Hans-Gerhard Gross

Component-Based Software Testing with UML

Hans-Gerhard Gross

Component-Based Software Testing with UML

With 121 Figures and 24 Tables

Springer

Hans-Gerhard Gross
Fraunhofer Institute
for Experimental Software Engineering
Sauerwiesen 6
67661 Kaiserslautern
e-mail: hans-gerhard.gross@iese.fraunhofer.de

Library of Congress Control Number: 2004111059

ACM Computing Classification (1998): D.2.4, D.2.5, D.2.13

ISBN 3-540-20864-X Springer Berlin Heidelberg New York

Cover design: KünkelLopka, Heidelberg
Production: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Typesetting: by the author
Printed on acid-free paper 45/3142/YL - 5 4 3 2 1 0

To Marc, Heiko, Isabell,
and Oliver

Foreword

The promise of object-oriented development to enable the flexible composition and reuse of software artifacts finds an extended realization in component technologies such as CORBA components, EJB, and .Net. Although off-the-shelf components and component composition, configuration and deployment methods are nowadays available, an intrinsic problem of component-based software construction is not yet well addressed. How can software engineers ensure that a component used in a certain component assembly fits the requirements and the context? How can software engineers select a component from a component repository? How can software engineers check its correctness and performance in that component assembly? Only if the overall efforts of developing reusable components, their composition into new component assemblies and correctness checks thereof are made more efficient than developing everything from scratch will component-based software development become a new software development paradigm in industry.

Traditional testing methods where the test system is independent from the tested system do not offer good support for the evaluation of software components in new contexts when procuring off-the-shelf components or reusing in-house components. Although testing is the means to obtain objective quality metrics about components in their target component assembly, the separation of functional component and test component in traditional testing leaves the problem up to the component user to design and develop tests that assess the correctness and performance of a component in a new component assembly. However, the argument of reuse should not only apply to a functional component but also to its test component. The idea is as simple as it is compelling: why not bring test components up to the same level of reuse as the components they are supposed to evaluate?

Built-in contract-based tests as developed and put forward in this book advocate a tight coupling of components with their test components. Built-in tests provide basic test functionality to flexibly assess the behavior of a software component. In addition, they enable checks by the component itself of its environment, as inadequate contexts will typically make a software component fail and should be detected by the component itself.

Although built-in tests follow well-known principles of hardware development, where readily built-in validation infrastructure belongs to any high-quality system, software component testing imposes new challenges when designing and developing built-in component tests as well as when applying and using them. First of all, as components may bear complex internal data and behavior that might show nondeterministic reactions when driven via component interfaces only, components must be made observable and controllable by opening the access to component internals in order to gain unambiguous, well-focused and repeatable results. Built-in tests provide solutions here. Secondly, as components will be typically used in new, possibly unforeseen contexts, the component tests must be made flexibly adaptable and extendable so as to analyze new usage scenarios and the reactions of a component in new assemblies. While the client components can freely compose new tests by invoking the base test functionality offered via the component's testing interface, the component itself can analyze its requirements that are preconditions for its correct behavior in a given context. Thirdly, testing a component assembly is an ongoing activity when constructing component assemblies, but when putting this assembly into production no checks or only sanity checks should be kept in order to avoid unnecessary code. Component technologies such as built-in testing help here as well.

However, the solution of built-in contract-based testing would only be half of a solution if not accompanied by approaches for how to design and construct them. This book puts the ideas of built-in contract-based tests into a UML-based development process and discusses the various aspects and views of how to develop them. The powerful KobrA development method is used as a basis, both for the system and the test system development, which brings test development not only onto the same level as system development but enables also the exchange and reuse of system artifacts in test development, and vice versa. In addition, it enables the exchange of information and the support of interaction between system architects, designers and developers on the one hand, and test architects, designers and developers on the other hand.

Hans-Gerhard Gross and I were members of the MDTS project which focused on "Model-Based Development of Telecommunication Systems" in which one of the major subjects was the integrated development of a system and its test system. While the first approaches were developed in this project, this book nicely completes the ideas and discusses the various parts in great detail. The fundamental concept of built-in contract-based tests, although not new in itself, is thoroughly discussed in the context of software components. Although some questions such as the quality of tests remain open, the book advances the state of the art in component testing. Of particular interest to me is the use of the UML 2.0 Testing Profile in this book and its application and extension to the special case of built-in tests. This demonstrates not only the capabilities of UML-based test specification in a standardized way but also the flexibility of the testing profile to address real needs from testing software components.

This book is an exhaustive compendium for component-based software testing based on UML, and it provides good examples for applying the developed approach. As a reader, I enjoyed the detailed discussion arguing about business-, technical- and process-oriented aspects, and I would like to see follow-ups.

Berlin,
July 2004

Ina Schieferdecker

Preface

This book summarizes and consolidates my primary research interests over the course of the last four to six years, and its main subjects represent the outcome of three major research projects, Component+, MDTS, and Empress, in which I was involved during my work at the Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, Germany. The three projects were carried out almost in a sequence that represents the "natural evolutionary steps" in developing the technologies introduced in this book.

The primary testing technology described here, built-in testing, with its two main incarnations, built-in contract testing and built-in quality-of-service testing, were developed within the Component+ project (www.component-plus.org) which was funded under the 5th IST European Framework Program. These technologies are described mainly in Chaps. 4, 5, and 7. The project consortium comprised a balanced number of academic research institutes and industry partners who applied the built-in testing approaches in a number of real-world case studies. I did not use these case studies as examples in this book because none of them addresses all the issues that are discussed in the book. I found it easier to explain the issues using two examples only, a thought-out vending machine specification and a real system specification from the Fraunhofer IGD's Resource Information Network. The descriptions of the industrial case studies and their results can be obtained from the Component+ project Web site.

The second project, MDTS, which stands for Model-Driven Development of Telecom Systems (www.fokus.gmd.de/mdts), funded by the German National Department of Education and Research (BMBF), devised the model-based testing and test modeling approaches that are introduced in the book. The validation of these concepts was based on the Resource Information Network (RIN), an application that was developed by Fraunhofer IGD, Darmstadt, Germany. The technologies of this project are described mainly in Chaps. 3 and 5.

The third research project, the ITEA project Empress (www.empress-itea.org), also funded by the German government, dealt with software evolution for embedded real-time systems. Here, my primary focus was on the adoption of search-based software engineering technologies for dynamic timing analysis and validation of real-time software according to modern object-oriented and component-based development principles. This work represents a combination of the main subject from Component+ with the work that I carried out at the University of Glamorgan, Wales, UK, prior to my position at IESE. This more visionary subject is treated in Chap. 7.

Another major subject of the book, and I am going to start with this, is the overall unifying framework for all introduced technologies: the KobrA method (www.iese.fhg.de/Projects/Kobra_Method). This is a component-based and model-driven software development method that was devised with significant involvement by IESE in another project funded by the German government, which resulted in a book by Colin Atkinson, and others, with the title "Component-Based Product Line Engineering with UML;" its title is similar to that of this book, and this reflects the similarity of the context. The work described in this book can be seen as an extension of the KobrA method in the area of testing. All the principles introduced are fully in line with the KobrA method, and the projects that I have mentioned above were applying, at least partially, principles of the KobrA method. I give an overview on the most fundamental and important topics of the KobrA method, so that the book can be read more cohesively. A much more comprehensive treatment of the KobrA method can be found in the other book. I introduce the KobrA principles in Chap. 2 of this book.

This book is by no means an exhaustive treatment of the combination of the subjects component-based development, modeling, and testing. It only deals with a few facets of these subjects, and, as such, only reflects my personal opinions and experiences in these subjects. Each of them are fields of extensive research and development in their own right. Some readers will disagree with my opinions, some others will be disappointed with the choice of subjects that I have treated in the book. I have tried to include pointers, to my best knowledge, to other similiar, related subjects of interest throughout the text.

Acknowledgements

Books are not simply cast into form. Besides the author, there are usually many more people who contribute in terms of ideas and discussions, and in the form of technical as well as emotional support. I cannot list all who helped to sustain me over the course of writing this book, and I apologize for any possible omissions.

First of all, I would like to thank all the people in the Component+ project in which most of the results that eventually led to the production of the book were produced. These are Håkan Edler and Jonas Hörnstein (IVF, Mölndal, Sweden), Franck Barbier, Jean Michel Bruel, and Nicolas Belloir (University of Pau, France), John Kinghorn and Peter Lay (Philips, Southampton, UK), Graham King (Southampton Institute, UK), Jonathan Vincent (Bournemouth

University, UK), Stefano di Panfilis and Matteo Melideo (Engineering Informatica, Rome, Italy), Aracéli Gomez, José Maria Lázaro, and Joseba Iñaki Angulo Redondo (Labein, Bilbao, Spain), Atanas Manafov (ISOFT, Sofia, Bulgaria), Mariusz Momotko, Bartosz Nowicki, and Witold Staniszkis (RODAN, Warsaw, Poland), Enn Õunapuu (Technical University of Tallinn, Estonia), Ivan Stanev (University of Rousse, Bulgaria), Istvan Forgacs and Eva Takacs (4D Soft, Budapest, Hungary).

In addition, I would like to thank the people from the MDTS project for the outstanding cooperation over the course of nearly three years of working together: Marc Born, especially for the crash course on component platforms, Julia Reznik (Fraunhofer Fokus, Berlin), Pedro Sanchez (Fraunhofer IGD, Darmstadt), for the provision of the RIN case study including ongoing support, and, in particular, Ina Schieferdecker (Humboldt University, Berlin), for advice on model-based testing and the TTCN-3, and for writing the Foreword.

For the fruitful cooperation in the Empress project I would like to thank, in particular, the people with whom I have cooperated most intensively: Stefan van Baelen (KU Leuven, Belgium), Linde Loomans (Jabil, Hasselt, Belgium), and Ruurd Kuiper (Technical University of Eindhoven, Netherlands).

Very special thanks go to Colin Atkinson, a former colleague of mine at the Fraunhofer Institute, now affiliated with the University of Mannheim. I see him as my mentor and teacher in the area of component-based development and modeling. I would also like to thank my current colleagues at Fraunhofer IESE: Christian Bunse for his advice and Christian Peper for reviewing the UML diagrams. My former colleagues Nikolas Mayer and Javier Mauricio Paredes deserve special recognition for their commitment to implementing and assessing optimization-based timing analysis.

In addition, I would like to thank the Springer team, in particular Ralf Gerstner, for ongoing support in technical issues, and the reviewers and the copy editor who provided me with valuable feedback and helped to improve the book considerably.

Finally, I would also like to thank my friends and family, for their support and love throughout the time of writing, and Isabell Beutke, especially, for tolerating my occasional phases of social disability over a considerable period of time.

Neunkirchen, *Hans-Gerhard Gross*
July 2004

Contents

1 Introduction

Component-based software development and software testing are two subdisciplines of what today is generally understood as software engineering. Software engineering is a discipline that attempts to bring software development activities, and testing is part of that, more in line with the traditional engineering disciplines such as civil engineering, mechanical engineering, or electrical engineering. The main goal of software engineering is to come up with standard ways of doing things, standard techniques and methods, and standard tools that have a measurable effect on the primary dimensions that all engineering disciplines address: cost and quality. Software engineering is still in its relative infancy compared with the more traditional engineering disciplines, but it is on its way.

Component-based software development directly addresses the cost dimension, in that it tries to regard software construction more in terms of the traditional engineering disciplines in which the assembly of systems from readily available prefabricated parts is the norm. This is in contrast with the traditional way of developing software in which most parts are custom-designed from scratch. There are many motivations for why people allocate more and more effort toward introducing and applying component-based software construction technologies. Some will expect increased return on investment, because the development costs of components are amortized over many uses. Others will put forward increased productivity as an argument, because software reuse through assembly and interfacing enables the construction of larger and more complex systems in shorter development cycles than would otherwise be feasible. In addition, increased software quality is a major expectation that people are looking for under this subject. These are all valid anticipations, and to some extent they have yet to be assessed and evaluated. Heineman and Councill [89] present a nice collection of articles that address these issues, and many of the discussions in this field are about the economics of software components and component-based software construction. However, this is not the subject of this book.

Testing directly addresses the other dimension, quality, and it has turned out that it must be dealt with somewhat differently in component-based software engineering compared with traditional development projects. This is because component-based systems are different and the stakeholders in component-based development projects are different, and they have contrasting points of view. How component-based systems are different and how they must be treated in order to address the challenges of testing component-based systems are the main subjects of this book. In the following sections we will have a look at the basics of component-based software engineering and what makes testing so special in this discipline, and we will look at the role that the Unified Modeling Language (UML), which emerges as a major accepted standard in software engineering, plays under this subject.

1.1 Component-Based Software Development

1.1.1 Component Definition

The fundamental building block of component-based software development is a component. On first thought it seems quite clear what a software component is supposed to be: it is a building block. But, on a second thought, by looking at the many different contemporary component technologies, and how they treat the term component, this initial clarity can easily give way to confusion. People have come up with quite a number of diverse definitions for the term component, and the following one is my personal favorite:

> A component is a reusable unit of composition with explicitly specified provided and required interfaces and quality attributes, that denotes a single abstraction and can be composed without modification.

This is based on the well-known definition of the 1996 European Conference on Object-Oriented Programming [157], that defines a component in the following way:

> A component is a unit of composition, with contractually specified interfaces and context dependencies only, that can be deployed independently and is subject to composition by third parties.

In this book I chose a somewhat broader terminology that avoids being independently deployable, since I am not specifically restricting the term component to contemporary component technologies such as CORBA, .NET, or COM, which provide independent execution environments. In this respect, I see the term component closer to Booch's definition, or the definition of the OMG:

> A component is a logically cohesive, loosely coupled module [20].
>
> A component is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces [118].

But there are many other definitions that all focus on more or less similar properties of a component, for example:

> A software component is an independently deliverable piece of functionality providing access to its services through interfaces [24].
>
> A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard [89].

From these definitions it becomes apparent that components are basically built on the same fundamental principles as object technology. If we consider an object-oriented language as the only deployment environment, we can also say that objects are independently deployable within this environment. The principles of encapsulation, modularity, and unique identities that the component definitions put forward are all basic object-oriented principles that are subsumed by the component paradigm [6]. However, the single most important feature of a component, in my opinion, is that it may be reused in a number of different contexts that have initially not been anticipated by the component producer. Only the component user decides whether it may be fit for a particular purpose, and this, in fact, is the single most important distinguishing factor of component-based development with respect to the traditional approaches. From the component definitions, we can actually derive a number of important properties for software components according to [24, 66, 89, 158]:

- Composability is the primary property of software components as the term implies, and it can be applied recursively: components make up components, which make up components, and so on.
- Reusability is the second key concept in component-based software development. Development for reuse on the one hand is concerned with how components are designed and developed by a component provider. Development with reuse on the other hand is concerned with how such existing components may be integrated into a customer's component framework.
- Having a unique identity requires that a component should be uniquely identifiable within its development environment as well as its runtime environment.
- Modularity and encapsulation refers to the scoping property of a component as an assembly of services that are related through common data. Modularity is not defined through similar functionality as is the case under the traditional development paradigms (i.e., module as entity with functional cohesion), but through access to the same data (i.e., data cohesion).

- Interaction through interface contracts; encapsulation and information hiding require an access mechanism to the internal workings of a component. Interfaces are the only means for accessing the services of a component, and these are based on mutual agreements on how to use the services, that is, on a contract.

In the following paragraphs we will take a closer look at the concepts that the component definitions put forward.

1.1.2 Core Principles of Component-Based Development

Component Composition

Components are reusable units for composition. This statement captures the very fundamental concept of component-based development, that an application is made up and composed of a number of individual parts, and that these parts are specifically designed for integration in a number of different applications. It also captures the idea that one component may be part of another component [6], or part of a sub-system or system, both of which represent components in their own right. As a graphical representation, composition maps components into trees with one component as the root of the parts from which it is composed, as shown in Fig. 1.1.

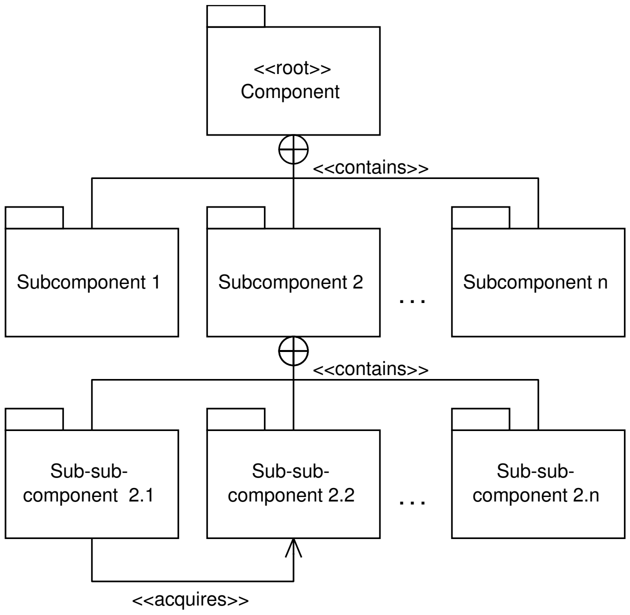


Fig. 1.1. Composition represented by a component nesting tree, or a so-called component containment hierarchy

Component Clientship

An additional important concept that is related to component composition and assembly is clientship. It is borrowed from object technology and subsumed by the component concept. Clientship or client/server relationship is a much more fundamental concept than component composition. It represents the basic form of interaction between two objects in object-oriented systems. Without clientship, there is no concept of composition and no interaction between components. Clientship means that a client object invokes the public operations of an associated server object. Such an interaction is unidirectional, so that the client instance has knowledge of the server instance, typically through some reference value, but the server instance needs no knowledge of the client instance. A clientship relation defines a contract between the client and the server. A contract determines the services that the server promises to provide to the client if the client promises to use the server in its expected way. If one of the two parties fails to deliver the promised properties, it breaks the contract, and the relation fails. This typically leads to an error in the clientship association. A composite is usually the client of its parts. A graphical representation of clientship forms arbitrary graphs, since clientship is not dependent on composition. This is indicated through the ≪acquires≫-relationship in Fig. 1.1. It means that `Sub-subcomponent 2.1` acquires the services of `Sub-subcomponent 2.2`, thereby establishing the client/server relationship. Clientship between contained components in a containment hierarchy is represented by the anchor symbols in Fig. 1.1. The minimal contract between two such entities is that the client, or the containing component, at least needs to invoke the constructor of its servers, or the contained components.

Component Interfaces

A component's syntax and semantics are determined through its provided and required interfaces. The provided interface is a collection of functionality and behavior that collectively define the services that a component provides to its associated clients. It may be seen as the entry point for controlling the component, and it determines what a component can do. The required interface is a collection of functionality and behavior that the component expects to get from its environment to support its own implementation. Without correct support from its servers at its required interface, the component cannot guarantee correct support of its clients at its provided interface. If we look at a component from the point of view of its provided interface, it takes the role of a server. If we look at it from the point of view of its required interface, the component takes the role of a client. Provided and required interfaces define a component's provided and required contracts. These concepts are illustrated in Fig. 1.2.

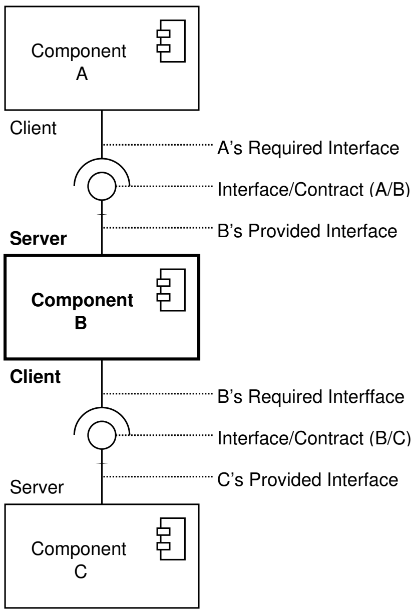


Fig. 1.2. UML-style representation of components with provided and required interfaces

Quality Attributes

Quality attributes have the same meaning for the non-functional aspects of a component that interfaces have for the functional and behavioral aspects of the component. Quality attributes define additional requirements of components, such as dependability and performance.

Quality Documentation

The documentation can be seen as part of a component's specification, or a refinement of its specification. Sometimes, a pure specification may be too abstract, so that it is difficult for users to see and understand how a component's operations may be called or can be applied. It is particularly useful in order to document how sequences and combinations of operation invocations add to the overall behavior. A documentation provides a deeper insight into how a component may be used in typical contexts and for typical usage profiles that the provider of the component had anticipated.

Persistent Component State

An additional requirement of Szyperski's component definition [157] that is often cited in the literature is that a component may not have a persistent state. It means that whenever a component is integrated in a new application, it is not supposed to have any distinct internal variable settings that result from previous operation invocations by clients of another context. This requires that a runtime component, a so-called component instance, will always be created and initialized before it is used. However, this is not practical for highly dynamic component systems such as Web services, which may be assembled and composed of already existing component instances that are acquired during runtime. It is not a fact that because components have persistent states they cannot be integrated into a running system. It is a fact that they must have a well-defined and expected persistent state so that they can be incorporated into a running application. The fact that a component may already have a state must be defined a priori, and it is therefore a fundamental part of the underlying clientship relation between two components. The invocation of a constructor operation, for example, represents a transition into the initial state of a component. This is also a well-defined situation that the client must know about in order to cooperate with the component correctly. In this respect, it may also be seen as a persistent state that the client must be aware of.

1.1.3 Component Meta-model

The previous paragraphs have briefly described the basic properties of a component and component-based development. The following paragraphs summarize the items that make up a component and draw a more complete picture of component concepts. I also introduce the notion of a UML component meta-model, which will be extended over the course of this book, to illustrate the relations between these concepts.

Figure 1.3 summarizes the concepts of a component and their relations in the form of a UML meta-model. It is a meta-model, a model of a model, because it does not represent or describe a physical component but only the concepts from which physical components are composed. The diagram defines a component as having at most one provided interface and one required interface. These two interfaces entirely distinguish this component from any other particular component. The provided interface represents everything that the component is providing to its environment (its clients) in terms of services, and the required interface represents everything that the component expects to get from its environment in order to offer its services. This expectation is represented by the other associated (sub-)components that the subject component depends upon, or by the underlying runtime environment.

Provided and required interfaces must be public, as indicated through the UML stereotype ≪public≫. Otherwise we cannot sensibly integrate the component into an application, because we do not know how the component will be connected with other components in its environment. Provided and required interfaces are also referred to as export and import interfaces.

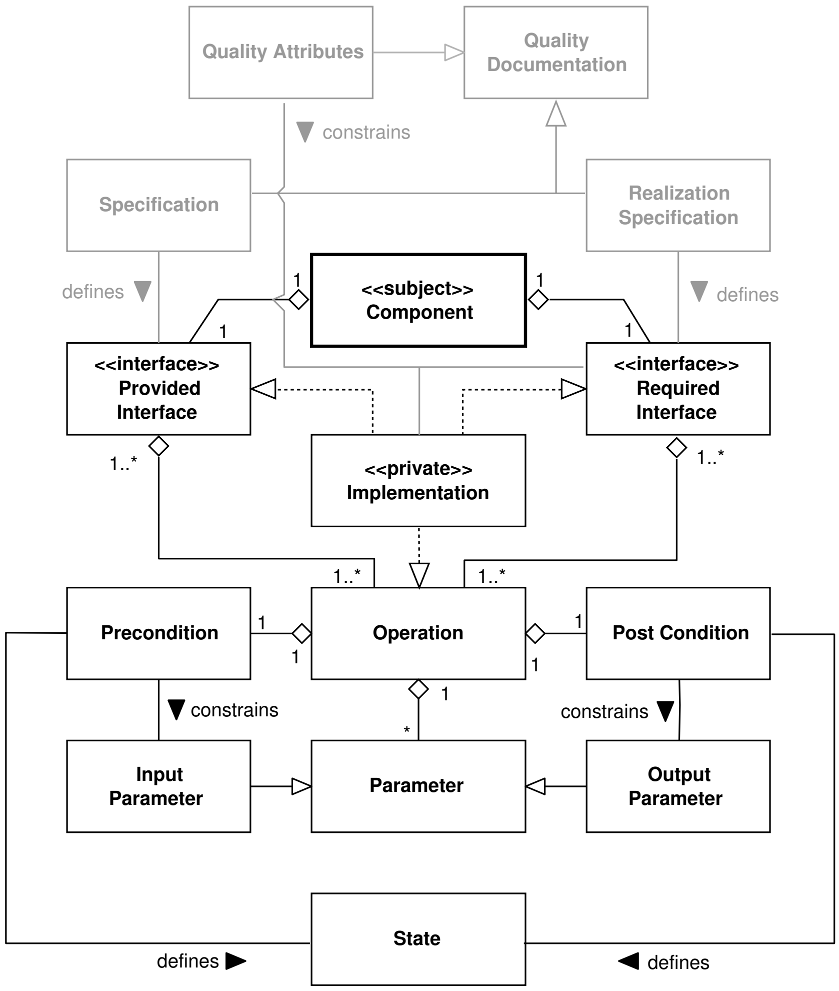


Fig. 1.3. Component meta-model

The implementation of a component realizes its private design. This is the collection of algorithms by which a component achieves its functionality, its internal attributes, internal and external operations, and operation calls to its associated (sub-)components. The implementation is hidden inside the encapsulating shell of the component, and is arbitrarily exchangeable through any other implementation that realizes the same external features.

Provided and required interfaces comprise operations. The operations in the provided interface are access points that the client of the component can use to control its functionality, and the operations in the required interface are the access points to other associated components that the subject component depends on or to the underlying runtime system. The functionality of an operation depends on pre and postconditions. A precondition is an invariant that must be true or fulfilled in order for the operation to guarantee the postcondition. A precondition constrains the input parameters of an operation and defines an initial state of the component that must be valid before the operation may be invoked. A postcondition constrains the output parameters of an operation to the guaranteed values and defines a final state of the component that becomes valid after operation invocation.

The combination of precondition, operation invocation with input parameters, and postcondition, with output parameters, represents a transition from one state to another. A state is a distinct combination of a component's internal attribute values that are constantly changed through the operations. These attributes may be owned by the component, or by any subordinate component that in itself may be seen as an attribute of the superordinate component. States are not directly visible outside a component's encapsulation boundary because they are determined through internal attributes. A state manifests itself through differing observable external behavior of a component only if the same operation is invoked; an operation may not be invoked at all under the conditions of a certain state. For example, the `storeItem()` operation of a storage component behaves differently if the storage is already completely used up. In this case the component won't store any more items.

So far we have looked only at the parts of a physical component in Fig. 1.3. A physical component is an executable binary version of the component. But a component is not really usable if it does not come with some additional description or documentation, which can be seen as the logical part of the component. Additional descriptive artifacts are the specification and realization of the component that represent its quality documentation and a specification of the quality attributes. A specification of a component is a collection of descriptive documents that define what a component can do. It comprises descriptions of the externally provided and required operations with their behavior and pre and postconditions and exceptions. A realization defines how a component implements its functionality, e.g., in terms of interaction with other subordinate components and additional algorithmic information. The quality documentation defines quality attributes that the component is abiding with, plus the quality attributes that it expects from its associated subordinate components. The quality attributes constrain the internal implementation of the component as well as the required interface. This means that a component can only accept an associated server component as long as it provides not only the expected function and behavior but, additionally, the expected quality features.

1.1.4 Component Engineering vs. Application Engineering

Component-based software construction may be subdivided into two distinct development activities: component engineering and application engineering.

While component engineering is concerned with the development of components as individual building blocks in component-based software construction, application engineering is concerned with the assembly and integration of these building blocks into new software applications. Component engineering is primarily performed by the provider of a component, and application engineering is mainly carried out be the component user. I have referred to these activities above respectively as development for reuse and development with reuse.

In its purest form, component-based development is only concerned with the second item, development with reuse (component integration), representing a bottom-up approach to system construction. This requires that every single part of the overall application is already available in a component repository in a form that exactly maps to the requirements of that application. Typically, this is not the case, and merely assembling readily available parts into a configuration will likely lead to a system that is not consistent with its original requirements.

Because the shape of the building blocks that are assembled and integrated during application engineering is so essential to component engineering, development for reuse (component engineering) is an important activity that needs also to be considered in the context of component-based development. After all, component engineering is initially responsible for how suitably and easily components can be composed and reused.

But there is another important dimension to the activities of component engineering and application engineering. Application engineering deals with how a system is decomposed into finer-grained parts that are individually controllable. Component engineering deals with how individual components will fit into the logical decomposition hierarchy that application engineering has come up with. Both are interdependent, and component-based development methods must deal with these two orthogonal approaches.

Component-based development is usually a mixture of top-down decomposition and bottom-up composition. In other words, the system is decomposed into finer-grained parts, that is, subsystems or components, and these are to be mapped to individual prefabricated building blocks. If no suitable components are found, decomposition is continued. If partially suitable components are found, the decomposition is repeated according to the needs of the candidate component. A found suitable component represents a feasible and acceptable solution for the entire system or the subsystem considered. The whole process is iterative and must be followed until all requirements are mapped to corresponding components or until the system is fully decomposed into the lowest desirable level of abstraction. If suitable third-party components are found, they can be composed to make up the system or subsystem under consideration. Such a process is always goal-oriented in that it accepts only components that are fit for the purpose of the system. It means that only these parts will be selected that somehow map to the system specification. The outcome of such a development process is usually a heterogeneous assembly consisting of combinations of prefabricated parts plus implementations.

All these considerations fall into the scope of component-based development methods that provide guidelines on what, when, and how such activities have to be carried out during component-based software construction. The next chapter (Chap. 2, "Component-Based and Model-Driven Development with UML") is devoted entirely to how component-based development should ideally be done in the context of a development method that follows the ideas of model-driven software construction. Development methods provide the framework for applying modern software engineering principles, and they are also responsible for integrating dynamic quality assurance techniques, the main subject of this volume.

1.2 Component-Based Software Testing

Component-based software development has been and still is considered to be the primary technology to overcome the software crisis [23, 89, 159]. Its main idea is to build new software products by reusing readily available parts, rather than by developing everything from scratch. The expected savings in product development are based on the assumption that software reuse has a much higher return on investment than pure software development. This is certainly true for product development, because parts of one system whose investment has already been amortized and written off are used again in another system without any extra cost. However, this is not entirely true. The assumption becomes a reality, and a component-based development project becomes a success, only if the costs of adapting all components to their new environment, and of integrating them in their new context is much lower than the whole system being developed from scratch. The cost of assembling and integrating a new product also includes the assuring of its quality.

The expectation that software developers and software development organizations place in component-based software engineering is founded on the assumption that the effort involved in integrating readily available components at deployment time is less than that involved in developing code from scratch and validating the resulting application through traditional techniques. However, this does not take into account the fact that when an otherwise fault-free component is integrated into a system of other components, it may fail to function as expected. This is because the other components to which it has been connected are intended for a different purpose, have a different usage profile, or may be faulty. Current component technologies can help to verify the syntactic compatibility of interconnected components (i.e., components that they use and that provide the right signatures), but they do little to ensure that applications function correctly when they are assembled from independently developed components. In other words, they do nothing to check the semantic compatibility of interconnected components so that the individual parts are assembled into meaningful configurations. Software developers may therefore be forced to perform more integration and acceptance testing to attain the same level of confidence in the system's reliability. In short, although traditional development-time verification and validation techniques can help assure the quality of individual units, they can do little to assure the quality of applications that are assembled from them at deployment time.

In the previous two paragraphs, I have already presented some ideas on the challenges that we have to consider and address in component-based software testing. In the following subsection we will have a closer look at the typical problems that emerge from this subject.

1.2.1 Challenges in Component-Based Software Testing

The software engineering community has acknowledged the fact that the validation of component-based systems is different from testing traditionally developed systems. This manifests itself in the number of publications in that field, in people's interest in the respective forums and workshops on the topic, and, in the production of this book; references include [65, 67, 77, 175, 176]. Component-based software testing refers to all activities that are related to testing and test development in the scope of a component-based development project. This comprises tests and testing activities during component engineering, carried out by the provider of a component, as well as all testing activities during application engineering, carried out by the user of the component. Here, it is important to note that the two roles, component provider and user, may be represented by the same organization, i.e., in case of inhouse components that are reused in a number of diverse component-based development projects.

A test that the provider performs will typically check the internal workings of a component according to the provider's specification. It can be seen as a development-time test in which the individual parts, classes, or subcomponents of the tested component are integrated and their mutual interactions are assessed. It concentrates on assessing the component's internal logic, data, and program structure, as well as the correct performance of its internal algorithms. Since the provider assumes full internal knowledge of the component, it can be tested according to typical white box testing criteria [11]. This component test will likely concentrate on the individual services that the component provides and not so much on the combination or sequence of the services. The aim of the component provider is to deliver a high-quality reusable product that abides by its functional and behavioral specification.

A second test that the user of the component will typically carry out is to assess whether the module fits well into the framework of other components of an application. This sees a component as a black box entity, so typical black box testing techniques [12] will be readily applied here. The aim of the component user is to integrate an existing component into the component framework, and assess whether both cooperate correctly according to the requirements of the user. For his or her own developments the user will assume the role of the producer and carry out typical producer tests. This is the case, for example, for adapter components that combine and integrate different third-party developments and enable their mutual interactions. Since the user assumes white box knowledge of these parts of an application they can also be tested based on typical white box testing criteria.

The following list summarizes the most important problems that people believe make component-based software testing difficult or challenging [66]:

- Testing of a component in a new context. Components are developed in a development context by the provider and reused in a different deployment context by the user. Because the provider can never anticipate every single usage scenario in which the component will be deployed by a prospective user, an initial component test can only assess the component according to a particular usage profile, and that is the one that the provider can think of. Components are often reused in contexts that the provider could have never imagined before, thus leading to entirely different testing criteria for the two stakeholders [175, 176]. I believe this is the primary challenge in component-based software testing, and the most important subject of Chap. 4, "Built-in Contract Testing", and the entire book is more or less devoted to this problem.
- Lack of access to the internal workings of a component [86, 175], which reduces controllability and testability. Low visibility of something that will be tested is a problem. Testing always assumes information additional to what a pure user of an entity is expecting. For most components, commercial ones in particular, we will not get any insight except for what the component's interface is providing. One part of the technology that I introduce in this book is devoted to increasing controllability and observability and, as a consequence, testability of components. This is the second most important subject of Chap. 4, "Built-In Contract Testing."
- Adequate component testing [142]. This is concerned with the questions of which type of testing, and how much testing a component provider should perform, and which type of and how much testing a component user must perform to be confident that a component will work. Unfortunately, I have no answer to this, and this book concentrates only on which testing criteria may be applied in a model-based development approach. This is the subject of Chap. 3, "Model-Based Testing with the UML." Defining adequate test sets (in particular), and adequate testing (in general) are still challenging subjects, and this represents one of the most significant problems in testing. There is not much empirical evidence of which testing criterion should be applied under which circumstances.

I believe the most fundamental difference between traditional testing and the testing of component-based systems lies in the different view points of the stakeholders of a software component, the provider and the user. This identifies the fundamental difference in the validation of traditional software systems and component-based developments. While in traditional system development all parts are integrated only according to a single usage profile, in component-based development they are integrated according to differing usage profiles that depend on their integrating contexts. In the first instance, system developers can integrate two parts of a system according to a single well-defined and known interface. Even if the two parts do not match, they can be adapted to their respective contexts and fully validated in their particular predetermined association. In the second instance however, this is not feasible. Neither the client nor the server role in an association can be changed, or will be changed, under the component paradigm. Here, component adaptation according to a different context is restricted to inserting an adapter between the two roles. But even if the adapter translates the interactions between client and server correctly, there is no guarantee that the semantic interaction between the two will be meaningful. The two deployed and integrated components have been initially developed in complete mutual ignorance for each other, so although a component is alright in one deployment context, e.g., at the component vendor's site, it may completely fail in another deployment context, e.g., in the component user's application. And this is not a problem of the individual components, because they may be individually perfect, but of their interaction, or their interaction within a new context. Even if such components have been developed by the same organization, and the development teams have full access to all software artifacts, something can go horribly wrong, as the ARIANE 5 crash 1996 illustrates.

1.2.2 The ARIANE 5 Failure

The first launch of the new European ARIANE 5 rocket on June 4th, 1996, ended in failure, and, consequently, in the rocket's destruction about 40 seconds after it had taken off. This is probably the most prominent real-world example to illustrate the fundamental problem in testing component-based systems. What had caused the failure was later identified through an extensive investigation which was published in a report by the inquiry board commissioned through the European Space Agency (ESA) [104]. In addition, the failure was analyzed by Jézéquel and Meyer [97].

The inquiry board found out that the on-board computer had interpreted diagnostic bit pattern from the inertial reference system as flight data, and wrongly adjusted the flight angle of the rocket. This had caused the rocket to veer off its intended flight path and disintegrate in increased aerodynamic loads. The inertial reference system did not send correct attitude data to the on-board computer due to a software exception which was caused through a conversion of a 64-bit floating point value into a 16-bit integer value. The floating point value was greater than what could be represented by the integer variable which created an operand error in the software. The error occurred in a component that was taken from the ARIANE 4 system and reused within the ARIANE 5 system, and that performed meaningful operations only before lift-off. According to ARIANE 4 specifications, the component continued its operation for approximately 40 seconds after lift-off and provided data representing the horizontal velocity of the rocket in order to perform some alignments. However, this was not a requirement of ARIANE 5. The exceptionally high values that caused the operand error were due to the considerably higher horizontal velocity values of the much more powerful ARIANE 5. The requirement for having the inertial reference component continue its operation for some time after lift-off came from the particular preparation sequence of ARIANE 4, which did not apply to the newer ARIANE 5.

1.2.3 The Lessons Learned

We can now have a closer look at why this is the typical failure scenario in component-based software construction. The reused component coming from the ARIANE 4 was obviously alright, and compliant with its specification. The ESA had used that component successfully and without any trouble for years in the ARIANE 4 program. So they could claim reasonably that this was a highly dependable piece of software that may also be used successfully in the new ARIANE 5 program. Or could they not?

Apparently, something went wrong because the importance of the context in component-based development was not considered. In this case the context is the integrating system (the new ARIANE 5) that exhibits an entirely different usage profile of the component from the original system (the old ARIANE 4). The much higher velocity of the new rocket generated values in the original inertial reference component that the designers of that component had not anticipated or considered at the time. At the time of its initial development, the ARIANE 4 engineers did not anticipate a new rocket that would require such velocity numbers. In other words, the new system exhibited a usage profile of the component that its original development team did not take into consideration at the time. The developers implemented the component in a way that provided more than enough margin for the failure of the application under consideration.

It is unlikely that contemporary component-based software testing technologies, some of which will be introduced in this book, could have prevented the crash. At that time, ESA's quality assurance program was aimed only at hardware failures, so that they would not have identified the software problem. Well, they could have, if they had considered software failures as a problem at the time, and in fact they do that now, and tested the system properly. But it clearly illustrates the effect that the integration of a correct and dependable component into a new context may have on the overall application. The fact that components may be facing entirely different usage scenarios, and that they have to be tested according to every new usage scenario into which they will be brought, is the most fundamental issue in testing component-based systems.

1.3 Model-Based Development and Testing

Modeling and the application of graphical notations in software development have been receiving increasing attention from the software engineering community over the past ten years or so. The most prominent and well known representative of graphical modeling notations is the Unified Modeling Language (UML) [118]. This has actually become a de-facto industry standard; it is readily supported by quite a number of software engineering tools, and the amount of published literature is overwhelming. The UML is fostered and standardized by the Object Management Group (OMG), a large consortium of partners from industry and the public sector (e.g. universities and research organizations) worldwide. The OMG's mission is to define agreed-upon standards in the area of component-based software development that are technically sound, commercially viable, and vendor independent. The UML, among other standards that the OMG has released over the years, is one of their earliest products, with its initial roots in the object-oriented analysis and design methods from the late 1980s and early 1990s [60]. It is a graphical notation, or a language, and not a method. I will introduce a method, the KobrA method [6], that is based on the UML, in Chap. 2. The UML can be used for specifying and visualizing any software development artifact throughout an entire software development project. The initial overall goals of the UML are stated as follows [44], and the language addresses these topics sufficiently:

- Model systems with object-oriented concepts.
- Establish an explicit coupling between conceptual as well as executable artifacts.
- Address the problems of scalability and complexity.
- Create a notation that can be used by humans as well as by machines.

In particular, the last item is being tackled more and more by researchers in the domain of generative programming that is based on, or follows the fundamental concepts of, Model-Driven Architectures (MDA) [22, 79]. The basic idea is to develop a system graphically in the form of UML models, pretty much according to the ideas of computer-aided design in mechanical engineering, and then translate these models into an executable format. The correlative technology for this second step in mechanical engineering is computer-aided manufacturing. Generative programming is not as simple as it may sound, and I will present some of the challenges of this subject in Chap. 2, but only marginally. I will give an overview on the primary concepts of the UML in Chaps. 2 and 3, and I will use the UML to specify the examples throughout the book.

1.3.1 UML and Testing

Testing activities that are based on models, or use models, are becoming increasingly popular. We can see this in the number of publications that have been emerging over the last few years, for example [1, 87, 88, 100, 119], to name only a few. UML models represent specification documents which provide the ideal bases for deriving tests and developing testing environments. A test always requires some specification, or at least a description or documentation of what the tested entity should be, or how it should behave. Testing that is not based on a specification is entirely meaningless. Even code-based, or so-called white box testing techniques, that initially only concentrate on the structure of the code, are based on some specification. The code is used only as a basis to define input parameter settings that lead to the coverage of distinct code artifacts. In Chap. 3, "Model-Based Testing with the UML," I will give a more extensive overview of these topics. Models are even more valuable if UML tools that support automatic test case generation are used. In general, we can discriminate between two ways of how to use the UML in tandem with testing activities; this is further elaborated upon in the following subsections, and in Chaps. 3 and 4:

- Model-based testing – this is concerned with deriving test information out of UML models.
- Test modeling – this concentrates on how to model testing structure and test behavior with the UML.

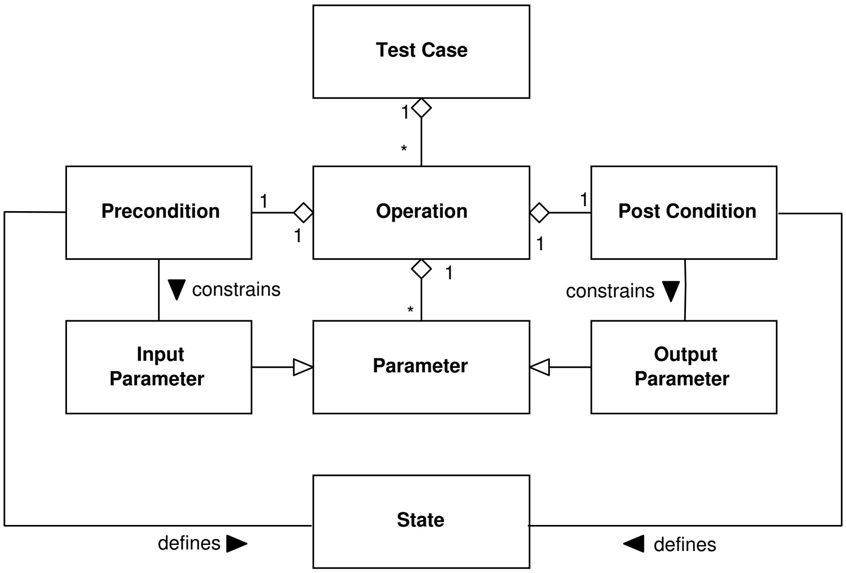


Fig. 1.4. UML-style representation of the concepts of a test case

1.3.2 Model-Based Testing

The UML represents a specification notation, and testing is the using or the applying of the concepts of a specification. A test case for a tested component, for example, comprises one or more operations that will be called on the tested object, a precondition that defines constraints on the input parameters for the test and determines the initial state of the object, and a postcondition that constrains the output of the tested operation, and defines the final state of the object after test execution. These concepts are depicted in Fig. 1.4, and it becomes apparent that this maps exactly to the lower part of the component meta-model depicted in Fig. 1.3 on page 8. So, we can map the concepts of a component exactly to the concepts of a test case for a component. Although, for a test we will need these concepts twice, once for the specification of what should happen in a test and once for the observation of what really happens. A validation action can then be performed to determine whether the test has failed or passed, and this is called the verdict of a test. The concepts of a test case are therefore a bit more complex, but the UML Testing Profile defines them sufficiently [120]. Hence, the UML readily provides everything that is necessary to derive tests and test cases for a component, and it even provides sufficient information to define entire application test suites. This will be described in Chap. 4,"Built-in Contract Testing," and Chap. 5, "Built-In Contract Testing and Implementation Technologies."

1.3.3 Test Modeling

Test cases or a test suite represent software. Any software, whether it performs any "normal functionality" or whether it is especially crafted to assess some other software, should be based on a specification. The UML is a graphical specification notation, and therefore it is also adequate for the specification of the test software for an application. Why should we apply a notation for testing that is different from the one we use for developing the rest of a system?

In general, the UML provides everything that is required to devise a testing framework for components or entire applications. However, there are some very special concepts that are important for testing, as I have stated in the previous subsection. These very special concepts are provided by the UML Testing Profile that extends the meta-model of the core UML with testing artifacts and testing concepts [120]. Chapters 3 and 5 concentrate on this topic, among other things.

1.4 Summary and Outline of This Book

This book addresses two of the three primary challenges in testing component-based software systems that I have identified in Sect. 1.2:

- Lack of access to a component's internal implementation, and as a consequence low observability, low controllability, and, thus, low testability of components.
- Testing of a component in a new context for which it had not been initially developed.

This book focuses on built-in testing for component-based software development in general, and built-in contract testing and related technologies for component-based development in particular. These technologies can, if they are applied wisely and organized intelligently, provide feasible solutions to these typical challenges. The fundamental idea of built-in testing is to have the testing for a software system directly incorporated into the system, although this may not always be the case, as we will see later on. Because testing, if it is viewed in that way, will be an essential part of an application, it must be integrated seamlessly into the overall development process of an organization. After all, testing in this respect is just another development effort that extends a component, or an assembly of components.

Since component-based system development and the UML go so well together, it is logical to set up the testing activities on this notation as well, and, additionally, to have it supplement an existing mainstream component-based development method, such as the KobrA method [6]. UML and testing are a perfect fit, and the component-based development method provides the common framework for putting the three main subjects of this book together: components, modeling, and testing.

The next chapter, Chap. 2 on "Component-Based and Model-Driven Development with the UML," introduces the KobrA method as the adhesive framework that incorporates all the other technologies. The chapter gives an overview on the KobrA method. It explains its phases, core development activities, and describes its artifacts that are mainly UML models. The chapter will also introduce an example that is more or less used consistently throughout the entire book to illustrate all the essential concepts and technologies.

The next chapter, Chap. 3 on "Model-Based Testing with the UML," describes why and how the UML and testing are a perfect fit. It briefly compares the more recent model-based testing techniques with the more traditional testing criteria, and then covers the two main areas extensively: model-based testing and test modeling. Under the first topic, I will introduce the diagram types of the UML (version 2) and investigate how these can be good for deriving testing artifacts. Under the second topic, I will introduce the recently published UML Testing Profile and investigate how this may be used to specify testing artifacts with the UML.

In Chap. 4 , "Built-in Contract Testing," which represents the main technological part of the book, we will have a look at what built-in testing means, and where it is historically coming from. Here, we will have a closer look at the two primary challenges in component-based software testing that this book addresses, and I will show why and how built-in contract testing presents a solution for tackling these. I will introduce the model of built-in contract testing, and describe extensively how testing architectures can be built and how modeling fits under this topic. This chapter concentrates on how built-in contract testing should be treated at the abstract, modeling level. The chapter also comprises the description of the built-in contract testing development process that can be seen as part of, or as supplementing, the development process of the KobrA method.

Chapter 5, on "Built-in Contract Testing and Implementation Technologies," looks at how the abstract models that have been defined in Chap. 4 and represent the built-in contract testing artifacts can be instantiated and turned into concrete representations at the implementation level. Here, we will have a look at how built-in contract testing can be implemented in typical programming languages such as C, C++, or Java, how contemporary component technologies, including Web services, affect built-in contract testing. An additional section is concerned with how built-in testing can be realized through existing testing implementation technologies such as XUnit and TTCN-3 [63, 69].

Chapter 6, on "Reuse and Related Technologies," concentrates on how built-in testing technologies support and affect software reuse as the most fundamental principle of, or as the main motivation for, applying component-based software development. It illustrates how the built-in contract testing artifacts can be used and reused under various circumstances, how they support earlier phases of component-based development, in particular component procurement, and how they can be applied to testing product families as generic representatives for software reuse at the architectural level.

Chapters 2 to 6 concentrate only on testing functional and behavioral aspects of component-based software engineering. Additional requirements that have to be assessed in component-based development belong to the group of quality-of-service (QoS) attributes, and Chap. 7, "Assessing Quality-of-Service Contracts," focuses on these. It gives an overview on typical QoS attributes in component contracts in general and characterizes timing requirements in component interactions, so-called timing contracts, in particular. It concentrates primarily on development-time testing for component-based real-time systems.

Additionally, I give an outline of an orthogonal built-in testing technology, built-in quality-of-service testing, that is typically used to assess QoS requirements permanently during the runtime of a component-based application.

2 Component-Based and Model-Driven Development with UML

Traditional, non-component-based software development in its purest form is typically performed in a top-down fashion in which the whole system is broken down into consecutively smaller parts that are individually tackled and implemented. Each part of such a system is specified, designed, and coded exactly according to the specification of the superordinate part that subsumes it. Hence, all the modules and procedures in such a custom development are crafted exactly to fit the overall application, but only that. In contrast, the main idea in component-based development is the reuse of existing building blocks that are usually quite generic, and that have been initially developed with no particular application context in mind, or with an entirely different application context of an earlier development. So, component-based development in its purest form is typically performed in a bottom-up fashion. In fact, it is a composition activity in which the whole system is put together out of more or less suitable parts that are already available in a component repository in a form that should somehow map to the requirements of the overall application.

In practice, the two approaches are intermingled. On the one hand, traditional development also applies component principles when it uses already existing functionality from system libraries or domain-specific modules. The lowest level of reuse is achieved when the system is implemented in source code. In theory, source code instructions and library calls may be regarded as the most basic generic building blocks that may be reused in a traditional development effort. On the other hand, component-based development is also contingent on typical top-down approaches, since merely assembling readily available parts into a configured application will quite likely lead to a system that is not consistent with its original requirements. Going from top down, as in traditional development, ensures that we get the system we are after.

A typical state-of-the-practice software development will therefore decompose the entire system into finer-grained parts that will be mapped somehow to existing functionality. If no such functionality is found that can be reused, the system will be separated further into smaller and smaller units, until the lowest desirable level of decomposition is achieved. These units will be implemented in source code, or reused if already existing component implementations may be found, and then integrated to compose the final product.

Software development methods and processes provide the support for all these aspects during all life cycles of a software project. A good method will guide the entire development team through all phases. It will support the developers in what they have to do, when they have to do it, and how they will be doing it. The method will give support on which steps to follow throughout the development process and on how these individual steps add to the overall product. A development method is like a recipe that identifies all the necessary ingredients and tells you how to put these together.

This chapter focuses on how component-based systems are ideally developed with UML by following a distinct model-based analysis and design process that is supported through a standard approach to constructing software, known as the KobrA method [6]. It gives an overview on the development principles according to a three-dimensional model that the KobrA method suggests (Sect. 2.1). Sections 2.2 to 2.4 concentrate on the main component modeling activities within the KobrA method that are associated with the first dimension of KobrA's development model, and how these are applied to come up with the primary KobrA UML artifacts, context realization, component specification, and component realization. The sections give an overview of the created UML models and how they can be used to define parts of an example application. Section 2.5 focuses in the second dimension of KobrA's three-dimensional development model, embodiment, that deals with how abstract representations in the form of UML models can be turned into concrete executable representations or physical components. It also explains the fundamental principles of how component reuse is addressed and how COTS components are treated by the method which can be seen as two additional aspects of component embodiment. The next section (Sect. 2.6) outlines how product family concepts may be treated within an overall development process, and it represents the third dimension of KobrA's development model. It also introduces the primary activities in product family development, framework engineering and application engineering. Section 2.7 introduces KobrA's mechanisms for dealing with documentation and quality assurance, but since this entire volume is on testing, the remaining chapters of the book will discuss the quality assurance issues. Section 2.8 summarizes and concludes this chapter.

2.1 Principles of the KobrA Method

Every serious attempt at developing software professionally should be based on a sound development method and process. Its role is to accompany the development with guidelines and heuristics describing where, when, and how advanced development technologies such as object-oriented design or modeling should be used. A method acts as a framework and a process in which the development effort will be carried out. Additionally, it defines the intermediate development artifacts, and provides guidelines on how these should be used as input to subsequent development cycles. It also ideally supports their verification and validation in some way. Applying a development method leads to all the necessary software documents that collectively make up the entire software project.

One example for a sound development method is the KobrA method [6] that has been developed primarily by the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern, Germany. It draws its ideas from many contemporary object-oriented and component-based methods, although it aims at combining their advantages while trying to iron out their disadvantages or shortcomings. The most influential methods that lend their concepts to the KobrA method are OMT [143], Fusion [33], ROOM [152], HOOD [140], OORAM [132], Catalysis [42], Select Perspective [2], UML Components [28], FODA [99], FAST [174], PuLSE [8], Rational Unified Process [96, 101], OPEN [70], and Cleanroom [116].

The KobrA method uses the UML as primary model-based notation for all analysis and design activities. In other words, most software documents that are created during the development with this method are UML models. There are other artifacts in natural language or in tabular form that will be introduced together with the models throughout this chapter, but KobrA follows the fundamental ideas of model-driven development approaches such as OMG's Model-Driven Architectures (MDA) [79] which propagate the separation of business or application logic from any underlying concrete implementation or platform technology. Hence, the essential structure and behavior of an application is captured in an abstract form, so that it can be mapped to various available concrete implementation technologies. Fully platform independent models (PIM) of this kind do not only allow a system architecture to exploit the most appropriate implementation technology available at the time, but additionally it endows the system with the capacity to endure inevitable technology evolution [6, 79].

The KobrA method only applies a limited number of concepts, and it follows strictly the principle of "separation of concerns" throughout an entire project. Separation of concerns is a time-honored strategy for handling complex problems in science and engineering. KobrA is also a method that follows the fundamental ideas of model-based development and component technology. It supports the first item, in that it guides the developer on how to use models, and provides a starting point for development and a process to follow throughout the development. Additionally, it supports the second item, in that it promotes component reuse, and follows typical object technology principles. These are the principles of modularity, encapsulation and information hiding, unified functions and data, unique identities, and incremental development.

Since KobrA's core principle is separation of concerns, it associates its main development effort with two basic dimensions that map to the following four basic activities in these dimensions, illustrated in Fig. 2.1:

- Composition/decomposition dimension. Decomposition follows the well established "divide-and-conquer" paradigm, and is performed to subdivide the system into smaller parts that are easier to understand and control. Composition represents the opposite activity, which is performed when the individual components have been implemented, or some others reused, and the system is put together.
- Abstraction/concretization dimension. This is concerned with the implementation of the system and a move toward more and more executable representations. It is also called embodiment, and it turns the abstract system represented by models that humans can understand into representations that are more suitable for execution on a computer.

KobrA defines an additional Genericity/specialization dimension that becomes important in a product family development context. Activities that are performed in this dimension are concerned with instantiating a generic framework architecture, a so-called product line or product family, into a specific product. I am leaving this dimension out of consideration for the moment since it only complicates the overall development process. I will introduce these concepts in Sect. 2.6 of this chapter. The following paragraphs describe in more detail the activities that are performed in each of the two main dimensions.

2.1.1 Decomposition

A development project always starts with the box on the top left hand side of the diagram in Fig. 2.1. This box represents the system that we would like to have, for example a vending machine. If we find a vending machine that fully satisfies the specification of that box, we are done, given that we are the customer of a vending machine provider. If we are the provider and make a living out of producing vending machines, we have to look at the individual parts that make up this system. Maybe we identify a box that accepts coins, one that dispenses change, one that issues the item that we would like to buy, some button-box for selecting it, and, additionally, some control logic that draws all these items together. By doing that we move down the composition/decomposition dimension and identify finer-grained logical or abstract components that we can add to our model. Eventually we might end up with a picture, like that in Fig. 2.2, that shows the organization of the boxes for our vending machine.

During the decomposition activity we always attempt to map newly identified logical components to existing components that we may be able to reuse. This requires a good understanding of the vending machine domain and, additionally, a sound knowledge of the available components for that particular do-

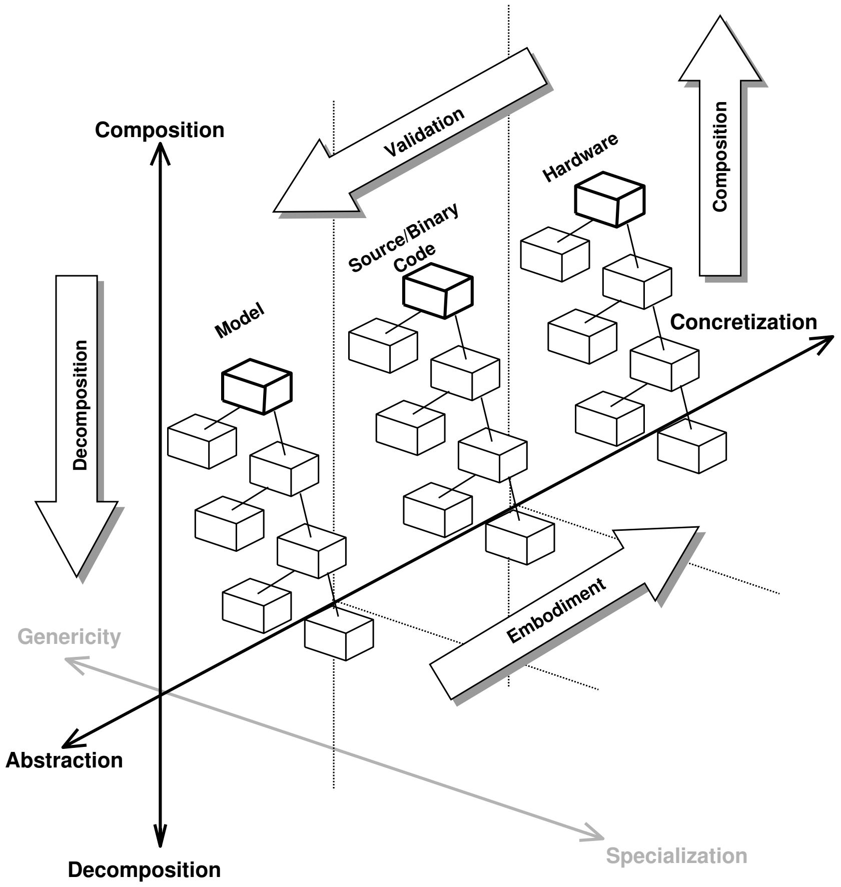


Fig. 2.1. The two main development dimensions of the KobrA method. An additional dimension that is used for domain and product line engineering is only slightly indicated

main. In fact, decomposition should always be directed toward identifiable and reusable existing components. Otherwise we end up decomposing our system into something odd, for which we will never find any suitable existing parts. This means that we entirely miss the fundamental goal of component-based system construction, and we end up developing everything from scratch. And this is indeed a problem for which all existing component-based development methods only provide very limited support. Mapping functional properties into suitable components still requires considerable human intelligence and experience of the domain expert.

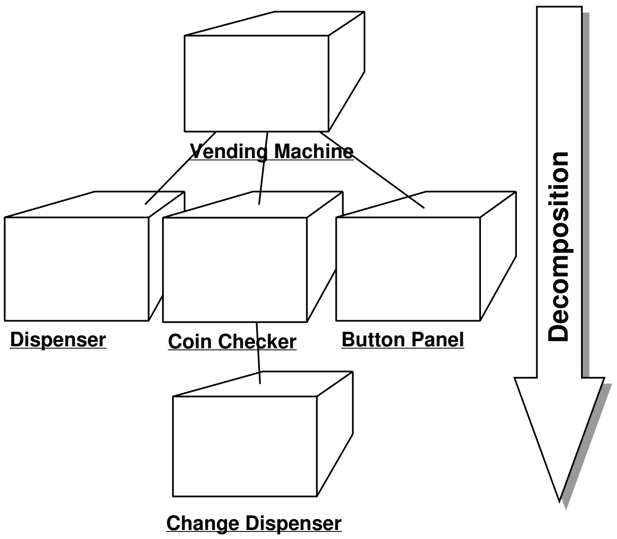


Fig. 2.2. Decomposition of a vending machine into finer-grained components

2.1.2 Embodiment

During decomposition, we identify and fully specify each box. This comprises a specification and a realization. Essentially, the specification defines the provided and required interfaces of the box. This is the connection to the superordinate and subordinate boxes. The realization defines how these interfaces are implemented. This comprises the implementation of the interactions with its subordinate boxes plus the algorithms by which it realizes its functionality. We will have a closer look at specifications and realizations in subsequent sections.

When we have specified a box entirely in an abstract way, for example through UML models, we can start to implement it by moving toward more concrete representations that are eventually executable by a computer. This activity is termed embodiment and it will typically involve some coding effort and, since we would like to increase component reuse, some component selection and validation effort. By performing an embodiment step we move down the concretization dimension displayed in Fig. 2.1. For our vending machine, it involves not only the development of some source code, but, since it is an embedded control system, also a decision of which parts of the abstract model will be implemented in software, in hardware, and in a combination of both. For example, at the abstract level of the UML model, we will call the operation `dispense(Item)` on the dispenser component of our vending machine. Since we are accessing a hardware component, at a more concrete level this call has to be translated into some electrical signal that runs through a cable and connects to the dispenser hardware.

2.1.3 Composition

After we have implemented some of the boxes, and reused some other existing boxes, we can start to put them together according to our abstract model so that we finally get the system running. This activity is termed composition, and it represents a move up in the composition/decomposition dimension. By doing this we connect the subordinate boxes with the superordinate boxes through some containment rules. For example, in code, it maps to a creation operation for the contained box or, if we reuse existing components, it maps to some message translation between the two boxes.

2.1.4 Validation

A final activity is the validation of the implemented boxes against their abstract models. This represents a move back in the abstraction/concretization dimension in Fig. 2.1. Validation is not necessarily the last activity in this cycle. We do not have to fully decompose the system in order to implement a box, and if we have implemented it, we can immediately validate it against its specification. However, if we have not implemented the subordinate boxes on which the current box relies, we will have to develop test stubs to simulate this missing behavior. If we decompose the system into the lowest desirable level and perform the embodiment by starting from the bottom, we will not have any missing links for performing the validation. Hence, we will not have to develop test stubs. But this, of course, is an idealized scenario, and typically we will do a combination of both, the integration of existing subordinate components and the development of test stubs for missing component implementations. For example, in embedded system development in particular, it is quite common for the software system to be developed at the same time or even before the hardware system. In this case, validation of the software system is dependent upon a hardware environment that is simulated through test stubs.

2.1.5 Spiral Model vs. Waterfall Model

Any development effort can always be attributed to one of the two dimensions, and the activities do not necessarily have to be performed in exactly that order. For example, whenever we have decomposed the system and identified some new boxes, we can have an embodiment step, a decomposition step, a composition step, and a validation step. At least we can do embodiment, and perform composition and validation at a more abstract level, i.e., in the UML model. In other words, we can integrate an existing component model into our hierarchy and define how we will test this integration. This is the beauty of working with abstract representations and applying models. Such a development model is called spiral model, because the development is spiraling between the different activities that may be performed at the same time [124].

This model is quite in contrast with the traditional waterfall model or the V-model as corresponding product model that imposes a much stricter sequence of the development activities [41, 124]. The waterfall model is suitable for more traditional development in which most software artifacts are developed from scratch, while the spiral model is more suitable for component-based development.

For our vending machine example, we may come up with a box structure as displayed in Fig. 2.3. This represents the component nesting of the vending machine system. Every UML package symbol in the diagram defines a set of model elements, for example, UML diagrams, that collectively and completely describe each component. The anchor symbol in the diagram represents the scoping semantics that is associated with the nesting of components [6].

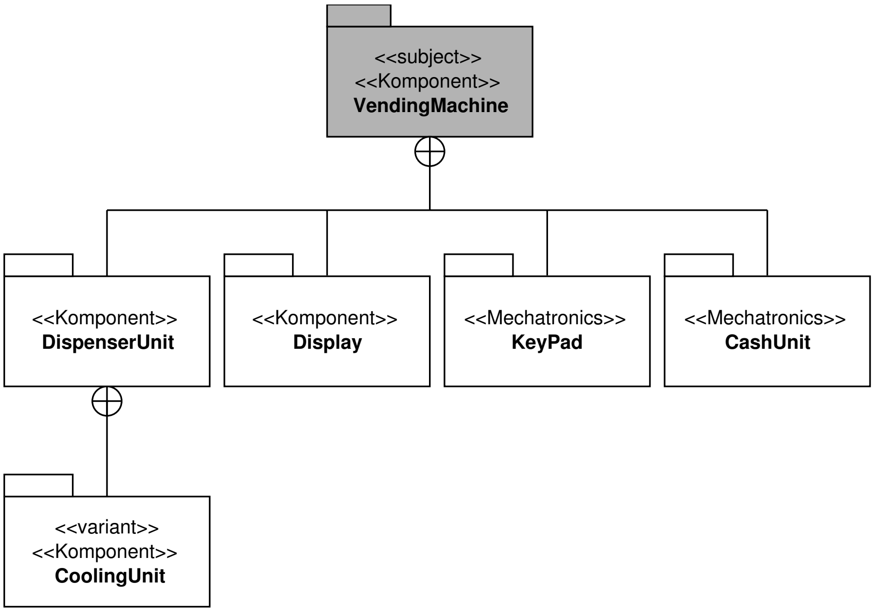


Fig. 2.3. Development-time containment tree for the vending machine example

Because we assume knowledge of the vending machine domain, we can decompose our vending machine system, represented by the shaded box in Fig. 2.3, into subcomponents for which we are quite likely to find reusable third-party developments. For example, we can define a `CashUnit` that will accept and pay back change, a `KeyPad` component from which the user can choose the item to buy, a `Display` component, and a `Dispenser` unit that issues the item. We may also decide that this component will have a `CoolingUnit`. The stereotype ≪Komponent≫ indicates that the UML package symbol represents a KobrA Component, so that it abides by certain rules which are fully described in [6].

The stereotype ≪variant≫ indicates that the component is part of a product family development. Each ≪variant≫ component indicates that we may have differing component implementations according to which final product we would like to instantiate. I introduce that for the vending machine because I would like to be able to change it or extend it later on. Through the ≪variant≫ stereotype we plan for possible future amendments of our product.

2.2 Context Realization

The initial starting point for a software development project is always a system or application specification derived and decomposed from the system requirements. User-level requirements are collected from the customer of the software, for example, in the form of use case diagrams and use case descriptions. They are decomposed in order to remove their genericity in the same way as system designs are decomposed to obtain finer-grained parts that are individually controllable.

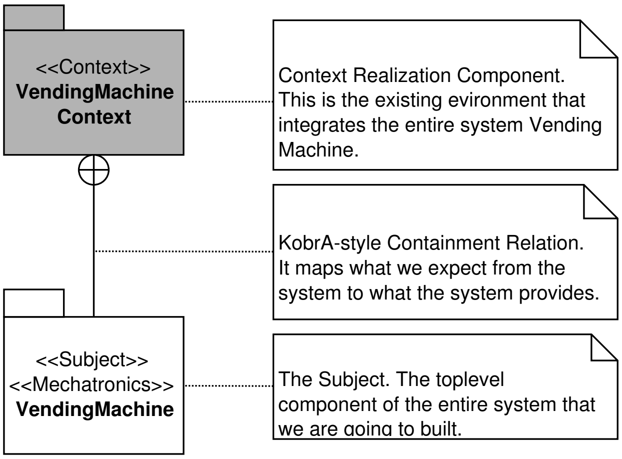

<<Context>>
VendingMachine Context
Context Realization Component. This is the existing evironment that integrates the entire system Vending Machine.
KobrA-style Containment Relation. It maps what we expect from the system to what the system provides.
<<Subject>>
<<Mechatronics>>
VendingMachine
The Subject. The toplevel component of the entire system that we are going to built.

Fig. 2.4. Excerpt from the containment tree: vending machine context realization and vending machine component

The system or application specification is supposed to reflect the expectations of the environment into which the system will be integrated. This may be the user of the system, associated other systems that have an effect on our system, and the processes by which all these stakeholders interact. The collection of all descriptive artifacts that define the environment of the system is termed the context realization, and the KobrA method defines that in the same way as a normal superordinate component that integrates the subject system. It is represented as if it were a component in its own right, and under the KobrA method it is therefore treated almost in the same way as any other component in the containment hierarchy. It happens to integrate the top-level component as displayed in Fig. 2.4.

The vending machine context realization defines the existing environment into which our vending machine will be integrated. In effect it defines the expected interface of the vending machine component. If we find a third-party component whose specification exactly satisfies the context we may get that and use it. Typically, this is quite unlikely, so we have to develop the vending machine according to the requirements of the context. The vending machine specification will therefore exactly reflect the context realization. This, in effect, is the same as custom development with no reuse taking place. The stereotype ≪Mechatronics≫ in Fig. 2.4 indicates that the system will be a combination of software, electronic, and mechanical parts. This is a concept from an extension of the KobrA method, the so-called Marmot method, that is specifically geared toward embedded system development [3].

The definition of the context represents the initial phase in any development project, and it is typically performed to gather information about

- which user tasks will be supported by the new system,
- which documents will serve as input to the system,
- which other systems will interact with our system, and
- which objects will be needed from the system to perform other activities in the context.

A usage model that is represented by use case diagrams is particularly suitable for performing this initial activity. Other models that may be used in the definition of the context are the enterprise or business process model, the structural model, and the activity and interaction models. Figure 2.5 shows the individual artifacts that make up a context realization.

2.2.1 Usage Model

Usage models specify high-level user interactions with a system. This includes the users or actors as subjects of the system, and the objects of the system with which the users interact. Thus, use case models may be applied to define the coarsest-grained logical system modules. Use cases mainly concentrate on the interactions between the stakeholders of a system and the system at its boundaries. A use case diagram shows the actors of the system, or the stakeholders, either in the form of real (human) user roles, or in the form of other associated systems that are using the system under development as server. Additionally, use case diagrams show the actual use cases and the associations between the actors and the use cases. Each use case represents some abstract activity that the user of the system may perform and for which the system provides the support. Overall, use case modeling is applied at initial

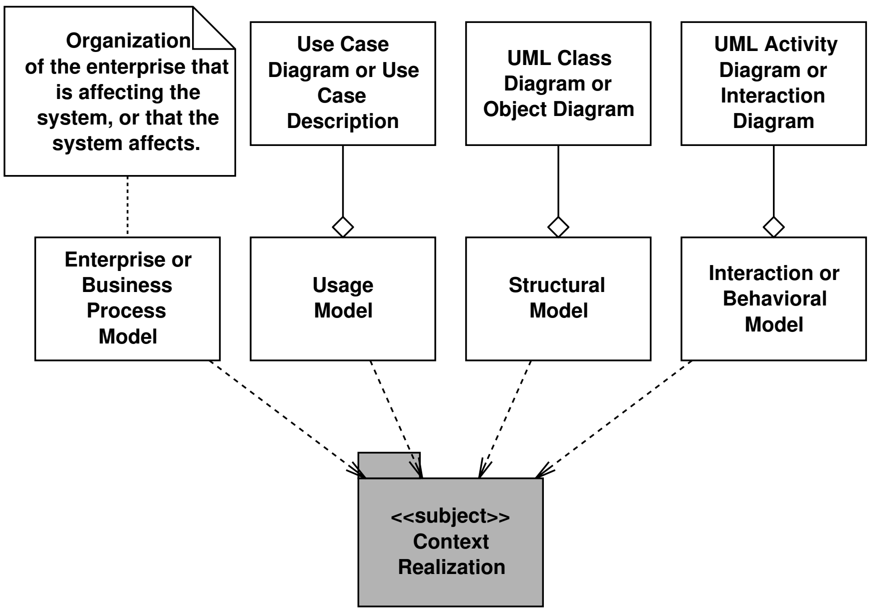

Organization of the enterprise that is affecting the system, or that the system affects.
Use Case Diagram or Use Case Description
UML Class Diagram or Object Diagram
UML Activity Diagram or Interaction Diagram
Enterprise or Business Process Model
Usage Model
Structural Model
Interaction or Behavioral Model
<<subject>> Context Realization

Fig. 2.5. UML-style definition of a generic context realization

requirements engineering phases in the software life cycle to specify the different roles that are using the individual pieces of functionality of the system, the use cases. Use case diagrams are often defined in terms of the actual business processes that will be supported by a system. Figure 2.6 shows the use case model for the vending machine. It comprises two primary actors, the main user who buys items from the machine, and the operator who performs the maintenance on the vending machine. The externally visible functionality of a vending machine is pretty simple, so that we can only identify a single use case, `PurchaseItem`, for the user.

Use case diagrams obviously do not give away much information, so they are typically extended by use case descriptions or use case definitions. A sole use case diagram is quite useless for the concrete specification of what a system is supposed to do. Each use case in a use case diagram is additionally individually specified and described according to a use case template. Table 2.1 shows an example use case template with the individual topic definitions taken from [19] and [31, 32]. It represents typical items of a use case that might be important for expressing interaction with a system from a user's perspective. Use case templates may be different according to the applying organization and the software domain in which they are applied. Table 2.1 represents an example of how to describe a use case in general terms.

Table 2.2 displays the description of the use case `PurchaseItem` from the usage model of the vending machine. The most important entries are the pre and postconditions and the description of the basic and alternative courses

Table 2.1. Use case template

Use Case No.	Short name of the use case indicating its goal (from the model)
Goal in Context	Longer description of the use case in the context.
Scope & Level	Scope and level of the considered system, e.g. black box under design, summary, primary task, sub-function, etc.
Primary/Secondary Actors	Role name or description of the primary and secondary actors for the use case, people, or other associated systems.
Trigger	Which action of the primary/secondary actors initiate the use case.
Stakeholder & Interest	Name of the stakeholder and interest of the stakeholder in the use case.
Preconditions	Expected state of the system or its environment before the use case may be applied.
Postconditions on success	Expected state of the system or its environment after successful completion of the use case.
Postconditions on failure	Expected state of the system or its environment after unsuccessful completion of the use case.
Description Basic Course	Flow of events that are normally performed in the use case (numbered).
Description Alternative Courses	Flow of events that are performed in alternative scenarios (numbered).
Exceptions	Failure modes or deviations from the normal course.
NF-Requirements	Description of non-functional requirements (e.g. timing) according to the numbers of the basic/alternative courses.
Extensions	Associated use cases that extend the current use case (≪extends≫ relation).
Concurrent Uses	Use cases that can be applied concurrently to the current use case.
Revisions	Trace of the modifications of the current use case specification.

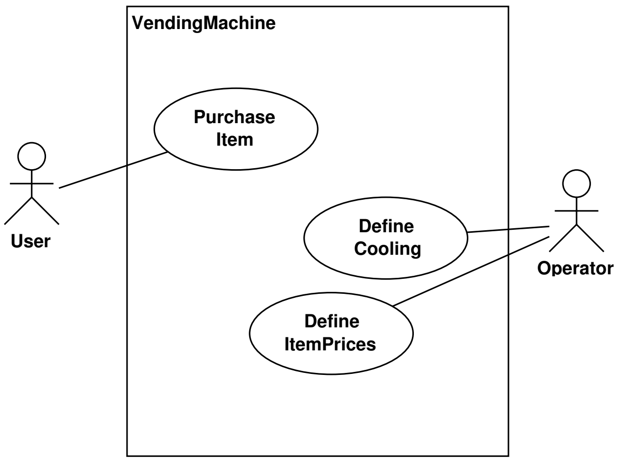


Fig. 2.6. Use case diagram for the vending machine

of the use case scenario. They present the basis for the derivation of the vending machine specification and its externally visible behavior in the next development step.

2.2.2 Enterprise or Business Process Model

The other context realization models view the environment of the system that we would like to build more from the perspective of a computer system, while the enterprise and business process models describe more the nature of the business itself for which the system is built, and represent the environment of the built system [6]. This becomes more apparent in typical business type applications, for example, in banking systems, and it is not so important for embedded systems. The functionality of our vending machine, for example, is not really dependent upon any organizational processes that we would have to describe and model, because they will change when we install the vending machine. In contrast, for a banking application, the business process is more important. Here, the enterprise model would represent the concepts that are relevant for the banking domain and the banking software system, for example, accounts, bills, exchange rates, and forms that bank customers have to fill out, that are processed through the system. The enterprise and business process models concentrate on how these existing concepts affect the computer system and how the computer system is going to change the processes in the bank's organization when it is installed.

2.2.3 Structural Model

The structural model in the context realization defines the structural organization of the entities that are outside the scope of the developed system

Table 2.2. Definition of the use case *Purchase Item* from the vending machine usage model

Use Case 1	Purchase Item
Goal in Context	Main scenario for purchasing an item from the vending machine.
Actors	User.
Trigger	User inserts coins into slot, or selects item on button panel.
Preconditions	Vending machine is operational.
Postconditions on Success	Item is provided.
Postconditions on Failure	Item is not provided.
Description Basic Course	1. User inserts sufficient amount of money. 2. User selects item on panel. 3. Selected item and return money are dispensed. 4. User takes item and returned money, if applicable.
Description Alternative Courses	1. User selects item. 2. Vending machine displays price. 3. <basic course>
Exceptions	1. [insufficient amount] user inserts more cash, or aborts. 2. [selected item not available] user selects different item, or aborts.
NF-Requirements	1. Item should be dispensed not more than 3 seconds after selection. 2. After 3 sec of no user selection use case is aborted and inserted cash returned.
Extensions	<left open>
Concurrent Uses	<left open>
Revisions	<left open>

but have yet an effect on the system. This comprises objects that the system requires as input, or produces as output, items that are somehow further processed, and other associated systems with which the subject system needs to interact to achieve its task.

Our vending machine in its simplest form will not have any external structure that we have to take into account in our context realization models. However, if we plan to develop a bit more advanced variants of our vending machine, for example, one that accepts credit cards, or another to be placed in hotel lobbies that accepts door key cards, so that items bought are booked

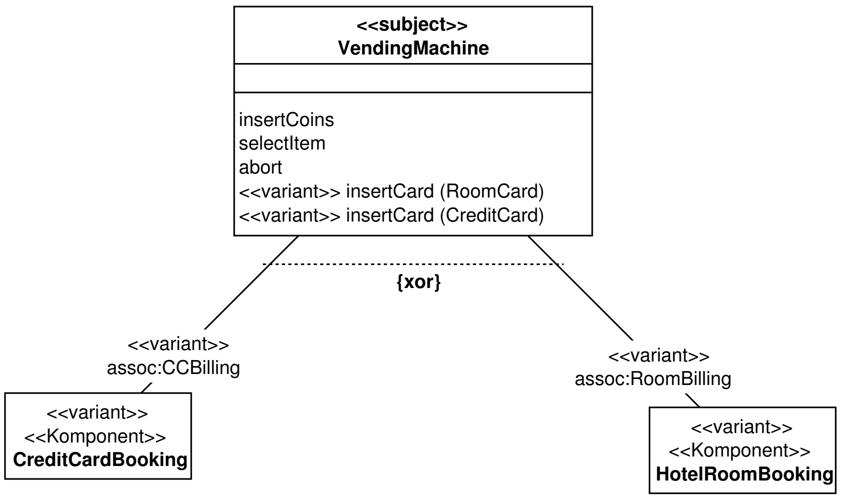


Fig. 2.7. Context realization structural model for the vending machine context

directly on the room account, we might have to indicate it in the context realization. Figure 2.7 displays the context realization structural model for such vending machine variations. Here, we have two systems with which the vending machine has to interact, and this affects the interface of the vending machine. We can have a connection either to the hotel booking system or to a credit card billing system, or neither connection. In the first case the vending machine will send a room identification record that includes the price and the room number. This is indicated through the ≪variant≫ `RoomBilling` association class in the diagram. In the second case we will have some credit card checking and billing indicated through the ≪variant≫ `CCBilling` association class in the diagram. In the third case we won't have any such connections. Currently, at such a high level of abstraction, we are not interested in how this will be implemented in detail. We include it in the context realization to indicate some additional features that we would like to have, so that we will not forget it later when we come to realizing the vending machine.

2.2.4 Activity and Interaction Model

Activity models in the context realization can be used to describe the procedures that the actors of the system may perform at its boundary. UML activity diagrams are very similar to traditional control flow graphs, although they are used on a much higher level of abstraction. Each rounded box in the activity diagram does not represent a block of source code statements as in a control flow graph, but an activity that the actor, for instance, a user, performs with the system. Figure 2.8 shows the context realization activity diagram that may be derived from the use case description in Table 2.2.

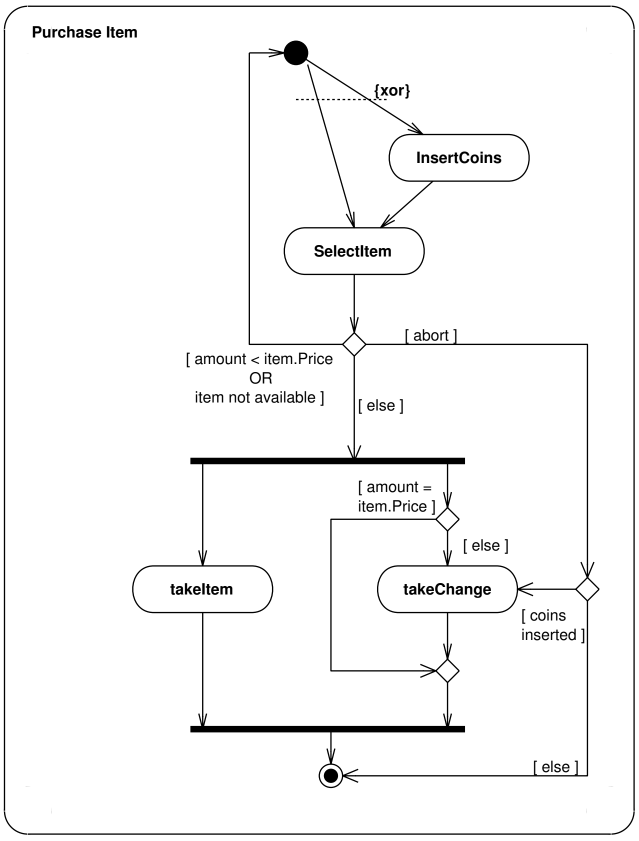


Fig. 2.8. Context realization activity model for the vending machine context according to the use case description for *Purchase Item*

We have two alternatives at the beginning of the diagram, either select an item and have its price displayed, or immediately insert coins and then choose an item to buy. This corresponds to the first entry of the basic course and the alternative course in the use case description. On the main path in the activity diagram, which relates to the basic course after the selection in the use case description, the user can take the item and the change if applicable. These two activities are concurrent because their sequence does not matter. The other paths in the activity diagram represent the different exceptional courses described in the use case definition.

We should always try to map the consecutive diagrams to previously created models. That way, we can identify and trace the items that we are talking about through the modeling process from higher-level to lower-level abstractions. For example, the use case description is very good for discussing things with the customer of our system, or with the business people in our own organization. In contrast, activity diagrams are already moving the subject toward more technical abstractions that may be more difficult for non-technical stakeholders to understand. But an activity model like the one displayed in Fig. 2.8 is much closer to subsequent modeling activities. We have only to refine this a little bit, and to add some artifacts, and we have a design diagram for our vending machine.

This strategy of small steps helps to keep the semantic gaps between the different abstraction levels at a minimum. And this is in fact why models are so useful for system development. We have only to add little bits and pieces, one after the other, and thus evolve the system recursively until we reach the point where we can implement it. The opposite strategy (and this sounds very familiar to me because it is common practice) is to take the use case descriptions and somehow cast them into source code.

Another model that may be used in a context realization is the interaction model. An interaction model can be useful to present a different view on the subject under consideration. The interaction model for the main user activity with the vending machine is displayed in Fig. 2.9.

The next step in the development cycle, after we have described the context of our system, is to define the top-level component that will be integrated in the context. This top-level component is in fact the entire system that we are going to build: the vending machine. Since component and system are interchangeable terms in component-based development, we cannot really draw the border between the two concepts. Somebody's component will almost always be somebody else's system, and somebody's system will almost always be somebody else's component. It depends on the point of view. Every component is described through a number of artifacts as represented in Fig. 1.3 in the Introduction.

The two most prominent items in the KobrA method that contain the other descriptive parts are the component specification and the component realization. Every KobrA component, except the context realization consists of these two items. The context realization, as the name implies, only comprises a very special component realization. The component realization of a superordinate component must always map exactly to the component specification of the subordinate component that the superordinate component contains. Otherwise they cannot have any meaningful interaction. In traditional system development, the modules are all custom designed to fulfill this requirement. With component reuse in place we will be quite unlikely to satisfy this requirement, because it will be difficult, if not impossible, to find a third-party component that exactly matches the required interface of the integrating component. We will be able to find existing components that match the requirements of an integrating component only to a certain extent. How existing components are reused is described in more detail in Subsec. 2.5.3.

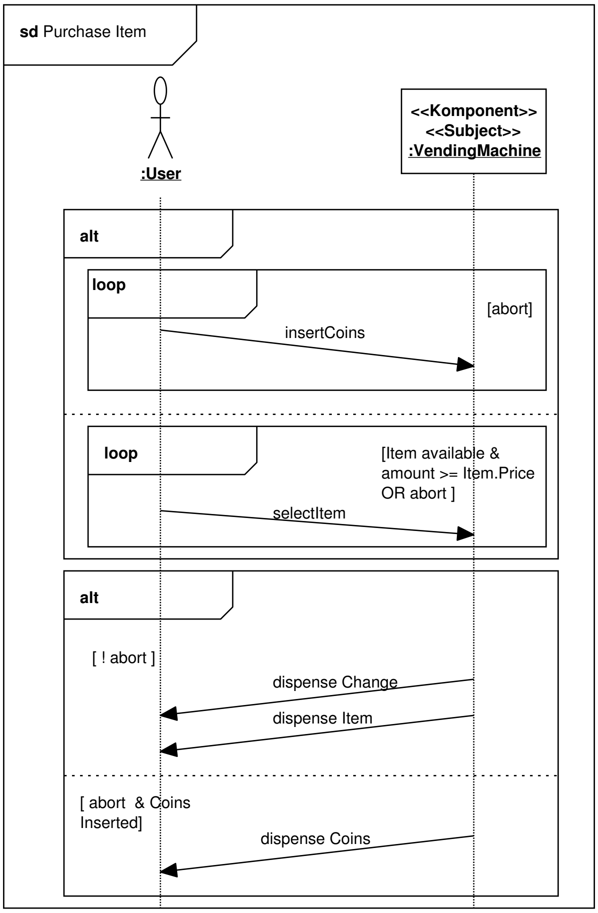


Fig. 2.9. Context realization interaction model for the vending machine context according to the use case description for *Purchase Item*

2.3 Component Specification

A component specification is a collection of descriptive documents that define what a component can do. Typically, each individual document represents a distinct view on the subject, and thus only concentrates on a particular aspect of what a component can do. A component specification may be represented by natural language, or through graphical representations, such as the UML, and formal languages, such as the Object Constraint Language (OCL) defined in the UML standard [118], or the QoS Modeling Language for specifying quality-of-service criteria [64]. Whatever notation is used, a specification should contain everything that is necessary to fully use the component and understand its behavior. As such, the specification can be seen as defining the provided and required interface of the component. Therefore, the specification of a component comprises everything that is externally knowable

- about its structure in the form of a structural model, and this comprises the other associated components, at its provided interface as well as at its required interface,
- about its function in the form of a functional model, and this comprises its provided and required operations,
- about its behavior in the form of a behavioral model, and this comprises its pre and postconditions.

Figure 2.10 illustrates these parts. Not all of them are mandatory, and they may change from project to project or from component to component; the picture in Fig. 2.10 represents rather a complete framework for a component specification. These artifacts are described in detail in the following. Additionally, a specification should comprise non-functional requirements; these represent the quality attributes stated in the component definition in Fig. 1.3. They are part of the quality assurance plan of the overall development project or the specific component. The quality assurance plan is summarized in Sect. 2.7, but this book is essentially about a sound quality assurance plan for the KobrA method that focuses on testing as a dynamic quality assurance technique. A complete documentation for the component is also desirable, and a decision model that captures the built-in variabilities that the component may provide. Such variabilities may be supported through configuration interfaces, but they are not considered here. The specification of a new component always represents a step down in the decomposition dimension in Fig. 2.2.

2.3.1 Structural Specification

The structural specification defines operations and attributes of the considered subject component, and the components that are associated with the subject (e.g., its clients and servers), as well as constraints on these associations. This is important for defining the different views that clients can have on the subject. A structural specification is traditionally not used in software projects. Only the advent of model-driven development approaches has increased its importance as specification artifact. As a UML class or object model, the structural specification provides a powerful means for the definition of the nature of the classes and the relationships by which a component interacts with its environment. It is also used to describe any structure that may be visible at its interface [6]. Figure 2.11 displays the specification structural model for the vending machine component. It is different from the context realization structural model since the focus is here on the system itself rather than its environment. The diagram is very similar to the containment hierarchy in Fig. 2.11 because the externally visible structure of the vending machine reflects the way in which the system may be ideally decomposed. The definitions of

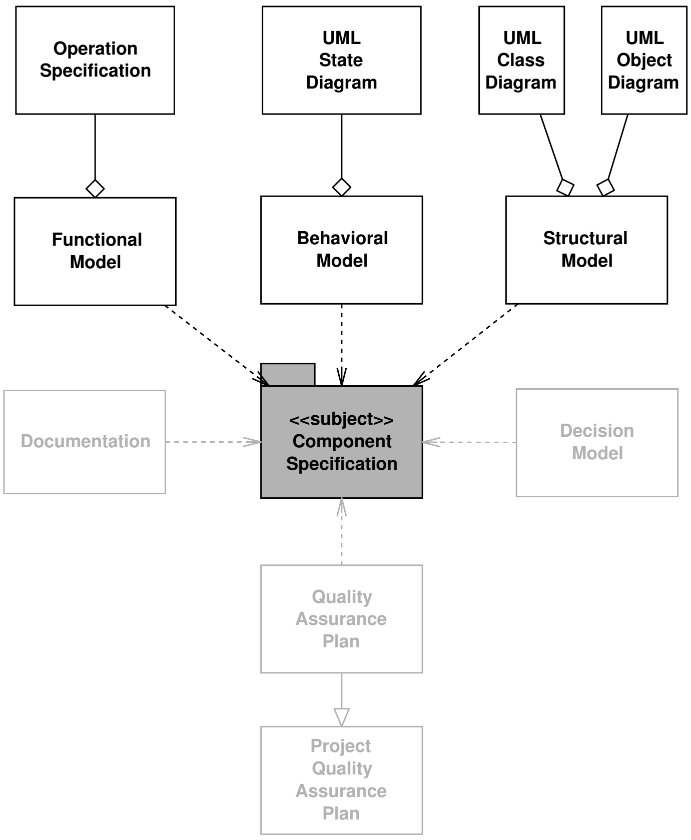


Fig. 2.10. Main descriptive artifacts in a component specification

the operations of the vending machine differ slightly from the original context realization structural model, in that we can now be more concrete about the component implementation. The specified operation `Abort` in the context realization will be implemented through the operation `SelectItem` with `Item=Abort` as input parameter. We can indicate that through the comment in the class `VendingMachine`. In order to maintain consistency between the two models, we may either change the context realization model after this decision, and indicate the implementation through a comment in the model, or amend the vending machine realization model that describes the implementation of this interface. This would map the operation `Abort` to the operation call `SelectItem(SelectionType Item=Abort)` where `Abort` represents a predefined key for this command. In any case, we have to go back to the other models and change them according to our recent design decisions.

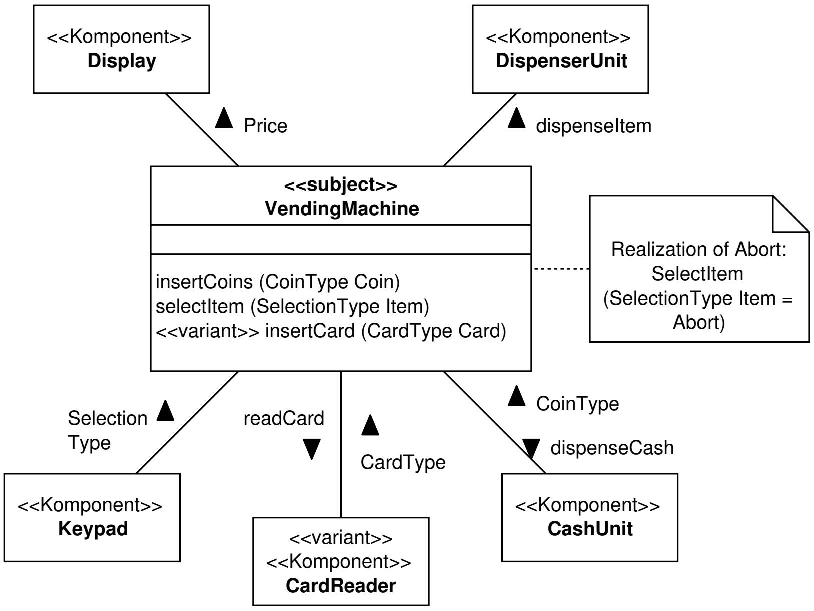


Fig. 2.11. Specification structural model for the vending machine component

2.3.2 Functional Specification

The purpose of the functional specification is to describe the externally visible effects of the operations supplied by the component. The collection of these operations is its provided interface. A template for a complete functional specification for one single operation of a component is depicted in Table 2.3. This template is used by the KobrA method, and has been initially proposed by the Fusion method [33]. The most important items on the list in Table 2.3 are the `Assumes` and `Result` clauses, which represent the pre and postconditions for the operation. These are essential for testing its correctness. The `Assumes` clause, or the precondition, defines what must be true for the operation to guarantee correct expected execution, and the `Result` clause, or the postcondition, describes what is expected to become true as a result of the operation if it executes correctly. It is possible to execute the operation if its `Assumes` clause is false, but then the effects of the operation are not certain to satisfy the postcondition. This corresponds to Meyer's design-by-contract principles [112]. The basic goal of the `Result` clause is the provision of a declarative description of the operation in terms of its effects. This means, it describes what the operation does, and not how it does it. Pre and postconditions typically comprise constraints on the inputs provided, the outputs provided, the state before the operation invocation (or its initial state), and the state after the operation invocation (or its final state) [6, 111]. Any notation for expressing pre and postconditions may be used depending on the intended purpose and domain of the system. A natural choice for expressing them may be the OCL. In the KobrA method, the functional specification is represented by a collection of all operation specifications that are defined by the template depicted in Table 2.3.

Table 2.3. Operation specification template according to the KobrA method and Fusion

Name	Name of the operation
Description	Identification of the purpose of the operation, followed by an informal description of the normal and exceptional effects.
Constraints	Properties that constrain the realization of and implementation of the operation.
Receives	Information input to the operation by the invoker.
Returns	Information returned to the invoker of the operation.
Sends	Signals that the operation sends to imported components (can be events or operation invocations).
Reads	Externally visible information accessed by the operation.
Changes	Externally visible information changed by the operation.
Rules	Rules governing the computation of the result.
Assumes	Weakest precondition on the externally visible states of the component and on the inputs (in receives clause) that must be true for the component to guarantee the postcondition (in the result clause).
Result	Strongest postcondition on the externally visible properties of the component and the returned entities (returns clause) that becomes true after execution of the operation with a valid assumes clause.

2.3.3 Behavioral Specification

The object paradigm advocates the encapsulation of data and functionality in one single entity, the object. This is one of the most fundamental principles of object technology. It leads to the notion of states, and the transitions between states, that typically occur in objects when they are operational. The component paradigm subsumes the principles of object technology as discussed before, and therefore it is based on exactly these principles as well.

Table 2.4. Example operation specification for *VendingMachine::SelectItem*

Name	SelectItem
Description	User selects an item to buy.
Constraints	<none>
Receives	`SelectedItem`.
Returns	`SelectedItem.Price` OR `SelectedItem`.
Sends	`SelectedItem.Price` to the Display OR/AND `DispenseItem` to the Dispenser.
Reads	`Amount` from the CashUnit.
Changes	Number of items in the DispenserUnit.
Rules	If [$CoinsInserted$] AND $SelectedItem = Abort$ dispenseChange; If [$Item.Empty$] display empty; If ([$Idle$]OR[$Amount < Item.Price$]) AND [!$ItemEmpty$] display `SelectedItem.Price`; If [$CoinsInserted$] and [$Amount >= Item.Price$] dispenseItem();
Assumes	Vending machine is `Idle` OR `CoinsInserted`.
Result	Item AND/OR Change (if applicable) dispensed; OR Item Empty/Item.Price displayed.

Components may have states. If a component does not have states, it is referred to as functional object or functional component, meaning it has no internal attributes that may be exhibited through its provided interface. In other words, a pure functional component does not exhibit externally visible states and transitions. It may, however, have internal states that are not externally visible, for example, states that could be exhibited by subsequent contained components.

The purpose of the behavioral specification (or the behavioral model) is to show how the component behaves in response to external stimuli [6]. It concentrates on the `Assumes` and `Result` clauses of the functional specification that define the pre and postconditions of an operation. If the component has no states, then the pre and postconditions do not define an initial state for which an operation invocation is valid, or a final state in which an operation invocation results. In this case, we are only concerned with distinct input with which the operation may be called that results in a distinct output of the operation call. For this, we do not necessarily need a behavioral model. The cohesiveness of functional objects is entirely arbitrary, because in object technology cohesion is defined through the data that the operations mutually access and change. Object technology is therefore focused on data cohesion while in traditional development we may refer to functional cohesion, meaning functions that have a similar purpose are grouped into components or modules. In any case, if the component is based on states, and most components are, the behavioral model expresses a great deal of the complexity in the pre and postconditions that are inherently defined through the collection or combination of all operation specifications.

The behavioral specification (or behavioral model) describes the behavior of the objects or instances of a component in terms of their observable states, and how these states change as a result of external events that affect the component instance [16, 111]. A state is a particular configuration of the data values of a component's internal attributes. A state itself is not visible. What is visible, or externally observable, is a difference in behavior of the component from one state to another when operations are invoked. In other words, if the same message is sent to a component instance twice, the instance may behave differently, depending on its original state before the message is received. A transition, or change from one state into another, is triggered by an event, which is typically a message arrival. A guard is a condition that must be true before a transition can be made. Guards are used for separating transitions to various states that are based on the same event [111]. Guards in state models are similar to decisions in traditional control flow graphs. They define input parameter domains for which the component behaves differently, i.e., switches into a different state, or performs different operations. The behavioral specification in KobrA is represented by one or more UML state diagrams or state tables. Figure 2.12 displays the specification behavioral model for the vending machine component, and Table 2.5 displays the corresponding state table. Both forms contain more or less the same information. The first form is easier to read and understand, and the second form is better for automatic processing and deriving test cases.

2.4 Component Realization

A component realization is an accumulation of descriptive documents that collectively define how a component is realized. A realization should contain everything that is necessary to implement the specification of a component. A higher-level component is typically realized through a combination of lower-level components (through composition) that are contained within it and act as servers to the higher-level component. Additionally, the realization describes the items that are inherent in the implementation of the higher-level component. This is the part of the functionality that will be local to the subject component, and not implemented through subcomponents. In other words, the realization defines the specification of the subcomponents, this is the expected interface of the component, and it contains additionally its own implementation. Its own implementation corresponds to its private

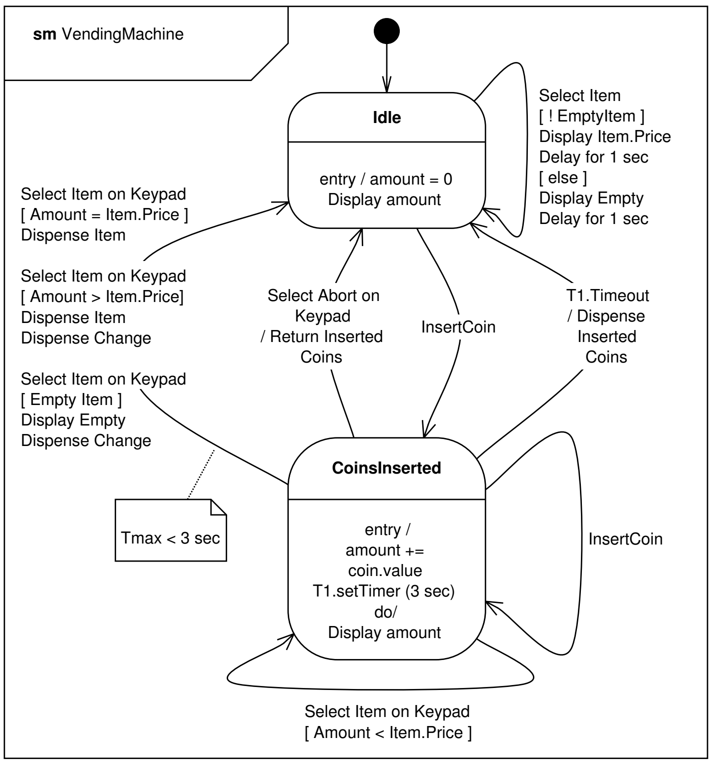


Fig. 2.12. Behavioral specification of the vending machine in the form of a statechart diagram

design that the user of the component does not see. Hence, a component realization describes everything that is necessary to develop the implementation of the specified component. This comprises the specifications of the other server components on which the subject component relies, in other words how its required interface is implemented, as well as its internal structure, and the algorithms by which it performs its specified functionality. Therefore, the realization comprises documents for specifying a component's

- internal structure through a structural model in the form of UML class and object diagrams,
- the algorithms by which it calculates its results through an algorithmic model in the form of UML activity diagrams, and
- its interactions with other components through an interaction model in the form of UML sequence and collaboration diagrams.

After we have specified a component's expected interfaces with other subordinate components can we start implementing it internally. I have summarized

Table 2.5. Behavioral specification of the vending machine in the form of a state transition table

Initial State & Precond (Guard)	Operation, Transition	Result	Final State & Postcond
Idle AND [!*EmptyItem*]	SelectItem (Item)	Display Price	Idle
Idle AND [*EmptyItem*]	SelectItem (Item)	Display Empty	Idle
Idle	InsertCoin (Coin)	Display Amount	CoinsInserted
CoinsInserted	InsertCoin (Coin)	Display Amount	CoinsInserted
CoinsInserted AND [$Amount < Item.Price$]	SelectItem (Item)		CoinsInserted
CoinsInserted AND [$Amount == Item.Price$]	SelectItem (Item)	dispense Item	Idle
CoinsInserted AND [$Amount > Item.Price$]	SelectItem (Item)	dispense Item dispense Change	Idle
CoinsInserted [*EmptyItem*]	SelectItem (Item)	Display Empty Dispense Change	Idle
CoinsInserted [$Item == Abort$]	SelectItem (Item)	Dispense Cash	Idle

the artifacts in a component realization in Fig. 2.13, and explain them in more detail in the next subsections.

2.4.1 Realization Structural Specification

The purpose of the realization structural model is to describe the nature of the classes, and their relationships, of which the component is made (i.e., its subcomponents), and the internal architecture of the component. In general, the structural model consists of a number of class and object diagrams [6]. Realization class diagrams describe the classes, attributes, and relationships between the classes of which a component is made. The component that is the focal point of the diagram is augmented with the stereotype ≪subject≫. Figure 2.14 displays the UML class diagram representing the internal structure of the `VendingMachine` component. The realization structural model is typically a refinement of the specification structural model, so it contains a superset of the information represented in the component specification; all elements that

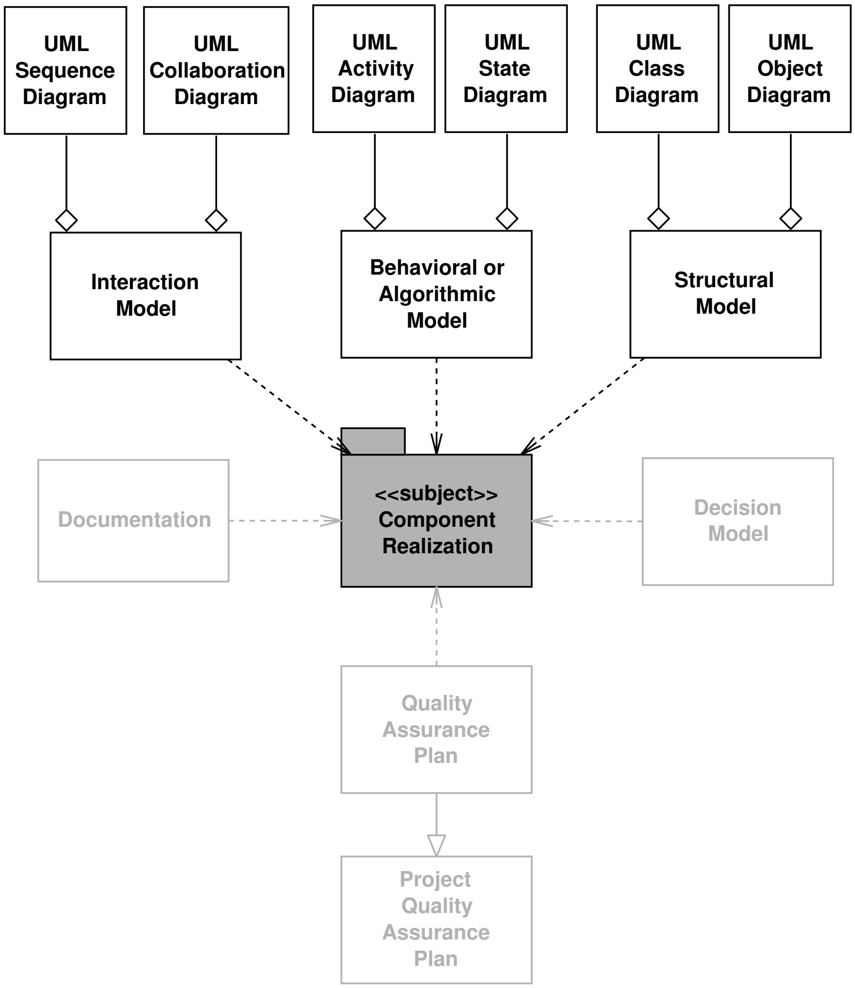


Fig. 2.13. Main descriptive artifacts in a component realization

are displayed there are also relevant to the realization, but are described here in more detail. Additionally, the realization comprises elements that are not visible at the specification level, since they are not important for using the component properly.

Figure 2.14 also displays the subordinate components of the subject indicated by the stereotype ≪Komponent≫, but there are additionally some classes that make up the internal implementation of the `VendingMachine`. Komponent stands for KobrA component [6]. We have a timer that triggers the return of change after some time in which the user shows no activity. We need a list to keep track of the items that have been sold and that are still available in the dispenser. This can be seen as an internal representation of the externally implemented real world, so important for embedded systems. The size of the list depends on the size of the dispenser, so we have a dependency association with the dispenser. We also need a list of possible coins that may be accepted by the `CashUnit`. And this is dependent upon the type of the `CashUnit` that we are going to buy from an external provider. For some of the subsequent components we can also begin already to define the expected interface. But this is a highly iterative process, and it depends heavily on which third-party component implementations we will find. According to their specifications, we have to change the required interface definition of the subject. For some component interactions we can be more concrete, in particular for those that invoke operations on the subject component; for others we can be only less specific. The definitions will be refined in subsequent modeling iterations when we have a clearer idea of how we will implement them.

2.4.2 Realization Algorithmic Specification

The algorithmic specification comprises a number of activity specifications that describe the algorithms by which the operations of a component are implemented. Figures 2.15 and 2.16 show the UML activity diagrams for the operations `insertCoin` and `selectItem` of the `VendingMachine` component. Some of the activities will have to be refined during later modeling phases, as soon as the collaborations with other components are determined more concretely. For example, we have specified that some signals are going to invoke operations, but we have still not specified how the signals will be handled and processed. Therefore, some of the activities in the diagram will map to additional private procedures. We will then have to include these procedures in the other models, for instance, in the realization structural model of the `VendingMachine`. The next step would be to define these procedures in the same way, as part of the realization algorithmic model. Activities that are not operations of a component may also be refined and specified through activity specifications that take the form and shape of operation specifications given in the template in Table 2.3. We could also define the operations in a programming language such as Java. In this case, we would come up with the most concrete representation of an operation, and we would have essentially completed the embodiment step for this particular item. But it would constrain us with respect to how we will have to implement the other components, and this is not really desirable in such early modeling phases.

2.4.3 Realization Interaction Specification

Activity diagrams provide a flowchart-like picture of the algorithm for an operation, and thus emphasize flow of control. Interaction models display similar information, but from the perspective of instance interactions rather than control flow [6]. Interaction diagrams describe how a group of instances collaborates to realize an operation, or a sub-activity of an operation. Figure 2.17 displays a UML collaboration diagram for the activity

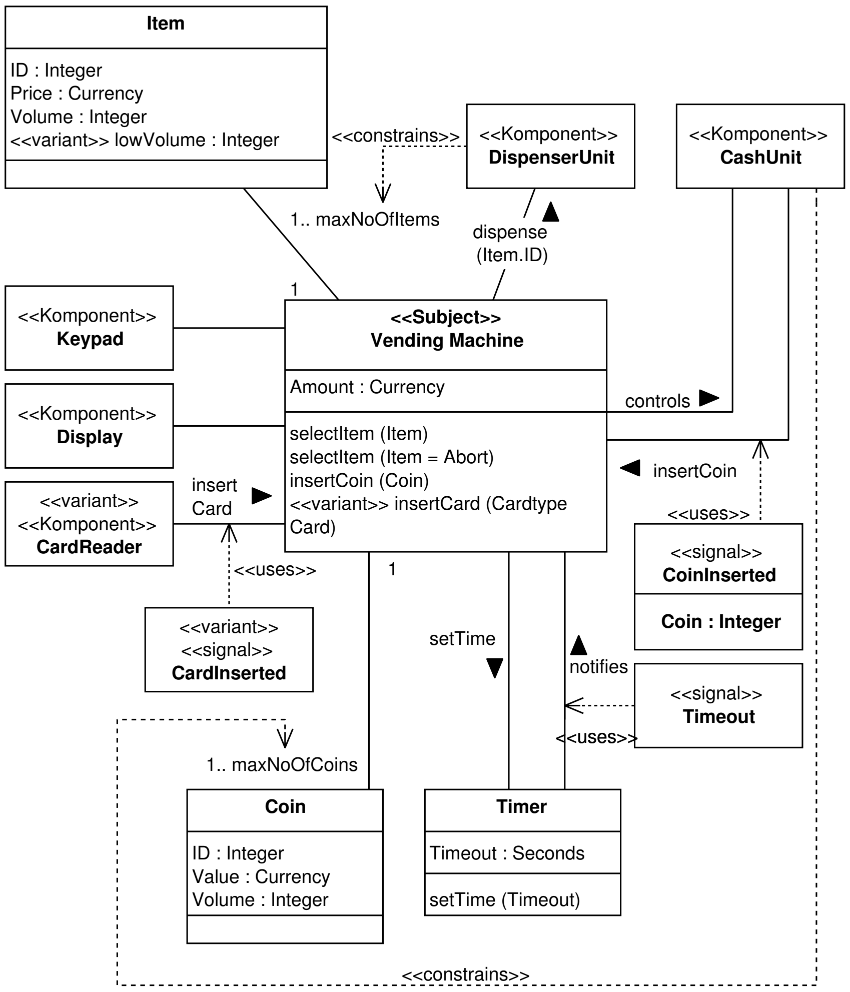


Fig. 2.14. Realization structural model of the *VendingMachine* component

`VendingMachine.insertCoin` and Figure 2.18 displays a UML collaboration diagram for the activity `VendingMachine.selectItem`. For small activities it is typically not necessary to create both, activity diagram and interaction diagram, because the algorithm may be quite clear. For larger activities it is often helpful to have both views available.

We have now created the first design models of our top-level component `VendingMachine`. For some of the subordinate components we have more concrete ideas of what they are supposed to look like; for others we are less concrete. The next step is to search for existing components that fulfill the relevant requirements of the expected interfaces of the `VendingMachine` component described in the realization models. Since we play the role of a vending

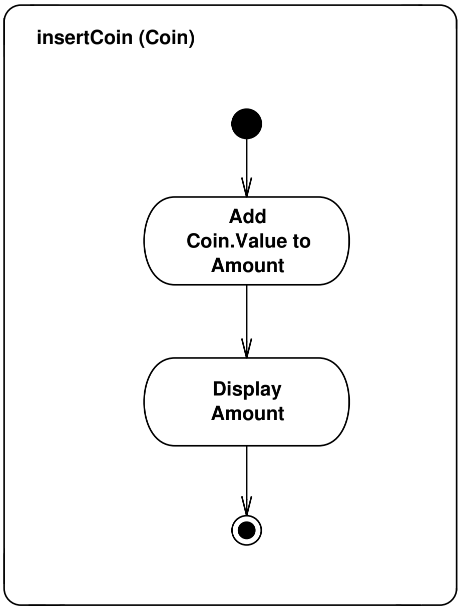


Fig. 2.15. Realization algorithmic model for the operation *insertCoin* of the *VendingMachine* component

machine provider, we assume some knowledge of our domain, and we expect to find the components that we have specified in the containment tree (Fig. 2.3). Typically, the next step is to integrate the components found into our existing model. This corresponds to an embodiment step, and additionally, a composition step as described in Sect. 2.1. Later, we will have a look at how real systems are devised from abstract models. Now, I will present only a brief overview. We will have a closer look at the embodiment activity in Chap. 5, which deals with implementation technologies.

2.5 Component Embodiment

Embodiment refers to all activities that aim at turning an abstract representation of our system, for example, a model of a component, into more concrete representations of that model, for example, an executable and deployable physical component. This corresponds more to the implementation activities and products of the traditional waterfall and V-model, while the more abstract component modeling activities along the other dimensions correspond more to the analysis and design products and activities that the waterfall and V-model define [41, 124]. Under the KobrA method, the entire system that will be developed is represented as a single component at the highest possible level of decomposition, and as a single component model at the highest possi-

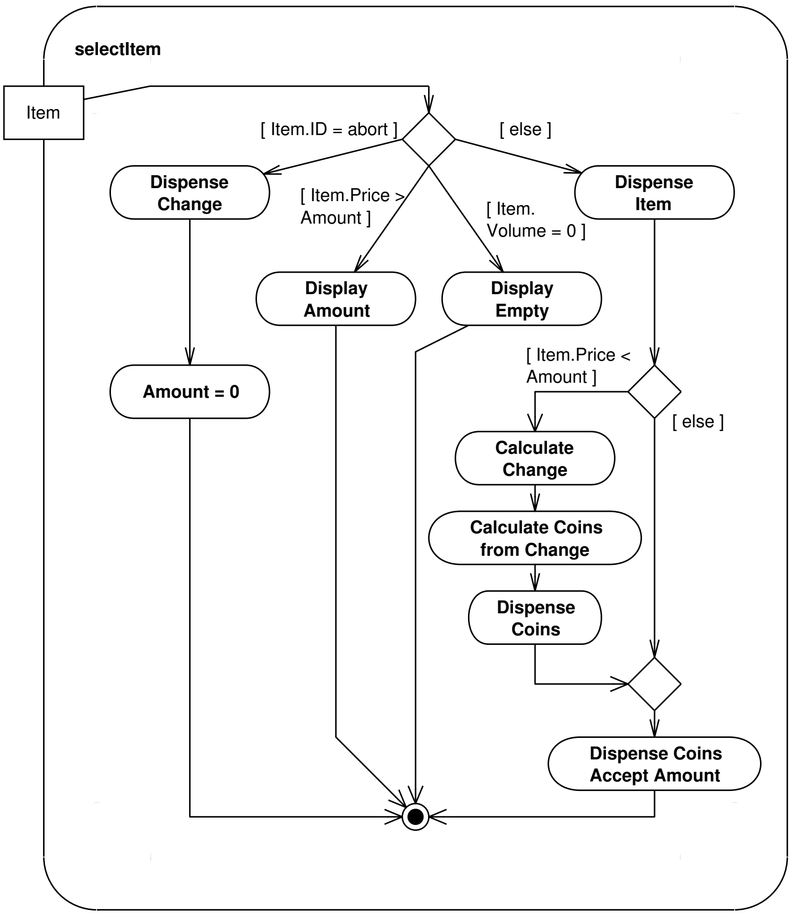


Fig. 2.16. Realization algorithmic model for the operation *selectItem* of the *VendingMachine* component

ble level of abstraction. So, before any software can be deployed and run, this high-level component must be transferred into lower-level components along the decomposition dimension as well as along the concretization dimension. Finally, the system can be brought into a binary form. These steps are illustrated in Fig. 2.19, and they involve modeling activities on lower levels of abstraction, as well as writing or generating code, invoking compilers, and employing configuration tools that, for example, essentially distribute the system over a number of nodes. The transition from the source code into a binary format is usually performed automatically through some translators, generators, compilers, linkers, etc. However, the transition from the model into the source code is much more difficult, and typically involves a quite extensive manual development effort. This transition is typically performed in a single

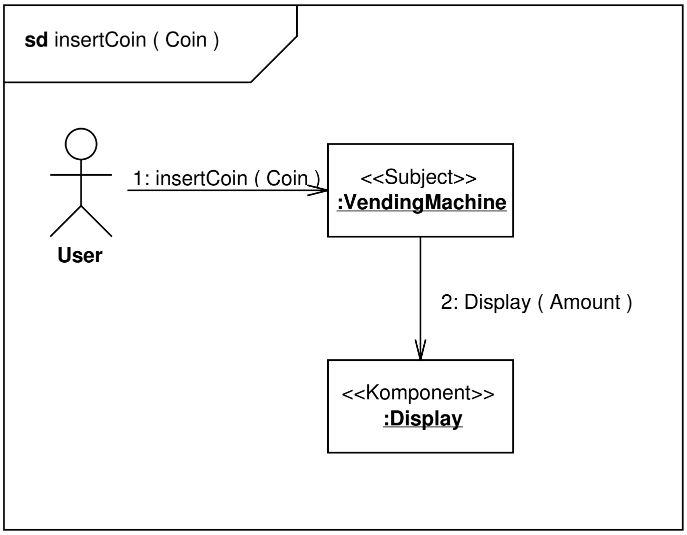


Fig. 2.17. Realization interaction model for the operation *insertCoin* of the *VendingMachine* component

step [26], as illustrated in Fig. 2.20, although it is not simply done by pressing a button and invoking a compiler.

The transition between model and code creates a semantic gap because it represents a change in the notation used. On one side we have the concepts of the UML, on the other side we have the concepts of the programming language used. In order to cross this gap, we have to translate the meaning of the concepts of the model into the appropriate concepts of the programming language. Initially, this does not seem to create any difficulties, because most object-oriented programming languages directly provide or at least support most of the UML's concepts. But the semantic gap becomes much more apparent if traditional procedural programming languages are used as the implementation technology. These support hardly any of the UML's primary concepts directly, because the UML is a notation that is heavily based on object technology. Even most object-oriented languages are quite diverse, and it is often difficult to exchange or even compare their concepts.

Performing the manual transformation directly from the model into the code, as suggested in Fig. 2.20, often requires implicit design decisions on the programming language level that may considerably deviate from what the model is actually defining. So, it is often difficult for different stakeholders of a system to understand that some source code is actually implementing a particular model. And this directly undermines the very principles of using and applying model-based approaches. The reason for this is that the modeling activities are typically performed only for high-level architectural specifications that provide a coarse-grained view of an entire system, and

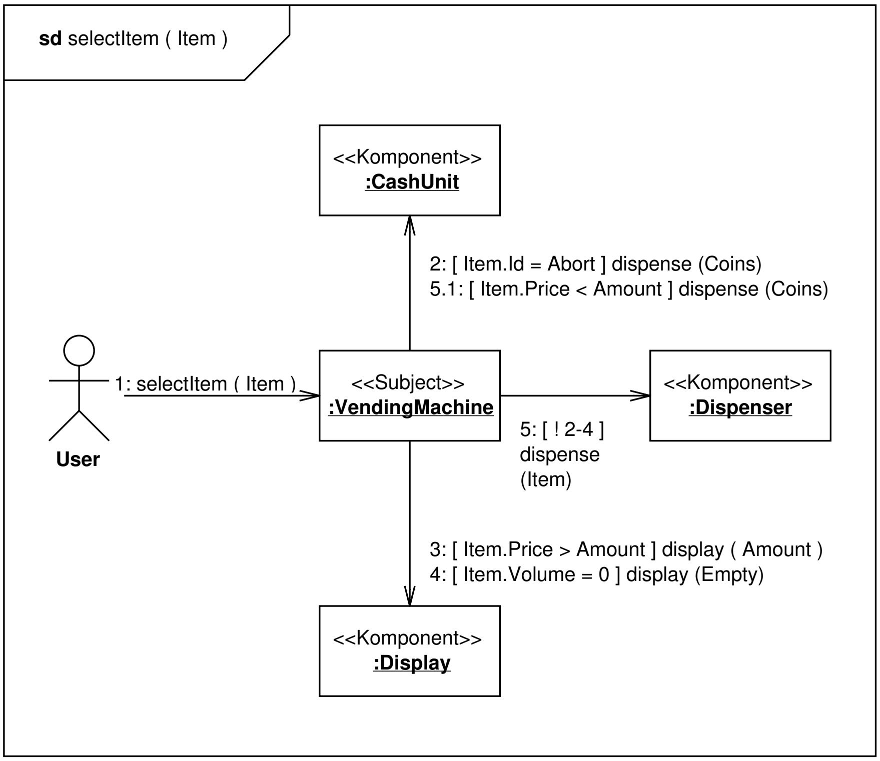

sd selectItem (Item)
<<Komponent>> :CashUnit
2: [Item.Id = Abort] dispense (Coins)
5.1: [Item.Price < Amount] dispense (Coins)
User
1: selectItem (Item)
<<Subject>> :VendingMachine
<<Komponent>> :Dispenser
5: [! 2-4] dispense (Item)
3: [Item.Price > Amount] display (Amount)
4: [Item.Volume = 0] display (Empty)
<<Komponent>> :Display

Fig. 2.18. Realization interaction model for the operation *selectItem* of the *VendingMachine* component

that the implementation is immediately based on such high-level design or analysis models [170]. Essential design decisions are therefore implicitly taken in the final source code but never documented in the models. This is because developers tend to intermingle the two separate and orthogonal activities, refinement and translation, and then have to deal with different representation formats and different levels of detail at the same time. In the next subsection I will briefly explain how the semantic gap between model and code may be bridged by separating these two concerns.

2.5.1 Refinement and Translation

In general, refinement is defined as a relation between two descriptions of the same thing or in the same notation. In other words, it is the description of the same thing at a level of greater detail [25]. Translation represents a relation between two different descriptions or notations usually at the same level of detail. Mixing these two different approaches in a software development project will quite likely lead to complex representations that are difficult to

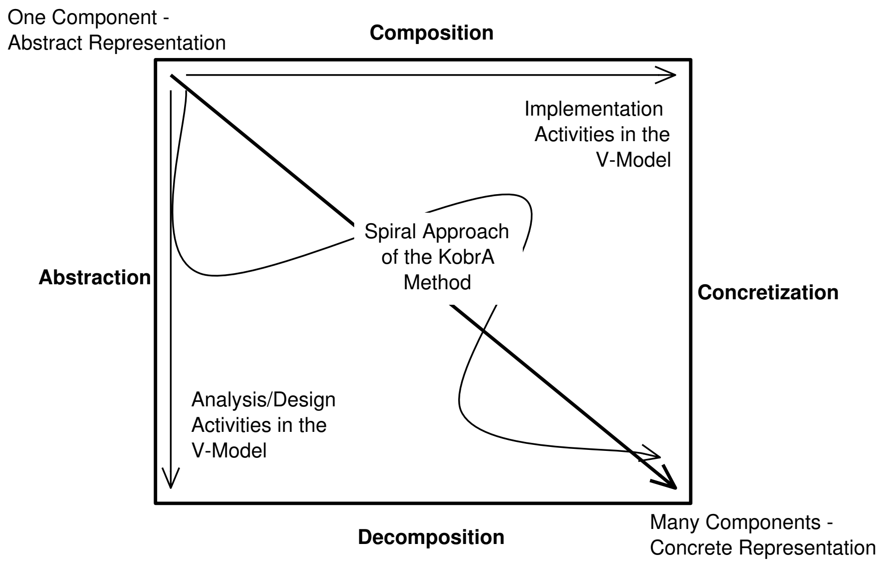


Fig. 2.19. Spiral approach of the KobrA method vs. approach of the waterfall/V-model

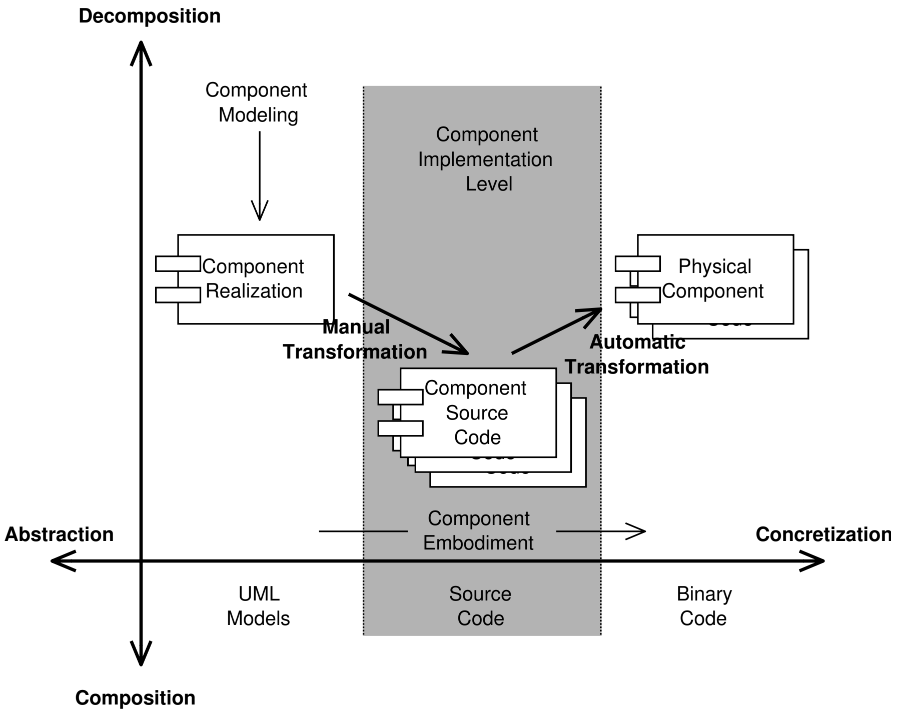


Fig. 2.20. Refinement and translation (Manual Transformation) in a single step

understand. In the worst case it will lead to a final system that does not implement its specification according to the models. Consequently, refinement and translation should be separated and performed as individual activities, as illustrated in Fig. 2.21. Here, the idea is to refine the existing models to a

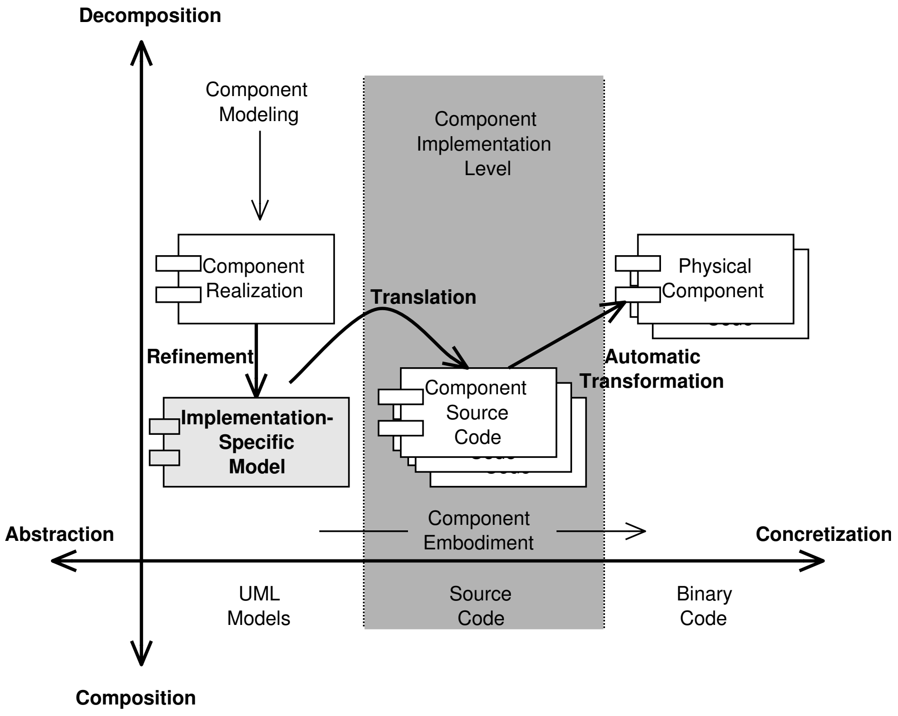


Fig. 2.21. Refinement and translation in two separated steps

predefined level of detail in a first step that can then be easily translated into code in a second step. This leads to the problem of defining the appropriate levels of detail that are suitable for translation into a particular programming language. This problem can be solved through implementation patterns such as the Normal Object Form that is introduced in the next subsection.

2.5.2 The Normal Object Form

UML profiles can be defined according to a number of different motivations, for example, a testing profile that adds testing concepts to the UML (this is described in more detail in Chap. 3). Implementation profiles are used to support the mapping of UML constructs into concepts of a programming language. A profile adds concepts to the core UML, together with associated constraints on these concepts, to define a set of modeling artifacts that are tailored specifically to support implementation in a particular programming language. For example, a Java implementation profile could be defined to capture Java's standard concepts in the UML, or a more exotic C implementation profile could be created to map UML concepts to C.

The Normal Object Form (NOF) [25, 27] represents such an implementation profile that maps the UML to the core concepts of object oriented-programming languages. It can be seen as a minimal set of artifacts for mapping the UML to any object-oriented programming language, and it should be refined to capture the more specific concepts of a particular object-oriented programming language, for example, Eiffel. As a UML profile, the NOF specification consists of [6]:

- a UML subset that contains modeling elements closely related to the implementation level,
- new model elements based on UML stereotypes,
- constraints on the use of existing and new modeling artifacts.

NOF models are valid UML models, with the difference that the UML comes only close to object-oriented programming language concepts, while NOF permits the UML to be used to write model-based programs in object-oriented languages. NOF models represent an object-oriented program in graphical form as if it would be written in source code. In fact, it represents a notation that, given the right tools, can be transformed automatically into the corresponding appropriate programming language. In this respect, it supports well the ideas of the Object Management Group's Model-Driven Architecture [79].

2.5.3 Component Reuse

Component reuse represents activities in both dimensions, composition/decomposition and abstraction/concretization. The normal case in reality is that the reused component will not be entirely fit for our expected interfaces, so component reuse typically involves

- changing our existing model to accommodate the deviating properties,
- changing the reused component in order to suit the integrating model, or
- coming up with an adapter that accommodates both views, that of our component and that of the reused component.

In the first instance we have to develop the required interface of our reusing component according to the provided interface of our reused component. But this really is bad practice, and it directly goes against the very ideas of component-based development. We would have to amend the `VendingMachine` component whenever we choose to integrate and deploy it with slightly differing other component implementations. In the second case, we would have to do the same for the reused component. Although in theory feasible, because we would have to amend the model of the reused component, this is normally much more difficult because, in particular for third-party components, we will not get any implementation-level representations but only binary code. In this case, we will fail in the embodiment dimension, so this is also not a solution. In the third case we have to insert another so-called adapter component that performs the mapping between the two deviating interfaces. For simple integrations this may only require a syntactic mapping of the interface operations, and the connection will work. Many of the contemporary component technologies provide such a syntactic mapping mechanism, but this is described in Chap. 5, where we will take a closer look at implementation technologies. For more complicated mappings, and this is by far the most common case, we will have to also build a semantic mapping between the two interacting entities. This translates what one component "means" into something that the other component can "understand." Such a wrapper component, that essentially encapsulates the server component is also called "glue code."

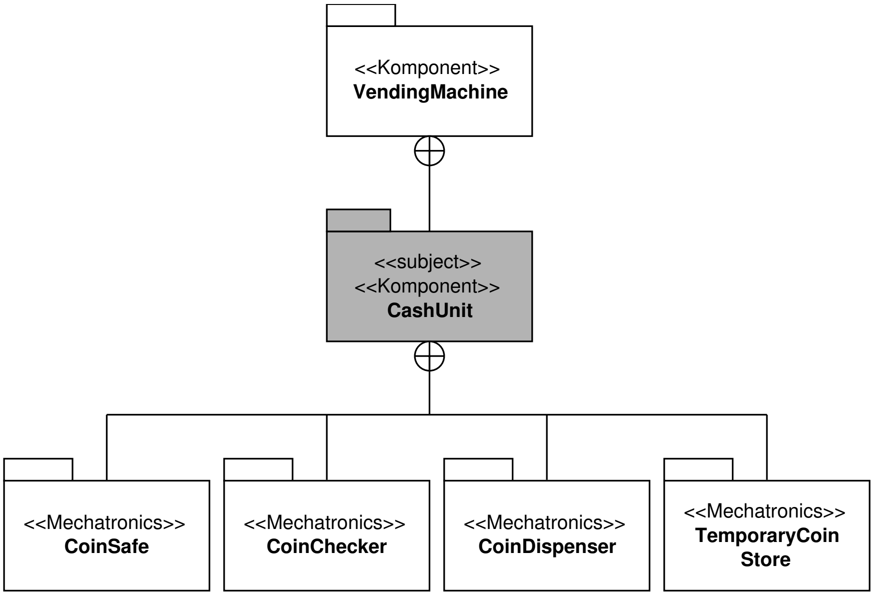


Fig. 2.22. Containment hierarchy for the *CashUnit*

If we do not find existing components, we have to realize them ourselves. In this case we have to decompose the missing components further until we find some matching implementations. Then, we can go back and try again to integrate them. For example, assume that for our `VendingMachine` we cannot find an appropriate commercial component that could act as our `CashUnit`. We have to see which other parts may be available on the market that could implement our specification of the `CashUnit`. If we have a look at the Web pages of relevant companies, we may come up with four subcomponents that are typically used for such a module. We can then refine the containment hierarchy to accommodate these components. This process is displayed in Fig. 2.22. Finally, we go through the same cycle of development steps for the `CashUnit` component that we have performed for the `VendingMachine` component. We start with the context of the `CashUnit`. In this case it is the component realization for the `VendingMachine` that we have already defined. When we have identified suitable third-party candidates that come close to our specification, we have to integrate the most suitable one into our overall component model. This activity is described in the next paragraph.

2.5.4 COTS Component Integration

The simplest way of reusing an existing component is possible if the specification of both the reusing and reused component are equivalent. This is the general case for any custom development where every contained subordinate component is crafted exactly according to the needs of the containing superordinate component. This scenario is displayed on the left hand side of Fig. 2.23. In contrast, the integration of third-party components usually deals with additional differences, syntactic as well semantic, in the required and the provided interfaces. This scenario is displayed on the right hand side of Fig. 2.23.

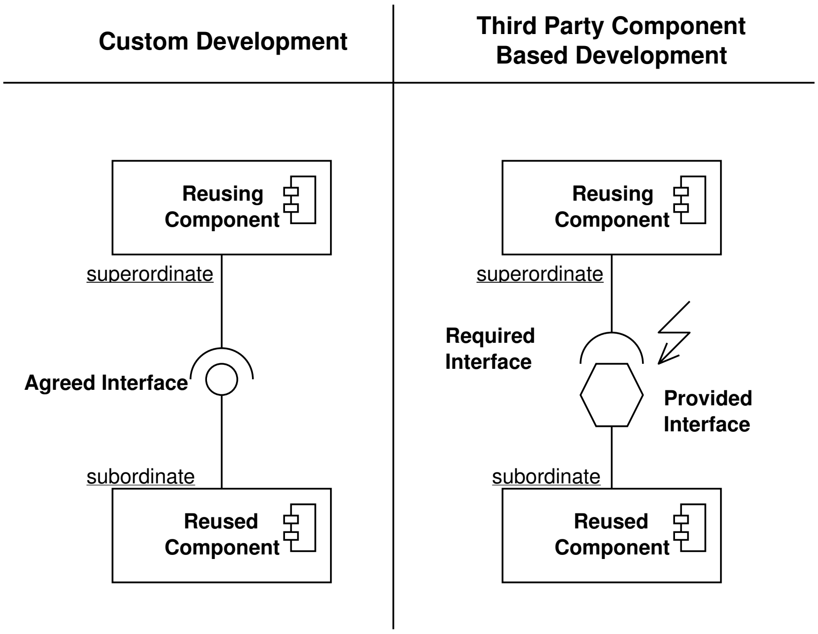


Fig. 2.23. Custom-designed component integration vs. third-party component integration

Component integration needs to take place at all abstraction levels of a development process. In other words, the description of the component needs to be adapted according to the description used throughout the development process. In our case, the descriptions are based on UML models. Consequently, in order to fully map what a third-party component offers to what our own integrating component requires, and to be able to compare both, we need to create a UML-compliant specification of the reused building block. This represents an integration effort at the abstract model level, and it is facilitated through a so-called conformance map [6]. The conformance map describes a COTS component's externally visible features in terms of a mapping between the notation of the reused component and the notation of our own development process. And it can only perform this mapping if the information provided by the documentation of the reused component is complete and correct with respect to its structural, behavioral, and functional information. For many COTS components this information may be present and usable, but it may well be distributed and organized in an odd way, so that it becomes difficult to perform the mapping. In general, the conformance map can be seen as a description of the COTS component according to our own prevailing notation, and it will comprise a component specification with its provided and required interfaces, and offer all relevant models that are needed to fully describe the COTS component and understand its behavior. We can only decide whether a component is fit for our purpose with full confidence if we have created the conformance map, and the required and provided interfaces for both components become directly comparable.

If we decide to use a component according to a positive evaluation of the conformance map, the next step is to devise a so-called semantic map. This concentrates on the similarities and differences of the two component specifications, and attempts to model a mapping between the two deviating interfaces. Basically, it describes how transactions from the reusing superordinate component must be translated into a form so that the reused subordinate component can understand it, and vice versa, so that both units will perform meaningful interaction in tandem. In reality, this amounts to the specification of a wrapper component that wraps around the original COTS component and encapsulates it. More generally, we may speak of a component adapter, that will be integrated between the two components, as providing an interface that is compliant with the superordinate component and requiring an interface that is compliant with the subordinate component. This adapter is a component in its own right and it is defined exactly according to the KobrA development principles that have been introduced in this chapter. Ideally, it will not be accommodated at the same level of abstraction as the two other functional components within the component containment hierarchy. It must be regarded more as belonging to a lower-level model further down in the concretization dimension, and it merely realizes the connection between two higher-level components; other than that it represents important functionality of the overall application. This process of COTS component integration is illustrated in Fig. 2.24.

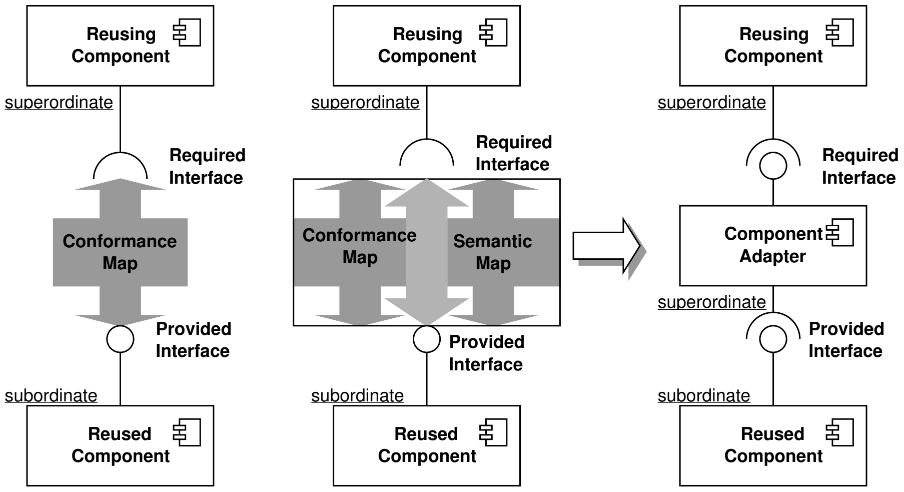


Fig. 2.24. Third-party component integration with adapter

2.5.5 System Construction and Deployment

There are two important steps missing to come up with a final system while moving down toward the end of the dimension that is associated with the embodiment activity: construction of the physical components and their deployment on the actual target platform. In the previous subsection, I have addressed the problems that are inherent in reusing and integrating existing components at a higher level of abstraction. There, we were still dealing with the model level, because with a COTS component we already have a physical deployable component. Here, I will give some details on how custom designed components are turned into physical executable artifacts if no appropriate third-party building blocks can be found. Development methods always follow the “divide and conquer” principle and encourage a logically organized architecture of a system. This is because development methods are centered around human needs, and the apparent lack of human capability of dealing with large and complex entities. There are many different ways of breaking a large system into smaller logical units that are individually more manageable. For deployment, any such logic organization is entirely meaningless. If a system will be a stand-alone package that runs on a single computer, it will quite likely be deployed as a single executable file. Logic components may be turned one-to-one into corresponding physical components, but usually there is no obvious reason for why physical components should follow this organization. If we deploy a typical component containment hierarchy on a component platform, such as Microsoft's DCOM, the platform will actually organize all components at the same level. It will ignore the nesting rules that the development method has applied. Although implicitly, if we execute such a system, it will abide by these nesting rules, because the top-level component will call services on lower-level components. But the hierarchy of these components on the deployment platform will be flat.

The construction of the final system and its deployment follows different rules, and so this activity is ideally separated from the implementation activities. Implementation effectively represents a flattening activity that results in a number of source files that may be compiled individually, and then linked with other compiled files. As a consequence, we can have different deployment scenarios:

- One or several logical components are transformed into one physical component. This would be a stand-alone executable deployed on a single computer, probably connected with other components on other computers.
- One logical component is transformed into several physical components. This means that functionality is distributed between nodes. The vending machine's `RoomBilling` component is an example of that. It will be separated into one physical component that will run on the physical vending machine and into another physical component that will run on the hotel's billing computer.
- Several logical components are transformed into several physical components. This is a combination of the previous two scenarios, and it represents the organization of typical distributed systems.

How logical components are organized in a physical system and how they are deployed on a platform is heavily dependent on the type of the system and the underlying deployment technologies used. I cannot give concrete guidelines on how a system should finally be constructed and deployed. UML deployment diagrams represent a suitable way of expressing and defining how logical components from KobrA's containment hierarchy should be distributed over real platforms, but this is not considered any further.

2.6 Product Family Concepts

Up to now, we have treated component-based system engineering as the ultimate way of realizing reuse. Here, components are seen as the primary reusable assets. We can either deal with components that have been developed to be used in a particular system (i.e., custom developments), and are found later to be suitable also for another system, or we can deal with generic components that have been developed with the idea of being used in a number of different applications from the beginning. Components of the first kind are assets that have been developed in an organization and maintained over some time, and that are used over and over again in different contexts. This is probably the most commonly understood way of software reuse to date. Components of the second kind usually have a much broader range of deployment, and they have been specifically purchased to be reused in a single application, or in a number of different applications. COTS components are typical representatives of this second kind.

In contrast, a product family elevates the principles of reuse to a much coarser-grained and more abstract level. Devising a product family is also termed product line engineering, and this in fact represents reuse on an architectural level. A product line or a product family is a generic system or a generic component framework that can be extended or adapted to instantiate a suite of similar final products. Product line engineering deals with exploiting the commonalities of similar systems and managing their differences, and it is separated into two distinct activities, framework engineering and application engineering.

Product line engineering is actually a way of organizing reuse for custom designed components. Most software organizations will operate only in a specific domain. So, most software components will be similar in most of their final products. For example, in the vending machine domain we will always have to deal with dispensers, checkers, and button panels that will reappear in all vending machine products, maybe organized in a different way or with slightly different functionality. Product line engineering organizes the common features of a vending machine in terms of a common core and supports the development of specific vending machine variants out of that common core in a certain way. This is illustrated in Fig. 2.25.

The following subsections introduce the two main activities of product line engineering, framework and application engineering, in more detail. But initially we will have a look at the mechanisms by which decisions are managed, and through which we can instantiate a concrete application from a generic framework.

2.6.1 Decision Models

Product line engineering concepts only make sense if an organization develops a number of similar products that are belonging to the same domain. This implies that all products have at least some features in common [6]; otherwise it would be difficult to say how these products may be similar. The domain of the products is consequently defined through these commonalities. Variabilities are features that may change from product to product. Simply defining what is common and what will be variable in a product line is not enough, because it does not state which variable features will be associated with which concrete product. Establishing this association is the role of the decision model. For each decision, it defines

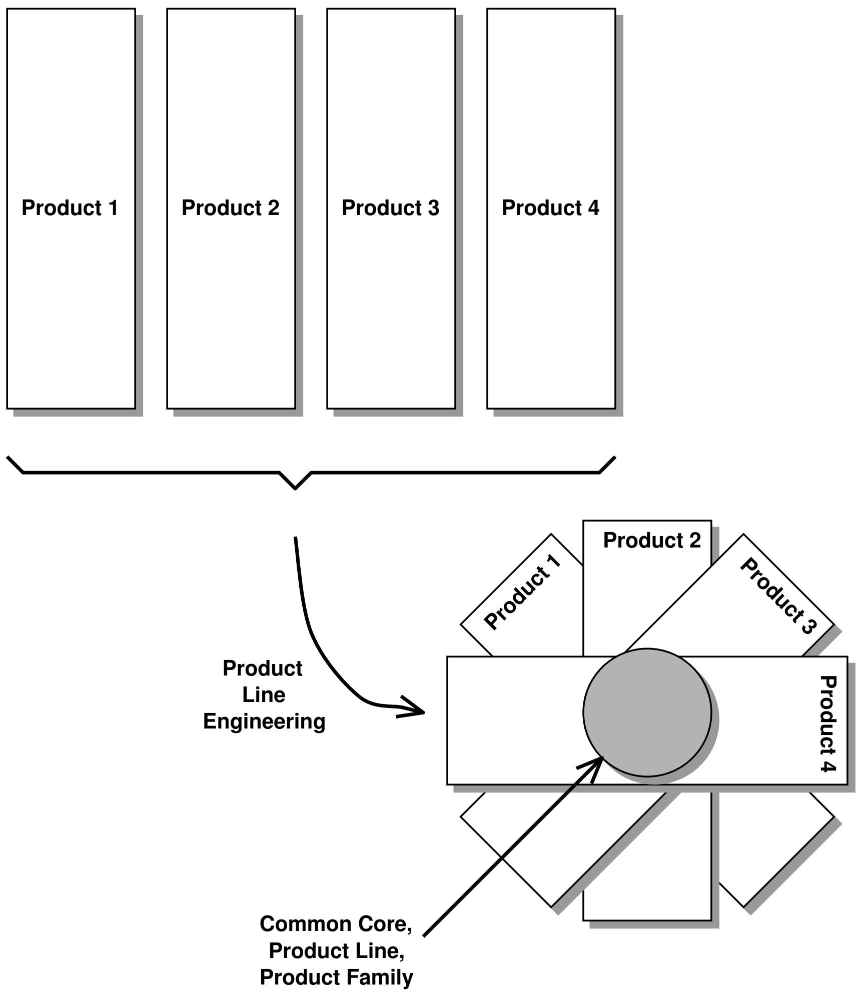


Fig. 2.25. Product line as intersection of different but similar individual products

- a textual question that is related to the domain and represents the decision to be made,
- a set of possible answers to that question where each answer maps to a specific instance of the product line (i.e. final concrete product), and
- the location where this decision is incorporated.

Additionally, it may comprise references to other decisions that have an effect on, or are affected by, the decision, and a set of effects from or on these other decisions. Table 2.6 shows a decision model in tabular form according to the context realization structural model in Fig. 2.7 on page 35.

Table 2.6. Decision model according to the context realization structural model of the *VendingMachine*

No.	Question	Variation Point	Resolution	Effect
1.a	CCBilling supported?	VendingMachine	no (default)	remove Component CCBilling
			yes	remove stereotype ≪variant≫
1.b	RoomBilling supported?	VendingMachine	no (default)	remove Component RoomBilling
			yes	remove stereotype ≪variant≫
	1.a and 1.b are alternatives according to the ≪xor≫ stereotype			

The decision model supports the instantiation of the product line into a particular final product. In order to achieve this, we will have to devise a decision model for each original model that is defined in the development process, such as the context realization models and the models for component specifications and realizations. Each model type, such as structural model, behavioral model, and interaction model will have its own decision model. The application engineer will have to give the right answers to the questions in each decision model to come up with a particular instance. Other answers to these questions in the decision models will lead to different final products. All the other development activities, such as decomposition, embodiment, composition, and validation, can be performed in the same way as described in the earlier sections. The product line engineering concepts do not affect these activities. They are only seen as a new development dimension as indicated in Fig. 2.26. This clearly separates the two main activities in that dimension, framework engineering, which relates to the development of the common core, and application engineering, which relates to the instantiation of the common core into a final application. These activities are described in the following subsection.

2.6.2 Framework Engineering

Framework engineering is concerned with the commonalities of a domain and the development of the common core for a product line. A framework can be devised by using essentially the same development principles that I have introduced in the earlier sections where I explained the principles behind the KobrA method. A framework is basically an incomplete assembly of components that provides a number of loose ends which have to be filled with additional artifacts. A single system represents a very special case of a prod-

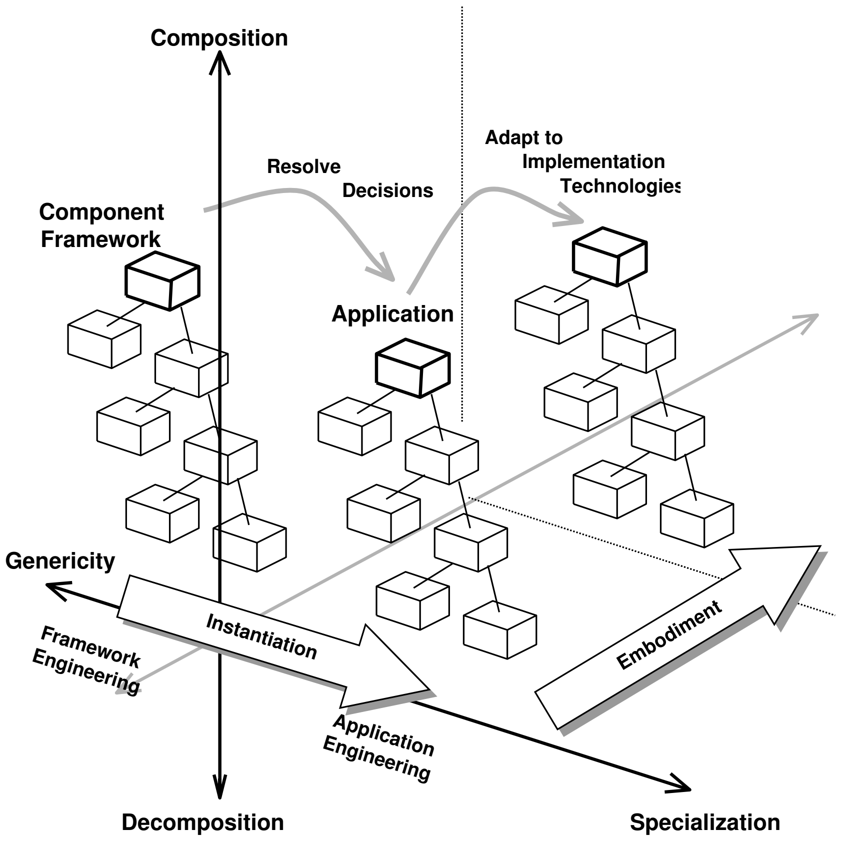


Fig. 2.26. Instantiation of a final application out of a component framework of a product line

uct family. In fact, a single system is the only representative or instance of the generic product family.

Any component in a component containment hierarchy stands for an entire system in its own right. It can represent either a specific system, in the case where no product line engineering concepts are applied, or it can represent a family of systems. In the first case, we may refer to it as an application or a final product. In the second case, the component represents a generic system that is made up of generic artifacts, and all its features refer initially to prospective final products that can be instantiated from the generic system. Any feature that is referring only to a single final product in the generic component may be indicated through the stereotype ≪variant≫. Variant features are those features that make a product different from the other products in the family. We require a model for the additional features that turn a generic system into a concrete system. This is provided by the decision model that associates the variabilities in the product family that the generic component represents to the specific artifacts of a particular product. In other words, all artifacts in a component specification or realization without the ≪variant≫ stereotype are common to all specific instances of a product line, and all the artifacts that are augmented with the ≪variant≫ stereotype represent locations where the individual products of a product family are different. The shaded intersection in Fig. 2.25 represents the set of features that are common to the four products in the same product line. These features will be modeled and developed in a normal way, as introduced in the previous sections of this chapter. Everything else, outside the shaded area, corresponds to variable features that do not belong to all members of the product family, but only to one member, or to some of the members. The differences between individual products may be quite substantial, including the presence or absence of components, or even entire component containment trees [6]. For example, Fig. 2.7 on page 35 shows two ≪variant≫ components, `CCBilling` and `RoomBilling`, which do not belong to the generic model of the `VendingMachine`. These variants represent very specific instances of `VendingMachine`, either one that offers credit card billing, or another one that offers hotel room billing. For example, a vending machine that provides room billing may be deployed in a hotel. If we remove the ≪xor≫ stereotype in the structural model, we may instantiate a system that provides both terms of payment at the same time.

Variabilities may appear on any level of abstraction or decomposition of a system. We can have variant subsystems, variant components, variant attributes and operations of components, and even variant activities within component operations if we look at the decomposition dimension. In addition, we will have a distribution of variable features through all levels of abstraction, so variabilities will be defined in the models and eventually penetrate into the source code where concrete code artifacts have to be added or removed accordingly. Any variability of a system will be initially defined in the models, i.e., variant use cases in the usage model, variant components and classes in the structural model, and variant functionality in the behavioral and functional models. A variant use case in the context realization, e.g., in Fig. 2.27, penetrates into all other models that deal with this part of the system's functionality. In this case, it leads to a variant use case description ≪variant≫`DefineCooling`, and eventually to a variant component ≪variant≫`Cooling`, depending on how the system architecture is derived from the usage model. At least we will have a variant operation specification ≪variant≫`DefineCooling` that specifies an operation for a final product that permits a service technician to set the cooling at the vending machine's local user interface. Another instance of the product may not permit the cooling to be adjusted, because it is predefined. The dependencies between variant features through all models of a product line may easily become quite complex, and it is a difficult task to manage them during a development project. Here, the primary difficulty is that the relationships between variant features often become apparent only through the decision model. The framework only indicates variant features; it cannot relate variant features to individual components according to the instance of a system. For example, a distinct variant operation in the class diagram requires a distinct class attribute. Both variant features will appear in the framework model, but there will be no indicator for whether these two variant features are related in some way.

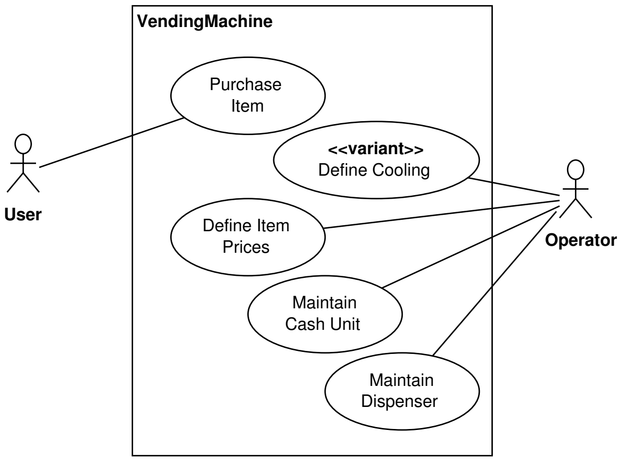


Fig. 2.27. Example of a usage model with a variable use case

Framework engineering represents a generalization of the component modeling or decomposition and of the embodiment activities, and it is enhanced to create generic frameworks rather than concrete products. In the remainder of this section I will summarize the activities that are affected by framework engineering [6].

Variability Identification

Variability identification is carried out throughout the entire product line engineering process. Whatever artifact is created within a single development, the developers have to decide whether it should be a persistent feature, an optional feature, or a feature that represents alternatives.

Decision Modeling

Decision modeling is carried out in two steps. The first step is to model variable features at the specification or realization model level and relate the artifacts to a distinct incarnation of the product line. The second step is to incorporate these features into the component containment hierarchy. It deals with how to relate variable component features with the other variable features at other levels in the composition hierarchy. These are the variable features of components above or below the component currently considered.

Component Identification

Component identification is concerned with defining good and sensible reusable building blocks for the domain in question. For example, factoring out and encapsulating the variabilities in individual components is a sensible way of identifying components. This elevates the product line engineering effort to the component level, so that ideally only complete components are added or replaced to instantiate a concrete product out of a product line.

Embodiment

Component embodiment in product line engineering can be performed in different ways. We can have either all various implementations for all products ready for deployment, and decide at start-up or runtime which incarnation of the system should be used, or we can have only those parts incorporated that are actually required for a particular system, so that the source code comprises only a minimal set of features. The following locations throughout the project life cycle represent feasible solutions for instantiating a component framework into a final product [6]:

- Implementation time. Certain variable features can be included in or excluded from the source code.
- Build time. The compiler and the linker can be controlled to include or exclude certain features. This can be done through simple preprocessor instructions, i.e., <include> in C.
- Installation time. When the system is finally installed on the target platform, we may select the modules that should be included or excluded. This must be supported through an installation tool.
- Start-up and runtime. Some of the features of a system can be selected by the user when it is starting up or running.

However, there is no standard way of approaching the organization of a product line-based system during embodiment. All the previously summarized techniques for dealing with variability can be used depending on the implementation technology used, the context, and the scope of a system development [40]. They are also often subject to strategic decisions of the project management.

2.6.3 Application Engineering

Application engineering is concerned with the generation of the specific artifacts that lead to the implementation of a final product. In a product line engineering project, it does this by instantiating the common core of a product line, and this is based on resolving the decisions that have to be made according to the decision models of the framework. The outcomes of this activity are a decision resolution model, the instance of a decision model, and a specific component framework with all variable features determined. Fig. 2.26 illustrates this move along the genericity/specialization dimension of the three-dimensional development model.

The decision resolution model records the steps that have been taken to come up with the final application, and it contains information about which set of choices led to the set of features that characterize a component. The resolution model is essential for recording the transition from a generic framework into a specific application, and it documents the application engineering activity in a traceable way. A decision resolution model is represented by a table similar to the decision model that associates a final decision with a decision of the decision model. The fundamental difference between the two models is that the decision model provides a range of decisions for each question to choose from, while the decision resolution model only represents a single decision that is associated with a question in the decision model.

Application engineering is typically performed in two steps. In the first step we have to instantiate the generic or variable features according to the decision model, and in the second step we have to adapt these features according to the development technologies used [6]. Hence, there is a clear temporal order between the modeling activities that are performed during decomposition of a product line-based system and the embodiment activities in the abstraction/concretization dimension. Application engineering must therefore be more closely related with the embodiment activity.

2.7 Documentation and Quality Assurance Plan

An essential part of a component specification, and a complete application specification, is the quality requirement that must be fulfilled to have an acceptable product. The specification cannot merely state that the product should exhibit high quality, or low failure rates, or the like. Such a terminology is too unspecific, and we can never assess that. Validation always implies a degree of goodness of an expected property; therefore, the property must be measurable. Additionally, the properties that define the expected degree of quality must in the first place be determined. Quality may be defined through the many differing attributes that a software product is expected to embody. In order to accommodate the various interpretations and requirements for the term quality, and to identify concrete practices and techniques from these abstract ideas, an effective quality assurance plan requires the following items [6]:

- A precise definition of what quality means for the development project considered, and how it manifests itself in different kinds of products.
- A precise description of what quality aspects are important for different kinds of products, and what quality levels are required.
- A systematic approach for judging the quality and improving it to the required levels.
- A plan and process to put the previous items together.

One part of a quality assurance plan determines the set of quality assurance techniques that should be applied in the software project, another part determines a set of test case selection techniques or test adequacy criteria. These two are fundamentally different. The former is concerned with theoretical background, models, and processes that will be applied and followed when a system is assessed qualitatively (e.g. stress testing, mutation testing, quality of service testing, contract testing). The latter is concerned with defining the concrete values for the input and the pre and postconditions of individual test cases according to some testing criteria (test coverage criteria, random testing, equivalence partitioning).

There are many standard test case selection techniques that may be applied within a component-based development project. Which techniques will be used in a project is subject to careful consideration, and this is typically part of defining the quality assurance plan. I cannot provide guidelines on which test case selection technique is the best for a particular purpose. This is clearly out of the scope of this volume, and it may be the subject of a book in its own right. I do, however, provide a proposition on how the models that collectively make up a system specification may be used to generate test suites for component-based software testing. The next chapter is devoted entirely to that.

2.8 Summary

Every serious software development project should be based on a sound development method. This represents the framework for all activities that a development team is supposed to perform and, ideally, it gives concrete guidelines on which activities must be carried out, how they should be carried out, when they should be carried out, and by whom they should be carried out. The KobrA method that I have briefly introduced in this chapter is one such framework. It is inherently based on the UML, because this is its primary and native notation, and it supports the concepts of the OMG's Model-Driven Architecture. But it provides a lot more.

One of the most important characteristics of this method is the provision of concrete guidelines for how the UML should be used in a component-based software development project. The core UML does not offer this. The method provides a well-defined way of approaching component-based developments through the identification of three development dimensions, composition/decomposition, abstraction/concretization, and genericity/specialization, with each of which each development activity can be associated. KobrA provides a number of concepts for dealing with specification artifacts at the system level as well as at the individual component level, and these are organized in component specification and realization. The way in which the method organizes component assemblies as containment trees realizes a recursive approach to component modeling that is based always on the same fundamental descriptive documents.

The remainder of this book can be regarded as devising a supplement to the KobrA method, which adds an entire UML-based testing framework for component-based developments. The initial step toward this target is to look at how the UML can support and realize testing, and this is the subject of the next chapter: "Model-Based Testing with UML."

3

Model-Based Testing with UML

Software testing is a widely used and accepted approach for verification and validation of a software system, and it can be regarded as the ultimate review of its specification, design, and implementation. Testing is applied to generate modes of operation on the final product that show whether it is conforming to its original requirements specification, and to support the confidence in its safe and correct operation [71, 91]. Appropriate testing should be primarily centered on requirements and specification not on code, which means that testing should always aim to show conformance or non-conformance of the final software product with some requirements or specification documents. Source code provides a great deal of information to guide the testing efforts according to testing criteria [11], but it cannot replace specification documents as a basis for testing. This is because code is a concrete representation of abstract requirements and design documents, and testing is supposed to show conformance of the concrete implementation with the abstract specifications. Testing based merely on source code documents shows that the tested program does what it does, but not what it is supposed to do.

The Unified Modeling Language (UML) has received much attention from academic software engineering research and professional software development organizations. It has almost become a de-facto industry standard in recent years for the specification and the design of software systems, and it is readily supported by many commercial and open tools such as Rational's Rose, Verimag's Tau, and VisualThought. The UML is a notation for specifying system artifacts including architecture, components and finer-grained structural properties, functionality and behavior of and collaboration between entities, and, at a higher level of abstraction, usage of a system. The UML may therefore be used to model and specify a computer system completely and sufficiently in a graphical and textual form, and to drive its realization. It provides most of the concepts of lower-level implementation notations. The combination of both, modeling and testing, is represented by the following two orthogonal dimensions that we have to consider:

- Model-based testing, which is the development of testing artifacts on the basis of UML models. In other words, the models provide the primary information for developing the test cases and test suites, and for checking the final implementation of a system. This is briefly introduced and related to traditional testing, in Sect. 3.2, "Model-Based Testing."
- Test modeling, which is the development of the test artifacts with the UML. In other words, the development of test software is based on the same fundamental principles as any other software development activity. So, in addition to using the UML to derive testing artifacts and guide the testing process, we can apply UML to specify the structural and behavioral aspects of the testing software. This is further elaborated in Sect. 3.3, "Test Modeling."

The subject of this chapter is driven mainly by the discussions of the Testing Panel held at the 5th International Conference on the Unified Modeling Language (UML 2002) in Dresden, Germany, that was initiated under the topic of whether the UML and testing may be a perfect fit. In the next section (Sect. 3.1), we will have a look at what makes model-based testing so different from traditional testing methods. Sections 3.2 and 3.3 describe the two primary activities that relate testing to model-driven development approaches, as I have stated above. Section 3.4 summarizes and concludes this chapter.

3.1 Model-Based vs. Traditional Software Testing

Traditional testing concepts (and it is really arguable whether we may refer to any testing technique as being traditional, as I will explain later) can be separated roughly into the following categories:

- Error classification. This aims at grouping different types of defects according to criteria such as nature, duration, extent, cost and consequences of a defect, and the classes that are used to assign a testing technique to address specific types of errors that can occur often.
- Testing artifacts. This comprises the test case with its individual features such as pre and postconditions, expected and observed outcome, input parameters, the test driver, test suite or test component that contains and executes the test cases, and the test bed or test stub that provides the simulation of the underlying software or hardware modules [124].
- Testing techniques and test case selection techniques. These are the typical black and white-box testing approaches with their numerous ways of defining test cases, most of which are described by Beizer [11].
- Testing phases and testing processes. These are the typical testing phases defined by the V-model, for example, unit test, integration test, system test, acceptance test, regression test, etc.

There may be other important testing concepts available which are well known, or some that are not so commonly known or applied, such as linear code sequence and jump testing (LCSAJ testing) [11], or there may be other relevant classes of testing concepts, for example, test adequacy criteria [142]. They all have in common that they are very fundamental testing concepts, and they are also equally valid for more modern object-oriented and component-based systems, or model-based testing. The target of testing does not change because of these modern development technologies; only the view on the different concepts may shift, or, in the case of models, the basis of testing changes. The most fundamental difference between traditional and more modern model-based approaches is probably the type of notations upon which testing is based. Traditionally, we used to derive testing from very abstract specifications, based on natural language at one extreme of the notations spectrum, and from the source code at the other extreme of this spectrum. Natural language is not very suitable as a basis for testing since it is not formal enough to derive all required testing artifacts. Source code is not really valuable any more if we consider component-based development where we will not be able to look at the implementations of most components. Models on the other hand are extremely useful for deriving all kinds of test artifacts at all levels of decomposition and abstraction, and they support all test development phases well. This entire chapter is devoted to that. In the following subsections I will give a number of examples on how we have to view and apply existing testing technologies differently when we deal with typical model-based concepts in object-oriented and component-based software engineering.

3.1.1 White Box Testing Criteria

Coverage is an old and fundamental concept in software testing. Coverage criteria [11] in testing are used based on the assumption that only the execution of a faulty piece of code may exhibit the fault in terms of a malfunction or a deviation from what is expected. If the faulty section is never executed in a test, it is unlikely to be identified through testing, so program path testing techniques, for example, are among the oldest software testing and test case generation concepts in software development projects [169]. This idea of coverage has led to quite a number of structural testing techniques over the years that are primarily based on program flow graphs [11] such as branch coverage, predicate coverage, or definition-use-(DU-)path-coverage [109], to name a few. These traditional coverage criteria all have in common that they are based on documents (i.e., flow graphs, source code) very close to the implementation level. Traditionally, these coverage criteria are applied only at the unit level, which sees the tested module as a white box for which its implementation is known and available to the tester. At a higher level of composition, in an integration test, the individual modules are treated only as black boxes for which no internal knowledge is assumed. An integration test is traditionally performed on the outermost subsystem that incorporates all the individually tested units, so that we assume white box knowledge of that outermost subcomponent but not of the integrated individual units. Traditional developments only separate between these two levels:

- white box test in unit testing, and
- black box test in integration testing.

Additionally, there may be an acceptance test of the entire system driven by the highest-level requirements. The more modern recursive and component-based development approaches do not advocate this strict separation, since individual units may be regarded as subsystems in their own right, i.e., components for which no internal knowledge is available, or integrating subsystems, i.e., components for which internal knowledge may be readily available. This duality that a component is always a system, and a system is always a component, and that white and black box knowledge may be assumed depending on the developer's point of view is part of the principles behind the composition/decomposition activity described in the context of the KobrA method in Chap. 2. For individual reused third-party components in a containment hierarchy we typically assume only black box knowledge. However, since we integrate these components to compose a coarser-grained system, we can assume white box knowledge of this system and, more specifically, white box knowledge of the interactions between the encapsulated components. In component-based developments we cannot strictly separate the term component from the term system; both testing approaches may be readily applied in parallel according to whether only black box information, e.g., external visible functionality and behavior, or, additionally, white box information, e.g., internal functionality and behavior, is available. In the first case we look at the component specification documents, its external specification, and check whether a component instance satisfies this. In the second case we look at the interactions between the components that collectively implement some higher-level functionality, and check these.

Typical white box strategies comprise statement coverage or node coverage at the lowest level of abstraction. In this instance, test cases may only be developed when the concrete implementation is available (i.e., for statement coverage), or if at least the implementing algorithm is known in the form of a flowchart (i.e., for node coverage). Statement coverage is not typically feasible or practical with the UML, unless we produce a model that directly maps to source code statements, but node coverage may be practical if it is based on a low-level UML activity diagram. Activity diagrams are very similar to traditional flowcharts, although activity diagrams may also represent collaboration between entities (i.e., through so-called swimlanes). Other coverage criteria such as decision coverage, condition coverage, and path coverage, may also be applicable under the UML, but this always depends on the type and level of information that we can extract from the model.

Testing that is based on the UML has many concepts in common with traditional code-based testing techniques as described in [11]. Source code can be seen as a concrete representation of a system, or its parts thereof, and UML models are more abstract representations of the same system. They are both located in the abstraction/concretization dimension discussed in the previous chapter. More concrete representations contain more detailed information about the workings of a system. They can be compared with zooming in on the artifacts considered, generating finer-grained representations but gradually losing the perspective on the entire system. Less concrete, or more abstract, representations contain less information about details but show more of the entire system. This can be compared with zooming out to a coarser-grained level of representation, making it easier to view the entire system but losing detail. The advantage of using model-based development techniques and the UML for development and testing is that a system may be represented entirely by one single notation over all levels of detail that range from very high-level and abstract representations of the system, showing only its main parts and most fundamental functions, to the most concrete possible levels of abstraction, similar and very close to source code representations. This means that in a development project we are only concerned with removing the genericity in our descriptive documents without having to move between and ensure consistency among different notations. The same is true when testing is considered. Code-based testing is concerned with identifying test scenarios that satisfy given code coverage criteria, and exactly the same concepts can be applied to more abstract representations of that code, i.e., the UML models. In that respect we can have model coverage criteria also for testing. In other words, more abstract representations of a system lead to more abstract test artifacts, and more concrete representations lead to more concrete test artifacts. Therefore, in the same way in which we are removing the genericity of our representations to receive finer-grained levels of detail and, eventually, our final source code representation of the system, we have to in parallel remove the genericity of the testing artifacts for that system and move progressively toward finer-grained levels of testing detail.

3.1.2 Black Box Testing Criteria

Most functional test-case generation techniques are based on domain analysis and partitioning. Domain analysis replaces or supplements the common heuristic method for checking extreme values and limit values of inputs [12]. A domain is defined as a subset of the input space that somehow affects the processing of the tested component. Domains are determined through boundary inequalities, algebraic expressions that define which locations of the input space belong to the domain of interest [12]. A domain may map to equivalent functionality or behavior, for instance. Domain analysis is used for and sometimes also referred to as partitioning testing, and most functional test case generation techniques are based on that. Equivalence partitioning, for example, is one technique out of this group that divides the set of all possible inputs into equivalence classes. Each equivalence relation defines the properties for which input sets belong to the same partition. Traditionally, this technique has been concerned only with input value domains, but with the advent of object technology it can be extended to the behavioral equivalence classes that we find in behavioral models. UML behavioral models such as statecharts, for example, provide a good basis for such a behavioral equivalence analysis, i.e., the test case design concentrates on differences or similarities in externally visible behavior that is defined through the state model.

Functional testing completely ignores the internal mechanism, of a system or a component (its internal implementation) and focuses solely on the outcome generated in response to selected inputs and execution conditions [91]. It is also referred to as black box testing, or specification-based testing, a term which is more meaningful and unambiguous. Binder [16] calls these techniques responsibility-based testing. This comes from the notion of a contract [112] between two entities that determines their mutual responsibilities. For example, meeting the contracted precondition assertion is the client's responsibility and meeting the postcondition is the server's responsibility, because this is what it promises to provide after completing a request [16]. Functional testing is primarily concerned with how test cases are derived from functional specifications, and there are several standard techniques that are briefly introduced in the following exposition.

Domain Analysis and Partition Testing

Domain analysis may be used as an input selection technique for all other subsequently introduced test case generation techniques. Domain analysis techniques are mainly applied in typical numerical software applications. A domain is defined as a subset of the input space that somehow affects the processing of the tested component [12]. Domain analysis is used for and sometimes also referred to as partition testing.

Equivalence Partitioning

Most functional test case generation techniques are based on partition testing. Equivalence partitioning is a strategy that divides the set of all possible inputs into equivalence classes. The equivalence relation defines the properties for which input sets belong to the same partition, for example, equivalent behavior (state transitions). Proportional equivalence partitioning, for example, allocates test cases according to the probability of their occurrence in each sub-domain [16, 124].

Category Partitioning

Category partitioning is traditionally used in the industry to transform a design specification into a test specification. It is based on the identification of the smallest independent test units and their respective input domains. Categories that may be considered in category partitioning are, for example, operand values, operand exceptions, memory access exceptions, and the like [16, 124].

State-Based Testing

State-based testing concentrates on checking the correct implementation of the component's state model. Test case design is based on the individual states and the transitions between these states. In object-oriented or component-based testing, any type of testing is effectively state-based as soon as the object or component exhibits states, even if the tests are not obtained from the state model. In this scenario, there is no test case without the notion of a state or state transition. In other words, pre and postconditions of each test case must consider states and behavior. The major test case design strategies for state-based testing are piecewise coverage, transition coverage, and round-trip path coverage, and they are described in more detail in Sect. 3.2.7.

Method Sequence-Based Testing

Method sequence-based testing concentrates on the correct implementation of a component's combinations, or sequences of operations provided. For component-based testing this is the most important test-case generation technique, since it concentrates on how individual component operations are working in combination and in sequences. These sequences represent distinct ways of using a component by a third party, i.e., the client of the component. Here, test case design is based on the behavioral model, such as a UML statechart diagram. The paths through the state model are checked. This may also include multiple invocations of the same operation according to a usage profile. Essentially, this applies all state-based testing coverage criteria that have already been introduced in the previous subsection.

Message Sequence-Based Testing

Message sequence-based testing checks the collaborations between different objects. Test case design is based on interaction models, such as the UML sequence or interaction diagrams. Message sequence-based testing is particularly important and advantageous for checking real-time applications [29].

The examples that I have put forward in this section and the discussions on some of the testing techniques make the similarities between what I have so far called traditional testing concepts and the more modern model-based concepts apparent. In my opinion, there is no fundamental difference. All these "traditional" concepts re-appear in model-based testing, and, in fact, they are essentially the same. The only two differences that I can observe are that we are now dealing with a somewhat novel notation, the UML, and that we have to consider testing techniques at a higher level of abstraction that have been devised for a much lower abstraction level, the implementation level. In the following section, we will look at the individual UML diagrams, and I will introduce their concepts and semantics and discuss how they may be used to extract black box as well as white box testing information.

3.2 Model-Based Testing

The UML provides diagrams according to the different views that we can have on a system. These views can be separated into user view and architectural view, which may be further subdivided into structural view and behavioral view, implementation view, and environmental view. These views can be associated with the different diagram types of the UML. The user view is typically represented by the use case diagram, and the structural view by class and object diagrams. Sequence, collaboration, statechart, and activity diagrams can be associated with the functional and behavioral views on a system, and component and deployment diagrams specify the coarse-grained structure and organization of the system in the environment in which it will be deployed. In essence, UML diagrams specify what a system should do, how it should behave, and how it will be realized. The entirety of all UML models therefore specifies the system completely and sufficiently. The fundamental question here is, what information can we extract from a UML model for driving the testing of the system, or what testing activities can we base on a UML model?

3.2.1 Usage Modeling

The initial phase of a development project is typically performed to gather information about which user tasks will be supported by a prospective system. This activity in the overall development process is termed usage modeling, and its outcome is the specification of the system's high-level usage scenarios. The main artifact in the UML that is concerned with this type of high level usage modeling is the use case diagram. Use case diagrams depict user communication with the system where the user represents a role that is not directly involved in the software development process, or represents other associated systems that use the system under development.

Use Case Diagram Concepts

Use case diagrams specify high-level user interactions with a system. This includes the users or actors as subjects of the system and the objects of the system with which the users interact. Thus, use case models may be applied to define the coarsest-grained logical system modules. Use cases mainly concentrate on the interactions between the stakeholders of a system and the system at its boundaries. A use case diagram shows the actors of the system (the stakeholders), either in the form of real (human) user roles, or in the form of other associated systems which are using the system under development. Additionally, use case diagrams show the actual use cases and the associations between the actors and the use cases. Each use case represents some abstract activity that the user of the system may perform and for which the system provides the support. Overall, use case modeling is applied at initial requirements engineering phases in the software life cycle to specify the different roles that are using the individual pieces of functionality of the system, the use cases. Use case diagrams are often defined in terms of the actual business processes that will be supported by a system. Figure 3.1 displays the use case diagram from Chap. 2. I have extended the model to concentrate on the operator's usage scenarios. I have added scenarios for maintaining the `CashUnit` and the `Dispenser`, i.e., take the cash off the safe, refill the dispensers, define the cooling of the machine, and determine the prices of the items sold by the machine.

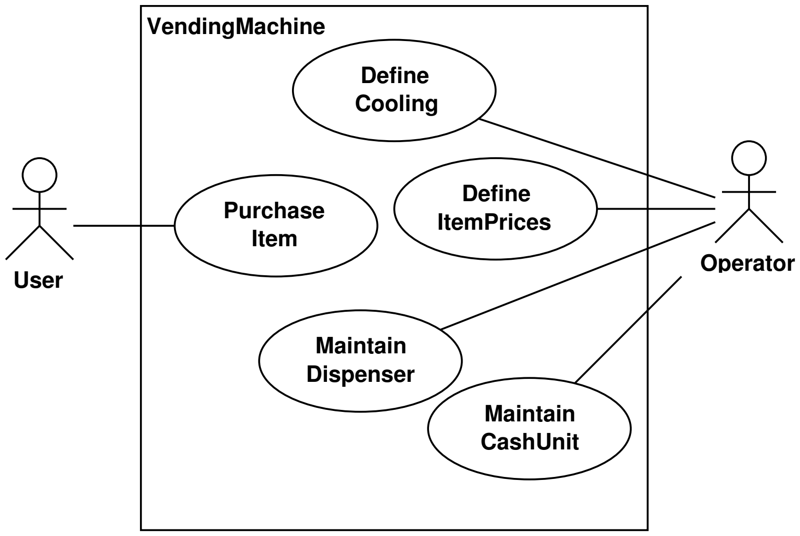


Fig. 3.1. Concepts of a use case diagram, specification of the vending machine

3.2.2 Use Case Diagram-Based Testing

Use cases are mainly applied for requirements-based testing and high-level test design. Testing with use cases can be separated into two groups according to the source of information that will be used for test development:

- Testing that is based on the use case diagram, which is mainly suitable for test target definition.
- Testing that is based on the information of the use case template, which is similar to typical black box testing, although at a much higher level of abstraction.

How a use case diagram can guide the testing and define the test target is described in the following paragraphs. The use case diagram does not permit typical test case design with pre and postconditions, input domains, and return values because it does not go into such a level of detail. However, we can define the following elements in a use case diagram, and their relations that may be suitable for the purpose of testing [16]:

- An actor can participate in one or several use cases: This will result in an acceptance test suite for each individual actor and for each individual use case, and the tests will reflect the typical usage of the system by that actor.
- A use case involves one or several actors: Each test suite will comprise tests that simulate the user's interactions at the defined interaction point that is the use case functionality. If several actors are associated with the same use case, we may additionally have concurrent usage of some functionality by different roles. For testing it means that we will have to investigate whether multiple concurrent usage is supported by the system as expected. We might therefore have to define a test suite that takes such a simulation into consideration.
- A use case may be a variation of some other use case (≪extends≫): If our test criterion is use case coverage, we will have to produce test suites that comprise all feasible usage permutations of the base use case and its extension. This is very similar to checking correct inheritance relations in object-oriented testing [16].
- A use case may incorporate one or more use cases (≪uses≫). For testing, this is essentially the same as the previous item.

A use case diagram can additionally indicate high-level components. This is specifically supported through use case descriptions. All the objects that are mentioned in a use case diagram or use case description are feasible candidates for high-level components at a system architectural level. Therefore, use cases and structural diagrams (class and component diagrams) are associated through the following relations, and this is actually how the semantic gap between use cases and the system architecture is bridged in a model-based development project:

- A use case is implemented through one or several nested and interacting components. Requirements-based testing should attempt to cover all components that are participating in the implementation of a use case. This is particularly important if requirements are changed. In that case, we will have to trace the changes to the underlying component architecture and amend the individual components accordingly. A regression test should then be applied to validate the correctness of these amendments. Traceability of requirements down to components is therefore an essential property. This can be done through a predefined stereotype ≪trace≫ `Component` that relates a use case to a component, or it can be written down in the use case description through an additional entry in the table in the form "realized through `Component`," for example.
- A component supports one or more use cases. Here, the component architecture is not functionally cohesive. In other words, individual components are responsible for implementing non-related functionality, leading to low cohesion in the components. This may be regarded bad practice but it surely happens, and often it is a requirement, or third-party components are organized in that way. In such an instance, use cases represent different and probably concurrent usage of the same component, and that must be reflected in the validation. We might therefore have to define a test suite that takes the concurrency situation into consideration as said before. With respect to traceability to the use case that a component supports, it would also be desirable to have a tracing mechanism, for example, in the same form as mentioned above. This could be done through a stereotype or through a UML comment. Following these suggestions, we can easily incorporate a two-way tracing mechanism in our models.

A use case diagram is mainly used for test target identification, and to achieve test coverage at a very high level of abstraction. For example, Binder defines a number of distinct coverage criteria that can be applied to use case diagrams to come up with a system-level acceptance test suite [16]:

- Test or coverage of at least every use case.
- Test or coverage of at least every actor's use case.
- Test or coverage of at least every fully expanded ≪includes≫, ≪extends≫, and ≪uses≫ combination.

These correspond to coverage of all nodes and arrows in a use case diagram, and are typical testing criteria, similar to the traditional test coverage measures that are based on program flow graphs as discussed in Sect. 3.1. Each of these criteria represents a test of high-level user interaction. Because there is only limited information, we can determine only which user functionality we will have to test, but not how to test it. Each test target can be augmented with information from the more concrete use case definitions to identify more concrete test artifacts. Therefore, each test target will map to a test suite and, eventually, when more information is added, to a number of more concrete test cases. The collection of all user-level tests which are developed in that way may be used for system acceptance testing.

Based on the previously described items, we can derive the test targets or tester components for the use case diagram depicted in Fig. 3.1. They are displayed in Table 3.1. We do not have ≪uses≫ or ≪extends≫ relations in our example, so the specification of the tester components for a system acceptance test suite are quite simple. These tests also represent full coverage of the use case diagram. We can model the organization of the tester components in the same way as we have modeled the rest of the system. For example, Fig. 3.2 shows the containment hierarchy of the tester components. This is an example only of how we can use models to define and describe the organization of the test for a system that is also model-driven. I will explain how the testing infrastructure is modeled in more detail in Sect. 3.3.

Table 3.1. Identification of external tester components for the *VendingMachine* from the use case diagram

Actor	Use Case
Usage Profile	Tester Component
User	Purchase Item
Operator	Define Cooling
Operator	Define ItemPrices
Operator	Maintain Dispenser
Operator	Maintain CashUnit

3.2.3 Use Case and Operation Specification-Based Testing

A use case diagram is quite useless for the concrete specification of what a system is supposed to do. Use case diagrams display only very limited information, so they are extended by use case specifications or use case definitions as introduced in Chap. 2. Each specification of a use case according to the use case template introduced in Chap. 2 corresponds to a component's operation specification according to the operation description template from Chap. 2. I have included both templates in this chapter again for convenience in Table 3.2 and Table 3.3. An operation specification comprises the full description of a piece of functionality that an object or component provides. For example, a class method may be regarded as such an operation.

In contrast, a use case represents a piece of functionality that is not attributed to a particular object in the system but to the entire system at its

Table 3.2. Use case template

Use Case No.	Short name of the use case indicating its goal (from the model)
Goal in Context	Longer description of the use case in the context.
Scope & Level	Scope and level of the considered system, e.g. black box under design, summary, primary task, sub-function, etc.
Primary/Secondary Actors	Role name or description of the primary and secondary actors for the use case, people, or other associated systems.
Trigger	Which action of the primary/secondary actors initiate the use case.
Stakeholder & Interest	Name of the stakeholder and interest of the stakeholder in the use case.
Preconditions	Expected state of the system or its environment before the use case may be applied.
Postconditions on success	Expected state of the system or its environment after successful completion of the use case.
Postconditions on failure	Expected state of the system or its environment after unsuccessful completion of the use case.
Description Basic Course	Flow of events that are normally performed in the use case (numbered).
Description Alternative Courses	Flow of events that are performed in alternative scenarios (numbered).
Exceptions	Failure modes or deviations from the normal course.
NF-Requirements	Description of non-functional requirements (e.g. timing) according to the numbers of the basic/alternative courses.
Extensions	Associated use cases that extend the current use case (≪extends≫ relation).
Concurrent Uses	Use cases that can be applied concurrently to the current use case.
Realized through	Component(s) in the component architecture that implement this use case.
Revisions	Trace of the modifications of the current use case specification.

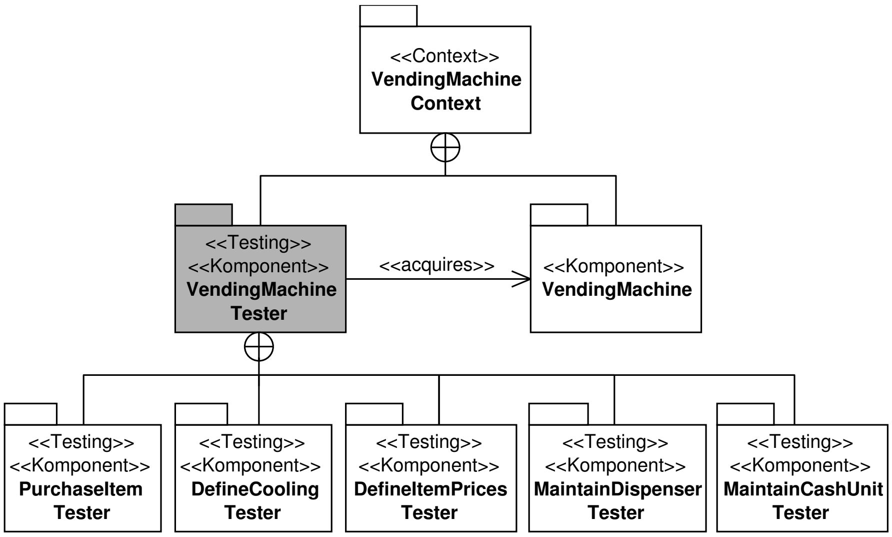


Fig. 3.2. Containment hierarchy of the test organization

boundary. If we consider an individual component to be a system in its own right, as is the case in the KobrA method, then the operation specifications of that component and its corresponding system-level use case specifications are essentially the same. The entries in the two templates in Table 3.2 and in Table 3.3 indicate this conceptual similarity. Both templates define name and description, pre and postconditions, and exceptions that can be related to constraints. The fundamental difference between the two models lies in the fact that use case descriptions in contrast with operation specifications define no concrete input and output types that easily map to input and output value domains, and that the pre and postconditions are not attributable to distinct objects in the system. In other words, use case descriptions represent similar information as operation specifications although at a much higher level of abstraction or decomposition and from a different point of view, according to the stakeholders to which we are referring. While use case specifications are used mainly for communication outside the development team, i.e., with the customer of the software, operation specifications are more suitable for communication between the roles within the development team, for example, system designers, developers, and testers. In any case, use case template-based testing and operation specification-based testing represent both functional or black box testing approaches, because both representations concentrate on external expected behavior. In other words, use case descriptions specify functionality at the system level, while operation specifications describe functionality at the object or component level.

Table 3.3. Operation specification template according to the KobrA method and Fusion

Name	Name of the operation
Description	Identification of the purpose of the operation, followed by an informal description of the normal and exceptional effects.
Constraints	Properties that constrain the realization and implementation of the operation.
Receives	Information input to the operation by the invoker.
Returns	Information returned to the invoker of the operation.
Sends	Signals that the operation sends to imported components (can be events or operation invocations).
Reads	Externally visible information that is accessed by the operation.
Changes	Externally visible information that is changed by the operation.
Rules	Rules governing the computation of the result.
Assumes	Weakest precondition on the externally visible states of the component and on the inputs (in receives clause) that must be true for the component to guarantee the postcondition (in the result clause).
Result	Strongest postcondition on the externally visible properties of the component and the returned entities (returns clause) that becomes true after execution of the operation with a valid assumes clause.

More abstract representations, such as use case models and use case descriptions, were not initially taken into account as bases for typical functional testing approaches such as the ones mentioned in Sect. 3.1 because traditional testing always used to be, and probably still is, focused on more low-level abstractions and concrete representations of a system such as code. Therefore, these functional testing techniques appear to be used more optimally in tandem with typical operation specification-type documents as the primary source for test development. However, since the two models, use case descriptions, and operation specifications are concerned with essentially the same information but at different levels of abstraction, all the functional testing techniques that I have introduced and discussed earlier can also be based on use case descriptions, although at a much more abstract level.

The use case template (Table 3.2) already contains a number of items that are suitable for the definition of system-level tests. The pre and postconditions (on success) in the template and the description of the basic course map to abstract test cases. For the first use case of the vending machine (`PurchaseItem`), these abstract test cases are displayed in Table 3.5. The tests are abstract because they do not indicate concrete values for the used variables, e.g., `Item`, `Item.price`, `InsertedCoins`, etc. Each of the abstract scenarios in the table may map to a number of concrete test cases that represent different distinct and concrete test scenarios that we can apply to check a functionality of the vending machine. The tests that we instantiate from Table 3.5 will be implemented in the `PurchaseItemTester` component in the testing structural model in Fig. 3.2. I have included the definition of the use case `PurchaseItem` in Table 3.4 below to demonstrate the mapping between the use case description and the derived tests more clearly.

The abstract tests displayed in Table 3.5 are all based on fundamental testing techniques that are summarized as follows [95]:

- Test of basic courses, testing the expected flow of events of a use case.
- Test of odd courses, testing the other, unexpected flow of events of a use case.
- Test of any line item requirements that are traceable to each use case.
- Test of features described in user documentation that are traceable to each use case.

The last two items in this list refer to global information that is not necessarily contained in the individual use case specifications. The line *Concurrent Uses* in the use case definition template indicates that the use case itself may be invoked concurrently, and that other use cases may be applied at the same time.

Additional testing techniques that may be used in tandem with use case modeling and the specification of use cases are scenario-based techniques, as described in the SCENT Method, but they require some additional modeling effort with dependency charts [144, 145, 146]. There are also distinct coverage criteria with these techniques, such as scenario-path coverage, event-flow coverage, exception coverage, that may be applied in use case based testing, or we can apply statistical usage-based testing techniques [133, 134].

3.2.4 Structural Modeling

High-level structural modeling is typically the next development step after use case modeling. Use case descriptions loosely associate system functionality with components at a high system architectural level. In other words, we can already define the very fundamental parts of the system in a typical divide-and-conquer manner when we develop the use case descriptions. All objects in the use case model have a good chance of becoming individually identifiable parts, such as components or classes, objects, modules, or subsystems, during

Table 3.4. Definition of the use case *Purchase Item* from the vending machine usage model

Use Case 1	Purchase Item
Goal in Context	Main scenario for purchasing an item from the vending machine.
Actors	User.
Trigger	User inserts coins into slot, or selects item on button panel.
Preconditions	Vending machine is operational.
Postconditions on success	Item is provided.
Postconditions on failure	Item is not provided.
Description Basic Course	1. User inserts sufficient amount of money. 2. User selects item on panel. 3. Selected item and return money are dispensed. 4. User takes item and returned money, if applicable.
Description Alternative Courses	1. User selects item. 2. Vending machine displays price. 3. <basic course>
Exceptions	1. [insufficient amount] user inserts more cash, or aborts. 2. [selected item not available] user selects different item, or aborts.
NF-Requirements	1. Item should be dispensed not more than 3 seconds after selection. 2. After 3 sec of no user selection use case is aborted and inserted cash returned.
Extensions	<left open>
Concurrent Uses	<left open>
Revisions	<left open>

design. Under the KobrA method they are all termed “Komponent” which stands for KobrA Component [6]. If we have defined the first components, we will typically decompose the system into smaller more manageable parts that are not immediately related to the high-level usage of the system; the models of these parts concentrate more on internal functional aspects of the system. This decomposition activity typically comes under the umbrella of system design and is normally detached from the usage modeling activity. The design activity

Table 3.5. Abstract tests, derived from the use case description for *purchaseItem*

No.	Pre-condition	Event	Postcondition	Result
1.1	operational	user inserts coin money		
1.2	money = item.price	user selects item on panel	operational AND item provided	user takes item
2.1	operational	user inserts coin money		
2.2	money < item.price	user selects item on panel	item not provided	
2.3	money = item.price	user selects item on panel	operational AND item provided	user takes item
3.1	operational	user inserts coin money		
3.2	money > item.price	user selects item on panel	operational AND item, change provided	user takes item AND change
4.1	operational	user selects item on panel	operational AND item not provided	price displayed

is more centered around technical requirements of the implementation, e.g., available components, safety or timing aspects, etc., rather than functional user requirements. Structural diagrams specify the architectural relationships between components as the most fundamental building blocks of the system.

Component Containment Diagram Concepts

Components are the basic construction entities in component-based development. A system's primary components are typically identified in the requirements engineering phase of a project through use case modeling and the definition of use case descriptions as outlined in the previous sections. We can identify good candidates for components because we apply domain knowledge that determines the architecture, or through naturally available system parts. In the first case, we can identify the components because in the past they used to be more or less the same, and we found that suitable. In the second case, we have existing components that are domain-specific. The fact that there are distinct prefabricated components on the market for our domain has mainly historic reasons. Because our domain has evolved in a certain way, we have certain ways of doing and organizing things, and this is reflected through the components on the market. Now, if we have identified some typical high-level components, they may be brought into a hierarchy that represents the coarsest-grained structural organization of the entire system. For example, Fig. 3.3 displays the containment hierarchy for the vending machine.

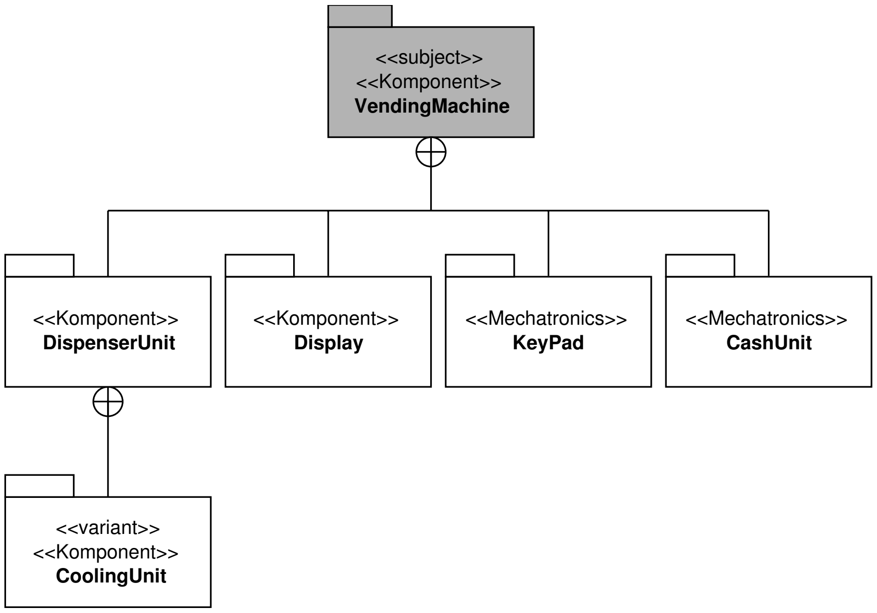


Fig. 3.3. Development-time containment tree for the vending machine

The KobrA method proposes to use the UML package symbol for representing components since a component is a collection or package of a number of different specification and documentation artifacts, e.g., a collection of UML models and tabular representations such as use case definitions and operation specifications. The symbol indicates the scoping of all these descriptive artifacts.

Each component in the hierarchy is described through a specification that comprises everything externally knowable about the component in terms of structure, function, and behavior, and a realization that encompasses everything internally knowable about these items. The specification describes what a component is and can do, and the realization how it does it. The subject component indicated through the stereotype ≪subject≫ represents the entire system under consideration. In our case it is the `VendingMachine` component. The context realization, in this case the component `VendingMachineContext`, describes the existing environment into which the subject will be integrated. It contains typical realization description artifacts as introduced in the specification of the vending machine in Chap. 2. The anchor symbol represents containment relations, or, in other words, the development-time nesting of a system's components. Component nesting always leads to a tree-shaped structure, and it also represents client/server relationship.

- A nested component is typically the server for the component that contains it, and
- a nesting component is typically the client of the components which it contains.

This is the case at least for creating a new instance of a contained component. The superordinate component is the context for the subordinate component, and it calls the constructor of the subordinate component. This can be seen as the weakest form of a client/server relationship. Containment trees can also indicate client/server relationships between components that are not nested. This is indicated through an ≪acquires≫ relationship between two components in which one component acquires the services of another component, as indicated between the component `VendingMachineTester`, and the component `VendingMachine` in Fig. 3.5. This type of an explicit client/server relationship leads to an arbitrary graph in the containment hierarchy.

A coarser-grained component at a higher level of decomposition is always composed of finer-grained components residing at a lower level of decomposition, and the first one "contains" or is comprised of the second ones. The nesting relations between these entities are determined through so-called component contracts. KobrA's development process represents an iterative approach to subsequently decomposing coarser-grained components into finer-grained components until a suitable third-party component is found, or the system is decomposed into the lowest desirable level of composition that is suitable for implementation. This was outlined with the vending machine example in Chap. 2.

Class and Object Diagram Concepts

Class diagrams and object diagrams are made up of classes or objects (class instances) and their associations. The associations define the peer-to-peer relations between the classes (and objects), so these diagrams are used primarily for specifying the static, logical structure of a system, a subsystem, or a component, or parts of these items. Associations come in different shapes with different meanings:

- A normal association defines any arbitrary relationship between two classes. It means they are somehow interconnected. It is indicated through a simple solid line between the classes.
- An aggregation is a special association that indicates that a class, i.e., the aggregate, is comprised of another class, i.e., the part. This is also referred to as "whole-part association" or class nesting. This is in fact similar to the containment model, and is not specific about who creates and owns the

subordinate classes. An aggregation is indicated through an open diamond at the aggregate side of the association line.
- Ownership between classes is indicated through the composition aggregation. This is a much stronger form of aggregation in which the parts are created and destroyed only together with the whole. This is indicated with a solid diamond at the aggregate end of the association.
- Generalization is a form of association that indicates a taxonomic relationship between two classes, that is, a relationship between a more general class and a more specific class. In object technology terminology this is also referred to as an inheritance relationship. It is indicated through an open triangle at the end of the association of the more general class.
- A refinement association indicates a relationship between two descriptions of the same thing, typically at two distinct abstraction levels. It is similar to the generalization association, although the focus here is not on taxonomy but at different levels of granularity or abstraction. This is indicated through a generalization with a dashed line.
- Dependency is another form of association that expresses a semantic connection between two model elements (e.g., classes) in which one element is dependent upon another element. In other words, if the independent class is changed, it typically leads to a change in the dependent class. This is indicated with a dashed arrow.

Multiplicity parameters at associations indicate how many instances of two interconnected classes will participate in the relation. Class symbols have syntax too. They consist of a name compartment, an attribute compartment, and an operation compartment. The last two define the externally visible attributes and operations that the class or object is providing at its interface, and which may be used by external clients of the class to control and access its functionality and behavior. For example, Fig. 3.4 shows the amended specification structural model of the `VendingMachine` component with the additional operations for the operator of the vending machine.

Package Diagram Concepts

A package diagram can comprise classes or objects, components, and packages. A package is only a grouping mechanism that can be linked to all types of other modeling elements, and that can be used to organize semantically similar or related items into a single entity. Subsystems, components, and containment hierarchies of classes may also be referred to as packages, since all these concepts encapsulate various elements within a single item in the same way as a package. For example, the KobrA development method uses the package symbol for specifying a Komponent (KobrA Component) since it represents a grouping of all descriptive documents and models that collectively define a component in terms of functionality, behavior, structure, and external and internal quality attributes.

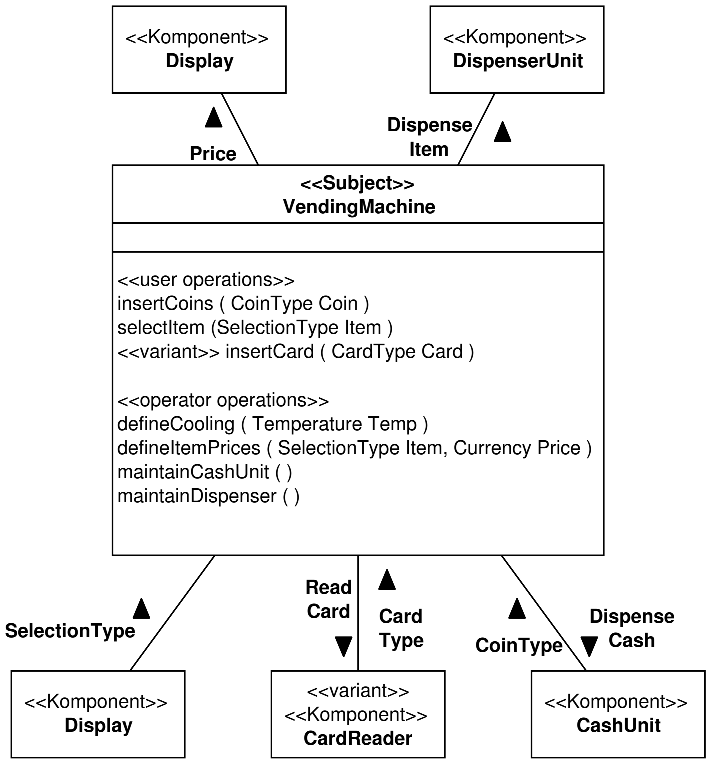


Fig. 3.4. *VendingMachine* amended specification structural model

Component Diagram Concepts

A component diagram organizes the logical classes and packages into physical components when the system is executed. It represents a mapping from the logical organization of a system to the physical organization of individually executable units. Its main focus is on the dependency between the physical components in a system. Components can define interfaces that are visible to other components in the same way as classes, so that dependencies between components can also be expressed as access to interfaces. Class and component diagrams are similar with respect to this property since they can both specify associations between modeling elements.

Deployment Diagram Concepts

A deployment diagram shows the actual physical software/hardware architecture of a deployed system including, computer nodes and types and hardware devices, along with their relations to other such entities. Important specifications in a deployment diagram, for example, are executable components which will be assigned to physical nodes in a network, and on underlying component platforms, runtime support systems, etc.

3.2.5 Structural Diagram-Based Testing

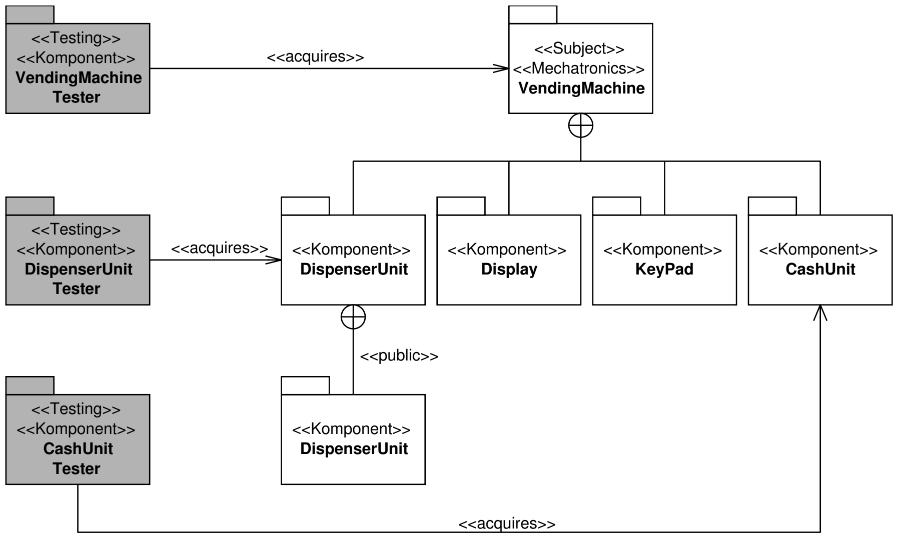
Intuitively it might seem odd to combine structural issues with testing activities because testing is always based on function or behavior rather than structure; so the value of structural diagrams for testing appears to be very limited at a glance. However, this is the case only for deriving concrete test cases from structural models, something that is clearly not feasible, since structural diagrams do not provide enough information for the definition of test cases, i.e., pre and postconditions, and behavior. Test case design can be done only in tandem with functional descriptions and behavioral models. What we can identify from structural models is what should be tested in a system that consists of many interacting entities. In other words, we can use structural models for test target definition in a way similar to that for use case models.

Structure represents the logical organization of a system, or the pairwise relations between the individual components. These pairwise relations are described through contracts that specify the rights and responsibilities of the parties that participate in a contract. When two components establish such a mutual relationship, we have to check the contract on which this relationship is founded. So, when we go through the structural models of a development project, we can identify a list of contracts for each of which we can formulate a test target that maps to a test suite or a tester component. Traditionally this testing activity belongs to integration testing.

Component Containment Diagrams and Testing

KobrA's component containment hierarchies can be seen as the most general and most abstract logical structural models. They display component nesting and client/server relations between a superordinate component and its contained subordinate components, represented by the anchor symbol, as well as client/server relations between components at the same or neighboring hierarchic levels, represented by arbitrary ≪acquires≫ relations. Both concepts indicate that one component requires the services of another component, so there must be an interface definition between the two parties in that client/server relationship. In object terminology, containment is equivalent to a class attribute that refers to a subordinate class. Each connection in a containment hierarchy relates to a test target and consequently to a test suite or to an additional tester component that specifically concentrates on testing that connection. This is illustrated in Fig. 3.5. A test of the system `VendingMachine` requires that the communication between all integrated components is tested in combination. The `VendingMachine` component expects to get some features from its subordinate components `Dispenser` and `CashUnit`, and in return these components expect to be used by `VendingMachine` in a certain way. These are mutually accepted contracts between the parties. These mutual expectations can be represented by test suites that can be executed as unit tests on the individual components before the components are finally integrated. Each test suite will contain only tests that simulate the access of the respective client component on the server. The `DispenserUnitTester` in Fig. 3.5 will comprise only tests that simulate the vending machine's usage of the tested subcomponent. Other tests are meaningless for the integration because such cases will never appear when the two components are interacting. If all the tests in all the test suites pass, we expect that the integration of all components was successful, and that the `VendingMachine` component will expose no more such errors that are related to the component interactions, given that we have applied adequate test sets.

The services that are exchanged through the connections of a containment hierarchy are more specifically defined in class/object diagrams and behavioral models, so that the actual test case definitions can only be carried out together with the other models that specify functionality and behavior. Here, we are concerned only with test target identification and test component specification.



<<Testing>> <<Komponent>> VendingMachine Tester
<<acquires>>
<<Subject>> <<Mechatronics>> VendingMachine
<<Testing>> <<Komponent>> DispenserUnit Tester
<<acquires>>
<<Komponent>> DispenserUnit
<<Komponent>> Display
<<Komponent>> KeyPad
<<Komponent>> CashUnit
<<public>>
<<Komponent>> DispenserUnit
<<Testing>> <<Komponent>> CashUnit Tester
<<acquires>>

Fig. 3.5. Tester components for the vending machine derived from the containment model

Class, Package, Component, and Deployment Diagrams and Testing

All the other structural diagrams in the UML such as class, package, component, and deployment diagrams are used to express the implementation and deployment of components. For example, class diagrams are mainly used in component specifications and realizations, but they can also express the distribution of logical software components over hardware nodes. Packages, components, and classes are very similar concepts, and the component term combines their individual particularities, i.e., the component provides a scoping mechanism to package a variety of concepts and artifacts such as classes and modules. Components provide interfaces as classes and, maybe, packages do, and they have states. In fact, a component's class properties are provided through the classes that the component contains. All testing concepts of the previous paragraphs are therefore applicable to classes and packages, that is, test target definition and identification of client/server contracts that need to be checked. The different diagrams merely represent slight variations of the modeled subjects. The fundamental difference between classes and components, for example, is that class interactions are likely to stay fixed for a longer period of time. Components may be seen as the fundamental building blocks of applications in component-based development, and they are often reused and integrated in different contexts. Classes, on the other hand, can be seen as the fundamental building blocks of components, so they are not so readily reused, because the classes in components are not subjected so much to constant change. The difference in the diagrams with respect to testing is not so much a concern for extracting different testing information from the models, but for the strategy of when tests will be ideally executed. The fundamental idea for identifying interesting locations in a system or a component for performing integration tests is the same in all diagram types. But we can extract more information on when these tests have to be performed. For example, a class diagram shows interaction between the fundamental building blocks of a component. They are likely to stay as they are during the lifetime of the component. So we integrate and test that integration once and for all.

Components are the building blocks of component-based systems. Whenever we put components together to come up with a new application, we have to check the validity of this integration through a test. Some components will be assigned dynamically, others will stay as they are. Component diagrams show this type of organization so that we can identify fixed contracts and loose contracts that are likely to change and will need retesting in a new context. Deployment diagrams represent a different view of the same problem. Here we assign components to nodes in a distributed environment. Some nodes will stay the same throughout the life span of a system and only need an initial check, but others might undergo constant change, so we will have to perform an integration test whenever a node is changed. The fundamental idea of test target definition with structural diagrams remains. We can see from a structural diagram which interactions should be tested under what circumstances.

Structural diagrams, in particular containment trees, play a seminal role in the development of the testing architecture for built-in contract testing in component-based application engineering. The concepts behind built-in contract testing and all the aspects of applying it in a component-based development project are subject of Chap. 4.

3.2.6 Behavioral Modeling with Statecharts

Structural modeling is part of system decomposition, and it identifies the subparts of the system that will be individually tackled in separate development efforts. Each part can be subdivided further into even smaller units. If such a part or component has been identified, its behavior must be described; this comprises its externally visible behavior at its provided interface, as well as the externally visible behavior at its required interface. The UML supports behavioral modeling through statechart diagrams and activity diagrams. Statechart diagrams represent the behavior of an object by specifying its responses to the receipt of events. Statecharts are typically used to describe the behavior of class or component instances, but they can also be used to describe the behavior of use cases, actors, or operations. Related to statechart diagrams are activity diagrams that concentrate on internal behavior of an instance, or, in other words, the control flow within its operations. Both diagram types are essentially based on the same fundamental concepts.

Statechart Diagram Concepts

Statechart diagrams are made up of states, events, transitions, guards, and actions. A state is a condition of an instance over the course of its life, in which it satisfies some condition, performs some action, or waits for some event. A state may comprise other encapsulated substates. In such a case, a state is called a composite state. A special state, the starting state, indicates the first condition throughout the life cycle of an instance. Another special state, the end state, indicates the last condition throughout the life cycle of an instance.

An event is a noteworthy occurrence of something that triggers a state transition. Events can come in different shapes and from different sources:

- A designated condition that becomes true. The event occurs whenever the value of an expression changes from false to true.
- The receipt of an explicit signal from somewhere.
- The receipt of a call of an operation.
- The passage of a designated period of time.

Events trigger transitions. If an event does not trigger a transition, it is discarded. In this case it has no meaning for the behavioral model. Events are therefore only associated with transitions. A simple transition is a relationship between two states indicating that an instance that is residing in the first state will enter a second state provided that certain specified conditions are satisfied. The trigger for a transition is an event. A concurrent transition may have multiple source states and multiple target states. It indicates a synchronization or splitting of control into concurrent threads without concurrent substates. A transition into the boundary of a composite state is equivalent to a transition to the starting state of the composite substate model. A transition may be labeled by a transition string to the following format:

```
event_name ( parameter_list )
[ guard_condition ] / action_expression
```

Here, a guard represents a conditional expression that lets an event trigger a transition only if the conditional expression is valid. It is a boolean expression written in terms of the parameters of the triggering event, plus attributes and links of the object that owns the state model. An action expression is a procedural expression that will be executed if the transition is performed.

3.2.7 Statechart Diagram-Based Testing

State-based testing concentrates on checking the correct implementation of the component's state model. The test case design is based on the individual states and the transitions between these states. In object-oriented or component-based testing, any type of testing is effectively state-based as soon as the object or component exhibits states, even if the tests are not obtained from the state model. In that instance, there is no test case without the notion of a state or state transition. In other words, pre and postconditions of every single test case must consider states and behavior. Binder [16] presents a thorough overview of state-based test case generation, and he also proposes to use so-called state reporter methods that effectively access and report internal state information whenever invoked. These are essentially the same as the state information operations that are defined by the built-in contract testing technology (state checking operations) that is the subject of Chap. 4. The following paragraphs describe the main test case design strategies or testing criteria for state-based testing:

Piecewise Coverage

Piecewise coverage concentrates on exercising distinct specification pieces, for example, coverage of all states, all events, or all actions. These techniques are not directly related to the structure of the underlying state machine that implements the behavior, so it is only incidentally effective at finding behavior faults. It is possible to visit all states and miss some events or actions, or produce all actions without visiting all states or accepting all events. Binder discusses this in greater detail [16].

Transition Coverage

Full transition coverage is achieved through a test suite if every specified transition in the state model is exercised at least once. As a consequence, it covers all states, all events, and all actions. Transition coverage may be improved if every specified transition sequence is exercised at least once; this is referred to as n-transition coverage [16], and it is also a method sequence-based testing technique.

Round-trip Path Coverage

Round-trip path coverage is defined through the coverage of at least every defined sequence of specified transitions that begin and end in the same state. The shortest round-trip path is a transition that loops back on the same state. A test suite that achieves full round-trip path coverage will reveal all incorrect or missing event/action pairs. Binder discusses this in greater detail [16].

Figure 3.6 shows the statechart diagram of the `VendingMachine` component specification. Each circled number in the statechart diagram refers to a single transition, specified in the state table for this diagram in Table 3.6. The state table is an alternative representation of the statechart diagram that focuses on the transitions rather than on the states. The state table contains no detailed calculations that are specified inside the state symbols of the statechart diagram. A state table is a great tool for representing many transitions between only a few states, and is additionally extremely useful for test case definition.

A test case comprises a precondition, a conditional expression that should be true before the test event is executed, an event, or a sequence of events, and a postcondition that should be true after the events have been executed. The pre and postconditions are made up of a number of conditional expressions that refer to the input parameter values of the event as well as to the internal attributes of an object before and after event execution. Sometimes they also include the actions that are performed on other associated objects, that is, if they effectively change the state of these other objects. We may also call this the outcome of an event. The combination of the object's internal attributes is its internal state. An internal state is any arbitrary combination of an object's attributes. Sometimes this is also referred to as state machine or physical state. In system development we are usually only concerned with value domains of attribute combinations that cause an object to exhibit different behavior. Such a value domain represents an externally visible or logical state.

A state table shows the logical states from the corresponding statechart diagram as part of the precondition that must be fulfilled before a state transition can take place. It shows the final states as part of the postcondition that should hold after the transition, and the event that triggers the transition. All these items come in a form that we can easily transfer into test cases. In fact all the entries in Table 3.6 represent tests that lead to full transition coverage

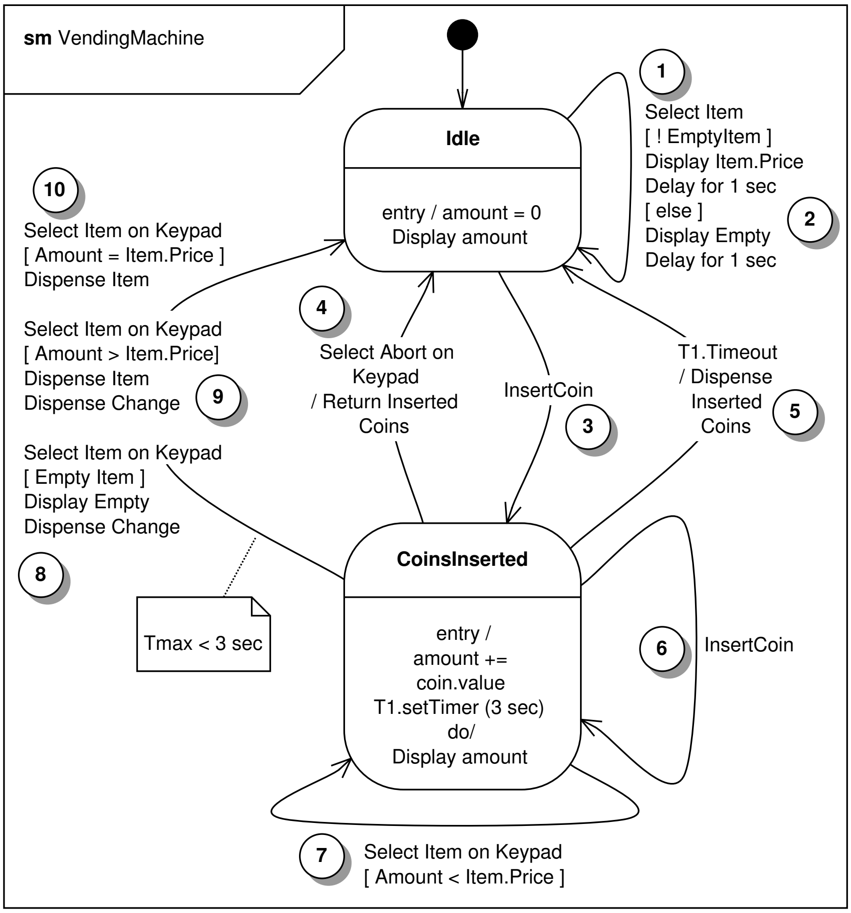


Fig. 3.6. Component specification statechart diagram for the *VendingMachine* component

of the state model. The tests, however are all abstract, because they do not define explicit values for the variables in the state table. When we have more information about the number of items that the vending machine is going to sell, their prices, and the currency that will be accepted by the machine, we can instantiate each abstract test and replace it by a number of different concrete test cases with real input values. Each of the abstract test cases in Table 3.6 can be instantiated according to these decisions. For example, Table 3.7 shows some instantiated test cases from Table 3.6. These tests are still abstract, since we cannot execute them. But they are a lot more concrete than the abstract tests in Table 3.6. We still have to add more specific information to come up with a concrete implementation of these test cases. For

Table 3.6. State transition table derived from the specification statechart diagram of the *VendingMachine* component

No	Initial State	Precondition	Transition	Postcondition	Final State
1	Idle	[!EmptyItem]	SelectItem (Item)	Display(Item.Price)	Idle
2	Idle	[EmptyItem]	SelectItem (Item)	Display(Empty)	Idle
3	Idle		InsertCoin (Coin)	Display(Amount)	Coins Inserted
4	Coins Inserted		abort	Return Inserted Coins	Idle
5	Coins Inserted		Timeout	Dispense Inserted Coins	Idle
6	Coins Inserted		InsertCoin (Coin)	Display(Amount)	Coins Inserted
7	Coins Inserted	[Amount < Item.Price]	SelectItem (Item)	Display(Amount)	Coins Inserted
8	Coins Inserted	[EmptyItem]	SelectItem (Item)	Display Empty; Dispense Change	Idle
9	Coins Inserted	[Amount > Item.Price]	SelectItem (Item)	Dispense Item; Dispense Change	Idle
10	Coins Inserted	[Amount = Item.Price]	SelectItem (Item)	Dispense Item	Idle

example, we have not stated any prices of the items yet, or we are not specific about the time that we have to wait before the timeout occurs in test case 5.2. These are specification pieces that are eventually determined during the specification and development of the vending machine. So, in the same way in that we remove the abstraction of our system, we can in parallel remove the abstraction of our testing system and move toward more concrete representations of the testing software. The activities that we have to perform along the abstraction/concretization dimension of the development process introduced in Chap. 2 are exactly the same for the development of the vending machine system and for the development of its testing system.

3.2.8 Behavioral Modeling with Activity Diagrams

Activity diagrams can be regarded as variations of statechart diagrams in which the states represent the performance of activities in a procedure and

Table 3.7. Instantiated concrete tests from the state table

No.	Initial State	Precondition	Transition	Postcondition	Final State
1.1	Idle	Item1 ==Empty	SelectItem (Item1)	Display (Empty)	Idle
1.2	Idle	Item2 ==Empty	SelectItem (Item2)	Display (Empty)	Idle
1.3	Idle	Item3 ==Empty	SelectItem (Item3)	Display (Empty)	Idle
1.4	Idle	Item4 ==Empty	SelectItem (Item4)	Display (Empty)	Idle
1.5	Idle	Item5 ==Empty	SelectItem (Item5)	Display (Empty)	Idle
...	...	...	...	...	...
2.1	Idle	Item1 !=Empty	SelectItem (Item1)	Display (Item1.Price)	Idle
2.2	Idle	Item2 !=Empty	SelectItem (Item2)	Display (Item2.Price)	Idle
2.3	Idle	Item3 !=Empty	SelectItem (Item3)	Display (Item3.Price)	Idle
2.4	Idle	Item4 !=Empty	SelectItem (Item4)	Display (Item4.Price)	Idle
2.5	Idle	Item5 !=Empty	SelectItem (Item5)	Display (Item5.Price)	Idle
...	...	...	...	...	...
3.1	Idle		InsertCoin (10ct)	Display (0.10)	Coins Inserted
3.2	Idle		InsertCoin (20ct)	Display (0.20)	Coins Inserted
3.3	Idle		InsertCoin (50ct)	Display (0.50)	Coins Inserted
3.4	Idle		InsertCoin (1EUR)	Display (1.00)	Coins Inserted
3.5	Idle		InsertCoin (2EUR)	Display (2.00)	Coins Inserted
4.1	Perform tests 6.1 to 6.3				
4.2	Coins Inserted		abort ()	CashUnit.dispense () == 0.80EUR	Idle
5.1	Perform tests 6.1 to 6.3 and wait for some time				
5.2	Coins Inserted		Timeout ()	CashUnit.dispense () == 0.80EUR	Idle
6.1	Perform test 3.1				
6.2	Coins Inserted		InsertCoin (20ct)	Display(0.30)	Coins Inserted
6.3	Coins Inserted		InsertCoin (50ct)	Display(0.80)	Coins Inserted
7.1	Perform test 3.1				
7.2	Coins Inserted	0.10 < Item1.Price	SelectItem (Item1)	Display(0.10)	Coins Inserted
...	...	...	...	...	...

the transitions are triggered by the completion of these activities. Activity diagrams depict flow of control through a procedure, so they are very similar to traditional control flow graphs, although activity diagrams are more flexible in that they may additionally define control flow through multiple instances. In general this is a procedural collaboration between objects. This is achieved through so-called swimlanes that group activities with respect to which instance is responsible for performing an activity. Essentially, an activity diagram describes flow of control between instances, that is, their interactions, and control flow within a single instance. Activity diagrams can therefore be used to model procedures at all levels of granularity, even at the business process level.

Activity Diagram Concepts

An activity diagram is comprised of actions and results. An action is performed to produce a result. Transitions between actions may have attached guard conditions, send clauses, and action expressions. Guard conditions have the same purpose as in statechart diagrams, and send clauses are used to indicate transitions that affect other instances. Transitions may also be subdivided into several concurrent transitions. This is useful for specifying parallel actions that may be performed in different objects at the same time.

While the statechart diagram displays an overview of all activities that can be performed with a component, the activity diagram shows how these activities are implemented in detail, and with which other objects or components they interact. This is the reason for why statechart diagrams are the better choice for component specifications and activity diagrams are the better choice for component realizations. Statechart diagrams concentrate more on “what” will be implemented and activity diagrams more on “how” it will be implemented.

3.2.9 Activity Diagram-Based Testing

UML activity diagrams are mainly used for typical structural testing activities; it means they provide similar information as source code or control flow graphs in traditional white box testing, although at a much higher level of abstraction if necessary. Developing activity diagrams may be seen in most cases as programming without a specific programming language.

Testing of Control Flow Within a Single Instance

Control flow testing within an instance corresponds to a typical white box unit test, though it is only valuable in component development and testing. Component-based testing is concerned more with the integration of objects and their mutual interactions rather than with their individual internal workings. For unit testing, activity diagrams provide typical traditional code coverage measures, although at a higher level of abstraction. An activity may be a single low-level statement, a block of such statements, or even a full procedure with loops and decisions. Typical code coverage criteria can be adapted easily to cope with activity diagram concepts. Traditional control flow graphs and UML activity diagrams are essentially the same. Beizer treats control flow-based testing thoroughly [11]. We can identify flows of control within an instance of an activity diagram that we can map to traditional coverage criteria:

- Testing the control flow graph through traditional coverage criteria.
- Control flow coverage of each activity in the activity diagram (solid arrow).

Testing of Control Flow Between Instances

Much more interesting for component-based testing with the UML is activity that is spread over a number of different objects. This reflects the collaborations of objects, their mutual effort toward a single goal. In this case it is the procedure of the activity. Such higher-level procedures cross component or object boundaries. At a boundary between two objects any flow of control is translated into some operation invocation or some signal invocation between the objects. The client object calls the methods of the server object. Here we have a typical contract at the particular connection between the two objects, so for testing we have to go back to the structural model and the behavioral model of each entity and derive appropriate test cases for assessing this interaction point. We can identify flows of control between instances of an activity diagram that we can map to traditional coverage criteria:

- Message flow coverage in the activity diagram (dashed arrow).
- Signal flow coverage in the activity diagram (dashed arrow).

High-level transactions are typically composed of lower-level transactions of many different objects. The decomposition activity is responsible for creating these calling hierarchies between transactions. If we base our testing on higher-level transactions, for example, on transactions in a use case model, activity diagrams display which objects are participating in a transaction. Each modeled transaction defines all its associated objects. So, when we start testing our system, we know which objects we will have to assemble and create, or for which objects in a transaction chain we will have to devise test stubs. Figure 3.7 shows the activity diagram for the `VendingMachine` operation `SelectItem`. It shows the internal flow of control of the operation and the other objects that participate in this activity. The other objects are invoked through signals in this particular instance. The circled numbers indicate the different paths through the diagram. They can be used to facilitate test case identification. Table 3.8 shows the corresponding test scenarios that can be de-

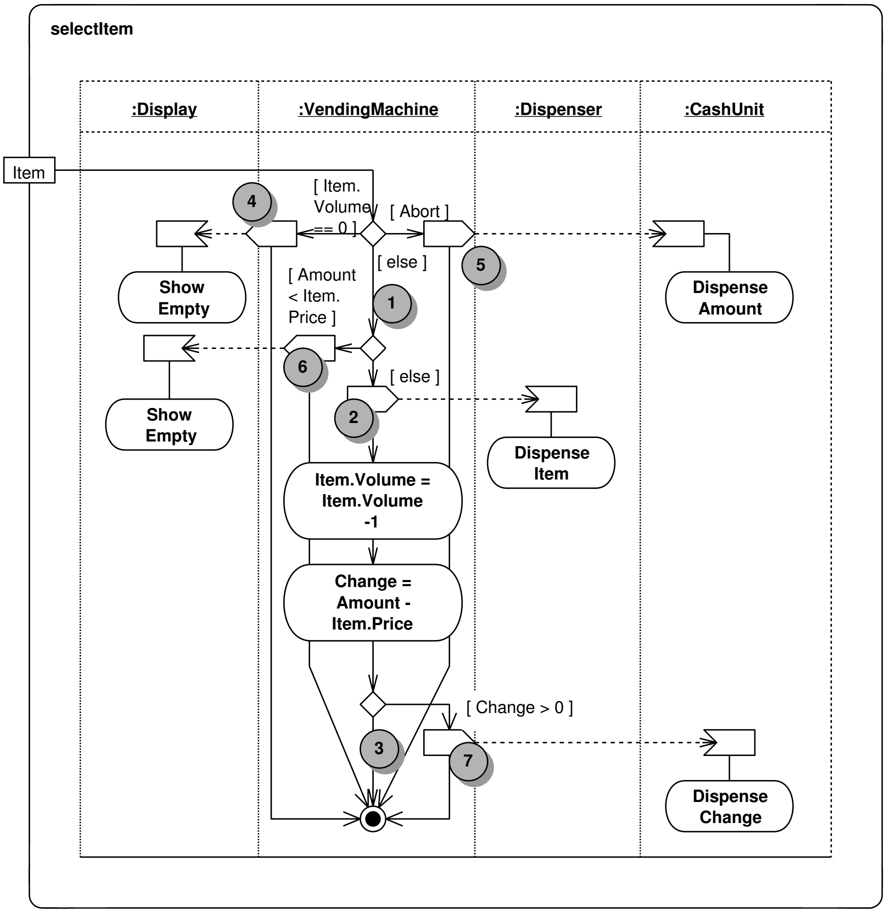


Fig. 3.7. Component realization activity model for the *VendingMachine* component operation *selectItem*

rived from the activity model. Each test scenario follows a distinct path along the control flow. The test cases represent full path coverage of the model. The conditions for each test case define the parameter settings that lead to the coverage of the defined path. After we have derived the test scenarios from the model, the next step is to define concrete test cases in the same way as in Table 3.7. In fact, we have already defined these test cases in Table 3.7.

3.2.10 Interaction Modeling

Interaction modeling represents a combination of dynamic and structural modeling. It concentrates mainly on the dynamic interactions between instances. The UML provides two diagram types for modeling dynamic inter-

Table 3.8. Test scenario identification from the activity model

No.	Covered Path	Conditions for the test case
1	1-2-3	[Item.Volume $>$ 0] [! Abort] [Amount == Item.Price]
2	1-2-7	[Item.Volume $>$ 0] [! Abort] [Amount $>$ Item.Price]
3	4	[Item.Volume == 0]
4	5	[Abort]
5	1-6	[Item.Volume $>$ 0] [! Abort] [Amount $<$ Item.Price]

action: sequence diagrams and collaboration diagrams. Their concepts are introduced in the next paragraphs.

Sequence diagrams and collaboration diagrams define the interactions on the basis of which objects communicate. This includes also how higher-level functionality, or a scenario, is spread over multiple objects, and how such a scenario is implemented through sequences of lower-level method invocations. Sequence and collaboration diagrams essentially show the same information content, but with a different focus, and they are both quite similar to activity diagrams.

Sequence Diagram

A sequence diagram shows interactions in terms of temporally ordered method invocations with their respective input and return parameters. The vertical axis shows the passage of time, and the horizontal axis the objects that participate in an interaction sequence. Through its focus on time passage, sequence diagrams also illustrate the life time of objects; that means through which occurrences they are created and destroyed. Labels can indicate timing properties for individual occurrences, so sequence diagrams are valuable for modeling and specifying real-time requirements. Messages that are sent between the instances can be synchronous, meaning that a sub-activity is completed before the caller resumes execution, or asynchronous, meaning that the caller resumes execution immediately without waiting for the sub-activity to finish. In the second case, the calling object and the called object execute concurrently. This is important for embedded system development. Messages in sequence diagrams can take the same format as transition labels in statechart diagrams, although they do not have the action expression. In other words, a message can also be made conditional through a guard expression. The format of a message is defined as follows:

```
[ guard_condition ] message_name ( parameter_list )
```

A sequence diagram starts with a single interaction; this is the considered scenario that triggers the whole sequence of messages which are spread over the participating objects. Figure 3.8 displays the sequence diagram for the operation `VendingMachine::SelectItem` according to the activity diagram in Fig. 3.7.

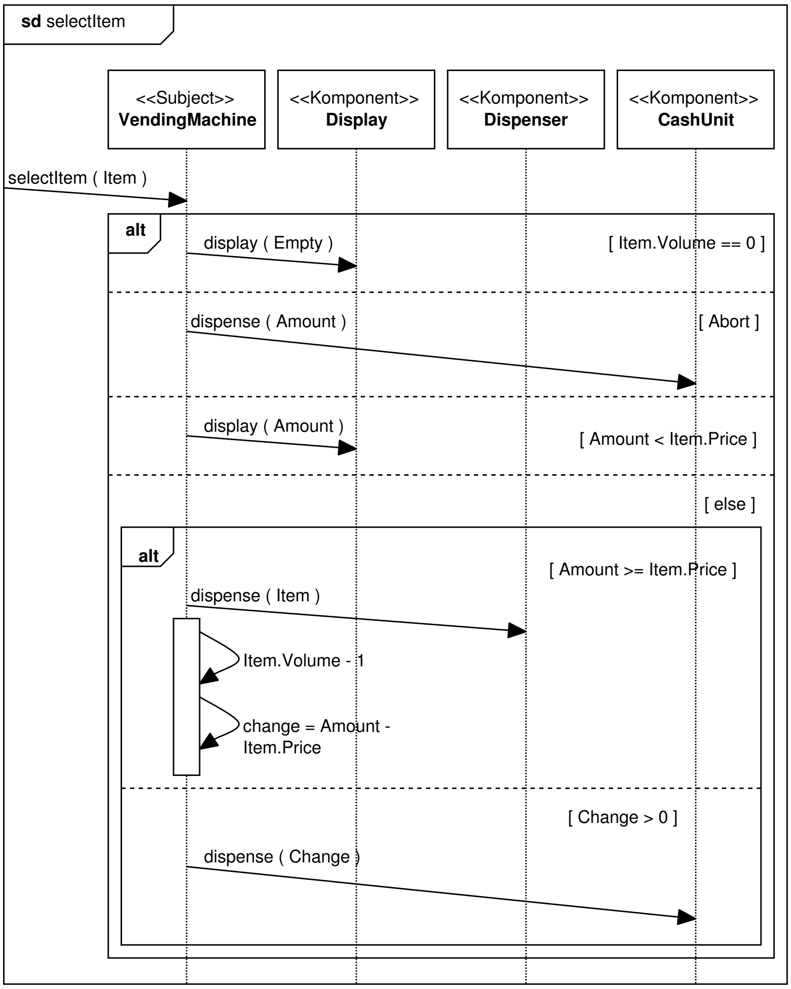


Fig. 3.8. Component realization sequence model for the *VendingMachine* component, operation *selectItem*

Collaboration Diagram

A collaboration diagram focuses more on structure and how it relates to dynamic interactions. It is similar to a class or object diagram, since it may also show internal realization of an object, that is, its subordinate objects. Essentially, a collaboration diagram shows the same interactions as the corresponding sequence diagram. However, in collaboration diagrams messages are numerically ordered, and associated with a single interaction between two objects rather than sequentially associated with a lifeline, as in the case of sequence diagrams. An interaction is a call path within the scope of a collaboration [16]. Interactions in a collaboration diagram have the following format:

```
sequence_number : [ guard_condition ]
  message_name ( parameter_list )

sequence_number : * [ iteration_condition ]
  message_name ( parameter_list )
```

The sequence number is an integer or sequence of integers which indicates the nesting level of a transaction sequence: 1 always starts the sequence, 1.1 represents the first sub-transaction on the first nesting level, 1.2 represents the second sub-transaction on the first nesting level, etc. 1.2a and 1.2b represent two concurrent messages which are sent in parallel. An asterisk indicates repeated execution of a message. The iteration is defined more specifically in the iteration condition. This is an expression that specifies the number of repetitive message executions. Figure 3.9 displays the sequence diagram for the operation `VendingMachine::SelectItem` according to the activity diagram in Fig. 3.7 and the sequence diagram in Fig. 3.8.

3.2.11 Interaction Diagram-Based Testing

Sequence and collaboration diagrams are typical control flow diagrams, although with slightly different foci. As the term interaction diagram implies, they concentrate on control flow through multiple interacting instances. For testing, the two diagram types may be represented as abstract control flow graphs that span multiple entities. With that respect we can apply all typical traditional control flow graph-based test coverage criteria as outlined in [11]. This includes path and branch coverage criteria as well as more exotic test case selection techniques such as round-trip scenario coverage [16]. Since UML diagrams are always also more abstract than traditional control flow graphs, the test targets may be more abstract. Binder identifies some typical problems that may be discovered through sequence diagram-based testing [16]:

- Incorrect or missing output.
- Action missing on external interface.
- Missing function/feature (interface) in a participating object.
- Correct message passed to the wrong object.

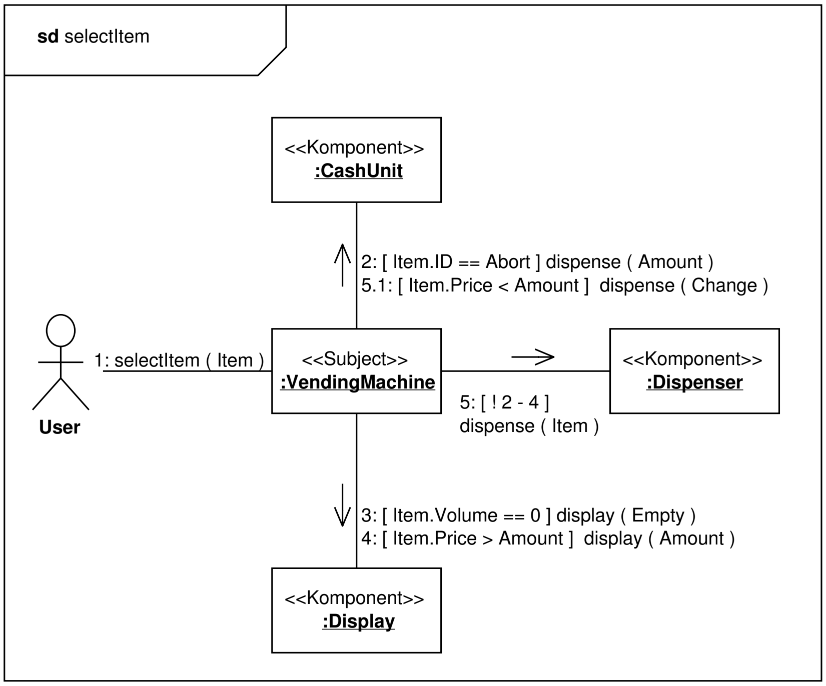

sd selectItem
<<Komponent>> :CashUnit
2: [Item.ID == Abort] dispense (Amount)
5.1: [Item.Price < Amount] dispense (Change)
1: selectItem (Item)
<<Subject>> :VendingMachine
<<Komponent>> :Dispenser
5: [! 2 - 4] dispense (Item)
User
3: [Item.Volume == 0] display (Empty)
4: [Item.Price > Amount] display (Amount)
<<Komponent>> :Display

Fig. 3.9. Component realization collaboration model for the *VendingMachine* component operation *selectItem*

- Incorrect message passed to the right object.
- Message sent to destroyed object.
- Correct exception raised, but caught by the wrong object.
- Incorrect exception raised to the right object.
- Deadlock.
- Performance.

The items in the list make the nature of interaction diagrams and their value for testing apparent. In addition to typical control flow issues these diagrams put considerable weight on collaboration and how that may be checked. From the sequence diagram in Fig. 3.8 we may derive the test sequence displayed in Table 3.9, and from the collaboration diagram in Fig. 3.9 we may derive the test sequence displayed in Table 3.10.

In this section we have looked at how UML models may be used to derive abstract test cases and test targets for a component-based system. It means we can use the model of the system for devising the testing model of the same system, and, eventually, the testing code, in an embodiment step. I call this model-based testing since the specification of the test software is derived from UML models.

Table 3.9. Test scenario identification from the sequence model

No.	Test Sequence	Collaboration
1	[Item.Volume == 0]	Display.Show (Empty)
2	[Item.Volume > 0] [Abort]	CashUnit.Dispense (Amount)
3	[Item.Volume > 0] [! Abort] [Amount < Item.Price]	Display.Show (Item.Price)
4	[Item.Volume > 0] [! Abort] [Amount >= Item.Price]	Dispenser.Dispense (Item)
5	[Item.Volume > 0] [! Abort] [Amount >= Item.Price] [Amount > Item.Price]	Dispenser.Dispense (Item) CashUnit.Dispense (Change)

Table 3.10. Test scenario identification from the collaboration model

No.	Collaboration	Condition for Collaboration
1	CashUnit.Dispense (Amount)	[Abort]
2	Display.Show (Empty)	[Item.Volume == 0]
3	Display.Show (Amount)	[Item.Price > Amount]
4	Dispenser.Dispense (Item)	[! Abort] [Item.Volume > 0] [Item.Price <= Amount]
5	CashUnit.Dispense (Change)	[Item.Price < Amount]

In the next section I give a brief overview on OMG's UML Testing Profile [120]. This is a specially defined way of using the UML for the design and implementation of the test software for a component-based system. It provides UML concepts that are specifically geared toward using this notation for the development of testing artifacts. I call this test modeling because based on this profile we can define how we are going to test a system in an abstract form.

3.3 Test Modeling

The OMG's Unified Modeling Language is initially concentrating only on architectural and functional aspects of software systems. This manifests itself in the following different UML diagram types:

- Use case diagrams describe the high-level user view on a system and its externally visible overall functionality.
- Structural diagrams are used for describing the architectural organization of a system or its parts thereof.
- Behavioral diagrams are used to model the functional properties of these parts and their interactions.
- Implementation diagrams can be used to describe the organization of a system during runtime, and how the logical organization of an application is implemented physically.

The modeling and development of the testing infrastructure also involves the description and definition of testing architectures, testing behavior, and physical testing implementation including the individual test cases. So, test development essentially comprises the same fundamental concepts and procedures as any other normal software development that concentrates on function rather than on testing. The testing infrastructure for a system is also software after all. Out of this motivation, the OMG has initiated the development of a UML testing profile that is specifically addressing typical testing concepts in model-based development [120]. The UML testing profile is an extension of the core UML, and it is also based on the UML meta-model. The testing profile supports particularly the specification and modeling of software testing infrastructures. It follows the same fundamental principles of the core UML in that it provides concepts for the structural aspects of testing such as the definition of test components, test contexts, and test system interfaces, and for the behavioral aspects of testing such as the definition of test procedures and test setup, execution, and evaluation. The core UML may be used to model and describe testing functionality since test software development can be seen as development for functional software properties. However, software testing is based on a number of very special additional concepts that are introduced in the following subsections and defined through the testing profile.

3.3.1 Structural Aspects of Testing

The UML testing profile defines the test architecture that copes with all structural aspects of testing in the UML. The test architecture contains test components and test contexts and defines how they are related to the specified system under test (SUT), the subsystem, or the component under test (i.e., the tested software). A test context represents a collection of test cases, associated with a test configuration that defines how the test cases are applied to the SUT. A test configuration may comprise a number of test components and describes how they are associated with the tested component, the system under test. A very special test component is the arbiter. It evaluates the test results and assigns an overall verdict to a test case. Feasible verdicts for a test result are pass, inconclusive, fail, and error. Figure 3.10 summarizes the structural concepts of the testing profile.

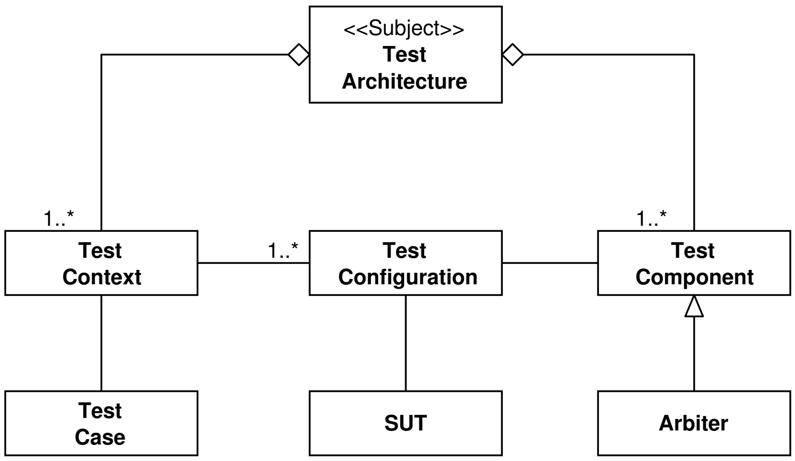


Fig. 3.10. Structural concepts of the testing profile

3.3.2 Behavioral Aspects of Testing

The test behavior is defined through a number of different concepts in the testing profile. The most important concept is undoubtedly the test case. It specifies what will be tested, with which inputs and under which conditions. Each test case is associated with a general description of its purpose. This is termed test objective and it essentially defines the test case. Each execution of a test case may result in a test trace. This represents the different messages that have been exchanged between a test component and the SUT. Finally, a test case also comprises its verdict. This indicates whether the test passed or failed. Figure 3.11 summarizes the concepts that are related to a test case.

The behavior of a test case comprises a test stimulus that sends the test data to the SUT to control it, and the test observation that represents the reactions of the SUT to the sent stimulus. The assessment of the SUT's reactions to a stimulus is performed by a validation action. Its outcome is the verdict for the test case. A pass verdict indicates that the SUT adheres to its expectations, a fail verdict indicates that the SUT differs from its expectations, an inconclusive verdict means that neither pass nor fail can be assigned, and the error verdict indicates an error in the testing system. The test behavior is summarized in Fig. 3.12.

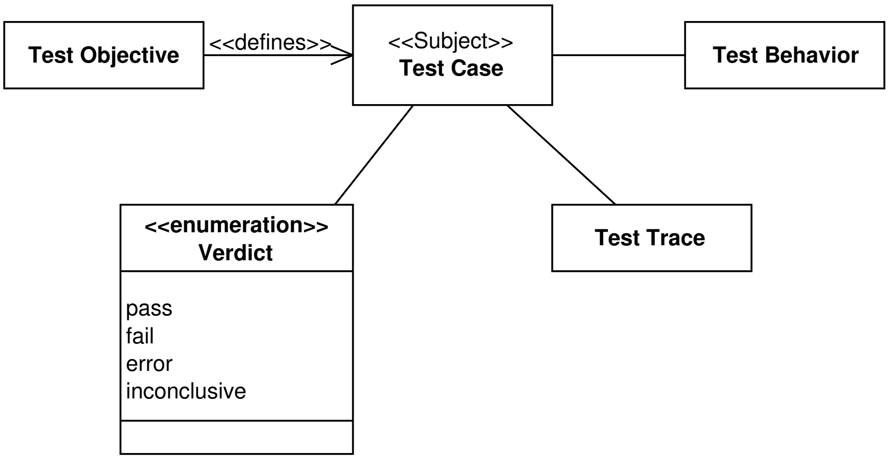


Fig. 3.11. Concepts of a test case in the testing profile

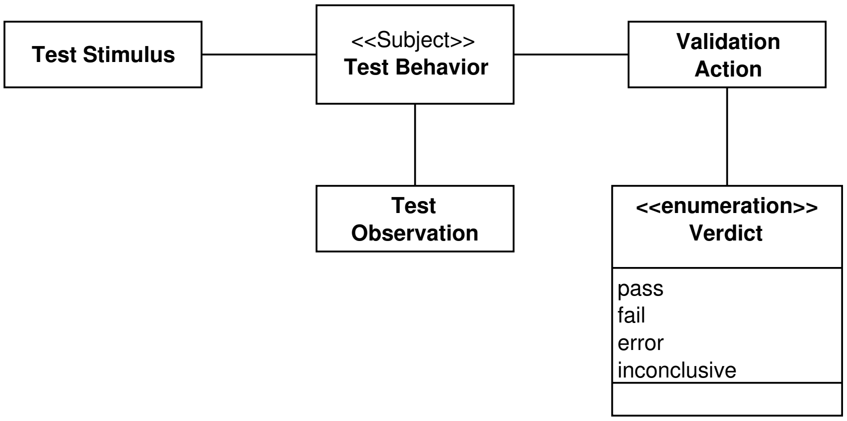


Fig. 3.12. Concepts of the behavior of a test case in the testing profile

Stimuli that are sent to an SUT and observations that are received from an SUT represent the test data of a test case. They are referred to as test parameter. A test parameter may comprise any combination of arguments, data partitions, or coding rules. An argument is a concrete physical value of a test parameter and a data partition is a logical value of a test parameter, such as an equivalence class of valid arguments. Coding rules are required if the interface of the SUT is based on distinct encodings, e.g., XML, that must be respected by the testing system. Figure 3.13 gives an overview of the test data associations in the testing profile.

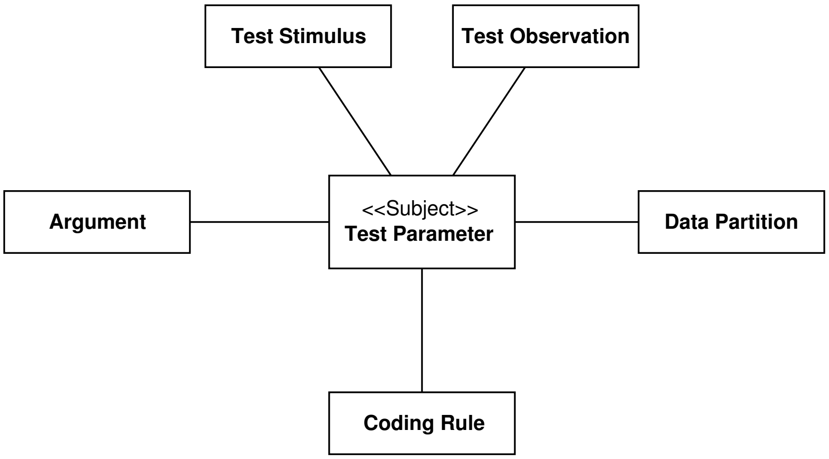


Fig. 3.13. Test data concepts in the testing profile

3.3.3 UML Testing Profile Mapping

The previous paragraphs have introduced the fundamental concepts of the UML Testing Profile. Its full specification may be found at the OMG's Web site [81]. Since in model-based testing all testing concepts are derived from UML models, and in test modeling they are expressed through the UML, they should be somehow related. In other words, we should be able to map the concepts of the core UML to the concepts of the UML Testing Profile. Basically, such a mapping provides some advice on how to apply the concepts of the profile in a model-driven development project. This is laid out in the following paragraphs.

Structural Concepts of the Testing Profile

Figure 3.10 identifies the structural concepts of the testing profile. These are the test architecture, made up of the test context with the test configuration that defines the test component. The test configuration is associated with the system under test (SUT). Additionally, we have the test case and the arbiter.

Test Architecture

The test architecture comprises everything that is related to the test of an application. It provides the framework for all testing models. On one side we will produce a system model whose structure is represented by the system architecture, and on the other side, for testing, we will devise a system that is represented by the test architecture. Both architectures are represented by UML structural diagrams. They can be developed independently from each other, in which case we will probably put them into separate models. So, we will have a system architecture model, e.g., the containment hierarchy of the `VendingMachine` and, additionally, a testing system architecture model, e.g., the containment hierarchy of the `VendingMachineTester`. This will comprise the entire testing system for the `VendingMachine`. Alternatively, we can develop both architectures in tandem so that they appear in the same model. The second approach is realized mainly through built-in contract testing that will be the subject of Chap. 4.

Both approaches have advantages and disadvantages, and they can be readily supported through typical product family concepts which I have introduced in Chap. 2. Both architectures should be somehow identifiable, otherwise it is difficult to separate testing functionality from original functionality. For example, this can be done through the special stereotype ≪testing≫ that indicates a model element as belonging to the test architecture. With the testing stereotype, we can then indicate a variant of the product family that specifically addresses testing issues, or that represents a testing system. How this can be done is laid out in Chap. 5.

Test Context

The test context represents a scoping concept that incorporates test cases, test components, and test configurations. It can be regarded as a superordinate test component in the spirit of a KobrA context realization. It can also be indicated through the stereotype ≪context≫ with an additional ≪testing≫ stereotype that separates it from the normal context of a component (in the system architecture). For example, we can have the context of the vending machine represented by the component ≪context≫`VendingMachineContext` and in parallel we can have ≪testing context≫`VendingMachineTesterContext`. With this organization we would define two entirely separate systems, the system of the vending machine, and the testing system for the vending machine.

Test Configuration

The test configuration represents an assembly of test components with test cases, and it defines how these are related to the tested system (SUT). Essentially this comes down to a sub-tree of a containment hierarchy that represents all testing for a system. It defines which test components are associated with which SUTs. We can use the full range of structural diagrams to express a test configuration. Here, the focus is on how the test architecture is associated with the system architecture and which test components are responsible for which functional components. I will give more details on how test configurations may be defined and used in model-based development in Chaps. 4 and 5.

Test Component

The test component is really a component in its conventional sense. It encapsulates everything that is necessary to organize and invoke test cases, so it encapsulates test data and test behavior, and it is associated with a tested component, i.e., SUT. I will give more details on how to design and use test components or tester components in Chap. 4.

System Under Test (SUT)

The system under test is usually also a component within a containment tree. So, the SUT can be represented by any of the UML's structural models. But it can also appear as an object in a sequence, collaboration, or activity diagram. In fact, the SUT is always the receiver of an event that is invoked from a test case in a test component. In Chap. 4, I will use the terms tested component or testable component for the SUT.

Test Case

The test case is the most fundamental concept in testing and it represents multiple associations between the UML concepts and the testing profile concepts. The specification of a test case requires pre and postconditions, expected and observed result, an event or an event sequence, input parameters, some sort of validation that compares expected and observed result, and a verdict as a result of this validation. Pre and postconditions are defined through behavioral models in the original system. We can derive these from the statechart diagram or from the functional model, i.e., operation specification or use case description. A precondition typically defines an initial state that is required for the test event to be invoked. This is represented by a sequence of operation invocations that set a tested component into an initial state according to the state model. A postcondition represents an expected final state after the execution of the event according to the state model. Usually, this is difficult to check unless a tested component provides explicit support through additional state checking operations, i.e., additional testing interface in Chap. 4. The specification of a test case has several sources in the UML model. The most important ones are the behavioral models, but structural models may also provide essential information, especially if a test case concentrates on message sequences. A test case as such is primarily defined through an activity diagram that defines the component interactions and the control flow for initializing the test case (precondition), finalizing the test case (postcondition and result), and executing the event. Each test case can be specified through an individual activity diagram if the test behavior is complex, or several test cases can be represented by a single activity diagram if the test behavior is simple.

Arbiter

The arbiter is a special test component that is responsible for assessing the observed outcome of a test case against the specification of the expected outcome. As a procedure, for example, as part of a test case or as part of an entire test component, it performs the validation. The arbiter will contain all attributes and methods that are required for checking the results of the test cases and assigning verdicts. Depending on the organization of the testing system, we can have an arbiter for each test case, for a number of test cases, or for an entire test component, and they are typically related or communicate through some attributes.

Behavioral Concepts of the Testing Profile

Figure 3.12 identifies the behavioral concepts of the testing profile. The test behavior is mainly related to the test case. But we can also have test behavior that draws a number of test cases together, and executes them as a sequence, for example. So we can also have test behavior at the test component level. The validation action is part of the arbiter, and it will be designed mainly in terms of a procedure in an activity diagram. It is responsible for evaluating the test result that the test observation is producing. The test stimulus is the actual event that is invoked on the tested component. All behavioral models can be used to define the test behavior, but the activity diagram is probably the primary model for describing testing behavior. Complex state models are not so common for testing.

Test Objective

The test objective is a general description of what should be tested. In the most general sense, the test objective that can be derived from a behavioral model is clearly the test of behavior. Since this is too broad a terminology, and a test objective may also be associated with a test case in the testing profile, we can be more specific and identify a number of test objectives, each of which maps to one or more test cases. Typical test objectives are coverage criteria, for example, model artifact coverage, i.e., state model coverage, and the like. A sensible approach would be to define a test component for each test objective.

Chapters 4 and 5 give some examples of how the concepts of the testing profile may be applied in real projects. Some of the structural concepts are readily applied in built-in contract testing, because this technology is based on a specific testing architecture. The behavioral concepts are more implicitly applied, and that is mainly within the test components. At higher levels of abstractions the structural concepts are clearly more important for the definition of the testing architecture and the abstract tests. At the implementation level the behavior becomes also more important and apparent. Chapter 5 gives some more details on the implementation-level activities. But I have to say that the focus of this volume is more on the abstract testing concepts.

3.3.4 Extension of the Testing Profile

In general, a profile is a predefined way of using or applying the UML with a particular focus. The focus of the testing profile is clearly to support the design and implementation of testing architectures. Other profiles may be domain- or company-specific. These support modeling and design with domain-specific concepts and artifacts, or with concepts and artifacts that are used only within a company. Hence, a profile represents a highly flexible collection of items that can be used for a particular purpose, in this instance the purpose of testing. But it must really be regarded as a core set of things that can be easily extended however developers or organizations see fit. In Chap. 4, I will introduce a number of concepts and modeling elements that are in fact extending the original testing profile of the UML. The testing interface is such a concept, for example. Or I will use special stereotypes, e.g., ≪testing≫. Stereotypes can be defined and their semantics described easily. I am not going to introduce these concepts formally, or present a formal extension model of the testing profile, but I am rather going to use them informally where necessary and explain their meanings.

3.4 Summary

UML and testing are a perfect fit. The UML can be used to specify any arbitrary aspect of a computer system, and testing is based specification documents in the same way as functionality. Specifications describe in a semi-formal way how a system is supposed to look, and what it is supposed to do. Testing validates whether the executable implementation of a system exactly conforms to what the specification defines. The UML can additionally be used to model and specify the testing framework for a computer system.

Model-based testing, or testing that is based on the UML, is not so different from the traditional testing criteria that have been around for years. In general, we will find the same concepts that code-based testing techniques propagate in model-based testing. This is the fact, because models are abstract representations of code, and whatever we apply at the concrete code level can be applied at a more abstract level, the model. The only difference is that more concrete representations of a specification will lead to more concrete testing artifacts, and more abstract representations of a specification will inevitably lead to more abstract testing artifacts. The model-based testing criteria and the testing artifacts that we can derive from a UML diagram clearly illustrate that. High-level diagrams lead to high-level testing artifacts, e.g., the use case diagram maps to testing targets, and lower-level diagrams lead to lower-level testing artifacts, e.g., the behavioral model leads to test cases, albeit abstract ones, according to the level of abstraction. This observation suggests that we can exploit almost anything in a model for testing. And in the near future we will be able to extract the testing artifacts automatically from the models, given that future tools will incorporate such capabilities.

The test software for a computer system is a software system in its own right, and this can and should be specified according to the same rules and with the same means applicable to the original tested system. Model-based testing and test modeling are in fact orthogonal activities that complement each other. After all, testing is only software development and execution. The recently specified testing profile of the UML supports test modeling elegantly in that it provides the right concepts for specifically addressing testing issues that the core UML does not incorporate.

Chapters 2 and 3 have looked at component-based development and model-based testing in isolation. They are now brought together and integrated in the built-in contract testing approach, and this is the subject of the next chapter.

4

Built-in Contract Testing

Component-based development is widely expected to revolutionize the way in which we develop, deploy, and maintain software [157]. The idea of component-based development is that we can generate new applications comparatively quickly, by assembling prefabricated parts, instead of continually custom designing and developing all our software from scratch, as is largely the case under the traditional development paradigm. A more important incentive for using component-based development techniques is that we are not limited only to our own prefabricated parts, but can buy high quality parts from professional component vendors. This way we can purchase other people's domain knowledge through their components, and we can eventually develop systems for which we are lacking actual experience. For example, we can buy and use third party networking components to integrate our system into a networked environment, and we can do that wireless now, because we can find the right components. One main outcome of component-based development is that we can develop larger and more complicated systems, and can do so much more quickly than before.

The vision of component-based development is to bring software engineering more in line with other engineering disciplines where assembling new products from standard prefabricated parts is the norm, and, at the same time, to save effort and improve quality. This vision of software development presents some challenges, however. With traditional development approaches, the bulk of the integration work is performed in the development environment, giving engineers the opportunity to pre-check the compatibility of the various parts of the system before it is integrated, and to ensure that the overall deployed application is working correctly. The fact that in a custom design all software is owned by the development team, and it can readily access all software documents, greatly facilitates testing and debugging.

In contrast, component assembly suggests that components have not been designed specifically for any particular purpose, or have been designed for an entirely different purpose. Furthermore, the component infrastructure that integrates a new component will have characteristics that are completely different from the component's original infrastructure for which it had been initially developed. In other words, it is likely that a component's original deployment environment, and thus its intended usage profile, will significantly differ from its new deployment environment into which it is going to be integrated when it is reused. The Ariane 5 accident was caused through a typical integration error of this kind. An additional difficulty is the late integration implied by component assembly, so that there is often little opportunity to verify the correct operation of applications before deployment time. In summary, we can say that although component-based development provides an attractive alternative to traditional custom software design, it inhibits validation. Even if component developers may adopt rigorous test methodologies and overall quality assurance techniques for their individual reusable units, this is not much use if such a unit is integrated into an environment for which it has not been specifically designed. This other environment may use the component in such a different way that the component developer may never have anticipated in advance, thus making the original tests obsolete. In component-based development most components are reused in a way as if they had never been tested at all, making an entirely new assessment of the component necessary [176].

The expectation that we put toward component-based software development is founded on the implicit assumption that the effort we have to put into component identification, acquisition, integration, and acceptance testing is significantly lower than the effort that we have to invest in traditional custom development where we produce and test all parts of the software and have access to all artifacts. The arguments that are put forward in favor of component-based development have substantial intuitive appeal but there is little empirical evidence to validate them [176]. In component-based development we do not have most of the artifacts that are typically required for testing, so we may be forced to perform more black box integration and acceptance testing at the system level to attain the same degree of confidence in the system's reliability. And this may offset the savings that we expect to get from component-based development entirely or at least to a large extent. In short, although we can apply rigorous testing strategies in the development of individual components, this helps little to assure the quality of entire applications that are assembled from them. The argument that the quality of an assembly of components is at least as good as the quality of its individual parts does not hold.

Built-in contract testing promises a way out of this quality assurance pitfall in component-based development. It is based on the notion of equipping individual components with the ability to check their execution environment, and their ability to be checked by their execution environment at runtime. When deployed in a new system, built-in contract test components check the contract compliance of all other associated components, including the runtime system [73]. This automatically validates whether a component will be "happy" within its new environment and whether the new environment will be "happy" with the newly deployed component.

This chapter concentrates on all aspects of built-in contract testing that are related to the design of a system. The next chapter (Chap. 5) will look at how built-in contract testing can be implemented on real component platforms. In the next section (Sect. 4.1), I will briefly describe the fundamental technologies behind built-in contract testing, assertions, and built-in testing. In Sect. 4.2, I will describe the motivation for applying built-in contract testing in component-based developments and its objective, and explain how the fundamental concept of a component contract affects testing. Section 4.3 introduces the two primary artifacts in built-in contract testing, the testing interface and the tester component, and how they should be ideally designed. Section 4.4 describes the development process that can be followed to come up with the built-in contract testing artifacts. Section 4.5 summarizes and concludes this chapter.

4.1 Concepts of Built-in Testing

In the literature, built-in testing typically refers to all concepts that are added to a component's code for facilitating testing or checking assertions and conditions at runtime. Built-in testing is usually not part of the original functional requirements of a component, and it does not stay incorporated in the code beyond the release of a component. But the main argument of this book is about exactly that: the development of components with permanent built-in testing, seen as an integral part of the components right from the beginning of their construction. There are some disadvantages under certain circumstances of leaving testing artifacts built-in permanently; but we will initially concentrate only on how we can apply built-in testing techniques in component engineering and component-based software testing, and will later on discuss how to organize them so that we may get rid of them again easily if we find them troublesome.

In the following, I will describe the two primary technologies of built-in contract testing, assertions and built-in testing, in more detail.

4.1.1 Assertions

Built-in testing is not a new concept. Among the first built-in testing concepts in software engineering were assertions, although we cannot really call them built-in testing. An assertion is a boolean expression that defines necessary conditions for the correct execution of an object or a component [16, 141]. Assertions in software follow the same principles as the self-checking facilities that are readily built into hardware components. Whenever the boolean expression of an assertion is executed, the assertion checks whether the value of some monitored variable is within its expected domain. In theory, assertions can be implemented at any arbitrary location in a component, but there are distinct places where they are more useful. For example, we can have an assertion before the entry into a procedure that checks all preconditions that must be true before we can call that procedure, or we can add an assertion at the exit from a procedure that assesses whether all postconditions of the procedure after its execution are fulfilled. Other assertions may also have to be true at all times, for instance an object invariant [16]. This is more difficult to validate because we have to add code that checks the assertion periodically. An assertion consists typically of three parts:

- a predicate expression that may be coded as a simple *if* statement,
- an action expression that in its simplest form points out the problem, i.e., a print statement, and
- an enable/disable mechanism that controls the execution of the assertions.

Assertions are typically programmed during development in the same way as any other functionality, and enabled or disabled during translation. During execution, an assertion may be either true, in which case it has not found any errors, or false. This second case is called an assertion violation that triggers the action expression to be performed. The action expression may be as simple as writing a message to the screen, a method that is quite readily applied in first year programming classes, or as complicated as invoking an entire recovery mechanism [16].

Assertions represent a valuable tool for catching many typical errors that can be made explicit through assumptions and conditions, such as responsibilities, i.e., pre and postconditions, or boundary conditions of input, output, and internal variables. Moreover, assertions address the single most important inhibiting factors of testing object-oriented systems: encapsulation and information hiding. Assertions enable us to readily check the validity of an object's internal states throughout its lifetime, through a continuous assessment of the object's internal state attributes that are changed through external events. Binder discusses assertions exhaustively [16].

4.1.2 Built-in Testing

In the literature, assertions and built-in testing are often used as synonyms, e.g., [16]. But I believe they are fundamentally different concepts. In my opinion, assertions are lacking one important ingredient for representing built-in testing, that is, the notion of a test or a test case. Assertions can be used in a test and provide valuable information for error detection during test execution, but we cannot say an assertion or its execution represents testing. A test is an experiment under controlled conditions that applies a set of test cases in a test procedure or test framework [91], so I believe we can only talk about built-in testing if we have built-in test cases. I have included assertions under the topic of built-in testing because they represent the most fundamental idea behind built-in contract testing in component-based development, that is, building the testing directly into components.

Built-in testing strategies comprise the built-in self-test metaphor whose idea is derived from the self-test capabilities commonly built into hardware components. Built-in self-test components are software modules that contain their own test cases. In sloppy terms, we could say that these are software components with their own tests for checking their own implementation.

One motivation for building self-tests into a software component is to check differing variations of component implementations that are all based on a single component specification [98, 103]. In other words, a component is not merely viewed as its implementation, or the physical thing that we will deploy in our system, but it is viewed as the collection of all descriptive documents that fully describe the component's interfaces, structure, and behavior, plus an arbitrary implementation of this model. This view is fully in line with KobrA's approach to define a component, which I have introduced in Chap. 2. Assume that we have a number of component realizations with a variety of different quality properties that are all consistent with the single component model. This could well comprise a number of different realizations in one programming language, or different implementations for different platforms. Each implementation consists of two distinct parts, the component's functional implementation and its test implementation. The definition of the test cases is the same for all different implementations, and so is the definition of the component's functionality. The implementation self-test strategy is a typical component development-time testing approach. It means it is a way to test individual units. It is not so suitable for checking component interactions in an assembly of individual units.

Object technology provides the right tools for the implementation of this type of object testing. We can have a base class in Java, for instance, that includes the implementation of the functionality , and then extend that with testing code through inheritance. The testing code of the extended class provides the test cases that will check the implementation of the base class. In Java that may be achieved through the following source code lines:

```
class Component {
  // functional interface and attributes
  ...
};

class testableComponent extends Component {
  // testing interface and test cases
  ...
};
```

The extension will also provide some interface that can be used to invoke the test. If we test such a component, we can instantiate its testable version, and start the tests through the additional testing interface. If we integrate the class in an existing component infrastructure, we may instantiate the original version, the base class. During the test, the extended class can access the original functionality through the inheritance mechanism as long as the class does not define any private items that are not inherited. The advantage of this type of built-in self-test is that we can break the encapsulation boundary of the tested object, but yet keep functionality and testing entirely separate in different classes. The fact that we can break the encapsulation boundary for testing can be seen both as a cure and curse. It is a cure because we can apply highly implementation-specific test cases. After all, we can access all internal attributes, and this results in high observability and controllability of the tested object. At a first glance this is very good for testing. However, breaking the encapsulation boundary in this way is also a curse, because if we apply such highly implementation-dependent test cases we will never be able to reuse the tests in a different implementation. For instance, we might access distinct attributes in a test which do not exist in another implementation, so that we can actually scrap the test.

Another similar approach proposed by Wang et al. with a slightly different motivation is to add component self-tests and leave them permanently in an object or component implementation for reuse [165, 166, 167, 168]. The main idea behind this strategy is to exploit the inheritance mechanism of typical object technologies to transmit not only the functionality of a class to its subclasses but, additionally, also its testing functionality in the form of readily built-in test cases. The test cases are invoked through an additional testing interface that the object provides. Users of an object may access its normal interface and get its specified functional behavior in normal mode, but they can also access the object's testing interface and execute its built-in test cases in test mode. At first sight, it might seem the same as the previous approach, but there is a subtle difference. In the previous instance, the approach was merely to have unit tests attached to and detached from a component implementation in a convenient way. Here, we have the tests always built into an object, and they are inherited from the base class by all extended classes. So, we have the same built-in testing facilities that the base class provides in all inherited classes. The main motivation of this approach is that software components or objects will get the same self-test capability that can be regarded as standard in most hardware components. Additionally, in contrast with hardware components, software components may inherit their features and thus provide the same self-test capabilities in subsequent versions.

This strategy might seem appealing at a glance, but software components differ from hardware components in one important respect. While hardware components are built from materials that can physically degrade over time, software components are not. Software components are encoded in digital formats which can easily be checked (and if necessary corrected) to ensure that there is no change over time. Thus, the concept of a self-test in software that is similar to the hardware approach [166, 168] is not directly applicable or useful in component-based software testing. This type of built-in testing is only useful in component evolution and maintenance because only such activities will in fact change the code of the component and make a regression test necessary. There is no point in a component rerunning previously executed tests on itself, because by definition the component itself does not change. What can and usually does change, however, is the environment in which a component finds itself. The objective of built-in testing should therefore not be a test of a component's own functionality, because we can check that through typical unit regression testing, but an assessment of the component's environment and how well it interacts with that. In other words, we have to assess that the component's environment into which it will be deployed does not deviate from what the component was developed to expect, and that the component does not deviate from what its new environment was developed to expect. This more obvious motivation is not clearly stated in the built-in testing approach of Wang et al. In the following sections I will state the motivation for built-in contract testing more concisely, and then introduce the methodology behind built-in contract testing that specifically addresses this testing problem.

4.2 Motivation for Built-in Contract Testing

The philosophy behind all built-in testing concepts is that an upfront investment on verification and validation infrastructure pays off during reuse. In component-based development, this follows the same fundamental ideas of gaining return on testing investment through reuse of the built-in testing artifacts. The more often a component will be reused, the higher is its rate of return, and this applies to its functionality as well as its testing. This is the case only if the testing artifacts are permanently built-in, and are always deployed, or at least purchased, together with the component.

4.2.1 Objective of Built-in Contract Testing

The objective of built-in contract testing is to ascertain that the environment of a component meets the component's expectations, and that the component meets the environment's expectations. These two views are the only things that can change in component integration. This is illustrated in Fig. 4.1. Typically, if we bring components together to form a new application, we will find a number of individual units that are nearly good enough for the purpose. It is extremely unlikely that we will find all components in forms that exactly match all other components that we are going to reuse. Normally, we will have to perform some syntactic and semantic mapping at the boundary between the deployed component and the component infrastructure in Fig. 4.1. In Chap. 2, I explained how this is typically done in a development method. In order to achieve this mapping we cannot usually change the individual units and make them work with one another. This is because we do not own amenable versions,

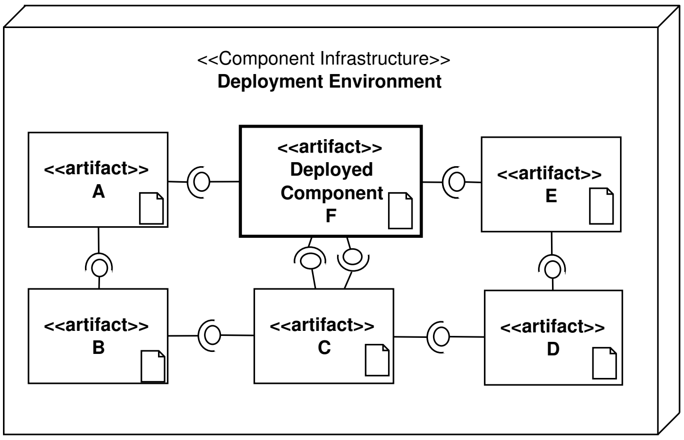


Fig. 4.1. Deployment of a component into an existing component infrastructure

i.e., we have bought COTS, or we are reluctant to change any of them. This is quite typical because although we perhaps own their source code, i.e., we use in-house development, we do not understand it well enough, and we do not want to mess it up. Attempting to change any of the components is anyway not a good idea in component-based software development. After all, we have decided to apply the component paradigm because we wanted to avoid this way of custom designing our software.

The only solution that remains is to add some so-called glue code between the components at their interfaces. This is an additional program that takes what one component "means" and transforms it into what the other component will "understand." So, for each interface between the deployed component and the component infrastructure in Fig. 4.1 we have to add such a transformation facility. At this point, the deployed component is integrated in the infrastructure, and the components may perform some meaningful interactions. I say "may" because we do not yet know for sure.

Before we can release this application we have to check the newly established component integration, and this is exactly where built-in contract testing comes into play. The idea of built-in contract testing is that every component in an infrastructure come equipped with its own built-in test cases. These tests are designed specifically to check the component's runtime environment, not its own implementation as in the previously described built-in testing approaches. Every component "knows" best what support it is expecting from its associated partners. The built-in test cases of the component represent this expectation. So, whenever a component interaction is established, the two parties in that interaction can automatically invoke their built-in tests and assess whether the interaction is sufficient.

This way of organizing component testing adds considerable value to the reuse paradigm of component-based software development because a component can complain if it is mounted into an unsuitable environment, and the environment can complain if it is given an unsuitable component. The benefit of built-in contract testing follows the principle which is common to all reuse methodologies: the additional effort of building the test software directly into the functional software results in an increased return on investment according to how often such a component will be reused. The component will be reused more frequently according to how easy it is to reuse, and built-in contract testing greatly simplifies the reuse of components. In the following section we will have a look at how built-in contract testing is designed and organized, but prior to that we look at the concepts that typically govern component interactions, component contracts.

4.2.2 Component Contracts

A prerequisite for the correct functioning of a system containing many components is the correct interaction of individual pairs of components according to the client/server model. Component-based development can be viewed as an extension of the object paradigm in which independent, self-contained units of behavior, i.e., objects or components, interact by sending messages to each other. The interaction of components is based on the client/server model. In a given interaction, one party plays the role of the client, and the other plays the role of the server. The client is the party that needs a service, and sends a request for this service to the server. The server then performs the requested service and returns the results without knowing the identity of the client. Client and server refer to roles that a component can play in an interaction. In different interactions a given component can play either the role of a server or a client, but in any particular interaction one of the parties involved plays the role of a server and the other the role of a client.

The correct functioning of a system of components at runtime is contingent on the correct interaction of individual pairs of components according to this client/server model. Component-based development can be viewed as an extension of the object paradigm in which, following Meyer [112], the set of rules governing the interaction of a pair of objects is typically referred to as a contract. This characterizes the relationship between a server object and its client object as a formal agreement, expressing each party's rights and obligations. This notion of a contract is extended to components, where different reuse and deployment contexts are so much more important, by parameterizing the pre and postconditions of components' contracts to address the configuration of components adequately that are deployed in new contexts. This extension is proposed by Reussner and others, and termed *parameterized contract* according to the "architecture-by-contract" principles. [135, 136, 137].

Testing the correct functioning of individual client/server interactions against the specified contracts goes along way toward verifying that a system of components as a whole will behave correctly; and this is fully in line with the notion of a parameterized contract and the architecture-by-contract principles mentioned above. The use of a software component is always based on a two-way contract:

- the component should operate according to its specification, and this is the server's contract,
- the system in which the component is integrated should meet the component's expectations, and this is the client's contract.

Built-in contract testing investigates whether a component deployed within a new runtime environment is likely to be able to deliver the services it is contracted to deliver; hence the name. Contracts are the most fundamental relations in component-based development. We typically have ownership and containment in a component-based application which leads to component nesting. And all these concepts inevitably lead to component contracts.

4.3 Model and Architecture of Built-in Contract Testing

As explained in the previous section, a component interacts with its environment, the other components in the infrastructure, at runtime by means of client/server interactions. One of the parties in an interaction plays the role of a client, the other that of the server. When an otherwise fault-free component is deployed in a new environment, there are only two basic things that could go wrong during its execution:

- either the component itself is used incorrectly by others,
- or one or more components that it uses and is depending on malfunction.

Both of these scenarios can be characterized in terms of the client/server relationship: the former implies that one or more of a component's clients behave incorrectly, while the latter implies that one or more of a component's servers behave inappropriately. Checking that these errors do not arise, therefore, can be characterized as checking that the contract between components is adhered to. The two situations identified above are highlighted in the generic deployment scenario illustrated in Fig. 4.1. In terms of this diagram, there are two things that should ideally be checked at runtime to ensure that a deployed component's contracts are adhered to:

- The deployed component (with the thickened border in Fig. 4.1) must check that it receives the required support from its servers. This includes the explicitly acquired servers, in this particular instance components C and E, and the implicitly acquired servers. The implicit servers are the components that the runtime support system provides, for example, a middleware platform or the operating system. They are servers in the same respect as any other explicit server component in the component

infrastructure of an application is a server. The only difference is that the components of the application implicitly assume their existence and use their services. This first scenario considers contract testing from the viewpoint of the deployed component.
- Clients of the deployed component must check that the component correctly implements the services that it is contracted to provide. In other words, clients of the deployed component must ensure that it meets its contract. This scenario considers contract testing from the viewpoint of the clients of the deployed component, in this particular instance the components A and C.

Every component in the infrastructure should therefore have its own test cases built in that reflect the component's expectation toward its environment. If a component F is a client of two other components E and C as displayed in Fig. 4.1, the purpose of the test cases built into F is to check E and C by invoking their methods, and to verify that they behave individually and collectively as expected. The same is true for components A and C that acquire the services of the deployed component F. The built-in test cases in components A and C will therefore invoke F's methods to check that F fulfills its contract in the way that A and C are expecting. Components A and C are quite likely to use F differently, so they have different built-in test suites. Each of the components' test suites will be designed according to each component's specific usage profile of F. Since the tests built into a client check the behavior of a server, they essentially check the semantic compliance of the server to the client relationship contract.

While most contemporary component technologies enforce the syntactic conformance of a component to an interface, they do nothing to enforce the semantic conformance. This means that a component claiming to be a stack, for example, will be accepted as a stack as long as it exports methods with the required signatures, i.e., `push` and `pop`. However, there is no guarantee that the methods do what they claim to do. Built-in tests offer a feasible and practical approach for validating the semantics of components, and this is in fact the main argument for built-in contract testing. In essence, this follows the same fundamental ideas of the previously introduced built-in self-tests. However, the built-in test cases here are not designed for checking a component's own implementation as in the previous cases, but for checking the new environment in which the component is deployed.

In general, testing a component involves testing state transitions as well as returned values. To support the two scenarios identified above (i.e., client viewpoint and server viewpoint), additional software artifacts must be provided in both the server and the client. This is illustrated in more detail in Fig. 4.2 and described as follows:

- In general, the client will contain built-in tests which are dedicated exclusively to checking the component's own deployment environment (its servers). This comprises explicit servers in the component infrastructure,

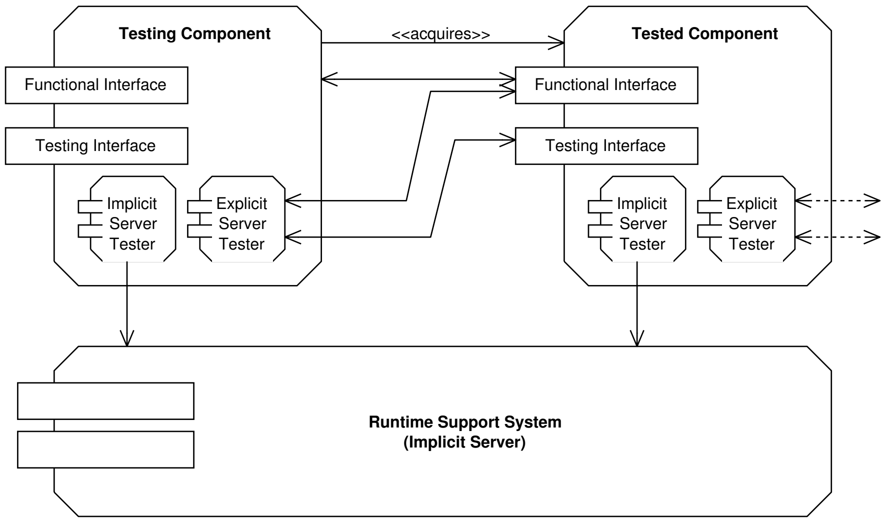


Fig. 4.2. Model of built-in contract testing

i.e., units of the application, and implicit servers that are provided through the middleware. The test cases are carefully designed to comply with trade-off requirements, and they are organized and contained in tester components. A client that contains built-in tests in that way is called a testing component. The tester component is a separate component and includes the test cases for another associated server component.

- The server will contain a contract testing interface. This adds to the client's normal functional interface and serves contract testing purposes. The contract testing interface consists of public methods for state setup and state validation. This enables access to the internal states of a component similar to what assertions provide, leaving the actual tests outside the encapsulation boundary. The clients of the component use the contract testing interface to verify whether the component abides by its contract. In this way, each individual client can apply its own test suite according to the client's intended use of the component, this means according to its own usage profile of the server. A server that provides such a testing interface is called a testable component.

The tester accesses the server's normal interface for executing the test cases, and the server's testing interface for state setup and state validation. The execution of the tester represents a full server test with which the client validates that the server provides its services correctly.

A component that plays both roles, that of the client and that of the server, will have both the built-in contract testing interface to support the testing that is performed by its own clients, and a number of tester components one for checking each of its associated servers. Such a component is called a testing and testable component.

4.3.1 Explicit vs. Implicit Servers

In general, two kinds of server components can be distinguished. The first kind, known as explicit servers, correspond to the server components defined explicitly as part of the development or implementation. These represent server components as typically understood. The second kind, known as implicit servers, correspond to services supplied by the runtime system. This kind of server is discussed further in this section.

When a computer program written in a particular language is executed, the computer must usually also include some additional software, known as the runtime system, which provides runtime support for some of the features of the used language. Typical examples are I/O operations and introspection operations that allow the program to find out some details (e.g., record size) about how objects are implemented. This runtime support software is invariably written by the vendor of the compiler technology used to develop the program, and is included in the final running system invisibly to the application program developer. The developers of such runtime support software naturally test it in the normal way to ensure that it satisfies the specified requirements. With the advent of object-oriented and component technologies, the trend has been to implement more of the features which were traditionally embedded within the runtime support software in predefined library classes. Java provides a good example of this trend, with many of the I/O and introspective facilities defined within the class library rather than invisibly in the runtime support. The difference between these library features and explicit server components of the first kind is that they are supplied implicitly. For example, the "in," "out," and "err" objects provide the standard I/O capabilities, and they are automatically visible to any Java object, but do not have to be explicitly acquired [37]. Therefore, in addition to tester components for testing the explicit servers of the kind described above, tester components are also needed for the

- implicit servers which are automatically provided as part of the standard setup,
- the invisible runtime support (RTS) software which provides support for any other features of the languages not handled by the previous item.
- support that is provided through typical component platforms, the so-called middleware.

In all the three cases the approach for handling implicit server components is the same as for explicit server components. In principle, therefore, a heavyweight runtime test of the runtime support software could include the manufacturer's acceptance tests to ensure that all the required features are properly supported. However, this would be extreme. In practice, a more carefully selected, lighter subset of these tests could be included in a component to provide a test of the runtime support software. In the next section we will have a closer look at testing interfaces in built-in contract testing and how they are designed, and, following that, we will see how they can be used in tandem with tester components.

4.3.2 The Testing Interface

Like objects, components are state machines and require state transition testing. Before a test can be executed, the tested component must be brought into the initial state that is required for a particular test. After test case execution, the test must validate that the outcome (if generated) is as expected, and that the tested component resides in the expected final state. By definition, however, the internal states of a component are hidden to outside entities by application of the principles of information hiding and encapsulation. In general, it is not possible for a system integrator to look inside a component. Components are black boxes whose functions can only be understood through the specifications of their interfaces. In fact, many components may be available only as binary images in the form of DLLs or some equivalent technology, making it impossible for an integrator to know what is inside the box. Even in the case of components where the source code is available, detailed design documentation may not be available, hence making it difficult to find faults in the component. A software component is a black box with an explicit set of provided and required interfaces. Each provided interface is a set of operations that the component provides, while each required interface is a set of operations that the component requires to perform its operations. Apart from this we have no further insight into a component. Typical components have low testability.

An external test software cannot usually set or get internal states except through the normal functional interface of the component. A specific sequence of operation invocations through the normal functional interface is usually required to set a distinct state for a test execution. Since the tests are performed to validate whether the functional interface behaves as it is expected to, it is unwise to use the functional interface to set and verify the internal states of a component and check the outcome of the tests. This means we should not use something for performing a test that is actually the subject of that test. Additionally, typical components will not readily provide output states in which the output of the component can be analyzed in order to draw conclusions on its current state. In other words, objects or components are often organized in a way that they accumulate data over some period of time and perform some internal calculations. Only eventually will they output a result that may give some hints on internally stored state information. In the meantime we could have had a number of faults that would never have emerged as failures because the component will never have exhibited these in any observable form. Any internal workings of a component are hidden; so are the faults that accumulate inside the component.

Consequently the test architecture needs to allow system integrators to use some internal information about the component that is essential for testing, without having direct access to the internal workings of the component. This problem can be circumvented by using an additional testing interface which contains special purpose operations for setting and retrieving the internal state of a component. In a general sense, a component can be viewed as a state machine. Every time a client invokes an operation on a component, its state changes. In addition, while a component performs some function its state will also change. Not all states will be significant to a system integrator. Certain key states will be of interest, for example, at the end of an operation, when something should have completed successfully. It might also be of interest to know of the progress of an operation.

The built-in contract testing architecture makes these states that are essential to operate a component visible to external clients. I call such externally visible states logical states, in contrast with the real internal states of a component, the so-called physical states. Logical states are essential for a client or user of a component in order to use the component correctly and understand its behavior. We may define a logical state as a domain or a set of physical states for which a component's operation invocations will exhibit the same abstract behavior. For example, the physical state of a stack is any arbitrary combination of its internal attributes. This comprises the number of stack entries and the value of each entry. If the value of an entry changes, the physical state of the stack changes. However, for using a stack and understanding its behavior, the value that is stored in a stack entry is completely irrelevant. It is only relevant in terms of whether I retrieve the same value from the stack that I have previously put into it. More important for understanding the behavior of a stack is the fact that I cannot retrieve a value from an empty stack, or I cannot store any more items in a full stack. "Empty and "full" are logical states that are externally visible to the user, because the stack operations `pop` and `push` will behave differently according to these states, that is, a `push` on a full stack and a `pop` on an empty stack will result in some error handling if the stack's contract defines such a handling mechanism.

Built-in contract testing represents a method and a process for making logical states of a component accessible to its external clients, and consequently to their built-in tester components. This part of the built-in contract testing architecture is concerned with component development. In other words, component providers have to add the testing interfaces to their products before they are released, so that component consumers can make use of this testability feature when they integrate such components into their applications. The state setup and state validation methods for a component can be derived from its own specification behavioral model. The behavioral model, i.e., a UML statechart diagram, shows a component's externally visible, feasible states through which it can proceed throughout its lifetime.

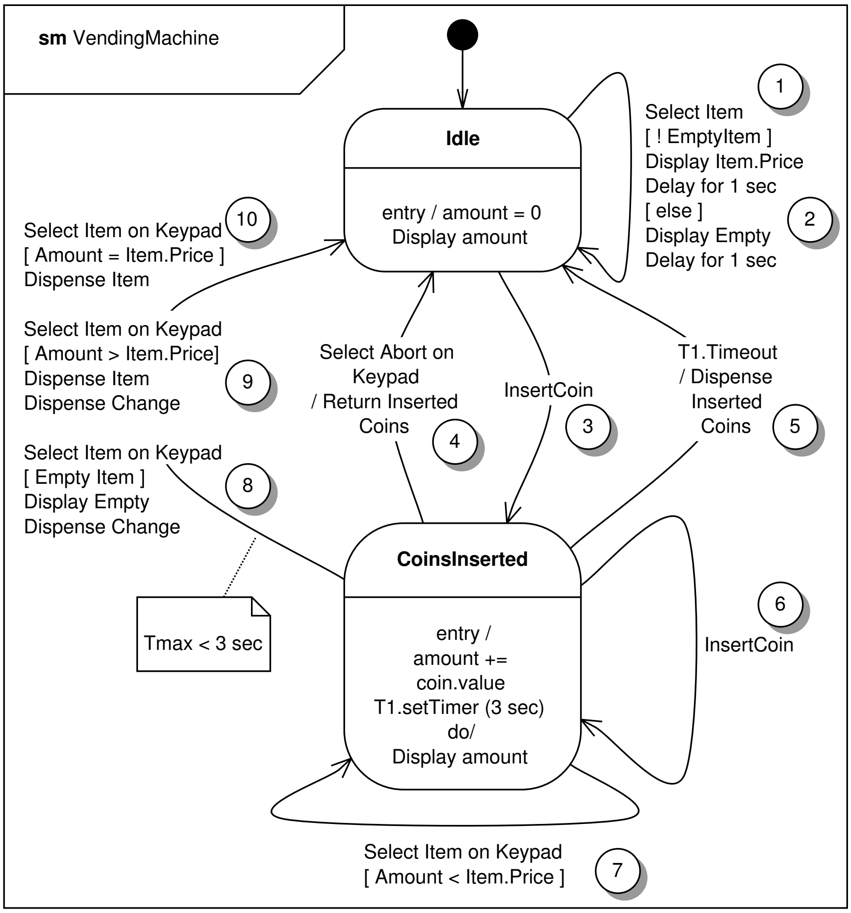


Fig. 4.3. Statechart diagram of the *VendingMachine* component

Figure 4.3 shows the specification behavioral model of the `VendingMachine` component from Chap. 3 again, and Fig. 4.4 shows the respective testing interface variants that we can derive from the behavioral model.

Every state symbol in the statechart diagram represents a starting point for a test execution and an end point. A test case can represent a single state transition between two such points, or a sequence of transitions through a number of states. We have discussed this in Chap. 3, where we have derived a state table from a statechart diagram and used it as the basis for testing. Table 3.6 on page 102 shows the abstract test cases for this example (Chap. 3). The fundamental concept here is that we can always identify an initial state before a test and a final expected state after a test, and both are derived from the specification statechart diagram. The testing interface will provide

- a state setting operation, e.g., `setToStateX():void`, and
- a state checking operation, e.g., `isInStateX():boolean`,

for each state symbol, with `X` representing the name of a particular state. These operations may be used by an external client of the component to bring the component into the desired state before a test is executed, or to check whether the component is residing in the expected state after the execution of a test.

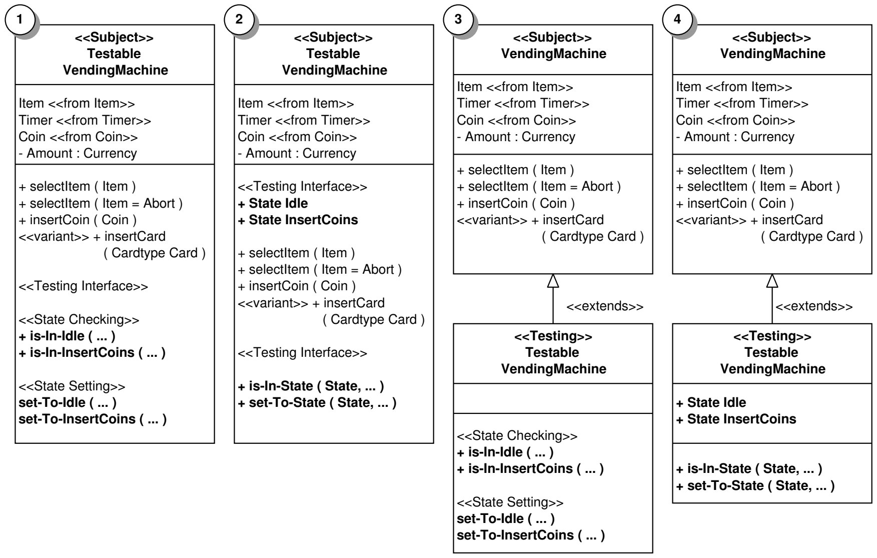


Fig. 4.4. Alternative UML class diagram representations for the testing interface of the *VendingMachine* component

Figure 4.4 displays four alternative implementations for such a state setting and state checking mechanism in the form of a UML class diagram:

1. The first class symbol of the `TestableVendingMachine` component on the left hand side in Fig. 4.4 represents the simplest way of implementing a component with a testing interface. It defines two testing interface operations for each state that are directly implemented in the original class.
2. A more advanced method is to have only two testing interface operations altogether, which take the names of the states as input parameters. The states are defined as public constants that internally switch to their respective implementations.

The internal code for setting and retrieving state information is essentially the same for both strategies. The advantage of the second way of doing this is that we have only to define two testing interface operations that always have the same signature for all components. Only the definitions of the states are different. This second method greatly facilitates and formalizes the code for the testing interface. For example, we could devise a predefined library that we could use as a testing interface development framework. This would provide an enumeration type for the states and a `case` statement-like construct for switching to the respective code sections for setting and retrieving state information. In fact, such a library does already exist, and we will have a closer look at it in Chap. 5.

The problem with the two architectural approaches on the left hand side of Fig. 4.4 for implementing testing interfaces is that the testing code is always permanently built into the component. In other words, after we have integrated such a component into its new environment and checked its contract compliance against its new component infrastructure, the testing code will always stay in the new application, although we will never need that again in this particular context. The two alternative architectural organizations on the right hand side of Fig. 4.4 solve this problem:

3. In the third instance we can have the implementation of the original component, in this case the `VendingMachine`, and extend it with a testing interface that comes as a separate implementation.
4. In addition, according to the second incarnation in Fig. 4.4, we can have only two operations that can be parameterized through state variables.

This way of organizing testing interfaces we can use a testable version of our component for integration and deployment by instantiating the extension, e.g., `TestableVendingMachine`. And, later on if we are confident that the application will work after we have applied some contract tests, we can instantiate and run the original, non-testable version of our application. The testing interface operations that we have discussed in the previous sections will provide the core testing functionality of a testable component. Their signatures will change from component to component according to the component's logical states. We will have either two testing interface operations per logical state or two operations in total, plus a public state variable that controls the states for the second alternative. Additionally, we can have a number of support operations that facilitate the access to the core testing interface. This comprises an operation that determines the type of built-in testing that the component provides, and an operation that can be used to set and invoke a component's own built-in tester components. The following list summarizes the operations of a typical testing interface:

- `IBITQuery`. Query operation that every component should provide as a default. This determines whether the component provides a testing interface according to the built-in contract testing philosophy. `BIT` stands for "Built-In Testing;" the signature `IBITQuery` indicates that the operation belongs to a special built-in testing interface that can be used to query the component on supported built-in test technology. I will introduce an additional built-in testing technology, built-in quality-of-service testing that uses also these conventions (Chap. 7).

Table 4.1. State-based unit testing of the *testableVendingMachine* component with access to its testing interface

No	Set To Initial State	Precondition	Transition	Postcondition	Check Final State
1	setTo (idle)	[Item.Empty]	SelectItem (Item)	Display (Empty)	isIn (idle)
2	setTo (idle)	[!Item.Empty]	SelectItem (Item)	Display (Item.Price)	isIn (idle)
3	setTo (idle)		InsertCoin (Coin)	Display (Amount)	isIn (insert Coins)
4	setTo (insert-Coins, ...)		SelectItem (abort)	CashUnit. dispense ()	isIn (idle)
5	setTo (insert-Coins, ...)		Timeout ()	CashUnit. dispense ()	isIn (idle)
6	setTo (insert-Coins, ...)		InsertCoin (Coin)	Display (Amount)	isIn (insert Coins)
7	setTo (insert-Coins, ...)	[Amount < Item.Price]	SelectItem (Item)	Display (Amount)	isIn (insert Coins)
8	setTo (insert-Coins, ...)	[Item.Empty]	SelectItem (Item)	Display (Empty) & CashUnit. dispense (Change)	isIn (idle)
9	setTo (insert-Coins, ...)	[Amount > Item.Price]	SelectItem (Item)	Dispenser. dispense (Item) & CashUnit. dispense (Change)	isIn (idle)
10	setTo (insert-Coins, ...)	[Amount = Item.Price]	SelectItem (Item)	Dispenser. dispense (Item)	isIn (idle)

- `IBITSetToStateX`. An operation that brings the component into one of the predefined states, according to a public state variable.
- `IBITIsInStateX`. An operation that checks whether the component is residing in a pre-defined state.
- `IBITSetTester`. An operation that assigns a dynamic tester component to a testing component. Dynamic tester components represent a special kind of tester components that make built-in contract testing much more flexible. This is explained later on, in Sect. 4.3.4.
- `IBITInvokeTester`. An operation that invokes the component's tester. It represents an external hook that the component provides through which external entities can start executing the built-in tests.

These proposed interface operations for testable components are fully in line with the initial built-in testing architecture and interface descriptions that have been developed as part of the Component+ project [34, 35]. In the following subsection, I will concentrate only on the two `IBIT` signatures `SetToState`X and `IsInState`X that represent the core testing interface operations for built-in contract testing. The other operations represent desirable features of a testing interface that are of a more general and organizational nature.

4.3.3 Optimal Design of the Testing Interface

The testing interface is not critical to the application of built-in contract testing, but it can simplify testing considerably. We have always tested components without the alternative access mechanism that the testing interface provides. Although, I have to say that the idea of a testing interface is quite natural in software development, and many components do in fact provide something like that. For example, most classes of the standard Java libraries provide many more features than one could possibly hope for, and many of these can be readily used in the same way as testing interface operations. Many of these operations provide simple "setter" and "getter" functionalities which effectively implement something that is quite similar to a testing interface as defined in built-in contract testing. Without a testing interface, testing is limited to the access mechanism that the component provides through its normal functional interface. If we decide that our components should have testing interfaces, we should design and implement them carefully to draw the greatest possible benefits from this additional development investment. The design and implementation of a good and useful testing interface provides some challenges, and it always demands considerable balancing our effort with the tradeoff for the users of our component.

Figure 4.5 shows the three alternatives that we may have with built-in contract testing.

- Sequence 1 represents a client testing the `VendingMachine` component without a supporting testing interface. We use the normal functional interface operations to bring the component into the required state for a test. We call the operation in that we are interested for this test. Hopefully, the test succeeds.

- Sequence 2 represents a fully testable server `VendingMachine` that supports the testing by its clients through a built-in contract testing interface. We can bring the component into the required state through the state setting operation. We call the operation that we would like to test, and we assess the correctness of the resulting state through the state checking operation.
- In sequence 3 the testable component only provides the state checking mechanism. This is a combination of the first and second testing sequence. We use the normal functional interface for setting up the state for a test; we check the state through the testing interface, so that we are sure that the component performed these operations correctly. Then, we execute the operations under test and check the resulting state again through the testing interface and what the tested operations have returned.

In the following paragraphs we discuss how testing interfaces are ideally designed and developed.

Set-To-State **Operations**

In the previous example I have defined state setting and state checking operations that can take a state as input parameter (Table 4.1). A logical state is an abstract representation of concrete parameter or attribute settings inside a component. It is a value domain similar to an input parameter domain for which a procedure follows distinct paths through its internal structure. The only difference is that the input parameters are internally stored attributes, and that the control flow is typically limited not only to one single procedure, but expands to a combination of method invocations that collectively represent the code of a component. Any logical state therefore requires a concrete physical state that belongs to the domain of the logical state. In other words, in order to bring a component into a distinct logical state, we have to tell the component which concrete values it should take for that state. Otherwise we cannot perform any meaningful testing with the component. This becomes apparent when we have to insert coins for testing our vending machine. The logical state `insertCoins` represents all feasible combinations of inserted coins. A test case, however, always embraces concrete values, for example, one 50 cent coin, 20 cent coin, and 10 cent coin for purchasing a muesli bar that will cost 80 Eurocents. We can achieve this particular state with the vending machine through the following invocation history (in Java):

```
insertCoin ( 0.50 );
insertCoin ( 0.20 );
insertCoin ( 0.10 );
```

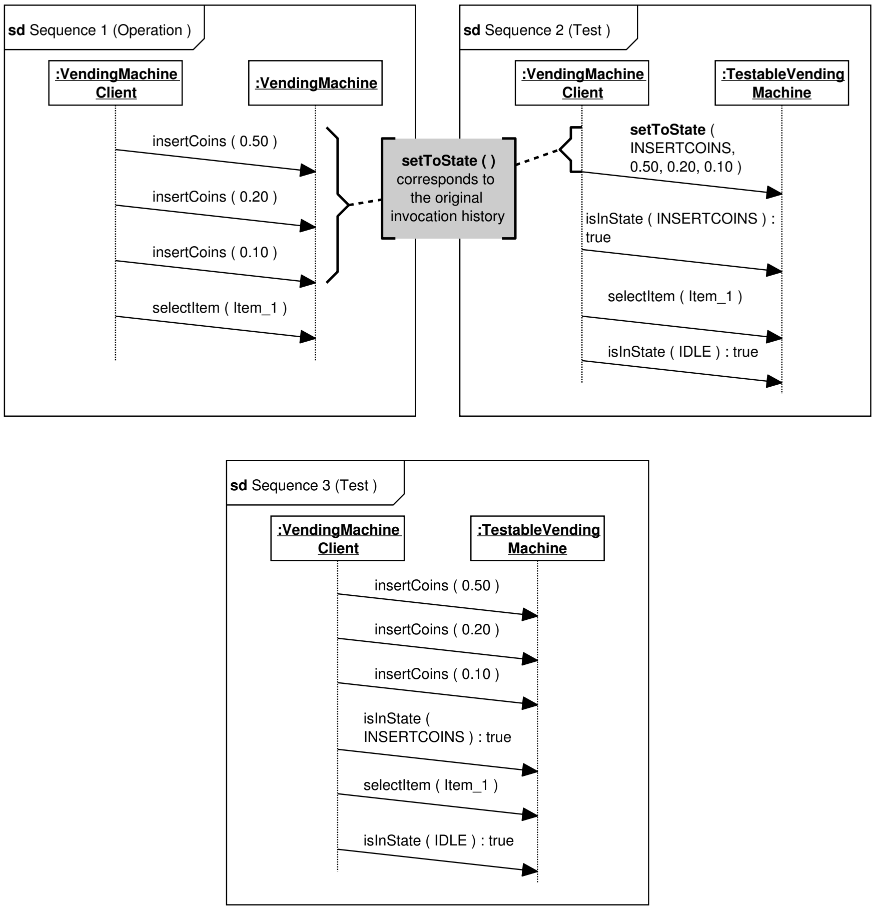


Fig. 4.5. Alternative invocation sequences for a test case according to the provided testing interface

The corresponding testing interface operation may look like this:

```
setTo ( insertCoins, 0.50, 0.20, 0.10 );
```

But this would restrain our state setting operation to exactly three coin values that it can handle, and this is not really what I would expect to get from a testing interface. A more flexible approach would be to pass a list of coin values as input parameter in this particular instance:

```
double [] ListOfValues = {0.50, 0.20, 0.10};
setTo ( insertCoins, ListOfValues );
```

But this makes the implementation of the testing interface operation more complex. The type of the input to the state setting operation will be constant, since we are always concerned with setting the same attributes in a component. But in this particular instance we have to accommodate the fact that we could get an arbitrary number of values. The setting operation will therefore have to provide the ability to process such input types, for example:

```
public class VendingMachine {
  ...
  // testing interface
  // logical states
  public final int IDLE = 0;
  public final int INSERTCOINS = 1;
  // operations
  setTo ( int state, double [] values ) {
    if ( state == IDLE ) {
      for ( int i=0; i<values.length; i++ ) {
        ... // do something with the values
      }
    }
    else if ( state == INSERTCOINS )
      ... // do something else with the values
    else ...
  }
}
```

There may be much better ways to implement the state constants in Java, for example, through the typesafe enumeration pattern [17], but the intention is merely to showcase the principles of a testing interface. I have included these little code examples to illustrate what we have to take into account when we design the state setting operations as part of the testing interface for a component. Sometimes we will find that the testing interface repeats a considerable part of the original functionality. In this case it is arguable whether the state setting mechanism provides any advantage over the normal functional interface. After all, it is heavily dependent upon the component for which we have to devise a testing interface. As I said before, we have to consider the tradeoffs carefully.

The following two issues suggest some criteria for which state setting operations are useful:

- States are difficult to reach through the normal functional interface. If the state model is complex it might be advantageous to have a single state setting operation that condenses a lengthy method invocation history into one single statement.
- A state is represented by a simple combination of internal attributes and the normal functional interface does not permit us to set these attributes directly.

The next issue describes the criterion for which state setting operations might be obsolete:

- A state is defined through a distinct data structure. The functional interface is used to build this data structure and the state setting operations repeat exactly this functionality. This is the case for most data storage components.

In general, we can observe that state checking operations are more important for built-in contract testing than state setting operations. This is laid out in the following paragraphs.

Is-In-State **Operations**

We can bring the component into any of its defined states by invoking the right methods of the normal functional interface in a distinct sequence. I call this the invocation history. Although the functional interface may be faulty, and we may end up in another state than we are actually expecting, we can call the state checking operations to assess that. If the component does not provide state setting operations, we can always use the normal interface to satisfy the preconditions for a test, call the state checking operations to check those preconditions, and finally invoke the test. So, in case we are undecided about whether or not our component should provide a state setting interface, we can always concentrate on an effective state checking interface.

Most of the is-in-state operations will likely be very similar to traditional assertions. Maybe the only difference is that assertions are typically assessed automatically and permanently during the execution of a component, and the state checking operations of built-in contract testing are usually explicitly invoked during a test. There is nothing keeping us from adding constant runtime assessments to a component in the form of traditional assertions, but their explicit invocation is probably more in line with the philosophy of built-in contract testing. If we separate the code for the testing interface from the component's normal functionality, as suggested on the right hand side in Fig. 4.4, we are anyway not going to get constant assertion checking through the testing interface. So, the design of a testing interface should always be driven by the goal of built-in testing; is it for

- permanent checking or
- integration testing?

State checking operations are usually much easier to design and implement than their respective state setting operations. If a state checking operation is supposed to determine whether the component is residing in a distinct state or not, its implementation is straightforward. In this case, the operation must check only the expressions for which the logical state is valid. This is most easily performed through a number of comparisons on the internal attributes that make up the state. For example, the following source code shows the state checking operation for the testing interface of the VendingMachine component (in Java):

```
public class VendingMachine {
  ...
  // testing interface
  // logical states
  public final int IDLE = 0;
  public final int INSERTCOINS = 1;
  // operations
  boolean isIn ( int state ) {
    if ( state == IDLE )
      return ( Amount == 0 );
    }
    else if ( state == INSERTCOINS )
      return ( Amount > 0 );
    else ...
  }
}
```

The only distinguishing feature between the two logical states is the internal attribute amount. In the first case it is zero, because no coins have been inserted, and in the second case it should be greater than zero, because the method insertCoins should have changed that.

Apart from the attributes that represent the state, we can certainly check any defined invariant of the component. Invariants are conditions that must be true at all times during execution. Assertions perform such assessments, and they are typically implemented in a way in which they can permanently check the invariants, or at least at distinct points during execution. In built-in contract testing, these distinct points can be made either explicit or implicit. In the first case we can design additional operations for each of the invariants for which we would like to have an assertion check during testing. In the second case we can attach the assertions to any of the state checking operations, and whenever a state checking operation is invoked through a test case, we can go through all assertions that we have additionally specified for a component. A third option is to group the checking of all internal invariants into one single operation that we may call the assertAll-operation of the testing interface, or if we define it according to the IBIT format: IBITassertAll(). Through this additional testing interface operation, we can separate state-related interface operations from the more traditional assertion checking mechanisms more clearly.

When we embrace several assertion checks with a single operation such as the IBITassertAll, and use boolean return values for the state checking operations, we get one single outcome, i.e., true or false, because we combine a number of different assertions into one answer. Such a combination will certainly impede error identification in the case of a failure, because it is not initially clear which assertion has actually caused it. This is a problem of reduced error observability and traceability with combined assertions. Ideally, we have to include a return mechanism that separates between several feasible causes of a failure, so that the tester knows whether it was an internal state that caused the failure, or whether it was a violation of any of the other invariants. How such a mechanism is realized clearly depends on the used underlying implementation platform and programming language. A convenient way of doing this is through exceptions and exception handling, but not all platforms support that. Another way may be through the traditional C language-style returned integer, where each integer represents a different cause of failure and zero represents success. However, I am not sure whether this would be my favorite approach. In Chap. 5 we will take a closer look at how built-in contract testing can be implemented in typical component technologies and programming languages. The following subsection describes how tester components are ideally developed.

4.3.4 Tester Components

Component integration, and therefore the establishment of a component interaction, usually requires some form of configuration. Configuration will in general involve the creation of all components in an application and, in particular, the creation of the client and server instances in a specific component interaction. This is usually done by an outside "third party," which I call the context of the components according to the KobrA method. This was outlined in Chap. 2. The context creates the instances of the client and the server, and passes the reference of the server to the client, thereby establishing the connection between them. This act of connecting clients and servers is represented by the KobrA style ≪acquires≫ stereotype. The context that establishes this connection may be the container in a contemporary component technology, or it may be the parent object. The two steps that are necessary to configure the pairwise relationships between components in an application are illustrated in Fig. 4.6. In the first step, the context creates the client and then the server, and in the second step it passes the reference of the server to the client. This must be done through invoking the operation in the client's configuration interface, for example, the operation `setServer(Server)`, in which `Server` represents a reference of the acquired server object.

In order to fulfill its obligations toward its clients, a component that acquires a new server must verify the server's semantic compliance to its contract. This means the client must check that the server provides the semantic service that the client has been developed to expect. The client is therefore augmented with built-in test software in the form of a tester component as shown in Fig. 4.7. The server tester component is executed when the client is configured to use the server. To achieve this, the client will pass on the server's reference to its own built-in server tester component, again through some sort of a `setServer(Server)` operation. The establishment of this relation is rep-

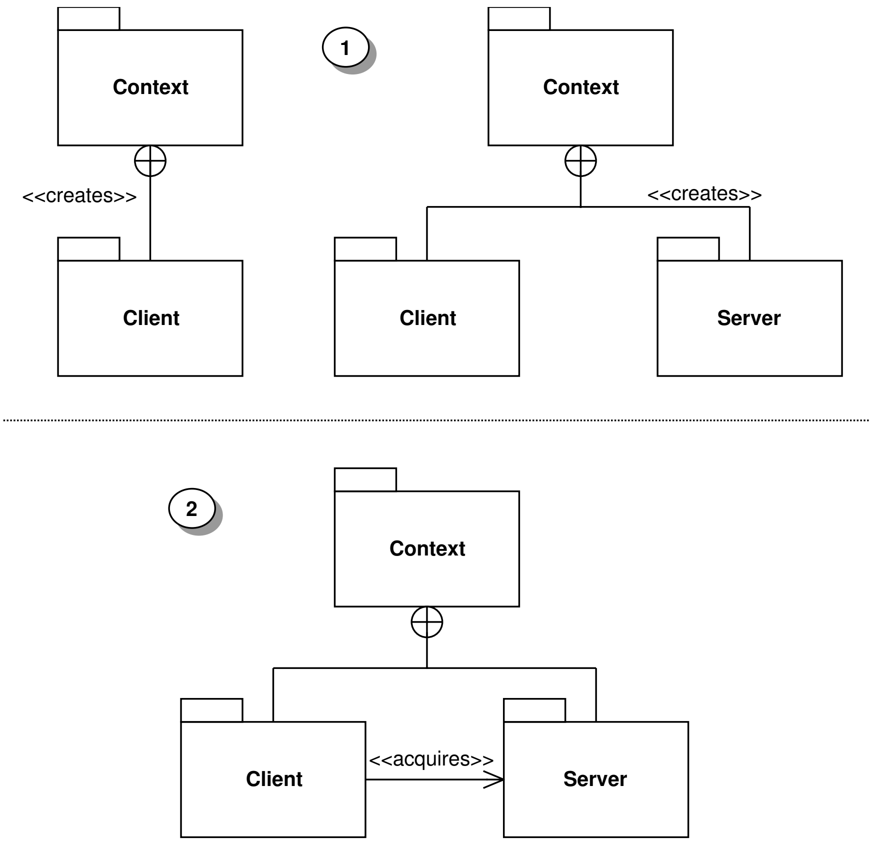


Fig. 4.6. Configuration of a pairwise component relationship

resented by an ≪acquires≫ association between the server tester component and the server in Fig. 4.7. The next step is that the server calls the testing sequence either through a separate method invocation, such as `startTest()`, or implicitly through the `setServer()` operation. If the test fails, the tester component may raise a contract testing exception and point the application programmer to the location of the failure.

In the previous two subsections I have demonstrated how the specification models of the `VendingMachine` component drive the development of its testing interface that it provides to its clients. In the next paragraph and following subsection we have a look at how the `VendingMachine` realization models drive the development of its built-in tester components for checking its associated servers. The two items together, testing interface and tester components, represent the component's complete additional built-in contract testing architecture.

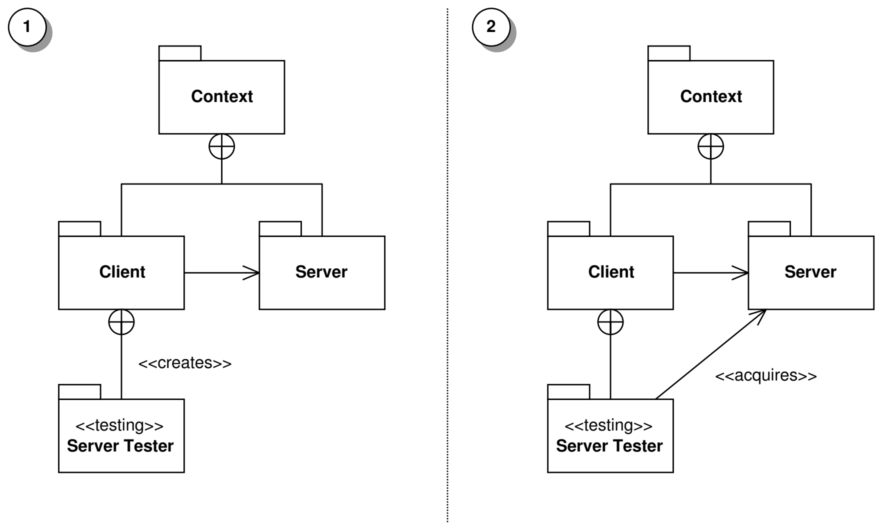


Fig. 4.7. Configuration of the tester component

The component realization fully describes a component's expectation toward its associated servers. This was outlined in Chap. 2. It comprises structure, the server components that the subject expects, and behavior. This is the behavior that the subject expects from its servers. The client's realization behavioral model defines a statechart diagram for each server component that the client acquires. In Chap. 2 I merely defined the structure of the vending machine's subcomponents, but no specific behavior. One of these components in the realization structural model is the `CashUnit`. Figure 4.8 displays the containment hierarchy of the `CashUnit` component. The context of the `CashUnit` is the `VendingMachine`, and its subcomponents are a `CoinChecker` that validates the incoming coins, a `TemporaryCoinStore` that holds the incoming coins as long as a user performs a `PurchaseItem` transaction, a `CoinDispenser` that pays back the change, and a `CoinSafe` that finally accepts and stores the coins. Figure 4.9 shows the realization behavioral model of the `VendingMachine` component for the `CashUnit`. This model represents the behavioral expectation of the vending machine toward a subordinate cash unit component. This behavior is what the vending machine requires from this part of its environment, and this is the basis on which the tests of the vending machine's built-in tester component for checking a subordinate cash unit implementation must be founded.

4.3.5 Optimal Design of a Tester Component

Now that I have explained how a tester component is defined in principle on the basis of a client's realization models, we can have a look at how tester

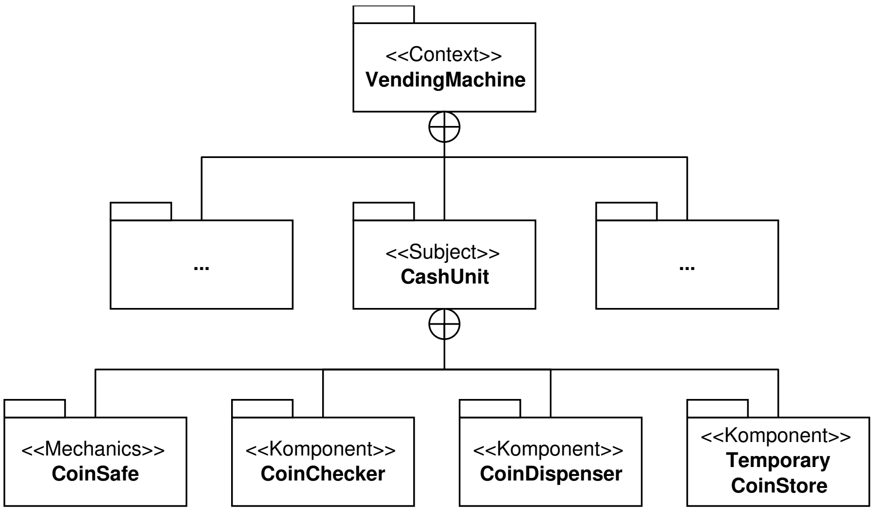


Fig. 4.8. Containment hierarchy of the *CashUnit* component

components are organized internally. In the previous sections I referred to "tests" or "test software" in an abstract sense, without being specific about the language construct used to represent them. A test is a normal piece of functional software, and in an object- or component-oriented context, this must be represented by one or more methods or method calls. The functional decomposition of a test into methods, i.e., the break down of the overall test method into sub-methods, is not of particular significance. However, the assignment of these methods to components is important. For the purpose of this section I will assume that there is a single method, known as the test method, which is responsible for executing a test.

Since built-in tests are logically performed by the clients of the components which they test, it might at first seem natural that the test method should belong there as well. In other words, if C is a component that contains a built in test of one of its servers S, the test method that executes this test would be regarded as a method of C. However, although this approach would work, and at first might seem intuitively appealing, it does not make optimal use of the reuse advantages offered by component and object technologies. Rather than associate a logical test with a single method, it is more in the spirit of component technology to encapsulate a test as a full component in its own right. This has two main advantages:

- First, it allows the logical tests to be written in tandem with the software they test by the original vendor of the functional components. Thus, component vendors would supply their components with accompanying tester components that have been written to make optimal use of a component's state-setting and checking methods.

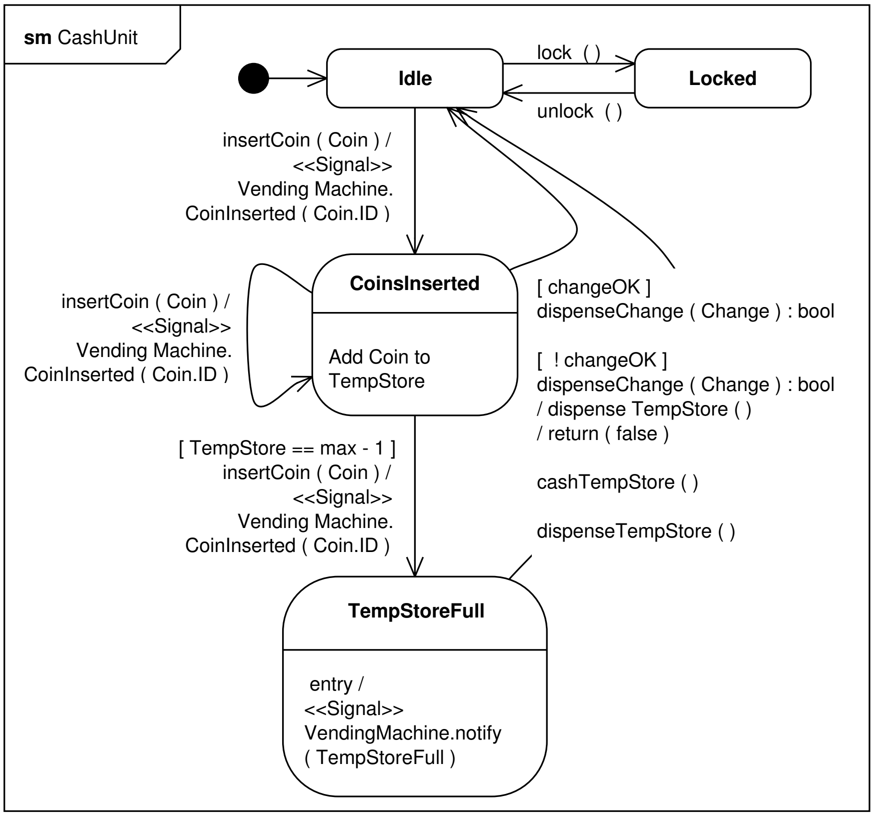

sm CashUnit
Idle
Locked
lock ( )
unlock ( )
insertCoin (Coin) / <<Signal>> Vending Machine. CoinInserted (Coin.ID)
CoinsInserted
Add Coin to TempStore
insertCoin (Coin) / <<Signal>> Vending Machine. CoinInserted (Coin.ID)
[changeOK] dispenseChange (Change) : bool
[! changeOK] dispenseChange (Change) : bool / dispense TempStore ( ) / return (false)
cashTempStore ( )
dispenseTempStore ( )
[TempStore == max - 1] insertCoin (Coin) / <<Signal>> Vending Machine. CoinInserted (Coin.ID)
TempStoreFull
entry / <<Signal>> VendingMachine.notify (TempStoreFull)

Fig. 4.9. Expected behavior of the *CashUnit* component in the realization behavioral model of the *VendingMachine* component

- Second, it provides the optimal level of flexibility with respect to test weight at both the runtime and development time. We can achieve development-time selection of the test weight by choosing which tester component to include in a given system, while runtime test weight selection can be performed by choosing which component instance is to perform a test at a given point in time.

With the concept of contract tester components, the test method is no longer associated with the client component but rather with the tester components which the client contains. The client may still provide a method which can be used to explicitly invoke the test method, but this need not always be the case. Test methods can also be implicitly invoked when servers are initially set, for example, within the constructor of a class in Java. Since heavy tests are usually extensions of lighter tests, i.e., they include all the test cases of the lighter tests but add more. Heavy tester components will usually contain lighter tester components. In other words, the test method for a heavyweight tester component will include a call to the next lightest tester component as well as the additional test cases that make it heavier. This is illustrated in Fig. 4.10. By hierarchically organizing test methods and tester components in this way, the replication of test cases at runtime can be avoided.

The tester component for state transition testing contains at least the minimal test set for state transition coverage identified in the state transition table. I have described that in Chap. 3. Each state transition in the state transition table maps to an individual test with state setup and provision of the required preconditions, an event, and the state verification with checking the expected postconditions. Here, the state transition table represents a minimal tester for state coverage. Performing any more testing and adding any more hierarchically organized tester components is a decision which must always be made according to some of the following criteria:

- Time of the test: How often is the test performed? When, during the product life cycle, is the test performed?
  - development time
  - specification-based test
  - regression test
  - runtime
  - deployment time
  - reconfiguration time
- Origin of the component: How much do we trust the component?
  - in-group
  - in-house
  - commercial off-the-shelf
- Mission criticality of the component or application: Is the system safety or mission critical?
  - extremely critical
  - critical
  - not critical
- Availability of resources: Do we have sufficient time and space to perform a test?
  - deeply embedded system
  - normally embedded system
  - real-time system
  - distributed system
  - information system

According to these criteria, differently sized tester components are feasible. For example, a thorough tester would be appropriate if the origin of the component is not known, while a very small lightweight tester would be best if the component is trustworthy and the time or the resources for executing a full test are sparse. Figure 4.11 summarizes the criteria for these decisions. In the next subsections we look at how the primary built-in contract testing artifacts, testing interface and tester component, are related to or associated with one another.

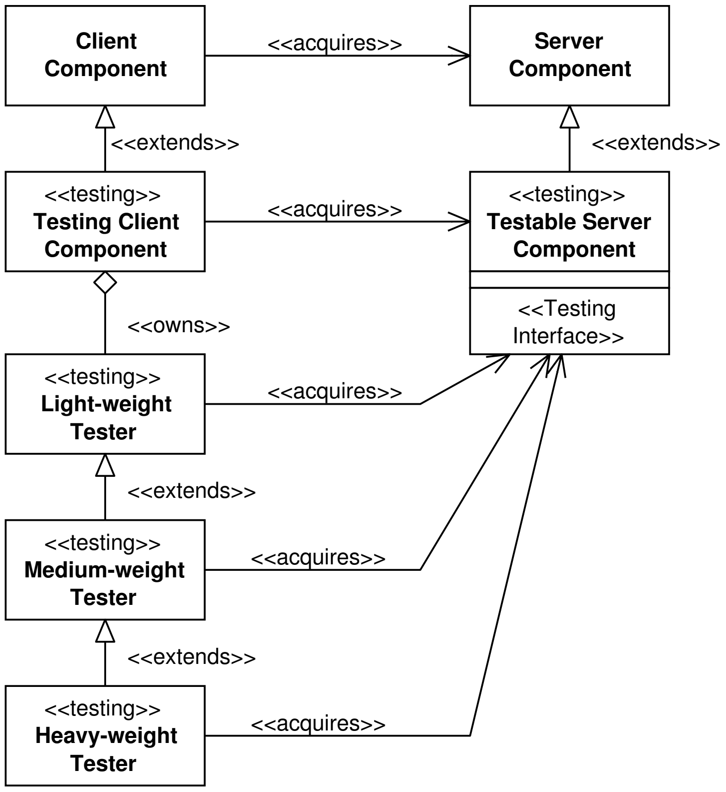

Client Component
<<acquires>>
Server Component
<<extends>>
<<extends>>
<<testing>> Testing Client Component
<<acquires>>
<<testing>> Testable Server Component
<<Testing Interface>>
<<owns>>
<<testing>> Light-weight Tester
<<acquires>>
<<extends>>
<<testing>> Medium-weight Tester
<<acquires>>
<<extends>>
<<testing>> Heavy-weight Tester
<<acquires>>

Fig. 4.10. Tester volumes and tester variations through extension

4.3.6 Component Associations in Built-in Contract Testing

Associations between Components in Built-in Contract Testing

The previous sections have introduced the two primary artifacts that must be developed for each component if it will provide built-in contract testing capabilities:

- the tester component, with the test cases for checking a component's server, and
- the testing interface that provides state setting and checking operations, among others, to enhance a component's testability.

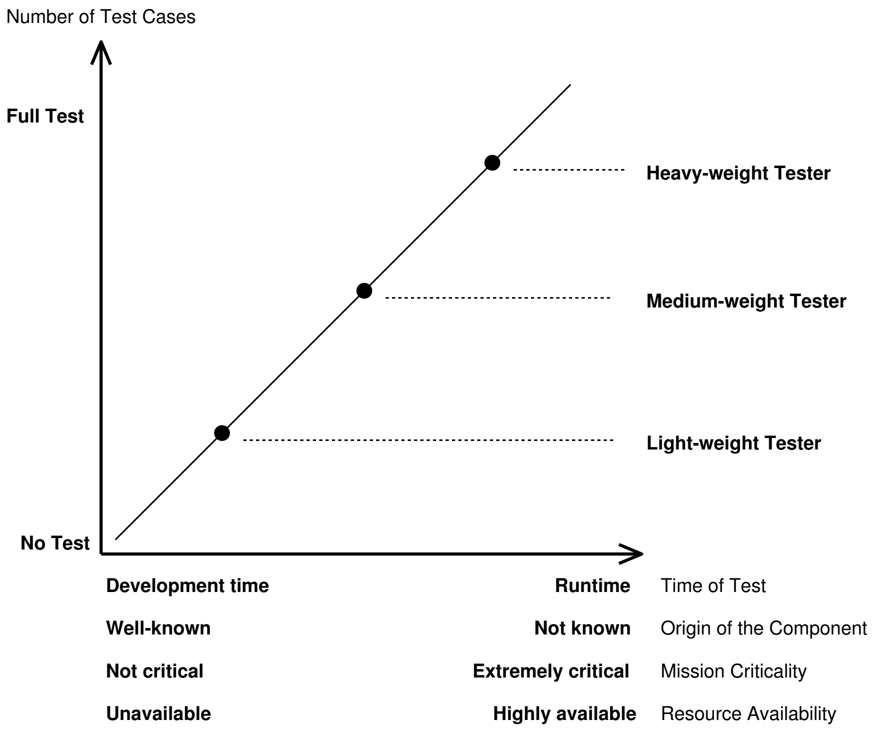


Fig. 4.11. Criteria for the size of the used test set

These represent additional functionality that the component developers have to add, aimed particularly at testing. The first one extends the client component, and it comprises the actual test cases that check the client's deployment environment, in other words, its associated server components. The second extends the interface of the server to enhance the server's observability and controllability, and make it more testable. If a server does not provide a testing interface, e.g., a COTS component that does not apply the principles of built-in contract testing, it does not mean that contract testing may not be used. Built-in contract testing is merely limited with respect to controllability and observability during a test, and the test cases in the client must be designed differently according to the missing testing interface. This is because the test suites that we have to develop if no testing interface is available tend to be much larger than if we have such an interface. The tests have to be designed in a way that they always end up in an externally observable state. Otherwise we cannot check anything. Observability is a prerequisite for testability. This is a well known problem with the testing of object-oriented systems. A test can only assess a result that becomes visible outside the object's encapsulation boundary. If an invoked object never returns any values back to the caller, the caller will never get any feedback on what has actually happened. Running tests on such an object is entirely meaningless. However, the tester component may be considered the more important part of built-in contract testing.

Associations between Client Component and Tester Component

In the client role, a component may own and contain a server tester component. This means the test cases, typically organized as components in their own right, are permanently encapsulated and built into the client. This is the simplest form of built-in contract testing, and it provides no direct runtime configurability with respect to the type and amount of testing the client component will perform when it is connected to its server components. This association can be expressed through the UML composition association. A more flexible way of built-in contract testing is realized through a loosely associated tester component that may be acquired by the testing client in the same way it acquires any other external source. Here, the component provides a configuration interface through which any arbitrary tester component that represents the client's view on a tested server may be set. This provides flexibility in terms of how much testing will be performed at deployment, and, additionally, in a product line development project it provides flexibility as to which type of tester will be applied according to the product line instantiated. A more loosely coupled association may be represented by a UML aggregation association, or, more specifically, through the KobrA stereotype ≪acquires≫ which indicates that the tester component is an externally acquired server. Figure 4.12 shows two alternative ways of attaching a server tester component to a testing client. The left hand side diagram shows a server tester component `B tester` that is directly built into `Testing Component A`. This means that the client of B contains and owns its own B tester. The second diagram on the right hand side of the figure shows a `B tester` component that is created and owned by the context, which creates and contains the other components also. `Testing Component A` and `B tester` are initially not associated or connected in any way. After creation of all components, the Client `Testing Component A` can acquire its `B tester` in the same way as it acquires its server component `Testable Component B`. However, the testing client has to provide a configuration interface that the context may access to pass the reference of `B tester` to client A. The following Java source code example illustrates how such a configuration facility may be implemented for the components in Fig. 4.12. The code represents a Java embodiment of the right hand side diagram in Fig. 4.12.

```
class TestingComponentA {

  Object serverB;
  Object Btester;

  // configuration interface for the acquisition of server B
  void setServer (Object o) {
    serverB = o;
  }
  // configuration interface for the acquisition of B tester
  void setTester (Object o) {
    Btester = o;
  }

}
```

The context is responsible for the creation of the three objects and their connection. This is illustrated in the following Java source code example:

```
class Context {

  TestingComponentA  TCA; // address of the client
  TestableComponentB TCB; // address of the server
  BTester            BT;  // address of the tester

  Context (void) { // constructor
    TCA = new TestingComponentA ();  // create client
    TCB = new TestableComponentB (); // create server
    BT  = new BTester ();            // create tester

    TCA.setServer ( TCB ); // client acquires server
    TCA.setTester ( BT  ); // client acquires tester
    BT.setServer  ( TCB ); // tester acquires server
  }

}
```

Testing Interface for Nested Components

In a server role, a component must be much more closely connected to its testing interface because the testing interface must be able to access the server's internal implementation (i.e., for setting and getting attribute variables). The testing interface is therefore directly built into the component and extends its normal functionality with some additional functionality that is intended for testing purposes. Another approach is to augment the functionality of the

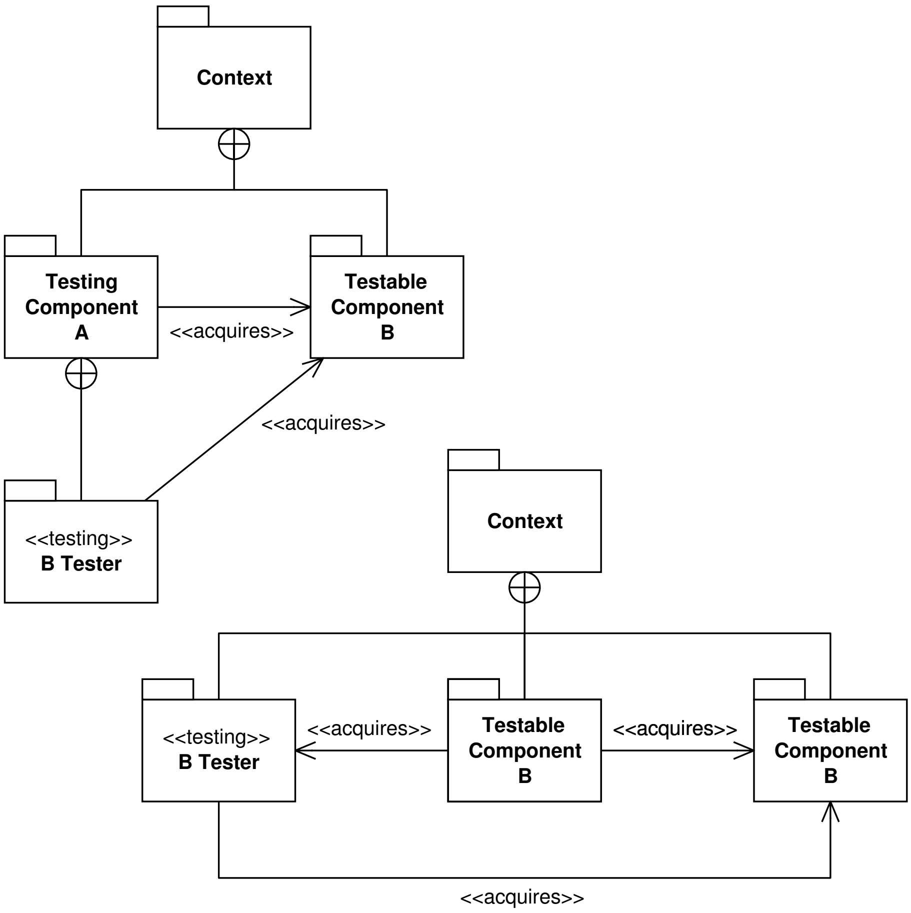


Fig. 4.12. Alternative ways of integrating a tester component

server with an additional testing interface by using a typical extension (inheritance) mechanism. In the KobrA method this is indicated through the UML extension symbol plus the ≪extends≫ stereotype. In any case, the testing interface of a component must be visible at its external boundary. For components with nested objects it means that each of these objects must be dealt with individually inside the component in a way that externally visible behavior that is implemented through these subordinate parts will be visible at the component boundary. This is the responsibility of the component developer.

Associations between Tester Component and Testing Interface

The tester component of the client and the server's testing interface interoperate in the same way as their respective functional counterparts. Since the testers and testing infrastructure are built into the system they are only additional functionality that happens to be executed when components are interconnected. The tester component must "know" only the reference of the tested server, and this is passed into the tester component when the test is invoked by the client. Testing in this context is executing some additional code that uses some additional interface operations. Therefore, built-in contract testing is initially only a distinct way of implementing functionality that is executed when components are interconnected during deployment. This concerns the architecture of a system, i.e., which components will expose additional interfaces, which components will comprise tester components. The actual test cases inside a tester component that are applied during deployment are arbitrary, and they can be developed according to any of the testing criteria introduced in Chap. 3.

Extended Component Meta-Model

Figure 4.13 shows how the two primary built-in contract testing artifacts fit into and extend the original component meta-model I introduced in Chap. 1. The testing interface is an extension of the original functional interface that is partially defined through the component's behavior (its states). The tester extends the component's required functional interface and adds the operations of the testing interface required of the associated subcomponent. The two shaded boxes indicate these two additional concepts. From the figure it becomes apparent that the impact of the built-in contract testing concepts on the component meta-model is only marginal.

4.4 Development Process for Built-in Contract Testing

The previous sections have set the foundations for developing and using built-in contract testing. For the development, this is mainly from the perspective of the component developer or provider. The subjects of these previous chapters and sections can be regarded as a required entry criterion, as a prerequisite, for introducing, implementing, and using the technology, which comprises:

- A sound development method such as the KobrA method that defines the
  - required quality criteria in the form of a quality plan, and thus a collection of test selection strategies,
  - the specification and realization specification of each component, ideally in the form of models, and operation specifications.
- A well-defined architecture with clear identification of the components and their associations in terms of
  - static component relations that will receive removable built-in contract testing artifacts, and
  - dynamic component relations that will receive permanently built-in contract testing artifacts.

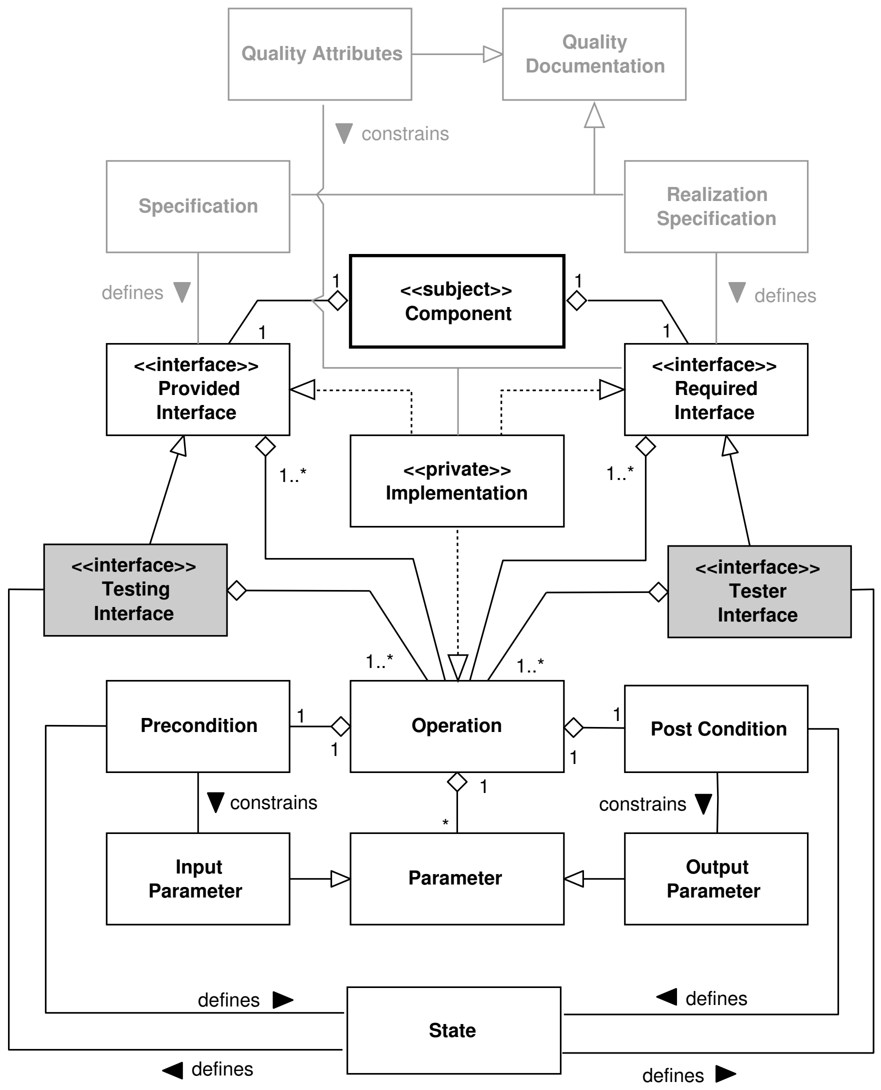


Fig. 4.13. Extended component meta-model with additional built-in contract testing concepts

We can apply the principles of built-in contract testing in projects that are lacking all these prerequisites, but they alleviate the design and implementation of built-in contract testing considerably, especially with respect to the degree of automatism that we can introduce within such a methodological framework. Without sound methodological support it is much more difficult to come up with a sound quality assurance plan, though this is also true for any development activity in any project.

The next sections provide a step-by-step guide for developing testing interfaces and tester components at a high level of abstraction; lower level abstractions and implementation technologies will be specifically dealt with in Chap. 5. Typical steps for the development of built-in contract testing that an application or component development team may follow are:

1. Identification of the tested interactions in the model of the overall application.
2. Definition and modeling of the testing architecture.
3. Specification and realization of the testing interfaces for the server role in the identified associations.
4. Specification and realization of the tester components for the client role in the identified associations.
5. Integration of the components (actually part of the component user's task).

Steps 1, 2, and 5 are not required if we merely consider pure component engineering without having to worry about an integrating application. If we are developing reusable components, all we need to worry about is the built-in testing interfaces that our components will provide and their built-in tester components. However, the first two steps are important if we plan to incorporate built-in contract testing within the realizations of our components as development-time testing infrastructure. Every component may be seen as a system or application in its own right, and its internal realization is performed by following exactly the same principles as for an entire component-based application. Essentially, there is no difference between the development of an application which is mainly an activity that comprises component reuse and integration, and the development of a "pure" component that may be focused on custom development, although reuse and integration are issues. The only fundamental difference between the component level and the application level is that we will quite likely remove the testing artifacts within a component, while we will probably leave the testing artifacts between the components in place for deployment. Component engineers and application engineers are in fact facing the same problems and can apply the same methods for solving them. So, steps 1, 2, and 5 are always part of the process, both for application engineering as well as for component engineering.

For a more extensive illustration of the steps introduced, I will refer to another example that will be used throughout this chapter. The example system is a Resource Information Network (RIN) from Fraunhofer IGD in

Darmstadt, Germany, that is distributed under General Public License (GPL) [92, 147]. It is a part of a larger communication system that supports working floor maintenance staff in their everyday tasks. The larger communication system, or the context of the RIN, hosts multiple communication devices that are interconnected through a radio network, and controlled and supported by a number of desktop working places. This organization is illustrated in the containment tree in Fig. 4.14.

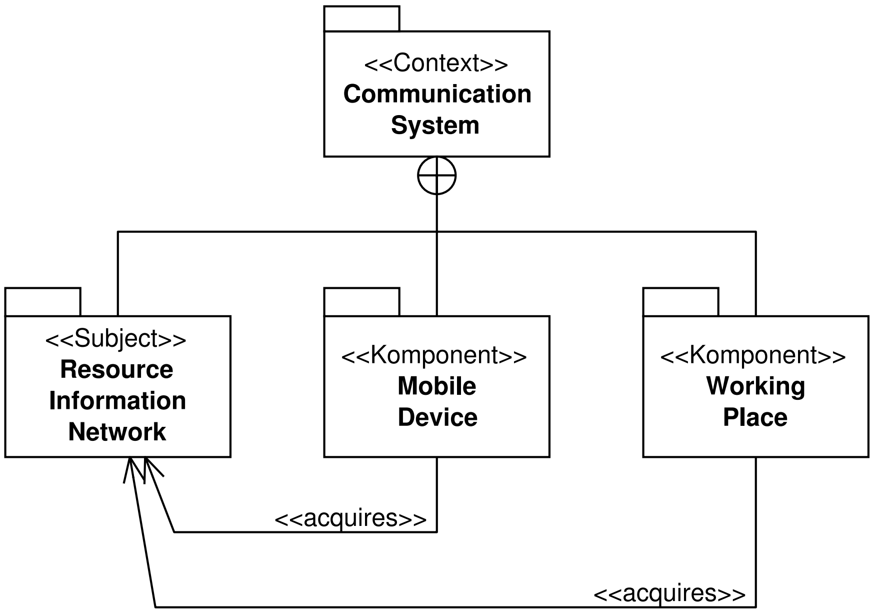


Fig. 4.14. Containment hierarchy for the RIN system within its context

The desktop working places help the maintenance staff achieve its tasks, and provide additional information. They can guide a worker through complex tasks by looking at the video signals from the worker's video facility, give advice to the worker through the audio device, and provide additional assistance, for example, the download of user manuals or video-based repair guides. Each of the communication devices has capabilities defined that are made public to all the other devices through the RIN. Every device that is part of the network will have a `RinClient`, a `RinServer`, and a number of `RinSystemPlugins` installed. The server controls the resource plug-ins and communicates with the client of another connected device. The client gets the information from an associated device's `RinServer`. All the devices within the range of the communication system can, before they communicate, retrieve information from their associated nodes through the RIN about which things they are capable of doing at a given moment in time. This way, the individual nodes are never overloaded with data that they cannot process as expected. For example, the video application of a desktop station may determine the current memory state of a handheld device and decide according to that information whether it can send colored frames or frames only in black and white; it may also reduce the frame rate or request the handheld device to remove some of its unused applications from its memory. These decisions depend on the profile of the user and the priority of the applications that use the RIN.

The RIN suite comprises a client component, a server component, and one or more system plug-in components. The client and server provide the basic communication between the nodes in a network. Every network node that will require resource information from other nodes will have a client component. Every network node that will provide resource information to other nodes will have a server component and some plug-in components, each for a distinct resource. The system plug-in provides the actual business logic capability, for example, the provision of memory and storage resource information. Requests from a client node are marshalled through the `RinClient` component, sent through the network and unmarshalled by the server component at the addressed node. The server communicates this request to the plug-in that generates the requested information and sends it back to the client the same way it arrived.

In Fig. 4.14, I described the RIN system as a part that resides within the context of a larger communication system. But the RIN can also be regarded as a system in its own right that can be used as a communication vehicle, or it can be regarded as a generic component that can be used by other applications that act as clients of the RIN to transfer their internal information to other such clients. This would move the RIN system into another context. We can even perceive that we deploy the RIN system in the context of the vending machine. After all, it is only a component that may be deployed in a number of different contexts. Under the vending machine scenario the RIN would be an ideal communication tool to find out about a particular vending machine's current resources, even remotely, given the fact that the machine is connected with some network. Such resources would comprise, for example, the number of items left in the dispenser unit, or the amount of cash left in the cash unit, and the like. The RIN system would have only to be provided with the right system plug-in component at the physical side of the vending machine. However, in the following subsections we will have a look at the RIN system in isolation. In other words, we will treat it as a full system. Its high level of abstraction containment hierarchy is displayed in Fig. 4.15, and a lower level of abstraction class diagram is displayed in Fig. 4.16. The following subsections illustrate the built-in contract testing process on the basis of the RIN system as an example. Each individual step in the development process is represented by a separate subsection.

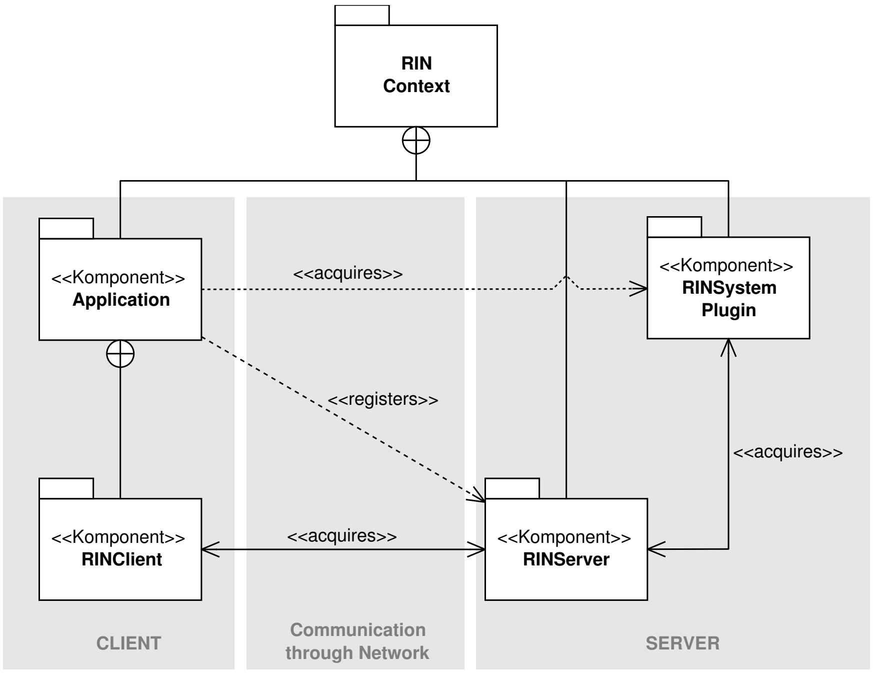


Fig. 4.15. High-level containment hierarchy of the RIN system

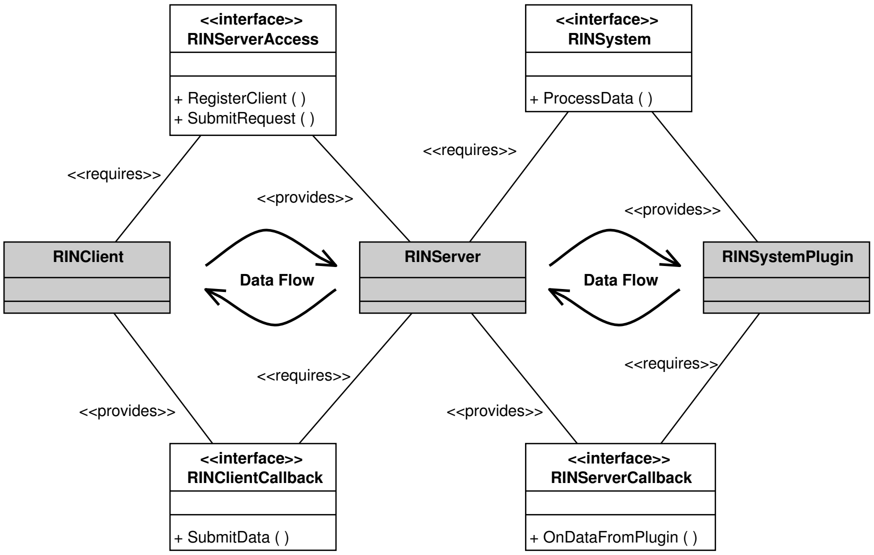


Fig. 4.16. Lower-level, implementation-specific class diagram-type representation of the RIN system

4.4.1 Identification of Tested Interactions

In theory, any arbitrary client/server relationship in component and application engineering may be checked through built-in test software. This is the case for both development of individual components as well as assembly of components into a configuration or final application. Built-in contract testing is in this respect a multipurpose testing technology that may be applied during all phases of the decomposition and embodiment dimensions displayed in Fig. 5.1 (Chap. 5). This is because the typical object and component principle of the client/server relationship is applied at all levels during development, and built-in contract testing is inherently founded on that. It is even valid under the non-object paradigm. The fundamental question is not about which parts of the application we are going to test with it, because we can apply it at all levels of composition and abstraction, and for all client/server interactions; but the most fundamental question is where it does make the most sense to have it built in permanently? In other words, under what circumstances does built-in testing provide the greatest return on investment with respect to software reuse in component-based application development? Therefore, as a development team, we have to answer the following questions:

- Where do we build in testing and have it removed after integration?
- Where do we build in testing and leave it permanently?

In general, any client/server interaction at the component level as well as at the application level may be augmented with a built-in testing interface and a built-in tester component. These interactions are represented by any arbitrary association in a structural diagram, for example, a UML component, class, and object diagram, as well as a KobrA-style composition, containment, nesting, and creation tree diagram. Essentially, every nesting association represents a client/server relationship. This is the case at least for UML because it provides no representation for creation associations and usage associations, in contrast with the KobrA method, that provides these, e.g., an instance that is created by a component but not used as a server by the component.

Associations between classes or objects that are encapsulated in a component are likely to stay fixed throughout a component's life cycle. Such associations may be augmented with removable built-in contract testing artifacts, because they can be tested once and for all when the individual parts of the component are integrated. After release, the component will be used as it is, and the execution of its own in-built contract tester components will not reveal any new component failures. Such an internal built-in contract testing infrastructure may be implemented through typical development- or compile-time configuration mechanisms, e.g., `#include` in `C++`, or through a runtime configuration interface that dynamically allocates tester components and testable components with testing interfaces.

Typically, reusable components will have permanent built-in testing interactions at their boundaries. This means that every external association that requires or imports an interface will be permanently augmented with a built-in contract tester component, and every external association that provides or exports an interface will be permanently augmented with a built-in testing interface. These external associations will change as soon as the component is reused in the different context of another application. So, the built-in tester components can be executed to assess the suitability of the new context for the component.

The diagram in Fig. 4.15 displays a distributed system that comprises a local part (`ApplicationContext`) and a remote system and plug-in part (`ServerContext`). The application context uses the underlying client/server interaction to communicate with the plug-in. The application uses the services from the plug-in. This means that the application "knows" the identity of the plug-in but the plug-in does not "know" the identity of the application. The application will be augmented with a plug-in tester component, and the plug-in will be augmented with a testing interface to support that tester. The application cannot directly see the plug-in, although it "knows" it is there, because all messages from the application to the plug-in and back are directed through the `RinClient` and the `RinServer`. In fact, we consider here the two development dimensions, decomposition and abstraction, at the same time. At an abstract level, the application acquires the plug-in indicated through the dashed arrow. At a lower level of abstraction, this access is realized through the client and server components that together implement the connection over the network.

Client and server as well as server and plug-in are associated through bidirectional ≪acquires≫ relations, meaning that each component plays both roles in a two-way contract. Each of these ≪acquires≫ relations is a feasible candidate for built-in contract testing, a tester component at the client role and a testing interface at the server role of the relationship. For the bidirectional associations it means that the client invokes services on the server component (e.g., `ProcessRequest)` and the server invokes services on the client component (e.g., `ReceiveDataFromServer`). The same is true for the association between the server and the plug-in.

4.4.2 Definition and Modeling of the Testing Architecture

The locations in the application where built-in contract testing makes the most sense can be identified through the ≪acquires≫ relationships between the units as said before. The stereotype ≪acquires≫ represents dynamic associations that may be configured at runtime according to the needs of the application, i.e., components that may be replaced. These are parts of the overall system that are likely to change over time, and the associations are therefore augmented with built-in contract testers on the client side and built-in contract testing interfaces on the server side.

Another indicator of a client/server relationship is the UML anchor symbol that stands for component nesting or containment, and it represents a very specific form of client/server relationship. A contained component is almost always the server of the containing component; this is the case at least for construction. The minimal coupling between the components in such a relationship is represented by the client calling the constructor method of the contained server component.

A third association type that deals with a client/server relationship is represented by the invisibly available runtime support system that is always there for a component to use because the component is executing within the context of that underlying platform. For the RIN system we can identify all the previously listed associations. They are summarized in the following and illustrated in Fig. 4.17:

- Implicit client/server relationship of the components with their respective runtime support systems. The runtime system is provided by the platform in which the component is going to be executed (e.g., operating system, virtual machine, component technology, etc.). This may change from configuration to configuration. Thus, for every new configuration we can have executed the component's built-in contract testers that are specifically designed to check the underlying runtime support system. Typically, the runtime system is part of the lower-level models and becomes only apparent in the embodiment phase. I have added the runtime support systems as components in the containment tree in Fig. 4.17 for illustration. The RIN system was initially developed under C++ for Microsoft's DCOM platform, and later transferred to CORBA Components (see Chap. 5).
- Explicit client/server relationship between the components and their creators indicated through the anchor symbol. These are the containment relations between `Application` and `RinClient`, and between `RinContext` and its two subcomponents `RinServer` and `RinSystemPlugin`. These relationships are likely to stay fixed in an application. Constructor calls are usually determined at compile time. Only if a component is integrated in a new application (i.e., when we bring it into a new context) may we invoke the built-in component tester of its superordinate component to assess whether the subordinate component is behaving correctly within its new hierarchy.
- Explicit client/server relationship between the application-level components are indicated through a simple ≪acquires≫ relation between the component symbols. In the RIN system these are bidirectional associations between `RinClient` and `RinServer`, and `RinServer` and `RinSystemPlugin`. For each role that a component is playing, client or server, we can define a tester component and a testing interface.

The decisions about where in the model we add built-in contract testing artifacts must somehow be documented in the structure of the system, i.e., in the model as well. This may be regarded as a simple additional software construction effort in the overall development process that amends the original functionality of the application or of the individual components within the

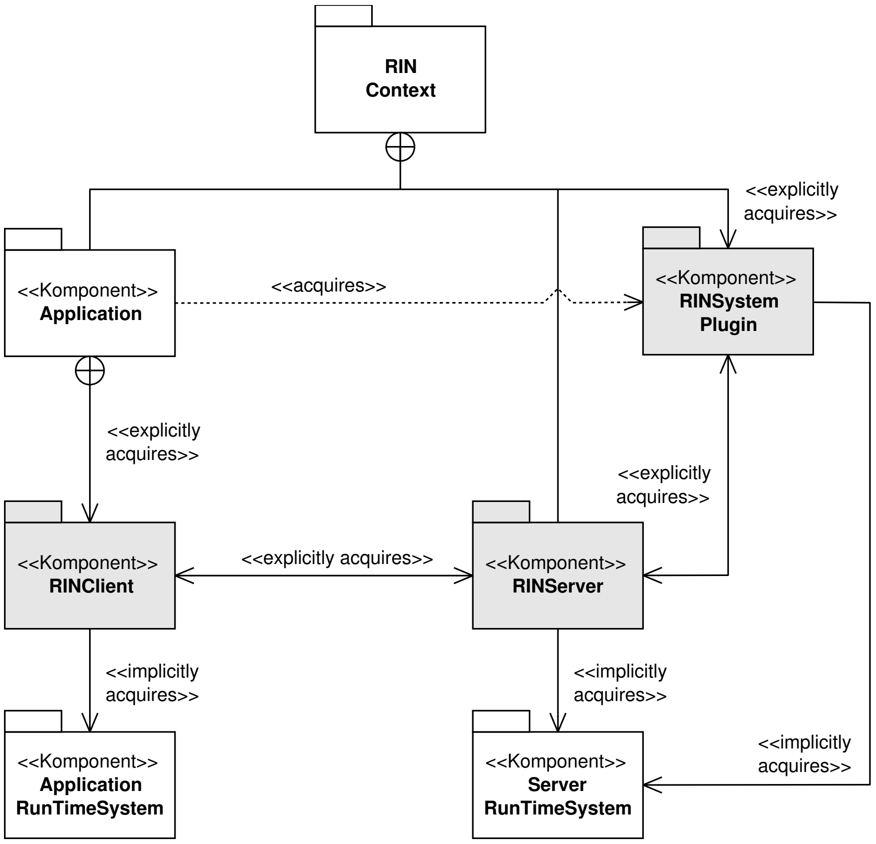


Fig. 4.17. Implicit and explicit client/server-relationships in the RIN system. The dashed line represents a logical association that, as such, will not be implemented

application. Figure 4.18 shows the new containment hierarchy of the RIN system with all the additional built-in contract testing artifacts in the shaded box. The `RinSystemPluginTester` on the left hand side of the diagram is part of the `Application` (through creation and containment), and it provides test cases which represent the `Application`'s usage of the `RinSystemPlugin`. All additional testing artifacts are indicated through the stereotype ≪testing≫. It can be seen as a custom extension to the original UML testing profile that is used to visually separate the original functionality from the built-in testing functionality. `Application` will contain a specific `RinSystemPluginTester` for each individual system plug-in that it will acquire. This tester represents an abstract view of the entire RIN system and checks the functionality accordingly. That is, it checks the responses of the server side RIN system plug-in, and implicitly, the underlying communication mechanisms that `RinClient` and `RinServer` realize. The additional testing interfaces that the three core

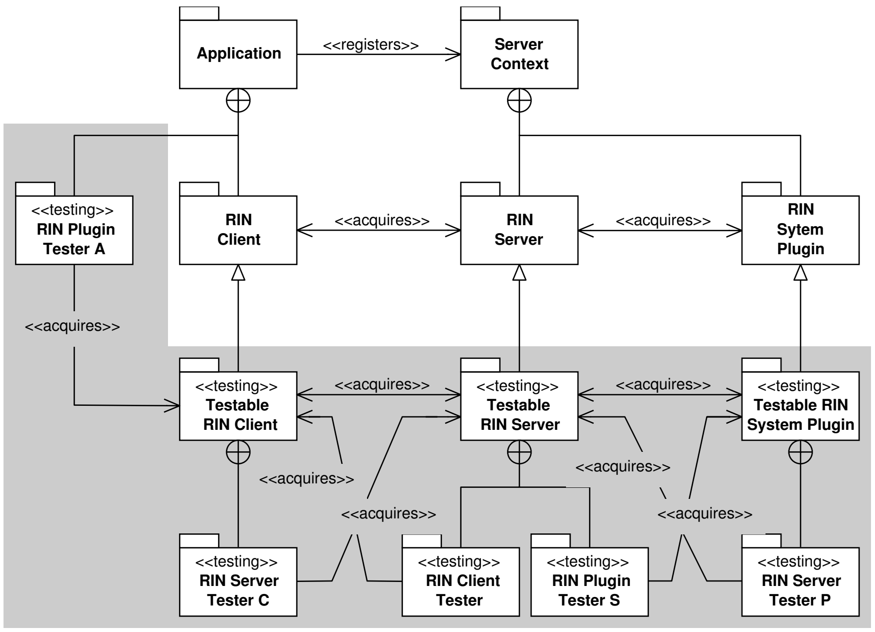

Application
<<registers>>
Server Context
<<testing>> RIN Plugin Tester A
RIN Client
<<acquires>>
RIN Server
<<acquires>>
RIN Sytem Plugin
<<acquires>>
<<testing>> Testable RIN Client
<<acquires>>
<<testing>> Testable RIN Server
<<acquires>>
<<testing>> Testable RIN System Plugin
<<acquires>>
<<acquires>>
<<acquires>>
<<acquires>>
<<testing>> RIN Server Tester C
<<testing>> RIN Client Tester
<<testing>> RIN Plugin Tester S
<<testing>> RIN Server Tester P

Fig. 4.18. Containment hierarchy of the amended RIN system including the built-in contract testing artifacts in the shaded area, but without the testing infrastructure for the implicit associations with the runtime support system

RIN components provide are represented by the three extension components to the original `RinClient`, `RinServer`, and `RinSystemPlugin`. Between these three components we have two-way client/server contracts, so we will have both built-in testing artifacts, testing interface and tester, for each component. Figure 4.18 displays the component `RinServerTesterC` that is contained in the `TestableRinClient` and represents the client's usage profile for the `RinServer`, and the `RinServerTesterP` that is contained in the plug-in and represents the plug-in's view on the server component. Both tester components are different because they represent different usage profiles for the `RinServer` coming from differing components. Additionally, we have a `RinSystemPluginTester` and a `RinClientTester` that represent the server's views on both components, client and plug-in, respectively. In the following subsection, we will have a look in more detail as how these testing artifacts are specified.

4.4.3 Specification and Realization of the Testing Interfaces

This step comprises the specification of an individual testing interface for the server role in an association. I have to add that this may be possible only if the component is an in-house development and we own its code. There is an exception with certain embodiment platforms which are considered in Chap. 5, but, in general, adding an interface to a component typically requires that we own a version of the component that permits such an amendment.

Entry criterion for the specification of the testing interfaces is a full functional specification for each operation of the tested component, for example, following the operation specification template of the KobrA method, or the behavioral model. These were introduced in Chaps. 2 and 3. Both models comprise sufficient information for the development of state setting and state checking operations that augment the functionality of the original component. The additional testing interface is used to set and retrieve state information of the component which is defined through the component's behavioral model. Each state in this model represents an item for which the behavior or an operation is distinctly different from any other item. The individual states that the behavioral model defines is therefore an ideal basis for specifying state setting and retrieving operations. Each state in the state model therefore maps to one state setting and one state checking method, but other state information can also be made public if required.

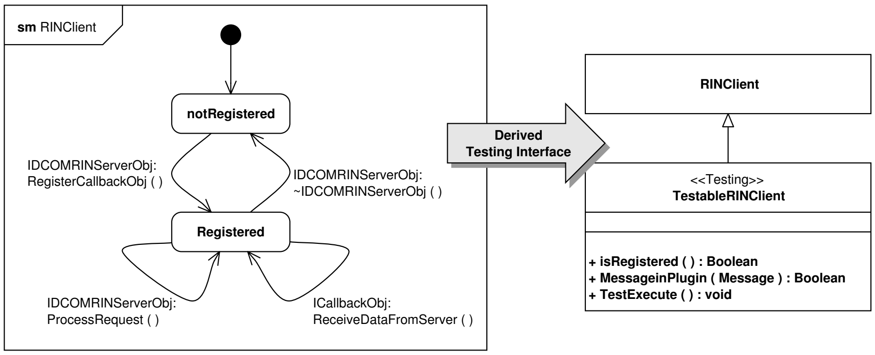


Fig. 4.19. Implementation-specific UML behavioral model of the *RinClient*, and the corresponding testing interface of *TestableRinClient*

The left hand side of Fig. 4.19 displays the `RinClient`'s behavioral model in the form of a UML statechart diagram. It proposes two states that may map to respective testing interface operations (indicated through the stereotype ≪testing≫) according to the built-in contract testing method introduced in the previous sections of this chapter:

```
<<testing>> isNotRegistered ( )
<<testing>> setNotRegistered ( )
<<testing>> isRegistered ( )
<<testing>> setRegistered ( )
```

The first two testing operations represent a state in which the object is not existing yet, because only the `RegisterCallbackObj` operation crates an instance of the component. If it resides in the state `notRegistered`, it does not provide any invocable service. The testing interface may therefore focus only on the second state in the behavioral model (`registered`), and only on the first specified operation, ≪testing≫`isRegistered`. The implementation of the `setRegistered` operation essentially repeats the implementation of the normal interface operation `RegisterCallbackObj`. Redundant implementations are useless, so the ≪testing≫`setRegistered` operation is omitted in the testing interface. The final specification of the testing interface is displayed through the structural model on the right hand side of Fig. 4.19. It defines two additional operations

- `MessageInPlugin` that is used to derive additional state information about the message that has been sent to the plug-in, and
- `TestExecute` that is used to invoke the `TestableRinClient`'s own built-in tester component, the `RinServerTesterC`.

The behavioral models and the corresponding testing interfaces for the server and plug-in components are depicted in Figs. 4.20 and 4.21, respectively. The testing interface for each tested component will be specified, e.g., according to the KobrA method, typically through structural models (e.g., class diagram) that show the signatures for the additional testing operations. The realization of these operations depends heavily on the realization of the functionality of the component. This is the main reason for why it is so important that the internal implementation of a component be available. The testing interfaces will have to access a component's internal attributes and manipulate them. In general, this is only feasible if we have access to the implementation.

4.4.4 Specification and Realization of the Tester Components

The first step has identified the relations between individual components that will be checked through the built-in contract testing approach. A client component in such a relation is referred to as testing component. Each of the testing components acquires a server, and it may be augmented with one or more built-in tester components (one for each server). Each of the tester components is developed according to the realization model of the testing component. In other words, each testing component owns a description of what it needs from its environment (its associated servers) to fulfill its own obligations. For example, in the KobrA method this is called the realization of the testing component, and it is defined in its realization model. It represents the expectation of the testing component toward its environment. In contract testing, the tests are not defined through the specification of the associated and tested server. In this case it would be merely a unit test of the server.

A tester contains the test cases and performs the tests on behalf of the testing component. A tester is typically realized as a component in its own

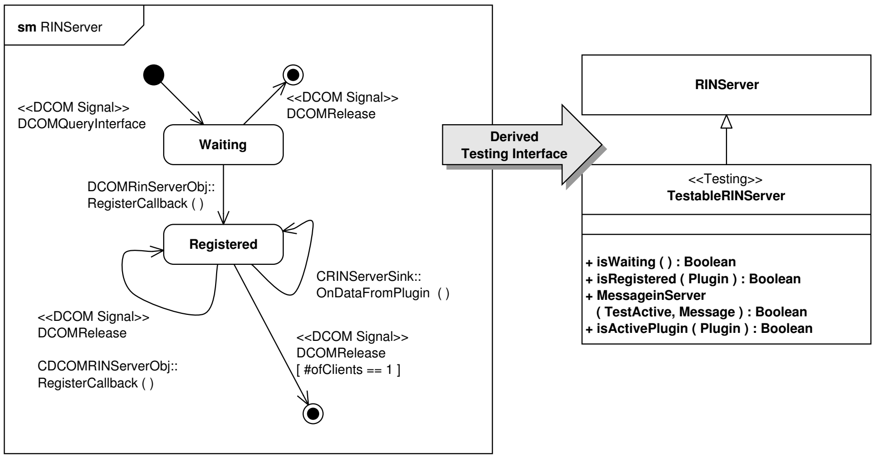


Fig. 4.20. Implementation-specific UML behavioral model of the *RinServer*, and the corresponding testing interface of the *TestableRinServer*

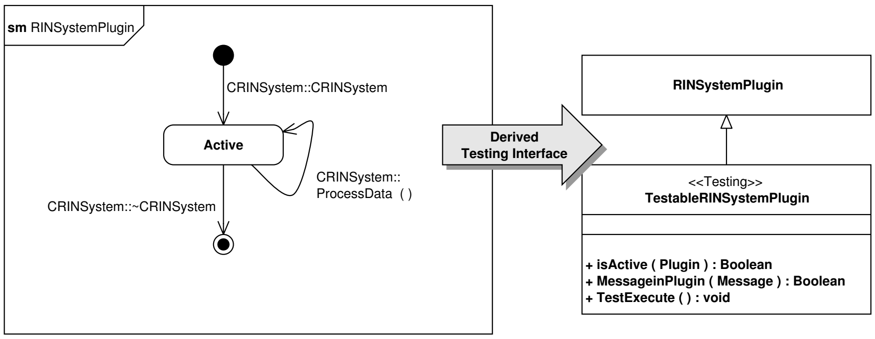


Fig. 4.21. Implementation-specific UML behavioral model of the *RinServer*, and the corresponding testing interface of the *TestableRinSystemPlugin*

right that the client acquires or contains. This was illustrated before, e.g., in Fig. 4.18. The difference between acquisition and containment is determined through the creating and using instance of the tester component. If the testing component *contains* its own built-in tester component, it will create it and keep a reference to it. In contrast, if it *acquires* its built-in tester component, it will need additionally a configuration interface through which the external context (or its own clients) may set its internal reference to a tester component. In this case, an instance of the tester component is created by the context and passed to the acquiring testing component. In essence, such externally acquired testing components can be seen as any other functional server components; they happen to comprise only code that runs a simulation of the transactions that the client typically performs on the server. The test cases inside the tester components are derived according to typical test case generation techniques such as domain analysis and partition testing, state-based testing, or method and message sequence-based testing. Models represent valuable sources for test case generation, as we have seen in Chap. 3.

Tables 4.2 to 4.6 specify the contents (test cases) of the respective tester components for the RIN System. The test cases are mainly based on the component's behavioral models.

Table 4.2. Specification of the test cases for the RIN client's *RinServerTester* component

No.	Initial State	Transition	Final State and Result
1	notRegistered	IDCOMRinServer:: RegisterCallbackObj ( )	Registered & Client added
2	Registered	IDCOMRinServer:: ProcessRequest ( )	Registered & Request processed
3	Registered	IDCOMRinServer:: ĨDCOMRinServer ( )	notRegistered & Client down

Table 4.3. Specification of the test cases for the RIN plug-in's *RinServerTesterP* component

No.	Initial State	Transition	Final State and Result
1	Registered	CRinServerSink:: OnDataFromPlugin ( )	Answer to the server & Registered

Table 4.4. Specification of the test cases for the RIN server's *RinClientTester* component

No.	Initial State	Transition	Final State and Result
1	Registered	IICallbackObject:: ReceiveDataFromServer ( )	Registered & Answer for Client

Table 4.5. Specification of the test cases for the RIN server's *RinPluginTesterS* component

No.	Initial State	Transition	Final State and Result
1	Registered	IICallbackObject:: ReceiveDataFromServer ( )	Registered & Answer for Client

The realization of the tester components is concerned with how an individual tester component will be organized and implemented, and which test suites it will contain. This may comprise tester subcomponents in the same way as realizations of normal functional components define subcomponents whose functionality they will acquire. Any tester component may in fact be a testing system in its own right, specified through a containment hierarchy. Testing systems may easily become quite voluminous. We can have a number of different tester components, each for a different purpose or objective, as the testing profile defines it. Each tester component specifies its own behavior that encompasses the various procedures for executing the test cases. Additionally, each test case is based on complex behavior, including the test stimulus that controls everything before, and some things during test execution, the test observation that controls some things during, and everything after test execution, and the validation of the outcome that leads to a verdict. All this testing complexity is supported through the UML Testing Profile. It supports the design and development of test cases and components with a number of items to take care of.

The following list summarizes the concepts that have been introduced in Chap. 3:

- Test objective that essentially defines a test case or a test suite.
- Test case that represents the application or execution of one sample to the system under test (SUT).
- Test behavior that describes the procedure of the test case.
- Test stimulus that defines what triggers the test; this comprises preconditions.
- Test observation that defines the outcome of the test, and the postconditions.
- Validation action that compares the observation with the expected outcome and eventually generates a verdict.

Figure 4.22 illustrates the mapping of the testing profile concepts to the `RINClientTester` component. Tables 4.2 to 4.6 define the test components in a tabular form. For the relatively simple built-in testing of the RIN system this may be sufficient. The tables define pre and postconditions, or initial and final state, plus expected outcome. This is enough information for realizing the tester components in an implementation language. How such testing models

Table 4.6. Specification of the test cases for the RIN application's *RinPlugin-TesterA* component

No.	Initial State	Transition	Final State and Result
1	registered & active	ProcessRequest ("bypass")	registered & active Req. bypassed
2	registered & active	ProcessRequest ("repeat")	registered & active Req. repeated
3	registered & active	ProcessRequest ("cancel")	registered & active Req. canceled
4	registered & active	ProcessRequest ("abstime" +	registered & active Req. timed
4.1	registered & active	"MemoryLoad")	registered & active total memory usage
4.2	registered & active	"TotalPhys")	registered & active phys. mem. usage
4.3	registered & active	"AvailPhys")	registered & active free phys. mem.
4.4	registered & active	"TotalPageFile")	registered & active page mem. usage
4.5	registered & active	"AvailPageFile")	registered & active free page mem.
4.6	registered & active	"TotalVirtual")	registered & active virt. mem. usage
4.7	registered & active	"AvailVirtual")	registered & active free virt. mem.
5	registered & active	ProcessRequest ("asap" +	registered & active instant Req.
5.1	registered & active	"TotalPhys")	registered & active phys. mem. usage
...	...	...	...

are turned into executable testing code in an embodiment step is the subject of Chap. 5.

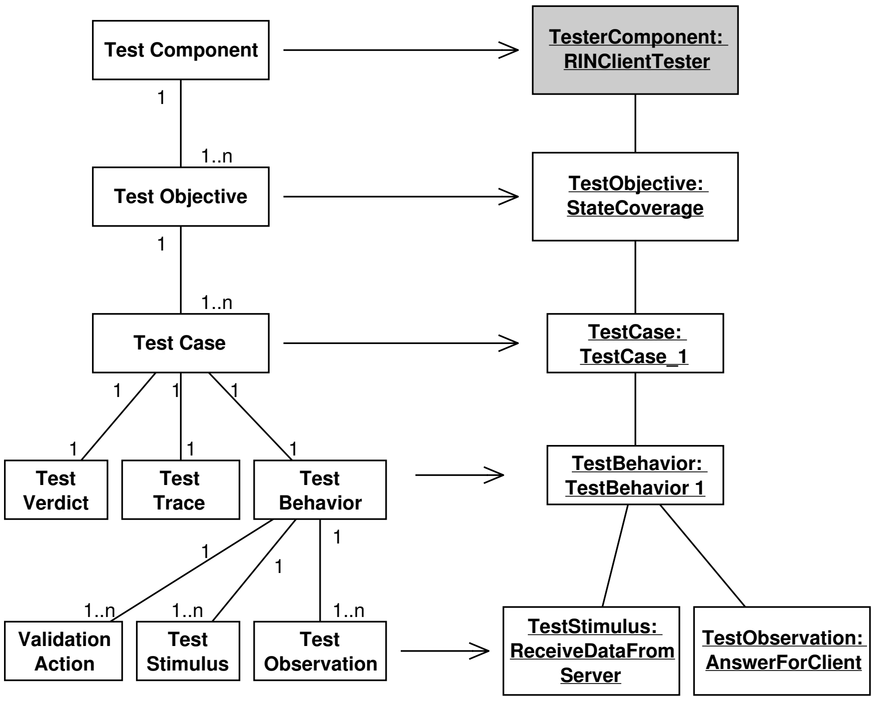


Fig. 4.22. Specification of a tester component through UML testing profile concepts for the *RinClientTester* component

4.4.5 Integration of the Components

Once all the functional component artifacts and the built-in contract testing component artifacts on both sides of a component contract have been properly defined and implemented, the two components can be integrated (plugged together). This follows the typical process for component integration, i.e., a wrapper is defined and implemented for the client and the server, or an adapter is designed and implemented that realizes the mapping between the two roles. In some implementation technologies, e.g., component platforms such as CORBA or CORBA Components, these mappings are readily supported through typical interface definition languages (IDLs). These can at least alleviate the efforts of implementing the syntactical mappings. Getting the semantics right represents another additional integration effort. This is part of the embodiment step and is laid out in more detail in Chap. 5.

Since the testing artifacts, tester component at the client role, and testing interface at the server role, are integral parts of the individual components on either sides of the contract, they are not subject to any special treatment. They are treated like any other normal functionality. Figure 4.23 illustrates this. Here, client and server have different required and provided interfaces, so that they must be mapped through an adapter. The adapter takes the operation calls from the client and transforms them to into a format that the server can understand syntactically as well as semantically. If the server produces results, the adapter takes the results and translates them back into the format of the client. Since the built-in contract testing artifacts are part of the client's and server's contracts, they will be mapped through the adapter as well.

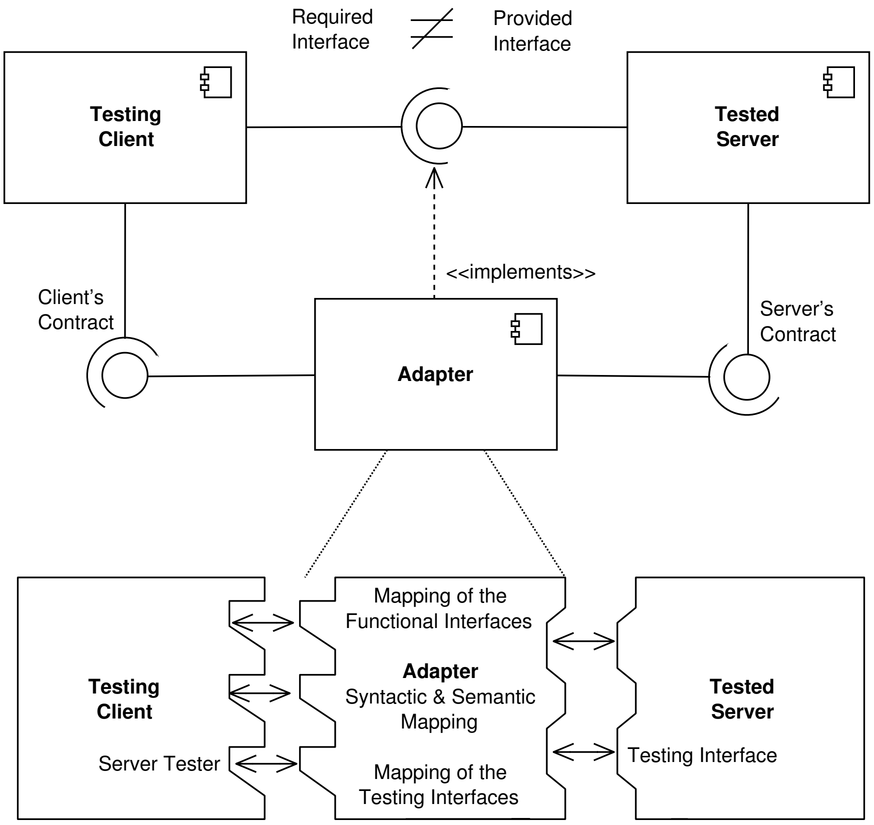


Fig. 4.23. Component integration through an adapter

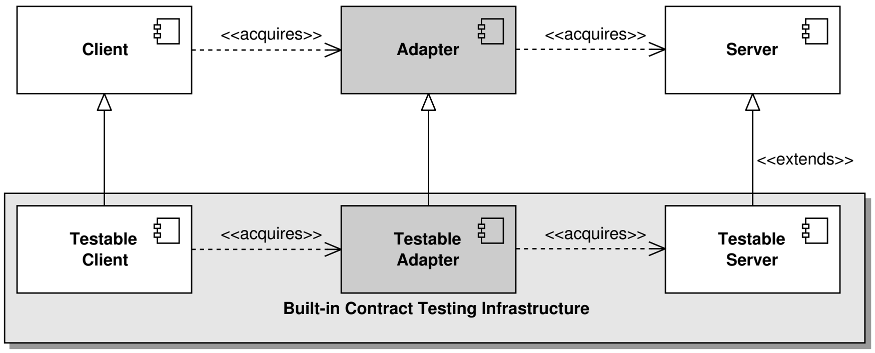


Fig. 4.24. Separation of an adapter into functional mapping and testing mapping through extension

I could even perceive the adapter to be designed and implemented fully in line with the principles of built-in contract testing. For the mapping between the pure functional components we may define a purely functional adapter, and for the testing infrastructure this is extended by mappings that additionally concentrate on the testing functionality. This leads to a structure that is displayed in Fig. 4.24. `Client` and `Server` are interconnected through `Adapter`, which mediates between the two differing contracts for the normal functional case. The testable versions of `Client` and `Server` augment both components with contract testing artifacts. This additional testing contract will be mapped through the extension of the adapter (`TestableAdapter` in Fig. 4.24) so that function and testing are fully separated in the two components as well as in the adapter.

In a containment hierarchy, integration is ideally performed bottom-up, so testing stubs for higher-level components that imitate the behavior of lower-level components may be omitted. Figure 4.25 displays two typical scenarios. The first one is the traditional top-down testing approach that requires stubs to replace required functionality that is not yet available for a test. The component ≪stub≫`C` represents this additional artifact. Ideally, in a component-based development project, component integration or composition is performed bottom-up, as discussed in Chap. 2, and the composition implementation is typically where the execution of the built-in tester components takes place. This means that all required subcomponents are available, so that the contract between components `B` and `C` in Fig. 4.25 can be checked before the contract between components `A` and `B`. The second interaction diagram indicates the invocation sequence of the tester components. Initially, the context calls the `startTest` operation on the highest-level component:

- First, this `startTest` operation calls the `startTest` operations of all subsequently associated components.

- Second, it calls its own built-in contract tester components for its immediate servers.

The first `startTest` invocation will essentially trigger the execution of the `startTest` of all subsequently nested components until the lowest level of the testing hierarchy is reached. So, the lowest-level built-in tester component is executed first. The invocation sequence will then move up hierarchic level by hierarchic level until the highest level is reached and its built-in tester components are executed. This testing approach follows the fundamental principle of application development that in its purest form is performed in a bottom-up fashion.

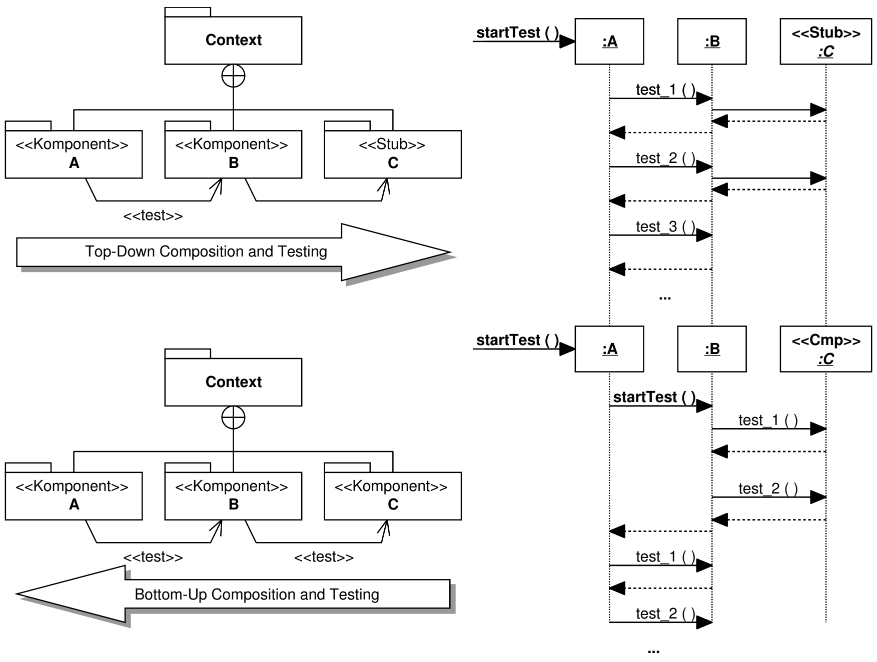


Fig. 4.25. Bottom-up vs. top-down composition and testing of components

4.5 Summary

Built-in contract testing addresses two of the three primary challenges in component-based software testing: low internal visibility of the tested object and component integration in various alien contexts that have never been anticipated by the object's original producer. The first issue is dealt with by the built-in testing interface that each reusable component is supposed to provide additionally to its original functional interfaces. Its design can be based on the externally visible states of the object, but more fundamental concepts such as assertions are also perceivable. In general, anything that makes a component more testable through external clients is fully in line with what built-in contract testing stands for.

The second issue is addressed by the built-in tester that each reusable component should readily provide. This contains tests that represent the object's expectation toward its associated server components and the underlying runtime system that in fact also represents a server component, albeit an implicit one. Built-in tester components are more fundamental to built-in contract testing because they assure that two components that have been brought together syntactically will also interact correctly in terms of their semantic contracts.

Tester component and testing interface will together make sure that a reusable component that is brought into a new context will behave according to what the context is expecting, and that the context will behave according to what the reusable component has been developed to expect. Built-in contract testing greatly alleviates the effort of integrating alien components in a component-based development project, because the components will readily provide their own in-built assessment strategies.

This chapter has concentrated mainly on the built-in contract testing architecture at higher levels of abstraction, or at the conceptual and modeling level. The next chapter will focus more on lower levels of abstraction, and how built-in contract testing may be realized through typical implementation technologies.

Built-in Contract Testing and Implementation Technologies

During the development of an application, after we have decomposed the entire system into subcomponents, we will iteratively move these components toward more concrete representations, starting from high-level UML models, through lower-level models, to code that is ready to compile and execute. All the design and development decisions that we take when we move toward more concrete representations are concerned with the concretization dimension in an iterative development process, and the activity is termed embodiment [6], as illustrated in Fig. 5.1.

Embodiment typically involves a last manual step along the abstraction dimension before the artifacts can be processed automatically. The outcome of this last step is a representation, some kind of source code (e.g., Java source code) that translators can understand and transform into a final executable form. In accordance with the prevailing terminology, this code representation is called the implementation of a system [6]. Most implementations traditionally exist in the form of code in high-level programming languages such as Cobol, C, Ada, C++, Java, etc., but recently component platforms such as COM, CORBA, and EJB are becoming more the implementation representation of choice when it comes down to modern distributed component-based systems and service-oriented products.

In the case of component technologies and component-based development, embodiment takes a very specific form. In component-based application engineering with third-party units, quite in contrast with traditional custom development, embodiment is not typically concerned with turning an abstract representation (i.e., a model) into a more concrete representation (i.e., source code). Here, it is more about integrating an existing implementation that is the readily deployable third-party component into the existing framework that the abstract model is providing. Some of this integration is readily supported by component platforms, but it also typically involves the creation of some additional models and code, the so-called "glue code." This adapts existing components to the integrating framework (or the other way around) in terms of syntax and semantics. It does this by transforming something that

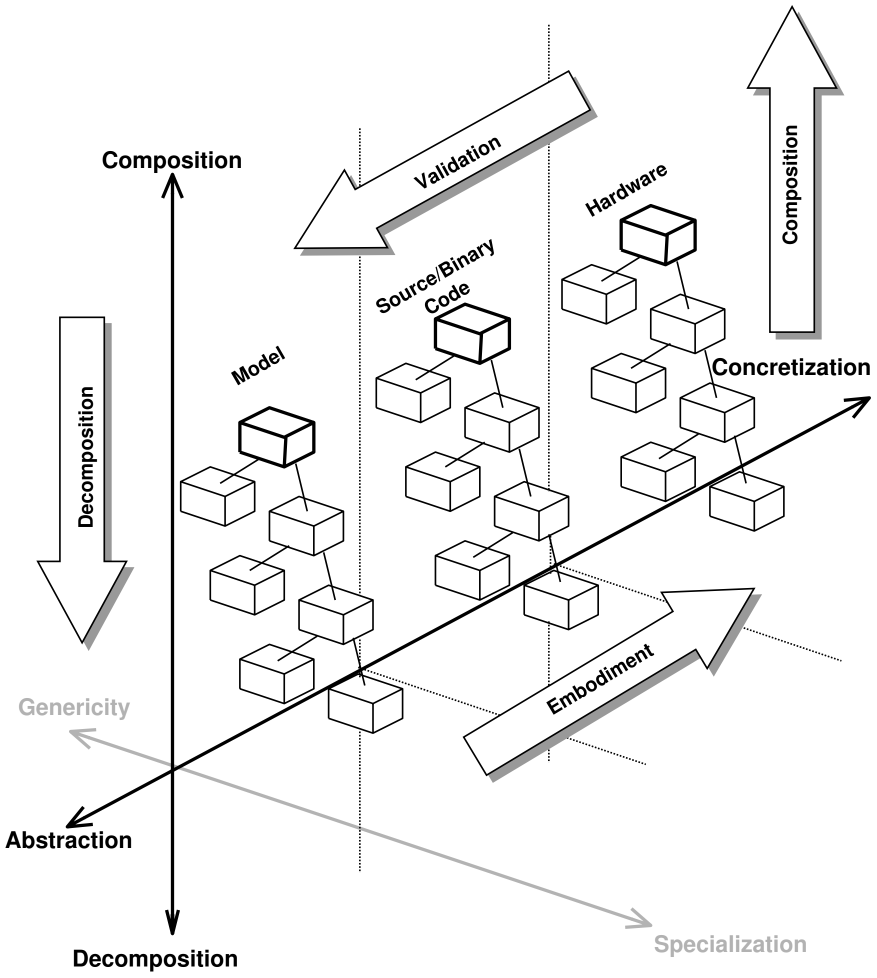


Fig. 5.1. Decomposition and embodiment in an iterative development approach

the framework is "meaning" into something that the component can "understand," and something that the component is replying into something that the framework will "understand."

Figure 5.2 illustrates this additional effort of refining the original model of the integrating framework and augmenting it with additional infrastructure to accommodate the existing and reused component. A translation from the model into some source code turns this infrastructure into its implementation form that in turn may be compiled and integrated with the existing and reused component. So, a typical embodiment activity in component-based development goes along the decomposition dimension to come up with a refined component infrastructure plus a single step along the concretization dimension to integrate existing executable reusable modules. These are steps along two dimensions, and they may also be supported by specific refinement pattern, in a form that I have introduced in Sect. 2.5 (Chap. 2). In other words, embodiment in component-based development is mainly a reuse activity plus a development activity for the actual integration effort, i.e., development of the "glue code." Most contemporary component platforms support this integration effort in some way or another. How they do this and how built-in contract testing will be embodied is the subject of this chapter.

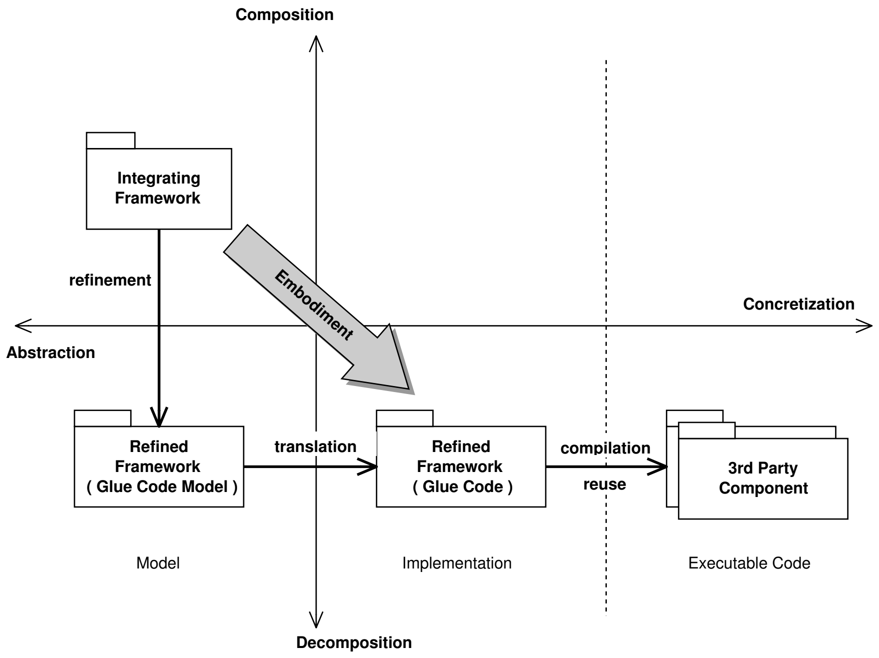


Fig. 5.2. Typical embodiment comprises refinement and translation

The implementation of a component, or in other words its binary executable version, is what most software organizations are interested in. Contemporary component platforms, or the so-called middleware platforms such as COM/DCOM, Java/EJB/J2EE, CORBA, and CORBA Components, represent modern runtime support systems for these binary modules. On the one hand, they reduce the manual effort of making different components that have been developed in mutual ignorance interact with each other. This probably represents the most fundamental feature of component platforms. On the other hand, and this has probably become much more important now, they provide typical operating system functionality. This includes containers that encapsulate components, networking and security infrastructure, hardware management, and the like. Essentially, we could argue that anything that represents typical operating system infrastructure that has not yet made it into the operating system can be called middleware. In effect, the middleware provides the link between the high-level user application and the lower-level operating system service, and the infrastructure for the link between the different components that are residing on a middleware platform. This is in fact what programming environments and operating systems can address only insufficiently. In an operating system we can only deploy and execute components that are specifically compiled and linked for that particular system, because, for example, components rely on specific libraries that the operating system provides. In a particular programming language we can interconnect only modules that have been developed in that language. We can possibly find interchange technologies between objects that reside in different runtime contexts, but they typically address only bindings between any two particular languages, and they have to be explicitly coded within the components. So, traditional component interchange technologies that are working at the operating system or programming language level are extremely restricted with respect to their flexibility and openness. Middleware platforms on the other hand mediate between the different restrictions that operating systems and programming languages pose on a development. They accept components that may be written in any arbitrary programming language, as long as there is a binding between the middleware platform and the programming language, and they accept any operating system, as long as there is a binding between the operating system and the middleware.

Contemporary middleware platforms represent only an excerpt of the feasible embodiment and implementation technologies. They provide only the execution framework for multiple diverting component instances. In order to get to a final representation of a component that can be executed in such a context, there are a number of additional artifacts and notations that can be applied and must be considered. They will comprise generators and other tools that are readily applied during the embodiment phase, and they have to be considered at all levels of abstraction and decomposition. All these technologies have an effect on how abstract representations are turned into executable formats, and how the built-in testing artifacts will be incorporated.

The following section (Sect. 5.1) shows how the embodiment of the built-in contract testing models is treated in general, in terms of a typical product line engineering activity. Section 5.2 concentrates on how the built-in contract testing can be realized through high-level programming languages such as C, C++, and Java. Additionally, I introduce the J/BIT Library that represents a Java support library for implementing testing interfaces for testable Java classes. Section 5.3 introduces the main concepts of the primary contemporary component technologies and how they may be applied in tandem with the built-in contract testing technology. A subsequent section (Sect. 5.4) focuses on how Web services, a similar technology, may be treated during testing, and Section 5.5 describes two solutions for implementing the built-in contract testing artifacts, the Testing and Test Control Notation and the XUnit testing framework. Finally, Sect. 5.6 summarizes and concludes this chapter.

5.1 Instantiation and Embodiment of Built-in Contract Testing

The embodiment activities that we have to perform for the testing functionality of our system are not different from those that we have to perform for the normal functionality of the system. Since contract testing is built-in, its development represents merely an additional modeling and implementation effort that follows the same principles and procedures as any other software development. These principles were introduced in Chap. 2. For each identified component in our development tree, we can define a testable version that provides an introspection mechanism, or a testing interface, and a testing version that owns or provides a tester component. These two additional artifacts represent the two views according to the two distinct roles in a client/server relationship:

- The testing interface is located at the server role in a client/server contract to provide additional access mechanism and facilitate or support the testing that clients may perform on a server.
- The tester component is directly built into or dynamically acquired by the client role in a client/server contract to perform an assessment of the associated server component during deployment.

The built-in contract testing interface enhances the functionality of a server to make it more testable for other associated client components, while the built-in contract tester component adds to the behavior of the client to enable it to apply tests on its environment. Both testing concepts are modeled and implemented as additional functionality that components may typically incorporate and use at any time like any other functionality. As long as the contract testing concepts are hardwired into the implementation of a component, there is no different way of dealing with it.

However, in order to apply a recursive component-based development method adequately, for example, the KobrA method that propagates a clear separation of concerns, and to fully use the capabilities of modern object and component technologies, it is essential to separate between a component's functional and testing properties. The two most important benefits from a clear separation of the two aspects, function and testing, in a system development are

1. that we can assign the right expertise and resources according to these aspects during component development and
2. that we can instantiate a testable and a non-testable version of the system, according to what is required.

The first one represents another way of applying the successful engineering strategy, "divide-and-conquer," in order to break the system down into more manageable parts and find and assign the right people for dealing with them individually, for example, developers for the function and testers for the testing. The second one organizes the testing part of a system in a way in which it cannot easily interfere with its original functionality and can be easily removed if necessary. This is important for a number of different reasons:

- Systems that are not evolving dynamically during runtime in terms of their internal architecture as well as their runtime environment need only built-in contract testing for their initial deployment. This is probably the case for most systems. As soon as all component interactions have been established and checked for their semantic compliance with their individual contracts, there is no reason for the built-in contract testing infrastructure to remain in place. The execution of any of the built-in contract tester components will not reveal any new problems.
- Systems with sparse resources (i.e., embedded systems) cannot accommodate the built-in contract testing infrastructure in their runtime environment, and if the tests are never used again they merely consume time and space.
- Permanently built-in testing code can be invoked by a client anytime during runtime, even if it is not intended. The invocation of a test may interfere with a system's normal operation.

In general, if we organize and implement built-in contract testing in an intelligent way, we can achieve the previously described variability in a system, and switch testability on and off according to the requirements at hand. If we view our system as a product family or product line we can apply all the powerful product line engineering concepts that I have briefly outlined in Chap. 2 in order to organize testability and testing. Product line engineering is concerned with how to organize a group of similar systems to fully exploit their commonalities and manage the differences between each of these systems. Here, the product line or product family represents a system core that is common to all products in that family. Each individual system in the product family represents a distinct variation or extension of that common core. These are also called variants [6] (see Chap. 2). In Chap. 2, I have introduced how product lines are dealt with in the scope of a component-based development method. In Chap. 6, we look at how built-in contract testing deals with the test of product line developments. In this section we will look only at how the implementation of built-in contract testing may be seen as a product line development and how that can facilitate the management of the built-in testing infrastructure.

During modeling, from the beginning of a development project, we can already identify many things that can serve for testing, as we have seen in Chap. 3. So, from the beginning of a project when we identify the coarsest-grained components we can already define some of the testing for these components.

Since in our case testing is built into the components from the beginning, we can separate the testing model from the functional model of our final product and view the testing model as a variant of the original functional model.

When we introduce the subject of dealing with variations into our process, we have to add a third dimension to our recursive development model depicted in Fig. 5.1 which is now extended in Fig. 5.3. The extension is the genericity/specialization dimension along which we can move in a product line development. The direction of genericity in Fig. 5.3 means that we are moving to a more generic system that is capable of serving more generic applications. This direction eventually leads to the product family core which is also referred to as the framework. The framework is so generic that we can use it and build from it all the various concrete final products in that product family. Dealing with the framework and developing it is termed framework engineering. The direction of specialization in Fig. 5.3 means that we are moving to a more specific system that is capable of serving a very special purpose only. This direction eventually leads to a single concrete final product. Dealing with this product is referred to as application engineering. A move along the genericity/specialization dimension from a more generic framework to a more specific application is called instantiation of an application out of a framework [6].

With these concepts in mind we can approach the development and implementation of the built-in contract testing infrastructure. The framework or the core of the product family represents the functional system; this is a complete working version of the system without any built-in testing. We can extend it with the built-in contract testing infrastructure. So, each component in the containment tree will be augmented with a testing interface and some tester components. The outcome of this instantiation is a testable version or a testable variant of the original core system. If we perform an embodiment step for the framework we will receive an individual workable non-testable version of our original system. If we perform an embodiment step for the testable variant we will end up with a fully testable workable version of that very same system. Instantiation adds all the contract testing artifacts and embodiment turns the product into a final testable implementation. This process is summarized in Fig. 5.3. The full development process will start at the top-level box on the left hand side with the decomposition of the system. Then we can perform an instantiation and add some testing models, or alternatively decompose the system a bit more. We can also start the embodiment for the parts that have been fully defined; this works for the original system as well as for the testable system.

The separation between artifacts that belong to the core and those that belong to a variant are typically indicated through a stereotype in the model. In the UML model the stereotype ≪variant≫ is placed in front of the variant features. I have introduced this concept for product line development in Chap. 2. However, for testing we should use a different stereotype, for example, ≪testing≫ in front of the testing artifacts in the model, to distinguish

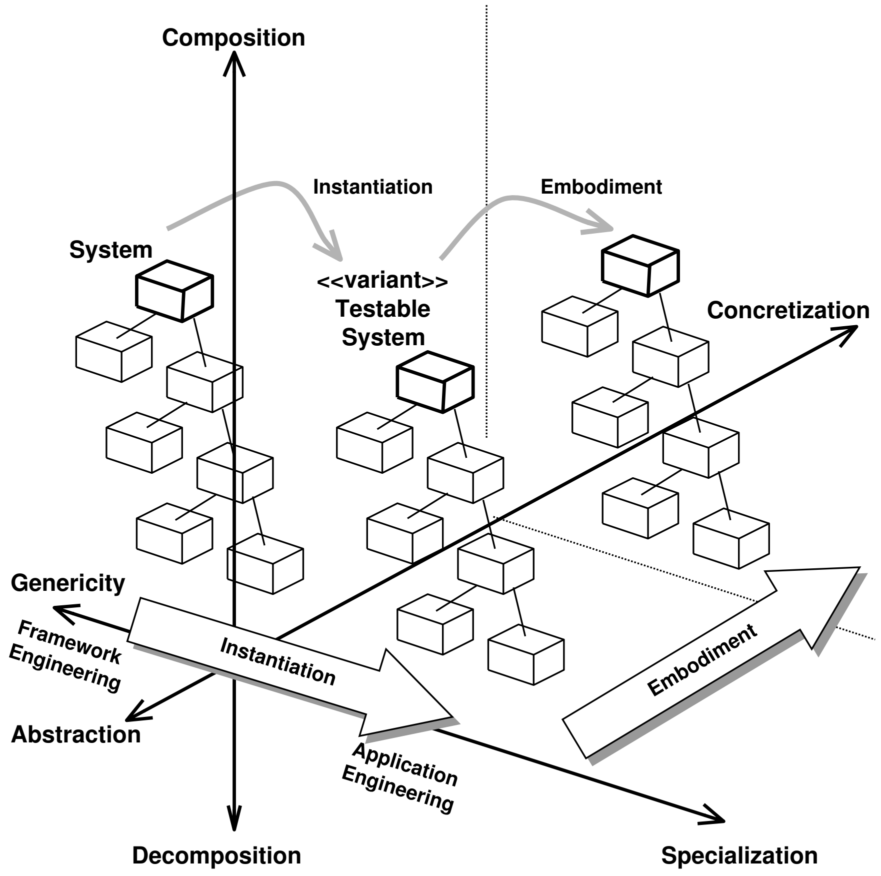


Fig. 5.3. Instantiation of the testable system from the original system, and embodiment of the testable variant

a typical product line development artifact from a typical built-in contract testing artifact. In a nutshell, product line concepts represent a well understood and standard way of dealing with testing variability, and in tandem with a recursive development process they can facilitate the organization and implementation of built-in contract testing considerably.

By following the previously described process, system engineers get a very clear view according to which aspect of a system they are currently working on, and how it adds to the overall development. This way of organizing built-in contract testing strictly follows the paradigm of "separation of concerns" that is put forward in all engineering disciplines. Now that we have had a look at how testing embodiment activities can be incorporated into the overall iterative development model of the KobrA method, we can now turn to how embodiment is performed more concretely in common implementation notations.

5.2 Built-in Contract Testing with Programming Languages

It is arguable whether artifacts in standard implementation notations such as Ada, C, C++, Pascal, or Java can be termed components. According to Szyperski's component definition [157, 158] that was established at the 1996 European Conference on Object-Oriented Programming (ECOOP'96), they are not. This defines a component as a unit of composition with contractually specified interfaces and context dependencies only, something that can be deployed independently and is subject to composition by a third party. The term "independently deployable" implies that such components will come in binary executable form, ideally with their own runtime environments if they are not supported by a platform. Implementations are not typically directly executable unless they are interpreted, such as Perl or Python. As said before, implementations always need this last transformation step to be independently deployable.

However, in a model-driven approach, as put forward in this book, an implementation is merely a section or a phase along the abstraction/concretization dimension as displayed in Fig. 5.1. It belongs to a transition between formats that humans can understand easily into formats that are easier for a machine to "understand" and process. The term component is related more to composition of individually solvable and controllable abstractions or building blocks. This terminology is motivated through a typical divide-and-conquer approach that splits a large problem into smaller and more manageable parts, so the term component is related more to the composition/decomposition dimension in Fig. 5.1. If we follow this philosophy, an abstract model (for example, one of the boxes in Fig. 5.1) may denote a component, and what it actually does. We can handle such a component in an abstract way, for example, we can perform some composition and incorporate it into an abstract component framework that is entirely defined in an abstract notation. So, whatever we can do with a concrete representation, i.e., at the code level, can be done in a more abstract representation, i.e., at the model level. Hence, under a model-driven approach all the properties of concrete executable components are meaningless because they are freed from the shackles of their concrete runtime environments. In my opinion, Szyperski's component definition does not explicitly separate between the two dimensions, composition and abstraction. In this respect, any artifact along the abstraction/concretization dimension may be regarded as a component as long as it is identified as a component in the composition/decomposition dimension. Typical implementation notations may therefore well be seen as complying with the component philosophy, and we can also include typical non-object-oriented implementations, because at the abstract level they can be treated as any other object at that level. In other words, any object technology principle at a high abstraction level can be transformed into non-object technology artifacts on a lower level of abstraction. The following two sections look at how built-in contract testing may be realized through C, C++, and Java implementations.

5.2.1 Procedural Embodiment Under C

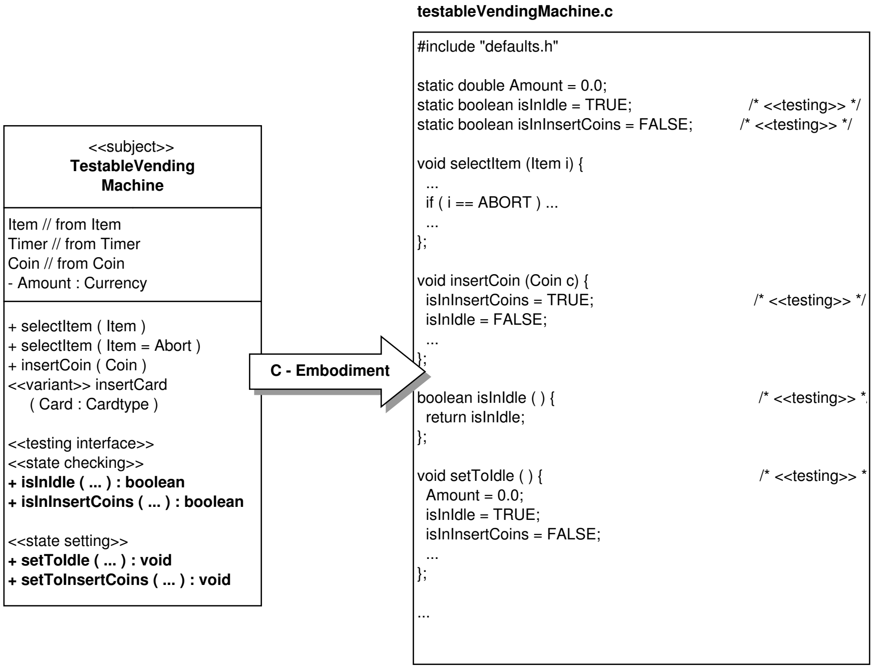
The C language belongs to the most widely used implementation technologies, and not only for very technical contexts or system programming for which it has been initially developed. C explicitly supports modular programming through the concept of source code files as modules or components and their separate compilation. It therefore also incorporates everything that is necessary for implementing information hiding, a basic principle of object technology [93]. The module concept manifests itself in a number of ways:

- Modules can be organized hierarchically. This adheres to the principles of composition and explicit dependencies or contracts between the hierarchically organized modules.
- Modules can be reused. This realizes the most rudimentary reuse principles and, in fact, follows Booch's component definition which sees a component as a logically cohesive, loosely coupled module that denotes a single abstraction [20].
- The two previous items lead to the notion of platform independency and abstraction. The module concept essentially hides and encapsulates underlying implementations that are platform-specific, such as system libraries, and other operations closer to the hardware.

C does not explicitly provide typical object-technology properties for implementing built-in contract testing, but since C++ can, and is usually implemented in C, the ideas behind built-in contract testing can well be adapted to C implementations. In C everything locally defined, i.e., through "static," will be accessible only through the procedures in the same scope or file. These can be seen as the attributes of an object in an object-oriented language or the data variables that the module encapsulates. The procedures that are comprised in the module can be seen as the methods of an object, or the module's external interface. The procedures collectively define the contract that the module is providing. The only difference with C++ objects is that their cohesiveness is determined through the data that the object encapsulates, while in a C module cohesiveness is determined through the functionality of the procedures (functional cohesiveness, or functional similarity).

Built-in Testing Interface

For built-in contract testing, we can implement a testing interface for each server module and a tester component for each client module. A testing interface can be implemented either if a C module encapsulates internal data, such as static variables, or if it will be used to control a module's built-in assertion checking mechanisms, which in fact represent internal state information as well. In C, a testing interface for a module is a collection of additional procedures that are added to the existing functionality. These additional procedures are local to the file of the module and provide an additional access mechanism to that module. Such an organization, internal static variables, and procedures to access these variables, effectively realize an encapsulated entity with class-like properties. The only difference with C++ is that in C we can have only one instance of such a module per process, i.e., per "a.out" file. This limits the value of a testing interface in C because the procedural philosophy of the language adheres to the separation of data and functionality that in effect leads to stateless components. But we can nevertheless easily implement contract testing interfaces in C. For example, Fig. 5.4 illustrates the embodiment of the `testableVendingMachine` component model that is taken from Fig. 4.4 on page 137 (Chap. 4). Here, all additional ≪testing≫ artifacts are hardcoded as specified in the original model.



```
#include "defaults.h"

static double Amount = 0.0;
static boolean isInIdle = TRUE;                /* <<testing>> */
static boolean isInInsertCoins = FALSE;        /* <<testing>> */

void selectItem (Item i) {
  ...
  if ( i == ABORT ) ...
  ...
};

void insertCoin (Coin c) {
  isInInsertCoins = TRUE;                      /* <<testing>> */
  isInIdle = FALSE;
  ...
};

boolean isInIdle ( ) {                         /* <<testing>> *
  return isInIdle;
};

void setToIdle ( ) {                           /* <<testing>> *
  Amount = 0.0;
  isInIdle = TRUE;
  isInInsertCoins = FALSE;
  ...
};

...
```

Fig. 5.4. Embodiment of the *testableVendingMachine* model in C

Built-in Tester Component

The built-in contract tester components at the client's side comprise test cases that simulate the client's access to the server module. There is no difference to other implementation technologies that are based on object technology. The value of built-in contract tester components is limited due to the limited number of feasible different usage profiles that a client can present. Since C modules do not typically encapsulate states that have an effect on the different operations, they are also not going to have any effect on the sequence or combination of operation invocations in the modules. All parameters that are provided to such a module are provided from outside its encapsulation boundary. It has no memory. Every operation of a module can therefore be seen as a stand-alone entity without dependencies on any other of the operations, and every operation can be tested in isolation to any other of the operations. Much of the complexity of testing object-oriented systems can be attributed to the interdependencies between individual class operations that are caused through the common data that they access. Under the procedural development paradigm, a unit test of a C module can therefore be regarded as a viable option even if the module will be used in a number of different applications. Other clients will not use such a module much more differently from the way it was initially intended to be used by the provider.

Built-in contract testing may offer only limited gain for the testing of component contracts in an application that is implemented in C. Where it may be much more successfully applied is at the interface between the C implementation and the underlying platform. One of the big challenges in C used to be, and still is, portability, or, in other words, a move of a C implementation from a development platform into a deployment platform. This particular case will be greatly supported by built-in contract tester components that are located at the transition between the user-level application and the underlying support platform that comprises not only the hardware but also the required support libraries and drivers. The built-in contract tester components can be invoked at deployment after the application has been brought to the new platform for the first time. They will contain tests that simulate the application's normal interactions with the underlying support software that is part of and installed on the platform. In this instance, built-in contract testing will alleviate the efforts of assessing whether an application will function properly on a particular platform, and it additionally points out the location of failure. This simplifies problem identification.

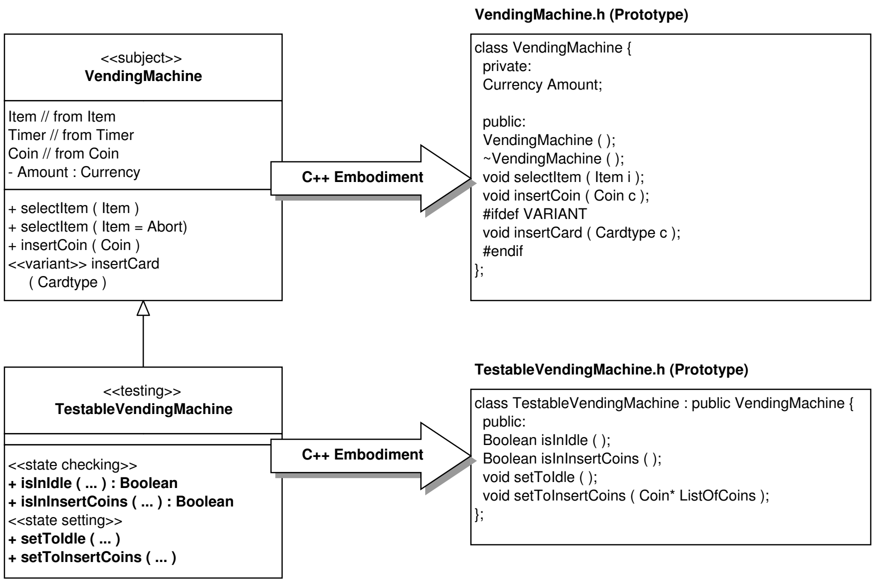
The C language lacks proper support for typical object technology properties that are advantageous for the implementation of built-in contract testing, such as dynamic assignment of components and an extension mechanism. This lack of basis technology prohibits dynamic assignment of tester components during runtime as well as dynamic instantiation of testable and non-testable components. In C most built-in testing artifacts will have to be hardcoded, so that they are available at all times in binary representation. This is mainly the case for the built-in testing interface, since C does not provide extension except through conditional preprocessing at the implementation level. The built-in tester components can be made dynamic to a certain extent, through the use of function pointers, though this is quite awkward compared with the capabilities of modern object-oriented languages and nobody will probably bother to use that. These limitations therefore require a well planned application of built-in contract testing technology in procedural languages such as C. In the next section we will have a look at how modern object-oriented languages support the use of built-in testing technology.

5.2.2 Object-Oriented Embodiment Under C++ and Java

C++ and Java are the most commonly and successfully used embodiment technologies for object-oriented implementations. They both represent success stories with respect to industrial penetration, though it is quite clear that Java is much more modern and represents a much cleaner and simpler way of realizing object-oriented programming than the C successor. C++ is halfway between traditional procedural programming à la C and object-oriented development as it is understood by Java. It is a superset of C, so it is still equipped with the procedural capabilities of its predecessor. Java lacks some of the typical C/C++ features that some programmers regard as cure and others as curse, for example, preprocessor instructions, direct memory access, multiple inheritance of implementation, implicit type conversion, and typical C legacy such as "goto," unions, global variables, and the like. Apart from a number of differences between the two programming languages that are thoroughly discussed in the literature (e.g., [110]), they both come equipped with the right means for implementing built-in contract testing in the way it is described in Chap. 4.

Built-in Testing Interface

Figure 5.5 shows a prototypical C++ implementation of the `TestableVending Machine` component. Figure 5.6 shows a prototypical Java implementation of the `TestableVendingMachine` component. In these implementation examples I have suppressed the variant feature that the model specifies, but I will come to that later on. Both implementations can be derived directly from their respective UML model, because both languages readily support UML's class concept; or, if we look at it the other way, because the programming languages are much older than the modeling notation, UML provides support for specifying classes how the two programming languages view it. For such simple specifications the mapping between model and code is straightforward. Java provides through the concepts of abstract classes and interfaces a much more advanced way of defining prototypes than C++. In my opinion, this leads to a clearer and simpler design in Java, although this is arguable.

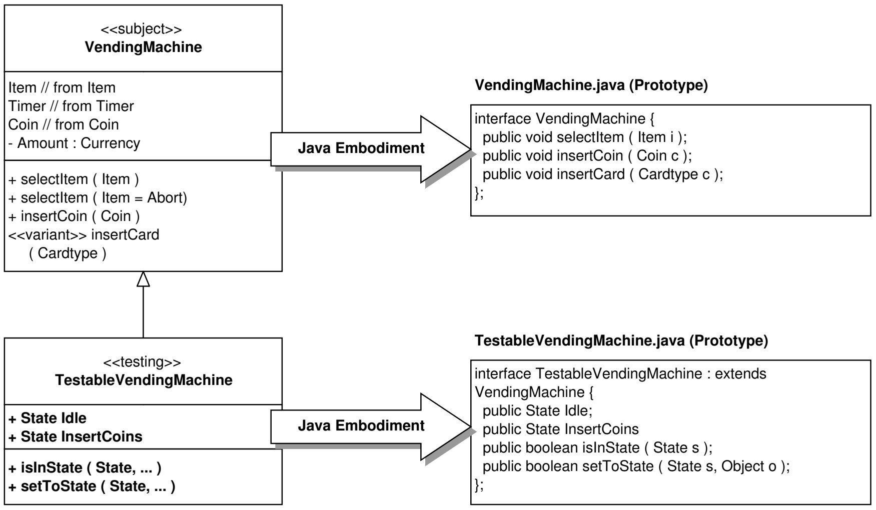


```
class VendingMachine {
  private:
  Currency Amount;

  public:
  VendingMachine ( );
  ~VendingMachine ( );
  void selectItem ( Item i );
  void insertCoin ( Coin c );
  #ifdef VARIANT
  void insertCard ( Cardtype c );
  #endif
};
```

```
class TestableVendingMachine : public VendingMachine {
  public:
  Boolean isInIdle ( );
  Boolean isInInsertCoins ( );
  void setToIdle ( );
  void setToInsertCoins ( Coin* ListOfCoins );
};
```

Fig. 5.5. Embodiment of the *testableVendingMachine* model in C++



```
interface VendingMachine {
  public void selectItem ( Item i );
  public void insertCoin ( Coin c );
  public void insertCard ( Cardtype c );
};
```

```
interface TestableVendingMachine : extends
VendingMachine {
  public State Idle;
  public State InsertCoins
  public boolean isInState ( State s );
  public boolean setToState ( State s, Object o );
};
```

Fig. 5.6. Embodiment of the *testableVendingMachine* model in Java

In the C example I have incorporated the variation point for dealing with the optional functionality of a vending machine that also supports credit card billing. This can be done in the same way for C++, because the language provides preprocessor instructions. For the Java example it does not work like that because Java does not provide a preprocessor that incorporates functionality according to optional design decisions. If we would like to implement this in Java, or implement it differently in C++, we have to change the model, and this represents a refinement and a translation step. Without the two steps we would implicitly assume design decisions that will never appear anywhere in the documentation of the system. For such a simple system as that we are dealing with, this might be okay. For larger systems, however, it is essential that such design decisions be documented well. I will briefly explain how the refinement of the models and the translation into Java code can be carried out.

The existing model in Fig. 5.5 maps only directly to C or C++. For a different Java implementation we will have to refine the model according to the limitations of Java in the way I have explained in Chap. 2. This refined Java implementation-specific model is displayed in Fig. 5.7. The variation point is realized as an extension to the original model, `ExtendedVendingMachine`, which can be extended through the testing interface, turning it into a `testableVendingMachine`. The translation step for this example is depicted in Fig. 5.8.

Built-in Tester Component

Table 5.1 repeats the specification of the `VendingMachineTester` component from Chap. 3. Initially, this test may be performed at the user interface of the vending machine, and it would be carried out through a real human tester who is performing the task described in the table. Alternatively, we can have the `VendingMachineTester` component access the `VendingMachine` component directly. So, the test will actually be applied like a simulation of a real user who performs transactions on the vending machine. Embodiment is essentially concerned with turning the test specification in Table 5.1 into source code, for example, in Java. This is laid out for the first test case in Table 5.1 in the following Java source code example. A prerequisite for executing this test is that the precondition, `ItemX == Empty` or `ItemX != Empty`, holds. The preconditions can be set through a stub that emulates the `Dispenser` component, or they can be set through the testing interface of that component if no stubs are used. This requires that the dispenser and the display component also provide testing interfaces according to the built-in contract testing paradigm through which a client tester can also set and check initial and final states. The organization of this testing system is represented by the containment hierarchy in Fig. 5.9.

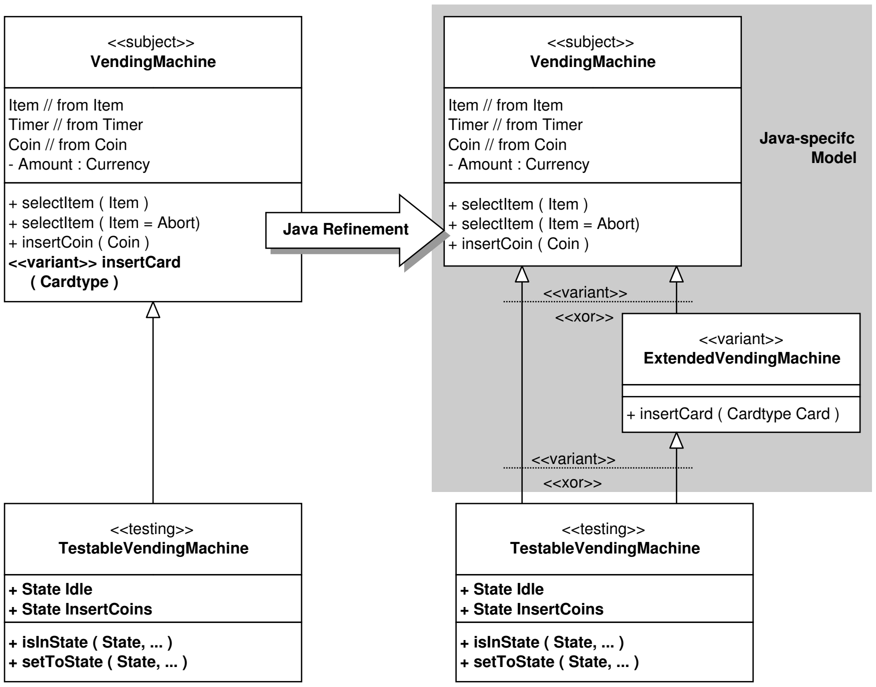


Fig. 5.7. Refinement of the *VendingMachine* model for a Java implementation

The following source code example represents a feasible Java implementation for the `VendingMachineTester` component specified in Fig. 5.9 and in Table 5.1. Here, I give only an excerpt of the full tester component:

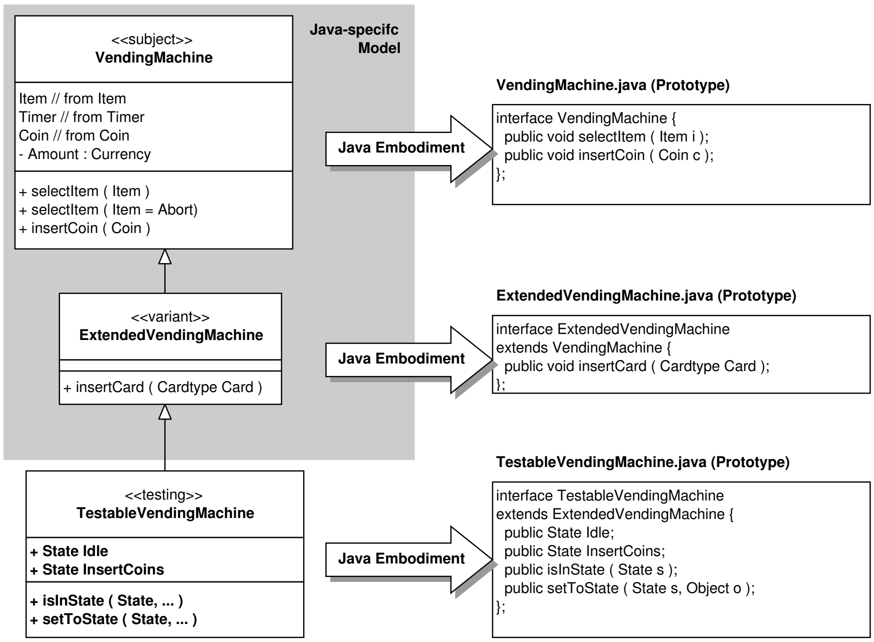
```
class VendingMachineTester {

  // configuration interface
  private object Dispenser;          // test bed
  private object Display;            // test bed
  private object TVM;                // tested component

  public void setDispenser (object testableDispenser) {
    Dipenser = testableDispenser;
  }

  public void setDisplay (object testableDisplay) {
    Display = testableDisplay;
  }

  void setTestableVendingMachine (object tvm) {
```



```
interface VendingMachine {
  public void selectItem ( Item i );
  public void insertCoin ( Coin c );
};
```

```
interface ExtendedVendingMachine
extends VendingMachine {
  public void insertCard ( Cardtype Card );
};
```

```
interface TestableVendingMachine
extends ExtendedVendingMachine {
  public State Idle;
  public State InsertCoins;
  public isInState ( State s );
  public setToState ( State s, Object o );
};
```

Fig. 5.8. Translation of the Java-specific model of the *testableVendingMachine* into a Java implementation

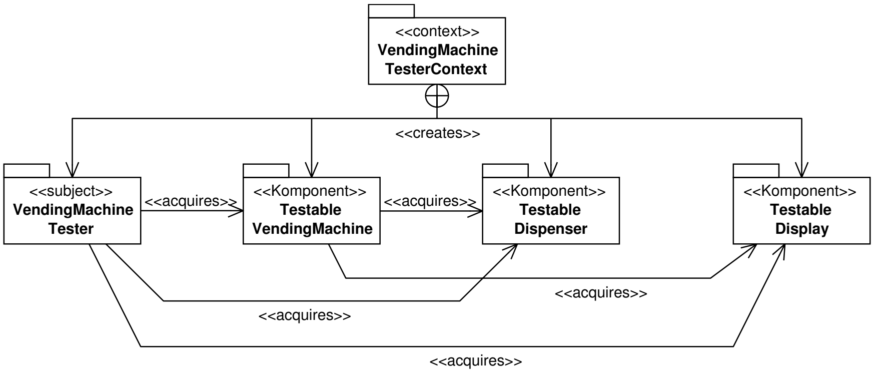


Fig. 5.9. Containment hierarchy of a testing system for the *VendingMachine* component

```
    TVM = tvm;
  }

  // start test

  public boolean startTest () {
    if (false == startTest11 ()) return false;
    if (false == startTest12 ()) return false;
    if (false == startTest13 ()) return false;
    if (false == startTest14 ()) return false;
    if (false == startTest15 ()) return false;
    if (false == startTest21 ()) return false;
    if (false == startTest22 ()) return false;
    if (false == startTest23 ()) return false;
    ...
    return true;
  }

  // test cases

  public boolean startTest11 () {   // TEST 1.1
    Dispenser.setTo (Item1, empty); // set precondition
    TVM.setTo(idle);                // set init state
    try {                           // expect exception
      SelectItem (Item1);           // call transaction
    } catch (DispenserItemEmptyException e) {
      if (TVM.isIn(idle) && Display.isIn(Empty))
        return true;                // check final state
      else                          // and postcondition
        return false;
    }
    return false;
  }

  public boolean startTest12 () {   // TEST 1.2
    Dispenser.setTo (Item2, empty); // set precondition
    TVM.setTo(idle);                // set init state
    try {                           // expect exception
      SelectItem (Item2);           // call transaction
    } catch (DispenserItemEmptyException e) {
      if (TVM.isIn(idle) && Display.isIn(Empty))
        return true;                // check final sate
      else                          // and postcondition
        return false;
    }
```

```
      return false;
    }

    public boolean startTest13 () {   // TEST 1.3
      Dispenser.setTo (Item2, empty); // set precondition
      TVM.setTo(idle);                // set init state
      try {                           // expect exception
        SelectItem (Item2);           // call transaction
      } catch (DispenserItemEmptyException e) {
        if (TVM.isIn(idle) && Display.isIn(Empty))
          return true;                // check final state
        else                          // and postcondition
          return false;
      }
      return false;
    }
    ...
    public boolean startTest31 () {    // TEST 3.1
      Dispenser.setTo(Item1, notEmpty);// set precondition
      TVM.setTo(idle);                 // set init state
      TVM.insertCoin (0.1);            // call transaction
      if (TVM.isIn(insertCoins) &&     // check final state
          Display.isIn(0.1))           // and postcondition
        return true;
      else
        return false;
    }
    ...
  }
```

If our implementation technology is restricted to a single programming language environment, we are actually done with the embodiment step when we have implemented the testing interface and the tester component according to each identified component contract in our component framework. The remaining work is to then integrate our individual components according to the mechanisms of the programming language.

Devising a source code implementation is always the first step if our deployment environment is a component platform or some middleware. If we target a component platform as the final embodiment infrastructure, we have to add platform-specific code to our source code development that is based on a single language. The component platform provides a binding to a particular language. This binding is similar to invoking library operations that the language provides. I will give some more details on how built-in contract testing may be realized on these component platforms in the subsequent sections.

Table 5.1. Behavioral tests for the *VendingMachine* component

No	Initial State	Precondition	Transition	Postcondition	Final State
1.1	idle	Item1 ==Empty	SelectItem (Item1)	Display (Empty)	idle
1.2	idle	Item2 ==Empty	SelectItem (Item2)	Display (Empty)	idle
1.3	idle	Item3 ==Empty	SelectItem (Item3)	Display (Empty)	idle
1.4	idle	Item4 ==Empty	SelectItem (Item4)	Display (Empty)	idle
1.5	idle	Item5 ==Empty	SelectItem (Item5)	Display (Empty)	idle
...	...	...	...	...	...
2.1	idle	Item1 !=Empty	SelectItem (Item1)	Display (Item1.Price)	idle
2.2	idle	Item2 !=Empty	SelectItem (Item2)	Display (Item2.Price)	idle
2.3	idle	Item3 !=Empty	SelectItem (Item3)	Display (Item3.Price)	idle
2.4	idle	Item4 !=Empty	SelectItem (Item4)	Display (Item4.Price)	idle
2.5	idle	Item5 !=Empty	SelectItem (Item5)	Display (Item5.Price)	idle
...	...	...	...	...	...
3.1	idle		InsertCoin (10ct)	Display (0.10)	insert Coins
3.2	idle		InsertCoin (20ct)	Display (0.20)	insert Coins
3.3	idle		InsertCoin (50ct)	Display (0.50)	insert Coins
3.4	idle		InsertCoin (1EUR)	Display (1.00)	insert Coins
3.5	idle		InsertCoin (2EUR)	Display (2.00)	insert Coins
4.1	Perform tests 6.1 to 6.3				
4.2	insertCoins		abort ()	CashUnit.dispense () == 0.80EUR	idle
5.1	Perform tests 6.1 to 6.3 and wait for some time				
5.2	insertCoins		Timeout ()	CashUnit.dispense () == 0.80EUR	idle
6.1	Perform test 3.1				
6.2	insertCoins		InsertCoin (20ct)	Display(0.30)	insert Coins
6.3	insertCoins		InsertCoin (50ct)	Display(0.80)	insert Coins
7.1	Perform test 3.1				
7.2	insertCoins	0.10 < Item1.Price	SelectItem (Item1)	Display(0.10)	insert Coins
...	...	...	...	...	...

Before that, we will have a brief look at how third-party components for which we do not own the source code can be augmented with built-in contract testing functionality. In the previous paragraphs I have concentrated mainly on how we can add testability features to our own classes for which the code is readily available. In the next paragraphs I will introduce a way to augment existing third-party Java classes with testing interfaces according to the built-in contract testing philosophy. This can be done through the BIT/J Library.

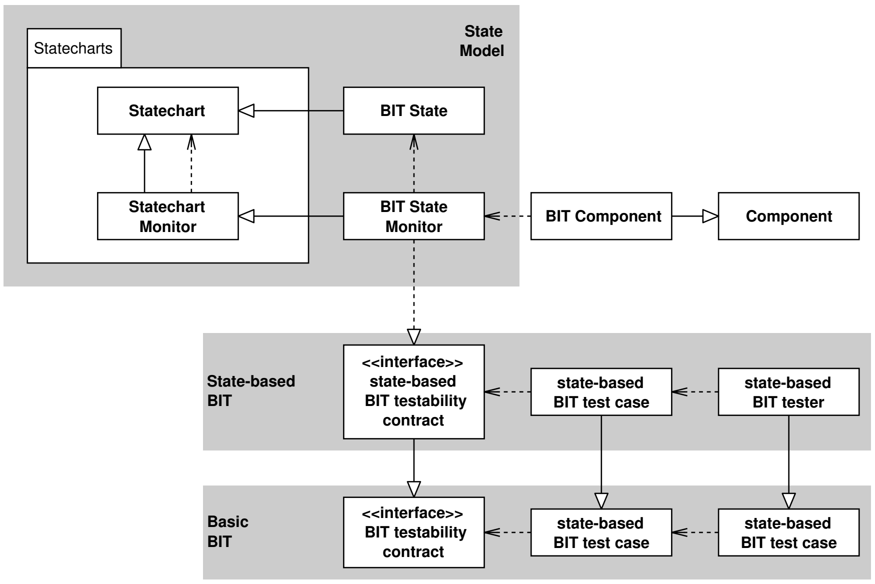

State Model
Statecharts
Statechart
BIT State
Statechart Monitor
BIT State Monitor
BIT Component
Component
State-based BIT
<<interface>> state-based BIT testability contract
state-based BIT test case
state-based BIT tester
Basic BIT
<<interface>> BIT testability contract
state-based BIT test case
state-based BIT test case

Fig. 5.10. Structural model of the BIT/J Library from the University of Pau, Laboratory of Computer Science (LIUPPA). BIT stands for Built-In Testing

BIT/J Library Java Support Framework for Built-in Contract Testing

Commercial third-party (off-the-shelf) components (COTS) cannot typically be augmented with additional built-in contract testing interfaces that provide a client of a server component with introspection capabilities for improved testability and observability [74]. Unless COTS vendors follow the built-in contract testing philosophy and incorporate testing interfaces into their products right from the beginning, such components can be tested only through their normal provided interfaces. Basically, this comes down to the traditional way of testing objects or components.

However, modern object-oriented implementation technologies such as Java do provide mechanisms that enable internal access to an encapsulated third-party component at a binary level. The BIT/J Library and tool suite developed by members of the Laboratory of Computer Science at the University of Pau, France (LIUPPA), is one such instance that is capable of implementing external access to existing third-party Java components. It is a free tool that has been developed as part of the Component+ project [38], and it is available through the LIUPPA Web site [13]. The BIT/J Library is based on the idea of incorporating the behavioral model of a third-party component, as it is defined through the specification of the component, directly into that component. It uses mainly the Java reflection mechanism to gain access to and retrieve internal information from a COTS component. Figure 5.10 displays the structural model of the BIT/J Library. Contract testing based on this library is initially concerned only with `BIT testability contract`, `BIT test case`, and `BIT tester` indicated through the shaded box labeled "Basic BIT" in Fig. 5.10. These are the three most fundamental extensions that any testing environment using BIT/J will require. The `BIT testability contract` represents the initial testing interface that copes with the assessment of results, execution environment, and faults. It is fully specified in [7]. The `BIT tester` comprises `BIT test case`s that access this interface for retrieving testability information from a component. State-based testing according to a component's behavioral model is added through their respective state-based versions indicated through the shaded box that is labeled "State-based BIT" in Fig. 5.10. These concepts add state-based testing interfaces as they are required for state information setup and retrieving according to the definition of the built-in contract testing technology, and an execution environment for the specified state model of a component, the `BIT state monitor`. The state model implementation is based on Harel's statecharts formalism [85] that is also adopted by the OMG in UML state diagrams. The statechart runtime environment is added through classes in the shaded box called "State Model" in Fig. 5.10. BIT/J essentially adds an executable state machine plus a number of access interfaces to a Java component. Such an augmented component is termed `BIT component` in Fig. 5.10. A more detailed specification of BIT/J is available through [7, 13]. So far, this type of built-in contract testing support library is available only for Java components. Other implementation technologies do not yet provide the proper support for realizing similar access mechanisms.

5.3 Component Technologies

Contemporary component technologies provide many modern concepts and tools to "glue" components together. They are sometimes referred to as object request brokerage (ORB) platforms, component brokerage platforms, or middleware. Szyperski calls these wiring standards [157] because they essentially provide all the connection mechanisms for the diverse modules that make up an application and that have been written in different programming languages and are available in various binary formats. Middleware platforms can be seen as a mediator between the various components on a single network node and remote components on a different node. Supporting interoperability between components that are physically residing on different networked computers is also an important service that middleware platforms realize. The principal organization of a component platform is depicted in Fig. 5.11. These technologies are called middleware because they have not yet found their way into the operating system, although they provide typical operating system services. They reside somewhere in between the services of the operating system and the applications that use the middleware services. A middleware platform may be compared with a traditional Unix command shell that draws together the components of a platform, the programs in the Unix toolbox, through a standardized mechanism. On a Unix platform this it is the pipe mechanism, a very simple yet powerful and effective component wiring standard. And it even provides ways of accessing remote computers.

Remote procedure calls (RPC) [39] can be regarded as an early attempt of the late 1980s or early 1990s, to realize such an open communication platform at a programming language level, and it is the ancestor of all modern component platforms. Many component platforms still bear many of the features from the early days.

The primary component technology standards come from Microsoft with its COM, OLE, ActiveX, COM+, and .NET technologies, Sun Microsystems with Java and JavaBeans, and the large industry consortium, the Object Management Group (OMG) with its CORBA standard. In the following subsections we will have a look at the features of some of these wiring standards, and then see how they affect the implementation of built-in contract testing.

5.3.1 JavaBeans and Enterprise JavaBeans

JavaBeans denotes the Sun/JavaSoft basic component model for Java. A Java component is called a "bean." Their primary motivation for developing this component technology was that Java applications, once written, should be able to be integrated and run, and thus reused, on any arbitrary execution platform. Additionally, the technology puts a strong emphasis on the fact that components once released should never be touched again, or, in other words, there is no required Java code-level programming effort involved in getting two alien beans from different vendors to interact with each other. This is probably the most important advantage of the JavaBeans component technology over pure Java. The JavaBeans specification defines a JavaBean as a reusable software component that can be manipulated visually through specific builder tools [115]. The main aspects of JavaBeans are summarized as follows [158]:

- Properties that a bean provides for customization. These can be accessed and changed through pairs of so-called setter and getter methods.

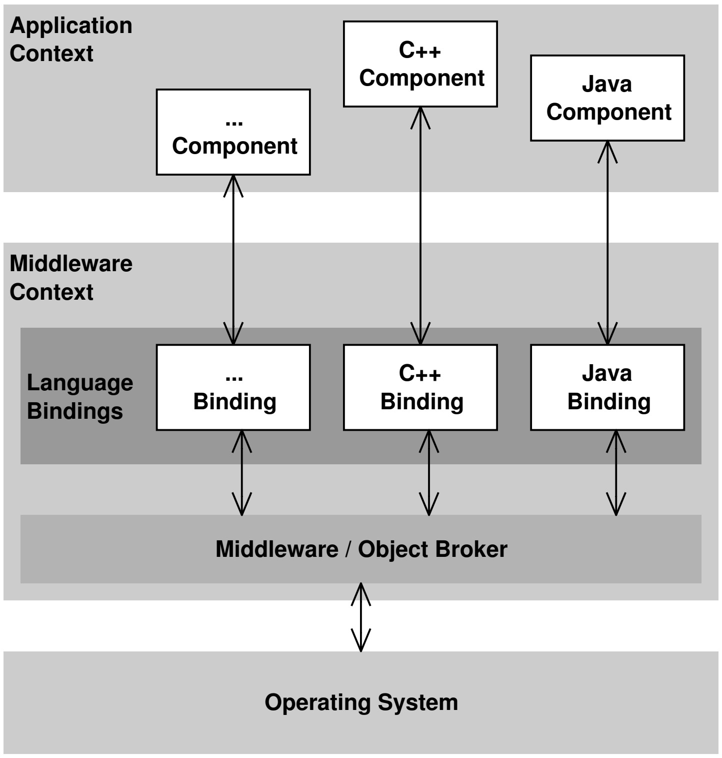


Fig. 5.11. Coarse-grained organization of a middleware platform

- Introspection with respect to properties, events, attributes, and methods that a bean provides. This is based on Java's reflection mechanism.
- Events that beans announce as "sources" and acquire as "sinks." An assembly tool can connect sources of one component with the respective sinks of other components, and vice versa.
- Persistence determines how the internal state information of a bean is stored and retrieved. By default this is controlled by the so-called automatic serialization mechanism of the Java language.

In addition to the main aspects of the JavaBeans component platform there are a number of advanced specifications:

- The Java containment protocol permits containment and nesting within the otherwise flat component hierarchy of JavaBeans. This allows an application to follow exactly the containment model of a development method.
- The Java service protocol enables top-level beans or containers to extend services to their contained subcomponents. This is required for handling the containment protocol.

- The Java activation framework (JAF) represents a registry for components that can be used in order to locate, load, and activate components that provide a particular operation over a particular type of data [158].
- The archiving model or long-term persistence model [158] represents an alternative to the serialization mechanism that uses XML or proprietary Java file formats. Archiving is limited to the part of a component's state that is accessible through its interface.
- The InfoBus specification represents a generic framework for component composition in a very particular fashion.

Enterprise JavaBeans (EJB) adds the distribution aspect to the suite of Java component models (Applets, Servlets, JavaBeans, EJB, Application Client Component). JavaBeans and the other component models concentrate more on the individual components and how they can be inspected and made to work together effectively through a specific wiring. Szyperski calls what JavaBeans provides "connection-oriented programming" [158].

EJB follows an entirely different path. EJB mainly provides an application server environment (e.g., JBOSS) and defines how components have to be designed and written by developers to be integrated in an EJB implementation [43]. Every EJB component comes equipped with a deployment descriptor that specifies how a component should be deployed in a specific context. This supports contextual composition that also other more advanced component platforms, such as COM+ or the CORBA Component Model (CCM), provide. EJB is concerned mainly with how individual component instances may be automatically connected to appropriate services and resources, i.e., through a so-called EJB Container. This container effectively wraps around its contained component instances and intercepts all communication that is going in and out. JavaBeans and EJB are fully specified in [114, 115].

5.3.2 COM, DCOM, ActiveX, COM+, and .NET

Microsoft's Component Object Model (COM) is the company's initial component "wiring" standard. COM is a binary standard that is based entirely on interfaces, so it does not even specify how a particular programming language relates to it. COM interfaces are organized in a subtle way and the platform provides mechanisms to find out about individual components' interfaces through `IUnknown`, a default interface that every COM component needs to provide. This comprises some default methods, such as `QueryInterface`, which identifies COM objects according to their provided interface references, and `AddRef` and `Release`, which are used to manage (add and release) interface references during an object's lifetime.

The Distributed Component Object Model (DCOM), sometimes also referred to as COM+ in later versions, represents the distributed extension of Microsoft's COM architecture. DCOM permits a client application to invoke a remote DCOM server object. It is designed specifically to perform within a network infrastructure made up of MS Windows nodes, and is also part of the Windows 2000 platform. DCOM supports transparent communication across not only process boundaries but also across platform or machine boundaries through the creation of client-side proxies and server-side stubs that hide the underlying networking infrastructure.

ActiveX-controls represent an advancement of Microsoft's original Object Linking and Embedding (OLE) standard. Both are inherently based on COM. In fact, ActiveX-controls are merely COM objects that are supported through a very special server and provide enhancements that are specifically designed to facilitate the distribution of components over slow network connections.

COM+ is an extension of COM and its successor. It integrates previously separate, somewhat colliding support technologies such as transactional processing, asynchronous messaging, load balancing, and clustering [158].

.NET aims at bringing the various different Microsoft standards, products, and services together into a single framework. In addition to other things, this includes Web services, as well as development and deployment platforms. The concrete services and facilities that the .NET framework offers are fully outlined by Szyperski [158].

5.3.3 CORBA, OMA and CCM

The Common Request Broker Architecture (CORBA) represents the OMG's initial attempt to make distributed components that are implemented in different languages interact with each other seamlessly. An initial version, 1.0, released in 1991, provides all the basic ingredients to achieve this goal, such as the CORBA Object Model, the Interface Definition Language (IDL), a core set of application programming interfaces (APIs) for dynamic request management and invocation, and an Interface Repository [78]. The subsequent version, 2.0, adds a number of extensions to the original version, such as OLE/COM compatibility, security and transaction services, and some datatype extensions. Other later versions provide programming language mappings for C++, Smalltalk, COBOL, Ada, and Java, as well as support for real-time features, fault tolerance, and common security.

CORBA is carefully standardized to allow for many different implementations and platforms [158]. A CORBA system essentially consists of three parts:

- The object request broker (ORB). The broker is responsible for providing the available object interfaces, receiving requests from clients, finding object implementations that may be able to process the request, and performing all transformations with the data in the request.
- A set of invocation interfaces. Invocation interfaces are the primary way through which client objects access the services that the ORB provides.
- A set of object adapters. Adapters are the primary way through which server object implementations access the services of the ORB.

An initial requirement for a CORBA system to work is that all object interfaces that the ORB provides are specified in a common language, the OMG interface definition language (OMG IDL) [54, 82]. Additionally, all languages that use the ORB need to be transferred into the format of OMG IDL. In other words, every supported language must have a so-called binding to the OMG IDL. Once an interface is expressed in IDL it can be compiled and deposited in the interface repository of the ORB. The OMG IDL compiler can also be used to generate stubs and skeletons for interfaces. A stub will be instantiated and installed at the client side of an ORB service, and it emulates the required server object for the local client. In reality, the stub will forward all invocations of the client through the ORB to the respective real target object. Stubs are often called client-side proxies [158]. A skeleton is installed at the side of the real object. It translates invocations coming from the ORB into a form that the server object can "understand." Figure 5.12 illustrates this organization.

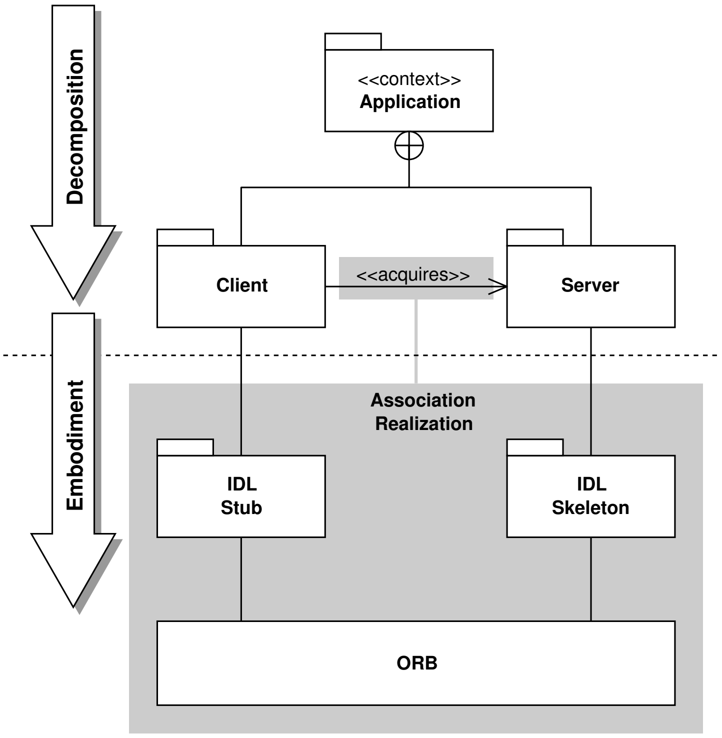


Fig. 5.12. Realization of a client/server interaction through an ORB

In order to broaden the applicability of CORBA, and to make remote object invocation more generic, the OMG has extended the CORBA standard and called it the Object Management Architecture (OMA). It embodies the OMG's vision for the entire component software environment. The OMA adds a number of items to the CORBA standard [80, 158]:

- A set of common object service specifications, the so-called CORBA Services. They standardize the life-cycle management of objects and provide operations for creation and access control of objects and their relationships to other objects. CORBA Services provide the generic environment in which objects can perform.
- A set of common facility specifications, the so-called CORBA Facilities. These provide computing solutions for typical business problems in a specific domain such as healthcare, manufacturing, finance, etc. [80].
- A set of application object specifications. These are application objects that perform specific tasks for the user. The application objects can be built relatively easily through modification of existing classes by generalization or specialization, e.g. from CORBA Services.
- The CORBA Component Model (CCM), also called CORBA Components. This represents the main contribution of the latest incarnation of the CORBA standards, CORBA 3 [158].

The CCM can be seen as the first non-proprietary, language and platform-independent component architecture [21]. It introduces a number of new features into the CORBA standards suite such as the portable object adapter that mediates between ORB and object implementation [158], and CCM components with very special features, CCM containers that contain components, and it supports EJB compliance. In other words, existing EJB solutions can be embedded in the CCM.

5.3.4 Component Technologies and Built-in Contract Testing

Component technologies are primarily used to interconnect component implementations that are written in different languages or are residing on different nodes of a network. So, each component will have to abide by the protocol of the component platform according to the binding of the component's implementation notation. In other words, each component implementation that will be deployed on a particular platform must be augmented with the right mechanisms so that the platform can handle the component and connect it to other such components. The so-called business logic of an application must thus be augmented with the infrastructure of the middleware platform. This is similar with what a linker in a typical programming environment does. It adds the infrastructure of the underlying runtime system to the original object code of a program and turns it into an executable representation for this particular runtime system.

In order to deploy a component on a middleware platform, we first have to add code artifacts that realize the communication between our component and the middleware, and, second, have to add the respective runtime system. If we are using a middleware platform, the deployed component is not entirely detached from the underlying runtime environment, and is not fully encapsulated by the middleware. Basically, constructing a component for some middleware can be seen as adding support libraries to an application, pretty much in the same way as any other support library that a programming language may be providing. The artifacts that we have to add to make a component implementation middleware compliant is usually some stub and client code for the client and server side proxies, i.e., defined through some interface definition language (IDL) and often generated automatically, and some operations to register with the object request broker and perform some other administrative tasks such as object construction and destruction. Figure 5.13 illustrates how a client/server association is realized with a Java ORB.

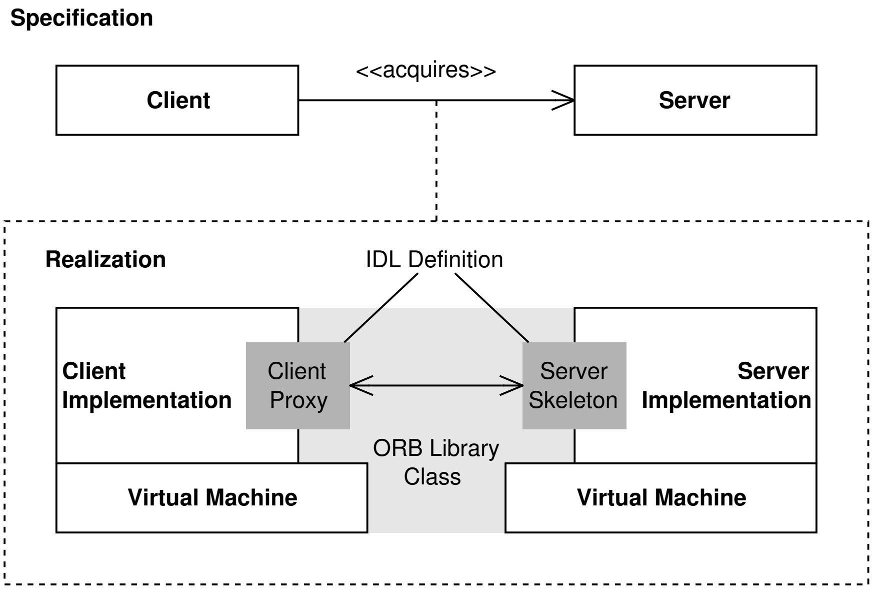


Fig. 5.13. Client-server association in a Java ORB

Since built-in contract testing is part of the components that are deployed on a middleware platform, there is no particular way of dealing with the contract testing artifacts. The testing interface provides an additional access mechanism to the component, and this is treated in the same fundamental way as all the other interface operations. The tester component is simply an add-on to the testing client. Both are realized through some implementation notation, e.g., in Java, and then augmented with the middleware platform code.

The only issue that we have to consider is the degree of flexibility of the contract testing artifacts with respect to the selected middleware platform. One of the final steps during embodiment is to distinguish between the logical components according to the containment model and the physical components that will be deployed on the middleware platform. Contract testing that will be directly built into the physical components cannot be removed later. Only built-in contract testing that will be organized at the physical component level can be allocated and deallocated within the component framework. So, in order to fully exploit the principles of built-in contract testing that I have introduced in Chap. 4, we have to look at whether and how component extension and inheritance, and dynamic component allocation and deallocation, are realized through a middleware platform. How component features may be inherited is important for how testing interfaces are implemented on a particular platform. How components are dynamically allocated and deallocated is important for how tester components are organized on a platform. We have to distinguish between things that are built-in at the programming language-level and things that are built-in at the component platform level. The following list summarizes these considerations:

- Both testing interface and tester component are permanently built-in at the programming language level. The built-in contract testing artifacts are hardcoded into the physical components. This form of built-in contract testing has no effect on the component platform, and it represents the most fundamental type of built-in contract testing. Additionally, we could built a configuration interface into the tested object that may be used to switch contract testing on and off at runtime. For example, this could be a different object constructor specifically for the testing artifacts.
- The tester component is dynamically allocated and deallocated at the component platform level. So, in addition to the functional components we will also have to deploy tester components on the middleware platform. These are treated in exactly the same way as the functional components, although the functional components that acquire tester components will need additional infrastructure to explicitly allocate the tester components at the platform level, i.e., explicit external interface definitions for the tester components. Since dynamic allocation and deallocation is what component platforms have initially been made for, this organization will work for all contemporary component platforms, although it will likely lead to flat component hierarchies at the component platform level. This is because not all middleware platforms support component hierarchies in the way in which KobrA containment trees define them.
- Dealing with the testing interface at the component platform level requires that the middleware supports an implementation extension or inheritance mechanism that actually works on the binary component, or at least at the object code level, e.g., a Java class.

The question that we have to answer in an embodiment step toward a middleware platform is whether the contract testing artifacts will be built directly into the functional physical components in the sense of built-in contract testing, or whether they will be available as separate components at the middleware platform level. In the first case, built-in testing will be available at all times during deployment and runtime of a binary component; in the second case it can be added or removed according to the requirements at hand, so it will provide the same flexibility at the binary component level that may be defined in the model or at the programming language level. Interface inheritance and dynamic component allocation and deallocation works for most contemporary component platforms, such as CORBA, CCM, COM, DCOM, ActiveX, Java Beans, and EJB. CORBA and DCOM do not allow implementation inheritance, but DCOM can achieve something similar to it through component aggregation and composition. Inheritance at a binary level on a readily compiled and linked component is not feasible because the binary level is lacking the programming language infrastructure that effectively carries out the inheritance.

So, for pure binary components (most off-the-shelf components will probably be available only in binary form for a particular middleware platform) we have to take another route to achieve high flexibility with respect to built-in contract testing. The easiest way to achieve this flexibility is to have two binary components ready for deployment: one with all the built-in contract testing artifacts permanently built-in (and this can be used for initial deployment and component integration testing) and one without the built-in testing artifacts that will be deployed in the final system. The contemporary component technologies are still some way from realizing flexible component embodiment in the form in which it was described earlier. We can observe some advances that aim at solving these problems, for example, as put forward through the CORBA Component Model. However, having similar mechanism that modern languages are readily providing in a component platform would really be most desirable.

In the next section we will take a look at how Web services, another form of a component broker platform, go together with built-in testing.

5.4 Built-in Contract Testing and Web Services

Web services represent a relatively new technology for the implementation of distributed component-based applications. They are commercial software applications that are executed on Internet hosts and provide individual services which are used to realize distributed component-based systems. These services are typically specified based on the Extensible Markup Language (XML) and they communicate with their clients through Internet protocols that also support the XML [43]. Web services fulfill all the requirements of Szyperski's component definition [157], that is, a service is described and used only based on interface descriptions, and, more importantly, is independently deployable. This means a Web-Service provides its own runtime environment so that a component-based application is not bound to a specific platform. Every part of such an application is entirely independent from any other part, and there is no overall runtime support system but the underlying network infrastructure. Web services represent the ultimate means of implementing component-based systems.

The fundamental idea behind the so-called service-oriented programming is that individual parts of an application communicate on the basis of a predetermined XML contract. This is not different from the way we have so far treated components. However, here the components of an application are not bound to a particular host, as in object-oriented programming, but are established dynamically throughout entire networks, for example, the Internet. This way, different implementations of a distinct service may be easily replaced by registering with a different Web service that provides the same specification of the required component or, in other words, the same interface. Architectures for service-oriented programming typically support the following concepts:

- Contract. This is the full specification of one or more interfaces that characterizes the syntax and semantics of a service (functional and behavioral specification).
- Component. This represents a readily usable and deployable object that provides functionality and exhibits behavior. This is the realization of the actual component that implements the functionality of the service.
- Connector, Container, and Context. These concepts realize the networking and runtime elements of a Web service. This means that they are responsible for establishing the connection between client and server, taking of the execution of an instance, and controlling its security.

Web services are in fact component platforms that communicate and establish the component interactions at a higher level of abstraction. They can be seen as extensions to existing component platforms. Examples of Web service architectures are Sun's Java 2 Enterprise Edition (J2EE) [156] and Microsoft's .NET architecture [113]. The core for both these systems is represented by a Web-based application server that essentially performs the same task as an object broker for typical middleware platforms. I am not going to present more details on how Web service architectures should be realized; that is clearly outside the scope of this book. In the following subsections, we will look at typical scenarios in which contract testing can be applied in the context of Web services.

5.4.1 Checking Web Services Through Contract Testing

Contract testing provides the ideal technique for checking dynamic and distributed component-based systems that are based on Web services. This is in fact the scenario for which built-in contract testing provides the most benefits. The syntactic compatibility between a client and a Web-based server is ensured through the XML mapping between their interfaces. Their semantic compatibility can be checked through the built-in server tester components inside the client that are executed to validate the associated server. These tests can be performed when the client is registering with a service for the first time, during configuration, or if the client requests the same specification of the server from a different Web service provider, during reconfiguration of the system.

Figure 5.14 displays the containment hierarchy of an example banking system that is based on a Web service. Here, I use the bank context as an example because embedded systems such as the vending machine and the resource information network are not typically realized through Web-based services, although this may be perceivable. Web services are actually more suitable for typical business domain systems such as systems for banking, insurance, and general administration and for Internet information applications.

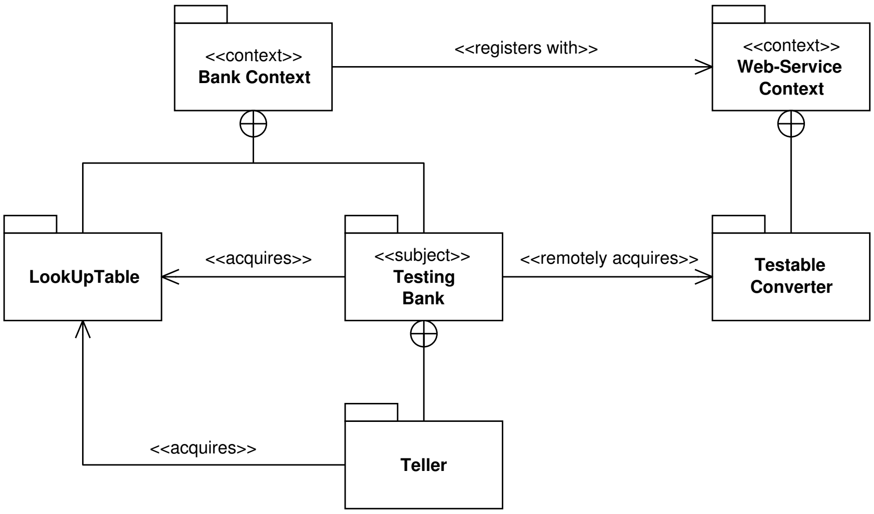


Fig. 5.14. Containment hierarchy of an example banking application that is based on Web services

The `TestableConverter` component on the right hand side of Fig. 5.14 represents the currency exchange rates converter of a distributed banking application, and it exports currency conversion operations. This component may be provided by a third party, an external Web service provider who specializes in selling banking services, and it may be updated on a daily basis according to the stock market exchange rates. The banking system connects to a new instance of the converter once a day at a given time, so that it always keeps the latest currency exchange rates in store for the respective modules of the banking application. The ≪remotely acquires≫ relationship indicates that the converter, in this case a testable converter, is not locally available. This means that the relationship will be implemented through some underlying networking or Web Service infrastructure. This infrastructure is realized through the connector and the container on the server side (the Web service) and a Web service-compliant implementation on the client side of the ≪remotely acquires≫ relationship. The stereotype ≪remotely acquires≫ hides the underlying complexity of the network implementation and considers only the level of abstraction that is important for testing. This representation format is termed "stratification" [5].

As soon as the connection between the two interacting components is established, however this is realized in practice, a normal contract test may be initiated. The server `TestableConverter` provides a suitable testing interface that the client's built-in tests can use. Client and server do not "know" that they communicate through Web interfaces. This connection is established through their respective contexts when the context of the `TestingBank` component registers with the context of the `TestableConverter`.

5.4.2 Testing of Readily Initialized Server Components

Web services typically provide instances that are ready to use. It means that the server component provided through the Internet service is already configured and set to a distinct required state. A runtime test is therefore likely to change or destroy the server's initial configuration, so that it may not be usable by the client any more. For example, a test suite for the `TestableConverter` component that is built into the `TestingBank` component in Fig. 5.14 may comprise test cases that change, add, or remove some of the exchange rates stored in that component, if this is possible. Clearly, for the client such a changed server is of no use and creates a fundamental dilemma for built-in contract testing of Web services.

Possible Destruction of the Server through the Testing Client

Under object-oriented runtime systems the client can solve this dilemma by asking the runtime system to create a clone of the tested component and to pass the clone's reference to the test software. This works because client and server are handled by the same runtime environment. For example, in Java this duplication is performed through the `Object.clone()` method. So, any Java object can be duplicated in that way. In this case the test software may completely mess up the newly created clone without any effect on the original instance; it is thrown away after the test, and the original is used as a working server.

However, in a Web service context, the runtime system of the client is different from that of the server so that the client cannot construct a new instance from an existing one. The client and server are residing within completely different runtime scopes on completely different network nodes. For example, the banking application from Fig. 5.14 may be based on Java, and the Web service component may be based on a Cobol runtime environment. This sounds a bit odd, but it happens in reality. In other words, only the Web service context may generate an instance of that Cobol component because it comprises a Cobol runtime environment. Contract testing can therefore be applied only in a Web service context if the Web service provides some way for the client to have a clone created and accessed for testing. Client and server, or the client's and the server's contexts, need to be made aware of this issue and provide additional access services accordingly to be able to cope with this. Some contemporary component technologies such as CORBA Components (CCM) are capable of doing exactly that. There, the container provides operations that generate exact copies of existing instances and make them available to their clients. In practice this will be initiated by the context of the client that requests the Web service to generate two instances of a server.

Possible Destruction of the Testing Client through the Server

A similar problem appears on non-networked platforms when the test software discovers a fatal failure that completely hangs the runtime system. During the integration phase of an application this is not a problem. However, during a reconfiguration of an operational system, however, this should not happen. The application should ideally reject an unsafe service and continue to operate with the existing configuration. The contract test should therefore be executed within its own thread to rule out any side effects.

A tested server component may acquire and lock a resource and then fail its test. The testing system that may be relying on this locked resource will not any more be able to commence its original service. This is a typical problem in distributed embedded applications, but it can be circumvented by another built-in testing technology: built-in quality-of-service testing. I will describe this technique in Chap. 7, where we will have a closer look at how quality-of-service in component contracts can be addressed.

For Web services, these problems are not an issue since the individual components are executed in different threads on different nodes. In this case a contract is always safe, and it will never fail the client application that is applying it, even if the test fails.

Now that we have discussed the primary implementation platforms, or wiring standards as Szyperski chooses to call them [158], for component-based development in general, we will have a look at specific implementation technologies for testing. These technologies are mainly geared toward the development of the client side in-built contract testing, the tester component, and its built-in test cases.

5.5 Implementation Technologies for Built-in Contract Testing

Built-in contract testing as I have introduced it thus far can be seen primarily as a method for facilitating the testing of component-based systems during development and deployment. The method provides a number of concepts and artifacts, and guidelines for applying these, and it can supplement and extend existing development methods such as the KobrA method in a natural way. How the principles of the technology are implemented in real development projects depends considerably on the type of a project, such as whether we are dealing with development for reuse or development with reuse, or, in other words, whether we take on the role of a component provider or a component integrator. Most projects will have to deal with both roles, because even if we are only reusing and integrating existing building blocks, we will inevitably have to develop the adapters that realize the connection between the diverse third-party units that we are integrating.

Another important issue in this respect is which implementation technologies are used; are we dealing with a specific programming language, or a distinct component platform? In the previous chapters and sections I have tried to shed some light upon all these aspects. The most basic and fundamental implementation technology will typically be a programming language. On top of that come some component platforms that integrate the binary implementations of the programming languages and make them interact with each other. These platforms can be seen as containers that organize the binary formats and support their execution and interactions. Szyperski calls this development beyond object-oriented programming [157], because these technologies are capable of making quite diverse implementations understand and interact with each other. Some of these technologies will already provide networking infrastructure to support distributed systems; for others we may have to add this in a separate development effort.

So, all the tools that we may employ to support built-in contract testing in one way or another are heavily dependent on the context of a project, or even on the entire development culture in an organization. Because there are so many implementation technologies, and they are swiftly changing and new ones emerging, I can give directions only on how the principles of built-in testing may be realized. Personally, I am a strong advocate of sound methodological support above all technologies, because technologies die quickly. Nevertheless, in the following subsections we will have a look at how well-known tools and implementation technologies, the XUnit framework above all, and the *Testing and Test Control Notation*, an emerging technology from the telecom domain, can be used to support built-in contract testing.

5.5.1 The XUnit Testing Framework

XUnit represents a framework for writing, applying and repeating unit tests. A unit test can be seen as the initial validation effort that is carried out to uncover errors and problems in a single unit in isolation. This means that the testing is performed according to the specification of the unit [66], and not according to the specification of its encapsulating context. While built-in contract testing concentrates on assessing a component in its context according to the specification of that context, a unit test assesses a component in terms of its own specification, free of any context into which it will be eventually integrated.

Although it seems initially that a unit testing framework is not suitable for supporting or implementing built-in contract testing, giving it a second thought we might well use this tool for our purpose. As I said earlier, the XUnit framework is a tool for writing and repeating tests. The fact that these tests may be unit tests, and they are designed as such, is only a question of the viewpoint or the criteria that we apply for our testing. According to which testing criteria these tests will be devised is not relevant for the tool or the framework. The fact that we apply unit tests is accounted for only through the basis upon which we develop these tests. If we derive the tests that fill the XUnit framework with life from the specification of the unit, we will end up with unit tests. If we derive these tests from the specification of the integrating context of the unit, we will end up with typical contract tests. The tool as such has no bearing on these criteria.

The XUnit testing framework in general and the JUnit testing framework in particular are based on the ideas of the Extreme Programming community [9, 102]. Essentially, these ideas state that before any code is written, the developer should have defined the test cases for the code, or, in a nutshell: test first! In my opinion this is a great idea and it probably took me most of Chaps. 2, 3, and 4 to propagate and communicate that. So, “testing first” is in line with the fundamental principles of built-in contract testing. However, I would rather prefer to say “plan the tests, or design the tests first,” because “testing” still bears a strong focus on test execution, and this is in fact how the Extreme Programming community sees it, a permanent testing and test execution effort during unit development. Their proposed development activities follow a cyclic procedure for developing a basic function, testing that basic functionality, developing another basic function, testing it, and so on. This process concentrates more on custom development and component-based development from the vendor’s viewpoint, and on “programming in-the-small.” We are more concerned with integrating existing units and making them interact, which could be more specifically referred to as “programming in-the-large.”

In general, the JUnit testing framework is for the tester component what the BIT/J library (Fig. 5.10) is for the testable component. Both are Java libraries that readily provide testing and testability concepts for Java classes.

Under the XUnit testing framework the test cases can be implemented through JUnit or one of its derivatives, according to which programming language is used, and applied to the final code. JUnit was written by Erich Gamma and Kent Beck [10, 59], and there already exist unit test frameworks for a number of different programming languages such as JUnit [62] for Java, CUnit or CppUnit for C and C++ [51, 61], PerlUnit for the Perl language [52], and many others. A summary can be found in [63]. Here, I will introduce the concepts of JUnit briefly, and then discuss how this may help support built-in contract testing. The concepts of JUnit are very similar to what the UML Testing Profile is actually defining, we have:

- A test case, a Java class that manages a single test case, and from which test case features can be inherited. Each test is initially separate from the others (unless a so-called test fixture is used), so it creates all the required test data for a test through the operation `setUp()`, and removes all test data after the test through the operation `tearDown()`. The test case defines its own input, event, action, and response in terms of pre and postconditions and expected results. These are concepts that the UML Testing Profile is also readily supporting.
- A test suite, a collection of test cases that the UML Testing Profile refers to as test component.
- A test result, the outcome of a single test or a test suite. The testing profile calls this test observation. The test observation is checked through the `assertTrue()` operation from the class `Assert` that will return a pass or a fail according to the result of the assertion.

In the next section we will see how JUnit can be used to implement tester components according to the principles of built-in contract testing.

5.5.2 JUnit and Built-in Contract Testing

If we use JUnit to implement a tester component, for example, the `Vending MachineTester`, the implementation in Java is not really much different from the Java implementation in Sect. 5.2. The JUnit framework readily provides support for typical testing concepts, so that many, or at least some, of the testing infrastructural code artifacts may be omitted. JUnit provides simple and ready-to-use solutions for these. The tester component will likely be smaller and structured more clearly. JUnit only supports the development of built-in contract tester components, and not testing interfaces for the testable components. So, in order to exploit the full range of built-in testability features, all the components will have to provide testing interfaces according to the built-in contract testing philosophy, or alternatively the BIT/J Library may be used.

The tester components will import the JUnit features and apply them. The configuration code that realizes the acquisition of the testable components is the same as for the previous Java implementation of the `VendingMachine`

`Tester` component in Sect. 5.2. These are in fact typical code segments for setting up the built-in contract testing infrastructure according to the model that may be found in Fig. 5.9. The public operations in the following Java source code example, `setTestableDispenser`, `setTestableDisplay`, and `setTestableVendingMachine`, realize the tester component's configuration interface. Its superordinate component can use these to connect the tester components with their associated server components. In this case, this superordinate component is the context of the system, because the vending machine component is the top-level component in our system's hierarchy. So, the tester component can be controlled by a superordinate component in any component hierarchy, and we may see this as performing a built-in contract test. Otherwise, as the case here, we invoke the tester directly as a stand-alone program. The JUnit framework provides the optimal support for this flexibility according to both these approaches. The `main` method, illustrated in the following source code example, invokes JUnit's built-in test runner that executes all defined test cases. In other words, the `VendingMachineTester`'s `main` method represents the context of the testing system, because this can be used to realize the associations between all components for a test, and starts the performance of the test.

Deviations from the defined assertions in each test case will be announced through thrown exceptions. They are typically caught and processed by JUnit's test runner operations, either in textual form or through a graphical user interface. If the tester component is invoked by another testing component deeper in the containment hierarchy, the exceptions indicate that there is something wrong with the tested server component. The testing component is then responsible for test execution and the analysis of the test results to take appropriate action. So, we can still perform a unit test according to JUnit's initial concepts if we devise the tests based on the unit's specification. But we can also perform a built-in contract test if the JUnit tester component is incorporated into the system, and invoked and controlled by a client component. These test cases will have been developed based on the specification of the client that owns the JUnit tester component.

The JUnit framework represents a convenient way for defining and executing unit tests as well as contract tests. The framework is designed in such an open way that both approaches can be easily dealt with. The following source code illustrates the principles of JUnit; it corresponds to the example in Sect. 5.2.

```
import junit.framework.*;

public class VendingMachineTester {

  // configuration interface
  private object Dispenser;          // test bed
  private object Display;            // test bed
private object TVM;               // tested component

public void setTestableDispenser
  (object testableDispenser) {
  Dipenser = testableDispenser;
}

public void setTestableDisplay
  (object testableDisplay) {
  Display = testableDisplay;
}

void setTestableVendingMachine (object tvm) {
  TVM = tvm;
}

// start test

public static Test suite () {
  TestSuite suite = new TestSuite ();
  suite.addTest (Test11.class);
  ...
  suite.addTest (Test31.class);
  ...
}

// test cases

public class Test11 extends TestCase {
  public void test () {
    Dispenser.setTo (Item1, empty);
    TVM.setTo (idle);
    try {
      SelectItem(Item1);
    } catch (DispenserItemEmptyException e){
      assertTrue (TVM.isIn(idle));
      assertTrue (Display.isIn(Empty));
    }
  }
}
...
public class Test31 {
  public void test () {
    Dispenser.setTo (Item1, notEmpty);
    TVM.setTo(idle);
```

```
          TVM.insertCoin (0.1);
          assertTrue (TVM.isIn(insertCoins));
          assertTrue (Display.isIn(0.1));
        }
      }
      ...
    }

    public static void main (String [] args) {
      // set up the test architecture
      VendingMachineTester VMT = new VendingMachineTester();
      VMT.setTestableDisplay(new testableDisplay());
      VMT.setTestableDispenser(new testableDispenser());
      VMT.setTestableVendingMachine
        (new testableVendingMachine());
      // execute the tests
      junit.textui.TestRunner.run(suite());
    }
```

The XUnit framework represents a way of implementing built-in contract testing in a particular language. It represents a language-specific and implementation platform-dependent form of developing built-in contract tests. In the following subsection I will introduce a language that is not bound to a particular programming language, TTCN-3, although it can be translated, into Java, for example.

5.5.3 The Testing and Test Control Notation – TTCN-3

TTCN-3, the Testing and Test Control Notation version 3, is a standard and a notation developed and put forward by the European Telecommunication Standards Institute (ETSI) that is particularly aimed at

- the specification and implementation of
  - test components,
  - test cases,
  - test behavior and execution sequence,
- functional/black box component testing
- distributed testing and testing of distributed systems.

ETSI's focus is on systems and standards in the telecom domain. So, TTCN-3 is initially targeted specifically at problems in that domain. It has already been applied in a number of typical telecom projects, such as mobile communication (GSM, 3G, TETRA), wireless LANs (Hiperlan/2), cordless phones (DECT), and Broadband technologies (B-ISDN, ATM), but also in areas that are not specific to the telecom domain, such as CORBA-based platforms and Internet protocols (IPv6, SIGTRAN, SIP, OSP) [94]. Although TTCN-3 has a strong telecom background, it is not at all limited to applications in that domain. It is in fact a multiple-purpose testing technology that can be used in a number of contexts, such as

- contract testing
  - interoperability testing,
  - performance testing,
  - integration testing
- robustness testing,
- regression testing,
- system testing,
- conformance testing.

The Testing and Test Control Notation is currently in its third incarnation. The original Tree and Tabular Combined Notation (TTCN) was first issued by the International Telecommunication Union (ITU) and the International Organization for Standardization (IS0) in 1992. TTCN is part of the ISO standard 9646 [53] that has been extended to TTCN-2, an intermediate release, and to TTCN-3 in its current version. TTCN-3 is a complete rework of the original TTCN test specification language with the intention to extend its applicability beyond pure OSI conformance testing utilization. In other words, TTCN-3 has been crafted specifically as a multiple-purpose testing technology. The core of TTCN-3 is based on a textual notation for test specification in a Java source code-like appearance. Its main characteristics are [150]:

- Dynamic test configurations that support concurrency, scalability, and distribution.
- Synchronous and asynchronous communication mechanisms for sequence-based and signal-based transactions.
- Data and signature templates that support rapid specification of the testing artifacts.
- Generic parameterization that allows custom types and behavioral specifications.

TTCN-3 Notation

The individual parts of the entire TTCN-3 suite are defined by ETSI standards that can be obtained from the ETSI Web site [94], and they are summarized as follows:

- Core Language [45]. It is separated into basic language elements, types and values, modules, test configurations, constants and variables, messages and timers, signatures and templates, operators, functions, and test cases. The core language looks similar to Java source code.
- Tabular Presentation Format [46]. This defines a graphical format for the specification of TTCN-3 test artifacts. It is basically an alternative way of displaying and manipulating the TTCN-3 artifacts. The core language

may be used independently of the tabular presentation format, but the tabular format cannot be used without the core language.

- Graphical Presentation Format [47]. This is the second alternative presentation format of the core language that is based mainly on message sequence chart (MSC) concepts [149].
- Operational Semantics [48]. This provides a state-based view of the execution of a TTCN-3 module. It introduces a number of different states and assigns meaning to the TTCN-3 constructs of the core language.
- Runtime Interface [49]. This describes the structure of an entire TTCN-3 test system in terms of Test Management (TM), TTCN-3 Executable (TE), SUT Adapter (SA), and Platform Adapter, and how these interact. Additionally, it defines the runtime interface in terms of operations that may be invoked by the entities in a TTCN-3 test system.
- Control Interface [50]. This presents a standardized way of adapting a TTCN-3 test system to a particular implementation notation such as ANSI C, or Java.

Figure 5.15 illustrates some of these individual TTCN-3 components and Figure 5.16 shows the organization of a TTCN-3 suite. The module is the top-

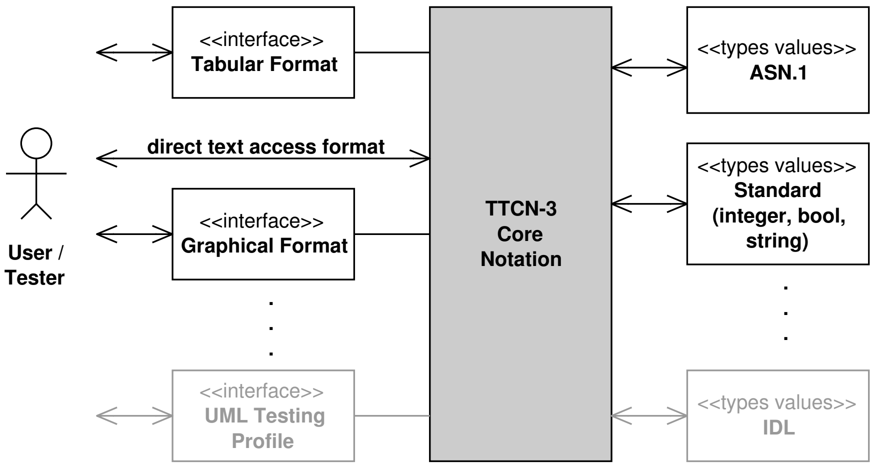


Fig. 5.15. Individual parts of the TTCN-3 test suite

level organizational unit of a TTCN-3 system. Each TTCN-3 module comprises a *definition part* and a *control part*. The first part contains all types and attributes that will be required during test execution and the second part acts as the main program that actually executes all the tests and defines their sequence and conditional branches of execution. Templates describe data in a generic form, mainly through matching operators and expressions. They can be used to define data ranges, data sets, constraints, and the like, and they

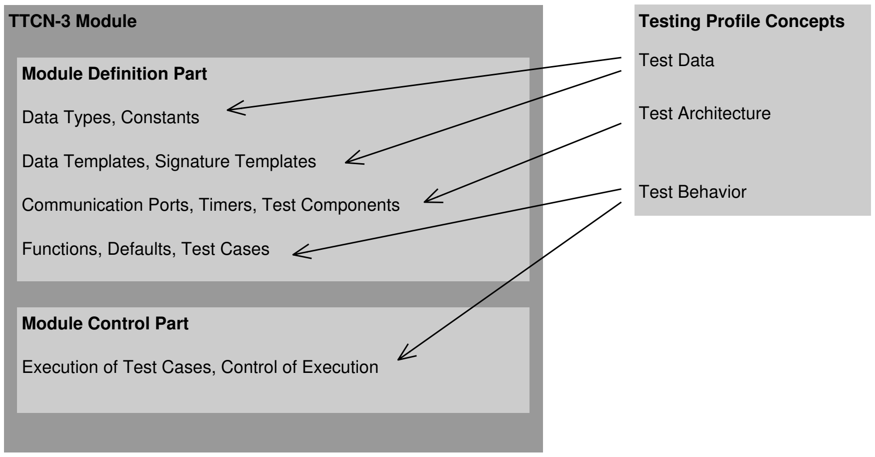

TTCN-3 Module
Module Definition Part
Data Types, Constants
Data Templates, Signature Templates
Communication Ports, Timers, Test Components
Functions, Defaults, Test Cases
Module Control Part
Execution of Test Cases, Control of Execution
Testing Profile Concepts
Test Data
Test Architecture
Test Behavior

Fig. 5.16. Organization of a TTCN-3 test system

may be applied to check the outcome of a tested component. A test case is a procedure that represents a sequence of test events, stimuli, and observations, and it generates a test verdict. The control part puts all defined items together into a meaningful test configuration.

TTCN Tools

TTCN is readily supported through a number of tools from a number of vendors or providers. I will give only a few examples.

OpenTTCN Tester

A testing tool suite provided by OpenTTCN [121].

TTthree

This is a TTCN-3 to Java compiler [107, 121]. It provides message-based and procedure-based communication with the system under test, as well as typical modularization concepts and for organizing and controlling the test cases. It realizes a graphical interface through which TTCN-3 can be accessed (Fig. 5.15), and it can be integrated with and supplement the popular Emacs development environment.

TTanalyze and OpenTTCN Analyzer

These are analysis tools similar to the parser of a programming language, and they can be used to verify TTCN-3 syntax and static semantics, declarations and parameters, and compatibility between tested and testing component [105, 121].

TTspec

This is the graphical user interface for the TTCN-3 tool suite [106]. It is used to specify the test cases on the basis of a graphical notation.

TTtwo2three

This is a TTCN-2 to TTCN-3 translator [108].

5.5.4 TTCN-3 and Built-in Contract Testing

In the following paragraphs we will have a brief look at how a test case may be implemented with TTCN-3. For example, the test cases of the RIN client's `RinServerTester` component, defined in Table 4.2, may be expressed in TTCN-3 notation. Each case can be directly derived from the behavioral model or from the table which represents an alternative view of the behavioral model. Additionally, we can specify the execution sequence of the three test cases through a testing behavior model (e.g., as a UML model) as displayed in Fig. 5.17. This will be implemented within the control part of the TTCN-3 module. The following TTCN-3 source code shows a simplified test case specification for the first test case, `#1 -- Registering`. It omits test types and data definitions [76].

```
external function
validClient() return Client;

altstep Default()
runs on RINClient {
  [ ] testPort.getreply {setverdict(fail); stop}
  [ ] catch.timeout {setverdict(fail); stop;}
}

testcase Registering()
runs on RINClient system RINServer
{
  activate(Default());
  // check the precondition
  testPort.call(IsInState(waiting),t) {
    [ ] testPort.getreply(IsInState(-): true)
    {setverdict(pass)}

    [ ] testPort.getreply(IsInState(-): false)
    {setverdict(inconc); stop}

    [ ] catch.timeout {setverdict(inconc); stop;}
  }
```

```
    // main test body
    testPort.call(Register(validClient),t) {
      [ ] testPort.getreply(Register(validClient))
      {setverdict(pass)}
    }
    // check the result
    testPort.call(IsRegistered(validClient),t) {
      [ ] testPort.getreply(IsRegistered(-): true)
      {setverdict(pass)}
    }
    // check the postcondition
    testPort.call(IsInState(registered),t) {
      [ ] testPort.getreply(IsInState(-): true)
      {setverdict(pass)}
    }
  }
```

TTCN-3 and the Testing Profile

It is not a coincidence that the concepts of TTCN-3, as introduced in the previous paragraphs, and the UML Testing Profile (introduced in Chap. 3) have many things in common (briefly indicated in Fig. 5.16). One source for the development of the testing profile was in fact the TTCN-3 notation, although the concepts of TTCN-3's Graphical Format, which comes closest to a graphical notation such as the UML, could not be transferred directly [148]. Table 5.2 gives only a brief overview of the UML testing profile versus TTCN-3. You will find a more thorough discussion of their similarities and differences in [148, 149].

TTCN-3 can be regarded as a generic test environment implementation technology. In the same way as coarser-grained models can be turned into finer-grained models and eventually into code closer to a typical implementation language, the TTCN-3 can be regarded as an intermediate generic implementation notation that will eventually be turned into code. This is illustrated in Fig. 5.18. The model on the left hand side represents the fundamental design activities for the testing environment. This is performed according to the built-in contract testing principles described in Chap. 4. The next step is to translate these graphical models into TTCN-3 that represents a generic implementation of the testing environment. The following step, turning the generic implementation into a concrete one, can be performed automatically through an available TTCN-3 to Java translator.

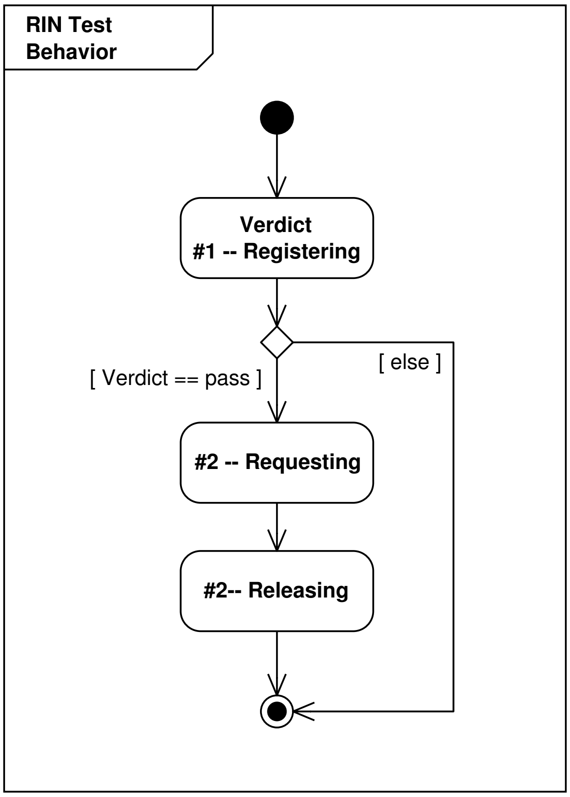


Fig. 5.17. Behavior of the RIN client's *RinServerTesterC* component from the RIN system, represented by a UML activity diagram

Table 5.2. Concepts of the UML Testing Profile vs. the concepts of TTCN-3

UML Testing Profile	TTCN-3
test architecture	TTCN-3 module definition part
test behavior	TTCN-3 module control part
tester component	test component
test case	test case
arbiter	test component
verdict	verdict
validation action	custom external function
test trace	test case

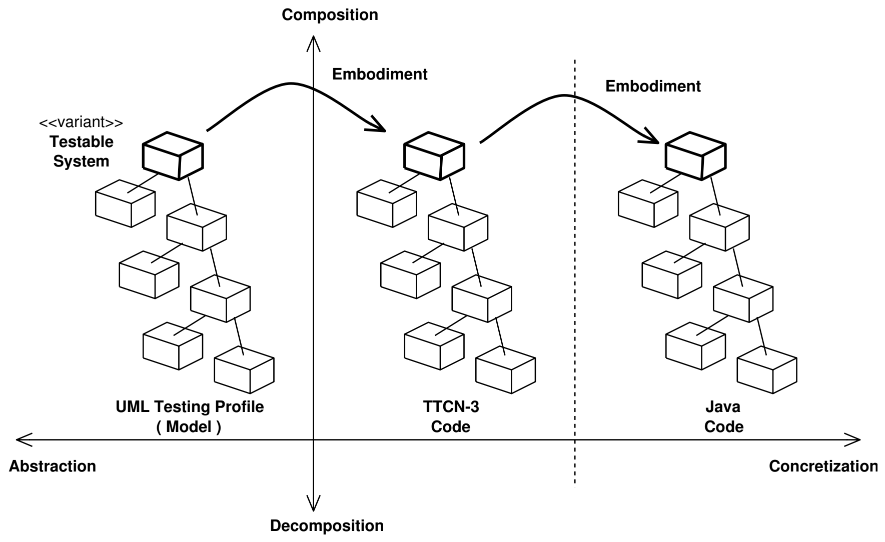


Fig. 5.18. Assignment of the UML Testing Profile notation and TTCN-3 notation to the embodiment dimension of component-based development

5.6 Summary

Embodiment is concerned with transferring abstract representations of a system into more and more concrete representations, thereby making decisions that reflect the particular requirements of the implementation platform used. The most basic implementation platform is a programming language. Any developed system will at some point become available as source code. In the case of model-driven development, the main activity that takes us to that point is a mapping from the concepts of the model space, the UML, into the concepts of the programming language. Contract testing is inherently built into the components that are transferred into code, so the mapping between the contract testing artifacts and the source code is the same as for any other functional artifact of the model. However, there are specific tools that we can apply to implement built-in contract testing, e.g., the XUnit testing framework for programming languages and the Testing and Test Control Notation. These tools and languages can take away a great deal of the coding effort that would otherwise be required to come up with an implementation for the built-in contract testing artifacts. For example TTCN-3, is capable of generating source code even for some middleware platforms.

The wiring standards, or the so-called middleware, and Web services also belonging to that group, represent more advanced administrative environments for reusable components. In general, they have no effect on how built-in contract testing should be treated. The testing is built-in at the programming language level, and the middleware code augments this with auxiliary infrastructure needed to build distributed component-based systems.

So, although these are important subjects, and embodiment is an essential phase in a component-based development project, its influence on the testing strategy as put forward in this volume is only marginal. This chapter has illustrated that. In the next chapter we will have a look at how reuse concepts affect or are affected by this testing technology.

6

Reuse and Related Technologies

In Chaps. 2 and 4, I introduced a development process based on the KobrA method that initially seems to propagate the idea that, once a system is decomposed into finer-grained parts, these parts should be developed from scratch. This assumption may be fueled by the strong focus on decomposition in the method described in which a system is recursively broken down into subsequently smaller and more manageable units, in a top-down fashion, that are individually turned into more concrete representations during embodiment. This represents only a single-sided view on a component-based development process such as is put forward by the KobrA method. The main motivation for applying a component-based development approach in a software project is that existing components from either a third party or developments in earlier projects that reside in a repository can be assembled and integrated relatively easily to form a new application. Embodiment, in this respect, also represents the activities that are necessary to integrate an existing reusable component implementation at the required location in the model. It is not concerned only with implementing a component from scratch out of the predefined component specifications.

Component-based development, in its purest form, actually encourages the opposite of this decomposition approach, which is that existing building blocks are successively assembled into larger units that eventually make up the entire system. This bottom-up exercise is represented by the composition activity described in Chap. 2. The difficulty with composition is in integrating an existing reusable component or its descriptive artifacts at the right level of decomposition and at the right level of abstraction within the overall component framework that the method provides. Another extreme is that a reusable component will exist only in the form of an executable binary representation at the highest possible level of concretization, or, if we look at it the other way round, at the lowest possible level of abstraction. Integrating components at higher levels of abstraction turns out to be a lot easier than if they are available only in executable formats with hardly any description.

The simplest form of reuse is instantiating an existing component implementation and using its services according to the defined clientship rules, if it matches exactly the required component specification. Typically, this simple form is not feasible because most existing components do not directly comply with the requirements of the integrating component framework, e.g., they provide less or more functionality because they are too specific or too generic, because they have been intended for a very distinct purpose which is different from the requirements of the new application, or because they have been never intended for a particular purpose at all, so that the new application requires extensive redesign and adaptation. Even if we decompose our system extremely carefully, so that we hit exactly all our existing component implementations, it is very unlikely that all components will fit together just like that in the end. Hence, a fundamental issue in component-based development is concerned with how an existing component can be integrated into the model of the component framework, and this requires that we determine the optimal location for this integration in the model. This can be done only if we decompose the entire application into logical units that may eventually be mapped to existing component implementations. In other words, on the one hand we have to find existing reusable components that will likely fit our predetermined decomposition hierarchy, but on the other hand we have to align the application model, the containment hierarchy, with the existing reusable component implementations that we have purchased. Component-based development and composition of concrete component realizations is therefore always contingent upon the right component decomposition at the logical or abstract level. So, for each iteration during component decomposition in which we subdivide our system into finer-grained parts we have to search for feasible candidates, existing components that may be mapped to the logical component specifications identified at the level of detail considered. The degree of reuse that we can achieve in a project depends heavily on how well we can map existing functionality to the requirements of the entire application. In Chap. 4, I introduced the concepts of syntactic and semantic mappings between the notations of the component framework and the reused component, in our case the UML, and the concept of component adapters where simple mappings are not enough. This represents reuse at a lower level of abstraction, that is to say at the level where we have already identified concrete component realizations that are suitable according to a specified component contract.

In this chapter we will look at reuse at a conceptual level. In other words, we will concentrate on how reuse is organized at the architectural level, i.e., through product families, how third-party components can be identified and evaluated, and how these activities relate to the model of built-in contract testing. But initially, in the next section (Sect. 6.1), we will take a closer look at how the built-in contract testing paradigm deals with software reuse in general. Then, in Sect. 6.2, I will describe how built-in contract testing may alleviate and support the earlier steps that are necessary before the actual reuse can take place, namely component identification and procurement. I will describe how such a testing technology can actually supplement an existing component brokerage platform, and how both technologies can be used in tandem to support component self-certification.

In the remainder of the chapter, we will focus on how built-in testing affects product families and product line engineering, which may be seen as the primary reuse technologies at the architectural level of a component-based development project. We will have a look at how built-in contract testing can be used to check product families and applications that are built on these. Another perception treats the development of a built-in contract testing architecture as a product line development in its own right, in which the non-testable original application is treated as the product family and the additional built-in testing system represents a particular variant of that product line.

6.1 Use and Reuse of Contract Testing Artifacts

Testing takes a big share of the total effort in the development of big or complex software systems. Guesses on software testing effort typically quote half the cost of the total development effort as cost for testing, although this clearly depends on the system and the type of organization. However, so far, component-based software engineering has mainly focused on cutting development time by reusing functional code. If existing reused components cannot be applied without extensive rework or retesting in the target applications, the time saved from component-based development becomes questionable [83]. Hence, there is a need to reuse not only functional code but also the tests and test environments that can be used to ensure that the components work on the target platform or within their target application. In order to achieve effective test reuse in software development, there are several aspects that must be taken into account:

- Increased testability through the use of built-in test mechanisms.
- Standardized testing interfaces.
- Availability of test cases.
- The possibility to customize the tests according to the target domain.

Built-in contract testing provides a flexible architecture that focuses on these aspects. It is the application of the built-in contract testing architecture that makes reuse possible. In the initial approach of built-in testing as proposed by Wang et al. [166], complete test cases are put inside the components and are therefore automatically reused with the component. While this strategy seems attractive at first sight, it is not flexible enough to suit the general case. A component needs different types of tests in different environments and it is neither feasible nor sensible to have them all built-in permanently.

Built-in contract testing separates the test cases from their respective components and places them into separate tester components. The client components that incorporate tester components will still have some built-in test mechanisms, but only to increase their accessibility for testing, i.e., some sort of a configuration mechanism that can be used to acquire tester components dynamically and invoke their execution. The actual testing and test case execution is carried out by the tester components, and these are connected to the tested server components through their respective testing interfaces. In the developed architecture, an arbitrary number of tester components can be connected to an arbitrary number of tested server components. This offers a much more flexible way of reusing test cases because they do not have to be identical to the ones originally defined by the provider and delivered with the server component. The tests can be customized to fit the context of the component at all stages in the component's life cycle.

Tested components have built-in mechanisms that increase their testability. For example, these mechanisms can be error detection mechanisms like assertions, methods to set and read the state of the component, and methods that report resource allocations. These mechanisms can be accessed through a standardized BIT interface, and are automatically reused with the component. We will have a closer look at these additional mechanisms in Chap. 7, which deals with checking quality-of-service contracts.

The overall concept of test reuse in built-in contract testing follows the fundamental reuse principles that are common to all object and component technologies. Because testing is inherently built into an application or its parts thereof (the components), testing will be reused whenever functionality is reused. In fact, testing code in this respect can be seen as any other normal functional code. Only the time when this functionality is executed, for example, at configuration or deployment, distinguishes it from the other non-testing functionality. In the following two subsections we will have a look at how built-in contract testing affects reuse during development-time, deployment-time, and runtime.

6.1.1 Development-Time Reuse

Individual components will be tested in two stages and according to two different viewpoints. An initial test will be carried out by the component vendor during its development. This test at the vendor's site ensures that the component's internal parts have been integrated properly. A second test will be carried out by the component user, or the application engineer, during its integration into a new component framework or context. These represent the two fundamental testing stages of a component. The two viewpoints according to which components are typically tested are defined by the criteria that both stakeholders apply during development.

The provider either releases a component that has been tested in isolation from any context, in this case a generic component that is suitable for a number of different purposes that may or may not have been explicitly defined by the component vendor. So the component test will be performed according to the requirements that the provider has defined for this particular component.

In most cases this will comprise a simple unit test that checks each individual service (or method) against its specification (or model). Another test of the provider will also consider message and method sequences, but these represent very specific usage profiles that the vendor has explicitly defined. In this case, the component has been devised for a very specific purpose in a very specific context. In fact, the test cases of the provider define this usage profile through examples.

The customer of the component performs an integration test within the new deployment environment. This new deployment environment is different from the provider's original development environment. So, the component customer's requirements will quite likely differ from the provider's requirements, so that the second test at the customer's or user's site will be performed according to the specification of the customer's integrating application. Because both requirements will likely be different, as I noted earlier, the test of the component provider is useless as a test for the user of the component. At least this is the case for an integration test within the new deployment environment. Weyuker has investigated this and found that a component that had been reused within two different projects was actually being used entirely differently by the contexts, with entirely different coverage of the component's code. The two contexts presented completely different usage patterns [175].

Components are often reused and integrated as if they have never been tested before [176]. The component provider can only try to perceive as many different feasible contexts as possible and devise tests accordingly for the released component. Each distinct context will map to a test suite and thus to an external tester component. Each tester component can extend previous tester components for other usage profiles, so that a more extensive and more general test suite will consist of a number of less extensive and more concrete test suites for particular usage profiles. Figure 6.1 illustrates this principle. The component testers on the left hand side of Fig. 6.1 represent the testing that the vendor or producer of the COTS component has already developed and performed. Ideally, these are provided openly, together with the vendor's original component. Each of these tester components represents an expected usage profile of the COTS component that the vendor of that component perceives for this product. In fact, the tester components are not entirely meaningless for a component user, as we will see.

Platform test

Initially, invoking a component's tester component, one delivered with the component, as a unit test is entirely meaningless. It will not discover any new errors within the component. This is because the provider should have run these tests on the component prior to its release and addressed any problems. The provided tester component is only meaningful for the customer if a typical contract testing scenario is considered. The provider's tester component invokes services on the component, but not in terms of a unit test

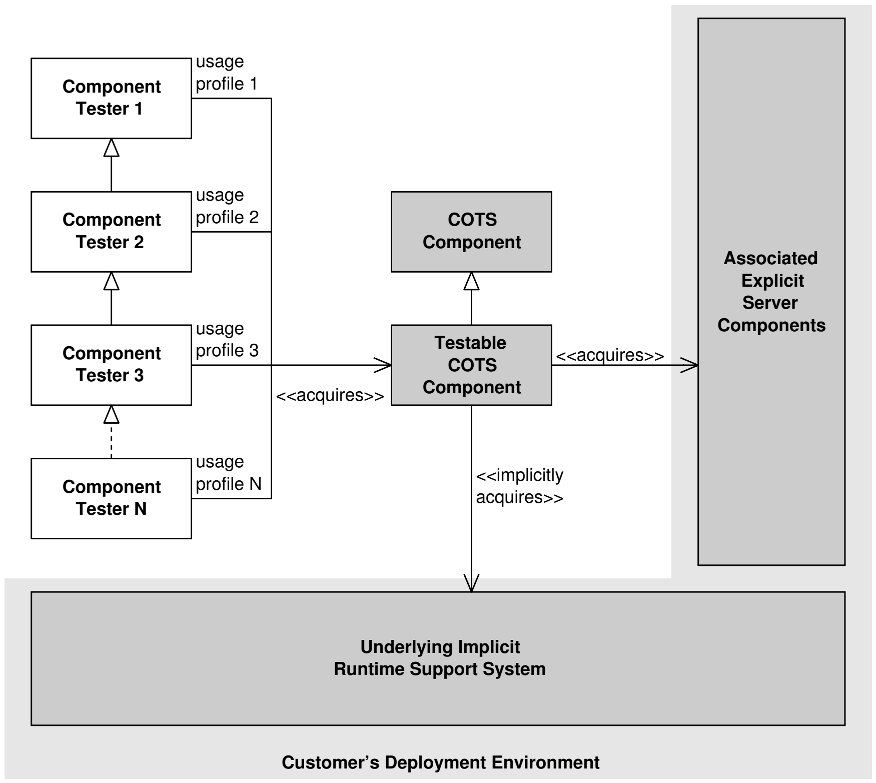


Fig. 6.1. COTS component with provider's tester components

as a stand-alone entity, but integrated within its new deployment environment. The tested component will in turn invoke services on its associated implicit and explicit servers, as indicated in Fig. 6.1. This represents a platform compliance test of the integrated component. The test cases provided can be executed to see, in general, whether the integrated component works in the component framework of the customer or user. These tests will not be adequate according to the system's usage of the component, i.e., the customer's client components of that COTS component, but they will identify at least fundamental problems of a new component in an existing application framework.

Built-in Documentation

The tester components that come with a COTS component illustrate how the vendor expects the component to be used, and how the vendor expects the component to use its own environment. Thus, the testers represent the vendor's interpretation of the component's specification of its provided and required interfaces. And this in fact realizes built-in documentation for the COTS component, because the test cases show a prospective user of that component what the component expects to get from its environment in terms of expected services as well as in terms of expected pre and postconditions. Additionally, tester components illustrate for which different usage profiles the component is suitable. This facilitates the decision as to whether the component is usable and can possibly be integrated into the intended framework of the application or not. For this purpose, the tester components can be compared with those that the user of the COTS component has developed to see whether his or her expectations map to what the COTS component provides. This is part of the component procurement process that is considered in more detail in Sect. 6.2 of this chapter.

6.1.2 Runtime Reuse

In the previous subsection, I concentrated on how built-in contract testing and the provision of testers together with third-party components facilitates application development. The whole technology has focused mainly on application development-time testing so far. But built-in contract testing provides a lot more. Since it is a built-in testing technology that might well remain in an application beyond deployment, in theory the tests can be invoked at any time during runtime. Consequently, built-in contract testing provides the support for performing and checking live updates of a system in the way that Web services provide. The entire ITEA project EMPRESS [126] was dedicated to the subject of software evolution in the context of embedded systems not only for development-time software evolution, but also for runtime evolution. The fundamental difference between the two evolution steps is that for development-time evolution the system is stopped, reconfigured, or updated, and started again. In fact, the new reconfigured system is deployed as if it were an entirely new application. In contrast, in a runtime evolution step, the system will remain operational during the reconfiguration and continue as an updated version without need of its being stopped or rerun.

Server–Client Test

Another scenario, quite common for telecom systems, is that the server component, e.g., a telecom service, needs to assess a registering client component, e.g., a mobile phone application. This scenario is illustrated through the structural diagram at the bottom of Fig. 6.2. Initially, in a first step, the client component registers with the server, and then, in a second step, it passes the reference of its own tester component to the server. This is not its own built-in tester, the `ServerTester`, but the tester component that would normally check the client. In this example, it is the `ClientTester` on the left hand side of Fig. 6.2. The first step is represented by the ≪acquires≫ relationship between the client and the server, and the second step by the ≪acquires≫

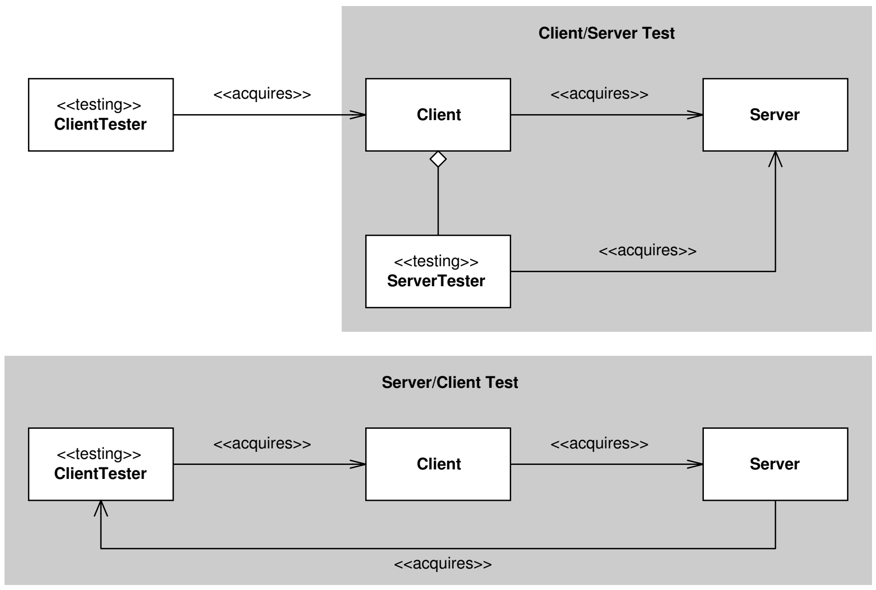


Fig. 6.2. Client–server test vs. server–client test

relationship between server and `ClientTester`. In reality, this simple model will probably be realized through the contexts of the two associated components. This is because the client needs to "know" the reference of its own tester to pass it to server, and this is usually not the case.

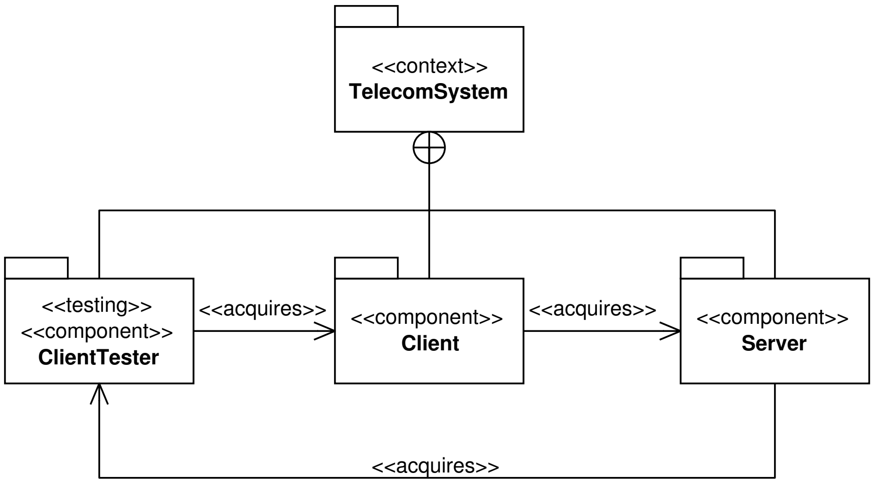


Fig. 6.3. Containment hierarchy for a server–client test

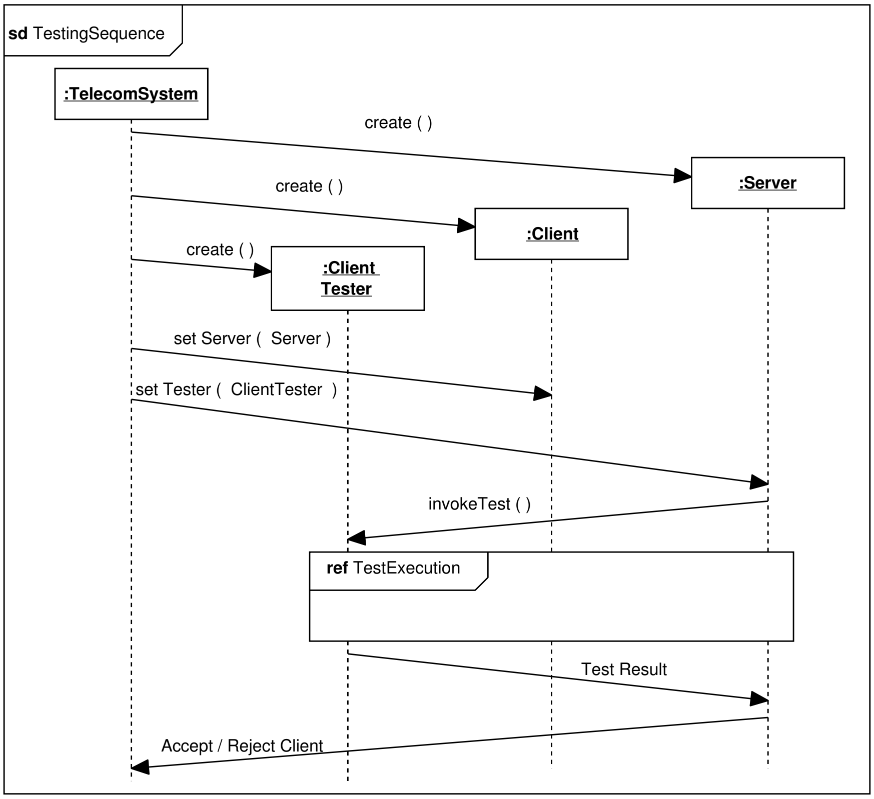


Fig. 6.4. Invocation sequence for setting up a server–client test

Fig. 6.3 shows the containment hierarchy with the context that is responsible for actually creating all components and interconnecting them, and Fig. 6.4 depicts the invocation sequences that the components have to perform to realize this type of testing. The context creates all component instances and connects the client with the server, through `setServer(Server)`, and the server with the `ClientTester`, through `setTester(ClientTester)`. The server can now invoke the test sequence that is stored in `ClientTester` to call the services of the client. The client is thus exercised with scenarios that are typical within this context and, as a consequence, the client invokes the services on the server component that are typical for this scenario. The outcome of this test is returned to the server because this component currently owns the `ClientTester`, and based on that the server can decide whether it will accept or reject the client, and communicate that to the context. This is similar to a platform test in the same way I have explained earlier (Fig. 6.1), but here the test is not initiated by the component, or the component framework, but by the underlying runtime system, which is represented by the server.

6.2 Component Certification and Procurement

Component-based software development, although intuitively quite natural and also economically appealing, has not had the breakthrough in the software domain that one could expect by looking at other traditional engineering disciplines such as mechanical or electronic engineering. Component-based software development may still be considered in its relative infancy compared with how systems are developed from readily available prefabricated parts in these other engineering disciplines [4]. This is not so much the lack of technological support in the software domain, because there is a sound theoretical basis, and the implementation technologies and deployment environments are quite advanced, as we have seen in Chap. 5. The most important reason that component technology has not had such an impact in the software industry lies in component procurement. The more traditional industries thrive on mature component markets in which component providers publish bulky catalogues of high quality, standardized bits and pieces. Application engineers also know and apply the standards, and are well aware of the vendors, their reputation, and their offerings in the market. They live in healthy cooperation. The software industry is almost entirely lacking such infrastructural foundations, and people are only slowly becoming aware of the fact that polished technologies do not necessarily make the component paradigm more emergent.

The European funded project CLARiFi [125] was a step forward in the right direction in that respect, as it developed a component broker platform in which component provider and purchaser find each other. CLARiFi provides the fundamental support for the certification process but it still poses some limitations that are related to the domain context. An extension to different domains is complex and only some guidelines have been provided. Certification can be seen as the assurance that a vendor's claims concerning a product are justified, and so far it is more or less a matter of trust between provider and customer, or it inevitably involves a third party. This third party may be an independent certification organization, the owner of the broker platform, or a user group that publishes the opinions of its members. However, the first requires sound, well-defined, and accepted component standards, the second requires a responsible and well-educated operator of the broker platform, and the third requires established criteria for component assessment. Even if standards for software quality, such as the ISO 9126 [56, 57, 58] and the ISO 14598-3 [55] are available, they suffer from limitations related to their customization to specific application environments. Without such criteria, every evaluation through a third party will inevitably lead to subjective verdicts. The only alternative to avoid this is to strengthen the role of the component customers and equip them with the right means to carry out their own certificates according to their own criteria. For the component providers this means they have to publish as much information as possible to enable the customers to assess a product themselves. There are two feasible ways for the component producer to provide more information:

1. Apply and provide component reliability principles such as put forward by Hamlet, Mason, and Woit [84], or by Reussner, Schmidt and Poernomo [138].
2. Provide testable components and component testers according to the built-in contract testing paradigm.

Both approaches are orthogonal and can be used in tandem. While the first ones are more formal and based on algebras for system design, the second one is more informal and based on what testing artifacts can offer. In the following subsections I will give a brief overview of how the advantages of built-in contract testing can be exploited to facilitate component certification of the second kind.

6.2.1 The CLARiFi Component Broker Platform

The goal of the CLARiFi project was the development of a component broker that is able to support the integrator in the process of software component location, evaluation, and selection [153]. This process includes several interactive steps that are based on the description of the application that the integrator is building [123, 155] and on a reliable description of the readily available components that are published through the CLARiFi platform [30]. Suppliers need to classify the functional and non-functional features of their components according to a taxonomy provided by the CLARiFi system. The classification process of components requires a certification to assure the correspondence between the description and the related implementation of the features. To address the certification problem, the CLARiFi project consortium identified a certification procedure, a set of data to collect and store inside the CLARiFi repository, and a set of roles that certifiers can play. The certification process requires that all these elements be provided through a reliable component broker. The certification process in CLARiFi includes two aspects of certification standards: quality standards [18] and technical standards. The first set includes both functional and non-functional qualities; the second one includes safety, security, and other additional criteria.

One of the goals of CLARiFi is to help integrators find the best set of components for building a system according to components' predefined features and qualities. To address this purpose, the CLARiFi certification model is mainly based on the two ISO standards, ISO 9126 and ISO 14598-3. The ISO 9126 standard provides a general model to specify and evaluate the quality of a software product from different perspectives such as development, maintenance, etc. This model addresses primary software quality issues and it is useful for most quality evaluations. The first part of the standard defines both internal and external characteristics, dealing with the software development process and with qualities of the final product [56, 57]. The second part of the standard defines so-called quality-in-use characteristics, which effectively concentrate on product qualities according to the user's point of view. Additionally, it defines a framework for performing measurements [58]. ISO 14598-3 defines guidelines or criteria with which quality models can be devised. It divides quality attributes into six categories that comprise specific subcategories:

- Functionality, which is defined as the ability of the software product to provide certain functionality. Subcategories are suitability, accuracy, interoperability, and security.
- Reliability, which is defined as the ability of the software product to respond in an appropriate way under specific conditions. Subcategories are maturity, fault tolerance, and recoverability.
- Usability, which is defined as the ability of the product for ease of use. Subcategories are understandability, ability to learn, operability, and attractiveness.
- Efficiency, which is defined as the ability of the software product to provide a certain level of performance through resource management. Subcategories are time behavior and resource utilization.
- Maintainability, which is defined as the ability of the software product to be modified. Subcategories are analyzability, changeability, stability, and testability.
- Portability, which is defined as the ability of the software product to be moved into another environment. Subcategories are adaptability, installability, coexistence, and replaceability.

Under CLARiFi the accreditation of the certifiers is done according to these categories, and these criteria represent a good starting point to approach the problem of certification. The standard provides suitable collection of guidelines, though not universally accepted due to the subjectivity involved, but well suited for customer self-certification.

6.2.2 Customer Self-certification

Certification is typically carried out by an independent authority that assesses the product according to predefined criteria. In software practice, however, it is difficult to find such criteria that are generally accepted as standards. Not only are we lacking standard evaluation criteria in the software industry, but we do not have standardized components in the first place, as is the case in other, more mature engineering domains. Moreover, component vendors and customers may have their own, entirely contrasting criteria for assessing the "goodness" or "badness" of a software component. For software, it is often difficult or even irrelevant to specify and certify a set of attributes because they depend on the environment into which a software component will be integrated, and the provider can never anticipate that. This becomes apparent when we look at performance and interoperability issues. Performance is highly dependent on the underlying middleware platform and on the hardware, and interoperability can be guaranteed only with respect to how much information the component vendor has on the other parts with which their software is supposed to interoperate. For example, a commercial Web browser plug-in component is integrated into a Web browser that has a weird interpretation of HTML. Obviously, the component is correct, but can we blame the vendor of the Web browser for not supporting a distinct way of interpreting a language, because it has been established by a competitor? The problems of certified and certifiable software cannot be solved simply through new technologies. They are clearly more fundamental and require all the stakeholders to go into a huddle and come up with acceptable and well-accepted standards. In the meantime we have to live with what we have.

One short-term solution for the previously described certification dilemma is to give the component customers the responsibility of performing the assessments for the components they purchase. Self-certification or self-assessment always requires that the product be available and accessible. So, the customer of a component must be given the opportunity to evaluate its function and behavior as well as all the other quality criteria that he is interested in. This is pretty much like a test drive with a new car that we intend to purchase. The customer needs an executable and testable version of the component. Maybe the vendor can offer a decaying or a trial version that the customer can play around with and integrate in the existing component infrastructure. We cannot really call this certification, since we are lacking well-defined and accepted standards according to which we can apply a certificate. It is more of a self-assessment, and the criteria and standards that we apply are not generally accepted by or relevant to other component customers. A self-assessment is always better than pure trust, although as a customer we will never get full access to a component in the same extent than a well-respected certification body. At least the assessment supports us a little bit in evaluating a commercial product. I will illustrate how built-in contract testing principles can help support and facilitate the self-assessment of commercial components, so that as customers we can replace trust by control, and belief by knowledge.

A component is a self-contained encapsulated piece of software, or a black box, that exhibits function and behavior according and in response to external stimuli that are invoked at its interface. Apart from what the interface provides, the testing software of a component, or its external test cases, do not "see" any internal implementation. This is detrimental to assessing the quality of a component from the outside, and we will be unlikely to get insight into a commercial component because the provider is eager to protect its asset. Built-in contract testing proposes to extend the normal functional interface of a component with an introspection mechanism, or a testing interface. The testing interface provides controlled access to the black box component and increases its controllability and observability during a test, without publishing the intellectual property of a commercial vendor that remains protected by the encapsulating hull of the component. The key idea of a testing interface is to provide some insight into the black box that is necessary to assess it, but to yet keep the encapsulation intact. This enables a user of the component to set and retrieve state information and assess the correctness of all performed transitions during a test. Each user of the component can thus apply scenarios that are typical to their very particular way of using it. The testing interface that, among other things such as assertions, makes the externally visible states accessible is capable of providing a great deal of testing information. This is especially important during extensive test runs and with reduced observability of the original component, for example, if the component does not naturally provide much output during execution. Both, internal state information and assertions can easily be implemented by any provider of commercial components, and they can be incorporated in a testable version of the component.

The aim of CLARiFi is the collection and management of software components to classify and retrieve required components. On the other hand, built-in contract testing provides a way to analyze the behavior and the functionality of a component at integration time. The two approaches are complementary even in terms of the component certification features. In fact, CLARiFi offers a brokerage service that can also provide components that are realized according to the model of built-in contract testing. Testable components are components after all, and the testing functionality is only additional functionality that happens to implement component testability and component testing. Testing and testability features may not be part of the component search and identification process. CLARiFi's brokerage system offers a way to find components according to their specific functional and non-functional features, and these are definitely the most important aspects for the selection of any software artifact. This is possible through formulating a query by choosing from a list of available properties that are offered by the brokerage system, and by setting the values of these properties according to the specific need of the customer. Customers can use these properties to specify which kind of certification aspect they are interested in. These properties are easily applicable to the customer.

6.3 Product Families and Testing

Product families and product line engineering deal with organizing and exploiting the commonalities of several similar systems and at the same time managing their differences. In Sect. 2.6 (Chap. 2), I presented an overview of product family concepts and how they are treated and applied in the KobrA method. In this section we will have a look at the challenges in testing product lines, and how built-in contract testing addresses them.

Product line engineering lies at the heart of component-based development. It represents reuse at the architectural level rather than at the single component level. It can be seen as elevating component-based development to a meta-level because components can be regarded as systems in their own right, and product line engineering is really about how to reuse entire systems or their parts. KobrA's containment hierarchy model is a good representation form to illustrate this principle, so product line engineering really spans all aspects that can be dealt with in this model. It is difficult to draw the line between what a product family is and what a component framework according to KobrA's containment model represents. We could argue they are the same, and, in fact, I believe they are. Product line engineering takes a different view on component-based development, and it organizes components more strictly into components that will always pertain to the common core of the product family and components that are optional or variable in a model. Product line engineering is not so much about dealing with how components are modeled, designed, and, above all, integrated; these can be seen as the primary focus or the core of component-based development. Product line engineering is more about how similarities and differences in diverse systems of a single organization are maintained and organized. So the notion of reuse is directed more toward how a component assembly can be reused as a whole in different systems that are somehow similar, and how diverse components can be attached to or detached from this whole.

Basically, a product line development comprises an assembly of components that are likely to stay fixed over a longer period of time. This represents the common core, the product line, or the product family. Additionally, it encompasses some other components that are more likely to vary because they represent the individual products that may be instantiated out of the product line. We had these discussions in Chap. 4 (Sect. 4.4.1), where we looked at how to identify the component contracts in a system that should ideally be augmented with built-in contract testing artifacts. There, I argued that module interactions that are likely to change over time should be enhanced by permanent built-in contract testing artifacts, while interactions that are likely to stay fixed over some period of time should be only temporarily enhanced by these artifacts. Ideally, they should be incorporated during development and deployment, but removed after that. I identified the component peripheral as an ideal place for permanent contract tests because components are the basic building blocks in component-based development. The classes inside the components may be tested once and for all, so that the built-in testing may be removed after integration into the final deployable component, because their internal configuration will not change any more. The same is true for product line developments, although at the "meta-component level". Here, we have components that will be configured in a distinct way and likely stay that way for some time. This configuration, the generic product family core, may be represented by a superordinate component at a higher level of composition than that acquired by other components that make up a specific final product. Essentially, by looking at it in this way, we are only applying the powerful recursive development principles of the KobrA method, and in the next subsection we will have a closer look at how this affects testing.

6.3.1 Testing of Product Families

A test of a product family comprises typical individual component tests in the same way that a normal component provider does. It checks the integrity of the component as an individual entity. The second step is to test the core assets or the component framework in the same way as is done for the integration of a component assembly. The third step is to check the variant features, and the interactions between the common core and the variant features. In this regard, testing a product line is at the architectural level not particularly different from testing a normal component framework or an application. Additionally, we may apply product-specific testing criteria in order to come up with adequate test sets [14, 111, 139, 160], but this is not the focus here. Built-in contract testing can be organized in a way that it greatly facilitates tests of product families.

A good product line architecture organizes the variant features at the inter-component level rather than at the intra-component level. This is illustrated in Fig. 6.5. In other words, variant functionality should always be accommodated in separate components, so that the modeling of these features is restricted to the assignment of a particular component. Otherwise, if variant features are defined inside a component, they are in fact intermingled with the features of the core framework. This is because most programming languages do not support dynamic allocation of component features inside the component, except for extension or preprocessing, such as `#<include>` in C. But even extension realizes an inter-component level organization of the variant features, because inheritance always defines two separate classes, the "super"-class and the "sub"-class.

The advantage of strictly separating optional and variant features from the product family core in the models as well as in the code is that only the points of variability, or the so-called component hooks, have to be defined in the common core. According to the product that should be instantiated out of a product line, we have to instantiate the respective variant component and assign it to the respective component hook in the framework. This is illustrated in Fig. 6.6. The instantiation of a product out of a product line then requires only that the respective component references be assigned correctly.

Essentially, the only artifacts in the core framework that deal with variability are the references of the associated optional and variable components, though I have to admit that this is not the entire truth. We will require different access mechanisms in the component framework or the product line core according to the concrete product we are instantiating. For example, if we instantiate a vending machine with credit card billing, it will require not only an associated credit card billing component, but the vending machine will also

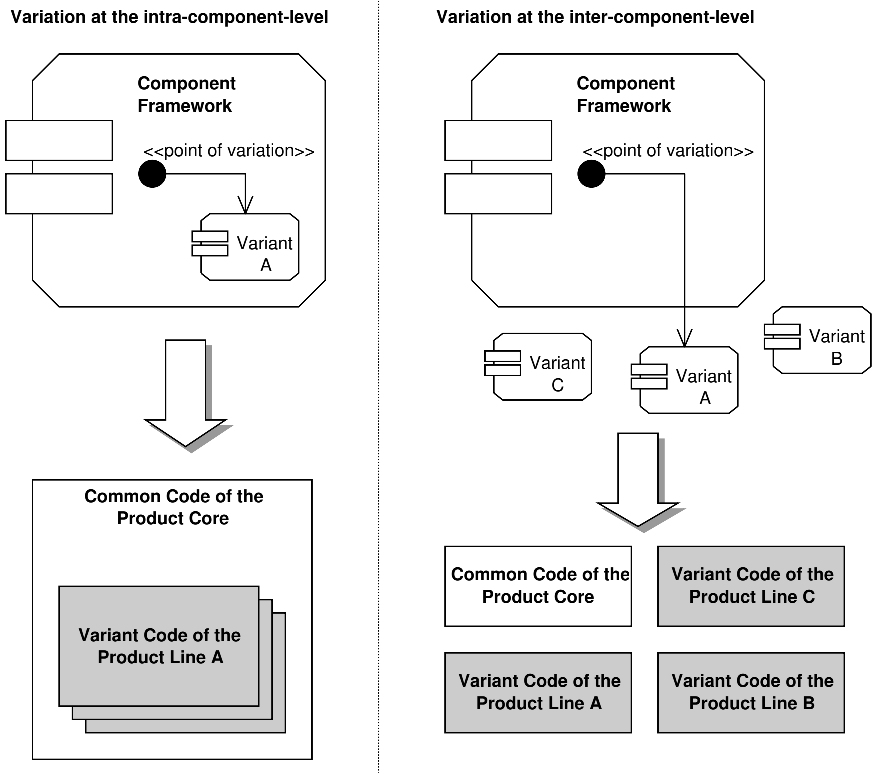

Variation at the intra-component-level
Component Framework
<<point of variation>>
Variant A
Common Code of the Product Core
Variant Code of the Product Line A
Variation at the inter-component-level
Component Framework
<<point of variation>>
Variant C
Variant A
Variant B
Common Code of the Product Core
Variant Code of the Product Line C
Variant Code of the Product Line A
Variant Code of the Product Line B

Fig. 6.5. Variation within a component vs. variation between components

have to provide an interface to access this functionality, e.g., the operation `insertCreditCard()`. The vending machine is the core system and it must be augmented with the optional credit card billing functionality that represents a final product in this product family. So, in order to separate core functionality from variant functionality we will need another vending machine component that caters to that additional feature. This is depicted in Fig. 6.7. The class `CCBillingVendingMachine` represents an extension to the original core vending machine and it comprises the artifacts that are necessary to realize this variable feature. These are the object reference `CCBilling`, which acquires the `CreditCardBilling` component in the form of an additional attribute, and the operation `insertCreditCard()`, which will be invoked from the user interface as an indicator that a customer intends to settle the bill with a credit card. Through this organization, we entirely separate the variant features at an individual component level.

This principle will then pervade the entire model or code corresponding to the abstraction level we are dealing with, e.g., the user interface. So, if the plan is to develop a vending machine without credit card billing, we have

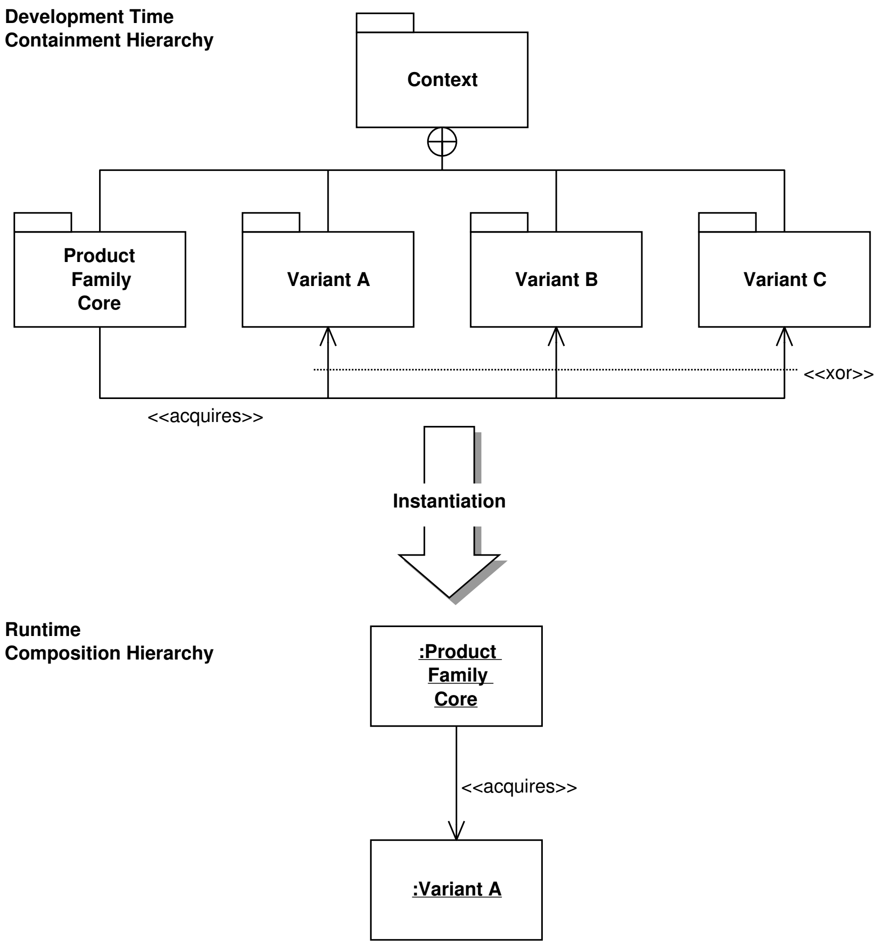


Fig. 6.6. Instantiation of a product out of a product line

to instantiate the core `VendingMachine` component, and if we aim at developing a credit card-enabled vending machine we have to instantiate the variant `CCBillingVendingMachine` plus the `CreditCardBilling` component (Fig. 6.7). If a product line is organized in this way, its pure composition is not different from any other component-based system. The decision models in the product line development will assign only different components to the core, according to the final product that will be instantiated. In other words, product line engineering merely represents a mechanism for including or excluding particular components from a model. For the general case, this is shown in Fig. 6.8. The common core is instantiated into a particular product by resolving decisions according to `Application_1` or `Application_2`, and this is initially done only at the logical component level in the model. To retrieve

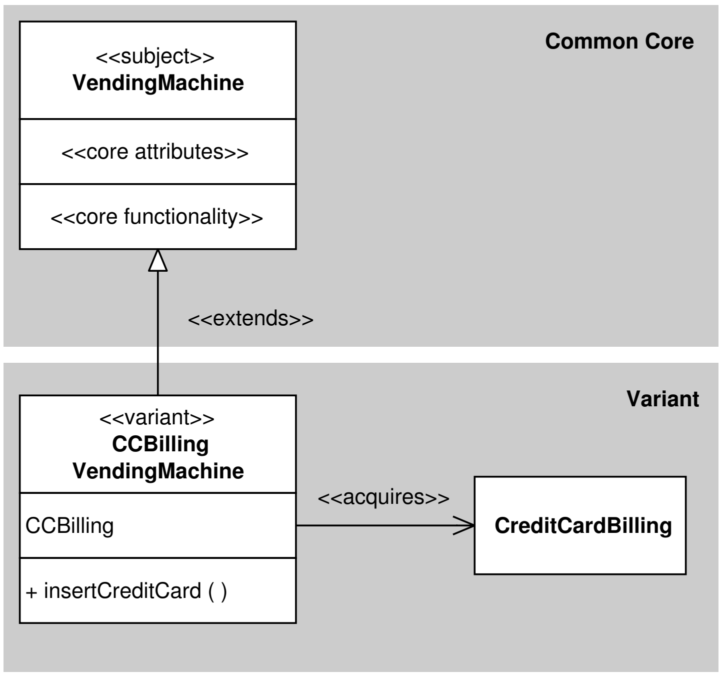


Fig. 6.7. Product line architecture for the vending machine example

`Application_1`, we need to add the two components `A` and `B`, plus the models that will realize the hooks for `A` and `B`, into the product family core. And to come up with `Application_2`, we have to add component `C` plus the additional framework within the product family core that is necessary to control component `C`. So, in addition to providing the respective variant components, we have to provide and instantiate the right models in the common core that are capable of dealing with these variants.

The product family core depicted in Fig. 6.8 is likely to stay the same for all possible products in the family. Otherwise it would not make any sense to define such a product family core. At a high level of abstraction we have a containment hierarchy that organizes the core into a number of nested logical components. At lower levels of abstraction, when we move down the concretization dimension, we can actually treat the component containment hierarchy that represents the product family core as a single component that encapsulates the logical component tree. This is indicated in Fig. 6.9. In other words, the product family core can be treated as a component in its own right at lower levels of abstractions, i.e., in the UML component model. By treating the product family in this way, we can apply the known principles of built-in contract testing and add the testing artifacts. Here, the most important issue is how the tester components are organized. We assume that the variant

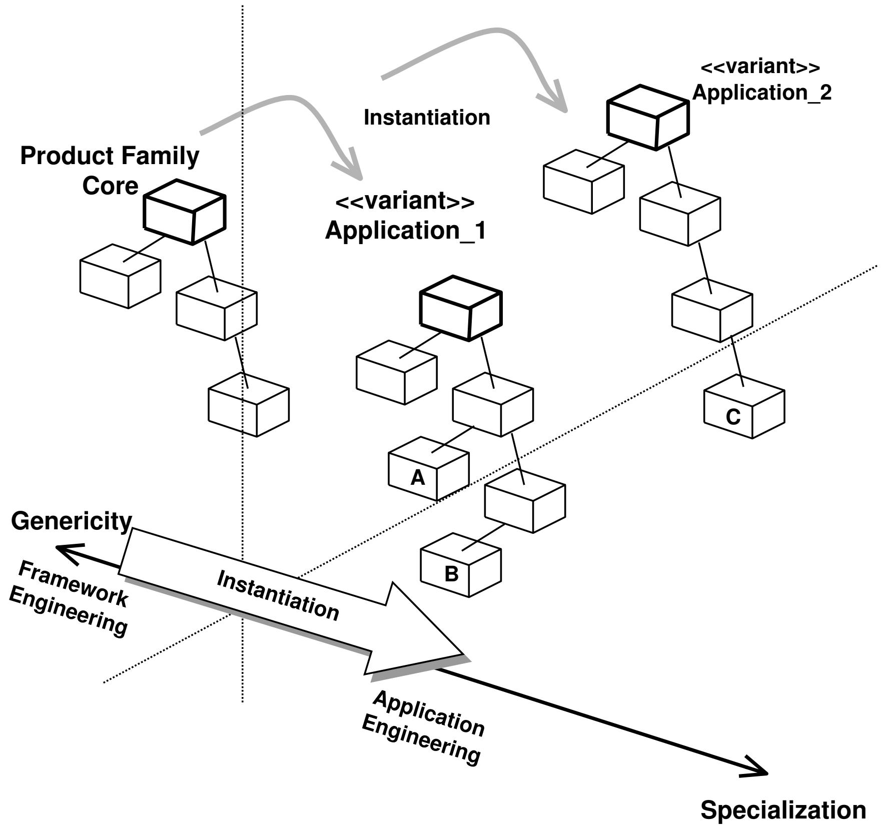


Fig. 6.8. Instantiation of two alternative applications from a single product family core at a high abstraction level

components `A`, `B`, and `C` in Fig. 6.9 will provide testing interfaces according to the rules defined in Chap. 4, thus making them `testableA`, `testableB`, and `testableC`.

The philosophy behind built-in contract testing is to add contract testing artifacts at the encapsulating hull of a component so that it can check its immediate environment. This comprises a testing interface at its provided interface and tester components at its required interface. Now that the product family core is treated as a component in its own right, we can add these artifacts easily. The testing interface of the core component is represented by the testing interfaces of the individual contained components, as depicted in Fig. 6.10; and at the required interface of the core component, we can add tester components according to the required interfaces of its contained subcomponents. The tester components are designed according to the client's usage

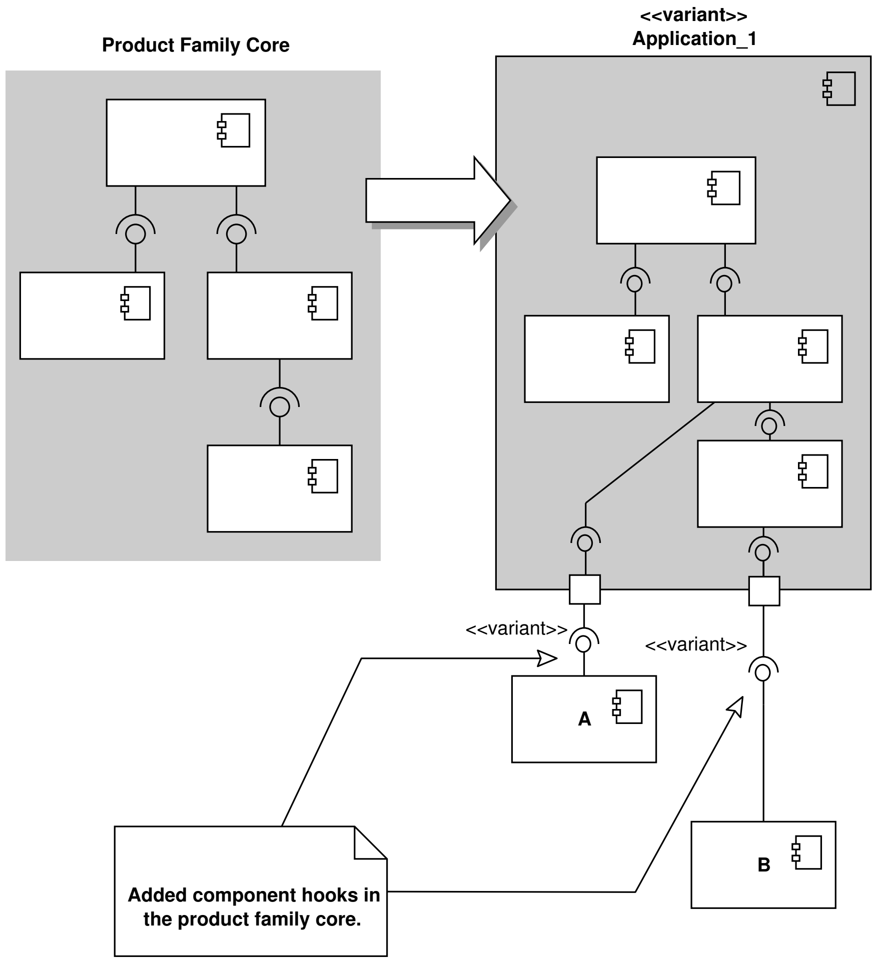


Fig. 6.9. Instantiation of a final application out of a product family core at a lower level of abstraction. The product family core may be treated as a component in its own right

profile of its associated servers. These servers are the components that represent the variablity and turn the product family core into a final application.

According to the contract testing philosophy, we will need a tester component for each client/server association between the product family core component and its extending variant components. Fig. 6.11 shows this architecture for the generic example depicted in Fig. 6.9. Each point of variation in the product line core will be associated with a distinct variant component plus a distinct tester component for this variant component. The tester components will be designed according to the expectation of the integrating framework,

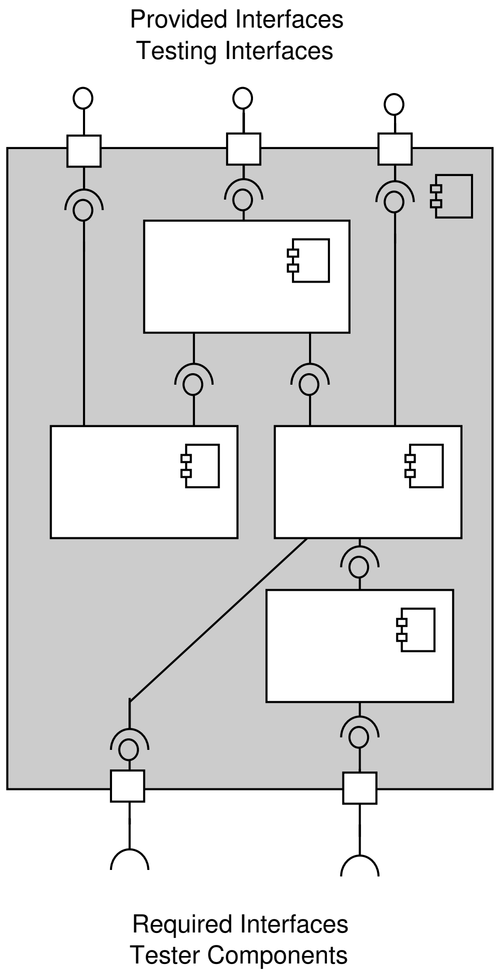


Fig. 6.10. Provided and required interfaces of the product family core are represented by the respective interfaces of its contained subcomponents

i.e., the product line core. To instantiate a `testableApplication_1`, we need testable components `TestableA` and `TestableB`, plus an `ATester` component and a `BTester` component associated with the clients of `A` and `B` respectively (Fig. 6.11), built into the product family core. In order to accommodate built-in contract testing in the core framework we have to add the respective "tester component hooks." These hooks for the built-in contract tester components are modeled, designed, and implemented in exactly the same way as their respective functional versions. The entire resulting product line architecture with the built-in contract testing artifacts is depicted as a containment tree in Fig. 6.12. This shows the development-time organization of a product line

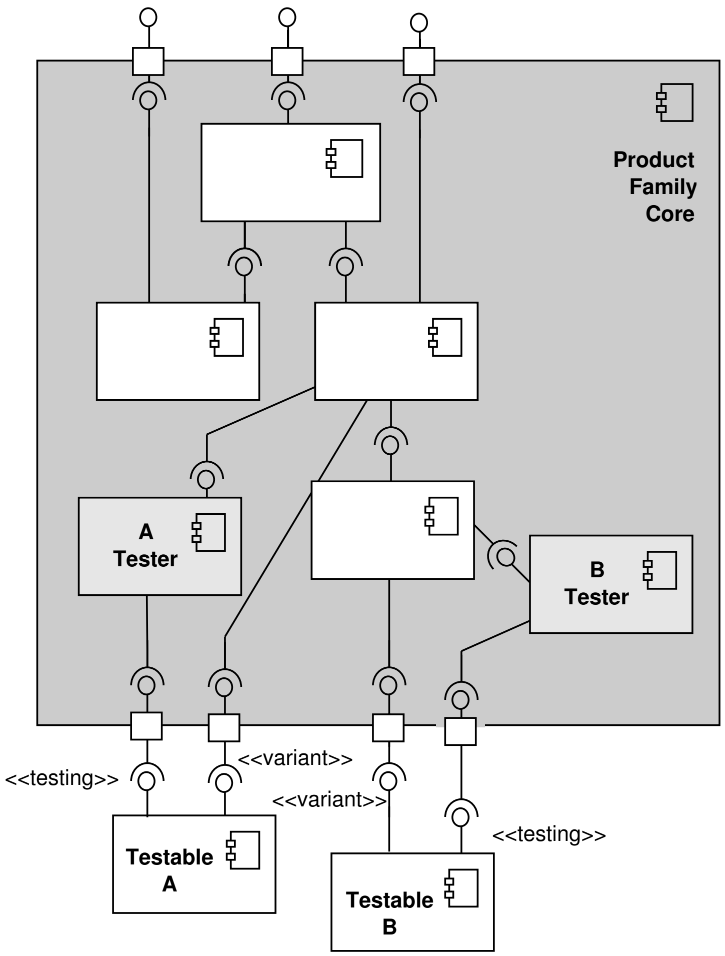


Fig. 6.11. Product family architecture with built-in contract testing components

architecture that comprises two products that can be instantiated out of the common core. If variant 1 will be instantiated, the framework will create the common core, and the components `TestableA` and `TestableB`; furthermore, because the tester components belong to the common core, the framework will have to create `ATester` and `BTester`. Instantiating variant 2 will then require the same steps for `TestableC` and `CTester`.

In Fig. 6.11, I have modeled the tester components as part of the product family core. In reality, they are better accommodated outside this encapsulation boundary, because it is unwise to build all tester components for all product line permutations into the product family core. If we instantiate a single application, we will need only the testing artifacts that are necessary

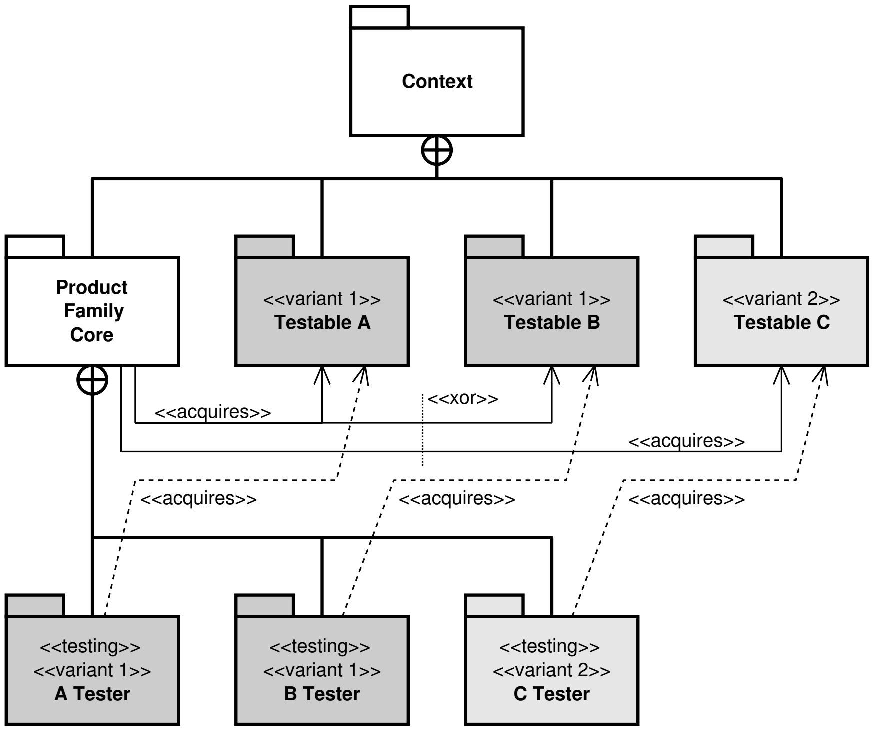


Fig. 6.12. Containment tree with the built-in contract testing artifacts for the product family core

for the application, nothing more. Any additional tester components will consume space. Figure 6.13 shows the alternative way of organizing tester components outside the encapsulating hull of the core component framework of a product line. For every variant component we will have a tester component that can be associated with the client component or the product line family core in the same way as the variant component. Every variant component is separated into two components, one that comprises the functionality for the variant features, and one that represents the testing for these variant features. The product family core will need two "component hooks" for each variant component, one for the original functional variant component and one for the tester of that component. The test cases in the tester components are developed according to the specification of the required interface of the product line core. This organization is more in line with the principles of built-in contract testing and component-based development.

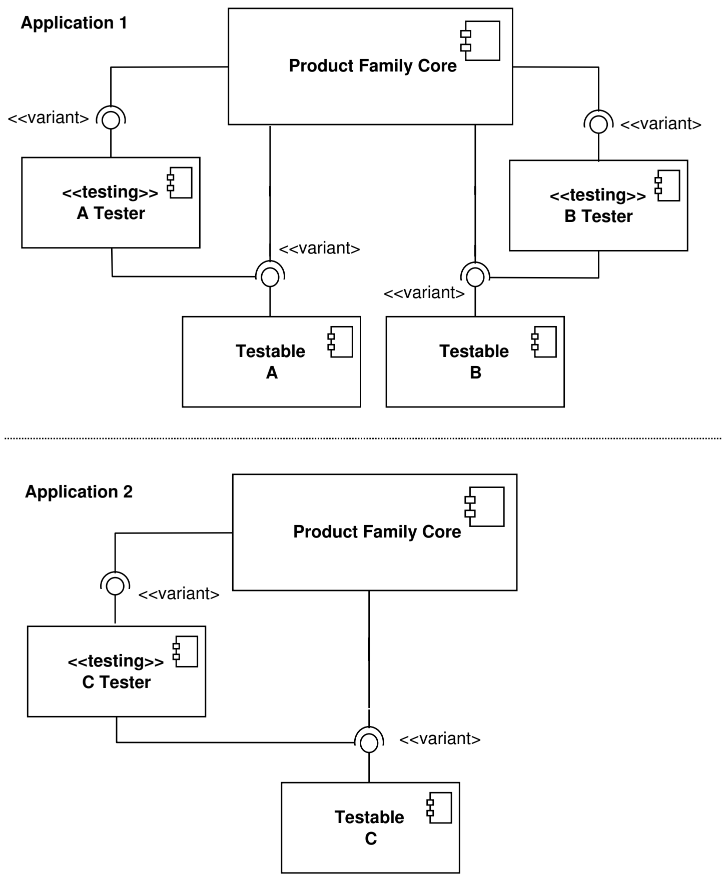


Fig. 6.13. Organization of the tester components outside the product family core

6.3.2 Testing as a Product Family Development

Built-in contract testing as I have introduced it and put forward throughout this book represents a typical product line engineering activity. Because the built-in contract testing artifacts require additional development effort within the functional system that we are constructing, a testing or testable system can be seen as a variant of the original functional system. In other words, we have an original system that provides the required functionality and behavior, plus an optional add-on to the system which provides testing and testability infrastructure. I have briefly introduced these principles in Chap. 5, and they are fully in line with the basics of product family development or product line engineering according to the KobrA method [6].

6.4 Summary

Reuse represents the most fundamental idea behind component-based development, and in its purest form building applications is solely based on assembling and integrating readily available third-party units. This comprises functional artifacts as well as built-in testing artifacts that should be readily provided and distributed with the respective components. In this chapter, we have concentrated on how built-in testing affects reuse and how it can supplement reuse-related technologies.

An important issue in this respect is that component manufacturers should readily distribute and openly provide their tester components, even if it seems redundant to re-test. There are a number of scenarios in which it can be advantageous to have tester components that the provider has distributed. I have identified some scenarios for which tester components can help during development time, in a platform test, and as built-in documentation for the component, as well as during runtime, in a server–client test.

Another advantage of applying built-in contract testing is that it supports customers of commercial components in component identification procurement and self-certification. As long as certification standards do not exist, it can considerably alleviate the earlier phases of component-based software development, that is, before the components are actually integrated into the component framework.

Product families realize component reuse at an architectural level, and they are regarded as providing the ultimate component reuse framework. In essence, a product family can be seen a normal application in which some of its parts are readily integrated for various final applications, and other parts are added or exchanged according to the final application that will be instantiated. Both represent typical scenarios for which built-in contract testing is suitable and applicable. Building the product family framework can be seen as component integration for which built-in testing may be afterwards removed, and instantiating an application out of the framework represents the typical contract testing scenario for which built-in testing will remain in place. In the next chapter we will discuss how quality-of-service requirements can be assessed under the component-based development paradigm.

Assessing Quality-of-Service Contracts

One of the most important motivations in practice for the application of object technology and subsequently component-based software engineering techniques is that new applications can be created with significantly less effort than in traditional approaches by assembling the appropriate prefabricated parts. However, contemporary object and component technologies are still some way from realizing the vision of application assembly, in particular when quality-of-service (QoS) issues are considered. With traditional approaches, the bulk of system integration work is performed in a distinct environment, the development environment, giving engineers the opportunity to pre-check the compatibility of the various parts of their system. This ensures that the overall application is working correctly in terms of functionality, behavior, and quality-of-service.

In contrast, the late integration implied by the assembly of readily deployable components means there is little opportunity to validate the correct functional and non-functional properties of the resultant application before deployment in the runtime environment. Although component developers may adopt rigorous test methodologies for checking component behavior, and deliver fault-free units, there is no guarantee that an assembly of such units will abide by the overall qualitative criteria that are required for the resultant application. In fact, even totally fault-free individual components that perfectly fulfill their individual qualitative measures may cause failures in the overall system if the platform does not provide the right support, or if the other components violate the overall qualitative criteria. The ARIANE 5 disaster is an example.

Compilers, configuration tools, and behavioral built-in contract testing can help by validating the syntactic and semantic compatibility of interconnected components, but they cannot check that individual components are functioning correctly in terms of an application's non-functional requirements, or quality-of-service specifications. As a result, component assemblies that may behave correctly according to expected function and behavior may not behave so well in terms expected response times, precision of the result, throughput, or other non-functional criteria. Such criteria are generally referred to as the overall quality of a service provided, whereby the service itself is determined by functional and behavioral aspects. How some of these additional non-functional, or quality-of-service criteria may be assessed in a component assembly is the subject of this chapter.

In the next section, Sect. 7.1, I will give an overview of quality-of-service requirements that affect component contracts in general and timing requirements in component contracts in particular. Because this chapter focuses primarily on real-time component contracts, in Sect. 7.2 we will have a closer look at how timing requirements have to be handled in component-based development. This includes how timing is specified and distributed over a number of objects and how these assemblies can be assessed in terms of their timings. Section 7.3 introduces the extended model of built-in contract testing that specifically addresses the problems of testing timing contracts between pairwise associated components. The extended model amends the basic built-in contract testing model from Chap. 4 by a special type of testing interface that provides timing notification, and a very specific tester component. The extended model addresses only execution time during application development, so the next section, Sect. 7.4, discusses the challenges of assessing timing contracts for dynamic updates that are performed during runtime. The subsequent section, Sect. 7.5, looks at how quality-of-service requirements can be carried out permanently, beyond deployment in component-based systems.

7.1 Quality-of-Service Contracts in Component-Based Development

Component contracts are not restricted to behavioral issues, as it may seem from the discussions of the previous chapters. They can come in four levels of increasingly negotiable properties according to Beugnard et al. [15]:

- Basic contract. This includes the signatures that a component provides or requires in terms of invocable operations, signals that a component sends or receives, and exceptions that a component throws or catches. This is also called syntactic contract, and it is specified either at the programming language level or through some interface definition language (IDL) provided by a component platform. This first level is required to make the components interact, and it is checked through the compiler of the language, the runtime system, or the underlying component platform, e.g., type checking that a Java compiler performs.
- Behavioral contract. This defines the overall functionality of a component in terms of pre and postconditions of operations and externally visible provided and required state transitions, according to Meyer's idea of design by contract [112] and Reussner's architecture by contract [135, 136, 137]. This second level is also termed semantic contract, and it is required to

achieve some meaningful interaction of two associated components toward a common goal. Built-in contract testing in the form that has been described so far in this book is aimed particularly at checking this type of contract.

- Synchronization contract. The behavioral contract assumes that services are atomic or executed as single transactions without specification of their sequences or interactions. The synchronization contract adds another dimension to the behavioral contract, which is the sequence or combination with which the interdependent operations of a component may be invoked. This is particularly important if different clients access a component's services at the same time. Throughout this book I have treated the second and third levels of component contracts as single entity, and referred to it as behavioral contract. In fact, a test of this additional dimension is also addressed through built-in contract testing in that we can have a number of different tester components that emulate the access of different clients. Method sequence and message sequence-based testing strategies are particularly useful for this type of contract.
- Quantitative contract or quality-of-service contract. A correct component integration according to the previously described contracts (levels 1 to 3) results in a system with correct syntactic and semantic component interactions. In other words, two components can perform some meaningful tasks together. Additionally, a quality-of-service contract quantifies the expected behavior or the component interaction in terms of minimum and maximum response delays, average response, quality of a result, e.g., in terms of a precision, result throughput in data streams, and the like.

QoS requirements for component assemblies compound the problems of integration testing in component-based application engineering. In the following sections we will have a closer look at response time as a QoS requirement and how it is dealt with during component integration and deployment. Later, in Sect. 7.5, we will also have a look at how response-time issues may be assessed permanently after application deployment, and how other quality-of-service requirements may be treated in a component-based development project.

Real-time requirements are typical for embedded systems, and they are affected or constrained not only by individual objects, but by all objects and object interactions that make up an application. This is illustrated through Fig. 7.1. A response-time requirement typically originates from a user requirement or the context of a system. In an embedded system this context is usually a natural process into which the software system is embedded, and which it monitors or controls. Such a high-level or system-level requirement is represented by ΔT_A in Fig. 7.1. Because of decomposition, a user transaction at the system boundary is usually realized through a number of lower-level transactions on subsequently nested components as indicated through the sequence diagram in the figure. Each client component, or the context at the highest nesting level, defines a response time bound to the services of its associated and nested server components. In a simplified form, the high-level timing requirement in Fig. 7.1 can be expressed as

$$\Delta T \geq \Delta T_A + \Delta T_B + \Delta T_C + \Delta T_D + ... \qquad (7.1)$$

ΔT represents the high-level real-time requirement defined by the context (equation 7.1). If the server component violates this requirement, the client component will likely fail to provide its own clients with their required response times.

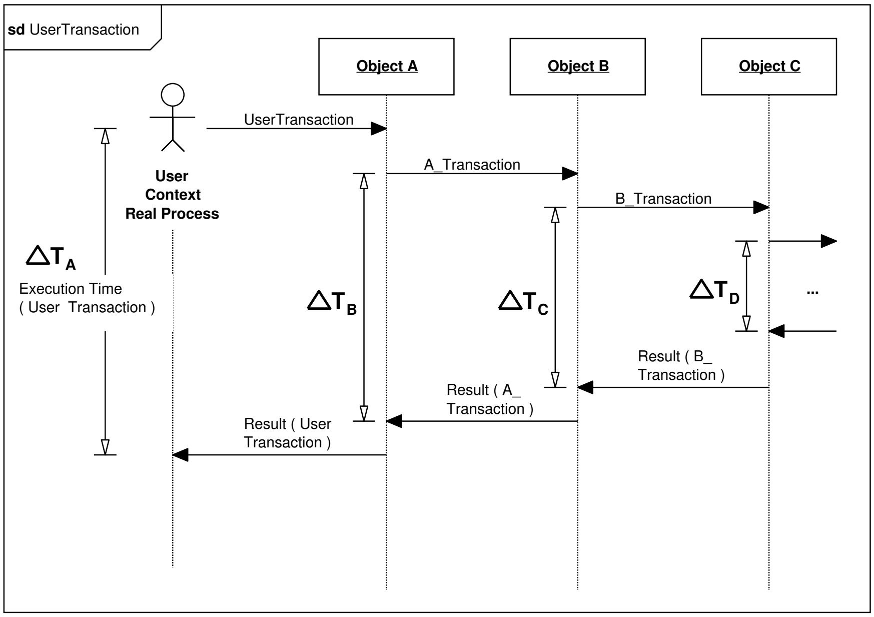


Fig. 7.1. Distribution of a high-level response-time requirement over multiple nested component interactions

Additional to the response-time dependencies between the components, each individual object or component may have a well-defined timing behavior on a particular runtime support system, or in a particular environment, for example, the development-time environment. This includes the underlying hardware platform as well as the operating system and its libraries, plus the component platforms that will show a different quality-of-service on a different hardware platform. Such dependencies are illustrated in the component diagram in Fig. 7.2. The QoS-contract that the *Component* in the center of Fig. 7.2 provides to its associated client depends on the quality-of-service that its own underlying runtime support platform (*ComponentRunTimeSystem*) as implicit server and the quality-of-service that its explicit server (*ServerRunTimeSystem*) provide. These dependencies are recursive. In other words, the quality-of-service that a component is providing to its clients can be only as good as the quality-of-service that the component is receiving from its associated implicit and explicit servers. QoS can never be assessed at the individual component level, and this makes the difficulty of guaranteed timing and timing analysis in component-based systems apparent.

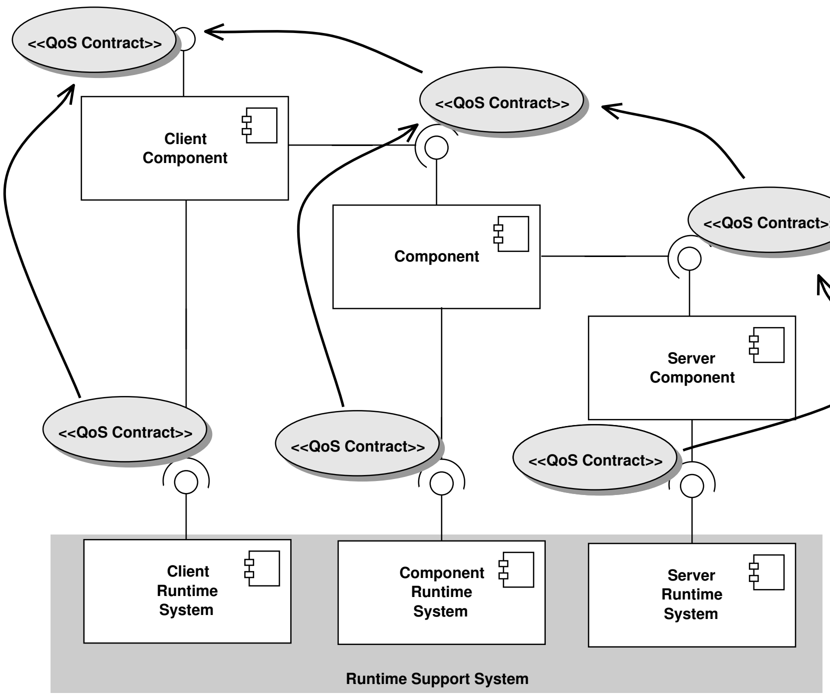


Fig. 7.2. Quality-of-service dependencies in a nested component hierarchy

The execution time of a component is completely changed if it is plugged to other components that implement functionality of a real-time application at a customer's site, for example, or if it is brought on a different platform with different underlying implicit servers and hardware. For any such changes in the implicit runtime system or in the associated explicit server components, the timing behavior of each possible combination of a component's feasible usage profiles with respect to these other components in situ on that particular platform must therefore be validated when the compliance to the timing schedule of such a system is assessed. Clearly, this cannot be done a priori for each component since the developer of that component can never anticipate its usage in a particular context. Execution-time analysis with an individual component is entirely meaningless. It can be performed only when components are assembled and put together into a new configuration on a new platform, although, the effort involved in plugging components together may be relatively small; therefore, the effort involved in validating the resulting assembly of components is showing the expected runtime behavior, may be much greater. This adds another dimension to the problem of component integration. Even if an entire application with all its component interactions readily in place, is moved to another platform, it must be completely reassessed in terms of its timing on that new platform.

7.2 Timing Analysis and Assessment with Components

Timing analysis in real-time system development is typically separated into two distinct activities that are supplementary:

- A constructive timing analysis is performed during component engineering to distribute high-level timing requirements over subcomponent functionality. This activity is essential for the schedulability analysis that enables the engineers to assign processing capacity of the scheduler to concurrent tasks. This can be supported by an adequate quality-of-service specification notation such as the QML [64].
- A validation timing analysis is performed to check whether the combination of all collaborating entities as a whole, including all scheduling issues, can satisfy its timing requirements. This is a test that verifies whether the first activity was successful and performed correctly.

Since this is a book on testing, I will concentrate only on the second item, that is, the assessment of whether a server component with all its included subcomponents will fulfill a client component's response-time requirements. However, the same principles of validation timing analysis can be applied to constructive timing analysis as well. I will give a brief overview on how timing issues must be addressed with respect to the first item.

Response-time requirements are derived directly from high-level user requirements, as I have illustrated in Fig. 7.1. The user of a component, or the customer of a system, is interested only in the response time at the component's or system's boundary, and whether this will fulfill the user's or customer's QoS requirements. He or she will not be interested in the timing of its individual parts or subcomponents. This is because the customer is interested only in a system's functionality and the quality of that functionality, e.g., its timing, and not in how the system or the component is organized internally. This is the case only if there is no user requirement stating that the system should be distributed.

The application engineer is responsible for the fulfillment of the application's real-time requirements at its boundary. And this depends on the timings of the individual components within a nesting hierarchy. In the following subsection we will look at typical problems that we have to address when we deal with timing requirements in component-based development.

7.2.1 Typical Timing Problems

For system developers or application engineers, the response times of the individual parts or components are important properties because they collectively define or affect the overall response time of the entire application as we have seen in Fig. 7.2. In general, the development of component-based real-time applications is performed in the same way as for applications without explicit response-time requirements This is the case at least for the decomposition activity in which the system is subdivided into finer-grained parts. An application cannot be decomposed into "the blue" because we have to consider already existing components that we will have to incorporate and reuse. In Chap. 2, I have emphasized that component-based development always represents an activity in two directions, decomposition, which is performed top-down, and composition, that is performed bottom-up. The first activity subdivides the system into sensible cohesive parts and the second activity tries to align the decomposed abstract units with already existing abstract component descriptions. Thus, decomposition is always oriented toward a specific goal, that is, the reuse of as many existing components as possible. This means that we will always have some existing parts of an application, even in the form of implementations in concrete programming languages. For dealing with real time in application development this means that we will not be able to change the response-time properties of these existing parts. This is even more difficult if these parts are third-party products. The only thing that we can do to vary the response times of these components is to select differing implementations that specifically address response-time issues, wherever this is feasible. For example, we could choose a component that is implemented in a faster language, or is realized with faster algorithms. However, this fundamental problem of response-time distribution between multiple interacting instances will always remain open.

The fact that we select a distinct component implementation unchangeably determines that for all the other remaining components there will be less leeway and choice in terms of their response time-specifications. This is illustrated in Fig. 7.3. If we select object A from a component vendor, its runtime will restrain the selection of the subsequently selected objects B and C. Depending on A's runtime without the nested components, $\Delta T_A - \Delta T_B$, we will have only ΔT_B time units left for the response time of the other components, B and C. This is valid only for a particular transaction that is realized through multiple objects. Any other transaction in which each of the components A, B, and C is participating needs to be assessed in the same way.

In order to achieve correct timing according to an application's real-time requirements, it is essential that all component integrations or all pairwise relationships between components be assessed in combination. This is in fact a search or optimization problem. What we are looking for is a feasible combination of component implementations that collectively satisfy all timing requirements. Here, feasible means that the combination of component implementations satisfy all the functional requirements, and this is a prerequisite for assessing the timing. It may be the case that a component's response time is acceptable if it is invoked as part of one particular user transaction, but it may not be acceptable in another transaction.

Such a problem is difficult to resolve. We could subdivide the two functionalities into two independent components. In that case we would effectively change our focus from data cohesion, which a component in fact represents, to something that we might call "real-time cohesion." In any case, we would have to intervene in the development beyond the component level and devise our own implementations, although this is not really what we would like to do in component-based application engineering. As a conclusion I have to admit that constructive timing analysis in a component-based development process is still challenging, and we will need considerably more research in that field in the future. Validation timing analysis it is a lot easier to perform

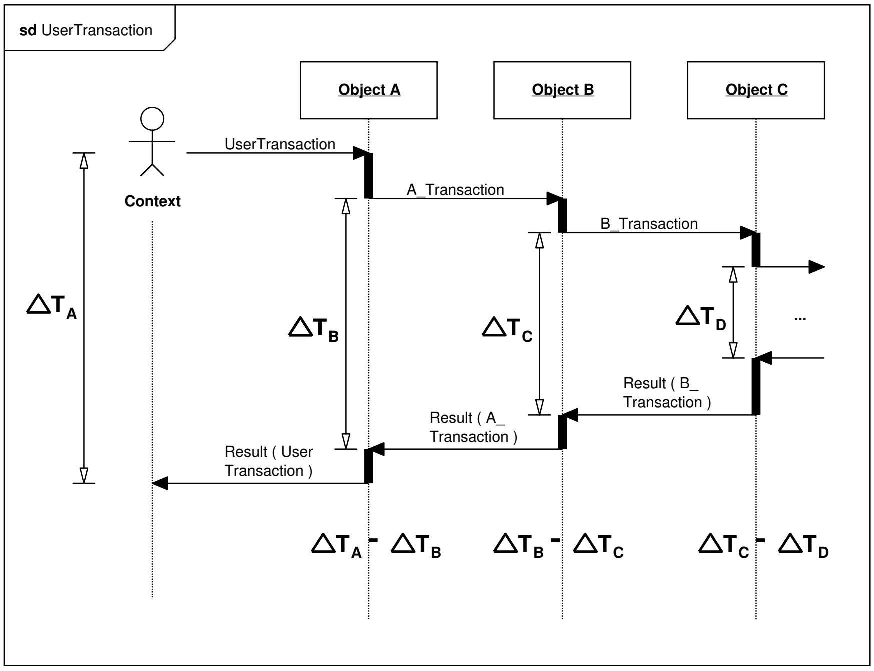


Fig. 7.3. Every additional component implementation restrains the selection of remaining components

in component-based application development. How this is done is laid out in the next section, after we have had a look at the different ways of performing timing analysis, and these are applicable for both, constructive and validation tasks.

7.2.2 Timing Analysis Approaches

Timing analysis can be performed in two oppositional ways: static timing analysis and dynamic timing analysis. I say oppositional not because they cannot be applied in tandem or cooperatively, but because they approach their subject, that is identifying a time bound, from two opposing directions. While static timing analysis estimates a time bound, for example, the worst-case execution time (WCET) of a task or transaction, from more pessimistic values, dynamic timing analysis approaches a time bound from more optimistic values. In other words, if static analysis is performed correctly, it always yields an overestimate, and dynamic analysis always yields an underestimate of the real time bound. This means, a time bound can never be determined exactly because of the inherent complexity of typical systems. We are able to retrieve only an estimate through timing analysis. This is illustrated in Fig. 7.4, and explained in the following paragraphs.

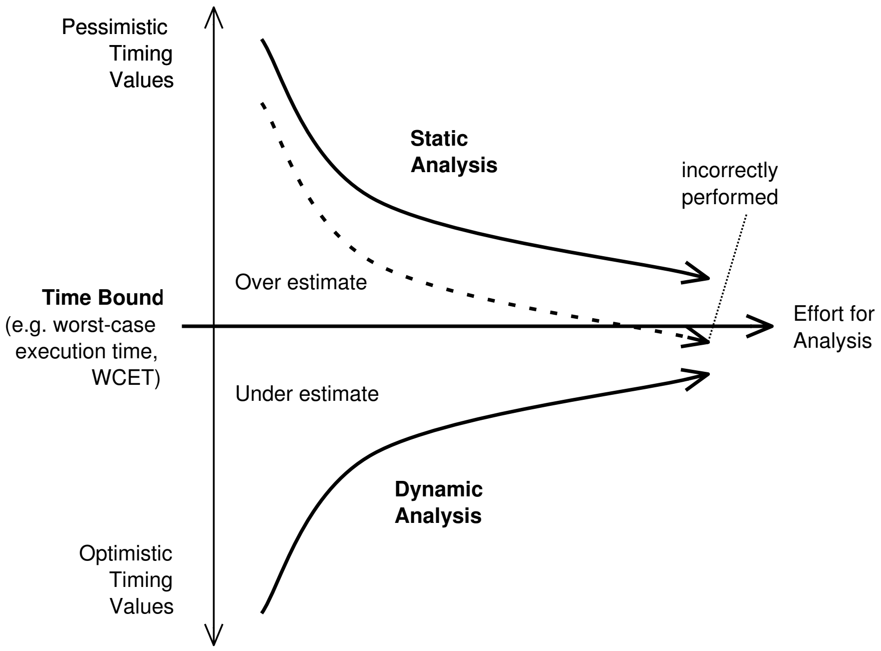


Fig. 7.4. Different ways of approaching a time bound in static and dynamic timing analysis

Static timing analysis refers to the traditional technique for analytically determining the best-, worst-, and average-case execution times of a task [71]. It follows a procedure that initially determines the required time for executing each basic block, that is, a segment of straight-line assembly code or a high-level language instruction, and then combines these into a cumulative total that represents the execution time for an entire task [117]. It is called static timing analysis, since it statically analyzes the code of a task for possible execution paths, and models the time of the code on the target hardware without executing the task or considering input parameter values [130]. In contrast with the static approaches, dynamic timing analysis only concentrates on the execution of a real-time task in its real environment on real hardware. In other words, the task or transaction is executed as a test and its response time is measured. Hence, dynamic timing analysis represents an experimental approach.

Neither approach is ideal, but there are no other ways to assess timing behavior. Static analysis, e.g. [127, 128, 129, 131], is difficult to perform, requires extensive human interaction, and is therefore also prone to error. For example, it requires a number of prerequisites such as knowledge about maximal iterations in the code, absence of recursion and function variables (which in fact restricts some object technology features), feasible sequences during execution, and path information. These items should be readily available and complete to guarantee correct static analysis [117, 122, 131]. An advantage of static analysis is that it always overestimates a time bound if the analysis is carried out correctly. As a consequence, it is suitable for safety-critical applications or so-called hard real-time applications in which the violation of a timing requirement will inevitably lead to a hazard, e.g., loss of life, or damage to the environment. The dynamic analysis approach is quite the opposite. Since it always underestimates the true time bound, it can never guarantee worst-case timing according to the specification of a safety-critical system. In general it must be considered unsafe for hard real-time system construction, and it is typically better suited for the so-called soft real-time systems for which the violation of some deadlines is acceptable.

Due to the fact that static solutions are so difficult, and adequate tools are still lacking, dynamic approaches are penetrating the real-time system domain. Dynamic timing analysis is easy to apply because it represents black box-style test case execution with time measurement. Additionally and more importantly, dynamic timing analysis does not come with any of the inhibiting factors of static analysis that defeat its application in modern object-oriented and component-based real-time application development. Dynamic execution does not require any internal, or implementation-specific information of the object under scrutiny. This is probably the single most important inhibiting factor for the application of static analysis in modern development technologies, such as component-based application engineering. On the other hand, dynamic timing analysis requires that the real application be readily available, including hardware and software components. In other words, before we can start with dynamic timing analysis, we have to have the implementation of the software system in an executable form and the real underlying hardware components, or at least simulators for them. This is often difficult in the real world because hardware and software are developed at the same time, or the software is even developed earlier. So far, to my best knowledge, there is no suitable generic methodology available for tackling these hardware/software integration and co-design issues. Since this is a book on testing, I concentrate only on dynamic timing analysis and how this can be applied to built-in contract testing. In the following section, I describe how the basic model of built-in contract testing may be extended to accommodate the capability to assess timing contracts in a way similar to that for behavioral contracts.

7.3 Extended Model of Built-in Contract Testing

The problems of component integration testing are compounded if QoS-requirements are considered, as we have seen earlier. Each individual object or component shows a very specific timing behavior in any arbitrary deployment environment. This environment includes other associated components as well as the underlying hardware platform. This very specific timing behavior is completely changed if the component is deployed in another environment with other associated client and server components or a different underlying hardware. Any other client component will apply a different usage profile and thus change the timing of the component considered, and any other server component (including an implicit server) will change the timing of its clients. A timing validation of a component can therefore never be done a priori. The producer or vendor of a component can never anticipate its usage in any arbitrary feasible deployment environment of any customer of that component. In the same way that isolated component tests are meaningless for behavioral contracts, they are even more so for QoS contracts in general and timing contracts in particular. Timing assessment can only be performed in situ, when the components are assembled and put together in a new configuration [75]. The basic model of built-in contract testing offers a feasible approach for assessing the correctness of behavioral contracts between pairwise associated components, and it may also provide a solution for addressing the assessment of real-time contracts between two such entities.

The essential difference in testing between the traditional procedural paradigm and the recent object-oriented development paradigm lies in how data and functionality are treated. We have discussed this in Chap. 4. While the procedural approach propagates a strict separation of data and functionality, the object or component paradigms encourage the opposite, that is, the combination and encapsulation of data and functionality in a single entity. This creates a fundamental difference in how modules of the two kinds are treated during testing. In object-oriented development, testing comprises the notion of an object state, so that a typical test for an object encompasses:

- an initial state defined through the combination of the object's internal attributes,
- an event with some input parameters,
- some output and behavior,
- some expected output and behavior, and a final state.

These items are part of any test case, for behavioral contract testing as well as for dynamic timing contract testing. Additionally, for timing assessment, we need to introduce the notion of execution time for an event, or a sequence of events, e.g., a transaction. The test case as such does not change much if we consider the test of a behavioral contract and a timing contract. What changes is the validation action of the test that will comprise some execution time measurement, and the event will have an auxiliary response-time specification. As a consequence, for the assessment of timing contracts between two associated components, we can have in principle exactly the same setup as for the basic built-in contract testing model. We can have a testing interface for the server role, for state setting and state checking, and a tester component at the client role of the association that contains and applies the test cases. Fig. 7.5 shows the extended model of built-in contract testing that is aimed at checking timing contracts. In the following subsection we will have a more

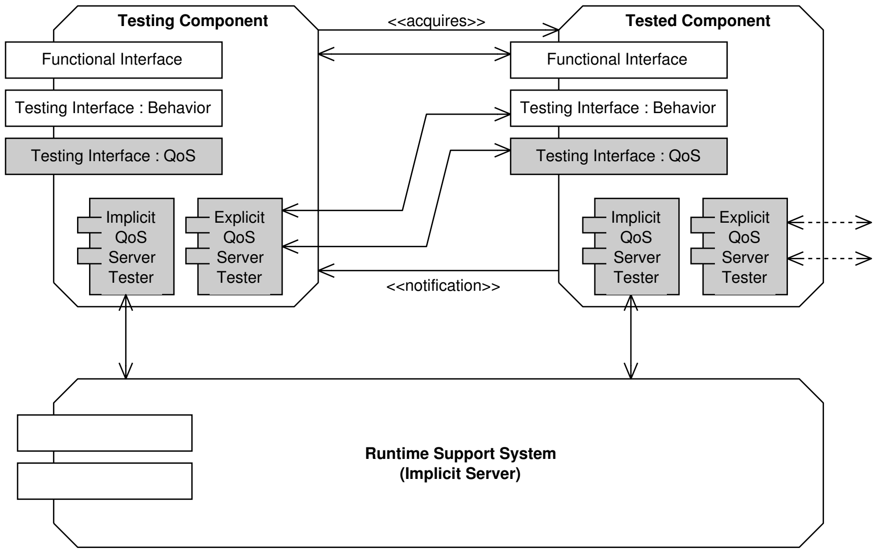

Testing Component
<<acquires>>
Tested Component
Functional Interface
Testing Interface : Behavior
Testing Interface : QoS
Implicit QoS Server Tester
Explicit QoS Server Tester
Functional Interface
Testing Interface : Behavior
Testing Interface : QoS
Implicit QoS Server Tester
Explicit QoS Server Tester
<<notification>>
Runtime Support System (Implicit Server)

Fig. 7.5. Extended model of built-in contract testing

detailed look at how the two artifacts, testing interface and tester component, need to be organized, and how they differ from the basic contract testing model.

7.3.1 Testing Interface for the Extended Model

In the initial model of built-in contract testing, I introduced the concept of "abstract states" that users of a component need to know to use it properly. These abstract state specifications can be used to derive a testing interface for the behavioral built-in contract testing according to the built-in contract testing development process described in Chap. 4. An abstract state is a domain of internal attribute settings for which an operation invocation shows the same external behavior (i.e., state transition). The response time of a component, or its timing behavior, is defined through its current concrete or physical state, which also includes the state of the underlying hardware and the input parameters of the timed event at the time before the event is invoked. So, for timing assessment, knowledge of or access to the concrete values that determine the internal states are essential. We have to deal with the physical state values in contrast with the behavioral contract test where the abstract state is often sufficient at least for checking the postcondition. The testing interface of a component according to the basic model of built-in contract testing must permit the setting of concrete state values to be useful for the assessment of the component's response time. Otherwise we have to fall back to the component's normal functional interface and set its internal state, as required for a particular test, through a sequence of normal operation invocations. In this case we have to carry out a distinct history of operation invocations with concrete input values to set the timed component to a particular physical state. Then, we have to invoke the operation or the sequence of operations for which we would like to assess the response time. For the second case we do not need the testing interface of the basic built-in contract testing approach.

Timing Measurement

The additional testing interface according to the extended model for QoS testing comprises timing measurement facilities. The additional QoS testing interface depicted in Fig. 7.5 provides operations that control, set, and read timers that the underlying runtime environment of the tested component implements. In theory, timing could well be measured inside the tester component at the client side because invoking and evaluating the timer is a typical process of the test software. However, if the tested and timed component is residing on a different network node than the tester component that provides the test scenarios for the tested component through some network, in addition, to the response time of the component's operations, we also measure the response time of the underlying networking infrastructure. Fig. 7.6 illustrates this. The timing testing interface thus enables remote testing. This is important in two particular cases:

- The tested component is running on an embedded controller that provides real-time features, and the platform of the testing system does not support

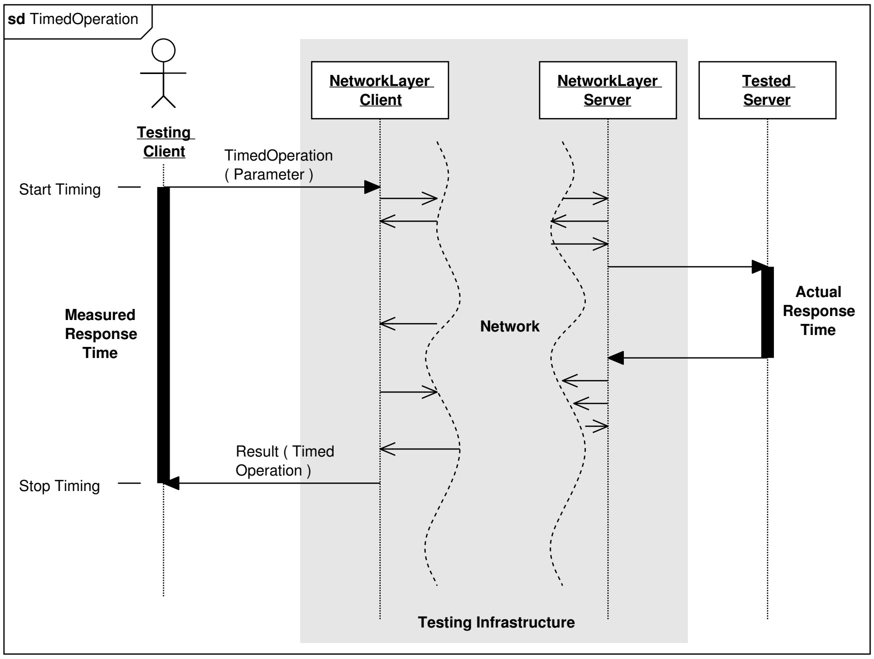


Fig. 7.6. Problem of timing measurement at the client role

real time. We are interested only in the timing of the component that runs on the embedded controller, so we have to measure that timing. If we measure the timing in the tester component, we assess the response time of the embedded component, the response time of the network connection, the runtime of the tester component, and some operating system activities of the tester component. This is clearly not what we are aiming for.

- The tested component is running on an embedded controller that does not provide the space for the tester component and the testing infrastructure.

Figure 7.7 shows the structural models for the extended testing interfaces of the Resource Information Network client and server components. The testing interface of the extended model for timing assessment can be realized as implementable interfaces (e.g., in Java), indicated by the ≪interface≫ stereotype. The good thing here, compared with behavioral contract testing, is that this additional interface may be the same for all components.

7.3.2 Tester Component for the Extended Model

The tester component at the client role is responsible for applying the test cases that are used to assess the timing of the server. This is the same principle as that for the basic model of built-in contract testing and checking the

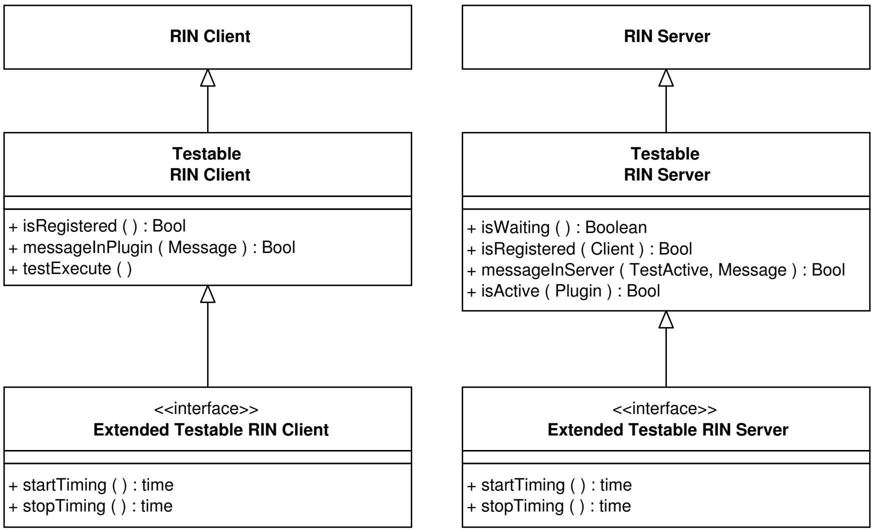

RIN Client
RIN Server
Testable RIN Client
+ isRegistered ( ) : Bool
+ messageInPlugin (Message) : Bool
+ testExecute ( )
Testable RIN Server
+ isWaiting ( ) : Boolean
+ isRegistered (Client) : Bool
+ messageInServer (TestActive, Message) : Bool
+ isActive (Plugin) : Bool
<<interface>> Extended Testable RIN Client
+ startTiming ( ) : time
+ stopTiming ( ) : time
<<interface>> Extended Testable RIN Server
+ startTiming ( ) : time
+ stopTiming ( ) : time

Fig. 7.7. Timing testing interface for the RIN system client and server components

behavioral contract between the two roles. Initially, these behavioral tests may also well be used for assessing the timing contract. They need to be somehow amended with the calls to the timing measurement operations at the server side that I introduced in the previous section.

We can implement the test cases in a way that they will always call the timing measurement operations, regardless of whether execution-time analysis is needed or not. Alternatively, the test cases in the built-in contract tester component may be organized in a way that the calls to the server's timing measurement interface can be turned on and off during runtime, according to which tests are required. This would imply that the testing interface of the client component whose tester performs the timing test on the server provides some way of engaging and disengaging a particular testing mode. The execution time for each transaction between a client and a server can thus be assessed relatively easily, by reading a timer before and after the transaction is invoked. Additionally, the tester component needs to provide a notification interface through which the tested (server) component can notify the tester component that the timed operation has been performed. This is important mainly for asynchronous interaction between client and server. Both interfaces, required for both roles in the extended model of contract testing, are depicted in Fig. 7.8.

Although this simple solution of measuring the execution time of the existing behavioral test cases might be appealing at a first glance, it is not sufficient to assess the execution times that are important in embedded real-time application engineering. Each component will have a number of timing

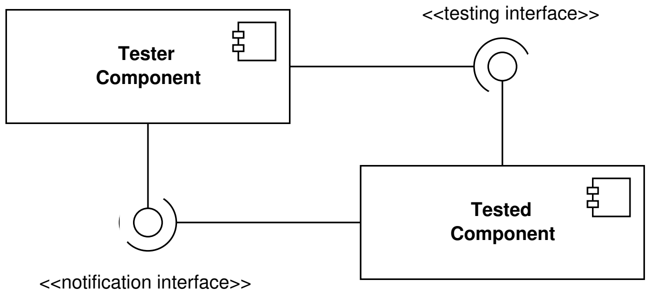


Fig. 7.8. Timing notification interface of the tester component and testing interfaces of the tested component in the extended BICT model

requirements that will have to be assessed in a test. These typical execution-time bounds are laid out in the following:

- Average case execution time (ACET). This time bound represents the average response time of a transaction between two roles. ACET is not a typical real-time requirement. It is actually more important for assessing the throughput of a system and for performance bottleneck analysis.
- Worst-case execution time (WCET). This is the most typical time bound for real-time systems. A violation of this timing requirement through a server means that the provided service is too late or the execution of a transaction is too slow to be useful for the client. A violation of this bound in a hard real-time system leads inevitably to a hazardous situation.
- Best-case execution time. This is another not so well-known time bound whose violation through a server means that the provided service is too early, or the execution of a transaction is too fast to be useful to the client. This will also lead to a hazardous situation in a hard real-time system.

While the role of the WCET is quite clear, it is often not so clear for the BCET. Executing too fast for a real-time system initially seems to be alright, but in many cases it is not. This becomes apparent if early results are buffered for later use by another process. If the process that produces a result is too fast, say twice as fast as the process that consumes the value, we will lose every second result, and it depends on the specification whether this is acceptable or not. So, synchronization is a key issue in real-time systems development. BCET and WCET determine the range in which response times of a component transaction are acceptable. This clearly emphasizes that real-time systems engineering has nothing to do with attaining fast execution, but with attaining the right response at the right time, not too early and not too late.

If we look at these time bounds, it becomes apparent that we gain nothing by measuring the execution times for the original samples of the behavioral contract tests in the way that I have described earlier. Each of these behavioral tests will yield only one execution time value for one particular sample, or for one particular input parameter combination. This is only one single sample out of a huge number of samples that are feasible for each test case. Because the input parameter values of a transaction determine its execution time, we have to execute a number of different parameter settings for each test and compare them. From the individual transaction executions we can determine the average time and pick the largest and smallest values.

The easiest way to generate a massive number of input parameter values is to use a random generator. In a loop, this will generate a set of parameter values for a transaction, apply them, and measure the time for a single execution of the transaction. The best-, worst- and average case-timing values can be permanently updated through simple comparisons and calculations. The following pseudocode illustrates such a test case.

```
begin test case
  timing sum = 0
  best case time  = max
  worst case time = min
  <<set optional state>>
  for number iterations
    randomize parameter list
    start timing
    invoke transaction ( parameter list )
    stop timing
    execution time  = stop timing - start timing
    timing sum      = timing sum + execution time
    best case time  = min ( best case time, execution time )
    worst case time = max ( best case time, execution time )
  end for
  average case time = timing sum / number iterations
end test case
```

Each original test case for behavioral contract testing thus maps to a volume of randomly generated test cases for which the execution time will be assessed. The original tests still represent a very valuable source for dynamic timing analysis because they will ideally define a complete usage profile for the tested component. The pseudocode above illustrates only a test case that does not consider any state-based testing. If we are dealing with states, and state transition testing, we have to take into account any operation prior to the transaction call, or the so-called invocation history, that is used to bring the tested component into its initial state for the test under scrutiny. I wrote earlier that the execution time of a component's transaction is determined through its own input parameter values and the component's internal state variables, and these are determined through the operations that have been invoked prior to the transaction. In this case, we have to replace the following pseudocode, for the stereotype ≪set optional state≫ in the pseudocode for the test case.

```
begin set optional state
  randomize parameter list 1
  invoke history transaction 1 ( parameter list 1 )
  randomize parameter list 2
  invoke history transaction 2 ( parameter list 2 )
  ...
  randomize parameter list n
  invoke history transaction n ( parameter list n )
end set optional state
```

Each history transaction in the pseudocode represents one operation invocation on the tested component that is adequate to bring the component into the desired state. It is quite common that many transactions according to the history need to be invoked before we can apply the actual tested and timed operation.

Although random testing is the most commonly used test case generation technique for assessing timing behavior today, it is not particularly good at it. In the following subsection we will have a look at a more advanced and effective technique for dynamic timing analysis and assessment, an evolutionary algorithm.

7.3.3 Optimization-Based Timing Analysis

Dynamic timing analysis and assessment can actually be regarded as a search or optimization problem. The goal of the optimization is to determine test cases that satisfy a specific test criterion. For timing analysis the test criteria are WCET and BCET. In other words, the task is to find and identify the parameter settings that represent the worst-case or best-case execution time of a component's transaction. Random testing belongs also to this group of search or optimization techniques, although it is merely a very simple one. More powerful representatives of this group are genetic algorithms (GA) [68], evolution strategies (ES) [151], and simulated annealing (SA) [164]. These belong to the group of evolutionary algorithms, and they are all characterized by the fact that the generation of new prospective solutions is based on the properties of old solutions for the optimization under consideration. The application of evolutionary algorithms to typical testing problems is termed evolutionary testing [171], which is boosted by a relatively new and vivid community and reflected by a number of publications in this area such as [72, 75, 130, 172, 173].

The ideas of evolutionary algorithms are loosely related to the mechanisms of natural evolution, which is founded on selection and reproduction, hence the name. They operate on populations of binary strings (GA) or real numbers (ES) that represent possible solutions to the search problem. Evolutionary algorithms recombine and mutate these strings, which are also called individuals, according to predefined operators, and the new individuals resulting are selected as a new generation according to a so-called fitness function. The operation of an evolutionary algorithm is characterized by three steps, reproduction, mutation, and selection that are carried out with a population. During reproduction, pairs of individuals are selected for recombination according to a selection strategy, and some parts of their strings form new individuals. This is termed crossover, and is controlled by the crossover operator. Some positions in the new individual are mutated according to a mutation operator. As an effect, recombination retains the information that is inherent in the entire population, and thus exploits the search volume that the population occupies. In contrast, mutation introduces new information into the population, and thus explores new locations of the search volume. Recombination and mutation yield new individuals which are tested and their fitness evaluated through the fitness function. This function assesses how well each individual solves the original search or optimization problem. Usually fitter individuals have a higher chance of being selected and recombined. The fittest individuals remain in the population and build the basis for the next iteration in a new generation. This procedure can be represented by the following pseudocode, with P, P1, P2, and P3 representing sets of feasible solutions or, in other words, populations:

```
begin ea
  initialize (P);
  while not breakCondition do
    P1 = selection (P);
    P2 = recombination (P1);
    P3 = mutation (P2);
    P = fittest (P3, P);
  end while
end ea
```

The iteration in this procedure is repeated for a predetermined number of times or until the stopping criterion is met, for example, after a number of generations without improvement. This evolutionary process generates new solutions based on information of existing solutions, so that later generations are likely to consist of fitter individuals that represent better solutions to the original optimization problem [71]. The evolutionary operators control the performance of the algorithm. A population can be regarded as a mapping of the problem space with each individual occupying a distinct location in this multidimensional volume. The number of optimized parameters determines the size of the search space. The crossover and mutation operators control the degree of exploitation and exploration of the search space by a population. The selection operator determines the individuals that are selected for recombination. Greater fitness of an individual represents a greater chance of being selected.

In order to realize dynamic timing analysis with an evolutionary algorithm, the optimization procedure can be used as a test case generator in the same way as the random generator. The optimization process generates input parameter values which determine the dynamic behavior of a component's transaction, and thus its response time. Each individual of the evolutionary algorithm corresponds to one set of input parameter values for one single invocation of the tested transaction. This also comprises the input parameter values for the invocation history that needs to be called to set the component to the initial state required for the test. The fitness function executes the test case with the set of inputs that is provided by the individual, and measures the execution time. This time is the only value that drives the optimization process. Individuals that produce longer execution times for WCET, or shorter times for BCET, are favored by the selection operator of the evolutionary algorithm. The recombination of highly fit individuals tends to produce highly fit offspring; this has been proven many times, e.g., [90]; this offspring is allowed to "spread through the population." Subsequent generations therefore consist of "fitter" individuals which produce longer execution times or shorter execution times, depending on the test target. Figure 7.9 illustrates the development of a typical evolutionary test for BCET and WCET compared with random testing. While the random test oscillates within a certain range and never reaches the values of the more sophisticated evolutionary algorithm, the evolutionary test approaches the extreme execution times noticeably, quickly and steep at the beginning, and flattened toward later generations. This picture is typical for most evolutionary tests [72].

7.3.4 Application to the RIN System

In the following paragraphs, I give a brief overview of how evolutionary testing can be applied to check the runtime behavior of the Resource Information Network (RIN) that I introduced in Chap. 4. I will concentrate only on one transaction that a user of the RIN system may invoke to illustrate how an individual for the evolutionary algorithm needs to be encoded for this type of testing.

The application, the user of the RIN system, defines a maximal or minimal expected response time, or a so-called latency requirement for an entire transaction that reads information about the state of an associated network device's physical memory. The application is the client of the RIN system. It has no knowledge of the RIN system's implementation and how it would possibly satisfy the client's response time requirement. The RIN system performs its own internal processing and interactions with other entities, so that the execution time is distributed among all entities that are participating in providing the requested service. This corresponds to schedulability analysis though the client is interested only in getting the requested service within

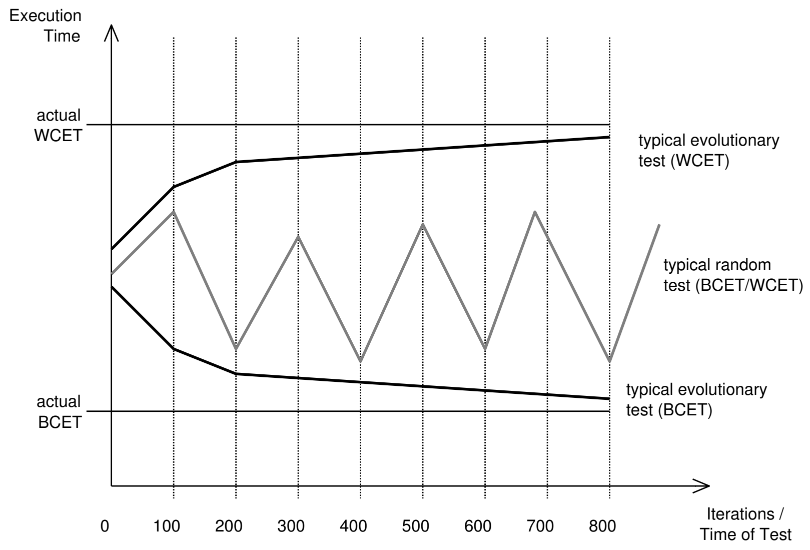


Fig. 7.9. Typical development of evolutionary testing compared with random testing

the expected and specified time bound of the latency requirement. The following list shows typical usage scenarios for the RIN system that the client application might want to invoke [72].

- The client creates a RIN server instance on the remote RIN host.
- The RIN server adds the available plug-ins to the RIN server's plug-in database.
- The RIN client instantiates a RIN server processing object in the server instance and a callback object in the client instance.
- The RIN client calls `RegisterCallback()` on the RIN server processing object and associates the callback object with that particular processing object. The registration comprises user name, application name, and host name. The `RegisterCallback()` operation will also cause the RIN server to add the client with its registration data to its internal connected client database.
- The client calls `ProcessRequest()` on the server processing object to enqueue the requested transaction to be further processed by the server and consequently the plug-in. This is the actual transaction that the application is interested in.
- The server performs a syntax check on the request sent by the client.
- The server calls `ProcessData(Message)` on the respective server plug-in component.
- The plug-in performs a syntax check on the plug-in request sent by the server.

- The plug-in performs the memory request on the local host's physical memory.
- The plug-in calls `OnDataFromPlugin(Message)` on the server component and notifies the server that the request has been processed.
- The server checks the syntax of the message received by the plug-in.
- The server calls `ReceiveDataFromServer(Message)` on the client's callback object.
- The client sends a signal to the application and notifies it of the message arrival.
- The application reads the message contents from the callback object. This is the end of the transaction that the application is interested in.
- The client closes the connection to the server and removes its own callback object.

The task of the search algorithm is to find the combination of operation invocations, with their respective input sets, that represents the worst-case or best-case execution time of a specified transaction or transaction sequence. This is one test case for WCET and one for BCET, out of a large number of possible test cases. In order to apply a genetic algorithm to this task we have to find a way of representing a test case in the form of a binary string, which is the basic item on which a genetic algorithm operates. Let us assume that the usage profile of the RIN system's client comprises a sequence of five different operation calls that the client calls on the RIN system by using the provided method `ProcessRequest(Plugin,Message)`. These five operations may be called by the client in any arbitrary sequence. The UML sequence diagram for the operation `ProcessRequest` is depicted in Fig. 7.10. The client of the operation is interested only in getting the response time for the entire execution of this operation, although it is separated into a number of different internal operation calls on a number of underlying objects. Each of the five operation invocations that the client of the RIN system performs has the following format:

```
ProcessRequest ( Plugin, Message );
```

`Message` has the following format:

```
FUNC, PARAM1 [, PARAM2]
```

We have a test case with five sequential operation invocations of `ProcessRequest`, e.g,

```
ProcessRequest ( Plugin, FUNC1, PARAM1, PARAM2 );
ProcessRequest ( Plugin, FUNC2, PARAM1, PARAM2 );
...
ProcessRequest ( Plugin, FUNC5, PARAM1, PARAM2 );
```

The variable FUNC is represented by the following plug-in operations:

```
FUNC = {BYPASS, REPEAT, ABSTIME, ASAP, COMPARE}
```

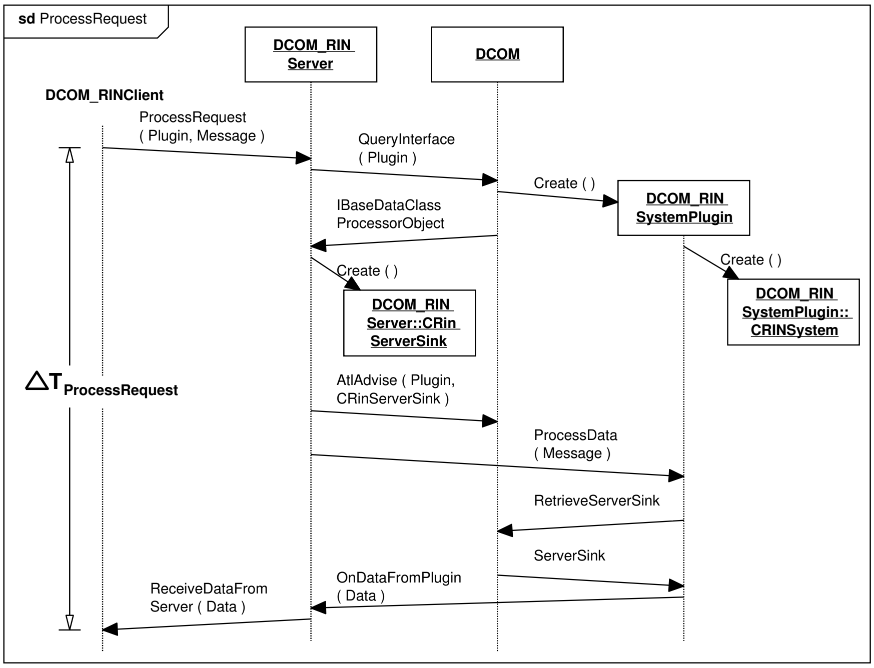

sd ProcessRequest
DCOM_RINClient
DCOM_RIN Server
DCOM
DCOM_RIN SystemPlugin
DCOM_RIN Server::CRin ServerSink
DCOM_RIN SystemPlugin:: CRINSystem
ProcessRequest (Plugin, Message)
QueryInterface (Plugin)
Create ( )
IBaseDataClass ProcessorObject
Create ( )
Create ( )
ΔT ProcessRequest
AtlAdvise (Plugin, CRinServerSink)
ProcessData (Message)
RetrieveServerSink
ServerSink
OnDataFromPlugin (Data)
ReceiveDataFrom Server (Data)

Fig. 7.10. UML sequence diagram for the RIN system's operation *ProcessRequest*

The variable PARAM1 can take the following values:

```
PARAM1 = {MemoryLoad, TotalPhys, AvailPhys,
  TotalPageFile, AvailPageFile, TotalVirtual,
  AvailVirtual}
```

Finally, the variable PARAM2 can take a 16-bit integer value plus one of the following operators:

```
==, !=, <=, >=, <, >
```

These last parameters are required to determine the memory size when the function `COMPARE` is called. It is used to assess whether an associated server provides enough memory space for performing a specific task.

Each individual in the search algorithm represents one sequence of five consecutive `ProcessRequest` calls with the respective input parameters. In our case, the first parameter `Plugin` is always constant since we intend to assess the timing behavior for the RIN's memory plug-in in the context of a particular application or configuration. The encoding of an operation call needs to be binary if we are using a genetic algorithm as test case generator. So, the input parameters for `ProcessRequest` have to be brought into a numerical representation that can be expressed as a binary string. The message that

`ProcessRequest` forwards comprises `FUNC1` to `FUNC5`, which can be expressed by five values, each an element of [0..4]. The same principle applies to the other parameters. We can encode `PARAM1` as a number in [0..5], and `PARAM2` as a number in [0..7]. For example, the representation $[4, 0, 3, 256]$ for an individual in the search algorithm will be expressed as the following operation invocation:

```
ProcessRequest ( MemoryPlugin,
                 ‘‘COMPARE’’,
                 ‘‘MEMORYLOAD’’,
                 ‘‘>=’’,
                 256
               )
```

The size for the second parameter will be represented by a 16-bit integer. In this example, the number 256 represents the size of the requested memory. Every test case will comprise five of these encodings, each for one operation invocation of `ProcessRequest`. These are subject to selection, recombination, and mutation within the genetic algorithm. The fitness function transforms the binary representations in real operation invocation sequences, calls them and measures the execution time for each such sequence. This is fed back into the genetic algorithm to determine the “goodness” of an individual with respect to best- or worst-case timing.

The following parameters for the genetic algorithm have been used for performing the timing tests of the previously introduced usage profile of the client; they are based on experience. The algorithm used is my C++ implementation of a GA [71]:

- Number of parents in the population: 12
- Number of children in the population: 12
- Number of individuals: 24
- Number of generations: 40
- Number of executed tests: 800
- Crossover probability, Pc: 0.3
- Mutation probability, Pm: 0.02
- Tournament selection: 0.5, tournament size = 3

The following list shows the software encoding for the operation sequence for the worst case execution-time found, 2.344 seconds on standard PC platform (the ordering is important):

- ProcessRequest (ASAP, MemoryLoad)
- ProcessRequest (BYPASS)
- ProcessRequest (REPEAT, AvailPhys)
- ProcessRequest (ABSTIME, MemoryLoad)
- ProcessRequest (COMPARE, MemoryLoad, <=, size)

This outcome means that on the platform used, this method sequence with these input parameter settings will result in worst-case execution time of

2.344 seconds. The platform used is a typical Pentium PIII 200 MHz with Microsoft's built-in DCOM middleware. If this value is below the client's expectation, the test passes and we can call this a successful integration of the two components, the client and the server. The same applies for the assessment of the best-case execution time, although here the test passes if the value found is above the client's expected value. The following list shows the encoding of the test case for the operation sequence according the best case execution time found (1.468 seconds):

- ProcessRequest (BYPASS)
- ProcessRequest (ASAP, MemoryLoad)
- ProcessRequest (REPEAT, AvailPhys)
- ProcessRequest (ABSTIME, MemoryLoad)
- ProcessRequest (COMPARE, MemoryLoad, <=, size)

The value of the 16-bit integer that represents the size in both cases does not seem to have an impact on the measured response times.

7.4 QoS Contract Testing for Dynamic Updates

In the previous section, I have introduced the extended model of built-in contract testing as a development-time testing technique. Dynamic updates of a component-based real-time application cannot be assessed in this way. A dynamic update of a system means that a component is replaced during runtime.

Before any component may be replaced dynamically, the correctness of the new, replacing implementation must be assessed according the requirements of the integrating component framework. If no real-time contract must be adhered to, this is relatively straightforward, and it can be realized through the basic model of built-in contract testing.

The integrating component framework will have to provide two component references, one for the original component that keeps operating as long as the new component is being tested, and another one for acquiring and testing the new implementation in parallel. The test process can be performed at the lowest priority, so that it does not interfere with the normal operation of the system; and this requires that there is still processing capacity left on the platform. After a positive test, the two components can be replaced. Although this sounds simple, replacing components is actually the most challenging task. For example, one of the most difficult issues here is to transfer the internal state from the replaced component instance to the replacing. The European research project EMPRESS has dealt with such issues in an embedded system context, and it proposes some solutions for handling dynamic updates, at least partially [126]. So although there are currently no sound methodological approaches available to perform live updates in component-based systems, the initial step, that is, assessing whether a new component is suitable for a new context, is addressed through built-in contract testing.

A timing contract test on a running system is not so straightforward. The fact that this test is not supposed to interfere with the system's normal operation is the major inhibiting factor for its application. If the timing test is performed at the lowest priority, it is completely useless, because the measured execution time will always be longer than it would be in normal circumstances. If we schedule the test normally, as if the tested component were running in its final execution environment, we will likely get inaccurate timings, and the test may easily threaten the system's normal operation that is supposed to provide its service seamlessly. This is because we need processing capacity for two components at the same time with the same priority. The only way out of this dilemma that I can perceive is to run the test at the lowest possible priority and apply a different timing measurement strategy. Such a timing strategy could, for instance, instead of measuring absolute runtime of the tested object, measure only the processor cycles that are required to execute the test. There are some tools that can actually perform such processor cycle timings, for example IBM Rational's Purify Plus [154], but I have not yet thought about to incorporate such a tool into a testing process of the kind that I have put forward throughout this book. There is definitely a lot of research still required in this area.

7.5 Built-in Quality-of-Service Runtime Monitoring

So far, I have described the two component integration steps that can be assessed through built-in contract testing. The first one is concerned with testing behavioral contracts between two components when they are integrated for the first time (Chap. 4), and the second one is the test of timing contracts between two associated components (this chapter). These testing steps are performed by the system integrator during application assembly and development. Neither behavioral nor timing contract testing can guarantee that all behavioral and timing failures are identified before the system is deployed. No quality assurance technique is capable of doing that. Therefore, an additional quality assurance measure can be constantly carried out during runtime of the final system, and this is the assertion checking mechanisms that I briefly introduced in Chap. 4. While assertion checking concentrates primarily on functional and behavioral errors that may arise during final execution of a system, we can also have built-in quality of service assessment that constantly monitors a running application. In the following paragraphs, I will describe the so-called built-in quality of service testing strategy that realizes constant monitoring of a component-based application after it has been released [161, 162, 163]. The approach is fully described in a number of extensive Component+ project reports [34, 35, 36].

Ideally, a test should reveal every single failure in a system, but in general it can never guarantee that. It is not realistic to expect the deployment of an entirely fault free application, even if the most advanced and effective quality assurance techniques have been applied during development and integration. The principle of built-in QoS testing is therefore to expand testing beyond deployment of a component-based application. It does that by the same fundamental concepts that I introduced for built-in contract testing. So, in general, every component should provide the same artifacts that contract testing provides, plus some additional items. These are summarized as follows:

- Testing interface for the testable component. This is a collection of operations that a component provides to support constant runtime monitoring.
- Tester component. Associated component that uses the services of the testing interface. This determines whether an error condition exists.
- Handler. This processes an error that a tester component or a testable component identifies.
- Constructor. This is a conceptual element that is responsible for the creation and interconnection of tester and testable components and handlers.

Built-in quality-of-service testing was initially designed to deal with deadlock situations and response-time issues during a system's normal runtime, but other types of runtime monitoring such as performance profiling, memory management, code and data integrity, and trace facilities are also perceivable [163]. Interfaces to provided test or monitoring services are named `IBITx`. This corresponds to what I have so far called testing interface. The `x` is a placeholder for the type of monitoring that the testing interface supports, i.e., `IBITDeadlock` for deadlock monitoring, and `IBITtiming` for response-time monitoring. The tester component provides the corresponding notification services, the `IBITDeadlockNotify` and the `IBITTimingNotify` interfaces, through which monitored components can notify their tester components of the relevant interactions. For example, the deadlock tester is notified of events that are relevant to detecting deadlocks such as resource requests or releases. Additionally built-in QoS testing adheres to the definition of `IBITQuery`, `IBITError`, `IBITErrorNotify`. The following list summarizes these additional interfaces:

- `IBITQuery` allows an external entity to query the availability of specific testing interfaces, and to acquire a handle on the corresponding `IBITx` interface.
- `IBITError` provides the basis for error propagation between components. It can be queried to determine a component's error status, or it can be configured to notify the appropriate `IBITErrorNotify` interface of an associated tester component.
- `IBITErrorNotify` is the general notification interface that can be used to support erroneous events other than timing and deadlock.

- `IBITRegister` provides the mechanisms to associate all participating components; this comprises construction, destruction, and the connection of components, testers and, handlers.

Figure 7.11 shows all concepts of the built-in quality-of-service runtime monitoring approach.

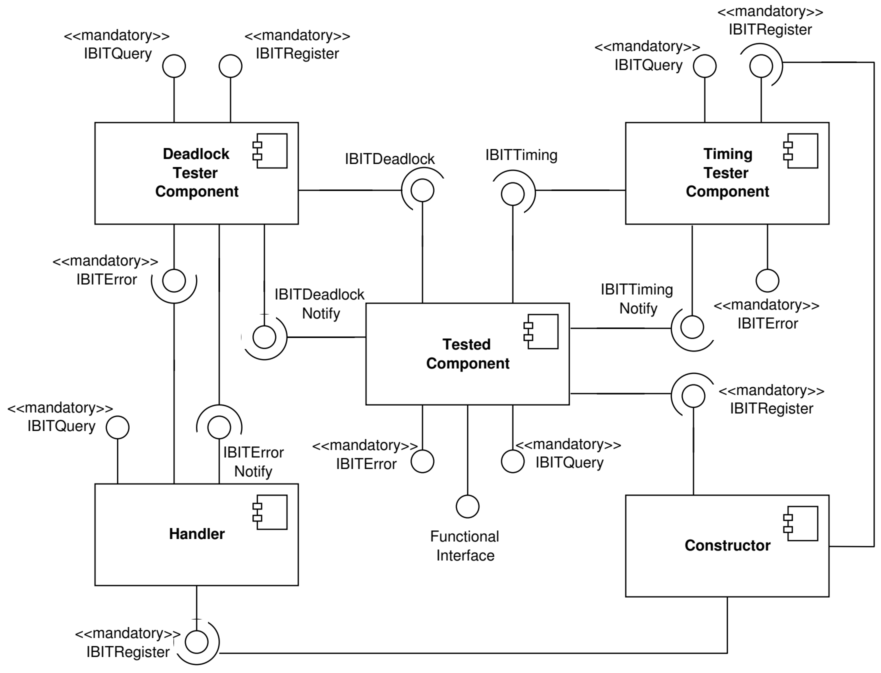


Fig. 7.11. Concepts of the built-in quality-of-service runtime monitoring

In contrast with built-in contract testing for behavioral contracts, the additional components in QoS runtime monitoring are never exclusively associated with a single tested component. This is probably the most important difference between the two technologies. For example, deadlocks can be detected only if several components that are competing for a resource are registered with the tester component that is responsible for monitoring this competition. Tester components in quality-of-service runtime monitoring are therefore located at an overall application level rather than associated with a particular client component, as is the case for contract testing. The system integrator must therefore consider a number of pairwise component interactions at a time, and not only a single one. The constructors are intended to perform this type of setup effort. Another difference is that for quality-of-service testing (and this includes the timing contract testing approach as well) the testing interfaces are always the same. They do not change from component to component according to the external states. This clearly facilitates standardization, so that the quality-of-service testing approaches as I have described them in this chapter are much easier to integrate into existing component models than built-in testing for behavioral contracts.

In the previous paragraphs I have described the architecture of built-in quality-of-service monitoring as it has been designed in the Component+ project [34, 35, 36]. I give an overview on which testing services apart from timing and deadlock monitoring the technology may be supporting. The fundamental mechanism to support all monitoring tasks is the `IBITError` interface. It is responsible for the propagation of error conditions between the different architectural entities of an application that supports this kind of runtime quality assurance. The developer of a component may augment this code with assertion-type mechanisms that make use of the provided interface. In that way, a component can easily support a number of additional testing and monitoring tasks such as pointer validation, completion of operations, I/O errors, and the like [163]. Moreover, the `IBITQuery` interface can be used to find out about the testing services that a third-party component provides. So, even if the technology is not yet standardized, it can provide a number of additional runtime quality assurance measures. I give the following examples according to [163]:

- Code and data integrity, which is an issue in embedded systems. Many of these systems do not provide proper memory management, so that the application memory can easily be corrupted. This follows the initial idea of built-in contract testing [166].
- Residual defects that show up only after some considerable execution period such as invalid pointers, memory allocation and deallocation, accumulated floating-point errors, etc.
- Trace as typical assertion mechanism. A trace is useful for locating an error.

7.6 Summary

Quality-of-service requirements are important not only for embedded real-time systems, but also for "normal" component-based systems that may have such requirements, e.g., throughput in Web applications. Embedded systems have especially stringent QoS requirements in general and timing requirements in particular. Response-time specifications are typically determined at the highest level of a system's decomposition, at its boundary, and they must be distributed among the decomposed objects. This activity is performed under the umbrella of constructive timing analysis, and its goal is to come up with a schedule for a system. This timing schedule must be assessed through a validation timing analysis activity, and the extended model of built-in contract testing is specifically geared toward that. It is concerned with how the timing requirements that are specified in terms of a timing contract between two interacting components can be assessed and validated dynamically.

In order to apply this extended model, the tested component needs to be augmented with an additional testing interface that provides timing notification services. The tested component needs to be augmented with a test case generator, typically a random generator, that applies a high volume of tests, and measures their timings. More effective test case generation techniques represent more sophisticated optimization strategies such as evolutionary algorithms. They can be applied in the same way as random testing, but result in much more accurate timings.

Permanent testing or monitoring of QoS requirements in general or real-time requirements in particular, beyond deployment, can be carried out through a supplementary quality assurance technique, built-in QoS testing. This provides a built-in testing framework that can be used to check code and data integrity, residual defects, deadlocks and timing, permanently during runtime of a component-based application.

Glossary

Abstraction

Abstraction is a powerful concept of hiding details and concentrating on essential information only. It refers also to a domain in the three-dimensional model of development in the KobrA method in the abstraction/concretization dimension.

Acceptance Testing

Acceptance testing refers to the test activities on the final application according to the user's or customer's requirements.

Application Engineering

Application engineering refers to all activities that deal with assembling and integrating existing components into a single application. It refers also to the instantiation of a final product out of a product family in product line engineering.

Assertion

An assertion is a boolean expression that defines the necessary conditions for the correct execution of an object or a component.

Average Case Execution Time

Average runtime of a program or a component's interaction over all feasible runs according to varying input parameters.

Basic Contract

A basic contract includes the signatures that a component provides or requires in terms of invocable operations, signals that a component sends or receives, and exceptions that a component throws or catches. It is also termed a syntactic contract.

Behavioral Contract

This is the same as the basic contract, plus it defines the overall functionality and behavior of a component in terms of pre- and postconditions of operations and externally visible, provided and required state transitions.

Behavioral Model

This is one part of a KobrA component specification which describes in terms of statechart diagrams or statechart tables how a component behaves in response to external stimuli.

Best Case Execution Time

This is the shortest feasible runtime of a program or a component over all feasible runs according to varying input parameters.

Built-in Testing

Built-in testing refers to all software artifacts that are built into a component, such as assertions, testing interfaces and tester components, to support testing activities.

COTS Component

A commercial off-the-shelf component is a ready-to-use physical component from a third party that can be incorporated into an application.

Certification

Certification is the process, carried out by a third party, of assuring that a vendor's claims concerning a product (e.g., a component) are justified. The third party may be an independent certification organization, the owner of a broker platform that distributes components, or a user group that publishes the opinions of its members.

Client

A client of a component instance *SI* is any other component instance that invokes the operations of *SI*. A client of component *S* (development-time description) is any other component with instances that are clients of *S's* instances.

Component

A component is a reusable unit of composition with explicitly specified provided and required interfaces and quality attributes, which denotes a single abstraction and can be composed without modification. In this book, the term component refers to a development-time description of a unit of composition, in contrast to a component instance or a physical component.

Component Adapter

A component adapter is an additional component in its own right that is inserted between two initially alien components to transfer what one component "means" into something that the other component "understands" and vice versa. It adapts the deviating contracts of two components so that they can interact in a meaningful way.

Component Contract

The interaction of components is based on the client/server model and the mutual agreement that the client meets the precondition of the server (client's contract) in order for the server to guarantee its postcondition (server's) contract. If the client fails to meet the precondition of a service, the server is not bound to its postcondition. This is a mutual agreement on which all component interactions are based, and termed a component contract.

Component Deployment

Component deployment is referred to as the act of configuring and running a component instance in its runtime environment.

Component Engineering

Component engineering represents all development activities that deal with developing individual components, rather than assemblies of components, that is applications.

Component Framework

This is an assembly of components that represents a product family. It can be instantiated, i.e., other components added, to retrieve the final application.

Component Instance

This is a component at runtime, according to the same idea that an object is an instance of its corresponding class.

Component Meta-model

This is the organization of a component expressed in terms of a modeling notation (a model of a model). A component meta-model describes the parts of a component description in a graphical form.

Component Realization

This is a collection of descriptive documents defined by the KobrA method that describe how the private design of a KobrA component fulfills its specification. The documents comprise structural models, activity models, interaction models, and a quality documentation.

Component Specification

This is a collection of descriptive documents defined by the KobrA method that describe a component's interfaces. The documents comprise structural models, behavioral models, functional models, non-functional specifications, and a quality documentation.

Component State

A state is a distinct setting of a component's internal attributes that determines the component's externally visible behavior.

Component Wrapper

This is the same as a component adapter.

Component-Based Development

This term represents all activities that deal with the development of software applications from existing reusable parts. It is separated into two subactivities, component engineering dealing with how individual components need to be built to be reusable, and application engineering dealing with how components need to be integrated into final applications.

Composition

This is the act of building or composing an application from existing parts. It is also a domain in the three-dimensional development model of the KobrA method in the composition/decomposition dimension.

Concretization

This is the act of turning an abstract software application described in the form of models into a more concrete software system, for example, described in the form of source code. It is also a domain in the three-dimensional development model of the KobrA method in the abstraction/concretization dimension.

Conformance Map

A conformance map describes a COTS component's externally visible features in terms of a mapping between the notation of the reused COTS component and the notation of our own development process, for example, KobrA and UML.

Containment

This is a development-time association defining a parent/child relation between components.

Context Realization

This describes the properties of a component's environment according to the KobrA method. It comprises all the models of a normal component realization, and, in addition, enterprise models that capture the abstract business for the intended system.

Decision Model

In a product family development, a decision model describes the groups of features possessed by the component for different members of the product family. It identifies the variation points and the corresponding resolution space in terms of effects of each resoltution option on the other models of the component.

Decomposition

This is the act of separating an application into finer-grained parts that are individually easier to tackle. It refers also to a domain in the three-dimensional development model of the KobrA method in the composition/decomposition dimension.

Deployment Environment

This is the runtime platform in which an application is eventually executed, in contrast to the development environment.

Development Method

A development provides a number of concepts, techniques, tools, and procedures that guide a software development in what engineers have to do, how they have to do it, and when they have to do it.

Embodiment

This is the act of transferring an application from a more abstract representation into a more concrete representation along the abstraction/concretization dimension of the KobrA method.

Encapsulation

Encapsulation is a key concept of object and component technology based on abstraction. Encapsulation separates what a component does from how it does it.

Execution Time Analysis

This is concerned with all activities carried out in order to assess a component's best-, average-, and worst-case timings.

Export Interface

This is the collection of services that a component provides. It is also termed provided interface.

Framework Engineering

This refers to all activities carried out to devise the core of a product family, the component framework.

Functional Model

This is the part of a component specification which describes the externally visible effects of the operations supplied by a component. Each component operation has one functional model.

Genericity

This is a domain in the three-dimensional development model of the KobrA method that refers to the genericity/specificity dimension. This dimension deals with the framework and application engineering activities in a product family development.

Import Interface

This is the collection of services that a component requires from its environment or runtime platform. It is also termed required interface.

Information Hiding

The principles of abstraction and encapsulation lead to the principle of information hiding. It refers to the principle that objects and components are fully described and usable according to their externally visible features without access to their internal implementations.

Instantiation

Instantiation refers to the act of turning an abstract entity into a concrete entitiy, that is, turning a component into a component instance, and instantiating a product family core representing a generic system into a concrete product.

Integration Test

This is a test that assesses component interactions according to their provided and required contracts when components are integrated to compose the final system.

Interface Definition Language (IDL)

This is a notation used by many contemporary component platforms to establish syntactic component interactions.

Invocation History

The invocation history refers to the sequence of operation invocations on a component, including their input parameter values, to bring a component into a distinct state.

KobrA Component

This is a component in accordance with the component model of the KobrA method, including a component specification, a component realization, and quality attributes with the required models.

Logical Component

This refers to an abstract component in a modeling hierarchy, in contrast to a concrete physical component in the runtime environment.

Logical State

This is an abstract externally visible description of the concrete internal attribute settings of a component. Logical states represent value domains of concrete physical component states. One logical state represents a number of internal physical states.

Middleware

The middleware refers to what is commonly understood as the collection of all services provided by contemporary component platforms. These services have not yet made it into the operating system environment, so that they reside in the middle between the operating platform and the application level.

Model-Based Testing

This refers to all activities and techniques that deal with the derivation of test artifacts from models.

Model-Driven Architecture

This is one of OMG's proposed approaches to future system engineering in which entire applications are modeled in graphical notations and then transformed automatically into executable entities.

Model-Driven Development

System development according to the principles of the model-driven architecture.

Normal Object Form (NOF)

This is a predefined implementation profile that contains elements of the core features of object-oriented programming languages, and thus can be simply translated into any mainstream object-oriented language.

Object Request Broker (ORB)

This is the part of a CORBA platform responsible for accepting and redirecting component requests. It works pretty much like an Internet proxy.

Physical Component

This is a concrete binary executable version of an abstract component.

Physical State

This is a concrete internal attribute setting of a component. A physical state represents a distinct value setting of an abstract component state that does not define values but only value domains.

Postcondition

This refers to the collection of all constraints on component properties of the server that must be fulfilled after successful completion of an operation invocation on a component. The postcondition is part of a server's component contract.

Precondition

This refers to the collection of all constraints on component properties of the client that must be fulfilled before an operation may be called on a server. The precondition is part of the client's component contract.

Product Family

This refers to all possible similar systems that are based on a common product family core, the component framework.

Product Line

This is the same as a product family.

QoS Contract

A quality-of-service contract quantifies the expected behavior or the component interaction in terms of minimum and maximum response delays, average response, quality of a result, e.g., in terms of precision, result throughput in data streams, and the like.

Quality Assurance Plan

This is a collection of documents defining what quality means for a development project, how it manifests itself in different kinds of products, what quality aspects are important for different kinds of products, and which quality levels are required and how they can be reached.

Refinement

Refinement is the representation of an entity in the same notation at a finer level of detail.

Semantic Map

This describes a mapping between the specification of a desired component and the KobrA specification of the interface offered by a foreign component.

Server

A server of a component instance *CI* is any other component instance whose operations *CI* invokes. A server of a component *C* (as a type) is any other component with instances that are servers of *C's* instances.

Structural Model

This describes classes and the nature of their attributes, operations, and relations in the form of UML diagrams.

Spiral Model

The spiral model represents an iterative approach to organize the phases of the software devlopment life-cycle. Each iteration in the process goes through planning, determining goals, alternatives, and constraints, evaluating alternatives and risks, developing and testing.

Synchronization Contract

The synchronization contract adds another dimension to the behavioral contract, which is the sequence or combination with which the interdependent operations of a component may be invoked.

Test

This term refers to an experiment under controlled conditions that applies a set of test cases or a test framework in order to validate wether a tested entity is consistent with its specification.

Test Case

This refers to an experimental execution of the component under consideration; a test case comprises an operation with parameters, expected and actual pre- and postconditions, a result, and a verdict.

Test Modeling

This refers to all activities and techniques that deal with the specification of test artifacts using models.

Test Stub

This represents fake functionality at a component's required interface with which it can be executed and assessed in a test run.

Testable Component

This refers to a component that provides an additional access mechanism to alleviate testing in built-in contract testing.

Tester Component

This refers to a component that contains test cases in built-in contract testing.

Testing Component

This refers to a component that owns a built-in tester component in built-in contract testing.

Testing Interface

This is an additional access mechanism of a component that facilitates its testing.

Translation

Translation is a transformation from one representation format into another one on the same level of detail.

UML Testing Profile

This is a variant of UML that addresses specifically the requirements of building test systems with UML.

Usage Model

This is a model that describes how a system is used.

Validation

Validation refers to all activities that are applied to assess whether "we build the right system." Typically, this comprises testing technologies that are applied in a translation relation.

Verification

Verification refers to all activities that are applied to assess whether "we build the system right." Typically, this comprises inspection and review techniques that are applied in a transformation relation.

V-Model

The v-model is one of the earliest product models that shows how the artifacts in a software development are related to one another. It aligns with the waterfall model as process model.

Waterfall Model

Software development process that proceeds linearly from requirements analysis through design, coding, and unit testing, subsystem testing and system testing. The waterfall model as process model aligns with the v-model as product model.

Worst-Case Execution Time

This is the longest feasible runtime of a program or a component over all feasible runs according to varying input parameters.

References

1. A. Abdurazik and J. Offutt. Using UML collaboration diagrams for static checking and test generation. In *International Conference on the Unified Modeling Language (UML 2000)*, York, UK, October 2000.
2. P. Allen and F. Frost. *Component-Based Development for Enterprise Systems: Applying the Select Perspective.* Cambridge University Press, 1998.
3. S. Amiri, C. Bunse, H.G. Gross, N. Mayer, and C. Peper. Marmot – Method for object-oriented and component-based embedded real-time system development and testing. http://www.marmot-project.org.
4. H. Apperly. The Component Industry Metaphor. In *Component-Based Software Engineering, Heineman/Councill (Eds)*, Boston, 2001. Addison-Wesley.
5. C. Atkinson, C. Bunse, H.-G. Gross, and T. Kühne. Towards a general component model for Web-based applications. *Annals of Software Engineering*, 13, 2002.
6. C. Atkinson et al. *Component-Based Product-Line Engineering with UML.* Addison-Wesley, London, 2002.
7. F. Barbier, N. Belloir, and J.-M. Bruel. Incorporation of test functionality into software components. In *2^{nd} International Conference on COTS-Based Software Systems*, Volume LNCS 2580, Ottawa, Canada, Feb. 2003. Springer.
8. J. Bayer et al. Pulse – A methodology to develop software product lines. In *Proceedings of the 5^{th} Symposium on Software Reusability (SSR'99)*, Los Angeles, May 21–23, 1999.
9. K. Beck. *Extreme Programming Explained.* Addison-Wesley, 1999.
10. K. Beck and E. Gamma. *Test Infected – Programmers Love Writing Tests.* CSLife and OTI, Zürich (http://members.pingnet.ch/gamma/junit.htm).
11. B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 1990.
12. B. Beizer. *Black-Box Testing, Techniques for Functional Testing of Software and Systems.* Wiley, New York, 1995.
13. N. Belloir, J.-M. Bruel, and F. Barbier. BIT/J User's Guide. Technical Report, University of Pau, LIUPPA, http://liuppa.univ-pau.fr/themes/aoc/aoc/bitj.php, 2003.
14. A. Bertolino and P. Inverardi. Architecture-based software testing. In *SIGSOFT'96 Workshop on Software Architectures*, San Francisco, CA, USA, 1996.

15. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Software*, 32(7):38–44, 1999.
16. R. Binder. *Testing Object-Oriented Systems: Models, Patterns and Tools.* Addison-Wesley, 2000.
17. J. Bloch. *Effective Java Programming Language Guide.* Sun Microsystems, 2001.
18. J. Bøegh. Quality evaluation of software products. *Software Quality Professional*, 1(2), 1999.
19. L. Bokhorst (Ed.). Requirements Specification Description Template. Technical Report, DESS Project (Software Development Process for Real-Time Embedded Software Systems), November 2001.
20. G. Booch. *Software Components with Ada: Structures, Tools and Subsystems.* Benjamin-Cummings, Redwood, CA, 1987.
21. M. Born, A. Hoffmann, A. Rennoch, and J. Reznik. The European CORBA Components open source initiative. *European Research Consortium for Informatics and Mathematics – News*, 55:33–34, October 2003.
22. J. Bosch (Ed.). *Generative and Component-Based Software Engineering*, Volume 2186 of *Lecture Notes in Computer Science.* Springer, Berlin, 2001.
23. F. Brooks. *The Mythical Man Month.* Addison-Wesley, 1995.
24. A.W. Brown. *Large-Scale, Component-Based Development.* Prentice Hall, 2000.
25. C. Bunse. *Pattern-Based Refinement and Translation of Object-Oriented Models to Code*, Volume 2 of *PhD Theses in Experimental Software Engineering.* Fraunhofer, Stuttgart, 2001.
26. C. Bunse and C. Atkinson. Improving quality in object-oriented software: Systematic refinement and translation from models to code. In *12th International Conference on Software & Systems Engineering and Their Applications*, Paris, France, 1999.
27. C. Bunse and C. Atkinson. The normal object form: Bridging the gap from models to code. In *2nd International Conference on the Unified Modeling Language*, Fort Collins, USA, 1999.
28. J. Cheesman and J. Daniels. *UML Components, A Simple Process for Specifying Component-Based Systems.* Addison-Wesley, 2000.
29. S. Chung et al. Testing of concurrent programms based on message sequence charts. In *IEEE International Symposium on Software Engineering for Parallel and Distributed Systems*, Los Angeles, CA, May 17–18, 1999.
30. J. Clark, C. Clarke, S. DePanfilis, G. Granatella, P. Predonzani, A. Sillitti, G. Succi, and T. Vernazza. Selecting components in large COTS repositories. *Journal of Systems and Software*, 2004.
31. A. Cockburn. Basic Use Case Template. Technical Report TR.96.03a, Human and Technology (http://alistair.cockburn.us), Salt Lake City, 1996.
32. A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley, Boston, 2001.
33. D. Coleman et al. *Object-Oriented Development. The Fusion Method.* Prentice Hall, 1994.
34. Component+ Consortium, www.component-plus.org. *Built-in Test Vade Mecum – Part 1: A common BIT architecture*, 2003.
35. Component+ Consortium, www.component-plus.org. *Built-in Test Vade Mecum – Part 2: Interface specifications, types, syntax and semantics*, 2003.
36. Component+ Consortium, www.component-plus.org. *Built-in Test Vade Mecum – Part 3: Quality of service testing*, 2003.

37. Component+ Consortium. Built-in Testing for Component-Based Development. Technical Report, Component+ Project, www.component-plus.org, 2001.
38. Component+ Consortium. Built-in Testing for Component-Based Development. Technical Report, Deliverable D3, www.component-plus.org, 2001.
39. J.R. Corbin. *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls.* Springer, 1991.
40. K. Czarnecki and U.W. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.
41. W. Dröschel and M. Wiemers. *Das V-Model 1997.* Oldenbourg, 1999.
42. D.F. D'Souza and A.C. Willis. *Objects, Components and Frameworks.* Addison-Wesley, 1998.
43. A. Eberhart and S. Fischer. *Web Services.* Hanser, München, 2003.
44. H.-E. Eriksson and M. Penker. *UML Toolkit.* Wiley, 1998.
45. European Telecommunications Standards Institute (ETSI). The Testing and Test Control Notation – Part 1: TTCN-3 Core Language. Technical Report, ETSI ES 201 873-1, 2003.
46. European Telecommunications Standards Institute (ETSI). The Testing and Test Control Notation – Part 2: TTCN-3 Tabular Presentation Format. Technical Report, ETSI ES 201 873-2, 2003.
47. European Telecommunications Standards Institute (ETSI). The Testing and Test Control Notation – Part 3: TTCN-3 Graphical Presentation Format. Technical Report, ETSI ES 201 873-3, 2003.
48. European Telecommunications Standards Institute (ETSI). The Testing and Test Control Notation – Part 4: TTCN-3 Operational Semantics. Technical Report, ETSI ES 201 873-4, 2003.
49. European Telecommunications Standards Institute (ETSI). The Testing and Test Control Notation – Part 5: TTCN-3 Runtime Interface. Technical Report, ETSI ES 201 873-5, 2003.
50. European Telecommunications Standards Institute (ETSI). The Testing and Test Control Notation – Part 6: TTCN-3 Control Interface. Technical Report, ETSI ES 201 873-6, 2003.
51. Cpp Test Framework for C++. http://cppunit.sourceforge.net.
52. Perl Unit Test Framework for C++. http://perlunit.sourceforge.net.
53. International Organization for Standardization (ISO). Information Technology - Open System Interconnection, Conformance Testing Methodology and Framework. Technical Report, ISO/IEC 9646:1998, 1998.
54. International Organization for Standardization (ISO). Information Technology - Open Distributed Processing - Interface Definition Language. Technical Report, ISO/IEC 14750:1999, 1999.
55. International Organization for Standardization (ISO). Software Engieering – Product Evaluation – Part 3: Process for Developers. Technical Report, ISO/IEC 14598-3:2000, 2000.
56. International Organization for Standardization (ISO). Software Engieering – Product Quality – Part 1: Quality Model. Technical Report, ISO/IEC 9126-1:2001, 2001.
57. International Organization for Standardization (ISO). Software Engieering – Product Quality – Part 2: External Metrics. Technical Report, ISO/IEC TR 9126-2:2003, 2003.

58. International Organization for Standardization (ISO). Software Engieering – Product Quality – Part 2: Internal Metrics. Technical Report, ISO/IEC TR 9126-3:2003, 2003.
59. M. Fowler. A UML Testing Framework. *Software Development*, April 1999.
60. M. Fowler and K. Scott. *UML Distilled*. Addison-Wesley, 1997.
61. C Unit Test Framework. http://cunit.sourceforge.net.
62. JUnit Test Framework. http://www.junit.org.
63. XUnit Test Framework. http://c2.com/cgi/wiki?testingframework.
64. S. Frolund and J. Koisten. QML: A language for quality of service specification. Technical Report HPL-98-10 980210, Hewlett-Packard, 1998.
65. J. Gao. Challenges and problems in testing software components. In *Workshop on Component-Based Software Engineering (ICSE 2000)*, Limerick, June 2000.
66. J.Z. Gao, H.-S.J. Tsao, and Y. Wu. *Testing and Quality Assurance for Component-Based Software*. Artech House, 2003.
67. S. Ghosh and A.P. Mathur. Issues in testing distributed component-based systems. In *Workshop on Testing Distributed Component-Based Systems (ICSE 1999)*, Los Angeles, May 1999.
68. D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
69. J. Grabowski and M. Schmitt. TTCN-3 - A language for the specification and implementation of test cases. *'at – Automatisierungstechnik*, 3, 2002.
70. L. Graham, B. Henderson-Sellers, and H. Younessi. *The OPEN Process Specification*. Addison-Wesley, 1997.
71. H.-G. Gross. *Measuring Evolutionary Testability of Real-Time Software*. PhD thesis, University of Glamorgan, Pontypridd, Wales, UK, June 2000.
72. H.-G. Gross. An evaluation of dynamic, optimisation-based worst-case execution time analysis. In *International Conference on Information Technology*, Kathmandu, Nepal, 2003.
73. H.-G. Gross, C. Atkinson, F. Barbier, N. Belloir, and J.-M. Bruel. Business Component-Based Software Engineering, Barbier (Ed.), Chapter Built-in Contract Testing for Component-Based Development (Chapter. 4). Kluwer, 2003.
74. H.-G. Gross, C. Atkinson, and F. Barbier. *Component-Based Software Quality, Cechich, Piattini, Vallecillo (eds.)*, Volume 2693 of *Lecture Notes in Computer Science (LNCS)*, chapter Component Integration through Built-in Contract Testing. Springer, Berlin, 2003.
75. H.-G. Gross and N. Mayer. Search-based execution-time verification in object-oriented and component-based real-time system development. In *8^{th} IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, Guadalajara, Mexico, January, 15–17 2003.
76. H.-G. Gross, I. Schieferdecker, and G. Din. *Quality in Component-based Development – Testing and Debugging, Beydeda (Ed.)*, Chapter Modeling and Implementation of Built-in Contract Tests. Springer, Berlin, 2004.
77. M. Grossman. Component testing: An extended abstract. In *Workshop on Component-Oriented Programming (ECOOP 1998)*, Brussels, July 1998.
78. Object Management Group. History of CORBA. Technical Report, http://www.omg.org, 1997–2004.
79. Object Management Group. Model driven architecture - resource page. Technical Report, http://www.omg.org, 1997–2004.

80. Object Management Group. Object management architecture - resource page. Technical Report, http://www.omg.org, 1997–2004.
81. Object Management Group. UML testing profile. Technical Report, http://www.omg.org, 1997-2004.
82. Object Management Group. CORBA - core specification. Technical Report, Version 3.0, December 2002.
83. D.S. Guindi, W.B. Ligon, W.M. McCracken, and S. Rugaber. The impact of verification and validation of reusable components on software productivity. In *22nd Annual Hawaii Intl Conference on System Sciences*, Pages 1016–1024, 1989.
84. D. Hamlet, D. Mason, and D. Woit. Theory of software reliability based on components. In *23rd International Conference on Software Engineering (ICSE-01)*, Los Alamitos, California, 2001. IEEE Computer Society.
85. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
86. M.J. Harrold, D. Liang, and S. Sinha. An approach to analyzing and testing component-based systems. In *Workshop on Testing Distributed Component-Based Systems (ICSE 1999)*, Los Angeles, May 1999.
87. J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *International Symposium on Software Testing and Analysis (ISSTA 2000)*, Portland, USA, August 2000.
88. R. Heckel and M. Lohmann. Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 82(6), 2003.
89. G.T. Heineman and W.T. Councill (Eds). *Component-Based Software Engineering.* Addison-Wesley, Boston, 2001.
90. J. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, Cambridge, 1975.
91. IEEE. *Standard Glossary of Software Engineering Terminology*, Volume IEEE Std. 610.12-1990. IEEE, 1999.
92. Fraunhofer IGD. RIN system specification. Technical Report, Fraunhofer Institute for Graphical Data Processing, Darmstadt, Germany (www.igd.fraunhofer.de), 2002.
93. J.A. Illik. *Programmierung in C unter UNIX.* Sybex, Düsseldorf, 1990.
94. European Telecommunications Standard Institute. www.etsi.org.
95. I. Jacobson. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.
96. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.
97. J.M. Jézéquel and B. Meyer. Design by contract: The Lessons of ARIANE. *IEEE Computer*, 30(1):129–130, Jan. 1997.
98. J.-M. Jézéquel, D. Deveaux, and Y. Le Traon. Reliable objects: Lightweight testing for oo languages. *IEEE Software*, July/August 2001.
99. K.C. Kang, S.G. Cohen, W.E Novak, and E.S. Petersen. Feature-oriented domain analysis (FODA) feasibility study. Technical Report, Software Engineering Institute, November 1990.
100. Y.G. Kim, H.S. Hong, D.H. Bae, and S.D. Cha. Test cases generation from UML state diagrams. *IEE Proceedings Software*, 146(4), 1999.
101. P.B. Kruchten. *The Rational Unified Process – An Introduction.* Addison-Wesley, 2000.
102. D. Lane. *JUnit – The Definitive Guide.* O'Reilly, 2004.

103. Y. Le Traon, D. Deveaux, and J.-M. Jézéquel. Self-testable components: from pragmatic tests to design for testability methodology. In *Technology of Object-Oriented Languages and Systems*, Nancy, France, June 7-11 1999.
104. J.T.L. Lions and ARIANE 501 Inquiry Board. ARIANE 5, flight 501 failure. Technical Report, European Space Agency, Paris, July 1996.
105. Testing Technologies IST Ltd. *TTanalyze, http://www.testingtech.de/products/TTanalyze.*
106. Testing Technologies IST Ltd. *TTspec, http://www.testingtech.de/products/TTspec.*
107. Testing Technologies IST Ltd. *TTthree, http://www.testingtech.de/products/TTthree.*
108. Testing Technologies IST Ltd. *TTtwo2three, http://www.testingtech.de/products/TTtwo2three.*
109. B. Marick. *The Craft of Software Testing.* Englewood-Cliffs, New Jersey, 1995.
110. R.C. Martin. Java vs. C++. Technical Report, Object Mentor, Inc., www.objectmentor.com, March 1997.
111. J.D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, 2001.
112. B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1997.
113. Microsoft. *Microsoft .NET.* http://www.microsoft.com/net.
114. Sun Microsystems. Enterprise JavaBeans technology specification, version 2.1 – final release. Technical Report, java.sun.com, 1995–2003.
115. Sun Microsystems. JavaBeans component architecture documentation. Technical Report, java.sun.com, 1995–2003.
116. H.D. Mills, R.C. Linger, and R.A. Hevner. Box structured information systems. *IBM Systems Journal*, 26(4), 1987.
117. K.D. Nilsen and B. Rygg. Worst-case execution time analysis on modern processors. *ACM SIGPLAN Notices*, 30(11):61–64, 1995.
118. Object Management Group. *Unified Modeling Language Specification*, 2000.
119. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In *International Conference on the Unified Modeling Language (UML 1999)*, Fort Collins, USA, October 1999.
120. OMG. *UML 2.0 Testing Profile Specification.* Object Management Group, www.omg.org, 2003.
121. OpenTTCN. *http://www.openttcn.com.*
122. C.Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5:31–62, 1993.
123. P. Predonzani, G. Succi, and T. Vernazza. *Strategic Software Production with Domain-Oriented Reuse.* Artech House, 2000.
124. R.S. Pressman. *Software Engineering: A Practioner's Approach.* McGraw-Hill, New York, 1997.
125. CLARiFi Project. http://www.clarify.eng.it.
126. EU ITEA Empress Project. http://www.empress-itea.org.
127. P. Puschner. A tool for high-level language analysis of worst-case execution times. In *10th Euromicro Workshop on Real-Time Systems*, Berlin, 1998.
128. P. Puschner. Worst-case execution-time analysis at low cost. In *Control Engineering Practice*, Volume 6, Pages 129–135, 1998.
129. P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. In *Real-Time Systems*, Volume 1, Pages 159–176, 1989.

130. P. Puschner and R. Nossal. Testing the results of static worst-case execution time analysis. In *19th IEEE Real-Time Systems Symposium*, Madrid, Dec. 1998.
131. P. Puschner and A. Schedl. Computing maximum task execution times – a graph-based approach. In *Real-Time Systems*, Volume 13, Pages 67–91, 1997.
132. T. Reenskaug, P. Wold, and O. Lehne. *Working with Objects: The OORAM Software Development Method*. Manning/Prentice Hall, 1996.
133. B. Regnell and P. Runeson. Combining scenario-based requirements with static verification and dynamic testing. In *4th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ98)*, Pisa, June 8–9, 1998.
134. B. Regnell, P. Runeson, and C. Wohlin. Towards the integration of use case modeling and usage-based testing. In *4th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ98)*, Pisa, June 8-9 1998.
135. R.H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos, Berlin, 2001.
136. R.H. Reussner. The use of parameterised contracts for architecting systems with software components. In *6th Intl Workshop on Component-Oriented Programming*, Budapest, Hungary, 2001.
137. R.H. Reussner and H.W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In *9th International Workshop on Component-Based Software Engineering*, Lund, Sweden, 2002.
138. R.H. Reussner, H.W. Schmidt, and I.H. Poernomo. Reliability prediction for component-based software architectures. *Systems and Software*, 66(3), 2002.
139. D.J. Richardson and A.L. Wolf. Software testing at the architectural level. In *SIGSOFT 1996 Workshop on Software Architectures*, San Francisco, CA, USA, 1996.
140. P.J. Robinson. *Hierarchical Object-Oriented Design*. Prentice Hall, 1992.
141. D.S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Jan. 1995.
142. D.S. Rosenblum. Adequate testing of component-based software. Technical Report, Dept. of Computer Science, University of California, TR 97-34, 1997.
143. J. Rumbaugh et al. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
144. J. Ryser and M. Glinz. A practical approach to validating and testing software systems using scenarios. In *Quality Week Europe*, Brussels, 1999.
145. J. Ryser and M. Glinz. SCENT: A method employing scenarios to systematically deriving test cases for system test. Technical Report, University of Zürich, 2000.
146. J. Ryser and M. Glinz. Using dependency charts to improve scenario-based testing. In *17th International Conference on Testing Computer Software (TCS2000)*, Washington, 2000.
147. P. Santos, T. Ritter, and M. Born. Rapid engineering of collaborative and adaptive multimedia systems on top of CORBA Components, K. Irmscher (Ed.). *Kommunikation in Verteilten Systemen, VDE*, 2003.
148. I. Schieferdecker, Z.R. Dai, J. Grabowski, and A. Rennoch. The UML 2.0 testing profile and its relation to TTCN-3. In Wiles Hogrefe (Ed.), *Proceedings of the 15th International Conference on Testing Communicating Systems*, Springer LNCS Volume 2644, Heidelberg, 2003.

149. I. Schieferdecker and J. Grabowski. The graphical format of TTCN-3 in the context of MSC and UML. In *International Workshop on SDL and MSC*, Springer LNCS Volume 2599, Heidelberg, 2003.
150. M. Schünemann, I. Schieferdecker, A. Rennoch, L. Mang, and C. Desroches. Improving test software using TTCN-3. Technical Report, GMD Forschungszentrum Informationstechnik (now Fraunhofer FOKUS), 2001.
151. H.P. Schwefel and R. Männer. *Parallel Problem Solving from Nature.* Springer, 1990.
152. B. Selic, G. Gullekson, and P. Ward. *Real-Time Object-Oriented Modeling.* Wiley, 1994.
153. A. Sillitti, G. Granatella, P. Predonzani, G. Succi, and T. Vernazza. Ranking and selecting components to build systems. In *International Conference on Enterprise Information Systems (ICEIS 2003)*, Angers, France, 2003.
154. IBM Rational Software. Purifyplus; http://www-306.ibm.com/software/awdtools/purifyplus.
155. G. Succi, W. Pedrycz, and R. Wong. Dynamic composition of components using Web-CODs. *International Journal of Computers and Applications*, 2002.
156. SunSoft. *Java 2 Enterprise Edition (J2EE).* http://java.sun.com/j2ee/.
157. C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, Harlow, England, 1999.
158. C. Szyperski. *Component Software, Beyond Object-Oriented Programming.* Addison-Wesley, London, England, second edition, 2002.
159. J. Udell. Componentware. *Byte Magazine*, 14(5):46–56, May 1994.
160. A. van der Hoek and H. Muccini. Towards testing product line architectures. In *ETAPS 2003 – Workshop on Test and Analysis of Component-Based Systems*, 2003.
161. J. Vincent and G. King. Built-in-test for run-time-testability in software components: Testing architecture. In *BCS Software Quality Management Conference*, 2002.
162. J. Vincent, G. King, P. Lay, and J. Kinghorn. Principles of built-in-test for run-time-testability in component based software systems. *Software Quality Journal*, 10(2), 2002.
163. J. Vincent, G. King, P. Lay, and J. Kinghorn. *Business Component-based Software Engineering, Franck Barbier (Ed.)*, chapter Built-In-Test for Run-Time-Testability in Software Components: Testing Architecture. Kluwer, Boston, 2003.
164. P.J.M. Von Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications.* Kluwer, 1997.
165. Y. Wang, G. King, I. Court, M. Ross, and G. Staples. On testable object-oriented programming. *ACM Software Engineering Notes*, 22(4), 1997.
166. Y. Wang, G. King, D. Patel, S. Patel, and A. Dorling. On coping with real-time software dynamic inconsistency by built-in tests. *Annals of Software Engineering*, 7, 1999.
167. Y. Wang, G. King, and H. Wickburg. A method for built-in tests in component-based software maintenance. In *IEEE International Conference on Software Maintenance and Reengineering (CSMR-99)*, Pages 186–189, 1999.
168. Y. Wang, D. Patel, G. King, and S. Patel. *BIT: A Method for Built-in Tests in Object-Oriented Programming*, chapter 47 in Handbook of Object Technology, Zamir (Ed.). CRC Press, 1998.

169. C.D. Warner. Evaluation of program testing. Technical Report, IBM Data Systems Division Development Laboratories, Poughkeepsie, NY, July 1964.
170. B.F. Webster. *Pitfalls of Object-Oriented Development.* M&T Books, 1995.
171. J. Wegener and M. Grochtmann. Verifying timing constraints by means of evolutionary testing. *Real-Time Systems*, 3(15), 1998.
172. J. Wegener, R. Pitschinetz, and H. Sthamer. Automated testing of real-time tasks. In *1st International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000.
173. J. Wegener, H. Sthamer, and A. Baresel. Application fields for evolutionary testing. In *EUROStar, Testing and Verification*, Stockholm, Sweden, November 2001.
174. D.M. Weiss and C.T.R. Lai. *Software Product Line Engineering – A Family-Based Software Engineering Process.* Addison-Wesley, 1999.
175. E.J. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5), 1998.
176. E.J. Weyuker. The trouble with testing components,. In *Component-Based Software Engineering, Heineman/Councill (Eds).* Addison-Wesley, 2001.

Index

Printing: Strauss GmbH, Mörlenbach
Binding: Schäffer, Grünstadt