A Container-Based Approach to Object-Oriented Product Lines

Jason O. Hallstrom, Nigamanth Sridhar, Paolo A.G. Sivilotti,
Anish Arora, and William M. Leal
Computer and Information Science
The Ohio State University
Columbus OH, USA
{hallstro,nsridhar,paolo,anish,leal}@cis.ohio-state.edu

June 23, 2003

Abstract

Software product lines improve the productivity of developers by structuring application development around a set of features common to a family of applications. While a number of product line development approaches have been proposed, we argue that these approaches primarily target product lines that vary with respect to their functional properties. We propose a complementary approach to developing object-oriented product lines that vary with respect to their non-functional characteristics. Our approach is based on the use of software containers*, similar to those used to host Enterprise Java Beans. We illustrate the approach in the context of a distributed middleware product line for Microsoft's .NET Framework. The individual products in this family vary with respect to their dependability properties.*

1 Introduction

In manufacturing enterprises like the automotive industry, companies focus their production efforts around families of similar products. This kind of *product line* manufacturing allows core production assets to be used across products in the same family. Volkswagen, for example, produces three different automobile models that share the same chassis component. The Jetta, Golf, and GTI are each manufactured by specializing a generic chassis to suit the needs of the individual models. In addition to the economies of scale created by this process, additional productivity improvements come by way of worker specialization. Production workers become familiar with the common components, the tools used to compose them, and the process by which individual products are manufactured.

Following the manufacturing metaphor, *software product lines* are developed around a core set of software assets, with individual products in the line developed by specializing those assets. Not surprisingly, product line development techniques fundamentally rely on commonality and variability analysis [Coplien et al. 1998]. *Commonality analysis* identifies the core set of features that are constant across a product family. Conversely, *variability analysis* identifies the features that vary among the individual products. Development strategies leverage this analysis by focusing product

development around the core set of software artifacts that implement behavior common to all products in the family. By providing hooks for implementing the possible variations, individual products are developed with only the incremental effort required to implement the application-specific variability. The result is improved time and cost-to-market through code reuse and developer specialization.

Software product lines are experiencing increasing popularity as software vendors are faced with mounting pressure to develop software more quickly and more economically. As a consequence, a number of product line development approaches have been proposed. The majority of these approaches, as we will see later, target variation along the functional dimension. We propose a complementary approach for building product lines that vary along the non-functional dimension. We have a particular interest in product lines that vary with respect to their dependability characteristics, and therefore focus primarily on these specialized product families.

Our development approach is based on the use of open object containers, similar to those used to host Enterprise Java Beans [SunMicrosystems 2001]. In this context, a *container* is an extensible environment that provides runtime support to the objects it hosts. To be clear, the traditional object-oriented programming literature [Meyer 1988, Musser and Saini 1996] uses the term *container* to refer to classes that serve as collections of objects (*e.g.* lists, stacks, queues). We refer to these classes as *collection classes*, reserving the term *container* to refer to an extensible object hosting environment. Similar to an operating system hosting processes, a container hosts objects, providing services to the objects it hosts *transparently*. That is, hosted objects are imbued with additional services just by virtue of executing within the container, with little or no additional programming effort on behalf of the class designer. Containers provide a model of object-oriented software development that supports a clean separation of concerns between core object functionality, and system-level peripheral services. This development model has been used to great advantage in factoring out non-functional commonalities within the same product. In this paper we explore the use of open object containers in factoring out non-functional commonalities along a line of similar products. As we will see, our approach has a number of advantages.

The remainder of this paper is organized as follows. We present an overview of existing product line development approaches in Section 2. Section 3 presents an overview of container architectures, and their use in developing object-oriented product lines. We then illustrate the approach through an example presented in Section 4. We conclude in Section 5, summarizing the advantages of the approach, and providing pointers to future work.

2 Software Product Lines

The product line development approach has a long history in the software engineering community. The roots of the approach are typically traced back to David Parnas [Parnas 1979, Parnas 1976], although many of the ideas on which his work is based date back even further to the concept of *programming families* proposed by Edsgar Dijkstra [Dijkstra 1970, Dijkstra 1972]. More recently, software product lines have gained considerable importance as an effective way of achieving large-scale software reuse. A number of development approaches have been proposed; in this section we briefly survey some of the most important.

- **Class Libraries**. One of the most basic approaches for developing object-oriented product lines is to provide a class library that implements the functionality common across a family of products. All of the products in the family implement their common behavior using library classes, specializing those classes as appropriate to the individual applications. These libraries typically provide classes that model entities and services specific to a particular domain, thereby supporting product lines within that domain. The basic approach is familiar to anyone who has used more generic class libraries like libg++ [Lea 1988], the C++ Standard Template Library [Musser and Saini 1996], COOL [Fontana et al. 1990], and Bertrand Meyer's Base Object-Oriented Component Libraries [Meyer 1994].

- **Component Repositories**. Similar to a class library, a component repository contains reusable artifacts that provide functionality common to one or more product families. Individual applications are built as *assemblies* of components from these repositories, as well as application-specific components. Variability is achieved by selecting different combinations of components, as well as varying how those components are assembled. Product line development strategies which leverage component-based software engineering principles are discussed in more detail in [Griss 2000, Griss 2001].

- **Object-Oriented Frameworks**. The framework-based approach to developing object-oriented product lines focuses not just on reusing the services supplied by individual classes, but also on reusing the collaborative structure embedded across a family of products. An *object-oriented framework* [Fayad and Schmidt 1997] provides a set of classes that collaborate in a precise manner to provide a common architectural framework on which a family of similar products can be built. The common collaborative structure is captured in the form of key methods, referred to as *template methods* in the design patterns literature [Gamma et al. 1995], that direct the flow-of-control, and call the appropriate *hook methods* of various classes. Hook method implementations are deferred to derived-class designers, who provide implementations appropriate to the individual applications. Individual products in a line are developed with only the incremental effort required to implement the application-specific variability. Framework-based product line strategies are discussed in more detail in [Batory et al. 2000, Schmidt 1997].

- **Component Frameworks**. Similar to object-oriented frameworks, component frameworks can be used to factor out architectural commonalities along a product line. They are distinct, however, in that component frameworks specify structural and behavioral constraints that must be met by components plugged into the framework (e.g. for specialization), as well as rules governing how the components must interact [Szyperski 1998]. Just as the notion of component repository is an analogue of class library, the notion of component framework is an analogue of object-oriented framework. A more detailed treatment of their use in developing software product lines can be found in [Speck and Pulvermüller 2000].

- **Step-Wise Refinement**. Another approach to building software product lines is based on Edsgar Dijkstra's concept of **step-wise refinement** [Dijkstra 1976]. The basic idea is to develop solutions to problems by starting with a high-level solution, and progressively refining that solution until the required level of detail is achieved. The work presented in [Batory et al.

1994, Batory, Lopez-Herrejon and Martin 2002, Batory et al. 2003] discusses a technique based on scaling step-wise refinement to *refinement layers* that cross-cut module boundaries. In that model, a refinement is a cross-cutting aspect [Kiczales et al. 1997] that refines the functionality of one or more classes. The common functionality of the product line is implemented in the base layer. Individual applications are developed by progressively refining this functionality by applying additional layers that specialize the core functionality as appropriate to the application.

- **Domain-Specific Languages**. A domain-specific language is a programming notation dedicated to solving problems in a certain area; its expressive power is tailored to solving problems in a particular domain. The domain commonalities are factored out implicitly, embedded in the semantics of the language. These tailored languages are especially useful in building specialized product lines because, while the language provides constructs specific to the domain, it also typically excludes constructs of a regular programming language that could detract from the problem domain at hand [Bentley 1986]. Individual applications can be generated as specifications in these languages (thus exploiting their simplicity), and then translated into production programming languages (thus remaining portable) [Batory, Johnson, MacDonald and von Heeder 2002].

3 Object-Oriented Containers

A software container is a runtime environment designed to manage the execution of objects. Containers provide a set of common services to the objects they host, without the objects having been explicitly programmed to support those services. An EJB Container [SunMicrosystems 2001], for example, provides persistence, queuing, reference, and other enterprise services to the instances it hosts. Objects hosted by the container are imbued with these services with little additional programming effort just by virtue of executing within the container. That is, container services are relatively transparent to the hosted objects, as well as their clients.

Software containers achieve their relative transparency by relying on an old folk theorem in computer science: any problem we are likely to encounter can be solved by introducing an extra level of indirection. This is strikingly visible in the case of container architectures, which leverage indirection to transparently mediate object collaborations. Container-hosted objects are not accessed by client objects directly, but rather through container-generated proxies [Gamma et al. 1995]. Object method invocations on proxy objects are *intercepted* by the container, which provides additional services before and after routing the invocations to the appropriate invocation targets. A high-level view of this model is illustrated in Figure 1.

Software containers provide a model of object-oriented development that supports a separation of concerns between core object functionality and container-supplied peripheral services. Application developers focus their attention on application-level services, leaving the peripheral services to the container vendor. In practice, this set of peripheral services is fixed, and is generally limited in scope to a handful of enterprise services that are broadly applicable across most domains. That is, container services are typically used to factor out a small (but complex) set of domain-independent commonalities. Our approach to product line development extends this perspective, extending the range of container services under consideration, and using those services to factor out commonal-

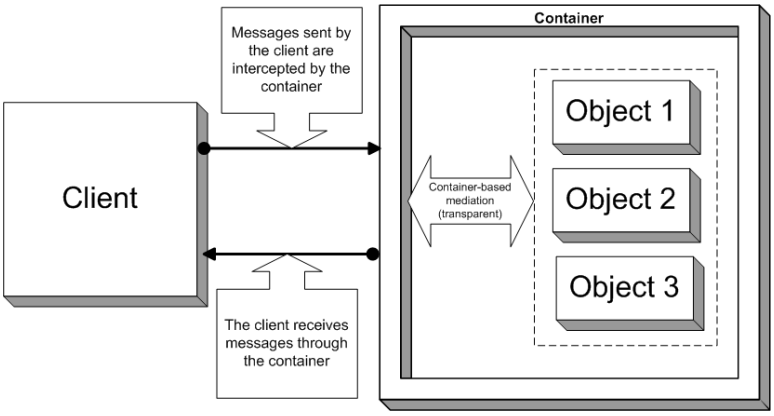

Figure 1: The container acts as a transparent layer around one or more objects. Client access to hosted objects is mediated by the container, which transparently injects services in the path of object collaborations.

ities, as well as manage non-functional variation among products in a family. At the heart of this approach is the ability to modularize container services as reusable modules.

As we have discussed in [Hallstrom et al. 2003], and will see in the following section, *interceptors* provide a means of modularizing a variety of novel container services across different domains. These interceptors are invoked automatically by the container in response to invocations on the objects it hosts. Interceptors operate on the logical invocation request and response messages that flow between objects in an object-oriented system. Whenever a method is invoked on a hosted object (through a proxy), the container generates an object that stores information about the call, and passes the resulting message object to the appropriate interceptors before performing the invocation on the target. Similarly, when the invocation completes, the container generates a new object that stores information about the call completion, and passes the resulting message object to the appropriate interceptors before returning control to the caller. An interceptor-based architecture [Schmidt et al. 2000] allows interceptors to modularize the behaviors injected in the path of target method invocations. Figure 2 illustrates a container linked with two such interceptor modules.

Since the interceptors we have described operate on call and response information, they can be viewed as meta-level objects that transform the behavior of the objects to which they are applied [Hallstrom et al. 2003]. Like other meta-level approaches, interceptors can modularize concerns that are typically thought of as cross-cutting. For example, a single interceptor instance can be used to provide invocation logging services to a set of objects of varying type[1]. This meta-level property is important to our approach because it allows us to modularize non-functional concerns, which almost always cross-cut module boundaries since systems are typically decomposed based on functional considerations. Consequently, we are able to develop interceptor libraries that provide a range of non-functional services.

Given a library of interceptor modules targeting non-functional concerns, we should like to vary the interceptors used by a container to achieve non-functional variation along products in a line.

[1]The resulting invocation log might be used to support debugging services, rollback and recovery, etc.

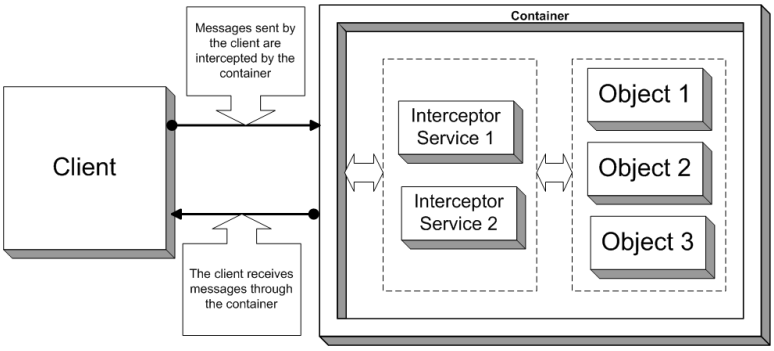

Figure 2: The container mediates access to its hosted objects through interceptor modules. Each interceptor module modularizes a particular service that should be transparently injected in the path of object collaborations.

Following the open-closed principle of modular decomposition [Meyer 1988], the set of services supplied by a container should be configurable without modifying the underlying container code. This is in conflict with the fixed-service view of containers that pervades present day software practice. As we have discussed in [Sridhar and Hallstrom 2003], one way of achieving this non-invasive configurability is to recast container architectures as parameterized templates. That is, each container is parameterized by a variable number of interceptors that enrich the behavior of the hosted objects and their interactions. Supporting a variable number of interceptors is possible because interceptors share a common interface. From the container designer's perspective, the container accepts one template parameter: a variable length array of interceptor objects that provide identical interfaces.

While much work has gone into using containers to factor out commonalities, there appears to be little work in using containers to manage variability. Our approach to product line development relies on a parameterized view of software containers, and a supporting interceptor library that provides a range of non-functional services. By varying the set of interceptors used by a container, we are able to factor out commonalities, as well as tailor the non-functional properties as appropriate to the individual applications.

4 Example: A Distributed Middleware Product Line

In [Hallstrom et al. 2003] we describe the design and implementation of an open object container that we have developed for Microsoft's .NET Framework. Our container architecture supports the parameterized view of containers outlined in the previous section. We have used this architecture to develop middleware that provides transparent distribution and lookup services, similar to those provided by Java RMI and CORBA. As we have a particular interest in dependability properties, we have additionally developed a library of interceptor modules that provide a range of dependability services to applications developed using this middleware. Using the modules provided by this library, we have deployed our middleware under different configurations depending on the level of dependability required by the individual applications. In this section we briefly describe a small set

of the interceptor modules that we have developed, and provide a silhouette of how those modules have been used in developing a dependable middleware product line.

4.1 Dependability Services

Replication and Fail-Over
In distributed scenarios it is often useful to maintain replicated objects so that object failures can be masked from the rest of the system. When an object fails, client requests can be transparently redirected to one of the replicas, preventing the fault from propagating to client objects. This service is implemented as three interceptor modules.

- **Multicast Interceptor**. The multicast interceptor is associated with a primary target, and is responsible for keeping a set of replicas synchronized with that target. Whenever an invocation is received by the primary object, the multicast interceptor forwards copies of that message to the replicas. These replicas serve as *hot-spares*, ready to be put into service should the primary fail.

- **Filter Interceptor**. The filter interceptor acts as a one-way filter that only allows messages from the primary object to pass through to clients. Messages sent from non-primary replicas are discarded, allowing clients to remain unaware of the replication.

- **Fail-Over Interceptor**. The fail-over interceptor detects when an object is unable to respond to invocation requests, and then selects a new primary object on behalf of the filter interceptor. We have implemented two versions of this module. The first implementation detects object failures by capturing exceptions from the underlying network. The second implementation relies on a simple heart-beat scheme that signals a failure if the primary does not send a heart-beat within a specified amount of time.

Distributed Recording
In providing masking fault-tolerance, an alternative to replicating objects is to provide support for checkpointing and recovery. The distributed recording service logs logically sequential events in a distributed system using vector clocks. The resulting log can be used to play back the events to bring the system into a consistent state should a fault occur[2]. This service is implemented as two interceptors that share a set of vector clocks and a common log. Each vector clock corresponds to the logical time associated with a single object.

- **Send Interceptor**. When the send interceptor receives a message for sending, it updates the logical clock of the sending object, and logs the message along with the time at which it was sent. If the target of the message is container-hosted, the interceptor adds the updated value of the clock as a time-stamp to the outgoing message. The message is then passed to the container for sending.

- **Receive Interceptor**. When the receive interceptor receives a message for delivery, it checks whether a time-stamp was included with the message. If a time-stamp was included, the

[2]The playback service is still under development.

interceptor removes the time-stamp, updates the logical clock of the receiving object, and logs the message along with the time at which it was received. The message is then passed to the container for delivery.

Load Balancing
When a system is subject to high demand, it is beneficial to distribute the processing load across all available hardware resources. Our load balancing service provides object-based load balancing across a set of replicated objects distributed across multiple processors. As the service is simple, it is implemented as a single interceptor.

- **Switch Interceptor**. The switch interceptor provides a load-balancing service to a set of replicated objects. The interceptor monitors the utilization of the containers in which the replicated objects are hosted. When an invocation request is received, the interceptor redirects the message to the object whose container has the most available resources.

4.2 Product Configurations

We have deployed our middleware using several different interceptor configurations. Each configuration corresponds to a unique product in a family of middleware applications. The selection of interceptor modules for each product is driven by the dependability needs of the applications that will use the resulting middleware. Distributed applications requiring support for masking fault-tolerance, for example, would be deployed with the interceptors required to support replication and fail-over, or checkpointing and recovery. Space and time tradeoffs of replication versus checkpointing, real-time constraints, and other considerations will drive the selection of one technique over the other. Middleware targeting applications with stringent real-time constraints, for instance, would be configured with the interceptors required to support replication and fail-over, as the time required to rollback from a fault would make checkpointing and recovery infeasible. The middleware product line might also be deployed with our load balancing service to keep the application load from interfering with meeting real-time requirements.

Every product in our product family is developed by non-invasively specializing our core middleware implementation. The configuration process is equivalent to instantiating the middleware implementation with one or more interceptor-based services. This process is light-weight, and can be done with little or no programming. We are, for example, investigating the use of XML-based configuration files for automating the configuration and deployment of individual products.

5 Discussion

As the example in the preceding section illustrates, object-oriented containers are technically viable for developing product lines that vary with respect to their non-functional properties. However, the motivations for (and hence the suitability of an approach to) building software product lines goes beyond technical feasibility. Software product lines introduce interesting opportunities for advancing the state of the software business. For a product line approach to be adopted, it is essential that these business (i.e. non-technical) issues be considered in some detail as well. In this section, we present a brief outline of how object-oriented containers offer solutions to some of these

non-technical issues. Much of the discussion covers our ongoing and future research directions in using containers to build commercial product lines.

5.1 Variability Analysis

Variability in a software product line is made explicit by introducing a number of *variation points* [Bosch et al. 2001]. Each of these variation points represents a particular design decision that is explicitly delayed until later in the development cycle. At the time of designing a product line, the engineer designs in the commonalities as the base features of the line, and leaves the variabilities open. In other words, the variation points are not fixed. Once the variation points are fixed, the designer has constrained the different ways in which the product line architecture can be specialized to yield a specific product. This phase of the product line's development is called *domain engineering*. The process of building particular products from such a product line architecture, known as *application engineering*, involves binding specific variants to each variation point.

In an object-based approach to product line architectures, such as the one we have proposed in this article, the different design decisions can be encapsulated in their own modules [Parnas 1972]. Once this is done, the application engineering process is reduced to simply picking the particular modules that include the appropriate variants for each variation point. This approach, in itself, introduces another variation point — the binding time of the variants to their variation points. In our object-based approach, depending on the implementation technology that is chosen, the binding time could range from compile-time through run-time. In fact, with interceptors as the implementation mechanism (as described in Sections 3 and 4), variants can be bound at run-time to their variation points. Further, our interceptor architecture allows for variants to be unbound and rebound during the lifetime of the software system. Such possibilities for rebinding open up new avenues for further investigation in product line variabilities.

Another important variability is the evolution of object interfaces [Svahnberg and Bosch 2000]. New implementations of existing abstract interfaces may add new functionality that was originally unknown to the rest of the system. Such interface modifications are handled in our interceptor-based architecture, since the level of granularity provided by interceptors can be as fine-grained as needed. An interceptor could, for example, intercept requests for a new method that did not exist in the original object, and delegate those requests to another object capable of providing the appropriate service. Further, since the interceptor architecture supports dynamic reconfiguration at various levels of scope [Hallstrom et al. 2003], variabilities can be introduced at the product line level, product level, object level, or even at the method level; and all these variabilities can be introduced and modified at run-time.

5.2 Product Line Economics

Economics is a very important motivation behind product-line engineering [Schmid 2001]. Indeed, it may well be the primary motivation for developing product families. The cost improvements of building a product family as opposed to several individual (yet related) products is considerable. Thus any new approach to building product lines must address this issue.

In our model, individual products in the product family are created by configuring the hosting

container with the appropriate set of container services. Even after a product has been deployed, the set of services it uses can be modified dynamically. Dynamic binding and rebinding of services to products brings up the interesting possibility of a *pay-on-use* economic model. The vendor of the services can use a cost model based on which particular services a product uses, and even when those services are bound.

Moreover, our model allows the product designer to view the container services from a *real options* perspective. The binding of a particular service to a product is viewed as an investment that the product designer is making. [Baldwin and Clark 2000] and [Sullivan et al. 2001] present software development as an investment activity. The models they propose (when adapted to our container-based approach) provide a strong economic basis for using software containers to build product-lines architectures.

5.3 Reasoning Issues

When an object is hosted by a container, the client's view of the object is transformed. The client views the hosted object as a variant of the original, augmented by the services supplied by its hosting container. So, when reasoning about compositional behavior, it is important to consider each object in conjunction with its host. However, since the abstract interface of the hosted object is modified as a result of placing it in the container, standard approaches to compositional reasoning can no longer be applied.

In an effort to overcome this problem, we have shown previously that containers can be recast as parameterized components, and when viewed this way, the distinction between containers and components disappears [Sridhar and Hallstrom 2003]. Several proof techniques are available for reasoning about parameterized components [Sitaraman and Weide 1994]. Further, tools exist to automatically generate reasoning tables for parameterized component models that use templates as the parameter binding mechanism [Sitaraman et al. 2003]. All of these techniques could potentially be used to reason about container-based product lines. At this point, however, we cannot use these methods to reason about systems developed using our interceptor architecture, since interceptors are meta-level components. We are currently working to develop a strong connection between such meta-level mediators and component-based reasoning.

References

Baldwin, C. and Clark, K.: 2000, *Design Rules: The Power of Modularity*, Vol. 1, MIT Press, Cambridge, MA.

Batory, D., Cardone, R. and Smaragdakis, Y.: 2000, Object-oriented frameworks and product lines, *in* P. Donohoe (ed.), *Proceedings of the First Software Product Line Conference*, pp. 227–247.

Batory, D., Johnson, C., MacDonald, B. and von Heeder, D.: 2002, Achieving extensibility through product-lines and domain-specific languages: A case study, *ACM Transactions on Software Engineering and Methodology* **11**(2), 191–214.

Batory, D., Lopez-Herrejon, R. E. and Martin, J.-P.: 2002, Generating product-lines of product families, *Proceedings of the 2002 Automated Software Engineering Conference*, Edinburgh, Scotland, pp. 81–92.

Batory, D., Sarvela, J. N. and Raushmayer, A.: 2003, Scaling step-wise refinement, *Proceedings of the* 25^{th} *International Conference on Software Engineering*, ACM, Portland, OR.

Batory, D., Singhal, V., Thomas, J., Dasari, S., Geraci, B. and Sirkin, M.: 1994, The GenVoca model of software-system generators, *IEEE Software* **11**(5), 89–94.

Bentley, J.: 1986, Programming pearls: Little languages, *Communications of the ACM* **29**(8), 711–721.

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J. H. and Pohl, K.: 2001, Variability issues in software product lines, *Software Product-Family Engineering: Proceedings of PFE 2001*, number 2290 in *LNCS*, Springer, Bilbao, Spain, pp. 13–21.

Coplien, J. O., Hoffman, D. and Weiss, D. M.: 1998, Commonality and variability in software engineering, *IEEE Software* **15**(6), 37–45.

Dijkstra, E. W.: 1970, Structured programming, *in* J. Buxton and B. Randell (eds), *Software Engineering Techniques*, NATO Scientific Affairs Division, pp. 84–87.

Dijkstra, E. W.: 1972, Notes on structured programming, *in* O. Dahl, E. Dijkstra and C. Hoare (eds), *Structured Programming*, number 8 in *A.P.I.C. Studies in Data Processing*, Academic Press, chapter 1, pp. 1–82.

Dijkstra, E. W.: 1976, *A Discipline of Programming*, Prentice Hall.

Fayad, M. and Schmidt, D.: 1997, Object-oriented application frameworks, *Communications of the ACM, Special Issue on Object-Oriented Application Frameworks* **40**(10).

Fontana, M., Oren, L. and Neath, M.: 1990, *COOL — C++ Object-Oriented Library*, Texas Instruments.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Griss, M. L.: 2000, Implementing product-line features with component reuse, *Proceedings of the* 6^{th} *International Conference on Software Reuse (ICSR-6)*, Springer, Vienna, Austria, pp. 137–152.

Griss, M. L.: 2001, Product-line architectures, *in* G. T. Heineman and W. T. Councill (eds), *Component-Based Software Engineering*, Addison-Wesley, chapter 22, pp. 405–420.

Hallstrom, J. O., Leal, W. M. and Arora, A.: 2003, Scalable evolution of highly-available systems, *IEICE/IEEE Joint Special Issue on Assurance Systems and Networks* . (to appear).

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J.: 1997, Aspect-oriented programming, *in* M. Akşit and S. Matsuoka (eds), *Proceedings European Conference on Object-Oriented Programming*, Vol. 1241, Springer-Verlag, Berlin, Heidelberg, and New York, pp. 220–242.

Lea, D.: 1988, The GNU C++ library, *Proceedings of the USENIX C++ Conference*.

Meyer, B.: 1988, *Object-Oriented Software Construction*, Prentice-Hall.

Meyer, B.: 1994, *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice-Hall.

Musser, D. and Saini, A.: 1996, *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, Addison-Wesley.

Parnas, D. L.: 1972, On the criteria to be used in decomposing systems into modules, *Communications of the ACM* **15**(12), 1053–1058.

Parnas, D. L.: 1976, On the design and development of program families, *IEEE Transactions on Software Engineering* **2**(1), 1–9.

Parnas, D. L.: 1979, Designing software for ease of extension and contraction, *IEEE Transactions on Software Engineering* **SE-5**(2), 128–138.

Schmid, K.: 2001, An initial model of product line economics, *Software Product-Family Engineering: Proceedings of PFE 2001*, number 2290 in *LNCS*, Springer, Bilbao, Spain, pp. 38–50.

Schmidt, D. C.: 1997, Applying design patterns and frameworks to develop object-oriented communication software, *in* P. Salus (ed.), *Handbook of Programming Languages*, Vol. I, MacMillan Computer Publishing.

Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F.: 2000, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Vol. 2, John Wiley & Sons Ltd", West Sussex, England.

Sitaraman, M., Gandi, D. P., Küchlin, W., Sinz, C. and Weide, B. W.: 2003, The humane bugfinder: Modular static analysis using a SAT solver, *Technical Report RSRG-03-05*, Clemson University, Clemson SC.

Sitaraman, M. and Weide, B.: 1994, Component-based software using RESOLVE, *Software Engineering Notes* **19**(4), 21–22.

Speck, A. and Pulvermüller, E.: 2000, Component frameworks for software generators, *Workshops of the GI Specialized Group on Programming Languages and Computing Concepts*, pp. 45–53.

Sridhar, N. and Hallstrom, J. O.: 2003, Generating configurable containers for component-based software, *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering*.

Sullivan, K. J., Griswold, W. G., Cai, Y. and Hallen, B.: 2001, The structure and value of modularity in software design, *Proceedings of the 9th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ACM Press, Vienna, Austria, pp. 99–108.

SunMicrosystems: 2001, J2ee 1.3 specification, http://java.sun.com/j2ee/download.html.

Svahnberg, M. and Bosch, J.: 2000, Issues concerning variability in software product lines, *Software Architectures for Product Families: Proceedings of IW-SAPF-3*, number 1951 in *LNCS*, Springer, Las Palmas de Gran Canaria, Spain, pp. 146–157.

Szyperski, C.: 1998, *Component Software: Beyond Object-Oriented Programming*, ACM Press and Addison-Wesley, New York, N.Y.