Reusing Patterns through Design Refinement

Jason O. Hallstrom[1] and Neelam Soundarajan[2]

[1] School of Computing, Clemson University, jasonoh@cs.clemson.edu
[2] Computer Sc. & Eng., Ohio State University, neelam@cse.ohio-state.edu

Abstract. Refinement concepts, such as procedural and data refinement, are among the most important ideas of software engineering. In this paper, we investigate the idea of *design refinement*, the process of refining a set of design patterns to arrive at application-specific design components, and ultimately, to system implementations. The approach also enables designers to refine a given pattern to arrive at more *specialized* versions of that pattern —*sub-patterns*— thus enabling the creation of *pattern hierarchies*.

We present three contributions: (*i*) We explore the concept of design refinement and consider what it means for such a refinement to be *correct*, in the sense of being faithful to the pattern being refined. (*ii*) We describe a two-part formalism for documenting patterns and subpatterns. A pattern *contract* captures the requirements and behavioral guarantees associated with a given pattern, while a *subcontract* captures the ways in which the pattern is specialized for use in a particular application or sub-pattern. Contracts and subcontracts serve as the basis for validating the correctness of a given refinement. (*iii*) We consider how related patterns may be organized into suitable hierarchies based on the notion of design refinement. We focus on variations of the standard *Observer* pattern as examples. A key feature of our formalism is that while it enables us to specify patterns and subpatterns precisely, it allows us to do so without compromising their *flexibility*.

1 Introduction

Refinement has been a central theme in software engineering since the inception of the field. Development techniques based on procedural refinement and data refinement provide a powerful set of methods for designing and implementing software. Equally important, they provide a foundation for ensuring software correctness. For each refinement technique, suitable reasoning methods and/or calculi [1, 2] have been developed to help software practitioners validate the correctness of their refinement steps. The result has been a dramatic improvement in software quality.

Our work is based on the observation that there is another form of refinement, *design refinement*, that has become increasingly important during the past decade as the use of design patterns has become ubiquitous in software practice. Design refinement corresponds to the process of transforming a set of design patterns into system design components, and ultimately, to system implementation components that exhibit specific behavioral properties — provided that the refinement steps applied respect the requirements dictated by the pattern. In this paper, we investigate the principles of design refinement and explore its

application in capturing hierarchies of related patterns in a manner that enables designers to reuse the effort involved in understanding them. We additionally consider how to ensure that the particular refinement steps applied in *specializing* a pattern satisfy the requirements associated with its correct usage.

While there is extensive literature documenting various aspects of patterns and the advantages of using them (*e.g.*, [3–5]), questions related to precisely specifying the requirements associated with applying patterns and associated techniques for checking if those requirements are met —i.e., ensuring *design correctness*— have not been fully addressed. Our goal is to develop such techniques. In our approach, the requirements that must be met when applying a pattern and the consequent behaviors that are expected as a result are expressed in the form of a *pattern contract*. Details concerning the specialization of the pattern as used in a particular system are expressed in the form of a corresponding *pattern subcontract*.

While pattern formalization can be expected to provide the usual benefits, such as eliminating ambiguity and serving as the basis for ensuring correctness, the process runs the risk of compromising pattern *flexibility* [3]. This is a serious concern; much of the power of patterns and the driving force behind their broad adoption derives from the flexibility they afford in applying them. The notion of *abstraction concepts*, an essential part of formalism, helps preserve this flexibility while simultaneously achieving specification precision. Each abstraction concept corresponds to a dimension of flexibility that must be preserved. Indeed, the process of identifying these abstraction concepts can help to identify latent dimensions of flexibility missing from standard pattern descriptions [6].

Thus, in our approach, a pattern contract specifies the requirements that must be satisfied to ensure the correct application of a given pattern, with the abstraction concepts used in its definition allowing for appropriate variations based on the needs of particular systems. The design refinement process effectively *"pins down"* these variations by providing suitable definitions for the abstraction concepts, while ensuring that the requirements dictated by the contract are satisfied. These definitions are supplied in a corresponding subcontract. To summarize, the information contained in a pattern contract applies to all possible uses of a given pattern, while the information contained in a subcontract captures how the pattern was specialized for use in a given application.

In some cases, however, it may be desirable to leave some of the abstraction concepts undefined, with the corresponding flexibility dimensions unbound. In this case, the subcontract will not capture a pattern application, but rather, a more specialized version of the original pattern — a *sub-pattern*. The contract for this new pattern is formed by the original contract as specialized by the subcontract for this refinement. This approach introduces an interesting possibility: Patterns related through a series of refinements can be classified in the form of a *pattern hierarchy*. The benefits of doing so are two-fold. First, pattern hierarchies can highlight the interconnections among related patterns, aiding developers in the pattern selection process. Second, pattern hierarchies enable designers to reuse the reasoning effort involved in understanding a given pattern

when reasoning about sub-patterns of that pattern. We illustrate these points by considering variations on the standard *Observer* pattern [4]. Although some of the variations in the resulting hierarchy have been documented in the literature, others, equally natural from the point of view of design refinement, have not.

The work reported in this manuscript represents a substantial revision and extension of our earlier work in pattern specification [6, 7]. Although the earlier work was also based on the idea of pinning down flexibility dimensions when documenting pattern applications, the kinds of abstraction concepts used in the formalism were limited. Most important, the formalism did not support variation in the interaction sequences among participating objects. Hence, the associated flexibility was also limited. Further, and partly as a result of this, the formalism could not help identify or characterize relations among patterns, nor organize them into suitable hierarchies.

Paper Organization. The remainder of the manuscript is organized as follows. Section 2 surveys related work in pattern formalization. Section 3 introduces the principles of design refinement, including the three types of abstraction concepts at its core. Section 4 summarizes the basic structure of pattern contracts and subcontracts. Section 5 demonstrates the principles of design refinement by constructing a hierarchy of patterns based on the standard Observer pattern. Finally, Section 6 concludes with a summary of contributions.

2 Related Work

A number of authors have investigated issues related to pattern formalization. Structural properties have been an important focus. Eden [8, 9] presents an approach to specifying the structural properties of patterns using a higher-order logic notation. Each set of pattern formulae specify the participating classes, methods, and inheritance hierarchies, and the corresponding relations among them. Kim and Carrington [10] present an *Object-Z*-based formalization of patterns using *role concept*s. Each role concept describes a pattern participant, such as a class, class feature, or like element. The resulting formalizations capture, in Object-Z, the structural relations among role concepts. Sunye *et al.* [5] and Dong [11] consider UML extensions used to model the structural aspects of patterns. Lano [12] also focuses on structural issues, using model transformations to formalize patterns. His work shows how a pattern can be viewed as a transformation from a given set of classes to another set of classes with the desired pattern properties. In contrast to the work of these authors, our focus is on *behavioral* properties — which are not readily captured using any of the above approaches.

Mikkonen *et al.* [13, 14] focus on behavioral properties using an action system notation that abstracts over the flow of control among participants. *Superposition* is used to support pattern refinement. And while the approach has been shown to be useful in reasoning about the temporal aspects of pattern behavior, the flexibility enabled by our abstraction concepts is richer. It is worth noting that Taibi and Ngo [15] combine the action system approach of Mikkonen *et al.* with the higher-order logic approach of Eden. While the resulting formalism is

more comprehensive, the behavioral portion of the formalism suffers the same flexibility limitations as Mikkonen *et al.* 's approach.

Closest to our work is that of Helm *et al.* [16], published before the seminal patterns book [4]. While the authors consider some structural issues, they focus on capturing behavioral properties. The specification notation includes support for refining object interactions and for arriving at application-specific behaviors. But the formalism's expressivity is limited. For example, the notion of a call sequence as a mathematical object is underdeveloped. It is impossible, for instance, to quantify over a call sequence to require that a particular method be invoked exactly once. There is also nothing similar to our use of *concept constraints* to prevent incorrect concept refinements. Nor can conditions be imposed on behaviors of methods not named in the pattern being specified. As a result, these *other* methods might nullify behaviors implemented by the named methods.

Before concluding this section, it is interesting to note that aspects of our approach are related to important issues identified by authors who use an informal approach to documenting patterns. According to Buschmann *et al.* [3], "You should be able to reuse the pattern in many implementations, but so that its essence is still retained. ... After applying a pattern, an architecture should include a particular structure that provides for the roles specified by the pattern, but adjusted and tailored to the specific needs of the problem at hand." What is the *essence* of a pattern and what types of "adjusting and tailoring" of roles are allowed? The answer to the former question is provided by pattern contracts, the answer to the latter by the notion of design refinement.

We should also mention the work on *generative reuse* [17–19]. Although not based on design patterns, the type of refinement that underlies this work is, in some ways, similar to design refinement. Hence our approach may also be applicable to reasoning about generative software.

3 Design Refinement and Abstraction Concepts

A key benefit of any refinement-based approach is the *flexibility* it provides in the form of abstractions that may be realized in various ways. While design refinement builds upon the ideas of procedural and data refinement, it affords much greater flexibility via more powerful types of abstractions that are unique to patterns. These abstractions can be classified into *structural abstraction*, *state-relation abstraction*, and *interaction abstraction*.

Consider the standard *Observer* pattern [4], which defines two *roles*, Subject and Observer. The pattern's intent is to maintain consistency between the state of the object playing the Subject role and the state(s) of the object(s) playing the Observer role. The subject maintains a set, obs, of references to the observers *attached* to the subject. Subject provides attach() and detach() methods for *attaching* and *detaching* observers, respectively. The subject must also provide a notify() method, which must be invoked whenever there is a *significant* change in the subject's state. notify() is required to invoke update() on each attached observer, which must in turn update the observer's state to make it *consistent* with the new state of the subject.

In a system built using Observer, a developer need not implement classes named Subject and Observer; application-specific names are likely to be more appropriate. Role methods, such as update() and notify(), may also be suitably renamed. Thus, in the example that Gamma *et al.* [4] consider, the subject is a *spreadsheet*, the observers being windows, each displaying the information in the spreadsheet in different formats such as a bar graph, pie chart, etc. The corresponding classes and their methods will be named appropriately. Further, in implementing application-specific versions of update() and notify(), the corresponding method signatures need not match those prescribed by the pattern. An application might, for instance, require additional parameters as part of the update() signature to pass state components from the object playing the Subject role to those playing the Observer role. *Structural abstraction* affords this flexibility. The *role maps*, corresponding to the various roles of the pattern, in the pattern subcontract for a given system, will specify the details of these refinements.

Consider now the update() method defined by the Observer role. As noted above, when this method is invoked on an observer, it must make the observer's state *consistent* with the current state of the subject. But what precisely does this mean? One possibility is that the observer makes a *copy* of the subject's state. While this is what some standard descriptions of the pattern suggest, it is inappropriate if an observer needs *partial* information about the subject. It is even possible that in an application, instances of two different classes, both playing the Observer role, might be simultaneously attached to a subject and maintain information about different aspects of the subject's state. The solution is to treat the notion of *consistent* as a *state-relation abstraction concept* —henceforth *relation abstraction concept*— between the states of the subject and the observers. The definition of the *Consistent()* concept will be tailored to suit the needs of particular applications or sub-patterns. Another relation abstraction used in specifying this pattern corresponds to the notion of *significant modification* in the state of the subject. The relation, *Modified()*, is defined between two states of the subject and used to determine which subject state changes trigger calls to notify(). The pattern contract will require that if *Modified*(s_1, s_2) is *true* (respectively, *false*), and the subject's state changes from s_1 to s_2, then notify() must (respectively, need not) be invoked. The subcontract for a system built using the pattern will provide definitions, applicable to that system, for both concepts.

Finally, consider the implementation of the notify() method. When notify() is called, it is required, according to standard descriptions, to invoke update() on each attached observer. While this strategy will achieve consistency with all the observers, there are other ways to accomplish this goal. For example, the observers might be arranged in a *chain*, with each observer maintaing a reference to the next. When update() is invoked on a given observer, it would then update its state and propagate the call to its successor. In this case, notify() need only invoke update() on the first observer in the chain. Alternately, the observers might be arranged in *clusters*, with one member of each cluster responsible for invoking update() on the others. In this case, notify() must invoke update() only on the designated cluster head within each cluster.

One could argue that such variations are not allowed by the Observer pattern, given its standard descriptions. But that is simply an issue of terminology. One could introduce a new pattern, *General Observer*, which only requires that notify() invoke update() on appropriate observers to ensure consistency of the entire observer set. Then Standard Observer and the variations described above would be legitimate refinements of that pattern. We call this type of refinement *interaction refinement* since it is the sequence of interactions among the objects that is being refined. In the pattern contract for Observer, *interaction abstraction concepts* are used to capture these points of flexibility. The subcontract for an application would provide definitions for these concepts applicable to that system. Similarly, the subcontract for a sub-pattern such as Standard Observer would provide definitions for these concepts — *without*, however, providing definitions for the other abstraction concepts, such as *Consistent()*.

4 Contracts and Subcontracts

Suppose that a system S is constructed using a pattern P. During the execution of S, there will be zero or more groups of objects interacting according to P. Each such group, p_i, is an *instance* of P, and each object in p_i is enrolled to play a role R in P. Further, each such object may be simultaneously enrolled in other instances. We use a *ghost* variable, players_i[], to denote the set of objects currently enrolled in p_i.

P's contract will include a *role contract* for each role. Role R's contract consists of an abstract data model for R and pre and post specifications of its methods. If a class C of S plays role R in some p_i, the subcontract will specify how the state of C maps to R's model and how the methods of C map to the methods of R. Correctness of this refinement requires that these methods of C, under the mappings specified in the role contract, satisfy the corresponding method specifications in the role contract.

R's contract will include an others clause that must be satisfied by any remaining (unmapped) methods of C to prevent those methods from violating the intent of the pattern. The role contract also includes *enrollment* and *disenrollment* clauses that specify how objects enroll in and dis-enroll from the role, respectively; we omit these details. Finally, P's contract will specify an *invariant* over the objects in the players[] array for each p_i. Ensuring appropriate relations between these objects is the purpose of applying P, thus the invariant is a key part of the pattern contract. The specifications of the role methods and invariant will be in terms of the models of P's roles and will include clauses involving the relation abstraction concepts of P, representing some of the ways in which P can be refined.

With each p_i, we associate an *instance trace* τ_i, a ghost variable that records information about the method invocations involving objects in p_i. At runtime, when such a call is made, an element is added to τ_i that records the method name, the identity of the target object, the calling object, and any parameter/return values; the states of the caller and callee are also included in the record. A similar post-conditional record is added to τ_i when the invocation completes.

Between the pre record and the post record, additional records may be added to τ_i, corresponding to calls made from the original method, calls from within *those* methods, etc. As long as the calls involve objects in p_i, they will be recorded on τ_i. These traces provide a *pattern-centric* view of the object interactions within S. For example, examining the records in τ_i that correspond to methods that enroll/dis-enroll objects in various roles yields the set of objects currently in p_i.

Both the pattern invariant and the role methods defined by P's contract may include conditions on τ_i. For example, the role method specification of `notify()` in the Standard Observer contract will require that upon completion, τ_i be extended by calls to `update()` on each `observer`. In general, these clauses will involve the *interaction abstraction concepts* of P, representing the ways in which the interactions of P may be refined. The contract will typically impose *constraints* on these concepts, as well as on the relation abstraction concepts, that govern the allowable definitions that may be supplied in a subcontract, lest the pattern invariant (and the correctness of the refinement) be violated.

Now consider the subcontract corresponding to S. It specifies how the abstractions of P are refined to satisfy the requirements of S. Structural refinement is achieved through the *role maps* defined by the subcontract. For each class C of S that plays a role R of P, the subcontract specifies a role map that maps the concrete state and methods of C to the abstract model and methods of R. These methods must satisfy, under the defined mappings, the corresponding method specifications defined by R's role contract. The subcontract also provides definitions for each relation abstraction concept and interaction abstraction concept. The definitions must satisfy the constraints specified in P's contract.

Now suppose that P_S is a specialized *sub-pattern* of P and R is a role of P. The simplest case is when the role contract for R is inherited by P_S from P. More interesting is the case when the data model for R is inherited, but the specification of one or more methods is *strengthened* by weakening the pre-condition and/or strengthening the post-condition. The role model may also be different, in which case the role map would be similar to that for a class playing role R. Two roles of P_S might also play role R. The role map for each would define mappings from the respective role models to R's model and might provide strengthened specifications for some of the methods, inheriting the rest from R.

Another possibility is that P_S may include a new role that does not map to any role of P. The corresponding role contract is not constrained by P's contract. The interaction traces for instances of P_S will record method calls on objects playing such additional roles. When checking whether the assertions of P_S imply the corresponding assertions of P —in particular, when dealing with the assertions over the traces— we effectively project out these elements. The P_S subcontract may also introduce new abstraction concepts and include constraints on these concepts; the constraints may involve the concepts inherited from P. (Of course, all constraints in the contract of P are inherited.) The precise syntax for the various elements defined within pattern contracts and subcontracts is part of our ongoing work. We omit these details due to space limitations, but illustrate some of the most important ideas in the next section.

5 Observer Hierarchy

Consider *General Observer*, the generalized Observer pattern intended to serve as the specialization base for (*i*) *Standard Observer*, (*ii*) *Chained Observer* (with the observers arranged in a chain), (*iii*) *Clustered Observer* (with the observers arranged in multiple clusters), and other sub-patterns. The contract for *General Observer* cannot require that notify() directly invoke update() on each attached observer since some of these specializations would not meet this requirement. But it would not be sufficient to simply require that when notify() finishes, each attached observer's state be *Consistent()* with the subject's state. This could be satisfied by, for example, simply resetting the subject's state back to its pre-conditional value (*i.e.*, before the change that triggered notify()) rather than updating the observers' states.

Another point related to the flexibility of the Observer pattern is worth noting. Many standard treatments of the pattern require that the other methods of the Observer role not make any changes to the observer's state, lest it become *inconsistent* with the subject's state. This is too restrictive since, for example, it doesn't allow an observer to change the format used to display information about the subject. In [6], we relaxed this to allow other methods of Observer to make changes as long as those changes left the observer in a state *consistent* with the same subject state that held at the start of the modifying Observer method. While this improves flexibility, it is still not flexible enough. For example, in the *MVC* architecture [3], both View and Controller play the Observer role. While View's other methods meet this requirement, Controller's methods do not. Indeed, in some *MVC*-based systems, the only way for a user to modify the subject's (*i.e.*, model's) state is via these methods. So, to maintain consistency, the state of the controller (and other observers) would have to be updated, not left unchanged.

To allow for the types of variation described above in regard to how update() is invoked on the various attached observers, we introduce the *AllObsUpdated()* interaction abstraction concept. The concept is defined over the subject state and the interaction trace and represents the notion of whether all attached observers have been updated, as necessary, to make them *consistent* with the current state of the subject. This intuition is captured in the following constraint, declared as part of the General Observer contract:

$$
\begin{array}{l}
\mathit{AllObsUpdated}(s, \tau) \Rightarrow \\
[\neg \mathit{Modified}(s, @\mathit{CurrState}(s, \tau)) \wedge \\
\forall ob \in s.\_obs : \ \mathit{Consistent}(s, @\mathit{CurrState}(ob, \tau))]
\end{array}
$$

@CurrState() is an auxiliary function that returns the state of the specified object in the most recent record in τ involving the object. Hence, the first clause of the consequent requires that the current subject state be *unmodified* from s, which represents the state of the subject at the start of the notify() call. That is, the clause requires that the subject's state not be modified while the observers are updated. The second clause of the consequent requires that for each observer in the *obs* set, the most current state recorded in τ be *consistent* with the subject's state. This allows the various updating strategies used in the sub-patterns, while

ensuring that all the observers are updated. Thus, we use *AllObsUpdated()* in the specification of notify() in the contract of General Observer. This ensures, given the above constraint, the intended behavior of the method.

The subcontract for each specialization of General Observer will provide an appropriate definition for *AllObsUpdated()*. For example, the subcontract for Standard Observer will define *AllObsUpdated()* to be *true* if τ contains a sequence of calls to update() on each element of *obs*, and *false* otherwise. The definitions corresponding to the subcontracts for Chained Observer and Clustered Observer will be more complex since they must account for the richer structure of the associated interaction sequences.

Interestingly, the variation in the behavior of the other methods of Observer, as in the Controller of *MVC*, can be represented without additional abstraction concepts. The others specification in the base pattern contract will require that changes in the observer state during execution of these methods must be due to intervening calls to update(), which themselves are due to changes in the subject state that result in calls to notify(). This requirement will be imposed by specifying suitable constraints on elements of τ as part of the others clause.

6 Conclusion

We have described a new form of *refinement* that plays a fundamental role in the design and implementation of object-oriented systems. *Design refinement* complements traditional refinement concepts and corresponds to the process of transforming a set of *design patterns* into design components, and finally, into implementations. Further, design refinement allows us to refine existing patterns to arrive at specialized *sub-patterns* and *pattern hierarchies*. This not only aids in pattern selection, but enables reuse of the effort involved in reasoning about a given pattern when reasoning about its variants.

We presented three contributions. First, we developed the idea of design refinement, including the three types of abstraction at its core. Second, we described an approach to specifying pattern requirements and behavioral guarantees in the form of *pattern contracts*, and to specifying *pattern subcontracts* that correspond to particular refinements of a pattern. A key consideration was to ensure that the flexibility of the pattern being specified was not compromised; the three types of abstraction concepts supported by the formalism ensure this. Indeed, a natural result of developing pattern contracts is that the contracts suggest partial refinements that correspond to specialized patterns and hierarchies. Thus, as the third contribution of the paper, we explored a hierarchy of *Observer* patterns.

Although the Observer pattern has been discussed widely in the literature, and various authors have suggested variations, our work seems to be the first to investigate them systematically. We were able to do so because the notion of design refinement, as well as the contracts for the various Observer variants, provided a natural foundation on which to base the hierarchy. In our future work, we intend to investigate other pattern hierarchies. This should be of great help to developers since each pattern in the hierarchy will be clearly specified.

References

1. de Roever, W., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and their Comparison. Cambridge (2001)
2. Morgan, C.: The specification statement. ACM Transactions on Programming Languages and Systems **10** (1988) 403–419
3. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons (1996)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
5. Sunye, G., Guennec, A., Jezequel, J.: Design patterns application in UML. In: The 14^{th} European Conference on Object-Oriented Programming. (2000) 44–62
6. Soundarajan, N., Hallstrom, J.: Responsibilities and rewards: Specifying design patterns. In: The 26^{th} International Conference on Software Engineering, IEEE Computer Society (2004) 666–675
7. Hallstrom, J., Soundarajan, N., Tyler, B.: Amplifying the benefits of design patterns. In: The 9^{th} International Conference on Fundamental Approaches to Software Engineering, Springer (2006) 214–229
8. Eden, A.: Formal specification of object-oriented design. In: The International Conference on Multidisciplinary Design in Engineering. (2001)
9. Eden, A.: LePUS: a visual formalism for object-oriented architectures. In: The 6^{th} World Conference on Integrated Design and Process Technology, IEEE Computer Society (2002) 149–159
10. Kim, S., Carrington, D.: Using integrated metamodeling to define OO design patterns with Object-Z and UML. In: The 11^{th} Asia-Pacific Software Engineering Conference, IEEE Computer Society (2004) 257–264
11. Dong, J.: UML extensions for design pattern compositions. Journal of Object Technology **1** (2002) 151–163
12. Lano, K.: Formalising design patterns as model transformations. In: Design Pattern Formalization Techniques. IGI Publishers (2007) 156–182
13. Mikkonen, T.: Formalizing design patterns. In: The 20^{th} International Conference on Software Engineering, IEEE Computer Society Press (1998) 115–124
14. Helin, J., Kellomki, P., Mikkonen, T.: Patterns of collective behavior in Ocsid. In: Design Pattern Formalization Techniques. IGI Publishers (2007) 73–93
15. Taibi, T., Ngo, D.: Formal specification of design patterns – a balanced approach. Journal of Object Technology **2** (2003) 127–140
16. Helm, R., Holland, I., Gangopadhyay, D.: Contracts: Specifying behavioral compositions in object-oriented systems. In: The European Conference on Object-Oriented Programming, ACM (1990) 169–180
17. Batory, D., Singhal, V., Thomas, J., Geraci, B., Sirkin, M.: GenVoca model of software-system generators. IEEE Software **11** (1994) 89–94
18. Biggerstaff, T.: A perspective of generative reuse. Annals of Softw. Eng. **5** (1998) 169–226
19. Neighbors, J.: Draco: a method for engineering reusable softw. sys. In: Software reusability: vol. 1, concepts and models. ACM (1989) 295–319