
**Fraunhofer** Institut
Experimentelles
Software Engineering

# Testing and the UML

## A Perfect Fit.

**Author:**
Hans-Gerhard Gross

In part supported by
the German Federal Department of Education and Research under the MDTS Project Acronym.

IESE-Report No. 110.03/E
Version 1.0
October 31, 2003

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

Testing activities that are based on models are becoming increasingly popular. UML models represent specification documents which provide the ideal basis for deriving test cases. They are even more valuable if UML tools are used that support the automatic test case generation. This report presents a summary of model-based testing techniques and test modeling techniques. These are the two fundamental aspects of testing with the UML. The first is concerned with deriving test information out of UML models, whereas the second concentrates on how to model test behaviour with the UML.

**Keywords:** Model-based testing, test modeling, unified modeling language, test case generation, test criterion.

# Table of Contents

# 1 Introduction

Software testing is a widely used and accepted approach for verification and validation of a software system, and it can be regarded as the ultimate review of its specification, design and implementation. Testing is applied to generate modes of operation on the final product that show whether it is conforming to its original requirements specification, and to support the confidence in its safe and correct operation. Appropriate testing should always primarily be requirements and specification centered and not code based, which means that testing should always aim to show conformance or non-conformance of the final software product to some requirements or specification document. Structural code information provides a big deal of information in order to guide the testing efforts according to testing criteria, but it cannot replace specification documents as a basis for testing.

The Unified Modeling Language (UML) has received much attention from academic software engineering research and professional software development organizations. It has almost become a de-facto industry standard in recent years for the specification and the design of software systems, and it is readily supported by many commercial and open tools such as Rational's "Rose", Verimag's "Tau", or "VisualThought". The UML is a notation for specifying system artifacts including architechture, components and finer grained structural properties, functionality and behaviour of, and collaboration between entities, and of course, at a higher level of abstraction, usage of a system. The UML may therefore be used to model and specify a computer system completely and sufficiently in a graphic and textual form, and drive its realization. It provides all the concepts of lower level implementation notations.

The combination of both, modeling and testing, is represented by two orthogonal dimensions that we have to consider under the subject:

- *Model-Based Testing* which is the development of testing artifacts on the basis of UML models. In other words, the models provide the primary information for developing the test cases and test suites, and checking the final implementation of a system. This is briefly introduced and related to traditional testing in Chapter 2: Model-based vs. Traditional Testing and further elaborated in Chapter 3. .
- *Test Modeling* which is the development of the test artifacts with the UML. In other words, the development of test software is based upon the same fundamental principles as any other software development activitiy, since they are in fact software artifacts with special testing purpose. So, additionally to using the UML to derive testing artifacts and guide the test-

ing process, it can be applied to specify the structural and behavioural aspects of the testing software. This is further elaborated in Chapter 4, UML Testing Profile.

The subject that is described in this report is mainly driven by the discussions of the Testing Panel of the 5th International Conference on the Unified Modeling Language (UML 2002) in Dresden, Germany, that was initiated under the topic of whether the UML and testing may be a perfect fit. The outcome of the German national funded MDTS project suggests that they are a perfect fit as indicated through the title of this report, so the document can be seen as a strong advocator for model-based testing and test modeling with the UML. Furthermore, it can be regarded as an initial attempt to summarize ongoing work in form of a state-of-the-art report on model-based testing techniques and model-driven test development that is based on the UML.

# 2 Model-based vs. Traditional Testing

Testing that is based on the UML has many concepts in common with traditional code-based testing techniques as described by Beizer [Bei90], for instance. Source code can be seen as a concrete representation of a system, or parts thereof, and UML models are more abstract representations of the same system. More concrete representations contain more and more detailed information about the workings of a system. It can be compared with zooming in on the considered artifacts, generating a finer grained representation but gradually loosing the overview on the entire system. Less concrete representations contain less information about details but show more of the entire system. This can be compared with zooming out to a coarser grained level of representation making it easier to overview the entire system but loosing the details out of sight. The advantage of using model-based development techniques and the UML for development and testing is that a system may be represented entirely through one single notation over all levels of detail, that goes from very high level and abstract representations of the system showing only its main parts and most fundamental functions, down to the most concrete possible levels of abstraction similar and very close to source code representations. It means that in a development project we are only concerned with removing the genericity in our descriptive documents without having to move between and ensure consistency among different notations. The same is true when testing is considered. Code-based testing is concerned with identifying test scenarios that satisfy given code coverage criteria, and exactly the same concepts can be applied to more abstract representations of that code, i.e. the UML models. In that respect we can certainly also have model coverage criteria for testing. In other words, more abstract representations of a system lead to more abstract test artifacts, and more concrete representations lead to more concrete test artifacts of that system. Therefore, in the same way in that we are removing the genericity of our representations in order to receive finer grained levels of detail and enventually our final source code representation of the system, in parallel we have to remove the genericity of the testing artifacts for that system and move progressively towards finer grained levels of testing detail. The testing profile that is the subject of Chapter 4 supports this parallel development effort of the testing artifacts.

## 2.1 Traditional White-Box Coverage Criteria and the UML

Coverage is an old and fundamental concept in software testing. Coverage criteria [Bei90] in testing are used, based on the assumption that only the execution of a faulty piece of code may exhibit the fault in terms of a malfunction or a

deviation from what is expected. If the faulty section is never executed in a test it is unlikely to be identified through testing, so program path testing techniques, for example, are amongst the oldest software testing and test case generation concepts [War64] in software development projects. This idea of coverage has led to quite a number of structural testing techniques over the years that are primarily based upon program flow-graphs [Bei90] such as branch coverage [Bei90], predicate coverage [Bei90], or definition-use-(DU)-path-coverage [Mar95], to name only a few. These traditional coverage criteria all have in common that they are based on documents (i.e. flow graphs, source code) very close to the implementation level. Traditionally, these coverage criteria are only applied at the unit level which sees the tested module as a white box for which its implementation is known and available to the tester. On a higher level, in an integration test, the individual modules are only treated as black boxes for which no internal knowledge is assumed. An integration test is traditionally typically performed on the outermost sub-system that incorporates all the individually tested units, so that we assume white-box knowledge of that outermost sub-component, but not of the integrated individual units. Traditional developments only separate between these two levels: white box test in unit testing, and black box test in integration testing. Additionally, there may be an acceptance test of the entire system driven by the highest-level requirements. More modern recursive and component-based development approaches do not advocate this strict separation since individual units may be regarded as sub-systems in their own right, i.e. components for which no internal knowledge is available, or integrating sub-systems, i.e. also components, for which internal knowledge may be readily available. Particularly in component-based developments where we cannot really strictly separate units from sub-systems both approaches may be readily applied in parallel according to whether only black-box information, e.g. external visible functionality and behaviour, or additionally white-box information, e.g. internal functionality and behaviour, are available.

Typical white-box strategies comprise statement coverage or node coverage on the lowest level of abstraction. In this instance, test cases may only be developed when the concrete implementation is available (i.e. for statement coverage), or if at least the implementing algorithm is known in form of a flow-chart (i.e. for node coverage). Statement coverage is typcially not feasible, or practical with the UML, unless we produce a model that directly maps to source code statements, but node coverage may be practical if it is based on a low-level UML activity diagram. Activity diagrams are very similiar to traditional flow-charts, although activity diagrams may also represent collaboration between entities (i.e. through so-called swimlanes). Other coverage criteria such as decision coverage, condition coverage, or path coverage, may also be applicable under the UML but it always depends on the type and level of information that we can extract from the model. Chapter 3 treats these items in much more detail for the individual UML diagram types.

## 2.2 Traditional Black-box Testing Techniques and the UML

Most functional test-case generation techniques are based upon domain analysis and partitioning. Domain analysis replaces or supplements the common heuristic method for checking 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 of the input space belong to the domain of interest [Bei95]. A domain may map to equivalent functionality or behaviour, for instance. Domain analysis is used for and sometimes also refferred to as partitioning testing, and most functional test case generation techniques are based on that. Equivalence partitioning, for example, is one technique out of this group that divides the set of all possible inputs into equivalence classes. This equivalence relation defines the properties for which input sets are belonging to the same partition. Traditionally, this technique is only concerned with input value domains but with the advent of object technology it can be extended to behavioural equivalence classes. UML behavioural models such as state charts for example, provide a good basis for such a behavioural equivalence analysis, i.e. test case design concentrates on differences or similarities in externally visible behaviour that is defined through the state model.

The following chapter looks at the individual UML diagrams, introduces their concepts and semantics, and discusses how they may be used in order to extract black-box as well as white-box testing information.

# 3 UML Diagram Types and Testing

The UML provides diagrams according to the different views that we can have on a system. These views can be separated into user view, architectural view, which may be further sub-divided into structural and behavioural view, implementation view, and environmental view. These views can be associated with the different diagram types of the UML. The user view is typically represented by use case diagrams, and the structural view by class and object diagrams. Sequence, collaboration, state chart and activity diagrams can be associated with the functional and behavioural views on a system, and component and deployment diagrams specify coarse-grained structure and organization of the system in its environment (deployment). In essence, UML diagrams specify what a system should do, how it should behave, and of course how it will be realized. The entirety of all UML models therefore specifies the system completely and sufficiently. The fundamental question here is which information can we extract from a UML model for driving the testing of the system, or which testing activities can we base upon a UML model. In the following we will look at the individual UML diagram types, discuss their features and semantics and how they can be used in order to derive testing information.

## 3.1 Usage Modeling with Use Case Diagrams

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

### 3.1.1 Concepts of Use Case Diagrams

Use case diagrams specify high-level user interactions with a system. This includes the users or actors as subjects of the system, and of course the objects of the system with which the users interact. Thus, use case models may be applied to define the coarsest-grained logical system modules. Use cases mainly concentrate on the interactions between the stakeholders of a system and the system at its boundaries. A use case diagram shows the actors of the system

(the stakeholders), either in form of real (human) user roles, or in form of other associated systems which are using the system under development as server. Additionally, use case diagrams show the actual use cases and the associations between the actors and the use cases. Each use case represents some abstract activity that the user of the system may perform and for which the system provides the support. Overall, use case modeling is applied at initial requirements engineering phases in the software life-cycle in order to specify the different roles that are using the individual pieces of functionality of the system, the use cases. Use case diagrams are often defined in terms of the actual business processes that will be supported by a system. Figure 1 displays an example use case diagram for an elevator system that indicates typical user interaction as well as the interactions that a maintenance engineer may perform with the system.

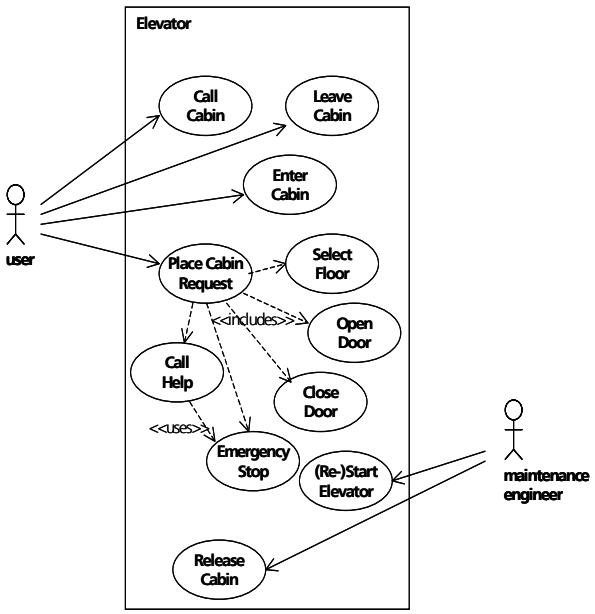



Figure 1:
Use case diagram for an elevator system.

Table 1: Basic use case template according to [Coc96, Coc01] and [Bok01]

| | |
|---|---|
| USE CASE # | Short name of the use case indicating its goal. |
| Goal in Context | Longer description of the goal in the context. |
| Scope & Level | Scope and level of the considered system, e.g. black-box under design, Summary, Primary Task, Sub-function, etc. |
| Primary, Secondary Actors | Role name or description of the primary and secondary actors for the use case, people, or other associated systems. |
| Trigger | Which action of the primary/secondary actors initiate the use case. |
| Stakeholder & Interest | Name of the stakeholder and interest of the stakeholder in the use case. |
| Preconditions | Expected state of the system or its environment before the use case may be applied. |
| Postconditions on success | Expected state of the system or its environment after successful completion of the use case. |
| Postconditions on failure | Expected state of the system or its environment after unsuccessful completion of the use case. |
| Description Basic Course | Flow of events that are normally performed in the use case (numbered). |
| Description Alternative Courses | Flow of events that are performed in alternative scenarios (numbered). |
| Exceptions | Failure modes or deviations from the normal course. |
| NF-Requirements | Description of non-functional requirements (e.g. timing) according to the numbers of the basic/alternative courses. |
| Extensions | Associated use cases that extend the current use case (<<extends>>-relation). |
| Concurrent Uses | Use cases that can be applied concurrently to the current use case. |
| Revisions | Trace of the modifications of the current use case specification. |

Use case diagrams display only very limited information, so they are typically extended by use case descriptions or use case definitions. A sole use case diagram is quite useless for the concrete specification of what a system is supposed to do. Each use case in a use case diagram is typically individually specified and described according to a use case template. Each of these descriptions is attached to a use case in the diagram. Table 1 shows an example use case template with the individual topic definitions taken from [Coc96, Coc01] and [Bok01]. This represents typical items of a use case that might be important for expressing interaction with a system from a user's perspective. Use case templates may be different according to the applying organization and the software domain in which they are applied. Table 1 only represents an example of how to describe a use case in general terms. Table 2 displays a concrete instance of this template applied to the use case *Select Floor* of the elevator system shown in Figure 1.

Table 2: Use case description according to the use case template in Table 1 for the elevator system.

| Use Case # **4.1** | **Select Floor** |
|---|---|
| Goal in Context | Main user scenario for getting to a different floor |
| Scope & Level | Primary user task |
| Primary Actor | User |
| Trigger | Select **SelectedFloor** number from the **CabinPanel** |
| Stakeholder | Analyst, Designer, Performance Engineer |
| Precondition | [ **SelectedFloor != CurrentFloor** ] |
| Postcondition on Success | [ **CabinPanel** lights up SelectedFloor indicator ] AND<br>[ **Cabin** eventually stops in the **SelectedFloor** ] |
| Postcondition on Failure | [ **CabinPanel** does not light up **SelectedFloor** indicator ] OR<br>[ Cabin does not stop in the SelectedFloor ] |
| Description Basic Course | 1. [ **SelectedFloor** < **CurrentFloor** ] AND [ **CabinDirection** == Down ] **Cabin** stops on the **SelectedFloor** on its way down<br>2. [ **SelectedFloor** > **CurrentFloor** ] AND [ **CabinDirection** == Up ] **Cabin** stops on the **SelectedFloor** on its way up<br>3. [ **SelectedFloor** < **CurrentFloor** ] AND [ **CabinDirection** == Up ] Cabin goes to **CurrentMaxFloor** first and approaches **SelectedFloor** on its return downwards<br>4. [ **SelectedFloor** > **CurrentFloor** ] AND [ **CabinDirection** == Down ] **Cabin** goes to **CurrentMinFloor** first and approaches **SelectedFloor** on its return upwards |
| Exceptions | [ **SelectedFloor == CurrentFloor** ] ring **Bell** AND open **Cabin/FloorDoor** within 300 ms |
| NF-Requirements | Floor/CabinDoors opens within **300 ms** from selecting CurrentFloor |
| Concurrent Uses | Use Case 4.1: any other floor selections, Use Cases 4.2, 4.3, 4.4, 4.5 |
| Extensions | None |
| Revisions | ... |
| | |

### 3.1.2 Use Case Diagrams and Testing

Use case descriptions are mainly used for requirements-based testing and high-level test design. Testing with use cases can be separated into two groups according to the source of information that will be used for test development. The first one is testing that is based upon the use case diagram which is mainly suitable for test target definition, and the second one is testing that is based upon the information of the use case template which is more similar to typical black-box testing although on a much higher level of abstraction. Both are considered in more detail in the following paragraphs.

Use case diagram based testing

The use case diagram does not permit typical test case design with pre- and postconditions, input domains and return values because it does not go into such a level of detail. However, we can define the following elements in a use case diagram and their relations according to [Bin00] that may be suitable for the purpose of testing:

- An actor can participate in one or several use cases: This will result in an acceptance test suite for each individual actor and for each individual use case, and the tests will reflect the typical usage of the system by that actor.

- A use case involves one or several actors: Each test suite will comprise tests that simulate the user's interactions at the defined interaction point, that is the use case functionality. If several actors are associated with the same use case we may additionally have concurrent usage of some functionality by different roles. For testing it means that we will have to investigate whether multiple concurrent usage is supported by the system as expected. We might therefore have to define a test suite that takes such a simulation into consideration.

- A use case may be a kind of some other use case (<<extends>>): If our test criterion is use case coverage we will have to produce test suites that comprise all feasible usage permutations of the base use case and its extension. This is very similar to checking correct inheritance in object-oriented testing [Bin00].

- A use case may incorporate one or more use cases (<<uses>>). For testing, this is essentially the same as the previous item.

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

- A use case is implemented through one or several nested and interacting components. Requirements-based testing should attempt to cover all components that are participating in the implementation of a use case. This is particularly important if requirements are changed. In that case, we will have to trace the changes to the underlying component architecture and amend the individual components accordingly. A regression test should then be applied in order to validate the correctness of these amendments.

– A component supports one or more use cases. Here, the component architecture is not functionally cohesive. In other words, individual components are responsible for implementing non-related functionality, leading to low cohesion in the components. This may be regarded bad practice but it surely happens, and often it is a requirement. In such an instance use cases represent different, and probably concurrent usage of the same component, and that must be reflected in the validation. We might therefore have to define a test suite that takes the concurrency situation into consideration as discussed before.

As said earlier, the use case diagram is mainly used for test target identification, and in order to achieve test coverage on a very high level of abstraction. For example Binder [Bin00] defines a number of distinct coverage criteria that can be applied to use case diagrams in order to come up with a system-level acceptance test suite:

– Test or coverage of at least every use case.
– Test or coverage of at least every actor's use case.
– Test or coverage of at least every fully expanded inclusion, extension and uses combination.

These correspond to coverage of all nodes and arrows in a use case diagram, and they are typical testing criteria similar to the traditional test coverage measures that are based on program flow-graphs as discussed in Chapter 2. Each of these criteria represents a test of high-level user interaction. Because there is only limited information we can only determine which user functionality we will have to test, but not how we can test it. Each test target can be augmented with information from the more concrete use case definitions in order to identify more concrete test artifacts. Therefore, each test target will map to a test suite and eventually, when more information is added, to a number of more concrete test cases. The collection of all user level tests that are developed in that way may be used for system acceptance testing.

Use case and operation specification based testing

Each specification of a use case according to the introduced use case template corresponds to a component's operation specification according to the operation description templates of typical development methods such as the KobrA method [Atk01]. Each use case is attributed to one or more high-level or abstract components that implement the use case's described functionality. An operation specification comprises such a full description of functionality and it is attributed to a distinct concrete object or component in the overall system. For example, a class method may be regarded as such an operation. KobrA's operation specification template is displayed in Table 3. In contrast, a use case represents a piece of functionality that is not attributed to a particular object in the system but to the entire system at its boundary. If we consider an individual component to be a system in its own right, as it is the case in the KobrA

method, then the operation specifications of that component and its corresponding system level use case specifications are essentially the same. The entries in the two templates in Table 1 and Table 3 indicate this conceptual similarity. Both templates define name and description, pre- and post conditions, and exceptions that can be related to constraints. The fundamental difference between the two items lies in the fact that use case descriptions in contrast to operation specifications define no concrete input and output types that easily map to input and output value domains, and that the pre- and postconditions are not attributable to distinct objects in the system. In other words, use case descriptions represent similar information as operation specifications although on a much higher level of abstraction, and from a different viewpoint. Whereas use case specifications are mainly used for communication outside the development team (e.g. with the customer of the software), operation specifications are more suitable for communication between the roles within the development team (e.g. system designers, developers and testers). In any case, use case template based testing and operation specification based testing represent both typical functional or black box testing approaches because both representations are merely concentrating on external expected behaviour. In other words, use case descriptions specify behaviour on the system level, whereas operation specifications describe behaviour on the object or component level.

Table 3: Operation specification template according to the KobrA development method.

| **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 |

More abstract representations such as use case models and use case descriptions were not initially taken into account as basis for typical functional testing approaches such as the ones mentioned in Chapter 2 because traditional testing always used to be, and probably still is, focused on more low-level abstractions and concrete representations of a system such as code. Therefore, these functional testing techniques appear to be more optimally used in tandem with typi-

cal operation-specification-type documents as primary source for test development. However, since the two models, use case descriptions and operation specifications are concerned with essentially the same information but on different levels of abstraction, the functional testing techniques can also be based on use case descriptions, although on a more abstract level, of course.

Examples for use case template based testing.

The use case template depicted in Table 2 already contains a number of items that are suitable for the definition of system level tests. The pre- and postconditions (on success) in the template and the description of the basic course map to the abstract test cases 1.x to 4.x in Table 4. The tests are abstract since they do not indicate concrete values for *Selected/CurrentFloor* variables, or the *CabinDirection*. Each of these abstract tests may instantiate to quite a number of concrete test cases (indicated through the x) that represent different scenarios of selecting floors and going up and down in an elevator. The test case 5.x in Table 4 represents an exception. It is derived from the *Postconditions on Failure* and the *Exceptions* in the use case specification. This abstract test case could for instance map to a concrete test for each possible *CurrentFloor* in order check that this facility for opening the door is actually working on every single floor.

Table 4: Abstract test cases derived from the use case specification displayed in Table 2.

| No. | Pre-Condition | Event | Post-Cond. | Result |
|---|---|---|---|---|
| 1.x | [ SelectedFloor != CurrentFloor ] & [ CabinDirection == Down ] | SelectFloor ( SelectedFloor < CurrentFloor ) | SelectedFloor lights up | Cabin stops on SelectedFloor |
| 2.x | [ SelectedFloor != CurrentFloor ] & [ CabinDirection == Up ] | SelectFloor ( SelectedFloor < CurrentFloor ) | SelectedFloor lights up | Cabin stops on SelectedFloor on next DownRun |
| 3.x | [ SelectedFloor != CurrentFloor ] & [ CabinDirection == Down ] | SelectFloor ( SelectedFloor > CurrentFloor ) | SelectedFloor lights up | Cabin stops on SelectedFloor on next UpRun |
| 4.x | [ SelectedFloor != CurrentFloor ] & [ CabinDirection == Up ] | SelectFloor ( SelectedFloor > CurrentFloor ) | SelectedFloor lights up | Cabin stops on SelectedFloor |
| 5.x | [ SelectedFloor == CurrentFloor ] | SelectFloor ( CurrentFloor ) | CurrentFloor does not light up AND Door opens | Cabin does not stop on SelectedFloor (Floor is not added to StopList) |
| ... | ... | ... | ... | ... |

The tests in Table 2 are all based on fundamental testing techniques summarized in the following [Jac92]:

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

The second two items in this list refer to more global information that is not necessarily contained in the individual use case specification. The line *Concurrent Uses* in the use case definition indicates that the use case itself may be invoked concurrently, and that other use cases may be applied at the same time. In the first case, we have to reflect the concurrency issue in the test suite for this use case. It means we can have any odd combination of the 5 abstract tests (Table 4) as a test sequence where every sequence maps to a single concrete test case with multiple events, i.e. leading to a sequence of elevator instructions. This identifies a fundamental specification deficiency in this example, and it shows how test considerations may actually improve the specification and design of a system before the tests are actually executed on the real thing. The use case specification indicates no explicit policy for handling concurrent floor selection requests. It means the specification says nothing about the sequence in which these requests will be handled. Of course common sense would suggest that the floors should be served in consecutive order of the cabin passing them. But this is only common sense because we know how an elevator is supposed to work, so we imply some domain knowledge. For other domains this might not be entirely clear. Hence, an elevator that would serve the floor requests in a temporal order of their appearence, that is the order in which users actually press the buttons, would be a correct solution according to this specification, and this is possibly not what the customer of an elevator has in mind.

Additional testing techniques that may be used in tandem with use case modeling and the specification of use cases are scenario-based techniques as described in the SCENT Method, but it requires some additional modeling efforts with dependency charts [RG99, RG00, RG00]. There are also distinct coverage criteria coming with these techniques that may be applied in use-case-based testing such as scenario-path coverage, event-flow coverage, or exception coverage, or even statistical usage-based testing techniques [RRW98, RR98].

## 3.2 Structural Modeling with Class/Object, Package, Component, and Deployment Diagrams

High level structural modeling is typically the next development step after use case modeling. Use case descriptions loosely associate system functionality with components on a high system architectural level. In other words, we can already define the very fundamental parts of the system in a typical devide-and-conquer manner (decomposition) when we develop the use case descriptions. All objects in the use case model have a good chance to become individually identifiable parts during design, such as components or classes, objects, modules, or subsystems. Under the KobrA method [Atk01] they are all termed Komponents. If we have defined the first components we will typically decompose the system into smaller more manageable parts that are not immediately related to the high-level usage models but are more concentrating on internal functional aspects. This activity typically comes under the umbrella of system design and it

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

### 3.2.1 Concepts of Structural Diagrams

Component Containment Diagram Concepts

Components are the basic construction entities in component-based development. A system's primary components are typically identified in the requirements engineering phase of a project through use case modeling and the definition of use case descriptions as outlined in the previous sections. This is typically an outcome from distinct domain knowledge that determines the architecture, or it is determined through naturally available system parts in that domain. The identified high-level components may be brought into a hierarchy that represents the coarsest-grained structural organization of the entire system. In the KobrA Method it is termed component containment model. Figure 2, for example, displays the architectural organization of an elevator. Such a component containment hierarchy plays a seminal role in KobrA's development process [Atk01]. KobrA uses the UML package symbol for representing components since a component is a collection or package of a number of different specification and documentation artifacts, e.g. a collection of UML models and tabular representations such as operation specifications. This indicates the scoping of such descriptive artifacts.

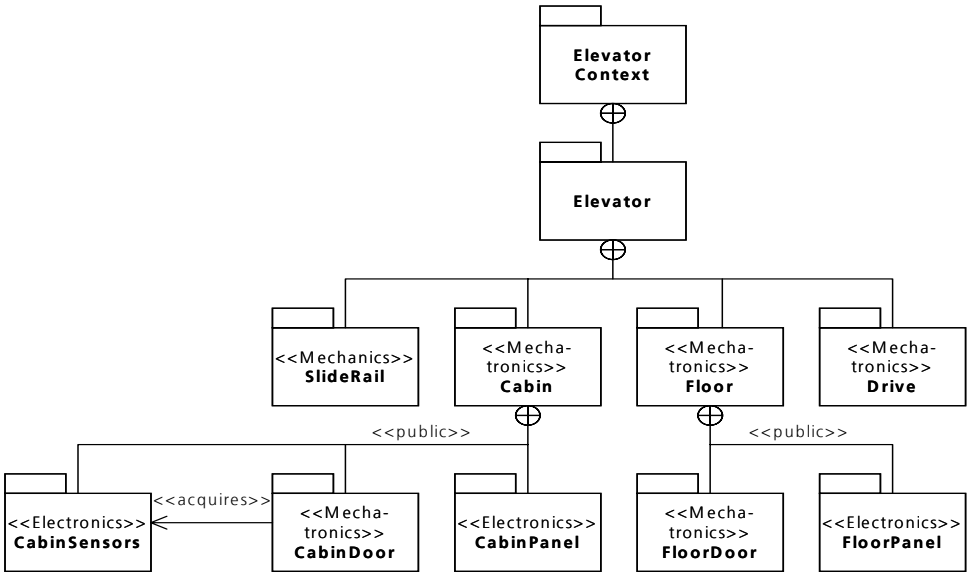
Each component in the hierarchy is described through a specification that comprises everything externally knowable about the component in terms of structure, function and behaviour, and a realization that comprises everything internally knowable about that component. The specification describes what a component is and can do, and the realization how it does it. The subject component indicated through the stereotype <<subject>> represents the entire system under consideration. For this example it is the *Elevator* component. The context realization, in this case the component *Elevator Context* describes the existing environment into which the subject will be integrated. It contains typical realization description artifacts. The anchor symbol represents containment relations, or in other words, the development time nesting of a system's components.

Component nesting always leads to a tree shaped structure, and it also represents client/servership. A nested component is typically always the server for the component that contains it, and a nesting component is always the client of the components which it contains. This is at least the case for creating a new instance of a contained component. The super-ordinate component is the context for the sub-ordinate component and it calls the constructor of the sub-ordinate component. This can be seen as the weakest form of client/servership between nested entities. Containment trees can also indicate client/server rela-

tionships between components that are not nested. This is indicated through an <<acquires>> relationship between two components in which one component acquires the services of another component as indicated between *CabinDoor* (client) and *CabinSensors* (server) in Figure 2. This type of explicit client/servership leads to an arbitrary graph.

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

Figure 2: Example Component Containment Hierarchy of an elevator.




## Class Diagram Concepts

Class diagrams and object diagrams are made up of classes or objects (class instances) and their associations. The associations define the peer-to-peer relations between the classes/objects, so class/object diagrams are primarily used for specifiying the static, logical structure of a system, sub-system, or a component, or parts of these items. Associations come in different shapes with different meanings. A normal association defines any arbitrary relationship between two classes. It means, they are somehow interconnected. An aggregation is a special association that indicates that a class (i.e. the aggregate) is comprised of an other class (i.e. the part). This is also referred to as "whole-part-association" or class nesting (similar to the containment model) and it is not specific about who

creates and owns the sub-ordinate classes. Ownership between classes is indicated through the composition aggregation. This is a much stronger form of aggregation in which the parts are only created and destroyed together with the whole. Generalization is a form of association that indicates a taxonomic relationship between two classes, that is a relationship between a more general class and a more specific class. In object technology terminology this is also referred to as inheritance relationship. A refinement association indicates a relationship between two descriptions of the same thing, typically on two distinct abstraction levels. It is similar to the generalization association, although the focus here is not on taxonomy but on different levels of granularity or abstraction. Dependency is another form of association that expresses a semantic connection between two model elements (e.g. classes) in which one element is dependent upon another element. In other words, if the non-dependent class is changed, it typically leads to a change in the dependent class. Multiplicity parameters at associtations indicate how many instances of two interconnected classes will participate in the relation.

Class symbols have syntax too. They are consisting of a name compartment, an attribute compartment, and an operation compartment. The latter two define the externally visible attributes and operations that the class or object is providing at its interface, and which may be used by external clients of the class in order to control and access its functionality and behaviour. Figure 3 shows an example class diagram of the elevator system.

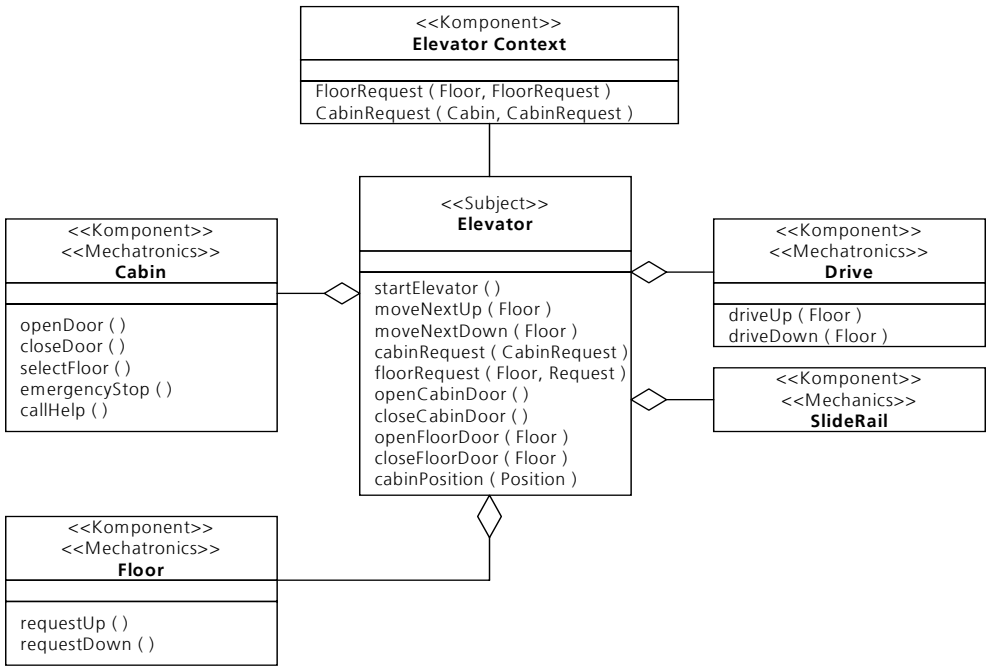



Figure 3:
Example Class Diagram of an elevator.

Package, Component and Deployment Diagram Concepts

A package diagram may comprise classes/objects, components, and packages. A package is only a grouping mechanism that can be linked to all types of other modeling elements, and that can be used to organize semantically similar, or related items into a single entity. Sub-systems, components and containment hierarchies of classes may also be referred to as packages since all these concepts encapsulate various elements within a single item in the same way as a package. For example, the KobrA development method [Atk01] uses the package symbol for specifying a Komponent (KobrA Component) since it represents a grouping of all descriptive documents and models that collectively define a component in terms of functionaliy, behaviour, structure, and external and internal quality attributes.

A component diagram organizes the logical classes and packages into physical components when the system is executed. It represents a mapping from the logical organization of a system to the physical organization of individually executable units. Its main focus is on the dependency between the physical components in a system. Components can define interfaces that are visible to other components in the same way as classes, so that dependencies between components can also be expressed as access to interfaces. Class and component diagrams are similar with respect to this property since they can both specify associations between modeling elements. The KobrA method [Atk01] advocates a special type of component diagram, the component containment hierarchy. KobrA is inherently component oriented from the very beginning of a development project. Components are therefore identified and handeled right from the early project life cycles.

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

### 3.2.2 Structural Diagrams and Testing

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

definition in the same way as use case models. In class diagrams we are simply more concrete about things, and this is what we can actually exploit in terms of testing, and we can add that information to the information that we derived from use case models.

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

### Component Containment Diagrams and Testing

KobrA component containment hierarchies [Atk01] can be seen as the most general and most abstract logical structural models. They display component nesting and consequently client/server relations between a super-ordinate component and its contained sub-ordinate components, represented through the anchor symbol, as well as client/server relations between components on the same or neighbouring hierarchic levels, represented through arbitrary <<acquires>> relations. Both concepts indicate that one component is requireing the services of another component, so there must be an interface definition between the two parties in that client/server relationship. In object terminology this is equivalent with a class attribute. Each connection in a containment hierarchy therefore relates to a test target and consequently to a test suite, or an additional tester component that specifically concentrates on testing that connection. This is illustrated in Figure 4. A test of the sub-system in Figure 4 requires that the communication between all integrated components is tested in combination. The *Cabin* component expects to get some features from its sub-ordinate components and in return these components expect to be used by *Cabin* in a certain way. This is a mutually accepted contract between these parties. In the same way, the *CabinDoor* component expects to get a distinct service from the *CabinSensors* component (indicated through the <<acquires>> relation). These mutual expectations can be represented through test suites that can be executed as unit tests on the individual components before the sub-system is integrated. Each test suite will only contain tests that simulate the access of the respective client component on the server. If all the tests in all the test suites pass, we expect that the integration will be successful, and that the sub-system or component *Elevator Cabin* will expose no more failures, given that we have applied adequate test sets.

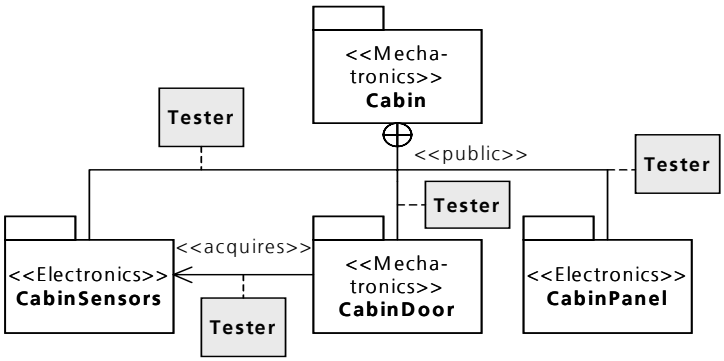



Figure 4:
Each client/server relation relates to a test suite or tester component.

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

Containment diagrams also contain information about the time in the development and deployment cycle of a system for performing a test. Components are the fundamental building blocks in a component-based development, so they are unlikely to be torn apart and have their internal parts being integrated into other entities. Components typically stay as they are and they are reused as they are. Everything inside a component's encapsulating boundary is therefore quite likely to stay the same all the time. So, inside a component during component development we are only concerned with development-time testing. Once the component's subordinate parts are successfully integrated and they pass their internal tests we are done, and we will never touch the thing again. We can therefore remove any built-in testing infrastructure that has been used during component testing. However, at the component's external boundary we should leave all the built-in testing infrastructure where it is. We can execute that whenever the component is integrated with other components to form a new system, i.e. at component integration and deployment time. In that respect all components are individual building blocks that are capable of checking their own deployment environment whenever they are reused in a new system. This technique is termed built-in contract testing and fully described in [Gro02a, Gro02b].

### Class Diagrams, Package, Component and Deployment Diagrams and Testing

All the other structural diagrams in the UML such as class, package, component, and deployment diagrams are used to express the implementation and deployment of components, for example class diagrams in KobrA are mainly used in component specifications and realizations, but they can also express the distribution of logical software components over hardware nodes. Packages, components and classes are very similar concepts, and the component term combines their individual particularities, i.e. the component provides a scoping mechanism

in order to package a variety of concepts and artifacts such as classes and modules. Components do provide interfaces in the same way as classes do, and maybe packages as well, and they have states. In fact, a component's class properties are provided through the classes that the component contains. All testing concepts of the previous paragraphs are therefore applicable to classes and packages, that is test target definition and identification of client-servership contracts that need to be tested. The different diagrams merely represent slight variations of the modeled subjects, or different views. The fundamental difference between classes and components, for example, is that class interactions are likely to stay fixed for a longer period of time. Components may be seen as the fundamental building blocks of systems, and they are often reused and integrated in different contexts, and classes can be seen as the fundamental building blocks of components, so they are not so readily reused because components are not so much subjected to permanent change. The difference in the diagrams with respect to testing is not so much concerned with extracting different testing information from the models but more with the strategy of when tests will be ideally executed. The fundamental idea of defining locations for performing integration tests is the same in all diagram types. But we can extract more information on when to perform these tests. A class diagram shows interaction between the fundamental building blocks of a component. They are likely to stay as they are during the lifetime of the component. So we integrate and test that integration once and for all. Compoents are the building blocks of systems. So whenever we put components together in order to come up with a new system, we will check this integration through a test. Some components will be assigned dynamically others will stay as they are. Component diagrams show this type of organization, so that we can identify fixed contracts and loose contracts that are likely to change and will need re-testing. Deployment diagrams represent a different view on the same problem. Here we assign components to nodes in a distributed environment. Some nodes will stay the same throughout the life span of a system and only need an initial check, but others might undergo constant change, so that we will have to perform an integration test whenever a node is changed. The fundamental idea of test target definition with structural diagrams does not change. We can still see from a structural diagram which interactions should be tested under which circumstances.

Built-In Contract Testing based on component-containment trees

The development of testing artifacts that are permanently built into software components according to the built-in contract testing paradigm [Gro02a, Gro02b] are heavily dependent upon structural models. Component containment trees identify the units of reuse in a component-based system, and they define the interfaces between the components that have to be augmented with permanent built-in contract testing interfaces and tester components. Class diagrams define structure that is typically more resilient to permanent change, so they define interfaces between components that will typically stay as they are for some longer period of time. Therefore, these contracts identify locations for removable built-in contract testing interfaces and tester components. The development principles for component containment trees and class diagrams are the

same with respect to built-in contract testing. They are fully described in [Gro02a, Gro02b]. Figure 5 displays the structural organization of a Resource Information Network that is used as a case study in the MDTS project funded by the German Federal Ministery of Education and Resarch [MDTS]. The built-in contract testing approach proposes a number of additional components in order to make this system fully testable for an integration test. Figure 6 displays this architecture. First of all, it adds testable interfaces to each component in the system that will be tested by a tester component. This turns a normal component into a testable component indicated through the term "Testable" in the component name. In object technology this is typically achieved through extending the main class in the component by operations that are setting and checking internal states or assertions. The development of these operations is driven by the behavioural model of a component. The second step is to add a tester component to each component that uses another component's interfaces. Figure 6 displays a number of tester components for some original component. They can also be put together. The tester components comprise the actual test cases and they simulate the behaviour of a client on the server for which they have been developed. The testing architecture is permanently built into the components, so that if they are integrated with other new components they can automatically check their deployment environment.

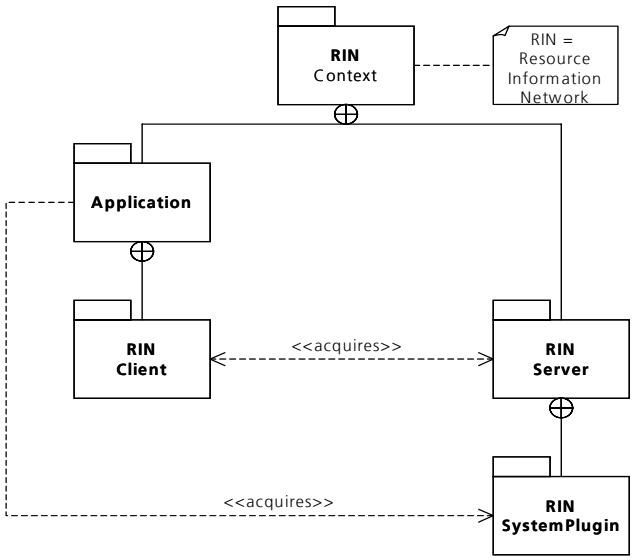



Figure 5: Component Containment of the Resource Information Network (RIN) System.

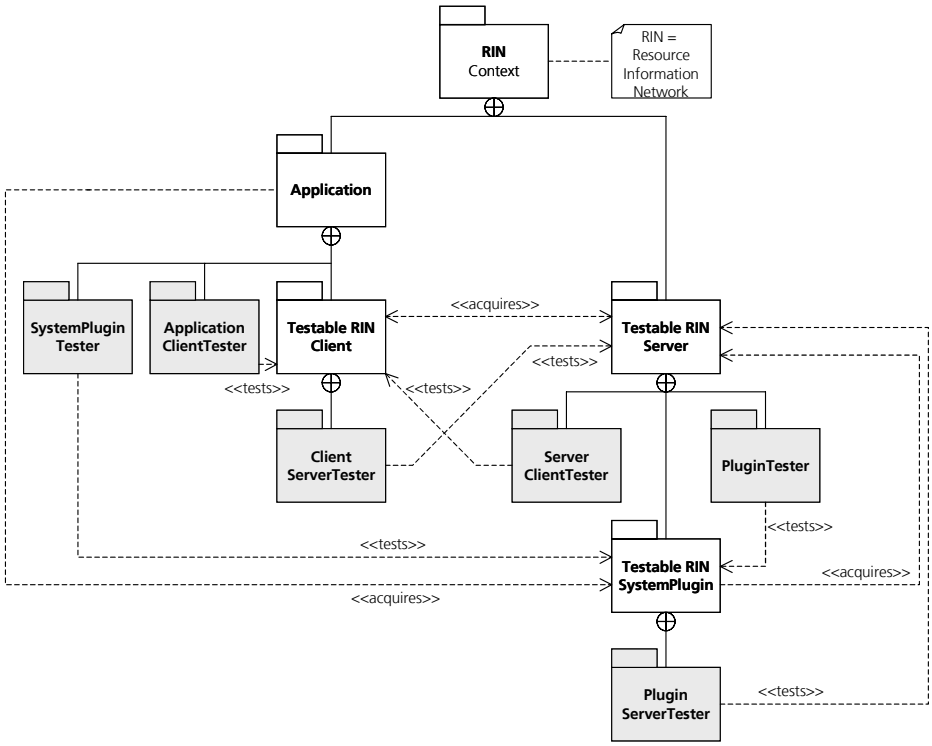



Figure 6: Component Containment Hierarchy of the RIN System with a complete built-in contract testing architecture.

Additional structural diagram based testing techniques are more geared towards implemenation-specific items, and they focus on such things as multiplicity of associations between classes representing boundary conditions on these associations, or testing of established required relationships, missing links, wrong links, dynamic class allocation, and acquisition. These testing concepts are further elaborated in [Bin00].

## 3.3 Behavioural Modeling with Statechart and Activity Diagrams

Structural modeling is part of system decomposition, and it identifies the subparts of the system that will be individually tackled in separate development efforts. Each part can be subdivided further into even smaller units. If such a part or component has been identified, its behaviour must be described, this comprises its externally visible behaviour at its provided interface as well as the externally visible behaviour at its required interface. The UML supports behavioural modeling through satechart diagrams and activity diagrams.

Statechart diagrams represent the behaviour of an object by specifying its responses to the receipt of events. Statecharts are typically used to describe the behaviour of class or component instances, but they can also be used to describe the behaviour of use-cases, actors, or operations. Related to statechart

diagrams are activity diagrams that concentrate on internal behaviour of an instance, in other words the control-flow within its operations. Both diagram types are essentially based upon the same fundamental concepts.

### 3.3.1 Statechart Diagram Concepts

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

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

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

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

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

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

### 3.3.2 Statechart Diagrams and 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 that are defined by the built-in contract testing technology (state checking operations) that has been developed in the European Union funded project *Component+* [Comp+]. The following paragraphs describe the main test case design strategies for state-based testing:

Piecewise coverage

Piecewise coverage concentrates on exercising distinct specification pieces, for example coverage of all states, all events, or all actions. These techniques are not directly related to the structure of the underlying state machine that implements the 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.

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].

Implementation of tester components for built-in contract testing

The coverage criteria described in the previous paragraphs can be applied to fill the built-in contract tester components with life. Every component of the system has a tester component associated with it, as indicated in Figure 6 for the RIN system example. The tester component is capable of checking the component's run-time environment in situ when it is integrated (at deployment) [Gro02a, Gro02b]. The test cases are developed according to the component's expectations towards its other associated components. In KobrA, a component's expec-

tation is defined through a realization behavioural model. In other words, every component has a model of the behaviour that it expects to get from its associated sub-components. Such realization behavioural models could for example look like the state chart diagrams in Figure 8 to Figure 10. These are in fact KobrA specification behavioural models of the respective components, defining which behavioural features these components are providing, but if the entire RIN system is integrated, superordinate realization models have to map exactly to subordinate specification models. This mapping is not a problem of testing, but of component composition and integration, so specification behavioural models must be the same as realization behavioural models of superordinate system parts. Otherwise we cannot integrate the components and test their interactions. Figure 7 displays the principle of the contract testing architecture. Each tested component provides a testing interface that extends the original component (shaded in Figure 7), and it provides testing operations that associated client tester components may use to support the testing. Each testing component (client) owns a server tester component. This contains tests that check the server's compliance to its contract with the client.

From the behavioural models in Figure 8 to Figure 10 we can devise a number of tests for tester components that follow the built-in contract testing paradigm. For example they can abide by the transition coverage criterion described in the previous paragraphs. The following tables, Table 5 to Table 8, illustrate the test cases that we can derive from these models (note, that they are not complete).

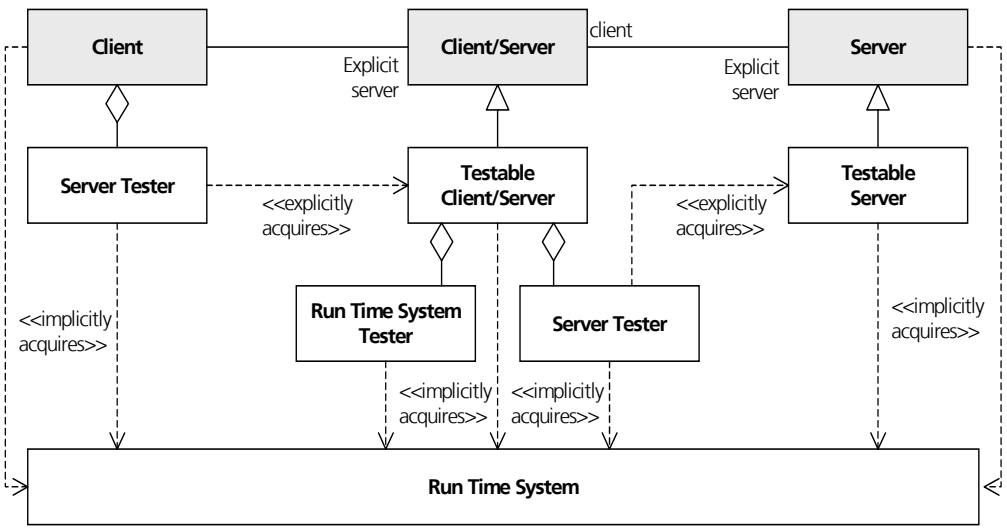



Figure 7:
Typical general built-in contract testing architecture.

Table 5:
Test cases for the Plugin Tester component of the RIN system.

| No. | Initial State | Event | Expected Outcome | Final State |
|---|---|---|---|---|
| 1 | Registered & Active | CRINSystem::ProcessData | Request to Plugin | Registered & Active |

Table 6:
Test cases for the Server ClientTester component of the RIN system.

| No. | Initial State | Event | Expected Outcome | Final State |
|---|---|---|---|---|
| 1 | Registered | ICallBackObj::ReceiveData-FromServer | Answer for a Client | Registered |

Table 7:
Test cases for the Plugin ServerTester component of the RIN system.

| No. | Initial State | Event | Expected Outcome | Final State |
|---|---|---|---|---|
| 1 | Registered | ICRinServerSink::OnData-FromPlugin | Answer for the Server | Registered |

Table 8:
Test cases for the SystemPluginTester component for the RIN system.

| No. | Initial State | Event | Expected Outcome | Final State |
|---|---|---|---|---|
| 1 | Registered & Active | IIDCOMRinServer::Pro-cessRequest("bypass") | Same message returned from the server to the client | Registered & Active |
| 2 | Registered & Active | IIDCOMRinServer::Pro-cessRequest("repeat") | The request remains automatically inside the plugin to repeat every-time it was programmed | Registered & Active |
| 3 | Registered & Active | IDCOMRinServer::Process-Request("cancel") | This cancels a specific request which remains active | Registered & Active |
| 4 | Registered & Active | IDCOMRinServer::Process-Request("abstime"+ | The request is executed in a date and time determined by the mes-sage | Registered & Active |
| 4.1 | | "MemoryLoad") | Value of the total memory usage | |
| 4.2 | | "TotalPhys") | Value of the total physical memory usage | |
| ... | ... | ... | ... | ... |
| 11 | waiting | IDCOMRinServer::Process-Request("bypass") | Error | waiting |
| 12 | waiting | IDCOMRinServer::Process-Request("repeat") | Error | waiting |
| 13 | waiting | IDCOMRinServer::Process-Request("cancel") | Error | waiting |
| 14 | waiting | IDCOMRinServer::Process-Request("abstime"+ | Error | waiting |
| 14.1 | | "MemoryLoad") | Error | |
| ... | ... | ... | ... | ... |

Design of built-in testing interfaces for built-in contract testing.

The development of the testing interface that extends each server component, is also based on the server's behavioural model. Every identified abstract state in the server's state chart diagram defines a distinct domain of feasible attribute combinations. Pre- and postconditions of operation invocations are dependent upon these domains. In other words, an operation can only be performed properly by a component if some conditions are satisfied, these are the preconditions, and the initial state of a component is part of that. This is termed the client's contract. The server performs its operation and adjusts its attributes according to some other conditions, these are the postconditions, and the final state of a component is part of that. If the client does not abide by its own contract, the server is not obliged to fulfill its side of the contract either. Built-in contract testing is a way of checking that such client/server interactions are correct.

In built-in contract testing every identified state can be represented by a state checking operation that essentially assesses whether all attributes are within the required domain for the state. This is important for state-based testing since not every state is also an output state. A testing interface that provides a built-in access mechanism to the component therefore adds considerable value to a component's testability and observability. Faults can therefore be detected when they appear and not when we observe some failure in a subsequent output state.

Additionally to developing a state checking mechanism, built-in contract testing also offers the concept of a built-in state setting mechanism. This brings the component into a state from which a transaction will be called in a test. This has the advantage that a test case can renounce the definition of a sequence of operations to get into a distinct state that is interesting for a test. However, state setting mechanisms are often difficult to develop. Figure 8 to Figure 9 show the testing interfaces for the RIN components that are partially derived from the respective state models. The other operations in the testing interfaces are based on typical built-in assertion concepts.

Figure 8: Behavioural model for the RIN System server component plus structural model for its testing interface.

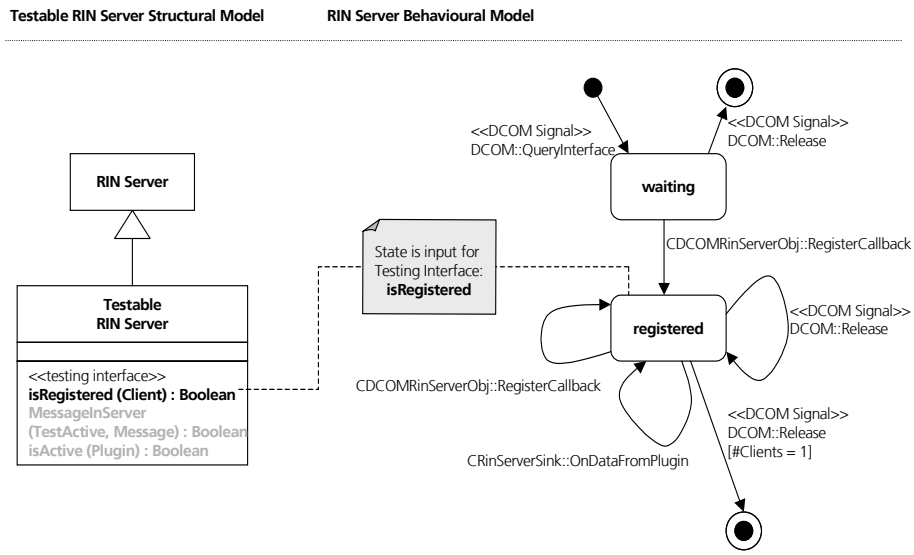
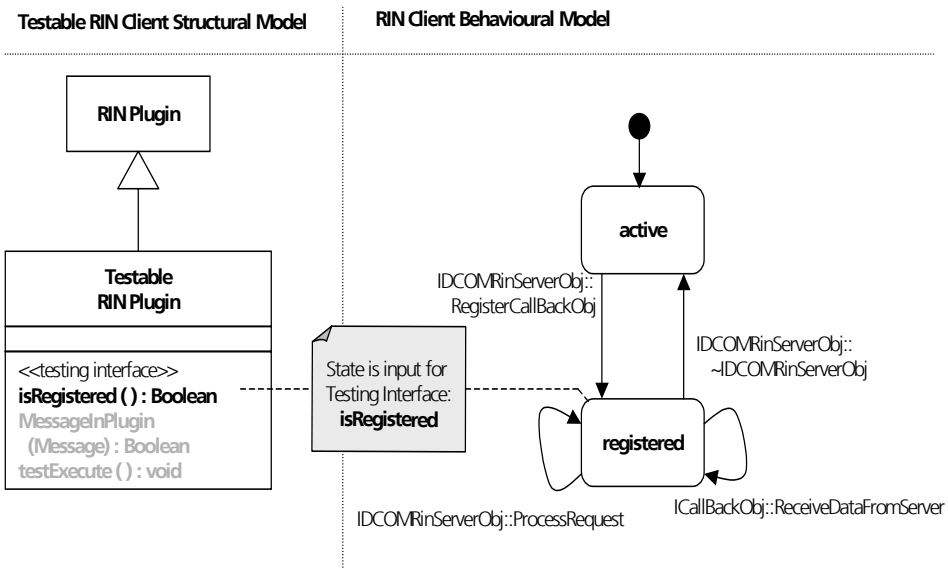



Figure 9: Behavioural model for the RIN client component plus structural model for its testing interface.



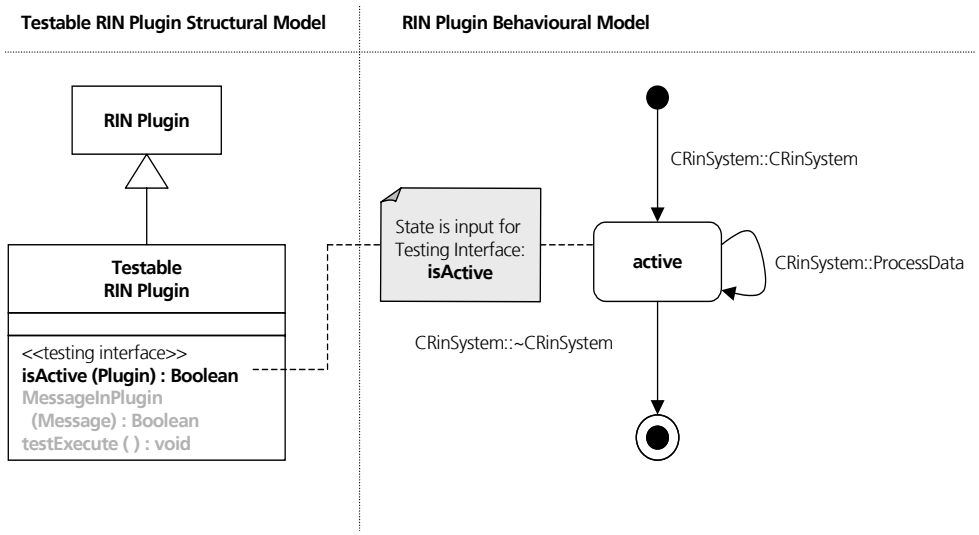



Figure 10: Behavioural model for the RIN plugin component plus structural model of its testing interface.

### 3.3.3 Activity Diagram Concepts

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

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

### 3.3.4 Activity Diagrams and Testing

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

Testing of control-flow within a single instance

Control-flow testing within an instance corresponds to a typical white-box unit test, though the value of white-box testing is limited in component development and testing. Component-based testing is more concerned with the integration of objects and their mutual interactions rather than with their individual internal workings. For unit testing, activity diagrams provide typical traditional code coverage measures, although on a higher level of abstraction. An activity may be a single low-level statement, a block of such statements, or even a full procedure with loops and decisions. Typical code coverage criteria can be adapted easily to cope with activity diagram concepts. Traditional control flow graphs and UML activity diagrams are essentially the same, and Beizer [Bei90] treats control flow-based testing thoroughly. We can identify several flows of control in activity diagrams that we can map to traditional coverage criteria according to Beizer:

- testing the control flow graph (traditional coverage criteria [Bei90]).
- control flow coverage (solid arrow)
- message flow coverage (dashed arrow)
- signal flow coverage (dashed arrow).

Testing of control-flow between instances

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

High-level transactions are typically composed out of lover-level transactions of many different objects. If we base our testing on higher-level transactions, for example on transactions in a use case model, activity diagrams display which objects are participating in a transaction. Each modeled transaction defines all its associated objects. So, when we start testing our system, we know which objects we will have to assemble and create, or for which objects in a transaction chain we have to devise test stubs. Figure 11 shows an activity diagram for the RIN system. If we would like the check the transaction "Process Request",

the activity diagram tells us which objects we have to develop (in code, or compiled), that is the "RIN Client", the "RIN Server" and the "RIN Plugin", and how they interact for performing the transaction. These are the sequences of transactions that are taking place between the objects.

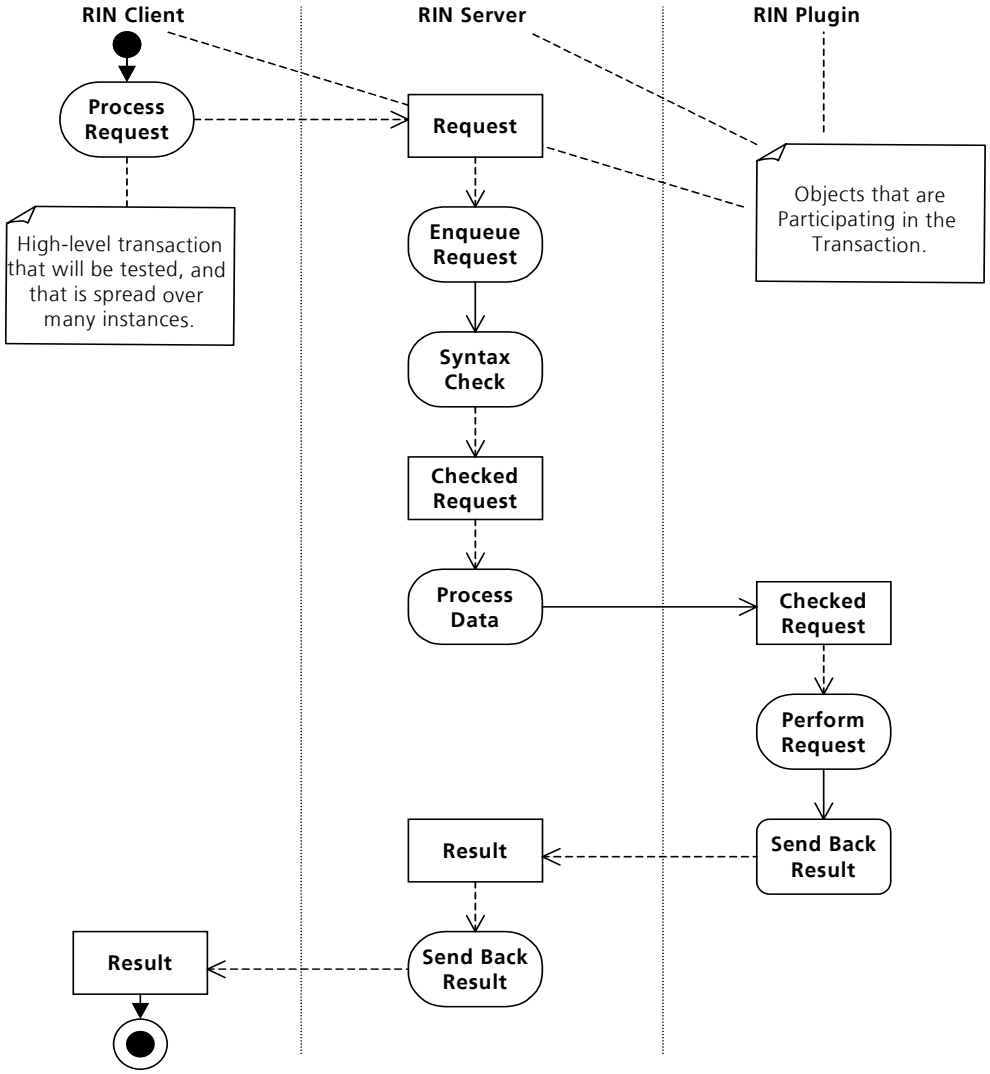



Figure 11: High-level RIN system activity diagram for processing a request.

## 3.4 Interaction Modeling with Sequence and Collaboration Diagrams

Interaction modeling represents a combination of dynamic and structural modeling. It mainly concentrates on the dynamic interactions between instances. The UML provides two diagram types for modeling dynamic interaction: Sequence Diagrams and Collaboration diagrams. They are both introduced in the next paragraphs.

### 3.4.1 Interaction Diagram Concepts

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

A sequence diagram shows interactions in terms of temporally ordered method invocations with their respective input and return parameters. The vertical axis shows the passage of time, and the horizontal axis the objects that participate in an interaction sequence. Through its focus on time passage, sequence diagrams also illustrate the life time of objects, it means through wich occurences they are created and destroyed. Labels can indicate timing properties for individual occurencs, so sequence diagrams are valuable for modeling and specifying real-time requirements. Messages that are sent between the instances can be synchronous, meaning that a subactivity is completed before the caller resumes execution, or asynchronous, meaning that the caller resumes execution immediatly without waiting for the subactivity to finish. The calling object and the called object execute concurrently in the second instance. This is important for embedded system development. Messages in sequence diagrams can take the same format as transition labels in state chart diagrams, although they do no have the action expression. In other words, a message can also be made conditional through a guard expression. The format of a message is defined as follows:

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

A sequence diagram starts with a single interaction, this is the considered scenario, that triggers the whole sequence of messages which are spread over the participating objects. Figure 12 displays an example sequence diagram for the Resource Information Network system.

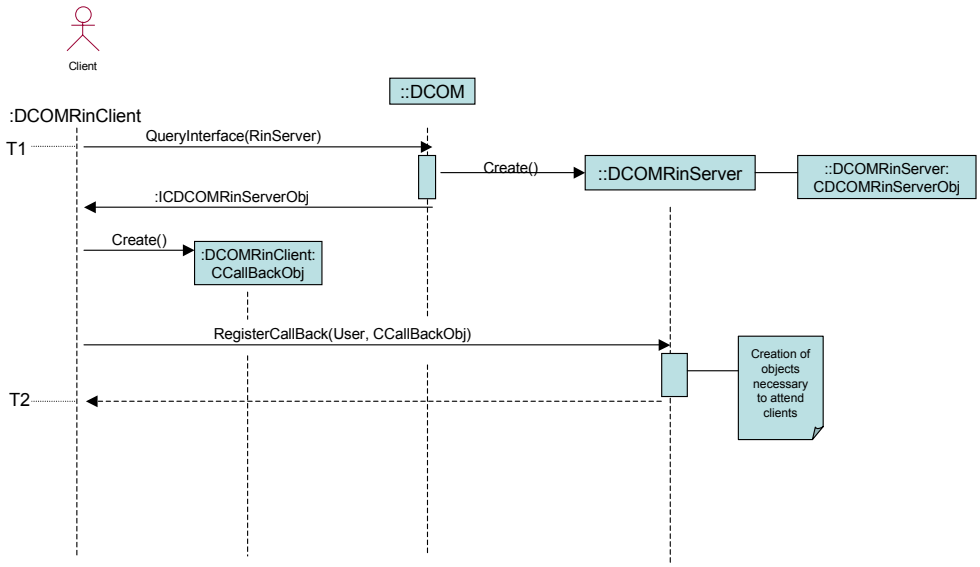
A collaboration diagram focuses more on structure and how it relates to dynamic interactions. It is similar to a class or object diagram, since it may also show internal realization of an object, that is its subordinate objects. Essentially, a collaboration diagram shows the same interactions as the corresponding sequence diagram. Although, here messages are numerically ordered and associated with a single interaction between two objects, rather than sequentially associated with a life line as it is the case in sequence diagrams. An interaction is a call path within the scope of a collaboration [Bin00]. Interactions in a collaboration diagrams have the following format:

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

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

The sequence number is an integer or sequence of integers which indicates the nesting level of a transaction sequence: 1 always starts the sequence, 1.1 represents the first sub-transaction on the first nesting level, 1.2 represents the second sub-transaction on the first nesting level, etc. 1.2a and 1.2b represent two concurrent messages which are sent in parallel. The asterisk indicates repeated execution of a message. Repetitions are more specifically defined thorugh iteration conditions. These are expressions that specify the number of repetitive message executions. Figure 13 shows the corresponding collaboration diagram according to the sequence diagram in Figure 12.

Figure 12: Example sequence diagram, RIN system initialization of the server.




### 3.4.2 Interaction Diagrams and Testing

Sequence and collaboration diagrams are typical control-flow diagrams although with slightly different foci. As the term interaction diagram implies they concentrate on control flow through multiple interacting instances. For testing, the two diagram types may be represented as abstract control-flow graphs that span multiple entities. With that respect we can apply all typical traditional control-flow graph based test coverage criteria as outlined in [Bei90]. This, of course, includes path and branch coverage criteria as well more exotic things such as round-trip scenario coverage [Bin00]. Since UML diagrams are always also more abstract than traditional control-flow graphs the test targets may be more abstract. Binder identifies some typical problems that may be discovered through sequence diagram-based testing [Bin00]:

- Incorrect or missing output
- Action missing on external interface.
- Missing function/feature (interface) in a participating object.
- Correct message passed to the wrong object.
- Incorrect message passed to the right object.
- Message sent to destroyed object.
- Correct execption raised, but caught by the wrong object.
- Incorrect exception raised to the right object.
- Deadlock.
- Performance.

The items in the list make the nature of interaction diagrams and their value for testing apparent. Additionally to typical control-flow issues these diagrams put considerable weight on collaboration and how that may be checked.

From the sequence diagram in Figure 12 we may derive the test sequence displayed in Table 9.

Table 9: Test sequence for checking the initialization procedure of the RIN server derived from the sequence diagram in Figure 12.

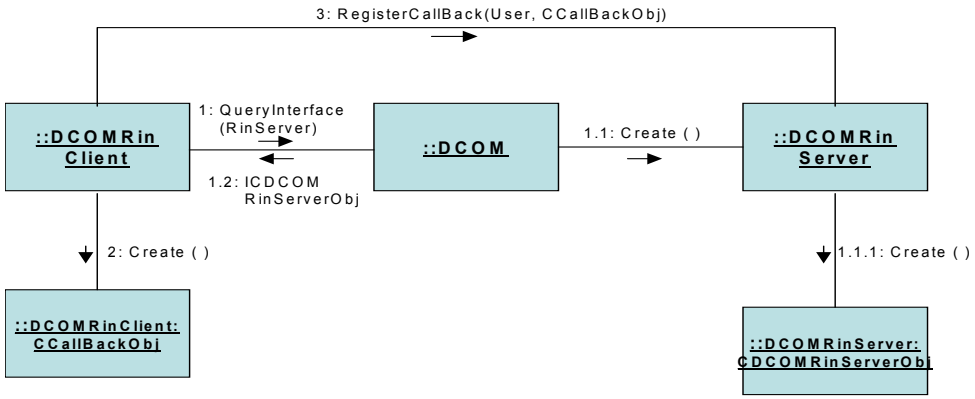
| No. | Initial State | Event | Expected Outcome | Final State |
|---|---|---|---|---|
| 1 | RinServer is not registered in DCOM | DCOM::QueryInterface (RinServer) | Return RINServerObject to Client | DCOM RinServer is registered & Active |
| 2.1 | DCOM RinServer is registered & active | RINClientCallBack-Obj::Create ( ) | Reference to the RinClientCall-backObj | |
| 2.2 | | IDCOMRinServer::Regis-terCallBack (User, RINCli-entCallBackObj) | OK | DCOM RinServer registered & active AND CallBackObj registered |

Any sequence may be augmented with additional timing specifications that are typically coming from high-level user requirements. For example, the registration of the RIN server in the DCOM environment may not take longer than 300 milliseconds. This is a timing requirement that will be attached to the sequence of several operation invocations as displayed in Figure 12. A test of this performance specification requires some additional infrastructural amendments in a tester component. In the test environment we need to read the timer before and after test execution and we have to amend the evaluation of the verdict that in this particular instance requires the calculation of the execution time. The test sequence in Table 9 will then be extended by timer operation calls as displayed in Table 10.

Table 10: Timer calls in a test sequence for performance assessment.

| No. | Initial State | Event | Expected Outcome | Final State |
|---|---|---|---|---|
| 1.1 | | ReadTimer (StartTime) | | |
| 1.2 | RinServer is not registered in DCOM | DCOM::QueryInterface (RinServer) | Return RINServerObject to Client | DCOM RinServer is registered & Active |
| 2.1 | DCOM RinServer is registered & active | RINClientCallBack-Obj::Create ( ) | Reference to the RinClientCall-backObj | |
| 2.2 | | IDCOMRinServer::Regis-terCallBack (User, RINCli-entCallBackObj) | OK | DCOM RinServer registered & active AND CallBackObj registered |
| 2.3 | | ReadTimer (StopTime) | | |

Figure 13: Example collaboration diagram, RIN system initialization of the server.




This chapter has concentrated on the individual UML diagram types and how they can be used in order to derive testing information. We have seen that typical structural models are more suitable for test target definition, and the development of testing architecture for a system. Test cases require more information and in particular more concrete information, specifically about functionality and behaviour. UML behavioural models are therefore indispensible for deriving a system's concrete tests that fill the testing infrastructure derived from structural models with life. The next chapter shows how the UML can be used to specify and model all the testing artifacts that are neccessary to check a system through a UML Testing Profile. Testing is just another functionality, so it can be modeled and designed just as any other functionality. Although, the testing profile provides additionally to the standard UML concepts that specifically support test development.

# 4 UML Testing Profile

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

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

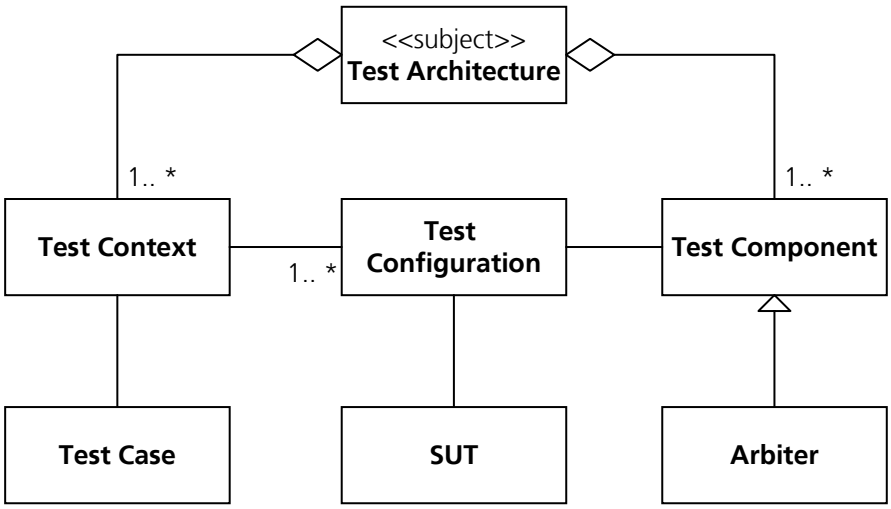
Testing also involves the description and definition of testing architectures, testing behaviour and physical testing implementation including the individual test cases. So, test development essentially comprises the same fundamental concepts and procedures as any other normal software development that is merely concentrating on function rather than testing. The testing infrastructure for a system is also software after all. Out of this motivation, the OMG has initiated the development of a UML testing profile that is specifically addressing typical testing concepts in model-based development.

The UML testing profile is an extension of the core UML, and it is also based upon the UML metamodel. The testing profile particularly supports the specification and modeling of software testing infrastuctures. It follows the same fundamental principles of the core UML in that it provides concepts for the structural aspects of testing such as the definition of test components, test contexts and test system interfaces, and behavioural aspects of testing such as the definition of test procedures, test setup, execution and evaluation. The core UML may be used to model and describe testing functionality since test software development can be seen as any other development for functional software properties. However, software testing is based on a number of very special addititional concepts that are introduced in the following and defined through the testing profile.

## 4.1 Structural Aspects of Testing

The UML testing profile defines the test architecture that copes with all structural aspects of testing in the UML. The test architecture contains test components and test contexts and defines how they are related to the specified system under test (SUT), the sub-system, or component under test (i.e. the tested software). A test context represents a collection of test cases, associated with a test configuration that defines how the test cases are applied to the SUT. A test configuration may comprise a number of test components and describes how they are associated with the tested component (SUT). A very special test component is the arbiter. It evaluates the test results and assigns an overall verdict to a test case. Feasible verdicts for a test result are pass, inconclusive, fail and error. Figure 14 summarizes the structural concepts of the testing profile.

Figure 14: Structural concepts of the testing profile.




## 4.2 Behavioural Aspects of Testing

The test behaviour is defined through a number of different concepts in the testing profile. The most important concept is undoubtedly the test case. It specifies what will be tested, with which inputs and under which conditions. Each test case is associated with a general description of its purpose. This is termed test objective and it essentially defines the test case. Each execution of a test case may result in a test trace. This represents the different messages that have been exchanced between a test component and the SUT. Finally, a test case also comprises its verdict. This indicates whether the test passed or failed. Figure 15 summarizes the concepts that are related to a test case.

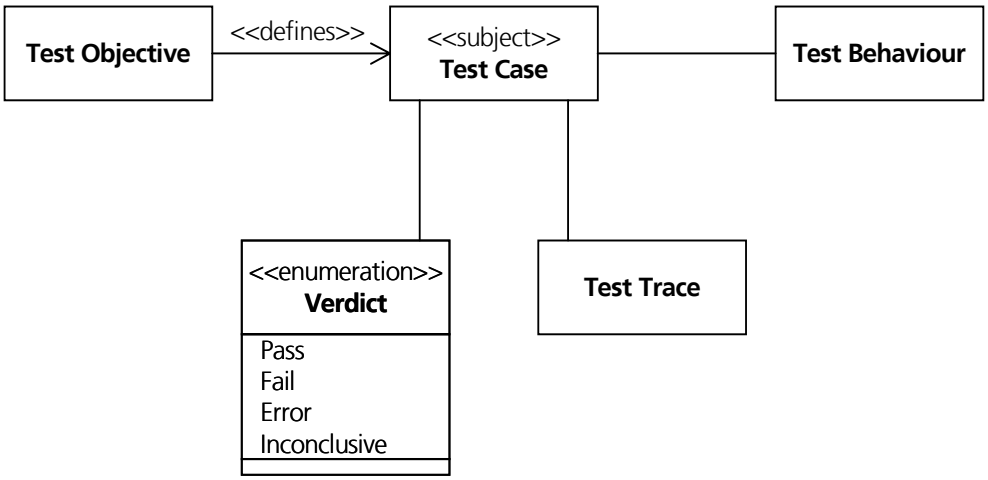



Figure 15: Concepts associated with a test case.

The behaviour of a test case comprises a test stimulus that sends the test data to the SUT in order to control it, and the test observation that represents the reactions of the SUT to the sent stimulus. The assessment of the SUT's reactions to a stimulus is performed by a validation action. Its outcome is the verdict for the test case. A pass verdict indicates that the SUT adheres to its expectations, a fail verdict indicates that the SUT differs from its expectations, an inconclusive verdict means that neither pass nor fail can be assigned, and the error verdict indicates an error in the testing system. The test behaviour is summarized in Figure 16.

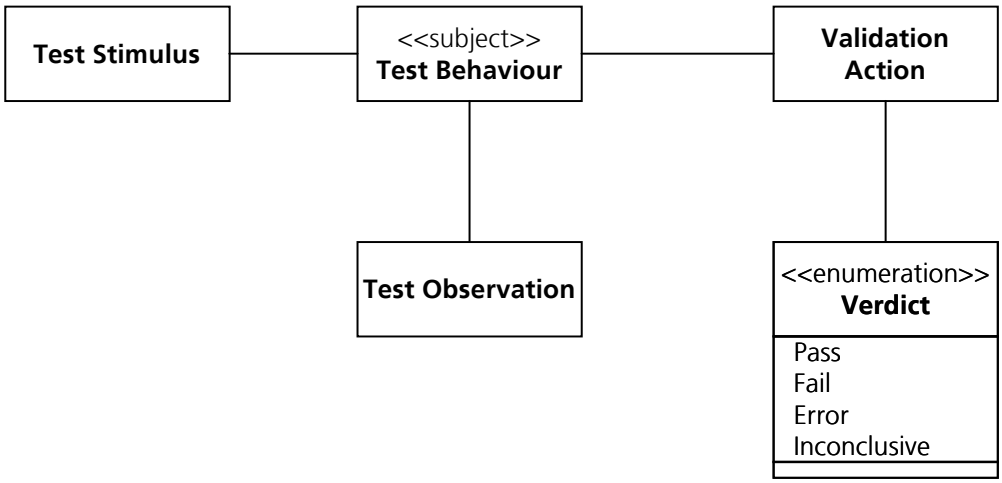
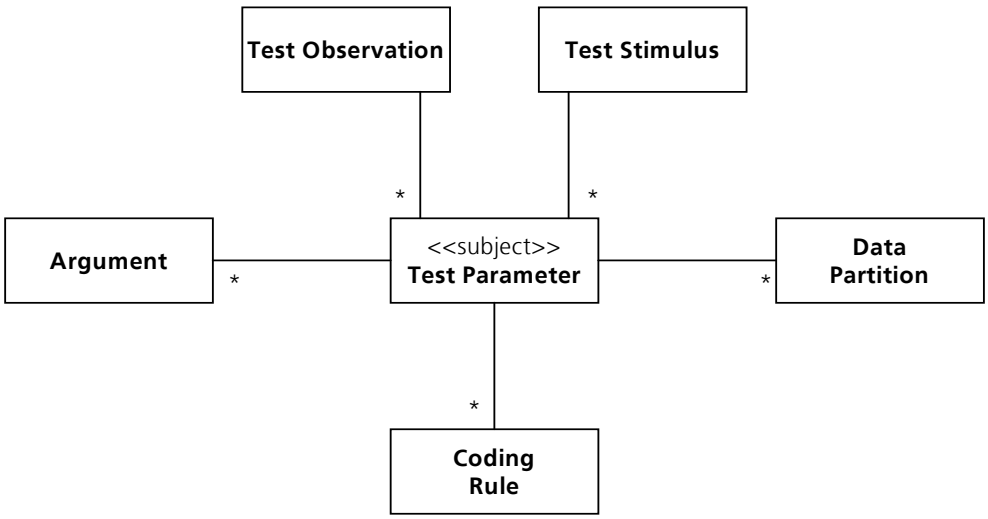



Figure 16: Concepts of the behaviour of a test case.

Stimuli that are sent to an SUT and observations that are received from an SUT represent the test data of a test case. They are referred to as test parameter. A test parameter may comprise any combination of arguments, data partitions, or

coding rules. An argument is a concrete physical value of a test parameter, and a data partition is a logical value of a test parameter such as an equivalence class of valid arguments. Coding rules are required if the interface of the SUT is based upon distinct encodings, e.g. XML, that must be respected by the testing system. Figure 17 gives an overview on the test data associations in the testing profile.

Figure 17: Test data concepts in the testing profile.




The previous sections have briefly introduced the concepts of the UML Testing Profile. In the following we look at how the UML and the testing profile are related.

## 4.3 Mapping to UML Testing Profile Concepts

Test Objective

The test objective is a general description of what should be tested. In the most general sense. For example the test objective that can be derived from a behavioral model is clearly the test of behavior. Since this is too broad a terminology, and a test objective is associated with a test case in the testing profile we can be more specific and identify a number of test objectives each of which maps to one or more test cases. The following items represent test objectives that can be associated with a state model:

– Piecewise Coverage: Piecewise coverage concentrates on exercising distinct specification pieces, for example coverage of all states, all events, or all actions. These techniques are not directly related to the structure of the underlying state machine that implements the behavior, so it is only 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 transi-tion 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].
- 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].

In the same way, we have can define the test of structure as a test objective that may be associated with a strucutral model. Although we would have to define more specifically what we mean by "test of structure".

Test Stimulus

A transition maps to a test stimuls. A stimulus is some transaction that is carried out on the tested object. A stimulus maps to a single operation invocation, or a series of operation invocations or events on a tested instance. A stimulus represents the input that is going to the tested instance. Sometimes, a distinct stimulus may only be sent to a tested object if some conditions or constraints hold (e.g. a distinct state). Such a state is also referred to as a pre-condition for sending the stimulus and it typically involves the execution of some previous stimuli in order to get an object into that distinct state. A state in the state model can map to a sequence of stimuli, and it consequently represents a history of transitions. A test stimulus can also comprise more. For example in built-in contract testing a stimulus may be considered, additionally to the call of the actual tested operation, the sequence of testing interface invocations in order to bring a component into a state from which a test will be performed.

Test Observation

A state represents part of a test observation. This is the final state after a test event has been executed. The test observation represents the output data from the tested instance. For example, an action expression maps to an observation, the return value of an event maps to an observation.

Verdict

The verdict is the assessment of whether the tested instance is correct or not. The evaluation of the test observation maps to a verdict for a single test stimulus. For this of course we need some sort of an oracle that defines the correct expected outcome of a test.

Test Case

A test case subsumes the previous items and compounds them into a single concept. It may comprise a test objective, a number of stimuli and observations as well as a verdict.

# 5 Summary and Conclusion

This report is motivated through the testing panel of the 5th International Conference on the Unified Modeling Language (UML 2002) that was focusing on the question of whether the UML and Testing may be a perfect fit. As it turns out, the UML and Testing are indeed a perfect fit and supplement each other in the two single most important respects: the UML as basis for the generation of testing artifacts, and the UML for the specification, design and modeling of test software. The first one sees the UML as the basis from which all testing for a software project is derived. Chapter 3 outlines which testing artifacts may be derived from which UML diagrams. It is important to note that testing which is traditionally perceived as a late activity in the development life-cycle, can also be developed on the basis of high-level specification and realization documents. One of the major outcomes of this work is that the testing for a system can be specified and designed in tandem with its normal functionality, and most notably, this can be performed at all decomposition levels and at all abstraction levels during development. As soon as more and more information about the workings of a system becomes available throughout the entire development effort, we can use that information and make the testing of the system more and more concrete. This way of developing the system and the testing of the system in parallel distributes the testing effort throughout the entire development cycle, instead of simply adding testing as a seperate effort after implementation, as it is the case in most traditional software development approaches. The system and the testing of the system are always in the same stage in terms of abstraction and decomposition. The testing and thus the test execution will therefore become available as soon as the final system becomes available and ready to execute. The second one simply adds concepts that are specific to testing to the core UML. This alleviates test development considerably, because the designers of the test software can simply use these concepts as basic building blocks. This is the case for all design patterns, not only for the ones related to testing. The UML Testing Profile was briefly summarized in Chapter 4. Overall, this report is a strong advocator for system modeling with the UML, because it is not merely a convenient and easy-to-learn notation for the design and specification of systems, but additionally it supports the quality assurance efforts throughout the system's life-cycle in a convenient way.

# 6 References


[AM01] Amyot, D., Mussbacher, G., Bridging the Requirements/Design Gap in Dynamic Systems with Use Case Maps, Tutorial, 23rd Int. Conf. on Software Engineering (ICSE-2001), Toronto, Canada, May 12-19, 2001.

[AM00] Amyot, D., Mussbacher, G., On the Extension of UML with Use Case Map Concepts, 3rd Int. Conf. on Unified Modeling Language (UML-2000), York, UK, September 2000.

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

[Bei90] Beizer, B., Software Testing Techniques. Van Nostrand Reinhold, New York, 1990.

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

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

[Bok01] Bokhorst, L.(ed.), Requirements Specification Description Template, ITEA DESS Project (Software Development Process for Real-Time Embedded Software Systems), Deliverable D.1.8.4, November, 2001.

[Coc96] Cockburn, A., Basic Use Case Template, Technical Report TR.96.03a, Human and Technology, Salt Lake City, April 1996.

[Coc01] Cockburn, A., Writing Effective Use Cases, Addison-Wesley, Boston, 2001.

[Comp+] EU-IST Component+ Project. http://www.component-plus.org.

[Gro02a] Gross, H.-G., Component+ Methodology -- Built-in Contract Testing: Technological Foundations. IESE Technical Report 073.02/E, Kaiserslautern, December 2002.

[Gro02b] Gross, H.-G., Component+ Methodology -- Built-in Contract Testing: Method and Process. IESE Technical Report 030.02/E, Kaiserslautern, October 2002.

[Jac92] Jacobson, I., et al., Object-oriented software engineering. Addison-Wesley, Reading, Mass., 1992.

[Mar95] Marick, B., The Craft of Software Testing: Subsystem testing. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[MDTS] BMBF Project MDTS: Model-driven Develpoment of Telecommunication Systems, http://www.fokus.fhg.de/mdts.

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

[PW99] Probert, R., Williams, A.W., Fast Functional Test Generation using an SDL Model, 12th Int. Workshop on Testing of Communicating Systems (IWTCS'99), Budapest, Hungary, September 1-3, 1999.

[RG99] Ryser, J., Glinz, M., A practical approach to validating and testing software systems using scenarios. Quality Week Europe, Brussels, 1999.

[RG00] Ryser, J., Glinz, M., SCENT: A method employing scenarios to systematically deriving test cases for system test. University of Zürich, Institute of Informatics, Technical Report, 03/2000.

[RG00] Ryser, J., Glinz, M., Using dependency charts to improve scenario-based testing. 17th International Conference on Testing Computer Software (TCS'2000), Washington, 2000.

[RR98] Regnell, B., Runeson, P., Combining scenario-based requirements with static verification and dynamic testing. 4th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'98), Pisa, Italy, June 8-9, 1998.

[RRW98] Regnell, B., Runeson, P., Wohlin, C., Towards integration of use case modeling and usage-based testing. 4th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'98), Pisa, Italy, June 8-9, 1998.

[War64] Warner, C.D., Evaluation of program testing. TR 00.1173, IBM Data Systems Division Development Laboratories, Poughkeepsie, N.Y., July 1964.

# Document Information

| | |
|---|---|
| Title: | Testing and the UML - A perfect Fit. |
| Date: | October 31, 2003 |
| Report: | IESE-110.03/E |
| Status: | Final |
| Distribution: | Public |