Fraunhofer Institut
Experimentelles
Software Engineering

Component+ Methodology

Built-In Contract Testing: Method and Process

Author:
Hans-Gerhard Groß

In part supported by
Matteo Melideo, Engineering Informatica, Rome, Italy
Franck Barbier, LIUPPA, Pau, France
and the Component+ Project Consortium.
under EC-IST-1999-20162

IESE-Report No. 030.02/E
Version 1.0
October 31, 2002

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.
The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competetive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach
Sauerwiesen 6
D-67661 Kaiserslautern

Executive Summary

This report represents one of the two parts of the methodology and process of Built-in Testing (BIT) technology that has been developed within the European Union IST 5th Framework Programme in the project Component+ (EC-IST-1999-20162).

The aim of this work is to devise a technology and methodology that can check the pairwise interactions of components in component-based software construction at integration and deployment time. Such pairwise interactions are also defined as contracts. Built-in contract testing is based on building test functionality into components, in particular tester components on the client side and testing interfaces on the server side of a pairwise contract. Since building test software into components has implications with the overall component-based development process, the technology is integrated and made to supplement an existing component-based development methodology, the KobrA method. This report initially outlines the concepts of the KobrA method that are important for built-in contract testing, provides a step-by-step guide on how to devise built-in contract testing artefacts on the basis of the KobrA method, and finally discusses further implications that built-in contract testing has on typical object and component technology concepts.

Keywords: Built-in Testing, Component-based Development, Component Contract, UML Model, Tester Component, Testing Interface, Development Process

Table of Contents

1 Introduction

The vision of component-based development is to allow software vendors to avoid the overheads of traditional development methods by assembling new applications from high-quality, prefabricated, reusable parts. Since large parts of an application may therefore be constructed from prefabricated components, it is expected that the overall time and costs involved in application development will be reduced, and the quality of the resulting applications will be improved. This expectation is based on the implicit assumption that the effort involved in integrating components at deployment time is lower than the effort involved in developing and validating applications 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 are themselves faulty. Current component technologies can help to verify the syntactic compatibility of interconnected components (i.e. that they use and 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 inter-connected components, so that the individual parts are assembled into meaningful configurations. Software developers may therefore be forced to perform more integration and acceptance testing in order 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 components, they can do little to assure the quality of applications that are assembled from them at deployment time.

1.1 Contracts in Component-Based Development

The correct functioning of a system of components at run time is contingent on 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, following Meyer [Mey97], the set of rules governing the interaction of a pair of objects (and thus components) is typically referred to as a contract. This characterizes the relationship between a component and its clients as a formal agreement, expressing each party’s rights and obligations. Testing the correct functioning of individual client/server interactions against the specified contract therefore goes along way towards verifying that a system of components as a whole will behave correctly.

1.2 Built-in Contract Testing

The testing approach described in this report is therefore based on the notion of building contract tests into components so that they can validate that the servers to which they are "plugged" dynamically at deployment time will fulfill their contract. Although built-in contract testing is primarily intended for validation activities at deployment and configuration-time, the approach also has important implications on the development phases of the overall software life-cycle. Consideration of built-in test artefacts needs to begin early in the design phase as soon as the overall architecture of a system is developed and/or the interfaces of components are specified. Built in contract testing therefore needs to be integrated with an overall software development methodology. In this report we explain the basic principles behind built-in contract testing, and how they affect component-based development principles, and show how it can be integrated with, and made to complement, a mainstream development method that is based on the creation of UML models.

1.3 Software Components

Since this work is mainly settled around the concepts of component based software engineering, it is important that we define the way in which we use the term component. For the sake of this report we choose to define a software component as a unit of composition with explicitly specified provided, required and configuration interfaces, plus quality attributes [Cmp+01]. This definition is based on the well known definition of the 1996 European Conference on Object-Oriented Programming [Szy99], that defines a component as unit of composition with contractually specified interfaces and context dependencies only, and that it can be deployed independently, and is subject to composition by third parties. We have intentionally chosen a broader definition that is avoiding the terminology “independently deployable”, since we are not specifically restricting ourselves to contemporary component technologies such as CORBA, .NET or COM. In this respect we are closer to Booch’s definition who sees a component as a logically, cohesive, loosely coupled module that denotes a single abstraction [Boo87]. From this it becomes apparent that components are basically built upon the same fundamental principles as object technology. The principles of encapsulation, modularity, and unique identities are all basic object-oriented principles that are subsumed by the component paradigm [Atk01].

1.3.1 Instance/Type Dichotomy of Components

Since we claim that components are not necessarily independently deployable, we can be sloppy about the role of the instance/type dichotomy. A type is a blueprint from which concrete instances can be created. In the object-oriented

terminology types are typically referring to classes and instances to objects. Consequently, components may be instances in concrete run-time environments when they are deployed into a system, but they can also be development-time incarnations when they act as blueprints. The only difference between run-time and development-time incarnations is that the latter do not encapsulate states nor offer invocable services. Over the course of this document we mean type if we refer to a component, and we mean instance if we refer to a component instance. In the scope of the Model Driven Architectures [OMG-MDA] types may already be seen as representing instances since the instance is created through a generative programming step. The same applies to source code components that are readily deployable through existing compiler technology, for example Java source code components that are easily translated into byte-code.

1.3.2 Class/Module Dichotomy of Components

Another important issue is whether components themselves present features such as operations and attributes, or whether they only act as containers for elements that do present these features. This is concerned with the question of whether components are class-like or module-like entities [Atk01]. The Components that are considered in this document are both, they may be seen as a unification of a class and a module.

1.4 Overall Structure of this Report

Built-in contract testing is greatly facilitated if it is used as part of an overall development process. A brief introduction to a sound development method is therefore essential. Chapter 2 lays the foundations for applying built-in contract testing technology properly, because it introduces specification techniques that are all part of the KobrA development method [Atk01]. It describes the artefacts that should ideally be available for a component if built-in contract testing will be applied and it builts the basis for the way in which a built-in contract testing architecture can be derived for an application. Because the principles of the contract testing approach extend the model-driven development paradigm of the KobrA method, and they extend this method in respect to testing, models modeling and model driven test case generation are all fundamental to this report. Chapter 3 discusses typical test case generation techniques that are ideal candidates for test case generation in object technology and component-based development. It also elaborates the main test case selection techniques that may be used with built-in contract testing. The following chapters Chapter 4 and Chapter 5 represent the core of this report. They concentrate on the main built-in contract testing methodology. Chapter 4 defines the actual artefacts that must be developed under the contract testing approach and describes how they are related. It elaborates built-in testing interfaces and tester components. Chapter 5 describes the process in detail that must be followed for developing modeling

and designing the testing functionality out of the modeling and design of the normal functionality of components. This may be seen as a step-by-step guide for built-in contract testing development. All subsequent chapters are concerned with the effects or implications that this technology imposes on other typical component technologies. Chapter 6 discusses how built-in contract testing supports the reuse paradigm of component-based software engineering. Chapter 7 looks at how the technology affects configuration management and configuration interfaces, and how these affect the technology. Chapter 8 describes how built-in contract testing may be used with commercially available third party components, so called COTS, that typically provide restricted internal access for testing purposes. Chapter 9 briefly discusses how built-in contract testing may be used with Web-Services, and finally, Chapter 10 summarizes and concludes this report.

2 Application Specification

The initial starting point for a software development project is undoubtedly a system or application specification derived and decomposed from the system requirements. Requirements are collected from the customer of the software. They are decomposed in order to remove their genericity in the same way as system designs are decomposed in order to obtain finer grained parts that are individually controllable. These parts are implemented and later composed into the final product. The decomposition activity is aiming to obtain meaningful, individually coherent parts of the system, the components. It is also referred to as component engineering or component development. The composition activity tries to assemble already existing parts, that may have been already used in other applications, into a meaningful configuration that reflects the predetermined system requirements. To sum this up, component engineering or component development is a top-down activity that decomposes a system into finer grained parts. Application engineering or component assembly is a bottom-up activity that builts up a complete system from readily available, prefabricated parts.

In its purest form, component-based development is only concerned with the second item, representing a bottom-up approach to development. 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 quite likely lead to a system that is not conformant with its original requirements. Component-based development is therefore usually a mixture of top-down decomposition and bottom-up composition. In other words, the system is decomposed into finer grained parts, that means sub-systems or components, and these are attempted to be mapped to individual prefabricated components. 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 considered sub-system. The whole process is iterative and must be followed until all requirements are mapped to corresponding components or until the system is fully decomposed onto the lowest desirable level of abstraction. If suitable third party components are found, they can be composed to make up the system or sub-system under consideration. Such a process is always goal-oriented in that it only accepts components that are fit for the purpose of the system. This means that only such 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 own implementations.

The decomposition process is based on the derivation of component specifications and realizations. The specification of a component comprises every information that is necessary to fully describe what a system part does, and the realization of a component contains full information that is necessary to implement this part. Their development and the required artefacts are described in the following sub-sections. But initially, we need to have a look at development methods that support the decomposition and composition activities.

2.1 Development Methods

Every serious attempt of 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 [Atk01]. A method acts as a framework and a process in which the development effort will be carried out. Additionally, it defines the intermediate development artefacts, and it provides guidelines on how these should be used as input to subsequent development cycles. It also ideally supports their verification in some way. Applying a development method consequently 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 [Atk01] that has been developed by Fraunhofer IESE. It draws its ideas from many of today's 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 [RBP91], Fusion [Col94], ROOM [SGW94], HOOD [Rob92], OORAM [RWL96], Catalysis [DW98], Select Perspective [AF98], UML Components [CD00], FODA [KCN90], FAST [WL99], PuLSE [BFK99], Rational Unified Process [JBR99,Kru00], OPEN [GHY97], and Cleanroom [MLH87].

The KobrA method uses the UML as primary notation. That means, most software documents that are created during the development with this method are UML models. However, there are other artefacts in natural language or in tabular form, but KobrA follows the concepts of OMG's Model Driven Architectures, so models are the primary development documents. Any other development method, or even no development method at all may be used to come up with the specification artefacts on which built-in testing is based. However, in the following sub-sections we describe the artefacts that are typically created in a KobrA development project since this supports built-in contract testing in a very natural way. All these specification artefacts may also be developed completely

arbitrarily, for instance in natural language. In fact, the Extensible Markup Language (XML) is more and more being used to express graphical specification artefacts such as models, for instance. XML is a tagged language that is often used in component technologies (e.g. CORBA Components) and for generative programming.

The following sub-sections concentrate on describing the specification artefacts that are typically created for each individual component within a development project. These are the products of the development method. How they are obtained as part of a development process is not subject of this report, however. Here, we concentrate merely on the software documents that represent the basis for creating the built-in testing specifications and model artefacts. But before that, we will discuss the importance of an overall quality assurance plan for a development project that should be part of the development method. Non-functional (quality) requirements are part of a quality plan, and they are usually defined for both, the entire project, and each individual component.

2.2 Definition of a Quality Assurance Plan

An essential part of an application specification comprises quality requirements that must be fulfilled in order to have an acceptable product. The specification cannot simply state that the product should exhibit high quality, or low failure rates, or the like. Such a terminology is too unspecific and it cannot be assessed. 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 be determined in the first place. Quality may be defined through many differing attributes that a software product is expected to exhibit.

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 [Atk01]:

- A precise definition of what quality means for the considered development project, 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.

This report develops a testing strategy and organization that applies built-in testing artefacts in component-based systems. This means that it is part and outcome of the considerations that must be taken when a quality assurance plan

for a development project is devised. However, it is not in the scope of this report to define and apply such a quality assurance plan for a particular software development project. In this document we merely underpin the importance of a concrete plan of which quality assurance techniques, or test adequacy criteria should be applied to satisfy which quality attributes to the expected level. Built-in testing is one such technique that can be used to control or assess the quality of component-based systems.

One part of a quality assurance plan determines the set of quality assurance techniques that should be applied in the software project, another part a set of test case selection techniques or test adequacy criteria. These two are fundamentally different. The first is concerned with theoretical background, models, and processes that will be applied and followed when a piece of software 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 post-conditions of individual test cases according to some testing criteria (test coverage criteria, random testing, equivalence partitioning). Built-in contract testing is clearly belonging to the first class. Though, since it requires and applies test suites, it has of course implications on the second class.

There are many standard test case selection techniques that may be applied within a development project. Which of these techniques will be used in a project is subject to careful consideration, and this is typically part of defining the quality assurance plan. We cannot provide guidelines on which test case selection technique is the best for a particular system. This is clearly out of the scope of this project and may be subject of a research project in its own right. What we do provide, however, is a proposition on how the models that collectively make up a system specification may be used to generate test suites for built-in testing. This is laid out in a separate section of this report (Section 3). There, we will focus on how test cases may be derived from the individual software specification artefacts that are created throughout the software development process. But now, we explain the specification artefacts in greater detail.

2.3 Component Specification

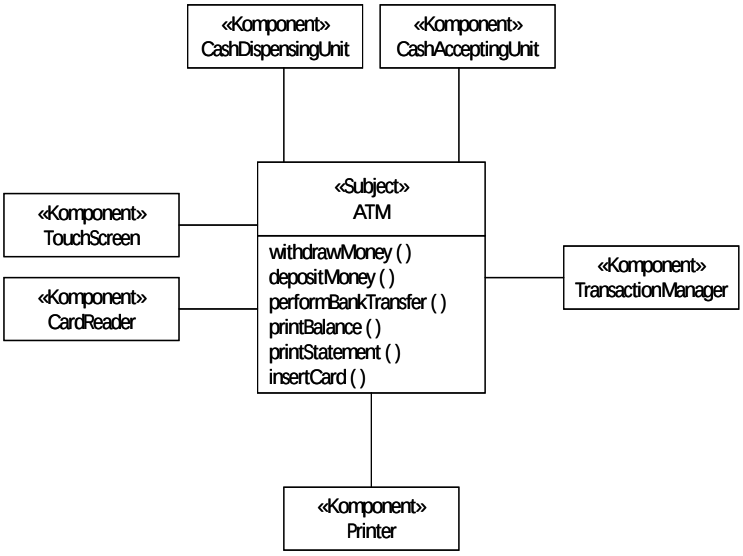
A specification is a collection of descriptive documents that collectively define what a component can do. Typically, each individual document represents a distinct view on the subject, and thus only concentrates on a particular aspect of what it can do. A component specification may be represented through natural language, or through some graphical representations and formal languages. Whichever notation is used, a specification should contain everything that is necessary in order to fully use the component and understand its behaviour. As such, the specification can be seen as defining the provided interface of the component. Therefore, the specification of a component comprises everything

that is externally knowable of its structure (e.g. associated other components, in form of a structural specification), function (e.g. provided operations, in form of a functional specification), and behaviour (e.g. pre- and post-conditions, in form of a behavioural specification). These parts are not mandatory and may change from project to project or from component to component. They rather represent a complete framework for a component specification. They 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. They are part of the quality assurance plan of the overall development project or the specific component. This was outlined in the previous sub-section. A complete documentation for the component is also desirable, and a decision model that captures the built-in variabilities that the component may provide. These variabilities are supported through configuration interfaces.

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 of component instances can have on the subject. Essentially, this maps to the prospective configurations of the subject, and thus its provided configuration interfaces. A structural specification is traditionally not used in software projects. Only the advent of model driven development approaches has increased its importance as specification artefact. As a UML class or object model, the structural specification provides a powerful means to defining the nature of the classes and relationships by which a component interacts with its environment, it is also used to describe any structure that may be visible at its interface [Atk01].

Figure 1:
UML-style structural specification.



The structural specification in KobrA is represented through a collection of UML class and object diagrams. An example for a class diagram is depicted in Figure 1. This displays the structure of the individual components that collectively make up an bank ATM system. The ATM system is the component that is developed from scratch, the other associated components are readily available third party components (CashAcceptingUnit, CashDispensingUnit, Printer) or in-house developments (TransactionManager, TouchScreen, CardReader).

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, this is its provided interface. A template for a complete functional specification for one single operation of a component is depicted in Table 1. The most important items of the list in Table 1 are the "Assumes" and "Result" clauses which represent the pre- and post-conditions for the operation. These are essential for testing the correctness of an operation. The "Assumes" clause defines what must be true for the operation to guarantee correct expected execution, and the "Result" clause 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 post-condition (compare with "design by contract" [Mey97]). 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. Pre- and post-conditions typically comprise constraints on the provided

inputs, the provided outputs, the state before the operation invocation (initial state), and the state after the operation invocation (final state) [Atk01, Gre01].

Any notation for expressing pre- and post-conditions may be used depending on the intended purpose and domain of the system. A natural choice for expressing them is the Object Constraint Language (OCL) defined in the UML standard [OMG00]. In the KobrA method, the functional specification is represented through a collection of operation specifications that are defined by the template depicted in Table 1.

Table 1: Operation specification template based on the Fusion method [Atk01, Col94].

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 component
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 pre-condition on the externally visible state of the component and on the inputs (in receives clause) that must be true for the component to guarantee the post condition (in the result clause)
Result	Strongest post-condition on the externally visible properties of the component and the returned entities (returns clause) that becomes true after execution of the operation with the assumes clause

The functional specification, that means the collection of all operation specifications, collectively define the provided interface of a component completely and sufficiently, including function and behaviour. However, depending on the complexity or the size of a component it may be difficult to understand and see the individual interactions with its environment. The behavioural specification defines this additional aspect of a component. It merely represents a different view on the component's specification (its operations), and in fact in the development process, the two are typically used to refine each other [Atk01]. The behavioural model shows a complete picture of the collection of all operation specifications, but it concentrates on the "Assumes" and "Result" clauses (the pre- and post-conditions) of all operation specifications.

Table 2: Example operation specification for the ATM component operation withdrawMoney.

Name	withdrawMoney
Description	On success: performs a withdrawal of a specified amount of cash from a specified bank account and prints out receipt on the Printer if requested. On failure: displays error message on TouchScreen, or locks CardReader
Constraints	cardLockRequest from Card locks the CardReader.
Receives	None.
Returns	On success: CashDispenseRequest to CashDispenser. Receipt to printer if requested. On failure: errorMessageto TouchScreen, and/or cardLock to CardReader
Sends	On success: withdrawalTransaction to TransactionManager
Reads	PinCode from TouchScreen. CustomerDetails from CardReader.
Changes	TransactionManager. CardReader. CashDispenserUnit.
Rules	Specified in the respective sub-operations (see activity diagram).
Assumes	Sufficient cash for CashDispenseRequest in CashDispenserUnit.
Result	TransactionManager is updated by withdrawalTransaction. CashDispensingUnit is updated by CashDispenseRequest.

2.3.3 Behavioural Specification

The object paradigm encapsulates 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. Our 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.

The purpose of the behavioural specification (or the behavioural model) is to show how the component behaves in response to external stimuli [Atk01]. It concentrates on the “Assumes” and “Result” clauses of the functional specification that define the pre- and post-conditions of an operation (Table 1 and Table 2). If the component has no states, then the pre- and post-conditions do not define an initial state for which an operation invocation is valid, or a final state in that 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. In this case, we do not necessarily need a behavioural 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 more focused on data cohesion whereas in traditional development we may refer to

functional cohesion, meaning functions are grouped into components or modules that have similar purpose. In any way, if the component is based on states, and most components are, the behavioural model expresses a big deal of the complexity of the pre- and post-conditions which are inherently defined in the collection of all operation specifications.

The behavioural specification (or behavioural model) describes the behaviour 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 [Gre01, Bin00]. 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 behaviour 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 [Gre01]. The behavioural specification in KobrA is represented through one or more UML state diagrams or state tables. An example for a state diagram is depicted in Figure 2, one for a state table is depicted in Table 3. Figure 2 displays the behavioural model for a banking card software that validates whether the card is activated with the right pin number. Table 3 represents the respective state table.

Figure 2: Behavioural specification in form of state diagram.

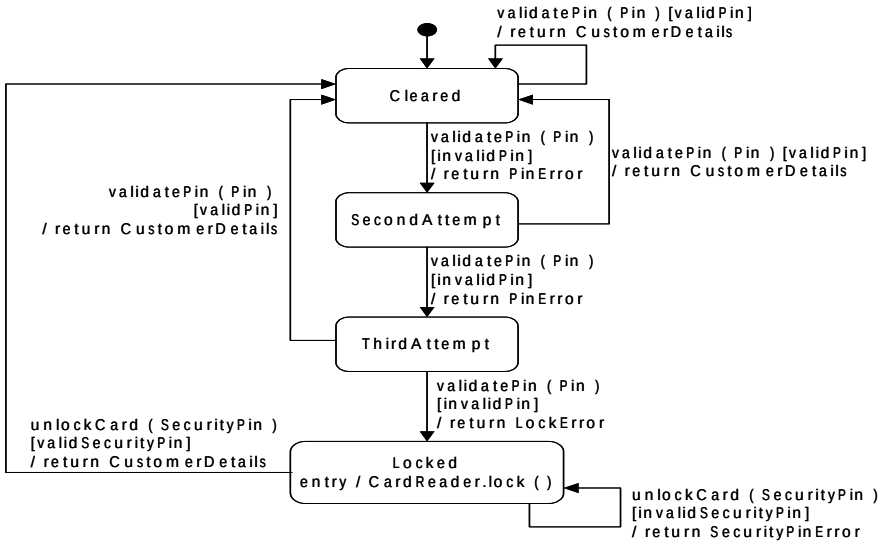


Table 3: Behavioural specification in form of a state table.

Initial State	Precondition	Event	Postcondition	FinalState
Cleared	[validPin]	validatePin (Pin)	return CustomerDetails	Cleared
Cleared	[invalidPin]	validatePin (Pin)	return PinError	SecondAttempt
SecondAttempt	[validPin]	validatePin (Pin)	return CustomerDetails	Cleared
SecondAttempt	[invalidPin]	validatePin (Pin)	return PinError	ThirdAttempt
ThirdAttempt	[validPin]	validatePin (Pin)	return CustomerDetails	Cleared
ThirdAttempt	[invalidPin]	validatePin (Pin)	return LockError	Locked
Locked	[validSecurityPin]	unlockCard (SecurityPin)	return CustomerDetails	Cleared
Locked	[invalidSecurityPin]	unlockCard (SecurityPin)	return SecurityPinError	Locked

The functional specification, that is the collection of all operation specifications, should contain sufficient information for creating the behavioural specification (see the specification the card operations in Table 4). Each specified operation represents one or several events in the behavioural model. Asynchronous events that affect the component as well also map to events in the state model. The "Rules" section in Table 1 defines the rules that govern the outcome of an operation in the operation specification, and the "Results" section defines that outcome. These rules define the different cases and input scenarios that affect the outcome of the operation. They are implicitly defined through the "Rules", "Assumes" and "Result" clauses (pre- and post-conditions). However, there is no standard process for developing an abstract state model from functional specifications. In the current state-of-the-practice this is a highly innovative and human-oriented activity. The development of a process for creating state-transition models represents a research project in its own right and is out of the scope of this report, so we do not go into more detail here.

Table 4: Functional specification for the banking card example (Figure 2).

Name	validatePin
Description	Validates a given Pin and, on success, returns the stored customerDetails. After three unsuccessful invocations (invalid Pin) the card is locked.
Constraints	cardLockRequest from card locks the cardReader (card is not returned to customer)
Receives	Pin: Integer
Returns	On success: customer details. On failure: invalid Pin error.
Sends	None.
Reads	None.
Changes	None.
Rules	Unless card is locked: return customer details. After third unsuccessful invocation [invalid Pin AND card not locked]: lock the card. After second unsuccessful invocation [invalid Pin AND card not locked]: allow one last unsuccessful attempt. After first unsuccessful invocation [invalid Pin AND card not locked]: allow two more unsuccessful attempts. After no unsuccessful invocations [invalid Pin AND card not locked]: allow three more unsuccessful attempts. One successful invocation clears the card from previous unsuccessful invocations.

Name	validatePin
Assumes	card not locked AND Number of unsuccessful attempts < 3
Result	(card locked AND Number of unsuccessful attempts = 3) XOR (card not locked AND Number of unsuccessful attempts < 3)

Name	unlockCard
Description	Unlocks a previously locked card, so that it may be used again.
Constraints	Only locked cards can be unlocked.
Receives	Security Pin: Integer.
Returns	On succes [valid SecurityPin]: CustomerDetails stored on the card.
Sends	On failure [invalid SecurityPin]: Security Pin Error.
Reads	None.
Changes	None.
Rules	On success [valid SecurityPin]: set card to cleared
Assumes	Card locked AND (valid SecurityPin OR invalid SecurityPin).
Result	(Card cleared AND valid SecurityPin) XOR invalid SecurityPin

2.4 Component Realization

A realization is a collection of descriptive documents that collectively define how a component is realized. A realization should contain everything that is necessary in order to implement the specification of a component. A higher-level component is typically realized through a combination of lower-level components that are contained within and act as servers to the higher-level component. Additionally, the realization describes the items that are inherent to 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 sub-components. In other words, the realization defines the specification of the sub-components, this is the expected interface of the component, and additionally it contains its own implementation. These items correspond to its private design that the client of the component does not see. The overall meta-model of a component with specification and realization is illustrated in Figure 6. This also underlines the importance of a quality assurance plan and quality documentation. However, these are not described in any further detail.

A component realization describes everything that is necessary in order to develop the implementation of the specified component. This comprises the specifications of the other server components upon which the subject component relies, as well its internal structure, and the algorithms by which it performs its specified functionality. Therefore, the realization comprises documents for specifying its internal structure (described in sub-section “Realization Structural Specification”), the algorithms by which it calculates its results (described in subsection “Realization Algorithmic Specification”), and the interactions with other

components, these are its own associated servers (described in sub-section "Realization Interaction Specification").

2.4.1 Realization Structural Specification

The purpose of the realization structural model is to describe the nature of the classes and their relationships out of which the component is made up (i.e. its sub-components), and the internal architecture of the component. In general, the structural model consists of a number of class and object diagrams [Atk01]. Figure 3 displays the UML class diagram representing the structure of an ATM component.

Realization class diagrams describe the classes, attributes and relationships between the classes out of which a component is made up. The component that is the focal point of the diagram is augmented with the stereotype <<subject>>. The realization class diagram is typically a refinement of the specification class diagram, so it contains a superset of the information represented in the specification class diagram. All elements that are displayed there, are also relevant to the realization, but here they are described 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.

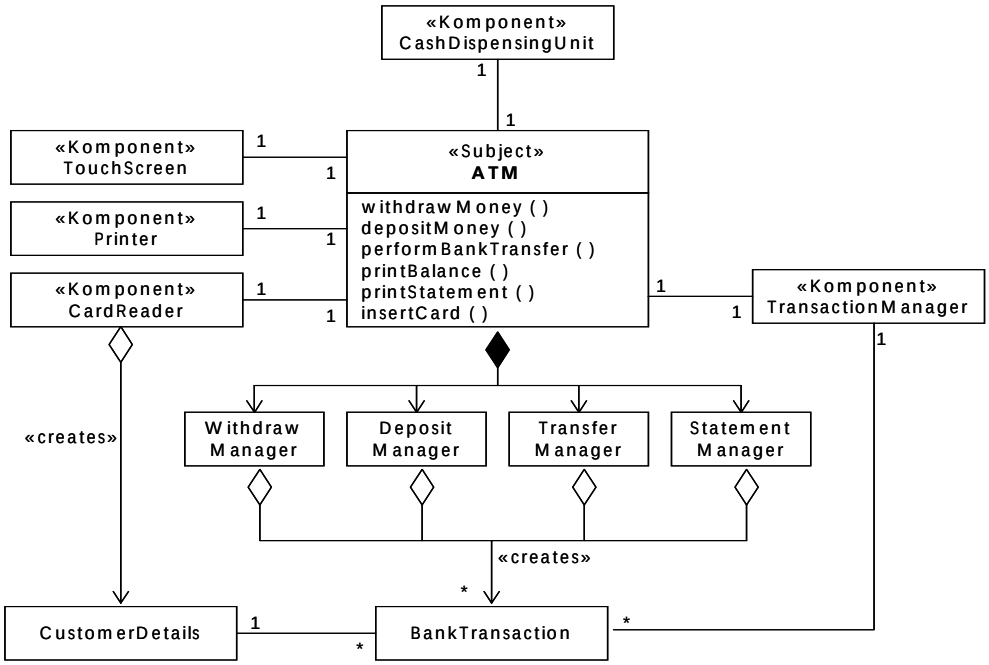


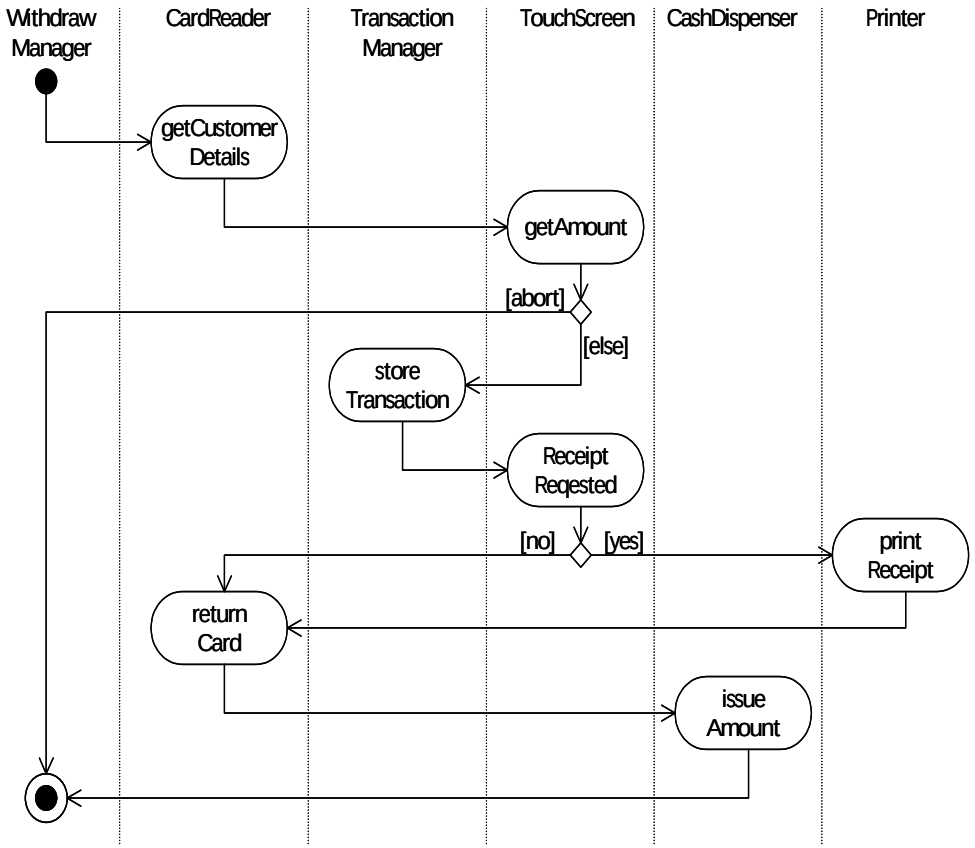
Figure 3: Realization class diagram for the ATM component.

2.4.2 Realization Algorithmic Specification

The algorithmic specification is comprised of a number of activity specifications that describe the algorithms by which the operations of a component are implemented. A UML activity diagram is a special kind of statechart diagram in which all the states are actions, these represent the nodes in a typical flow-graph notation, and all transitions are triggered by the completion of the actions. Figure 4 shows the UML activity diagram for the ATM's operation *withdrawMoney*. This represents a high-level of abstraction specification artefact, and it is quite likely decomposed into finer grained parts (they are not displayed here). Essentially, the activities in the diagram map to private individual procedures belonging to the classes that are labeled in the UML "swimlanes". These operations are typically further refined in subsequent activity diagrams that are related to the respective objects.

Activities which are not operations of a component may also be refined and specified through activity specifications that take the same form and shape of operation specifications (example is displayed in Table 4).

Figure 4: Activity diagram for the operation *withdraw Money* of an ATM component.



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 [Atk01]. Interaction diagrams describe how a group of instances collaborate to realize an operation, or a sub-activity thereof. Figure 5 displays a corresponding UML collaboration diagram for the activity displayed in Figure 4.

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.

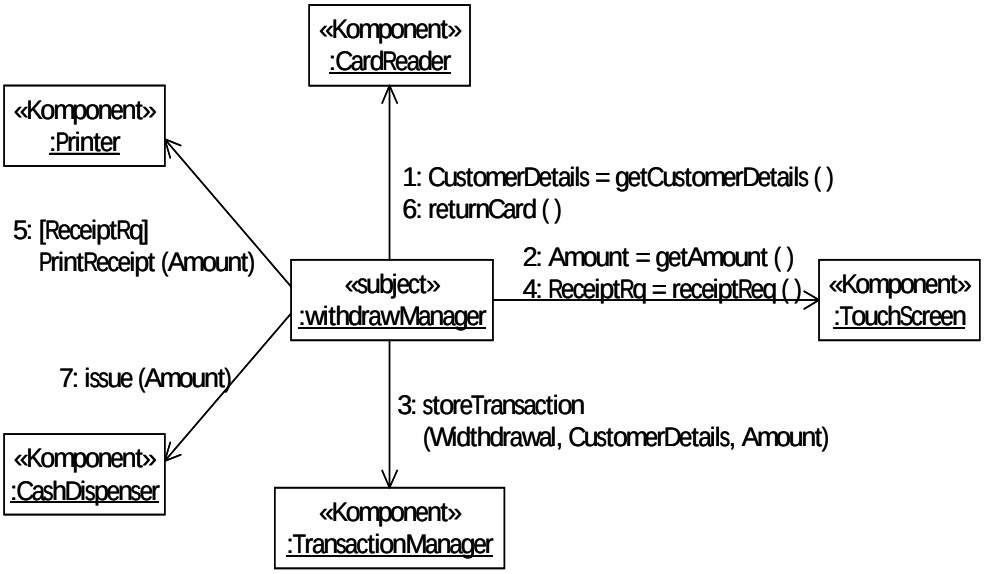


Figure 5: Collaboration diagram for the ATM operation *withdrawMoney*.

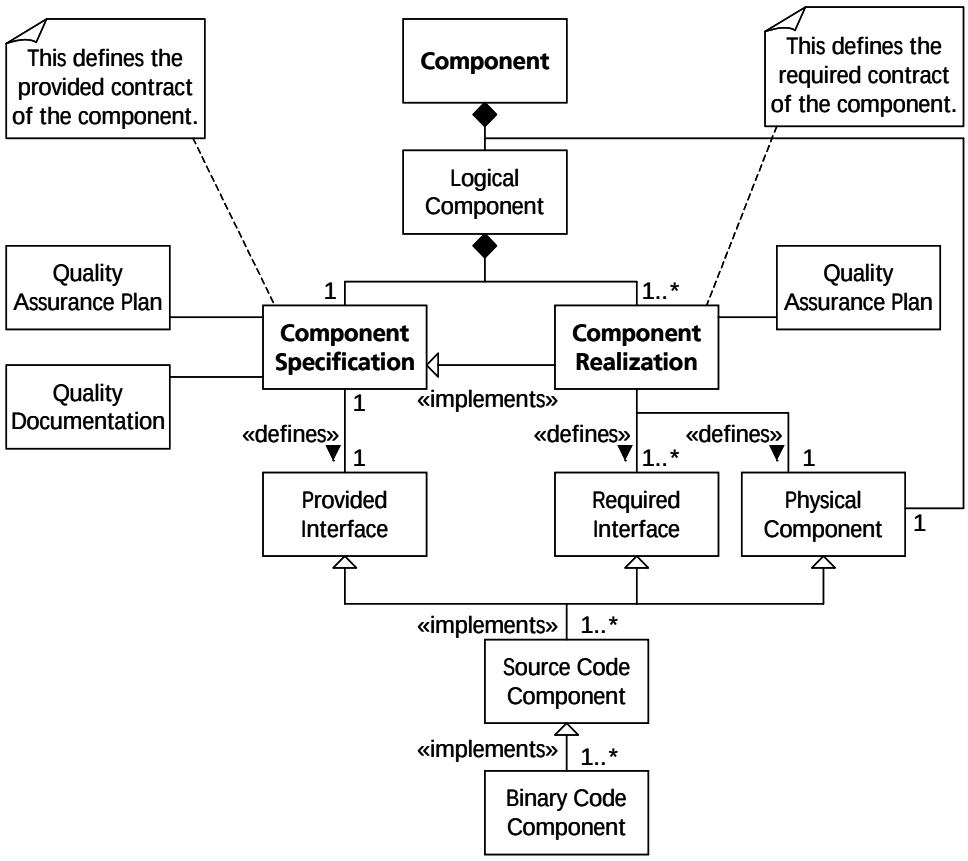


Figure 6: Metamodel that describes the concepts of a Component.

2.5 System Specification

Component realization and specification represent the overall logical component that exists only in form of specification documents. In this case this is mainly models plus the operation specifications. The realization is additionally comprising a description of the physical component. This describes how the individual component operations will be implemented and how these access the associated server components whose individual interfaces are laid out in the specification of the required interface. Each of these will be implemented as a component in its own right with all the component specification artefacts that have been introduced in the previous sections. This comprises everything that is externally knowable of its structure (e.g. associated other components), function (e.g. provided operations), and behaviour (e.g. pre- and post-conditions). Consequently, component realizations map to sub-ordinate component specifications, and vice versa. The source code component implements the provided interface and the operation calls to the required interfaces in the chosen programming language, plus the code that realizes internal operations of the physical component. Such a source code component is implemented through a

binary component. Figure 6 displays the meta model of a component that is used under this context.

Specification and realization provide a precise and abstract description of what a component does and how it works. Additionally, they define the provided and required interfaces of the component. These represent the contracts of the component on the basis of which it interacts with its environment (other associated components). This is also concerned with other important relationships of components, namely containment and clientship. These are important concepts that describe how components are related to other components in a hierarchical tree. Components are comprised of other sub-components that are comprised of sub-ordinate server components. In effect a higher-level component is a system in respect to the contained sub-components. A system and a component are therefore the same thing in respect to which level of abstraction is considered. In other words, a component may be a single stand-alone entity, and in this case referred to as a system in its own right, or it may only be part of a larger grained component, and thus only be referred to as a component. This follows the idea that sombody's component may be somebody else's system.

Containment

The specification of a system/component is always subject to containment rules relating a component to its parent component. A containment tree represents the way in which such relations are established. The UML component diagram in Figure 7 represents the containment hierarchy for a simple banking application that uses a web-service. The root node in a containment hierarchy is typically the context of the application. In its simplest form the context is the main method in a Java program that creates the instances of *LookUpTable* and *Bank*, for example.

Clientship

Clientship relations are the most fundamental associations between components or objects. They are subdivided into relations in which the component acts as a server, and relations in which the component acts as a client. The server relationship determines the services that a component must provide and the nature of the information that is passed through its provided interface. The client relationship determines the services that a component expects to get from other components and the nature of this interaction. The two are determined through the specification and the realization of the component. On an inter-component-level these relations are represented through UML style associations with additional stereotypes that indicate the nature of client/servership. Figure 7 represents the client-server relations in a simple banking application. The relations are defined through the arrows, the component to which the arrow points represents the server side of the relation, and the <<acquires>> stereotype. This indicates that the client is given a reference to an existing server instance, rather then creating its own server instance.

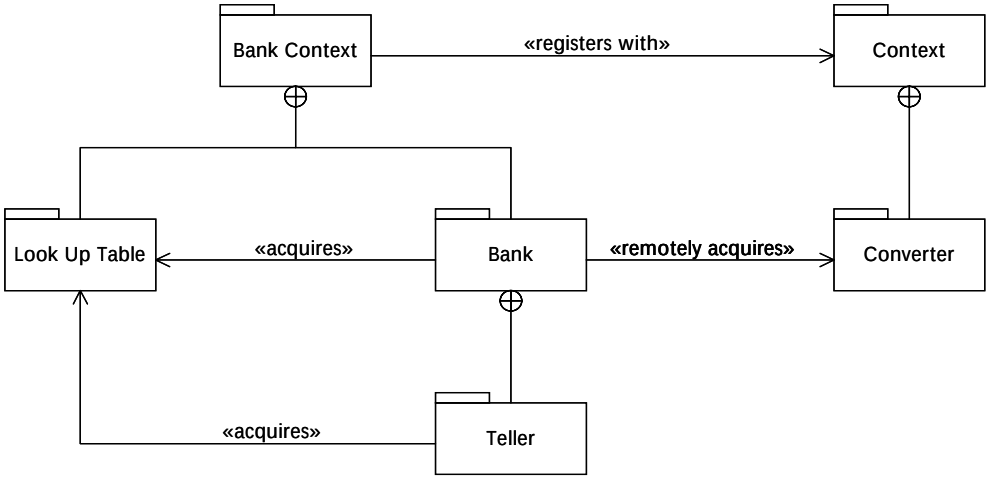


Figure 7: Containment and clientship relations in a banking application.

2.5.1 Context Realization

The previous sections have described the specification and realization of a component, which when applied in alternation form the basis for an iterative development process that is founded on recursive decomposition along the containment hierarchy. Every specification is defined in respect to an encapsulating realization, and every realization is defined with respect to an associated specification [Atk01]. This sub-section explains where this process initially starts.

The overall system, that means the collection of all components that make up an application, is represented through a component containment tree (as described earlier) whose initial root node represents the realization of the application's context or environment. This environment encapsulates the component or system very much like a normal component, therefore the terminology *context realization*. It provides interfaces (e.g. to the outside world of the component) and expects services from the component. Typically the context of a component or system is another system, a user interface or an existing business process into which the new application will be embedded.

A context realization is comprising the same fundamental artefacts as a normal component realization such as structural model, algorithmic model and interaction model (see the section on Component Realization). Additionally, it has an enterprise model. This model focuses on the nature of the enterprise for which the system is built. It describes the relevant aspects of the enterprise in a way that completely ignores that the system will be developed as a computer system How to derive a context realization is detailed in [Atk01].

2.5.2 Relation Between Specification and Realization in Application Engineering

Applications are made up of many interacting components. Each of these components is defined by a full specification and a full realization. The specification models collectively describe what a component provides to its clients, and the realization models collectively describes how a component is implemented and what it requires from other components.

3 Test Case Selection Techniques

The previous chapter has introduced typical concepts and artefacts for specifying a component-based application. This should also be supported by a process that gives guidelines on when and how these concepts and artefacts should be applied and produced. However, a description of such a process is not part of this report. This may be found in the standard literature on development methods [RBP91, Col94, SGW94, Rob92, RWL96, DW98, AF98, CD00, KCN90, WL99, BFK99, JBR99, Kru00, GHY97, MLH87, and Atk01]. This chapter introduces the most important test case selection techniques that may be used with the built-in contract testing approach. They are particularly aimed at component-based application testing and suitable for model-based testing, but any other testing methodology from the standard literature [Bei90, Bei95] may also be applied. This of course depends upon the quality criteria that are defined in the quality assurance plan for a component-based development project (see the section on the Definition of a Quality Assurance Plan). An investigation on which testing techniques are suitable under which circumstances is presented in [Veg00].

Since the BIT technology is concentrating on software components, built-in contract testing is primarily related to functional testing techniques, that is also referred to as black-box testing. A better term for functional testing that actually reflects its original definition is specification-based testing. These are test case generation techniques that only concentrate on the externally visible function and behaviour of components according to their specifications [IEEE99]. It means that when we devise the test cases for a component we do not apply any internal knowledge of its implementation. We cannot therefore base component testing on structural testing criteria. This is quite natural for externally acquired third party components (i.e. COTS components), anyway, because we do not know their internals. Built-in contract testing is therefore initially developed for applications that are consisting of third party components, comprising third party in-house developments.

However, larger grained components, from third parties or in-house developments, are made up out of smaller grained components. These are assembled together in order to built the larger grained components just in the same way using the same fundamental principles as for entire applications. The only difference between the two levels of abstraction (component level and application level) is that the individual parts of the larger grained components will always stay integrated together as encapsulated module once they have been developed. Contract testing may therefore be applied in order to integrate these individual parts into meaningfully configured encapsulated components but once

this integration has been performed there is no need for constantly having the contract testing artefacts built into the components. Figure 8 illustrates this. On application level (inter-component level) we have two components A and B as reusable assets that are assembled and put together. Component A contains another tester component (in this case the class Component B Tester) that checks the server component B. On component level (intra-component level) we might have tester components as well, such as Y Tester in component B that checks its server Y. Since in most organizations components are the primary reuse artefacts, it is unlikely that they will change very often internally.

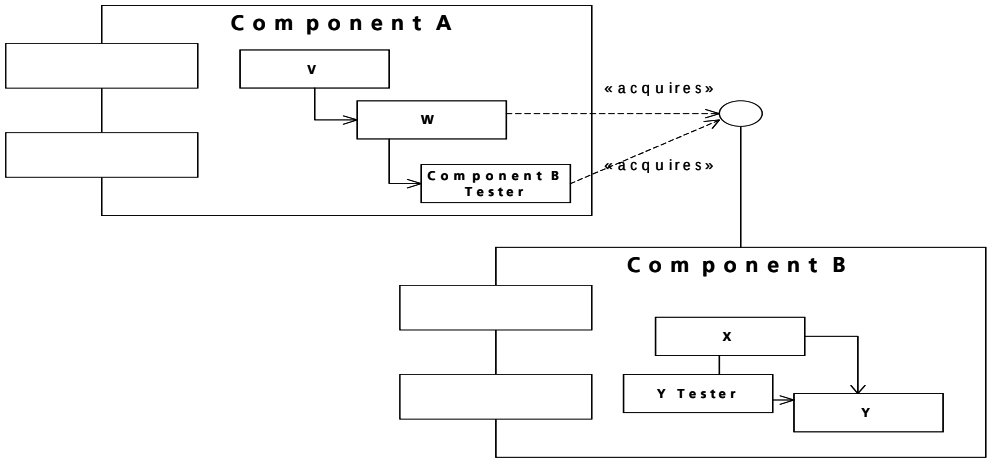


Figure 8: Component-level contract testing and application level contract testing.

Therefore, after all classes that are internal to component B have been put together and checked through contract testing, and this in fact is a typical integration test although on an intra-component level, the built-in test software becomes obsolete. Hence, the only difference between application level integration testing and component-level integration testing is that in the second case the built-in test software should be removed after the integration test. Application integration testing and component white-box testing are therefore essentially the same under the contract testing paradigm. Tester components are defined in Chapter 4 where the built-in contract testing artefacts are described in greater detail.

The following sections introduce functional or black-box test techniques, and after that, structural or white-box test techniques are considerd. An important issue in object and component testing is that black and white box testing cannot be separated strictly. Component engineering takes a fractal-like view on software systems where components are made of components that are made of components. The terminology of black and white boxes has only a meaning for the level of abstraction that we are looking at. A white box test for a super-ordinate component maps to a black box test for a sub-ordinate component and so on. Therefore, in some sense in component engineering we are also only deal-

ing with black-box test techniques. Real white-box testing in its traditional meaning is only concerned with testing the code that the physical component is providing by itself, and not the black-box specification that it contains to describe its sub-components.

3.1 Functional Testing Techniques

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 [IEEE99]. It is also referred to as black-box testing, or specification-based testing, which is a much more meaningful and unambiguous terminology.

Binder [Bin00] calls these techniques responsibility-based testing. This comes from the notion of a contract [Mey99] between two entities that determines their mutual responsibilities. For example, meeting the contracted pre-condition assertion is the client's responsibility, and meeting the post-condition is the server's responsibility, because this is what it promises to provide after completing a request [Bin00]. 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.

3.1.1 Domain Analysis and Partition Testing

Domain analysis may be used as input selection technique for all other subsequently introduced test case generation techniques. Domain analysis techniques are mainly applied in typical numerical processing software applications. It replaces the common heuristic method for testing extreme values and limit values of inputs [Bei95]. 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 in the input space belong to the domain of interest [Bei95]. Domain analysis is used for and sometimes also referred to as partitioning testing.

Equivalence Partitioning

Most functional test case generation techniques are based upon 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 are belonging to the same partition, for example equivalent behaviour (state-transitions). Proportional equivalence partitioning, for example, allocates test cases according to the probability of their occurrence in each subdomain.

Category Partitioning

Category partitioning is traditionally used in 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, or memory access exceptions, and the like.

3.1.2 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 effectively any type of testing is 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 a state-transition. In other words, pre- and post-conditions of every single test case must consider states and behaviour. Binder [Bin00] presents a very thorough investigation 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 used in this report that are described later on (state checking operations). The major test case design strategies for state-based testing are described in the following:

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 behaviour, so it is only accidentally effective at finding behaviour faults. It is feasible 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 [Bin00].

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, this 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 [Bin00], and it is also a method sequence based testing technique (see Section 3.1.3).

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 [Bin00].

3.1.3 Method-Sequence-Based Testing

This test case generation technique concentrates on the correct implementation of a component's combinations, or sequences of provided operations. Test case design is based on the behavioural model, such as a UML state chart diagram. Here, the paths through the state model are checked. This may also include multiple invocations of the same operation. Essentially, this applies all state-based testing coverage criteria that have already been introduced in the previous section (see Section 3.1.2). Table 5 shows feasible test cases that are derived based on method sequences for the banking card behavioural model displayed in Figure 2. This particular example is based on round-trip path coverage (for the first round-trip: from state Cleared back to state Cleared, Figure 2).

Table 5: Test case design based on method sequences according to the behavioural model of the banking card.

#	Initial State	Precondition	Event	Postcondition	FinalState
1	Cleared	[validPin]	validatePin (Pin)	return CustomerDetails	Cleared
2	Cleared	[invalidPin]	validatePin (Pin)	return PinError	
		[validPin]	validatePin (Pin)	return CustomerDetails	Cleared
3	Cleared	[invalidPin]	validatePin (Pin)	return PinError	
		[invalidPin]	validatePin (Pin)	return PinError	
		[validPin]	validatePin (Pin)	return CustomerDetails	Cleared
4	Cleared	[invalidPin]	validatePin (Pin)	return PinError	
		[invalidPin]	validatePin (Pin)	return PinError	
		[invalidPin]	validatePin (Pin)	return PinError	
		[invalidSecurityPin]	unlockCard (SecurityPin)	return SecurityPinError	
		[validSecurityPin]	unlockCard (SecurityPin)	return CustomerDetails	Cleared
...	...	...	...	...	...

3.1.4 Message-Sequence-Based Testing

This 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 [Chu99].

3.2 Structural Testing Techniques

Structural testing takes into account the internal mechanism of a system or component for generating test cases. It is also referred to as white-box or glass-box testing, or more expressively as implementation-based testing [IEEE99, Bin99]. Implementation-based testing evaluates observable behaviour with respect to a test model that is derived from the realization or implementation of the component. There is a vast number of structural testing techniques in the standard literature [Bei90], however, for component-based application engineering its ben-

efit is rather limited. The internals of components may be validated through tests that have been developed according to structural criteria, but application integration testing techniques such as built-in contract testing are solely based on black-box testing.

3.3 Model-based Testing and Testing Techniques

Model-based testing is a relatively old idea that has undergone a renaissance recently through the introduction of model-based development and the OMG's Model Driven Architecture [MDA]. Model driven testing follows two different routes. The first one is to have a meta-model for testing concepts, or a so-called testing profile that is currently being developed under the OMG umbrella [UMLT]. Figure 9 displays an example concept space for a test case in form of a UML class diagram. The OMG approach goes down that line, although their concept spaces are laid out in a much more detailed and comprehensive way. The second route to model-based testing concentrates on how tests may be derived from graphical notations such as the UML. Such techniques are traditionally used for the development of safety critical and real-time systems, and more recently it concentrates upon approaches on how to derive test information from individual UML models. Binder gives an overview on which UML models may be used for which types of testing [Bin99].

3.3.1 Test case selection and test information extraction techniques from models

Models represent a solid foundation for test case generation that is primarily based on the specification, and therefore mainly functional. Models use powerful (semi-) formal abstract notations in order to express requirements specifications. Having good requirements is crucial not only for the development of a system but additionally for the development of its testing infrastructure. If requirements are additionally testable they are the perfect source for instant test scenario generation.

Class diagrams

Class diagrams represent structure, that is associations between entities plus externally visible attributes and operations of classes. They are a valuable source for testing. Specification class diagrams (server) represent the interfaces that individual components export to their clients and therefore show which operations need to be tested, which operations support the testing and which external states are important for a unit. These can directly guide the construction of tester components for a server component. Realization class diagrams (client) represent the operations of the servers that a client is associated with. They only contain externally visible server operations and attributes that a client is actually using. It means such a diagram restricts the operational profile of a client in terms of operations. This helps to determine the range of operations that a tester component must consider. Class diagrams may be used to generate test

cases according to boundary conditions and component interaction criteria [Bin99].

State diagrams

State diagrams are a valuable source for testing in many ways. This has already been demonstrated in previous sections of this report (see Section 3.1.2 and Section 3.1.3) and in the literature [Bin95, Off99, Rob99], and it is one of the main sources for test case generation in built-in contract testing. State diagrams concentrate on the dynamics of components in terms of externally visible states and transitions between the states. State chart diagrams may be used to generate test cases according to class hierarchy and collaboration testing criteria [Bin99].

Interaction diagrams

While state diagrams concentrate on the behaviour of individual objects, UML collaboration diagrams represent the behavioural interactions between objects. They describe how the functions of a software are spread over multiple collaborating entities and how they interact in order to fulfil higher-level requirements [Abd00]. Collaboration diagrams represent two views on an entity, a structural view, and a behavioural view. Additionally, they pose constraints on a system. Since collaboration diagrams realize a complete path for a higher-level use case they may be used to define complete message sequence paths (see Section 3.1.4) according to the use case [Abd00]. This leads to testing coverage of a use-case-based requirements document. Interaction diagrams are useful to control testing criteria such as round-trip sequences and polymorphic testing [Bin99].

Package diagrams

KobrA's notation for a component is the UML package diagram since a KobrA Component combines class-like and module-like properties. This is elaborated in [Atk01]. For testing they represent a similar source as Class diagram although on a coarser grained level of abstraction. In built-in contract testing component diagrams (component trees) are used to indicate variability in an application and therefore mark the associations between components that need to be augmented with built-in contract testing artefacts. This is further explained in Section 4.1 and depicted in Figure 10.

Use cases operational profiles and scenarios

Many organizations define use cases as their primary requirements specifications, for example [Mey98]. Additionally, they use operational profiles in order to determine occurrences and probabilities of system usage. Use case models thereby map to operations in an operational profile. Another application of use cases is the generation of state chart diagrams [Quasar] from use-case driven requirements engineering, or the generation of collaboration diagrams, as briefly described in a previous paragraph. Use cases may be used to generate test cases according to combinational function and category partitioning criteria [Bin99].

Scenarios are used to describe the functionality and behaviour of a software system from the user's perspective in the same way as use cases do this. Scenarios

essentially represent abstract tests for the developed system that can be easily derived by following a simple process. This is laid out in the SCENT Method [Rys99].

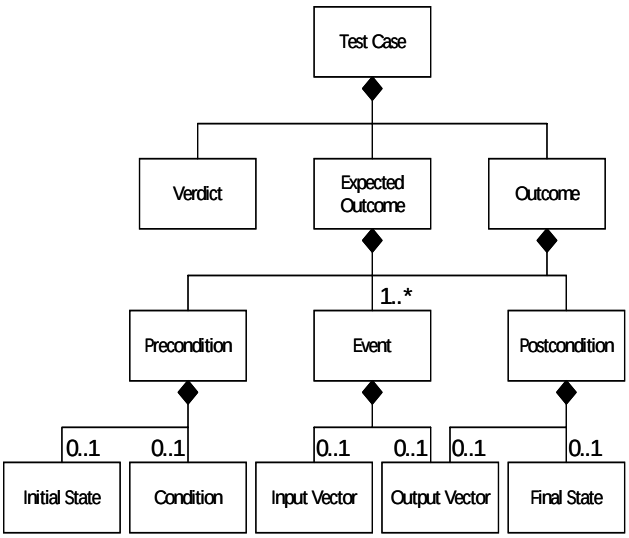


Figure 9: Concepts of a test case displayed through a structural model.

4 Specification of the BIT Artefacts

The previous chapters have laid out the foundations for the development of the BIT artefacts. They can be seen as an entry criterion for using built-in contract testing. This is a sound development process that is ideally based on models, though other notations may be acceptable as long as they provide similar contents, plus a testable requirements specification (that is part of the method) from which the tests may be derived. This of course comprises a quality plan as well. This chapter discusses the two primary built-in contract testing artefacts, the server tester that is built into the client of a component in order to test the component when it is plugged to the client, and the testing interface that is built into (extends) the normal interface of the component and provides introspection mechanisms for the testing.

Meyer [Mey97] defines the relationship between an object and its clients as a formal agreement or a contract, expressing each party's rights and obligations in the relationship. This means that individual components define their side of the contract as either offering a service (this is the server in a client-server relationship) or requiring a service (this is the client in a client-server relationship). Built-in contract testing focuses on verifying these pairwise client/server interactions between two components when an application is assembled. This is typically performed at deployment time when the application is configured for the first time, or later during the execution of the system when a re-configuration is performed.

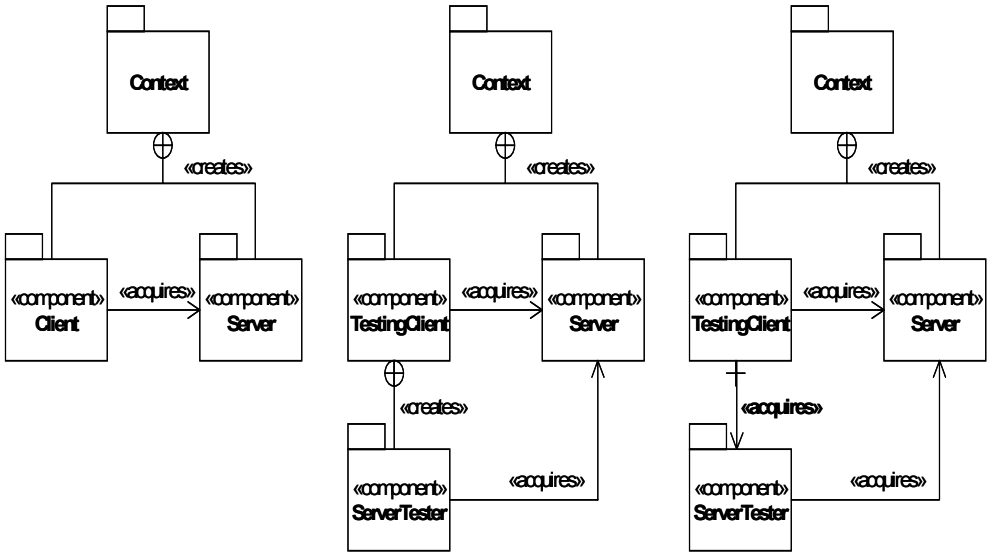
4.1 Built-in Server Tester Components

Configuration involves the creation of individual pairwise client/server relations between the components in a system. This is usually done by an outside "third party", which we refer to as the *context* of the components. This creates the instances of the client and the server, and passes the reference of the server to the client (i.e. thereby establishing the clientship connection between them). This act of configuring clients and servers is represented through the KobrA style <<*acquires*>> stereotype illustrated in Figure 10. The context that establishes this connection may be the container in a contemporary component technology, or it may simply be the parent object.

In order to fulfil its obligations towards its own clients, a component that acquires a new server must verify the server's semantic compliance to its clientship contract. It 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 in-built test software in form of a tester component as shown in Figure 10. This is called a server tester component, and it is executed when the client is configured to use the server [Cmp01]. In order to achieve this, the client will pass the server's reference to its own in-built server tester component. This is represented through an <<*acquires*>> association between the server tester component and the server in Figure 10. If the test fails, the tester component may raise a contract testing exception and point the application programmer to the location of the failure.

Figure 10: KobrA style component containment hierarchy without and with built-in contract testing.



A test involves the invocation of the methods of an associated component with predefined input values according to a precondition and the checking of the returned results against the expected results according to a post-condition. A test suite inside a server tester component contains a number of test cases that are developed according to testing criteria that are defined in the quality assurance plan (see Section 2.2). Any of the testing criteria from Chapter 3, or any other test case generation technique from the literature [Bei90] is considered an acceptable method of deriving test cases. This of course depends upon the type of project and the management of the project, for example safety standards may apply that require entry-exit-path coverage as test criterion.

A client that owns or contains a tester component and performs a contract test on its acquired server is termed a *testing client* or a *testing component* [Cmp+01]. The tester component may be a component in its own right, and it can also be acquired dynamically, not only owned by and contained in the client. This is illustrated through the bold <<acquires>> relation between *TestingClient* and *ServerTester* in Figure 10 in the right hand side diagram. This way of organizing the association between client and server tester provides maximal

flexibility. The server tester is attached to the client in the same way as the normal server through a configuration interface. Figure 11 displays a generic specification of these relationships in form of a UML class diagram.

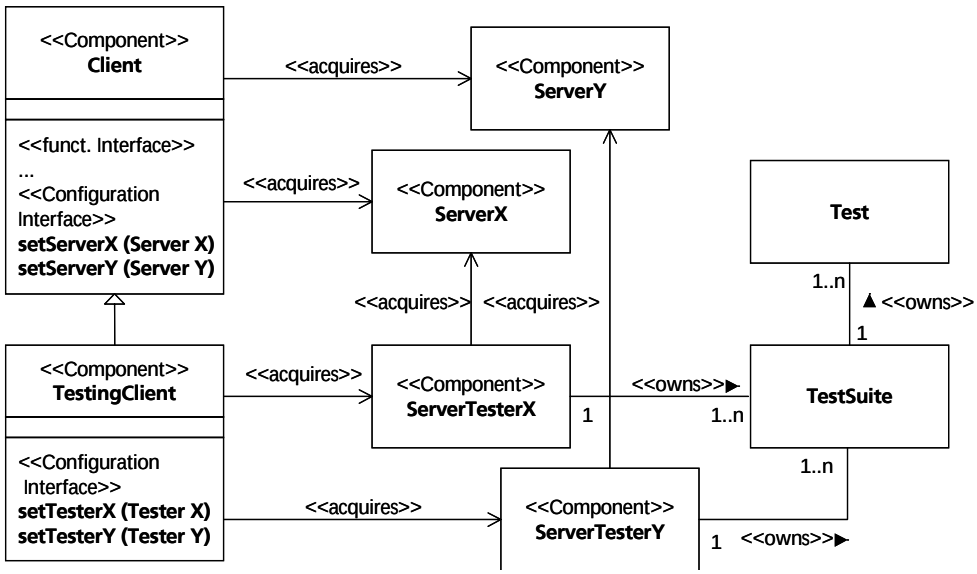


Figure 11: Generic Specification of a testing component and associated server tester components.

The *TestingClient* in Figure 11 extends the functionality of the *Client* component by a configuration that is used to set the type of tester that will check the associated servers of the *Client*. This is established by passing the references of the respective servers to the *TestingClient* (through call *TestingClient::setTesterX/Y*) whenever the configuration interface in the *Client* is setting a new server component. Figure 12 displays the configuration sequence for this example without considering the built-in testing, and Figure 13 displays the same sequence including built-in testing.

The testing artefacts represent normal development artefacts which are dealt with in the same way as any other development item. It means they fit naturally in the overall development method. In the KobrA Method such artefacts are additionally augmented with the stereotype <<Testing>> in order to separate testing functionality from normal (non-testing) functionality. This is important if the identifier of a testing component does not indicate that the component serves testing purposes and no consistent rules for identifiers are defined for a project. A component with the stereotype <<Component>> that serves testing purposes will therefore have the stereotype <<Testing Component>>. A method that is only used for testing will have the <<Testing>> stereotype as well. This corresponds to the strategy that the KobrA Method offers for handling variability. In this case it provides the stereotype <<Variant>> or <<Variant Component>> [Atk01]. This organization provides the highest level of flexibility for testing because we can develop a specific testing variant of an application

out of a more general application, that is without the testing, by using exactly the same principles for handling variability that are employed in product-line engineering. Through this mechanism we can easily instantiate an application that comprises all the testing, so we can check its integration. After successful integration we can create an instance of the original system without all the built-in testing and deploy it on an embedded controller, for instance.

Figure 12: Sequence diagram that specifies the configuration sequence of the scenario displayed in Figure 11 without built-in testing.

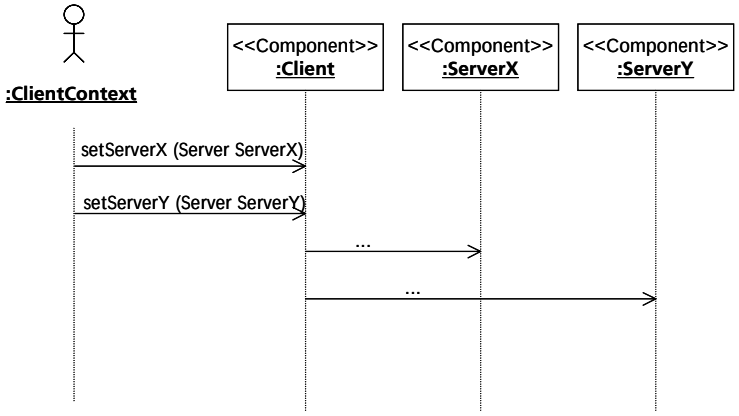


Figure 13: Sequence diagram with the configuration of built-in testing.

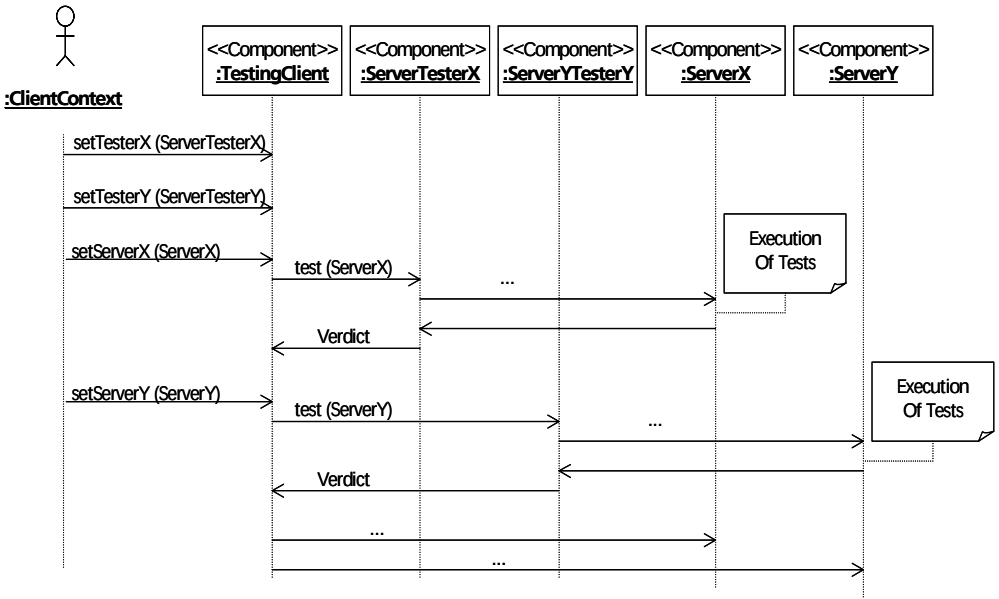


Figure 14 displays the class diagram for a testing client that shows points of testing variability represented through the stereotypes <<Testing>>. The tester components are also augmented with the stereotype <<Testing Component>>.

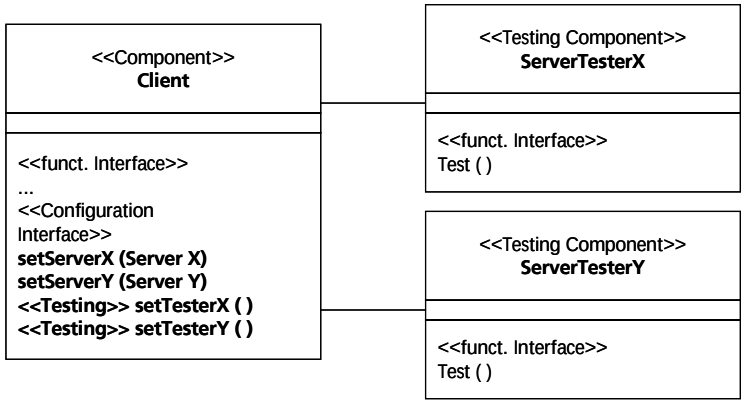


Figure 14: Structure of the testing client with stereotypes.

4.2 Built-in Testing Interface

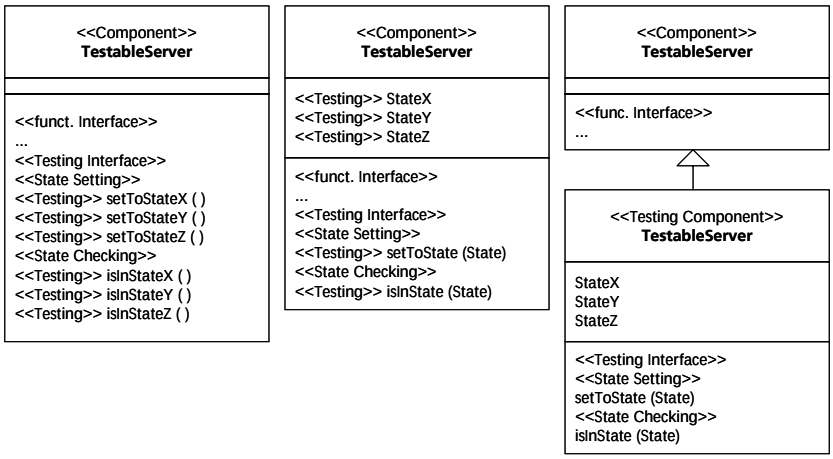
The previous section has concentrated on the specifications that need to be considered on the client side of a client-server relationship, the tester component, and how this is embedded into the normal development. This section defines how the server side in the client-server relationship may be augmented in order to support a client's built-in tester component.

The object-oriented and as a consequence the component-based development paradigm builds on the principles of abstract data types which advocate to the combination of data and functionality in a single entity. State transition testing is therefore an essential part of component verification. In order to check whether a component's operations are working correctly it is not sufficient simply to compare their returned values with the expected values. The compliance of the component's externally visible states and transitions to the expected states and transitions according to the specification state model must also be checked. These externally visible states are part of a component's contract that a user of the component must know in order to use it properly. However, because these externally visible states of a component are embodied in its internal state attributes, there is a fundamental dilemma.

The basic principles of encapsulation and information hiding dictate that external clients of a component should not see the internal implementation and internal state information. The external test software of a component therefore cannot get or set any internal state information. The user of a correct component simply assumes that a distinct operation invocation will result in a distinct externally visible state of the component. However, the component does not usually make this state information visible in any way. This means that expected state transitions as defined in the specification state model cannot normally be tested properly.

The contract testing paradigm is therefore based on the principle that components should ideally expose externally visible state information by extending the normal functional server as displayed in Figure 15. In other words, a component should ideally not only expose its externally visible signatures, but additionally it should provide the model of its externally visible behaviour openly. A testing interface therefore provides additional operations that read from and write to internal state attributes that collectively determine the states of a component's behavioural model.

Figure 15: Concepts of testable components and testing interfaces represented through a class diagram.



A component that supports its own testing by external clients through an additional testing interface is called *testable component*. Figure 15 shows three alternative ways of implementing a testable component. The first class on the left hand side represents a testable server component that has all built-in testing artefacts directly built-in. This is the normal functional interface plus the additional testing interface that comprises operations for setting and getting internal state information (*setToStateXYZ* and *isInStateXYZ*). These are the specified externally visible states according to the component's behavioural model. In the KobrA specification the state setting and checking operations are augmented with the *<<Testing>>* stereotype in order to indicate their special purpose. The diagram in the middle (Figure 15) represents an alternative implementation that has public state variables and only one state setting and one state checking operation that take the state variables as input instead of two setting and checking operations per state as in the first instance. This alternative representation is supported by the Java Library that comes with contract testing as an example support system (see Section 8.3 and [BBB03]). Additional flexibility will be achieved if the testing interface extends the normal functionality of the component as represented by the right hand side diagram in Figure 15. This implements the testing interface as a component extension in its own right so that the implementation of the testing software is encapsulated and strictly separated from the normal functional software.

The state checking operations of the testing interface verify whether the component is currently residing in a distinct logical state (for verifying the pre- and post conditions of a test case). The state setting operations set the component's internal attributes to represent a distinct logical state (for satisfying the preconditions of a test case). State checking operations are more fundamental than state setting operations. The latter may often involve quite considerable development effort. Sometimes they will re-implement exactly the same functionality as the normal interface. Thus, in most cases state setting will be achieved by invoking the operations of the normal functional interface. Subsequently, the state checking methods may be used to verify that the pre-conditions (initial state) for a test case are satisfied. In the server tester component, this is the client side of the contract, there are consequently two alternatives for implementing a test case. In the first way the state setting operations, if applicable, are invoked to ensure the preconditions required for a test case, the tested operation is invoked with the predetermined input parameters according to the testing criterion, and finally, the state checking operations are invoked to verify the post conditions required for a test case. The second way is applied when no state setting operations are provided by the tested component. Then, the operations of the component's normal functional interface have to be invoked to bring the component into the desired initial state for a test. Since these operations are part of the software that should be tested, the state checking operations have then to be invoked to verify the correct precondition for the application of a test case. Finally, the test method is called, and the state checking operations are used to verify the post conditions against the expected outcome. These two alternatives are illustrated through the sequence diagrams in Figure 16. Each of these alternatives depicts the same test case.

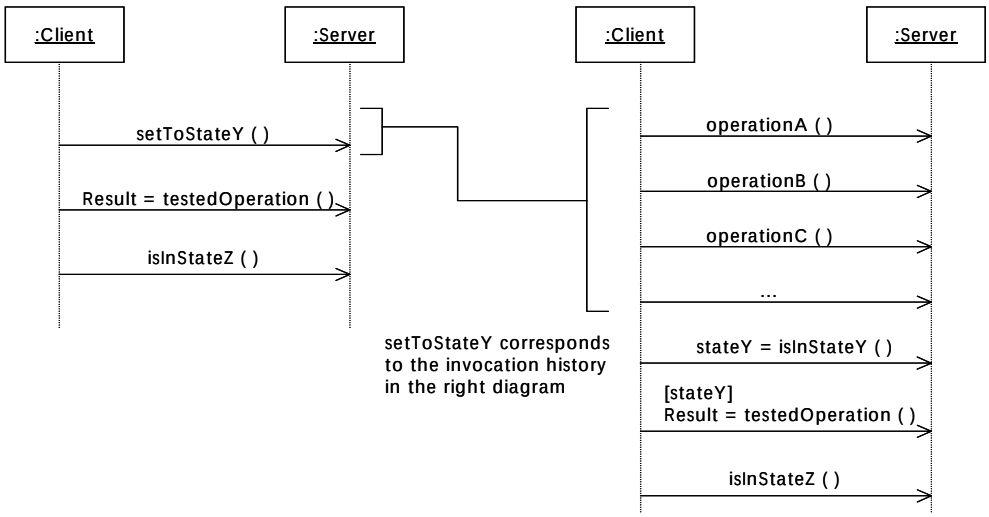


Figure 16: Alternative invocation sequences in a test case.

4.3 Associations between Components in Built-in Contract Testing

The previous sections have introduced the two primary artefacts that must be generated for built-in contract testing: tester component and testing interface. These represent additional functionality that is particularly aimed at testing. The first one extends the client component, and it comprises the actual test cases that check the client's deployment environment, that is its server components. The second extends the interface of the server in order to make the server more testable. If a server does not provide a testing interface (e.g. a COTS component) it does not mean that contract testing may not be used. It is simply limited in respect with controllability and observeability during a test, and the test cases in the client must be designed differently, that means according to the missing testing interface. This is because the test units that may be considered if no testing interface is available tend to be larger, because they have to be designed in a way that they always end up in an externally observable state. Otherwise we cannot check anything. Observeability is a prerequisite for testability. The tester component may be considered the more important part of built-in contract testing.

4.3.1 Associations between Client Component and Tester Component

In the client role, a component may own and contain its tester component. It 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 run-time configureability 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 sources. 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 which product line will be instantiated. A more loosely coupled association may be represented through a UML aggregation association, or more specifically through the KobrA stereotype *<<acquires>>* that indicates that the tester component is an externally acquired server.

4.3.2 Associations between Server Component and Testing Interface

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 in to the component and extends its normal functionality with some additional functionality that happens to be intended for testing purposes. Another approach is to augment the functionality of the 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 behaviour that is implemented through these smaller parts will be visible at the component boundary.

4.3.3 Associations between Tester Component and Testing Interface

The tester component of the client and the server's testing interface are interoperating in the same way as their respective functional counterparts. Since the testers and testing infrastructure is built into the system it is only additional functionality that happens to be executed when components are interconnected. The tester component must only "know" the reference of the tested server, and this is simply passed in to the tester component when the test is invoked by the client. Testing in this respect is only 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 only concerns the architecture of a system (i.e. which components will expose additional interfaces, which components will comprise tester components), the test cases that are applied during deployment are arbitrary, and they can be developed according to traditional criteria.

5 Development of the BIT Artefacts & Step-by-Step Process

The previous chapters and sections of this report have set the foundations for developing and using built-in contract testing. They can be seen as a required entry criterion, as a prerequisite, of introducing and implementing the technology. This comprises a sound development process that defines the required quality criteria in form of a quality plan, the specification and realization specification of each component, ideally according to the KobrA method with UML models and operation specifications, plus fundamental object and component technology features. The next sections provide a step-by-step guide for developing testing interfaces and tester components. That is guided by examples.

The following sub-section headings represent the respective steps that have to be taken. These steps are:

1 Identification of the Tested Interactions.

2 Definition and Modeling of the Testing Architecture.

3 Specification of the Testing Interfaces for the Identified Associations

4 Realization of the Testing Interfaces

5 Specification of the Tester Components

6 Realization of the Tester Components

7 Integration of the Components

5.1 Identification of Tested Interactions - Step 1

In theory, any arbitrary server-clientship relationship in component and application engineering may be checked through built-in test software. This is true for both development of individual components and assembly of components into meaningful configurations. Built-in contract testing is in this respect a multiple purpose testing technology that may be applied at all stages during unit and system development. This is because the typical object- and component-principle of client-servership is applied at all levels during development (even under the non-object paradigm) and built-in contract testing is inherently founded on that. The 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 abstraction and for all client-serv-

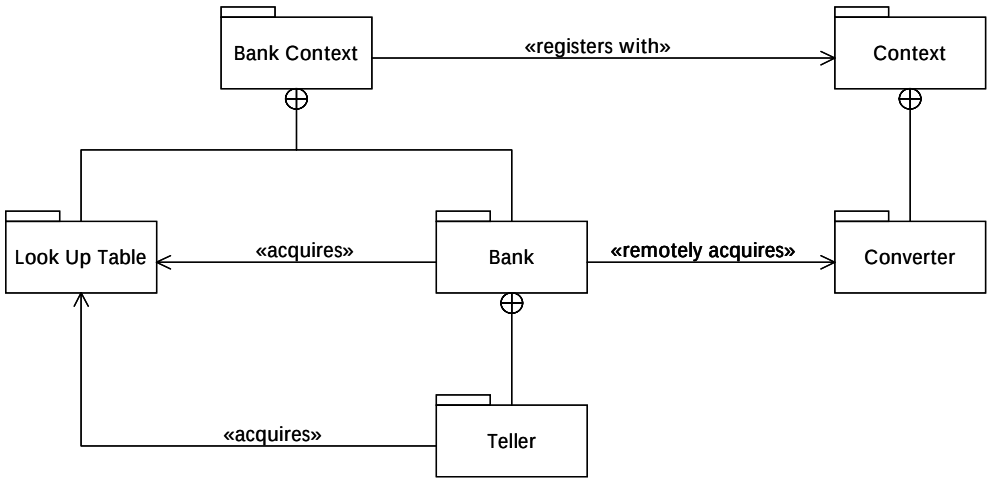
ership interactions, but the fundamental question is where it makes the most sense to have it built-in permanently. In other words, under which circumstances does built-in testing provide the greatest return on investment with respect to software reuse in component-based application development. Therefore we have to decide where in an application we have to built in the testing permanently, and where we can have it removed after integration. The development team must therefore answer the following questions:

– Where do we built in testing and have it removed after integration?
– Where do we built in testing and leave it permanently?

In general, any client-servership interaction may be augmented with a built-in testing interface and built-in tester component. These interactions are represented through any arbitrary association in a structural diagram, for example UML component, class and object diagrams, as well as KobrA composition-, nesting- and creation-tree diagrams. In other words, every nesting association represents client-servership. At least this is the case for UML because it provides no representation for creation associations and usage associations in contrast to the KobrA Method that does provide these (e.g. an instance that is created by a component but not used as a server by this component).

Associations between classes 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 artefacts. This may be implemented through development-or compile-time configuration mechanism (e.g. #include in C++), or through a run-time configuration interface that dynamically allocates tester components, and testable components with testing interfaces. Typically, reusable components will have permanent built-in testing interactions. 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.

Figure 17:
KobrA containment hierarchy for a simple banking application.



Example 1

Figure 17 displays a KobrA containment hierarchy for a simple banking application that shows the individual components and their clientship relations. The decision on which associations in the diagram will be augmented with built-in contract tester components and interfaces depends on the estimation of how likely the individual components will be replaced. Initially, any <<acquires>> association is a possible candidate for built-in contract testing. The diagram displays a distributed system that comprises a local part (bank context) and a remote converter part (context). The components of the bank context maybe more likely to stay integrated over a longer period of time, and may only be put together once. This integration may be checked by built-in contract testing. Since this part of the system is not likely to change in the future, the built-in testing artefacts remain only in the assembly throughout its integration and are later removed. For the remote associations this is different. The converter component is more likely to be changed over time (e.g. whenever the system is restarted, or whenever new exchange rates are required, etc.), and therefore it makes sense to have the contract testing artefacts permanently built in at that location.

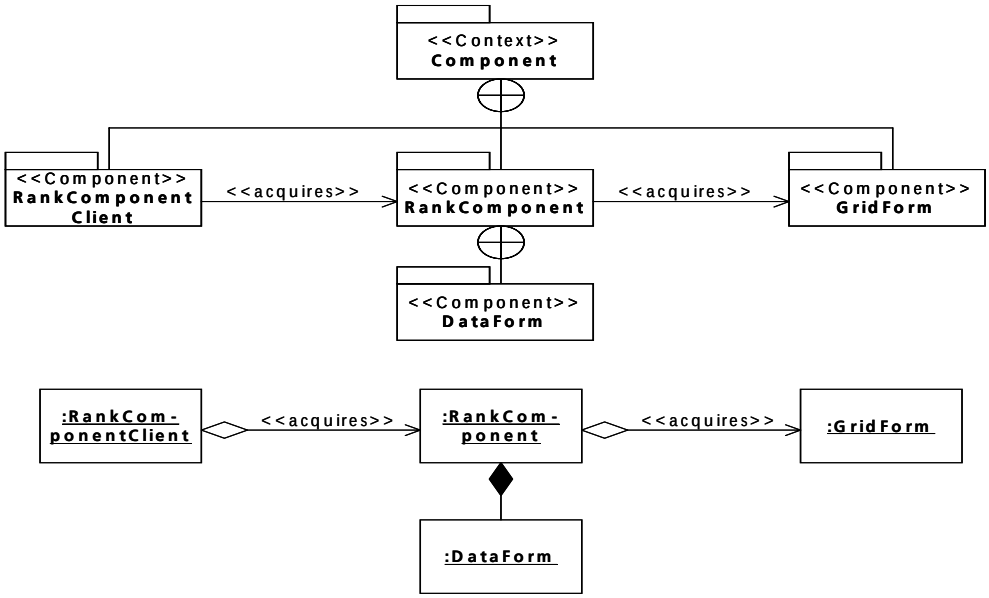


Figure 18: Example KobrA development-time containment tree (top) with the respective organization of the run-time instances (bottom).

Example 2

Figure 18 displays the development-time containment hierarchy and the run-time organization of an example component (*RankComponent*) that will also be used throughout this step-by-step guide. The UML collaboration between *RankComponent* and *DataForm* represents client-servership with component containment that will permanently remain established (i.e. *RankComponent* creates and holds the reference of *DataForm*). This means we can integrate the two as a single encapsulating unit and test that integration once and for all. This connection will not change in the future. The new component diagram will therefore not show the *DataForm* component any more, because it is completely hidden by the encapsulating unit *RankComponent*, that means it belongs to the realization of *RankComponent*. All other associations with the stereotype <<acquires>> are possible candidates for built-in contract testing nevertheless.

There are many different ways of enabling or disabling a component's functionality (in this case it is testing functionality), for example through a configuration interface. Under Java, this may be achieved through exploiting the extension mechanism plus a type cast.

Client and Server Source Codes

The following Java source code examples illustrate how this may be done:

```
class client {
  testableServer server; // acquired server
  bitTester tester; // acquired built-in tester
  ...
  // configuration interface
```

```
  public void setServer (testableServer s) {
    server = s;
  }
  ...
  public void setTester (bitTester t) {
    tester = t;
  }
  public performTest () {
    if (server != null || tester != null)
      tester.start ();
    else
      ; // if testing is not configured do nothing
  }
}

class server {
  // public interface
  ...
}

class testableServer extends server {
  // testing interface
  ...
}
```

Context Source Code

The context of the two components can enable or disable the built-in testing functionality.

```
class context {
  public static void main (String [] args) {
    client C = new client();
    server S = new server();
    testableServer tS = new testableServer();
    bitTester T = new bitTester();
    ...
    // client acquires a normal server
    C.setServer ((testableServer) S);
    // client acquires a testable server
    C.setServer ((testableServer) tS);
    // this sets the server tester in the client
    C.setTester (T);
    // this executes the test on the server
    C.performTest();
  }
}
```

5.2 Definition and Modeling of the Testing Architecture - Step 2

The locations of 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 according to the needs of the application (i.e. components 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 of the relationship.

The decisions on where we would like to integrate built-in contract testing artefacts must be somehow documented in the structure of the system. This may be regarded as a simple additional software construction effort in the overall development process that adds functionality to the application.

Example 1

Figure 19 displays the additional built-in contract testing artefacts that must be included in the simple banking application in order to check the compliance of the remote converter component to its clientship contract. The bank component that acquires the remote converter component as a server will have a persistent *ConverterTester* component built in that accesses the converter component for checking purposes. The server on the other hand, should ideally (but not necessarily) provide a testing interface that supports the testing. The additional testing functionality is indicated through the stereotype <<Testing ...>>. Adding the testing functionality results in an extended architecture of the system (Figure 19). The bank component is extended by the *converter tester* sub-component, and the *converter* component is replaced by the *testable converter* component (through extension).

Example 2

The extended system architecture of the second example is displayed in Figure 20. This shows the additional development artefacts that must be generated for built-in contract testing. These are indicated through the stereotype <<*Testing*>>. The modeling and realization of the <<*Testing*>> artefacts follows the same principles that are used for realizing the normal functionality (KobrA Method). In fact, these additional artefacts are realized through normal software development activities, they only happen to implement testing functionality, and this is the only peculiarity. Once the additional features (components) have been identified they need to be specified in detail. This is described in the next subsections, first for server's testing interface and then for the client's server tester.

Figure 19:
Containment tree with additional contract testing components for checking a remote server component.

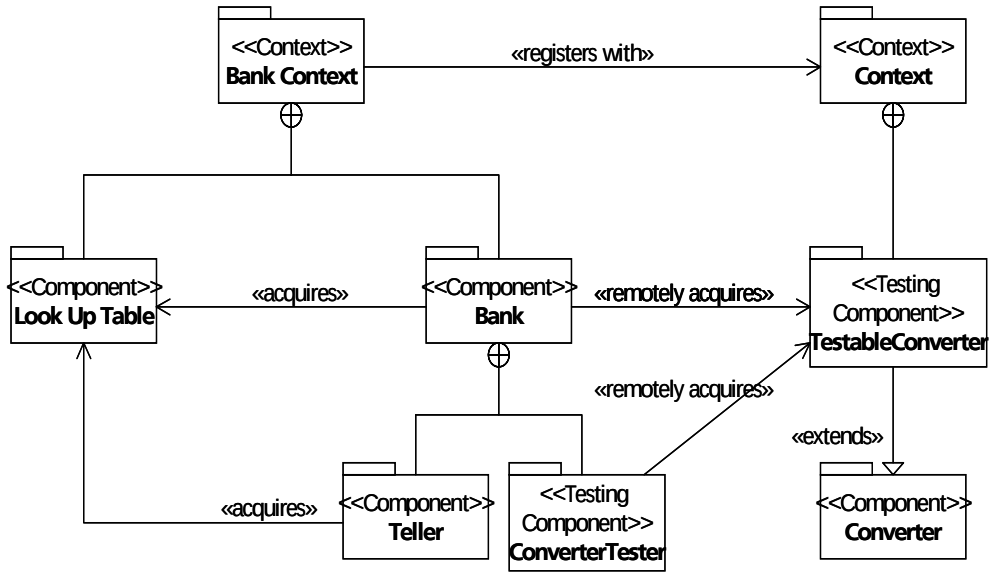
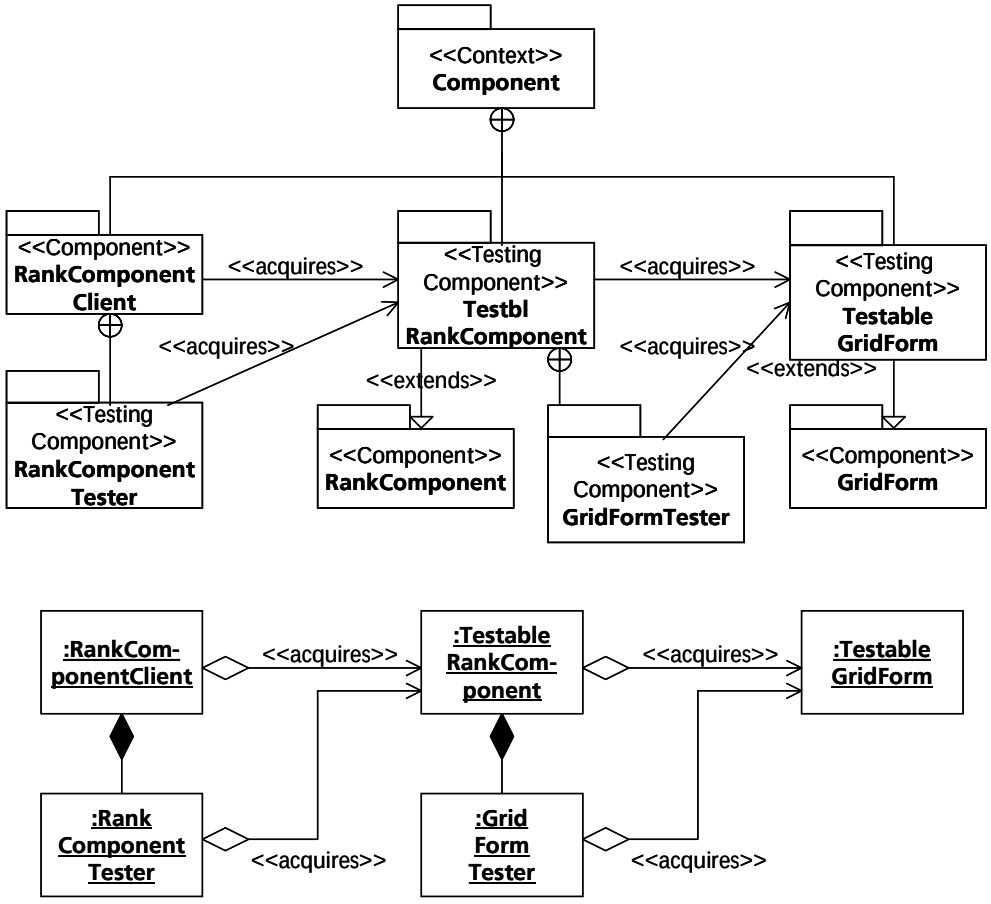


Figure 20:
New amended and extended architecture of the example application from Figure 18.



5.3 Specification of the Testing Interfaces for the Identified Associations - Step 3

This step comprises the specification of an individual testing interface for the server role of an association. It must be noted that it may only be performed if the component is an in-house development, with the exception that Java is used as implementation platform, or that appropriate technologies are in place that enable the extension of third party components, e.g. Java's reflection mechanism.

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 (Table 6), or the behavioural model. Both comprise sufficient information for development of state setting and state checking operations that augment the functionality of the original server component.

The additional testing interface is used to set and retrieve state information of the component. This is defined in the component's behavioural model. Each state in this model represents an item for which the behaviour or an operation is distinctively different from any other items. The individual states that the behavioural 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.

Table 6: Operation specification template based on the Fusion method [Atk01, Col94].

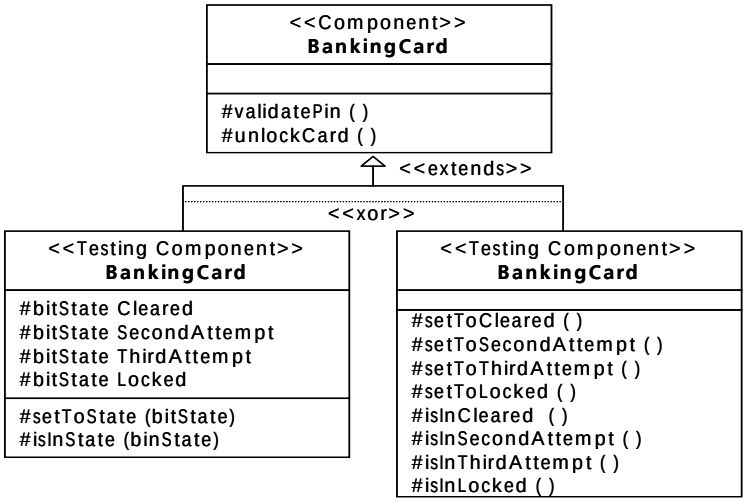
Name	**Name of the operation**
Description	identification of the purpose of the operation, followed by an informal description of the normal and exceptional effects
Constraints	Properties that constrain the realization and implementation of the component
Receives	Information input to the operation by 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 pre-condition on the externally visible state of the component and on the inputs (in receives clause) that must be true for the component to guarantee the post condition (in the result clause)
Result	Strongest post-condition on the externally visible properties of the component and the returned entities (returns clause) that becomes true after execution of the operation with the assumes clause

Example 1

See the specification of the ATM components in Section 2.3.2 and Section 2.3.3, and the behavioural model in Figure 2 and Table 3. The specification of the additional testing interface for the banking card example is displayed in Figure 21. It is derived from the behavioural model of Figure 2 that indicates four

distinct states: *Cleared*, *Second-* and *ThirdAttempt* for three unsuccessful trials of providing a correct pin number, and *Locked* for another unsuccessful attempt. Each of these map to two operations of the testing interface, one for setting the state, and one for checking whether the component is residing in that state. An alternative specification is the definition of each state as public attribute, plus two parameterized operations *isInState* and *setToState* that take these attributes as input.

Figure 21: Structural model of the banking card testing interface.



Example 2

The operation specification for the method *RankComponent::setDataTransfer-Area* is given in Table 7. Such a specification should be available for every single provided operation of the component *RankComponent*. The behavioral model of this component is displayed in Figure 22. The component is in charge of ranking data according to some algorithms. It provides many functions (only a part is considered here) and it is composed of an aggregation of many more sub components (e.g. *GridForm* and *DataForm* according to Figure 18). The component supplies the basic functionality to find similar cases using either continuous or categorical attributes or a combination of both, and the best attributes with a heuristic search. The best attributes with the heuristic search routine will return the complete set of attributes excluding the predicted attribute. Full error checking is implemented in this component (defensive development).

The operation *SetDataTransferAarea* creates an area where all the data can be stored while it is being analyzed, and this is the primary condition to perform all of the other operations. If this area is not created no other operations can be invoked. It is a volatile storage area used just for the purpose of the required calculations. Its operation specification is given in Table 7.

Table 7:
Operation specification for RankComponent::SetDataTransferArea.

Name	**RankComponent::SetDataTransferArea**
Description	On Success: create a storage area where all the data can be stored and whose size is given by the parameter passed to the function. Return an ERR_SUCCESS message. On Failure: return an ERR_MESSAGE (depends on the error type).
Constraints	None.
Receives	Number of cases and number of attributes to build up the storage size.
Returns	On Success: a message ERR_SUCCESS On Error: a message depending on the error occurred.
Sends	On Success: activate the DataForm.
Reads	Number of cases and number of attributes from the DataForm.
Changes	Nonne.
Rules	None.
Assumes	Another data storage does not exist. The number of cases is between 1 and 200-The number of attributes is between 1 and 200.
Result	Data Storage is created and it is ready to store data.

Figure 22:
Class and Behavioural model of *RankComponent*.

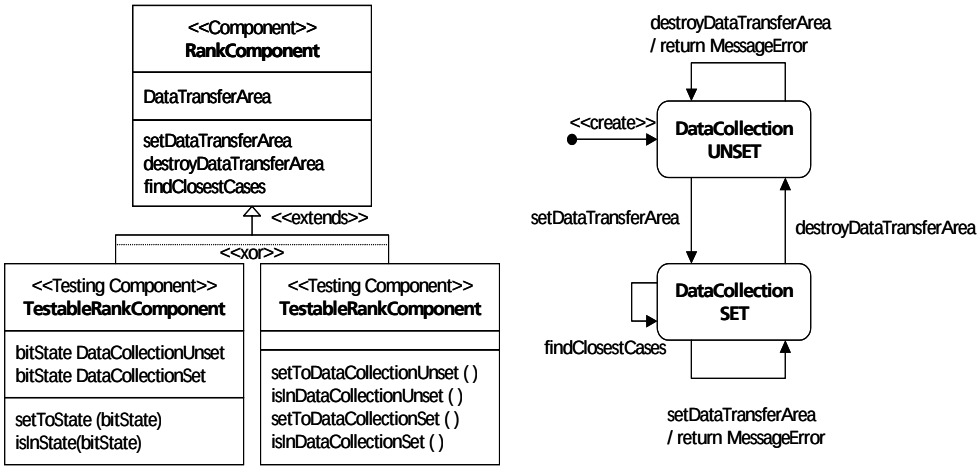


Figure displays the behavioural model of the component *RankComponent* (Example 2). This identifies two distinct states for which the externally observed behaviour of that component is different. That is if the state *DataCollectionUNSET* is true, the operation *setDataTransferArea* builts up an internal data structure so that the operation *findClosestCase* may be invoked, otherwise in the state *DataCollectionSET* this operation leads to nothing.

The behavioural model maps to the specification of the testing interface, as displayed in the structural diagram in Figure 22. Each state maps to two additional operations, one for bringing the component into that state, and one for checking whether the component is residing in that state. Two alternative implementations are feasible. The first one defines the states as public attributes that are

parameters for the two multiple purpose state setup and checking operations. The second one defines a state setup and checking operation for each individual state. Whereas the second method specifies a testing interface that is only consisting of two operations plus all defined states, the first one defines two operations for each state.

5.4 Realization of the Testing Interfaces - Step 4

The testing interface for each tested component will be specified according to the KobrA method, typically by a structural model (e.g. class diagram) that shows the signatures for each additional operation (see previous section). The realization of these operations depends heavily upon the realization of the functionality of that component.

Example 1

The banking card example exhibits four different states that represent a counter for the number of unsuccessful attempts of providing the correct banking card pin, so the implementation of the state setup and checking operations is straight forward. The realization of the testing interface is represented by the activity diagrams in Figure 23. The figure shows only the implementation of the testing interface for the first alternative, that is two interface operations, one for state setting and one for state checking, plus the public state variables that these operations may take as input parameters. The implementation for the second alternative has one of these operations for each respective state in the behavioural model.

Example 2

The realization of the *RankComponent* has some implications on the specification of the testing interface. This is explained in the following. The *setToState* operations of the testing interface in Figure 22 are specified to bring the component into some states, so that the *DataTransferArea* is accessible or not accessible. This is exactly what the two operations *setDataTransferArea* and *destroyDataTransferArea* do anyway. The implementations of the *setToState* operations are therefore exactly the same as the two operations of the normal functional interface. We could therefore simply call the respective normal operations as sub-operations from the newly specified testing interface methods, or we could re-implement the same functionality, ideally in a different way. Both solutions are complete nonsense, the first one, because it makes no sense to use something that we would like to check for correctness as a means to assess its correctness, and the second one, because nobody will actually implement the same thing twice merely for assessing it. A sensible approach to tackle this dilemma is therefore to implement only the state checking mechanisms (*isInStateDataAreaSet* and *isInStateDataAreaUnset*). These may be easily realized without code duplication, and they check whether the *DataTransferArea* has been created or destroyed correctly. The new specification, and additionally, the realization of the *isInState* operation are depicted in Figure 24. The contract testing interface in this case is only comprising a single method (or two methods depending on

the implementation according to the <<xor>> stereotype in Figure 24) that checks whether the DataTransferArea has been created or destroyed correctly. The parameters of the *isInState* operation control the respective states of the behavioural model (Figure 22) and the Reference represents the *DataTransfer-Area* that should have been stored correctly. Whether the second parameter is needed depends on the implementation of the *RankComponent* and who is actually providing or creating the instance of *DataTransferArea*.

Figure 23: Activity diagrams for the realization of a testing interface according to the structural diagram.

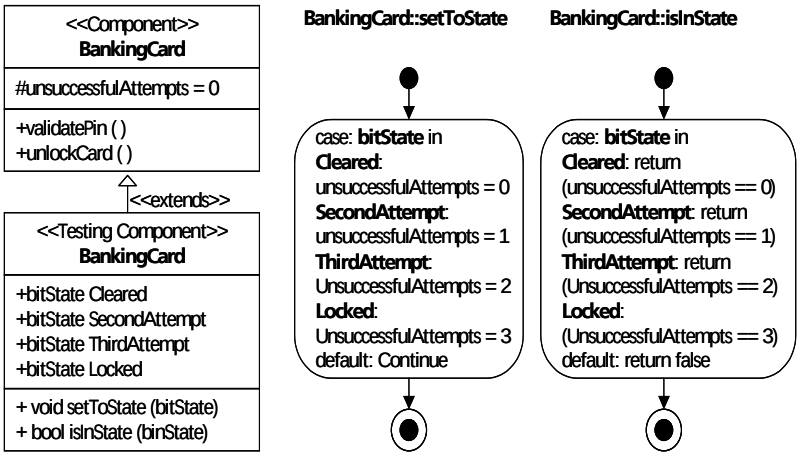
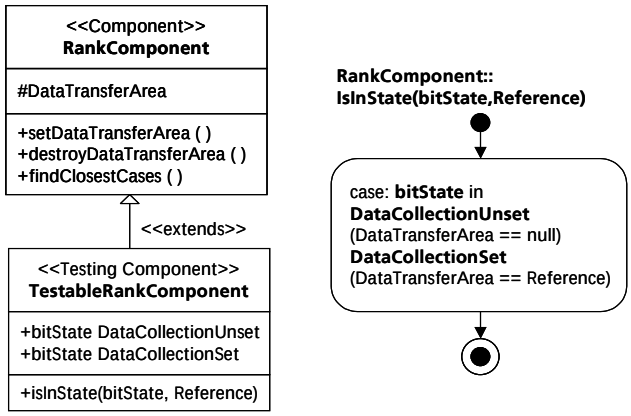


Figure 24: Specification and realization of the testing interface for *RankComponent*.



5.5 Specification of the Tester Component - Step 5

Step one 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 will be referred to as testing component in the following. Each of these testing components acquires a server, and it may be augmented with one or more built-in tester components (one for each server). Each of these tester components is developed according to the realization model of the testing com-

ponent. In other words, each testing component owns a description of what it needs from its environment (its associated servers) in order to fulfill its own obligations. This is defined in the realization model of the testing component. It represents the expectation of the testing component toward its environment. The tests are not defined by the specification of the associated server - in this case it would only be a unit test of the server.

Example 1

Every testing component that acquires another server component will own or acquire a built-in tester component for that server. Figure 19 displays a component *Bank* that acquires a remote *Converter* component for currency conversions. The *Bank* component owns a built-in *ConverterTester* component that is able to check the compliance of the acquired converter to the *Bank's* clientship contract. The *ConverterTester* represents the expectation of the *Bank* toward its server in form of individual samples, the test cases. These test cases are designed according to test criteria that have been defined in the quality assurance plan. Chapter 2 has briefly described a quality assurance plan, and Chapter 3 has outlined feasible test case selection techniques that may be applied in a project.

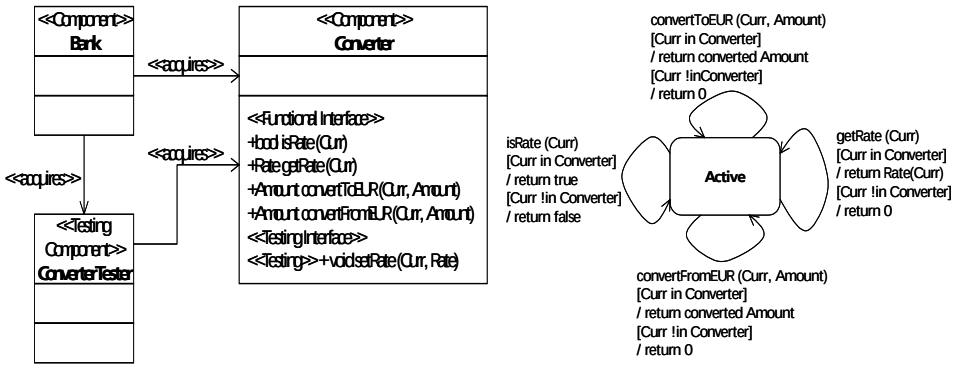
The specification of the tester component is performed in the same way as for any other functional component by simply following the steps that the development process suggests. In this case, this would be simply following the KobrA approach for designing a new component. The test cases may be realized in a tabular form and later on implemented as code and integrated in the component body. Figure 25 shows the realization structural and behavioural models of the *Bank* component. The realization defines what the Bank component is made out of. In this case, it is a *Converter* component that the *Bank* acquires with a number of features that the *Bank* is expecting to be supported with. The operation signatures are displayed in the realization structural model, and their semiformal specification are contained in the realization behavioural model. An additional requirement of the Bank is the testing interface operation *<<Testing>> setRate (Curr, Rate)*. This is required for the Bank's Converter tester component.

The test cases that will go into the *Converter* tester may be summarized in the following Table (Table 8). They are derived according to the Bank's realization models. The tests are derived according to coverage of the realization behavioural model that corresponds to full functional coverage. The states of the *Converter* are not essential for the *Bank* component since it cannot add or remove any entries. It can only change existing entries.

Table 8: Specification of a test suite for Bank's Converter Tester Component.

#	Initial State	Precondition	Event	Postcondition	FinalState
1			setRate (Curr, Rate)	Curr in Converter	
		Curr in Converter	isRate (Curr)	true returned	
2		Curr !in Converter	isRate (Curr)	false returned	
3			setRate (Curr, Rate)	Curr in Converter	
		Curr in Converter	getRate (Curr)	Rate returned	
4		Curr !in Converter	getRate (Curr)	0 returned	
5			setRate (Curr, Rate)	Curr in Converter	
		Curr in Converter	convertToEUR (Curr, Amnt)	Converted Amnt returned	
6		Curr !in Converter	convertToEUR (Curr, Amnt)	0 returned	
7			setRate (Curr, Converter)	Curr in Converter	
			convertFromEUR (Curr, Amnt)	Converted Amnt returned	
8		Curr !in Converter	convertFromEUR (Curr, Amnt)	0 returned	

Figure 25: Realization structural and behavioural models for the *Bank* component.



Example 2

The tests that the *Client* of *RankComponent* performs are derived according to the realization structural and behavioural models of the *Client*. This represents the *Client's* view on the functionality and behaviour of *RankComponent*. Figure 26 shows the realization structural model of its *Client*.

Table 9: Specification of a test case for RankComponent according to the RankComponent's behavioural model.

#	Initial State	Precondition	Event	Postcondition	FinalState
1	DataCollectionUnset	InvalidData	setDataTransferArea ( )	ERR_OUT_MIN_CASES	DataCollectionUnset
	DataCollectionUnset	InvalidData	setDataTransferArea ( )	ERR_OUT_MAX_CASES	DataCollectionUnset
	DataCollectionUnset	InvalidData	setDataTransferArea ( )	ERR_OUT_MIN_ATTRIBUTES	DataCollectionUnset
	DataCollectionUnset	InvalidData	setDataTransferArea ( )	ERR_OUT_MAX_ATTRIBUTES	DataCollecitonUnset
	DataCollectionUnset	ValidData	setDataTransferArea ( )	ERR_SUCCESS	DataCollectionSet
	DataCollectionSet	ValidData	setDataTransferArea	ERR_DTA_ALREADY_EXISTS	DataCollectionSet
	DataCollectionSet		destroydataTransferArea		

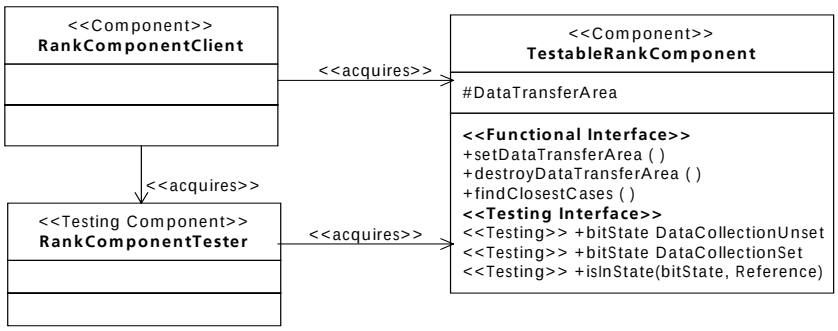


Figure 26: Realization structural model for the *RankComponent*.

5.6 Realization of the Tester Components - Step 6

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 sub-components in the same way as realizations of normal functional components define sub-components whose functionality they will acquire.

Example 1

The realization of the *Bank's Converter* tester component comprises a number of models that represent the test suites (e.g. activity diagrams). However, testing functionality is often so straightforward that we won't probably need any models. The realization may therefore simply be an implementation of the test plan in code or in a test notation such as the *Testing and Test Control Notation* [TTCN-3] that may be used as an input to automatic code generators. The test plan is represented by a table or a collection of tables according to the different testing criteria in a project (e.g. Table 8). The following Java code excerpt shows an partial implementation of the test plan in Table 8:

```
class ConverterTester {
  private Object converter = null;
  private boolean pass = true;

  public setConverter (Object C) {
    converter = C;
  }
  public boolean performTest ( ) {
    test_1 ();
  }

  private void test_1 ( ) {
    converter.setRate (''USD'', 0.9956);
    if (!converter.isRate (''USD'')
      pass = false;
  }
```

```
    private void test_2 ( ) {
       ...
    }
    ...
}
```

Example 2

Figure 27 shows a Java source code example with for an implementation of the tester component for *RankComponent*.

Figure 27: Excerpt of a Java implementation.

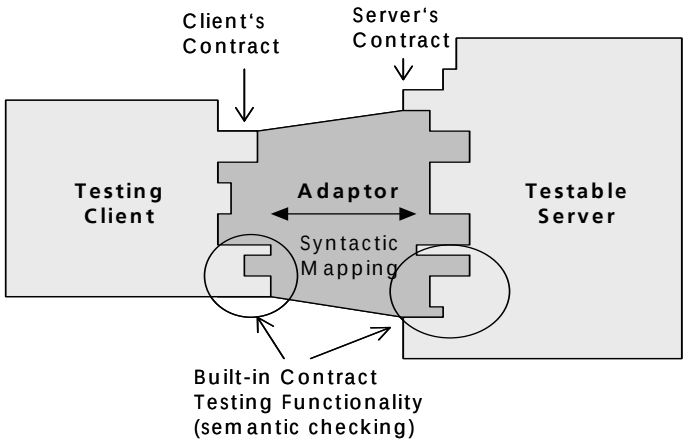
```
...
public class TesterRankComponent {
    public boolean executeTest (RankComponent rc) {


// Test Case 1. Parameters in the range but DataTranferArea exists
        /* Handle the Precondition*/
        if (isDataColllectionUnSet())
            rc.setToDataCollectionSet();
            // Handle the Event
            int res1 = rc.setDataTranferArea(20, 20);
            // Handle the Postcondition: expect ERR_DTA_ALREADY_CREATED
            if (res1 = RankComponentException.ERR_DTA_ALREADY_CREATED && rc.isDataCollectionSet()) {
                System.err.println ("1: final state: OK");
            else {
                System.err.println ("1: final state: FALSE");
            }
        } else System.out.println("Test failed");

//Test Case 2: .....
...

//Test Case 3: .....
...

// Test Case 6. Parameters in the range and DataTransfer not set
        // Handle the precondition
        if (isDataCollectionSet())
            rc.setToDataCollectionUnSet();
            //Handle the event
            int res6 = rc. setDataTranferArea (20, 20);
            // expect ERR_DTA_ALREADY_CREATED
            if (res6 = RankComponentException.ERR_SUCCESS && rc.isDataCollectionSet()) {
                System.err.println ("2: final state: true");
            else {
                System.err.println ("2: final state: FALSE");
        } else  System.out.println("Test failed");
    }
}
```

5.7 Integration of the Components - Step 7

Once all the functional component artefacts and the built-in contract testing component artefacts 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 client or the server, or an adaptor is designed and implemented that realizes the mapping between the two roles. Since the testing artefacts are integral parts of the individual components on either sides of the contract they are not subject to any special treatment, they

are just as any normal functionality. Figure 28 illustrates this. Here, client and server have different expected and provided interfaces, so that they must be mapped through an adaptor. The adaptor takes the operation calls from the client and transforms them to into a format that the server can understand. If the server produces results, the adaptor takes these and translates them back into the format of the client. Since the built-in contract testing artefacts are part of the client's and server's contracts they will be mapped through the adaptor as well. Component platforms such as *CORBA Components* do already provide support for this type of mapping.

Figure 28: Component integration through an adaptor.



Sequence of Integration

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 29 displays two typical scenarios. The second one is the traditional top-down testing approach that requires stubs to replace required functionality that is not yet available for a test. Ideally, component integration is performed bottom-up. This means that all required sub-components are available, so that the contract between component B and C in Figure 29 can be verified before the contract between component A and B. This follows the fundamental principle of application development that is in its purest form performed in a bottom-up fashion.

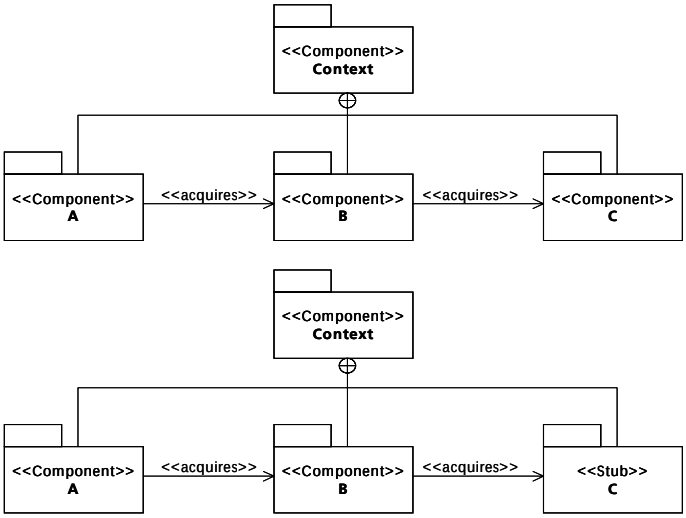


Figure 29: Integration sequence that supports built-in contract testing.

6 Test-Suite Use and Reuse

Testing takes a big share of the total effort in the development of big/complex software. However, component-based software engineering has mainly been focused on cutting development time by reusing functional code. If the components cannot be applied without extensive rework or retesting in the target domains, the time saving becomes questionable [Gui89]. Hence, there is a need to reuse not only functional code but also the tests and test environments that can be used to verify that the components work on the target platform or their target application. To get an 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 (BIT) mechanisms.
- Standardized test interfaces.
- Availability of test cases.
- The possibility to customise the tests according to the target domain.

The contract testing approach to built-in testing includes a flexible architecture that focuses on these aspects. It is the application of this architecture that makes reuse possible. In the initial approach of built-in testing as proposed by Wang et. al. [Wan00], 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.

Under the contract testing paradigm test cases are separated from their respective components and put in separate tester components. The BIT components still have some built-in test mechanisms, but only to increase their accessibility for testing. The actual testing is done by the tester components that are connected to the BIT components through their BIT interfaces. In the developed architecture, an arbitrary number of tester components can be connected to an arbitrary number of BIT components. This offers a much more flexible way to reuse tests as the tests used do not have to be identical to the ones originally delivered with the BIT component. The tests can be customised to fit the context of the component at all stages in the component's life cycle. BIT components have built-in mechanisms that increase the testability of the components. These mechanisms can be for example 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 standardised BIT interface, and are automatically reused with the component.

The overall concept of test reuse in built-in contract testing follows the fundamental reuse principles of all object and component technologies. Because testing is inherently built into an application or parts thereof (the components) testing will be reused whenever functionality is reused. In fact testing in this respect is normal functionality. Only the time when this functionality is executed distinguishes it from the other non-testing functionality, that is at configuration or deployment time.

6.1 Test Reuse at Development Time

Component-based development separates two development activities. The first one takes a top-down view on the development, this is when specified system functionality is distributed over individual components. This is also referred to as component engineering. The components are the building blocks of component-based systems, and so they are the primary units of reuse. Therefore, component engineering is termed development for reuse. The second one takes a bottom up perspective on development, this is when the individual reusable building blocks are assembled and put together in order to create a meaningful configuration of a system. This is also referred to as application engineering and it performs the development with reuse.

Components that are eventually reused in a new application abide by the same basic object and component principles. In other words, what an application is on a higher level of abstraction that is a component on a lower level of abstraction: an assembly of other sub-components. This is quite natural for components: somebody's component is somebody else's system. The only distinguishing feature between a component or an application/system is its reusing context, or in other words, the arbitrary definition of a unit of composition that a user of a component determines. Lower-level component assemblies that are made up of other sub-components may be supported by the same built-in contract testing techniques and architectures as application level components. Only the question of whether a component will be a stand alone reusable entity in a repository, or be subsumed in an assembly of other components determines whether it makes sense to have the contract testing artefacts built-in permanently.

Test reuse through built-in contract testing may therefore be applied at all levels of development whenever larger-grained units are integrated out of finer-grained units. After successful integration the built-in testing artefacts may be removed.

6.2 Test Reuse at Configuration and Deployment Time

In the typical scenario for component-based software engineering, a component can be bought off-the-shelf or taken from an in-house repository and inserted into a new application. In our vision, these components should all be components that support built-in contract testing. This means:

- They should have well-defined test interfaces that make them testable. These can be developed by the component producer according to the BIT methodology and process described in the previous chapters of this document.
- They should also provide and publish their own tester components. These are the tester components for
  - checking the component itself, and views the component as the server component.
  - checking the servers of that component. These are the tester components that validate the environment.

The built-in contract testing artefacts that should be provided with a such a component are illustrated in Figure 30.

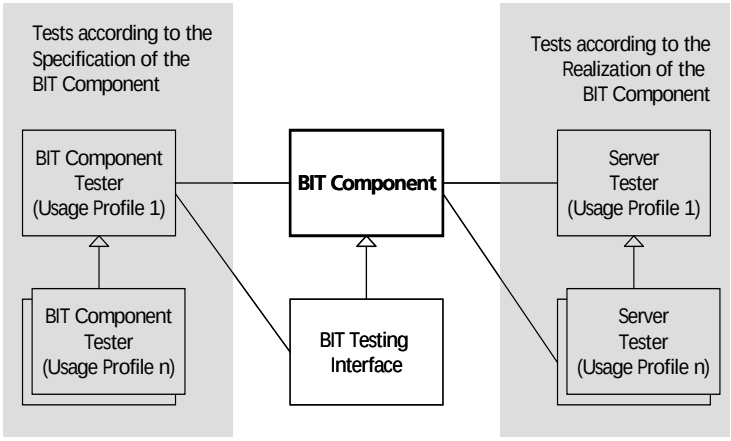


Figure 30: BIT interface and tester components that should be provided with any BIT component.

6.2.1 BIT Component Testers

The BIT component testers on the left hand side of the BIT component in Figure 30 represent the testing that the vendor or producer of the BIT component has already developed and performed. These are source code components that comprise the vendor's original test cases for checking this particular BIT component (its provided interfaces). Each of these tester components represents an expected usage profile of the BIT component that the vendor of that BIT component had imagined for their product. This in fact is a test suite for unit testing

because it is developed according to the specification of the unit and not according to the specification of the user of that unit.

Reusing and reexecuting these tests might initially seem redundant since the vendor of the BIT component will have already applied these tests in their own development environment. However, the operation environment (the expected interfaces) of the customer of that BIT component is likely to be different from that of the vendor. Reusing and reapplying the existing test components validates the functionality of the BIT component within its new context. The application of these BIT component testers will cause the BIT component to use its own server components upon which it depends, thus validating the contracts between the BIT component and its servers. In other words, the application of the BIT component test suites validates the contract compliance of the BIT component's associated servers. In this case, the combination of the BIT component plus its server environment will be validated.

6.2.2 Server Tester Components

The server tester components on the right hand side of the BIT component in Figure 30 represent the testing that the vendor or producer of the BIT component has already developed and performed to check the BIT component's development-time environment. These are source code components that comprise the vendor's original test cases for checking the servers of this particular BIT component. Each of these tester components represents a feasible usage profile of the BIT component's environment upon which itself depends. In other words, these server tester components represent the different types of environments for which the BIT component has been developed originally.

Reusing and reexecuting these tests will be performed whenever the server environment of the BIT component is changed, that is if the BIT component is moved into a new run-time environment, or if it is integrated into a new application.

6.2.3 Built-in Documentation through the Provided Tester Components

The tester components for the provided and required interfaces determine how the BIT component vendor expects its component to be used, and how the BIT component vendor expects the BIT component to use its own environment. Thus, the tester components represent the vendor's interpretation of the BIT component's specification and realization documents. This realizes built-in documentation of the BIT 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 post-conditions.

Tester components additionally illustrate for which different usage profiles the component is suitable. This facilitates the decision of whether the component can be integrated into the intended scope of the application or not. For this purpose, the tester components can be compared with the tester components that the user of the BIT component has developed in order to see whether the user's expectations map to what the component provides.

6.3 Test Reuse at Operation Time

In theory, built-in contract testing may also be applied at operation time when an application is dynamically changed. However, this may not initially be defined as operation time, since the application will be out of service for the duration of such a dynamic update. This process may rather be defined as a re-configuration and re-deployment of an application, which is the typical scenario for using and applying built-in contract testing.

6.4 Test Reuse throughout Maintenance

If the individual components that make up an application are in-house developments, the built-in contract tester components may also be used for regression testing of the individual parts. Since each tester component represents a valid usage profile of the associated component, the sum of all tester components from all different dedications of the component may be re-executed when a particular unit is amended. In other words, built-in contract testing also provides a vehicle to perform specification-based development-time regression testing.

6.5 Reuse of Standardised Tester Components

The full vision of component-based development can only become a reality if components implement standard interfaces for standard functionality. This adopts the typical mechanical engineering view on physical components to the software domain. It means that technical gadgets are built out of standard parts that are made according to standardised specifications. For software engineering this corresponds to building software components according to well defined, and standardised functional specifications. Each of these specifications is generic to all feasible implementations of such a specification. This means that a generic tester component for a generic specification can be used to test each specific component implementation. A component market place would therefore not only have implementations of components on offer but also their generic specifications and generic tester components for these specifications. Component vendors may therefore not only sell functionality in form of components but also testability in form of tester components for distinct specifications. These generic tester components will be plugged into component assemblies

and reused in the same way as functional components are currently reused in application engineering. For contract testing this means that a client which acquires a standard component, executes its standardised in-built tester component in order verify that the new server is an acceptable implementation of that standard interface.

However, this vision can currently only be realized on an organizational level due to the lack of suitable component market places. An organization that is developing under the component paradigm will have internal repositories of standard components for their particular domain, from which the development departments will draw their reusable assets. These repositories may be realized as simple source or binary code libraries, or if more advanced, component repositories with underlying middleware platforms that may be accessed remotely through the organizational intranet. Such reusable assets will typically comprise multiple realizations, different versions, for different purposes, or different product lines that are implemented according to standard interfaces. In other words, for each of their domain-specific standard interfaces they will have a number of feasible implementations of that interface, or versions, that abide by the specified contract, and additionally realize a variety of different non-functional requirements. Such requirements may define size and performance properties of the code, or conformance to different quality and safety standards, or they may simply define specific run-time environments or hardware configurations under which a version may be executed. All these different versions of one single interface specification are expected to provide identical functionality and behaviour. The different versions are only used according to the different required non-functional specifications. Therefore, there will only be one single tester component for each of these standard interfaces.

7 Configuration Management and Interfaces

Configuration management activities are responsible for how changes in the application are achieved. Configuration interfaces are one way to realize changes, and they play an important role in the component-based development paradigm. They represent additional functionality that is used to configure a server component according to the particular needs of a distinct client, or a client to use a distinct server. Each client uses a component according to a particular usage profile. This is equivalent with a view on a component's functionality, and this typically appears in systems with high functional variability, for instance in product line development. Each particular product line will exhibit different behaviour at certain points of variability of the overall system, and a component must be able to accommodate this variability through some means of configuration. Built in contract testing discriminates two different types of configuration interfaces:

- Existing functional configuration interfaces that are typical for product line development, for example.
- Configuration interfaces that set up the type and thoroughness of the required built-in testing.

7.1 Functional Configuration

The first type of configuration is concerned with the effect that existing functional configuration interfaces might have on any BIT artefact, or vice versa. Such configuration interfaces are concerned with existing variability in functionality that is controlled through configuration interfaces. These are readily defined in the functional specification of a product line, and they implement normal functionality of the component as far as testing is concerned. Two scenarios are conceivable:

- The server is configureable, and a client acquires an instance of a configured server component. This means that the creator of the component (this is not the client) has already configured its functionality according to the needs of the acquiring client, and the client "knows" nothing about the different views that this server may provide to other clients. In this case, the client receives a particular instance of a component that provides it with a particular service. Before using this acquired server, the client may execute its own built-in tests that verify the semantic conformance of the server to its clientship contract. In this case, there are no implications of the server's configureability that affect built-in contract testing.

– The client is configureable, and it may be able to acquire a range of different types of servers according to the intended functionality of the overall application. In this case, the configuration interface of the client will have to provide some means of choosing the type of server to be acquired from a list of feasible types of servers for this particular point of variation. In this case, the configuration interface must also provide some means to set the tester for a particular type of server. Though, this is done in the same way as for configuring the functional variability. This is performed according to choosing the test weight in contract testing. The set up is displayed in Figure 31.

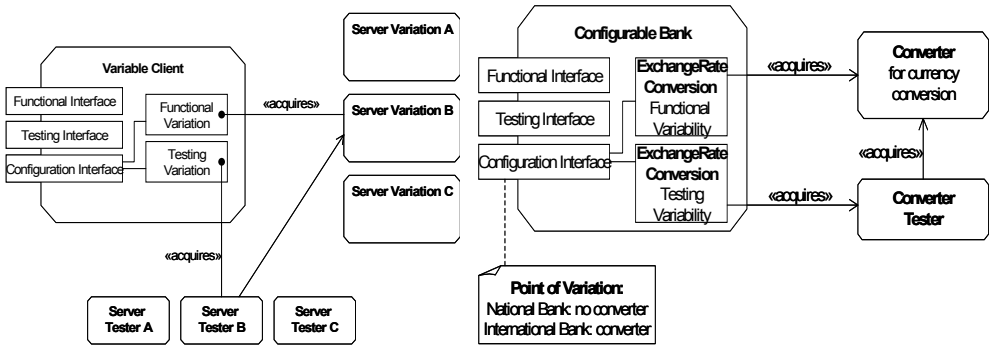


Figure 31: Configureable client, for example a national and international Bank.

7.2 Test Configuration

This is concerned with how different testing requirements of the BIT technology are configured. Configuration of a BIT-component can be considered from several angles; the impact of functional configuration (at compile or deployment time), the impact of configuration on test service availability, and run-time configuration of test services. It is important to recognize that the configuration of the functional aspects of the component has little impact on the testing functionality.

Testing configuration interfaces are specific to the particular type of contract testing that will have to be performed according to the intended use of a component. With respect to the type and required thoroughness of test, any available tester component may be included in a client either statically, or dynamically implemented through references. The previous section has considered different types of testers that may be set according to different functionality at points of variation in an application. This is one way to introduce configuration interfaces for testing. The second one concentrates on the aspect of thoroughness of performed test. This is governed by criteria such as the time of the test, the origin of a component or the availability of resources, for example in an embedded system. In this respect, the architecture of built-in contract testing is completely open to configuration since different types and degrees of testing can be easily introduced.

8 Built-in Contract Testing and Commercial Off-the-shelf Components (COTS)

The integration of commercially available third party components into own applications represents one of the main driving factors for component-based software development since it greatly reduces own development efforts. Such components are aimed at solving typical problems in distinct domains, and ideally they can be purchased off-the-shelf and simply plugged into an application. However, this ideal scenario seems far from reality. Although they reduce development effort for functionality, third party components do typically greatly increase the integration effort of the overall application, since they are normally only available in a shape that permits or at least inhibits their easy integration. For example, they may provide syntactically or semantically different interfaces from what is expected and required, or they may not be entirely fit for the intended purpose. In any way, the usage and integration of third party components typically requires either the development of wrappers or adaptor components that hide and compensate these differences, or changes in the design and implementation of the integrating client component. This underlines the need for component markets that are based on and driven by the provision of standard interfaces reflecting standard solutions to typical domain problems.

If our own component can communicate syntactically with a provided COTS component through some mechanism, the next step is to make sure that they can also communicate semantically. In other words, the fact that two components are capable of functioning together says nothing about the correctness of that interaction. This is where built-in contract testing provides its valuable services, and this is where it exhibits its greatest benefits.

8.1 COTS Components with BIT Capability

Ideally, all commercially available components should provide testability measures such as the introspection mechanism that is realized through built-in contract testing interfaces. Such components will naturally fit into applications that are driven by the built-in contract testing paradigm. They simply need to be interconnected syntactically, and this of course comprises the functionality as well as the testing aspects of the two components. The built-in contract testing paradigm may be seen as a strong advocate of not only providing testing interfaces with commercial components but also providing these according to well-defined interfaces, so that the syntactic integration effort may eventually be wiped out completely. However, since the technology has not yet penetrated into the component industry, it is likely that component vendors will not provide

their components with such testability measures. Though, this is not too much a difficulty as explained in the following sub-section.

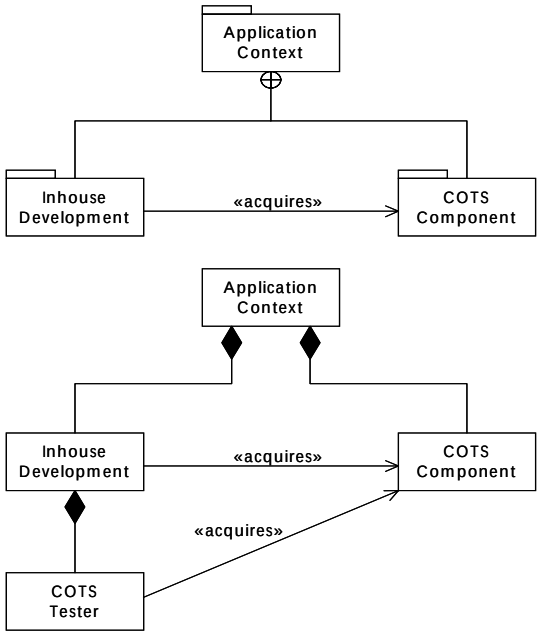


Figure 32: In-house development that acquires a COTS component without testing interface.

8.2 COTS Components without BIT Capability

If we bought a COTS component today we will typically expect it not to expose any testability interfaces, or its own server testers. In other words, we get a component that only provides its specified service. This is a unit with reduced controllability and observeability compared with the full controllability and observeability that components built-in contract testing paradigm expose through their testing interfaces. For the client of such a component it means it cannot apply a server tester that uses this testing interface, so it cannot set or get any internal information of the component except through the normal interface. In fact, this is the situation that testers have been facing right from the very advent of object technology. They can only use the exposed operations of a component in order to validate it, nothing more. Although the testability of such components is decreased it does not change the validity of the built-in contract testing approach. Figure 32 illustrates this scenario. It displays an in-house component that is equipped with a built-in server tester component (through the context) that checks the server's contract compliance. The server does not exhibit any testing interfaces, so its testability is limited.

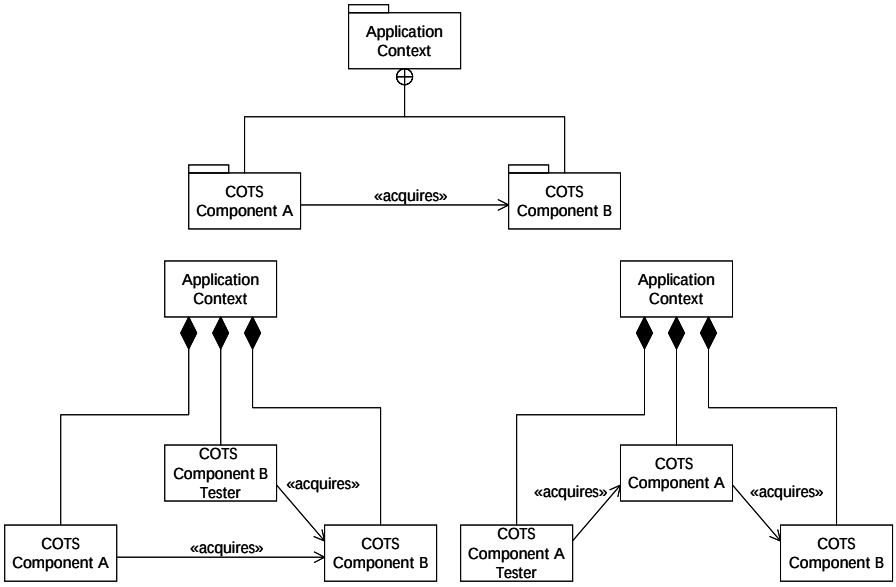


Figure 33: COTS component that acquires another COTS component.

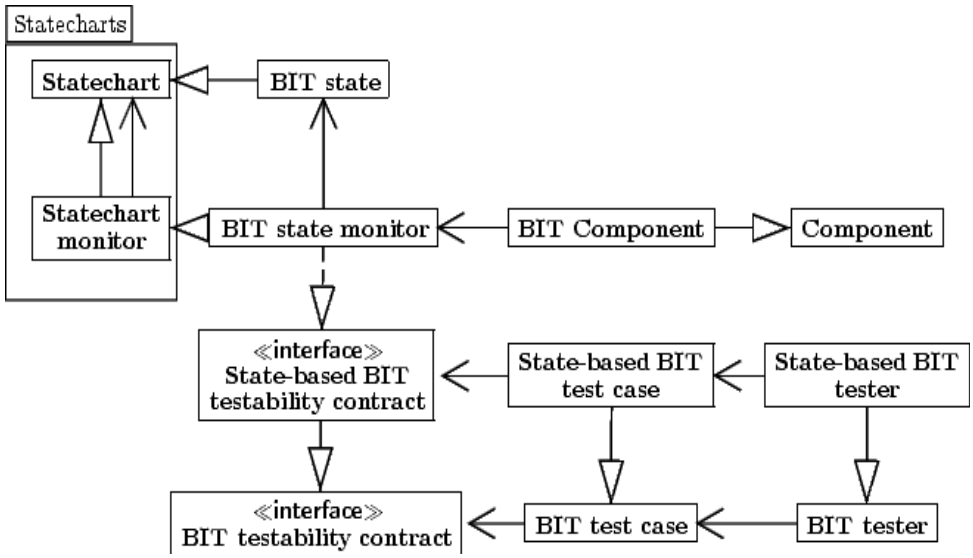
The same principle applies when two COTS components without testing interfaces are integrated. This is illustrated in Figure 33. COTS component A acquires and uses COTS component B as server. Initially, A cannot test its server B. However, the context, this is the glue code that combines the two COTS components, can provide a tester component that is developed according to A's required interface specification. Before the context lets the client component use its server it can pass it to the tester component that checks it for the client. In this case, any arbitrary scenario is feasible where the context provides tester components for the COTS components. In this case, the testing organization depends entirely on the capability of the context.

8.3 Extended COTS Components with Added Built-in Contract Testing Capability

Commercial third party components cannot typically be augmented with an additional built-in contract testing interface that provides a client with an introspection mechanism for improved testability and observeability. This leaves COTS components only to be tested through their provided functional interface, as it is traditionally the case in object and component testing. However, modern object languages or component platforms such as Java do provide mechanisms that enable internal access to a component. They can break the encapsulation boundary of binary components in a controlled way and offer internal access. Such mechanisms can be used to realize testing interfaces according to the built-in contract testing philosophy for any arbitrary third party component. How this is achieved exemplary with the Java platform as implementation technology by using a suitable Java Library that is based on Java's own reflection

mechanism is subject of the following paragraphs. The architecture of the library is displayed in Figure 34 [BBB03]. Built-in contract testing can initially be carried out by using three primary concepts. These are the testability contract, the tester and test case. These are the fundamental features that support the assessment of test results, control of the execution environment, and actions to be taken if faults are encountered. Additionally, the library provides state based testing support that is more essential to built-in contract testing. These concepts are the state-based testability contract, the state-based tester, and the state-based test case. The state-based concepts abide by the principles of Harel's state machines [BBB03].

Figure 34: Organization of the built-in contract testing support library [BBB03].



The Java library is bounded to the original Java COTS component either through an extension mechanism (inheritance) or through a containment relationship. The second case realizes a new component that owns the original COTS component as a sub-component (i.e. as an attribute). The new component acts as a wrapper around the original COTS component. The access to the COTS component is realized through Java's Reflection mechanism inside the library. The behavioural model that is required for the contract testing interface must be defined according to the generic behavioural facilities that the library is providing and it is completely incorporated in the newly created wrapper. Through this technique, the wrapper component represents an executable behavioural model of the original COTS component. This requires that the COTS component is properly documented. The wrapper component therefore provides a built-in contract testing interface through overwriting the isInState and setToState operations that are implemented through the library. This technique is detailed and explained on the basis of a case study in [BBB03]. It must be noted that this technique heavily depends upon the capability of the implementation technology.

9 Built-in Contract Testing and Web-Services

Web-Services 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. Web-Services fulfill all the requirements of Szyperski's component definition [Szy99], that is a service is only described and used based on interface descriptions and more importantly, it is independently deployable. This means, a Web-Service provides its own run-time 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 run-time 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. Initially, this is not different from the way we have treated components so far, for instance objects. However, here the components of an application are not bound to a particular host such as in object-oriented programming, but they are established dynamically throughout entire networks, for example the Internet. In this way, different implementations of a distinct service may be easily replaced through registering with a different Web-Service that provides the same specification of the required component. 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 behavioural specification).
- Component. This represents a readily usable and deployable object that provides functionality and exhibits behaviour. 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 run-time specific elements of a Web-Service. It means they are responsible for establishing the connection between client and server, take care of the execution of an instance, or control its security.

9.1 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. In fact this is the scenario for which built-in contract testing provides the most benefits. The syntactic compatibility between client and a Web-based server is ensured through the XML mapping between their interfaces. Their semantic compatibility is checked through the built-in server tester components inside the client that are executed to validate the associated server. These tests are performed when the client is registering with a service for the first time, this means during configuration, or if the client requests the same specification of the server from a different Web-Service provider, this means during re-configuration of the system.

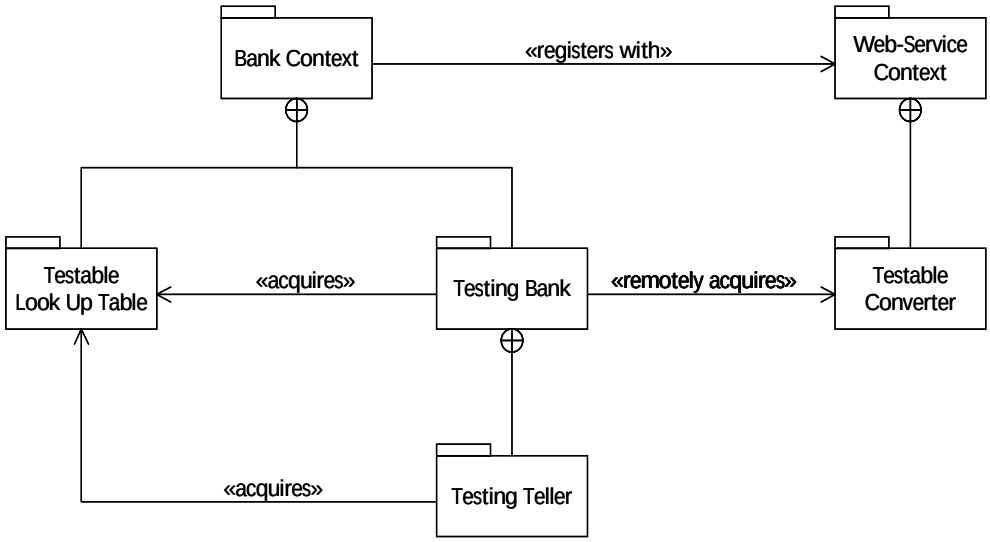


Figure 35: Example containment hierarchy of a simple international banking system on the basis of a Web-Service.

Example 1

Figure 35 displays the containment hierarchy of a simple international banking system that is based on a Web-Service. The converter component contains all currency exchange rates and exports the conversion operations. It may be provided by a company (Web-Service provider) that is specialized on 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, so that the latest currency exchange rates are always available on-line to the banking application. The <<remotely acquires>>-relationship indicates that the converter (in this case it is a testable converter) is not locally available. It means that this relationship will be implemented through some underlying networking infrastructure. This is realized through the Connector and the Container on the server side (the Web-Service) and a Web-Service conformant implementation on the client side of the <<remotely acquires>>-relationship.

The stereotype <<remotely acquires>> hides the underlying complexity of the network implementation and only considers the level of abstraction that is important for testing. This technique is termed “stratification” [Atk02]. In other words, 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 may provide a suitable test 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 Bank component registers with the context of the Converter component (Figure 35).

Example 2

Figure 36 displays a containment hierarchy of a web-service-based video-conference system that represents the core functionality of a health-care surveillance application. It shows a number of components, including one for the camera control at the *CentralSystem* and one for the actual video camera at the *RemoteSystem*.

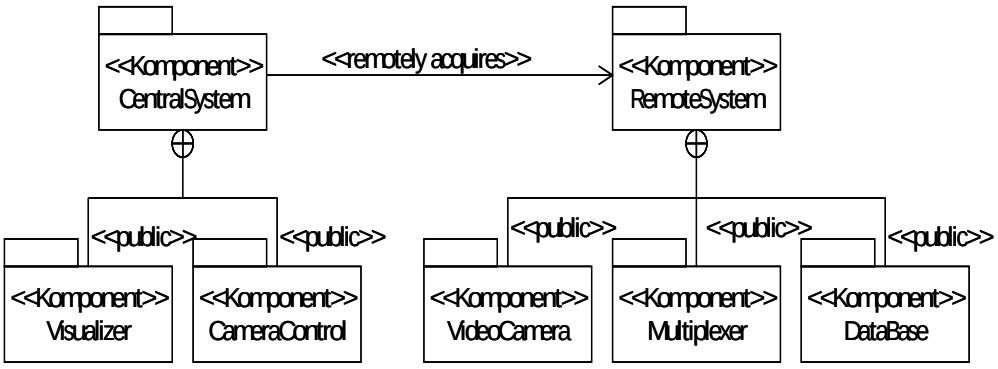


Figure 36: Development-time containment hierarchy of a web-based video-conferencing system from a healthcare project.

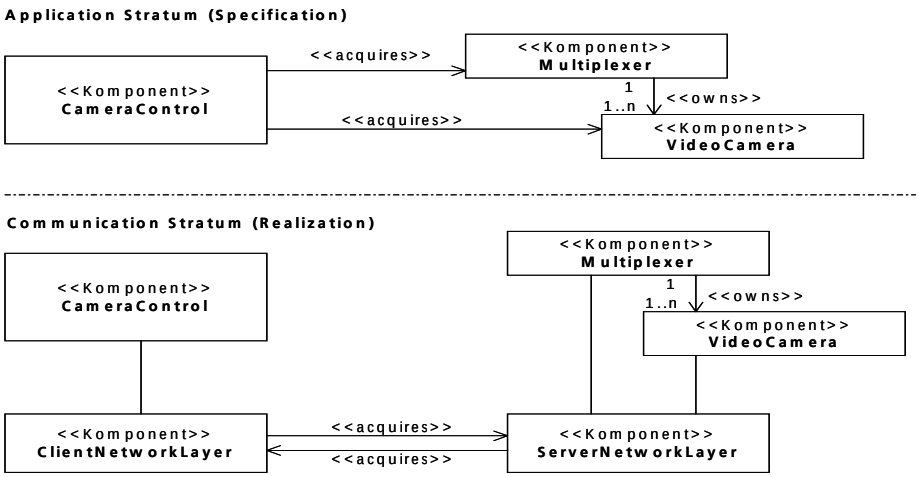


Figure 37: Specification and Realization of a Web-Service in two Strata.

Testing in this example should consider different system views at different levels of abstraction (so-called strata). This is indicated in Figure 37. On an application level, we are only interested whether the camera control component on the left side can control the camera on the right side correctly. And we can test that through appropriate test cases in the camera control component. However, failures may also be rooted in the underlying network infrastructure. This is similar to the applications run-time system, and it may therefore be tested as well. The camera control component may therefore be augmented with a tester component for checking the underlying network infrastructure plus one for the application-level camera control component. The testing sequence can then start by executing the network related tests, and then, if these are correct, perform the application-level tests that we are actually interested in. The decision on whether we will include a network tester depends upon the trust that our component has in the underlying infrastructure, and on how loosely the two components are interconnected. If the underlying infrastructure is likely to change, it might well make sense to have a permanently built-in network tester in the camera control component. The same is true for the camera component. If it never changes it probably does not make much sense to have a camera tester built-in permanently. Such a tester is detailed in the following paragraph.

Figure 38: Structure of the camera control system with a *CameraTester* Component.

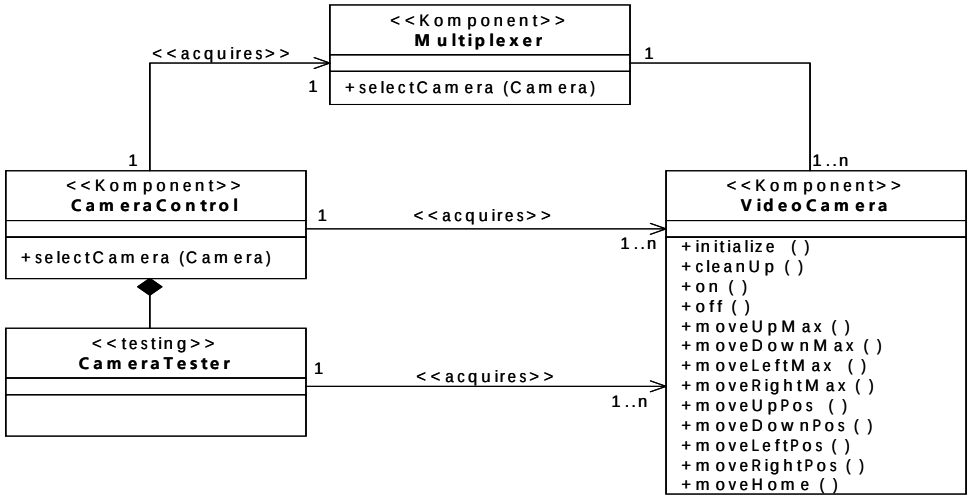


Figure 38 displays the structural view on the camera control functionality of the system, and Figure 39 and Figure 40 depict the behavioural views on this system part, with Figure 40 refining the ON-state in Figure 39. The *CameraTester* component will be developed according to the behavioural models. Every state-transition maps to one or more test cases for the *CameraTester*. If the camera control unit acquires a new camera it can simulate typical requests from that camera through executing *CameraTester* component.

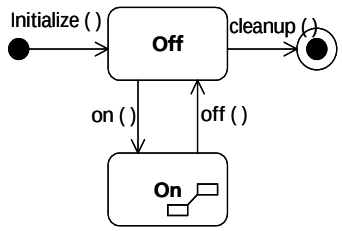


Figure 39: Behavioural model of the camera control functionality.

Figure 40: Refinement of the ON-state in Figure 39 in tabular form.

		Final State					
		Home	MaxUp	MaxDown	MaxRight	MaxLeft	OtherPos
Initial State	<<starting>> **Home**		moveMaxUp	moveMaxDown	moveMaxRight	moveMaxLeft	moveUpPos moveDownPos moveRightPos moveLeftPos
	MaxUp	moveHome		moveMaxDown	moveMaxRight	moveMaxLeft	moveDownPos moveRightPos moveLeftPos
	MaxDown	moveHome	moveMaxUp		moveMaxRight	moveMaxLeft	moveUpPos moveRightPos moveLeftPos
	MaxRight	moveHome	moveMaxUp	moveMaxDown		moveMaxLeft	moveUpPos moveDownPos moveLeftPos
	MaxLeft	moveHome	moveMaxUp	moveMaxDown	moveMaxRight		moveUpPos moveDownPos moveRightPos
	OtherPos	moveHome	moveMaxUp	moveMaxDown	moveMaxRight	moveMaxLeft	[pos<MaxUp] moveUpPos [pos<MaxDownPos] moveDownPos [pos<MaxRightPos] moveRightPos [pos<MaxLeftPos] moveLeftPos

9.2 Testing of Readily Initialized Server-Components

Web-Services are typically providing instances that are ready to use. It means that the server component that is provided through the Internet service is already configured and set to a distinct required state. A run-time 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 Converter component in Figure 35 may comprise test cases that change, add or remove some of the exchange rates stored in that component. Clearly, for the client, such a changed server is of no use and this creates a fundamental dilemma for built-in contract testing of Web-Services.

9.2.1 Possible Destruction of the Server through the Testing Client

Under object-oriented run-time systems the client can solve this dilemma by simply creating a clone of the tested component and passing the clone to the test

software. This works because client and server are handled by the same run-time environment. For example, in Java this is performed through the *Object.clone* method. In this case, the test software may completely mess up the newly created clone without any effect on the original instance, it is simply thrown away after the test, and the original is used as working server. However, in a Web-Service environment the run-time 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 run-time scopes on completely different network nodes. For example, The banking application in Figure 35 may be based on Java, and the Web-Service component may be based upon a Cobol run-time environment. In other words, only the Web-Service context may generate an instance of that Cobol component because it comprises a Cobol run-time environment.

Contract testing can therefore only be applied in a Web-Service context if the Web-Service provides somes way for the client to have a clone created and accessed for testing. Some contemporary component technologies such as CORBA Components are capable of doing exactly that. Here, 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.

9.2.2 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 up the run-time system. During the integration phase of an application this is not a problem because it is not used under real conditions. During a re-configuration of an operational system, however, this may 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 in order to rule out any side effects. For Web-Services this is not an issue since the individual components are executed in different threads on different nodes, anyway. So in this case, a contract is always safe, and it will never fail the client application that is applying it.

10 Summary

This report has described the methodology and process of built-in contract testing in model-driven component-based application construction. It is a technology that is based on building the test software directly into components, built-in tester components at the client side of a component interaction and built-in testing interfaces at its server side. In this way, every component may check whether it has been brought into a suitable environment, and it may be checked by the environment whether the component provides the right service. This enables system integrators who reuse such components to validate immediately and automatically whether a component is working correctly in the changed environment of an newly created application. The benefit of built-in contract testing consequently follows the same principles that reuse technologies offer. The effort of building test software into individual components is paid back depending on how often such a component will be reused in a new context, and a component is reused depending on how easily it may be reused. Built-in contract testing greatly simplifies the reuse of a component because once it has been integrated syntactically in its new environment its semantic compliance with the expectations of that new environment may be automatically assessed.

11 References

[Abd00] Abdurazik, A. and Offutt, J., "Using UML Collaboration Diagrams for Static Checking and Test Generation", 3rd Intl. Conf. on the Unified Modeling Language (UML'00), pp. 383-395, York, UK, October 2000.

[Atk01] Atkinson, C., et al. "Component-Based Product-Line Engineering with UML", Addison-Wesley, London, 2001.

[Atk02] Atkinson, C., Bunse, C. Groß, H.-G., Kühne, T. Towards a General Component Model for Web-based Applicaitons. Annals of Software Engineering, Vol. 13, 2002.

[AF98] Allen, P., Frost, F., Component-based Development for Enterprise Systems: Applying the Select Perspective, Cambridge University Press, 1998.

[BBB03] Barbier, F., Belloir, N., Bruel, J.M., Incorporation of Test Functionality in Software Components. To appear: 2nd Intl. Conference on COTS-based Software Systems, Ottawa, Canada, 10.-12. February 2003.

[Bei90] Beizer, B., Software Testing Techniques, Thompson Computer Press, 1990.

[Bei95] Beizer, B., Black-box Testing, Techniques for Functional Testing of Software and Systems, John Wiley & Sons, New York, 1995.

[Bin00] Binder, R., Testing Object-Oriented Systems - Models, Patterns, and Tools, Addison-Wesley, 2000.

[BKF99] Bayer, J., et al., PuLSE - A Methodology to Develop Software Product Lines, SSR'99, 1999.

[Boo87] Booch, G., Software Components with Ada: Structures, Tools and Subsystems, 1987.

[CD00] Cheesman, J., Daniels, J., UML Components, A Simple Process for Specifying Component-based Systems, Addison-Wesley, 2000.

[Chu99] Chung, S., et al., "Testing of Concurrent Programs Based on Mes-

sage Sequence Charts", In IEEE International Symposium on Software Engineering for Parallel and Distributed Systems, Los Angeles, CA, May 17 - 18, 1999.

[Cmp+01] Component+ Project Technical Report, "Built-in Testing for Component-based Development", http://www.component-plus.org

[Col94] Coleman, D., et al., Object-oriented Development. The Fusion Method, Prentice Hall, 1994.

[DW98] D'Souza, D.F., Willis, A.C., Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley, 1998.

[GHY97] Graham, L., Henderson-Sellers, B., Younessi, H., The OPEN Process Specification, Addison-Wesley, 1997.

[Gre01] McGregor, J., Sykes, D., A Practical Guide to Testing Object-Oriented Software, Addison-Wesley, 2001.

[Gui89] Guindi, D. S., Ligon, W. B., McCracken, W. M., Rugaber, S., "The impact of verification and validation of reusable components on software productivity", Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, pp. 1016-1024, 1989.

[IEEE99] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std-610.12-1990, 1999.

[JBR99] Jacobson, I., Booch, G., Rumbaugh, J., The Unified Software Development Process, Addison-Wesley, 1999.

[KCN90] Kang, K.C., Cohen, S.G., Novak, W.E., Petersen, E.S., Feature-oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI90-TR-21, Software Engineering Institute, November 1990.

[Kru00] Kruchten, P.B., The Rational Unified Process - An Introduction, Addison-Wesley, 2000.

[MDA] OMG's Model Driven Architecture, www.omg.org/mda.

[Mey97] Meyer, B., Object-oriented Software Construction, Prentice Hall, 1997.

[Mey98] Meyer, S. and Sandfoss, R., "Applying Use-Case Methodology to SRE and System Testing", STAR West Conference, October 1998.

[MLH87] Mills, H.D., Linger, R.C., Hevner, R.A., Box-structured Information Systems, IBM Systems Journal, 26(4), 1987.

[OCL00] Object Management Group, OCL Specification Documentation, www.omg.org/ocl.

[Off99] Offutt, J. and Abdurazik, A., "Generating Tests from UML specifications", 2nd Intl. Conf. on the Unified Modeling Language (UML99), pp. 416-429, Fort Collins, CO, October 1999.

[Quasar] German national funded Quasar Project, http://www.first.gmd.de/quasar/.

[RBP91] Rumbaugh, J., et al., Object-oriented Modeling and Design, Prentice Hall, 1991.

[Rob92] Robinson, P.J., "Hierarchical Object-oriented Design", Prentice Hall, 1992.

[Rob99] Robinson, H., "Finite State Model-Based Testing on a Shoestring", STAR West, October 1999.

[RWL96] Reenskaug, T., Wold, P., Lehne, O., Working with Objects: The OORam Software Development Method, Manning/Prentice Hall, 1996.

[Rys99] Ryser, J. and Glinz, M., "A Scenario-Based Approach to Validating and Testing Software Systems using Statecharts", 12th Intl. Conf. on Software and Systems Engineering and their Applications (ICSSEA'99), Paris, France, 1999.

[SGW94] Selic, B., Gullekson, G., Ward, P. Real-Time Object Oriented Modeling, John Wiley & Sons, 1994.

[Szy99] Szyperski, C., Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1999.

[TTCN-3] Testing and Test Control Notation 3, European Telecommunication Standards Institute (ETSI), www.etsi.org.

[UMLT] UML Testing Profile, http://www.fokus.gmd.de/u2tp.

[Veg00] Vegas, S.V., Title, PhD Thesis 2000.

[Wan00] Wang, Y., King, G., Fayad, M., Patel, D., Court, I., Staples, G., Ross, M., "On Built-in Test Reuse in Object-Oriented Framework Design",

ACM Journal on Computing Surveys, Vol. 32, No. 1, March 2000.

[WL99] Weiss, D.M., Lai, C.T.R., Software Product Line Engineering - A Family Based Software Engineering Process, Addison-Wesley, 1999.

Document Information

Title:	Component+ Methodology: Built-In Contract Testing Method and Process
Date:	October 31, 2002
Report:	IESE-030.02/E
Status:	Final
Distribution:	Public