1 chapter

Introduction

A software product line consists of a family of software systems that have some common functionality and some variable functionality. To take advantage of the common functionality, reusable assets (such as requirements, designs, components, and so on) are developed, which can be reused by different members of the family. Clements and Northrop (2002) define a **software product line** as "a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" (p. 5).

The interest in software product lines emerged from the field of software reuse when developers realized that they could obtain much greater reuse benefits by reusing software architectures instead of reusing individual software components. The software industry is increasingly recognizing the strategic importance of software product lines (Clements and Northrop 2002).

The idea of a product line is not new. There are examples of product lines in ancient history; the pyramids of Egypt (Figure 1.1) might have been the first product line! A modern example of product lines comes from the airline industry, with the European Airbus A-318, A-319, A-320, and A-321 airplanes, which share common product features, including jet engines, navigation equipment, and communication equipment (Clements and Northrop 2002).

The traditional mode of software development is to develop single systems—that is, to develop each system individually. For software product lines, the development approach is broadened to consider a family of software systems. This approach involves analyzing what features (functional requirements) of the software family are common, what features are optional, and what features are alternatives. After the feature analysis, the goal is to design a software architecture

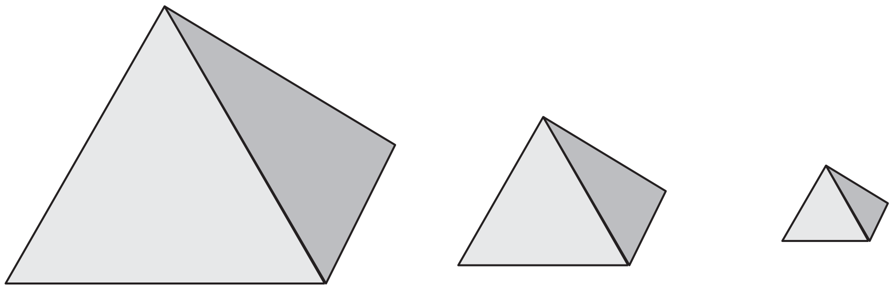

Figure 1.1 *A product line from ancient history: the pyramids of Egypt*

for the product line, which has **common components** (required by all members of the family), **optional components** (required by only some members of the family), and **variant components** (different versions of which are required by different members of the family). To model and design families of systems, the analysis and design concepts for single-product systems need to be extended to support software product lines.

This chapter presents an overview of software reuse and software product lines. It also gives an overview of using the Unified Modeling Language (UML) to develop component-based software product lines.

1.1 Software Reuse

Software reuse has been a goal in software engineering since 1968, when the term *software engineering* was first coined. This section briefly surveys different approaches to software reuse, starting with the most common form of software reuse (using reuse libraries) and leading up to software product lines.

1.1.1 Software Reuse Libraries

In traditional software reuse, a library of reusable code components is developed. This approach requires the establishment of a library of reusable components and of an approach for indexing, locating, and distinguishing between similar components (Prieto-Diaz and Freeman 1987). Problems with this approach include managing the large number of components that such a reuse library is likely to contain and distinguishing among similar though not identical components.

The reusable library components are the building blocks used in constructing the new system. Components are considered to be largely atomic and ideally unchanged when reused, although some adaptation may be required. Depending on the development approach, the library could contain functional or object-oriented components. This reuse approach has been used with both functional and object-oriented systems.

When a new design is being developed, the designer is responsible for designing the software architecture—that is, the overall structure of the program and the overall flow of control. Having located and selected a reusable component from the library, the designer must then determine how this component fits into the new architecture.

An example of this traditional reuse is a subroutine library, which consists of a collection of reusable subroutines in a given application area—for example, a statistical subroutine library. Another example is an object-oriented toolkit, which consists of a set of related and reusable classes designed to provide useful, general-purpose functionality. These reuse libraries and toolkits emphasize code reuse.

Apart from certain specific domains, such as mathematical libraries, the benefits of the traditional software reuse approach have been limited in general. With this approach, overall reuse is relatively low, and the emphasis is on code reuse.

1.1.2 Software Architecture and Design Reuse

Instead of reusing an individual component, it is much more advantageous to reuse a whole design or subsystem, consisting of the components and their interconnections. This means reuse of the control structure of the application. Architecture reuse has much greater potential than component reuse because it is large-grained reuse, which focuses on reuse of requirements and design.

1.2 Software Product Lines

The most promising approach for architecture reuse is to develop a product line architecture, which explicitly captures the commonality and variability in the family of systems that constitutes the product line. Various terms are used to refer to a software product line. A software product line is also referred to as a **software product family**, a **family of systems**, or an *application domain*. The terms used most often in the early days of this field were *domain analysis* (Prieto-Diaz 1987) and *domain modeling*. The architectures developed for application domains were referred to as *domain models* or *domain-specific software architectures* (Gomaa 1995).

Parnas referred to a collection of systems that share common characteristics as a *family of systems* (Parnas 1979). According to Parnas, it is worth considering the development of a family of systems when there is more to be gained by analyzing the systems collectively rather than separately—that is, when the systems have more features in common than features that distinguish them. A family of systems is now referred to as a *software product line* or a *software product family*. Some approaches attempt to differentiate between these two terms. This book, in common with other approaches, does not differentiate between the terms *software product line* and *software product family*, assuming that they both refer to a family of systems.

The architecture for a software product line is the architecture for a family of products. Some product line development approaches provide a **generic architecture**, or **reference model**, for the product line, which depicts the commonality of the product line but ignores all the variability. Each application starts with the generic architecture and adapts it manually as required. Although this approach provides a better starting point in comparison to developing a system without any reuse, it fails to capture any knowledge about the variability in the product family.

A more desirable approach is to explicitly model both what is common and what is different in the product family. A software product line architecture should therefore describe both the commonality and variability in the family. Depending on the development approach used (functional or object-oriented), the product line **commonality** is described in terms of common modules, classes, or components, and the product line **variability** is described in terms of optional or variant modules, classes, or components.

The term **application engineering** refers to the process of tailoring and configuring the product family architecture and components to create a specific application, which is a member of the product family.

1.2.1 Modeling Variability in Software Product Lines

Modeling commonality and variability is an important activity in developing software product lines. Early advocates of product line development included proponents of the FAST (Family-Oriented Abstraction, Specification, and Translation) approach and its predecessors (Coplien et al. 1998; Weiss and Lai 1999), work at the Software Engineering Institute (Clements and Northrop 2002; Cohen and Northrop 1998; Kang et al. 1990), work at the Software Productivity Consortium (Pyster 1990), and the EDLC (Evolutionary Domain Life Cycle) approach (Gomaa 1995; Gomaa and Farrukh 1999; Gomaa and O'Hara 1998; Gomaa, Kerschberg, et al. 1996). Jacobson et al. (1997) introduced the concept of variation points as a way to model variability in use cases and UML.

Variability in a software product line can be modeled at the software requirements level and at the software design level. The most widely used approach for modeling variability in requirements is through feature modeling, as described next.

1.3 Modeling Requirements Variability in Software Product Lines: Feature Modeling

Features are an important concept in software product lines because they represent reusable requirements or characteristics of a product line. The concept of a feature is quite intuitive and applies to all product lines, not just software product lines. Consider a vehicle product line. Several different models of cars might share common characteristics. For example, a vehicle product line (analogous to a software product line) might have a common chassis, which represents a *common* feature; a choice of engine size, where each engine represents an *alternative* feature; and optional cruise control, which represents an *optional* feature. For any individual car (analogous to a software member of the product line), a buyer would have no choice of chassis (there is no choice because this is a common feature), would have a choice of engine size (one of the alternative features), and could decide whether to have cruise control or not (the optional feature).

The FODA (feature-oriented domain analysis) method uses features, which are organized into a feature tree (Cohen and Northrop 1998; Kang et al. 1990). Features may be mandatory, optional, or mutually exclusive. The feature tree is a composition hierarchy of features, in which some branches are mandatory, some are optional, and others are mutually exclusive. In FODA, features may be functional features (hardware or software), nonfunctional features (e.g., relating to security or performance), or parameters (e.g., red, yellow, or green). Features higher up in the tree are composite features if they contain other lower-level features.

Other product line methods have used the FODA approach for modeling product line features (Dionisi et al. 1998; Griss et al. 1998). Feature modeling has also been prominent in other product line methods, such as the EDLC method (Gomaa 1995; Gomaa and Farrukh 1999; Gomaa, Kerschberg, et al. 1996). The feature concept is not an object-oriented concept, and it is not used in UML modeling of single systems. Chapter 5 describes an approach for modeling and describing features in UML.

1.4 Modeling Design Variability in Software Product Lines

Techniques for modeling variability in design include modeling variability using parameterization, modeling variability using information hiding, and modeling variability using inheritance (Gomaa and Webber 2004). The relative merits of each technique depend on the amount of flexibility needed in the product line. These approaches are described briefly in Sections 1.4.1 through 1.4.3.

1.4.1 Modeling Variability Using Parameterization

Parameters can be used to introduce variability into a software product line. The values of parameters defined in the product line components are where the variation comes in. Different members of the product line would have different values assigned to the parameters.

Parameterization approaches use four different types of parameters:

1. **Compilation parameters**. Values are set at compile time. This approach is sometimes used to conditionally compile code depending on the parameter setting.
2. **Configuration parameters**. Values are set at system configuration time, which is sometimes referred to as system generation (sysgen) time or system installation time.
3. **Runtime initialization parameters**. Values are set at system initialization time. The component could provide operations that are called at system initialization to initialize or change the values of parameterized attributes.
4. **Table-driven parameters**. Some reusable applications are configured with parameters that are stored in tables. This can be an effective way of allowing users to configure the application.

Several product line analysis methods use parameterization. The FODA method developed at the Software Engineering Institute (SEI) uses features to characterize the domain (Kang et al. 1990). In FODA, one kind of feature is the parameter. In the Domain-Specific Software Architecture (DSSA) program, generic components were parameterized to simplify customization for particular applications (Hayes-Roth 1995). The FAST product line method (Weiss and Lai 1999), which emphasizes analyzing commonality and variability in a product line, uses parameterization extensively as one of the approaches to achieve variability. The EDLC model for software product lines uses parameterization at configuration time to define the values of component parameters (Gomaa and Farrukh 1999).

A well-documented case study of a product line that used parameterization is CelsiusTech (Bass et al. 2003). CelsiusTech's Ship System 2000 (SS2000) is a product line for command and control of naval systems. The SS2000 consisted of parameterized components for which the application engineers supplied the parameter values. The SS2000 features 3,000 to 5,000 parameters that must be individually set for each member system of the product line. Managing such a large number of parameters could be a problem because little or no guidance is given on how to ensure that parameters are not in conflict with each other.

1.4.2 Modeling Variability Using Information Hiding

Information hiding can also be used to introduce variability into a software product line. Different versions of a component have the same interface but different implementations for different members of the product line. Thus the variability is hidden inside each version of the component. In this case the variants are the different versions of the same component but must adhere to the same interface. This approach works well as long as changes can be limited to individual components and no changes to the interface or assumptions about how the interface will be used are required. From a reuse perspective, the application engineer selects a component from a limited set of choices and uses it in the application.

Most product line methods use information hiding. In addition to parameterization, the FODA, DSSA, FAST, and EDLC methods already described all use information hiding.

1.4.3 Modeling Variability Using Inheritance

A third way of introducing variability into a software product line is to use inheritance. In this case, different versions of a class use inheritance to inherit operations from a superclass, and then operations specified in the superclass interface are redefined and/or the interface is extended by the addition of new operations. For a given member of the product line, the application engineer selects the version of the component.

With the widespread use of object-oriented techniques, variability using inheritance has become increasingly common in software product lines. The KobrA approach (Atkinson et al. 2002), which is an object-oriented customization of the Product Line Software Engineering (PuLSE) method (Bayer et al., 1999; DeBaud and Schmid 1999), uses inheritance to model component variability. In the EDLC product line method (Gomaa and Farrukh 1999), inheritance is used to model different variant subclasses of an abstract class, such that the variants are used by different members of the product line.

1.4.4 Comparison of Approaches for Modeling Design Variability

Modeling variability using parameterization allows application engineers to define the values of product line attributes, which are maintained by various product line components. If all the variability in a product line can be defined in terms of parameters, then this can be a simple way to provide variability. However, the variability is limited in that no functionality can change. In some product lines parameterization is used to select among functionality alternatives, although it is a complicated and error-prone approach and needs to be used with care.

Modeling variability using information hiding allows application engineers to choose variants from a limited set of choices in which the interface is common but the implementations are variable. Information hiding allows a higher degree of variability than parameterization does because both functionality and parameters can be varied.

Modeling variability using inheritance allows application engineers to choose variants whose functionality can vary. Because of the nature of inheritance, this approach also allows a wider range of variants by extending the superclass interface in variant subclasses.

In most product lines, a combination of all three approaches is needed. The object-oriented approach to software development helps by supporting all three of these approaches to modeling variability. However, other approaches are also needed, as explained next.

1.5 Reusable Design Patterns

A different approach for providing design reuse is through design patterns. A **design pattern** describes a recurring design problem to be solved, a solution to the problem, and the context in which that solution works (Buschmann et al.

1996; Gamma et al. 1995). The description specifies objects and classes that are customized to solve a general design problem in a particular context. A design pattern is a larger-grained form of reuse than a class because it involves more than one class and the interconnection among objects from different classes. A design pattern is sometimes referred to as a *microarchitecture*.

After the original success of the design pattern concept, other kinds of patterns were developed. The main kinds of reusable patterns are

- **Design patterns**. In a widely cited book (Gamma et al. 1995), design patterns were described by four software designers—Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—who were named in some quarters as the "gang of four." A design pattern is a small group of collaborating objects.
- **Architectural patterns**. This work was described by Buschmann et al. (1996) at Siemens. Architectural patterns are larger-grained than design patterns, addressing the structure of major subsystems of a system.
- **Analysis patterns**. Analysis patterns were described by Fowler (2002), who found similarities during analysis of different application domains. He described recurring patterns found in object-oriented analysis and described them with static models, expressed in class diagrams.
- **Product line–specific patterns**. These are patterns used in specific application areas, such as factory automation (Gomaa 1998) or electronic commerce.
- **Idioms**. Idioms are low-level patterns specific to a programming language—for example, Java or C++. These patterns are closest to code, but they can be used only by applications that are coded in the same programming language.

From the perspective of software product lines, the biggest benefit can usually be obtained through the reuse of software architectural patterns, which is described in more detail in Chapter 10.

1.6 Modeling Single Systems with UML

Object-oriented concepts are considered important in software reuse and evolution because they address fundamental issues of adaptation and evolution. Object-oriented methods are based on the concepts of information hiding, classes, and inheritance. Information hiding can lead to systems that are more self-contained and hence are more modifiable and maintainable. Inheritance provides an approach for adapting a class in a systematic way.

With the proliferation of notations and methods for the object-oriented analysis and design of software applications, the Unified Modeling Language (UML) was developed to provide a standardized notation for describing object-oriented models. For the UML notation to be applied effectively, however, it needs to be used together with an object-oriented analysis and design method.

Modern object-oriented analysis and design methods are model-based and use a combination of use case modeling, static modeling, state machine modeling, and object interaction modeling. Almost all modern object-oriented methods use the UML notation for describing software requirements, analysis, and design models (Booch et al. 2005; Fowler 2004; Rumbaugh et al. 2005).

In **use case modeling**, the functional requirements of the system are defined in terms of use cases and actors. **Static modeling** provides a structural view of the system. Classes are defined in terms of their attributes, as well as their relationships with other classes. **Dynamic modeling** provides a behavioral view of the system. The use cases are realized to show the interaction among participating objects. Object interaction diagrams are developed to show how objects communicate with each other to realize the use case. The state-dependent aspects of the system are defined with statecharts.

1.7 COMET: A UML-Based Software Design Method for Single Systems

An example of a UML-based software design method for single systems is COMET (***C***oncurrent ***O***bject ***M***odeling and Architectural Design M***et***hod), which is described in Gomaa 2000. COMET is a highly iterative object-oriented software development method that addresses the requirements, analysis, and design modeling phases of the object-oriented development life cycle. The functional requirements of the system are defined in terms of actors and use cases. Each use case defines a sequence of interactions between one or more actors and the system. A use case can be viewed at various levels of detail. In a *requirements* model, the functional requirements of the system are defined in terms of actors and use cases. In an *analysis* model, the use case is realized to describe the objects that participate in the use case, and their interactions. In the *design* model, the software architecture is developed, addressing issues of distribution, concurrency, and information hiding. Sections 1.7.1 through 1.7.3 discuss each of these phases of object-oriented development.

1.7.1 Requirements Modeling

During the **requirements modeling** phase, a requirements model is developed in which the functional requirements of the system are defined in terms of actors and use cases. A narrative description of each use case is developed. User inputs and active participation are essential to this effort. If the requirements are not well understood, a throwaway prototype can be developed to help clarify the requirements.

1.7.2 Analysis Modeling

In the **analysis modeling** phase, static and dynamic models of the system are developed. The *static model* defines the structural relationships among problem domain classes. The classes and their relationships are depicted on class diagrams. Object-structuring criteria are used to determine which objects should be considered for the analysis model. A *dynamic model* is then developed in which the use cases from the requirements model are realized to show the objects that participate in each use case and how they interact with each other. Objects and their interactions are depicted on either communication diagrams or sequence diagrams. In the dynamic model, state-dependent objects are defined with statecharts. A **statechart** is a graphical representation of a finite state machine in the form of a hierarchical state transition diagram.

1.7.3 Design Modeling

In the **design modeling** phase, the software architecture of the system is designed; that is, the analysis model is mapped to an operational environment. The analysis model (which emphasizes the problem domain) is mapped to the design model (which emphasizes the solution domain). Subsystem structuring criteria are provided to structure the system into subsystems, which are considered as aggregate or composite objects. Special consideration is given to designing distributed subsystems as configurable components that communicate with each other using messages.

Each subsystem is then designed. For sequential systems, the emphasis is on the object-oriented concepts of information hiding, classes, and inheritance. For the design of concurrent systems, such as real-time, client/server, and distributed applications, it is necessary to consider concurrent tasking concepts in addition to object-oriented concepts.

1.8 Modeling Software Product Lines with UML

The field of software reuse has evolved from reuse of individual components toward large-scale reuse with software product lines. Software modeling approaches are now widely used in software development and have an important role to play in software product lines. Modern software modeling approaches, such as UML, provide greater insights into understanding and managing commonality and variability by modeling product lines from different perspectives.

The UML-based software design method for software product lines described in this book is called PLUS (***P***roduct ***L***ine ***U***ML-Based ***S***oftware Engineering). The PLUS method extends the UML-based modeling methods that are used for single systems to address software product lines. With PLUS, the objective is to explicitly model the commonality and variability in a software product line. PLUS provides a set of concepts and techniques to extend UML-based design methods and processes for single systems to handle software product lines.

The PLUS method is similar to other UML-based object-oriented methods when used for analyzing and modeling a single system. Its novelty, and where it differs from other methods, is the way it extends object-oriented methods to model product families. In particular, PLUS allows explicit modeling of the similarities and variations in a product line.

In order to understand the product line and develop a model of it, an analyst needs to consider several different perspectives of the product line. A product line model is therefore a multiple-viewpoint representation of the product family, such that each viewpoint presents a different perspective on the family. The different viewpoints are developed iteratively. By analyzing the different viewpoints of the family, an analyst can get a better understanding of the product line. For product line analysis and modeling, the object-oriented analysis and design method used for individual systems is extended to product lines. Requirements, analysis, and design models of the product line are developed.

1.9 UML as a Standard

This section briefly reviews the evolution of UML into a standard. The history of UML's evolution is described in detail by Kobryn (1999). UML 0.9 unified the modeling notations of Booch (1994), Jacobson (1992), and Rumbaugh et al. (1991). This version formed the basis of a standardization effort, with the additional involvement of a diverse mix of vendors and system integrators. The standardization effort culminated in submission of the initial UML 1.0 proposal to the Object Management Group (OMG) in January 1997. After some revisions, the final UML 1.1 proposal was submitted later that year and adopted as an object modeling standard in November 1997.

The first widely used version of the standard was UML 1.3. There were minor revisions with UML 1.4 and 1.5. A major revision to the notation was made in 2003 with UML 2.0. Many books on UML, including the revised editions of the major UML references—*The Unified Modeling Language User Guide* by Booch et al. (2005) and *The Unified Modeling Language Reference Manual* by Rumbaugh et al. (2005)—conform to UML 2.0. Other books describing the UML 2.0 standard include the revised edition of *UML Distilled* by Fowler (2004), *UML 2 Toolkit* by Eriksson et al. (2004), and the revised edition of *Real-Time UML* by Douglass (2004).

1.9.1 Model-Driven Architecture with UML for Software Product Lines

The OMG maintains UML as a standard. In the OMG's view, "modeling is the designing of software applications before coding." The OMG promotes model-driven architecture as the approach in which UML models of the software architecture are developed prior to implementation. According to the OMG, UML is methodology-independent; UML is a notation for describing the results of an object-oriented analysis and design developed via the methodology of choice.

A UML model can be either a platform-independent model (PIM) or a platform-specific model (PSM). The platform-independent model is a precise model of the software architecture before a commitment is made to a specific platform. Developing the PIM first is particularly useful because the same PIM can be mapped to different middleware platforms, such as COM, CORBA, .NET, J2EE, Web Services, or another platform. The approach in this book is to use the concept of model-driven architecture to develop a component-based software architecture for a product line, which is expressed as a UML platform-independent model.

An object-oriented analysis and design method for software product lines needs to extend single-system analysis and design concepts to model product lines, in particular to model the commonality and variability in the product line, and to extend the UML notation to describe this commonality and variability. The goal is to extend UML for software product lines using the standard UML extension mechanisms of stereotypes, constraints, and tagged values (Rumbaugh et al. 2005) (see Appendix A, Section A.10).

1.10 Related Texts

This book presents a comprehensive UML-based object-oriented method addressing requirements modeling, analysis modeling, and design modeling for software product lines. This book complements books in other areas.

There are several books on UML-based object-oriented analysis and design for single systems. These include the set of three books by Booch, Jacobson, and Rumbaugh (Booch et al. 2005; Jacobson et al. 1999; Rumbaugh et al. 2005), as well as other UML books, such as Eriksson et al. 2004 and Fowler 2004, books on tool usage with UML (Quatrani 2003), books on real-time design with UML (Douglass 2004; Gomaa 2000), and books on using patterns with UML (Douglass 2002; Larman 2002).

There are few books on software reuse and software product lines. The Jacobson book on software reuse (Jacobson et al. 1997) addresses object-oriented software reuse in general and introduces the topic of variability in use cases and classes using the variation point concept. Weiss's book on software product lines (Weiss and Lai 1999) is a comprehensive treatment of this topic, with particular emphasis on how information hiding can be used effectively in product line design. The book on software product lines (Jazayeri et al. 2000) consists of a collection of papers published as a result of a European industry/university collaborative project. The book provides interesting research-oriented perspectives on product lines, including some papers on software architecture. Bosch also addresses software architecture and product lines in an interesting book (Bosch 2000) that covers a range of topics on software architecture and addresses both technical and management issues in software product lines. Clements and Northrop (2002) provide a very good introduction to the field of software product lines and coverage of the major management issues (in particular the practice areas of technical management and organizational management for product lines), as well as detailed case studies of organizations that have successfully developed software product lines.

There are a growing number of books on component technology, which describe the design of individual components (Brown 2000; Szyperski 2003) or UML components (Cheesman and Daniels 2001). These books complement this book but do not address how to design components so that they can be incorporated into software product lines.

There are several books on software architecture, including Shaw and Garlan 1996 and Bass et al. 2003, and also a UML-based book (Hofmeister et al. 2000). Bass et al. 2003 has some chapters on software product lines, a topic that is not addressed by the other two books. Finally, there are several books on architecture and design patterns, most notably Buschmann et al. 1996, Gamma et al. 1995, Larman 2002, and Schmidt et al. 2000. However, none of these books describe patterns in terms of how they could be incorporated into software product lines.

This book places an emphasis on object orientation, UML, designing software architectures for product lines, and addressing how to incorporate component and pattern technology into software product lines.

1.11 Summary

This chapter discussed software reuse and the reason for developing software product lines, which are also referred to as software product families, as well as modeling variability in software product lines. It also introduced the UML notation, which is used throughout this book, and described the concept of model-driven architecture. Chapter 2 will describe important design and architecture concepts for software product lines. Chapter 3 will describe the software product line engineering process. Appendix A provides an overview of the UML notation.