# A Survey of Product-Line Architectures

Maarit Harsu
Software Systems Laboratory
Tampere University of Technology
P.O. Box 553, 33101 Tampere
e-mail: `firstname.lastname at tut.fi`

## Contents

**1 Introduction** **3**

**2 Domain analysis and domain engineering** **5**
2.1 Domain and domain analysis . . . . . . . . . . . . . . . . . . . . . 5
2.2 Domain engineering . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Commonality analysis . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Variation management** **10**
3.1 Variation categories . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.2 Variation levels . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
3.3 Variation mechanisms . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.4 Modeling variation with design patterns . . . . . . . . . . . . . . . 14

**4 Design and styles** **18**
4.1 Architectural styles . . . . . . . . . . . . . . . . . . . . . . . . . . 18
4.2 Architecture design . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.3 An example of an architectural design model . . . . . . . . . . . . 23

**5 Modeling and description** **25**
5.1 Different views of architectural description . . . . . . . . . . . . . 25
5.2 Architectural description languages . . . . . . . . . . . . . . . . . 26
5.3 Examples of architectural description languages . . . . . . . . . . 28
5.3.1 Koala . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
5.3.2 C2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
5.4 Architectural description with UML . . . . . . . . . . . . . . . . . 30

5.4.1 Introducing UML . . . . . . . . . . . . . . . . . . . . . . 30
5.4.2 Applying UML . . . . . . . . . . . . . . . . . . . . . . . 31
5.5 Variation modeling . . . . . . . . . . . . . . . . . . . . . . . . 34

**6 Analysis and assessment** **36**
6.1 Quality attributes . . . . . . . . . . . . . . . . . . . . . . . . . 36
6.2 Architectural assessment . . . . . . . . . . . . . . . . . . . . . 37
6.2.1 Scenario-based assessment . . . . . . . . . . . . . . . . . 37
6.2.2 Questionnaire-based assessment . . . . . . . . . . . . . . 39
6.2.3 Checklist-based assessment . . . . . . . . . . . . . . . . . 39
6.2.4 Simulation-based assessment . . . . . . . . . . . . . . . . 40
6.2.5 Metrics-based assessment . . . . . . . . . . . . . . . . . . 41

**7 Development and evolution** **43**
7.1 Architecture development . . . . . . . . . . . . . . . . . . . . . 43
7.2 Architecture evolution . . . . . . . . . . . . . . . . . . . . . . . 44
7.2.1 Classifying the modifications . . . . . . . . . . . . . . . . 45
7.2.2 Managing the modifications . . . . . . . . . . . . . . . . . 46

**8 Recovery and reengineering** **49**
8.1 Reengineering steps . . . . . . . . . . . . . . . . . . . . . . . . 49
8.2 Recovery steps . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
8.2.1 Recovering single systems . . . . . . . . . . . . . . . . . 51
8.2.2 Recovering a product-family . . . . . . . . . . . . . . . . 52
8.3 Recovery approaches . . . . . . . . . . . . . . . . . . . . . . . 52
8.4 Migration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**9 Reuse** **55**
9.1 Reuse situations . . . . . . . . . . . . . . . . . . . . . . . . . . 55
9.2 Reuse problems . . . . . . . . . . . . . . . . . . . . . . . . . . 55
9.3 Reuse supported by frameworks . . . . . . . . . . . . . . . . . 58

**10 Testing** **59**
10.1 Testing in general . . . . . . . . . . . . . . . . . . . . . . . . . 59
10.2 Testing in product lines . . . . . . . . . . . . . . . . . . . . . . 61
10.2.1 Testing in core asset development . . . . . . . . . . . . . 62
10.2.2 Testing in product development . . . . . . . . . . . . . . 62
10.3 Architecture testing based on formal descriptions . . . . . . . . . 63

# 1 Introduction

The architecture of a software system defines that system in terms of computational components and connections among those components [SG96, p. 1]. A software product line, in turn, is a set of systems which share a common software architecture and set of reusable components [Bos00, p. 2]. According to Jazayeri et al., a product family software architecture (a product-line architecture) defines the concepts, structure, and texture necessary to achieve variation in features of variant products while achieving maximum sharing parts in the implementation [JRvdL00, p. 27].

Concerning software architectures in general, different architectures may have different styles and different architectures support different quality attributes. Architectural styles are closely related to patterns such that a certain style may be best suited a particular type of problem. To make sure that an architecture fulfills its quality requirements, the architecture must be analyzed and assessed against these requirements. Creating an architecture is the first step in creating a system. Architecture description is needed when moving from an architectural design to a code framework. For explicit description, there are architectural description languages.

Referring especially to product-line architectures, an important task is to analyze the domain and to identify the commonalities and variabilities of the objects and operations of that domain. Product-line architectures can also be found from existing systems by analyzing their commonalities. This is called architecture recovery. After using the products of a product line, new requirements usually arise for these products. These requirements may suggest modifications also to the product-line architecture. Thus, architectures can evolve.

This report provides an overview to product-line architectures considering the aspects described above. The following areas concerning the topic are discussed:

- domain analysis and domain engineering,
- variation management,
- design and styles,
- modeling and description,
- analysis and assessment,
- development and evolution,

- recovery and reengineering,
- reuse,
- testing.

The areas considered in the report are not separate. For example, variation among the products of the same family (Section 3) is analyzed during domain engineering (Section 2), and variation among components (Section 3) enables reusing those components (Section 9). In addition, architectural recovery (Section 8) is associated with domain engineering (Section 2) because recovery exploits commonality analysis belonging to domain engineering. Thus, these areas are connected to each other, although they are considered in different sections. Besides the subjects considered in this report, there are areas that are related to product-line architectures but not discussed here in detail. Such are, for example:

- components,
- patterns,
- frameworks,
- architectural methods.

# 2 Domain analysis and domain engineering

Domain analysis considers the scope of the domain covering the objects, operations, and relationships of a certain area. The result of domain analysis is domain model which describes the identified objects, operations, and relationships. Domain analysis is an early step in any programming project, not only in product-line architectures. Domain engineering, in turn, is associated with product-line architectures. It considers both the commonalities and variabities among the product-family. According to Ardis et al., the first stage of domain engineering is domain analysis in which domain experts collect and document their knowledge of the product family [ADH+00]. Besides domain analysis and domain engineering, there is a related term, ”scoping”, meaning determination the boundaries of the product line.

## 2.1 Domain and domain analysis

The term ”domain” can be used in several associations [Sch00]:

- business area,
- collection of problems (problem domain),
- collection of applications (solution domain),
- area of knowledge with common terminology.

Domain analysis can apply different approaches. It can concentrate on describing what is inside the domain, what is the boundary of the domain, or what is outside the domain [Sch00]. The first case describes the items that constitute the domain, or it identifies other domains that together form the actual domain (domains can have sub-domains). The second case describes the rules of inclusion and exclusion. In addition, structure and context diagrams can be produced both to describe the boundary of the domain and to show the relation of the domain to the outside.

Domain analysis is concerned with product-line architectures, but it can be exploited in other contexts, too. It can be used in considering legacy systems and in exploring how to transform legacy systems into a common architecture as follows [BCC+99, pp. 31–32]:

- Legacy products are analyzed to consider whether they are appropriate to be used in product lines. This includes comparing the functional capabilities of the products to those needed in product lines.

- The general product concept can be analyzed for feasibility. Domain analysis is used to give understanding about the structure and state of the domain to which a product line should be constructed. Legacy products represent the history of a domain while the product line to be built represents its future, and former informs the latter.

- Domain requirements can be analyzed in order to maintain them according to changing market needs and technologies. When the product line evolves, also the domain model must evolve. Moreover, the domain model must accommodate to new product requirements.

## 2.2 Domain engineering

Domain engineering is associated with product-line architectures. It studies how the products of the same family shares the common basis and how they differ from each other [ADH+00]. According to Bergey et al., domain engineering means development and acquisition of the core assets of the product line [BFG+00, p. 5].

Domain engineering is a part of the Family-Oriented Abstraction, Specification, and Translation (FAST) process [ADD+00, ADH+00, CHW98, WL99]. The FAST process is a product-line development process covering all the phases of producing families of architectures. Figure 1 introduces FAST process which is divided into two phases: domain engineering and application engineering. The purpose of domain engineering is to understand the relationships among the products of the product family: both their commonalities and variabilities. This understanding is translated into technology such as a common set of subroutines or a domain-specific language. This technology is called application engineering environment. Application engineering uses this environment to produce the members of the product family. Feedback from application engineering suggests modifications to the application engineering environment. The modifications are made after considering their impact on the original domain analysis effort.

Domain engineering (and FAST process) involves cost estimation of the product-line approach. It should be decided whether there are sufficient potential family members to justify the investment in domain engineering. It should also be considered to what extent it pays to generate family members. At the early initiation stage, the product-line approach requires more costs and provides less benefit than producing a single product. However, further deployment of products from the product line will be more efficient.

Domain engineering is very close to scoping that defines which products and

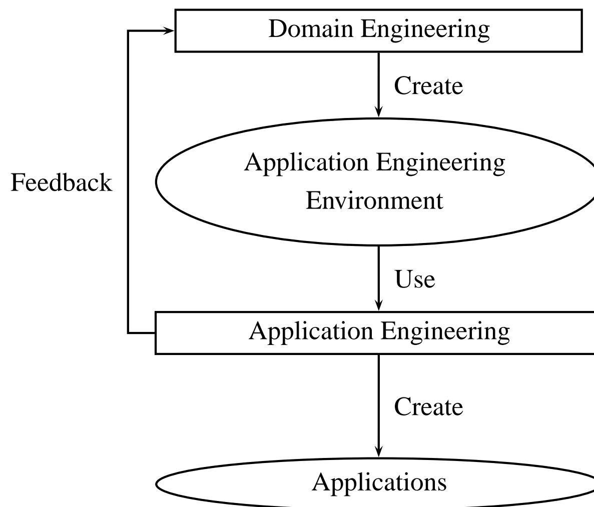


Figure 1: Domain engineering according to the FAST process [ADH+00]

features are included in the product line and which ones are excluded. There are three different levels of scoping: product line, domain and asset base [Sch00]:

**Product-line scoping**
: identifies the specific requirements and individual products that should be part of the product line.

**Domain scoping**
: identifies appropriate boundaries for conceptual groups of functionality that are relevant to the domain.

**Asset scoping**
: identifies the various elements that should be made reusable.

## 2.3 Commonality analysis

As mentioned, domain engineering considers the similarities and differencies between the members of the product family. This kind of analysis is called commonality analysis, although it covers also considering the variabilities.

Commonality analysis identifies and makes useful the abstractions that are common to all family members [Wei98]. There are two main sources of abstrac-

tion: terminology and commonalities. Terms concerning product-line architecture make communication among developers easier and more precise. As another source of abstraction, commonalities are actually assumptions that are true for all family members. Besides commonalities, it is important to consider variabilities among family members. Variabilities provide a way to prepare for potential changes by pointing those decisions concerning family members that are likely to alter over the lifetime of the family.

The result of commonality analysis is commonality document consisting of the following sections [ADH+00]:

**Overview**
: describes the domain and its relation to other domains.

**Definitions**
: provide a standard set of technical terms.

**Commonalities**
: consist of a structured list of assumptions that are true for every member of the family.

**Variabilities**
: consist of a structured list of assumptions about how family members differ.

**Parameters of variation**
: consist of a list of parameters that refine the variabilities, adding a value range and binding time for each.

**Issues**
: form a record of important decisions and alternatives.

**Scenarios**
: are examples used in describing commonalities and variabilities.

Commonality analysis can be used for several purposes [Wei98]. It can be used in the further phases of the FAST process, for example, in designing a domain specific language and then in generating code and documentation from the language specification for each product. It serves as a basis for a family architecture and as reference documentation. Commonality analysis can also be exploited in reengineering the members of a product family. It can be used as a training aid for new project members. In addition, a plan for evolution of the family can be derived from commonality analysis.

Variation management is an important topic in software product lines. Although it is a part of domain analysis, it is considered as an own section (Section 3).

# 3 Variation management

The products belonging to the same product family has much in common. However, there are also variation among the products of the same family. Variation is provided according to different users or different design and implementation requirements. Variation can be identified during domain analysis. It is important to take variation into account in the early phases in designing a product-line architecture and handling variation in the architecture level instead of code level [CAJ98].

Variation is associated with reusability. To enable reuse, components should be applicable in different contexts which suggests that there must be variation among the components. Adjusting a component is achieved in two ways: via component variation and via component adaptation [Bos00, pp. 224–]. Variation is possible in particular variation points in which the behavior of the component can be changed. These variation points should be decided during component design. Adaptation is needed when the variability of a component is not sufficient.

Variation is also associated with architecture evolution. Evolution affects on how variability is handled in software product lines [SB00]. For example, variability can be handled by selecting component implementations. In addition, component interfaces may evolve, affecting the way they can be used in variant products.

## 3.1 Variation categories

The requirements or feature properties of product lines can be divided into following categories [CAJ98, KCH+90, LM97, TCY93, Tra95]:

- mandatory requirements are supported in all systems in a domain,
- optional requirements are only required in some systems,
- alternative requirements are alternatives for each other,
- prerequisite requirements are needed for other requirements.

From another point of view, variability can be divided into three categories called axes of variability [DMNS97, MHM98]:

**Feature variability**
: means variation in the definition and implementation of a specific feature or additional features. Such are, for example, variation in checking the duplication of messages, or providing a choice of pleasing alerts in addition to a standard alert.

**Hardware platform variability**
: means variation in the type of microcontroller, memory, and devices that need to be supported.

**Performances and attributes variability**
: means variation in the required performances such as number of back-to-back messages to be received, and in the attributes such as failure handling and concurrency support.

## 3.2 Variation levels

Variability can occur at different levels in the design [SB00]. These levels are:

- product line level,
- product level,
- component level,
- sub-component level,
- code level.

Variability at product line level defines how different products in the product line varies. Components for different products are selected, and product-specific code to be used is selected or generated.

Variability at product level defines the architecture and choice of components for a particular product. The components are fitted together to form a product architecture, and the product-specific code is customized for the particular product variation. At this level, it is also considered how to cope with evolving interfaces.

Variability at component level defines the component implementations to be selected into the product. A component can be considered as an abstract object-oriented framework with several framework implementations. At this level, the set of framework implementations is selected. It is taken into account how to enable addition and use of several component implementations, and how to design the component interface to adapt to the addition of more concrete implementations.

Variability at sub-component level defines the features to form a component for a particular product. All features of a component are not needed in all products. Thus, to avoid dead code, these unnecessary features should be removed.

However, removing features may affect to other components, and their implementations may require modifications, too. As a consequence, variability at this level concerns removing and adding parts of a component.

The actual evolution and variability described above is implemented at code level. If variability has been considered properly at the upper levels, the code level calls only for checking that the provided class interfaces match the required interfaces. However, when the components and classes evolve, also their interfaces may change. Several components and component implementations use these interfaces. In addition, different products may use different versions of the component implementation and its interface. All these variabilities must be considered at code level.

As shown, variabililty occurs at different levels. However, it is important to consider variability already at the higher architectural level, not only at code level [CAJ98]. It is more intuitive to think about variation at a higher level before implementing them at a lower level. Business goals and constraints can be expressed more naturally at a higher level, when implementation details need not be considered yet. Different types of variation can be implemented by customizing architecture at the design level. Moreover, variation at the architecture level may be equivalent to variations at the code level. Thus, less work is required at the code level, if variation is considered already at higher levels.

## 3.3 Variation mechanisms

There are different mechanisms to enable variation [Bos00, SB00]:

**Inheritance**

Inheritance can be used if the component is implemented as a class in an object-oriented language. Through inheritance and late binding, each component can be specialized from its superclass for each specific context.

**Extensions**

Extension means such kind of variation that the user selects one of different behavioral variants. There is typically the stable functionality, and each variable functionality is modeled as an independent entity. The user can select an existing variant entity or introduce a new one. For example, the strategy design pattern [GHJV95] uses extensions.

**Configuration**

Configuration allows variation where all variants are present at all variation

points. The user may select appropriate files and set parameters to connect modules and components to each other.

**Parametrization, templates, and macros**
These variability techniques are used when parameters or macro expressions can be introduced and later instantiated with the actual parameter or by expanding the macro. In template instantiation, components are configured with application-specific types. This variation can be applied in list or queue implementations for different element types.

**Generation**
A generator requires its input to be a specification written in some domain-specific or component-specific language. The generator then translates this specification into a source-code-level component which can be attached to the product or application. For example, graphical user interfaces can be generated from graphical or textual specifications.

**Compiler directives**
Compiler directives (like `ifdef` in C++) can be used at compile-time to select between different implementations in the code.

Different variation mechanisms can be applied at different variation levels. Configuration is mainly used in product-line level, product level, and component level. At the product-line level, it can be applied to select the components and the product-specific code. At the product level, the selected components are connected together. At the component level, the actual concrete implementations are selected to include into the product. Configuration may also be used at sub-component level, if components have been designed as a collection of disjoint sub-components. In this case, configuration management can be used to select the specific parts of the components.

The same three levels (product-line level, product level, and component level) may also apply compiler directives and parametrization. At the product-line level, compiler directives can be used to remove unnecessary product-specific code. At the product level, both of the techniques can be used to connect components to each other. However, they allow only a static way of connecting components. Compiler directives and parametrization at sub-component level are not recommended because they may lead to dead code and the complexity of the code.

Inheritance and extensions can be used at all the levels of variability. However, they are especially important at class level (code level). They provide a way to divide the source code into several files. Instead of using inheritance, templates can

usually be chosen. However, at sub-component level, templates are not always suitable. Usually more than one extension is allowed to be present in a system, and this is not technically possible when using templates.

Generation is best suited for product level. At that level, code needs to be instrumented with product-specific code and to connect the components to each other.

Karhinen and Kuusela introduce a different division for variation mechanisms and their applicable levels [KK98]:

**Implementation configuration**
: provides one design for all products. The design is very simple because variation is managed at implementation level. At that level, conditional compilation and source code configuration management are applied. However, the more products the product line comprises, the more complex the implementation becomes.

**Customization**
: supports variation by one universal product that can be customized, for example, for different customers. All possible components must be present in the design and implementation of the product. The active set of components is selected for each product.

**Modularization**
: places variation to structural elements of the design. The common part of the design is a framework, and variants are produced by selecting existing and specifying new components.

**Design configuration**
: handles variation at the design level. Each product has a different design. The management of different designs depends on the tool support handling the dependencies between the products of the family.

## 3.4 Modeling variation with design patterns

Besides the aforementioned approaches to model variation, variation can be represented using patterns [KM99]. Patterns provide reusable, routine solutions to certain types of problems and support the reuse of the solution implementations. A pattern is typically presented as a class diagram and as an object diagram.

In constructing a family model that supports commonality and variability, patterns associated with discriminants can be applied. A discriminant is any feature (or requirement) that distinguishes one system from another. There are three basic types of discriminants [KM99]:

**Single discriminants**
: are a set of mutually exclusive features, only one of which can be used in a system. For example, a mobile phone has one display which can vary between different phones.

**Multiple discriminants**
: are a set of optional features that are not mutually exclusive. At least one of them must be used. For example, there must be at least one way of giving a call on mobile phones, but there can be several alternatives such as pressing the digits, pressing redial, or voice dialing.

**Option discriminants**
: are single optional features that might or might not be used. For example, mobile phones can have an internet connection facility, but it is not mandatory.

The above division of discriminants is closely related to the division of feature properties into mandatory, optional, alternative, and prerequisite requirements introduced in Subsection 3.1. However, discriminants do not provide mandatory nor prerequisite elements, while multiple discriminant do not belong to the feature properties.

The single discriminant can be presented as an inheritance hierarchy in which generic features are described in a base class and specific ones correspond the subclasses. Only one subclass can be instantiated in a system. The set of subclasses is called realm. When necessary, this set can be extended for a new system (see Figure 2). Virtual functions are used in the base class in order to enable also other classes to access the methods in a subclass (Operation `B` in Figure 2). It is possible to refer to a base class instance without knowing the used subclass. However, this single subclass instance must have a well-known name to enable references to it. The single instance is implemented using the singleton pattern [GHJV95]. A subclass must not introduce new methods because otherwise replacing it by other subclasses would not be possible.

The multiple discriminant is also presented as an inheritance hierarchy like the single discriminant. However, now more than one subclass can be instantiated in a system. As shown in Figure 3, each subclass has one instance and the set of

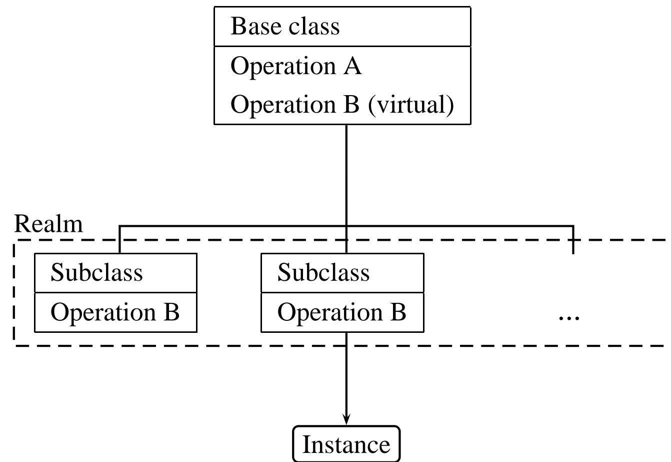


Figure 2: A class and object diagram for a single adapter pattern [KM99]

instances is held in a collection. The collection is needed to identify and to collect different subclass instances by a name or another unique identification. This is necessary because accessing methods in a particular subclass must be possible.

Optional features are presented by creating two associated peer classes. (Inheritance is not used because it is not optional.) The associated classes must have a 0-1 relationship on at least one end (see Figure 4). Class `B` is optional for class `A`. `A` does not assume that `B` exists. Thus, `A` can be reused whether or not `B` is reused.

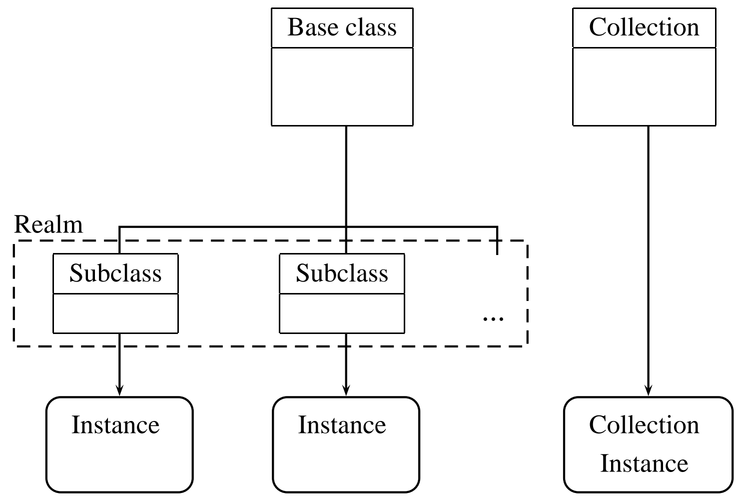


Figure 3: A class and object diagram for a multiple adapter pattern [KM99]

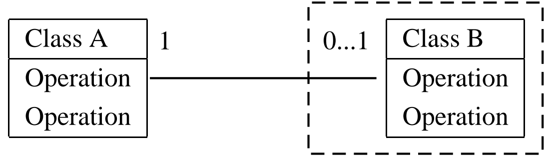


Figure 4: A class and object diagram for the option pattern [KM99]

# 4 Design and styles

Architectures consist of components and connections between these components. However, there are different alternatives to relate these components and connections together. These alternatives can be called architectural styles. Styles are associated with architectures in general, not especially with product-line architectures. However, architectural styles show the common aspects between different architectures. They can also be used in guiding the design of an architecture. When designing an architecture, the existing architectural styles can be taken into account and considered which of them best suits the current situation.

## 4.1 Architectural styles

Architectural styles can also be called system patterns (cf. design patterns) [BCK98, p. 93] or architectural patterns [BMR+96]. An architecural style represents a collection of design decisions that have already been made and can be reused. It consists of a few key features and rules for combining those features so that architectural integrity is preserved. An architectural software style is determined by the following [BCK98, p. 94]:

- a set of component types performing some function at run-time (such as data repository, process, and procedure),
- a topological layout of the components according their relationships at run-time,
- a set of semantic constraints (for example, a data repository is not allowed to change the values stored in it),
- a set of connectors providing communication, coordination, or co-operation among components (such as subroutine call and data streams).

Actually, a style defines a class of architectures. In other words, it is an abstraction for a set of actual architectures applying that style. We cannot usually find clear occurencies of particular styles in system designs, instead, the styles appear in slightly different forms. However, style catalogs are important because they reveal when two identified styles are similar. They also inform about the situations in which a particular style can be applied. A small catalog of architectural styles is shown below [BCK98, SG96]:

**Data-centered architectures**

emphasize integrability of data. They are appropriate for systems that describe the access and update of a widely accessed data store. Subtypes of

these architectures are repository, database, hypertext, and blackboard architectures.

**Data-flow architectures**
: emphasize reuse and modifiability. They are appropriate for systems that describe transformations on successive pieces of input data. Data enters the system and then flows through the components one at a time until some final destination is reached. Subtypes of these architectures are batch-sequential and pipe-and-filter architectures.

**Virtual machine architectures**
: emphasize portability. They simulate such functionality that is not native to the hardware or software on which it is implemented. They can, for example, simulate platforms that have not yet been built (such as new hardware) or "disaster" modes that would be too complex or dangerous to test with the real system (such as flight and safety-critical systems). Examples of these architectures are interpreters, rule-based systems, and command language processors.

**Call-and-return architectures**
: emphasize modifiability and scalability. They are the most general architectural styles in large software systems. Subtypes of these architectures are main-program-and-subroutine architectures, remote procedure calls, object-oriented systems, and hierarchically layered systems.

**Independent component architectures**
: emphasize modifiability by separating various parts of the computations. They consist of independent processes or objects that communicate through messages. They send data to each other but do not directly control each other. The messages can be passed to named receivers or they can be broadcast such that interested participants pick up the messages. Subtypes of these architectures are communicating processes and event systems.

Besides the above styles, Buschmann et al. introduce the following styles and patterns to be exploited in each style [BMR+96]:

**Distributed systems**
: include one architectural pattern: broker. The broker pattern consists of decoupled components that interact by remote service invocations. A broker component coordinates communication by forwarding requests and by transmitting results and exceptions.

**Interactive systems**
: include model-view-controller and presentation-abstraction-control as their

patterns. Both of these patterns support human-computer interaction. The model part of the former one contains the core functionality and data. Views display information to the user, while controllers handle user input. Views and controllers together form the user interface. The latter kind of pattern provides a hierarchy of co-operating agents. Each agent manages a specific functional part of the system and consists of three components: presentation, abstraction, and control. With this division, human-computer interaction can be separated from the functional aspect of each agent and from the mutual communication of the agents.

**Adaptable systems**
include two patterns: microkernel and reflection. These patterns support applications to extend and to adapt to evolving technology and changing requirements. The microkernel pattern supports adaptation to changing requirements by separating the minimal functional core from the extended functionality and customer-specific parts. The reflection pattern supports dynamic changes to software systems. In this pattern, an application is divided into two parts. The meta level provides information about the selected system properties and makes the system self-aware. The base level includes the application logic.

Architectural styles can be associated with architecture analysis and quality attributes (considered in Section 6). The resulting styles are called attribute-based architecture styles [KKB+99]. Architectural styles define the conditions under which they should be used. In addition to defining the components and connectors, attribute-based architecture styles include quality-attribute-specific model that declares the behavior of component types interacting with each other. For example, with pipe-and-filter architectures, it should be considered how performance is handled. In addition, attention should be paid to the assumptions made by filters that effect their reuse.

## 4.2 Architecture design

Architecture design covers different aspects some of which are considered elsewhere in this report. For example, Section 2 introduces scoping, and architecture assessment is included in Section 6. However, in this subsection, we describe the outline of the design process concerning product-line architectures.

Bosch has considered the design of a product-line architecture [Bos00, pp. 189–]. In his model, the design consists of the following steps:

- business case analysis,
- scoping,
- product and feature planning,
- actual design of the product-line architecture (consisting of):
  - functionality-based architectural design,
  - architectural assessment,
  - architecture transformation,
- component requirement specification,
- validation.

Business case analysis makes sure that the software product-line approach will be paying. This phase also considers different ways to move to product-line-based production. Examples of these ways are revolutionary and evolutionary paths.

Scoping uses the results of the business case analysis as a basis in selecting the products and product features which will be included in the product line. Scoping also defines which features include in which products.

Product and feature planning considers the characteristics of subsequent versions of the product-line architecture. As the architecture evolves, it becomes actual to include new products and new features in the product-line architecture. Future inclusion is easier, if these potential new products and features have been considered before-hand.

Design of the product-line architecture is the main step of the process. Design process can be considered to consist of three steps: functionality-based architectural design, architectural assessment and architecture transformation. These steps are considered below.

Functionality-based architectural design first defines the product context in which the software architecture operates. In the case of product-line architectures, the contexts are not necessarily specified for the product line as a whole. Instead, single products of the same product line differ from each other according to their supported context. The next step in functionality-based architectural design is the identification and definition of the core abstractions of the product line. Correspondingly, the components are defined as instances of these abstractions.

The final step in functionality-based architectural design verifies the suitability of the selected abstractions and the ability of the current architecture to represent all variations of the product.

Architecture assessment evaluates the product-line architecture against its quality requirements. Architecture assessment techniques are, for example, scenarios, simulation and mathematical models (see Section 6). However, evaluating all the products of the family would be too expensive and time-consuming. Thus, we can concentrate on assessing those products that contain the critical features in the product line. Alternatively, we can concentrate on evaluating extreme products such as the largest, the smallest, etc. Assessment also covers evaluating the capability of future inclusion of new features and products. This kind of evaluation tells about the evolvability and maintainability of the product-line architecture.

Architecture transformation is concerned with improving the quality attributes of a software architecture. A product-line architecture should support three transformation aspects: variants, optionality, and conflicts. Most often, it is necessary to provide two or more solutions (variants) for subsets of the product line. For example, the layered architectural style supports variants because variants can be encapsulated as layers. In addition, many design patterns [GHJV95] (such as abstract factory, strategy, and mediator) support variants. Optionality means that for some products in the product line, certain components can be excluded. Transformations may be needed to reduce the dependency on optional components. However, accessing these components should be allowed when needed. For example, the blackboard architectural style supports optionality, since components depend only on the blackboard, not on other processes. In addition, some design patterns (such as proxy and strategy) support optionality. The design considerations of the product-line architecture may reveal conflicts between the product-line architecture and the requirements of individual products. If the conflict is handled in the product-line architecture, some transformations are needed. Some design patterns (such as adapter, proxy, and mediator) may prove to be useful in resolving conflicts.

Component requirement specification is important in the design phase because software architecture defines a set of components that implement the required behavior. When exploiting components, the software engineer has to know the functional and quality requirements of the components: which products use the components and how the components should be instantiated for each product.

Validation ensures that the product line supports the features defined during scoping, that it can be easily instantiated for each product, and that planned new

```
(a)                                    (b)
 S = {a, b, c}                          S := a | b | c ;
 T = {d[S], e[S], f[S]}                 T := d S | e S | f S ;
 W = {n[W], m[W], p, q[T, S]}           W := n W | m W | p | q T S ;
```

Figure 5: Realms, components, and grammars [BSC99]

feature can be easily included.

## 4.3 An example of an architectural design model

As an example of an architectural design model we consider GenVoca [BO92, BG97, BSC99, BCS00]. GenVoca is a design method for building architecture-based software that is extensible via component addition and removal. GenVoca is based on stepwise refinement. Refinements are scaled to components or layers. Complex applications are expressed as a composition of a few refinements rather than a huge amount of small refinements.

GenVoca model is based on components and sets of components called realms [BO92]. Each component has an interface and an implementation. Every component is a member of a realm `T`, where all members of `T` realize exactly the same interface but in different ways. Thus, the members of the same realm are plug-compatible and interchangeable.

In Figure 5a, realms `S` and `T` have three components, while realm `W` has four. An application is a named composition of components, for example, as follows:

```
A1 = d[b];
A2 = f[a];
```

Application `A1` composes component `d` with `b`, and `A2` composes `f` with `a`. These applications are interchangeable implementations of `T` (because the outermost components of both belong to realm `T`). Composing components is equivalent to stacking layers, and thus, GenVoca considers the terms ”component” and ”layer” as synonyms.

Figure 5 shows the close relationship between realms and grammars. Realms and their components define a grammar whose sentences are applications. Figure 5a shows realms, and Figure 5b represents the corresponding grammars. As the

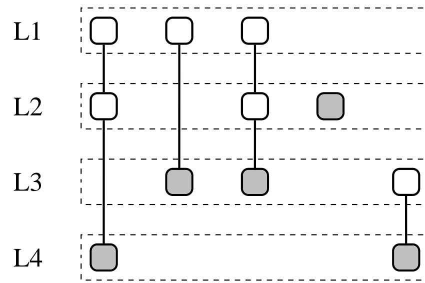


Figure 6: Creating inheritance hierarchies by composing layers [BCS00]

set of all sentences defines a language, the set of all component compositions defines a product line. Adding a new component to a realm is equivalent to adding a new rule to a grammar.

Figure 6 shows the situation from the point of view of layers (components and layers are treated as synonyms). The squares stand for classes, and the shaded classes are terminal classes. The solid vertical lines represent subclass relationships or refinements. (Although other horizontal relationships may exist between classes, they are not shown.) Large-scale refinements involve subclassing to several classes at the same time. The encapsulation of these refinements defines a layer. Each layer is enclosed by a dashed rectangle.

A layer implements a feature that can be shared by many applications of a product line. An application that supports particular features is defined by a composition of layers that implements those features. Thus, layers are the basic building blocks for families of applications.

GenVoca provides a flexible way to design product-line architectures. It shows the building blocks of frameworks and framework instances. It is possible to define the features that are needed in framework classes and instance classes. If these features require modification, GenVoca model allows adding, swapping, and removing components from previously defined compositions.

# 5 Modeling and description

Architectures are typically presented as boxes and lines connecting the boxes. However, it is not necessarily easy to understand the meaning of those items in architectural description. The boxes can, for example, represent components of the system, programs, source code fragments, or logical groupings of functionality. Correspondingly, the lines or arrows (connectors) can, for example, represent compilation dependencies, data-flow, control-flow, calling relationships, or part-of relationships. All such items and relationships is hard to describe in a single figure. There are different ways to describe an architecture, and for explicit modeling there are architectural description languages. As an alternative for these languages, UML (Unified Modeling Language) can also be used to describe an architecture.

Architectural description is associated with architectural styles and design. Architectural description methods are applied in the design process of the architecture. Architectural description is also concerned with variation. During the design process of a product-line architecture, it should be considered how to describe variation between products.

## 5.1 Different views of architectural description

Architectural description can be represented via several views such as 4 + 1 view model [Kru95]. Each of the first four views describes the system from a different view point while the fifth view illustrates and validates the earlier views. The five views are:

**Logical view**
: describes the functionality of the system provided for the end user. The abstractions of this view are derived from the problem domain. Logical view supports object-oriented architectural style.

**Process view**
: describes concurrency and synchronization aspects of the system. This view also takes into account performance, scalability, system integrity, and fault-tolerance. Process view supports several architectural styles, for example, pipes-and-filters and client-server styles.

**Development view**
: describes the organization of the system into modules or subsystems as hi-

erarchical layers. Each subsystem layer can be developed separately. Thus, development view supports layered architectural style.

**Physical view**
: describes the mapping of the system onto the hardware. Elements identified in the earlier views must be mapped onto different physical configurations, for example, according to different customers.

**Scenarios**
: ties the other views together and illustrates them with selected use cases or scenarios. Scenarios show how the other views work and fit together.

There are also different divisions of views. For example, Garlan and Sousa introduce four views: problem domain view, code view, run-time view, and deployment view [GS00].

## 5.2 Architectural description languages

Architectural descriptions can be specified in an explicit and precise way by using an architectural description language (ADL). These languages provide notations for representing architectural structures such as components, connectors, systems, properties, and styles. ADLs allow formal description of architectures. In addition, they support early analysis and feasibility testing of architectural design decisions [BCK98, p. 268].

ARES project has set some requirements for ADLs [JRvdL00, pp. 35–]:

**Level of abstraction**
: Representing the architecture of a large and complex system usually needs abstractions. Architectural views can be considered as some kind of abstractions. The semantics of components may also be complex, and thus, require abstraction.

**System construction in addition to system documentation**
: ADLs provide system documentation through different views. In addition, when the system changes, it would be useful to make the modification in the system description and to automatically propagate it to the system implementation.

**Handling of variations within a product family**
: Handling of variations in ADLs is usually poorly supported. However, it would be desirable to represent the variability within a product family at

an architectural level rather than at the program code level. Thus, an ADL should provide means to describe both the common architecture and the variable parts of each product.

**Definition of dynamic structures**
Most ADLs concentrete on representing static components and connectors. However, support for dynamic structures is needed for the run-time development of the system.

**Multiple system views and attributes**
It should be possible to make the relationships clear among different views and between the views and the basic architecture. However, it might not be sensible to show this information with the ADL itself, instead, the ADL would provide a framework for managing this kind of information.

**Description of hierarchical and layered architectures**
Hierarchical and layered architectures should be described at different levels of abstraction and detail. Similarly, components should be represented with various amounts of details. This makes large systems easier to manage.

**Tool support for architectural visualization and manipulation**
In addition to textual representation, there should be ways to visualize the architecture in a graphical form. Graphical representations support system comprehension, system navigation, and consistency management.

In addition to architectural description languages, there are architecture interchange languages such as ACME [GMW97]. These languages are meant to interchange architectural specifications across ADLs. For this purpose, ADLs should have a common basis for architectural representation including the following aspects:

- components,
- connectors,
- systems (configurations of components and connectors),
- ports (interaction points i.e. interface of a component),
- roles (interaction points i.e. interface of a connector),
- representations to model hierarchical compositions,
- mappings from the internal architecture of a composite component or connector to the external interface.

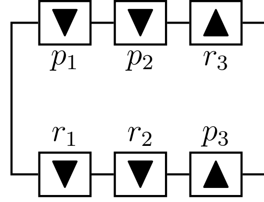


Figure 7: A Koala component [vO98]

## 5.3 Examples of architectural description languages

This subsection introduces two architectural description languages: Koala and C2. More examples of these languages and their mutual comparison is provided in [MT00].

### 5.3.1 Koala

Koala is meant to describe software architectures of embedded systems. It provides means to describe components, interfaces, configurations, bindings, etc. [JRvdL00, vO98]. An example of a Koala component is presented in Figure 7. Components can communicate with their environments through interfaces. Koala divides interfaces into incoming and outgoing interfaces. A component provides functionality through incoming interfaces, and in order to do so it requires functionality through outgoing interfaces. In Figure 7, components are shown as rectangles and interfaces as small squares containing triangles. The direction of a triangle indicates the direction of the corresponding function call.

Configurations are represented by connecting the interfaces of components. A requires-interface must always be bound to precisely one provides-interface, but a provides-interface may be bound to more than one (or zero) requires-interfaces.

To enable to describe large systems, Koala provides a recursive component model. It means that any combination of components can again be viewed as a component with provides- and requires-interfaces. In addition, Koala provides means to describe diversity, as will be considered in Subsection 5.5.

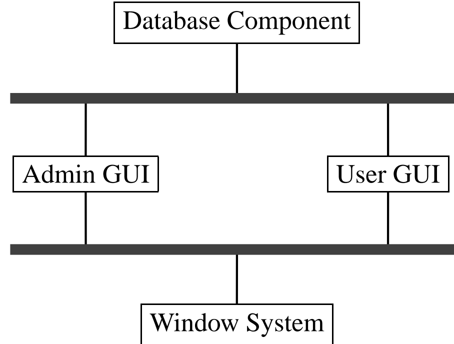


Figure 8: An example C2 architecture for a database application [RMRR98]

#### 5.3.2 C2

C2 is a software architecture style for user interface intensive systems [MT97, MR99, RMRR98, TMA+96]. C2 SADL is an architectural description language for describing C2-style architectures. Thus, we use C2 to refer to the combination C2 and C2 SADL. In C2-style architecture, connectors transmit messages between components. Components, in turn, maintain state, perform operations, and exchange messages with each other via two interfaces called top and down. Each interface defines a set of messages that may be sent and a set of messages that may be received. A message can be a request for a component to perform an operation or a notification that the component has performed the operation or changed state.

In C2 style, components may not directly exchange message, instead, they communicate via connectors. Each component interface may be attached to at most one connector. A connector may be attached to any number of other components and connectors. Request messages may only be sent upward through the architecture, while notification messages may only be sent downward. An example of this situation is shown in Figure 8. The depicted system consists of four components (database server, window system, and two graphical user interfaces) and of two connectors (the dark horizontal balks). When one of the user interfaces is used to request a modification, a request message is sent upward to the connector, and then to the database. When the database performs an operation, a notification message is sent to the connector and further to the GUI components.

## 5.4 Architectural description with UML

UML (Unified Modeling Language) can be used to describe an architecture [HNS99, MR99, RMRR98]. Although UML is meant to model the design of object-oriented systems, it can be used more widely because it supports describing various elements and relations between them. Thus, it is applicape to describe software architectures.

Actually, both ADLs and UML has advantages and disadvantages in describing architectures [MR99, RMRR98]. Architectural description languages are special-purpose notations having a great deal of expressive power. However, they are not well integrated with common development methods. These more widely used methods like UML, in turn, are accessible to developers, but they lack the semantics needed for extensive analysis.

This subsection first introduces UML and after that shows how to apply UML to architectural description.

### 5.4.1 Introducing UML

A UML model of a software system consists of the following partial models [MR99, RMRR98]:

- classes and their declared attributes, operations, and relationships,
- the possible states and behavior of individual classes,
- packages of classes and their dependencies,
- example scenarios of system usage including kinds of users and relationships between user tasks,
- the behavior of the overall system in the context of a usage scenario,
- examples of object instances with actual attributes and relationships in the context of a scenario,
- examples of the actual behavior of interacting instances in the context of a scenario,
- the deployment and communication of software components on distributed hosts.

UML is an extensible language that allows adding new constructs without changing the existing syntax or semantics of the language. There are three mechanisms to allow such extension [MR99, RMRR98]:

- constraints place semantic restrictions on particular design elements,
- tagged values allow new attributes to be added to particular elements of the model,
- stereotypes allow groups of constraints and tagged values to be given descriptive names and applied to other model elements.

### 5.4.2 Applying UML

To apply UML to describe an architecture, the architecture can be divided into four views [HNS99]:

- conceptual,
- module,
- execution,
- code.

Note that the above division has much in common with Kruchten's 4 + 1 division [Kru95] presented in Subsection 5.1.

The conceptual architecture view describes the architecture in terms of domain elements. Such elements are components with ports enabling interactions, and connectors with roles defining the relationships between the connectors and ports. The components and connectors are associated with each other to form a configuration. To join also ports and roles in that configuration, their protocols must be compatible with each other. Components can be decomposed into other components and connectors. These elements, with their associated behavior and relationships are shown in the upper part of Table 1. Conceptual architecture view uses the following UML features:

- UML class diagrams for showing the static configuration,
- UML sequence diagrams or state diagrams for showing the protocols connected to ports,
- UML sequence diagrams for showing a particular sequence of interactions among a group of components.

The module architecture view describes the decomposition of the software and its organization into layers. Subsystems are decomposed into modules, and modules are related to layers according to their use-dependencies (see the second part of Table 1.) There is no configuration for the module view because it only shows the relationships among modules but not the combination of modules in a particular product. The module architecture view uses the following features:

- tables for describing the mapping between the conceptual and module views,
- UML package diagrams for showing subsystem decomposition dependencies,
- UML class diagrams for showing use-dependencies between modules,
- UML package diagrams for showing use-dependencies among layers and the assignment of modules to layers.

The execution architecture view is the run-time view of the system. It shows how modules are combined into a particular product by mapping modules to run-time images. The execution view also defines communication among modules and assigns modules to physical resources. The run-time images and communication paths are associated to each other to form a configuration (see the third part of Table 1). The execution architecture view uses the following UML features:

- UML class diagrams for showing the static configuration,
- UML sequence diagrams for showing the dynamic behavior of a configuration, or the transition between configurations,
- UML state diagrams or sequence diagrams for showing the protocol of a communication path.

The code architecture view contains files and directories. It maps the modules and interfaces of the module view to source files, and the run-time images of the execution view to executable files. The code view does not have a configuration. It defines relationships to be applied across all products, not just to a particular product. The elements and their relations are shown in the last part of Table 1. The code architecture view uses the following features:

- tables to describe the mapping between elements of the module and execution views and elements of the code view,
- UML component diagrams for showing the dependencies among source, intermediate and executable files.

| View | Elements | Behavior | Relations |
|---|---|---|---|
| Conceptual | •Component<br>•Port<br>•Connector<br>•Role | •Component functionality<br>•Port protocol<br>•Connector protocol | •Component decomposition<br>•Port-role binding (for configuration) |
| Module | •Module<br>•Subsystem<br>•Layer | •Interface protocol | •Module implements<br>•Conceptual component<br>•Subsystem decomposition<br>•Module use-dependency |
| Execution | •Run-time image<br>•Communication path | •Communication protocol | •Run-time image contains<br>•Module<br>•Binding (for configuration) |
| Code | •Source<br>•Intermediate<br>•Executable<br>•Directory | | •Source implements module<br>•Source includes source<br>•Intermediate compiled from run-time image<br>•Executable implements run-time image<br>•Executable linked from intermediate |

Table 1: Elements of different architecture views [HNS99]

There are also other ways to describe architectures with UML. Garlan and Kompanek propose several alternative ways to describe each architectural concept in UML [GK00]. They discuss the advantages and disadvantages of each way. Medvidovic et al. shows how UML can be used to describe C2-style architectures [MR99, RMRR98]. They apply the built-in extension mechanism of UML to map architectural models expressed in C2-style into object notations.

## 5.5 Variation modeling

An important aspect in product-line architectures is variation among products. However, variation is difficult to model in architectural descriptions. Although many models provide means to describe hierarchical systems (ways to decompose systems into smaller subsystems), they do not always support the description of variation.

The model introduced by van den Hamer et al. distinguishes abstract components and component variants from each other [vdHvdLStS98]. An abstract component can be one of several components. For example, transmission of a car can be either manual or automatic. Thus, an abstract component can be considered as a place-holder for one or more implementations (component variants) of the same basic idea or design. A particular component variant can have a known internal structure. Another variant for the same component can have a different structure consisting of either same or different (abstract) components.

The model of van den Hamer et al. is recursive. A component variant can itself be a family. This family may have a fixed structure defined with abstract components. However, it achieves its diversity by supporting alternative variants for the lower level components.

Describing variation is also possible in Koala language [vO98] (introduced in subsection 5.3.1). In Koala, variation is divided into internal diversity (within components) and structural diversity (between components). To enable diversity, components and configurations are separated from each other. Flexible mechanisms are needed to instantiate components and bind them into configurations.

Internal diversity is implemented via parametrization. In Koala, diversity parameters are declared as functions in requires-interfaces. Thus, their implementation lies outside of the component. Such requires-interfaces that contain diversity functions are called diversity interfaces. However, they can be used like normal requires-interfaces.

Structural diversity is implemented with switches. A switch defines connections between interfaces. The top side of a switch must be connected to the tip of a triangle describing an interface, and its bottom side must be connected to the triangle base of a different interface. Switches are needed, for example, in following situations. Suppose component `A` uses component `B1` in one product and `B2` in another. It would be possible to define two configurations to handle the implementation. However, `A` may be a part of a complex combound component, and it is not desired to duplicate the rest of it. Thus, a switch can be used between the requires-interface of `A` and the provides-interfaces of `B1` and `B2`.

In addition, Koala provides optional interfaces to handle diversity. A new version of a component may differ from the existing ones such that it has different interfaces. Instead of introducing a new component with a new unique name, it is more reasonable to allow adding new (optional) interfaces into existing configurations.

# 6 Analysis and assessment

In architecture assessment or evaluation, the architecture is analyzed against its quality requirements. Architecture analysis can also reveal potential risks concerning the architecture. Architecture assessment is based on the quality attributes that describe the value of the architecture.

Architecture evaluation should be performed already at an early stage of the development of a product-line architecture [ABC+97]. Problems found early i.e. from requirements, specification or design are easy to correct. Software quality cannot be appended afterwards, instead, it must be inherent from the beginning and built-in by design. Thus, as soon as there are artifacts to be evaluated, it should be checked how well the system will meet its requirements.

## 6.1 Quality attributes

The purpose of architecture analysis is to evaluate how well an architecture satisfies its requirements. This evaluation can be made against the quality attributes set for the architecture. The following classification of quality attributes is presented in [BCK98, pp. 79–]:

- quality attributes measurable at run-time (such as performance, security, availability, functionality, and usability),
- quality attributes not measurable at run-time (such as modifiability, portability, reusability, integrability, and testability),
- business qualities (such as time-to-market, cost and projected lifetime of the system),
- qualities of the architecture (such as conceptual integrity, buildability, correctness, and completeness).

Architectural assessment concerns software evaluation in the design phase. Several design metrics can be found in the literature [JRvdL00, p. 69]:

- size,
- complexity,
- modularity,
- cohesion,

- coupling.

Quality attribute specifications may contain profiles [Bos00, pp. 82–]. A profile is a set of scenarios, generally with some relative importance associated with each scenario. For example, usage profile is a set of usage scenarios that describe typical uses for the system. The usage profile can be used as a basis to specify some quality requirements such as performance and reliability. For other quality attributes, other profiles are used. For example, hazard scenarios can be used for safety requirements, while change scenarios are suitable for assessing maintainability.

## 6.2 Architectural assessment

Architectural evaluation methods can be classified in different ways. They can be divided into questioning and measuring techniques [ABC+97, BKW97, DN00, JRvdL00]. Questioning techniques generate qualitative questions to ask about an architecture. They can be applied to evaluate an architecture for any given quality. This category includes scenarios, questionnaires, and checklists. Measuring techniques, in turn, suggest quantitative measurements to be made on an architecture. They are used to address specific software qualities, and thus, they are more restrictly applicable than questioning techniques. Examples of these techniques are metrics, simulations, prototypes, and experiments. These different approaches are considered in the following subsections.

### 6.2.1 Scenario-based assessment

When creating and organizing scenarios, different stakeholders should be taken into account. It is important especially when the assessment team should specify and communicate different stakeholder requirements [CGY99, DWW98, DWW00]. Examples of stakeholders are [BCK98]:

- end user,
- customer,
- marketing specialist,
- system administrator,
- developer,

- maintainer.

As an example of a scenario-based assessment, we consider software architecture analysis method (SAAM) [BCK98, pp. 193–]. It can be used to evaluate a single architecture or to compare several architectures against each other. SAAM consists of the following steps (that are also depicted in Figure 9):

**Develop scenarios**
: A scenario is a brief description of some anticipated or desired use of a system. Scenarios can describe the different uses of the system as well as the potential changes of the system. In addition, this step considers the different users of the system and defines the qualities the system should satisfy. Thus, scenarios represent tasks for different stakeholders.

**Describe candidate architecture**
: In this phase, the candidate architecture is described in some architectural notation. The representation should cover the computation of the system and its components with their connections.

**Classify scenarios**
: Scenarios are classified into direct and indirect ones. Direct scenarios are those that the system supports directly. It means that the anticipated use defined by the scenario requires no modification to the system. This can be concluded from the demonstration showing how the architecture would behave when performing the scenario. Indirect scenarios are those that suggest some changes to the system that could be represented architecturally. These changes are, for example, an addition of a component to provide an activity, a modification of a component to provide an activity in a different way, a new connection between components, etc.

**Perform scenario evaluations**
: This phase produces a list of desired modifications required by the indirect scenarios. In addition, the cost of each modification is estimated.

**Reveal scenario interaction**
: Two or more (indirect) scenarios interact in a component if they suggest modifications to the same component. Scenario interaction is important because it may reveal several meanings in a single component. It indicate the points where the designer should pay accurate attention. Scenario interaction is related to such metrics as structural complexity, coupling, and cohesion. High interaction between different components corresponds to low cohesion and high structural complexity, while high interaction between similar components signals high cohesion.

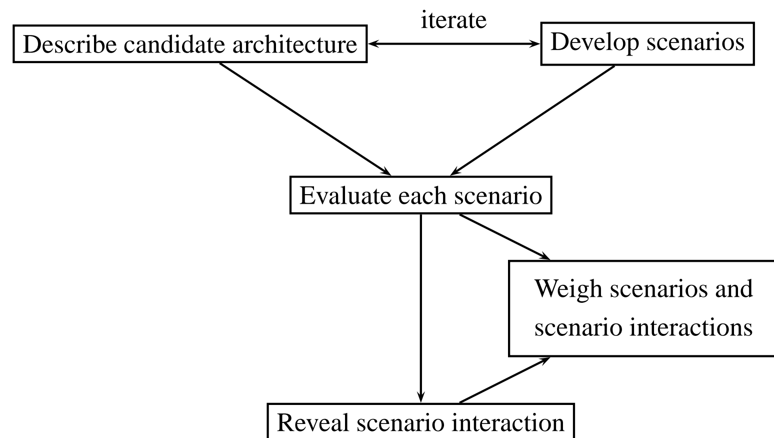


Figure 9: Overview of the SAAM method [DWW00]

**Overall evaluation**

If the purpose is to compare several architectures, a weight should be assigned to each scenario according to its importance. This action avoids the situation in which the first candidate architecture is best on the half of the scenarios and the second one wins on the other half of the scenarios.

### 6.2.2 Questionnaire-based assessment

A questionnaire consists of a list of general questions applicable to all architectures [ABC+97]. The questions concers, for example, the way the architecture is generated and documented or details of the architecture description itself. There is a team to review the answers and to focus some relevant aspects with additional questions. Example questions are:

- Is there a designated project architect?
- Is a standard description language used?
- Are all user interface aspects separated from functional aspects?

### 6.2.3 Checklist-based assessment

Checklists are used to help keeping a balanced focus on all areas of the system [ABC+97]. A checklist contains a detailed set of questions developed after ac-

quiring much experience in evaluting several systems. Checklists typically concentrate more on particular qualities of the system than questionnaires. Example checklist questions are:

- Does the system write the same data multiple times to disk?
- Have you considered handling of peak as well as average loads?

### 6.2.4 Simulation-based assessment

Simulation-based assessment is a dynamic method simulating the context in which the software system is supposed to execute. It consists of the following steps [Bos00]:

**Define and implement the context**
This phase identifies the interface of the architecture to its context. The purpose is to find out the way to simulate the behavior of the context at the interfaces. In addition, the appropriate level of abstraction is chosen. The balance between cost and benefit should be found to be able to implement only those things that are really needed.

**Implement the architectural components**
The interfaces and connections of components needed in this phase can be found from the architectural design descriptions. The accuracy of the implementation depends on the quality attributes that the software architect is interested in.

**Implement profile**
Again, the implementation of the profile depends on the quality attributes under consideration. However, it should be possible to activate individual scenarios and run a complete profile.

**Simulate system and initiate profile**
So far, the complete simulation including context, architecture and profile(s) is ready for use. Thus, the system can be simulated and the results are collected. For several quality attributes, simulation actually runs two profiles. For example, when assessing safety, the system runs its usage profile automatically and, in addition, the hazard scenarios are activated. Each time a hazard scenario occurs, data about the the system context is collected.

**Predict quality attribute**
Finally the collected information is analyzed and the assessed quality attributes based on that information are predicted. The amount of the gath-

ered information may be large. Thus, automated support to process this information may be needed.

### 6.2.5 Metrics-based assessment

Assessing an architecture can be based on mathematical models such as McCabe's cyclomatic complexity metric [McC76]. Mathematical models allow static evaluation of the architectural design model, and they are suitable for assessing operational quality attributes. This kind of assessment consists of the following steps [Bos00]:

**Select and abstract a mathematical model**
There are well-established mathematical models for the assessment of quality attributes. However, these models are usually too detailed to be used at the architectural level. Thus, the selected models need to be abstracted.

**Represent the architecture in terms of the model**
The selected model do not necessarily assume that the system consists of components and connections. Instead, it may assume the system to be represented by using tasks. Thus, the architecture must be adapted to the model.

**Estimate the required input data**
The mathematical model may require such input data that is not included in the architecture definition. This data has to be estimated and concluded from the requirement specification and the architecture design.

**Predict the quality attribute**
After the earlier phases, the software architect is now able to calculate the resulting prediction for the assessed quality attribute.

As a summary of the assessment techniques, Table 2 shows the properties of the evaluation approaches [ABC+97].

| **technique** | **generality** | **detail level** | **phase** | **target** |
|---|---|---|---|---|
| questionnaire | general | coarse | early | artifact, process |
| checklist | domain-specific | varies | middle | artifact, process |
| scenarios | system-specific | medium | middle | artifact |
| metrics | general or domain-specific | fine | middle | artifact |
| prototype, simulation, experiment | domain-specific | varies | early | artifact |

Table 2: Properties of the assessment techniques [ABC+97]

# 7 Development and evolution

Architecture development covers modifying activities made in the design phase of a product-line architecture. Architecture evolution, in turn, addresses the modifications made after using the products of the product line.

Architectural development and evolution are associated with design decisions and variation. Considering the extensibility and variation management of the architecture already at its design phase makes the future evolution easier to handle.

## 7.1 Architecture development

Essential in developing products based on a product line is that the product architecture can be derived from the product-line architecture [Bos00, pp. 261–]. This derivation consists of four steps:

- architecture pruning,
- architecture extension,
- conflict resolution,
- architecture assessment.

The product-line architecture contains features that are common for the products of the family. However, the product-line architecture should support all the members of the family. Thus, it may contain also those features that are common for a (large) subset of the product family. Consequently, when developing a product architecture from a product-line one, it may be necessary to prune those features that are not necessary for the particular product. The product-line architecture may also provide variant parts from which a product usually exploits one. Thus, the other variants should be excluded from the product architecture.

The pruned architecture does not always fulfill the requirements of the product. Instead, the product is expected to implement those (product-specific) parts that are not included in the common architecture. Thus, the product architecture should be extended to cover the product-specific requirements, too. These requirements can be either functional or quality requirements. The former ones can be implemented as pure component extensions such as product-specific extensions to product-line components or complete product-specific components. Product-specific quality requirements are more difficult to manage because they may require reorganization of the whole architecture. This can be due to such

products that are at the boundary of what the product line should provide without violating the requirements of the core family members.

The product-line architecture and the product architecture have a close relationship but different aims. Thus, there may occur conflicts in the derivation. First, the functionality of a product-line component may conflict with the product requirements. The functionality can be embedded in the behavior of the common component, and it cannot be separated without wide modification of the component. Second, a product-line component does not always provide the functionality required in the product. The insertion of the missing functionality can be difficult if the different parts of the functionality should be added in several points. Third, the quality attributes of the architectures can be conflicting. For example, the flexibility attribute of the product line can be conflicting with the real-time and reliability requirements of the product. Fourth, a quality attribute that is needed in the product architecture is not necessarily provided in the product line. This may require architecture transformations. Each of these situations requires consideration whether to transform the product-line architecture or the product architecture. Transformation of the product-line architecture is usually undesired but in some situations it cannot be avoided.

During the three earlier steps, several modifications to the product-line architecture have been made. These improvements may have had effects on the other requirements of the architecture. Thus, it is reasonable to evaluate the architecture to make sure that its requirements are still fulfilled.

## 7.2 Architecture evolution

A product-line architecture joins the products of the family together, and thus, limits their evolution. The architectures of the family support the development and modification that they are prepared for. Some potential changes are easy to anticipate before-hand. In addition, domains are different: some are more exposed to changes than others. The common architecture must change and evolve, when it can no longer support the members of the family [Kuu99].

This subsection first classifies the modifications and then shows how to manage the modifications needed in architectural evolution.

### 7.2.1 Classifying the modifications

After using a product belonging to a product family, the customers may require new functionality or other improvements for the product. These new requirements lead to considerations whether they should be applied to a single product or the whole product line. If the requirements cover the whole product line, it must evolve. The required changes can be classified as follows [Bos00, BR00, SB99a, SB99b]:

- new product line,
- introduction of new product,
- adding new features,
- extend standards support,
- new version of infrastructure,
- improvent of quality attribute.

Introduction of a new product line based on an existing one can have several reasons. First, the variability between the products of the existing product line can become too large. This may be, for example, due to recurring introduction of new products to the same product line. Second, it is sometimes necessary for business reasons to provide for an major customer an own product line based on the existing one. Third, it can be necessary to incorporate independent products into the product line, for example, after merging or buying another company in the same market segment. Such products do not share the same architecture and they cannot be converted at once. Thus, they may require a new product line. The fourth reason for a new product line is the geographical distance between the product line and a subset of the products in that product line. Fifth, cultural conflicts between different organizational units can lead to separate product lines. A new product line can be introduced either via cloning or via specialization.

Another type of evolution is the introduction of a new product. This can be the case when there occurs need for a new product in the market. In addition, it is sometimes reasonable to extend the scope of the existing product line by adding a new product at the high-end or at the low-end of the product line. When introducing a new product, the commonalities between the product and the product line should be identified. In addition, the architecture of the new product should be matched with that of the product line. The similar parts of the architectures suggest replacing product-specific components with product-line components. The

variant parts of the new product should be considered as well. It may be necessary to extend the components to support the new product. In addition, product-specific code can be either developed in the case of adding a totally new product or reengineered in the case of integrating an existing product to the product line. Finally, the new product should be instantiated based on the product-line assets and verified against the requirements of the product.

Adding new features is the most common evolution type. The pressure to add new features can come from customers or from competitors. Alternatively, new technological opportunities can lead to develop a product with new features. The new features do not necessarily cover all the products of the family. This may lead to problems, because the new features can have negative effects on those products that do not need the new feature. One possible solution is to create a hierarchical product line.

The fourth type of evolution is the extension of standard support by the products of the product line. Examples of these standards are network communication protocols, component communication standards (JavaBeans, CORBA) and file systems. The first release do not necessarily support a standard completely, only a subset of the standard. Then the subsequent releases extend the support gradually until the whole support is achieved. This kind of evolution is often concerned with changing an existing framework implementation to add functionality.

The new version of the infrastructure (hardware, operating system) usually provides new functionality. The similar functionality must have earlier been provided by the application (the product line). It is not reasonable to implement the same functionality on two levels. Thus, the product line should be modified to use the functionality available in the lower layers.

The last kind of evolution concerns with the quality attributes of the product-line members and of the assets in the product line. Quality requirements usually have effects on the architecture of the product. For example, more strict performance requirements lead to modifications of component implementations.

### 7.2.2 Managing the modifications

There are different ways to manage the modifications during architecture evolution. Knowledge concerning the evolution of a system can be stored in a design decision tree (DDT) [KK98, RK96]. The order in which design decisions are made is significant. Design decisions are restricted by the constraints of the ear-

lier decisions. In addition, the newer decisions are more fine-grained and more specialized than the older ones. Thus, design decisions can be represented as a tree-like structure. However, a DDT is a directed graph. The nodes are design decisions consisting of the actual decision, the context or requirements of the decision, and the implications of the decision.

Design decision trees support tracing design decisions to requirements. In addition, they enable considering the alternative solutions. It is easy to find the first decision to be changed when a requirement has changed. Thus, it is possible to estimate the cost of the modification.

As another way to manage the modifications, Scherlis proposes four ideas to handle evolutionary changes in the product line [Sch98]:

- systematic techniques to carry out structural change,
- maintenance of design records to avoid information loss,
- mixing informal and formal steps in structural evolution,
- simultaneous multiple views.

There are several techniques to made structural changes in software. Such are, for example, class hierarchy reorganization, systematic representation change, and specialization of types and classes. When restructuring code, the relationship between code and the actual architecture changes. Thus, it is important to keep records through the development and evolution process.

Information loss can be avoided by maintaining design records. Structural changes can be made by using formal techniques. Such tools can also be exploited in gathering information and in finding and creating the links between code fragments and their context in design documents. Structural manipulations have also effects on reuse. Manipulation tools make the component interfaces more clear and support component specialization, and thus, improve the reuse of these components.

It can be necessary to use both informal and formal methods in structural evolution. Usually, when mechanical techniques cannot be used, manual methods must be used in structural changes. In this case, design record maintenance can be more difficult. Thus, formal binding of annotations to code should be maintained even when the annotations themselves are informal and textual.

It is important to provide multiple views of a system. Creating a new structural view involves analysis and manipulation effort. The new view provides an alternative perspective on the system. It shows the dependencies between abstractions and elements belonging to a particular evolutionary step. However, computationally significant modifications can make existing views invalid.

# 8 Recovery and reengineering

The purpose of architectural recovery is to study legacy software and to find out the common architecture of the systems in order to reengineer them to constitute a product family. Architectural recovery is very close to reverse engineering and design recovery. Each of them gathers information about an existing system and tries to get a conception of the system, its components and the relationships between the components. Necessary information can be collected from source code, from different documents and from domain and application experts. However, architectural recovery concentrates especially on deriving architectural information from these available sources of information concerning the system.

When developing a new system, starting from scratch is very expensive. Thus, a successful legacy system can be used as a basis in the development process. The recovery process may reveal several systems sharing the same basic architecture. Collecting these systems under the same product line is important because maintenance and further development of similar but separate systems would be more difficult than the systems belonging to the same program family.

Architectural recovery is related to architecture description because the captured architecture needs to be described. Architectural recovery is also associated with domain engineering because commonality analysis (a part of domain engineering) can be used in architectural recovery and reengineering. Commonality analysis is needed in comparing the architectural aspects of legacy systems and in deciding whether these systems should belong to the same program family or not.

## 8.1 Reengineering steps

Reengineering typically consists of three steps: analysis of an existing system, logical transformation, and development of a new system (see Figure 10). The first step goes up the left side of the horseshoe, the second goes across the top of the horseshoe, and the third goes down the right side of the horseshoe. However, reengineering can be applied in different levels. It is not always necessary to raise up to the architectural level. Instead, reengineering can be concerned with source code (the lowest level) or any other level shown in Figure 10.

The first step is called architecture recovery and conformance. In its complete form, architecture is recovered by extracting artifacts from source code. The recovered architecture is analyzed to find out whether it conforms to its design. It is also evaluated against its quality attributes such as performance, security, or reli-

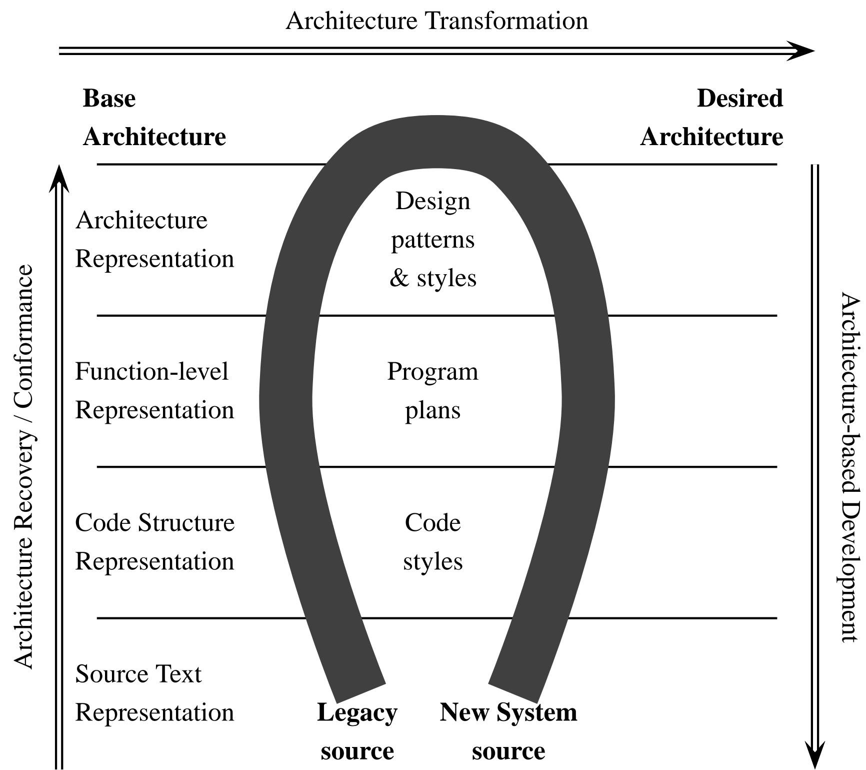


Figure 10: Horseshoe model to integrate reengineering and software architecture views [BSWW99]

ability. (This step is considered in Subsections 8.2 and 8.3.)

The second step is architecture transformation. The recovered architecture is reengineered to a desired form. It is evaluated against the quality requirements of the system and other organizational and economic constraints.

The third step, architecture-based development, instantiates the desired architecture. Packaging issues are decided and interconnection strategies are chosen. Code-level artifacts from the legacy system can be wrapped or rewritten to fit into this new architecture. (This step is considered in Subsection 8.4.)

## 8.2 Recovery steps

Architecture recovery can be divided into two steps [JRvdL00, pp. 99–]:

- recovering and describing the architecture of single systems,
- recovering and describing the program family architecture on the basis of the architecture of several single systems.

### 8.2.1 Recovering single systems

Recovering the software architecture of single systems is useful for the following activities:

**Redocumentation**
: Architecture recovery produces documents about the system needed in further development.

**Reasoning and assessment**
: Architecture recovery gives confidence in the system at a high level, verifies the fulfillment of its requirements and reveals the limitations of its architecture.

**Architecture redesign**
: Architecture recovery aids in understanding the system which is important when changes are needed in the architecture of the system.

**Maintenance**
: Architecture recovery helps get a conception of the system and measure the impact of changes which are needed in maintenance.

**Evolution**
Architecture recovery shows the components of the system and their mutual relationships. This information is needed in implementing new functionality or in other evolution activities.

**Development of a new system or a successor from a single system** Architecture recovery produces information which is needed in developing a new system that meets the requirements of customers and market better than the existing one.

### 8.2.2 Recovering a product-family

Recovering the architecture of a product family is useful for the following activities:

**Development of successors**
Architecture recovery decreases the development cost because the architecture, documentation or even source code can be reused among the program family. In addition, many changes are easier to made because they can be anticipated in the products of the family.

**Maintenance of original systems**
Architecture recovery identifies the common parts of the systems. This improves understanding of the overall system and makes error repair and enhancement easier.

**Development of product families that share requirements**
Architecture recovery supports reuse even when developing another product family with similar requirements such as safety or security.

## 8.3 Recovery approaches

Architectural recovery can be based on the properties of an architecture [BGHE97, BG98, EOGB97]. An architectural property represents a specific design decision related to requirements of the architecture. It can be described using different architectural descriptions. The following list shows examples of architectural properties. Possible techniques to implement each property are given within parentheses:

- information exchange (parameter passing, global or local variables),
- system control (active, passive),

- dynamic behavior (process, concurrency),
- system structure (layered, kernel/shell, pipe/filter),
- safety (redundancy, hardware tests, time-outs, checksums),
- security (encryption, digital signature, traffic padding),
- variance (standards, customer needs, hardware).

The recovery process goes as follows. Architectural properties are first identified (for example, according to the above list). After that, appropriate implementation techniques for each property are considered. Occurencies of these implementations are identified. This identification may exploit different recovery methods such as reverse engineering tools, tools for providing different views about the source code, string matching or code browsing tools.

Recovery process based on architectural properties can apply Hot-Spots technique [GB98]. Hot-Spots are architecture-related elements representing an essential concept of the application domain. Hot-Spots technique consists of several steps. The essential concepts of the domain are first identified. After that, implementations of these concepts (Hot-Spots) are seached from the code. The results of the search are programming-oriented concepts such as files, modules, functions, data structures, etc. In addition, different tools can be exploited to show, for example, the call graph representing related functions and data structures to the identified programming-oriented concepts.

In addition to architectural properties, recovery can be based on architectural structures, i.e. components and connectors [EOGB97, EKO+98]. Components are typically implemented as a set of modules. However, in such languages that do not share object-oriented paradigm or not have the concept of module, component identification can be difficult. In these cases, identification can be based on file structure, variability of software, coupling, design documentation, or scheduling structure. Components can be organized in layers. In such cases, components at different levels can be identified using different techniques. For example, recovering components on the highest level can be based on documentation.

Architectural recovery can apply string or pattern matching [KTW98]. For example, a software architecture for state transitions and state machines is typically implemented using selection statements such as `if`- and `switch`-statements in C and in C++. Code fragments implementing state transitions usually refer to a state variable storing the current state. Whenever an event arrives, the current state

is determined using a selection statement. Then, the event is determined using another selection statement nested in the first selection statement. After that, the appropriate program code is executed in the corresponding branch of the selection statement. Code fragments as described above, obey a particular pattern. Thus, they can be identified using specific pattern or string matching techniques.

## 8.4 Migration

Traditional reengineering is based on deep program understanding and reverse engineering. Understanding plays a critical role in evolving legacy systems into reusable assets to be exploited in a product line approach. However, legacy code can be very difficult to understand for several reasons. Legacy software may have been implemented via ad hoc methods or unstructured programming style. During its life-time, legacy software may have been a target for several maintenance actions applied by several maintainers. Documents are not always up-to-date. There may be little or no conceptual integrity in the architecture and design of a legacy system.

As program understanding is usually a very slow and time-consuming activity, Weiderman et al. proposes moving from the deep understanding of the internals of software modules (white-box reengineering) to the understanding of the interfaces between software modules (black-box reengineering) [WNS+97, WBST98]. Black-box reengineering is preferred due to new technologies such as CORBA, Java, and the Web. The cost of deep understanding is linear relative to the size of the program, but the benefit provided by the new technology cannot be exploited. Instead, the alternative way (interface understanding) combined with the new technology provides economic benefits.

# 9 Reuse

Software architectures consist of components. Components are highly reusable because their interface and functionality are explicitly defined. A product-line architecture has common components shared by all products of the same family. In addition, it provides variant components exploited by a subset of the products. These variant components are essential for reuse, because together they cover a large set of situations in which they can be applied and reused. Thus, an important benefit of product-line architectures is to provide reusable components which can be reused among the products of the same family and even between different product lines.

We consider different situations to reuse software components as well as problems related to reuse. As an example of architectural reuse, we consider an object-oriented framework called FRED (Framework Editor for Java).

## 9.1 Reuse situations

Bosch has identified three levels of component reuse [Bos00, p. 215]:

**System versions**
: Components over subsequent versions of a system can be reused.

**Software product lines**
: Components can be used for subsequent product versions and for a family of products containing similar but not identical functionality.

**Third-party components**
: Components can reused over product versions, a family of products and within different organizations. This is called commercial-off-the-shelf (COTS) technology.

There is no consensus about the meaning of reusable assets. Bosch has noticed that academic perception of reusable assets is rather different from that of industry [Bos99, Bos00] (see Table 3).

## 9.2 Reuse problems

Bosch has identified several problems concerning reusable assets [Bos99]. The problems can be devided into three categories:

| Research | Industry |
| --- | --- |
| • Reusable assets are black-box components. | • Assets are large pieces of software with a complex internal structure and no enforced encapsulation boundary. |
| • Assets have narrow interface through a single access point. | • The asset interface is provided through entities such as classes in the asset. These interface entities have no explicit differences to non-interface entities. |
| • Assets have few and explicitly defined variation points that are configured during instantiation. | • Variation is implemented through configuration and specialization or replacement of entities in the asset. Sometimes multiple implementations (versions) of assets are used for variation. |
| • Assets implement standardized interfaces and can be traded on component markets. | • Assets are primarily developed internally. Externally developed assets go through considerable (source code) adaptation to match the product-line architecture requirements. |
| • Focus is on asset functionality and on the formal verification of functionality. | • Functionality and quality attributes such as performance, reliability, code size, reusability, and maintainability have equal importance. |

Table 3: Academic versus industrial conception on reusable assets [Bos99]

- multiple versions of assets,
- dependencies between assets,
- assets in new contexts.

Multiple versions of assets can be due to conflicting requirements. Despite the same functionality, different products in the product line may have different quality requirements. For example, performance or code size can have a very high priority. Some other products, instead, should fulfill other quality requirements that are conflicting with those aforementioned. Thus, the reuse of the assets is restricted to those ones that share the same quality attributes. The second reason for multiple asset versions is caused by variability. Variability can be implemented, for example, through configuration or compiler switches. However, this can be complicated if variation covers the whole asset. In these cases, variation is often implemented via different versions of the product. Third, multiple versions of assets are needed because of the differences among the products in the product line. Some products may need only a restricted subset of the functionality provided by the product line. The unused part of the functionality brings no benefit to these products, instead, it unnecessarily increases the code size and makes the interfaces more complex. For example, in embedded systems these factors can be so critical that it is reasonable to provide also a lightened version.

The reusable assets are part of the same product-line architecture. Thus, they have dependencies between each other. The initial design of the assets typically introduces only necessary dependencies. The evolution of the assets has the tendency to bring new dependencies that are not always necessary. For example, during evolution the size of the reusable assets often increases. Thus, it can become actual to split the reusable component into two components. These two components have a lot of connections to each other. Even after redesign, several connections remain because of the common origin of the components. As another example, development of the product-line architecture may extend its functionality. This new functionality usually covers more than one asset, and thus, creates dependencies among the involved assets. The third example concerning dependencies between assets is related to extension of assets. Evolution of an asset component may require new connections to other components. Avoiding these connections could have been possible in the initial design with a different modularization, but afterwards they are difficult to handle.

To maximize reuse, assets should be used in as many products and domains as possible. However, the new context usually requires some changes to the asset. This makes reuse of the assets more complicated. The assets are usually difficult

to be applied in new domains. Although it is easy to accomodate to anticipated variabity, is not so simple to manage unexpected variability.

## 9.3 Reuse supported by frameworks

Frameworks can be used in implementing product-line architectures. A framework provides the common reusable part of an architecture. In addition, it defines the ways how to specialize the architecture to support also the variant parts. Design patterns [GHJV95] are closely related to frameworks because they provide natural specialization points for frameworks.

FRED (Framework Editor for Java) [HHK$^+$01] is an object-oriented framework for architecture-oriented Java programming. Instead of design patterns, FRED introduces a programming pattern consisting of a collection of roles for various language structures such as classes, methods, attributes, etc. and a set of constraints on the roles. Each role corresponds to a base class defining the usage and extension way of that class. Extending a class can be considered as binding a user-given role for a pattern under the role-specific constraints.

FRED provides a task list based on pattern definitions to guide the user during the program development process and to remind her about the things still uncompleted. The task list also checks that the used patterns are bound to the context in the required manner. The task list contains uncompleted tasks such as unbound roles of a pattern and broken constraints of existing bindings. Tasks can have orderings among each other. For example, a task for defining a subclass must naturally be carried out before another task for defining an overriding method within that subclass. Carrying out a task may lead to succeeding tasks added to the task list. The task is completed, if the user assigns a suitable program element to the role.

As mentioned, FRED uses patterns in framework specialization to express reusable structures. Roles are considered as abstract representations of recurring program elements. Thus, they corresponds to the common base of a product line. Like the constraints in FRED, also the constraints of a product line define the common features of the products and defines the way how the products can vary.

# 10 Testing

Testing is needed in software development process to identify faults and to be able to correct them. Another purpose of testing is to determine whether a software system fulfills its requirements. Product-line architectures need testing to make sure about the correctness of both the product line and the individual products of the family.

Architecture testing can be based on formal description of the architecture. In this case, test criteria can be defined on the basis of the formal description and test plans can be derived from this description.

## 10.1 Testing in general

Testing consists of different kinds of testing tasks. Each task can be divided into the following activities [Nor01]:

**Analysis**
identifies appropriate test cases from the material to be tested. Analysis may reveal some defects such as poor testability. The output of the activity is a detailed test plan.

**Construction**
builds the artifacts needed in performing the test cases identified in the earlier activity. Examples of such artifacts are test drivers, test data sets, and software implementing the actual tests.

**Execution and evaluation**
perform testing and analyze its results. Depending on the results, next process steps are decided.

It is not realistic to test software with all possible inputs. Thus, testing can be considered as searching for those inputs that most likely lead to a failure. However, the following aims can be laid to testing [Nor01]:

- testing is objective such that testing criteria are selected under guidance of the satisfaction of the requirements,
- testing is systematic such that criteria are selected according to an algorithm describing the reason to select each criterion,
- testing is thorough such that criteria could be argued as being complete by some definition,

- testing is integral to the development process such that during developing a software system, plans are made tell how to assess the system.

Testing has many forms to be carried out in different phases of software development. Such are [Nor01]:

**Analysis and design model validation**
: Each phase in software development process may include creation of different models. The syntax of the models should be verified, and the models should be validated against the required system. Test cases are constructed from the requirements and constraints.

**Unit testing**
: A unit to be tested can be a function, class, or component. These tests are carried out during coding. The tests should ensure that the unit does everything claimed in its specification and does nothing that it should not. There are two strategies to select the test inputs for a test case. The functional testing strategy selects test inputs according to the specification of the unit. The structural testing strategy, in turn, makes the selection according to the structure of the code that implements the functionality of the unit.

**Subsystem integration testing**
: Subsystems consist of units. The units have been tested before integrating them into a subsystem. However, there may occur failures when using units in connection with each other. Thus, the subsystem integration test plan should describe tests that cover communication between pairs of components.

**System integration testing**
: The result of composing subsystems together is the whole system or product. System integration testing ensures that a product does what it is supposed to do. These representative tests cover the complete specification about the functionality of the system. Tests can be based on frequency of use or on the criticalility of the function.

**Regression testing**
: Regression testing is critical in environments emphasizing reuse, for example, in product-line architectures. Regression tests determine periodically whether the components remain correct and consistent over time.

**Conformance testing**
: Conformance testing determines whether a component can be used in a specific role in an application. Thus, conformance testing focuses on an initial validation.

**Acceptance testing**

To validate the claims of the provider, the consumer performs acceptance testing. The application being tested is introduced into the consumer's actual environment.

**Deployment testing**

Deployment testing can be considered both as an extension and as a precursor to acceptance testing. Deployment testing is performed before releasing the software to customers for acceptance testing. It covers all the unique system configurations upon which the is to be deployed. Deployment testing focuses on the interaction between the product and platform-specific libraries, device drivers, and operating systems.

## 10.2 Testing in product lines

Testing cornerning product lines covers the core asset software, the product-specific software and interactions between them. In addition, testing may be distributed across different parts of the organization. Testing in product lines also requires efforts and produces artifacts that can be reused across the various products of the product line. Thus, this reuse aspect should be considered in test plans.

Concerning testing in product lines, the testing software should be structured such that it supports reuse. Consequently, the test code should be traceable to the code that it tests. When correcting errors, the test code is executed several times during development. This requires changes to the product code, and corresponding changes may be needed to the test code. Due to the demand of traceability, the test software should reflect the product-line architecture. For example, when two parts of the product-line architecture have a particular relationship, the test code for those parts should also be related. This reduces maintenance costs of the test software because it is easier to identify points requiring modification.

Reuse of test assets can be supported such that they have the same granularity as the production assets to which they correspond. This makes it easy to reuse the test assets in testing individual products of the same family.

The product-line architecture can provide support for testing. This includes special test interfaces allowing a self-test functionality. The basic support for self-testing can be defined in the product-line architecture and applied by specific products according to their individual features.

### 10.2.1 Testing in core asset development

As mentioned, testing produces reusable core assets. There are three categories of such assets [Nor01]:

- documents such as test plans and test reports,
- test data sets,
- test software.

The test documents can be organized hierarchically according to the relationships among the products in the product line. A skeletal test plan supports reuse by reflecting to the commonalities among the products. This also enables common features to be tested in the same way for every product.

Software core assets are usually components. In addition to testing a component against its specification, the behavior of the component in integrated situations should also be examined.

In product-line architectures, different products need different components. Thus, components can be variants for each other. The test plan for individual components is divided into functional and structural test suites. Functional tests can be used for all variant components. Structural tests, in turn, must be modified for each different variation.

### 10.2.2 Testing in product development

In product-line architectures, products are generated from core assets. This makes integration testing an important activity. This kind of testing focuses on the interactions between components. Defining the interfaces of the components carefully makes these tests simpler.

In addition, functional tests are needed for products. They ensure that the product behaves as it is supposed to behave. The functionality of the product may have been tested at product-line level. However, a product usually has some product-specific features and they have to be tested in the product level. However, the tests used at the product level can be derived from the tests or test templates created at the product-line level.

Note that the test cases used at the product level are the same whether or not the product is defined within a product line. However, the effort required to be ready

to test is different. The effort in the case of product-line software can be smaller because at least some test analysis artifacts can be reused from the product-line level.

## 10.3 Architecture testing based on formal descriptions

Architecture testing is closely related to architectural description because test plans can be derived from descriptions. (Section 5 considers these different ways to describe an architecture.) For example, when using UML (Unified Modeling Language) to describe an architecture, UML statecharts can be used as a basis to generate test cases [Abd00, OA99]. There are also other languages that can be associated to testing. For example, Standard ML provides parametric polymorphism and type inference. These features correspond to generic properties of architectures. In testing, parametric polymorphism can be exploited to reduce necessary test cases because testing of polymorphic values need not very large test set [AC98].

In addition to architecture description languages, there are other formal methods to describe an architecture. For example, CHAM (Chemical Abstract Machine) provides means to formal description that is associated with testing [PW92, IW95]. CHAM is based on a framework for architectural description that defines an architecture specification to consist of elements (processing, data and connecting elements) and form (relationships among the elements) [PW92]. CHAM views a software system as chemical solutions with their reactions. Molecules are the basic elements of a CHAM. Solutions are multisets of molecules corresponding to the states of a CHAM. Chemical solutions can evolve according to transformation (reaction) rules. This corresponds to the state changes of the machine.

Applying CHAM in testing purposes is based on its division of architectures into data elements, processing elements, and connecting elements [BI98, BCIM00, RW96]. Testing an architecture requires studying these elements and their relationships. CHAM defines all these aspects that should be studied during testing at the architectural level. Thus, it can be used as a basis for architecture testing and especially for architecture integration testing.

Architectural testing can be applied in different testing phases. It can concentrate on regression testing [Har98] or on conformance testing [You98]. Most often, architectural testing is associated with integration testing (for example, CHAM formalism). Architecture is composed of components or even COTS

(commercial-off-the-shelf) components. In these cases, it is important to validate that the components integrate correctly [LR98].

In addition, there are different methods for architectural testing. For example, chaining resembles program slicing [HRB90, Wei84] such that it reduces the parts of an architecture needed to be tested [SRW97].

# References


[ABC+97] Gregory Abowd, Len Bass, Paul Clements, Rick Kazman, Linda Northrop, and Amy Zaremski. Recommended best industrial practice for software architecture evaluation. Technical Report CMU/SEI-96-TR-025, Software Engineering Institute, Carnegie-Mellon University, 1997.

[Abd00] Aynur Abdurazik. Suitability of the UML as an architecture description language with applications to testing. Technical Report ISE-TR-00-01, Information and Software Engineering, George Mason University, Virginia, February 2000.

[AC98] Stuart O. Anderson and Daniele Compare. Type systems, software architecture and testing. In *International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, June–July 1998.

[ADD+00] Mark Ardis, Peter Dudak, Liz Dor, Wen-jenq Leu, Lloyd Nakatani, Bob Olsen, and Paul Pontrelli. Domain engineered configuration control. In Patric Donohoe, editor, *Software Product Lines, Experience and Research Directions*, pages 479–493. Kluwer Academic Publishers, 2000.

[ADH+00] Mark Ardis, Nigel Daley, Daniel Hoffman, Harvey Siy, and David Weiss. Software product lines: a case study. *Software — Practice and Experience*, 30(7):825–847, 2000.

[BCC+99] John Bergey, Grady Campbell, Paul Clements, Sholom Cohen, Lawrence Jones, Robert Krut, Linda Northrop, and Dennis Smith. Second DoD product line practice workshop report. Technical Report CMU/SEI-99-TR-015, Software Engineering Institute, Carnegie-Mellon University, 1999.

[BCIM00] A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Deriving test plans from architectural descriptions. In *22nd International Conference on Software Engineering (ICSE'2000)*, pages 220–229, June 2000.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BCS00] Don Batory, Rich Cardone, and Yannis Smaragdakis. Object-oriented frameworks and product lines. In Patric Donohoe, editor, *Software Product Lines, Experience and Research Directions*, pages 227–247. Kluwer Academic Publishers, 2000.

[BFG+00] John Bergey, Matt Fisher, Brian Gallagher, Lawrence Jones, and Linda Northrop. Basic concepts of product line practice for the DoD. Technical Note CMU/SEI-2000-TN-001, Software Engineering Institute, Carnegie-Mellon University, 2000.

[BG97] Don Batory and Bart J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, 1997.

[BG98] Berndt Bellay and Harald Gall. Reverse engineering to recover and describe a system's architecture. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 115–122. Springer, 1998.

[BGHE97] Berndt Bellay, Harald Gall, Vesna Hasler, and Wolfgang Eixelsberger. Architecture recovery based on architectural concepts. Technical Report TUV-1841-97-13, Distributed Systems Group, Information Systems Institute, Technical University of Vienna, September 1997.

[BI98] Antonia Bertolino and Paola Inverardi. Reaction graps for the testing and analysis of software architectures. In *International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, June–July 1998.

[BKW97] Mario R. Barbacci, Mark H. Klein, and Charles B. Weinstock. Principles for evaluating the quality attributes of a software architecture. Technical Report CMU/SEI-96-TR-036, Software Engineering Institute, Carnegie-Mellon University, 1997.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components.

*ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[Bos99] Jan Bosch. Evolution and composition of reusable assets in product-line architectures: A case study. In *First Working IFIP Conference on Software Architecture*, February 1999.

[Bos00] Jan Bosch. *Design and Use of Software Architectures: Adopting and evolving a product-line approach*. Addison-Wesley, 2000.

[BR00] Jan Bosch and Alexander Ran. Evolution of software product families. In Frank van der Linden, editor, *Software Architectures for Product Families, International Workshop IW-SAPF-3*, volume 1951 of *Lecture Notes in Computer Science*, pages 168–183. Springer, 2000.

[BSC99] Don Batory, Yannis Smaragdakis, and Lou Coglianese. Architectural styles as adaptors. In Patric Donohoe, editor, *Software Architecture*, pages 203–224. Kluwer Academic Publishers, 1999.

[BSWW99] John Bergey, Dennis Smith, Nelson Weiderman, and Steven Woods. Options analysis for reengineering (OAR): Issues and conceptual approach. Technical Note CMU/SEI-99-TN-014, Software Engineering Institute, Carnegie-Mellon University, September 1999.

[CAJ98] Yu Chye Cheong, Akkihebbal L. Ananda, and Stan Jarzabek. Handling variant requirements in software architectures for product families. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 188–196. Springer, 1998.

[CGY99] L. Chung, D. Gross, and E. Yu. Architectural design to meet stakeholder requirements. In Patric Donohoe, editor, *Software Architecture*, pages 545–564. Kluwer Academic Publishers, 1999.

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, 1998.

[DMNS97] Serge Demeyer, Theo Dirk Meijler, Oscar Nierstrasz, and Patrick Steyaert. Design guidelines for tailorable frameworks. *Communications of the ACM*, 40(10):60–64, 1997.

[DN00] Liliana Dobrica and Eila Niemelä. A strategy for analysing product line software architectures. Technical Report 427, Technical Research Centre of Finland (VTT), 2000.

[DWW98] Tom Dolan, Ruud Weterings, and J. C. Wortmann. Stakeholders in software-system family architectures. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 172–187. Springer, 1998.

[DWW00] Tom Dolan, Ruud Weterings, and J. C. Wortmann. Stakeholder-centric assessment of product family architecture. In Frank van der Linden, editor, *Software Architectures for Product Families, International Workshop IW-SAPF-3*, volume 1951 of *Lecture Notes in Computer Science*, pages 225–243. Springer, 2000.

[EKO+98] Wolfgang Eixelsberger, Manfred Kalan, Michaela Ogris, Häkon Beckman, Berndt Bellay, and Harald Gall. Recovery of architectural structure: A case study. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 89–96. Springer, 1998.

[EOGB97] Wolfgang Eixelsberger, Michaela Ogris, Harald Gall, and Berndt Bellay. Software architecture recovery of a program family. Technical Report TUV-1841-97-21, Distributed Systems Group, Information Systems Institute, Technical University of Vienna, November 1997.

[GB98] Harald Gall and Berndt Bellay. The Hot Spots technique to scavenge for architectural elements. Technical Report TUV-1841-97-11, Distributed Systems Group, Information Systems Institute, Technical University of Vienna, June 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. O. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GK00] David Garlan and Andrew J. Kompanek. Reconciling the needs of architectural description with object-modeling notations. In *Third International Conference on the Unified Modeling Language (UML 2000)*, October 2000.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *IBM Centre for Advanced Studies Conference (CASCON'97)*, pages 169–183, November 1997.

[GS00] David Garlan and João Pedro Sousa. Documenting software architectures: Recommendations for industrial practice. Technical Report CMU-CS-00-169, School of Computer Science, Carnegie Mellon University, October 2000.

[Har98] Mary Jean Harrold. Architecture-based regression testing of evolving systems. In *International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, June–July 1998.

[HHK+01] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki, Antti Viljamaa, and Jukka Viljamaa. Task-driven specialization support for object-oriented frameworks. Technical Report 22, Software Systems Laboratory, Tampere University of Technology, February 2001.

[HNS99] C. Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with UML. In Patric Donohoe, editor, *Software Architecture*, pages 145–159. Kluwer Academic Publishers, 1999.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, 1995.

[JRvdL00] Mehdi Jazayeri, Alexander Ran, and Frank van der Linden. *Software Architecture for Product Families: Principles and Practice*. Addison-Wesley, 2000.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University, November 1990.

[KK98] Anssi Karhinen and Juha Kuusela. Structuring design decisions for evolution. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 223–234. Springer, 1998.

[KKB+99] Mark H. Klein, Rick Kazman, Len Bass, Jeromy Carriere, Mario Barbacci, and Howard Lipson. Attribute-based architecture styles. In Patric Donohoe, editor, *Software Architecture*, pages 225–243. Kluwer Academic Publishers, 1999.

[KM99] Barry Keepence and Mike Mannion. Using patterns to model variability in product families. *IEEE Software*, 16(4):102–108, 1999.

[Kru95] Philippe B. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[KTW98] Roland Knor, Georg Trausmuth, and Johannes Weidl. Reengineering C/C++ source code by transforming state machines. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 97–105. Springer, 1998.

[Kuu99] Juha Kuusela. Architectural evolution. In Patric Donohoe, editor, *Software Architecture*, pages 471–478. Kluwer Academic Publishers, 1999.

[LM97] W. Lam and J. A. McDermid. A summary of domain analysis experience by way of heuristics. *Software Engineering Notes*, 22(3):54–64, 1997.

[LR98] Chang Liu and Debra Richardson. Software components with retrospectors. In *International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, June–July 1998.

[McC76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[MHM98] Jacques Meekel, Thomas B. Horton, and Charlie Mellone. Architecting for domain variability. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 205–213. Springer, 1998.

[MR99] Nenad Medvidovic and David S. Rosenblum. Assessing the suitability of a standard design method for modeling software architectures. In Patric Donohoe, editor, *Software Architecture*, pages 161–182. Kluwer Academic Publishers, 1999.

[MT97] Nenad Medvidovic and R. N. Taylor. Exploiting architectural style to develop a family of applications. *IEE Proceedings. Software Engineering*, 144(5–6):237–248, 1997.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[Nor01] Linda Northrop. *A Framework for Software Product Line Practice — Version 3.0*. Software Engineering Institute, Carnegie-Mellon University, January 2001. Available from http://www.sei.cmu.edu/plp/framework.html.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *Second International Conference on the Unified Modeling Language (UML'99)*, pages 416–429, October 1999.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[RK96] Alexander Ran and Juha Kuusela. Design decision trees. In *8th International Workshop on Software Specification and Design (IWSSD-8)*, March 1996.

[RMRR98] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating architecture description languages with a standard design method. In *20th International Conference on Software Engineering (ICSE'98)*, pages 209–218, April 1998.

[RW96] Debra J. Richardson and Alexander L. Wolf. Software testing at the architectural level. In *Second International Software Architecture Workshop*, pages 68–71, October 1996.

[SB99a] Mikael Svahnberg and Jan Bosch. Characterizing evolution in product line architectures. In *Third International Conference on Software Engineering and Applications (IASTED)*, pages 92–97, October 1999.

[SB99b] Mikael Svahnberg and Jan Bosch. Evolution in software product lines: Two cases. *Journal of Softare Maintenance: Research and Practice*, 11(6):391–422, 1999.

[SB00] Mikael Svahnberg and Jan Bosch. Issues concerning variability in software product lines. In Frank van der Linden, editor, *Software Architectures for Product Families, International Workshop IW-SAPF-3*, volume 1951 of *Lecture Notes in Computer Science*, pages 146–157. Springer, 2000.

[Sch98] William L. Scherlis. Structural views, structural evolution, and product families. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 235–240. Springer, 1998.

[Sch00] Klaus Schmid. Scoping software product lines. In Patrick Donohoe, editor, *Software Product Lines, Experience and Research Directions*, pages 513–532. Kluwer Academic Publisher, 2000.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SRW97] Judith A. Stafford, Debra J. Richardson, and Alexander L. Wolf. Chaining: A software architecture dependence analysis technique. Technical Report CU-CS-845-97, Department of Computer Science, University of Colorado, September 1997.

[TCY93] Will Tracz, Lou Coglianese, and Patrick Young. A domain-specific software architecture engineering process outline. *Software Engineering Notes*, 18(2):40–49, 1993.

[TMA+96] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[Tra95] Will Tracz. DSSA (domain-specific software architecture) pedagogical example. *Software Engineering Notes*, 20(3):49–62, 1995.

[vdHvdLStS98] Peter van den Hamer, Frank van der Linden, Alison Saunders, and Henk te Sligte. An integral hierarchy and diversity model for describing product family architectures. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 66–75. Springer, 1998.

[vO98] Rob van Ommering. Koala, a component model for consumer electronics product software. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 76–86. Springer, 1998.

[WBST98] Nelson Weiderman, John Bergey, Dennis Smith, and Scott Tilley. Can legacy systems beget product lines? In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families, Second International ESPRIT ARES Workshop*, volume 1429 of *Lecture Notes in Computer Science*, pages 123–131. Springer, 1998.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[Wei98] David M. Weiss. Commonality analysis: A systematic process for defining families. In Frank van der Linden, editor, *Development and Evolution of Software Architectures for Product Families*, volume 1429 of *Lecture Notes in Computer Science*, pages 214–222. Springer, 1998.

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[WNS+97] Nelson Weiderman, Linda Northrop, Dennis Smith, Scott Tilley, and Kurt Wallnau. Implications of distributed object technology for reengineering. Technical Report CMU/SEI-97-TR-005, Software Engineering Institute, Carnegie-Mellon University, June 1997.

[You98] Michal Young. Testing complex architectural conformance relations. In *International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, June–July 1998.