Testing framework-based software product lines

Raine Kauppinen

Contents

6   Conclusions   **59**

References   **63**

1   Introduction

A *software product line* can be seen as a process that can be used to implement a set of software products that share common features [Nor01]. The resulting set of software products is a *software product family* [Ard00]. The commonality across a software product family can be exploited when individual software products are designed and implemented. The entire product family is designed so that every application in the family has a similar software architecture. The architecture contains features that can be reused in every application of the family.

An implementation strategy for a software product line is an *object-oriented application framework* [FHB00]. It is a partial design and implementation of an architecture and basic functionality for an application that belongs to a given family. Frameworks allow easy capturing of commonalities between different applications and reuse of features from application to application in a software product family while allowing variation among its members.

In the *product line approach*, as in general in software engineering, testing is essential. The framework of a product family must be reliable and well-tested, because all applications of the family share common parts implemented in the framework. However, also application specific parts as well as the application as a whole need to be tested thoroughly. The product line approach requires a carefully planned testing process that can be easily adapted and used for product families in different application domains.

Unfortunately, testing of product families and product lines is a challenging task. Especially, testing a framework of a product line may be difficult, because a framework is only a partial implementation of a product line application. Assuring the reliability of the framework requires careful testing before applications are derived from it. In practice, however, frameworks are often tested only by testing

applications that are derived from them, which makes it difficult to distinguish errors in the framework from errors in the application specific code. Also, the current testing methods for frameworks and product lines are quite immature, so there is a clear need for more mature testing methods. The testing process should also be supported by testing tools and automated as much as possible.

In this thesis, methods that have been developed or can be applied to the testing of software product lines are studied. Because a framework-based product line can be seen as an incomplete program, testing of partial programs is also of interest. Component testing is also studied, because the specific features of applications that do not belong to the framework can be implemented as components whose interfaces are defined in the general architecture of the product line. Object-oriented testing methods are also of some interest, because software product lines, especially frameworks, are usually designed and implemented with object-oriented techniques.

In addition to the state-of-the-art study of product lines and product line testing, we introduce the concepts of *hook* and *template coverages* for framework-based product lines. These coverages define how much of the functionality of the application specific code expanding the framework has been covered with existing tests. Also, we introduce the design of *RITA*, a framework integration and testing application that can be used in the testing of product families. RITA is an environment that makes it possible to test incomplete framework-based product lines. Moreover, it can be used to manage and keep track of the testing process of the product line. The tool also provides metrics for estimating the testing coverage over the product line, including hook and template coverages.

This thesis is organized as follows. Sections 2 to 4 are a state-of-the-art study of product lines and product line testing. Section 2 is an introduction to software product lines and application frameworks in general while Section 3 contains an

overview of the traditional object-oriented testing methods that form the foundations for product line testing practices. Section 4 introduces testing methods that have been developed especially for or can be applied to product line testing. Then, Section 5 introduces the RITA tool for product line testing and finally, Section 6 summarizes and concludes this work.

2 Product line approach to object-oriented software development

Software engineering is traditionally seen as a development process of a single software application. This approach corresponds to handcrafting every product from scratch regardless of possible similarities between different products. After the introduction of the object-oriented paradigm, reuse of software artifacts such as classes and components has been emphasized. Object-oriented software engineering techniques provide a feasible way of reusing components that can be used as such or with slight modifications in several software products. Lately, many companies have focused on these commonalities between different products and started to capture them in software architectures where they can be used as a foundation for a product line of software products [JRL00].

This section describes the product line approach to software development. Also, a specific way to implement a software product line using object-oriented application frameworks is introduced. First, Chapter 2.1 describes software product lines and families in general. In Chapter 2.2, the key components of a framework-based product line, namely frameworks and framelets, are introduced.

2.1 Product lines

Software product lines have recently received attention in research and especially in industry. Instead of creating software from scratch for each product, many companies have focused on the commonalities between different products and started to capture those in product line architectures [Coh02]. In the research area, case studies of the product line approach have been made [Ard00, Bos99]. In addition to the case studies, a few larger projects have been launched to study the use of product lines in software development. The projects include, for example, the European ARES, ESAPS and CAFÉ projects [JRL00, Lin02a] and SEI's (Software Engineering Institute) framework for product line practice [Nor01].

From the industrial viewpoint, product lines in software engineering are similar to product lines in manufacturing that are used, for example, by Ford (to build cars) and Boeing (to build airplanes) and virtually all companies who exploit the commonalities of their products to enable mass-production. Thus, software product lines are expected to provide similar advantages as product lines in manufacturing. The main advantages are expected to be reduced time to market and lower development costs once the product line for an application domain is functional. Another important advantage, related to object-oriented software engineering techniques, is reuse of software components that is possible in large scale when product lines are used. In this sense, the product line approach to software development is a natural step from handcrafted software products tailored for one customer to mass-production of software products [Nor01].

The creation of a software product line can be seen as a three-step process. It consists of *commonality analysis*, *prototyping*, and *creation of an application engineering environment and an architecture* for the product line [Ard00]. In commonality analysis, common features of all the products in the product line are identified,

as are parameters of variation. The result is a document that typically contains two structured lists: one is a list of assumptions that are true for every member of the product line (commonalities), and the other is a list of assumptions about how the product line members differ (parameters of variation).

After commonality analysis, a prototype of a generic product line member is implemented. The prototype is useful, because it helps to discuss topics related to the product line. The design of the product line, including commonalities and parameters of variation, is revised at this stage. It is worth noticing, however, that the prototype is developed very quickly and is therefore not useful as a real product.

Based on the commonality analysis and the prototype, an application engineering environment and a product line architecture are developed. A new product is formed by taking useful components from the environment and by using preplanned variation mechanisms, such as parameterization and inheritance, to tailor them as necessary. After application specific parts have been added, the application is ready. Object-oriented application frameworks can be used as the application engineering environment that represents the product line architecture [Ard00, Nor01]. The use of frameworks as the core of product lines is covered in more detail in Chapter 2.2.

An example of a software product line is the server product line used in Axis Communications AB. Axis develops, among other products, storage, camera and scanner servers. The product line is illustrated in Figure 1. It is based on a product line architecture that is divided into three product family architectures, namely storage server, camera server and scanner server architectures. The storage server architecture is further divided into CDROM and file server architectures. Each of the architectures contains variations, for example, for different kinds of hardware and standards that are supported by the servers. An object-oriented framework
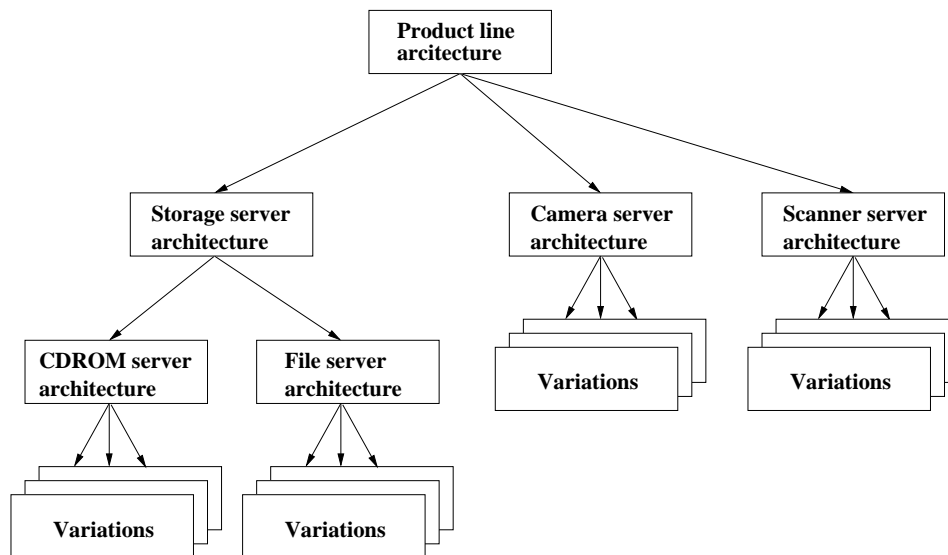
Figure 1: The architecture of the software product line used in Axis Communications AB [Bos99].

is used to implement the product line [Bos99].

It is expected that product lines become the dominating software production paradigm in the near future [JRL00]. Product flexibility is currently a very important factor in the software market and, with product lines, the promise of tailor-made systems built specifically for the needs of particular customers or customer groups can be fulfilled. Especially for large companies, product lines offer an efficient way to exploit the commonalities shared by different products to achieve economies of production [Ard00, Nor01].

Currently, many companies have been successful in developing and fielding products using the product line approach [JRL00]. Examples of such companies are, for example, Axis Communications AB with their server product line, Lucent Technologies with their product lines for different domains [Ard00] and Nokia Mobile Phones with their mobile browser product line [Jaa02]. However, while many companies have succeeded in adopting the product line approach as their standard way of doing business, there are also many companies that either are in

the middle of the adoption process or do not use the product line approch at all. Also, only few companies yet have adequate data to show meaningful gains from the approach, but they still remain committed due to the promise of financial and other benefits of the product line approach [Coh02].

Of course, the product line approach is not without problems. The problems encountered in include, for example, the fact that the use of product lines requires plenty of resources, at least when starting a product line in a new application domain. Smaller companies may not have the resources needed. In addition, there are some technical diffulties with the product line approach, because it is relatively new in software engineering. For example, a well-defined software engineering process for developing and testing product line applications is needed. Also, tool support is needed for developing, testing and maintaining a full-scale software product line [Bos99, Nor01].

2.2   Frameworks and framelets

A natural core of a product line is an object-oriented application framework. Frameworks allow companies to capture the commonalities between applications for the domain they operate in. Thus, framewoks provide an easy way to reuse software components and create applications. This is very appealing to companies who use the product line approach as described in Chapter 2.1 [FHB00, GuB01].

An object-oriented application framework can be defined as a partial design and implementation for an application in a given domain. In this sense, a framework is an incomplete system. It is a set of objects that captures the special expertise in some application domain to a reusable form, but leaves application-specific functionality unimplemented. To create an application from this skeleton, missing

functionality and possible new features must be added [GuB01, Vil01].

Object-oriented frameworks take advantage of techniques that object-oriented programming languages provide. These techniques include, for example, abstract classes, polymorphism, dynamic binding and object composition. Often, objects in frameworks are described as abstract classes. For the framework, they represent the design of its components and, for applications, they serve as templates for creating subclasses. The design of a component includes its interface and usually a core for its implementation. An abstract class, for example, can define a skeleton for an algorithm where every step is defined as a call to an abstract method either in the same class or in another class. Default implementations for these methods can also be defined in the framework or they can be left unimplemented. If the implementation language separates classes and interfaces, like Java, frameworks written in such a language usually contain both an interface and an abstract class for a framework component [Vil01].

The process of reusing a framework to build an application is called *framework adaptation*. Frameworks are adapted by adding the application specific parts to the structure the framework provides [FSJ99, GuB01]. The parts in the framework that are open to customization are called *hot spots* (or *variation points*) while stable parts are called *frozen spots*. Hot spots can be discovered in the domain analysis or they can be defined by a domain expert. *Design patterns* are usually used to organize framework code so that proper parts become hot spots [Gam94]. Different technical variations of hot spots can be captured with *metapatterns* [Pre95].

Metapatterns are a way to use the basic mechanisms of object-oriented programming languages to compose different kinds of object structures with different properties [Pre95]. In practice, metapatterns simply name certain aspects of design structures. More specifically, a metapattern is a named expression for the placement of two methods, a *calling method* and a *called method*, in classes.

In metapatterns, methods of classes are categorized into *template methods* and *hook methods* [Pre95]. A method can be a hook method, a template method or both, depending on its context. Template methods are high-level function bodies that invoke lower-level operations – hook methods – defined somewhere else. Hook methods can be abstract (defined in subclasses), delegated (defined in other objects), regular (the lowest level of action with no succeeding calls), or they can be recursively new template methods for other hooks. A class that contains a template method is called a *template class*, and a class that holds a hook method is called a *hook class*. A class can also be both a template and a hook class. A hook class parameterizes its template class, so hooks are points that must be defined or modified when deriving a new application from the framework.

There are seven metapatterns that result from the combination of three aspects: the separation of hook and template classes, the cardinality of the reference relationship between template and hook classes, and the possible inheritance relationship between hook and template classes. Four of the metapatterns are illustrated in Figure 2. The remaining metapatterns result when the cardinality of the reference from template to hook class is 1:1 instead of 1:*.

In *unification*, both the template method and the hook method are combined in the same class, so adaptations can only be done by overriding the hook method in the subclasses. More flexible approach is *connection*, where template and hook methods are separated to different classes. In connection, one template class can be connected to many instantiations of the hook class. Unification and connection are metapatterns that are most often used to implement hot spots of a framework.

In the *recursive metapatterns* the template and the hook methods have the same signatures and they are defined in the same class hierarchy. In *recursive unification* the template method and the hook method are declared as being the same method. In *recursive connection* the template method resides in a subclass of the hook class.

Unification

1:* Connection

1:* Recursive Unification

1:* Recursive Connection

T = Template class
H = Hook class
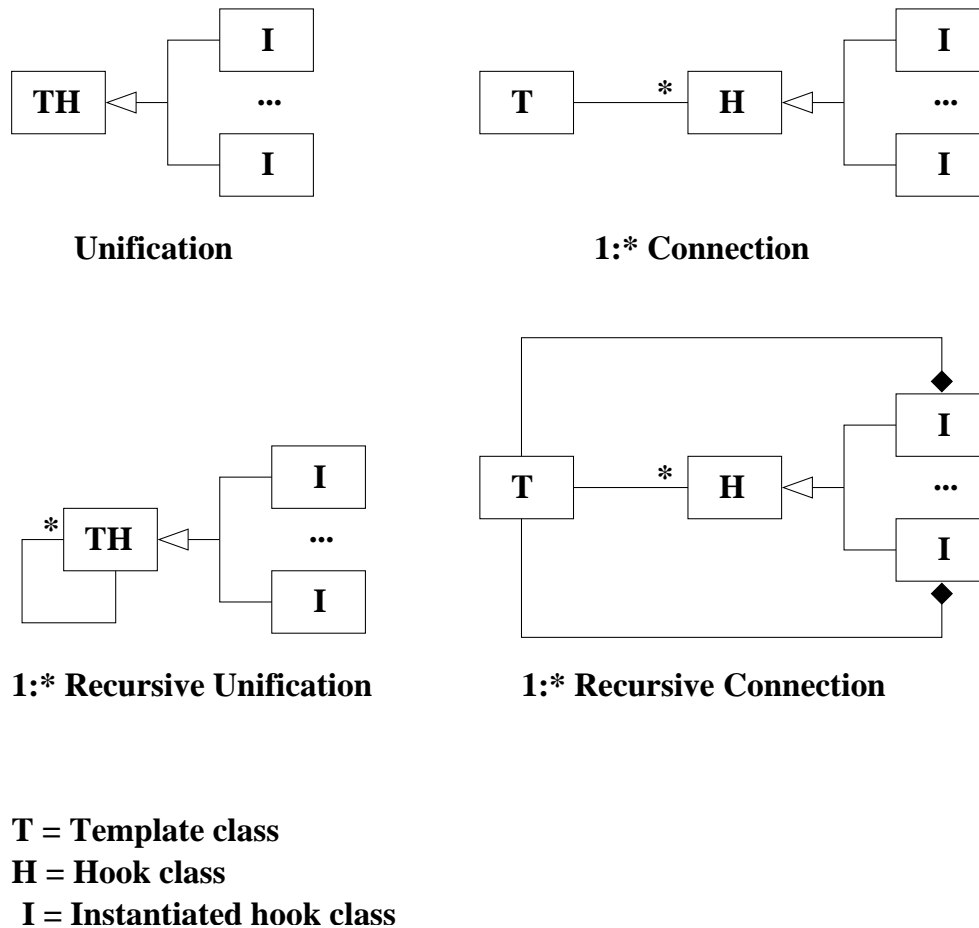 I = Instantiated hook class

Figure 2: Four metapatterns.

Although they allow more flexible implementation of hot spots, the recursive metapatterns are not used as often as unification and connection [Pre95].

An implementation of unification and connection metapatterns is illustrated in Figure 3 as UML class diagram [JSL02]. The framework contains both the frozen spot (template) and the hot spot (hook). The hot spot is instantiated by components in the application specific part. In unification, the frozen spot and the hot spot are in the same class that contains both the template and the hook method. The hook method is abstract and it is overridden by its instantiation in the application specific part. In connection, template and hook methods in the framework are in separate classes. The hook class is instantiated by implementing the ab-

Unification

1:N Connection

Figure 3: An implementation of unification and connection metapatterns as UML class diagram [JSL02].

stract hook class in the application specific part.

Object-oriented frameworks are traditionally designed so that they contain the main event loop and have full control over the application. This is called the *inversion of control* or the *Hollywood principle* (*"Don't call us, we'll call you"*), because the framework itself, instead of the user-created application part, is responsible for the flow of control [Vil01]. However, this is not always the case. A distinction can be made between *calling* and *called frameworks*: calling frameworks contain

the main loop whereas called frameworks do not [SBF96]. Good examples of called frameworks are *framelets*. They are mini-frameworks that contain less than ten classes and have a simple, clearly defined interface [Pre95]. Framelets do not assume the main control of the application although they follow the Holly-wood principle. They are usually used as architectural elements in product line architectures, but they can also be used, for example, to integrate legacy software components to frameworks and product lines [PrK99].

Figure 4 shows an example of the structure of an application derived from a product line based on an object-oriented framework. The framework provides a skeleton for the application. It defines templates that contain hot spots where the application specific parts can be plugged in. In this case, the framework has two templates and three hot spots. However, the application uses only two of the hot spots. The application specific parts, in this case a framelet and an application code element, are connected to their respective hot spots via interfaces. Two of the three hot spots of the framelet are used to extend the framelet with an application code element and with another framelet by using the same mechanism. The hot spots in the two templates of this framelet are left unimplemented.

Figure 5 shows the interface between the template and the hot spot marked with the filled rectangle in Figure 4. The interface is implemented using the connection metapattern where the cardinality of the connection between the template class and the hook class is 1:1. The template class contains a template method that uses the hook class. The hook class is abstract, so the class and its hook method are implemented by a class in the framelet that extends the framework via this interface.

In addition to calling and called frameworks, frameworks can also be classified by the way they are adapted to derive new applications [GuB01]. A *white-box framework* is a framework that can be used by inheritance only. The framework
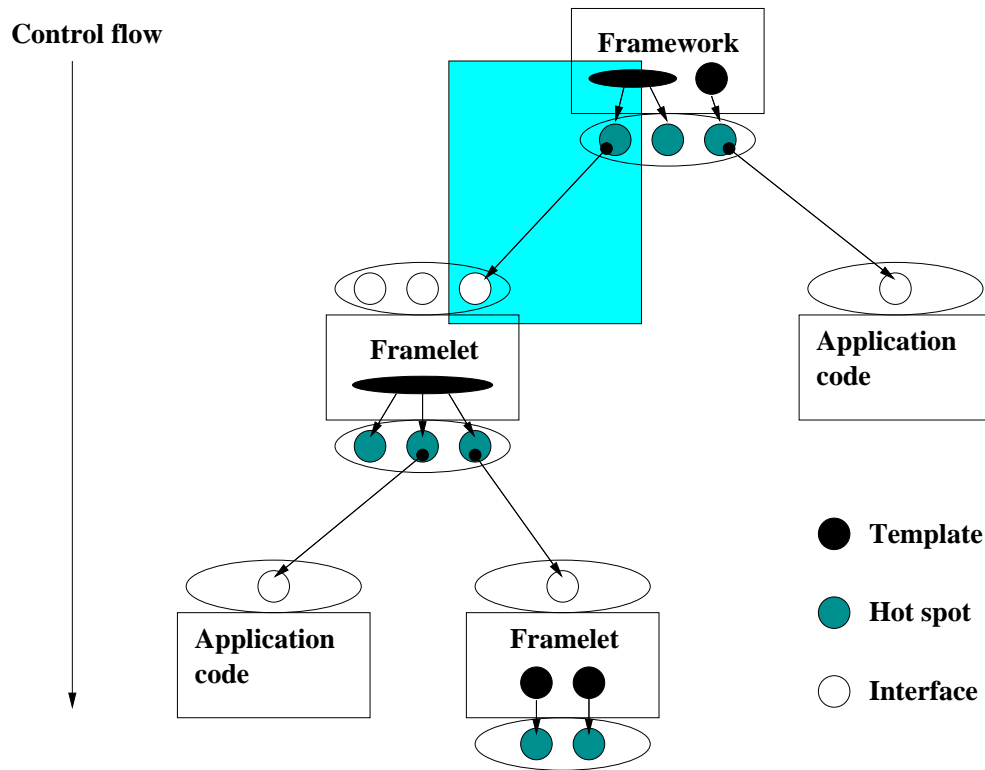
Figure 4: An example of the structure of an application derived from a product line based on an object-oriented framework. The interface between the template and the hot spot marked with the filled rectangle is shown in more detail in Figure 5.

shown in Figure 4 is a white-box framework. It is impossible to use a white-box framework without understanding how it works internally. A *black-box framework*, on the other hand, can be used by configuring existing components. A black-box framework is easier to use than a white-box framework, because its internal mechanisms are – at least partially – hidden from the developer. The drawback is that this approach is less flexible, because the capabilities of a black-box framework are limited to what has been implemented in the set of provided components. A framework that has some characteristics from both white-box and black-box frameworks is sometimes called a *gray-box framework* [Vil01].
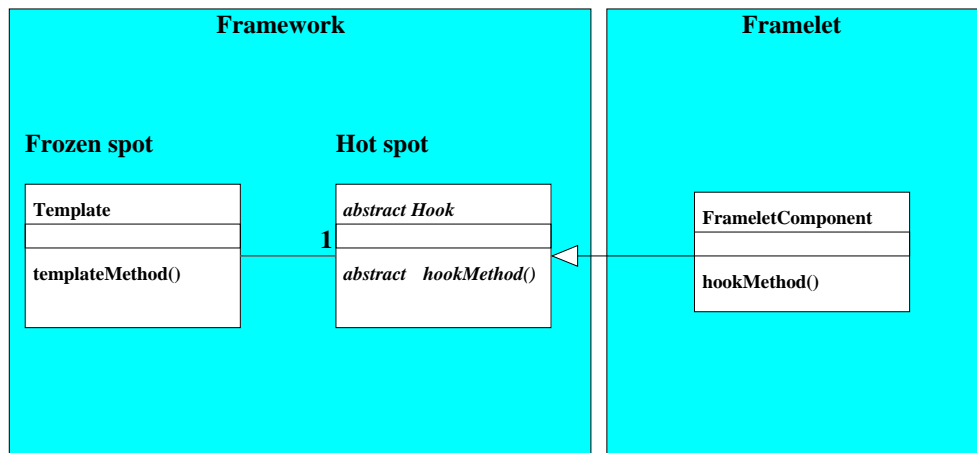
Figure 5: An implementation of the interface in Figure 4 as UML class diagram.

Although object-oriented application frameworks fit very well to the product line approach and are widely used, there are some well-known difficulties with them. These difficulties may not appear before frameworks are used in real projects, so they may be hard to deal with [Bos00]. Typical difficulties include, on one hand, the *framework gap* and, on the other hand, the *overlap of framework entities*. The framework gap occurs when the structure of a framework does not cover the requirements of an application. The overlap of framework entities may occur if the same real-world entity is modeled in more than one way in the framework. However, these difficulties can be avoided with careful design of the framework [FSJ99]. Despite the difficulties encountered, increased reusability of software components and reduced time to market for applications make frameworks feasible to use [FHB00, GuB01].

3   Testing of object-oriented software systems

Testing of traditional procedural applications is a well-established software development practice. However, the object-oriented paradigm has created new

challenges to testing, because object-oriented applications have a different kind of structure than traditional procedural applications. Especially, dealing with instantiations of classes and their collaboration can be very difficult when testing is performed. These new challeges to testing of object-oriented applications have been met by extensive research that still produces new results. The results provide ways to apply traditional testing methods in the object-oriented context. Also, new testing theory and methodology designed specifically for object-oriented applications has been proposed [AlA01, BOP00, Lab00].

This section is an overview to object-oriented testing methods that form foundations for testing practices specific to product lines discussed in Section 4. First, the traditional object-oriented testing process and the need to design for testability are discussed in Chapter 3.1. Then, in Chapter 3.2, the basic methods for testing object-oriented software are introduced. Chapter 3.3 discusses metrics for estimating testing adequacy. Finally, Chapter 3.4 gives an overview of how object-oriented testing can be automated and supported by testing tools.

3.1 Object-oriented testing process and design for testability

The overall goal of testing is to provide confidence in the correctness of an application by executing and studying its code or parts of it. Traditionally, testing has been used in two ways to achieve this goal. First, testing has been used to assist developers to find faults in applications so that they can be repaired. Second, testing has been used to determine whether an application can perform as specified by its requirements [Bei90].

Recently the role of testing has been expanded to provide techniques for estimating the reliability of a software component. This is especially important when object-oriented programming is used, because software components are being

constantly reused [AlA01, Nor01]. However, testing of object-oriented software is not an easy task, because the object-oriented paradigm is more complex than the traditional procedural one. Useful testing methods for object-oriented programming are available, but they may not be effective if not used properly [AlA01]. Unfortunately, the gap between reasearch knowledge and actual testing practice can be very large [McK94].

Before applying testing methods, a software engineer must understand the basic principles that guide testing [Pre97]. According to these principles testing should be objective, systematic, thorough and integral to the development process. Furthermore, the tests used should be planned long before the actual testing begins. Also, testing should begin from the small and progress towards the large, which means that the first tests planned and executed focus on individual components of the application under test and the focus of testing moves towards the testing of entire application as a whole as the testing progresses.

Testing requires careful planning and a well-defined process that can be integrated with the software development [McK94]. The testing of an object-oriented software system can be modeled, for example, with the *standard V-model* [FeG99]. The V-model is illustrated in Figure 6. According to the model, software is developed in four levels. The top level is *requirement specification* where the specifications of the software are gathered. The second level is *architectural design* in which the overall architecture of the system is designed. The third level is *detailed design* where each component of the system is designed. The lowest level is *code level* that contains the concrete implementation of the system in a programming language.

Testing is performed at every level of the V-model. On the left side, tests that will be used are designed and written during the design of the system at each level. The designed models can also be validated using *inspections* [GiG93]. In
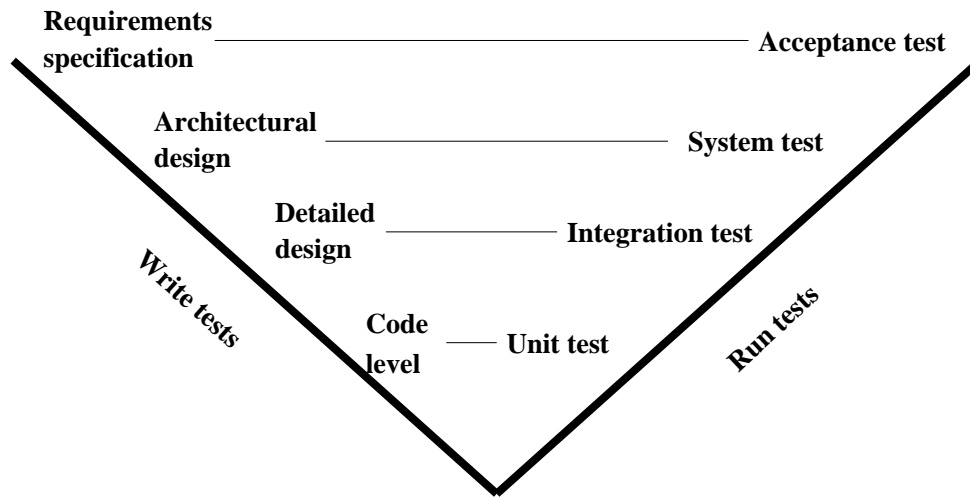
Figure 6: The V-model for testing object-oriented software applications [FeG99].

an inspection, a design model or part of it is reviewed thoroughly before it is implemented. Reviewers can be members of the team that developed the model or they can be other experts who have the knowledge needed to evaluate the model under inspection. The validation of design models via inpections is usually not considered testing, because no application code is executed. However, inspections are sometimes seen as testing that is performed long before the actual implementation of the designed application has started [McS01].

On the right side of the V-model, the designed tests are executed. At different levels different testing strategies and methods are needed. At the code level, classes and their methods are tested via *unit testing*. As classes are integrated into components at the detailed design level, *integration testing* is performed. At the architectural design level the focus of testing moves to testing the entire system as a whole with *system testing*. Finally, at the requirements specification level *acceptance testing* is used to determine whether the system meets its specifications or not [FeG99]. Object-oriented testing strategies and methods are discussed in more detail in Chapter 3.2.

The cost and difficulty of testing object-oriented systems can be reduced with a strategy called *design for testability*. Design for testability means that every component of a system is designed so that it can be easily tested. Easy testing of components requires that testing is directly supported for example via build-in testing capabilities. Design for testability is especially important when the product line approach to software development is used although the strategy has received relatively little consideration in software development [Bin94].

3.2   Testing methods for object-oriented systems

The only way to completely guarantee the correctness of a program is to execute it on all possible inputs, which is usually impossible or at least impractical. Thus, systematic testing techniques generate a representative set of *test cases* to provide adequate testing for the program [Bei90]. Metrics for determining adequacy of testing are covered in Chapter 3.3.

In its most general form, a test case is a pair *(input, expected result)*, in which *input* is a description of an input to the software under test and *expected result* is a description of the output that the software should give for this particular input. Test cases are organized into a *test suite*. Test suites are usually organized based on the kinds of test cases. Developing and organizing a test suite so that it is correct, observable and adequate is one of the greatest challenges in testing [McS01].

There are two general forms of testing, namely *program-based* and *specification-based*. Program-based testing is called *white-box testing* (sometimes also *glass-box testing*) and specification based testing is called *black-box testing* [Pre97]. In white-box testing, certain aspects of the code of a program, such as statements, branches, data dependencies or paths are used to select test cases. White-box test

cases can often be generated from the program code with testing tools.

In black-box testing, test cases are designed to show that a program satisfies its specifications. The specifications can be functional or non-functional. Functional specifications define features that the program has and non-functional specifications define other properties of the program, such as performance, security and maintainability. Both functional and non-functional specifications must be considered when black-box test cases are defined. Black-box test cases are usually generated manually based on requirements derived from the specifications. Since white-box and black-box testing complement each other, both types of testing are usually performed [McS01].

Most systematic testing techniques are used to validate program units. In object-oriented programming, these units include objects and classes. Classes are often seen as the smallest units of an object-oriented program that should be tested [McS01]. In this case, unit testing in object-oriented programming means class testing. Unlike unit testing of conventional procedural software, which focuses on the algorithmic details and data flow, class testing for object-oriented software is driven by the operations encapsuled by the class and the state behavior of the class [Pre97]. A variety of white-box methods for testing classes have been proposed and used for testing of the object-oriented programs [AlA01, Bal00].

Additional testing is required when units are combined or integrated into subsystems. Integration testing focuses on the interfaces between units and subsystems. Interface problems include, for example, errors in input and output formats and misunderstood entry or exit parameter values. In addition to interface problems similar to procedural programs, integration testing of object-oriented programs can be very difficult due to lack of hierarchical control structure, which makes it difficult to use the traditional top-down or bottom-up integration strategies. Instead, object-oriented integration testing can be performed, for example, by

testing clusters of collaborating classes or components [Pre97] or by exploiting an underlying class dependency diagram to determine the test order of classes [Lab00]. Although most integration testing methods are black-box methods, also white-box methods for integration testing have been developed [HMF92].

The overall architecture of a system is tested using system testing, where the entire system is tested as a whole. At this level, the focus of testing shifts completely to black-box testing. The adequacy of system testing is based on the number of the possible ways to use the system. In object-oriented systems, for example, use cases can be used to derive test cases for system testing [Pre97]. In addition, acceptance testing is used to verify that the system meets its requirements. When a system is functional, *regression testing* is periodically performed to ensure the correct functionality of the system. Regression testing is done, for example, when changes are made to components of the system [Bei90].

Whenever test cases are executed, a *test driver* is used. A test driver runs each of the test cases in a test suite and reports the results. It creates one or more instances of a class to run a test case. A test driver can take a number of forms. It can be, for example, a separate class that is used to run the tests. Sometimes, when only parts of a program are tested, *stubs* may also be required. Stubs are used to replace missing parts of a program needed to be able to complete a test, that is, run the program. A stub can be, for example, an empty implementation of a method that returns a default value that the calling method can properly use [FeG99, McS01].

3.3   Metrics for estimating testing adequacy

Because an application usually cannot be tested on all possible inputs, the goal of testing is to adequately test the application, as mentioned in the beginning of

Chapter 3.2. However, it is not easy to define when an application is adequately tested. There are many aspects to consider when addressing this question. The amount of testing required should be determined relative to the long-term and short-term goals of the project and to the software being developed. For example, the expected lifetime of an application is one viewpoint. Another may be the type of the application. For example, a life-critical real-time software system requires very extensive testing before it is released, whereas many information system applications can be released after a relatively short testing period [Bei90].

Adequacy of testing can be measured based on the concept of *coverage*. Coverage is a measure of how completely a test suite exercices the capabilities of a piece of software. Coverage can be measured in at least two ways. One way is in terms of the number of the requirements in the specification, which conforms to black-box testing. For example, the percentage of the requirements that are tested by test cases in a test suite can be measured. This is called *requirements coverage* [FeG99]. Another, more common way is in terms of the amount of code executed when running a test suite. A test suite might be adequate if, for example, some portion of the source code has been executed at least once. This approach matches the approach of white-box testing.

Three commonly used measures of adequacy for white-box testing of object-oriented software are *code-based coverage*, *state-based coverage* and *constraint-based coverage* [McS01]. Code-based coverage is based on counting how much of the code that implements a class is executed across all test cases in a test suite. The goal is to make sure that every logical unit (for example, every statement in *statement coverage* or every condition in *condition coverage*) in or every path through the code implementing a class is executed at least once when all test cases have been executed. If certain units of code (or paths) have not been reached, the test suite needs to be expanded or the code needs to be corrected to remove unreachable

parts. Even with full code coverage, the test suite for a class might not be adequate because it might not exercise interactions between methods as state-based and constraint-based coverages do. So, the use of one or both of those coverages to complement code-based coverage – and vice versa – is necessary.

State-based coverage is based on counting the number of transitions in the state transition diagram of a component covered by a test suite. If any of the transitions is not covered, the component has not been tested adequately and more test cases are needed. If test cases are generated directly from the state transition diagram, they achieve this measure [Bal00]. Even if all transitions are covered, adequate testing is doubtful, because states usually embrace a range of values for various object attributes. So, in addition to the test cases that achieve state-based coverage, further testing is needed for typical and boundary values of the attributes to improve testing coverage.

Parallel to adequacy based on state transitions, adequacy can be expressed in terms of how many pairs of *pre-* and *postconditions* have been covered (constraint-based coverage). If, for example, the preconditions for an operation are *A* or *B* and the postconditions are *C* or *D*, the test suite should contain test cases for all the valid combinations (*A=true, B=false, C=true, D=false; A=false, B=true, C=true, D=false*; and so on). If one test case is generated to satisfy each requirement, the test suite meets this measure of adequacy [McS01].

The three measures of coverage are illustrated in Figures 7-10. Figure 7 shows a partial Java code of a simple state machine class. Figure 8 shows a test suite that executes all the statements of the code in Figure 7. Every statement can be reached with a relatively small test suite. The creation of a `SimpleStateMachine` instance and two state changes are all that is needed for full code-based coverage. However, test suites generated according to state-based and constraint-based coverages test the class more thoroughly than the code-based test suite.

```
1:   public class SimpleStateMachine {

2:       private int currentState;
3:       private boolean validStates[];

     ...

         // constructor
4:       public SimpleStateMachine() {
5:           currentState = 0;
6:           validStates = {true, true, false};
7:       }

     ...

         // accessor
8:       public boolean changeState(int state) {
9:           boolean changeOk = true;

10:          if (validStates[state] == true)
11:              currentState = state;
12:          else
13:              changeOk = false;

14:          return changeOk;
15:      }

     ...

16: }
```

Figure 7: Partial Java code of a simple state machine class.

```
Number    Description                        Coverage         Test case
1         Creation of a class instance       Lines 1-7        SimpleStateMachine sm = new SimpleStateMachine()

2         Legal state change from 0 to 1     Lines 8-11, 14-16  sm.changeState(1)

3         Illegal state change from 1 to 2   Lines 8-10, 12-16  sm.changeState(2)
```
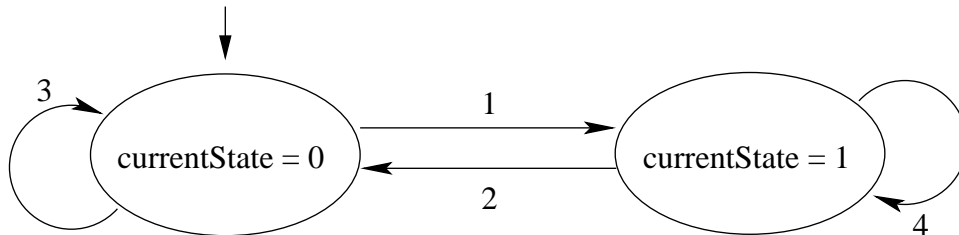
Figure 8: Test suite based on statement coverage for the simple state machine in Figure 7.

For state-based coverage, the state transition diagram has to be created. In this case there is only one variable that affects the state of the class: `private int currentState` declared on line 2. Because this simple state machine can only be in states 0 and 1 (as stated on line 6), there are two possible states and four transitions. The state transition diagram is shown in Figure 9. The figure also shows a test suite that covers all the transitions. The test suite contains four independent test cases. Each test case starts from the initial state (`currentState = 0`) and covers one transition that is not covered by other test cases. It is also worth noticing that transitions 3 and 4 are taken when state is changed to the

state where the state machine already is or when an illegal transition to any other state than `0` or `1` is tried, so there are several test suites that cover all the transitions. Moreover, the instantiation of the class is not tested with this test suite, because the starting state – from which there is a transition to the initial state of the machine – is omitted from the diagram.

State transition diagram

Corresponding test suite

Number	Coverage	Test case
1	Transition 1	changeState(1)
2	Transitions 1 and 4	changeState(1), changeState(2)
3	Transitions 1 and 2	changeState(1), changeState(0)
4	Transition 3	changeState(2)

Figure 9: The state transition diagram for the simple state machine in Figure 7 and a corresponding test suite.

Finally, contraint-based coverage is illustrated in Figure 10. The pre- and postconditions of the simple state machine class depend on the same variable `private int currentState` as in the state transition diagram. Here, the precondition is the state the class is in and the postcondition is the state to which the class moves after the method `public boolean changeState(int state)` is executed. In this case, the local `changeOk` variable can be used as the other part of the postcondition: the variable (and the return value of the method) is `true` if transition is made and `false` if the state is not changed due to illegal target state. As can

be seen, the constraint-based coverage is very similar to the state-based coverage although constraint-based coverage is in this case slightly more thorough than the state-based coverage.

```
Number    Precondition      Postcondition        Test case
1         not instantiated  currentState = 0,    SimpleStateMachine sm = new SimpleStateMachine()

2         currentState = 0  currentState = 0,    changeState(0)
                            changeOk = true
3         currentState = 0  currentState = 1,    changeState(1)
                            changeOk = true
4         currentState = 0  currentState = 0,    changeState(2)
                            changeOk = false

5         currentState = 1  currentState = 0,    changeState(0)
                            changeOk = true
6         currentState = 1  currentState = 1,    changeState(1)
                            changeOk = true
7         currentState = 1  currentState = 1,    changeState(2)
                            changeOk = false
```

Figure 10: Test suite based on pre- and postconditions for the simple state machine in Figure 7.

3.4 Automated testing and tool support

Object-oriented testing, as well as testing of traditional procedural programs, requires automation and tool support [FeG99]. Without automation, less testing will be done or greater cost will be incurred to achieve a given testing goal. An automated testing environment needs functions to initialize a system and its environment, execute test suites and replay them under predefined conditions. Many testing tools that have been originally developed for testing of traditional procedural programs can also be used to test object-oriented applications with minor changes. Also, several test tools that provide a testing environment especially designed for object-oriented testing have recently become commercially available [FeG99].

Object-oriented testing can be automated in several ways with testing tools. The level of automation provided by different tools varies from very low (the tool automates only some specific testing tasks such as test driver and stub generation) to very high (the tool provides an entire testing environment). However, every

tool that can automate a part of the testing process is useful, because manual testing of object-oriented programs is a tedious task [Bal00, DHS02].

Figure 11 shows different kinds of tools that can be used at different levels of the V-model. Testing can be managed during the entire software development process with *management tools*. On the left side of the model, tests can be designed and written with *test design tools* at the requirement specification, architectural design and detailed design levels. Also, *static analysis tools* can be used to create unit tests at code level. On the right side of the model, where the actual testing is performed, *coverage tools* can be used to evaluate testing coverage of unit testing. *Dynamic analysis tools* and *debugging tools* can be used with both unit and integration testing. At higher levels of testing – system and acceptance testing – *performance simulator tools* are useful. In addition, *test execution and comparison tools* can be used at every level on the right side of the model to run the tests and compare the obtained results with the expected ones [FeG99].
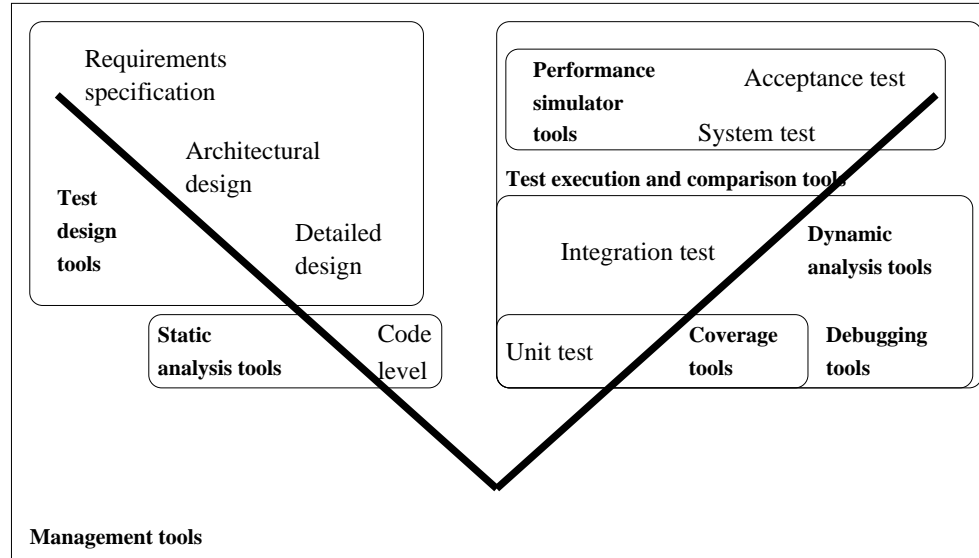

Figure 11: Testing tools and V-model levels [FeG99].

In addition to commercial tools, testing automation has been under extensive academic research. At lower level of automation, techniques and algorithms for automatic test driver and stub generation as well as for automatic test case generation have been proposed [AlA01, Bal00, BOP00]. In addition, entire testing environments for unit testing have been developed. For example, the *JUnit framework* is designed for object-oriented unit testing [Fow99]. The use of JUnit is a flexible way to design, execute and manage unit tests. JUnit can also be used with commercial tools, for example, with the *JTest environment* to further automate unit testing [Par03]. Another testing environment is the *Roast framework* [DHS02], which generates test drivers with embedded test cases and provides unit operations for automated class testing.

4   Testing in the product line approach

The traditional object-oriented testing methods discussed in the previous section form foundations for product line testing. However, when a product line is tested, or multiple product lines are tested, product line specific issues arise. These issues include, for example, problems of scale, because all applications of the product line need to be tested. The key idea in product line testing is to reuse test cases and other testware throughout the entire product line instead of testing every application as an independent software product.

In this section, product line specific testing issues are discussed. In Chapter 4.1, the overall testing process for product lines is described. Then, testing methods specific to framework-based product lines are introduced in Chapter 4.2. Metrics for product line testing are discussed in Chapter 4.3, and Chapter 4.4 focuses on the automation and tool support needed in product line testing. Finally, Chapter 4.5 gives an overview of the state-of-the-art of product line testing.

4.1 Product line testing process

Product line testing, like the traditional testing of a single application, is usually performed according to the standard V-model described in Chapter 3.1. Although product line testing can also be performed according to the V-model, the testing process has to be integrated with the higher level product line business process [Lin02a].

Unlike the V-model that has become a standard approach for testing of single applications, there is no standard process framework that would integrate the overall product line business process to the product line testing process. Instead, there are several process frameworks that can be used. For example, the ESAPS and CAFÉ projects and SEI have their own frameworks [Lin02b, Nor01].

The process framework of the CAFÉ project, *CAFÉ Process Reference Model* (CAFÉ-PRM) is illustrated in Figure 12. It is an improved version of the framework developed in the ESAPS project that divides software engineering to two main levels: *software family reverse engineering activities* and *derivation activities*. Both levels are divided further into six phases that correspond to the phases in the traditional *waterfall process model* for software engineering [Roy70].

At the software family reverse engineering level, the core of the product line is developed. This corresponds to the development of the application framework in the framework-based product line approach. This level deals with issues that concern all the applications in the domain, while issues specific to a single application belong to the application engineering level. As can be seen in Figure 12, testing is performed in four of six phases at both levels of the CAFÉ-PRM framework [Lin02b].

Between the two main levels resides a repository of *reusable assets* that contains domain specific *artifacts*, like documents, application components or framelets

that can be reused throughout the application domain. In other words, reusable assets are components that can be used in every application in a domain, either as such or with slight modifications (see Chapter 4.2 for more information).
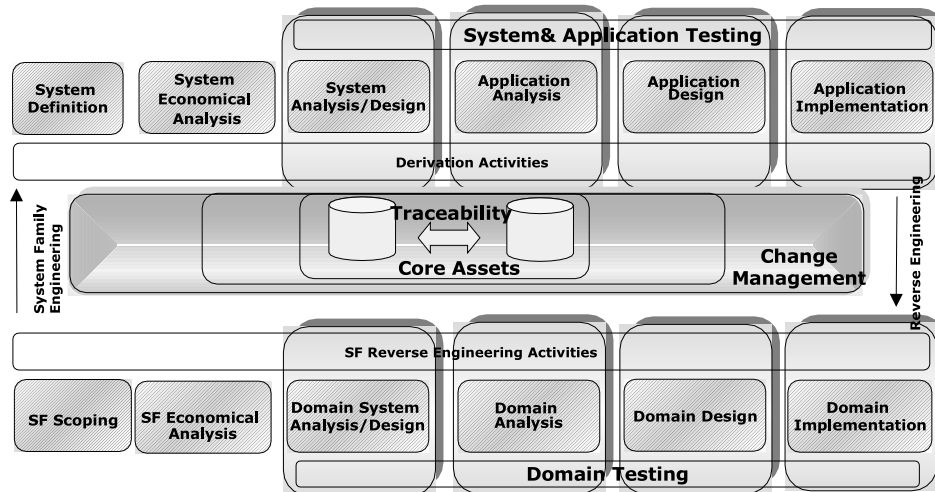

Figure 12: CAFÉ Process Reference Model (CAFÉ-PRM) [Lin02b].

SEI's *Framework for Product Line Practise* is built on the *CMMI process framework* [JoS02]. It is very similar to the CAFÉ-PRM. It also has domain and application engineering levels and specific development phases at both levels. The use of reusable assets is also emphasized in this framework. In addition, SEI's framework has an additional key aspect, *management activities* for the product line practice. Management activities cover the essential managerial aspects of the product line engineering [Nor01].

Integrating testing to product line process frameworks has proved to be somewhat problematic in practice. It seems to be unclear from the business point of view where testing belongs in the overall product line process. For example, both CAFÉ-PRM and SEI's frameworks imply that testing is performed according to the standard V-model, but neither framework explicitly integrates the V-model to the overall product line process. However, SEI's framework has a more de-

tailed description of different testing phases related to development phases of the framework.

4.2 Testing methods for product lines

In the product line approach, testing is based on generating, managing and using *reusable test assets* that contain test suites and other *test artifacts*. To effectively manage test assets, a *test asset repository* is needed. Reusable assets in Figure 12 contain artifacts shown in Figure 13, which illustrates *CAFÉ Asset Reference Model* (CAFÉ-ARM). It describes reusable assets that can be reused throughout the lifecycle of a product line. The assets include, for example, different documents, use cases, scenarios, classes and other software components [Lin02b]. Testing related assets include, for example, test plans, test suites and test reports.
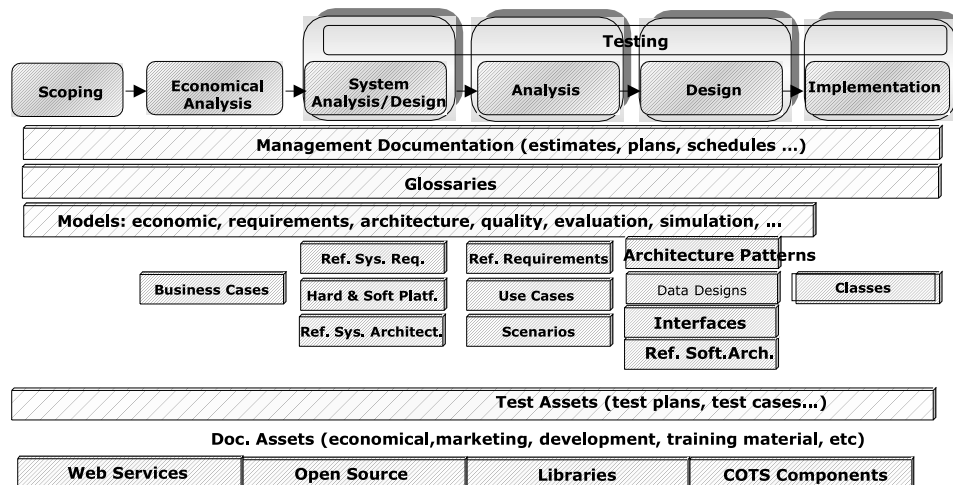

Figure 13: CAFÉ Assets Reference Model (CAFÉ-ARM) [Lin02b].

The relationship between test assets and a product line is illustrated in Figure 14. The figure shows a product line architecture from which product A is derived. On the application domain level, the product line has a test plan and corresponding

test cases that are designed during the domain engineering process. Since these artifacts already exists when product A is implemented during the application engineering process, they can be used to derive an application specific test plan and application specific test cases. The generated test plan and test cases can also be reused later, when new applications are derived from the architecture.
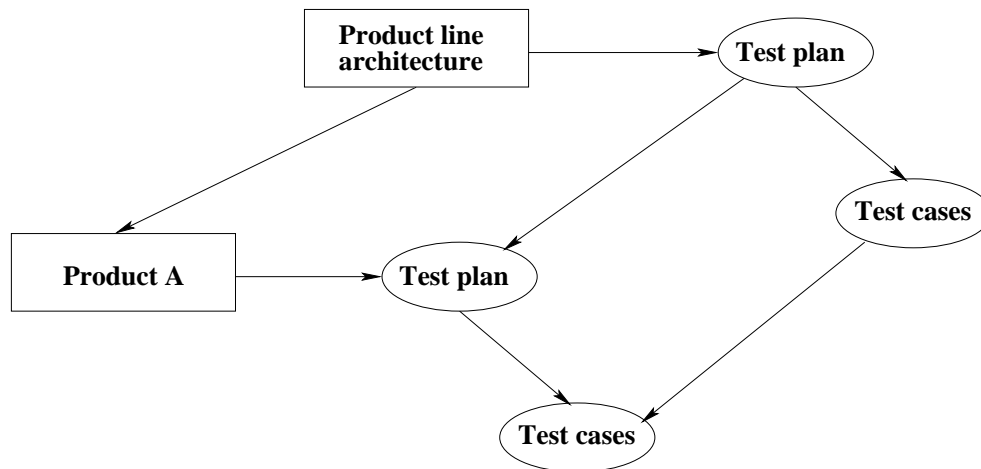
Figure 14: The relationship of the test assets and a product line [McG01].

Two basic approaches have been proposed for managing test assets. The first one is to have a separate asset repository – usually a database management system – where reusable assets, including test assets, are kept. In this approach, the application code is kept separate from the test assets [McG01]. The idea of separating the code and the assets fits very well to the CAFÉ-PRM and is illustrated in its representation. The asset repository is located between the two main levels of the framework. The reason for the separation is reusable and thus repetitive nature of test assets. Also, a typical product line contains huge amount of test assets, so an asset repository provides a feasible way to organize them.

However, the code and the assets are not always separated. The second approach to manage test assets is to integrate them with application code. For example, documentation and test cases along with their expected results can be integrated

directly to Java code with a Javadoc extension [HoW02]. In this approach, there is no separate asset repository. Instead, the necessary artifacts are extracted from the application code with specific tools when needed.

The integration of application code and testing assets corresponds closely the ideas of *generative programming*[CzK00] . In generative programming, applications are built by extracting the necessary features from the base code to the application during compilation. This idea can be seen as an alternative approach for building a product line. In this case, commonalities of different products of a product line are captured as features and added to a new product line application via extraction from a base feature set. Integrating testing assets into the feature set would therefore fit nicely to the ideas of generative programming .

In the framework-based product line approach, both a separated and an integrated asset repository can be used. The standard way to implement an asset repository is to separate and manage application code and test assets with a DBMS. However, object-oriented application frameworks also provide a simple way to integrate the assets and the application code via inheritance mechanism. This idea has been applied in the form of *built-in tests* [Wan00]. The idea is to integrate tests into classes of a framework. In this way, tests can be inherited to the classes that implement the application framework. In other words, built-in tests can be used as default properties or services of the classes in a framework [JSL02]. This approach makes it possible to build *self-testing components* for a software product line in similar way as they are used in electrical engineering [Mau93].

While test assets and their use in the product line approach have been studied, the current product line practice does not describe which testing methods should be used when product lines are tested. The asset repository and test artifacts are used to support testing, but no product line specific testing methods

have been presented apart from build-in tests that can be used if the product line is framework-based. Instead, it is assumed that the methods are similar to the object-oriented testing methods presented in the previous section. This assumption is the main problem with the current product line testing practice and it is discussed in more detail – along with the state-of-the-art of the product line testing in general – in Chapter 4.5.

4.3   Metrics for estimating product line testing

Traditional object-oriented measures of coverage (see Chapter 3.3) can be applied at the component level when product lines are tested. In addition, product line specific measures of coverage that can be used to evaluate the overall testing coverage over a product line are also needed. We propose two new coverage criteria for framework-based product line applications: *hook coverage* and *template coverage*. They define how much of the functionality of hand-written application code elements and framelets extending an application framework via hot spots have been covered with existing test suites.

Let us have a template class t that has template methods $tm_1 \ldots tm_n$ as shown in Figure 5. Each template method references one or more hooks $h_1 \ldots h_n$. Each hook is an abstract method, either in a unified template and hook class (the unification metapattern) or in an abstract hook class (the connection metapattern) as shown in Figure 2. We say that a hook is *connected* if the abstract hook method is instantiated either by overriding it in the unified template and hook class (the unification metapattern) or by deriving a class that implements the abstract hook class and its hook method (the connection metapattern). Thus, a hook is connected only if it is expanded by hand-written application code or by a framelet. Especially, this means that an abstract hook method can not be connected.

The template class t is *connected* if one or more of its hook methods are connected. In the unification metapattern, the unified template and hook class is connected if one or more of its abstract hook methods are instantiated by overriding them in hand-written application code or in a framelet. In the connection metapattern, the template class is connected if the abstract hook class is instantiated by deriving a class that implements it in hand-written application code or in a framelet.

Let us now have a test suite ts that contains test cases $tc_1 \ldots tc_n$. We say that a test case *reaches* a connected hook method if it executes some or all of the code in the hook method. We say that a connected hook method is *tested* if one or more of the test cases in the test suite ts reach it and the test cases provide 100% statement coverage for the hook method. If one or more of the test cases in the test suite ts reach the hook method, but the statement coverage they provide is below 100%, the hook method is *partially tested*. The hook method is *not tested* if it is not reached by any test case in the test suite ts. We use the statement coverage, because it measures how much of the code is reached with the test cases used. Especially, the statement coverage gives an overview of how much of the code is not tested at all. However, other method level coverages could also be applied. We say that a connected hook class is *tested* if all of its hook methods are tested and *partially tested* if one or more of its hook methods are tested or partially tested; otherwise the hook class is *not tested*.

Let us furthermore have a template method tm in a connected template class t. The template method tm references one or more of its hook methods $h_1 \ldots h_n$ via hook method references $hr_1 \ldots hr_n$. In other words, a hook method reference is a method call from a template class to an instantiated hook method that resides either in hand-written application code or in a framelet. We say that a test case *goes through* a hook method reference if the test case invokes the hook method reference, that is, if the test case executes the statement containing the method

call from the template class to the instantiated hook method. We say that a hook method reference is *tested* if it is gone through by one or more test cases and *not tested* if the hook method reference is not gone through by any test case.

We say that the template method tm is *tested* if its every hook method reference is tested and *partially tested* if one or more of its hook method references are tested; otherwise the template method is *not tested*. A connected template class is *tested* if all of its template methods are tested and *partially tested* if one or more of its template methods are tested or partially tested; otherwise the template class is *not tested*.

We can now define *hook method coverage*, *hook class coverage*, *template method coverage* and *template class coverage* for connected templates and hooks. We define the hook method coverage HMC, provided by a test suite ts, for a connected hook method hm as follows:

$$HMC = the\ statement\ coverage\ of\ the\ hook\ method\ \texttt{hm}\ provided\ by\ the$$

$$test\ cases\ in\ the\ test\ suite\ \texttt{ts}\ that\ reach\ the\ hook\ method\ \texttt{hm}.$$

Based on this definition, we define the hook class coverage HCC for a connected hook class hc as follows:

$$HCC = \frac{HMC_1 + \ldots + HMC_n}{n},$$

where $HMC_1, \ldots, HMC_n$ are the hook method coverages of the hook methods residing in the hook class hc and n is the number of all the hook methods in the hook class hc.

In a similar way, we define template method coverage and template class coverage. The template method coverage TMC for a template method tm is as follows:

$$TMC = \frac{HR_{tested}}{n},$$

where HR_{tested} is the number of tested hook references in the template method tm and n is the number of all the hook method references in the template method

tm. The template class coverage TCC for a template class tc is as follows:

$$TCC = \frac{HRC_{tested}}{n},$$

where HRC_{tested} is the number of tested hook method references in the template class tc and n is the number of all the hook method references in the template class tc.

These coverages are illustrated in Figures 15 and 16. Figure 15 shows a framework with three hot spots. Two of the hot spots, numbers one and three, are connected. Hot spot number one is connected to a framelet with the connection metapattern and hot spot number three is connected to application code with the unification metapattern. Hot spot number two is not connected.

The method `templateMethod1` in the class `FrameTemplate` has one reference to method `hookMethod1` in the class `FrameletClass1` and one reference to method `hookMethod1` in the class `FrameletClass2`. The method `hookMethod1` of the class `FrameletClass1` is not tested whereas the method `hookMethod1` of the class `FrameletClass2` is tested, so the hook method coverage for `hookMethod1` in `FrameletClass1` is 0 and the hook method coverage for `hookMethod1` in `FrameletClass2` is 1. The hook class coverage of `FrameletClass1` is 0, because its only hook method is not tested. The hook class coverage of `FrameletClass2` is 1, because its only hook method is tested. In this case, the reference from `templateMethod1` to untested `hookMethod1` can not be tested, since there is no test case that executes any of its code. However, the reference from `templateMethod1` to `hookMethod2` is tested. This yields the template method coverage of $1/2$ for `templateMethod1`, since one of its two hook method references are tested. Because there are no other template methods in the class `FrameTemplate1`, its template class coverage is also $1/2$.

The method `templateMethod3` references only the tested hook method `hook-`

`Method3`, whose hook method coverage is 1. Four of the five hook method references of the method `templateMethod3` are tested, so the template method coverage of the method is $4/5$ as is the template class coverage of the class `FrameTemplateHook1`. The hook class coverage of the class `ApplicationCodeClass1` is 1, because its only hook method is tested.

Figure 15: Hot spots of a framework and its hook method, hook class, template method and template class coverages.

Figure 16 shows an example of a framelet. The framelet uses the unification metapattern and has one unified template and hook class, `FrameletTemplateHook1`. The class contains two template methods and three abstract hook methods. Two of the hook methods are instantiated by the class `ApplicationCodeClass1` that extends the framelet.

The method `templateMethod1` has three references to the partially tested hook method `hookMethod2`. The hook method `hookMethod2` is partially tested, be-

cause its hook method coverage is $1/3$. Two of the three hook method references from `templateMethod1` to `hookMethod2` are tested and is one not tested, so the template method coverage for `templateMethod1` is $2/3$. The method `templateMethod2` has one reference to the hook method `hookMethod1` which is not tested, so the hook method coverage for the method `hookMethod1` is 0 and the template method coverage for the method `templateMethod2` is also 0.

These coverages yield the template class coverage of $1/2$ to the class `FrameletTemplateHook1`, because two of its four hook method references are tested. The hook class coverage of the class `ApplicationCodeClass1` is $1/6$, since the hook method coverage for the method `hookMethod1` is 0 and the hook method coverage for the method `hookMethod2` is $1/3$.
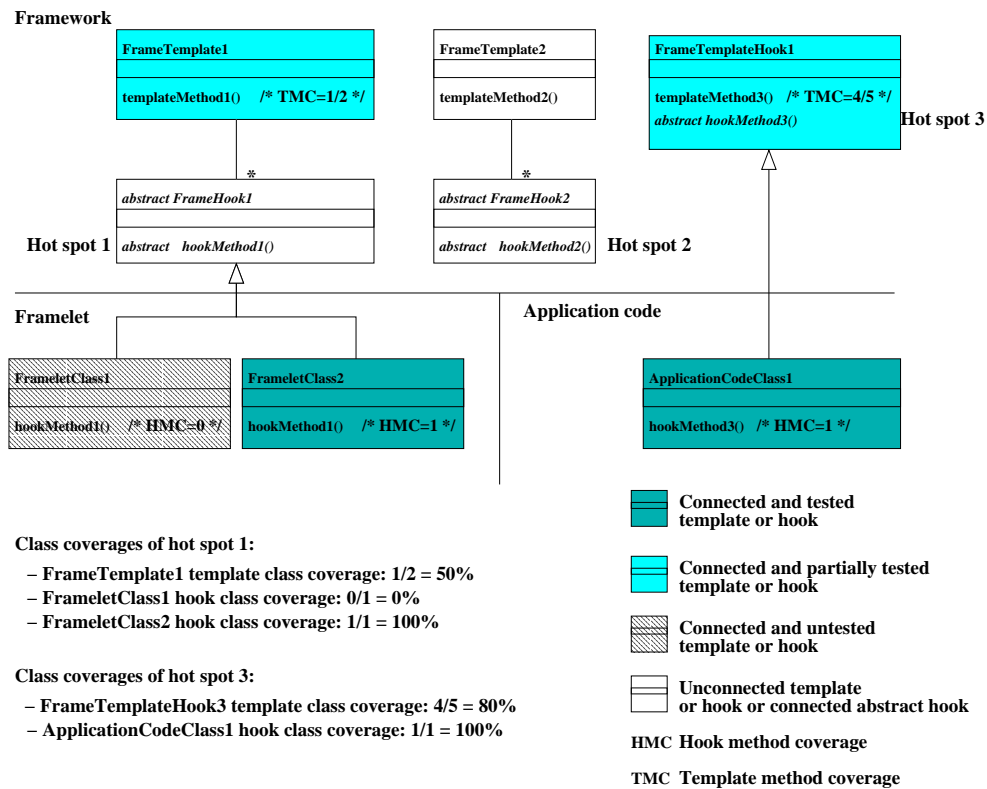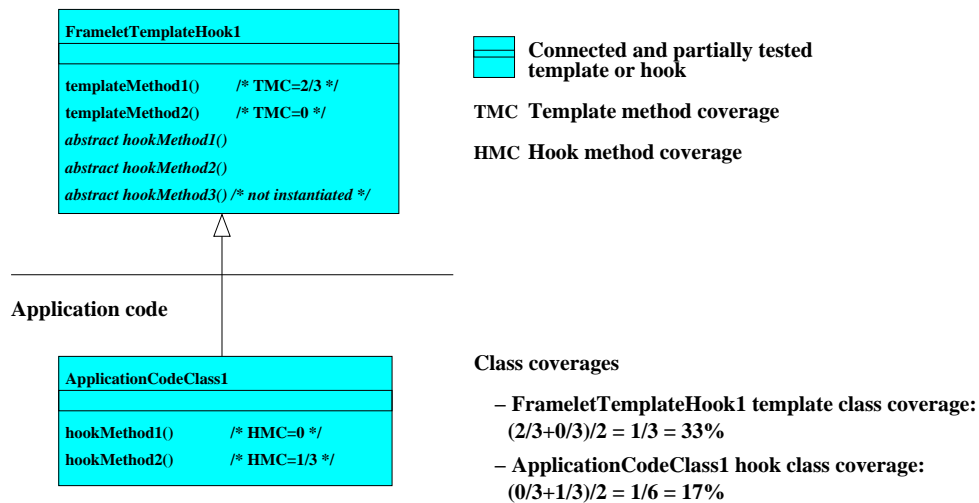

Figure 16: Hot spots of a framelet and its hook method, hook class, template method and template class coverages.

Finally, we define hook coverage and template coverage. We define the hook coverage HC of an application a as follows:

$$HC = \frac{HCC_1 + \ldots + HCC_n}{n},$$

where $HCC_1, \ldots, HCC_n$ are the hook class coverages of all the classes that con-

tain instantiated hook methods in the application a and n is the number of all the hook classes that contain instantiated hook methods in the application a. The hook coverage can also be used to measure the coverage of a single framework or framelet. In this case, $HCC_1, \ldots, HCC_n$ and n are counted from a single framework or framelet instead of the entire application.

We define the template coverage TC of an application a as follows:

$$TC = \frac{TCC_1 + \ldots + TCC_n}{n},$$

where $TCC_1, \ldots, TCC_n$ are the template class coverages of all the template classes of the application a and n is the number of all the template classes in the application a. The template coverage can also be used to measure the coverage of a single framework or framelet if $TCC_1, \ldots, TCC_n$ and n are counted from a single framework or framelet instead of the entire application.

The hook and template coverages are illustrated in Figure 17. The figure shows an application that uses the framework shown in Figure 15, the framelet shown in Figure 16 and two application code elements. Hook and template coverages are shown for the framework, the framelet and the entire application. The hook coverage of the framework is $2/3$, because the hook class coverages of the hook classes in the framework are 0, 1 and 1. The template coverage of the framework is $13/30$, because the template class coverages of the template classes in the framework are 0, $1/2$ and $4/5$. The hook coverage of the framelet is $1/6$ and the template coverage is $1/2$, because the framelets has only one hook and one template class and their coverages are $1/6$ and $1/2$. These coverages yield the total hook coverage of $5/12$ (42%) and template coverage of $7/15$ (47%) for the entire application.

Application:
  – hook coverage:
    (2/3+1/6)/2 = 5/12 = 42%
  – template coverage:
    (13/30+1/2)/2 = 7/15 = 47%

Framework:
  – hook coverage:
    (0+1+1)/3 = 2/3 = 67%
  – template coverage:
    (0+1/2+4/5)/3 = 13/30 = 43%

Framework

Framelet:
  – hook coverage:
    1/6 = 17%
  – template coverage:
    1/2 = 50%

Framelet

Application code

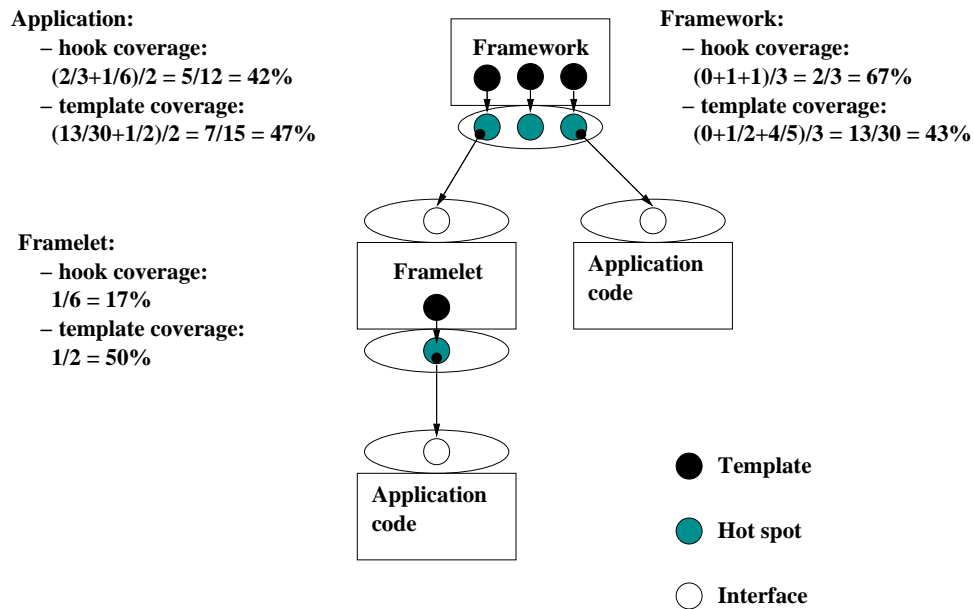**Application code**

● Template

● Hot spot

○ Interface

Figure 17: An application using the framework shown in Figure 15, the framelet shown in Figure 16 and two application code elements and their hook and template coverages.

4.4 Automated testing and tool support for product lines

In the product line approach, tool support is even more important than in traditional object-oriented testing. This is because the scale of a product line containing several applications derived from the same architecture is greater than the scale of a single application. For example, there is much more code and related test assets to manage. It is also possible that an organization has multiple product lines which means even more problems of scale. In the product line approach, tool support can be used to automate the generation and usage of test assets. For example, test execution and test analysis should be automated. Moreover, regression testing should also be automated, since test cases related to the core of the product line have to be repeated every time a new product is derived from the core [Nor01].

Figure 11 in Chapter 3.4 shows different kinds of testing tools that can be used at different levels of the V-model. In product line testing, all of these tools can and should be be applied. Compared to traditional testing, the testing management tools are of more importance in the product line approach than in testing of a single application. This is because of additional management overhead due to complexity of a product line. Also, an asset repository requires additional tool support in the product line approach. If the repository is separated from application code, a DBMS-based tool support is required. Otherwise, if the repository is integrated, extraction tools for integrated assets are needed to build the repository when necessary [HoW02, McG01].

The main problem concerning product line testing automation is that most of the tools available do not support the product line approach. Usually this kind of tools can also be used with product lines, but they are applicable only to low-level testing, like unit testing. For example, the JUnit, JTest and Roast Framework tools mentioned in Chapter 3.4 are unit level tools that can be used at least partially. Furthermore, management tools for testing usually assume that a single application is tested and therefore do not support reuse of testing assets that is essential for the product line approach. However, it is already possible to reuse testing assets in small scale, for example, with regression testing tools that automate the execution of existing test cases [FeG99] such as the *WinRunner* tool [Mer03].

A natural approach for tool support in product line testing would be an environment that integrates different types of tools shown in Figure 11. The integrated environment should have a repository for reusable test assets and provide support for automatic and manual generation, execution and analysis of test suites. Unfortunaly, such environments do not yet exist [TPK02]. However, a design of a prototype of a testing environment for product lines is discussed in Section 5.

4.5   State-of-the-art of product line testing

As discussed in the previous chapters, the managerial and business views of product lines and their testing have been under extensive research. Testing related results include, for example, process frameworks for the product line approach. Frameworks by the ESAPS and CAFÉ projects [Lin02a, Lin02b], SEI [Nor01], and ideas of reusable testing assets and their management [McG01] are examples of the work in progress.

The difficulty with process frameworks is that none of them is as widely accepted as a standard reference model for product line testing like the V-model is with respect to testing of a single application. Furthermore, the presented frameworks do not fully integrate the testing levels of the V-model into their development phases although testing is supposed to be performed according to these levels. Another problem with process frameworks is that they have not been compared with each other. Thus, there is no indication of how they relate to each other, and whether the different frameworks are better suited for different application domains.

The main problem in the current product line approach is that the work that has been done has focused mainly on acceptance and system levels of testing. However, since individual software components are extensively reused in the product line approach, their thorough low-level (for example, unit level) testing should also be assured. In other words, current research and practise focuses mainly on the high-level view of product line testing. The low-level view of the product line approach, where the actual testing is done, has received very little attention. It is assumed that traditional object-oriented testing methods can be used in the product line testing process without change. This assumption leaves many important questions open. For example, it is not clear which of the object-oriented testing

methods should be used and how these methods can be scaled to the product line level. Furthermore, it is unclear if there is a need for new, product line specific, testing methods.

The focus of current work on product line testing related to the standard V-model is illustrated in Figure 18. The work is focused on acceptance and system levels and related requirements specification and architectural design levels. Existing product line specific testing methods have been developed mainly for these levels. Integration and unit testing levels have got very little attention and product line specific testing methods for these levels are basically non-existent. Because of this, the existing high-level methods are often not used and product line applications are tested independently as single applications using traditional testing methods. This means, for example, that reusable testing assets are not used as effectively as would be possible, and that the entire testing process is repeated for every application derived from a product line.
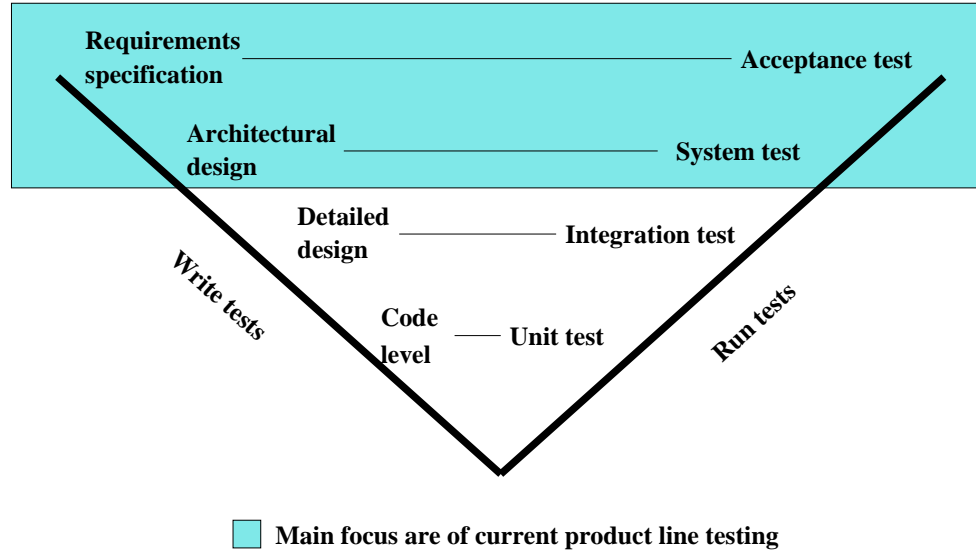
Figure 18: The focus of current work on product line testing related to the standard V-model.

In the framework-based approach to product lines, for example, the application framework is the core of a product line and should be well-tested before any applications are derived from it [TPK02]. However, framework-based product line applications are in practice often tested without using any information about the product line which they are derived from. An example of this that – unfortunately – characterizes the state-of-the-art of the product line testing today is the following comment regarding development of mobile browsers in a product line at Nokia:

> *"Moreover, testing a product line is more complex than testing a single software product. We must test the product line in its various configurations, which easily multiplies the number of test cases. To manage this complexity, we system tested individual product releases instead of testing the whole product line. This kept testing simple and guaranteed the quality of product releases. However, we did not build a pool of tested core assets ready for integration, which probably increased the need to test each product release."*
> [Jaa02]

Product line testing also lacks specific metrics and automatic tools. There is a need for product line specific testing measures of coverage in addition to the traditional ones. There is also a need for product line specific testing tools that should, on one hand, help to manage reusable testing assets and, on the other hand, automate the execution of tests and analysis of their results. A feasible approach for this would be to use a product line specific, integrated testing environment.

5 RITA: Framework integration and testing application for product lines

As discussed in the previous section, work has been done to describe the product line testing process based on reusable assets. Also, existing process frameworks for product line engineering have a very high-level view to product lines. However, this business-related view does not define how the actual testing should be done. Because of this, product line specific testing methods are basically non-existent. Plethora of object-oriented testing methods that can be used in the product line testing process do exist, but there are basically no results on how they should be scaled to the product line level.

In this section, building on the theory discussed in the previous sections, we introduce the RITA tool for product line testing developed in the CAFÉ project by the University of Helsinki [TPK02]. First, motivation for the tool is discussed and the main ideas behind it are introduced in Chapter 5.1. Then, the specific features of the tool are discussed in Chapter 5.2. Chapter 5.3 describes the views to an application under testing provided by RITA. Chapter 5.4 describe the relation between RITA and product line testing, specifically the relation between RITA and the standard V-model and the relation between RITA and the product line testing process. Finally, current state and future work regarding the tool and the theory it is based on are presented in Chapter 5.5.

5.1 Motivation and introduction

The *RITA tool* for product line testing is designed to tackle the key question of how the traditional object-oriented testing methods should be used when product lines based on object-oriented application frameworks are tested. The tool can

also be used for evaluating if new product line specific testing thory and methodology are needed [TPK02]. RITA provides an environment specifically designed for testing of framework- and framelet-based product lines. It includes services for interface class identification, code profiling, coverage criteria analysis, driver and stub generation, test management and statistics. The tool can be used, on one hand, to apply the existing object-oriented testing methods in the product line context and, on the other hand, to explore new, framework-based product line specific testing methods, such as hook and template coverages introduced in Chapter 4.3.

An overview of the tool is shown in Figure 19. The figure shows inputs and outputs of the tool. The main input for RITA is the application code that is composed of the used framework, framelets and application code. In addition, the tool needs template and hook class information of the application. Testing related inputs are test materials and drivers and stubs needed for testing of the possibly partial application. These can be either generated manually or by the RITA tool. Outputs of the tool include test results that contain information about passed and failed tests, statistics of the tests (for example, different coverages), new test cases generated by the tool (based, for example, on template and hook class interfaces) and new drivers and stubs that are generated by the tool if needed.

5.2 RITA features

The RITA integration and testing environment consists of testing management elements that offer services for white-box testing in framework-based product line environments. The elements are *template and hook class identifier*, *profiler*, *coverage analyzer*, *driver and stub generator*, *test environment* and *statistics generator* [TPK02].

A new application is first analyzed to identify template and hook classes. This in-
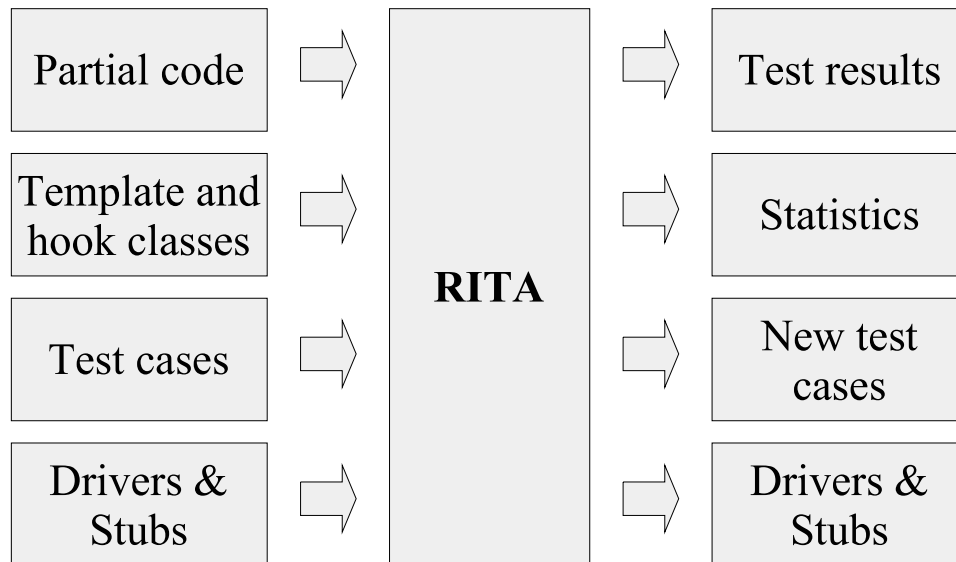
Figure 19: Overview of the RITA tool showing its inputs and outputs.

formation is used in testing and coverage analysis. External tools can be used to import this information to RITA. For example, if the framework of the application is generated with a framework editor, the hook and template class information may be imported from the editor. One such editor is the *FRED* tool [Vil01]. Another possibility is to import the information from a tool that can regognize the template and hook classes of the framework as patterns, such as the *MAISA* tool [Nen00]. If an external tool is not used, the template and hook identifier of RITA can be used. The template and hook class identifier first recognizes potential template and hook class candidates, and the end user has to choose the actual classes from a list of candidates. The template and hook class information is important, since it allows RITA to recognize interfaces between frameworks, framelets and application code.

When a newly added application is being tested, it is first run through a profiler. The profiler preprocesses application code such that RITA can gather data from actual code execution. This element allows later RITA elements to analyze the

executed code, and also to create graphical presentations of the code. In the first stage RITA profiler accepts Java code as input with all standard Java structures. Only exceptions are forbidden in the first version, since exception processing can easily violate the normal execution flow of the program. In the later versions also exception processing will be under research. Also, the RITA profiler will be defined such that it can be later expanded to programming languages other than Java.

The coverage analyzer is one of the main elements of RITA. It accepts a pro-filer generated file as input and generates white-box -based coverage information from the material. In the first prototype standard coverage criteria, such as code and branch coverage, are supported. In addition, hook and template coverages, as defined in Chapter 4.3, are supported.

From testing point of view, a designed product line is an incomplete program. The framework and framelets may be present while some or all of the applica-tion specific code is unavailable. Because of this, RITA offers services for auto-matic driver and stub generation. In the first prototype of RITA, drivers are based on template and hook class interfaces. RITA uses identified template and hook classes to generate drivers, and coverage information to hint for possible test ma-terials. RITA is also an integration tool, since it allows framelets and application-specific code to be tested separately from the framework.

In addition, RITA offers a complete test management environment. Executed tests are stored into a database and expected results may be compared to test results. Regression testing is also supported. From the tester's point of view this is probably the most important service of RITA although from the research point of view this element is not very interesting.

Finally, RITA will generate various reports and graphical presentations of the test process and tested software. The first prototype will include at least various coverage percentage based on executed tests, graphical coverage information of chosen classes and information about template and hook classes and their test states.

5.3 RITA views

The RITA tool provides four different views to the application under testing. The views are *framework view*, *template view*, *class view* and *method view*. Each view offers view-specific testing services for the application. Figure 20 shows an example application to be tested with RITA. The application is a real-time database management system application derived from a framework-based DBMS product line. The framework of the example provides database services and offers three hot spots for application specific expansions. Two of the hot spots are extended, both with framelets. Framelets provide real-time capabilities and disk management functions. The framelet providing real-time capabilities has three hot spots two of which are extended with application specific code. One extension provides services for scheduling and the other provides service for transactions. This example is used in the following chapters to describe the four views of the RITA tool.

The framework view is the product line level view to the application under test. The view offers services for testing between framework elements (the framework and framelets and application code elements connected to the framework). The framework view includes, for example, black-box testing services and statistics of different coverages including hook and template coverages for the entire application and for a single framework or framelet. At this view frameworks, framelets
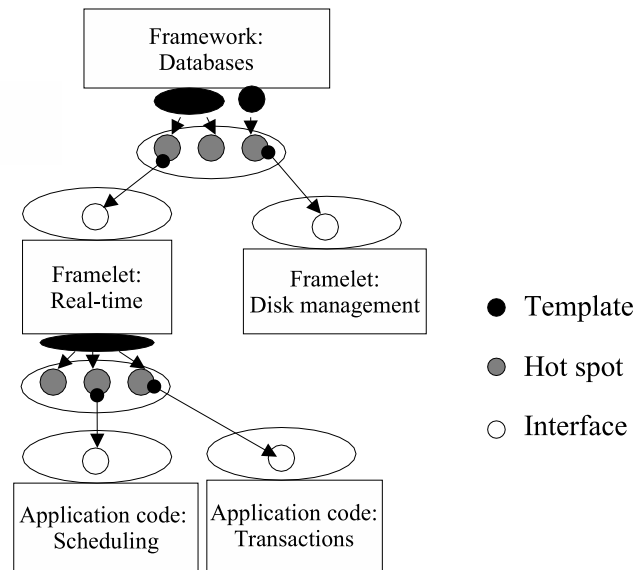
Figure 20: An example of a real-time DBMS application of a framework-based DBMS product line.

and application code elements and their connections are shown. Unconnected hot spots are also visible, but the class hierarchy of elements is not shown.

The framework view of the application in Figure 20 is shown in Figure 21. The view shows the framework, the two framelets and the two application code elements of the application. When one of these elements is clicked, the corresponding template view is opened. The view also shows the two unconnected hot spots of the application.

The template view shows a UML class diagram of a framework, a framelet or an application code element. The classes also include template and hook classes of the element, so interfaces between the element in question and other framework elements is visible at this level. The template view offers services for interface testing. This level also includes hook and template coverages for hook and template classes and methods. The exact services of this view are under design and will not be implemented in the first prototypes of RITA, but they can include, for

Figure 21: The framework view of the example application shown in Figure 20.

example automatic test case generation for interface testing between framework elements. Also, automatic driver and stub generation for framework element interfaces is possible at this level.

The template view of the framelet providing real-time capabilities, `Framelet 1` in Figure 21, is illustrated in Figure 22. The view shows the UML class diagram of the framelet. In the framework area, the abstract hook class `RealTimeService` is shown. Its counterpart, the instantiated hook class `RealTimeService` that is used to extend the framework is shown in the framelet area of the view. If the framework area surrounding the class `RealTimeService` is clicked, the template view of the framework is shown. If the class itself is clicked, the corresponding class view is shown.

In addition to the instantiated hook class, the framelet area shows the other classes of the framelet. They include classes that implement the functionality of the framework (`RTDispatcher`, `RTQueue` and `RTSimpleStateMachine`), and also template and hook classes that provide the three hot spots of the framelet. The first hot spot is provided by classes `RTParellelScheduler` and `RTSimpleScheduler`, the second by the class `RTSerialScheduler` and the third by classes `RTTransactionHandler` and `RTTransaction`. The first and the third hot spots use the connection metapattern and the second uses the unification metapattern. The hot spot that uses the classes `RTParellelScheduler` and `RTSimpleScheduler` is not implemented. When a class in the framelet area is clicked, the corresponding class view is shown.
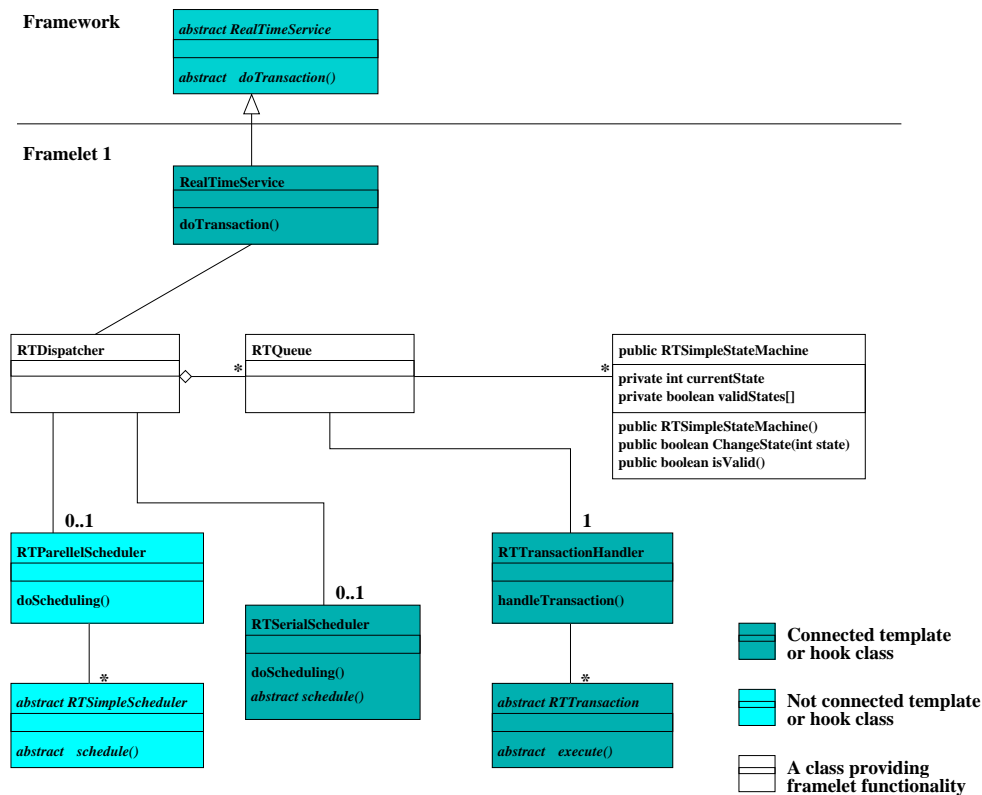

Figure 22: The template view of `Framelet 1` in Figure 21.

The class view offers two presentations for a class. The first shows the method references of each method of the class and the other shows the code of the class. The class view offers services for class testing. It includes standard object-oriented testing services. The services include, for example, interface testing of a class, testing of a class state and method collaboration testing. Driver and stub generation is supported by providing places where manually generated drivers and stubs can be plugged in. The exact services of this view are under design and will not be implemented in the first prototypes of RITA.

The class view of the class `RTSimpleStateMachine` in Figure 22 is shown in Figure 23. The class is a slightly modified version of the class used earlier (see Figure 7). Figure 23 shows both presentations of the class view: the method reference presentation of the class is shown on the left and the code presentation on the right. In this case, the class contains only three methods. The constructor of the class, `RTSimpleStateMachine`, has no references to other methods. Method `changeState`, however, uses the method `isValid`. The methods of this class do not contain method calls to other classes. Such references are shown in the method reference presentation so that the arrow from the calling method is connected to the class which holds the called method. When a method is clicked in the method reference presentation or in the code presentation, the corresponding method view is shown.

The method view offers services for unit testing at the method level. This view includes various coverage criteria (for example, code-based, state-based and constraint-based coverages described in Chapter 3.3) and a list of recognized indepented paths through a method. Also this view provides places for manually generated driver and stubs.

The view offers two presentations for a method. The first is a standard flowchart view to the method and the other is the code of the method which could be used,

```
1:  public class RTSimpleStateMachine {

2:      private int currentState;
3:      private boolean validStates[];

    ...

        // constructor
4:      public RTSimpleStateMachine() {
5:          currentState = 0;
6:          validStates = {true, true, false};
7:      }

    ...

        // accessors
8:      public boolean changeState(int state) {
9:          boolean changeOk = true;

10:         if (this.isValid(state) == true)
11:             currentState = state;
12:         else
13:             changeOk = false;

14:         return changeOk;
15:     }

    ...

16:     public boolean isValid(int state) {
17:         if (this.validStates[state] == true)
18:             return true;
19:         else
20:             return false;
21:     }
22: }
```

┌─────────────────────────────────────┐
│ public RTSimpleStateMachine()   │
└─────────────────────────────────────┘

┌─────────────────────────────────────┐
│ public boolean changeState(int state)│
└─────────────────────────────────────┘
                  │
                  ▼
┌─────────────────────────────────────┐
│ public boolean isValid(int state)   │
└─────────────────────────────────────┘

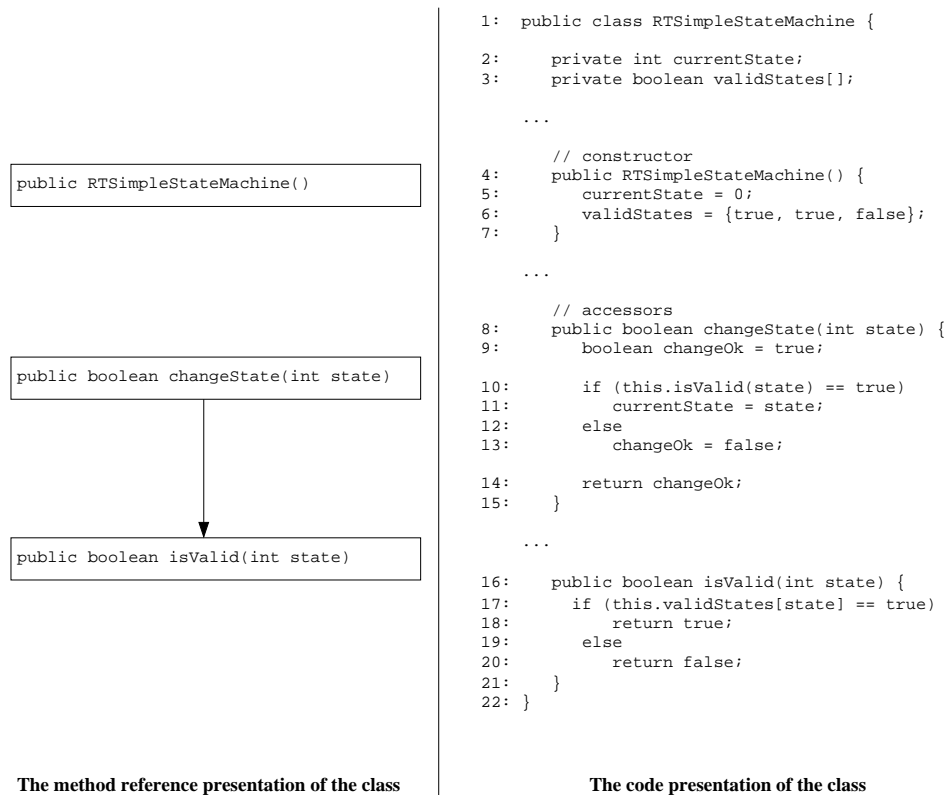**The method reference presentation of the class**   **The code presentation of the class**

Figure 23: The class view of `RTSimpleStateMachine` in Figure 22.

for example, in illustrating the statement coverage in addition to showing the actual method code. The method view of the method `changeState` in Figure 23 is shown in Figure 24. The figure shows the flowchart of the method on the left and the code on the right.

5.4 Product line testing with RITA

The RITA tool is designed to be a complete testing environment. As such it can provide services to all levels of the standard V-model. In this sense, RITA can bee seen as an environment where all the areas of tool support for testing illustrated in Figure 11 can be supported. However, all the designed features are not implemented in the first version of RITA. Also, the focus of the tool is to support low-level white-box testing of framework-based product lines, so the higher

The flowchart presentation of the method

```
1: public boolean changeState(int state)
2: boolean changeOk = true;
3: if (this.isValid(state) == true)
4: currentState = state;
5: changeOk = false;
6: return changeOk;
```

```
public boolean changeState(int state) {
    boolean changeOk = true;

    if (this.isValid(state) == true)
        currentState = state;
    else
        changeOk = false;

    return changeOk;
}
```
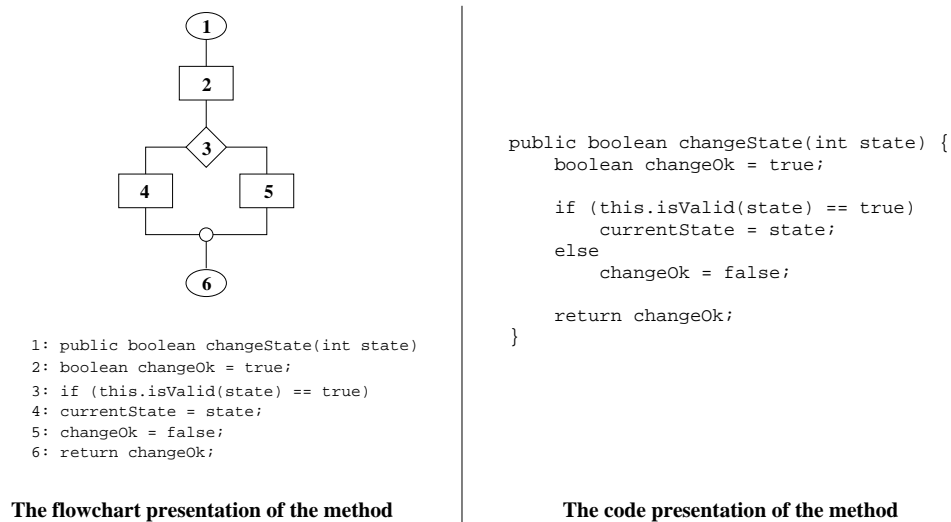
The code presentation of the method

Figure 24: The method view of the method `changeState` in Figure 23.

levels of the V-model are not supported in the first version of the tool.

The current focus of RITA in relation to the standard V-model is illustrated in Figure 25. The tool is focused on unit and integration testing of framework-based product lines. Unit and integration testing are supported by the method, class and template views of RITA. Unit tests can be executed and analyzed via method and class views. Similarly, integration tests can be executed and analyzed via template view. RITA has also some test design and test management features, so code and detailed design levels of the V-model are supported, but they are not main focus areas of the tool. System and acceptance testing nor architectural level and requirements specification of the V-model are not directly supported, but the tool is designed so that it is possible to manage, execute and analyze system tests via framework view.

Another way of describing how RITA can be used in product line testing is to relate it with a generic software testing process presented in Figure 26. The figure also shows how RITA relates to this process, which is originally developed for testing of single applications, but can also be used as a basis for a product line
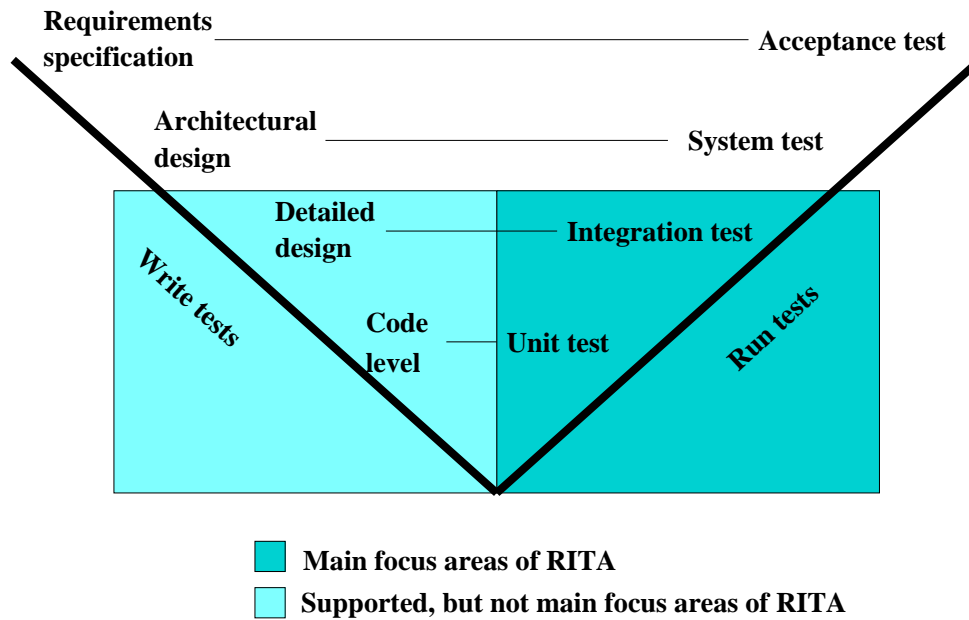
Figure 25: Focus areas of RITA in relation to the standard V-model.

testing process. In this process, requirements engineering produces requirements that are used in test planning to generate a test plan. The test plan is used when test cases are designed in test case specification. Then, test cases are scripted so that they are in executable form. Executable test cases are run, which produces test results. The results are evaluated and a test report is produced. Test management is needed in every phase of the process [FeG99, McS01].

RITA is focused on test execution and analysis, so RITA supports test scripting, test execution and test evaluation. Related test management activities are also supported. Furthermore, RITA can be used in test case specification, because new test cases can be generated with the tool automatically and manually. However, this is not the main focus of RITA. The current design of RITA does not support requirements engineering or test planning.

Based on the generic testing process in Figure 26, the testing process supported by RITA is illustrated in Figure 27. The process requires a test plan based on requirements as input, since RITA does not support test planning or requirements
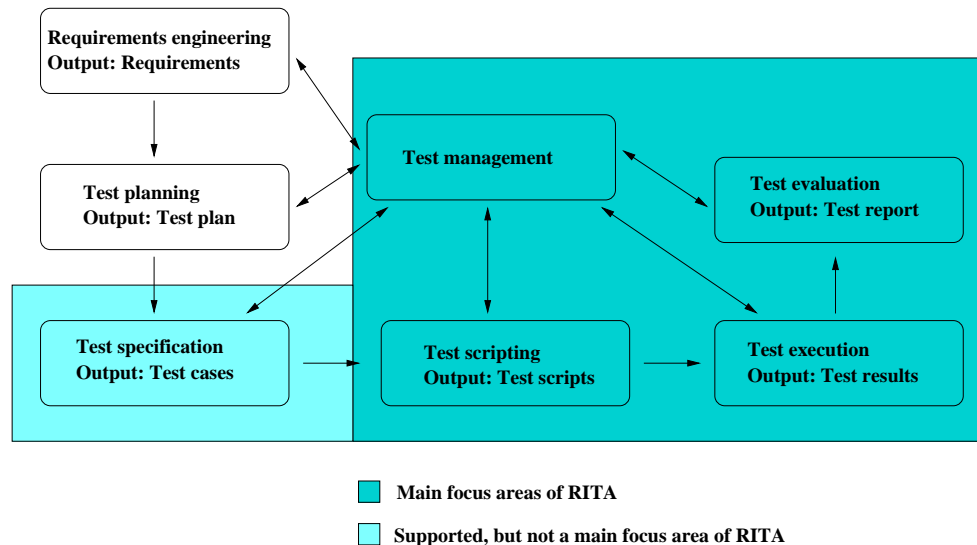
Figure 26: Focus areas of RITA in relation to a generic software testing process.

specification. Test cases are generated mostly manually, but the tool can generate additional test cases automatically based on, for example, hook and template class information. After test cases are generated and selected for execution, the tool scripts and executes the tests. After tests are run, results can be evaluated based on the test report generated by the tool. RITA also manages testing assets thoughout the process, for example, by maintaining a test asset repository.

The testing process supported by the RITA tool is iterative. Based on the results of executed tests, new test cases can be generated and tests scripts may be updated. The existing tests can also be repeated without change, for example, when regression testing is performed. When new test cases are generated, either manually or automatically, the process is repeated for them. It is also possible that test scrips need to be changed, for example, because interfaces of the application under test have been changed.

The testing process illustrated in Figure 27 can be used in testing of a single application. In this case, the test plan is designed specifically for the application and the testing process is iterated according to its lifecycle. However, the RITA tool is
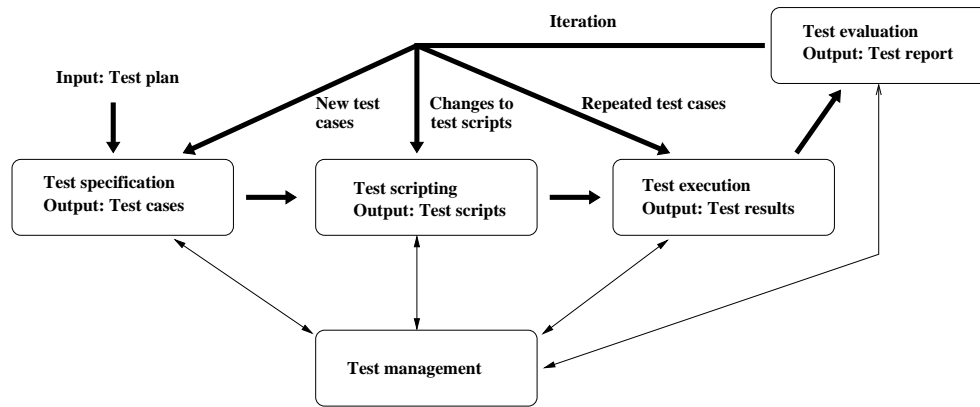
Figure 27: Testing process supported by RITA.

focused on testing of product lines instead of testing of single applications. When product lines are tested, the test plan covers the entire product line. It specifies how the framework of the product line is tested before applications are derived from it. The test plan also specifies how the derived applications are tested. In this case, the testing process is iterated throughout the lifecycle of the product line. In this way, actual product line testing can be performed and existing test assets can be effectively reused.

5.5   Future work

We are currently implementing the first version of the RITA tool. The first prototype is expected at the end of the second quarter of year 2003. This prototype will be used as a testbench for research of existing and new testing methods for framework-based product lines. The tool can also be used to evaluate the feasibility of the approach to product line testing presented in this thesis. At this point it seems that there is a clear need for both the tool itself and for new testing theory that will bind the high-level product line testing process frameworks to the low-lewel testing methodology in the product line context.

The first prototype of the tool will include the profiler, the coverage analyzer and the test environment. In other words, the prototype can load and profile an application, execute test cases and analyse the results in the form of hook, template and traditional coverages. However, the first prototype does not include automatic template and hook class identifier, driver and stub generator or statistic generator. This means that template and hook classes have to be identified with an external tool or manually, and incompleted parts of the application cannot be fully tested with the prototype. Also, the statistic generator and the database-based test asset repository are missing from the prototype. The prototype supports the framework, class and method views, but the template view is not fully implemented in the first version.

In the future, the implementation of the tool will continue, and for example, the missing parts will be implemented as needed. The RITA tool is also expected to evolve into a useful environment for product line testing that can also be used in practical software engineering projects in addition to its research use. However, to be useful for real product line projects, the environment must contain all the features discussed in Chapter 5.2. This will take time and resources and require co-operation with industrial partners. Also, thorough evaluation of the tool and the theory behind it are needed. The work with these issues will be started in the follow-up projects of CAFÉ.

6 Conclusions

Software product lines have recently received attention in the research community, but especially in industry. Instead of creating software from scratch for each product, many companies have focused on the commonalities between their different products and started to capture those in product line architectures. In the

research area, case studies of the product line approach have been made. In addition to the case studies, a few larger projects have been launched to study the use of product lines in software development.

A natural way to implement a software product line is an object-oriented application framework. A framework can be seen as a partial design and implementation for an application in a given domain. Therefore, a framework is an incomplete system, a set of objects that captures the special expertise in some application domain to a reusable form. Frameworks are feasible to use, because they can provide increased reusability of software components and reduced time to market for applications.

It is expected that product lines will soon become a dominating software production paradigm. Product flexibility is currently a very important factor in the software market and, with product lines, the promise of tailor-made systems built specifically for the needs of partical customers or customer groups can be fulfilled. Especially for large companies, product lines offer an efficient way to exploit the commonalities shared by different products to achieve economies of production.

In the product line approach, as in all software engineering, testing is essential. The framework of a product family must be reliable and well-tested, because all applications of the family share the common parts implemented in the framework. However, also the application specific parts as well as the application as a whole have to be tested thoroughly. The product line approach requires a carefully planned testing process that can be easily adapted and used for product families in various application domains.

The state-of-the-art of product line testing is immature, so there is a clear need for a more mature testing methodology. Most of the research so far has concen-

trated on the product line testing process and on the assets that can be reused thoughout the process. Regarding these issues, product line process frameworks and ideas of asset repositories of product lines have been formulated. However, there is a gap between the product line testing process and the practical testing methods, because it is not clear which object-oriented testing methods can be effectively used in this particular context. Testing also lacks necessary tool support and automation that are essential in the product line approach.

However, the work to tackle the problems encountered is under way. For example, product line testing process is under extensive research and new testing methodology is being developed. For example, the CAFÉ project and its followup projects in Europe and SEI in the USA are currently studying new testing theory and deriving practical methodology and tools to be used with product lines.

Examples of the work done in the CAFÉ project are the definition of hook and template coverages and the development of the RITA tool for testing of framework-based product lines. RITA provides a testbench for traditional as well as newly developed testing methods for product lines. The tool includes services for interface class identification, code profiling, coverage criteria analysis, driver and stub generation, test management and statistics. It can be used to apply the existing testing methods in the product line context and to explore new, framework-based product line specific testing methods including hook and template coverages. Hopefully, in the future, the tool can also be used as a complete testing environment for product line testing.

Acknowledgements

bers of the RITA project for their advice and comments throughout the writing process. The author would also like to thank the CAFÉ project members from Nokia Research Center for their hands-on experiences about software product lines and information about the relevant material for this work.

References

AlA01   Alkadi, I., Alkadi, G., Algorithms that Compute Test Drivers in Object-Oriented Testing. *Proceedings of the IEEE Aerospace Conference*, Big Sky, Montana, USA, March 2001, Volume 1, 115–119.

Ard00   Ardis, M., et al., Software Product Lines: A Case Study. *Software – Practice and Experience*, Volume 30, Number 7, June 2000, 825–847.

Bal00   Ball, T., et al., State Generation and Automated Class Testing. *Software Testing, Verification and Realibility*, Volume 10, Number 3, September 2000, 147–170.

Bei90   Beizer, B., *Software Testing Techniques*. Van Nostrand Reinhold, Second Edition, 1990.

Bin94   Binder, R., Design for Testability in Object-Oriented Systems. *Communications of the ACM*, Volume 37, Number 9, September 1994, 87–101.

BOP00   Buy, U., Orso, A., Pezzé, M., Automated Testing of Classes. *Proceedings of the International Symposium on Software Testing and Analysis (IS-STA'00)*, Portland, Oregon, USA, August 2000. Software Engineering Notes, Volume 25, Number 5, 2000, 39–48.

Bos99   Bosch, J., Product Line Architectures in Industry: A Case Study. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, California, USA, May 1999, 544-554.

Bos00   Bosch, J., et al., Object-Oriented Framework-based Software Development: Problems and Experiences. *ACM Computing Surveys*, Volume 32, Number 1 (electronic supplement), March 2000, 3–7.

Coh02   Cohen, S., *Product Line State of the Practice Report*. Technical Note CMU/SEI-2002-TN-017, Carnegie Mellon University, Software Engineering Institute, October 2002.

CzK00   Czarnecki, K., Eisenecker U., *Generative Programming : Methods, Tools, and Applications*. Addison-Wesley, 2000.

DHS02   Daley, N., Hoffman, D., Strooper, P., A Framework for Table Driven Testing of Java Classes. *Software – Practice and Experience*, Volume 32, Number 5, April 2002, 465–493.

FeG99   Fewster, M., Graham, D., *Software Test Automation – Effective Use of Test Execution Tools*. Addison-Wesley, 1999.

FHB00   Fayad, M., Hamu, D., Brugali, D., Enterprise Frameworks Characteristics, Criteria, and Challenges. *Communications of the ACM*, Volume 43, Number 10, October 2000, 39–46.

Fow99   Fowler, M., *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

FSJ99   Fayad, M., Schmidt, D., Johnson, R., *Building Application Frameworks*. Wiley and Sons, 1999.

Gam94   Gamma, E., et al., *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

GiG93   Gilb, T., Graham, D., *Software Inspection*. Addison-Wesley, 1993.

GuB01   van Gurp, J., Bosch, J., Design, Implementation and Evolution of Object-Oriented Frameworks: Concepts and Guidelines. *Software – Practice and Experience*, Volume 31, Number 3, March 2001, 277–300.

HMF92   Harrold, M., McGregor, J., Fitzpatrick, K., Incremental Testing of Object-Oriented Class Structures. *Proceedings of the 14th International Conference on Software Engeneering (ICSE'92)*, Melbourne, Australia, May 1992, 68–80.

HoW02   Hoffman, D., Wilkin, S., JUnit Extensions for Documentation and Inheritance. *Proceedings of the 20th Pacific Northwest Software Quality Conference (PNSQC'02)*, Portland, Oregon, USA, October 2002.

Jaa02   Jaaksi, A., Developing Mobile Browsers in a Product Line. *IEEE Software*, Volume 19, Number 4, July/August 2002, 73–80.

JoS02   Jones, L., Soule, A., *Software Process Improvement and Product Line Practice: CMMI and the Framework for Software Product Line Practice*. Technical Note CMU/SEI-2002-TN-012, Carnegie Mellon University, Software Engineering Institute, July 2002.

JRL00   Jazayeri, M., Ran, A., van der Linden, F. (eds.), *Software Architectures for Product Families: Principles and Practice*. Addison-Wesley, 2000.

JSL02   Jeon, T., Seung, H., Lee, S., Embedding Built-in Tests in Hot Spots of an Object-Oriented Framework. *ACM SIGPLAN Notices*, Volume 37, Number 8, August 2002, 25–34.

Lab00   Labiche, Y., et al., Testing Levels for Object-Oriented Software. *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, Limerick, Ireland, June 2000, 136–145.

Lin02a   van der Linden, F., Software Product Families in Europe: The Esaps & Café Projects. *IEEE Software*, Volume 19, Number 4, July/August 2002, 41–49.

Lin02b   van der Linden, F., ESAPS-CAFÉ Inputs. *Proceedings of the 3rd ITEA Symposium*, Amsterdam, Netherlands, October 2002, URL: http://www.itea-office.org/symposium/ [March 13, 2002].

Mau93   Maunder, C., A Universal Framework for Managed Built-in Test. *Proceedings of the International Test Conference*, Altoona, Pennsylvania, USA, October 1993, 21–29.

McG01   McGregor, J., *Testing a Software Product Line*. Technical Report CMU/SEI-2001-TR-022, Carnegie Mellon University, Software Engineering Institute, December 2001.

McK94   McGregor, J., Korson, T., Integrated Object-Oriented Testing and Development Process. *Communications of the ACM*, Volume 37, Number 9, September 1994, 59–77.

McS01   McGregor, J., Sykes, D., *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001.

Mer03   Mercury Interactive, *WinRunner Data Sheet*. URL: http://www-svca.mercuryinteractive.com/products/winrunner/whitepapers [March 13, 2003].

Nen00   Nenonen, L., et al., Measuring Object-Oriented Software Architectures from UML Diagrams. *Proceedings of 4th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Sophia Antipolis, France, June 2000, 87-100.

Nor01   Northorp, L. (director), *A Framework for Software Product Line Practice – Version 3.0*. Software Engineering Institute, Carnegie Mellon University, 2001, URL: http://www.sei.emu.edu/plp/framework.html [March 13, 2003].

Par03   Parasoft   Corporation,   *Automating   and   Improving Java   Unit   Testing: Using   Jtest   with   JUnit.*   URL: http://www.parasoft.com/jsp/products/tech_papers.jsp?product= Jtest [March 13, 2003].

Pre95   Pree, W., *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

Pre97   Pressman, R., *Software Engineering – A Practitioner's Approach.* McGraw-Hill, Fourth Edition, 1997.

PrK99   Pree, W., Koskimies, K., Rearchitecting Legacy Systems – Concepts and Case Study. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA'99)*, San Antonio, Texas, USA, February 1999, 51–64.

Roy70   Royce, W., Managing the Development of Large Software Systems: Concepts and Techniques. *Proceedings of the IEEE Western Electronic Show and Convention (WESCON)*, Los Angeles, California, USA, August 1970, A/1–1–A/1–9.

SBF96   Sparks, S., Banner, K., Faris, C., Managing Object-Oriented Framework Reuse. *Computer*, Volume 29, Number 9, September 1996, 52–61.

TPK02   Taina, J., Paakki, J., Kauppinen, R., RITA - a fRamework Integration and Testing Application. *Proceedings of the Finnish Data Processing Week (FDPW'02)*, Petrozavodsk, Russia, July 2002, to appear.

Vil01   Viljamaa, A., *Pattern-Based Framework Annotation and Adaptation – A Systematic Approach.* Licentiate Thesis, Report C-2001-52, University of Helsinki, Department of Computer Science, 2001.

Wan00   Wang, Y., et al., On Built-in Test Reuse in Object-Oriented Frame-
         work Design. *ACM Computing Surveys*, Volume 32, Number 1 (elec-
         tronic supplement), March 2000, 7–12.