture	APIs	Target Environment	Subsystems	Design Object Model	Design Scenarios	Design OIDs	Design State Models	Design Class Description	Rejected Design Alternatives	Coding Guidelines	Physical Packaging Plan	Development Environment	Source Code	User Support Materials	Test Cases	Glossary	Historical Work Products
Problem Statement		◆	◆	◆	◆															◆																									◆		◆	
Use Case Model				◆		◆									◆					◆		◆					◆	◆	◆																◆			
Nonfunctional Requirements				◆																						◆	◆	◆	◆		◆	◆	◆	◆	◆	◆	◆	◆							◆			
Prioritized Requirements						◆			◆	◆	◆		◆																		◆																	
Business Case									◆	◆	◆			◆																																		
Acceptance Plan															◆																																	
Intended Development Process										◆	◆								◆							◆				◆											◆							
Project Workbook Outline												◆							◆							◆				◆											◆							
Resource Plan										◆	◆																																					
Schedule									◆		◆																																					
Release Plan									◆	◆																																						
Quality Assurance Plan							◆	◆							◆				◆							◆				◆											◆							
Risk Management Plan										◆	◆																																					
Reuse Plan									◆	◆	◆						◆		◆											◆	◆			◆								◆		◆				
Test Plan							◆		◆	◆		◆							◆											◆																◆		
Metrics									◆	◆		◆			◆																																	
Project Dependencies									◆	◆	◆		◆																		◆			◆								◆						
Issues	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆		◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆	◆
Analysis Guidelines								◆												◆	◆	◆	◆	◆	◆																							
Subject Areas									◆																																							
Analysis Object Model																				◆			◆	◆	◆										◆												◆	
Analysis Scenarios																							◆	◆												◆											◆	
Analysis OIDs																				◆	◆			◆	◆												◆											
Analysis State Models																							◆		◆													◆									◆	
Analysis Class Descriptions																																							◆									
User Interface Guidelines								◆																			◆	◆	◆																			
Screen Flows															◆													◆	◆							◆		◆	◆							◆			
Screen Layouts																											◆		◆							◆		◆	◆							◆			
UI Prototype																											◆	◆																				
Design Guidelines								◆																							◆	◆	◆	◆	◆	◆	◆	◆	◆									
System Architecture																																◆	◆	◆	◆	◆	◆	◆	◆	◆			◆		◆			
APIs																																												◆	◆	◆		
Target Environment																															◆										◆	◆	◆		◆			
Subsystems																															◆											◆						
Design Object Model																																		◆			◆	◆	◆								◆	
Design Scenarios																																					◆	◆									◆	
Design OIDs																																			◆			◆	◆									
Design State Models																																					◆		◆								◆	
Design Class Description																																											◆				
Rejected Design Alternatives																																															◆
Coding Guidelines								◆																																			◆				
Physical Packaging Plan																																											◆	◆			
Development Environment																																									◆						
Source Code																																																
User Support Materials																																																
Test Cases																		◆																														
Glossary																																																
Historical Work Prodcuts																																																

Table 8-2 (Page 2 of 2). Work Product Traceability.

Note:

- Subsystems impact to subsystem workbooks install procedures.
- Historical work products should relate to their ancestors and descendants.
- Subject Areas work product impacts the workbook of each Subject Area.

Legend:

Work products listed at left impact work products listed at top when a ◆ appears at the intersection, for example:

- Problem Statement impacts Use Case Model, Nonfunctional Requirements, Prioritized Requirements, Business Case, Subject Areas, User Support Materials and Glossary.
- Alternatively, Analysis Object Model is impacted by Issues, Analysis Guidelines, and Analysis OIDs.

9.0 Requirements Work Products

The requirements chapter of the workbook represents the specification of the project. As such it is an important part of the contract to which both customer and development team bind themselves. The term "customer" is used here in its accepted sense, even though it is understood that many development projects may not have a direct customer. In this case the term should be understood to mean someone who is acting on behalf of customers of the product in order to determine and validate requirements. This might in practice be a potential user of the system, a domain expert, or a development team member nominated to represent the technical interests of customers.

Customer involvement in some form or other is an important part of object-oriented development, and a central part of requirements gathering in particular. The contract between the development team and the customer may be formal or it may be informal but the concept of an agreed contract is nevertheless important. Whether formal or informal, it is obviously important that all parties agree in advance on the task to be performed. It is vital that the requirements are expressed clearly, simply, and unambiguously. Both customer and development team must understand the requirements, and it must be obvious that their understandings are identical. Poorly understood or constantly changing requirements are a frequent cause of project failures. Achieving a common understanding of requirements gives a project a much better chance of success.

The requirements section of the project workbook consists of the following work products:

- Problem Statement
- Use Case Model
- Nonfunctional Requirements
- Prioritized Requirements
- Business Case
- Acceptance Plan

These work products all "inherit" the common work product attributes described in Section 8.1, in addition to which they have specialized attributes of their own. The work products and their specialized content are defined and commented upon in the following sections.

The first step towards a complete, formalized set of requirements is a Problem Statement, which is a succinct statement of the problem that the system is intended to solve.

A complementary work product is the Business Case, which presents the commercial (or some other) justification for the project from the point of view of the development organization.

The Problem Statement and Business Case together act as important points of focus for the subsequent requirements-gathering activity.

There are many aspects to the requirements themselves. Broadly, requirements are divided into functional requirements, which specify what the system is to do, and Nonfunctional Requirements, which specify constraints on the system. Examples of nonfunctional constraints are reliability and performance.

The work products of the requirements project phase are not directly related to the use of an object-oriented software development approach at all. By their nature, they concentrate on the system interfaces, which after all, is how the system is going to be judged. How the system will work internally is not of direct concern to the customer. Having said that requirements are not directly related to the use of object technology, some forms of requirements specification turn out to be more convenient starting points for object-oriented development than others. One form that is very useful in practice for capturing functional requirements is the Use Case Model. Other forms of requirements work products may additionally, however, be necessary in order to conform to organizational development standards. For example, what is traditionally known as a functional requirements document is a list of the features that the system will support. This kind of document usually duplicates material presented in a Use Case Model. If possible, the two should be brought together, for example by using the Use Case Model work product as the format for the functional requirements document.

The Prioritized Requirements work product defines the overall priorities of the system requirements by looking at both the functional requirements, represented by the Use Case Model, and Nonfunctional Requirements.

The final requirements work product is the Acceptance Plan, which is the way in which the customer agrees to decide whether the system does indeed satisfy the requirements. The Acceptance Plan closes the contractual loop and binds the technical requirements to the contract of which it is a part.

Of course, the set of requirements work products described above is not necessarily sufficient to satisfy a development team that the project is feasible within the constraints. Only by actual development or prototyping effort, and other means such as adopting and exploiting an iterative and incremental schedule, may the risks of commitment to a set of requirements be reduced; almost certainly they cannot be eliminated entirely. The point being made is that the existence of particular requirements work products in this chapter does not mean that *only* these should be produced prior to a developing team committing to develop a solution that addresses the requirements. How much development or prototyping effort should be expended before commitment is as much a political and economic question as a technical one.

Similarly, a customer may require that considerable analysis is performed before agreeing to a requirements document. There may also be projects that are so complex that much analysis is needed to gain a good understanding of the requirements work products. The requirements chapter of this book contains only those work products that are unique to requirements gathering. The process of requirements gathering may in addition necessitate the production of work products that are normally associated with other project phases, and hence documented in other chapters of this book.

9.1 PROBLEM STATEMENT

Description

A Problem Statement should describe the *business* requirements of the application to be developed. These are not functional requirements but a short description of the business problem that the application should solve. The Problem Statement does not say what the application should be or do, but instead concentrates on why it is needed at all.

It should be written by people familiar with the domain and the focus should be on the business needs of the intended users. Design and implementation topics should not be present; however, it is valid to address needs such as integration with an existing environment.

Purpose

This is done to communicate to everyone involved what the project objectives are. By writing a Problem Statement, you ensure common understanding and agreement. While it may surface issues or areas of disagreement, the sooner this is done the better.

Participants

The customer writes the Problem Statement. It may be written by an executive, or a project manager, or someone paying for the development effort.

Timing

Preferably before any work on a project begins.

Technique

Have the customer write a brief text answering:

1. What are we trying to accomplish?
2. Why are we making this effort?

Do not be concerned with how the problem will be solved.

Strengths

- A Problem Statement provides a way of determining whether everyone connected with the project agrees on its objectives.
- It can put the project into context for the development team.
- Problem Statements help to identify candidate objects (nouns) and behaviors (verbs).

- They are good for determining application boundaries.
- They usually generate questions about scope, function, et cetera.

Weaknesses

- Problem Statements are rarely complete and are often incorrect.
- Achieving consensus on the Problem Statement can be challenging.
- It can be difficult to write for a project that is aimed at creating something new, rather than automating or improving some existing process or service. If new roles are being created, it is often difficult to find someone able to articulate what the new world will look like.

Notation

Free format text.

Traceability

This work product has the following traceability:

Impacted by:
- Issues (p. 176)

Impacts:
- Use Case Model (p. 96)
- Nonfunctional Requirements (p. 106)
- Prioritized Requirements (p. 111)
- Business Case (p. 115)
- Subject Areas (p. 187)
- User Support Materials (p. 341)
- Glossary (p. 355)

Advice and Guidance

1. The Problem Statement should be reviewed at the first team meeting when the project is initiated.

   It is important that the team have a shared understanding of what the business purpose of the project is. Look for and record issues.

2. Completeness and correctness

   Note that a Problem Statement is rarely complete or correct. It may be written by someone who may not have a working knowledge of the day-to-day operations of a particular area. Therefore, you should always review the Problem Statement with people working in the domain.

3. Avoid things like "be best of breed" unless competitive analysis is an aspect of the project (i.e., it can be clearly articulated what it means to be "best").

4. Consensus

   When the project is started, there may be a lack of consensus or even disagreements on the accuracy of the Problem Statement. This may happen because the sponsor is not familiar with the day-to-day operations of the business problem being addressed. If this is the case then a consensus must be achieved. The team may wish to rewrite the Problem Statement and then review it with the project sponsor(s).

   It may also be that there is not a consensus within a business area and this must be addressed and addressed. While this can be time consuming, the alternative (nonconsensus) is not desirable.

5. Additional information this work product should contain:
   - Objectives
   - Highest priority problem (may be useful when weighing trade-offs)
   - Intended users

6. The typical Problem Statement is between one-half and two pages in length.

Verification

- Check that the problem and not the solution is described.
- Check that the Problem Statement includes an explanation of why a solution is needed (preferably from a business perspective).

Example(s)

The following is a Problem Statement from an object-oriented project to develop an Education Administration application:

> Our company spends over $5 million a year on education for our employees, but we don't know where the money is going or if it is being spent effectively. We are not sure if we are getting value for our money (for example, is the education being purchased helping us to meet our company's strategic objectives?).
>
> There is a concern in management that some employees are getting too much or too little education. Some staff are required to maintain their professional standing by completing certain courses. Others are expected to attend courses as part of their career development.
>
> We are looking for some means of tracking and measuring the effectiveness of our education spending. It has been suggested that a means of doing this might involve creating a catalogue and tracking staff's education and doing evaluations and follow-ups.
>
> An external consultant has suggested that this education tracking system should be tied in to our Human Resources system.

References

See [Rumbaugh91a] for discussion and an example of a Problem Statement.

Importance

Essential. It is critical for the entire team to understand the reasons for the project. If you can't articulate this, then the project could be in serious trouble.

9.2 USE CASE MODEL

Description

A Use Case Model is a convenient form in which to express top-level functional requirements. A Use Case Model consists of a set of actors (representing external agents), Use Cases (representing usages of the system by the actors or vice versa), and links between the actors and the Use Cases.

A Use Case Model is the central part of a requirements document for the object-oriented development approach recommended in this book. It states what the proposed system is to do. This is in contrast to the Nonfunctional Requirements that impose constraints on the system: performance, reliability, availability, et cetera.

A Use Case describes a particular, observable, system behavior. An observable behavior is one that is visible externally via a system interface. The set of observable behaviors define the functional system requirements. Use cases are guaranteed to be observable by the fact that they must be connected to one or more actors. Examples of Use Cases are *Query bank account* and *Aircraft enters controlled airspace*. Examples of actors are *Customer* and *Radar*. Actors may be human users or other systems such as database management systems.

A Use Case Model and the Analysis Scenarios (see Section 11.4) which are derived from it together constitute a complete set of functional requirements. Particularly if the same person or team is responsible for both the Use Case Model and the Analysis Scenarios, as is often the case, the Use Case Model, itself, need not be particularly detailed. Detail can be added during analysis. The degree of detail required is, of course, related to the formality of the requirements document as a legal contract. Use case models are initially highly informal, but through iterative analysis phases they become more formal and more complete. Actors and Use Cases are usually documented using a combination of diagrams and text.

Some development organizations require functional requirements to be captured using a particular format, usually a flat list of system features. Such a functional requirements document is no replacement for a Use Case Model, although it can be used as a good source of information from which to construct one if the document already exists. If not, then the Use Case Model should be written first and the functional requirements document derived from that.

Purpose

A Use Case Model captures the customer's expectations of the functionality of the system. This must be expressed clearly so that both sides can commit themselves to the project requirements, so that misunderstandings can be avoided.

It is vitally important that the requirements document, in general, and the Use Case Model, in particular, are written so that it can be easily understood by customers, domain experts, and end users. If customers cannot understand the functional requirements, they will not feel committed to them. If users or domain experts cannot understand the functional requirements, they will not be able to check that the requirements are correct. The intuitive and simple form of a Use Case Model helps to achieve this common understanding.

A Use Case Model, as opposed to any other format, is used to represent top-level functional requirements, because it emphasizes interfaces and end-to-end functionality. When you identify an actor in a Use Case Model, you are making a statement about where the boundary of the system is, and about what interfaces the system will need. When you identify a Use Case in a Use Case Model, you are describing how the system will be used by one or more of the actors, or how actors will be used by the system.

There is no chance of falling into the common trap of making internal design statements because only observable behaviors can be Use Cases. This is so because they must be linked directly to actors. This helps the Use Case Model to ensure that it concentrates on the system boundary, and not on internal structure, mechanisms, or algorithms. If internal details such as these are allowed to creep into requirements documents, the document will become unwieldy, it will not be understandable by potential users, and it may constrain design unnecessarily.

Participants

Who defines the Use Case Model depends on the context of the project. Whoever defines the requirements, it is the responsibility of the project manager to ensure that they are formalized appropriately, that they are adequate, and that the customer understands them.

If possible, the Use Case Model should be written by a small team which represents both the customer and the development team (analysts) and includes the project manager and the team leader. At least one domain expert should be included, as well as one end user, if possible. As mentioned in the introduction to this chapter, it is important that customer interests are represented. If the development team has no direct customer, a potential end user should take this role.

Timing

The Use Case Model forms part of the contract between the customer and the development team; it is written before any commitment is made on either side.

Technique

A starting point for the Use Case Model is the Problem Statement. This is a succinct statement of the problem that the system is to solve. Turning the Problem Statement into a Use Case Model involves discovering exactly where the boundaries of the system are, who the users of the system are, and what each type of user expects from the system. This is usually done by interviewing or observing users and domain experts. Involving these people in regular reviews of the Use Case Model is very important.

Naturally, not all user expectations can be met, and the authors of the Use Case Model must bear in mind that requirements have costs. The Problem Statement and the Business Case together help to decide which expectations are reasonable.

Strengths

A simple, clear Use Case Model capturing functional requirements encourages commitment and enthusiasm by customers and end users, as well as the project team. This is a prerequisite for success. Vague or ambiguous functional requirements virtually guarantee problems later in the project life cycle.

Weaknesses

The Use Case Model is not typically a standard format for a functional requirements document. If feasible, consider adjusting your development process to bring it into line with planned object-oriented development deliverables by using a Use Case Model as the format for the functional requirements document. If that is not possible and you need to do both a Use Case Model and a business as usual functional requirements document, try to do it in such a way as to avoid confusion and redundancy of information. While a Use Case Model is not a conventional format, it covers conventional needs and should be a more than adequate replacement.

Notation

A Use Case Model of functional requirements is best documented using a combination of diagram and text.

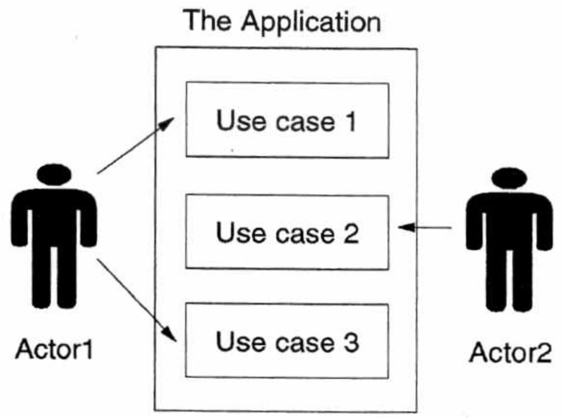


Figure 9-1. The Form of a Use Case Diagram.

Figure 9-1 shows how a Use Case Model may be shown in a diagram. The box represents the system to be constructed. Inside the box are the named Use Cases that are to be supported by the system. The Use Cases are shown with their links to the external, named actors. One could imagine using different graphics to represent different kinds of actors, for example external systems as opposed to humans, although this is by no means necessary. A very small system might be shown on one diagram such as this. A larger system might use several. While little by way of concrete requirements are shown on this kind of diagram, it is very good at indicating the nature of the system interfaces in an intuitive manner.

Links between actors and Use Cases are directional, with the direction indicating which of the two (the system or the actor) initiates communication. An actor may be linked to many Use Cases, and a Use Case may be linked to many actors.

Other forms of notation can be added to a Use Case diagram to express, for example, inheritance between actors, and *uses* or *extends* relationships between Use Cases. See [Jacobson92] for a full description of the possibilities. These extensions should only be used if they aid understandability of the Use Case Model in practice.

In addition to the Use Case diagram, textual descriptions of each Use Case and actor should be provided. For each Use Case, a template of the following form may be completed.

Use case name ____________________
Definition ____________________
Notes ____________________
____
____

For each actor, a template of the following form may be completed.

Actor ____________________
Definition ____________________
Notes ____________________
____
____

The names of the actors and Use Cases connect their textual descriptions to their representations in the Use Case diagram. The *Definition* slots provide space for the actors or Use Cases to be described in whatever detail is considered appropriate. The templates also contain slots to enable notes to be recorded. Design or interfacing ideas, suggestions, and/or constraints will arise during requirements gathering. The *Notes* slots enable these comments to be noted while still separating requirements from design detail. Interfacing notes may consist of or refer to sketches of Screen Layouts, GUI Prototypes, or relevant standards such as CUA or RS-232C.

The pictorial representation shown in Figure 9-1 is often a very convenient way of showing the links between actors and Use Cases. An alternative format for representing the same information, which may be more suitable for large systems, is a two-column table of actors and Use Cases. Yet another way of representing the actor to Use Case links would be to augment the textual actor and Use Case templates with a *Links* slot. It is vital that the links between actors and Use Cases are defined. The method of representation can be chosen to suit the tools available for the size of project.

Traceability

This work product has the following traceability:

Impacted by:

- Problem Statement (p. 93)
- Issues (p. 176)

Impacts:

- Prioritized Requirements (p. 111)
- Acceptance Plan (p. 119)
- Test Plan (p. 164)
- Subject Areas (p. 187)
- Analysis Scenarios (p. 203)
- Screen Flows (p. 237)
- Screen Layouts (p. 242)
- UI Prototype (p. 247)
- User Support Materials (p. 341)

Advice and Guidance

- Involve customers, domain experts, and end users in the formulation and review of the Use Case Model.
- Drive all development activity from the Use Cases—analysis in particular. All development activity should be traceable back to the Use Cases. This ensures that only the required system is built. A practical way of doing this is the scenario-driven approach to development described in Section 18.7.
- Drive the system acceptance tests from the Use Cases, thus closing the project loop.
- If a particular format of functional requirements document has to be written, build the Use Case Model first and derive the functional requirements document from that. This is the preferable way round as the Use Case Model is a good vehicle to use for communicating with customers and users, and hence it is appropriate to agree on the Use Case Model first. If a functional requirements document already exists, then the information contained in it should be used as input to a Use Case Model which should still be reviewed in its own right with customers, domain experts, and users.
- Consider the Use Case Model and the Analysis Scenarios as a complementary pair of work products. The Use Case Model identifies system boundaries, external agents, and top-level system requirements; the Analysis Scenarios elaborate on the requirements and tease out the behavioral variations of the system. Together they constitute the functional requirements of the system. They may also be considered to be the abstract functional specification of the system. The specification is abstract in the sense that it omits interfacing details. These details are added during design; the Design Scenarios may be considered for the concrete functional specification of the system.
- If the system being defined is large, then there may well be a large number of Use Cases. In this situation the Use Cases must be organized in some manner. The organizing principle chosen should be consistent with the way that the requirements will subsequently be analyzed, in order to simplify that analysis.

  For example, if the system is a real-time one, and the analysis is largely driven by state models, then it will be appropriate to organize the Use Cases according to the possible states of the system. If the system is more static, then the analysis is likely to be driven more by the object model, and it will be more appropriate to organize the Use Cases according to the parts of the system that they affect. This process is eased if a domain analysis has already been performed (see Section 18.1) and problem domain classes have already been identified. Use cases can then be grouped according to the classes to which they relate.

- A Use Case Model should not be avoided because of the amount of detail that is apparently required. A minimal Use Case Model would consist of a list of actors, a list of Use Cases, and a representation (pictorial or otherwise) of the links between them. A minimal Use Case Model such as this would be well worth doing and maintaining.
- Do not duplicate information between the Use Case Model and the Analysis Scenarios. Only document the Use Case Model sufficiently to enable the Analysis Scenarios to be written.
- Don't forget to include external systems as actors.
- If the Use Case Model is part of a workbook describing a subsystem, then the other subsystems will appear in the Use Case Model as actors.
- Feel free to record design or interfacing notes in the *Notes* slot of a Use Case or actor template, but be sure that these are not confused with requirements.
- Each Use Case and actor should be documented textually using no more than one page unless it is exceptionally complex. As a rule of thumb, it will take about one day to define each Use Case, but this will depend on familiarity with the domain and the degree to which the system boundaries and the system requirements are "obvious."
- To estimate the total number of Use Cases, spend an hour drawing a Use Case diagram. Count the Use Cases identified and add 25 to 50 percent depending on how familiar you think you are with the domain.
- If the estimated number of Use Cases exceeds 50, either your Use Cases are too small (closer to Scenarios, see Section 11.4) or you should consider splitting the project into subprojects.
- Large numbers of Use Cases need to be grouped in some way for convenience and clarity. This can either be done by subject area or by Actor. Grouping by Actor consists of asking each Actor, in turn, which Use Cases that Actor is involved in, and then documenting these Use Cases as a single group. These groups will naturally overlap as some Use Cases are related to more than one Actor. When this happens, simply document each Use Case once fully, somewhere, and reference this documentation from each duplicate. If the lists are simply of names, with each Use Case being described more fully in a separate, flat list, then the problem of duplication is not important.

- Grouping by subject area consists of first identifying distinct areas of concern. These might follow an already completed partitioning of the system into subsystems, or they may stem from a natural decomposition in terms of business domains, for example, Administration, Accounts, Security, et cetera. Each subject area is then visited in turn, and the Use Cases in the subject area listed. Duplicate entries in multiple lists are unlikely in a grouping by subject area.
- Use the Use Case Model as an important vehicle of communication with customers, users, and domain experts. The intuitive nature of the model will facilitate this. Use the model to check both completeness and correctness with customers. This is most easily done by explaining each group of Use Cases, whether grouped by subject area or by Actor, and trying to find gaps, misplaced system boundaries, additional Actors, errors, and the like.
- The Use Case Model can also be used as the basis for interviewing customers to determine requirements. Such an interview would consist of asking which people are to use the system, which reports are to be generated, and with which external systems the system is to interact. This information results in a draft list of Actors. The Use Cases related to each Actor can then be teased out by asking the different roles of each Actor, and how each Actor is to use (or be used by) the system in each role. It may be that it is the roles that form the Actors, instead of the Actors that were originally identified. Actors should be logically distinct agents rather than physical people or systems. As the Use Cases are listed, or subsequently, they can be marked to indicate their subject area. All the identified Use Cases in a particular subject area can then be extracted and examined for completeness. Almost certainly this cross checking will find omissions. Many variations on this kind of interview are possible.

Verification

- Check that actors and Use Cases are connected appropriately. This can be done by considering all possible *roles* of each actor, and all *tasks* of each actor in each role, and ensuring that every task is enabled by adequate connections to the appropriate Use Cases. If the system uses the actors, instead of the other way round, the analysis should also be reversed: Ask about the roles and tasks of the system, and whether they are adequately covered by connections to actors.
- Check that the Use Case Model includes a representation of the system boundary.
- Check that the descriptions of each Use Case focus on visible system functionality and not on the internal behavior of the system.

Example(s)

Figure 9-2 shows a Use Case diagram for a fax recognition and forwarding application. The application has interfaces to three actors: users, administrators, and fax devices.

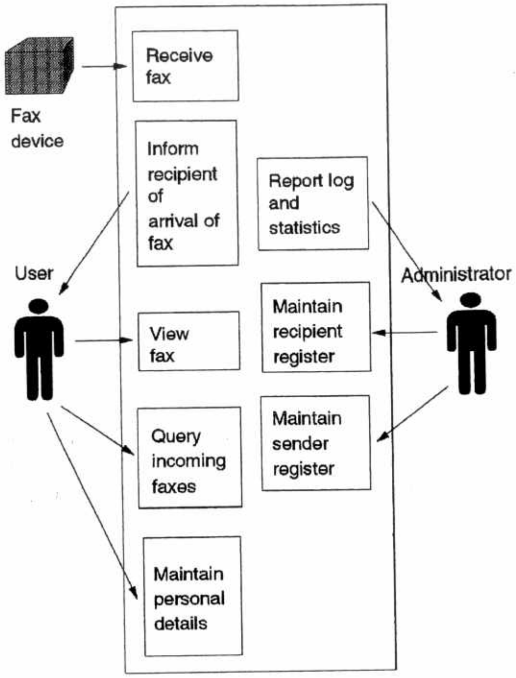

Fax device
Receive fax
Inform recipient of arrival of fax
Report log and statistics
User
Administrator
View fax
Maintain recipient register
Query incoming faxes
Maintain sender register
Maintain personal details

Figure 9-2. An Example of a Use Case Diagram.

The description of the *Receive fax* Use Case might look like the following:

Use case name	Receive fax.
Definition	The action of receiving an incoming fax, determining its intended recipient user if possible, and notifying that person. When a fax is received, the fax sheet images are stored. Optical character recognition is performed to determine the recipient, sender, and subject of the fax. If a recipient is discovered, she is notified of the fax arrival. In any case, the fax is added to the incoming fax list.
Notes	• Use a list of known cover page patterns to aid cover page interpretation. • Maintain a database of known senders and recipients, and the mapping from senders to the cover page patterns that they use. • Think about notification by voice mail in release 2.

The description of the *User* actor might look like the following.

Actor name	User.
Definition	A human user of the fax recognition application. Users must be known to the application; their details are defined by an administrator and maintained by the user herself.
Notes	Users using the fax application are notified of the arrival of a fax message by a dialog window that appears on their workstation. The dialog box has a button so that the user can ask to view the fax.

References

See [Jacobson92] for a full presentation of Use Cases and their role in driving the development process. Jacobson uses Use Cases in a different way to that proposed in this book. Jacobson positions Use Cases as the centerpiece around which all development activities are structured. In this book we advocate placing Scenarios in that role instead of Use Cases. Our reasons are the following.

- Scenarios describe end-to-end behaviors that connect requirements, development work, and test cases in an intuitive and straightforward manner. We use Use Cases to determine system boundaries and to serve as the roots from which Scenarios are subsequently derived. The Use Cases that flow from a Jacobson Use Case Model are not necessarily end to end; many of the Jacobson Use Cases are *components* of the top-level, initial Use Cases.

- Introducing Scenarios allows a useful separation of concerns: a Use Case Model is used to determine the system boundaries and the top-level system behaviors. Scenarios are then used to tease out the assumptions, outcomes, and variations of each of these. The Jacobson form of Use Cases cannot be conveniently used in this way.

- A focus on Scenarios yields a *declarative* description of a system in terms of visible behaviors and what these behaviors mean. This leaves the designer maximum freedom to structure the system as she wishes. A Jacobson Use Case analysis tends to encourage a much more *operational* style of system description that might lead to premature design.

Importance

Functional requirements are essential in some form. We recommend that functional requirements be represented as a Use Case Model and Scenarios.

9.3 NONFUNCTIONAL REQUIREMENTS

Description

Nonfunctional Requirements are the collection of system requirements that are not directly related to what the system should do. Examples of Nonfunctional Requirements include statements of reliability, availability, performance, and details of components (hardware and software) that are to be used or reused. Nonfunctional Requirements usually take the form of constraints on how the system should operate.

Nonfunctional Requirements involving constraints on hardware and software components are often represented pictorially.

Purpose

The Nonfunctional Requirements are a vital component of the requirements document. While the functional requirements drive the analysis process, the Nonfunctional Requirements drive the design. For example, performance constraints will be one factor determining the application Architecture, but they do not affect the analysis of the problem. Another example is the Nonfunctional Requirement that distributed clients and servers should run under OS/2 using TCP/IP sockets for communications. Once again, this affects system design, but not problem analysis.

There is one sense where Nonfunctional Requirements can impact problem analysis: Nonfunctional Requirements may shed light on the external agents with which the system must interact. These agents may well then become *actors* in the Use Case Model that documents the system's functional requirements, see Section 9.2.

Like all other aspects of system requirements, it is essential that Nonfunctional Requirements are expressed clearly and understood by all parties.

Participants

The planners, project manager, and team leader set the Nonfunctional Requirements with customer representatives.

Timing

Nonfunctional Requirements are identified and agreed on as part of the requirements gathering phase.

Technique

One way to generate or to check the coverage of Nonfunctional Requirements, is to pass over the Use Case Model that represents the top level of functional requirements, see Section 9.2. For each Use Case and for each Actor to Use Case link, ask what are the appropriate constraints. For each such interface, the Nonfunctional Requirements related to some predefined set of headings should be listed. The set of headings should include:

- Reliability
- Availability
- Security
- Performance
- Standards
- Look and Feel.

Also, any components, hardware or software, that the system must, or should, use should be listed or depicted graphically.

Strengths

Clear statements of Nonfunctional Requirements are vital to avoid surprises later in the development process. Design churn is one symptom of indecision over Nonfunctional Requirements.

Weaknesses

None.

Notation

Free format text augmented by diagrams, if appropriate, showing constraints on hardware and software configurations. Nonfunctional Requirements may be grouped under headings such as those mentioned above.

Traceability

This work product has the following traceability:

Impacted by:

- Problem Statement (p. 93)
- Issues (p. 176)

Impacts:

- Prioritized Requirements (p. 111)
- User Interface Guidelines (p. 234)
- Screen Flows (p. 237)
- Screen Layouts (p. 242)
- UI Prototype (p. 247)
- System Architecture (p. 257)
- APIs (p. 265)
- Target Environment (p. 272)
- Subsystems (p. 274)
- Design Object Model (p. 281)
- Design Scenarios (p. 293)
- Design OIDs (p. 298)
- Design State Models (p. 306)
- User Support Materials (p. 341)

Advice and Guidance

- Use a set of standard headings, such as those listed above, to organize and to check the completeness of Nonfunctional Requirements.
- Consider prototyping activity to determine realistic constraints.
- Ask "What if?" questions to check the completeness of the Nonfunctional Requirements.
- Some constraints will be on the border between functional and Nonfunctional Requirements, such as user interface constraints. Document them in one place and reference them from the other.
- Include Nonfunctional Requirements that refer to how the system might be expected to change in the future. This will be very valuable when it is decided what flexibility to build into the design.
- Include Nonfunctional Requirements that refer to the hardware and software environment in which the system must run, but include only those environmental constraints that are truly requirements and not just design expectations. Design *decisions* about the software environment, as opposed to *imposed requirements*, should be documented in the Architecture work product, see Section 13.2. Use the Target Environment work product, see Section 13.4., to document hardware environment decisions.

Verification

- Check for coverage by preparing a list of the broad areas that are appropriate to be covered by Nonfunctional Requirements. (This list will vary depending on the type of application and Development Environment.) For each of these areas, check that adequate Nonfunctional Requirements have been produced.
- Check that each Nonfunctional Requirement is in fact nonfunctional, that it is a constraint on system behavior and not a description of new behavior.
- Check that each Nonfunctional Requirement is really required. In particular, check that it is not an attempt to do some premature design.
- Check for contradictions between Nonfunctional Requirements.
- Check for tensions between Nonfunctional Requirements and record these as Issues.

Example(s)

Nonfunctional Requirements for an application with a GUI might include the following.

- All queries must be completed within 0.5 second.
- All updates must be completed within 1 second.
- The system should support up to 20 concurrent users.
- MTTF (Mean Time to Failure) should be at least three months.
- Object technology should be used as a development technology to ease future maintenance and enhancement.
- The IBM Open Class collection class library should be used to represent all collection classes.
- Distributed System Object Model must be used for interprocess communications.
- User interfaces must conform to IBM's Common User Access (CUA '91) standards.
- The hardware architecture shown in Figure 9-3 is to be used as a customer requirement.

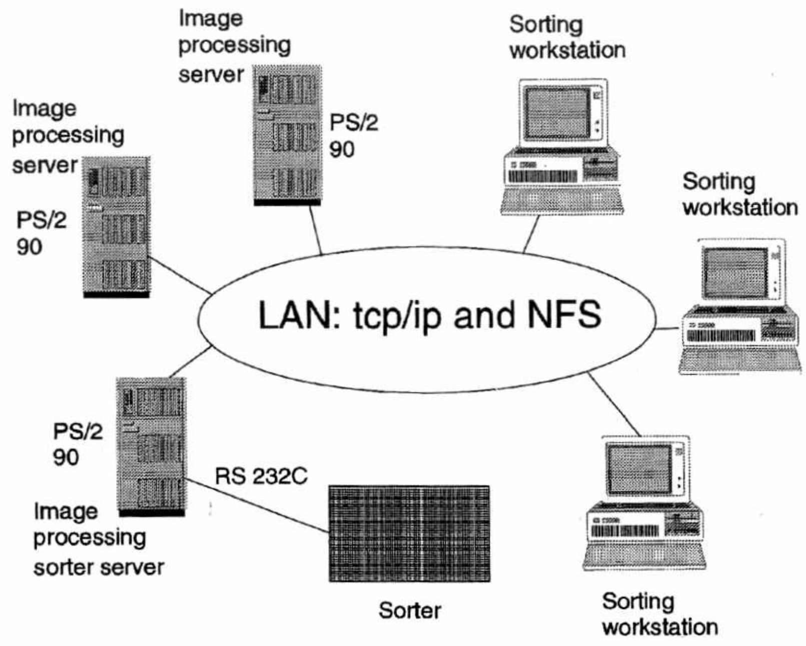


Figure 9-3. Example Environment Requirements Diagram.

References

None.

Importance

Essential. Clearly understanding the Nonfunctional Requirements for a system is critical to ensuring that the system built satisfies the full breadth and depth of customer requirements.

9.4 PRIORITIZED REQUIREMENTS

Description

The Prioritized Requirements work product defines the relative priorities of system requirements: both the functional requirements that are represented by the Use Case Model and the Nonfunctional Requirements. The prioritization can be formal or informal as appropriate and should reflect the views of the customer and users.

Note that a Prioritized Requirements primarily represents input from the customers and users. By itself it cannot dictate the priorities of the development team as these are also affected by dependencies between requirements, development risks, et cetera. This information must be factored in before scheduling decisions are made.

Purpose

Whether the customer and potential users have been involved in the writing of the Use Case Model and the Nonfunctional Requirements, these two work products alone do not contain sufficient information to permit planners to begin to construct a Release Plan. Two further kinds of information are required: the *importance* of each requirement, and the *urgency* of each requirement.

It is very easy, when interviewing customers or users, for requirements to take the form of "wish lists" of functionality that would be nice to have but which are not necessarily essential. It is clearly important for a development team to be know which requirements are vital and the ones that are wishes. Similarly, and particularly if delivery of the system is to be phased over a long period, the relative urgency of requirements is very relevant. Customers cannot necessarily have all their requirements satisfied. Both when constructing the initial Release Plan and subsequently during development, trade-offs will have to be made. Requirement X will take longer than we thought: Should we slip it or hit that deadline at the expense of requirement Y? In such situations it is obviously necessary for the priorities of the customer and the development team to be in tune. Disappointment and surprise will be the results otherwise. Customers might be willing to accept a slippage in the overall schedule if certain key features are provided early. You need to find out what those key features are.

Participants

The customer, project planner, project manager and team leader are the people most directly involved in writing this work product.

Timing

Prioritized Requirements is written as part of the process of requirements gathering. If it is necessary for requirements to be prioritized in great detail, then it may continue into the analysis phase.

Technique

The basic technique for prioritizing requirements is very simple: The requirements are listed, and customers and users are asked to evaluate the relative importance and urgency of each. The role of the planner, manager, and team leader is to ensure that the customers and users understand the costs and other implications of each requirement.

The degree of detail appropriate for a Prioritized Requirements will vary from project to project. For example, a small project with a "friendly" customer can afford to be fairly informal in its prioritization. On the other hand, a large, complex project must be prioritized in greater detail. You will have to decide the appropriate level of detail for your project.

Detail is introduced in two ways: when listing requirements and when prioritizing them. When listing requirements, one can be content with working at the Use Case level only. Alternatively, one might wait till Analysis Scenarios are available and prioritize at that level (similarly for priority levels). When deciding the relative urgency and importance of each requirement, at whatever level of detail the requirement is described, a coarse or a fine grain spectrum of options might be used. No one level of granularity will fit all projects.

Strengths

Making the priorities of the customers and users explicit in this way reduces the risk of surprises and shocks later. It also helps the development team to make the inevitable trade-offs that arise in design in a consistent and hence effective way.

Prioritization is the language of negotiation. It is both natural and vital to prioritize when faced with project risks.

Weaknesses

In a very small development effort with few requirements and where no incremental releases are planned, prioritization of requirements may not be needed.

Asking customers to prioritize requirements might seem inappropriate for someone who is trying to present a "can do" image. Why ask about priorities if I want to create the impression that I can do anything? If you can in fact do anything then you can forget about prioritization; the rest of us need a way of knowing how to deal with resource constraints, with unseen design difficulties, and with all the other kinds of project risks, in ways that will be acceptable to the customer.

Notation

The precise format of a Prioritized Requirements will depend on the size and nature of the project, as suggested above. In general, however, the project requirements are listed, and each requirement is assessed for both urgency and importance. For most projects it is adequate to list functional requirements at the Use Case level; it is probably not necessary to descend into the detail of Analysis Scenarios. It will probably be necessary to prioritize each Nonfunctional Requirement separately.

For each requirement, its urgency and importance can be specified in greater or lesser detail. For many projects a three-value scale for each is probably appropriate.

Importance	1 (vital), 2 (important), 3 (would be nice)
Urgency	1 (immediate need), 2 (pressing), 3 (can wait)

Requirements can either be listed by name, by numerical or other identifier, or the prioritization can be added to the Use Case Model and Nonfunctional Requirements work products directly. If the prioritization is separate from these other work products, then a simple three row or column table is adequate.

Traceability

This work product has the following traceability:

Impacted by:

- Problem Statement (p. 93)
- Use Case Model (p. 96)
- Nonfunctional Requirements (p. 106)
- Issues (p. 176)

Impacts:

- Acceptance Plan (p. 119)
- Resource Plan (p. 135)
- Schedule (p. 139)
- Release Plan (p. 144)
- Risk Management Plan (p. 152)
- System Architecture (p. 257)

Advice and Guidance

- Involve customers, users, development team members, and management in prioritizing requirements.
- Once you have drawn up the Prioritized Requirements table, go back to the Acceptance Plan, which should identify the acceptance tests that are most important to the customer. If you meet 95 percent of the customer's requirements and the customer tests first the 5 percent you didn't meet, you will have failed in the customer's eyes. On the other hand, if you meet the most important 5 percent of customer requirements (those that the customer will test first), you will probably have earned the customer's willingness to tolerate some lower priority fixes if needed.
- Start with a coarse-grained set of urgency and importance values as suggested above. Only if these result in too little information should you then use a more fine-grained spectrum of possibilities.

- Start with prioritizing Use Cases and only if this is problematic, or if you estimate that particular, individual Use Cases involve great effort, should you switch to prioritizing functional requirements at the Analysis Scenario level.
- The Prioritized Requirements framework can be extended to deal with more than just input from customers and users. Additional columns or rows could list estimates of risk, complexity, et cetera. This information, in addition to dependencies between requirements, can then be used as direct input to the scheduling process. An additional row or column, a "scheduling prioritization index" might be added as a rough numerical aggregate of the rest of the information accumulated for each requirement.
- Adding the prioritization information directly to the Use Case Model and Nonfunctional Requirements work products makes sense, if they do not fragment the information unacceptably. If the prioritization information is not entered as a separate table then it must be possible to extract this table from the other work products. This depends on tool support.
- Be sensitive to the statements of the customer and users. If they seem to imply that part but not all of the functionality of a Use Case is vital, then explore this further with Analysis Scenarios, if only informally by asking about different situations.
- It may well be that it is appropriate to prioritize at the level of Scenarios instead of Use Cases, but be selective; do not do this automatically and uniformly.

Verification

- Check that writing the Prioritized Requirements has yielded real information. Having all requirements listed with importance "2" is not useful. In such cases either use a finer-grain set of values and/or force the issue by asking questions such as "What if I could only supply this or that. Which would you choose?"
- Check that prioritization has been adequately detailed for listing requirements. If you anticipate that particular Use Cases will be refined into many Analysis Scenarios then try prioritizing a few of the Scenarios to see whether their priority values differ from those of their Use Case.
- Present the prioritization table to the developers and ask for their feedback. They may have insights into risks and complexity that warrant additional questions being asked about priorities.
- Small projects might well make do with an informal statement of priorities. A few lines of text might be sufficient to say what is most important and pressing, and what can be slipped or dropped if necessary.

Example(s)

The following is a simple example of a Requirements Prioritization. The scale of urgency and importance values is as suggested in the Notation section earlier in this work product description.

Table 9-1. *An Example of a Requirements Prioritization.*

Requirement	Importance	Urgency
Export to spreadsheet	2	3
Determine net worth	1	1
Get detailed changes	1	2
Purchase securities	1	2
Transfer funds	1	2
Make loan application	1	3

References

There are no references for this work product.

Importance

Optional, but very important for projects that may have more requirements than available resources can reasonably handle.

9.5 BUSINESS CASE

Description

A Business Case provides a justification for the undertaking of a project. The Business Case is complementary to the Problem Statement: The Problem Statement states the problem to be solved whereas the Business Case justifies the effort. There are two kinds of Business Case that one encounters: quantitative, which is based on hard measurements (e.g., cost-benefit analysis); or qualitative, which is based on something less tangible and not necessarily measurable (here, the move to objects may be based on a sponsor's vision and may not be subject to a cost-benefit analysis).

Besides providing the justification for a project, a Business Case is important as a focus for requirements. For example, if a Business Case rests on the project feeding certain reusable parts into a parts center then this commitment must be taken into account during all development phases.

A Business Case has to justify a project from a business perspective. It often takes the form of a cost/benefit analysis. The cost of the project is an estimation of the resources that it will consume. The benefits of the project have many possible sources including the following:

- Revenues
- Productivity
- Customer goodwill
- Accomplishment of organizational mission statement
- Development experience gained, for example the project will result in three developers sufficiently trained in C++ design and coding that they can work independently in future projects
- Reusable components, for example, a telephony domain analysis and a framework for simple telephony applications that can be reused in a future application.

Purpose

Too many teams begin projects without knowing why. Or perhaps each senior team member knows why but the reasons differ from person to person. This is obviously undesirable. The Business Case assists with achieving a shared vision and understanding of the project's value.

Participants

It is usually the project manager who is responsible for articulating a Business Case, though planners and financial analysts typically play a major role.

Timing

Either before or during requirements gathering.

Technique

Standard business techniques for justifying a project should be used.

Strengths

- A Business Case ensures that expectations are explicit and constant.
- The Business Case helps to focus attention on why the project has been undertaken. Surprisingly many projects are started without any clear idea about why they have been undertaken.
- Agreeing on a written Business Case is a good test of common commitment and direction. This is obviously vital during the planning phase of a project, but it is also very helpful during development. For example, when selecting an Architecture, various alternatives with different trade-offs may be considered. Understanding the justification for a project can at times assist with making decisions during the project.

- Depending on the circumstances, a Business Case may be essential to obtaining project funding.

Weaknesses

Business Cases can be unreliable in their forecasts and it is often difficult to accurately quantify the benefits of a project. Having to develop, sell, and defend a Business Case is also hard work at a time when it might be more interesting to refine requirements and begin to build project plans.

It is, however, essential and projects are much more likely to fail, or be perceived to have failed, without one.

Notation

Free format text.

Traceability

This work product has the following traceability:

Impacted by:	**Impacts:**
• Problem Statement (p. 93)	• Resource Plan (p. 135)
• Issues (p. 176)	• Schedule (p. 139)
	• Release Plan (p. 144)
	• Reuse Plan (p. 158)

Advice and Guidance

- Include a description of the analysis, design, and code assets that the project is expected to accrue. This might include a domain analysis, architectural or utility frameworks, or reusable code.
- Include reference to the skills that will be acquired as a result of carrying out the project.
- Use the Business Case as input into design decisions to ensure that design trade-offs are consistent with each other and with the overall project objectives.
- Understanding the justification for the effort can be a major benefit in assisting the team to make decisions relating to functionality.

Verification

- Check that the Business Case focuses on why the project should be undertaken by the development team, rather than on why the customer needs the system.

Example(s)

Below we present examples of qualitative and quantitative Business Cases. We have chosen to show examples that are different in scope from others presented in this book in that they are not specifically related to a project employing object technology. Rather, they are examples from two companies of the Business Cases for the deployment of object technology in their software development communities.

We have chosen this approach since Business Cases for software development is really a business as usual work product but for many projects, the question of how to justify the deployment of object technology is a harder question. These are examples of how two companies that we have had some involvement with approached this problem.

The first example is based on an actual Business Case but has been modified for reasons of confidentiality. In making these changes, the arithmetic may not stand up, but it is intended as an illustration only.

Initial plan outline for a move of 100 people to object technology:

- Our company has an Information Technology group consisting of 200 programmers.
- It is estimated that going to objects will improve productivity by 50 percent over time.
- Within the first three years we plan to convert at least 50 percent of our development staff, roughly 100 persons, to using objects.
- It is estimated that the cost of training will average three months per person. The investment per person is a rather complex calculation, bearing in mind that after some initial training, programmers will be productive but at a lower productivity rate than usual.
- Assume productivity improvements achieve their target levels in year two of the transition.
- There will be an additional cost of six full-time mentors at a cost of $900,000 per year (planned for 2 years).
- Administrative and infrastructure costs of the transition are a net cost of $1 million (rather complex accounting comes into play here, and there are tax benefits that will not be covered here).
- Software acquisition costs are not factored in as they are covered by usual budget.
- Costs:
  - 25 person years of effort (100 persons for 3 months) for a total cost of $2.5 million (assuming a $100,000 internal cost for a person year)
  - Mentoring Costs = $1.8 million (for 2 years)
  - Infrastructure Overhead = $1 million (for 2 years)
  - Total Costs = $5.3 million (for 2 years)
- Benefits
  - Starting in year 2, the 100 programmers will be 50 percent more productive.
    - Yielding a benefit of $5 million per year (100 programmers will produce the work of 150 programmers).
    - Because of the current application development backlog, the productivity gain will be applied to reducing the backlog (i.e., there is no plan to reduce the workforce).
  - The investment will be recovered early in year 3 of the transition.
  - Some internal object technology mentors will emerge from the group of 100 programmers.

Figure 9-4. Example of a Quantitative Business Case.

The following example outlines an actual Business Case for a company piloting the use of object technology. While some of the benefits could be measured (for example, the time to introduce a new product), the company did not produce a quantitative Business Case.

The following represents a qualitative Business Case—the benefits were not subjected to hard measurement.

The company plans to adopt object technology hoping to achieve the following objectives:

- Deliver software that better meets our users' needs.
- Improve our ability to deliver new products to the marketplace in a timely fashion.
- Improve programmer productivity.
- Develop an infrastructure to support software reuse and lessen software development time.
- Involve users more in the software development process via Object-Oriented Analysis techniques.

Figure 9-5. Example of a Qualitative Business Case.

References

- Chapter 4 of Goldberg and Rubin [Goldberg95] has some advice on how to make the business case and obtain management commitment to use object technology.
- Chapter 1 of Malan [Malan96] offers advice and a case study on getting management buy-in.

Importance

Essential. All projects need some level of justification before any significant amount of resources are expended on them.

9.6 ACCEPTANCE PLAN

Description

The Acceptance Plan is a description of the sequence of steps by which the customer, or a potential end user playing the role of the customer, will verify that the constructed system does indeed satisfy its requirements. Each step of the plan should be clearly and fully described in terms of how to set up the test, and the acceptable system behaviors in response to the test. These behaviors should be stated in such a way that it will be obvious whether or not the test is passed.

The contract with the customer should include a clause to the effect that the system will be accepted formally if the acceptance tests are all successful. If there is no direct customer, the contract and the Acceptance Plan will be informal, but their existence is still relevant.

Purpose

Requirements documents are often long. Verifying that a given system satisfies every statement of the requirements document can take a prohibitive length of time, and can be open to interpretation, particularly if the customer is looking for excuses not to accept. Furthermore, the burden of proof that the requirements are satisfied lies with the development team. Proving correctness is notoriously difficult, at best. For example, the requirements document may state that a particular interface should use a particular protocol. Proving that the code correctly generates the protocol is almost certainly not feasible. Running tests is the only alternative, but which tests and how many? The scope for disagreement is considerable. Bearing in mind that disagreement might mean lack of payment, it is up to the development team to ensure that this situation does not arise.

The Acceptance Plan is an integral part of the requirements document, and as such is agreed to by both parties when the project is started. If requirements change during the lifetime of the project then both new statements of requirements and a new Acceptance Plan will have to be negotiated.

Participants

The Acceptance Plan will typically be developed and agreed to by a team made up of representatives with decision-making responsibilities from the project team (perhaps the project manager and/or the team leader) and the customer (or someone representing the customer set).

Timing

The Acceptance Plan is written after the rest of the requirements document but before commitment to the project is made by either side. If there is no direct customer, and hence no formal contract or commitment, this timing can be relaxed and the Acceptance Plan written in parallel with development work.

Technique

The Acceptance Plan should be designed to demonstrate in a positive manner that the project requirements have been met. It should be driven by the functional requirements of the system, that is, the set of Use Cases, see Section 9.2. The Use Cases each summarize one way in which the system is to be used. For each of them, the Nonfunctional Requirements must be applied and an adequate set of tests decided upon. The tests should all be closed. That is, it must be guaranteed that each can be determined to be either successful or unsuccessful within a definite period of time. Sometimes this means that time-limits or other bounds must be applied to tests.

Strengths

The existence of an adequate Acceptance Plan ensures that the customer is committed to a definite way of determining whether or not to accept the system.

Weaknesses

The writing of an Acceptance Plan is time-consuming. In particular, it consumes scarce resources at the start of the project that are also required for project planning. The temptation exists to claim that an Acceptance Plan is not urgent and can be constructed as the project proceeds. Once the contract has been signed, however, the negotiating power of the development team is considerably lowered, and there is no compelling reason for the customer to commit to the plan at all. If there is no direct customer, of course, there is less justification for writing an Acceptance Plan early.

Notation

The format of the plan is a grouped sequence of individual steps of the following form.

Test group name ______________________
Test number ______________________
Prerequisites 1. ______________________
2. ______________________
Set-up instructions 1. ______________________
2. ______________________
Test instructions 1. ______________________
2. ______________________
Acceptable behavior 1. ______________________
2. ______________________
Test performed on date ______________________
Test performed at time ______________________
Name of customer signatory ______________________
Name of project signatory ______________________
Test result (pass/fail) ______________________
Signed for customer ______________________
Signed for project ______________________

The Acceptance Plan should include instructions for the repetition of tests in the event of failure. For example, it might be stipulated that if a test fails, then the development team is permitted a given length of time to make corrections before the group of tests of which that test is a part is repeated; the rest of the Acceptance Plan does not need to be repeated.

Traceability

This work product has the following traceability:

Impacted by:
- Use Case Model (p. 96)
- Prioritized Requirements (p. 111)
- Issues (p. 176)

Impacts:
- Test Plan (p. 164)

Advice and Guidance

- Review each test, asking under what circumstances it could fail. If any of these circumstances depend on the environment of the system, for example, they depend on system loading, then add set-up or test instructions to ensure that a limit is not exceeded.
- Set-up and test instructions should be constructive. That is, they should specify a definite sequence of steps for achieving the test. They should not, for example, say that system loading must not exceed some specified value, because this does not state how the permissible loading is to be achieved. Instead, they should specify the user actions to be performed by a definite number of users. This can be achieved and controlled precisely.
- Place time-limits on tests. No test should be allowed to drag out with no predetermined time-limit.
- Link tests together by means of prerequisites to save complex test set-up instructions.
- If there is no direct customer, the need for strict criteria for determining release is still valid.

Verification

- Check that each use case of the Use Case Model is covered by adequate acceptance tests.
- Check that each Nonfunctional Requirement is covered by adequate acceptance tests.

Example(s)

The following is an example of an Acceptance Plan from a group which is testing that an application can be installed successfully as per specifications:

Table 9-2. Example of an Acceptance Plan.

Test group name	Installation
Test number	1
Prerequisites	1. Hardware as specified in user guide 2. Software as specified in user guide
Set-up instructions	1. OS/2 running
Test instructions	1. Execute "A:\INSTALL" from OS/2 command line. 2. Follow installation instructions in dialog boxes, using all provided defaults. 3. Reboot. 4. Execute "PPP" from OS/2 command line.
Acceptable behavior	Successful completion of installation procedure within 10 minutes. The PPP command results in the application sign-on dialog appearing.
Test performed on date	
Test performed at time	
Name of customer signatory	
Name of project signatory	
Test result (pass/fail)	
Signed for customer	
Signed for project	

References

None.

Importance

Optional. An Acceptance Plan is certainly a good idea and a good way to ensure that the project team and the customer have the same understanding as to what will be delivered and in what state. But, it is possible to have a successful project without such a plan.

10.0 Project Management Work Products

The Project Management chapter of the workbook describes work products that are important to the successful control and management of the project.

Many software development projects are large and complex. Throw in the possibility of unclear and unstable requirements, dependencies on other organizations, and the use of new technologies and processes (such as with object technology) and the difficulty of successfully completing a project grows enormously. The management team and the project's technical leaders must deal with staffing the project, scheduling it, selecting development processes, managing risk, setting quality goals, and identifying and resolving risks, issues, and dependencies.

The projects that succeed in that kind of environment are those on which the team has a strong commitment to applying good Project Management principles to the project throughout its life.

The project management section of the project workbook consists of the following work products:

- Intended Development Process
- Project Workbook Outline
- Resource Plan
- Schedule
- Release Plan
- Quality Assurance Plan
- Risk Management Plan
- Reuse Plan
- Test Plan
- Metrics
- Project Dependencies
- Issues

These work products all "inherit" the common work product attributes described in Section 8.1, and have specialized attributes of their own.

Many of these work products are just as necessary in projects not using object technology as in those that are following an object-oriented software development approach. In most cases, the changes to these work products caused by a move to object technology are subtle. Yet, they should still be treated as essential work products in the project workbook for any object-oriented project.

The Intended Development Process provides a broad outline of the development process that the development team will be following. The general principles for development set forth in it will be used in other work products that set specific schedules and plans.

The Project Workbook Outline may in some sense be particular to those projects using a workbook-centered approach to development, but if seen as documentation of the act of specifying what work products will be produced during each phase of a particular project, it is important to all projects. An understanding of what work products will be built certainly impacts most of the other Project Management work products.

The Resource Plan identifies the resources needed, and the timing of when they are needed, to produce a solution that addresses the requirements laid out in the Requirements work products on the necessary Schedule. The Resource Plan must be coordinated with Schedule, Release Plan, Project Dependencies, Issues, and various others work products.

The Schedule is a means for understanding the work that needs to be completed and when in order for the project to complete successfully. Like many of the Project Management work products it needs to be closely coordinated with the other work products described in this chapter.

The Release Plan is used to place Requirements into manageable units of work reflected by project releases, versions, iterations, and increments and is closely aligned with the Schedule and Resource Plan.

The Quality Assurance plan sets project goals for quality, which usually have a lot of customer input, and outlines the activities that will take place throughout the life of the project to meet those goals.

Most projects involve some level of risk. The Risk Management Plan is used to identify potential risks and details plans to address them. Risks can impact Prioritized Requirements which in turn can affect the Schedule and the Release Plan.

The Reuse Plan work product recognizes the fact that reusing existing components, and producing new components that can be reused by others, are often integral parts of an object-oriented approach to software development. It also recognizes that reuse will not happen unless it is planned in advance and the necessary time and resources are allotted for it.

A Test Plan is a critical part of any software development project as testing is typically needed to ensure that the project's Requirements are being met, and testing is usually a large piece of any Quality Assurance Plan.

Metrics can be a very useful tool in projecting necessary resources, setting schedules, and identifying issues and risks.

The Project Dependencies work product is an important tool for understanding those items that your project requires, perhaps from other organizations, to successfully complete a solution that addresses the project's requirements on the required schedule.

Finally, the Issues work product is an important tool for capturing, understanding, and addressing the inevitable issues that will occur throughout the life of the project.

The following sections provide more detail on each of these work products.

10.1 INTENDED DEVELOPMENT PROCESS

Description

The Intended Development Process work product documents the broad outlines of the development process that is to be used for this project. By itself it is not a plan; other parts of the project workbook apply the information of this work product to the particular project to form plans, schedules, workbook guidelines, et cetera. For example, this work product may decree that the project will be incremental, with increments lasting approximately six weeks. The project manager will then use this information, and much more, to generate a work schedule. The process documented in this work product is project-independent, although it has obviously been selected for its appropriateness to deal with the problems and constraints of the current project. The Intended Development Process should be thought of as a development process template.

This work product is broad but shallow. That is, it addresses the entire project life cycle, but only at a very high level. The work product should address the development process itself, the techniques and notations to support the process, and the tools to automate some of the development activities.

There are two aspects to the Intended Development Process that may be described. First, the overall project profile, and secondly, the individual project life cycle phases. The overall project profile describes the large-scale shape of the project. That is, whether it is to be iterative, incremental, waterfall, or some combination of these. Each of these profiles has parameters that must be set for a project, for example, the length of each increment, and the phases that constitute each increment and their durations. All of these parameters have their approximate "ideal" values that consider the requirements to manage complexity, communications, and change. These issues are common to all projects, and their solutions are to a certain extent independent of the details of any particular project. External and organizational constraints and project peculiarities are, however, significant influences. A presentation in this work product of both the ideal and the actual Intended Development Process will help the issues to be thought through and will make the work product more reusable in future projects.

Besides the overall project profile, the individual phases of the project life cycle may be described in outline in this work product. The phases and their definitions will vary from organization to organization, but they must encompass the chapters of the workbook described in this book.

For each of the life cycle phases, the techniques to be used, the notations to support the techniques, and the tools to automate the techniques should be described, as far as is reasonable at this early stage. The goal is to define a development process, leaving the localized details of the steps to be dealt with later as appropriate. For each development phase, the following issues may also be addressed.

- The process by which the work products are to be validated for correctness and verified for internal consistency.
- Version and change control.
- The way that the size of the activities may be estimated, and both progress and quality monitored.
- The technology and techniques to be used to construct and use various kinds of prototypes.

For the project planning phase, the planning work products may be described, along with planning and scheduling tools that may be useful in producing and maintaining these.

For the requirements phase, the format of the requirements document may be presented, along with tools that are capable of assisting the gathering and sorting of requirements. The way that customers and end users can be involved in the process may be discussed.

For the analysis and design phases, the modeling notations may be specified, for example Booch or OMT. Modeling notations tend to be neutral to the development process used, but they influence tool and education choices, and may be controlled by organizational rules. If possible, tools should be selected for their appropriateness to the Intended Development Process and techniques, and not for their support of a particular notation. The global impact of development tools, and hence of development techniques and notations, makes them a proper subject of discussion in this work product although they do not strictly fall under the heading of "process."

For the implementation phase, the target programming language may be specified, if this is appropriate, along with code generation, programming and programming support tools. The principles of code integration should be presented.

For the code testing phases, the testing principles may be established, along with any testing or test generation tools that are relevant.

The Intended Development Process work product is used as the basis from which to write the guidelines work products of each chapter of the workbook. The difference between these is that the Intended Development Process work product is aimed at project managers and leaders, whereas the guidelines work products of the workbook are aimed at the team members responsible for completing the work products covered by the guidelines.

Purpose

The Intended Development Process work product enunciates the principles behind the process to be followed, and some of their details. This is done to separate these principles from their application within the context of a particular project. This in turn is for three reasons.

- Documenting principles separately from their application helps the project manager to separate them mentally.
- The principles represent a body of project management techniques that are independent of a particular project, and hence they can be reused from project to project, provided of course that they are applicable. It is difficult to reuse something unless it is documented separately.
- This work product can be reviewed at the end of the project to judge what worked, what didn't work, and why. Reviewing in this way a project-independent statement of development process principles makes the lessons learned more reusable than simply reviewing the various project-specific plans.

Participants

The project manager must approve the Intended Development Process. It will probably be written jointly by the team leader and the project manager.

Timing

The Intended Development Process is input to the other project planning steps. As such it is a very early deliverable, at least in draft form. It is probably one of the first project management work products to be completed.

Technique

Reuse the documented development process of a previous, successful project. If no such process is available to be reused, use the process given in the example below as a starting point.

Strengths

A well-documented Intended Development Process can prevent later confusion and unnecessary discussions about how the project is to be tackled. It represents the reusable portion of the project management chapter of the project workbook. Thinking through and documenting the complete development process at a very early stage permits tool and education requirements to be anticipated.

Weaknesses

It is difficult to document an Intended Development Process if this is a first-time project for the team. Giving guidance and still giving the team necessary flexibility is a difficult balance to strike. A well-documented Intended Development Process takes considerable time to produce.

Notation

Free format text, but you might use one heading for the overall project profile, and one for each of the phases of the project life cycle. Under each life cycle step there should be subheadings for techniques, notation, tools, metrics, et cetera.

An alternative, more formal approach would be to represent the Intended Development Process as a complete, idealized process. If that approach is followed, this work product would contain, in summary form, a complete project management document. Such a document would be generic, because it would be independent of the details of any particular project. These details are instead supplied in parametric form: the total effort of the project, the project "profile," staffing, et cetera. The actual project management details could then be supplied by "filling in the blanks," which would obviously have great advantages. This is an approach similar to that used by automated project estimation and scheduling packages.

An advantage of this more formal approach is that it makes explicit the parametric dependencies of the plan, and hence provides more guidance (for example) on how to estimate effort and track progress. Furthermore, standardizing in this way on process not only makes projects more repeatable, but means that metric data is more widely applicable.

A compromise can be reached. For example, an informal approach can be used for an initial project, after which the project management workbook chapter of that project can be summarized and abstracted to form the Intended Development Process for later projects.

Traceability

This work product has the following traceability:

Impacted by:
- Quality Assurance Plan (p. 147)
- Test Plan (p. 164)
- Issues (p. 176)

Impacts:
- Schedule (p. 139)
- Release Plan (p. 144)
- Analysis Guidelines (p. 183)
- User Interface Guidelines (p. 234)
- Design Guidelines (p. 253)
- Coding Guidelines (p. 322)

Advice and Guidance

- Where appropriate and when time permits, note the reasons for selecting a particular process. This will help when this work product is reused by other projects, or assessed for its reuse potential.
- It should be clear that the development team can deviate from the intended process if there are good and well-documented reasons for so doing.
- The Intended Development Process work product should not duplicate material presented in the Analysis, User Interface, Design, and Coding Guidelines work products. Use references to those work products to provide information on how the Intended Development Process is to be applied.

Verification

- Check that all reusable process management material is included in or referred to from this work product.
- Check for balance of release process documentation and development process documentation.
- Check that tool support, if covered in the Intended Development Process, ensures that manually-entered information needs to be entered only once.

Example(s)

An Intended Development Process might include details such as the following:

- Use the workbook and basic techniques described in the book *Developing Object-Oriented Software: An Experience-Based Approach* (that's this book!).
- Use a combination of a waterfall and an iterative and incremental process. Requirements gathering, initial project planning, analysis, and architectural definition phases are performed in a waterfall manner, followed by a series of incremental development cycles. The work products of these initial phases, like all others, are subject to iterative improvement during the life cycle of the project.
- Use conventional estimation techniques to provide initial project sizing estimates. From the end of the analysis phase, estimate total classes (including utility and graphical user interface) in the ratio of 1:6 analysis classes to final implementation classes. Use an initial estimate of one person-month (PM) per implementation class for all project phases through to delivery. (This is an estimate for experts. For intermediates, use 2 PM's per implementation class; for novices, use 3 PM's per implementation class.) Use these figures to update the original estimate of project size.
- The incremental part of the development process follows a seven-month release schedule. Each release consists of three eight-week increments, followed by a four-week system test cycle, culminating in the release.

- Each increment has three distinct phases of two weeks each:
  - design and interface development
  - implementation and documentation
  - integration, final verification test (FVT), metric data analysis, process adjustment, and plan adjustment.

  Further, the activities associated with the first two phases include a week of developing the work products followed by a week of review and iterative rework. As an estimate, a person can design, implement, and document the solutions to five scenarios throughout the two week cycle (including the review and iteration).

- The very first development increment follows a depth-first approach (Section 17.1), consisting of a series of four minicycles of two weeks' duration each. The goals of these minicycles are to establish the Development Environment, to give all team members a taste of all development phases, and to establish the basic Architecture.

References

Succeeding with Object Technology by Goldberg and Rubin [Goldberg95], in particular the chapters on "Strategies for Developing with Objects," and "What is a Process Model."

Importance

Optional but important. All projects should have an understanding of what their approach is going to be before starting. But, it is possible that this understanding can be gained through some of the other work products such as Schedule and Resource Plan.

10.2 PROJECT WORKBOOK OUTLINE

Description

The Project Workbook Outline is an organized list of work products that will comprise the project workbook. As the name suggests, it takes the form of an outline. The key items in the outline are work product types (e.g., Problem Statement). It is recommended that those items be grouped by the phase or facet of work to which they belong (e.g. Project Management, Requirements).

The outline shows not only the intended content of the workbook but also the order that the items will appear in the workbook. It represents a commitment by the development team to produce that set of work products and to record them in the workbook.

Remember that a project workbook is a logical container of concrete work products. The workbook needs to have a place for every agreed to work product, though the actual work products may physically be stored in a database managed by a CASE tool or a configuration and version management tool. In fact a physical work product might even be stored in a file drawer.

Not all work products are "deliverables," as in "customer deliverables," but all that the team agrees to produce do need to be kept and be made accessible to the development team via the project workbook.

Purpose

The Project Workbook Outline is needed to find agreement among the development team members about the set of work products to be created for the project. It is also needed to establish a commitment from the development team to record each of the identified work products in the project workbook.

There is no reason not to create a Project Workbook Outline. The mere existence of a project workbook implies a de facto structure or outline. Establishing the outline first forms a plan for consistency and completeness.

If an outline is not created first, development team members will not know which work products to create and record for the project. A good development team will question the lack of a Project Workbook Outline. A team less experienced will probably set off creating the work products that they used in their last project. If there is more than one team, there will be several different outlines.

Participants

The Project Workbook Outline should be developed by the team leader and the project manager.

Timing

The Project Workbook Outline should be decided and recorded during the project management phase, but it will be retroactive to the requirements phase.

Technique

Start your outline with the essential work products listed in Table 8-1. Add other work products to the outline depending on the size and risks of your project. Review the work product list with the project manager and team leaders to ensure that each work product in the outline is suitable for the project.

Strengths

If you have a published Project Workbook Outline, everyone in the project will know which work products they need to create and which they can expect to find and work with from other team members.

Weaknesses

None.

Notation

Since the outline will only be used to establish the workbook, any form (numbered or not) of two level outline will suffice.

The recommended structure for a project workbook is one major section per phase or facet of work and one subsection for each work product type associated with that phase. Intuitively, the order of phases and work product types within the phase should approximate the chronological order in which they are normally produced.

The Project Workbook Outline should match the intended workbook structure.

Traceability

This work product has the following traceability:

Impacted by:
- Quality Assurance Plan (p. 147)
- Issues (p. 176)
- Analysis Guidelines (p. 183)
- User Interface Guidelines (p. 234)
- Design Guidelines (p. 253)
- Coding Guidelines (p. 322)

Impacts:
- Quality Assurance Plan (p. 147)
- Analysis Guidelines (p. 183)
- User Interface Guidelines (p. 234)
- Design Guidelines (p. 253)
- Coding Guidelines (p. 322)

Advice and Guidance

Especially for new projects with teams who have not worked together before, create and distribute the Project Workbook Outline early. It helps the team to see the scope of work that they will be performing and the work products they will be expected to create and record.

During team orientation meetings make sure that the entire team understands the nature of each work product and the relationships that exist among them (i.e. the common facets).

Ensure that the team agrees with the necessity of each work product in the outline.

Verification

Use either the complete development team (for a small project) or the team leaders (for a large project) to review the Project Workbook Outline. Ensure that the review group understands what each work product is and how it is related to others. Seek consensus from the review group that each work product is necessary and that the list is sufficient for the success of the project.

Example(s)

For a small project, the following outline would be a reasonable starting point, with other things like User Interface Model work products or the Application Programming Interfaces work product added if applicable to the application.

- Requirements
  - Problem Statement
  - Use Case Model
  - Nonfunctional Requirements
  - Business Case
- Project Management
  - Project Workbook Outline
  - Resource Plan
  - Schedule
  - Test Plan
  - Issues
- Analysis
  - Analysis Object Model
  - Analysis Scenarios
  - Analysis Object Interaction Diagrams
- Design
  - System Architecture
  - Target Environment
  - Design Object Model
  - Design Scenarios
  - Design Object Interaction Diagrams
  - Design Class Descriptions
- Implementation
  - Coding Guidelines
  - Source Code
  - User Support Materials
- Testing
  - Test Cases
- Appendix
  - Glossary

Figure 10-1. *Example of a Project Workbook Outline.*

References

Just this book!

Importance

Essential. If you are going to create a project workbook, you need an outline. There's no way around this.

10.3 RESOURCE PLAN

Description

A Resource Plan identifies requirements of the project in terms of staff, training, equipment, services, and budget. It should state the types and quantities of the resources, and when they are required. This item is a normal project management deliverable.

Purpose

Early in a project's life it is essential to identify all of the resources that will be required during the life of a project for its successful completion. During the life of the project the Resource Plan needs to be updated and reviewed periodically to manage resource exposure risk.

Participants

The project or resource manager would normally perform this task. To acquire the necessary resources it will require negotiation with the resource owners.

Timing

A basic Resource Plan should be completed as early as possible. The plan should then be updated and refined as the project proceeds and more details of the project are available. It should be reviewed regularly to identify risks.

Technique

A possible approach to building a Resource Plan is:

- Choose and obtain a project management tool. A site or development organization will normally have standardized on one, and there are numerous choices. Example tool sets that have been used with the approach described in this book can be found in Appendix C.
- Estimate the resources needed for each phase of the project. The estimates for later stages are likely to be less accurate than the initial ones; therefore, they should be reevaluated as the project progresses.
- Enter this data into a project management system.
- As resource requirements and timings change, update the data in the project management tool.
- Pay heed to warnings about resource exposures that the tool gives.
- Do not underestimate time and effort associated with activities such as installing tools, educating staff on object technology and the tools to be used, and ramping up staffing.

There is little difference in resource planning between traditional development and object technology-based development.

Strengths

It ensures that the project identifies the resources needed for successful completion. It helps in the management of risk.

Weaknesses

It requires effort and discipline to establish and maintain but is a good investment in all but the smallest projects.

Notation

Many available tools support PERT charts, time lines, and critical-path analysis. These capabilities are essential to good project management.

Traceability

This work product has the following traceability:

Impacted by:
- Prioritized Requirements (p. 111)
- Business Case (p. 115)
- Schedule (p. 139)
- Release Plan (p. 144)
- Reuse Plan (p. 158)
- Test Plan (p. 164)
- Metrics (p. 169)
- Project Dependencies (p. 173)
- Issues (p. 176)
- Subject Areas (p. 187)

Impacts:
- Schedule (p. 139)
- Release Plan (p. 144)

Advice and Guidance

Building a Resource Plan is difficult, but it really needs to be done to have control of a project and to minimize the risks of not having enough resources to successfully complete the project. Some simple advice and guidance follows:

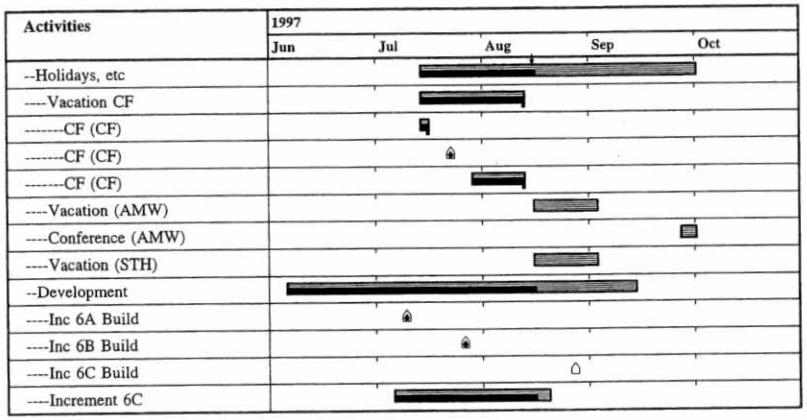
- If this is your first project using an object-oriented software development process and have no idea how to estimate resources, it might be useful to start by estimating the resources as you would if this were not an object-oriented project. You can then add some appropriate buffer (say 10 to 25 percent) to cover the fact that you are using a new approach, and it will likely cost some time and resources during this initial attempt to use it.
- Factor in costs for training, mentors, tools installation and support, process definition and deployment, and other factors related to the move to the use of object technology.
- Revisit the Resource Plan often (certainly after each iteration of your project) and make adjustments based on the new knowledge you will have at hand.
- Keep a history of Resource Plans and adjustments to them to build a base of data from which to estimate resource needs for future object-oriented development projects.

Verification

- Check that all appropriate kinds of resource have been addressed.
- Check that the plan identifies when and for how long each resource is needed.
- Check that the timing of each resource is appropriate; not too early and not too late.
- Check that resources are not overcommitted in the sense that the same resource is required to do too much at once.
- Check that resources specified are adequate for each task.

Example(s)

The following example demonstrates that a Resource Plan should cover specific project activities as well as things like vacations and other "absences" in order to be truly useful:



References

Succeeding with Object Technology by Goldberg and Rubin [Goldberg95], in particular the chapter "Plan and Control a Project."

Importance

A Resource Plan is essential for ensuring that the appropriate resources are available to a project at the proper times. While a very small project might choose to document its Resource Plan informally, it still needs to have some idea of resource needs and availability.

10.4 SCHEDULE

Description

The schedule is a set of:

- Activities
- Start dates and durations for each activity
- Work assignments
- Milestones

It is closely allied to the Resource Plan. At the start of a project a schedule will be created based on the functions to be delivered and the deadlines that have to be met, using project estimation Metrics, comparisons to similar projects, and general experience.

Purpose

The schedule is produced to understand the work completed and the work needed to be completed for the project. It is used to plan and measure project progress.

Initially it will be a feasibility exercise to see whether the project can deliver the required functions on the desired deadlines. During the life of a project, it provides a view of the progress of the project.

Comparison of the actual dates achieved for the items in the schedule to the planned dates provides feedback to the project managers on the progress of the project and whether it is on track. If the project is behind schedule, it provides a driver for corrective action to bring the project back on schedule. The reasons for the differences between planned and actual have to be analyzed to determine their causes so that remedial action can be taken as appropriate. If appropriate, metric data used for project estimations should be updated to reflect actual schedule data.

Participants

The project manager and planner(s) share responsibility for the schedule with help from the project team leaders.

Timing

A basic schedule should be completed as early as possible. The schedule should then be updated and refined as the project proceeds and actual details of the project become available. It should be reviewed regularly to identify divergence from the plan so that corrective action can be initiated as appropriate.

Technique

- Obtain a project management tool. Example tool sets that have been used with the approach described in this book can be found in Appendix C.
- Build an initial schedule by entering milestones, tasks (with durations), resources, and work assignments.
- Often, initially, the minimum task duration is a week. As the project plan is refined, activities can be broken up into more detail and shorter duration activities.
- Link dependent activities together. Group related activities together under a hammock (an abstract superactivity). This helps to avoid micromanagement by hiding small items from the project plan.
- Examine critical path(s) to see if plan alternatives can be found to reduce the criticality. A critical path is a set of linked tasks within a project that have no float (gap between the end of one task and the start of the next task) and end with some deliverable of the project. This means that if any one task in this critical path slips, that deliverable slips and possibly the entire schedule slips.
- Critical paths in a project should be flagged as high-risk elements during risk management planning.

Strengths

It ensures that the project identifies the critical activities that the project team has to achieve on time for successful completion of the project. It alerts project management when the project is slipping behind the plan.

Weaknesses

It requires effort and discipline to establish and maintain but is a good investment in all but the smallest projects.

Notation

There are a variety of notational styles for schedules and they are typically tool-dependent. Please see the example section of this work product description for a look at one possible notation.

Traceability

This work product has the following traceability:

Impacted by:

- Prioritized Requirements (p. 111)
- Business Case (p. 115)
- Intended Development Process (p. 127)
- Resource Plan (p. 135)
- Release Plan (p. 144)
- Risk Management Plan (p. 152)
- Reuse Plan (p. 158)
- Test Plan (p. 164)
- Metrics (p. 169)
- Project Dependencies (p. 173)
- Issues (p. 176)

Impacts:

- Resource Plan (p. 135)
- Release Plan (p. 144)

Advice and Guidance

Building a schedule is not easy, but it is critical to the success of the project. Some simple advice and guidance follows:

- If this is your first project using an object-oriented software development process and have no idea how to build a schedule, it might be useful to start by estimating time needed as you would if this were not an object-oriented project. You can then add some appropriate buffer (say 10 to 25 percent) to cover the fact that you are using a new approach and it will likely cost some time and resources during this initial attempt to use it.
- Factor in time for training, mentors, tools installation and support, process definition and deployment, and other factors related to the move to the use of object technology.
- Revisit the Schedule often (certainly after each iteration of your project) and adjust it based on the new knowledge you will have at hand.
- Keep a history of Schedules and adjustments to them to build a base of data from which to estimate resource needs for future object-oriented development projects.
- Keep initial iterations small both in time and content. This will allow you to begin to shake out your development process and environment and give the team some early experience with object-oriented software development. This can be a valuable tool for getting an early assessment of how good your schedule is and for making any needed adjustments.

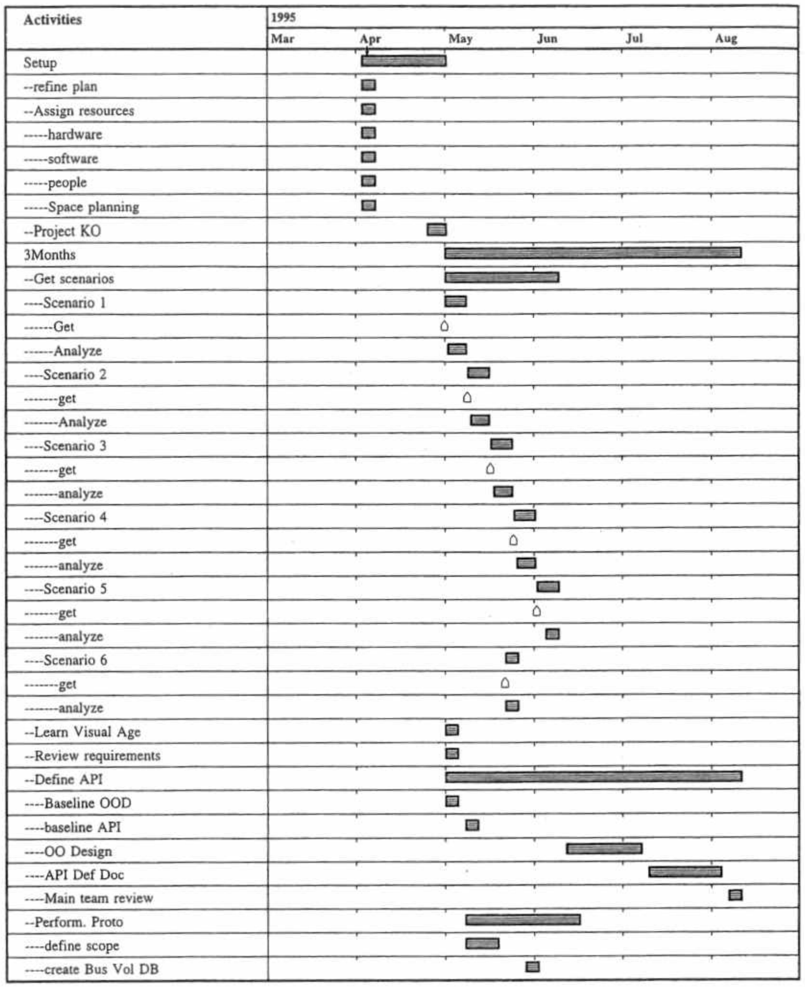
Verification

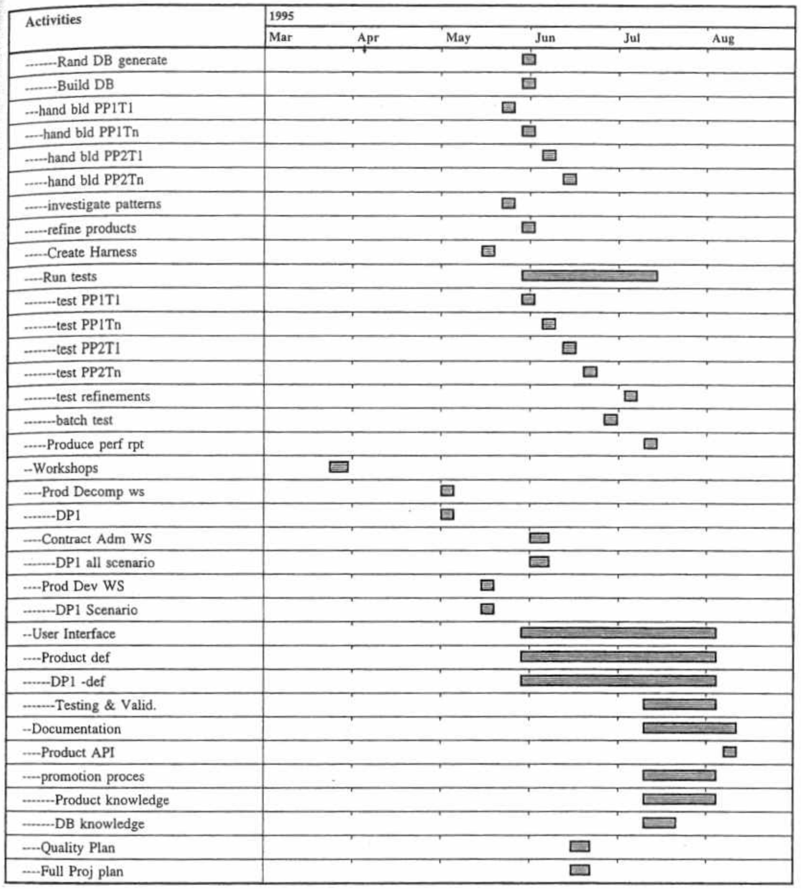
- Check that the schedule takes account of technical dependencies implied by the Subsystem Model.
- Check that the critical part through the schedule is marked.
- Check that the milestones are adequate.
- Check that all identified risks have been addressed in the schedule.

- Check that adequate buffers of time exist in the schedule.

Example(s)

The following example is the initial plan for a three-month feasibility study:





References

Succeeding with Object Technology by Goldberg and Rubin [Goldberg95], in particular the chapters "Plan and Control a Project" and "Case Studies of Process Models."

Importance

Schedules are essential for planning the project and its resources and for tracking progress against that plan.

10.5 RELEASE PLAN

Description

The Release Plan maps the requirements gathered in the Requirements section of the workbook into the releases, versions, iterations, and increments during which the system will be developed (or changed) to handle them.

At the lowest level of granularity, the plan will group the phases and activities that are scheduled. These phases and activities are derived from the Intended Development Process that serves as the model to be followed during the development of the release.

Purpose

The Release Plan is essential to set the scope of the project and divide the work up into manageable units. Having the scope of a release, version, iteration and increment helps to avoid the various forms of paralysis that can stall the project. For example, analysis activities can often result in the identification of additional desired function, the exploration of which can lead to "analysis paralysis" (i.e., the inability to determine when to stop and move on to the next phase). By clearly setting the scope of the requirements to be addressed in each increment, new function that is identified should be allocated to another increment unless its development is certain not to affect the current schedule. New functions that are deemed to be high priority should displace lower priority requirements into later increments if necessary to maintain the plan integrity.

Participants

The Release Plan is the responsibility of the project manager in conjunction with the planners and team leaders. Marketing representatives will also be involved to the extent that they are representing the customers and their requirements. Together, they insure that the overall plan delivers an application that meets the requirements within the window of opportunity and is cost-acceptable to the customers.

Timing

An initial Release Plan is built early in the project cycle; however, it is not a static document. Using an "iterative and incremental" approach, one starts the Release Plan after the requirements have been prioritized and the dependencies between them noted. Customer priorities may change—also technical challenges may arise that require retuning of the plan. After each increment is completed, the plan is adjusted to reflect any changes that have occurred that could affect delivery of the releases.

Technique

Usually, it is best to work first on the highest value, most complex requirements with a near-term window of opportunity. This will make it easier to adjust to problems and/or cancel the project altogether (the latter might occur if it turns out that the problems are insurmountable). It is always better to find the problems sooner than later, so do not put the hard stuff off until later.

To avoid plan "paralysis" do not try to plan the entire project in detail up front. Instead, break the project into 6 to 12 week increments per release, allocating the high priority requirements to the first increments. You might put fewer requirements in the early increments to allow room to adjust for learning curves and other changes to the team productivity "model."

To avoid plan "churn" do not stop work to change the plan for the current increment. If for some reason a work product is not going to be finished on time, decide whether to:

1. "Stretch" the phase to allow it to be completed as planned as part of the current increment. Do this when a high-priority requirement that many others depend on is in jeopardy.

2. "Shift" the work product in question to the next increment to keep the date intact. Do this with lower priority requirements or in early increments where a learning curve (or other forms of "paralysis") may be involved. Moving to the next increment as scheduled can get your team "over the hump."

3. "Share" the work by allocating additional resources to keep both the date and the deliverables of the current increment intact. Do this for a complex work product that can easily be decomposed into two subsystems and worked on separately. Of course, this solution assumes that some resources have been left "in reserve" for this purpose.

Plan some time after each increment to adjust the plan for the later increments and releases. During this period, the complexity measurements of the work products should be used to revise productivity estimates and facilitate Release Plans that are more and more accurate as the project unfolds.

Strengths

- As discussed above, two of the strengths of this approach are that a team can avoid "plan paralysis" and "plan churn." It allows a team to get going in the face of uncertainty and build up a baseline of knowledge to reduce that uncertainty within the lifetime of the project.

- This work product allows the development team and, more importantly, the customer, to see what functionality will be delivered in what time frame.

Weaknesses

This approach makes it more difficult for inexperienced teams to commit to the exact content of any increment or release ahead of time because of the lack of good productivity estimates to apply to the plan. However, the ability to identify problems early and adjust in a systematic fashion without disrupting development more than makes up for the uncertainty at the beginning.

Notation

Free format text is sufficient.

Traceability

This work product has the following traceability:

Impacted by:
- Prioritized Requirements (p. 111)
- Business Case (p. 115)
- Intended Development Process (p. 127)
- Resource Plan (p. 135)
- Schedule (p. 139)
- Risk Management Plan (p. 152)
- Reuse Plan (p. 158)
- Project Dependencies (p. 173)
- Issues (p. 176)

Impacts:
- Resource Plan (p. 135)
- Schedule (p. 139)

Advice and Guidance

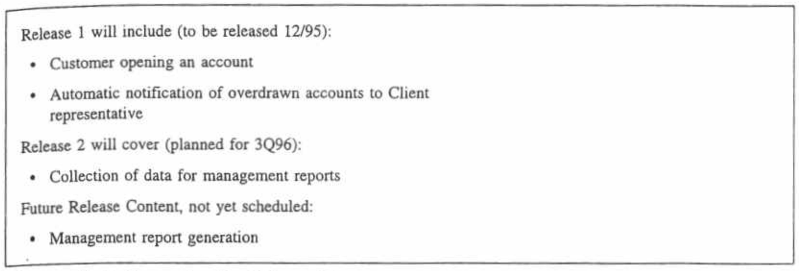
Releases normally are planned with the customer. Usually functionality is prioritized and this is a key driver of releases. It is also impacted by the schedule, the resources available, the technology available, and costs.

Verification

- Check that the functionality of each release can be tested.
- Check that the plan is risk-driven.

Example(s)

While the following is not a complete plan, it should give you some idea of what a Release Plan contains. It is based on a recent banking project:



Release 1 will include (to be released 12/95):

- Customer opening an account
- Automatic notification of overdrawn accounts to Client representative

Release 2 will cover (planned for 3Q96):

- Collection of data for management reports

Future Release Content, not yet scheduled:

- Management report generation

Figure 10-2. Example of a Release Plan.

References

None.

Importance

Optional for a project with only a single release. Essential for projects planning multiple releases.

10.6 QUALITY ASSURANCE PLAN

Description

The Quality Assurance Plan in some form is a traditional part of most software development processes and is not unique to the use of an object-oriented development process.

The Quality Assurance Plan involves:

- Establishing quality goals, defining success criteria, and defining expected results
- Validation and tracking
- Removing defects
- Addressing global quality aspects

A Quality Assurance Plan is a required item in the development processes of many companies. The details of the Quality Assurance Plan should be specified in terms of the measurements that will be taken and the expected results. This will help drive the review part of the process in a much more objective and systematic fashion.

An organization may have a common Quality Assurance Plan that is a template for all development projects. Based on that template each project will have a project specific Quality Assurance Plan. The specific form that a Quality Assurance Plan takes may vary by organization. This section will, therefore, only address Quality Assurance Plans generally.

Quality Assurance Plans generally include:

- Customer Satisfaction goals: overall satisfaction and satisfaction for areas such as usability, capability, and performance
- Code Quality Goals: typically, still, some ratio of "Total Valid Unique Problems per Thousand Lines of Shipped Source Instructions"
- Tools and approaches: some specification of the tools or methods that will be used to track customer satisfaction and code quality
- Defect removal models: showing the number of defects that are to be uncovered at each phase of the development cycle
- A Quality Management section: describing processes for change control and defect management, how quality will be tracked and assessed, and specific quality improvement line items or actions that will be taken

Purpose

A Quality Assurance Plan forces a focus on quality and defect removal throughout the development process and is a valuable tool for ensuring that quality is built into the product from the beginning. If the quality goals are realistic and based on past product results, then the Quality Assurance Plan can allow managers to quickly assess whether the current product is going to achieve its quality objectives.

Additionally there may be other reasons why a Quality Assurance Plan is useful:

- There may be organizationwide quality activities, processes, goals, et cetera, that are best captured in a separate document.
- Quality data is input to future planning (for example, code quality influences maintenance costs, usability influences user support costs).
- ISO 9000 or other processes may require a plan to track quality related aspects of a project separately.

Participants

The project manager owns the Quality Assurance Plan, but typically, many people in related groups such as testing, usability, performance, and service will contribute to the development and implementation of the plan.

Timing

- Quality goals and their verification criteria are one kind of Nonfunctional Requirement. They are defined at the beginning of the project. These quality-related requirements are either given by the organization as cross-project quality goals or are set by the customer. They are quite often based on previous versions of the product or on the results of similar products.

- Quality Plans typically have activities and goals that are performed and tracked throughout the life cycle during the development of the work products.
- Overall results are reported when the project is shipped and they continue to be modified as customer problem data is reported.

Technique

The four aspects listed previously in the Description Section are discussed here in more detail:

1. Establishing quality goals, success criteria, and defining expected results:

   As stated above quality goals and criteria are part of Nonfunctional Requirements. There may be different requirements that address specific aspects such as usability, performance, or code quality. Each product may, because of its nature, place a different importance on different aspects. All functional and Nonfunctional Requirements are addressed and validated in the appropriate work product. For example:

   - Usability may be addressed in the user interface design work product, which is validated through usability tests. For this, the overall quality goal is likely defined with the user.
   - The Acceptance Plan described in Section 9.6, also defines quality criteria that need to addressed in the Quality Assurance Plan.
   - Performance is addressed during design and architecture and validated in a prototype or early drivers.
   - Code quality is an aspect of implementation and will be validated by tests or code inspections.

   A Quality Assurance Plan may then select a few key quality-related requirements and assign priorities. (For example, there may be an organization-specific focus on code quality, which may receive higher priority than usability or a product may have to achieve certain performance measurements in order to be competitive in the market place.)

   These key quality criteria need to be defined together with the customer of the product (or with the funding organization).

2. Validation and tracking

   - Each work product described in this book has advice and guidance on how to validate the work product.
   - Causal analysis and defect prevention are concepts used to improve the quality beyond single projects.
   - Tracking design changes (and errors found during validation activities and testing) will give quantified feedback about quality levels.

- Metrics need to be established that define what is tracked. Section 10.10 introduces various metrics relevant for an object-oriented project.
- The Test Plan, described in Section 10.9 will be used to plan the detailed activities needed for validating and tracking various quality metrics.

3. Removing defects

- As stated above, most Quality Assurance Plans include defect removal models. They normally describe the number of defects that are expected to be uncovered during each phase of the development process.

  The Quality Assurance Plan should also define the steps that will be taken to uncover and remove these defects. These actions can range from reviews (i.e., of analysis or design work products) to testing (of the code at various iterations).

  The object-oriented development process described in this book is very front-end intensive. This suggests that with the proper reviews of analysis and design work products, many defects should be removed early in the development cycle. Additionally, we recommend an iterative and incremental process which, among its other benefits, provides for testing of functional code far earlier in the product development process than would occur using a waterfall approach.

4. Addressing global quality aspects.

   There are various technical or organizational decisions that can be made early in a project life cycle that can have a positive impact on the overall quality of the project. Some examples of these kinds of decisions include:

- Using proven class libraries, such as IBM's Open Class Library, allows the reuse of important functions provided in code of exceptional quality. The Open Class Library has the added advantage of being available on multiple platforms. The library reuses a large, common, platform independent code base. This ensures common quality on all platforms.
- Solutions aimed at a particular customer or set of customers can (and should) include the customer(s) to identify required functions and to help define quality criteria and quality assessment plans.
- Agreeing on a specific object-oriented development approach for the project or across projects will allow you to reap the benefits of object-oriented development.
- Most products can adapt an iterative and incremental development process as described in this book. This allows for early verification of particular implementations of functions.
- Larger development areas can benefit from common quality procedures across multiple product development efforts.

Strengths

A Quality Assurance Plan defines a formalized approach to delivering a product of high quality and demonstrates an organizational commitment to quality. A thorough Quality Assurance Plan can help a project to judge quality and react to any problems throughout the development cycle.

Weaknesses

A Quality Assurance Plan can take a lot of time to develop and execute and as such requires a very strong organizational commitment to a planned approach to quality.

Another pitfall to be aware of is that overemphasis of one or a few quality aspects (too often code quality) can often lead to overlooking other potential quality problems (for example, performance).

Notation

English text or graphics, showing customer feedback or error removal rates, can serve as notation for this work product.

Traceability

This work product has the following traceability:

Impacted by:
- Project Workbook Outline (p. 132)
- Test Plan (p. 164)
- Metrics (p. 169)
- Issues (p. 176)

Impacts:
- Intended Development Process (p. 127)
- Project Workbook Outline (p. 132)
- Test Plan (p. 164)
- Analysis Guidelines (p. 183)
- User Interface Guidelines (p. 234)
- Design Guidelines (p. 253)
- Coding Guidelines (p. 322)

Advice and Guidance

- Ask customers to define key quality requirements
- Follow the Quality Assurance Plans defined for your organization
- Use the Advice and Guidance that are listed for each work product

Example(s)

Please refer to the case study for an example (page 508).

References

- *Succeeding with Object Technology* by Goldberg and Rubin [Goldberg95], in particular the chapter "What is Measurement?"
- The Defect Prevention Process (DPP), found in [Mays90], discusses improving quality by preventing defects from getting into a product.
- Chapter 7 of [Malan96], discusses metrics and defect tracking.
- The International Standards Organization (ISO) has a standard for quality management called ISO 9000. The document: *'ISO 9000 International Standards for Quality Management'* is available from the ISO Central Secretariat.
- [Schulmeyer90] proposes statistical controls over the development process to produce quality software.
- [Kaplan95] discusses 40 innovations that have lead to improved quality software.
- The Total Quality Management (TQM) approach to improving software quality is discussed in [Arthur92].

Importance

Each work product described in this book has a description of how to verify it. This helps to build quality into every aspect of the process recommended herein. A Quality Assurance Plan is therefore optional, unless the organization where the project is performed requires the use of a separate Quality Assurance Plan.

10.7 RISK MANAGEMENT PLAN

Description

A *risk* is anything that may jeopardize the success of the project. Risks in a software project do not relate only to technical matters. Developers must also deal with things like politics, competition, window of opportunity, credibility, reputation, shrinking market, unproven target environment, fuzzy requirements, fickle clients, et cetera. For example, the possibility that a key project architect may leave the company midproject may be a risk that has to be addressed.

A Risk Management Plan identifies risks associated with a project and provides plans to manage them. Risk management is important in any type of software development project, but the use of object technology does introduce certain risks of its own.

A risk, once identified and assessed for probability and impact, may be managed at the following levels.

1. **Elimination of risk**. Some risks can be eliminated entirely. For example, the risk that a particular key architect may leave midproject may be eliminated by bringing into the project someone else with comparable skills. If the architect leaves, the second architect could step in. The event under discussion (architect leaving) may still occur during the project, but it is no longer a risk. Of course, most risks cannot be eliminated in this way.

   One particular form of risk elimination is risk transference, which involves transferring the risk to another party. For example, a project might be concerned about not being able to recover its expenses to develop a product that has questionable market value. This risk could be transferred by having the customer pay expenses up front, the expenses being recoverable if the project is successful. This kind of strategy, of course, only eliminates the risk for the development team.

2. **Risk reduction**. If a risk cannot be eliminated, the next form of risk management to be considered is to try to lower the probability that the event identified as a risk will occur in practice. An example of this in the case of the architect who may leave might be to improve that person's working conditions. Depending on the anticipated reasons for leaving, this strategy may make it less likely that the architect will actually leave.

   Another example of risk reduction is reducing announced release requirements to reduce the probability of schedule slip.

3. **Damage control**. If it has proved impossible to lower sufficiently the probability of the event associated with the risk, then the possibility that the event will actually occur must be faced. If it does, then by definition the project will be harmed in some way. Steps must then be taken to limit this damage as far as possible. As an example, if the architect leaves midproject, an adequately qualified person from elsewhere in the company must be found as a replacement. Before the replacement is found and up-to-speed, the project leader will provide cover.

   Other examples of damage control are car bumpers, sprinkler systems (for fires), circuit breakers (for electrical overload), and running tasks in separate processes (OS/2).

4. **Contingency planning**. If damage control steps are planned, their implementation must be effective as soon as possible after the risk event has occurred. For this to happen, contingency plans must be laid to remove obstacles from the critical path to damage control. For example, if the architect leaves, then the finding of a replacement will be speeded up by preparing in advance a list of qualified people who might be available. Another contingency plan might be to provide time for the project leader to familiarize herself with the application Architecture.

   Other examples of contingency planning are spare parts, backup systems, and archiving.

For each identified risk, the Risk Management Plan should address each of these levels as appropriate. A risk management strategy for handling a particular risk may combine elements of more than one of the risk management levels. This does not matter; the above list should serve as a checklist only.

Purpose

Innovative projects inherently involve risk. It is important for people to recognize and deal with all facets of the risks that they face. Awareness breeds readiness, competence, and confidence. Not recognizing and managing risks will lead to unpleasant surprises that in turn lead to budget and schedule overruns, or noncompetitive products.

Some projects suffer because of factors outside the control of the project, while others are harmed by internal factors. Analyzing and managing the risks in your project can limit damage and reduce wasted investments.

With an iterative and incremental development process, criteria must be agreed upon for deciding how to sequence development work. Probably the most important criterion is risk. Requirements should be partitioned between development cycles to minimize risk to the project. This is impossible without knowing what the risks are, and without a plan to manage them.

Participants

Risk management should be part of the way all project members do business. Of course, broader and more expensive risks should be coordinated by the project management, but everyone should be aware of risks, thereby documenting, and planning for them.

Timing

Risk management is a continuing activity throughout the project life cycle. Obviously, a concentrated focus on risk assessment and planning occurs while the initial project plan is being constructed. Risk management should be done, however, throughout the project life cycle.

Technique

For each project phase (requirements gathering through to maintenance) and for every release and development cycle, all the risks for that phase, component, release, or activity should be identified, weighed, prioritized, assigned ownership, and scheduled for resolution. Risk resolution is the process of determining how to manage a particular risk. The likelihood of failure (the undesirable event identified by the risk actually occurring) compounded by the cost of that failure should be used to rank the risks. Those with the highest rank should be resolved first and assigned to the most competent people for the earliest completion.

Doing this involves looking at nearly everything as a potential risk:

- If we start too late, we will never get done.
- If the network delay is too great, the product won't perform well.

- If the user interface isn't more intuitive and powerful than the competition, the product won't sell.
- If we miss the window of opportunity we will have to be xx percent better to catch up, and we will have yy percent less money to do it.
- If the product is buggy, we will lose our reputation.
- The users are fickle; we don't know what they want.

Besides analyzing each risk directly as it is recognized, groups can run brainstorming sessions to develop checklists of risk management strategies that might be appropriate for different kinds of risk.

Strengths

A risk is anything that may jeopardize the success of the project. Anticipating them and laying plans to manage them is vital for any form of project management. A formal Risk Management Plan helps a manager track risks and acts as a management checklist.

Weaknesses

Risk management requires effort at times when project resources are probably already stretched.

Judgment must be exercised when deciding what is a risk worth managing formally. Categorizing everything as a potential risk will grind the project to a stop.

Notation

A template of the following form can be used to document each identified risk:

Test group name ____________

Description ____________

Owner ____________

Deadline ____________

Dependencies (on other risks) ____________

Likelihood of occurrence ____________

Cost of occurrence (without applying strategy) ____________

Cost of applying strategy ____________

Cost of occurrence (when strategy is applied) ____________

Priority [1 to 10 (max)] ____________

Management strategy ____________

Traceability

This work product has the following traceability:

Impacted by:
- Prioritized Requirements (p. 111)
- Project Dependencies (p. 173)
- Issues (p. 176)

Impacts:
- Schedule (p. 139)
- Release Plan (p. 144)

Advice and Guidance

- Assess the probability of Project Dependencies being met. Some of these may be risks.
- Technical risks may frequently be managed effectively through prototyping. It is usually employed as a risk elimination strategy. Technical risks often take the form of not knowing in advance whether something is technically feasible or efficient. Prototypes are experiments designed to answer these questions and hence eliminate the risk.
- Another technique for managing technical risks is the judicious use of an iterative and incremental development process. Early development cycles may be used to implement a component with which some form of risk is associated, such as low performance. If this implementation is judged to be unacceptable, later development cycles may be used to rework its design and implementation. The use of the development process in this way is damage control. The damage of an unsatisfactory design is controlled by redesign. The contingency planning required is to advance the implementation of this component to a sufficiently early development cycle so that there is adequate time for the subsequent rework if it proves necessary.
- The use of object technology and object-oriented development can introduce some specific risks into a software project where each must be managed. The risks of the use of object technology include the following:
  - The learning curve might swamp the project.
  - Inexperience with the object-oriented approach might result in a low-quality design.
  - The immaturity of object-oriented tools might affect development productivity.
  - An iterative and incremental development schedule might lead to design churn and/or unnecessary development.
  - The emphasis sometimes placed on object-oriented analysis and a lack of experience with object-oriented design may lead developers to try to implement the analysis directly, without adequate consideration of the many design Issues.
  - Uncertainty about the problem or how to proceed might lead to "analysis paralysis."
- An initial checklist for use in brainstorming sessions to identify risk management strategies might include the following:

  - Prototype to eliminate a well-defined technical risk. If the risk can be phrased as a well-defined technical question, for example: Is it efficient to use a relational database to store image objects; the answer can frequently be obtained by constructing a prototype. See Section 17.4 for a discussion of the use of prototypes in risk management.
  - Schedule work in an early development cycle to manage technical risks arising from the uncertainty of untried designs. This differs from prototyping because a prototype is targeted at answering a specific question whereas a development increment is a check of the complete design, albeit a design with reduced functionality.
  - Reduce risks associated with productivity uncertainty by estimating effort using a parameterized formula and comparing the estimated and actual productivity and effort values. Use updated estimates in later development cycles.
  - Reduce the risk that the user interface may not be as the customer wants by involving the customer in user interface prototyping activity.
  - Reduce the risk of external drivers not arriving in time by omitting this functionality from early development cycles, or by building scaffolding to simulate the driver. For example, a project dependent on a database interface to be supplied by another project may omit persistence from the requirements of the first development cycle.
- Each subsystem should have its own Risk Management Plan that should be reprioritized during each development cycle. The highest ranked items, based on whatever realistic weighting scheme the project chooses, should be addressed in the current or imminent cycle.

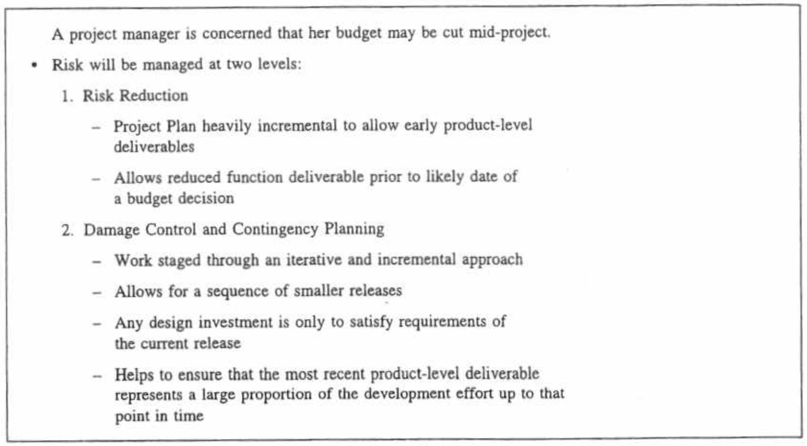
  Leaders of projects involving the parallel development of multiple subsystems should have an integrated Risk Management Plan for the entire system. Subsystem and system plans may be connected either by *delegating* a risk item from the master plan to a particular subsystem, or by *promoting* a risk item from a subsystem plan to the master. Risk promotion may be effected by having the project manager personally weigh and integrate the subsystem risk plans or by having the subsystem team leaders prepare an update to the master plan for the project manager to review.

Verification

- Check that all appropriate kinds of risks have been addressed. This can be done by producing a list of the risks that are appropriate to your project (which varies from project to project), and checking for coverage under each of these headings.

Example(s)

The following is a Risk Management Plan for a project that is under threat of budget cuts:



A project manager is concerned that her budget may be cut mid-project.

- Risk will be managed at two levels:
  1. Risk Reduction
     - Project Plan heavily incremental to allow early product-level deliverables
     - Allows reduced function deliverable prior to likely date of a budget decision
  2. Damage Control and Contingency Planning
     - Work staged through an iterative and incremental approach
     - Allows for a sequence of smaller releases
     - Any design investment is only to satisfy requirements of the current release
     - Helps to ensure that the most recent product-level deliverable represents a large proportion of the development effort up to that point in time

Figure 10-3. *Example of a Risk Management Plan.*

References

- See [Rakos90] for a discussion of risk management in general
- See [Boehm88] on the use of risk management to guide the development process.
- [Boehm89] is a collection of papers on Software Risk Management.

Importance

Optional. But, many projects would find it essential. You must assess the level of risk your project has and decide whether you need this work product. Understanding and planning for risks is an important part of successfully completing software projects.

10.8 REUSE PLAN

Description

A Reuse Plan is a statement of which existing software parts are going to be reused, which reusable parts are to be built, and the costs and benefits of doing this. In the case of building reusable parts, the Reuse Plan must include a business justification and details of how the parts are to be supported.

Effort, costs, and the schedule of a project greatly depend on the amount of "reuse" the project will be doing. This may include savings if the project can reuse existing reusable parts. It also may include additional effort if the project decides to build reusable parts.

So, a Reuse Plan is very intertwined with the Business Case and the overall project plan (Schedule, Resource Plan, and concrete actives such as Design).

A Reuse Plan might also explain why parts of the projects had to be written from scratch and why parts of the project could not be made reusable.

Purpose

Both reusing existing parts and building reusable parts will affect a project plan and have to be taken into consideration.

There are two major items that are affected by reuse:

1. The Business Case (for example, development costs): Saving through reuse will affect the Business Case and may be the decisive factor for an investment decision. Any costs from reusing parts or building parts have to be taken into account as well.

2. The Project Plan (for example Schedule, Resource Plan, and concrete activities such as design): Reusing parts can lead to savings in time and effort and creating reusable parts can increase the required time and effort on a project. Thus effective planning of project activities requires an understanding of what reuse activities are planned.

Development for reusable parts has to be justified, planned, and tracked separately from the rest of the project, so this has to be documented as a separate item in the project plans.

In general, additional efforts have to be planned and there has to be management commitment to these efforts. Otherwise a project under short-term pressure may stop reuse efforts in which the value is not seen until much later in the project or perhaps not until subsequent projects.

If a project does not exploit the benefits of reuse, it may want to document which steps were made to try reuse, which reuse options were considered, and why the reuse options were rejected. Reuse is one of the best ways to improve productivity and quality so it is important to understand impediments to doing it.

Participants

The project planner, the architect, team leaders, and the manager all have input on the project's reuse efforts.

Timing

Planning for reuse should be done when:

- Preparing the Business Case
- Preparing a detailed project plan
- Reviewing project plans after each development cycle

Technique

- Analyze opportunities for reuse (what could be reused and what could be built).
- Use previous experience (your own, or that of other projects), use prototyping, and make estimates for costs and savings based on experience or prototyping.
- Step through all development phases and consider activities, efforts, and savings.
- Make appropriate sizings.
- Factor results into the appropriate plan documents.

Strengths

Reuse requires concerted effort and planning outside of what is "normal" for a development project. Committing those efforts in a written plan will encourage the allocation of the proper time and resource to ensure the necessary activities are done.

Listing reused components will document increased Project Dependencies.

Weaknesses

If reuse is not a strong commitment of the entire project team, including management, the investments needed to identify reuse opportunities and to build reusable parts may be the most likely to be cut when the project experiences schedule pressure. Actual opportunities for reusing existing parts may not be known during project planning. Unless you have past experience and/or a very clear understanding of your technical requirements and the technical capabilities of the reusable part, you may discover during development that you can't use a particular part. Take this into consideration and refine your plan for each development cycle.

Notation

Free format text.

Traceability

This work product has the following traceability:

Impacted by:

- Business Case (p. 115)
- Issues (p. 176)

Impacts:

- Resource Plan (p. 135)
- Schedule (p. 139)
- Release Plan (p. 144)
- Project Dependencies (p. 173)
- Analysis Guidelines (p. 183)
- Design Guidelines (p. 253)
- System Architecture (p. 257)
- Subsystems (p. 274)
- Physical Packaging Plan (p. 326)
- Source Code (p. 334)

Advice and Guidance

For reusing existing parts:

- Reusing existing parts is not free. Plan for the "costs of reuse." If your people are not familiar with the parts you must allocate time to scan available reusable parts, to evaluate the reusable parts early in the project cycle, to "look for reuse opportunities" during inspections, et cetera.
- Plan for the savings, including savings for design, development, test, and maintenance.
- Understand what's available.
- Understand your needs. Depending on the kind of project, this may be easy ("we need just the same as last time") or hard ("we never worked in this domain, so we don't know what we really need").
- Before you commit yourself to use a reusable part, know your requirements and understand if they are met.
- Reusing existing parts creates an additional dependency. You have to manage this dependency. You may have to make compromises in order to be able to use a generally reusable part.
- Consider giving a team member time and responsibility for driving reuse in the project much as you would give someone responsibility for other line items.

For building reusable parts:

- There is no firm requirement to develop a reusable part completely within your project. But ideally you should make a complete analysis and design including aspects outside your immediate need. Otherwise the parts may be too project specific and not reusable in other projects.
- After a (relatively) complete analysis and design you may still decide to implement only the pieces that you need in your project. Any future extensions then should not require changes to the existing code.
- If there is no time to make at any parts fully reusable, you should at least make some effort to enable future projects to "harvest" pieces from your project.
- Building reusable parts increases the short-term costs. There is a rule of thumb that reusable software costs three times as much as "normal" software. These costs may be spread over several projects or iterations, but the actual savings may only occur after the third usage. The decision to spend extra effort and how much will be spent has to depend on usage projections. For example, if parts are only going to be reused a couple of times, extensive documentation will be unnecessary. It may be more effective to get direct help from the owner. On the other hand for parts that are expected to be reused hundreds of times it becomes essential to reduce the cost of reuse by providing good documentation, examples, usage patterns et cetera.

- Be aware that others will become dependent on you.
- Strongly consider building a small separate team responsible for building the reusable parts.
- Get support from a management "sponsor." Usually projects will face schedule pressure. You need strong management support to keep the investment for the reusable parts in your schedule.
- Understand what's available; for example, don't build what's already available.
- Do a domain assessment to select where to invest. A domain assessment helps to decide whether parts are needed in the domain, if the domain is mature enough to invest in parts, or if this is a domain where your organization will work in the future.
- Include the necessary efforts in your plan. For example, make sure there is appropriate time allocated for analysis, design, development, et cetera. for the reusable parts. They don't just "happen."
- Be aware that even after the project is finished, somebody has to maintain the reusable part.
- Consider adding reuse topics to your local process guidelines.

Verification

- Check that all intended reuse work tasks have been planned and appropriate staff assigned.
- Check that effort is being made to search for reusable work products.
- Check that a minimum of new development is being undertaken.
- Check that reasonable effort is being made to enable the work products of the project to be reusable.

Example(s)

The following example shows a potential Reuse Plan of a client-server application. The application is supposed to deal with archiving data.

Reused Assets:

Open Class

The project will use the class libraries that are shipped with VisualAge C++ (Open Class).

Effort: Three members of the development team need training for the use of the class libraries early in the project so that design can be done with those class libraries in mind.

Justification: Rewriting Graphical User Interface classes or collection classes would destroy the Business Case for the project.

DSOM

Distribution will be done through the Distributed System Object Model (DSOM).

Effort: Two members of the team need training for SOM/DSOM.

Justification: Developing a mechanism for distributing objects from scratch would destroy the Business Case for the project.

No application specific reusable parts

After a brief survey we couldn't find any reusable parts in the application domain.

Building Reusable Assets:

Reusable Classes for Archiving Data

The project will not be able to build fully reusable classes or class libraries, but since archiving data is a common problem, there will be an attempt to make this service reusable. The goal will be to have classes that can be easily extracted and made reusable in a separate project.

Effort: To make classes reusable an additional person month will be spent during design.

Justification: The common need for an archive function will offset the small investment needed to make the classes for this function easily extractable.

Domain Model

Although the concrete classes will not be fully reusable, we plan to have a complete model for archiving data.

Effort: To make the domain model as complete as possible, two person months of effort will be spent to interview domain experts and to review the domain model with domain experts.

Justification: It is important that the model for this critical function be complete and accurate.

Figure 10-4. Example of a Reuse Plan.

References

- Chapters 9 through 11 of Goldberg and Rubin [Goldberg95] offer a good introduction on issues related to reuse.
- [Tracz95] provides a good overview of a number of reuse topics.

Importance

Optional, but, for projects doing reuse it is important that any major reuse activities are explicitly documented and committed and would therefore be essential for them.

10.9 TEST PLAN

Description

A Test Plan is a document answering the *who*, *when*, *what* and *how* of testing. It specifies which kinds of tests should be performed and in what order. If any specific environment is needed (hardware, software) it is also described there.

Here the term "testing" refers only to the testing of code. Any noncode testing, such as the validation and verification of work products other than code, can be put either into the Quality Assurance Plan, the Intended Development Process, or the various Guidelines work products.

Planning for testing must take into account the acceptable quality level of the product, as defined in the Quality Assurance Plan, that the project is delivering with given resources, because that is the ultimate requirement that governs the testing. For example, a drawing program might be quite acceptable if it has fewer than one error reported per week of exploitation, while such a rate would clearly not be acceptable for a life-support machine's software. Depending on the required quality level that the product must fulfill, the amount of resources dedicated to testing will be different.

If an iterative and incremental development process is being used, testing is done both as an integral part of each development phase of each increment, and also as a distinct phase in its own right in order to function test the executable deliverables produced by the increment. A Test Plan should refer to both increment function testing as well as testing the whole deliverable.

Purpose

It is produced to ensure that testing is complete, feasible, and successful, and that adequate resources for testing are identified and planned for. A Test Plan ensures that the testing strategy is agreed to in advance.

Participants

A testing team, in cooperation with the team leader, planner, developers, human factors, and information developers build and maintain the plan. Coordination between the testers, the team lead, and the project planner is necessary in order to ensure that development and testing activities are synchronized (in the Test Plan). This is especially important when the development process is incremental and iterative.

Timing

This typically happens at the same time that the project and iteration plans are made. Like any other planning activity, it is impossible to make a final plan before the beginning of the project, but that should not be an excuse for not having a plan. An initial Test Plan, as complete as possible, with clearly identified dependencies on the Resource and Release Plans and Schedules, offers a good document that should be updated as the project progresses from iteration to iteration.

Technique

For each activity identified in the iteration plan, an appropriate testing period needs to be allocated in the Test Plan. What the appropriate time is depends on the particular project, iteration, and work products concerned:

- Projects new to object-oriented technology may want to allocate more time for testing to allow for thorough testing of all work products, while projects already experienced in object-oriented technology can plan to focus more on testing those work products that in their previous engagements were identified as not being at the required quality level.
- Every iteration consists of analysis, design and implementation work products, only their ratios are different from iteration to iteration. This results in different times being required for testing code in different iterations.
- Implementation work products (code) can be reviewed, but are largely tested by running Test Cases that test different aspects of their quality (function, performance, usability, et cetera). Planning to complement code reviews with code testing will yield the best results, as those activities are complementary and used to discover different categories of problems. When planning for testing of code, one must plan resources for Test Case development, execution, and maintenance. If the projected number of Test Cases turns out not to be sufficient (for example, because during development the code was discovered to be more complex than was initially anticipated), the Test Plan should be revisited and updated accordingly. The best approach is to plan to review the Test Plan on a regular basis.

Strengths

- Identifies potential bottlenecks of testing resources.
- Enables planning of resources.

Weaknesses

- If the project and/or iteration plan is incorrect, the Test Plan will likely be wrong as well.
- Not knowing the required quality level of the product may lead to too little testing (potentially dangerous), or too much testing (unnecessary).

Notation

The same syntax used for project and iteration plans can be used. Typically, management tools have scheduling possibilities, the output of which can be printed and included in the Test Plan document.

Traceability

This work product has the following traceability:

Impacted by:
- Use Case Model (p. 96)
- Acceptance Plan (p. 119)
- Quality Assurance Plan (p. 147)
- Metrics (p. 169)
- Issues (p. 176)
- Screen Flows (p. 237)

Impacts:
- Intended Development Process (p. 127)
- Resource Plan (p. 135)
- Schedule (p. 139)
- Quality Assurance Plan (p. 147)
- Analysis Guidelines (p. 183)
- Design Guidelines (p. 253)
- Test Cases (p. 346)

Advice and Guidance

1. Determine the required quality level for the product.
2. Identify all software and hardware dependencies and when they are needed (for example: before the first iteration, after the second iteration, et cetera).
3. Based on the projected size of the project, required quality level, and initial number of iterations, allocate the resources accordingly. The higher the required quality level, the more time is needed in testing. As a very rough estimate, for projects with a required quality level that is:
   - **High**: Plan for 95 percent or more of overall project resources for testing (no errors are acceptable, for example, a nuclear power-plant management system).[10]
   - **Medium**: Plan for between 20 to 50 percent of overall project resources for testing (a few errors are fine, providing the fixes are available quickly, for example a customer solution for assisting bank managers in granting loans);
   - **Low**: Plan for up to 5 percent of overall project resources for testing (a few errors are fine, even if the fixes are not available until the next release, for example a general-purpose text editor).

[10] From a talk given by Nancy Leveson on "System Safety and Software Design" describing experiences about designing software for safety-critical systems, delivered at OOPSLA '94, Portland, Oregon, on October 26, 1994.

Verification

- Check the testing process: Test Case generation, testing, test result logging, and analysis of testing must be as automated as possible.

Example(s)

A project is developing a customer solution for an investment company. It will need to be able to handle information pertaining to about 10,000 customers from several different countries and to process any of their transactions (transfer, buy, and sell) in no more than five seconds. The customer's hardware is one RS/6000® 990, as a central server, and five RS/6000 590's, as local servers for each country. The operating system of all machines is AIX 4.1.2, with TCP/IP communication over fiber-optic cables between the servers. The contract states that the solution provider will provide 24 hour on-line help, and will find and fix any reported problem within 24 hours from notification, otherwise a penalty of $20,000 per exceeding day is due.

The initial analysis model consists of 20 classes and 7 Use Cases that expand into 16 scenarios. The design model is likely to add 20 new classes, bringing the total to 40 classes. DB2/6000 will be used for storing all the customer and transaction data. The development is planned as four iterations of six weeks, as follows:

1. User management scenarios
2. User transactions on a local machine
3. Concurrent user transactions
4. Backup and recovery procedures.

Given information about the project, we note the following:

1. The required quality level for the product is medium to high.
2. The hardware requirements are:
   - One RS/6000 990 with 30GB disks for data and 512MB of RAM (needed before iteration 4)
   - Five RS/6000 590 with 6GB disks for data and 512MB of RAM (one needed before iteration 1, the rest needed before iteration 3)
   - Fiber-optic communication boards (needed before iteration 3)
3. The Software requirements are (needed before iteration 1):
   - AIX 4.1.2
   - TCP/IP
   - DB2/6000

As this is a project with a medium to high required quality level, the testing time should take between 50 to 70 percent of resources, as follows:

- Before iteration 1:
  - Get the testing hardware and software (note that this is different from the Development Environment).
  - Prepare the testing environment.
- During iteration 1:
  - Plan for four days for review of analysis and design models.
  - Based on accepted design model, plan for designing, implementing, and executing five Test Cases per class and five Test Cases per scenario for code being developed during this iteration.
- During iteration 2:
  - Plan for three days for review of analysis and design models.
  - Based on accepted design model plan for designing, implementing, and executing five Test Cases per class and five Test Cases per scenario for code being developed during this iteration, and an additional 40 Test Cases that explicitly test the integration with code developed during iteration 1. Re-execute all the Test Cases that failed during iteration 1.
- During iteration 3:
  - Plan for two days for review of analysis and design models.
  - Based on accepted design model plan for designing, implementing, and executing five Test Cases per class and five Test Cases per scenario for code being developed during this iteration, and an additional 20 Test Cases that explicitly test the integration with code developed during iterations 1 and 2. Also plan for design, implementation, and execution of 30 performance and 10 stress Test Cases. Do the usability testing with at least 10 people. Re-execute all the Test Cases that failed during iterations 1 and 2.^{00}
- During iteration 4:
  - Plan for a day for review of analysis and design models.
  - Based on accepted design model plan for designing, implementing, and executing five Test Cases per class and five Test Cases per scenario for code being developed during this iteration, and an additional 20 Test Cases that explicitly test the integration with code developed during iterations 1, 2, and 3. Continue with performance and usability testing. Do the regression testing based on a random selection of 40 Test Cases in addition to all that have failed before.^{00}
- After iteration 4:
  - Plan three days for installation test.
  - Plan two weeks for system test in conditions as close to real life as possible.
  - Plan three days for "internal" user acceptance test.

Note that in every iteration, the test plan addresses reviewing the analysis and design models. This is done to obtain an understanding of how the application will be used as well as how it is being designed in order to build the necessary test cases for that iteration.

References

- See [Pressman92] for a general discussion of software testing strategies.
- For a general review of object-oriented software testing, see [Siegel96].
- For an overall review of quality processes, see [Whitten89].

Importance

A Test Plan is an essential means for projects to ensure that their products achieve and maintain the required quality levels.

10.10 METRICS

In order to understand and control a project or activity it is necessary to understand the critical factors that will influence its successful delivery. These factors should then be expressed as Metrics that can be estimated and tracked.

Description

Metrics are measurements of essential elements of the development project that enable successful planning and tracking of projects. A measurement is a particular value of a metric at some time. For instance the value of the average lines of code per class metric might be 200 at the end of the first iteration of a project. Metrics fall into four general categories.

Size Metrics Metrics that measure the dimensions of a project and its components. This will include things like number of classes and number of people.

Productivity Metrics Metrics that measure the rate at which project members are able to produce project deliverables.

Quality Metrics Metrics that provide a measure of the quality of the design and implementation.

Reuse Metrics These Metrics focus on the exploitation of existing assets, both internal and external to the project.

There are two distinct uses of these Metrics that are related to the separate tasks of planning and tracking the project. In performing these tasks, estimates must first be made of the size of the project, and of the productivity that will be achieved. This provides the information to do schedule planning. Other Metrics, for example related to quality, must also be selected and estimated as part of the planning process. The chosen set of Metrics should then be tracked during the project. Estimation of metric data is an essential part of project planning; the tracking of the same Metrics is vital for determining progress and conformance to plan. The actual project size may vary from the estimate; it usually grows by a factor of two to three due to poor estimation, poor understanding of the requirements, and the expansion of project scope. Productivity assumptions, like all other metric data estimates, should be validated against the actual productivity being achieved.

Note that, like good objects, individual development work products are responsible for their own metric data. For example, the Design Object Model work product should contain the metric data relevant to it, perhaps total class count and total method count. For each of these, the estimated and actual values should be included. The *Metrics* attribute common to all work products is used to do this. The Metrics work product presented in this section is used to document which Metrics are to be used in the project and to contain summary estimated and actual measured metric data.

Purpose

The definition, collection, and analysis of metrics can be very useful in building project plans and schedules, tracking projects, managing project risks, and providing benchmarks for process improvements. Metrics can:

- Provide data on which to produce a project plan.
- Provide a baseline for tracking project progress.
- Manage risk, which includes schedule and quality.
- Provide benchmarks for development process improvements.

Participants

Many people are involved, but there are two main types of people, users, and collectors of Metrics. Ideally the tools being used in the development process will provide Metrics collection. The project managers, planners, and team leads will be users of the Metrics. The project manager and planner in conjunction with team leaders also select the Metrics to be used.

Timing

During the planning of the project, existing metric data should be used to estimate the resources required for the project. During the life of the project the Metrics should be collected and tracked against the planned values.

Technique

- Decide on the way in which the project will be estimated and tracked in terms of size, schedule, resource expenditure, and quality.
- Choose an appropriate set of Metrics. The following basic Metrics are recommended by the authors:
  - Number of Use Cases (if used).
  - Number of Scenarios for each Use Case.
  - Number of events for each scenario.
  - Number of classes and instances for each scenario.
  - Number of classes at analysis, design, and implementation.
  - Number of methods for each class at analysis, design, and implementation.
  - Lines of code (LOC) for each method.
  - Development effort per class at analysis, design, and implementation.
- Make a projection of the values that will be expected during the development iterations, increments, and phases.
- Collect the actual measurements of the chosen Metrics and review them against the planned values.
- Analyze variations to determine reason, evaluate risk to project, and take remedial actions as appropriate. For example if LOC per method is predicted to be in the range 15 to 40 and a method is found to have 150 LOC, investigate why this is the case. It may be found that there is a rational reason, the method is implementing a complex algorithm for instance.
- Complexity Metrics can also be collected. These Metrics measure things like class coupling, cohesion, and method complexity.

See the references section of this work product description for references that can provide more complete sets of possible metrics.

Strengths

A set of Metrics by which a project is planned and tracked is essential for a successful project. They help to make the assumptions of the project plan explicit, and provide a means to measure conformance to the plan. Metrics from previous projects enable better use of resources and more accurate planning.

Weaknesses

Getting developers to collect Metrics is a difficult task. Automatic collection by tools should be strongly considered.

Notation

If using a tool, it will have a particular format for collecting data.

Traceability

This work product has the following traceability:

Impacted by:

- Issues (p. 176)

Impacts:

- Resource Plan (p. 135)
- Schedule (p. 139)
- Quality Assurance Plan (p. 147)
- Test Plan (p. 164)

Advice and Guidance

- Determine which set of metrics are most important to your ability to plan, run, and track your project.
- Using tools, basic Metrics can be collected for tracking purposes and for subsequent use in estimating. Tools can be very useful in monitoring Metrics that seem to be out of line.

Verification

- Check that the parameters of heuristics or algorithms used to estimate the project size are being tracked.
- Check that the measurement data needed to improve the process of estimating your next project are being gathered.
- Check that the data required to check quality targets are being gathered.

Example(s)

Here is a very brief example from a project that was piloting object technology and the use of C++. It is provided only as an example of the types of metrics that could be interesting and useful to collect.

Table 10-1 (Page 1 of 2). *Example of Project Metrics.*

Metric	Actual Values
Number of Classes Implemented (total)	24
Root Classes (total)	11
Number of Classes Reused	3
Lines of code	13423
Number of Methods	194
Maximum depth of hierarchy	3

Table 10-1 (Page 2 of 2). *Example of Project Metrics.*

Metric	Actual Values
Analysis Person Days	180
Design Person Days	100
Implementation Person Days	75
Test Person Days	30
Documentation Person Days	40
Education Person Days	240
Methods Per Class	8.1
LOC/Class	559.3
Lines of Code Per Method	69.2

References

- [Lorenz94] discusses specific metrics for objects.
- [Henderson-S96] provides a detailed discussion on various metrics.
- Chapter 20 of Goldberg and Rubin [Goldberg95] offers a good introduction on issues related to metrics and measurement.

Importance

Optional. Our experience indicates that metrics are essential to the efficient planning and tracking of medium and large projects, but we also understand that many projects have managed to complete projects without doing this. Even if they are not needed at a project level, an understanding of metrics can have a great benefit organizationally. The more metrics on hand, the easier to estimate and plan future object-oriented development efforts.

10.11 PROJECT DEPENDENCIES

Description

Project Dependencies are items that your project requires for successful completion. You identify these items and build assumptions that these items will be provided at specific times during the course of your project. If they are not, then the project is at risk of failure. Anything is valid as a Project Dependency, but it most often involves prerequisite artifacts, personnel, skills, hardware, or software function to be delivered by a group outside of the management scope of your project.

Purpose

If you could assume everyone had a memory like an elephant, you would never need to write anything down. Since this is not the case, you need to track the important items that might kill your project. Keeping a record of these items and periodically reviewing them to ensure completeness is essential.

Many times dependencies are not under your span of control. These items need to be watched carefully, since their owners may not have the same priorities as you. Should a dependency default, you need to know as quickly as possible to be able to react.

Participants

The planner is responsible for maintaining the list. Everyone in the project is responsible for identifying dependencies and assumptions. The project manager may need to get involved to resolve issues with organizations that own specific dependencies.

Timing

The list should be created when the project and iteration plans are made. At this time only global assumptions may be identified. Dependencies may only be flushed out during low level design or implementation, so a checkpoint during each iteration must be created to review status of existing dependencies and assumptions. Any fallout from this review should be addressed and new items should be added. This checkpoint can be placed at the end of each iteration.

Technique

Each dependency should be documented separately and tracked. These items should also be integrated into the project plan. Any dependency that is in the critical path must be flagged and tracked tightly as a key plan item.

Dependencies on other organizations should be discussed with that organization. If critical, a document of understanding may be created to ensure both parties understand the agreement. At a minimum, for critical dependencies, they should have in their project plan a reference to your dependency date.

Dependencies should be kept in a database of sorts for tracking, updating, and reporting.

Strengths

If dependencies can be integrated into the project plan, you should be able to know if they are being met, and if there are any problems, you can react quickly. Maintaining a list and reviewing it frequently will help in keeping the project on track and possibly identifying other assumptions or dependencies.

Weaknesses

In the past, this type of activity was documented very poorly, if at all. Most of the time, assumptions existed in the heads of the architect or developers and were forgotten or not shared with the rest of team. Building a worthwhile list is dependent on the team's participation and commitment to help to identify, document, track, and resolve dependencies.

Notation

Each dependency should have the following information provided:

- Short description of dependency
- Full explanation of dependency
- Organization responsible for dependency and responsible contact
- Date dependency required/committed
- Person in your team who is tracking this dependency
- History of activity and current status

Traceability

This work product has the following traceability:

Impacted by:
- Reuse Plan (p. 158)
- Issues (p. 176)

Impacts:
- Resource Plan (p. 135)
- Schedule (p. 139)
- Release Plan (p. 144)
- Risk Management Plan (p. 152)
- System Architecture (p. 257)
- Subsystems (p. 274)
- Physical Packaging Plan (p. 326)

Advice and Guidance

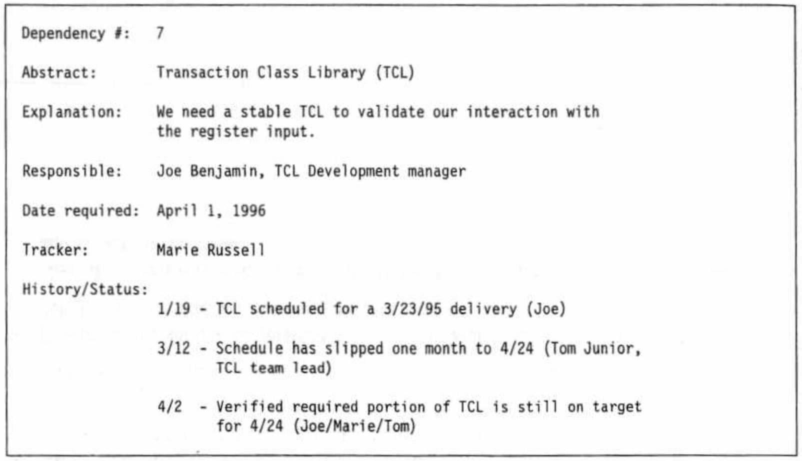
- Make sure the owner of what you are dependent on is aware of your dependency. If possible, have them document a date in their plan to provide the deliverable to you.

Verification

- Check that the people and projects on which you are dependent are aware of your dependency.
- Check that a process is in place for tracking each dependency.

Example(s)

The following example is for a project with a dependency on receiving a class library in stable, working order:



```
Dependency #:   7

Abstract:       Transaction Class Library (TCL)

Explanation:    We need a stable TCL to validate our interaction with
                the register input.

Responsible:    Joe Benjamin, TCL Development manager

Date required:  April 1, 1996

Tracker:        Marie Russell

History/Status:
                1/19 - TCL scheduled for a 3/23/95 delivery (Joe)

                3/12 - Schedule has slipped one month to 4/24 (Tom Junior,
                       TCL team lead)

                4/2  - Verified required portion of TCL is still on target
                       for 4/24 (Joe/Marie/Tom)
```

Figure 10-5. *Example of a Project Dependency.*

References

None.

Importance

Optional. But, you should assess whether your particular project does have dependencies. If it has a number of them then you might find this work product to be essential. Your project should record dependencies in some manner and periodically review their status and effect upon your schedule.

10.12 ISSUES

Description

During the software development process, it will often occur that there are disagreements, areas of uncertainty that require clarification, or questions requiring answers or decisions by users or a sponsor. These should be recorded as issues to ensure that they are not forgotten. All resolved issues or decisions made should be recorded and remembered. Recording decisions is an important aspect of this activity.

Note that the issues being discussed here are global, project impacting issues (ones that a project manager or team lead would want to be aware of). We have said elsewhere in this document that an individual work product may have issues associated with it. Those issues should be considered private to the work product owner.

Purpose

Issues are worth their weight in gold. It is vital that issues are surfaced, recorded, tracked, and resolved in order to keep the project moving forward as planned.

Identifying and resolving issues is a major objective of the application development process. The earlier in a project that issues can be identified, the better.

Participants

The collection of all issues has an owner who is responsible for ensuring that all issues are resolved in a timely fashion. The project manager has the ultimate responsibility for ensuring issues are tracked and resolved. This responsibility is often shared with team leaders. Anyone can identify and submit an issue.

Each issue is assigned to someone who is responsible for its resolution.

Timing

Throughout the duration of the project.

Technique

Gathering issues is a natural part of the application development process and they should be actively solicited. Any area of disagreement should be recorded and added to the issues list. As part of the project management process, issues should be reviewed on a regular basis. They should be prioritized, assigned, and due dates for resolution determined.

Strengths

Tracking issues is simple and a powerful means of ensuring that the correct application is built.

Weaknesses

For the effort involved when compared with the risk of not recording and tracking them, there is no downside.

Notation

Issues should be dated, numbered, and kept in a list or a book or even on-line in a Lotus Notes® database. The general form for an issue is:

- Number or identifier
- Title
- Owner
- Assignee
- Status (such as unassigned, assigned, or closed)
- Description
- Priority (such as High, Medium, or Low)

- Open date
- Close date
- Due date (used when there are activities dependent on resolution of this issue)
- Action plan
- Activity log
- Decision (record rejected alternatives)

There are many possible outlines for an issue and it should be structured to meet the project's needs.

Note that some of these attributes are inherited from the common work product attributes already defined (Section 8.1) and so are redundant. They are reiterated here for emphasis.

Traceability

This work product has the following traceability:

Impacted by:	**Impacts:**
• None.	• All Other Work Products

Advice and Guidance

- It is best to have someone on the team assigned the responsibility of managing the issues list.
- The transcribe and converge technique described in Section 18.3 is a terrific source of issues. You will frequently observe differences of opinion when using this technique. These differences often merit recording for subsequent review.
- Watch out for emotional arguments during analysis and/or design. It is best to record them as issues and move on. The assignee can then do whatever fact-finding is best and present findings to the team.
- If you ever encounter a situation where there is consensus but an individual still holds out for his or her position (often identifiable by the phrase "Yes, but ..."), suggest recording as an issue for later investigation. Often, over time, the issue goes away.
- Issue resolution: OIDs (see Section 11.5) are a useful means of working out a resolution. Either find or invent a scenario that covers the area of dispute, construct an OID that addresses the concern, and see how it works out.
- Issues should be reviewed on a regular basis by management.
- Closed issues should be retained for future reference, and they should include proper detail to ensure that the reasoning can be recalled.
- Note that decisions made, even if made on the spot, normally represent identification and closure of an issue.

For example, during a planning session someone asks, "What UI standard are we following?" and after a very brief discussion, it is decided that the CUA89 standard will be adopted. This should be documented as the issue, "What UI standard will be adopted" and then the reasons for the decision should be recorded.

- The use of Lotus Notes, or some similar mechanism, for managing an Issues database is recommended.

Verification

- Ensure that the issue list has an owner.
- Check that all issues are being tracked. During most project-related meetings, someone should be responsible for recording issues.
- Check that the status of all issues is correct.

Example(s)

The following issue is from a project to develop a tool for host database management:

Table 10-2. Example of an Issue.

Issue: 29

Title: Resolve Alert Management Process

Owner: Charles

Assignee: Mary

Status: Closed

Description: It was recorded in the requirements that alerts would be sent to an Operator's console or proxy. However during analysis, the domain expert (Mike) recommended that the notification should go directly to the Data Base Administrator.

Priority: Urgent (Alert design cannot proceed until resolved).

Open date: Feb. 23, 1996

Close date: Mar. 11, 1996

Action Plan: Resolve by asking the expected users of the application

Activity log:

- 2/25 - Mary asked our second expert (Sheila) her opinion. She suggested this should be a programmable option that would be specified at install time.
- 3/1 - Mary asked User Council. After much discussion, they favored the programmable option.
- 3/3 - met with design team leads to discuss cost. Determined that this would be a costly addition as install program development has already begun.
- 3/4 - asked sponsor who concurred with the decision below.

Decision: While the suggestion has merit, due to the impact on the project, it has been decided that this will not be built into the first release—especially as the User Council did not express strong views. It will be closed and recorded as a suggestion for Release 2.

References

None.

Importance

Essential. Identifying and resolving issues drive the software development process.

11.0 Analysis Work Products

The Analysis portion of the project workbook details those work products that are created during the analysis phase of the project.

Analysis is the separation of a whole into its component parts; an examination of a problem, its elements, and its relationships.

Object-oriented analysis is the process of identifying objects that are relevant to the problem to be solved and their relationships. The process includes classifying the objects and finding relationships among the classes.

During object-oriented analysis, we apply object-oriented methods and techniques to understand, develop, and communicate the functional requirements of an application.

From the modeling viewpoint, we model the problem domain by focusing on both the static and the dynamic aspects of a problem. The static and dynamic work products are separate, but tightly coupled, as is their development.

The *static* aspects are best described in an object model, which shows the objects in the system and how they relate to other objects.

The *dynamic* aspects, the transition between object states and the interaction between objects, are best explored by analyzing the behavior of objects as they collectively fulfill the goals set up by the Use Cases. Use Cases and their Scenarios provide the input to the dynamic modeling process in the form of object interaction diagrams. Such diagrams also provide sample test cases for the system.

Analysis is focused on the *problem domain*, and not on the application. Thus, as far as possible, the only classes that should be included in an analysis are those that an end user would recognize. An area that is problematic is the system boundary. On the one hand the analysis is restricted by the boundaries of the application; on the other hand the design of the system interfaces is not an analysis concern. A solution that works in practice is to produce *a problem domain analysis that covers the application*. That is, it is aimed squarely at the problem domain (and not at the application *per se*), and it ignores as far as possible all the details of the system interfaces. The analysis *covers* the application in the sense that it includes the abstractions mentioned in the Use Case Model, but is not much broader in scope.

According to this viewpoint, the Actors of the Use Case Model become analysis classes with responsibilities and collaborations like any other analysis classes. After all, they are first-class entities in the problem domain. This enables communications across the system boundary to be described in abstract terms during analysis, deferring their design till design time.

One of the problems many teams have to deal with is to decide how detailed the analysis should be and/or for how long one should perform the analysis activities. *Analysis paralysis* is the name given to the phenomenon of a team being extremely reluctant to leave the analysis phase. The analysis phase as a result consumes much more resource than planned. The following is a list of rules of thumb for avoiding analysis paralysis.

- Adopt an iterative and incremental development process. This means accepting, among other things, that the analysis of early cycles is provisional and subject to change following feedback from developers and customers.
- If an iterative and incremental approach is used, define the requirements of each development cycle as a series of scenarios. Only perform enough analysis to enable the scenarios of the current cycle to be expressed.
- Obtain feedback from customers and domain experts. One cause of analysis paralysis is a lack of confidence in the domain. Ask the experts.
- If analysis discussion is getting bogged down, make a working assumption and record the matter as an Issue to be checked off-line.
- Use predefined, although maybe rotating, roles during each analysis session. Roles should include a leader, a timekeeper, and a scribe. The role of leader should not rotate.
- Do not use modeling CASE tools until the central analysis work products have stabilized.
- Consider using a depth-first approach (see Section 17.1) to development. One of the causes of analysis paralysis is uncertainty in the boundary between analysis and design. Depth-first development ignores this boundary for a small, initial set of scenarios.
- Beware of introducing design detail into analysis. If in doubt, err on the side of minimal analysis. Design details are characterized by constraints. If an analysis team finds itself worrying about efficiency or performance, then it has crossed the border into design.
- Try using transcribe and converge (see Section 18.3) to achieve consensus.

The analysis section of the project workbook consists of the following work products:

- Guidelines
- Subject Areas
- Object Model
- Scenarios
- Object Interaction Diagrams
- State Models
- Class Descriptions

These work products all "inherit" the common work product attributes described in Section 8.1, and they have specialized attributes of their own.

Analysis Guidelines are documented rules on how analysis will be performed in the project. The intent is to ensure that process for developing the other analysis work products and the format in which they will be delivered is consistent and well-understood across the development team.

Subject Areas allow a large system to be partitioned into smaller and distinct business domains allowing for a more effective divide and conquer approach to analysis.

The Object Model, a static model of the problem domain, is a critical work product that provides a decomposition of the system into classes of objects.

Analysis Scenarios provide refinements of the Use Case Model and are necessary for building Analysis Object Interaction Diagrams that provide a graphical representation of the interactions between objects.

State Models describe the life cycle of classes in graphical form and can lead to better understanding of the nature of a class.

Analysis Class Descriptions provide a summary of information about classes.

Much more detail is provided on these work products in the following sections.

11.1 ANALYSIS GUIDELINES

Description

Analysis Guidelines are the set of rules intended to document the way in which analysis is to be performed on a particular project. There are, in general, two kinds of Analysis Guidelines: work product guidelines and process guidelines.

Work product guidelines describe the nature of the analysis work products that are to be produced by this project. This includes both the range of work products expected and the nature of each. This information will, in general, depend on both application and team, although many guidelines will be common. In addition to describing standard notations that should be used, the analysis work product guidelines should add project-specific rules, if any, for documenting analyses. These additional rules often fall into the following categories.

- Boilerplate conventions, which describe the overall format of the analysis work products. This information can be provided most simply by a set of templates. Documentation templates too should be provided, for example to describe the information to be provided per class, per association, et cetera.
- Naming conventions, if relevant.
- Diagramming conventions. If any variations on the basic notation are to be used then they should be documented here.

Process guidelines provide guidance on the process by which analysis is to be performed. These guidelines may take the form of a list of suggested techniques, or may be more prescriptive. The guidelines might also include tools advice and recommendations for the usage of the tools.

Purpose

The point of Analysis Guidelines is to ensure that the analysis deliverables and process are planned in advance, and that team members are consistent in their application of the process to achieve the deliverables. This does not mean that analysis must necessarily be planned in a step by step manner, but that thought is put into the analysis procedures and are made publicly available. The last thing that a team needs is to be involved in procedural discussions when it should be doing the analysis itself. The existence of guidelines also acts as insurance that there will be relatively few surprises when the analysis deliverables come to be reviewed. The reviewed analysis work products may contain mistakes, but their format should be as expected.

Participants

The team leader and analysts are the people most likely to document the Analysis Guidelines. The guidelines must be written by someone with considerable experience in software engineering, and object-oriented analysis in particular.

Timing

The guidelines must obviously be in place before any analysis activity is started. If there is to be any analysis education then this must be consistent with the guidelines. If the education is just-in-time then the guidelines should be established before the education is delivered, and the education should take them into account, if this is possible. This involves the educators being flexible, and giving them sufficient time to tailor their material. The alternative is to adopt the set of guidelines taught by the educator provided that these are appropriate and well documented.

Technique

Beg, steal, or borrow some existing guidelines. Interview experienced developers to check that the guidelines are reasonable and complete. Consider asking a mentor for advice on Analysis Guidelines, or to comment on those that have already been assembled.

Strengths

Ensures that analysis time is spent doing analysis, as much as possible, and not in meta-discussions on what analysis is, or how to do it.

Weaknesses

A potential problem is that the guidelines may be overly restrictive or prescriptive. The author of the guidelines should only provide rules for cases in which, if they were not followed, it would be difficult to understand or review the work product, or the work product would be hard to use subsequently.

Another potential drawback is that the writing of guidelines will be seen as a distraction from the real work. The guidelines should be minimal, within the constraint that they are sufficient to ensure that the analysis work products are understandable, usable, and consistent.

Notation

Free format text augmented by work product templates or examples.

Traceability

This work product has the following traceability:

Impacted by:

- Intended Development Process (p. 127)
- Project Workbook Outline (p. 132)
- Quality Assurance Plan (p. 147)
- Reuse Plan (p. 158)
- Test Plan (p. 164)
- Issues (p. 176)

Impacts:

- Project Workbook Outline (p. 132)
- Subject Areas (p. 187)
- Analysis Object Model (p. 192)
- Analysis Scenarios (p. 203)
- Analysis OIDs (p. 208)
- Analysis State Models (p. 219)
- Analysis Class Descriptions (p. 227)

Advice and Guidance

- Group the guidelines into lists of guidelines by work product.
- Use work product templates where relevant.
- Use standard guidelines where they are available and appropriate. Modify these only as necessary.
- If they are novel, publish your guidelines for others to comment upon and to use.
- Harvest good templates from the workbooks of other projects.
- Use the guidelines as entry criteria to analysis reviews.
- If the project team consists largely of object-oriented novices, guidelines that are more prescriptive might be appropriate.
- Briefly review the guidelines at the end of each iteration to check their adequacy and completeness.
- Change or add to the guidelines during the project if this is considered necessary.

- Include statements about what analysis is and what it is not. These statements should be backed up by some criteria for deciding when to stop analysis. For example, it might be stated that if analysis modeling activity has started to discuss alternative ways in which the system may be designed or implemented, then it has strayed too far towards design.

Verification

- Check for coverage of each analysis work product. That is, use the list of analysis work products as a checklist of headings under which to add guidelines.
- Check that adequate guidance is provided for the usage of all tools, where appropriate.
- Check that the guidelines provide for maximal integration of analysis work products. For example, are the Analysis Class Descriptions being generated automatically from the Analysis Object Model and other models?

Example(s)

Analysis Guidelines for a particular development team or site might include the following:

- Use Object Modeling Technique (OMT) as a modeling notation and Select OMT as a modeling tool. The tool should be used for documentation only; during modeling sessions use white boards, post-it stickers that double as Class Description cards, and a flip chart for Glossary entries.
- All classes, as soon as they are proposed, should be defined with a Glossary entry.
- Glossary entries, when stable, should be transferred to Select OMT®.
- The analysis process is highly iterative, but the following cycle should broadly be followed.
  - From each of the requirements Use Cases generate a set of Scenarios by considering the different sets of assumptions under which the system would behave in an essentially "straight-line" (unconditional) manner.
  - Generate an initial object model, possibly preceded by a semantic net brainstorming session to generate ideas. Focus on the essence of the business in question and ignore infrastructure and peripheral concerns. Do not worry about cardinality or aggregation at this point. Use transcribe and converge to achieve consensus on the key concepts and relationships. Do not worry for the moment about whether the classes are strictly required, or about identifying generalizations.
  - Produce an object interaction diagram for each scenario. This involves identifying responsibilities and assigning them to classes. As this is done, add the responsibilities to the class descriptions. Distribute responsibilities evenly throughout the model as far as possible. Where helpful, but only then, identify data attributes. Iterate between OID modeling and object modeling in order resolve any tensions that arise. Do not iterate after each OID, as the model will be too unstable if this is done. Wait till a critical mass of OIDs, perhaps five, are available before returning to the object model.
  - Document the outcomes of each scenario in terms of objects created, destroyed, and modified.
  - Optimize the object model by pruning disconnected classes, identifying generalization/specialization hierarchies, and promoting responsibilities and relationships up the hierarchies if appropriate.
- Avoid including design detail in analysis models. Design detail is characterized by constraints: if proposed detail is included only to satisfy a constraint, omit it.
- Be sure not to do more modeling work than necessary: Only do a sufficient amount to understand how the object model can support the scenarios, without introducing concerns about constraints or other Nonfunctional Requirements.
- For those classes with important and significant dynamic aspects, such as transactions, units of work, or some real-time interfacing classes, draw state transition diagrams to capture their dynamics. Use the OIDs to produce an initial diagram, which is then completed manually.

References

None.

Importance

Optional, but it is our opinion that Analysis Guidelines are important. The existence of these guidelines can keep analysis focused on analysis issues and ensure that time allotted to analysis is spent productively.

11.2 SUBJECT AREAS

Description

A Subject Area is a distinct domain of interest that can be identified at analysis time. It is a recognizable part of the problem domain that can be analyzed as a separate unit. Subject Areas are to Analysis what Subsystems are to Design. Subject Areas might or might not become Subsystems at design time—that is a design issue.

In an object-oriented approach, Subject Areas tend to be defined as clusters of analysis classes that are closely related to each other by inheritance ("is-a"), aggregation ("has-a"), and other ("uses") associations. So it is common (and useful) to associate key classes with a primary Subject Area—the one with the most closely related classes.

Note, however, that it is unlikely that classes only have relations with other classes in their own Subject Area. Subject Areas are usually not self-contained and it is common to have some classes within a Subject Area interact with classes in other Subject Areas. It is useful to promote these "uses" relations from the class level to the Subject Area level. Thus, Subject Areas will "use" or depend on each other.

Purpose

The use of Subject Areas permits a large system to be partitioned very early in its development cycle. It encourages a separation of analysis concerns, and provides a means of organizing work products. Without Subject Areas, or an equivalent organizing concept, analysis work products can become unmanageable due to their size and quantity. Subject Areas provide a means of breaking the analysis into manageable chunks.

Many large applications have natural Subject Areas that may usefully be explored separately at analysis time. This allows different domain experts to be used for (say) an Account Subject Area and a Audit Subject Area, or it might simply allow parallel development in an organization.

Partitioning an analysis using Subject Areas facilitates reuse as it is then easier to examine and to extract information or whole work products related to a common Subject Area shared by applications.

Participants

The analysts, in consultation with customers, domain experts, and end users partition the analysis into Subject Areas.

Timing

Subject Areas are identified during the analysis phase and used to organize the analysis work.

Technique

Subject Areas are either defined "up front," as a way of getting into the analysis of a large application, or they are introduced during analysis as a means of organizing the evolving analysis work products.

If Subject Areas are defined up front, then a very early analysis step is to identify the general categories of objects suggested by the Problem Statement and Use Cases. Identify the *objects*, especially the indirect objects, in *actor-verb-object* statements of functional requirements. Then cluster them into their natural Subject Areas. This will probably, but not necessarily, take the form of a partitioning of an early version of the Analysis Object Model.

Don't be influenced by Nonfunctional Requirements: "User Interface" may be a great idea for a Subsystem (design), but it's probably not a Subject Area (analysis) from the problem domain. Subject Areas should be structured along lines that domain experts would recognize. Thus, things such as Customer Management or Account Maintenance, but not User Interface or Persistence, would make useful Subject Areas.

If Subject Areas emerge as an organizing principle *during* rather than *before* analysis work, then it will probably be as a result of the realization that the quantity and size of the analysis work products is getting unmanageable, and that some way of organizing them must be found. The primary way to discover Subject Areas once analysis has started is to partition the Analysis Object Model into a few clusters that are essentially independent of each other, but which are closely related internally. The remaining analysis work products can then be allocated to these Subject Areas according to their prime focus. But how do you find the "prime focus"?

Since Use Cases (see Section 9.2) are specified in the form of *actor-verb-object*, we can use the *object* part of each Use Case to associate it with its Subject Area. For example, if an analysis of a banking problem has identified Subject Areas: *Accounts, Journals, Audits, Operations, ...*, then the Use Case: "Customer deposits funds to savings account" would, most likely, be assigned to the *Accounts* Subject Area (We are assuming here that *savings account* is a class in the *Accounts* Subject Area). Note that the direct object of the Use Case ("funds") is rather passive (often an attribute of the verb, e.g., *deposit-funds*) and the indirect object ("savings account") is the active receiver. Also note that the other Subject Areas (e.g., Journal, Audit) will eventually be used by the Account Subject Area, probably during the development of the Analysis Object Interaction Diagram (see Section 11.5).

Strengths

Just as the design of large systems needs organization, so does their analysis. Structuring the analysis work products according to Subject Areas allows each aspect of the application to be examined and understood in isolation. It also facilitates checking for consistency and completeness among the closely related classes within a Subject Area.

Weaknesses

Identifying and maintaining Subject Areas takes some time and organization. Partitioning an analysis into Subject Areas erects barriers that might result in inconsistencies between Subject Areas. Effort must be put into communication and reviewing to ensure that this does not happen.

Notation

A simple table of the Subject Areas is sufficient. For each Subject Area, the following is relevant:

- Name of Subject Area
- Brief description
- Key Classes
- Dependencies (Subject Areas used by this one)
- Workbook

The last item refers to the fact that each Subject Area may be given an entire workbook of its own. This workbook might be a part of the overall system workbook or a distinct book or file. Obviously, a Subject Area (like a Subsystem, see Section 13.5) need only include those work products that are relevant to it. It will probably include most of the analysis work products. At design time it will be decided whether to proceed with design at the Subject Area level by adding design work products to each Subject Area workbook, or to proceed with design at the system level by adding design work products to the main workbook. Subject Area workbooks, like Subsystem workbooks, are logically part of the overall project workbook.

Alternatively, most object-oriented CASE tools support the concept of Subject Areas via *views* or hierarchical *layers*. They take the form of named boxes connected by dependency ("uses") arrows. They often show the key member classes as attributes but hide the description in a "properties" panel. See Figure 25-5 on page 554 for an example of "Category Diagram" generated by Rational Rose.

Traceability

This work product has the following traceability:

Impacted by:

- Problem Statement (p. 93)
- Use Case Model (p. 96)
- Issues (p. 176)
- Analysis Guidelines (p. 183)
- Analysis Object Model (p. 192)
- Analysis OIDs (p. 208)

Impacts:

- Resource Plan (p. 135)

Advice and Guidance

- Ensure that Subject Areas partition the analysis and not the design work products. Subject Area descriptions should refer to the business domain and not to design artifacts.
- Use Subject Areas as candidate Subsystems, but ignore this use of Subject Areas at analysis time.
- Be prepared to adjust Subject Area boundaries, or even to split or merge Subject Areas, as analysis proceeds.
- At design time, each Subject Area may form the basis for one or more Subsystems that may be described and developed in their own workbook.

Verification

- Check that definitions and glossary entries are consistent across Subject Area boundaries.
- Check the completeness of the set of Analysis Scenarios in each Subject Area independently.
- Check the dependencies between Subject Areas implied by the Analysis OIDs.
- Ensure each analysis class is represented in one and only one Subject Area.

Example(s)

The following demonstrates a nongraphical presentation of Subject Areas for a banking application:

Accounts		
	Description	The various types of accounts managed by the bank.
	Key classes	Account, Savings Account, Checking Account, Loan
	Uses	Journals
	Workbook	BANKACCT
Journals		
	Description	Transaction recording facets of the bank.
	Key classes	Log, Transaction Log, ATM Log, EFT Log, Teller Tally
	Uses	Operations
	Workbook	BANKJRNS
Audits		
	Description	Error and fraud detection mechanisms of the bank.
	Key classes	Audit, Teller Audit, Branch Audit, ATM Audi, EFT Audit, Account Audit
	Uses	Journals, Accounts
	Workbook	BANKAUDT
Operations		
	Description	Personnel and scheduling facets of the bank.
	Key classes	Schedule, Bank Schedule, Personnel Schedule, Employee, Manager
	Uses	(none)
	Workbook	BANKOPER

Figure 11-1. Example of Subject Areas.

References

- Our notion of Subject Areas is similar to the "layers" discussed by Grady Booch [Booch94].
- The concept of "domains" of Sally Shlaer and Steve Mellor [Shlaer92] is similar to Subject Areas.

- The concept of "Subjects" of Peter Coad [Coad90] is similar to Subject Areas.
- James Rumbaugh discusses "subsystems" in OMT [Rumbaugh91a]

Importance

Optional in small and medium-sized projects but essential in large projects. Subject Areas provides an important means for dividing analysis work products into more manageable, more understandable pieces.

11.3 ANALYSIS OBJECT MODEL

Description

The Analysis Object Model is a static model of the part of the problem domain relevant to the Problem Statement. In common with the Design Object Model, it consists of classes and relationships between classes. Three kinds of relationships are normally used: associations, aggregations, and generalizations/specializations (inheritance). The Analysis Object Model is a key object-oriented analysis work product.

Purpose

An object model is the fundamental way to document the static aspects of objects in the problem domain. Object modeling is what makes object-oriented development different from traditional development. The basic idea is to decompose a system down into classes of objects that cooperate by passing messages to get the job done.

The power of object modeling, as opposed to data flow or control flow modeling, is that by focusing on modeling complete abstractions (objects), which encapsulate both function and data, it is possible to use the same basic concepts during all development phases from analysis to code. By contrast, one might analyze a problem conventionally in terms of data and then design a solution in terms of function.

Participants

Object modeling is the task of architects and analysts. It is essential to get the active participation of clients and/or domain experts in this modeling activity. Since the real-world objects represented in this model belong to the problem domain; clients, end users, and domain experts are the right audience to give inputs and to validate the model. With their participation, the problem can be better understood, and less mistakes will happen in the analysis.

Timing

The Analysis Object Model is developed primarily during the analysis phase but it needs to be maintained during design or implementation if domain understanding changes.

Technique

- Identify key problem domain abstractions that satisfy the criteria of objects: identity, state, and behavior
- The behavior of objects at analysis time can be activities and/or services
- Define each identified candidate class using a short glossary entry
- Connect the candidate classes by identifying relationships between classes: associations and specializations/generalizations
- Add responsibilities
  - Key attributes
  - Behavior
  - Aggregations
- Check consistency with OIDs and state diagrams
- Iterate until model is stable
- Update class descriptions
- Restructure and refine as required over time

After the Analysis Object Model is complete, if the domain is large enough, you may choose to partition the model into *Subject Areas* (cf. *Class Category* in Booch [Booch94] or *subsystem* in OMT [Rumbaugh91a]). Large models require internal organization [Rumbaugh95a] and partitions help us to group classes and concentrate our attention on a subset of the model at a time. Partitioning borders between analysis and design. The boundary between object-oriented analysis and object-oriented design is not as clear as it was between structured analysis [Demarco79] and structured design [Stevens81]. The clustering of classes can already begin during analysis as is proposed in [Nerson92]. Dividing the analysis into Subject Areas facilitates parallel development by various teams.

Subject Areas are decided by logical criteria aimed at producing a clear and simple analysis. The goal of partitioning a design into Subsystems (Section 13.5) is different. While analysis Subject Areas and design Subsystems might be aligned in some systems, that is incidental and not inherent in the definition of the work products.

Strengths

A key deliverable for capturing and communicating problem domain understanding.

An effective means of communication between team members, customers, and domain experts.

Weaknesses

Only shows the static relationships.

Maintenance of the Analysis Object Model is often neglected. This stems from the (incorrect) assumption that the design supersedes the analysis. This weakness is shared by all analysis work products.

Notation

An Analysis Object Model is a specialized form of work product whose purpose is to capture the relationships between classes of objects in a system or an application. An Analysis Object Model is best represented visually as a class diagram. The following information is usually shown in a class diagram:

- Classes
- Relationships
  - Generalization/specialization (IsA)
  - Association (KnowsAbout)
  - Aggregation (HasA)
- Attributes
- Behavior

Instance objects can also be added if this aids domain understanding.

Some forms of documentation show these aspects separately while others combine them all on one diagram. The choice is up to you and your team members, but our recommendation is that you document all of them in the single diagram described below.

Each of these aspects is discussed in detail below:

Classes: A class can be drawn as a 3-part box, with the class name in the top part, a list of attributes (with optional types) in the middle part, and a list of services or operations (see Figure 11-2).

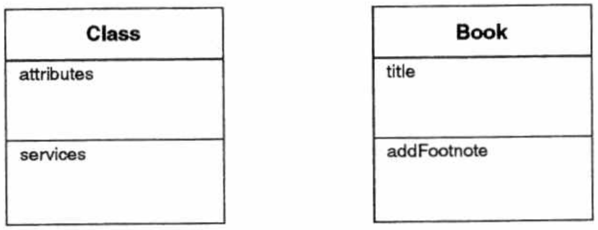


Figure 11-2. *Class Notation.*

Generalization/specialization: Generalization/specialization is usually shown as a hierarchy of super/sub type relationships where the sub types inherit the properties (both attributes and services) of the supertypes. Figure 11-3 shows the notation for class inheritance hierarchy. A Truck, a Car, or a Van are considered to be specializations of Vehicle.

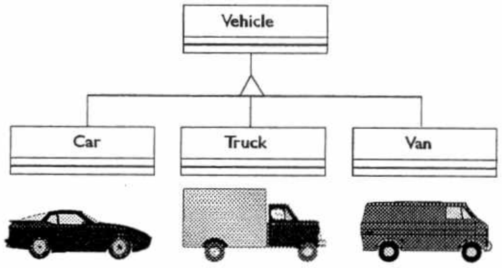


Figure 11-3. *Inheritance Notation.*

Association: Association is the simplest form of relationship. It represents knowledge of the existence of other objects. An association can be thought of as a class in its own right, so that it can support attributes, services and other relationships to enforce the details of the contract. Different notations handle this in different ways, but the basic idea is that associations are the fundamental way to describe how one object uses another to complete a task.

If two classes have an association between them, then instances of these classes are, or might be, linked. These links between instances can be thought of as instances of the association between the classes. Often it is useful to model these links with state and behavior and not just identity. That is, associations can be classes in their own right. For example, the Student-Course association in Figure 11-4 might have the attribute "grade" (*link attribute* in OMT [Rumbaugh95a]).

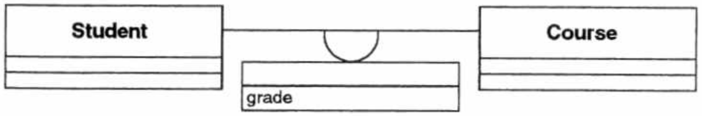


Figure 11-4. *Associations Notation.*

Associations have cardinality. The cardinality shows how many instances of the class can be associated with one instance of the other class. Cardinality can be 0 or 1 (hollow ball), 1 (no marker), 0 or many (solid ball), or some other integer range. Cardinality is important if it is so for the problem domain, for example, a customer can only place one order, et cetera.

An association is, figuratively speaking, the connection through which messages will be passed to access the attributes and services of other model components.

An association is a binary relationship at the analysis level. It reflects *conceptual or physical links* between objects of associated classes [Rumbaugh91a]. At the analysis level, no determination is made whether an association represents conceptual or physical links. This distinction becomes more important at design time and will be discussed later in Section 13.6, Design Object Model. It is sometimes useful to tag associations on an object diagram with a name. These names are often verbs, as the associations usually exist so that instances of one class can "do something" to or with instances of another. For example, Factory and Employee classes might have an "employs" association between them.

Aggregation: Aggregation is a special form of association and shows another view: namely the part-whole hierarchy relating object classes in the model. For example, when a car is decomposed into body and engine (see Figure 11-5), it is thought to "contain" the components, i.e. body and engine; they are its parts. Aggregation often means ownership: The lifetime of the whole encompasses that of its parts. If you are not sure whether a relationship is an association or an aggregation then leave it as association.

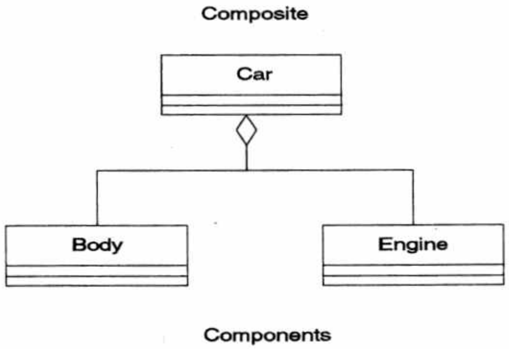


Figure 11-5. Aggregation Notation.

During analysis, aggregations are used to model type composition in a domain. Aggregations can be classified by the cardinality or multiplicity of the aggregate:

- Assembly or container; where a component cannot be part of more than one whole or aggregate (cf. *Aggregates* in [Civelo93])
- Collection; where a composite can have many components [Civelo93]
- Catalog; a component can be used in more than one composite (cf. *Catalog aggregation* [Rumbaugh95a])

Attributes: Attributes represent the structural properties of a class. For example in Figure 11-6 the static properties of the class *Car* are *Make, Model,* and *Year.* Each instance of that class will contain its own set of values (for example, "Porsche," "Carrera," "1987") for the attributes.

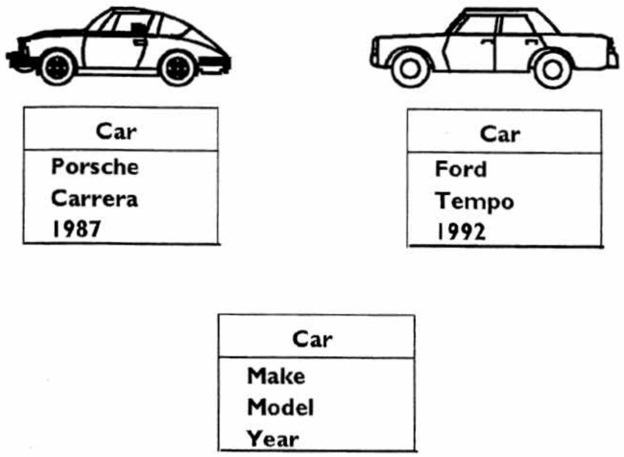


Figure 11-6. Attributes Notation.

Behavior: The behavior of a class, often documented as services or operations, is a statement of the responsibilities of the class. Behavior is what separates object modeling from traditional forms of data modeling (such as data models that result in Entity-Relationship-Attribute diagrams). The fundamental aspect of an object class is to encapsulate both data and function into one package and exploit inheritance, polymorphism, and contracts to get a high degree of reuse. Services encapsulate this function.

During analysis we model real world objects that can be physical or conceptual. The representation of a physical object such as a customer is modeled if it exhibits interesting behavior from the model point of view, in other words, if it carries out activities that influence or communicates with the model. For example, a *member* object (Part 5, Video Store Case Study) could perform the activity: *cancel membership*.

The differences between activities and services are that activities are not invoked. They are performed by objects by their own initiative. However, we treat activities and services in the same way. Conceptually, we could think of an object sending a message to itself. We have a different situation with view class, for example, *CustomerView*. One should not include view classes in an Analysis Object Model, because they do not influence the model classes, and only communicate with the model classes for their own benefit.

The operations or services of a class can have formal parameters, a return type, and a textual description.

Traceability

This work product has the following traceability:

Impacted by:

- Issues (p. 176)
- Analysis Guidelines (p. 183)
- Analysis OIDs (p. 208)

Impacts:

- Subject Areas (p. 187)
- Analysis OIDs (p. 208)
- Analysis State Models (p. 219)
- Analysis Class Descriptions (p. 227)
- Design Object Model (p. 281)
- Glossary (p. 355)

Advice and Guidance

- Don't overburden the modeling notation with unnecessary complexities. The simpler, the better. Object models must be readable by the expert and the beginner.
- The model should not contain any design decisions. Design, not analysis, is driven by the system's Nonfunctional Requirements that represent constraints in the way that the system works; for example, performance or availability constraints. Do not add detail to an analysis model simply to satisfy a Nonfunctional Requirement. Leave it for the design work products.
- Objects in the Analysis Object Model should relate to problem domain objects and mean something to the end user.
- Avoid being overly abstract; use the names that people familiar with the domain use (for example, in the bank-lending domain, name objects and services as a Loans Officer would).

- Name objects and services consistently and meaningfully.
  - Name object classes with common noun phrases (for example, customer)
  - Name services that modify objects with active verbs
  - Name services that query objects with verbs indicating queries
- Watch out for objects like computers or databases that may represent implementation constructs. Ask the user what the function is that they provide and try to name accordingly (for example, call a customer database a customer repository).
- Avoid controller objects that control the rest. One (slightly tongue-in-cheek) test for controller objects: Ask all developers which object they would not like to implement. If they all agree on one then you have a controller. Controller classes are often characterized by names such as "controller," "manager," and the like. A goal of using an object-oriented approach is the distribution of function in the system. Controllers act against this trend.
- Avoid multiple inheritance (inheriting from more than one superclass) unless following this guideline would result in a clumsy model.
- Eliminate unconnected objects from the object model.
- Decompose objects to the most primitive components that have meaning to the user.
- Keep any inheritance trees as shallow as possible to reduce the impact of any changes in superclass methods on lower-level subclasses. Most designs can be captured in three or less levels.
- When determining operation ownership, the service should be associated with the provider (server), not the requester.
- It is generally better to have many simple objects than a few complex ones. An overly complex object with too many attributes may be a sign that the object can be split into smaller objects. In other words, ensure that each class represents only one abstraction. One sign that this rule is being violated is an inability in find a good name for the class. Another danger sign exists if a glossary entry cannot simply be phrased in the form "An instance of this class is a ...." Avoid glossary entries that simply list attributes or services: concentrate on the entire abstraction.
- Don't worry about efficiency or minimization of classes in the analysis object model.
- Associations at analysis time are bidirectional, as it is too early to decide which of the two objects will have the responsibility to keep the information about the other or neither object may actually know about the other. During design we may decide to invent a third object that will keep this information (relating to the link between two objects).

Associations should be labeled. Some people insist on labeling both ends of associations, others are comfortable with only one end being labeled (since the inverse associate can be derived).

- Watch out for processor and data objects

  Data objects superficially have behavior but only access functions. Focusing on the abstract responsibilities of a class instead of its concrete services and attributes helps to clarify whether an object is there only to encapsulate data.

- When in doubt, use association instead of aggregation; it is more general.
- When a class is identified, record a short definition of it in some form of Glossary. With the help of a tool the object model and the class descriptions could probably be different representations of the same information.
- If the model is large (more than what fits in a diagram), consider breaking it up into Subject Areas

Verification

- Check that class names are appropriate. Names should convey intent. Be suspicious of names such as *Controller* and *Manager* as these often indicate a centralization of control.
- Check each class against the criteria that a class should have identity, state, and behavior.
- Check necessity and consistency of cycles of associations.
- Check for symmetry of associations, for instance, that related associations are at related levels in the inheritance hierarchies.
- Check that all aggregations imply lifetime encapsulation.
- Check for an absence of overlapping aggregations such as aggregations with the same components might be owned by multiple aggregates at the same time.
- Check the correctness of all unitary cardinalities.
- Check for absence of design artifacts and bias.
- Check that inheritance always implies specialization.

Example(s)

The following requirements describe a library system containing accounts of those users who want to access library documents. A document can be contained either directly in a library, or in a folder. A folder can be contained inside another folder or inside a library. Each account has an associated capability. When a user wants to access a document or a folder, her account's capability is checked against the threshold required by the document or folder. A user can logon the library, if the user has an account. The user with the right capability can open, delete, and copy a folder or a document. A document can be edited also by the user.

Based on the system requirements and using the approach described in the technique section of this work product description, the project team has made the following observations:

- Class *User* has an *m:1* association with class *Library*, called *Logon*;
- Class *User* has an *1:m* association with class *Account*, called *Own*;
- Class *User* has an *m:n* association with class *LibraryItem*, called *Access*, and the association class is *Library*;
- Class *User* is not within the system to be developed;
- Class *Library* has an *1:m* aggregation relation with class *Account*; It also serves as an association class between *User* and *LibraryItem*;
- Classes *Folder* and *Document* have common behaviors which can be generalized into a class called *LibraryItem*;

  1. Their objects have the behavior of checking capability through the *Security* objects.
  2. Their objects have the behavior of being contained by a *Library* object, or a *Folder* object directly.
  3. Their objects also have the behavior of being opened, deleted, and copied by a user object.

- Class *Library* has an *1:m* aggregation relation with class *LibraryItem*;
- Class *Account* has an aggregation relation with class *Capability*;
- Class *LibraryItem* has aggregation relations with class *Security*;
- Class *Security* has an association with class *Capability*, called *VerifiedBy*, and the association has an association class *Threshold*.

The object diagram at the analysis level for this example, rendered as a Class Diagram, is shown in Figure 11-7.

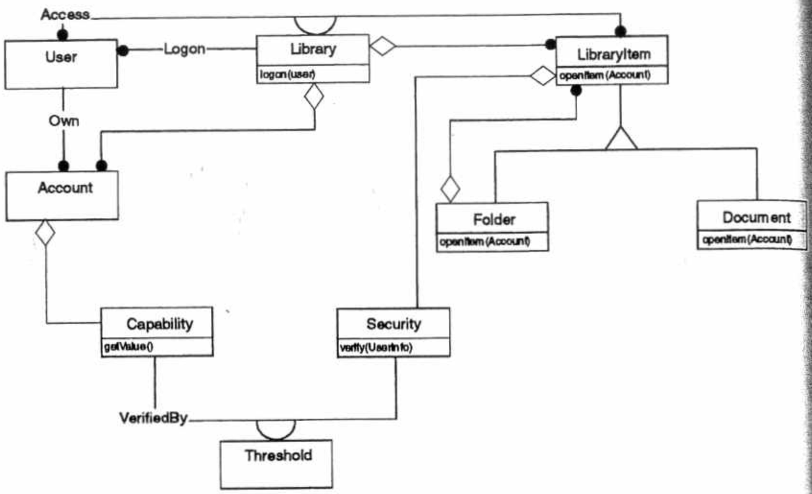


Figure 11-7. *Class Diagram Representation of the Object Model at the Analysis Level.*

References

How to build an object model is well documented in *Object-Oriented Modeling and Design* [Rumbaugh91a] and in various papers by James Rumbaugh.

Importance

An Analysis Object Model is absolutely essential. Decomposing the system into classes of objects that encapsulate data and functionality allows the use of the same concepts throughout the development cycle.

11.4 ANALYSIS SCENARIOS

Description

A Scenario is an elaboration of a Use Case. Use Cases are statements of high-level functional requirements; Scenarios add more detail and describe factors that may result in behavioral variations of a given Use Case.

A Scenario can be defined as follows:

Scenario = Use Case + Assumptions (initial conditions) + Outcomes

A Scenario describes the behavior of the system in a particular situation. The Use Case Model and the set of all Scenarios together constitute the functional requirements of a system. These requirements may be stated as formally or informally as considered appropriate.

Purpose

Use Cases are statements of user needs; however, they are not sufficiently detailed to enable the development of analysis models. Scenarios are refinements of Use Cases and are used to develop Object Interaction Diagrams. A single Use Case can generate multiple Scenarios, and Scenarios derived from the same Use Case can involve the interplay of different classes.

It is very effective to define the requirements of each cycle of an iterative and incremental development schedule in terms of the Scenarios that must be implemented in that cycle.

Participants

An analysis team, led by a qualified analyst who is knowledgeable in object technology should create the Scenarios. It is essential that domain experts or people familiar with the domain participate.

Timing

Start in analysis, after some Use Cases have been identified, and continue throughout the analysis phase.

Technique

Scenarios can be generated directly from Use Cases. They are constructed by taking a Use Case and identifying possible different outcomes (for example, loan granted vs. loan rejected) and different conditions that might result in different kinds of collaborations (for example, the loan requiring a cosigner that in turn results in different kinds of interactions with different classes). At times, this is not as simple as it sounds—sometimes it is hard to imagine different outcomes or assumptions. There will be times when you will have to watch for these happenings while building Object Interaction Diagrams.

There are other sources of information to augment the Use Cases:

- The brains of people with domain knowledge

  A brainstorming session is a useful means of doing this

- Problem Statement
- Reviewing or walking through case studies
- Functional requirements (if a separate functional requirements document exists)
- Variations of other Scenarios

  It is possible to generate new Scenarios by varying the assumptions and outcomes

- Working out Object Interaction Diagrams.

Scenarios are generated by considering each Use Case in turn. For each Use Case the possible behavioral variations of the Use Case are considered. Each variation is documented as a separate Scenario. The behavior of the Scenario is captured by describing the assumptions that the Scenario makes, that is, the initial conditions that must be true, and the outcomes (results) of the Scenario. No information on *how* the Scenario is to be performed is provided, only the conditions before and after the Scenario. The conditions may be informal, textual statements, or they may be formal preconditions and postconditions of the Scenario stated in terms of the states and attribute values of the participating objects.

To generate Scenarios while building Object Interaction Diagrams, watch for questions that will determine what the next request will be or where it should be directed (for example, when processing a loan application, it may make a difference if the customer is known to the financial institution or is a new client). When making an assumption to process an OID, make sure it is explicit and added to the corresponding Scenario. Once this is done, it is easy to generate variations by altering the assumptions (change an existing customer to a new customer, change good credit rating to bad, et cetera).

Like all work products, analysis Scenarios are subject to iterative rework. For example, if state modeling discovers new states of a class, then the language of Scenario assumptions and outcomes has effectively been enriched. It may then be possible to restate the assumptions and outcomes of Scenarios more precisely, or it might be possible to identify new Scenarios.

Strengths

One cannot overstate the importance of Scenarios. They are vital to identifying ways in which the system must respond to real-world situations, or initiate activity in the real world. By identifying assumptions and outcomes we are better able to get a handle on variations that may occur, which may in turn drive out different responsibilities and participants.

Weaknesses

It can be difficult to identify underlying assumptions, as they are often implicit in the situation. One needs to be quite rigorous in searching for assumptions.

Notation

Scenarios are recorded textually; see the examples for a suggested format. The key Scenario attributes to record are its assumptions and outcomes. If doing so would help, then lists of participating objects and parameters may also be added as Scenario attributes.

Traceability

This work product has the following traceability:

Impacted by:

- Use Case Model (p. 96)
- Issues (p. 176)
- Analysis Guidelines (p. 183)

Impacts:

- Analysis OIDs (p. 208)
- Analysis State Models (p. 219)
- Design Scenarios (p. 293)
- Glossary (p. 355)

Advice and Guidance

1. Keep in a list and assign each one a permanent number (even if it is retired or abandoned).
2. Assign a Scenario the same number as its corresponding Use Case.

   As one Use Case may generate many Scenarios, it is useful to extend the Scenario number scheme. Thus Use Case #7 corresponds to Scenario #7.x (that is to say 7.1, 7.2, ...).

3. Watch out for implicit assumptions: Try to make all assumptions explicit.

   This makes it easier to vary Scenarios and identify potential situations that may involve different participants. For example, when a customer applies for a loan (in a banking domain) it makes a difference whether or not they are "known to the bank" (for example, an existing customer). If a customer is new to the bank and applying for a loan, a quantity of background information will be requested. On the other hand, if an existing customer, this step will be bypassed; however, the service person will likely get current account information (for example, existing loan and credit card balances, previous loan history, et cetera).

4. New Scenarios are often developed by finding variations of old ones. When this happens, make sure that the discriminating assumption is added to the original Scenario.
5. Present a Scenario in terms of generic parameters and participating objects, that is, in terms of formal parameters. When the formal parameters are not obvious, document

them explicitly. A Scenario should not refer to specific data values or objects unless doing so is a necessary part of the Scenario. Scenario attributes that document the participants and parameters of the scenarios are particularly useful if there are many participants and/or parameters.

6. If multiple instances of the same class participate in a Scenario, then role names should be given to each. Both role names and class names can be defined in the participant's Scenario attribute, if these are used. If only one instance of a class participates then no role name is necessary, unless it would be helpful to indicate it.

7. If a Scenario participant's attribute is used then only include in it those objects that are mentioned in the assumptions and outcomes lists, not any additional objects that might appear on the object interaction diagram for the Scenario.

8. If doing so would be helpful, use the names of possible states of objects to express the assumptions and outcomes.

9. If a formal style of Scenario presentation is used, Scenario assumptions and outcomes will "feel" more like preconditions and postconditions, and can be labeled as such.

10. If preconditions and postconditions refer to object attributes, they will need to distinguish between the values before the Scenario is performed, and the values after. A useful convention is that all references to after values use the attribute name decorated with a prime symbol. For example a postcondition stating that the new account balance minus the old account balance equals AmountToBeTransferred (a Scenario parameter perhaps) might be written:

```
Account.balance' - Account.balance = AmountToBeTransferred
```

Verification

- Check that each scenario description is as generic as possible. Scenarios must refer to instances, but the description of each instance should be as generic as possible, and not unnecessarily referring to specific instances such as account_512678.

  Instance data is sometimes useful as a means of communicating with users during analysis sessions (for example, Mary the Clerk rather than Clerk).

- To ensure coverage of scenarios, vary all of the assumptions that have been identified and try different combinations and permutations of assumptions and outcomes.

Example(s)

The following are some Scenarios from a banking application that are derived from the Use Case of "customer applies for loan":

```
Use Case 1: Customer applies for loan

Scenario 1.1: Customer Applies for Loan (Granted)

Assumptions:
 • Customer is known to bank (an existing customer)
 • Applied for amount is within Loans Officer's lending authority
 • Customer is employed
 • Customer has a good credit rating
Outcomes:
 • Loan is granted to customer

Scenario 1.2: Customer Applies for Loan (Declined)

Assumptions:
 • Customer is unknown to bank (not an existing customer)
 • Applied for amount is within Loans Officer's lending authority
 • Customer is employed
 • Customer has a bad credit rating
Outcomes:
 • Loan application is declined

Scenario 1.3: Customer Applies for Loan (Marginal Case - approved)

Assumptions:
 • Customer is known to bank (an existing customer)
 • Applied-for amount is within Loans Officer's lending authority
 • Customer is employed
 • Customer has a marginal credit rating
 • Bank requires marginal applications to supply a cosigner
 • Customer provides cosigner
Outcomes:
 • Loan is granted to customer
 • Cosigner is bound by loan contract
```

Figure 11-8. Analysis Scenario for Customer Loan Application.

The following is a scenario from a DB performance monitor project:

```
Use Case 1: Monitor Database

Scenario 1.1: Data Collection–First Sample (successful)

Assumptions:
 • User has defined expression (page activity = page read + page write)
 • All objects have been created
 • Only shows the capture of the first sample
 • Only shows collection for 1 database (there could be many)
Outcome:
 • Successful
```

Figure 11-9. Analysis Scenario for Monitor Database.

References

- [Jacobson92] has extensive discussion on Use Cases.
- [Spivey88] defines behavior in terms of assumptions and outcomes.

Importance

Absolutely essential. Scenarios allow the refinement of Use Cases necessary for building a complete analysis model.

11.5 ANALYSIS OBJECT INTERACTION DIAGRAMS

Description

An Analysis Object Interaction Diagram (OID) is a graphical representation of an Analysis Scenario, expressed in terms of the interactions between real-world or analysis objects. An Analysis OID presents the dynamics of an Analysis Scenario by showing how the objects that participate in the Scenario collaborate in order to achieve its desired outcomes. Bearing in mind that Analysis Scenarios are derived directly from Use Cases, Analysis OIDs complete the link between the requirements and the Analysis Object model. Although Analysis OIDs present the dynamics behind a Scenario, they stay at the analysis level of abstraction. The key for developing effective Analysis OIDs is to focus on the real-world objects only for understanding and abstracting the problem and business, instead of defining solutions.

Purpose

Analysis OIDs provide a high-level view of how objects or instances of those classes defined in the Analysis Object Model interact in order to carry out the Scenarios that are the requirements in the system. The graphical medium shows the end-to-end execution flows in a simple and sufficient way. Analysis OIDs are used to discover responsibilities that are needed to carry out Scenarios, and to assign those responsibilities to classes. As Analysis OIDs link Scenarios to the Analysis Object Model, they may be used either to help derive the Object Model or to validate an existing Object Model.

The expressiveness of Analysis OIDs in showing the dynamics of real-world objects from the user's perspective makes this work product essential to the object-oriented software development, especially in a scenario-driven process.

Participants

Developing Analysis OIDs is the task of a development team led by an analyst. The team consists of analysts, designers, domain experts, and developers. It is very important for clients and domain experts to participate in this modeling activity. It is vital that the Scenarios and their Analysis OIDs that refine those Scenarios should represent their views, business, and requirements. With their participation, the problem will be better understood, and fewer modeling errors will be made.

Timing

Analysis OIDs should begin to be developed early in the analysis phase. As soon as a primitive Analysis Object Model is ready, relevant Analysis OIDs may be developed to drive the analysis object modeling, such as, enlarging the model with more analysis classes, validating its model, and assigning responsibilities and behaviors to its classes. In an iterative process, it will be performed continuously and incrementally throughout the development to represent the evolving understanding of the problem domain objects.

Technique

An Analysis OID is created for a Scenario by recording how objects in the classes from the Analysis Object Model could cooperate in order to perform the Scenario. The record takes the form of a sequence of messages sent between objects. Writing an Analysis OID forces one to take decisions about which classes are to have what responsibilities, or to validate previous decisions. It is expected that Analysis OID writing and Object Modeling take place jointly and iteratively. Analysis OID's frequently "break" the Object Model that must then be modified.

Writing an Analysis OID with a primitive Analysis Object Model involves the following:

- Deciding which objects need to participate in the Scenario. These objects are inserted into the Analysis OID as named vertical lines. Instance names should suggest the roles that the objects play in the Scenario, for example, "sourceAccount" or "destinationAccount."
- Deciding the class of each of these participating objects. Unless the Analysis OID breaks the Analysis Object Model, which frequently happens, the class will be one that has already been identified in the model. If no appropriate class exists in the model, it must be added. As a result, the Analysis Object Model is enhanced.
- The question "what happens now" is asked repeatedly, preferably of a domain expert, in order to carry out the Scenario. The object behaviors modeled in Analysis OIDs are presented as *messages* and *internal activities*. Both must be explicitly represented in an Analysis OID with regard to their occurring sequences. Use the responsibilities identified for the classes of the participating objects as menus from which to select messages. If no appropriate responsibility exists, or if it is assigned to an inappropriate class, then the Analysis Object Model is broken, and needs to be fixed. For

each message, it must be decided which object sends it, which one is to receive and carry it out, and which parameters are appropriate.

Analysis OID modeling is an iterative process that includes a parallel development of the Analysis Object Model. Whether the Analysis OID modeling drives the Analysis Object Modeling or vice versa will depend on the problem to be modeled, as well as the modeling style. For a behavior-centric problem such as real-time systems, Analysis OID modeling often drives the analysis. On the other hand, for a data-centric problem such as information system applications, Analysis Object Model often plays a more important role in analysis. In either case it is expected that information will flow in both directions as the Analysis OID and Analysis Object Model are developed and made consistent and robust.

Strengths

The strengths of Analysis OIDs can be summarized as dynamics, intuitiveness, and expressiveness. The notation of time lines is intuitively appealing and simple but very powerful in expressing what will happen to objects when the Scenario is in progress. An Analysis OID is a very succinct way of expressing the dynamic and functional aspects of a system through the interactions among objects. It should be noted that the notation of Analysis OIDs is not novel to object-oriented software development, but their application within it is highly effective.

Weaknesses

Each Analysis OID can only describe at most one Scenario of a system. A system can consist of hundreds of Scenarios under different conditions and status; Therefore, the amount of work can be too big to accomplish in a short period of time. One solution proposed in this approach is to model only the key Scenarios of a system or to combine several Scenarios into one.

Another weakness is that using Analysis OIDs might lead developers into premature object design. Sometimes it is better to determine object interaction sequences only at design time. Focus on problem domain objects, but even that rule will not always guarantee that an Analysis OID overspecifies a Scenario into the design level. The only real solution is to use your judgment to avoid design decisions, and reflect your clients' point of view, instead of developers'.

Notation

The notation for the Analysis OIDs is mostly straightforward. We will give a complete description of all the concepts, although only a few may be used for any particular Analysis OID.

1. **Object**: Objects or instances involved in the current Scenario are listed. Each object is identified by a name identifying its class and a name indicating its role in the Scenario, in the format: *<object name>:<class name>*. These names are usually placed at the top of the time line for that object. Role names can be thought of as instance names, except that an attempt is made in an Analysis OID to refer to generic rather than specific objects.

2. **Time Line**: Time lines are the vertical lines representing each object. They are used as sources and targets of the time-ordered sequence of messages sent between objects.
3. **Message**: Messages represent object interactions. Every message in an Analysis OID has one sender and one receiver. Sometimes the sender can be omitted if objects from more than one class will send the message. Messages can either request information from an object, or change its state. A message sent to *self*, that is, the current object, is represented by a message whose source and destination is the current object. A message can be either a synchronous or asynchronous one.
   - **Synchronous Message**: The sender of a synchronous message expects a returning one represented by a message line with *return(parameter)*. The sender of the message waits for the return before proceeding.
   - **Asynchronous Message**: The sender of an asynchronous message does not wait for an explicit return message before proceeding. This is the only distinction between synchronous and asynchronous messages.

   Incoming messages to a time line represent sync-points.
4. **Message Parameter**: A message can carry parameters in an interaction.
5. **Condition**: A condition within a pair of square brackets represents a decision point in the message sender object. If the condition is true, the message attached will be sent. In any case the object continues with its time line. Using the condition notation can help make the Analysis OID more general, with weaker assumptions. The alternatives to using conditions (and loops) are to use several Analysis OIDs to present the options for a single Scenario, to split up the Scenario, or to omit certain details. Each of these may be the best choice depending on circumstances.
6. **Loop**: In order to perform repeated actions, a loop can be attached to an object's time line. A loop is also associated with a loop condition, which is the condition under which the loop is performed again.
7. **Internal Activity**: An internal activity is any internal behavior that is relevant to a time line, but whose internal details are either not appropriate to provide, or not yet known. An internal activity is represented by the name of the activity in curly brackets appearing at the appropriate point on a time line. Internal activities might be entirely local to that object, or they may involve communications. The internal activity can be further specified in an attached note. An example of an internal activity might be "*{Identify the right account}*." A description of an internal activity can be provided as a footnote to the Analysis OID.

The notation of an Analysis OID is shown in Figure 11-10. We understand that the notation we use is not completely supported by every CASE tool. Users are encouraged to tailor the notation, based on the CASE tool they use. Most commonly supported concepts are *objects, messages*, and *time lines*.

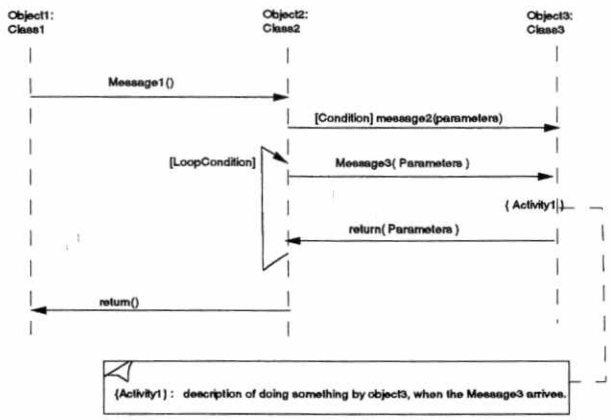


Figure 11-10. Format of Analysis Object Interaction Diagram.

Traceability

This work product has the following traceability:

Impacted by:

- Issues (p. 176)
- Analysis Guidelines (p. 183)
- Analysis Object Model (p. 192)
- Analysis Scenarios (p. 203)
- Analysis State Models (p. 219)

Impacts:

- Subject Areas (p. 187)
- Analysis Object Model (p. 192)
- Analysis State Models (p. 219)
- Analysis Class Descriptions (p. 227)
- Design OIDs (p. 298)

Advice and Guidance

- Make Analysis OIDs consistent with the business logic. Analysis OIDs are used for recording what happens in the real world. Any design and architectures related to the system solution should be kept out of it.
- Keep Analysis OIDs as simple as possible. It can help avoid mixing in any design objects and decisions.

- Focus on object general behaviors, instead of methods. Methods are only meaningful under the design, and should be created at the object-oriented design phase. The messages sent between two objects in an Analysis OID help model the behaviors, and these messages are not the methods for the message receiver (but they could be so in the Design Object Interaction Diagram discussed later in this document).
- Make Analysis OIDs consistent with the Analysis Object Model. If object A sends a message to object B, A's class should have a certain relationship with B's class in the Object Model.
- Due to the fact that hundreds of OIDs can be developed, it is important to capture only the major ones that drive out the principal problem features. Do not explore every exceptional condition unless it is significant to the users.
- Avoid Analysis OIDs overspecifying their Scenarios. One common mistake is for analysts to descend into unnecessary detail. If this happens, design objects and decisions creep into the analysis model, resulting in clutter and overspecification. Avoid this by focusing on using the problem domain objects in the Analysis OIDs, instead of implementing the Scenarios. Internal activities in the OIDs can be used to defer low-level object interactions that lead to possible overspecification.
- An important step ignored by many developers is the detailing of assumptions before developing an Analysis OID for a Scenario. Check hidden assumption and outcomes.
- Avoid both overly passive objects (pure data objects), and overly active objects (managing and controlling objects). Distribute the responsibilities throughout all objects.
- Identification and dating are very important. In the case of Analysis OIDs it makes sense to assign to them exactly the same identification as their corresponding Scenarios (as they have a 1:1 relationship).
- The object role names and parameters, or formal arguments, in a Scenario should be consistent with its Analysis OID.
- Write an Analysis OID in terms of specific objects such as "*anAccount*" or "*Account1*," if this promotes clarity. The goal of Analysis OID modeling is not for completeness, but an end-to-end understanding that is missing in the Analysis Object Modeling.
- Make Analysis OIDs general and independent of technology, design, and constraints. There are some situations in which the detail of object interactions is necessarily a design Issue. In such cases leave the Scenario without a corresponding Analysis OID, or leave the problematic interactions as internal activities in the Analysis OID. Postpone the resolution of the design issue to modeling of Design Object Interaction Diagrams.
- Watch out for implicit assumptions and make them explicit.

- Make sure that assumptions are proven in the Analysis OID. For example, if we are processing a customer change of address with the assumption that there is no territory change, we still need to show the object that makes the decision related to territory and its sale representatives in that Analysis OID.

Verification

- A walk-through of Analysis OIDs is an effective technique to check whether the model reflects the reality.
- Check for an even distribution of system intelligence and control.
- Check for an absence of overly passive objects.
- Check that an Analysis OID does not overspecify its Scenario in the sense that it introduces decisions which can better be made at design time.
- Check that Scenario assumptions are necessary and sufficient for each Analysis OID.
- Check that Scenario outcomes indeed occur in each Analysis OID.
- Check for hidden assumptions and outcomes in an Analysis OID.

Example(s)

This example is simplified from a real project. It deals with users accessing the items in a library system. The example presents two scenarios as well as a modification to the second scenario.

A user has an account associated with the library system. The account has security levels associated with each library item. Each library item's security object will check whether the user is permitted to access that library item.

The first scenario shows the business logic for the *access* process for the library file system.

This example is also explored at the design level in Section 13.8, Design Object Interaction Diagrams.

- **Analysis Scenario 1**: A user wants to access a library item.
- **Assumption**: The user has already logged into the library, and a unique identity, *current user*, is established.
- **Outcome**: A library item is permitted to be viewed, if the user passes the security check for that particular library item.
- **Description**: When a library item receives a request for access from a user it first passes the user's identity (account) to its own security object to check whether the user is permitted access. If the user has the clearance to access this library item, the item will display its own content. Otherwise, the request is rejected.
- **Analysis OID**: The Analysis OID for this scenario is shown in Figure 11-11

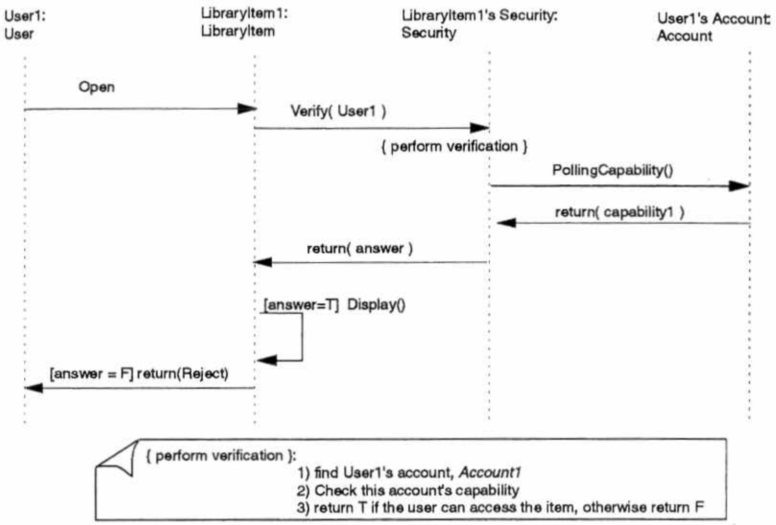


Figure 11-11. Analysis OID for User Access to Library Item.

Our understanding of requirements can change over time. Analysis OIDs can facilitate this process. Consider the following example of the Use Case "Customer Changes Address" in a marketing company. The Scenario being addressed is:

- **Analysis Scenario 2**: Customer Changes Address
- **Assumptions**:
  - Caller is an existing customer
  - Address change is immediate (not in the future)
- **Outcome**: Customer's address successfully changed
- **Description**: A customer's address change usually involves an update of the customer's file and notification to the customer's agent.

This could be a situation of a Help Desk (Clerk) for an Insurance Company receiving a phone call from a customer who wishes to notify the company that he or she has moved.

Figure in Figure 11-12 shows a possible Analysis OID for this Scenario.

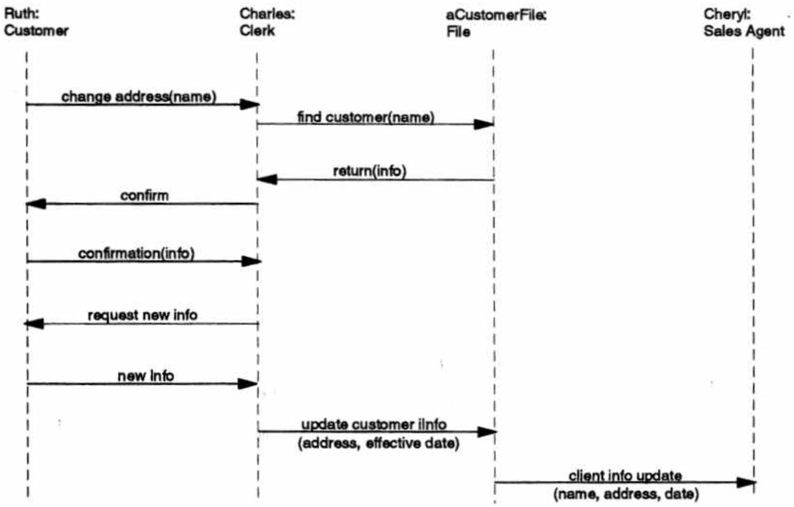


Figure 11-12. Initial Analysis OID for Change of Address.

The basic Scenario is that a customer calls the Help Desk, to notify the company of an address change. The customer's file is found in the customer filing cabinet, its identification is confirmed and the sales agent is notified. This OID satisfies the requirement - at least at first glance. However, later during analysis, we learn that agents have territories, and we must revisit this OID.

With this new information, we can modify the Scenario as follows:

- **Analysis Scenario 2 (modified)** : Customer Changes Address
- **Assumptions**:
  - Caller is an existing customer
  - Address change is immediate (not in the future)
  - Move involves change of territory and new agent must be notified
- **Outcome**: Customer's address successfully changed
- **Description**: A customer's address change usually involves an update of the customer's file and notification to the customer's agent.

We are faced with a problem - who (which object) knows about sales agents' territories. Certainly it would be outside the responsibility of the customer's file. This is an important question as we may be discovering the need for a new object. Figure 11-13 shows a possible solution to the modified Scenario. It only shows the portion of the OID which would be modified.

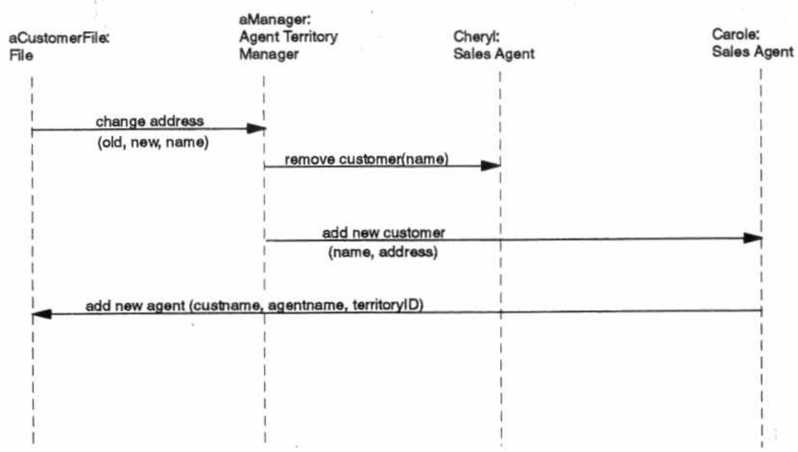

aCustomerFile: File
aManager: Agent Territory Manager
Cheryl: Sales Agent
Carole: Sales Agent
change address (old, new, name)
remove customer(name)
add new customer (name, address)
add new agent (custname, agentname, territoryID)

Figure 11-13. Modification to Part of the Analysis OID for Change of Address.

In the modified OID, the customer's file delegates the task of territory management to the Agent Territory Manager object. This object knows about and understands the company's approach to territory management (it can get quite complex). This object will apply current business policies and determine who needs to be informed of the address change (in this case the old and new Sales Agents). If the address change had not involved a change of territory, it would simply have notified the existing agent of the address change and then sent a no change notice back to the customer's file. Otherwise, the customer will be removed from his/her current agent, and forwarded to another agent who is in the territory that the customer moves to. The customer's file is also updated for the change.

This example illustrates a number of points:

1. Analysis OIDs assist with the discovery process and understanding a business area;
2. Always present the business knowledge and logic in Analysis OIDs;
3. Delegation of responsibilities can change with our understanding of the business area;
4. We must always be prepared to learn new information.

References

- Jacobson *et al.* [Jacobson92] were the first group employing Use Cases and OIDs in dynamic modeling.
- OMT [Rumbaugh91a] has its own name for OIDs, that is, "event trace diagrams."
- [Booch94] also has interaction diagram.
- The Unified Modeling Language [Booch96] also utilizes the OIDs under the name "Sequence Diagram" for dynamic modeling.

Importance

Essential. Analysis OIDs are a powerful tool for showing the dynamics of the real-world objects being modeled.

11.6 ANALYSIS STATE MODELS

Description

A state model, as used in object-oriented analysis, describes the life cycle of a class. It describes states that a class may attain and transitions that cause a change of state. The state transitions, representing external stimuli or events, show an object's state changes. A state model is represented by a *state diagram* or a state transition table.

Purpose

A state model represents an object's life cycle in graphical notation or in a tabular form. It gives an overview of how an object reacts to external events without getting into code details. A state model is much easier to develop and understand in comparison with high-level textual descriptions. It is useful because of the insight it can yield about the nature of a given class.

Participants

At the analysis level, state models should be developed by the analysis team. Customers should participate in this activity, so that the modeling can be as accurate as possible based on clients' requirements and knowledge of objects in their domain.

Timing

Analysis State Modeling is done after an initial Object Model and a dynamic model with scenarios and Object Interaction Diagrams (OIDs) have been completed. It is performed for those classes whose OIDs show that they have significant dynamics. This is indicated by incoming messages whose arrival order is vital to the class.

Technique

- Select a class to model.

  During analysis, we look for classes with an interesting or unclear interesting life cycle (for example in a lending application the Loan class is quite interesting).

- Identify how the object comes into being. This will be a state transition leading to an initial state.
- From the initial state, add all transitions that can occur and the states that they lead to.
- Repeat this process for all identified states until complete.
- Note that it is valid to include transitions that lead back to the same state (i.e. loops).

  For example, a Loan is in an active state, a customer makes a payment, is shown as a transition leading back to the active state.

- For completeness, it is advisable to probe whether or not there are transitions leading between various states in the model.

Strengths

A state model provides a complete picture of the life cycle of a class. It is another way of looking at an object, and object behaviors can be clearly presented through such a model.

Weaknesses

The weakness of state models is that one model can only show information limited to one class. It can be difficult to show simultaneous changes of state that occur between multiple, collaborating classes.

Notation

The graphical notation of a state diagram representing the state model is shown in Figure 11-14.

The key elements of the notation are:

- A class's states are shown in circles (or boxes).
- Transitions are shown as directed arcs between states.
- Transitions should be labeled as doer-action tuples (for example, customer makes loan payment).

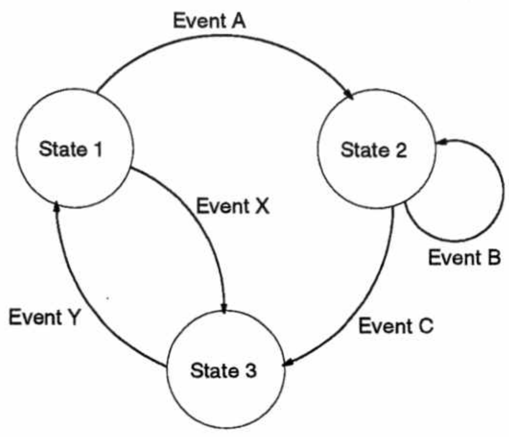


Figure 11-14. State Diagram Notation.

Traceability

This work product has the following traceability:

Impacted by:
- Issues (p. 176)
- Analysis Guidelines (p. 183)
- Analysis Object Model (p. 192)
- Analysis Scenarios (p. 203)
- Analysis OIDs (p. 208)

Impacts:
- Analysis OIDs (p. 208)
- Analysis Class Descriptions (p. 227)
- Design State Models (p. 306)
- Glossary (p. 355)

Advice and Guidance

- One of the best ways of building a state diagram is to go through a role playing exercise. This can be quite effective. Pretend that you are the object being examined, assume a state and ask "what can happen to me now?"
- Examine the state chart and ponder whether it is possible to get from one state to any other state. This sometimes uncovers behaviors or actions that otherwise might not be anticipated.
- In analysis, it is important to avoid representing design decisions in the state models; we observed that it is quite easy to fall into this trap.

- Watch that you don't wind up building flowcharts. The best way to avoid this is to ensure you keep a consistent point of view. If you are the "Loan," then you only concern yourself with the "Loan's" point of view.
- When working with "naive" users (i.e., noncomputer people) do not use terms such as finite state machine or state transition diagram. Just say you are going to build a picture of the life cycle of the object, how it gets created, what things can happen to it, what causes it to go away, and so forth. Say that the reason for doing this is in order to achieve a deeper understanding of the domain.
- State diagrams are built on a very selective basis during analysis. In a typical domain, maybe one or two per 100 classes. This will increase dramatically in domains that have a real-time aspect about them (like manufacturing, process control, monitoring, et cetera).

Verification

- At every state close your eyes and ask "what can happen to me now?" and ensure that all state transitions (and their respective states) are represented.
- From every state ask if it is possible to visit every other state in the model (for example, "can I possibly get from here to there?" where "there" is every other state in the model).
- If two states exhibit the same behavior, then collapse them into one provided they exhibit the same behavior (i.e., have same entry and exit conditions).

Example(s)

Figure 11-15 presents a possible state diagram for an Account class for a bank application with our basic state model notation. The life cycle for an *Account* object from creation through being closed is demonstrated in the diagram.

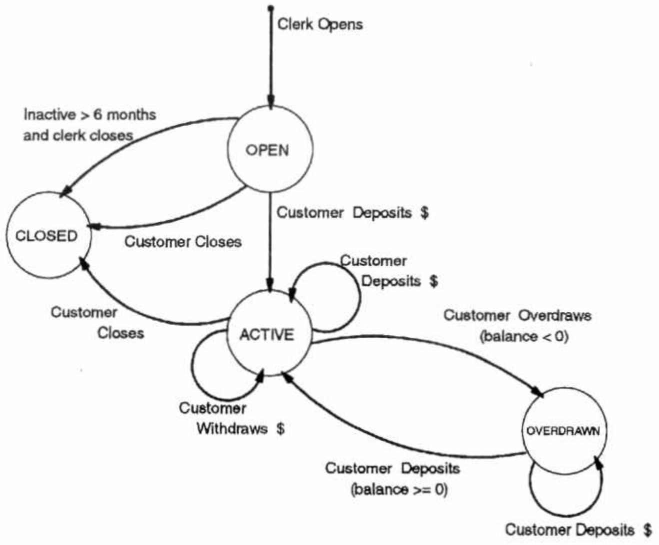


Figure 11-15. *State Diagram for the Account class.*

Figure 11-16 shows another example of a state diagram, this time for a loan class. This was built with the help of a Loans Officer from a bank. If this process is presented to users as being an inquiry about the life cycle of a loan object, users have little difficulty assisting with producing such a diagram (i.e. it is best to avoid describing the process as finite state machines or state modeling when speaking to users).

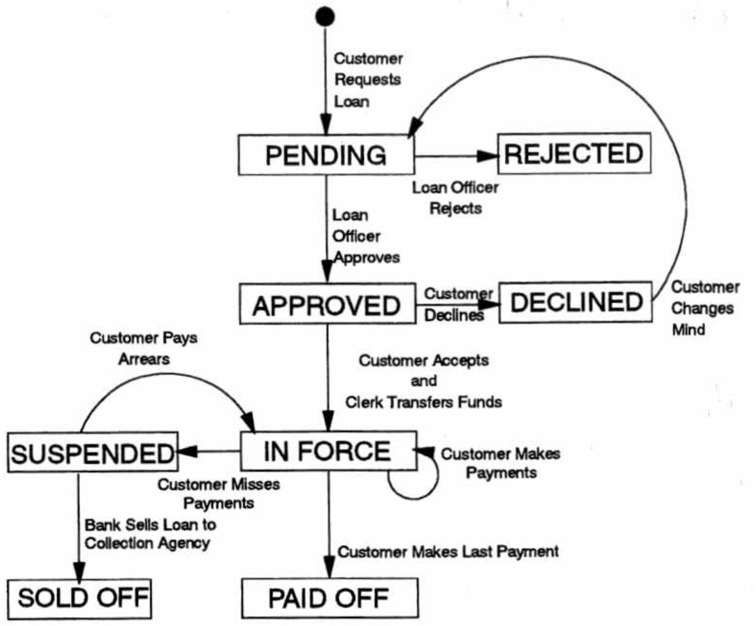


Figure 11-16. State Diagram for a Loan.

State Transition Tables: State Transition Tables are useful when State Transition Diagrams (or Statecharts) are not feasible; for example, they must convey state model in an ASCII medium, say e-mail; drawing tools are missing, limited, or not standardized in the development (or documentation) environment.

There are at least two types of state transition table formats that show the same information in different ways.

State Transition Matrix (STM): Has one row per state and one column per event. The content of the cell is the Next state. (See Table 11-1)

State Transition Table (STT): Has three columns labeled *Current State*, *Event*, and *Next State*. (See Table 11-2)

STMs work well when there are few states and events but have a rich set of possible transitions. They force one to address completeness and consistency.

Table 11-1. *State Transition Matrix for Account Class.*

Event-> State	Clerk Closes	Customer Closes	Customer Deposits [bal+amt >=0]	Customer Deposits [bal+amt <0]	Customer Withdraws [bal-amt >=0]	Customer Withdraws [bal-amt <0]
Open	Closed	Closed	Active	(can't happen)	(can't happen)	(can't happen)
Active	Closed	Closed	Active	(can't happen)	Active	Overdrawn
Overdrawn	(can't happen)	(can't happen)	Active	Overdrawn	(can't happen)	(can't happen)
Closed	(can't happen)	(can't happen)	(can't happen)	(can't happen)	(can't happen)	(can't happen)

STTs work well when there are many states and/or events but sparse transitions; that is, when very few states are sensitive to each event. These are less cluttered and easier to create, but hide incompleteness.

Table 11-2. *State Transition Table for Account Class.*

State	Event	Next State
Open	Clerk Closes	Closed
Open	Customer Closes	Closed
Open	Customer Deposits	Active
Active	Customer Closes	Closed
Active	Customer Deposits	Active
Active	Customer Withdraws [bal-amt>=0]	Active
Active	Customer Withdraws [bal-amt<0]	Overdrawn
Overdrawn	Customer Deposits [bal+amt<0]	Overdrawn
Overdrawn	Customer Deposits [bal+amt>=0]	Active

Using the full Statechart notation [Harel87], events can be elaborated with attributes, conditions, and actions. States can also have entry and exit actions as well as internal activities.

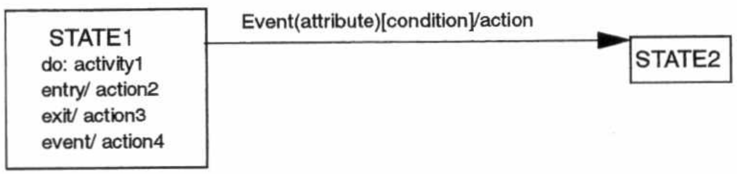


Figure 11-17. *Extended Notation for State Diagrams.*

STMs and STTs can be annotated (extended) with adornments from the underlying modeling technique. For example, the entry/exit actions above can be associated with the Event entry for the transition. Table 11-3 is an elaboration of Table 11-2 that shows event attributes and actions.

Table 11-3. *Elaborated State Transition Table for Account Class.*

State	**Event(attribute)[condition]/action**	**Next State**
Open	[6 months inactive]/send notice	Closed
Open	Customer Closes	Closed
Open	Customer Deposits(amt)/bal:=amt	Active
Active	Customer Closes/send check for balance	Closed
Active	Customer Deposits(amt)/bal:=bal+amt	Active
Active	Customer Withdraws(amt) [bal-amt>=0]/ bal:=bal-amt	Active
Active	Customer Withdraws(amt) [bal-amt<0]/ bal:=bal-amt	Overdrawn
Overdrawn	Customer Deposits(amt) [bal+amt<0]/ bal=bal+amt	Overdrawn
Overdrawn	Customer Deposits(amt) [bal+amt>=0]/ bal=bal+amt	Active

References

- Rumbaugh et al [Rumbaugh91a], and Booch [Booch94] recommend using state diagrams to model the dynamic behaviors of objects.
- Statechart, a diagramming notation for states, was developed by Harel in 1987 [Harel87].

Importance

Optional, although as a rule it is extremely useful to use in business domains such as banking where *Loan* would be very interesting or in insurance where an *InsurancePolicy* can be very interesting.

In real-time domains this becomes essential as there will be objects with strong state-dependent behaviors, such as the *Connection* class in a telephony application.

11.7 ANALYSIS CLASS DESCRIPTIONS

Description

Analysis Class Descriptions are summaries of all the information known about a class at the analysis level. They are similar to the CRC cards of Wirfs-Brock et al [Wirfs-Brock89]. Class-related information exists in several places in the analysis chapter of the project workbook, for example in different parts of an object model, in scenarios, and in state models. For each class, a class description provides a concentrated summary.

Class descriptions are not intended to overlap or conflict with object or dynamic models. Good tool support should enable class descriptions to be updated automatically whenever other model views are modified, for example to add a new responsibility to a class in an object model view. Much of the information of a class description should be generated automatically from the data of the various development models. Similarly, Class Descriptions and Glossary entries (see Section 16.1) may in practice be generated from the same source data.

Purpose

Analysis Class Descriptions are provided for two reasons.

1. To provide a place to put class-specific information, such as a short description, key attributes, responsibilities, and the like, which might otherwise slip down cracks between the other analysis work products.
2. To provide a single point of contact for analysis information regarding a particular class.
3. To provide a place to record information that won't fit on other diagrams (Object Model, OIDs, et cetera). This may take the form of pictures, descriptions, standards, related documents, and the like.

Participants

The analyst who owns a class has the responsibility to maintain the Analysis Class Description for that class. Tools that automatically derive the Class Description information from other work products (for example, the Object Model and the State Model) significantly reduce the effort needed to maintain Class Descriptions.

Timing

Class descriptions are provided at analysis, design, and implementation levels. (Implementation class descriptions are used only if they are needed as a repository for new information about the class.) Thus a particular class may have several class descriptions, one for each level of abstraction in which it is involved: analysis, design, and implementation. These multiple class descriptions may refer to each other, but they are different work products. The reason for this apparent redundancy is that the definition of a class may be subtly different at each level of abstraction.

The Analysis Class Description serves as a repository for summary information as the analysis modeling activity proceeds, and is completed in time to be reviewed along with the remainder of the analysis work products. A class description is opened as soon as there is a need to summarize the properties of a class.

Technique

The creating or updating of class descriptions should be one of the activities performed after each analysis modeling session.

Exactly how the class description is filled in will depend on its format. A class description will (obviously) have a name and it should have a short textual definition. Names are very important and should be chosen to reflect the nature and intent of the class. A vague or ambiguous name is often a sign that the abstraction named is insufficiently understood or inappropriate. The abstraction might be inappropriate because it refers to more than one concept, in which case it is a candidate for splitting into multiple classes (each with more precise names). Another problem that often surfaces with names is that an abstraction is actually a function and not an object. Class names that are verbs like *Connect* or *Initiate* might indicate this. Names such as *Controller* or *Manager* are often hints that the system is too centralized; its intelligence has not been distributed in an appropriate manner. This is a common mistake among experienced developers who are new to object technology. Names should, therefore, be taken seriously.

Even if a modeling team agrees on a name, it frequently happens that they find out later that they do not agree on what the class actually represents. This problem can be solved by insisting that modeling sessions that propose new (or modified) classes also agree on a short textual definition. The shorter the definition the better; one line is perfect although sometimes inadequate. The definition cannot and should not define all aspects of the class. It should, however, capture enough of the meaning of the class to ensure that all developers are thinking along the same lines, and that newcomers to the project can gain an immediate understanding of why the class exists in the model. Agreeing on definitions is often difficult, but if it is not done then the outstanding inconsistencies will dig themselves into the model, requiring considerable excavation work subsequently.

Names and textual class definitions are most conveniently captured initially as Glossary entries, and then incorporated into Class descriptions as the need to summarize information about the classes arises.

Strengths

Class descriptions are an important part of all published development methods. If properly used, they can also significantly improve the readability of a workbook by providing cross-references.

Weaknesses

An appropriate format must be chosen. There are many published formats for class descriptions. It is important that a format is fitted around a development method and not vice versa. Tools complicate the issue by providing support for Class Descriptions, but not necessarily the ability to tailor the format.

Notation

Beyond names and definitions, the format of class descriptions should serve the selected analysis, and not the other way around. If a responsibility-driven approach [Wirfs-Brock90] is being used, then class descriptions should resemble CRC cards. If OMT [Rumbaugh91a] is being used, then the class descriptions will concentrate more on attributes and operations. Whatever the method, it is important that the class description format is decided in advance. Although there will be close correspondences between the analysis and design class descriptions of particular classes, they may each have different formats to support the different nature of the modeling work at each level.

While Analysis Class Descriptions are very important, they do not tend to be as structured as those at the design level, for less emphasis is placed on documenting interfaces at the analysis level. The following template is adequate for Analysis Class Descriptions, although it should be modified to suit the development method if required.

Name ______________________________
Definition ______________________________
Operations ______________________________
____
____
Key attributes ______________________________
____
____
Relations ______________________________
____
____
States ______________________________
____
____
Documentation ______________________________
____
____

In the above box, "Name" identifies the class name. Under "Definition," a sentence or two describing the class should be provided. "Documentation" is a place holder for any materials related to the analysis of the class that cannot be expressed in the structured forms of the other work products, such as the Analysis Object Model, the Analysis OIDs, et cetera. An example of this might be an existing customer document describing part of the problem domain.

It is useful to note specific key attributes as these often contribute significantly to an understanding of the class. They are, however, to be understood as *logical* attributes, and not attributes that will necessarily appear in the design. An example of a logical attribute might be "Age." The presence of this attribute in an Analysis Class Description should not be taken to imply that a corresponding attribute will exist in the design. The design might, for example, employ a "dateOfBirth" attribute instead.

Attributes and responsibilities that appear in Class Descriptions should include those inherited from other classes, and they should be marked as such.

Traceability

This work product has the following traceability:

Impacted by:
- Issues (p. 176)
- Analysis Guidelines (p. 183)
- Analysis Object Model (p. 192)
- Analysis OIDs (p. 208)
- Analysis State Models (p. 219)

Impacts:
- Design Class Description (p. 311)

Advice and Guidance

- Do not manually duplicate all class-related information in the class descriptions. For example, do not copy association information manually from the object model into the class descriptions. Good tool support will help to store the analysis work products in a nonredundant fashion.
- Class descriptions should be consistent with the information specified in other analysis work products. Thus, if the Analysis Scenarios have Object Interaction Diagrams (OIDs) which specify messages between objects as operations with parameters and results, then the class descriptions should record these operations, parameters and results. If the OIDs specify only message names, then the class descriptions should record only message names, et cetera.

  There is obviously tension between this piece of advice and the one above. Judgement must be used to decide how best to document a model. It is reasonable to summarize operations from scenarios as the class-centered view will otherwise be missing. It is probably unreasonable to include associations, because the graphic object model will do a better job of that, and no summarizing value-add is provided by the class descriptions.

- When initially opening class descriptions to document modeling sessions, don't worry too much about whether it is correct to include a class. If a class turns out not to be relevant to the model, then it will be isolated in the object model and, more tellingly, it will not participate in any scenarios. The class description can be removed at that point. It is often quicker to define classes that *may* be relevant and to proceed with the modeling, than to worry prematurely about the relevance of classes. However tentatively a class is included, though, it should be well defined.
- Design the class description formats and do not simply accept the format that a particular tool provides.

Verification

- Check that the class descriptions are being maintained.
- Check that the descriptive text accompanying each class description describes the intent of the class, and not its internal structure.
- Check for the completeness of class responsibilities, namely, that there are no missing responsibilities.
- Check attributes and relations for completeness, for relevance to the problem domain, and for relevance to analysis.

Example(s)

The following is an example of a class description from a banking application:

Advice and Guidance

If you have a UI-intensive application or a general public-oriented application, create a UI Prototype to resolve the risk of unintentionally creating a nonintuitive or unfriendly user interface for your application.

- Use a specialized UI team (or human factors team) to do the prototype for you if you can, it will save time.
- Use specialized UI Builder tools to create and exercise the UI Prototype.
- Limit the scope of the prototype to UI issues and concerns (don't overload the prototype with other issues like performance, persistence, distribution, communication, et cetera).
- Develop the UI Prototype in stages (incrementally) and test them with customers before tuning (iterating) or proceeding to the next increment. Don't pester the customer with too many test sessions, but do use the test sessions to develop rapport with customer.
- When performing this activity, don't waste a lot of time responding to minor changes (like colors, placement, and such). Listen to the feedback and get back to work.

Verification

- Not applicable.

Example(s)

Not applicable, since UI Prototype is an executable program.

References

- Good arguments for rapid prototyping are offered in [Connell89]. Although it is not specific to object-oriented technology, many general principles still apply.
- A general discussion of principles and guidelines for Graphical User Interface (GUI) design can be found in [Mayhew92].

Importance

Optional, although very important for UI-intensive applications and applications oriented to the general public (for example: banking, library, event kiosks, exhibitions, lobby facilities).

13.0 Design Work Products

The Design portion of the project workbook details those work products that are created during the design phase of the project.

Object-oriented design is the process of determining the Architecture for and specifying the classes needed to implement a software product. It involves making global and local decisions about a planned implementation based on constraints, Nonfunctional Requirements, and available alternatives.

During analysis we focused on problem-domain objects; however, during design we focus on solution domain objects. During design, the emphasis is on defining a solution [Monarchi92].

The problem domain classes encountered during analysis are refined during design. New objects and classes are created during design as shown in Figure 13-1.

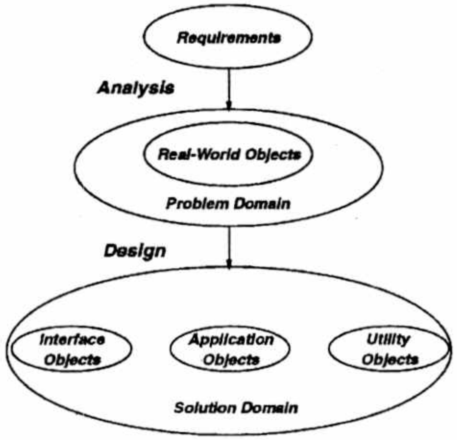


Figure 13-1. The Relationship Between OO Analysis and OO Design.

As part of the design activity, the system is partitioned into subsystems to make the design process more manageable.

In summary:

- Object-oriented design transforms the analysis model into the design of a software solution.
- Definition of the overall software architecture (global decisions) is done during design.
- Attributes, operations, and algorithms are defined for all design objects during design.
- New objects, not domain-derived but implementation-oriented, are defined during design process.

The boundary between design and implementation work products is both flexible and subjective. The goal of design is to achieve sufficient agreement on interface definition and internal structure that implementation work may then proceed independently in parallel teams. The amount of design detail that is needed will depend on many factors including the size of implementation groups relative to the size of the project, the degree of coupling between the team components, and the like. In practice this means that each project must decide for itself exactly where to draw the line. Wherever that is, the following work products are relevant, but projects must choose the degree of detail to include in each. As mentioned above, this decision is project-dependent but is by no means arbitrary. The amount of detail is chosen to enable independent parallel implementation of the design after it has been completed and reviewed.

The design section of the project workbook consists of the following work products:

- Design Guidelines
- System Architecture
- Application Programming Interfaces (APIs)
- Target Environment
- Subsystem Model
- Design Object Model
- Design Scenarios
- Design Object Interaction Diagrams
- Design State Models
- Design Class Descriptions
- Rejected Design Alternatives

These work products all "inherit" the common work product attributes described on Section 8.1, and they have specialized attributes of their own.

Design Guidelines are a set of rules that help in defining the design deliverables and guiding the design process and as such have an impact on most of the other design work products.

System Architecture is the set of broad design principles that allows the system design to be coherent and consistent. It, too, has an impact on most of the other design work products.

The Application Programming Interfaces (APIs) are a set of visible classes and their interfaces that enable the system to be used without the need to understand its internal details.

The Target Environment defines the environment(s) in which the system is intended to operate. This is typically part of the Nonfunctional Requirements or at the least is impacted by that work product. This can have a big impact on the System Architecture.

The Subsystem Model partitions a system into smaller entities and delegates certain system responsibility to those entities. The main benefit of this is the breaking up of large, complex systems into more manageable entities. Each Subsystem will have its own set of design work products such as Design Object Model, Design Scenarios, and Design Object Interaction Diagrams.

The Design Object Model is a static model that represents the structure of the classes and their relationships with each other in the implementation of the system.

Design Scenarios enumerate the possible assumptions and resultant outcomes (starting and ending states) for each intended behavior of the planned system.

Design Object Interaction Diagrams graphically depict the collaborations between objects that is required to support the Design Scenarios.

Design State Models represent the dynamic behavior of design classes and are done for all classes that have strong state-dependent object behavior.

Class Descriptions contain all of the information known about a class at the design level. They provide the starting point for implementation work on a class.

Rejected Design Alternatives provided a repository of those major design directions that were considered but rejected. They are valuable when the need arises to revisit design decisions as often happens in complex system development projects.

The work products and their specialized content are defined and discussed in more detail in the following sections.

13.1 DESIGN GUIDELINES

Description

Design Guidelines are the set of rules intended to define the design deliverables and to guide the design process of a particular development project. In this respect they are similar to the Analysis Guidelines (Section 11.1), but are broader in scope to reflect the greater diversity of the design deliverables. Similarly to Analysis Guidelines, Design Guidelines may be divided into work product guidelines and process guidelines.

Analysis Guidelines must address object, scenario, and state modeling, and the processes to perform these. Design Guidelines must address these too. For the common aspects, the Design Guidelines can probably just refer to their analysis counterparts. Some additional notational concerns that are relevant at design time are those of concurrency, distribution, and association directionality. Notational extensions should be defined to indicate process boundaries and to make distinctions between synchronous and asynchronous messaging on Design OIDs. The particular process for deciding upon process boundaries and message types should be described in the guidelines also.

The other work products that are unique to design, such as System Architecture, Subsystem Model, and Application Programming Interface also need guidelines to assure that their documentation is consistent and useful. For example, the API for a subsystem might be documented as a set of figures, tables, and descriptive text, or as a collection of programming "header files," or as both formats. The Design Guidelines must make it clear what is expected of all the participants. Contracts between subsystems are also useful documentation that the guidelines might insist upon.

A very important part of the design process is the way in which it treats design Issues and their resolution. The Design Guidelines should provide guidance on this matter.

Purpose

Both Analysis and Design Guidelines are important, but for slightly different reasons. Analysis Guidelines are required primarily to agree on notation and to help the team get going, particularly if the team includes novices to object-oriented software development. Design Guidelines are used much more to guide the development process. In analysis, freedom is encouraged because the emphasis is on understanding the requirements, documenting and analyzing them, and verifying them with the customer. The nature of design demands a disciplined approach to process and documentation.

Participants

The Analysis and Design Guidelines will probably be written by the same people, perhaps the team leader and the architect. Considerable experience is required of both software engineering in general and object-oriented software development in particular.

Timing

The Design Guidelines must be completed before design can begin. They must, therefore, be written either during the first analysis cycle, or earlier.

As is the case for Analysis Guidelines, Design Guidelines and team education are interdependent. What is special to object-oriented design (as opposed to structured design) is principally in the realm of process. It is therefore the design process that is concentrated on in the Design Guidelines, and in the educational effort targeted at design. This dependency means that the Design Guidelines might have to be agreed upon even before design education has begun.

Technique

Beg, steal, or borrow some existing design guidelines from existing object-oriented projects. Customize these and review them with your project. Interview experienced developers to check that the guidelines are still reasonable and complete. Consider asking a mentor for advice on Design Guidelines, or to comment on those that have already been assembled.

Strengths

Design Guidelines ensure common style and consistency among the various design work products.

Weaknesses

None.

Notation

Design Guidelines take the form of check lists and/or templates addressing each work product, process, and tool intended to be used by the developers.

Design guidelines are usually organized by work products.

Traceability

This work product has the following traceability:

Impacted by:

- Intended Development Process (p. 127)
- Project Workbook Outline (p. 132)
- Quality Assurance Plan (p. 147)
- Reuse Plan (p. 158)
- Test Plan (p. 164)
- Issues (p. 176)

Impacts:

- Project Workbook Outline (p. 132)
- System Architecture (p. 257)
- APIs (p. 265)
- Target Environment (p. 272)
- Subsystems (p. 274)
- Design Object Model (p. 281)
- Design Scenarios (p. 293)
- Design OIDs (p. 298)
- Design State Models (p. 306)
- Design Class Description (p. 311)

Advice and Guidance

- The guidelines should be minimal though sufficient to ensure that the design work products are understandable, usable, and consistent.
- Use existing guidelines where they are available and appropriate. Modify these only as necessary.
- If your guidelines are novel, publish them for others to evaluate and use.
- Use work product templates where relevant.

- Harvest good templates from the workbooks of other projects.
- Use the guidelines as entry criteria to design reviews.

Verification

- Check that guidelines exist for the process, notation, and tool usage for each anticipated type of work product.
- Check that guidelines exist for the usage of all tools, where appropriate.
- Check that the guidelines provide for maximal integration of design work products. For example, are the Design Class Descriptions being generated automatically from the Design Object Model and other models? Is API documentation being generated directly from the Design Object Model?

Example(s)

Sample Design Guidelines might be:

Process	The design process is scenario-driven. That is, design proceeds by incrementally transforming the analysis OIDs into design OIDs, each transformation being the result of a design decision.
API	The Application Programming Interface should include C++ header files for each subsystem.
Scenarios	The first few scenarios chosen for transformation should be central to the business of the application. These scenarios are used to drive out the main architectural features of the application. These initial scenarios are modeled by the entire design team under the leadership of the project architect. Subsequent scenarios may be designed by subteams, but the architect must be consulted if a subteam is being forced to make a generic design decision. A generic design decision is one that has implications beyond the single scenario. Daily design meetings are to be held at which the evolving Architecture is presented and discussed. If there is significant doubt as to how a scenario is to be designed, a formal Issue is opened and tracked. An Issue is specified as a problematic scenario and a list of alternative OIDs. The Issue is resolved by selecting one of the OIDs, and documenting the decision including the rejected alternatives.
Issues	Issues are to be tracked by the architect.

Figure 13-2. *Example of Design Guidelines.*

References

None.

Importance

Formal Design Guidelines are optional, but useful in guiding the design process for a development project.

13.2 SYSTEM ARCHITECTURE

Description

In order for a design to be coherent and for the design process to be efficient, it is necessary for certain broad design principles to be established in advance. This agreed-upon set of underlying principles is the System Architecture. Restated with a slightly different emphasis, the System Architecture is the set of global, projectwide design decisions.

A System Architecture can be broad, encompassing many aspects of design, or it can be narrow. Any design statement can be considered to be architectural if it is agreed that it is to have general applicability. The "obvious" areas of architectural interest are the following.

- Structure, the way in which the software is to be layered or partitioned
- The key communication patterns between components of this structure
- Communication (interprocess communication for example)
- Distribution
- Persistence
- Security
- Error Handling
- Recovery
- Debugging
- The use or reuse of specific, existing hardware and software configurations.

Within these and other areas, architectural statements may be strong, imposing a well-defined structure on designs, or they may be weak, insisting on only a minimal structure.

A System Architecture may prescribe new structures, or may insist that the application must use particular, existing class libraries or frameworks.

A distinction exists between the *problem domain* explored during analysis and the *solution domain* defined during design. The solution domain can often be subdivided usefully into the following subdomains.

- Application subdomain
- Application support subdomain
- Utility subdomains

The application subdomain consists principally of (design versions of) those classes identified during design. The application support subdomain consists of application-specific classes that the application classes will need in order to deliver their functionality. The utility subdomain consists of those support classes that are application-independent. A number of utility subdomains might be defined according to subject matter such as collections, communications, and the like. Within the context of a banking application, an example of an application subdomain class is Account; an example of an application support subdomain class is CreditStrategy; an example of a utility subdomain class is ISequence.

The definition of the application, the application support, and the utility subdomains, and the mapping of problem domain classes onto classes in these subdomains is what design, as a whole, is all about. It is the global design decisions of the *System Architecture* that provide a framework within which this mapping can take place in a coherent and consistent manner. The architect must balance the need to guide designers with the need to keep the actual design separate from the statements of the principles that underlie the design, the System Architecture. In practice this usually means doing the following.

- Identifying the utility subdomain class libraries.
- Identifying the broad structures and communication patterns of the application support subdomain.
- Providing guidelines to help designers map the problem domain classes into the solution domain.

As discussed below, it is in response to the various kinds of Nonfunctional Requirements that these decisions are made.

Purpose

If the process of design takes place in the absence of a strong System Architecture, design decisions will tend to be *ad hoc* and unrelated. Furthermore, all design decisions must be made from scratch, which is very time consuming. Successful projects tend to be characterized by System Architectures that are simple, strong, coherent and that have been enforced throughout the project.

As the design process proceeds, more and more people gradually become involved. Initially only the architect works on the design. At a certain point individuals and then teams begin to work in parallel. The System Architecture serves to capture those global design decisions that have to be made centrally by the architect before parallel work begins. After this point, making global design decisions is much more difficult and much less efficient.

A good technique for driving the design process as a whole is scenario-driven design, see Section 18.7, which uses bundles of scenarios as the requirements for each cycle of an iterative and incremental development schedule. A potential problem of the technique, however, is that it is so requirements-driven that software developed in this way may be brittle; changes in the requirements might be difficult to implement. The solution to this problem is to perform scenario-driven development within the context of a System Architecture driven not only by project-specific Nonfunctional Requirements, but also Nonfunctional Requirements related to good software engineering principles, such as modularity, anticipating changes, and reuse. The need to balance scenario-driven design in this way is another factor motivating the development of a strong System Architecture.

Participants

A System Architecture is usually the work of one person. Committees lack the focus to develop the required simplicity and coherence. The project architect is responsible for the System Architecture and for ensuring that its principles are followed.

Timing

System Architecture definition is usually the first activity to be performed in the design phase. All other design work may be thought of as mapping the analysis model onto the System Architecture.

Even in the context of an iterative and incremental development process, it usually pays to develop the bulk of a System Architecture very early. Architectural decisions can and should be tested in development increments, of course, and if necessary adjusted in the light of experience, but right from the beginning of design there must be an architectural vision of how the whole system will work. If this is not done, the necessary rework may be too expensive. Iterative rework is to be expected, but it should be as localized as possible. Reworking a System Architecture, or imposing one after the design and implementation is well advanced, will require the kinds of large-scale changes which most projects will not be able to afford. The most likely results of such an attempt will be a weak System Architecture and a complex design lacking coherence.

Having said that a System Architecture must be defined early, the *implementation* of the architectural ideas can happily be spread over a number of development increments, in a risk-driven manner of course. Initial increments might, for example, ignore persistence and distribution, although parallel, early prototyping activity might test proposed resolutions to persistence and distribution issues.

Technique

In the way that the functional requirements (best expressed as a use case model and a set of scenarios) drive the analysis phase, the Nonfunctional Requirements drive the design phase. It is the constraints of the Nonfunctional Requirements that force design decisions. As the System Architecture consists of those design decisions that must be made globally, architectural design is driven by those Nonfunctional Requirements that have a global impact. Selecting a System Architecture, therefore, uses the following iterative process, however formally or informally.

1. List Nonfunctional Requirements that have a global impact
2. Make whichever design decisions must be taken at a global level to meet these Nonfunctional Requirements
3. Assess the effect of these decisions by transforming a representative sample of OIDs in the light of this candidate System Architecture
4. Iterate

Note that the Nonfunctional Requirements addressed by a System Architecture must include not only those that appear explicitly in the requirements document. These requirements capture constraints related to performance, availability, persistence, distribution, security, et cetera. In addition to these *external* constraints, a System Architecture must also address two further categories of Nonfunctional Requirements: software engineering requirements and *internal* requirements. A System Architecture must address the Nonfunctional Requirements that stem from the need to perform good software engineering. These include requirements related to modularity, anticipating changes, and simplifying future reuse of the software. Reuse will not happen unless it is planned and designed to happen. Nonfunctional Requirements such as these are often not surfaced during requirements gathering which, by definition, focuses on the external system constraints. If not, then they should be made explicit as part of the architectural design activity. A System Architecture must also address such *internal* Nonfunctional Requirements as error recovery, naming conventions, messaging, data integrity, heterogeneous environments, multiplatform dependencies, multiple vendors, wrappering, languages, tools, hardware configurations and dependencies, et cetera. These Nonfunctional Requirements too will typically not be surfaced during requirements gathering. In short, all design topics for which global solutions are required should be addressed by the System Architecture.

Strengths

A System Architecture makes explicit the underlying principles of the design. By doing this, the principles can be applied uniformly, and their appropriateness checked. Factoring out design decisions and making them globally prevents the development team reinventing the wheel and ensures a degree of application consistency.

Weaknesses

Developing a strong System Architecture is challenging. It consumes scarce resources when the architect is probably under pressure to allow the project developers to start to do something.

Notation

A System Architecture takes the form of free format text augmented by design and/or code structures and diagrams of hardware and software configurations. To enhance the traceability of the architectural decisions, it helps to pair (or reference) the Nonfunctional Requirements with the corresponding architectural decision. Other than that, you should organize the System Architecture (grouping the decisions) by category, such as persistence, error handling, recovery, and so forth.

Traceability

This work product has the following traceability:

Impacted by:

- Nonfunctional Requirements (p. 106)
- Prioritized Requirements (p. 111)
- Reuse Plan (p. 158)
- Project Dependencies (p. 173)
- Issues (p. 176)
- Design Guidelines (p. 253)
- Target Environment (p. 272)
- Subsystems (p. 274)

Impacts:

- APIs (p. 265)
- Target Environment (p. 272)
- Subsystems (p. 274)
- Design Object Model (p. 281)
- Design Scenarios (p. 293)
- Design OIDs (p. 298)
- Design State Models (p. 306)
- Design Class Description (p. 311)
- Rejected Design Alternatives (p. 316)
- User Support Materials (p. 341)

Advice and Guidance

- The temptation exists to define very generic, abstract System Architectures that will solve whole categories of problem. It is good for a System Architecture to be generic but it is more important that it supports and facilitates actual application development. The simplest way of keeping a System Architecture with its feet on the ground is to insist that it is always related to a particular application and not developed in a vacuum.
- Only explicit architectural statements can be policed.
- If the System Architecture does not address a particular design issue, the architect must accept that no uniform resolution to that issue will necessarily be adopted by the design teams.
- The stronger a System Architecture, the greater the independence of design teams.
- Remember to address the requirement to anticipate changes. Likely modifications or additions to both functional or Nonfunctional Requirements should be considered. Addressing these requirements for change in the System Architecture is a matter of achieving decouplings of some form or another. Use [Gamma95] as a source of decoupling techniques.
- Include statements of the design trade-offs that have been assumed in the System Architecture. These will help people understand the basis of the System Architecture, assist developers design and implement consistently with the architectural trade-offs, and enable the trade-offs to be reviewed at a later date.
- Developers need not remain idle or unassigned while a System Architecture is being developed. There is considerable work to be done setting up the Development Environment and performing prototyping or basic implementation work. Consider using a depth-first development strategy (see Section 17.1) to get developers up to speed with the development techniques, tools, and domains, in parallel with further development of the System Architecture.

- A System Architecture is often developed iteratively together with a Subsystem Model as these are closely related.

Verification

- Check for completeness by developing a list of topics and issues relevant to System Architecture, such as the one at the beginning of this section. Check the System Architecture work product against each item listed.
- Check that anticipated changes to the design are addressed in the System Architecture.
- Check that all design decisions that require global coordination across the project are included in the System Architecture work product.
- For each architectural statement, can a framework (reusable subsystem) or a component of a framework be used to enforce it automatically?
- Check that the System Architecture is driven by the requirements of an application and that it has not been designed "in a vacuum."
- Check that all design decisions that involve risk are documented appropriately in the Risk Management Plan, see Section 10.7, and that adequate, timely prototyping activity has been scheduled to check the decisions.
- Check that the System Architecture has used existing components and technologies where this is feasible and appropriate.

Example(s)

The following example relates to the image processing system whose Subsystem Model is shown in outline in Figure 13-7. The Nonfunctional Requirement from which each architectural statement is derived appears first in italics. This is done here to show the dependencies between System Architecture and Nonfunctional Requirements. Traceability back to Nonfunctional Requirements as explicitly and at such a fine granularity as this may not always be done. Flat lists of architectural statements and diagrams under headings such as *Structure, Error Handling, Reuse*, et cetera, are frequently used.

Table 13-1 (Page 1 of 2). Example of System Architecture.

Nonfunctional Requirement: *Utilize existing C functions to perform basic image processing operations.*
Derived architectural decision: The basic image processing functions are implemented by a component consisting solely of the legacy C code. An interfacing subsystem invokes the legacy functions. This involves extracting images from the image database, placing them in memory in the format expected by the legacy functions, interpreting their return values, returning images to the image database, and the like.

Nonfunctional Requirement: *Use familiar, off-the-shelf technology to distribute images between image processing nodes, at least in early development cycles.*
Derived architectural decision: Store images one per file and distribute images using NFS.

Table 13-1 (Page 2 of 2). Example of System Architecture.

Nonfunctional Requirement: *Reuse as much as possible of this application in similar applications that also exploit image processing technology.*
Derived architectural decision: There is an image processing System Architecture that is independent of the specifics of the application itself. The image processing System Architecture provides a distributed image database, and access to image processing functions.

Nonfunctional Requirement: *Enable the basic image processing functions to be switched on the fly.*
Derived architectural decision: The legacy image processing functions are encapsulated within a subsystem that metamodels the interface to these functions to enable interfaces to new functions to be created dynamically.

Nonfunctional Requirement: *Anticipate new kinds of images. The algorithms to manipulate these new images must be added on the fly.*
Derived architectural decision: The *singleton* and *abstract factory* design patterns [Gamma95] are used to create families of objects related to particular kinds of images. A singleton registry of concrete image factories is maintained. New image code is provided in the form of a DLL that contains the new concrete image factory class and the classes related to the new image that the new factory instantiates. On being loaded, the DLL instantiates a singleton concrete factory and registers it. Image classes are instantiated by means of the appropriate factory obtained from the registry.

Nonfunctional Requirement: *The image processing System Architecture should provide test-bed facilities to enable experimentation with different database access algorithms.*
Derived architectural decision: The *strategy* design pattern [Gamma95] is used to define different database access strategies and to associate them dynamically with image objects.

Nonfunctional Requirement: *Employ uniform error handling mechanism.*
Derived architectural decision: Use C++ exception handling throughout to flag exceptional conditions. Legacy image processing functions currently employ a standard set of return codes. All application-specific exception classes are to be derived from a common class that encapsulates a return code. Legacy code wrappers must check return codes and throw exceptions as appropriate.
Conditionally include code to check all method preconditions identified during design. Preconditions should be coded as protected methods of the invoked class. Preconditions of subclass methods should invoke superclass method preconditions before performing any subclass-specific checking. Do not encode postconditions or invariants.

Nonfunctional Requirement: *Extensible architecture.*
Derived architectural decision: The subsystems will be allocated to the hardware as shown in Figure 13-3.

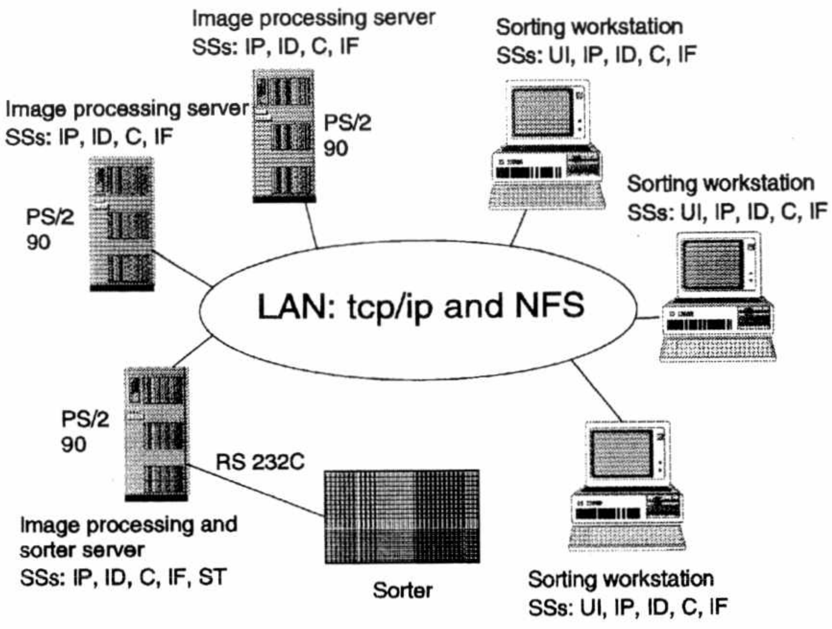


Figure 13-3. Example Architecture Diagram Showing Hardware and Software Configurations.

The software structure of an image processing server is as shown in Figure 13-4.

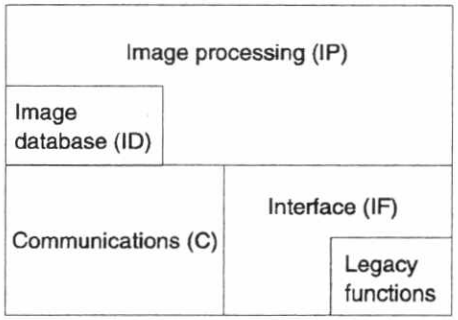


Figure 13-4. Example Architecture Diagram Showing Software Layers.

References

[Booch94] discusses Architecture.

Importance

Essential for ensuring that the design process does not become ad hoc.

13.3 APPLICATION PROGRAMMING INTERFACES (APIS)

Description

The term "API" actually stands for Application Programming Interface, but more accurately it refers to an Architected Program Interface. The point of this pun is to emphasize that APIs need not be interfaces to end-user applications, and that an API needs to be designed. An API work product documents an internal or external API provided by the system.

In object-oriented terms, an API is simply a set of visible classes and their interfaces. It also includes associated global types, data, and functions but these, hopefully, are kept to a minimum if they cannot be eliminated entirely. The classes that appear in an API are a subset of those contained in the subsystem to which the API is an interface. An API can, therefore, be considered to be a particular *view* or *filter* of subsystem classes.

As described previously in Section 2.4, a workbook can describe either a whole system or a subsystem. API work products, therefore, relate either to a whole system or to a particular subsystem depending on the workbook of which they form a part. The Subsystem Model, see Section 13.5, defines subsystems and their interdependencies in terms of *contracts* that are coherent collections of responsibilities. It is subsystem APIs that implement contracts. This mapping of contracts onto APIs is specified in the Subsystem Model.

Purpose

A system or subsystem API is defined to enable the system or subsystem to be used without the need to understand all its internal details.

Participants

Each API, if it is not prescribed completely by the system requirements, has an owner who is responsible for its integrity. In accordance with the workbook approach, the API owner is responsible for documenting it into this work product. The API owner is likely to be a designer or developer.

Timing

APIs are usually defined shortly after subsystem partitioning, as the APIs often provide access to individual subsystems. The need for a particular API usually becomes apparent during the construction of the Use Case Model but its details can only be provided considerably later.

Technique

In the sense that an API is a filtered view of subsystem classes, at least part of the documentation of an API should be obtainable by using the filtering and documentation facilities of a design tool or class browser.

An API cannot be developed entirely independently of the classes that support that API. The need for a particular API often arises from a particular Use Case that Scenarios elaborate into a collection of related system behaviors. The triggers for these Scenarios are implemented by calls to the methods that form the API.

If it is a subsystem being described, as opposed to a system, then Use Cases may not have been defined, although there is no reason why they should not have been, particularly if the subsystem is intended to be reused. If Use Cases do not exist then the driver of an API is the set of Scenarios for that subsystem, or a clustering of those Scenarios into *contracts*. If Scenarios do not exist either, then an API is driven directly by the contracts that define the interdependencies between subsystems.

Strengths

Precise documentation of APIs enables design work to be done in parallel. Subsystems depend on each other and if subsystem interfaces are not defined, no autonomous subsystem work is possible, since the internals of each subsystem will have to be developed simultaneously. Subsystem interfaces modularize a solution.

Weaknesses

None.

Notation

In addition to bare-bones type signatures, such as what C++ header files provide, API documentation should include references and information to permit the API to be understood independently of the system design. The degree of independence will depend, of course, on the intended audience for the documentation: It might be appropriate for API documentation to assume knowledge of the system, or it might not. If the API documentation is to be completely stand-alone, and if it is complex, then an object model and perhaps a set of OIDs should be provided to enable the reader to understand the semantics of the interface. Sample client code is often a good way of illustrating usage while hiding the internal details that OIDs would expose. If the audience for the API documentation might be expected to customize the interface, customization OIDs and/or code should also be included.

The structure of a document describing an API might take the following form:

Chapter 1: Purpose of API
Chapter 2: API structure
Chapter 3: Class interfaces
Chapter 4: Usage scenarios

The motivating Use Case should be used to guide the content of the introductory chapter. The API structure chapter might contain an Object Model together with a textual walkthrough of the model. See the example below for a suggested format for documenting class interfaces. Each usage scenario should contain a description of the scenario, statements of assumptions and outcomes, and an OID and/or sample code.

Traceability

This work product has the following traceability:

Impacted by:
- Nonfunctional Requirements (p. 106)
- Issues (p. 176)
- Design Guidelines (p. 253)
- System Architecture (p. 257)

Impacts:
- Source Code (p. 334)
- User Support Materials (p. 341)
- Test Cases (p. 346)

Advice and Guidance

- APIs should be defined in development increments as early as possible.
- Consider using the *facade* design pattern [Gamma95] to encapsulate the interface or to provide alternative, simplified interfaces.
- Exploit the capabilities of your toolset to generate as much as possible of the API documentation automatically.
- A minimal API document obviously lists just the type declarations relevant to client programmers. The degree of additional documentation: usage scenarios, method descriptions, and the like, should be scaled according to the intended audience and usage of the API.

Verification

- Check API for conformance to Coding Guidelines.
- Check the documentation of APIs intended for publication for conformance with appropriate publication guidelines.
- Depending on the intended audience for the API and its complexity, check that documentation includes Scenarios and sample code showing how the API may be used.
- Check API for appropriate levels of completeness and extensibility.
- Check API for unnecessary exposure of internal data representations.

Example(s)

Here is an example of some documentation that was written to describe a simple C++ database server API. The document as a whole followed the structure outlined earlier. The excerpt that follows is from a section documenting the *DbSession* class interface. Note that this documentation is intended for writers of *client* code, not *customization* code. It is for this reason that only the public class interface is described. Note also that the documentation is intended for object technology novices. For this reason the names and short descriptions of inherited methods are included in the description of each class interface. This would not normally be done, but it was felt that this audience would expect to find all methods relevant to a particular class documented there.

Definition

A DbSession is an object representing a database session. FileProxies for database files are created by DbSession methods; the contents of the files are then accessed by means of FileProxy methods without further direct reference to the DbSession.

DbSessions *know* the FileProxies that are associated with them, and deletion of a DbSession object causes deletion of its FileProxies. To ensure that FileProxy construction and deletion only occur at the correct time (and not, for example, when a DbSession object is copied), copy and assignment operations are not part of the DbSession public interface.

Public methods

Static methods:

ApplicationManagerInstance	Get the DbSession initiated by the application manager

New methods:

OpenFile	Open an existing database file
NewFile	Create a new database file
OpenBinarySegmentFile	Open an existing binary segment database file
NewBinarySegmentFile	Create a new binary segment database file
OpenVoiceSegmentFile	Open an existing voice segment database file
NewVoiceSegmentFile	Create a new voice segment database file
GetHandle	Get the database server handle for this session
GetPath	Get the path used by the database server as the database file directory

Inherited, overridden, or instantiated methods:

None

Special methods:

Constructor	Create a DbSession
Destructor	Delete a DbSession

Descriptions of new methods

ApplicationManagerInstance

Purpose Get the DbSession instance that represents the database session initiated by the application manager. If the instance does not yet exist, the method will construct it.

Format

```
static DbSession& ApplicationManagerInstance(const GsiName
gsiName)
```

Parameters

gsiName The NetBIOS name of the GSI for which the session is being used.

Notes The method wraps the vmsfopen function.

OpenFile

Purpose Open a database file and return a reference to a proxy for it.

Format

```
virtual FileProxy& OpenFile(
      const FileName fileName,
      const DbSession::AccessMode accessMode = DbSession::READ_ONLY )
```

Parameters

fileName The name of the file to be opened.
accessMode The access mode of the new file proxy.

Notes

NewFile

Purpose Create a new database file and return a reference to a proxy for it. The underlying semantics of the method are those of CreateDA in the existing API.

Format

```
virtual FileProxy& NewFile(
      const FileName fileName,
      const int keyLength,
      const int recordLength,
      const int degree = 32 )
```

Parameters

fileName The name of the file to be created.
keyLength The length of the key. Possible values are 1 through 49.
recordLength The length of the record (both key and data).
degree Parameter influencing database index creation algorithm; default 32. Use the default unless otherwise instructed.

Notes The new file proxy is read-write.

Constructor

Purpose Create a DbSession object.

Format

```
DbSession(
      const ClientName clientName,
      const GsiName gsiName,
      const PVOID ioArea = 0,
      const int ioAreaLength = 0,
      const int timeout = 0,
      const int sessionCount = 3,
      const IBoolean dllServer = False,
      const int adapter = 255 )
```

Parameters

clientName The NetBIOS client name.
gsiName The NetBIOS name of the GSI with which the session is being opened.
ioArea A pointer to the area to be used for GSI requests and responses; default 0. If the pointer is zero (0), the API allocates the storage, but in that case the storage will be overwritten by each GSI request via this session.
ioAreaLength The length of the user-supplied ioArea.
timeout The time allowed, in seconds, for a response from the LAN; default 0. The maximum value is 127. A value of zero (0) specifies that no timeout is to be used.
sessionCount Only relevant if a real network is being used. The number of network sessions opened for the process; default 3. Only the sessionCount parameter for the first session (of any kind) created for each process is used.
dllServer Whether or not the call is made from a DT/2 DLL server; default False.
adapter The LAN adapter number as defined to OS/2 system configuration; default 255. Possible values are 0 through 3 (LAN) or 255 (local). If an adapter has not been installed, use value of 255, which means that all operations take place locally.

Exception codes As for OpnSesDAE.

Notes

Supporting declarations

enum AccessMode {
  READ_ONLY, READ_WRITE }

References

None.

Importance

API work products are optional but may be essential if your project has numerous Application Programming Interfaces or may be required by your customer. APIs work products are important for allowing a system or subsystem to be used without the need for a complete understanding of that system or subsystem.

13.4 TARGET ENVIRONMENT

Description

The target environment is the environment in which the application will operate. The specification of the Target Environment usually comes from the Nonfunctional Requirements and it has a strong influence on the System Architecture. It should specify hardware platform, operating system, and the runtime environment. In a distributed or a client/server system, a Target Environment may describe more than one physical system.

Purpose

It is critical that the environment in which the application will operate is clearly specified. The purpose of this documentation is to ensure that both the end user and system architect share a common understanding of the that operating environment. A change of the Target Environment may result in a selection of a different System Architecture, therefore, its documentation is important.

Participants

The specification of the target environment may come from a customer requirement or a decision by system architects. The architect of the project or project planner should document the Target Environment of the application.

Timing

The information concerning the Target Environment is collected and documented at requirements gathering time. At times, this information may be elaborated during design.

Technique

Any customer or user input to the choice of Target Environment should be obtained and validated like any other Nonfunctional Requirements. If the selection of Target Environment is made by system architects, this specification should be sent to the customer to verify their agreement with the selection.

Strengths

The strength of this work product is to document clearly the operating environment of the system or application.

Weaknesses

None.

Notation

The Target Environment is usually documented as free format text with diagrams as appropriate.

Traceability

This work product has the following traceability:

Impacted by:

- Nonfunctional Requirements (p. 106)
- Issues (p. 176)
- Design Guidelines (p. 253)
- System Architecture (p. 257)

Impacts:

- System Architecture (p. 257)
- Coding Guidelines (p. 322)
- Physical Packaging Plan (p. 326)
- Development Environment (p. 330)
- User Support Materials (p. 341)

Advice and Guidance

It is important that the environment for the application is clearly specified. All the known information on the environment should be recorded in this work product, namely, hardware platforms, operating systems, network protocol, language system runtime requirements, and so forth.

Verification

- Check if the target environment specification is complete with respect to all operational dependencies.
- If more than one system is involved, check to make sure all target environments are specified.

Example(s)

The following are examples of Target Environment specifications.

- The target environment is RISC System/6000 running AIX version 3.2.
- The target environment is ES/9000 running CICS/ESA version 3 and LE/370 version 1 release 2 under MVS/ESA SP version 5.1.
- The target environment for the client is IBM PC running OS/2 WARP and the server is RISC System/6000 running AIX version 3.2.

Figure 13-5. Examples of Target Environment.

References

None.

Importance

The Target Environment is essential since it has a major impact on the System Architecture.

13.5 SUBSYSTEM MODEL

Description

A Subsystem Model is a partitioning of a system into subsystems, and a delegation of system responsibilities to the subsystems. The term "subsystem" is used in this book to refer to *any* large design component. A database management system can, therefore, be a subsystem, as can a user interface component or an application framework. Subsystems, because they are large structures at the design level, must take into account the System Architecture.

This definition of the term "subsystem" is similar to that used by Booch [Booch94]. OMT [Rumbaugh91a] uses the term differently: a logical grouping of classes. This is more of an analysis concept than design, and we prefer to distinguish the two kinds of partitioning. Both analysis and design need a means of clustering, but the goals of analysis and design clustering are not the same, and hence analysis and design cluster divisions may not coincide. We use the term "subsystem" solely for design clustering in order to avoid confusion.

Our definition also differs from what Shlaer and Mellor call "domains" [Shlaer88] which are horizontal partitions each consisting of all the classes of a system in a particular Subject Area. Shlaer and Mellor domains are similar to the "layers" of Booch. Domains are used in a very definite manner by Shlaer and Mellor.

Another design concept that is not necessarily identical to that of the subsystem is the unit of work allocated to a team of developers. There is no reason why units of work need necessarily be aligned with subsystems, although pragmatic constraints may insist that this is the case in particular projects.

A subsystem is not necessarily a design for a particular physical component. The design of physical components will probably be expressed in terms of subsystems, but not all subsystems represent physical components. Some subsystems may contain nested subsystems.

A Subsystem Model identifies existing subsystems that are to be reused as well as those that need to be constructed.

Purpose

The principal reason why a system is subdivided into subsystems is that the system is too complex or too large to understand or to be worked on as a whole; it needs to be partitioned into smaller units to be manageable.

This rationale has two implications. Firstly, each subsystem must be understandable in isolation. Otherwise, no understanding would be gained by partitioning the system. This means that each subsystem must have its own Object Model, its own Scenarios, and its own OIDs. This does not mean, for example, that a class cannot appear in more than one subsystem's Object Model, but the ownership of classes by subsystems must be clear. Secondly, the public subsystem interfaces must be clearly and fully described. This is done so that the developers of one subsystem need not know about the internal structure of other subsystems to use them. If this were not the case, the benefits of partitioning the system would be much reduced.

Another important reason for constructing a system from subsystems is reuse. It may be possible to reuse existing components or to identify problem-independent functionality that can be "harvested" for subsequent reuse. In either case, there is a need to structure the system into large, isolated, design components.

Participants

The architect is responsible for the definition of the subsystems. The project manager must also be involved, as the subsystem partitioning may affect the way in which work can be assigned to teams and the dependencies between the teams.

Timing

Subsystem partitioning is usually done iteratively together with the development of an Architecture, as a result of addressing Nonfunctional Requirements related to defining physical boundaries, achieving modularity, reusing existing components, and designing for further reuse. Both the System Architecture and the Subsystem Model are revised iteratively as design proceeds.

A large system may, however, be partitioned into subsystems at an earlier stage, before analysis, in order to address logically separate aspects of the system independently. Sometimes, parts of previous System Architectures and Subsystem Models can be reused for new systems.

Technique

How to partition a system into subsystems depends on when the partitioning is performed. If it is decided to partition before analysis then a clear idea of how the system can be cleaved into subsystems must already exist; otherwise, such an early partitioning would not be attempted. These lines of cleavage may be those of an existing application that is being re-engineered, or they may be based on the boundaries of existing systems that are being integrated. If the system is completely new, then an early partitioning may stem from a domain analysis, see Section 18.1, which has surfaced a clear separation of concerns in the domain.

If a subsystem partitioning is performed as an initial design step, then it will be based on knowledge gained during analysis, an awareness of the components available for reuse, and physical boundaries. The analysis model might naturally have generated some clusterings of classes that are obvious candidates for encapsulation as subsystems. Physical packaging requirements may reinforce this initial partitioning, or they may impose additional demands of their own. If a project is distributed then the system partitioning must to a certain extent correspond to geography.

If a system is partitioned into subsystems only after an initial Design Object Model and Design OIDs exist, then the partitioning can be performed by clustering the classes of the (system) Object Model and splitting the (system) OIDs into separate OIDs for each of the newly created subsystems. When splitting off a subsystem OID in this way it is often appropriate to represent other subsystems by *Facades* (in the sense of the Facade design pattern of [Gamma95]). A Facade is an object that encapsulates a whole subsystem. In this way the use of subsystem interfaces is emphasized, and the internals of one subsystem hidden from others. The messages sent to a Facade form the interface to that subsystem. Facades are useful when constructing subsystem OIDs irrespective of when subsystem partitioning is performed.

Whenever the partitioning is performed, some heuristic for clustering must be used; a rule of thumb is that subsystem partitioning should minimize the dependencies between subsystems. Obviously, the clustering must also take into account physical boundaries, reused and reusable components, and architectural principles.

In addition to identifying subsystems, a Subsystem Model must also document the way in which subsystems depend on each other. Individual methods or responsibilities are, however, usually too fine-grain to be used as the basis for documenting relationships between subsystems. It is necessary to cluster the responsibilities of a subsystem into *contracts* and to use these to show the links between subsystems. A contract is a coherent collection of related responsibilities. (This is the definition of a contract used in [Wirfs-Brock90].) A subsystem is dependent on a contract of another subsystem, if it employs the responsibilities of that contract to deliver its functionality. It is these subsystem-contract connections that are shown on the subsystem diagram described below.

Each subsystem that is identified is, in principle, treated as a system in its own right. As described earlier in Section 2.4, Workbook Structure, a separate logical workbook is devoted to each subsystem. By giving each its own workbook, the structure exists to manage, gather requirements for, analyze, develop, and test each subsystem independently. Of course, subsystems will not always be developed quite so autonomously or formally, although deriving separate requirements and analysis work products for certain subsystems will greatly assist the understandability and future reuse of these subsystems. If requirements gathering for a subsystem is performed, then a subsystem Use Case Model will be produced. There is a correspondence between a subsystem Use Case Model and contracts in the Subsystem Model of the enclosing system. The contracts presented by a subsystem are the reasons why the subsystem is needed in the Subsystem Model; the subsystem Use Case Model captures the requirements on the subsystem. The subsystem Use Case Model must, therefore, support the subsystem contracts that are implemented by subsystem APIs.

Strengths

The Subsystem Model is an extremely important design document. It straddles the earlier, systemwide design and the later per-subsystem design. To a certain extent it summarizes the System Architecture of the application. When design teams communicate, it is at the subsystem level of abstraction; therefore, it is very important that the subsystem level of design is well documented to satisfy the varied goals of the subsystem partitioning process.

A Subsystem Model provides a basis for identifying reusable or harvestable design components.

Weaknesses

None.

Notation

Logically, each subsystem has its own workbook similar to that used for the system as a whole. Physically, the subsystem workbooks may be parts of the system workbook or they may be separate, as appropriate to the development team size and structure. If the subsystems are in turn broken down into subsubsystems, the latter, too, have their own subsystems.

The work products of the individual subsystems, including the APIs that define the subsystem interfaces, their object models, their code, and the like, belong, therefore, to the subsystem workbooks. What is important in the parent system workbook is to indicate which subsystems exist and how they are interrelated. All other information is delegated to the subsystem workbooks.

The following template may be used for each subsystem identified in the Subsystem Model:

Subsystem name
Description
Workbook
Contracts

The *Workbook* slot is used to identify the workbook in which the subsystem is documented. All further documentation of the subsystems is delegated to the subsystem workbooks.

Contract name ______________________
Description ______________________
API ______________________
Notes ______________________
____
____

The *API* slot is used to identify the subsystem API that implements the contract. All documentation of the API is delegated to the relevant API work product of the subsystem workbook. In addition to the use of the above templates, a summary diagram showing the subsystems and the contractual dependencies between subsystems is frequently helpful. The form of a subsystem diagram is shown in Figure 13-6.

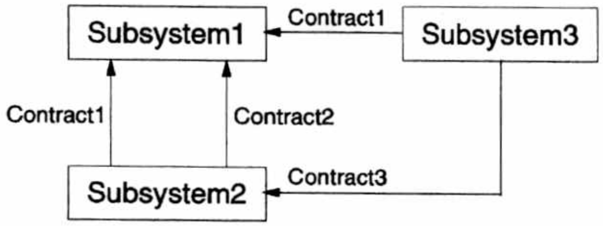


Figure 13-6. *The Form of a Subsystem Diagram.*

Each box of the subsystem diagram represents a subsystem. The arrows connecting the subsystems represent dependencies between subsystems with the arrows pointing towards the server subsystems. The names on the arrows are those of the contracts of the server subsystem being depended upon by the client subsystem.

Traceability

This work product has the following traceability:

Impacted by:
- Nonfunctional Requirements (p. 106)
- Reuse Plan (p. 158)
- Project Dependencies (p. 173)
- Issues (p. 176)
- Design Guidelines (p. 253)
- System Architecture (p. 257)
- Design Object Model (p. 281)

Impacts:
- System Architecture (p. 257)
- Physical Packaging Plan (p. 326)

Note: The Subsystem Model also impacts lower level subsystem work products, most notably Use Case Models and APIs.

Advice and Guidance

- The subsystem partitioning is likely to change from time to time. This is inevitable as experience is gained of system responsibilities.
- Reduce dependencies between subsystems as far as possible.
- Make subsystems problem-independent, if possible, to increase the chance of the subsystem being reused.
- The (logically) multiple subsystem Object Models will almost certainly be represented as divisions within a single model maintained by a design tool.
- After a subsystem partitioning, OIDs may need to be reworked to reassign responsibilities to optimize subsystem decoupling.

Verification

- Check that all subsystem APIs are, or will be, defined in time to enable them to be used by the developers of other subsystems.
- Check the subsystem partitioning for optimal decoupling of subsystems. (This is not necessarily the same as maximal decoupling.)
- Check the frequency and size of intersubsystem communications.
- Check the subsystem partitioning for reuse opportunities; both the reuse of existing components, and the future reuse of new work products.
- Check that the boundary of the system to be built is explicitly represented in the Subsystem Model diagram.

Example(s)

Figure 13-7 shows a subsystem diagram for an image processing architecture. An important design goal for this System Architecture was to develop subsystems that were reusable in several image processing applications. The *Image functions* subsystem consist of many, large legacy functions written in a non-OO language that are to be reused without change. To enable this, the *Image processing* subsystem is responsible for extracting the images from the image database, placing them in memory as expected by the legacy functions, invoking the legacy functions, and interpreting their return values.

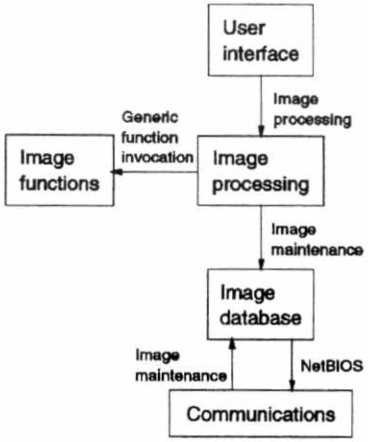


Figure 13-7. *An Example of a Subsystem Diagram.*

The description of the image database subsystem might be as follows:

Table 13-2. *Example of Subsystem Description.*

Subsystem name	Image database
Description	A distributed image database
Workbook	IPA_ID
Contracts	Image maintenance

The description of this contract may be as follows:

Table 13-3. *Example of Contract Description.*

Contract name	Image maintenance
Description	An API for maintaining a distributed image database.
API	IM
Notes	1. All image processing operations are implemented by subclasses of the *Image* abstract class. 2. By default, images are stored in the local node. 3. Dynamic load balancing may result in images being moved, but that is functionally transparent. 4. Images may be accessed from any node, regardless of where the image is stored. 5. A mapping from C++ pointers to unique system image IDs is maintained within the subsystem, but is not exposed.

References

- [Booch94] describes subsystems.
- [Wirfs-Brock90] describes contracts.

Importance

The Subsystem Model is optional in small and medium sized projects but essential in large projects. Developing a Subsystem Model provides a means for partitioning a system into more manageable units.

13.6 DESIGN OBJECT MODEL

Description

The Design Object Model is a structural representation of the software objects (classes), that comprise the implementation of a system or application. A Static Model is comprised of design object classes and their attributes, responsibilities, operations, and interrelationships, expressed as association, aggregation, and inheritance links. The object model is a key object-oriented design work product.

Purpose

An Object Model is the fundamental way to document the static aspects of an object-oriented solution to a problem. The focus of the Object Model during design is the structure of the software system as opposed to the structure of the problem domain during analysis (Section 11.3, Analysis Object Model).

The objects in object-oriented design are called "design objects" (vs. "problem-domain objects" in object-oriented analysis). Classes of problem-domain objects can be mapped to one or more classes of the corresponding design objects, depending on the underlying

System Architecture. For example, class *Folder* may be mapped to two classes *Folder* and *FolderView* in design (see examples on page 291). The architectural mechanism here is the Model-View-Controller framework for a system with graphical user interfaces. In this case, the real-world *Folder* object becomes two objects in the solution-domain: The *Folder* data and behavior is part of the "Model," but its graphical representation and on-screen behavior is part of the "View" domain.

Many new classes are invented at this phase, as architectural and design decisions are applied. This is one of the most creative phases of the object-oriented development life cycle. Programmers should have a very clear knowledge of how to proceed with implementation of the system when all the design work products are defined.

Participants

The Design Object Model is created by architects, designers and developers.

Timing

The Design Object Model is primarily developed during the design phase, but it will need to be maintained during the implementation phase as design and implementation decisions are worked out.

Technique

The best way to develop a Design Object Model is to start with the Analysis Object Model and expand it into a design model. The steps to doing this are:

1. Start with a copy of the Analysis Object Model.
2. Add new classes from the solution domain, for example, view and utility classes (identified while developing Design Object Interaction Diagrams).
3. Validate and assign responsibilities from Design OIDs and Design State Models in terms of:
   - attributes (for structural aspects)
   - methods or operations (for behavioral aspects)
4. For every class and association in the model, consider the operations that can result in an instance being created or deleted [D'Souza95].
5. Optimize the object model for performance and reuse. Introduce new associations or modify existing ones to optimize access based on Nonfunctional Requirements (e.g., fast look-up). Consider making associations that were thought to be permanent features of the class into temporary links between objects that are either passed as arguments in methods or created as temporary objects within a method body.
6. Eliminate or collapse structures representing unnecessary information, for example, those not substantiated by Design OIDs.

7. Add metaclass if your implementation language is Smalltalk or SOM. A metaclass is a class whose instances are themselves classes [Booch94]. Metaclasses enable classes to be manipulated as objects in their own right.
8. Transform generalization (or inheritance) structures to delegation when appropriate in order to decouple elements of the design. If a subclass does not pass the substitution (Is-A) test, change Is-A to Uses or Has.
9. Determine which classes will be persistent (for example, based on some startup reference or update Design OID).
10. Specify the accessibility of operations:
    - *Public*: any class can invoke the operation
    - *Protected*: only subclasses can invoke the operation
    - *Private*: no other class can invoke the operation

    If your implementation language has no support for *Private* or *Protected* use Coding Guidelines to enforce them, for example, *Protected* could be expressed in SOM by using the `private` IDL brackets and the `private` emitter switch.
11. Specify the implementation details of associations:
    - Directionality (cf. `using` in Booch [Booch94]). Remember that during analysis associations were bidirectional.
    - Name, visibility, and mutability of the attribute implementing the association.
12. Determine the implementation of the "many" ends in associations:
    - A collection class as an attribute in the "from" object
    - A custom class (usually a subclass of a collection class or an aggregate including a collection class object) as an attribute in the "from" object
    - A reference to an association class that contains one or more attributed references to other objects.
    - No implementation (if the link will not be used in that direction).
13. Determine ownership (Who creates or instantiates objects of a class? Who deletes them?)
    - Determine which aggregations (in particular) represent lifetime encapsulation. That is, the lifetime of the aggregate completely encloses the lifetime of the component (cf. `server binding` in [Coleman94]). Ownership usually implies lifetime encapsulation.
14. Establish the kind of reference (cf. `physical containment` in [Booch94]) to be used by the attributes implementing associations and aggregations:
    - By *reference*: contains a pointer or a reference
    - By *value*: contains an object (applicable only for aggregations at the design level of abstraction. During analysis, reference/value distinctions are usually ignored)

- By *key*: contains a key that can be converted into a reference by some means (for example, a "name resolution mechanism").

15. Determine scope of associations:
    - *Operation Scope* (dynamic reference): when an object A gets a reference to an object B through a method parameter or by a local variable of a method
    - *Class Scope* (persistent reference): reference from object A to object B needs to persist between method calls
16. Determine mutability: The *mutability* of a reference indicates whether it can be reassigned after initialization [Coleman94]. The `const` adornment can be used to indicate immutability. An attribute can be declared as mutable, that is, the value of the attribute can be changed after initialization, for objects declared as constant, when the `const` adornment applies to the entire object.
17. Determine placement of class operations and class attributes (those that do not apply to specific objects, i.e., instances of the class). In many cases they become attributes and operations of a collection class. For example, the extension of a class, namely, the set of all instances, could be maintained in a class attribute (`static data member` in C++) that is a collection class.
18. Determine special or advanced properties of classes and operations (for example, C++'s template, virtual, inline, const, friend, et cetera). Advanced properties are normally implementation language dependent.
19. Determine the types of all attributes.

Strengths

The Design Object Model is a key deliverable for capturing and communicating solution domain understanding to the development team and other interested parties. Its ability to be simple or richly adorned, capturing all static design information, allows it to be extremely useful throughout the development life cycle. Its clear intuitive notation makes it a favorite work product of all analysts, designers, and developers.

Weaknesses

- A Design Object Model only shows the static relationships.
- It needs to be maintained over time. Neglect of the Design Object Model often stems from the (incorrect) assumption that program Source Code supersedes the design documentation.

Notation

The Design Object Model notation enhances the Analysis Object Model notation (Section 11.3), with the emphasis on architecture and design class description. The enhancement includes:

Directionality

Associations should show the direction(s) of reference between the objects. Association classes and n-ary associations should be transformed to show how they will be implemented (e.g., by containment of a collection class of references, or by reference to a directory of associations).

Class adornments

Whether a class is abstract or parameterized could be shown in the Design Object Model.

Attribute and accessibility adornments

The accessibility of attributes and methods, such as whether they are public, private, or protected, should be shown explicitly in the Design Object Model.

Aggregation and association adornments

Whether associations are "by reference" (owner or not) or "by value" should be shown explicitly.

Visibility dependency

Argument dependency and temporary variable dependency need to be represented in Design Object Diagram, for example, by a dotted line with an arrow head. A class has an argument dependency on another class if one of its methods refers to the other class when defining its formal parameters. On the other hand, a class has a temporary variable dependency on an other class, if it instantiates the class as a local temporary variable (See Figure 13-8).

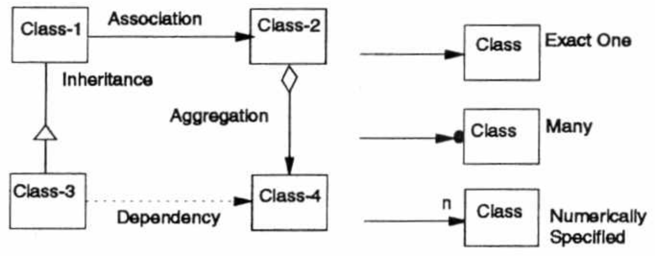


Figure 13-8. Design Object Model Notation.

Traceability

This work product has the following traceability:

Impacted by:

- Nonfunctional Requirements (p. 106)
- Issues (p. 176)
- Analysis Object Model (p. 192)
- Screen Flows (p. 237)
- Screen Layouts (p. 242)
- Design Guidelines (p. 253)
- System Architecture (p. 257)
- Design OIDs (p. 298)

Impacts:

- Subsystems (p. 274)
- Design OIDs (p. 298)
- Design State Models (p. 306)
- Design Class Description (p. 311)
- Glossary (p. 355)

Advice and Guidance

- If you are using an object-oriented CASE tool, copy the relevant part of the Analysis Object Model into a new file (keep separate versions of the Analysis Object Model and the Design Object Model).
- Design Object Model diagrams can become complicated and cluttered. We recommend layering the diagram to reduce clutter and having separate views that focus on particular topics, for example, one with associations but no attributes or operations, and another that focuses on class attributes and operations.
- Not all class dependencies should be depicted as direct associations. An object can be referenced by another if it is passed as an argument to an operation of the other class. When this happens, it is better to model it as a *visibility dependency* between these two classes.
- There is no “association class” in design. Association classes should be converted into normal classes (perhaps derived from or containing a collection class) having direct associations with the other classes.
- Some languages, including C++, permit subclasses to hide inherited services. This is not considered good style. In an inheritance hierarchy a subclass should support all the services provided by its superclass, so that it can be used anywhere the superclass is used. This is known as the substitutability principle[11], i.e. that objects of the subclass are substitutable for objects of the superclass.
- Used properly, inheritance is a key means to achieve object-oriented designs that have well-formed abstractions and whose class structures are reusable and extensible. The misuse of inheritance, however, can lead to designs that are brittle and applications that are error-prone and difficult to understand and modify. When designing inheritance structures, a number of process guidelines should be considered:
    - Verify the class hierarchy has structural and behavioral conformance. Type-safe use of inheritance considers the conformance of client-server object interactions. Structural conformance concerns the client's expectations of the supplier's properties and relationships with other classes. Behavioral conformance concerns the client's expectation of the supplier's behavior. When accessing properties or navigating relationships or invoking behavior, the client should not need to know if it is dealing with a supplier that is an instance of a class or one of its subclasses.
    - Use abstract classes for interface reuse—to enforce common interfaces across subclasses.
    - Use structural composition and operational delegation instead of structural subclassing and operational inheritance if the only reason for inheritance was “sharing” of code. Factor-out classes into more atomic behaviors so that large hierarchies become "forests" of trees.
    - Consider replacing inheritance with aggregation and states (an object *Has-A* state that determines its behavior) when objects are expected to change types during their life cycle (see “State design pattern” in [Gamma95]).
    - Limit the depth of inheritance hierarchies to a maximum of 4 to 5 levels. Deep inheritance hierarchies are a conceptual burden, cause maintenance problems, and can make it extremely difficult to reuse or extend a design.
    - Design the Inheritance Hierarchy consistently with Design Patterns. Many design patterns exploit abstract superclasses and inheritance-based relationships, for example Adapter, Bridge, Builder, Composite, Iterator, Mediator, Proxy and Strategy. The underlying principles are often collaborations between superclass and subclass methods using so-called “template” and “hook” methods [Pree95].
    - Document the intent of Inheritance. If inheritance is used for code reuse—to “borrow” methods from superclasses—then class hierarchies quickly loose semantic cohesion. Document cases where inheritance is not behaviorally or structurally conforming and how the design would have to change to preserve conformance.
- Document your design decisions (including rejected ones).
- Look for design pattern opportunities. The chosen architecture template may already advocate the use of certain design patterns but many others will come from the discovery of how to use patterns in innovative ways. For example, looking for recurring patterns of interaction among analysis objects often suggests design patterns that will structure the solution.

[11] The concept of substitutability also covers the semantic relationships between services, i.e. “require no more, promise no less.” See Chapter 10 “Proper Inheritance” in *C++ FAQs*[Cline95] .

- Use design patterns to insulate the design of the subsystem. When considering collaborations with other design levels (subsystems) do not bring components from other levels into your design directly. Adopt a design pattern that allows you to access the other level yet protects your system's design against changes to that other level, for example: *facades, adapters, proxies*, and *mediators*.
- Apply design patterns that ease the reuse of an asset. Do not constrain the design session by imposing asset interfaces that seem either wrong, inadequate, or unnatural for the current system; the Adapter pattern helps you to maintain design elegance in the face of unnatural asset interfaces.
- Apply design patterns that protect the reuser against changes to the reused component. If the component is a major concept in the model, widely used, and is likely to change, then apply the *Bridge* pattern. The cost of changing components that reference the asset component is likely to be greater than that of applying the *Bridge* pattern. If the new component needs only a subset of the capabilities of the existing one, introduce a new component using the *Adapter* pattern [Gamma95].

- The following questions can help you to decide on visibility and ownership for aggregations:
  - Is the identity of the contained object used outside of the containing object?
  - What is the cardinality of the containment relationship?
  - Are there other containing objects that contain the same component?
  - Should the contained object know its containing object?
  - Is the lifetime of the components bounded by the containing object or not?
- External Agents are normally out of scope in design (although as part of the problem domain their presence in the analysis model is justifiable).
- Use a "managed pointer" whenever it is not obvious who and when will delete the objects referenced (for example one object is referenced from entries in two different collections).
- Attributes implementing superclass associations should be declared as `Protected`., or as `Private` with `Protected` accessor operations.
- Use a Reuse-based Approach. Further structuring of the solution arises from requirements that the design presents to itself—for example, objects that coordinate or mediate the activities of other objects and objects that organize queues and manage events.
  - Generate Requirements During Design. As the design proceeds new requirements emerge for the design itself. These should be formulated as design specifications and used to guide the search for reusable assets and candidate design patterns.

  - Use Frameworks. A framework stipulates how specific application subclasses should behave. Abstract classes and deferred methods require subclasses to fulfill expectations of 'template' methods. Other methods and classes of the framework should be left well alone.
  - Use standard utility classes where possible. Many of the productivity and quality benefits of object-orientation stem from the reuse of libraries and frameworks. We should always approach the design task by being knowledgeable about what already exists and that should shape the way we solve problems.
- Once the object model stabilizes and there is confidence in the completeness of the requirements, you should inspect the classes to look for opportunities to refactor the design.
  - Consider removing classes that are leaves or orphans of the inheritance structure, that is, those that are not instantiated nor inherited from. It can be the case that these classes are poor abstractions that should be reformed either by creating a more general concept, introducing a missing superclass or by reallocating their behavior.
  - Consider merging classes that have related responsibilities. Small classes should be inspected as candidates for features that have a common generalization. By being combined they might offer a more useful abstraction.
  - Consider splitting those classes that have too many attributes or methods. Large classes often exhibit a bias towards a procedural rather than behavioral decomposition of the system. Factor out common behavior and attributes to a superclass. Look for disjoint subsets of coupling between attributes and methods as an obvious way to split the class. Refactor on the basis of behavioral consistency with other parts of the system. Inspect behavior within the class and consider design patterns that can restructure the class.

Verification

See the checklist for an Analysis Object Model, Section 11.3, which is also appropriate here (with the obvious exception of "check for design artifacts").

- Check that Object Model documentation indicates where, how, and why design patterns have been used, where this is appropriate.
- Check the *intent* of design patterns that are used.
- Check that internal representations of classes may be changed independently of other classes, except where explicit dependencies exist. That is, check that knowledge of internal representations is localized as much as possible.
- For all inheritance links, check that public inheritance always implies substitutability, i.e., that objects of the subclass are substitutable for objects of the superclass; that

superclass assertions are honored by redefined methods and that deep inheritance hierarchies are designed correctly, particularly regarding flexibility and the advisability of avoiding downcasts.

- Check that there is no intensive communications around one class. If one class dominates the communication pattern, it may have too many responsibilities and knows too much about too many parts of the system. This could become a fragile part of the system: If any of the parts it knows about changes, it may have to change. If it is a large and complex component, then the chances of introducing an error or propagating changes is higher.
- Check that there are no unnecessary intensive communications between classes. Two classes communicating with each other intensively evidently need to know a lot about each other. Perhaps they should be one component or there is a third abstraction waiting to be found.
- Check that there is no redundant or duplicated behavior. Are there classes with the same behaviors? Are there behaviors that can be merged? Are there behaviors that could be removed either by moving the behavior to other methods or by elimination altogether?
- For each class, verify that names are unique and descriptive and, where necessary, a stereotype has been given and overrideability has been specified. Verify that attributes and operations are functionally coupled—there are no disjoint cut sets that could suggest splitting of the class into new classes. Check how instances are created and destroyed.
- For all operations, verify that operation names are descriptive and expressive of their purpose and, where relevant, an overrideability attribute has been defined; that operations are conceptually the same across classes that have the same names; that the types of all arguments and any return results are defined and that the constraints relating to referential integrity, attribute values, preconditions, and postconditions of operations are implemented.
- For all attributes, check that their names are descriptive; that there are no attributes that represent the same thing; that the types of all attributes are defined, either explicitly, as for C++, or using a signature-style for Smalltalk and that the attribute represents a value-based property and is not an object reference that is implied by an association/aggregation or that represents a missing association/aggregation.
- For all relationships, verify that the relationship is named descriptively, that multiplicity and directionality of references has been specified, and that the relationship is always true. Check that associations do not reference too specific an object; that they do not warrant becoming classes in their own right; and that they should not be modeled as aggregations.

- For all aggregations, review visibility and ownership. Is the identity of the contained object used outside of the containing object? If so, then verify that existence constraints are properly observed outside of the containing object. Are there other containing objects that contain the same component? If so, are multiple owners responsible for controlling the existence of the contained object? Are these owners members of the same class? If so, then indicate that the aggregation itself has a multiplicity of "many" by placing a "dot" next to the aggregation diamond. Should the contained object know its containing object? If so, then indicate that visibility is bidirectional.

Example(s)

The example shows a snapshot of the Object Model at the design level for the Library system example previously introduced in Figure 13-9. By comparing the same Object Model at the analysis level (Figure 11-7), it can be found that many *view* classes and directed associations have been added to this model. On the other hand, *User* class has been removed, since it is not within the system scope. By incorporating directed associations, the Object Model at the design level reflects the principle, derived from the `Observer` pattern (cf. Model-View-Controller mechanism) that "only view objects know model objects, but not vice versa."

The key part of the model-view used in the example is the *Observer* design pattern [Gamma95]. This pattern helps determine how to map the classes defined in analysis to those in design. In the example here, only part of that pattern is shown with the assumption that a certain superclass will handle the notification.

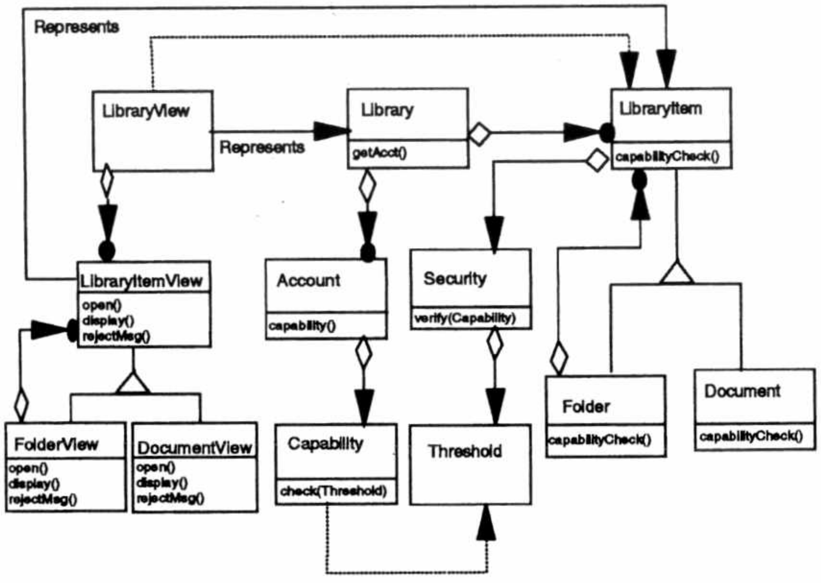


Figure 13-9. Example of a Design Object Model.

Associations representing conceptual links are dropped in the Object Model at the design level. The analysis association between classes `Capability` and `Security` represent a conceptual link as is shown in Figure 11-7. It is replaced in the design by an association of operation scope. This can be documented in the definition part of the class specification at the design level as a "Class Visibility Dependency" clause. At run time, objects of `Capability` can be passed to objects of `Security` for security checking, through a method parameter. There is an argument dependency from class `Security` to class `Capability`. In other words, `Capability` is visible to `Security`. The argument dependency is captured in the OIDs (see Figure 13-12) and can be added to the Design Class Description, it is not normally shown in the object model. The designers of the Library application have decided to explicitly show in the object model the other two argument dependencies: `LibraryView` to `LibraryItem` and `Capability` to `Threshold`. `Account` and `Security` also have views; however, they are not shown in this diagram because they belong to another subsystem.

References

- How to build a Design Object Model is documented in *Object-Oriented Modeling and Design*[Rumbaugh91a] and in various papers by J. Rumbaugh.
- Object visibility and class adornments are well documented in *Object-Oriented Analysis and Design with Applications* by G. Booch [Booch94].
- A good classification of visibility relationships can be found in *Object-Oriented Development: The Fusion Method* by D. Coleman, et al. [Coleman94].
- Design patterns are described in *Design Patterns: Elements of Reusable Object-Oriented Software* by E. Gamma, et al. [Gamma95].
- Proper inheritance is discussed in *C++ FAQs* by M. Cline and G. Lomow [Cline95].

Importance

The Design Object Model is absolutely essential during design. It is a fundamental way to document the static aspects of a system.

13.7 DESIGN SCENARIOS

Description

Analysis Scenarios define required systems behaviors at an abstract level; Design Scenarios define systems behaviors at a concrete level. In particular, they refer to design rather than analysis objects in describing their behavior, assumptions, and outcomes, and they include information about how to trigger the scenarios. In all other respects they are like Analysis Scenarios (see Section 11.4).

Design Scenarios can be used to define either systems behaviors or subsystems behaviors depending on whether they are used in the workbooks of systems or of subsystems.

A Design Scenario specifies a Design Object Interaction Diagram (OID) in the sense that it states formally what a Design OID must do, and how the behavior of the Design OID is triggered.

Purpose

Many projects require a complete set of functional specifications that define the externally-visible behaviors of a system—the *behaviors* and not just the interfaces that trigger those behaviors. Design Scenarios fill that role. If the Design Scenarios are part of a subsystem workbook then they will, of course, provide a functional specification of that subsystem.

If OIDs are used as a vehicle for discussing design alternatives, documenting design decisions, and validating Object Models, then the assumptions and outcomes of these OIDs must be defined precisely. Design Scenarios define the assumptions and outcomes of Design OIDs.

The approach to design advocated in this book is to transform the analysis models taking design considerations into account in an incremental manner. One way to do that is to take the Analysis OIDs, transform them incrementally and then transform their corresponding Analysis Scenarios (in order to create Design Scenarios that capture the design-level assumptions and outcomes of the new Design OIDs). An alternative is to take the Analysis Scenarios, transform them incrementally, and only then transform their corresponding Analysis OIDs, before performing internal design by continuing to transform the new Design OIDs. The former approach focuses on the development of *mechanisms* to drive design; the latter approach focuses on the development of *interfaces* to drive design. Both approaches are valid. Both have a requirement for Design Scenarios.

Participants

Design Scenarios are written by designers. If the Design Scenarios are part of a subsystem workbook then they will be written by the designers responsible for that subsystem.

Timing

Design Scenarios are written after the System Architecture and the Subsystem Model have been defined.

If Design OIDs are the key vehicle for design, then Design Scenarios will be written immediately after the Design OIDs as a way of capturing their assumptions and outcomes. An alternative is to write the Design Scenarios first, modify (or construct) the Design OIDs next, and then to proceed with design by continuing to transform the Design OIDs. A Design Scenario is the external abstractions of a system behavior, whereas a Design OID captures the internal details. The Design OID needs a Design Scenario to establish its function, and starting and ending states, but Design OIDs often discover missing assumptions and outcomes that need to be added to the Design Scenarios.

In any case, the Design Scenarios, like all other design work products, will have to be revisited whenever design issues force design rework.

Technique

Design OIDs are the key vehicle for performing design. If Design OIDs are written before Design Scenarios, then when a Design OID is produced, its corresponding Design Scenario is then created or updated. The new Design Scenario takes into account the previous version of the Design Scenario (or the corresponding Analysis Scenario if no Design Scenario previously existed), and the design issues that the Design OID was forced to address. Writing the Design Scenario involves reflecting these design issues in the description of the system behavior that the Design Scenario represents. This might, for example, involve replacing an abstractly expressed outcome by one that specifies the result of the scenario on newly-introduced design objects that the Design OID has just surfaced.

If Design Scenarios are written before Design OIDs, the change is that instead of reflecting a design decision that has been made while the Design OID was written, the design decision must be made now. Making that decision involves deciding how the Design Scenario's specification of system behavior has to change in order to accommodate the design issue under consideration. This might, for example, involve introducing new design objects to set up the scenario. This will involve (at least) adding or changing the assumptions of the Design Scenario to refer to the new objects.

A Design Scenario states the way in which the system behavior specified by the Design Scenario and depicted in the Design OID is to be triggered. A Design Scenario is usually triggered by a method call or a user action. In the case of a user action, user interfacing standards, styles, et cetera. obviously have to be taken into account. See Section 12.0, User Interface Model Work Products for further details. In the case of a method call, the first object shown in the OID is usually the one whose method is called to trigger the behavior. Only this object's class and method declaration are described as the Design Scenario trigger; not the object that invokes the method.

Strengths

Just as Analysis Scenarios are a very good way to formalize the functional *requirements* imposed on a system, so do Design Scenarios provide a functional *specification* of the system (or subsystem). The functional specification includes details that are omitted in the more abstract functional requirements. Documenting behaviors in this way supports the use of OIDs as a design vehicle. It enumerates all the variations that Design OIDs must deal with and establishes the starting and ending states for each OID. Precise subsystem functional specifications facilitate the parallel and independent development of subsystems. The enumeration of Design Scenarios provides another concrete metric for use in project estimation and tracking.

Weaknesses

Writing Design Scenarios is a lot of work. The temptation will exist to use only Design OIDs as a vehicle for design. Worse, you may be tempted to skip both Design Scenarios and Design OIDs due to the work involved. Without a formal language of design, however, it is difficult to control the process of building or maintaining large systems or components.

Notation

Similar to Analysis Scenarios (see Section 11.4) except that the scenario trigger must additionally be described. The following template may be used:

Table 13-4 (Page 1 of 2). *Example of a Design Scenario Notation Template.*

Scenario name ______________________
Trigger ______________________
Assumptions ______________________
____
Outcomes ______________________
____

Table 13-4 (Page 2 of 2). Example of a Design Scenario Notation Template.

Notes	

The trigger is either a method declaration or a description or a user action. User actions can be described by free-format text, perhaps including a reference to a User Interface Model work product.

Traceability

This work product has the following traceability:

Impacted by:

- Nonfunctional Requirements (p. 106)
- Issues (p. 176)
- Analysis Scenarios (p. 203)
- Design Guidelines (p. 253)
- System Architecture (p. 257)

Impacts:

- Design OIDs (p. 298)
- Design State Models (p. 306)
- Glossary (p. 355)

Advice and Guidance

Design Scenarios differ from Analysis Scenarios in that their assumptions and outcomes are specified in terms of starting and ending states of design objects versus analysis objects, that is, objects and states from the Design Object Model (DOM) vs. the Analysis Object Model (AOM). Often however, scenarios can be carried over from analysis to design when the referenced objects and states are carried over from AOM to DOM. In this case, Design OIDs can be started from Analysis Scenarios and OIDs. If and when new assumptions or outcomes are discovered during Design OID development, the Design Scenarios can be explicitly documented.

So, although Design Scenarios can be documented before starting on Design OIDs, developers often find it more efficient to discover and record them while developing the Design OIDs.

Care should be taken to distinguish between discoveries affecting simply the design versus those that truly affect the analysis work products. For example, if it is "discovered" that there is a "requirement" that the system must interface with some external software system, it should be questioned to determine whether it is a functional or a nonfunctional requirement.

- If it is determined that the requirement was functional and contractual, for instance "send the transaction and its completion status to the XYZ Audit System," then that should be recognized as an Actor affected by the system and be recorded in the Use Case Model, Analysis Scenarios, Analysis Object Model, State Models, and Class Descriptions.

- If, however, it is determined that the "requirement" was nonfunctional, for instance "use the internal transaction journalizer from the previous system," then that should be recognized as a design decision and result in Design Scenarios whose collaborations or outcomes reflect that design.

In the latter case, the analysis work products are not affected, because the function of the system, the nature of the problem, was not affected.

Verification

- Check that the Design Scenarios cover all anticipated changes to requirements. This might mean reworking the Analysis Scenarios but it is most easily checked by reference to the Design Scenarios.
- Check that all assumptions and outcomes refer to design objects, their states, and their attributes, and not to the analysis model.
- Check the completeness of Design Scenarios by checking how the opposite assumptions are dealt with (play "what if" games).
- Whenever an object state is mentioned in a Design Scenario assumption, check whether the other possible states of the object are adequately addressed by Design Scenarios.

Example(s)

This example refers to the image processing subsystem shown in Figure 13-7. One of the scenarios discovered at design time for this subsystem is that of *Invoke basic image processing function.* This Scenario is purely internal to the image processing architecture. In particular it is architecture-specific. The public interfaces to image processing applications do not know about the basic image processing functions. For both these reasons, the scenario cannot appear in the list of Analysis Scenarios: It is a subsystem scenario identified at design time as a result of projecting a system scenario onto the *Image processing* subsystem. Note that this scenario is used by *all* system scenarios that require usage of the built-in image processing functions.

Table 13-5 (Page 1 of 2). Example of a Design Scenario.

Scenario name	Invoke basic image processing function

Trigger	The scenario is triggered by a request to invoke a particular basic image processing function on the local node.
	`ISequence<IfResult> Image::perform(`
	`        IfParms legacyFunctionParameters )`

Assumptions	• The specified image processing function is implemented locally. • The images on which the function is to operate are currently stored in the image database, but not necessarily all locally.

Table 13-5 (Page 2 of 2). *Example of a Design Scenario.*

Outcomes	• The image processing function is invoked. • The result images (if any) are stored in the image database. • The return values (if any) are returned to the caller. These may include references to the ids of any result images.
Notes	The images specified as parameters to the function must be retrieved from the image database.

References

- [Jacobson92] has extensive discussion on Use Cases.
- [Spivey88] defines behavior in terms of assumptions and outcomes.

Importance

Design Scenarios are essential for creating Design Object Interaction Diagrams. Sometimes, Design Scenarios are created during Design OID construction as new assumptions and outcomes are discovered relating to design artifacts while working from Analysis OIDs or Scenarios. If you don't create Design Scenarios explicitly, you will have to create them before completing Design OIDs.

13.8 DESIGN OBJECT INTERACTION DIAGRAMS

Description

A Design Object Interaction Diagram (OID), used for *dynamic modeling* in the object-oriented design, is a graphical representation of object collaborations in a Design Scenario either derived from analysis or for design only, in terms of design objects and their interactions.

It is the result of transforming the corresponding Analysis Object Interaction Diagram, if one exists. The driving force behind the transformation is the System Architecture.

One major difference between a Design OID and its analysis equivalent (see Section 11.5) is its emphasis on those design classes intended for implementation. Some of these are directly derived from problem domain classes found in analysis, and others are invented or reused in design to provide control, interface, communication, distribution, and storage function.

The analysis-to-design transformation, of which writing Design OIDs is a vital part, is architecture-driven. For example, if a system has a client/server architecture, some analysis objects will be mapped into the design ones on the client side, some on the server side, and some mapped on both sides.

Purpose

The flexibility and the extensibility of a system depend on an adequate allocation of identified behaviors in terms of coupling and cohesion. A good allocation of behaviors leverages the reusability and interchangeability of objects. The strength of Design OIDs are their intuitive expressiveness for representing the end-to-end dynamic control and information flows among objects, under one specific System Architecture.

Modeling with Design OIDs is an effective driver of the development process. It is the main vehicle for allocating responsibilities to objects, discovering problems, holding design discussions, and considering design alternatives.

The dynamics of a System Architecture can be expressed well using Design OIDs. Knowledge of how objects interact is vital to the definition of an Architecture.

Dynamic modeling with Design OIDs is one of the most important steps. It directly impacts most implementation work products. Any design Issue should be resolved at the Design OID level.

Design OIDs are a means to:

- Decide and graphically depict the design objects' behaviors and responsibilities.
- Discover, present, and understand the function to be accomplished by each object.
- Visualize the distribution of system responsibilities among objects.
- Realize the conversion of classes of the Analysis Object Model into those of the Design Object Model.
- Identify, apply, and present patterns for structuring the design.

The following consequences could result from failing to conducting the Design OID modeling:

- Inconsistent use of design principles, design mechanisms, and the reuse of design patterns.
- Failure to discover opportunities for component and framework reuse.
- Inadequate identification of trade-offs between reusability, modifiability, and efficiency.
- Failure to preform commonality and variability analysis across subsystems.
- Incomplete refactoring the design for better resilience to change.
- Incomplete and erroneous specification of classes to be implemented.
- Potential miscommunication and misunderstanding among development team members.

Participants

Developing Design OIDs is the task of system architects, designers, and developers, led by a system architect. It is very important to have key programmers participate in this step, since they understand well the system constraints, environment, and language limitations. These factors must be reflected in the Design OIDs.

Timing

Design OIDs should be developed as soon as a primitive Design Object Model exists, and the system boundary and Architecture is defined in the design phase. The Design OIDs might be used as a vehicle to drive the development of the system structure. Both the Analysis Object Model and Analysis OIDs provide the basis for those at the design level. When a primitive Design Object Model is ready, it is time to develop the Design OIDs for the system objects. In the Design OID modeling process, more design objects are usually created.

In an iterative and incremental process, design modeling, especially Design OID modeling, should be carried out in each development cycle. Even in the implementation, Design OIDs are often referenced or modified for new design or architecture ideas.

Technique

The Design OIDs are developed by transforming those at the analysis level while considering the underlying Architecture, frameworks, design patterns, and system constraints. Usually, the contents of Design OIDs are more detailed, since they serve as part of the specification for subsequent coding. When an Architecture is in place, Design OIDs are used to assign responsibilities to design classes. For example, if the Model-View-Controller architecture is used, an analysis class is transformed into two design classes, a view class and a model class. The former captures the knowledge of end-user interface, and the latter possesses the business logic. When a persistent layer is used for a multilayer architecture, an analysis class can be transformed into another two design classes: a persistent class and a model class. The former is used only for handling the database interface. Developing a Design OID involves the following:

- Start with Use Cases, Analysis and Design Scenarios, and Analysis OIDs. Identify their corresponding Design OIDs and related design classes.
- Find the object responsibilities identified in the Analysis Object Model. Its corresponding design object may take over some of the responsibilities with additional ones from the System Architecture.
- Analyze the interactions of design objects. Examine the responsibilities for dependencies. For example, if an object is responsible for a specific action, but does not possess all the knowledge needed to accomplish that action, it must collaborate with objects defined in other classes that do possess the knowledge. Any two objects that have direct collaboration should have a directional association or a visibility dependency between their corresponding classes defined in the Design Object Model.
- Identify collaborations by asking the following question for each responsibility of every class: Are the class's objects capable of fulfilling this responsibility itself? If not, what does it need? From which other class can it acquire what it needs? Each responsibility that you decided to share between classes also represents a collaboration between their objects. Check what other objects need the result or information, and make sure that each object that needs the result collaborates with the one getting it.
- Make sure the parameters are well defined for messages including method return values, so that data flow and storage is explicitly documented.
- Add threading and control information, defined in the notation subsection, into your Design OID to show when, where, and how an object behavior is performed or waits for a response.

Strengths

Strengths of Design OIDs are their intuitiveness and expressiveness for representing dynamic interactions between objects, and System Architecture. It is a powerful and effective tool to enable developers discussing design Issues in depth before and during implementation.

Weaknesses

Limitations of Design OIDs are similar to those at the analysis level. A Design OID can only specify the execution of a system for one Scenario. A system can consist of hundreds of Scenarios under different assumptions. The amount of work required to create OIDs for each Design Scenario is daunting. Even if all Design OIDs are written, their number tends to defeat their objective of providing a clear understanding of system dynamics. The solution to this problem is to focus on Design OIDs that are effective in the sense that they impact either the System Architecture or system key function. Avoid those Design OIDs which do not contribute to the system understanding or design.

Notation

Design OID notation is mostly the same as the analysis counterpart (See *Notation* in Section 11.5, "Analysis Object Interaction Diagrams" on page 208).

The additional notation in Design OIDs is architectural, and includes:

- Focus-of-control
- Multitasking
- Process and subsystem boundary.

A focus-of-control shows whether an object is active. It is either in the state of execution, or in the blocking state waiting for a returning message. It is represented by a long rectangle box on the concerned object line time. In a complex system, process and subsystem boundaries as well as multithreading execution can be explicitly represented in the Design OIDs. The Design OID can demonstrate which process and which subsystem an object is in, how many threads the object has at one time point, and what activity one thread is engaged in. One notation format is shown in Figure 13-10.

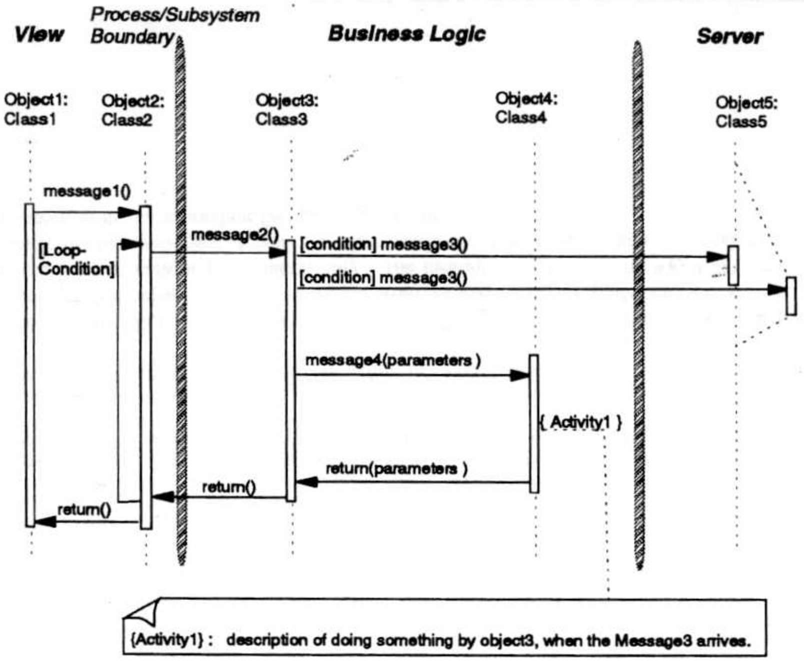


Figure 13-10. Format of a Design Object Interaction Diagram.

Traceability

This work product has the following traceability:

Impacted by:

- Nonfunctional Requirements (p. 106)
- Issues (p. 176)
- Analysis OIDs (p. 208)
- Screen Flows (p. 237)
- Screen Layouts (p. 242)
- Design Guidelines (p. 253)
- System Architecture (p. 257)
- Design Object Model (p. 281)
- Design Scenarios (p. 293)
- Design State Models (p. 306)

Impacts:

- Design Object Model (p. 281)
- Design State Models (p. 306)
- Design Class Description (p. 311)

Advice and Guidance

- Consistency should be maintained between the Design Object Model and Design OIDs. If object A sends a message to object B in a Design OID, either A's class has an association (including aggregation) directed to B's class, or there is an argument or typing dependency from A's class to B's class. The latter happens when B is passed to A through method parameters. Both cases should be reflected in the object model. Also, class attributes and operations referenced in the Design OIDs should appear in the Design Object Model.
- Due to the fact that a system may involve hundreds of Scenarios, it is suggested that only the major Scenarios are documents and developed into Design OIDs. Frequently, exceptional conditions are documented as notes at the bottom of each Design OID.
- The System Architecture impacts Design OIDs and should be expressed explicitly.
- Distribute responsibilities evenly among design objects.
- Avoid overly passive objects and overly active objects. The design may fall back to structured System Architectures.
- Determine the frequency of interobject message exchanges to identify any possible system bottleneck, to find ways to improve system performance.
- Check that every message arrow is named and has specified the required parameters.
- Check that the Scenario assumption/start-state is used/needed and that the expected outcome occurs.
- The integration test cases should be the direct result from the Design OID, while the system test cases should closely follow the Design Scenarios.

Verification

See the checklist for Analysis OIDs, Section 11.5, which is also appropriate here.

- Check that all navigations from object to object implied by a Design OID are in practice made possible by the appropriate attributes, parameters, and operations.
- Check that the frequency of interprocess communications is acceptable.
- Check the adequacy of all message parameter lists.
- Check whether the System Architecture is represented in Design OIDs.

Example(s)

We use the same example presented in the Analysis Object Interaction Diagram section, and the Design Scenario is derived from the one used in the corresponding Analysis OID (Section 11.5).

This example is about a library system in which users can access its library items. A user has an account associated with each library that has a unique level security capability. Each library item's security object will check whether the user is permitted to access the current library item.

- **Design Scenario**: A user wants to access a library item.
- **Trigger**: The scenario is triggered by the current user who clicks the mouse on the view icon of the library item to be accessed.
- **Assumption**: The user has already logged into the library system and his or her identity is established within the library.
- **Outcome**: A library item is permitted to be viewed if the user passes the security checking for this library item.
- **Description**: When a library item gets a request from a user for access, it first finds the account information from the library and passes the user's information to its own security object to check whether the user is permitted access. If the user has the proper authorization, the item will display its own content. Otherwise, request is rejected.

Figure 13-11. Example of a Design Object Interaction Scenario.

The Design OID example for this Scenario (Figure 13-12) shows a 2-tier System Architecture that separates the *view* classes from *business-logic* or *model* classes, so that view classes take care of presentation only. The messages are passed one way from view classes to the business-logic classes. In other words, the dependency is unidirectional, as is also shown in its corresponding Design Object model.

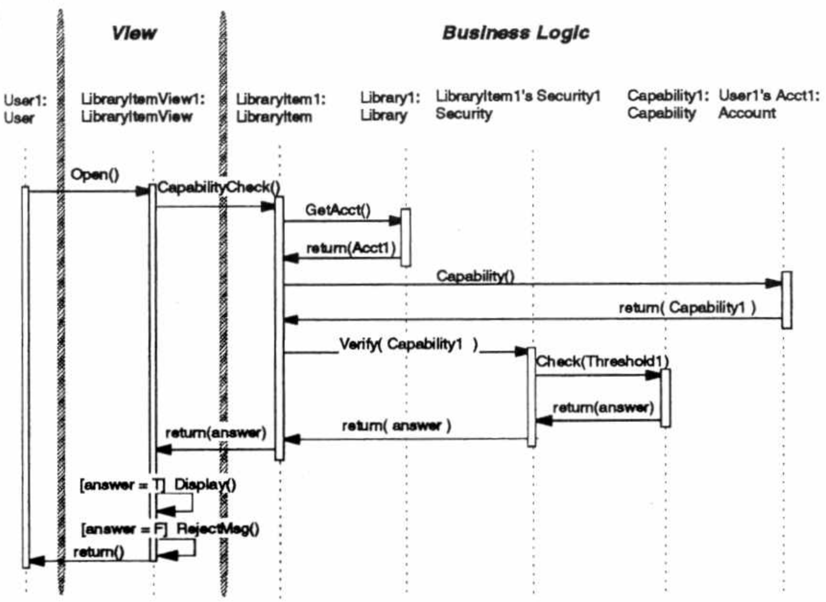


Figure 13-12. Example of a Design Object Interaction Diagram.

Comparing this Design OID (Figure 13-12) with the Analysis OID in Figure 11-11, one can find that this one is specified with much more design detail and extra design decisions. These design decisions reflect the Model-View-Controller architecture with separated concerns. In this architecture, the view objects are responsible for presenting the graphics and passing information from users to the model objects. The model objects are those possessing the business logic. One part of the business logic presented here is how to verify a user's security for the tasks she or he is allowed to perform.

References

Jacobson et al [Jacobson92] were the first group employing Use Cases and Object Interaction Diagrams (OID) in object-oriented dynamic modeling. Rumbaugh's OMT [Rumbaugh95b] and Booch's method [Booch94] have very similar work products. For example, OMT has OID-like constructs, called "Event Trace Diagrams" that are used for analysis. We define both Analysis OID and Design OIDs to be a far more expressive work product, by separating the notation for problem abstraction (Analysis OID) from the one for solution representation (Design OID), and adding concepts such as *conditions*, *loops*, *message synchronization*, and *process boundaries*. We also notice that the Unified Modeling Language [Booch96] begins to address issues related to conditions, processes, and message synchronization.

Importance

Dynamic modeling with Design OIDs is essential to specify the system object behaviors and it should be one of the major activities of the design phase. It is the key step to identify and specify methods for each class.

For one Scenario, several alternative Design OIDs may be produced. Each of these represents a different design solution. Modeling with Design OIDs forces solutions to be complete and brings to the surface Issues that must be resolved during system design. It is difficult to do object-oriented design without this form of system dynamics.

13.9 DESIGN STATE MODELS

Description

A Design State Model is a representation of the dynamic object behaviors for design classes. The Design State Model can use the same notation as one in the Analysis State Model (see Section 11.6), even if their contexts are different.

It should be emphasized that only a few key classes with strong state-dependent object behaviors need to be described by the State Model. Dynamic modeling with Design Object Interaction Diagrams (OIDs) is sufficient to express most object interaction and behavior.

For classes with complex states, design state modeling is the place to identify whether the *State* design pattern [Gamma95] can be used.

Purpose

A State Model can show the specification of object behaviors related to each class based on State Diagrams, Tables, or Matrixes. It is easy to understand the life cycle for one object through this mechanism. A State Model provides a convenient, visual means of understanding the life cycle of an object.

Design OIDs often provide clues for building a Design State Model. When a message is sent from one object to another, two things happen. First, the message-sending object will take an *action* in its particular state to accomplish this message sending. Second, an event is formed and arrives at the message-receiving object. This prompts the receiving object to perform some action and potentially change its state.

Participants

At the design level, State Models should be defined by system architects and designers. It is very important to have some key programmers participate this activity, to ensure that the design can be efficiently implemented. Architects should help verify whether the specification meets the requirements described in the State Models at the analysis level.

Timing

Design State Modeling is usually performed iteratively together with the development of a Design Object Model and Design OIDs.

Technique

During design and implementation phases, it may be necessary to develop State Models for certain classes whose dynamic behavior needs to be better understood. The difference between building a State Model for analysis vs. for design is that the design model can be impacted by System Architecture, frameworks, environment, and system constraints. The key objective here is to specify how to accomplish the responsibilities assigned to the object.

Strengths

The strength of State Models is their capability to describe the life cycles and state-dependent behavior of design objects clearly and efficiently. It is also a centralized place to model the behavior for one object.

Weaknesses

State models can be very trivial for some classes, and it is tedious to develop a state diagram for every class. A State Model describes a single class but not interactions between classes and their states. Thus State Models should be built selectively.

Notation

A State Model at the design level uses the same notation as an Analysis State Model (Section 11.6).

Traceability

This work product has the following traceability:

Impacted by:

- Nonfunctional Requirements (p. 106)
- Issues (p. 176)
- Analysis State Models (p. 219)
- Screen Flows (p. 237)
- Screen Layouts (p. 242)
- Design Guidelines (p. 253)
- System Architecture (p. 257)
- Design Object Model (p. 281)
- Design Scenarios (p. 293)
- Design OIDs (p. 298)

Impacts:

- Design OIDs (p. 298)
- Design Class Description (p. 311)
- Glossary (p. 355)

Advice and Guidance

It is wise to present the State Models in such detail that the tough design decisions can be easily discussed. Often, developers hesitate to make design decisions until the coding phase. This is a hidden danger to a project, since if a wrong decision is made by a particular programmer without being fully discussed at the design level, the system might not be built in its best form.

- Classes whose state dependencies have been explored in an Analysis State Model do not necessarily need to have their dynamics explored further in a Design State Model. Design State Models may be written either for newly introduced design classes or for already identified analysis classes as considered appropriate. The criterion is whether the writing of a Design State Model would add significantly to system understanding.

- Once a Design State Model is stable, ensure that it is consistent with the other design work products, in particular, the Design OIDs, the Design Object Model, and the Design Class Descriptions. How this will be done will be dependent on the way it is decided to implement the state-dependent behavior of the target class. In general, there are two ways: directly or by using the State design pattern (see [Gamma95]).

- If a Design State Model is to be implemented using the State design pattern, the following must be done:

  - Classes representing each of the states must be identified and defined. Each of these state classes will implement the services and data relevant to their substate; all services inapplicable to the state are also provided, but their implementation suggests that an exceptional situation has arisen. Default exceptional implementations can be inherited from a common state superclass. The common state superclass can also define any non-state-changing services that are common to all states.

  - All the services of all the states are part of the interface of the target class, and these are delegated to the current state. The target class has an attribute that maintains the current state.

  - State-changing logic is provided either by the target class or by the state classes, as considered appropriate. Do not create new state objects on each state change; reuse existing state objects where possible, both for efficiency and to implement state-specific data that must persist from state to state. If no state-specific data are required, then state objects may be instantiated using the *Singleton* design pattern.

  Implementations with state patterns expand the number of classes implemented slightly, but they separate concerns of different states and avoid repetitive code that tests state variables. A key advantage of the state pattern is that adding a new state or changing the details of an existing state, becomes a local and simple change.

- If a Design State Model is to be implemented directly, then all services and state variables are implemented directly by the target class. Each service must, if it is state-dependent, test the state variables and act accordingly. State transitions that are ruled out by the Design State Model are excluded by service *preconditions*, which must be identified and implemented for each service.

Verification

See the checklist for Analysis State Models, Section "Verification" on page 222, which is also appropriate here.

- Check for black holes: A group of states which, once entered, cannot be exited (as a group). Note that not all black holes suggest modeling errors.
- Check the state transition table. Even if the table is not explicitly produced, check each possible combination of states for a possible transition.
- Check for the absence of race conditions.
- Check semantics of absent transitions. An absent transition denotes a behavior that should not happen. Check that this prohibition is enforced. Absent transitions correspond to method preconditions; they should be documented as such.
- Check that each State Model contains initial and final states.
- Check that every noninitial state has at least one incoming transition.
- Check that every nonfinal state has at least one outgoing transition.
- Check that every transition is labeled with the event that triggers it.
- Check the events of all outgoing transitions from each node for completeness: Is another event possible in these circumstances?
- Check that every state is named.
- Check that the State Model identifies the class to which it refers.
- Check the consistency between Design OIDs and their related objects' State Models. If there is a message received by an object in a Design OID, the message should be

interpreted as an event occurring in that object's State Model. The object should experience a state change.

Example(s)

The TCP connection can be modeled using Design State Model. A TCP connection has three states, closed, listen, and established. With different events coming to different states, the connection reacts differently. The state diagram for a TCP connection is shown in Figure 13-13.

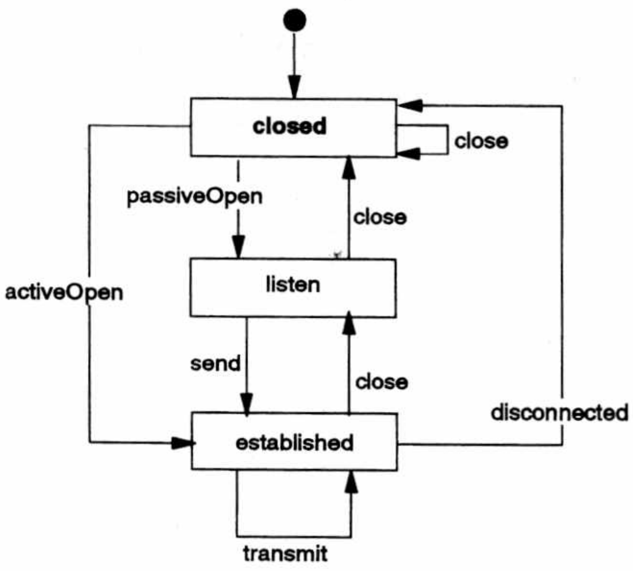


Figure 13-13. *Example of Design State Model.*

References

Sally Shlaer and Steve Mellor [Shlaer92] discuss using state modeling techniques for system analysis and design. Other references for this section are the same as those in the Analysis State Model (see Section 11.6).

Importance

Design State Models are optional but extremely useful for understanding the life cycle of objects with much state-dependent behavior.

13.10 DESIGN CLASS DESCRIPTIONS

Description

Design Class Descriptions are containers of all the information known about a class at the design level. Comments written about Analysis Class Descriptions in Section 11.7 are valid here too, and will not be repeated. Two points should be noted, however, when comparing analysis and Design Class Descriptions.

- They are different work products, although the Design Class Description will refer to its analysis counterpart if one exists. This is because the definition of the design class may be slightly different from that of the analysis class. For example, an analysis class may have been split into several design classes, perhaps to separate the model and view aspects of the class.
- The main audience for Design Class Descriptions is an implementation team. This must be borne in mind when planning the Design Class Description format.

Design Class Descriptions are likely to be generated automatically by the toolset responsible for maintaining the object model, the state models, et cetera.

Purpose

Design Class Descriptions are needed for the same reasons as Analysis Class Descriptions (See Section 11.7) and also serve the vital purpose of providing the starting point for class implementation work.

Participants

The designer who owns a class has the responsibility to maintain its Design Class Description.

Timing

Design Class Descriptions are opened as soon as the need for that design class is identified. The Class Description then serves as a repository for class oriented summary information as the design modeling activity proceeds.

Technique

As soon as the need for a particular class has been identified in a modeling session, a class description is opened for the class. It is a reasonable assumption that most of the analysis classes will become design classes, so an initial design activity might be the creation of Design Class Descriptions from Analysis Class Descriptions. The updating of class descriptions should be one of the activities performed after each design modeling session. Class Descriptions, Glossary entries (see Section 16.1), and the Design Object Model should be integrated, with a tool taking care of consistency and eliminating redundancy.

Use the Design Class Descriptions as a place to document any class or method design decisions that cannot be expressed more succinctly using one of the other design work products. Class invariants fall into this category, as do informal implementation notes on class design.

Strengths

Design Class Descriptions are vital starting points for the implementation of the classes. They organize all known information about each class and often record information that fits nowhere else.

Weaknesses

There is a danger that Class Descriptions may overlap in a redundant manner with other work products such as the Object Model and State Models. Redundancy may lead to inconsistency and a waste of resources. The solution is to ensure that the Class Descriptions are integrated into the chosen toolset so that information need to be entered and maintained only once. This should be considered to be an important criterion for tool selection.

Notation

Design Class Descriptions drive low-level design activity in the subsequent implementation phase. The format of the descriptions must, therefore, be appropriate to support that activity. Design Class Descriptions are necessarily sensitive to the target programming language. The class description template should, therefore, be tailored to that language. The following template is specific to C++; a Smalltalk template would differ only slightly.

A suitable Design Class Description template might be the following. There is no need for a "Name" slot in the template as each class description is expected to be an individual work product in its own right. All work products *inherit* the common work product attributes (defined in Section 8.1), including an identifying name.

Table 13-6 (Page 1 of 2). *Example of a Design Class Description Template.*

Description	______________________________
States	1.____________________________

Table 13-6 (Page 2 of 2). *Example of a Design Class Description Template.*

Relationships	1.____________________________ 2.____________________________
Public members	1.____________________________ 2.____________________________
Protected members	1.____________________________ 2.____________________________
Private members	1.____________________________ 2.____________________________
Notes	1.____________________________ 2.____________________________

Class invariants, if they are used, can be documented in an additional "Invariant" slot in the template.

At the design level it is assumed that defined operations will specify concrete types for parameters and results. Depending on the development method, the division of the design, the implementation phases, et cetera, it may be considered useful or necessary to provide pseudo-code for each operation. "Notes" can also include any free-format notes, hints, or implementation instructions.

Depending on tool support, the Design Class Descriptions should either include all design information relevant to a particular class, or the detailed information should be easily accessible from the class descriptions.

Traceability

This work product has the following traceability:

Impacted by:

- Issues (p. 176)
- Analysis Class Descriptions (p. 227)
- Design Guidelines (p. 253)
- System Architecture (p. 257)
- Design Object Model (p. 281)
- Design OIDs (p. 298)
- Design State Models (p. 306)

Impacts:

- Source Code (p. 334)

Advice and Guidance

- Use Design Class Descriptions as a place to hold method specifications or pseudo-code if appropriate.
- The degree of detail to be entered in Design Class Descriptions is dependent on the development process used. In general, however, the Design Class Descriptions should define the various interfaces of the classes, and provide enough information to allow

the class to be coded autonomously. Design Class Descriptions should be uniform in their level of detail.

- If provided automatically by tool support, do not copy information, for example associations, from other design work products into the Design Class Descriptions.
- A class should be defined independently of its subclasses. The subclass relationships are, however, included in the Design Class Descriptions in order to complete the picture of the class. Once again, good tool support is assumed in order to ensure that this transfer of information from the Design Object Model to the Design Class Descriptions is automatic.
- Tool support should include the visualization of inheritance class structures.

Verification

See the checklist for Analysis Class Descriptions, Section 11.7, which is also appropriate here.

- Check that each Design Class Description is an adequate basis for coding the class.
- Check that each Class Description is consistent with the agreed coding guidelines.
- Check that all methods are defined in adequate detail.
- Check that all attributes, parameters, and returns have types.
- Check that all key attributes and relations have been identified in the class descriptions.
- Check that all n-way associations have been (or will be) implemented by a collection class of the appropriate type.

Example(s)

The following example is for a system that was developed in C++.

Table 13-7 (Page 1 of 3). Example Showing Design Class Descriptions.

Name

BankAccount

Description

An agreement between a bank and a customer that enables the customer to deposit funds in the bank and, within certain limits, to withdraw funds.

States

- Active
- Withdrawn (substate of Active)
- InCredit (substate of Active)
- Suspended
- Open
- Closed (no account is ever destroyed)

Relationships

- Subclasses: CheckingAccount, SavingAccount
- Association: ownedBy(Customer)

Public members

Table 13-7 (Page 2 of 3). Example Showing Design Class Descriptions.

Several other public member functions, like constructors, destructor, uninteresting accessor functions, and others have been omitted here.

- `virtual bool isCreditWorthy( Amount creditAmount) const`
Check if this account can be granted a credit of the amount specified. Returns true if the account is creditworthy.

- `virtual TransactionResult modifyOverdraftLimit( Amount overDraftAmount )`
Modify the limit by which this account can be overdrawn. If the amount specified is positive, the limit is increased. If the amount specified is negative, the limit is decreased. The new limit cannot become negative, it will be zero in this case. Returns OK, if the overdraft limit was modified successfully, otherwise a TransactionResult that indicates the reason for failure.

- `virtual TransactionResult replaceCustomerDetails( CustomerDetails* customerDetails)`
Replace the details that are kept for the customer owning this account with the details specified. Returns OK, if the customer details were replaced successfully, otherwise a TransactionResult that indicates the reason for failure. The customer details are passed as pointer to also allow objects of concrete subclasses to be passed in without object slicing. Account is adopting the new customerDetails object and will delete it on its own destruction or if it is replaced again.

- `virtual const CustomerDetails& queryCustomerDetails() const`
Return the details that are kept for the customer owning this account. The customer details are returned by reference to also allow objects of concrete subclasses to be returned without object slicing.

- `virtual TransactionResult`
`addGuarantee( BankAccount& guarantor, Amount guaranteedAmount )`
Add to this account the guarantee provided by the specified account. This additional guarantor will provide a guarantee up to the amount specified. Returns OK, if the guarantee was added successfully, otherwise a TransactionResult indicates the reason for failure.

- `virtual TransactionResult removeGuarantee( BankAccount& guarantor)`
Remove the guarantee provided for this account by the account specified. Returns OK, if the guarantee was removed successfully, otherwise a TransactionResult indicates the reason for failure.

- `virtual TransactionResult addSignatory( const Customer& additionalSignatory )`
Add to this account the customer specified as additional person with the rights to sign. Returns OK, if the signatory was added successfully, otherwise a TransactionResult indicates the reason for failure.

- `virtual TransactionResult removeSignatory( const Customer& additionalSignatory )`
Remove the authorization to sign for this account for signatory specified. Returns OK, if the guarantee was removed successfully, otherwise a TransactionResult indicates the reason for failure.

- `virtual TransactionResult close() = 0;`
Close this account. Returns OK, if the account was closed successfully, otherwise a TransactionResult indicates the reason for failure. This is an abstract member function that needs to be overridden by the derived concrete classes, because closing an account must involve different actions depending on the concrete account type (checking, saving, et cetera).

- `virtual State state() const;`
Return the state of the account.

Protected members

Table 13-7 (Page 3 of 3). Example Showing Design Class Descriptions.

Most other set/get accessor functions for private attributes have been omitted here ...

- `virtual void state( State newState)`
  Set the state of the account.

Private members

- AccountCode accountCode
- Name name
- Address address
- Date creationDate
- AccountBalance currentBalance
- CreditArrangement overdraftFacilities

Notes

- **BankAccount is an abstract base class.**

References

See [Wirfs-Brock90], [Rumbaugh91a], and [Booch94] for examples of Class Description formats.

Importance

Design Class Descriptions are essential as a starting point for class implementations.

13.11 REJECTED DESIGN ALTERNATIVES

Description

Rejected Design Alternatives are those design decisions that were considered but, for one reason or another, were rejected. The mainstream design work products document the decisions that *were* taken; this work product documents those that were *not* taken.

Purpose

Documenting the Rejected Design Alternatives is very important, because without them a design is disconnected from its history and from the effort that was required to construct the design. The practical implication of not documenting rejected alternatives is that this information will be lost, either because the members of the design team have changed, or because the details of the issues have simply been forgotten. This matters because design decisions need to be reviewed from time to time. Perhaps the pattern of communication between objects, as shown in Object Interaction Diagrams (OIDs) for example, has become unbalanced, and a design review has suggested that alternatives be considered. Perhaps the project requirements have changed, a not uncommon occurrence, and design trade-offs have to be evaluated anew. Sometimes the need to review a design decision arises informally: A designer wants to be convinced that a decision is correct. In all these cases, if the history of a design decision has been lost, the various alternatives must be rediscovered and reevaluated. This duplication of effort is, of course, an undesirable overhead.

The effort required to consider design alternatives seriously affects the success of an iterative and incremental development process. If considering alternatives, and hence switching to alternatives, is too expensive then design decisions will tend to carry themselves along by their own momentum: "We don't know why a decision was made but it would be too time-consuming to find out." This is not intended to imply that actually making a design switch is trivial or cheap, but that without access to the history of a design decision the switch frequently won't even be considered.

Documenting design alternatives also improves communications between members of design teams. Questions of why something is the way it is can often be answered by a referral to the documented alternatives.

This problem of documenting design decisions can be tackled in one of two ways: one positive and one negative. The positive approach is to capture each design decision that is made and to document it together with its alternatives and their trade-offs. Desirable though this approach is in theory, it is, in practice, very difficult, because design decisions are being made continuously during the design process. What should be documented? How many design decisions does a single OID embody? Many: Design is a matter of continual decision-making, sometimes major, frequently minor.

The negative approach is to capture design alternatives that were rejected for some reason: to state them and to explain why they were rejected. In practice this turns out to be more feasible than a positive documentation approach. The process of design generally flows forward, but occasionally there are problems. Progress halts when alternatives are considered, weighed, and a decision made, perhaps after constructing some prototypes to evaluate the alternatives. This is the time to document what has happened: The facts are recorded as a set of Rejected Design Alternatives. By recording only negative decisions, there is an automatic filtering of the mass of minor decisions that were "obvious" and that would simply clutter the workbook and reduce progress if they were documented. The minor decisions are not recorded because they did not result in any serious consideration of alternatives. If positive decisions are recorded, it is in practice difficult to perform this filtering.

The reason why remembering the history of design decisions is more important than remembering that of analysis decisions is that analysis decisions are usually made in order to capture domain knowledge, or for reasons of modeling "goodness." When performing analysis, we deliberately ignore factors of efficiency, System Architecture, the Target Environment, et cetera. Analysis decisions are, therefore, much less exposed than design decisions to reconsideration. The exception is the case of a requirements change or a change in domain understanding, but in these cases the reworking of the analysis will reflect the new requirements, and alternative ways of modeling the old requirements will be of secondary importance. This is not to say that recording rejected analysis decisions is not sometimes useful, but it is not as vital as it is for design.

Participants

The team leaders, designers and architects who own the design work products are responsible for recording and documenting the alternatives that were considered and that were ultimately rejected.

Timing

Rejected Design Alternatives are documented as the design proceeds, as soon as practically possible after the design decision has been made. Design details are often highly complex and the facts will be forgotten if designers are told to document rejected alternatives only after the design has been completed.

Technique

The common work product structure, see Section 8.1, includes an Issues attribute. The raising of an issue often triggers the consideration of design alternatives. At some point the issue will be resolved by the taking of a decision and the rejection of alternatives. The rejected alternatives are then documented and cross-references made between the relevant work products, the Rejected Alternatives, and the issues that generated them. Not all rejected decisions stem from matters raised as explicit Issues however.

Strengths

Recording Rejected Design Alternatives enables a design team to backtrack without the overhead of rediscovering and reevaluating design alternatives.

Weaknesses

Recording Rejected Design Alternatives imposes an overhead on the project that must be recognized and budgeted. This work is an investment in future rework and will not pay in the very short term. It can be thought of as a form of reuse: reuse of design decisions. The benefits of any form of reuse are often difficult to quantify and sometimes difficult to justify.

Notation

The format in which Rejected Design Alternatives can be documented is very similar to that of design patterns [Gamma95].

Table 13-8. Example of a Rejected Design Alternative Template.

Description ______________________

Context ______________________

Assumptions ______________________

Alternatives 1.______________________
2.______________________

Considerations ______________________
____

Decision ______________________

The design alternatives documented in the above template can be described using either text or OIDs.

Traceability

This work product has the following traceability:

Impacted by:
- Issues (p. 176)
- System Architecture (p. 257)

Impacts:
- Historical Work Products (p. 359)

Advice and Guidance

- Record Rejected Design Alternatives as soon as possible.
- Include or refer to rejected Scenarios, OIDs, Object Models, et cetera. if these are relevant.
- The documentation should focus on enabling another designer in the future to return to this design decision and to reevaluate the alternatives, possibly with a different set of assumptions.
- A work product design review should briefly review the rejected alternatives that relate to the accepted design.
- The documentation should be relatively self-contained. References to other work products are fine, but the Issue, its context, and its assumptions should be understandable by someone reading this documentation alone.

Verification

- Review documented decisions to check that they are still reasonable.

Example(s)

In an application to configure distributed computer systems in a centralized manner, the following design issue arose:

Table 13-9. Example of Rejected Design Alternative.

Description	Representation of collected configuration data.
Context	Actual configuration data can be collected from the target nodes, and defined configuration data can be specified by a user. The actual and defined configuration object models have a common structure, as both represent the various alternative ways in which this operating system can be configured. The question is whether the same object model should be used for both actual and defined cases or whether two separate object models should be used.
Assumptions	1. While there are operations that are common to both actual and defined data, there are operations that are only applicable to one. For example, it is only valid to check defined data (against actual nodes), and it is only valid to validate actual data (for consistency). 2. There is an import method that turns valid actual data into defined data.
Alternatives	1. Represent actual and defined configuration data using two separate object models. This permits each of these object models to be as simple as possible, with only those alternatives and operations that are valid. 2. Represent actual and defined configuration data using the same object model. Where necessary, configuration objects use the state pattern to distinguish between the two kinds of representation. The subclasses of the state objects have interfaces that permit only operations appropriate to that state. The state pattern can also be used to distinguish between validated and unvalidated actual data.
Considerations	1. The first alternative involves simpler structures and fewer classes. 2. The replication of structure implicit in the first alternative will make the design less modular and more difficult to maintain. 3. The use of the state pattern in the second alternative enables the design to deal with other issues (distinction between validated and unvalidated data) by means of the same mechanism.
Decision	The second alternative was selected.

References

The format for design patterns that also works for Rejected Design Alternatives can be found, along with a complete discussion of design patterns in [Gamma95].

Importance

Recording Rejected Design Alternatives is optional, but we recommend that it be done as it makes revisiting of design decisions much easier.

14.0 Implementation Work Products

The purpose of the implementation phase is to construct the deliverable components of the product based on the plans detailed in the design work products. For software products, the most obvious deliverables are the executable form of the software (a.k.a. binaries) and the User Support Materials (a.k.a. documentation). But it is not a simple task to transform design work products into those deliverables. Since the final step in producing software and documentation is an automated one (e.g., compilation, formatting), we need to focus on the development of the intermediate work products that lead up to that step.

The implementation section of the project workbook consists of the following work products:

- Coding Guidelines
- Physical Packaging Plan
- Development Environment
- Source Code
- User Support Materials

These work products all "inherit" the common work product attributes described in Section 8.1 and have specialized attributes of their own. The work products and their specialized content are defined and commented on in the following sections, but let's briefly describe them here.

Before dispatching scores of programmers to generate millions of lines of code whose quality affects your company's health and whose maintenance defines your future liability, it is a good idea to have some Coding Guidelines. The nature and value of these guidelines is best understood by veteran programmers who have endeavored to extend or maintain someone else's source code.

Since source code files, libraries, and executables are organized and managed in the Development Environment for ease of development and testing rather than for ease of installation and use by a product user, a plan is needed that shows how all the deliverable components will be collected and packaged for delivery and installation. This is known as the Physical Packaging Plan.

The collection of tools, processes, conventions, and organizational infrastructure that a programming shop needs to establish to carry out the implementation of software defines a very complex "environment." A significant part of that involves setting up and running the configuration management and version control system. The Development Environment is often called "that well-oiled machine" because it is big, important, and requires a lot of care to set up and keep running. The Development Environment work product attempts to define that environment.

2

Evaluating a Software Architecture

> Marry your architecture in haste and you can repent in leisure.
> —Barry Boehm
> from a keynote address: *And Very Few Lead Bullets Either*

How can you be sure whether the architecture chosen for your software is the right one? How can you be sure that it won't lead to calamity but instead will pave the way through a smooth development and successful product?

It's not an easy question, and a lot rides on the outcome. The foundation for any software system is its architecture. The architecture will allow or preclude just about all of a system's quality attributes. Modifiability, performance, security, availability, reliability—all of these are precast once the architecture is laid down. No amount of tuning or clever implementation tricks will wring any of these qualities out of a poorly architected system.

To put it bluntly, an architecture is a bet, a wager on the success of a system. Wouldn't it be nice to know in advance if you've placed your bet on a winner, as opposed to waiting until the system is mostly completed before knowing whether it will meet its requirements or not? If you're buying a system or paying for its development, wouldn't you like to have some assurance that it's started off down the right path? If you're the architect yourself, wouldn't you like to have a good way to validate your intuitions and experience, so that you can sleep at night knowing that the trust placed in your design is well founded?

Until recently, there were almost no methods of general utility to validate a software architecture. If performed at all, the approaches were spotty, ad hoc, and not repeatable. Because of that, they weren't particularly trustworthy. We can do better than that.

This is a guidebook of software architecture evaluation. It is built around a suite of three methods, all developed at the Software Engineering Institute, that can be applied to any software-intensive system:

- ATAM: Architecture Tradeoff Analysis Method
- SAAM: Software Architecture Analysis Method
- ARID: Active Reviews for Intermediate Designs

The methods as a group have a solid pedigree, having been applied for years on dozens of projects of all sizes and in a wide variety of domains. With these methods, the time has come to include software architecture evaluation as a standard step of any development paradigm. Evaluations represent a wise risk-mitigation effort and are relatively inexpensive. They pay for themselves in terms of costly errors and sleepless nights avoided.

Whereas the previous chapter introduced the concept of software architecture, this chapter lays the conceptual groundwork for architectural evaluation. It defines what we mean by software architecture and explains the kinds of properties for which an architecture can (and cannot) be evaluated.

First, let's restate what it is we're evaluating:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. [Bass 98]*

By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract some information about the system (otherwise there is no point looking at the architecture—we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction (see the sidebar What's Architectural?).

The architecture defines the components (such as modules, objects, processes, subsystems, compilation units, and so forth) and the relevant relations (such as calls, sends-data-to, synchronizes-with, uses, depends-on, instantiates, and many more) among them. The architecture is the result of early design decisions that are necessary before a group of people can collaboratively build a software system. The larger or more distributed the group, the more vital the architecture is (and the group doesn't have to be very large before the architecture is vital).

One of the insights about architecture from Chapter 1 that you must fully embrace before you can understand architecture evaluation is this:

> *Architectures allow or preclude nearly all of the system's quality attributes.*

This leads to the most fundamental truth about architecture evaluation: If architectural decisions determine a system's quality attributes, then it is possible to evaluate architectural decisions with respect to their impact on those attributes.

What's Architectural?

Sooner or later everyone asks the question: "What's architectural?" Some people ask out of intellectual curiosity, but people who are evaluating architectures have a pressing need to understand what information is in and out of their realm of concern. Maybe you didn't ask the question exactly that way. Perhaps you asked it in one of the following ways:

- What is the difference between an architecture and a high-level design?
- Are details such as priorities of processes architectural?
- Why should implementation considerations such as buffer overflows be treated as architectural?
- Are interfaces to components part of the architecture?
- If I have class diagrams, do I need anything else?
- Is architecture concerned with run-time behavior or static structure?
- Is the operating system part of the architecture? Is the programming language?
- If I'm constrained to use a particular commercial product, is that architectural? If I'm free to choose from a wide range of commercial products, is that architectural?

Let's think about this in two ways.

First, consider the definition of architecture that we quoted in Chapter 1 of this book. Paraphrasing: A software architecture concerns the gross organization of a system described in terms of its components, their externally visible properties, and the relationships among them. True enough, but it fails to explicitly address the notion of context. If the scope of my concern is confined to a subsystem within a system that is part of a system of systems, then what I consider to be architectural will be different than what the architect of the system of systems considers to be architectural. Therefore, context influences what's architectural.

Second, let's ask, what is *not* architectural? It has been said that algorithms are not architectural; data structures are not architectural; details of data flow are not architectural. Well, again these statements are only partially true. Some properties of algorithms, such as their complexity, might have a dramatic effect on performance. Some properties of data structures, such as whether they need to support concurrent access, directly impact performance and reliability. Some of the details of data flow, such as how components depend on specific message types or which components are allowed access to which data types, impact modifiability and security, respectively.

So is there a principle that we can use in determining what is architectural? Let's appeal to what architecture is used for to formulate our principle. Our criterion for something to be architectural is this: It must be a component, or a relationship between components, or a property (of components or relationships) *that needs to be externally visible* in order to reason about the ability of the system to meet its quality requirements or to support decomposition of the system into independently implementable pieces. Here are some corollaries of this principle:

- *Architecture describes what is in your system.* When you have determined your context, you have determined a boundary that describes what is in and what is out of your system (which might be someone else's subsystem). Architecture describes the part that is in.
- *An architecture is an abstract depiction of your system.* The information in an architecture is the most abstract and yet meaningful depiction of that aspect of the system. Given your architectural specification, there should not be a need for a more abstract description. That is not to say that all aspects of architecture are abstract, nor is it to say that there is an abstraction threshold that needs to be exceeded before a piece of design information can be considered architectural. You shouldn't worry if your architecture encroaches on what others might consider to be a more detailed design.
- *What's architectural should be critical for reasoning about critical requirements.* The architecture bridges the gap between requirements and the rest of the design. If you feel that some information is critical for reasoning about how your system will meet its requirements then it is architectural. You, as the architect, are the best judge. On the other hand, if you can eliminate some details and still compose a forceful argument through models, simulation, walk-throughs, and so on about how your architecture will satisfy key requirements then those details do not belong. However, if you put too much detail into your architecture then it might not satisfy the next principle.
- *An architectural specification needs to be graspable.* The whole point of a gross-level system depiction is that you can understand it and reason about it. Too much detail will defeat this purpose.
- *An architecture is constraining.* It imposes requirements on all lower-level design specifications. I like to distinguish between when a decision is made and when it is realized. For example, I might determine a process prioritization strategy, a component redundancy strategy, or a set of encapsulation rules when designing an architecture; but I might not actually make priority assignments, determine the algorithm for a redundant calculation, or specify the details of an interface until much later.

In a nutshell:

> *To be architectural is to be the most abstract depiction of the system that enables reasoning about critical requirements and constrains all subsequent refinements.*

If it sounds like finding all those aspects of your system that are architectural is difficult, that is true. It is unlikely that you will discover everything that is architectural up front, nor should you try. An architectural specification will evolve over time as you continually apply these principles in determining what's architectural.

—MHK

2.1 Why Evaluate an Architecture?

The earlier you find a problem in a software project, the better off you are. The cost to fix an error found during requirements or early design phases is orders of magnitudes less to correct than the same error found during testing. Architecture is the product of the early design phase, and its effect on the system and the project is profound.

An unsuitable architecture will precipitate disaster on a project. Performance goals will not be met. Security goals will fall by the wayside. The customer will grow impatient because the right functionality is not available, and the system is too hard to change to add it. Schedules and budgets will be blown out of the water as the team scrambles to back-fit and hack their way through the problems. Months or years later, changes that could have been anticipated and planned for will be rejected because they are too costly. Plagues and pestilence cannot be too far behind.

Architecture also determines the structure of the project: configuration control libraries, schedules and budgets, performance goals, team structure, documentation organization, and testing and maintenance activities all are organized around the architecture. If it changes midstream because of some deficiency discovered late, the entire project can be thrown into chaos. It is much better to change the architecture before it has been frozen into existence by the establishment of downstream artifacts based on it.

Architecture evaluation is a cheap way to avoid disaster. The methods in this book are meant to be applied while the architecture is a paper specification (of course, they can be applied later as well), and so they involve running a series of simple thought experiments. They each require assembling relevant stakeholders for a structured session of brainstorming, presentation, and analysis. All told, the average architecture evaluation adds no more than a few days to the project schedule.

To put it another way, if you were building a house, you wouldn't think of proceeding without carefully looking at the blueprints before construction began. You would happily spend the small amount of extra time because you know it's much better to discover a missing bedroom while the architecture is just a blueprint, rather than on moving day.

2.2 When Can an Architecture Be Evaluated?

The classical application of architecture evaluation occurs when the architecture has been specified but before implementation has begun. Users of iterative or incremental life-cycle models can evaluate the architectural decisions made during the most recent cycle. However, one of the appealing aspects of architecture evaluation is that it can be applied at any stage of an architecture's lifetime, and there are two useful variations from the classical: early and late.

Early. Evaluation need not wait until an architecture is fully specified. It can be used at any stage in the architecture creation process to examine those architectural decisions already made and choose among architectural options that are pending. That is, it is equally adept at evaluating architectural decisions that have already been made and those that are being considered.

Of course, the completeness and fidelity of the evaluation will be a direct function of the completeness and fidelity of the architectural description brought to the table by the architect. And in practice, the expense and logistical burden of convening a full-blown evaluation is seldom undertaken when unwarranted by the state of the architecture. It is just not going to be very rewarding to assemble a dozen or two stakeholders and analysts to evaluate the architect's early back-of-the-napkin sketches, even though such sketches will in fact reveal a number of significant architecture paths chosen and paths not taken.

Some organizations recommend what they call a *discovery review*, which is a very early mini-evaluation whose purpose is as much to iron out and prioritize troublesome requirements as analyzing whatever "proto-architecture" may have been crafted by that point. For a discovery review, the stakeholder group is smaller but must include people empowered to make requirements decisions. The purpose of this meeting is to raise any concerns that the architect may have about the feasibility of *any* architecture to meet the combined quality and behavioral requirements that are being levied while there is still time to relax the most troubling or least important ones. The output of a discovery review is a much stronger set of requirements and an initial approach to satisfying them. That approach, when fleshed out, can be the subject of a full evaluation later.

We do not cover discovery reviews in detail because they are a straightforward variation of an architecture evaluation. If you hold a discovery review, make sure to

- Hold it before the requirements are frozen and when the architect has a good idea about how to approach the problem
- Include in the stakeholder group someone empowered to make requirements decisions
- Include a prioritized set of requirements in the output, in case there is no apparent way to meet all of them

Finally, in a discovery review, remember the words of the gifted aircraft designer Willy Messerschmitt, himself no stranger to the burden of requirements, who said:

> *You can have any combination of features the Air Ministry desires, so long as you do not also require that the resulting airplane fly.*

Late. The second variation takes place when not only the architecture is nailed down but the implementation is complete as well. This case occurs when an organization inherits some sort of legacy system. Perhaps it has been purchased on the open market, or perhaps it is being excavated from the organization's own archives. The techniques for evaluating a legacy architecture are the same as those for one that is newborn. An evaluation is a useful thing to do because it will help the new owners understand the legacy system, and let them know whether the system can be counted on to meet its quality and behavioral requirements.

In general, when can an architectural evaluation be held? As soon as there is enough of an architecture to justify it. Different organizations may measure that justification differently, but a good rule of thumb is this: Hold an evaluation when development teams start to make decisions that depend on the architecture and the cost of undoing those decisions would outweigh the cost of holding an evaluation.

2.3 Who's Involved?

There are two groups of people involved in an architecture evaluation.

1. *Evaluation team.* These are the people who will conduct the evaluation and perform the analysis. The team members and their precise roles will be defined later, but for now simply realize that they represent one of the classes of participants.
2. *Stakeholders.* Stakeholders are people who have a vested interest in the architecture and the system that will be built from it. The three evaluation methods in this book all use stakeholders to articulate the specific requirements that are levied on the architecture, above and beyond the requirements that state what functionality the system is supposed to exhibit. Some, but not all, of the stakeholders will be members of the development team: coders, integrators, testers, maintainers, and so forth.

   A special kind of stakeholder is a project decision maker. These are people who are interested in the outcome of the evaluation and have the power to make decisions that affect the future of the project. They include the architect, the designers of components, and the project's management. Management will have to make decisions about how to respond to the issues raised by the evaluation. In some settings (particularly government acquisitions), the customer or sponsor may be a project decision maker as well.

   Whereas an arbitrary stakeholder says what he or she wants to be true about the architecture, a decision maker has the power to expend resources to *make* it true. So a project manager might say (as a stakeholder), "I would like the architecture to be reusable on a related project that I'm managing," but as a decision maker he or she might say, "I see that the changes you've identified as necessary to reuse this architecture on my other project are too expensive, and I won't pay for them." Another difference is that a project decision maker has the power to speak authoritatively for the project, and some of the steps of the ATAM method, for example, ask them to do precisely that. A garden-variety stakeholder, on the other hand, can only hope to influence (but not control) the project. For more on stakeholders, see the sidebar Stakeholders on page 63 in Chapter 3.

The client for an architecture evaluation will usually be a project decision maker, with a vested interest in the outcome of the evaluation and holding some power over the project.

Sometimes the evaluation team is drawn from the project staff, in which case they are also stakeholders. This is not recommended because they will lack the objectivity to view the architecture in a dispassionate way.

2.4 What Result Does an Architecture Evaluation Produce?

In concrete terms, an architecture evaluation produces a report, the form and content of which vary according to the method used. Primarily, though, an architecture evaluation produces information. In particular, it produces answers to two kinds of questions.

1. Is this architecture suitable for the system for which it was designed?
2. Which of two or more competing architectures is the most suitable one for the system at hand?

Suitability for a given task, then, is what we seek to investigate. We say that an architecture is suitable if it meets two criteria.

1. The system that results from it will meet its quality goals. That is, the system will run predictably and fast enough to meet its performance (timing) requirements. It will be modifiable in planned ways. It will meet its security constraints. It will provide the required behavioral function. Not every quality property of a system is a direct result of its architecture, but many are, and for those that are, the architecture is suitable if it provides the blueprint for building a system that achieves those properties.
2. The system can be built using the resources at hand: the staff, the budget, the legacy software (if any), and the time allotted before delivery. That is, the architecture is *buildable.*

This concept of suitability will set the stage for all of the material that follows. It has a couple of important implications. First, suitability is only relevant in the context of specific (and specifically articulated) goals for the architecture and the system it spawns. An architecture designed with high-speed performance as the primary design goal might lead to a system that runs like the wind but requires hordes of programmers working for months to make any kind of modification to it. If modifiability were more important than performance *for that system*, then that architecture would be unsuitable *for that system* (but might be just the ticket for another one).

In *Alice in Wonderland,* Alice encounters the Cheshire Cat and asks for directions. The cat responds that it depends upon where she wishes to go. Alice says she doesn't know, whereupon the cat tells her it doesn't matter which way she walks. So

> *If the sponsor of a system cannot tell you what any of the quality goals are for the system, then any architecture will do.*

An overarching part of an architecture evaluation is to capture and prioritize specific goals that the architecture must meet in order to be considered

Why Should I Believe You?

Frequently when we embark on an evaluation we are outsiders. We have been called in by a project leader or a manager or a customer to evaluate a project. Perhaps this is seen as an audit, or perhaps it is just part of an attempt to improve an organization's software engineering practice. Whatever the reason, unless the evaluation is part of a long-term relationship, we typically don't personally know the architect, or we don't know the major stakeholders.

Sometimes this distance is not a problem—the stakeholders are receptive and enthusiastic, eager to learn and to improve their architecture. But on other occasions we meet with resistance and perhaps even fear. The major players sit there with their arms folded across their chests, clearly annoyed that they have been taken away from their *real* work, that of architecting, to pursue this silly management-directed evaluation. At other times the stakeholders are friendly and even receptive, but they are skeptical. After all, they are the experts in their domains and they have been working in the area, and maybe even on this system, for years.

In either case their attitudes, whether friendly or unfriendly, indicate a substantial amount of skepticism over the prospect that the evaluation can actually help. They are in effect saying, "What could a bunch of outsiders possibly have to tell us about *our* system that we don't already know?" You will probably have to face this kind of opposition or resistance at some point in your tenure as an architecture evaluator.

There are two things that you need to know and do to counteract this opposition. First of all, you need to counteract the fear. So keep calm. If you are friendly and let them know that the point of the meeting is to learn about and improve the architecture (rather than pointing a finger of blame) then you will find that resistance melts away quickly. Most people actually enjoy the evaluation process and see the benefits very quickly. Second, you need to counteract the skepticism. Of course they are the experts in the domain. You know this and they know this, and you should acknowledge this up front. But you are the architecture and quality attribute expert. No matter what the domain, architectural approaches for dealing with and analyzing quality attributes don't vary much. There are relatively few ways to approach performance or availability or security on an architectural level. As an experienced evaluator (and with the help of the insight from the quality attribute communities) you have seen these before, and they don't change much from domain to domain.

Furthermore, as an outsider you bring a "fresh set of eyes," and this alone can often bring new insights into a project. Finally, you are following a process that has been refined over dozens of evaluations covering dozens of different domains. It has been refined to make use of the expertise of many people, to elicit, document, and cross-check quality attribute requirements and architectural information. This alone will bring benefit to your project—we have seen it over and over again. The process works!

—*RK*

suitable. In a perfect world, these would all be captured in a requirements document, but this notion fails for two reasons: (1) Complete and up-to-date requirements documents don't always exist, and (2) requirements documents express the requirements for a *system*. There are additional requirements levied on an architecture besides just enabling the system's requirements to be met. (Buildability is an example.)

The second implication of evaluating for suitability is that the answer that comes out of the evaluation is not going to be the sort of scalar result you may be used to when evaluating other kinds of software artifacts. Unlike code metrics, for example, in which the answer might be 7.2 and anything over 6.5 is deemed unacceptable, an architecture evaluation is going to produce a more thoughtful result.

We are not interested in precisely characterizing any quality attribute (using measures such as mean time to failure or end-to-end average latency). That would be pointless at an early stage of design because the actual parameters that determine these values (such as the actual execution time of a component) are often implementation dependent. What we are interested in doing—in the spirit of a risk-mitigation activity—is learning where an attribute of interest is affected by architectural design decisions, so that we can reason carefully about those decisions, model them more completely in subsequent analyses, and devote more of our design, analysis, and prototyping energies to such decisions.

An architectural evaluation will tell you that the architecture has been found suitable with respect to one set of goals and problematic with respect to another set of goals. Sometimes the goals will be in conflict with each other, or at the very least, some goals will be more important than other ones. And so the manager of the project will have a decision to make if the architecture evaluates well in some areas and not so well in others. Can the manager live with the areas of weakness? Can the architecture be strengthened in those areas? Or is it time for a wholesale restart? The evaluation will help reveal where an architecture is weak, but weighing the cost against benefit to the project of strengthening the architecture is solely a function of project context and is in the realm of management. So

> *An architecture evaluation doesn't tell you "yes" or "no," "good" or "bad," or "6.75 out of 10." It tells you where you are at risk.*

Architecture evaluation can be applied to a single architecture or to a group of competing architectures. In the latter case, it can reveal the strengths and weaknesses of each one. Of course, you can bet that no architecture will evaluate better than all others in all areas. Instead, one will outperform others in some areas but underperform in other areas. The evaluation will first identify what the areas of interest are and then highlight the strengths and weaknesses of each architecture in those areas. Management must decide which (if any) of the competing architectures should be selected or improved or whether none of the candidates is acceptable and a new architecture should be designed.[1]

2.5 For What Qualities Can We Evaluate an Architecture?

In this section, we say more precisely what suitability means. It isn't quite true that we can tell from looking at an architecture whether the ensuing system will meet *all* of its quality goals. For one thing, an implementation might diverge from the architectural plan in ways that subvert the quality plans. But for another, architecture does not strictly determine all of a system's qualities.

Usability is a good example. Usability is the measure of a user's ability to utilize a system effectively. Usability is an important quality goal for many systems, but usability is largely a function of the user interface. In modern systems design, particular aspects of the user interface tend to be encapsulated within small areas of the architecture. Getting data to and from the user interface and making it flow around the system so that the necessary work is done to support the user is certainly an architectural issue, as is the ability to change the user interface should that be required. However, many aspects of the user interface—whether the user sees red or blue backgrounds, a radio button or a dialog box—are by and large not architectural since those decisions are generally confined to a limited area of the system.

But other quality attributes lie squarely in the realm of architecture. For instance, the ATAM concentrates on evaluating an architecture for suitability in terms of imbuing a system with the following quality attributes. (Definitions are based on Bass et al. [Bass 98])

- *Performance:* Performance refers to the responsiveness of the system—the time required to respond to stimuli (events) or the number of events processed in some interval of time. Performance qualities are often expressed by the number of transactions per unit time or by the amount of time it takes to complete a transaction with the system. Performance measures are often cited using *benchmarks*, which are specific transaction sets or workload conditions under which the performance is measured.
- *Reliability:* Reliability is the ability of the system to keep operating over time. Reliability is usually measured by mean time to failure.

[1] This is the last time we will address evaluating more than one architecture at a time since the methods we describe are carried out in the same fashion for either case.

- *Availability:* Availability is the proportion of time the system is up and running. It is measured by the length of time between failures as well as how quickly the system is able to resume operation in the event of failure.
- *Security:* Security is a measure of the system's ability to resist unauthorized attempts at usage and denial of service while still providing its services to legitimate users. Security is categorized in terms of the types of threats that might be made to the system.
- *Modifiability:* Modifiability is the ability to make changes to a system quickly and cost effectively. It is measured by using specific changes as benchmarks and recording how expensive those changes are to make.
- *Portability:* Portability is the ability of the system to run under different computing environments. These environments can be hardware, software, or a combination of the two. A system is portable to the extent that all of the assumptions about any *particular* computing environment are confined to one component (or at worst, a small number of easily changed components). If porting to a new system requires change, then portability is simply a special kind of modifiability.
- *Functionality:* Functionality is the ability of the system to do the work for which it was intended. Performing a task requires that many or most of the system's components work in a coordinated manner to complete the job.
- *Variability:* Variability is how well the architecture can be expanded or modified to produce new architectures that differ in specific, preplanned ways. Variability mechanisms may be run-time (such as negotiating on the fly protocols), compile-time (such as setting compilation parameters to bind certain variables), build-time (such as including or excluding various components or choosing different versions of a component), or code-time mechanisms (such as coding a device driver for a new device). Variability is important when the architecture is going to serve as the foundation for a whole family of related products, as in a product line.
- *Subsetability:* This is the ability to support the production of a subset of the system. While this may seem like an odd property of an architecture, it is actually one of the most useful and most overlooked. Subsetability can spell the difference between being able to deliver nothing when schedules slip versus being able to deliver a substantial part of the product. Subsetability also enables incremental development, a powerful development paradigm in which a minimal system is made to run early on and functions are added to it over time until the whole system is ready. Subsetability is a special kind of variability, mentioned above.
- *Conceptual integrity:* Conceptual integrity is the underlying theme or vision that unifies the design of the system at all levels. The architecture should do similar things in similar ways. Conceptual integrity is exemplified in an architecture that exhibits consistency, has a small number of data

and control mechanisms, and uses a small number of patterns throughout to get the job done.

By contrast, the SAAM concentrates on modifiability in its various forms (such as portability, subsetability, and variability) and functionality. The ARID method provides insights about the suitability of a portion of the architecture to be used by developers to complete their tasks.

If some other quality than the ones mentioned above is important to you, the methods still apply. The ATAM, for example, is structured in steps, some of which are dependent upon the quality being investigated, and others of which are not. Early steps of the ATAM allow you to define new quality attributes by explicitly describing the properties of interest. The ATAM can easily accommodate new quality-dependent analysis. When we introduce the method, you'll see where to do this. For now, though, the qualities in the list above form the basis for the methods' capabilities, and they also cover most of what people tend to be concerned about when evaluating an architecture.

2.6 Why Are Quality Attributes Too Vague for Analysis?

Quality attributes form the basis for architectural evaluation, but simply naming the attributes by themselves is not a sufficient basis on which to judge an architecture for suitability. Often, requirements statements like the following are written:

- "The system shall be robust."
- "The system shall be highly modifiable."
- "The system shall be secure from unauthorized break-in."
- "The system shall exhibit acceptable performance."

Without elaboration, each of these statements is subject to interpretation and misunderstanding. What you might think of as robust, your customer might consider barely adequate—or vice versa. Perhaps the system can easily adopt a new database but cannot adapt to a new operating system. Is that system maintainable or not? Perhaps the system uses passwords for security, which prevents a whole class of unauthorized users from breaking in, but has no virus protection mechanisms. Is that system secure from intrusion or not?

The point here is that quality attributes are not absolute quantities; they exist in the context of specific goals. In particular:

- A system is modifiable (or not) with respect to a specific kind of change.
- A system is secure (or not) with respect to a specific kind of threat.
- A system is reliable (or not) with respect to a specific kind of fault occurrence.
- A system performs well (or not) with respect to specific performance criteria.
- A system is suitable (or not) for a product line with respect to a specific set or range of envisioned products in the product line (that is, with respect to a specific product line *scope*).
- An architecture is buildable (or not) with respect to specific time and budget constraints.

If this doesn't seem reasonable, consider that no system can ever be, for example, completely reliable under all circumstances. (Think power failure, tornado, or disgruntled system operator with a sledgehammer.) Given that, it is incumbent upon the architect to understand under exactly what circumstances the system should be reliable in order to be deemed acceptable.

In a perfect world, the quality requirements for a system would be completely and unambiguously specified in a requirements document. Most of us do not live in such a world. Requirements documents are not written, or are written poorly, or are not finished when it is time to begin the architecture. Also, architectures have goals of their own that are not enumerated in a requirements document for the system: They must be built using resources at hand, they should exhibit conceptual integrity, and so on. And so the first job of an architecture evaluation is to elicit the specific quality goals against which the architecture will be judged.

If all of these goals are specifically, unambiguously articulated, that's wonderful. Otherwise, we ask the stakeholders to help us write them down during an evaluation. The mechanism we use is the *scenario*. A scenario is a short statement describing an interaction of one of the stakeholders with the system. A user would describe using the system to perform some task; these scenarios would very much resemble *use cases* in object-oriented parlance. A maintenance stakeholder would describe making a change to the system, such as upgrading the operating system in a particular way or adding a specific new function. A developer's scenario might involve using the architecture to build the system or predict its performance. A customer's scenario might describe the architecture reused for a second product in a product line or might assert that the system is buildable given certain resources.

Each scenario, then, is associated with a particular stakeholder (although different stakeholders might well be interested in the same scenario). Each scenario also addresses a particular quality, but in specific terms. Scenarios are discussed more fully in Chapter 3.

2.7 What Are the Outputs of an Architecture Evaluation?

2.7.1 Outputs from the ATAM, the SAAM, and ARID

An architecture evaluation results in information and insights about the architecture. The ATAM, the SAAM, and the ARID method all produce the outputs described below.

Prioritized Statement of Quality Attribute Requirements

An architecture evaluation can proceed only if the criteria for suitability are known. Thus, elicitation of quality attribute requirements against which the architecture is evaluated constitutes a major portion of the work. But no architecture can meet an unbounded list of quality attributes, and so the methods use a consensus-based prioritization. Having a prioritized statement of the quality attributes serves as an excellent documentation record to accompany any architecture and guide it through its evolution. All three methods produce this in the form of a set of quality attribute scenarios.

Mapping of Approaches to Quality Attributes

The answers to the analysis questions produce a mapping that shows how the architectural approaches achieve (or fail to achieve) the desired quality attributes. This mapping makes a splendid rationale for the architecture. Rationale is something that every architect should record, and most wish they had time to construct. The mapping of approaches to attributes can constitute the bulk of such a description.

Risks and Nonrisks

Risks are potentially problematic architectural decisions. Nonrisks are good decisions that rely on assumptions that are frequently implicit in the architecture. Both should be understood and explicitly recorded.[2]

Documenting of risks and nonrisks consists of

- An architectural decision (or a decision that has not been made)
- A specific quality attribute response that is being addressed by that decision along with the consequences of the predicted level of the response
- A rationale for the positive or negative effect that decision has on meeting the quality attribute requirement

An example of a risk is

> The rules for writing business logic modules in the second tier of your three-tier client-server style are not clearly articulated (*a decision that has not been made*). This could result in replication of functionality, thereby compromising modifiability of the third tier (*a quality attribute response and its consequences*). Unarticulated rules for writing the business logic can result in unintended and undesired coupling of components (*rationale for the negative effect*).

An example of a nonrisk is

> Assuming message arrival rates of once per second, a processing time of less than 30 milliseconds, and the existence of one higher priority process (*the architectural decisions*), a one-second soft deadline seems reasonable (*the quality attribute response and its consequences*) since the arrival rate is bounded and the preemptive effects of higher priority processes are known and can be accommodated (*the rationale*).

For a nonrisk to remain a nonrisk the assumptions must not change (or at least if they change, the designation of nonrisk will need to be rejustified). For example, if the message arrival rate, the processing time, or the number of higher priority processes changes in the example above, the designation of nonrisk could change.

2.7.2 Outputs Only from the ATAM

In addition to the preceding information, the ATAM produces an additional set of results described below.

Catalog of Architectural Approaches Used

Every architect adopts certain design strategies and approaches to solve the problems at hand. Sometimes these approaches are well known and part of the common knowledge of the field; sometimes they are unique and innovative to the system being built. In either case, they are the key to understanding whether the architecture will meet its goals and requirements. The ATAM includes a step in which the approaches used are catalogued, and this catalog can later serve as an introduction to the architecture for people who need to familiarize themselves with it, such as future architects and maintainers for the system.

[2] Risks can also emerge from other, nonarchitectural sources. For example, having a management structure that is misaligned with the architectural structure might present an organizational risk. Insufficient communication between the stakeholder groups and the architect is a common kind of management risk.

Approach- and Quality-Attribute-Specific Analysis Questions

The ATAM poses analysis questions that are based on the attributes being sought and the approaches selected by the architect. As the architecture evolves, these questions can be used in future mini-evaluations to make sure that the evolution is not taking the architecture in the wrong direction.

Sensitivity Points and Tradeoff Points

We term key architectural decisions *sensitivity points* and *tradeoff points*. A sensitivity point is a property of one or more components (and/or component relationships) that is critical for achieving a particular quality attribute response. For example:

- The level of confidentiality in a virtual private network might be sensitive to the number of bits of encryption.
- The latency for processing an important message might be sensitive to the priority of the lowest priority process involved in handling the message.
- The average number of person-days of effort it takes to maintain a system might be sensitive to the degree of encapsulation of its communication protocols and file formats.

Sensitivity points tell a designer or analyst where to focus attention when trying to understand the achievement of a quality goal. They serve as yellow flags: "Use caution when changing this property of the architecture." Particular values of sensitivity points may become risks when realized in an architecture. Consider the examples above. A particular value in the encryption level—say, 32-bit encryption—may present a risk in the architecture. Or having a very low priority process in a pipeline that processes an important message may become a risk in the architecture.

A *tradeoff point* is a property that affects more than one attribute and is a sensitivity point for more than one attribute. For example, changing the level of encryption could have a significant impact on both security and performance. Increasing the level of encryption improves the predicted security but requires more processing time. If the processing of a confidential message has a hard real-time latency requirement then the level of encryption could be a tradeoff point. Tradeoff points are the most critical decisions that one can make in an architecture, which is why we focus on them so carefully.

Finally, it is not uncommon for an architect to answer an elicitation question by saying, "We haven't made that decision yet." In this case you cannot point to a component or property in the architecture and call it out as a sensitivity point because the component or property might not exist yet. However, it is important to flag key decisions that have been made as well as key decisions that have not yet been made.

2.8 What Are the Benefits and Costs of Performing an Architecture Evaluation?

The main, and obvious, benefit of architecture evaluation is, of course, that it uncovers problems that if left undiscovered would be orders of magnitude more expensive to correct later. In short, architecture evaluation produces better architectures. Even if the evaluation uncovers no problems that warrant attention, it will increase everyone's level of confidence in the architecture.

But there are other benefits as well. Some of them are hard to measure, but they all contribute to a successful project and a more mature organization. You may not experience all of these on every evaluation, but the following is a list of the benefits we've often observed.

Puts Stakeholders in the Same Room

An architecture evaluation is often the first time that many of the stakeholders have ever met each other; sometimes it's the first time the architect has met them. A group dynamic emerges in which stakeholders see each other as all wanting the same thing: a successful system. Whereas before, their goals may have been in conflict with each other (and in fact, still may be), now they are able to explain their goals and motivations so that they begin to understand each other. In this atmosphere, compromises can be brokered or innovative solutions proposed in the face of greater understanding. It is almost always the case that stakeholders trade phone numbers and e-mail addresses and open channels of communication that last beyond the evaluation itself.

Forces an Articulation of Specific Quality Goals

The role of the stakeholders is to articulate the quality goals that the architecture should meet in order to be deemed successful. These goals are often not captured in any requirements document, or at least not captured in an unambiguous fashion beyond vague platitudes about reliability and modifiability. Scenarios provide explicit quality benchmarks.

Results in the Prioritization of Conflicting Goals

Conflicts that might arise among the goals expressed by the different stakeholders will be aired. Each method includes a step in which the goals are prioritized by the group. If the architect cannot satisfy all of the conflicting goals, he or she will receive clear and explicit guidance about which ones are considered most important. (Of course, project management can step in and veto or adjust the group-derived priorities—perhaps they perceive some stakeholders and their goals as "more equal" than others—but not unless the conflicting goals are aired.)

Forces a Clear Explication of the Architecture

The architect is compelled to make a group of people not privy to the architecture's creation understand it, in detail, in an unambiguous way. Among other things, this will serve as a dress rehearsal for explaining it to the other designers, component developers, and testers. The project benefits by forcing this explication early.

Improves the Quality of Architectural Documentation

Often, an evaluation will call for documentation that has not yet been prepared. For example, an inquiry along performance lines will reveal the need for documentation that shows how the architecture handles the interaction of run-time tasks or processes. If the evaluation requires it, then it's an odds-on bet that somebody on the project team (in this case, the performance engineer) will need it also. Again, the project benefits because it enters development better prepared.

Uncovers Opportunities for Cross-Project Reuse

Stakeholders and the evaluation team come from outside the development project, but often work on or are familiar with other projects within the same parent organization. As such, both are in a good position either to spot components that can be reused on other projects or to know of components (or other assets) that already exist and perhaps could be imported into the current project.

Results in Improved Architecture Practices

Organizations that practice architecture evaluation as a standard part of their development process report an improvement in the quality of the architectures that are evaluated. As development organizations learn to anticipate the kinds of questions that will be asked, the kinds of issues that will be raised, and the kinds of documentation that will be required for evaluations, they naturally preposition themselves to maximize their performance on the evaluations. Architecture evaluations result in better architectures not only after the fact but before the fact as well. Over time, an organization develops a culture that promotes good architectural design.

Now, not all of these benefits may resonate with you. If your organization is small, maybe all of the stakeholders know each other and talk regularly. Perhaps your organization is very mature when it comes to working out the requirements for a system, and by the time the finishing touches are put on the architecture the requirements are no longer an issue because everyone is completely clear what they are. If so, congratulations. But many of the organizations in which we have carried out architecture evaluations are not quite so sophisticated, and there have always been requirements issues that were raised (and resolved) when the architecture was put on the table.

There are also benefits to future projects in the same organization. A critical part of the ATAM consists of probing the architecture using a set of quality-specific analysis questions, and neither the method nor the list of questions is a secret. The architect is perfectly free to arm her- or himself before the evaluation by making sure that the architecture is up to snuff with respect to the relevant questions. This is rather like scoring well on a test whose questions you've already seen, but in this case it isn't cheating: it's professionalism.

The costs of architecture evaluation are all personnel costs and opportunity costs related to those personnel participating in the evaluation instead of something else. They're easy enough to calculate. An example using the cost of an ATAM-based evaluation is shown in Table 2.1. The left-most column names the phases of the ATAM (which will be described in subsequent chapters). The other columns split the cost among the participant groups. Similar tables can easily be constructed for other methods.

Table 2.1 shows figures for what we would consider a medium-size evaluation effort. While 70 person-days sounds like a substantial sum, in actuality it may not be so daunting. For one reason, the *calendar* time added to the project is minimal. The schedule should not be impacted by the preparation at all, nor the follow-up. These activities can be carried out behind the scenes, as it were. The middle phases consume actual project days, usually three or so. Second, the project normally does not have to pay for all 70 staff days. Many of the

Table 2.1 Approximate Cost of a Medium-Size ATAM-Based Evaluation

		Stakeholders	
Participant Group ATAM Phase	**Evaluation Team (assume 5 members)**	**Project Decision Makers (assume architect, project manager, customer)**	**Other Stakeholders (assume 8)**
Phase 0: Preparation	1 person-day by team leader	1 person-day	0
Phase 1: Initial evaluation (1 day)	5 person-days	3 person-days	0
Phase 2: Complete evaluation (3 days)	15 person-days	9 person-days + 2 person-days to prepare	16 person-days (most stakeholders present only for 2 days)
Phase 3: Follow-up	15 person-days	3 person-days to read and respond to report	0
TOTAL	36 person-days	18 person-days	16 person-days

stakeholders work for other cost centers, if not other organizations, than the development group. Stakeholders by definition have a vested interest in the system, and they are often more than willing to contribute their time to help produce a quality product.

It is certainly easy to imagine larger and smaller efforts than the one characterized by Table 2.1. As we will see, all of the methods are flexible, structured to iteratively spiral down into as much detail as the evaluators and evaluation client feel is warranted. Cursory evaluations can be done in a day; excruciatingly detailed evaluations could take weeks. However, the numbers in Table 2.2 represent what we would call nominal applications of the ATAM. For smaller projects, Table 2.2 shows how those numbers can be halved.

If your group evaluates many systems in the same domain or with the same architectural goals, then there is another way that the cost of evaluation can be reduced. Collect and record the scenarios used in each evaluation. Over time, you will find that the scenario sets will begin to resemble each other. After you have performed several of these almost-alike evaluations, you can produce a "canonical" set of scenarios based on past experience. At this point, the scenarios have in essence graduated to become a checklist, and you can dispense with the bulk of the scenario-generation part of the exercise. This saves about a day. Since scenario generation is the primary duty of the stakeholders, the bulk of their time can also be done away with, lowering the cost still further.

Table 2.2 Approximate Cost of a Small ATAM-Based evaluation

		Stakeholders	
Participant Group ATAM Phase	**Evaluation team (assume 2 members)**	**Project Decision Makers (assume architect, project manager)**	**Other Stakeholders (assume 3)**
Phase 0: Preparation	1 person-day by team leader	1 person-day	0
Phase 1: Initial evaluation (1 day)	2 person-days	2 person-days	0
Phase 2: Complete evaluation (2 days)	4 person-days	4 person-days + 2 person-days to prepare	6 person-days
Phase 3: Follow-up	8 person-days	2 person-days to read and respond to report	0
TOTAL	15 person-days	11 person-days	6 person-days

Table 2.3 Approximate Cost of a Medium-Size Checklist-based ATAM-Based Evaluation

		Stakeholders	
Participant Group ATAM Phase	**Evaluation Team (assume 4 members)**	**Project Decision Makers (assume architect, project manager, customer)**	**Other Stakeholders (assume the customer validates the checklist)**
Phase 0: Preparation	1 person-day by team leader	1 person-day	0
Phase 1: Initial evaluation (1 day)	4 person-days	3 person-days	0
Phase 2: Complete evaluation (2 days)	8 person-days	6 person-days	2 person-days
Phase 3: Follow-up	12 person-days	3 person-days to read and respond to report	0
TOTAL	25 person-days	13 person-days	2 person-days

(You still may want to have a few key stakeholders, including the customer, to validate the applicability of your checklist to the new system.) The team size can be reduced, since no one is needed to record scenarios. The architect's preparation time should be minimal since the checklist will be publicly available even when he or she begins the architecture task.

Table 2.3 shows the cost of a medium-size checklist-based evaluation using the ATAM, which comes in at about $\frac{4}{7}$ of the cost of the scenario-based evaluation of Table 2.1.

The next chapter will introduce the first of the three architecture evaluation methods in this book: the Architecture Tradeoff Analysis Method.

2.9 For Further Reading

The For Further Reading list of Chapter 9 (Comparing Software Architecture Evaluation Methods) lists good references on various architecture evaluation methods.

Zhao has assembled a nice collection of literature resources dealing with software architecture analysis [Zhao 99].

Once an architecture evaluation has identified changes that should be made to an architecture, how do you prioritize them? Work is emerging to help an architect or project manager assign quantitative cost and benefit information to architectural decisions [Kazman 01].

2.10 Discussion Questions

1. How does your organization currently decide whether a proposed software architecture should be adopted or not? How does it decide when a software architecture has outlived its usefulness and should be discarded in favor of another?
2. Make a business case, specific to your organization, that tells whether or not conducting a software architecture evaluation would pay off. Assume the cost estimates given in this chapter if you like, or use your own.
3. Do you know of a case where a flawed software architecture led to the failure or delay of a software system or project? Discuss what caused the problem and whether a software architecture evaluation might have prevented the calamity.
4. Which quality attributes tend to be the most important to systems in your organization? How are those attributes specified? How does the architect know what they are, what they mean, and what precise levels of each are required?
5. For each quality attribute discussed in this chapter—or for each that you named in answer to the previous question—hypothesize three different architectural decisions that would have an effect on that attribute. For example, the decision to maintain a backup database would probably increase a system's availability.
6. Choose three or four pairs of quality attributes. For each pair (think about tradeoffs), hypothesize an architectural decision that would increase the first quality attribute at the expense of the second. Now hypothesize a different architectural decision that would raise the second but lower the first.

3

The ATAM—A Method for Architecture Evaluation

> There is also a rhythm and a pattern between the phenomena of nature which is not apparent to the eye, but only to the eye of analysis. . . .
>
> —Richard Feynman
> *The Character of Physical Law*

This chapter will present the first of three methods for architecture evaluation that are the primary subject of this book. It is called the Architecture Tradeoff Analysis Method (ATAM). The ATAM gets its name because it not only reveals how well an architecture satisfies particular quality goals but it also provides insight into how those quality goals interact with each other—how they *trade off* against each other.

Having a structured method makes the analysis repeatable and helps ensure that the right questions regarding an architecture will be asked early, during the requirements and design stages when discovered problems can be solved relatively cheaply. It guides users of the method—the stakeholders—to look for conflicts and for resolutions to these conflicts in the software architecture.

The ATAM can also be used to analyze legacy systems. This need arises when the legacy system needs to undergo major modifications, integration with other systems, porting, or other significant upgrades. Assuming that an accurate architecture of the legacy system is available (which frequently must be acquired and verified using architecture extraction and conformance testing methods), applying the ATAM results in increased understanding of the quality attributes of the system.

The ATAM draws its inspiration and techniques from three areas: the notion of architectural styles; the quality attribute analysis communities; and the Software Architecture Analysis Method (SAAM), which was the predecessor to the

ATAM. Styles and quality attribute analysis are introduced in this chapter but discussed more thoroughly in Chapter 5. The SAAM is presented in Chapter 7.

This chapter introduces the steps of the ATAM, then describes the steps in more depth, and concludes by discussing how the steps are arranged in phases carried out over time.

3.1 Summary of the ATAM Steps

The main part of the ATAM consists of nine steps. (Other parts of the ATAM, including preparation before and follow-up after an evaluation, are detailed later in this chapter.) The steps are separated into four groups:

- Presentation, which involves exchanging information through presentations
- Investigation and analysis, which involves assessing key quality attribute requirements vis-à-vis architectural approaches
- Testing, which involves checking the results to date against the needs of all relevant stakeholders
- Reporting, which involves presenting the results of the ATAM

Presentation

1. **Present the ATAM.** The evaluation leader describes the evaluation method to the assembled participants, tries to set their expectations, and answers questions they may have.
2. **Present the business drivers.** A project spokesperson (ideally the project manager or system customer) describes what business goals are motivating the development effort and hence what will be the primary architectural drivers (for example, high availability or time to market or high security).
3. **Present the architecture.** The architect describes the architecture, focusing on how it addresses the business drivers.

Investigation and Analysis

4. **Identify the architectural approaches.** Architectural approaches are identified by the architect but are not analyzed.
5. **Generate the quality attribute utility tree.** The quality attributes that comprise system "utility" (performance, availability, security, modifiability, usability, and so on) are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritized.

6. **Analyze the architectural approaches.** Based upon the high-priority scenarios identified in Step 5, the architectural approaches that address those scenarios are elicited and analyzed (for example, an architectural approach aimed at meeting performance goals will be subjected to a performance analysis). During this step architectural risks, nonrisks, sensitivity points, and tradeoff points are identified.

Testing

7. **Brainstorm and prioritize scenarios.** A larger set of scenarios is elicited from the entire group of stakeholders. This set of scenarios is prioritized via a voting process involving all the stakeholders.
8. **Analyze the architectural approaches.** This step reiterates the activities of Step 6 but uses the highly ranked scenarios from Step 7. Those scenarios are considered to be test cases to confirm the analysis performed thus far. This analysis may uncover additional architectural approaches, risks, nonrisks, sensitivity points, and tradeoff points, which are then documented.

Reporting

9. **Present the results.** Based upon the information collected during the ATAM evaluation (approaches, scenarios, attribute-specific questions, the utility tree, risks, nonrisks, sensitivity points, tradeoffs), the ATAM team presents the findings to the assembled stakeholders.

Sometimes there must be dynamic modifications to the schedule to accommodate the availability of personnel or architectural information. Although the steps are numbered, suggesting linearity, this is not a strict waterfall process. There will be times when an analyst will return briefly to an earlier step, or will jump forward to a later step, or will iterate among steps, as the need dictates. The importance of the steps is to clearly delineate the activities involved in the ATAM along with the output of these activities. The next section offers a detailed description of the steps of the ATAM.

3.2 Detailed Description of the ATAM Steps

3.2.1 Step 1: Present the ATAM

The first step calls for the evaluation leader to present the ATAM to the assembled stakeholders. This time is used to explain the process that everyone will be following, allows time to answer questions, and sets the context and expectations for the remainder of the activities. In particular, the leader will describe

- The ATAM steps in brief
- The techniques that will be used for elicitation and analysis: utility tree generation, architecture approach-based elicitation and analysis, and scenario brainstorming and prioritization
- The outputs of the evaluation: the scenarios elicited and prioritized, the questions used to understand and evaluate the architecture, a utility tree describing and prioritizing the driving architectural requirements, a set of identified architectural approaches, a set of risks and nonrisks discovered, and a set of sensitivity points and tradeoffs discovered

A standard set of slides is used to aid in the presentation; the outline of such a set is given in Figure 6.6.

3.2.2 Step 2: Present the Business Drivers

The evaluation's participants—the stakeholders as well as the evaluation team members—need to understand the context for the system and the primary business drivers motivating its development. In this step, a project decision maker (ideally the project manager or the system's customer) presents a system overview from a business perspective. An outline for such a presentation is given in Figure 3.1. The presentation should describe

- The system's most important functions
- Any relevant technical, managerial, economic, or political constraints
- The business goals and context as they relate to the project
- The major stakeholders

Business Context/Drivers Presentation (~12 slides; 45 minutes)

- Description of the business environment, history, market differentiators, driving requirements, stakeholders, current need, and how the proposed system will meet those needs/requirements (3–4 slides)
- Description of business constraints (e.g., time to market, customer demands, standards, cost, etc.) (1–3 slides)
- Description of the technical constraints (e.g., commercial off-the-shelf [COTS] products, interoperation with other systems, required hardware or software platform, reuse of legacy code, etc.) (1–3 slides)
- Quality attributes requirements and from what business needs these are derived (2–3 slides)
- Glossary (1 slide)

Figure 3.1 Example Template for the Business Case Presentation

- The architectural drivers (major quality attribute goals that shape the architecture)

3.2.3 Step 3: Present the Architecture

In this step, the lead architect (or architecture team) makes a presentation describing the architecture at an appropriate level of detail. What the "appropriate level" is depends on several factors: how much of the architecture has been designed and documented, how much time is available, and the nature of the behavioral and quality requirements. The architectural information presented will directly affect the analysis that is possible and the quality of this analysis. Frequently the evaluation team will need to ask for additional architectural information before a more substantial analysis is possible.

In this presentation the architect should cover

- Technical constraints such as an operating system, hardware, or middleware prescribed for use
- Other systems with which the system must interact
- Architectural approaches used to meet quality attribute requirements

Architectural views, as described in Chapter 1, are the primary vehicle that the architect should use to present the architecture. Which views the architect chooses to present will, of course, depend on what about the architecture is important to convey. Functional, concurrency, code, and physical views are useful in almost every evaluation, and the architect should be prepared to show those. Other views should be presented in addition if they contain information relevant to the architecture at hand, especially information relevant to achieving important quality attribute goals. As a rule of thumb, the architect should present those views that he or she found most important to work on during the creation of the architecture.

At this time the evaluation team begins its initial probing for and capturing of architectural approaches as a prelude to Step 4.

An outline for the architecture presentation is shown in Figure 3.2. Providing a template like this to the architect well in advance of the ATAM meeting helps ensure that he or she presents the right information and also helps to ensure that the exercise stays on schedule.

3.2.4 Step 4: Identify the Architectural Approaches

The ATAM focuses on analyzing an architecture by understanding its architectural approaches. In this step they are captured by the evaluation team but are not analyzed. The team will ask the architect to explicitly name any identifiable approaches used, but they will also capture any approaches they heard during the architecture presentation in the previous step.

Architecture Presentation (~20 slides; 60 minutes)

- Driving architectural requirements, the measurable quantities associated with these requirements, and any existing standards/models/approaches for meeting these (2–3 slides)
- High-level architectural views (4–8 slides)
  - Functional: functions, key system abstractions, and domain elements along with their dependencies, data flow
  - Code: the subsystems, layers, and modules that describe the system's decomposition of functionality, along with the objects, procedures, and functions that populate these and the relations among them (e.g., procedure call, method invocation, callback, containment)
  - Concurrency: processes, threads along with the synchronization, data flow, and events that connect them
  - Physical: CPUs, storage, and external devices/sensors along with the networks and communication devices that connect them
- Architectural approaches, styles, patterns, or mechanisms employed, including what quality attributes they address and a description of how the approaches address those attributes (3–6 slides)
- Use of commercial off-the-shelf (COTS) products and how they are chosen/integrated (1–2 slides)
- Trace of 1–3 of the most important use case scenarios, including, if possible, the run-time resources consumed for each scenario (1–3 slides)
- Trace of 1–3 of the most important growth scenarios, describing, if possible, the change impact (estimated size/difficulty of the change) in terms of the changed components, connectors, or interfaces (1–3 slides)
- Architectural issues/risks with respect to meeting the driving architectural requirements (2–3 slides)
- Glossary (1 slide)

Figure 3.2 Example Template for the Architecture Presentation

The ATAM concentrates on identifying architectural approaches and architectural styles[1] because these represent the architecture's means of addressing the highest priority quality attributes, that is, the means of ensuring that the critical requirements are met in a predictable way. These architectural approaches define the important structures of the system and describe the ways in which the system can grow, respond to changes, withstand attacks, integrate with other systems, and so forth.

[1] As we mentioned in Chapter 1, we use the term *approaches* because not all architects are familiar with the language of architectural styles and so may not be able to enumerate a set of styles used in the architecture. But every architect makes architectural decisions, and the set of these we call *approaches*. These can certainly be elicited from any conscientious architect.

An architectural style includes a description of component types and their topology, a description of the pattern of data and control interactions among the components, and an informal description of the benefits and drawbacks of using that style. Architectural styles are important since they differentiate classes of designs by offering experiential evidence of how each class has been used along with qualitative reasoning to explain why each class has certain properties and when to use it.

For example, Figure 3.3 shows the Concurrent Pipelines style. This style consists of a set of pipelines, each of which consists of a sequence of processes. Each input message is incrementally transformed by each process in the appropriate sequence. Systems built like this have several benefits, many of which result in enhanced modifiability.

- The system can be understood easily as a sequence of data transformations.
- Each element of the pipeline can be modified or replaced, in principle, without affecting any of the other elements.
- Elements can be reused elsewhere.
- Evaluators can reason about this style in terms of its performance implications.

A style can be thought of as a set of constraints on an architecture—constraints on component types and their interactions—and these constraints define the set or family of architectures that satisfy them. By locating architectural styles in an architecture, the evaluators can see what strategies the architect has used to respond to the system's driving quality attribute goals.

A specialization of an architectural style, called *attribute-based architectural styles* (ABASs), is particularly useful in the ATAM. An ABAS is an architectural

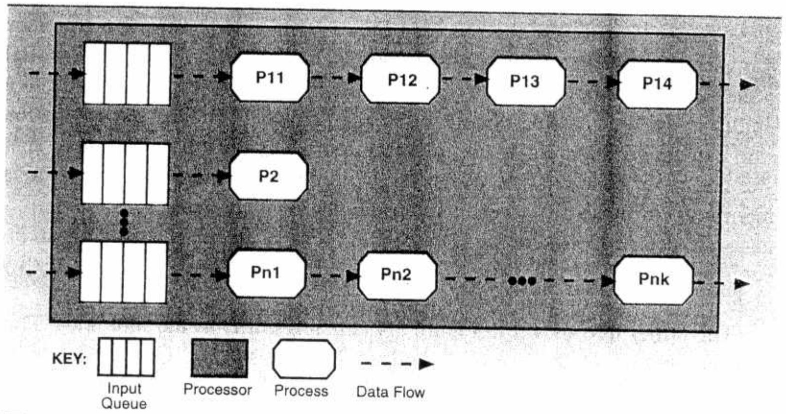


Figure 3.3 Concurrent Pipelines Style

style along with an explanation of how that style achieves certain quality attributes. That explanation in an ABAS leads to attribute-specific questions associated with the style. For example, a performance-oriented ABAS highlights architectural decisions in a style that is relevant to performance—how processes are allocated to processors, where they share resources, how their priorities are assigned, how they synchronize, and so forth. The derivative questions probe important architectural decisions such as the priority of the processes, estimates of their execution time, places where they synchronize, queuing disciplines, and so forth: in other words, information that is relevant to understanding the performance of this style. ABASs are discussed in Chapter 5.

3.2.5 Step 5: Generate the Quality Attribute Utility Tree

In this step the evaluation team works with the project decision makers (here, the architecture team, manager, and customer representatives) to identify, prioritize, and refine the system's most important quality attribute goals. This crucial step guides the remainder of the analysis. Lacking this guidance, the evaluators could spend precious time analyzing the architecture ad infinitum without ever touching on issues that mattered to its sponsors. There must be a way to focus the attention of all the stakeholders and the evaluation team on the aspects of the architecture that are most critical to the system's success. This is accomplished by building a utility tree.

The output of the utility tree–generation step is a prioritization of specific quality attribute requirements, realized as scenarios. The utility tree serves to make concrete the quality attribute requirements, forcing the architect and customer representatives to define the relevant quality requirements precisely.

Utility trees provide a mechanism for directly and efficiently translating the business drivers of a system into concrete quality attribute scenarios. For example, in an e-commerce system two of the business drivers might be stated as, "Security is central to the success of the system since ensuring the privacy of our customers' data is of utmost importance," and "Modifiability is central to the success of the system since we need to be able to respond quickly to a rapidly evolving and very competitive marketplace." Before the evaluation team can assess the architecture, these system goals must be made more specific and more concrete. Moreover, the team needs to understand the relative importance of these goals versus other quality attribute goals, such as performance, to determine where the team should focus its attention during the architecture evaluation. Utility trees help to prioritize and make concrete the quality goals. An example of a utility tree is shown in Figure 3.4.

The utility tree shown in Figure 3.4 contains *utility* as the root node. This is an expression of the overall "goodness" of the system. Quality attributes form the second level of the utility tree. Typically the quality attributes of performance, modifiability, availability, and security are the children of utility,

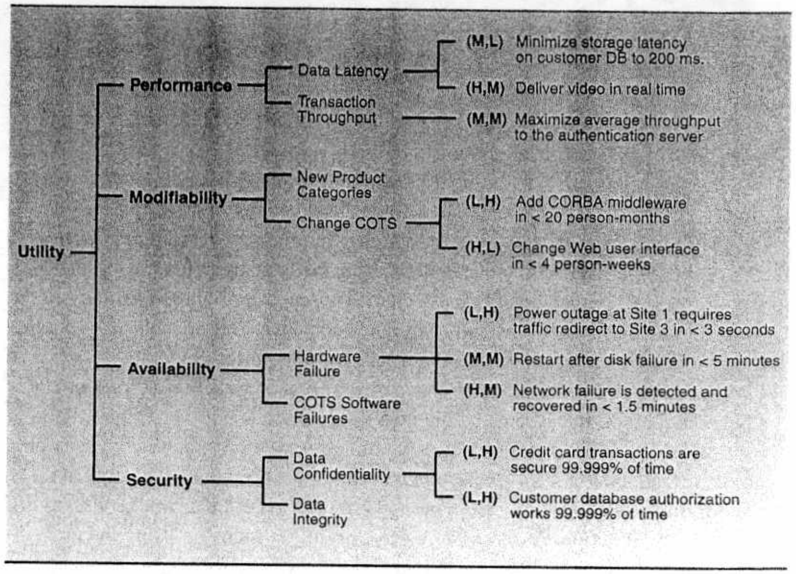

Utility
Performance
Data Latency
Transaction Throughput
(M,L) Minimize storage latency on customer DB to 200 ms.
(H,M) Deliver video in real time
(M,M) Maximize average throughput to the authentication server
Modifiability
New Product Categories
Change COTS
(L,H) Add CORBA middleware in < 20 person-months
(H,L) Change Web user interface in < 4 person-weeks
Availability
Hardware Failure
COTS Software Failures
(L,H) Power outage at Site 1 requires traffic redirect to Site 3 in < 3 seconds
(M,M) Restart after disk failure in < 5 minutes
(H,M) Network failure is detected and recovered in < 1.5 minutes
Security
Data Confidentiality
Data Integrity
(L,H) Credit card transactions are secure 99.999% of time
(L,H) Customer database authorization works 99.999% of time

Figure 3.4 Sample Utility Tree

although participants are free to name their own quality attributes. Sometimes different stakeholder groups use different names for the same ideas (for example, some stakeholders prefer to speak of "maintainability"). Sometimes they introduce quality attribute names that are meaningful in their own culture but not widely used elsewhere. Whatever names the stakeholders introduce are fine, as long as the stakeholders are able to explain what they mean through refinement at the next levels.

Under each of these quality attributes are specific quality attribute refinements. For example, in Figure 3.4 performance is decomposed into "data latency" and "transaction throughput." This is a step toward refining the attribute goals into quality attribute scenarios that are concrete enough for prioritization and analysis. "Data latency" is then further refined into "minimize storage latency on customer database" and "deliver video in real time" because these are both kinds of data latency relevant to the system of the example.

These quality attribute scenarios (see the sidebar Scenarios) at the leaf nodes are now specific enough to be prioritized relative to each other and (equally important) analyzed. The prioritization may be on a 0 to 10 scale or may use relative rankings such as High (H), Medium (M), and Low (L). (We prefer the High/Medium/Low approach since we find that it works well for our purposes and takes less time than trying to assign precise numbers to an imprecise quantity.)

Scenarios

Typically the first job of an architecture evaluation is to precisely elicit the specific quality goals against which the architecture will be judged. The mechanism that we use for this elicitation is the *scenario*.

We use scenarios heavily in all three of our evaluation methods, and many other software and system analysis methods use them as well. They have been used for years in user interface engineering, requirements elicitation, performance modeling, and safety inspections. Why do so many fields rely on such a seemingly simple and innocuous device? The answer is threefold: they are simple to create and understand, they are inexpensive (it doesn't take much training to generate or work with them), and they are effective.

A scenario is a short statement describing an interaction of one of the stakeholders with the system. A *user* would describe using the system to perform some task; his or her scenarios would very much resemble *use cases* in object-oriented parlance. A *maintainer* would describe making a change to the system, such as upgrading the operating system in a particular way or adding a specific new function. A *developer*'s scenario might focus on using the architecture to build the system or predict its performance. A *product line manager's* scenario might describe how the architecture is to be reused for a second product in a product line.

Scenarios provide a vehicle for taking vague development-time qualities such as modifiability and turning them into something concrete: specific examples of current and future uses of a system. Scenarios are also useful in understanding run-time qualities such as performance or availability. This is because scenarios specify the kinds of operations over which performance needs to be measured or the kinds of failures the system will have to withstand.

Structure of Scenarios

Scenarios tell a very brief story about an interaction with the system from the point of view of a stakeholder. We ask stakeholders in evaluation exercises to phrase scenarios using a three-part format that helps keep the descriptions crisp and to make sure that the scenario provides enough information to serve as the basis for investigation. The three parts are stimulus, environment, and response.

The *stimulus* is the part of the scenario that explains or describes what the stakeholder does to initiate the interaction with the system. A user may invoke a function; a maintainer may make a change; a tester may run a test; an operator may reconfigure the system in some way; and so on.

The *environment* describes what's going on at the time of the stimulus. What is the system's state? What unusual conditions are in effect? Is the system heavily loaded? Is one of the processors down? Is one of the communication channels flooded? Any ambient condition that is relevant to understanding the scenario should be described. By convention, if the environment is simply "under normal conditions," then it is omitted.

The *response* tells us how the system—through its architecture—should respond to the stimulus. Is the function carried out? Is the test successful? Does the reconfiguration happen? How much effort did the maintenance change require?

The response is often the key to understanding what quality attribute the stakeholder who proposed the scenario is concerned about. If the response to a user-invokes-a-function stimulus is simply that the function happens, then the stakeholder is probably interested in the system's functionality. If the stakeholder appends "with no error" or "within two seconds" to the end, that indicates an interest in reliability and performance, respectively. By noticing the quality attribute of interest, the evaluation leader can prod the stakeholder to clarify or refine the scenario if necessary. Perhaps the stakeholder is interested in performance but left out a statement of the scenario's environment. The leader could ask about the ambient workload on the system to see if that played a part in the stakeholder's interest. Or the leader could ask whether the stakeholder was interested in worst-case, average, or best-case performance.

Ideally, all scenarios are expressed in this stimulus–environment–response form. In practice we don't usually achieve this ideal. In the heat of brainstorming we don't always pause to insist that every scenario be well formed. That's why the template in Figure 3.5 includes a place for the stimulus, environment, and response to be captured. That way, we can spend time structuring only those scenarios selected for analysis.

Types of Scenarios

In the ATAM we use three types of scenarios: *use case scenarios* (typical uses of the existing system, used for information elicitation), *growth scenarios* (anticipated changes to the system), and *exploratory scenarios* (extreme changes that are expected to "stress" the system). These different types of scenarios are used to probe a system from different angles, optimizing the chances of surfacing architectural decisions at risk. Examples of each type of scenario follow.

Use Case Scenarios

Use case scenarios describe a user's intended interaction with the completed, running system. For example:

1. The user wants to examine budgetary and actual data under different fiscal years without reentering project data (usability).
2. A data exception occurs and the system notifies a defined list of recipients by e-mail and displays the offending conditions in red on data screens (reliability).
3. The user changes a graph layout from horizontal to vertical and the graph is redrawn in one second (performance).
4. A remote user requests a database report via the Web during a peak period and receives it within five seconds (performance).

5. The caching system is switched to another processor when its processor fails, within one second of the failure (reliability).
6. There is a radical course adjustment before weapon release while approaching the target that the software computes in 100 ms. (performance).

Notice that each of the above use case scenarios expresses a specific stakeholder's desires. Also, the stimulus and the response associated with the attribute are easily identifiable. For example, in scenario 6 above, "radical course adjustment before weapon release while approaching the target," the stimulus and latency goal of 100 ms. is called out as being the important response measure. For scenarios to be well formed it must be clear what the stimulus is, what the environmental conditions are, and what the measurable or observable manifestation of the response is.

The quality attribute characterizations (which are discussed in Chapter 5) suggest questions that can be helpful in refining scenarios. Again consider scenario 6, which considers thse analysis questions:

- Are data sampling rates increased during radical course adjustments?
- Are real-time deadlines shortened during radical course adjustments?
- Is more execution time required to calculate a weapon solution during radical course adjustments?

Growth Scenarios

Growth scenarios represent typical *anticipated* changes to a system. Each scenario also has attribute-related ramifications, many of which are for attributes other than modifiability. For example, scenarios 1 and 4 below will have performance consequences and scenario 5 might have performance, security, and reliability implications.

1. Change the heads-up display to track several targets simultaneously without affecting latency.
2. Add a new message type to the system's repertoire in less than a person-week of work.
3. Add a collaborative planning capability with which two planners at different sites can collaborate on a plan in less than a person-year of work.
4. Double the maximum number of tracks to be handled by the system and keep to 200 ms. the maximum latency of track data to the screen.
5. Migrate to an operating system in the same family, or to a new release of the existing operating system with less than a person-year of work.
6. Add a new data server to reduce latency in use case scenario 5 to 2.5 seconds within one person-week.
7. Double the size of existing database tables while maintaining a one-second average retrieval time.

Exploratory Scenarios

Exploratory scenarios push the envelope and stress the system. The goal of these scenarios is to expose the limits or boundary conditions of the current design, exposing possibly implicit assumptions. Systems are seldom conceived and designed to handle these kinds of modifications, but at some point in the future these might be realistic requirements for change, so the stakeholders might like to understand the ramifications of such changes. For example:

1. Add a new three-dimensional map feature and a virtual reality interface for viewing the maps in less than five person-months of effort.
2. Change the underlying Unix platform to Macintosh.
3. Reuse the 25-year-old software on a new generation of the aircraft.
4. Reduce the time budget for displaying changed track data by a factor of ten.
5. Improve the system's availability from 98% to 99.999%.
6. Make changes so that half of the servers can go down during normal operation without affecting overall system availability.
7. Increase the number of bids processed hourly by a factor of ten while keeping worst-case response time below ten seconds.

Each type of scenario—use case, growth, and exploratory—helps illuminate a different aspect of the architecture, and together provide a three-pronged strategy for evaluation. Because scenarios are easy to formulate and understand, they are an excellent vehicle with which all stakeholders can communicate their architectural interests, and serve as the backbone for many evaluation methods.

—*RK*

Participants prioritize the utility tree along two dimensions: (1) by the importance of each scenario to the success of the system and (2) by the degree of difficulty posed by the achievement of the scenario, in the estimation of the architect. For example, "minimize storage latency on customer database" has priorities (M,L), meaning that it is of medium importance to the success of the system and the architect expects low difficulty to achieve, while "deliver video in real time" has priorities (H,M), meaning that it is highly important to the success of the system and the architect perceives the achievement of this scenario to be of medium difficulty.

Clearly the scenarios marked (H,H) are the prime candidates for scrutiny during the analysis steps of the ATAM. After those are handled, then either the (M,H) or (H,M) scenarios are attacked, depending on the consensus of the participants. After those, time permitting, will come the (M,M) scenarios. A scenario garnering an L rating in either category is not likely to be examined since it makes little sense to spend time on a case of little importance or little expected difficulty.

Refining the utility tree often leads to interesting and unexpected results. For example, Figure 3.4 came from an actual evaluation in which the stakeholders initially told us that security and modifiability were the key quality attributes. It turns out that another group of stakeholders thought that in fact performance and availability were the important drivers, and this discrepancy did not surface until the utility tree step. Creating the utility tree guides the key stakeholders in considering, explicitly stating, and prioritizing all of the current and future driving forces on the architecture.

The output of utility tree generation is a prioritized list of scenarios that serves as a plan for the remainder of the ATAM. It tells the ATAM team where to spend its (relatively limited) time, and in particular where to probe for architectural approaches and risks. The utility tree guides the evaluators to look at the architectural approaches involved with satisfying the high-priority scenarios at the leaves of the utility tree. Additionally, the utility tree serves to make the quality attribute requirements concrete, forcing the evaluation team and the customer to define their quality requirements precisely. Statements commonly found in requirements documents such as "The architecture shall be modifiable and robust" are untenable here because they have no operational meaning: they are not testable.

3.2.6 Step 6: Analyze the Architectural Approaches

At this point, there is now a prioritized set of concrete quality requirements (from Step 5) and a set of architectural approaches utilized in the architecture (from Step 4). Step 6 sizes up how well suited they are to each other. Here, the evaluation team can probe the architectural approaches that realize the important quality attributes. This is done with an eye to documenting these architectural decisions and identifying their risks, nonrisks, sensitivity points, and tradeoffs. The team probes for sufficient information about each architectural approach to conduct a rudimentary analysis about the attribute for which the approach is relevant. The goal is for the evaluation team to be convinced that the instantiation of the approach in the architecture being evaluated is appropriate for meeting the attribute-specific requirements for which it is intended.

Outputs of this step include

- The architectural approaches or decisions relevant to each high-priority utility tree scenario. The team should expect that all the approaches identified have been captured in Step 4; if not, the team should probe to find out the reason for the discrepancy. The architect should identify the approach and the components, connectors, and constraints involved.

- The analysis questions associated with each approach, geared to the quality attribute with which its scenario is associated. These questions might come from documented experience with approaches (as found in ABASs and their associated quality attribute characterizations, discussed in Chapter 5), from books on software architecture (see For Further Reading at the end of this chapter), or from the prior experiences of the assembled stakeholders. In practice all three areas are mined for questions.
- The architect's responses to the questions.
- The risks, nonrisks, sensitivity points, and tradeoff points identified. Each of these is associated with the achievement of one or more quality attribute refinements in the utility tree with respect to the quality attribute questions that probed the risk.

In effect, the utility tree tells the evaluation team where to probe the architecture (because this is a highly important factor for the success of the system), the architect (one hopes) responds with the architectural approach that answers this need, and the team can use the quality attribute–specific questions to probe the approach more deeply. The questions help the team to

- Understand the approach in detail and how it was applied in this instance
- Look for well-known weaknesses with the approach
- Look for the approach's sensitivity and tradeoff points
- Find interactions and tradeoffs with other approaches

In the end, each of these may provide the basic material for the description of a risk, and this is recorded in an ever-growing list of risks.

For example, assigning processes to a server might affect the number of transactions that server can process in a second. Thus the assignment of processes to the server is a sensitivity point with respect to the response as measured in transactions per second. Some assignments will result in unacceptable values of this response—these are risks. Finally, when it turns out that an architectural decision is a sensitivity point for more than one attribute, it is designated as a tradeoff point.

The analysis questions are not an end unto themselves. Each is a starting point for discussion and for the determination of a potential risk, nonrisk, sensitivity point, or tradeoff point. These, in turn, may catalyze a deeper analysis, depending on how the architect responds. For example, if the architect cannot characterize client loading and cannot say how priorities are allocated to processes and how processes are allocated to hardware, then there is little point in doing a sophisticated queuing or rate-monotonic performance analysis. If such questions can be answered, then the evaluation team can perform at least a rudimentary or back-of-the-envelope analysis to determine whether these architectural decisions are problematic or not vis-à-vis the quality attribute requirements they are meant to address. The analysis is not meant to be comprehensive and detailed. The key is to elicit enough architectural information to establish some link between the architectural decisions that have been made and the quality attribute requirements that need to be satisfied.

A template for capturing the analysis of an architectural approach for a scenario is shown in Figure 3.5.

For example, Step 6 might elicit from an architect the following information, shown in Figure 3.6, in response to a utility tree scenario that required high availability from a system.

As shown in Figure 3.6, based upon the results of this step the evaluation team can identify and record a set of sensitivity points and tradeoff points, risks and nonrisks. All sensitivity points and tradeoff points are candidate risks. By the end of the ATAM, each sensitivity point and each tradeoff point should be categorized as either a risk or a nonrisk. The risks, nonrisks, sensitivity points, and tradeoffs are gathered together in separate lists. The numbers R8, T3, S4, N12, and so on in Figure 3.6 simply refer to entries in these lists.

Analysis of Architectural Approach				
Scenario #: *Number*	**Scenario:** *Text of scenario from utility tree*			
Attribute(s)	*Quality attribute(s) with which this scenario is concerned*			
Environment	*Relevant assumptions about the environment in which the system resides, and the relevant conditions when the scenario is carried out*			
Stimulus	*A precise statement of the quality attribute stimulus (e.g., function invoked, failure, threat, modification . . .) embodied by the scenario*			
Response	*A precise statement of the quality attribute response (e.g., response time, measure of difficulty of modification)*			
Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
Architectural decisions relevant to this scenario that affect quality attribute response	*Sensitivity Point #*	*Tradeoff Point #*	*Risk #*	*Nonrisk #*
. . .	. . .	. . .	. . .	. . .
. . .	. . .	. . .	. . .	. . .
Reasoning	*Qualitative and/or quantitative rationale for why the list of architectural decisions contribute to meeting each quality attribute requirement expressed by the scenario*			
Architectural Diagram	*Diagram or diagrams of architectural views annotated with architectural information to support the above reasoning, accompanied by explanatory text if desired*			

Figure 3.5 Template for Analysis of an Architectural Approach

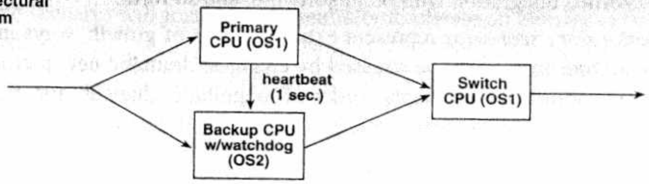
Analysis of Architectural Approach				
Scenario #: *A12*	**Scenario:** *Detect and recover from HW failure of a primary CPU*			
Attribute(s)	*Availability*			
Environment	*Normal operations*			
Stimulus	*One of the CPUs fails*			
Response	*0.999999 availability of the switch*			
Architectural Decisions	**Sensitivity**	**Tradeoff**	**Risk**	**Nonrisk**
Backup CPUs	*S2*		*R8*	
No backup data channel	*S3*	*T3*	*R9*	
Watchdog	*S4*			*N12*
Heartbeat	*S5*			*N13*
Failover routing	*S6*			*N14*
Reasoning	• *Ensures no common mode failure by using different hardware and operating system (see Risk R8)* • *Worst-case rollover is accomplished in 4 seconds as computing state takes that long at worst* • *Guaranteed to detect failure with 2 seconds based on rates of heartbeat and watchdog* • *Watchdog is simple and proven reliable* • *Availability requirement might be at risk due to lack of backup data channel (see Risk R9)*			
Architectural Diagram	 Primary CPU (OS1) heartbeat (1 sec.) Backup CPU w/watchdog (OS2) Switch CPU (OS1)			

Figure 3.6 Example of Analysis of an Architectural Approach

At the end of this step, the evaluation team should have a clear picture of the most important aspects of the entire architecture, the rationale for key design decisions that have been made, and a list of risks, nonrisks, sensitivity points, and tradeoff points.

At this point the team can now test its understanding of the architectural representations that have been generated. This is the purpose of the next two steps.

3.2.7 Step 7: Brainstorm and Prioritize Scenarios

Scenarios drive the testing phase of the ATAM. Generating a set of scenarios has proven to be a great facilitator of discussion and brainstorming, when greater numbers of stakeholders (see sidebar Stakeholders) are gathered to participate in the ATAM. Scenarios are used to

- Represent stakeholders' interests
- Understand quality attribute requirements

While utility tree generation is primarily used to understand how the architect perceived and handled quality attribute architectural drivers, the purpose of scenario brainstorming is to take the pulse of the larger stakeholder community. Scenario brainstorming works well in larger groups, creating an atmosphere in which the ideas and thoughts of one person stimulate others. The process fosters communication and creativity and serves to express the collective mind of the participants. The prioritized list of brainstormed scenarios is compared with those generated via the utility tree exercise. If they agree, great. If additional driving scenarios are discovered, this is also an important outcome.

In this step, the evaluation team asks the stakeholders to brainstorm three kinds of scenarios.

1. *Use case scenarios* represent the ways in which the stakeholders expect the system to be used. In use case scenarios the stakeholder is an end user, using the system to execute some function.
2. *Growth scenarios* represent ways in which the architecture is expected to accommodate growth and change in the moderate near term: expected modifications, changes in performance or availability, porting to other platforms, integration with other software, and so forth.
3. *Exploratory scenarios* represent extreme forms of growth, ways in which the architecture might be stressed by changes: dramatic new performance or availability requirements (order-of-magnitude changes, for example), major changes in the infrastructure or mission of the system, and so forth. Whereas growth scenarios provide a way to show the strengths and weaknesses of the architecture with respect to anticipated forces on the system, exploratory scenarios attempt to find more sensitivity and tradeoff points that appear at the stress points of the architecture. The identification of these points helps assess the limits of the system's architecture.

Stakeholders are encouraged to consider scenarios in the utility tree that have not been analyzed; those scenarios are legitimate candidates to put into the brainstorm pool. This gives the stakeholders the opportunity to revisit scenarios from Steps 5 and 6 that they might think received too little attention. This is in keeping with the spirit of Steps 7 and 8 as testing activities.

Once the scenarios have been collected, they must be prioritized. First, stakeholders are asked to merge scenarios that they believe represent the same behavior or quality concern. Then stakeholders vote for the scenarios they think are most important. Each stakeholder is allocated a number of votes equal to 30 percent of the number of scenarios,[2] rounded up. So, for instance, if there were eighteen scenarios collected, each stakeholder would be given six votes. These votes can be allocated in any way that the stakeholder sees fit: all six votes for one scenario, one vote each for six scenarios, or anything in between.

Each stakeholder casts his or her votes publicly; our experience tells us it's more fun that way, and the exercise builds unity among the participants. Once the votes are tallied, the evaluation leader orders the scenarios by vote total and looks for a sharp drop-off in the number of votes. Scenarios "above the line" are adopted and carried forth to subsequent steps. For example, a team might consider only the top five scenarios.

Figure 3.7 shows a few example scenarios from an evaluation of a vehicle dispatching system (only the top five scenarios of the more than thirty collected are shown here), along with their votes.

At this point in an evaluation, we pause and compare the result of scenario prioritization with the results of the utility tree exercise from Step 5 and look for agreement or disagreement. Using group consensus, each high-priority brainstormed scenario is placed into an appropriate leaf node in the utility tree. First, we agree on which quality attributes each scenario is addressing. In Figure 3.8, the highly ranked scenarios from Figure 3.7 are shown along with an indication of the quality attribute or attributes that each scenario affects most heavily; this helps to place the scenarios in appropriate branches of the utility tree.

When a brainstormed scenario is placed into the utility tree, one of three things will happen:

1. The scenario will match and essentially duplicate an already-existing leaf node.

#	Scenario	Votes
4	*Dynamically replan a dispatched mission within 10 minutes*	*28*
27	*Split the management of a set of vehicles across multiple control sites*	*26*
10	*Change vendor analysis tools after mission has commenced without restarting system*	*23*
12	*Retarget a collection of diverse vehicles to handle an emergency situation in less than 10 seconds after commands are issued*	*13*
14	*Change the data distribution mechanism from CORBA to a new emerging standard with less than six person-months' effort*	*12*

Figure 3.7 Examples of Scenarios with Rankings

[2] This is a common facilitated brainstorming technique.

Scenario	# Votes	Quality Attributes
4	28	Performance
27	26	Performance, modifiability, availability
10	23	Modifiability
12	13	Performance
14	12	Modifiability

Figure 3.8 Highly Ranked Scenarios with Quality Attribute Annotations

2. The scenario will go into a new leaf node of an existing branch (or, if it addresses more than one quality attribute, it will be placed into the leaves of several branches after rewording to ensure its relevance to each particular quality attribute is clear).
3. The scenario will fit in no branch of the tree because it expresses a quality attribute not previously accounted for.

The first and second cases indicate that the larger stakeholder community was thinking along the same lines as the architect. The third case, however, suggests that the architect may have failed to consider an important quality attribute, and subsequent probing here may produce a risk to document.

From the introduction to this chapter, recall that Step 7 is the first of two so-called testing steps.The utility tree generation and the scenario brainstorming activities reflect the quality attribute goals, but via different elicitation means and from the point of view of different groups of stakeholders. The scenarios elicited in Step 5 are being tested against those representing a larger group of stakeholders.

The architects and key developers created the initial utility tree. The widest possible group of stakeholders is involved in the scenario brainstorming and prioritization. Comparing the highly ranked outputs of both activities often reveals disconnects between what the architects believe to be important system qualities and what the stakeholders as a whole believe to be important. This, by itself, can reveal serious risks in the architecture by highlighting entire areas of concern to which the architects have not attended. Table 3.1 highlights the differences between Step 5 and Step 7.

Before this step, the utility tree was the authoritative repository of all the detailed high-priority quality attribute requirements from all sources. After this step, when the high-priority brainstormed scenarios are reconciled with the utility tree, it still is.

Table 3.1 Utility Trees versus Scenario Brainstorming

	Utility Trees	Scenario Brainstorming
Stakeholders	Architects, project leader	All stakeholders
Typical group size	evaluators; 2–3 project personnel	evaluators; 5–10 project-related personnel
Primary goals	Elicit, make concrete, and prioritize the driving quality attribute requirements Provide a focus for the remainder of the evaluation	Foster stakeholder communication to validate quality attribute goals elicited via the utility tree
Approach	General to specific: begin with quality attributes, refine until scenarios emerge	Specific to general: begin with scenarios, then identify quality attributes they express

Stakeholders

A multitude of people have a stake in a system's architecture, and all of them exert whatever influence they can on the architect(s) to make sure that their goals are addressed. The users want a system that is easy to use and has rich functionality. The maintenance organization wants a system that is easy to modify. The developing organization (as represented by management) wants a system that is easy to build and will employ the existing work force to good advantage. The customer (who pays the bill) wants the system to come in on time and within budget.

The architecture is where these concerns converge, and it is the architect who must weigh them carefully and mediate their achievement. It can be a very daunting task.

First of all, many of these concerns are not expressed as actual requirements. Put another way, the requirements specification for a system conveys only a fraction of the constraints that an architecture must satisfy to be acceptable. The model taught in software engineering classes, wherein the requirements document comes sailing through the transom window and lands with a thud on the architect's desk, after which he or she can confidently design the system, is just not true. This model ignores the other stakeholders for the architecture who have concerns beyond the behavior and functionality. They also levy additional goals on the architecture, and although these goals are not quite requirements, the architecture will be as much of a failure if those goals are not met.

For example, the project manager may need to utilize an otherwise-idle staff who are experts in databases and may exert pressure on the architect to include a database in the system. The presence of a recently purchased tool environment, the cost of which the management is eager to justify, exerts pressure to develop with an architectural style that can be easily handled by that environment. The need to conform to new standards, to incorporate new kinds of peripheral devices, to migrate to new versions of the operating system, or to accommodate new functionality are architectural constraints that rarely make it into a requirements document in ways other than the vague and not-very-helpful dictum, "The system shall be easy to change."

And, as we have already seen, none of the concerns expressed as a generic quality (such as "The system shall be easy to use") can stand as the basis for any sort of design decision.

Worse, many of the stakeholders' wants and needs are likely to be in direct conflict with each other. Users' desire for speed may conflict with the maintainers' desire for modifiability, for instance. The architecture serves as the focal point for all of the often-conflicting pressures manifested by the different stakeholders' concerns. All of these wants and desires converge in the architecture, and the architect is often in the position of mediating the conflicts.

Therefore, a critical step of any architecture evaluation is to elicit exactly what the goals for the architecture in fact are, and the stakeholders who own those goals are the ones from whom they are elicited. The requirements specification, if it exists, is but a starting point. The other goals, and the stakeholders who champion them, must be taken into account. This leads to one of the most important principles about architecture evaluation:

> *Active participation by the architecture's stakeholders is absolutely essential for a high-quality evaluation.*

The following table shows some of the architecture's stakeholders who might be involved in an architectural evaluation. Of course, not all are needed for every architecture evaluation; the specific context will determine which are appropriate and which are not.

Of course, it is often impractical to assemble this many people for a two- or three-day meeting. Many of the stakeholders may not even work for the organization in which the architecture is being developed. The goal is to have the important stakeholders' views represented. If, for example, the system is envisioned to have a long lifetime of evolution and modification, then you should recruit people who can speak to the kinds of modifications that the architecture should enable. And not all systems have the same set of stakeholders—perhaps performance is simply not a concern, or there will be no system administrator.

If you cannot assemble all of the architecture's stakeholders, then try to assemble people who can represent them. Having stakeholders missing or

Stakeholders for an Architecture Evaluation

Stakeholder	Definition	Interest
Producers of the System		
Software architect	Person responsible for the architecture of the system and responsible for making tradeoffs among competing quality pressures	Moderation and mediation all of the quality concerns of the other stakeholders
Developer	Coder or designer	Clarity and completeness of architecture description, high cohesion of parts, limited coupling of parts, clear interconnection mechanisms
Maintainer	Person making changes after initial deployment	Maintainability, ability to locate every place in the system that must change in response to a modification
Integrator	Developer responsible for integrating (assembling) the components	Same as developer
Tester	Developer responsible for testing the system	Integrated, consistent error-handling protocols; limited component coupling; high component cohesion; conceptual integrity
Standards expert	Developer responsible for knowing the details of standards (current and future) to which the software must conform	Separation of concerns, modifiability, interoperability
Performance engineer	Person who analyzes system artifacts to see if the system will meet its performance and throughput requirements	Understandability, conceptual integrity, performance, reliability
Security expert	Person responsible for making sure that the system will meet its security requirements	Security
Project manager	Person who allocates resources to teams, is responsible for meeting schedule and budget, interfaces with customer	Clear structuring of architecture to drive team formation, work breakdown structure, milestones and deadlines, etc.

(continued)

Stakeholders for an Architecture Evaluation (*continued*)

Stakeholder	Definition	Interest
Product-line manager or "reuse czar"	Person who has a vision for how this architecture and related assets can be (re-) used to further the developing organization's long-range goals	Reusability, flexibility
Consumers of the System		
Customer	Purchaser of the system	Schedule of completion, overall budget, usefulness of the system, meeting customers' (or market's) expectations
End user	User of the implemented system	Functionality, usability
Application builder (in the case of a product-line architecture)	Person who will take the architecture and any reusable components that exist with it and instantiate them to build a product	Architectural clarity, completeness, simple interaction mechanisms, simple tailoring mechanisms
Mission specialist, mission planner	Representative of the customer who knows how the system is expected to be used to accomplish strategic objectives; has broader perspective than end users alone	Functionality, usability, flexibility
Servicers of the System		
System administrator	Person running the system (if different from user)	Ease in finding the location of problems that may arise
Network administrator	Person administering the network	Network performance, predictability
Service representatives	People who provide support for the use and maintenance of the system in the field	Usability, serviceability, tailorability
Persons Interfacing or Interoperating with the Software		
Representatives of the domain or community	Builders or owners of similar systems or systems with which the subject system is intended to work	Interoperability

Stakeholders for an Architecture Evaluation (*continued*)

Stakeholder	Definition	Interest
System architect	Architect of the entire system; person who makes tradeoff decisions between hardware and software and who selects the hardware environment	Portability, flexibility, performance, efficiency
Device expert	Person who knows the devices with which the software must interface; can predict future trends in hardware technology	Maintainability, performance

represented with low fidelity exposes you to the risk that the architecture will not be analyzed (or will be analyzed with low fidelity) with respect to the missing stakeholders' concerns. The acceptability of that risk to you will determine whether or not you wish to proceed with the evaluation anyway.

> *The quality of a software architecture evaluation depends in very large part on the quality of the stakeholders whom you are able to assemble for it.*

We wish to call out one stakeholder in the preceding in particular: the software architect. The next principle of sound architecture evaluations is this:

> *Insist on having the architect, or the architecture team, present at the evaluation.*

Therefore, whenever crafting the list of stakeholders to be present during an evaluation, make sure the architect is first on the list.

Having the architect present is essential for several reasons. First of all, failure to identify an architect is a sure sign of trouble on a project. Second, an architecture evaluation will likely be the first time that an architect will have the luxury of having all of the stakeholders in the same room at the same time, articulating their goals for the architecture in a facilitated environment. If the architect is not there, the experience will be wasted. Third, the architect will be the one who will take away action items requiring attention in the architectural design. And fourth, somebody has to present the architecture to be evaluated and explain how it will meet the articulated goals. Who better than the architect?

Remember that you need to not just identify the *kinds* of stakeholders who should be present (or represented) but also to identify *names of specific individuals* who will serve the stakeholder roles. You can help the client identify the kinds (using the preceding table), but the client will have to assign names and ensure their participation. If the client cannot identify individuals

> and vouch for their participation in time for the evaluation, then you should call time-out until he or she can.
>
> —PCC

3.2.8 Step 8: Analyze the Architectural Approaches

After the scenarios have been collected and analyzed, the evaluation team guides the architect in the process of carrying out the highest-ranked scenarios from Step 7 on whatever architectural descriptions have been presented. The architect explains how relevant architectural decisions contribute to realizing the scenario. Ideally this activity is dominated by the architect's explanation of scenarios in terms of previously discussed architectural approaches.

In this step the evaluation team performs the same activities as in Step 6, mapping the highest-ranked newly generated scenarios onto the architectural artifacts thus far uncovered. Assuming Step 7 didn't produce any high-priority scenarios that were not already covered by previous analysis, Step 8 is a testing activity: it is to be hoped that little new information will be uncovered.

3.2.9 Step 9: Present the Results

Finally, the collected information from the ATAM needs to be summarized and presented back to the stakeholders. This presentation typically takes the form of a verbal report accompanied by slides but might, in addition, be accompanied by a more complete written report delivered subsequent to the ATAM. In this presentation the evaluation leader recapitulates the steps of the ATAM and all the information collected in the steps of the method, including the business context, the driving requirements, the constraints, and the architecture. Most important, however, is the set of ATAM outputs:

- The architectural approaches documented
- The set of scenarios and their prioritization
- The set of attribute-based questions
- The utility tree
- The risks discovered
- The nonrisks documented
- The sensitivity points and tradeoff points found

These outputs are all uncovered, publicly captured, and catalogued during the evaluation. But in Step 9, the evaluation team produces an additional output: risk themes. Experience shows that risks can be grouped together based on some common underlying concern or systemic deficiency. For example, a group of risks about inadequate or out-of-date documentation might be grouped into a risk theme stating that documentation is given insufficient consideration. A group of risks about the system's inability to function in the face of various kinds of hardware and/or software failures might lead to a risk theme about insufficient attention paid to backup capability or provision of high availability. For each risk theme, the evaluation team identifies which of

> **Sensitivities, Risks, and Nonrisks**
>
> The ATAM relies heavily on the identification of sensitivity points and risks in the architecture. It relies on sensitivity points to not only locate potential problems (risks) in the architecture but also to find the *strengths* of the architecture. One aspect of the method that is often overlooked is that we can find nonrisks as well as risks.
>
> Recall that nonrisks, like risks, are related to architectural responses stemming from architectural decisions, based on some assumed stimuli. But with nonrisks we say that the architectural decision is appropriate—the way that the architecture has been designed meets the quality attribute requirements. We want to record this information, as we record information about risks, because if these architectural decisions ever change, we must examine their effect on the nonrisk to see if it still poses no risk. For example, we might choose to store some important shared system information in a flat file located on a centrally accessible server. We know that this poses no risk because the file is small, there are no security concerns associated with accessing the information in the file, and the various programs that need to use it never attempt to access it simultaneously. If any of these assumptions ever changed—for example, if the size of the file grew dramatically, or it began to hold confidential information, or if different programs might contend for exclusive access to the file—then we would have to revisit the architectural decision. Using a flat file might now be a risk.
>
> As we stated in Chapter 2, a sensitivity point is a property of one or more components (and/or component relationships) that is critical for achieving (or failing to achieve) a particular quality attribute response. And as part of an ATAM effort every sensitivity point should be explicitly classified as a risk or a nonrisk, depending upon whether the desired response is achieved or not.
>
> But there are other potential problems that we find during an ATAM exercise, and these can stem from factors beyond just architectural decisions. During the evaluation we might uncover potential problems that are managerial or process-related in nature. We might find supplier-related problems. We might discover that funding is insecure or schedules are unreasonable or that technical decisions are being made for political reasons. These findings are called *issues* and we record those separately (see the Issues sidebar in Chapter 4).
>
> —RK

the business drivers listed in Step 2 are affected. Identifying risk themes and then relating them to specific drivers precipitates two effects. First, it brings the evaluation full circle by relating the final results to the initial presentation. This provides a satisfying closure to the exercise. Second, it elevates the risks that were uncovered to the attention of management. What might otherwise have seemed to a manager like an esoteric technical issue is now identified unambiguously as a threat to something that manager cares about.

Because the evaluation team is systematically working through and trying to understand the architectural approaches, it is inevitable that, at times, the evaluation team sometimes makes recommendations on how the architecture might have been designed or analyzed differently. These mitigation strategies may be process related (for example, a database administrator stakeholder should be consulted before completing the design of the system administration user interface), they may be managerial (for example, three subgroups within the development effort are pursuing highly similar goals and these should be merged), or they may be technical (for example, given the estimated distribution of customer input requests, additional server threads need to be allocated to ensure that worst-case latency does not exceed five seconds). However, offering mitigation strategies is not an integral part of the ATAM. The ATAM is about locating architectural risks. Addressing them may be done in any number of ways.

Table 3.2 summarizes the nine steps of the ATAM and shows how each step contributes to the outputs that the ATAM delivers after an evaluation.

3.3 The Phases of the ATAM

Up to this point we have enumerated and described the steps of the ATAM. In this section we describe how the steps of the ATAM are carried out over time. The ATAM comprises four phases, corresponding to segments of time when major activities occur.

Phase 0 is a setup phase in which the evaluation team is created and a partnership is formed between the evaluation organization and the organization whose architecture is to be evaluated. Phase 1 and Phase 2, the evaluation phases of the ATAM, comprise the nine steps presented so far in this chapter. Phase 1 is architecture-centric and concentrates on eliciting architectural information and analyzing it. Phase 2 is stakeholder-centric and concentrates on eliciting stakeholder points of view and verifying the results of the first phase. Phase 3 is a follow-up phase in which a final report is produced, follow-on actions (if any) are planned, and the evaluation organization updates its archives and experience base.

The four phases are detailed next.

Table 3.2 Steps and Outputs of the ATAM, Correlated

Outputs of the ATAM:						
Steps of ATAM:	**Prioritized Statement of Quality Attribute Requirements.**	**Catalog of Architectural Approaches Used**	**Approach- and Quality-Attribute-Specific Analysis Questions**	**Mapping of Architectural Approaches to Quality Attributes**	**Risks and Nonrisks**	**Sensitivity and Tradeoff Points**
1. Present the ATAM						
2. Present business drivers	*a				*b	
3. Present architecture		**			*c	*d
4. Identify architectural approaches		**	**		*e	*f
5. Generate quality attribute utility tree	**					
6. Analyze architectural approaches		*g	**	**	**	**
7. Brainstorm and prioritize scenarios	**					
8. Analyze architectural approaches		*g	**	**	**	**
9. Present results						

Key: ** = the step is a primary contributor to the output; * = the step is a secondary contributor.

a. The business drivers include the first, coarse description of the quality attributes.
b. The business drivers presentation might disclose an already-identified or long-standing risk which should be captured.
c. The architect may identify a risk in his or her presentation.
d. The architect may identify a sensitivity or tradeoff point in his or her presentation.
e. Many architectural approaches have standard risks associated with them.
f. Many architectural approaches have standard sensitivities and quality attribute tradeoffs associated with them.
g. The analysis steps might reveal one or more architectural approaches not identified in Step 4, which will then produce new approach-specific questions.

3.3.1 Phase 0 Activities

Phases 1 and 2 of the ATAM are where the analysis takes place and so are considered the "heart" of the method. But before Phase 1 can begin, a partnership must be established between the sponsor of the evaluation and the organization carrying it out. A statement of work must be signed and agreements arranged about times, dates, costs, and disposition of work. The evaluation team must be formed. This is the purpose of Phase 0.

Phase 0 includes all the groundwork that must be laid before an architecture evaluation can begin. Properly executed groundwork will ensure that the exercise will be a success. When the preparation phase is completed, you should be ready to begin the evaluation exercise with confidence that your client will understand what is involved, that the necessary resources will be at hand, and that the goals and outcomes of the evaluation are clear to all parties.

Phase 0 consists of two parts: establishing a partnership with the client and preparation for the evaluation phases.

Partnership

Phase 0 involves communication with the person(s) who commissioned the evaluation, whom we will refer to as the *client*. How you and the client make initial contact is beyond the scope of this book, as are any arrangement made concerning compensation for the work performed. Whatever the circumstances, we assume that you and the client have conversed about the possibility of performing an architecture evaluation. Now is the time to solidify the agreement.

The client needs to be someone who can exercise control over the project whose architecture is the subject of evaluation. Perhaps the client is a manager of the project. Or perhaps the client is someone in an organization who is acquiring a system based on the architecture. If the acquisition is a major one, the developing organization may well agree to having its architecture evaluated by outsiders. The client may or may not work for the same organization as the acquirer. In any case, it is assumed that the client has enough leverage to cause the development project to take the necessary time out so that the architecture can undergo the evaluation. It is also assumed that the client has access to a broad selection of stakeholders for the architecture.

The following issues must be resolved before the client gives the go-ahead for the evaluation to take place.

1. The client should have a basic understanding of the evaluation method and the process that will be followed. This can be handled by a briefing or by a written description of the method. This book is also a definitive source of information but may contain more than your client wishes to know. It is a good idea to make a videotape of a method briefing to give to prospective clients.
2. The client should describe the system and architecture being evaluated. This enables the evaluation leader to decide whether or not there is enough material present—that is, whether the architecture is far enough along—so that an ATAM-based evaluation will be useful. At this point, the evaluation leader will have to make a "go/no-go" decision.

3. Assuming the decision is "go," a contract or statement of work should be negotiated and signed. This lets both sides make sure that the following issues are understood:
   - Who is responsible for providing the necessary resources (such as supplies, facilities, a location, the attendance of stakeholders, the presence of the architect and other project representatives, and so on) for the evaluation
   - What is the period of performance for the evaluation a window in which the evaluation will be carried out
   - To whom will the final report be delivered and by when
   - What is the team's availability (or nonavailability) for follow-up work
4. Issues of proprietary information should be resolved. For example, the evaluation team might need to sign nondisclosure statements.

The negotiation phase is also a good time to talk to the client about the costs and benefits of architecture evaluation. Perhaps by the time the client has come to you, he or she already believes in the intrinsic value of the activity, but one of the goals of the premeeting is to share with the client any data you have about past benefits so that the client will feel confident about proceeding. Also, the client and the architect may belong to a different organization (such as when the client is acquiring the architecture from a separate development organization), in which case the client may be convinced of the value of the evaluation, but the architect and his or her organization may not be. Arming the client with cost/benefit data will help convince the developers of the value.

To increase client buy-in, you can do the following:

- Share with your client cost data and associated benefits from public sources, such as this book, and your own data from previous evaluations.
- As your repository of evaluations grows, share the (sanitized) comments about benefits with the client. Add them to your method overview presentation.
- Cite any instances you can of one organization asking for multiple evaluations.

Preparation

The preparation half of Phase 0 consists of

- Forming the evaluation team
- Holding an evaluation team kickoff meeting
- Making the necessary preparations for Phase 1

If your organization does not have a standing evaluation team, then you will have to form one. This means choosing individuals to take part in the exercise. Finding the individuals, scheduling their time, and clearing their participation with their respective supervisors are all part of forming the team.

Once you have assembled a team (or even if you have a standing team) you will need to assign each member a role in the upcoming evaluation. It is always a good idea to rotate the roles among team members from exercise to exercise; this way there are more people available to perform any given role in the event of a personnel shortage. The table below defines those roles and responsibilities for an architecture evaluation using the ATAM.

There is not necessarily a one-to-one correspondence between people and roles: a person may assume more than one role, or more than one person may collectively carry out a role. It is up to the evaluation team leader to assign people to roles (and roles to people). These rules of thumb may help.

- The minimum complement for an evaluation team should be four people.
- One person can usually carry out the process observer, timekeeper, and questioner roles simultaneously.
- The team leader's responsibilities occur primarily outside the evaluation exercise; hence, that person can double up on any other in-exercise role. The team leader is often the evaluation leader, because both tend to be senior people, but not always.

Table 3.3 Evaluation Team Individual Roles and Responsibilities

Role	Responsibilities	Desirable Characteristics
Team leader	Sets up the evaluation; coordinates with client; makes sure client's needs are met; establishes evaluation contract. Forms the evaluation team. In charge of seeing that final report is produced and delivered (although the writing may be delegated).	Well-organized, with managerial skills. Good at interacting with client. Able to meet deadlines.
Evaluation leader	Runs evaluation. Facilitates elicitation of scenarios; administers scenario selection/prioritization process; facilitates evaluation of scenarios against architecture. Facilitates on-site analysis.	Comfortable in front of an audience. Excellent facilitation skills. Good understanding of architectural issues. Practiced in architecture evaluations. Able to tell when protracted discussion is leading to a valuable discovery, or when it is pointless and should be redirected.
Scenario scribe	Writes scenarios on flip chart or whiteboard during scenario elicitation process. Carefully captures agreed-upon wording of each scenario and doesn't let discussion continue until exact wording is captured.	Good handwriting. Willingness to be a stickler about not moving on before an idea (a scenario) is captured. Able to quickly absorb and distill the essence of technical discussions.
Proceedings scribe	Captures the proceedings in electronic form on a laptop computer or in-room workstation. Captures the raw scenarios. Captures the issue(s) that motivated each scenario, as this is often lost in the wording of the scenario itself. Captures the resolution of each scenario when applied to architecture(s). Generates a printed list of adopted scenarios for hand-out to all participants.	Good, fast typist. Well-organized to allow rapid recall of information. Good understanding of architectural issues. Able to assimilate technical issues quickly. Must not be afraid to interrupt the flow of discussion (at opportune times) to test understanding of an issue, so that the appropriate information is captured.
Timekeeper	Helps the evaluation leader stay on schedule. Helps control amount of time devoted to each scenario during evaluation phase.	Willingness to brazenly interrupt discussion to call time.
Process observer	Keeps notes on where the evaluation process itself could be improved or deviated from the plan. Usually a silent observer, but may make process-based suggestions to the evaluation leader, discretely, during the evaluation. After evaluation, reports on how the process went and what lessons were learned for future improvement. Also responsible for reporting experience to architecture evaluation team at large.	Thoughtful observer. Knowledgeable in the evaluation process. Should have previous experience in the architecture evaluation method.
Process enforcer	Helps the evaluation leader remember and carry out the steps of the evaluation method.	Should be fluent in the steps of the method. Willing and able to provide guidance to the evaluation leader in a discrete manner.
Questioner	Raises issues of architectural interest that perhaps the stakeholders have not considered.	Good architectural insights; good insights into needs of stakeholders. Experience with systems in similar domain. Not afraid to bring up possibly contentious issues and pursue them doggedly. Familiarity with attributes of concern.

(continued)

Table 3.3 Evaluation Team Individual Roles and Responsibilities (*continued*)

- Questioners should be chosen so that the appropriate spectrum of expertise in qualities of interest (performance, reliability, maintainability, and so on) can be brought to bear on the system being evaluated.

In Chapter 10, we will discuss setting up a standing evaluation team in which individuals are rotated on and off and trained so as to spread the expertise throughout the organization. This practice has many organizational benefits and is a major step in adopting mature architecture-based development processes within an organization. But more to the immediate point, it obviates the problem of recruiting individuals willing to serve on the team: the team is already formed.

After the team is formed, a kickoff meeting should be held in which all available knowledge about the evaluation should be shared and team roles assigned. Preparations for Phase 1 include taking care of the myriad logistical details to assure that everyone shows up at the right time and place prepared to work—this means the project's representatives in addition to the evaluation team.

3.3.2 Phase 1 Activities

In Phase 1, the ATAM team meets with a subset of the team whose architecture is being evaluated, perhaps for the first time. This meeting has two concerns: organization of the rest of the analysis activities and information collection. Organizationally, the manager of the architecture team needs to make sure that the right people attend the subsequent meetings, that people are prepared, and that they come with the right attitude—a spirit of nonadversarial teamwork.

With a small group of key people, Phase 1 concentrates on Steps 1 through 6. The evaluation team presents the ATAM method; a spokesperson for the project presents the business drivers; the architect presents the architecture. The group catalogs architectural approaches and builds the utility tree. The high-priority utility tree scenarios then form the basis for analysis.

Besides carrying out the six steps, the evaluation team has another purpose to fulfill during Phase 1. It needs to gather as much information as possible to determine

- Whether the remainder of the evaluation is feasible and should proceed. If not, then Phase 1 is an opportune cutoff point, before the larger group of stakeholders is assembled for Phase 2.
- Whether more architectural documentation is required and, if so, precisely what kinds of documentation and how it should be represented. If this is the case, the evaluation team can work with the architecture team during the hiatus between Phase 1 and Phase 2 to help them "catch up" so that Phase 2 can begin on a complete note.
- Which stakeholders should be present for Phase 2. An action item at the end of Phase 1 is for the evaluation's sponsor to make sure that the right stakeholders assemble for Phase 2. (See the Stakeholders sidebar on page 63.)

There is a hiatus between Phase 1 and Phase 2 in which ongoing discovery and analysis are performed by the architecture team, in collaboration with the evaluation team. As we described earlier, the evaluation team does not build detailed analytic models during this phase, but they do build rudimentary models that will give the evaluators and the architect sufficient insight into the architecture to make the Phase 2 meeting more productive. Also, during this hiatus the final composition of the evaluation team is determined, according to the needs of the evaluation, availability of human resources, and the schedule. For example, if the system being evaluated is safety critical, a safety expert might be recruited, or if it is database-centric, an expert in database design could be recruited to be part of the evaluation team.

3.3.3 Phase 2 Activities

At this point, the evaluation team will have understanding of the architecture in sufficient detail to support verification of the analysis already performed and further analysis as needed. The appropriate stakeholders have been identified and have been given advance reading materials such as a description of the ATAM, perhaps some scenario examples, and system documentation including the architecture, business case, and key requirements. These reading materials aid in ensuring that the stakeholders know what to expect from the ATAM. Now the stakeholders are gathered for Phase 2, which can involve as few as 3 to 5 stakeholders or as many as 40.[3]

Since there will be a broader set of stakeholders attending Phase 2 and since a number of days or weeks may have transpired between the first and second meetings, Phase 2 begins with an encore of Step 1: Present the ATAM. After that, Steps 2 through 6 from Phase 1 are recapped for the new stakeholders. Then the evaluation proceeds by carrying out Steps 7, 8, and 9.

Table 3.4 lists the steps and typical categories of attendees for Phase 1 and Phase 2.

A Typical ATAM Agenda for Phase 1 and Phase 2

In Figure 3.9 we show an example of a typical ATAM agenda for Phases 1 and 2. Each activity in this figure is followed by its step number, where appropriate, in parentheses. While the times here need not be slavishly followed, this schedule represents a reasonable partitioning of the available time in that it allows more time on those activities that experience has shown to produce more results (in terms of finding architectural risks).

[3] Strive for about 10–15 stakeholders. A much larger crowd than that is feasible but will require excellent facilitation skills on the part of the evaluation leader and more time than is shown in the sample agenda in Figure 3.9.

Table 3.4 ATAM Steps Associated with Stakeholder Groups

Step	Activity	Participants for Phase 1	Participants for Phase 2
1	Present the ATAM	Evaluation team and project decision makers	Evaluation team, project decision makers, and all stakeholders
2	Present business drivers		
3	Present architecture		
4	Identify architectural approaches		
5	Generate quality attribute utility tree		
6	Analyze architectural approaches		
7	Brainstorm and prioritize scenarios	N/A	
8	Analyze architectural approaches		
9	Present results		

3.3.4 Phase 3 Activities

On the back end of the ATAM, the final report (if called for by the agreement between the evaluation organization and the evaluation client) must be written and delivered. But equally important from the point of view of maintaining an ATAM capability, repositories of artifacts must be updated, surveys and effort measures taken, and the evaluation team debriefed to try to identify ways in which the method could be improved. Phase 3 is the follow-up phase.

In Phase 3, the following tasks must be accomplished:

1. Produce the final report.
2. Collect data for measurement and process improvement.
3. Update the artifact repositories.

Producing the Final Report

If the contract with the client includes a written final report, it is produced during Phase 3. Producing the final report is a matter of cataloging (a) what you did, (b) what you found, and (c) what you concluded. By using a standard report template (such as the one outlined on page 203 in Chapter 6) and assigning responsibility for specific sections to team members at the start of the evaluation, writing the report can be accomplished quickly and efficiently.

Start	Activity	
8:30 am	Introductions/ATAM Presentation (Step 1)	Phase 1
10:00	Present Business Drivers (Step 2)	
10:45	Break	
11:00	Present Architecture (Step 3)	
12:00	Identify Architectural Approaches (Step 4)	
12:30 pm	Lunch	
1:45	Generate Quality Attribute Utility Tree (Step 5)	
2:45	Analyze Architectural Approaches (Step 6)	
3:45	Break	
4:00	Analyze Architectural Approaches (Step 6)	
5:00 pm	Adjourn	
		Hiatus
Day 1		Phase 2
8:30 am	Introductions/ATAM Presentation (Step 1)	
9:15	Present Business Drivers (Step 2)	
10:00	Break	
10:15	Present Architecture (Step 3)	
11:15	Identify Architectural Approaches (Step 4)	
12:00	Lunch	
1:00 pm	Generate Quality Attribute Utility Tree (Step 5)	
2:00	Analyze Architectural Approaches (Step 6)	
3:30	Break	
3:45	Analyze Architectural Approaches (Step 6)	
5:00 pm	Adjourn for the Day	
Day 2		
8:30 am	Introductions/Recap ATAM	
8:45	Analyze Architectural Approaches (Step 6)	
9:30	Brainstorm Scenarios (Step 7)	
10:30	Break	
10:45	Prioritize Scenarios (Step 7)	
11:15	Analyze Architectural Approaches (Step 8)	
12:30 pm	Lunch	
1:30	Analyze Architectural Approaches (Step 8)	
2:45	Prepare Presentation of Results/Break	
3:30	Present Results (Step 9)	
5:00 pm	Adjourn	

Figure 3.9 A Sample ATAM Agenda for Phases 1 and 2

The Two Faces of the ATAM

The Strength of Scenarios

The predecessor of the ATAM is the SAAM (Software Architecture Analysis Method), which you will read about in Chapter 7. For a long while I thought SAAM was short for Scenario-based Architecture Analysis Method. It could have been, since scenarios are the root of its success.

The SAAM involves facilitating a scenario brainstorming session to generate a list of scenarios. A subset of the scenarios is then used to illuminate the architecture by identifying the components that would be involved if the architecture were actually able to execute the scenario. If the architecture cannot support the scenario, the exercise determines which components will have to change in order to execute the scenario. Some informal analysis is also performed, for example, by looking for components that are involved in "executing" many disparate scenarios but perhaps shouldn't be.

The SAAM is a successful method primarily for two reasons:

1. The brainstorming and prioritization of scenarios generally foster a level of stakeholder interaction and cooperative creative thinking that has never occurred before.
2. The generated collection of scenarios often is the best representation of system requirements that the stakeholders have seen up to that point.

The participating stakeholders generally feel that they learn something about the architecture and possibly each other. Moreover, in many cases, a SAAM-based evaluation is the first time that all of the stakeholders are assembled in the same place at the same time.

Clearly these were benefits that we wanted to retain when creating the ATAM. One of the faces of the ATAM is, in effect, the SAAM.

Putting "Analysis" in the Analysis Method

When we set out to create the ATAM, one of our goals was to strengthen the "analysis" aspect of the software evaluation. In fact, my attitude (despite the success of the SAAM) was that the ATAM should be predominantly about analysis and that scenario generation and stakeholder interaction was too touchy-feely. The real benefits accrue when you collect hard-core architecture data (process execution times, process priorities, details of encryption strategy, estimates of component failure rates, places where coupling between components is too strong, etc.) and then use some analytic methods to draw some firm engineering conclusions.

I then participated in my first ATAM-based evaluation exercise. Many stakeholders were present, but one "participant" was conspicuously absent: the

architecture. However, what I found to be most amazing was that nobody seemed to miss it. Despite the absence of hard-core architectural information, we successfully executed the ATAM and the customer derived significant benefits from the experience. The participants felt that the discussion generated as a result of brainstorming scenarios and subsequently recalling (and sometimes creating) the architecture in real time was very valuable. Once again, scenarios carried the day. We had actually carried out a multi-attribute SAAM-based evaluation exercise under the guise of the ATAM.

Well, I became a believer in scenarios. However, the ATAM team still felt that the analysis aspects of the ATAM could and should be strengthened, but we needed additional methodological machinery that would enable us to carry out analysis in real time in the presence of many stakeholders.

Our solution was to incorporate the concept of architectural approaches (as discussed in Chapter 1) into the ATAM. The core of the idea was that approaches or patterns would be helpful in illuminating the "shape" of the architecture, whereas scenarios illuminated only selected paths through the architecture. We felt that understanding the shape (or shapes) of the architecture would be very helpful in analyzing its properties since different approaches are important for achieving different attribute-specific requirements. The new and improved method would work something like this:

- Identify the attributes that are key to the success of the system (using utility trees).
- Determine the architectural approaches that are used within the architecture to meet those attribute requirements.
- Ask attribute-specific questions (derived from the various attribute characterizations) to understand the extent to which those approaches are suitable for meeting the attribute-specific requirements.

Armed with our new methodological machinery, we tried out our new version of the ATAM. It proved to be somewhat successful. We found some approaches and many risks, but the analysis that we performed was cursory at best. Moreover, some of the stakeholders were confused, asking questions like:

- "What's an architectural approach and how do I go about finding one?"
- "Is portability an aspect of modifiability?"
- "Why don't we add manageability to the utility tree?"

Some of our other observations were:

- When generating the utility tree we found that the most vocal folks dominated the discussion.
- No one quite understood attribute-specific questions, so we ended up generating most of those ourselves.

- It wasn't clear why we needed both utility trees and scenarios. They seemed redundant.

This mode of operation was not amenable to large-group interaction and required more up-front preparation. It seemed better suited for a smaller group centered around the architects. Even with a more robust set of steps in our method, eliciting detailed architectural information and carrying out analysis in real time in the presence of many stakeholders was inherently awkward and difficult.

Two Faces Are Better Than One

We needed a new mode of interaction but didn't want to sacrifice the benefits of the old mode. That's when we decided that two faces are better than one. The same set of steps can be used at different times with different emphases. One face is very much architecture-centric and concentrates on eliciting architecture information and analyzing the architecture. The second face is very much stakeholder-centric and concentrates on eliciting stakeholder points of view and verifying the results of the first phase. These two modes of interaction are compared in the following table.

Two Faces of the ATAM

Face 1 for Phase 1	Face 2 for Phase 2
Few stakeholders (2–4)	Many stakeholders (10–15)
Architecture-centric	Stakeholder-centric
Solution-oriented	Problem-oriented
Analysis-oriented	Verification-oriented
Fosters an understanding of architecture	Fosters stakeholder interaction
Scenarios primarily used in creating utility tree	Scenarios primarily used to verify utility tree
High bandwidth, informal technical conversations	Organized meeting

—*MHK*

Collecting Data

Each evaluation provides a convenient, inexpensive opportunity to collect data so that you can improve your ideas about the costs and benefits of performing evaluations, and also to collect participants' impressions about what worked particularly well and what could stand to be improved.

Data comes from two sources: the evaluation team and the client. In both cases, you should collect improvement data (ideas about what worked particularly well and what could have been improved) and cost/benefit data. The client may not recognize benefits until well after the evaluation, so we recommend a follow-up survey to be taken about six months after the evaluation to gauge the longer-term effects.

We recommend sending out five short surveys:

1. An improvement survey to the participants, asking their impressions of the evaluation exercise
2. An improvement survey to the team members, asking for their impressions of the exercise
3. A cost survey to the client
4. A cost survey to the evaluation team
5. A long-term benefits survey to the client

It will help to categorize the cost data in terms of before-exercise activities, during-exercise activities, and post-exercise activities. Examples for many of these surveys may be found in Chapter 10.

Updating the Artifact Repositories

You should maintain repositories of the artifacts you used or produced during each previous evaluation. These will serve you during future evaluations.

In addition to recording the cost and benefit information, store the scenarios that you produced. If future systems that you evaluate are similar in nature, you will probably find that the scenarios that express the architecture's requirements will converge into a uniform set. This gives you a powerful opportunity to streamline the evaluation method: you can dispense with the scenario brainstorming and prioritization steps of the ATAM altogether and simply use the standard scenario set for your domain. The scenarios have in some sense graduated to become a checklist, and checklists are extremely useful because each architect in the developing organization can keep a copy of the checklist in his or her desk drawer and make sure the architecture passes with respect to it. Then an evaluation becomes more of a confirmation exercise than an investigatory one. Stakeholders' involvement becomes minimal—as long as you have confidence in the applicability and completeness of the checklist with respect to the new system under evaluation—thus reducing the expense of the evaluation still further.

Besides the scenarios, make a list of the analysis questions you used; these are the evaluation team's best tools, and growing your toolbox will make future evaluations easier and give you something to show to newly added team members as part of their training.

Add participants' comments to a repository as well. Future evaluation leaders can read through these and gain valuable insights into the details and idiosyncrasies of past evaluations. These exercise summaries provide excellent training material for new evaluation leaders.

Finally, keep a copy of the final report, sanitized if necessary to avoid identifying the system or cleansed of incriminating remarks. Future evaluation teams will appreciate having a template to use for the reports they produce.

3.4 For Further Reading

As this book was going to press, an initial draft of a training course on the ATAM was being alpha-tested. You can watch for details at the SEI's architecture tradeoff analysis Web site [SEI ATA].

The ATAM's analysis steps are based on questions about architectural approaches and quality attributes. The former come from descriptions of architectural styles in books [Bass 98, Shaw 96, Buschmann 96, Schmidt 00], and those chapters of Smith and Williams that deal with architecting performance-critical systems [Smith 01]. Quality attribute questions come from resources on performance evaluation [Klein 93, Smith 01], Markov modeling for availability [Iannino 94], fault tolerance [Jalote 94], reliability [Lyu 96], security [SEI NSS], and inspection and review methods (such as the SAAM, described in Chapter 7) for modifiability.

To read an interesting treatment of quality attribute requirements and their relationship to design decisions, see Chung et al. [Chung 00]. They refer to an early paper by Boehm et al. [Boehm 76] that presents a tree of software quality characteristics very similar to the utility trees presented in this chapter.

3.5 Discussion Questions

1. Prepare a short, informal presentation on the ATAM and present it to your colleagues.
2. Think of a software system in your organization whose architecture is of some interest. Prepare a presentation of the business drivers for this system, using the template given in this chapter.
3. What do you suppose the quality attributes of interest are for this system? Sketch a utility tree for the system. For each quality attribute of interest, write a refinement of it, and then write a scenario or two that make the concern concrete.

4. If you were going to evaluate the architecture for this system, who would you want to participate? What are the stakeholder roles associated with this system, and who could you get to represent those roles?
5. For each stakeholder role you identified, write a couple of scenarios that represent that role's point of view of the system.
6. How do the quality-attribute-focused scenarios you wrote for question 3 compare with the stakeholder-role-focused scenarios you wrote for question 5? Do they cover the same or different issues?
7. Pick a few of the scenarios you wrote for question 5 and try to understand how the architecture you've selected would respond to them. Use the analysis template given in this chapter.
8. Think of a system in your organization that is not yet ready for an architecture evaluation. What is it about the system that made you choose it? Can you generalize your answer to establish criteria for whether or not a project is ready for an architecture evaluation?