



**UNIVERSITÄT PADERBORN**  
*Die Universität der Informationsgesellschaft*

Faculty of Computer Science, Electrical Engineering and  
Mathematics

PhD Thesis

**Integrating Contract-based Testing into Model-driven  
Software Development**

by Barış Güldali

A dissertation submitted to the  
Faculty of Computer Science, Electrical Engineering, and Mathematics of  
the University of Paderborn

Supervisors:

Prof. Dr. rer. nat. Gregor Engels  
(University of Paderborn)

Prof. Dr. rer. nat. Mario Winter  
(Cologne University of Applied Sciences)

Paderborn, June 3th, 2014



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Model-based Software Development . . . . .	4
1.3	Model-based Testing . . . . .	6
1.4	Problem Statement . . . . .	7
1.5	Solution . . . . .	9
<b>I</b>	<b>Foundations and Related Work</b>	<b>15</b>
<b>2</b>	<b>Fundamentals of Model-based Testing</b>	<b>17</b>
2.1	Software Development Methodology . . . . .	17
2.2	Model-based Software Development . . . . .	18
2.2.1	Modeling . . . . .	20
2.2.2	Model Transformations . . . . .	20
2.3	Testing in Development Process . . . . .	25
2.3.1	Test Levels . . . . .	25
2.3.2	Test Activities . . . . .	28
2.3.3	Test Automation Techniques . . . . .	33
2.4	Model-based Testing . . . . .	35
2.4.1	MBT Process . . . . .	37
2.4.2	Different Approaches . . . . .	39
2.4.3	Different Paradigms . . . . .	42
<b>3</b>	<b>Contract-based Testing</b>	<b>47</b>
3.1	Characteristics . . . . .	47
3.1.1	Reference Model for Contract Modeling . . . . .	48
3.1.2	Reference Model for Testing with Contracts . . . . .	49
3.2	Approaches in the Literature . . . . .	51
3.2.1	AutoTest . . . . .	54
3.2.2	Korat . . . . .	56

3.2.3	WeSUF . . . . .	56
3.2.4	LTG/B . . . . .	57
3.2.5	WSTVC . . . . .	58
3.3	Tabular Comparison of Approaches . . . . .	59
3.4	Summary . . . . .	61
<b>4</b>	<b>Visual Contracts</b>	<b>63</b>
4.1	Modeling with Visual Contracts . . . . .	63
4.1.1	Running Example: Online Shop . . . . .	65
4.1.2	Visual Contracts . . . . .	67
4.1.3	Semantics of Visual Contracts . . . . .	69
4.2	Application Areas . . . . .	72
4.3	Experiences with Visual Contracts . . . . .	73
<b>5</b>	<b>Summary of Part I</b>	<b>77</b>
5.1	Improvement Potential in CBT Approaches . . . . .	77
5.2	Requirements on a novel Testing Approach . . . . .	81
<b>II</b>	<b>Approach</b>	<b>85</b>
<b>6</b>	<b>General Approach</b>	<b>87</b>
6.1	Development Process Overview . . . . .	88
6.2	Implementation . . . . .	92
6.3	Test Design . . . . .	94
6.4	Test Implementation . . . . .	96
6.5	Test Execution . . . . .	98
6.6	Summary . . . . .	101
<b>7</b>	<b>Unit Testing</b>	<b>103</b>
7.1	Development Scenario . . . . .	104
7.2	Test Design . . . . .	107
7.2.1	Approach 1: Artificial Prestate . . . . .	108
7.2.2	Approach 2: Natural Prestate . . . . .	116
7.3	Test Implementation . . . . .	125
7.4	Test Execution . . . . .	126
<b>8</b>	<b>Integration Testing</b>	<b>129</b>
8.1	Development Scenario . . . . .	130
8.2	Test Design . . . . .	133
8.3	Test Implementation and Execution . . . . .	136

<b>9</b>	<b>System Testing</b>	<b>143</b>
9.1	Development Scenario . . . . .	144
9.2	Test Design . . . . .	150
9.3	Test Implementation . . . . .	154
9.4	Test Execution . . . . .	158
<b>10</b>	<b>Tool Support and Evaluation</b>	<b>161</b>
10.1	Unit Testing . . . . .	162
10.1.1	Approach 1: Artificial Prestate . . . . .	164
10.1.2	Approach 2: Natural Prestate . . . . .	169
10.2	Integration Testing . . . . .	175
10.3	System Testing . . . . .	177
10.4	Evaluation . . . . .	181
10.4.1	Evaluation by interviews and case studies . . . . .	181
10.4.2	Evaluation by quantitative methods . . . . .	182
10.4.3	Evaluation by peer reviews . . . . .	184
<b>III</b>	<b>Closure</b>	<b>187</b>
<b>11</b>	<b>Conclusion and Future Work</b>	<b>189</b>
11.1	Contributions . . . . .	191
11.2	Epilogue . . . . .	195
11.3	Roadmap for further Research . . . . .	197
11.3.1	Further selection criteria for test data . . . . .	197
11.3.2	Binding real test data . . . . .	197
11.3.3	Agile development . . . . .	199
	<b>Bibliography</b>	<b>200</b>



# Abstract

Model-based testing (MBT) aims at improving the manual test design process by using test models for automated test case generation, which is systematic and efficient. However, MBT is not for free: test models must be created and maintained; tools and techniques for test case generation and execution are required. Furthermore, testing activities must be integrated into the model-driven development process. For a seamless integration of the development and test processes, models should be interchangeable between developers and testers at each stage of the development process. This poses challenges for both developers and testers in their modeling, implementing and testing activities. Most of the existing approaches propose using detailed and complete models for development and testing, which requires advanced modeling skills and tools. Experts agree that these challenges are the main reasons why model-driven development is still not there it needs to be. As a reaction, a former work proposed using Visual Contracts for a light-weight and semi-automated development process. However, the questions regarding testing and its integration remained unanswered.

In this thesis, we fill this gap and extend the proposed development process by a Visual Contract-based testing process. Our approach proposes using Visual Contracts as a test basis for an automated test process. Thereby, test cases and test scripts are systematically derived from Visual Contracts using formal selection criteria. The Visual Contract language follows the Design-by-Contract paradigm resulting in intuitive test models specifying precondition and postconditions for test object operations. Visual Contracts allow creating partial models enabling starting testing activities before all implementation details are known. The originality of our approach compared to other contract-based approaches lies in its support of all test levels during the model-driven development process such that a complete integration of development and testing processes is given. Furthermore, we apply many novel techniques and tools (e.g. model transformations, model checking, code generation, model-driven monitoring) for realizing the testing activities resulting in a sophisticated test process.





# Zusammenfassung

Modellbasiertes Testen (MBT) hat das Ziel den manuellen Testentwurfsprozess durch den Einsatz von Testmodellen zur automatischen Testfallgenerierung zu verbessern. Dadurch wird der Prozess effizienter und systematischer. Allerdings MBT bedeutet erst ein Mal investieren: Testmodelle müssen erstellt und gewartet werden, Werkzeuge für die Generierung und Ausführung müssen entwickelt werden. Ausserdem müssen diese neuen Techniken und Werkzeuge in den modellbasierten Entwicklungsprozess (MBE) integriert werden. Für eine nahtlose Integration müssen die Modelle zur jedem Zeitpunkt der Entwicklung austauschbar unter den Entwicklern und den Testern sein. Diese Anforderung bedeutet neue Herausforderungen für die Beteiligten bei ihren Entwicklungs- und Testaktivitäten. Viele der aktuellen MBT- und MBE-Ansätze erfordern den Einsatz von detaillierten und vollständigen Modellen, was fortgeschrittene Fähigkeiten von Entwicklern und Testern notwendig macht. Experten sind sich einig, dass genau diese neuen Herausforderungen die Entwicklung und die Verbreitung von modellbasierten Techniken erschwert. Als eine Reaktion zu diesen Entwicklungen wurde die Softwareentwicklung mit den Visuellen Kontrakten als eine leichtgewichtige und teil-automatisierte Technik eingeführt. Allerdings blieben viele Fragen bzgl. des Testens unbeantwortet.

In dieser Dissertation füllen diese Lücke zwischen der modellbasierten Entwicklung mittels visuellen Kontrakten und dem modellbasierten Testen. Dabei verwenden wir die Visuellen Kontrakte als Basis für den automatisierten Testfallentwurf, wo die Testfälle und die Testskripte mittels formalen Auswahlkriterien von den Visuellen Kontrakten abgeleitet werden. Unser Ansatz setzt auf das Design-by-Contract-Paradigma und spezifiziert intuitive Testmodelle mittels Vor- und Nachbedingungen an Testobjekten. Die Testmodelle dürfen unvollständig sein und erlauben dadurch einen früheren Start mit den Testaktivitäten, ohne dass alle Einzelheiten der Implementierung bekannt sein müssen. Der Beitrag unseres Ansatzes im Vergleich zu anderen kontraktbasierten Testansätzen ist es, dass wir im Gegensatz zu anderen Ansätzen alle Teststufen behandeln, was die Integration der Entwicklungs-

und Testaktivitäten zu jeder Zeit möglich macht. Dabei setzen wir viele innovative Software Engineering-Techniken (z.B. Modelltransformationen, Modelchecking, Code-Generierung, Model-driven Monitoring), um diese Integration zu ermöglichen.

# Chapter 1

## Introduction

Manual testing is an expensive and error-prone task. Test automation can help in making the test process more efficient and more precise. In the modern times of model-based software development, model-based testing proposes using models for supporting test automation. Thereby, test cases or test scripts are generated from models. In the literature of model-based testing, many paradigms and notations are proposed for creating models. For adapting model-based testing in the software development process, it is important to select appropriate paradigms and notation addressing the needs of the development context. Design-by-Contract (DbC) is a well-known paradigm for specifying software. In this thesis, we discover the embodiment of DbC paradigm and appropriate contract-based notations into model-based testing (MBT). In this chapter, we roughly describe the context of model-based testing and motivate the need for a novel approach for contract-based testing (CBT).

### 1.1 Motivation

The expectations of users on high quality software are growing. They are expecting more functionality in shorter version-cycles for less cost. Besides functionality, they also expect good non-functional qualities, like high performance, easiness in usability etc. Thus, both functional and non-functional qualities [ISO01] have a direct influence on the success of software. In addition to the end user's expectations, also rapid changes in market requirements and in legal issues make the development of software more and more complex.

For assuring the requirements on high quality software, constructive and analytical activities are required throughout the development process. While the constructive activities aim at building the software correctly, analytical

activities aim at checking the correctness of the software with respect to a specification after or during the construction. Process models for software development define how these activities are to be organized. Depending on the development context different process models can be applied, e.g. RUP [Kru99], V-model [Boe84], Scrum [SB01].

The V-model is the most traditional process model which defines the relation between constructive and analytical activities. Figure 1.1 illustrates the V-model<sup>1</sup> based on Basili and Pezze [BP06]. In V-model, the development process starts with capturing the requirements on software and ends with the deliverable software. In-between, design and implementation constitute the constructive activities and testing constitutes the analytical activity. During the design, which is illustrated on the left branch, the requirements on software are translated into a specification. During the implementation, which is illustrated in the right branch of V-model, the software is coded with respect to its specification. Then, in order to assure the correct implementation, the software is tested against its specification by means of test cases. If errors in the software are detected, they have to be corrected. If the desired quality of software is reached, it can be delivered.

As the complexity of software to be developed grows, also the complexity of the development activities increases. This leads to more effort and to more costs in software development, not only for the constructive part but also for the analytical part. A current analysis of development efforts in [SS10] show that testing can make a draft of up to 40% of total development costs. The main reason for these costs is that many testing tasks are done manually. Typical manual activities in testing are test case design, test execution and evaluation of the test results. The manual work is time-consuming and error-prone. Even if almost the half of the total development efforts is invested in testing, software can still contain undetected errors at its delivery. Depending on their severity, these errors can lead to failures in software systems which can harm human life or which can result in high financial loss. Thus appropriate testing techniques should be selected for effective quality assurance. The appropriateness of testing techniques strongly depends on the characteristics of the software under test and also on the characteristics of the development process [VB05].

In order to handle the complexity of software development, model-based techniques are proposed. Thereby, abstract models of software are used for capturing the requirements, for designing the software and for code gener-

---

<sup>1</sup>This illustration is slightly different than the illustration of Boehm [Boe84]. While Boehm places the constructive activities on the left branch and the analytical activities on the right branch, Basili and Pezze place the constructive activities (design and implementation) on the outer V, and testing on the inner V.

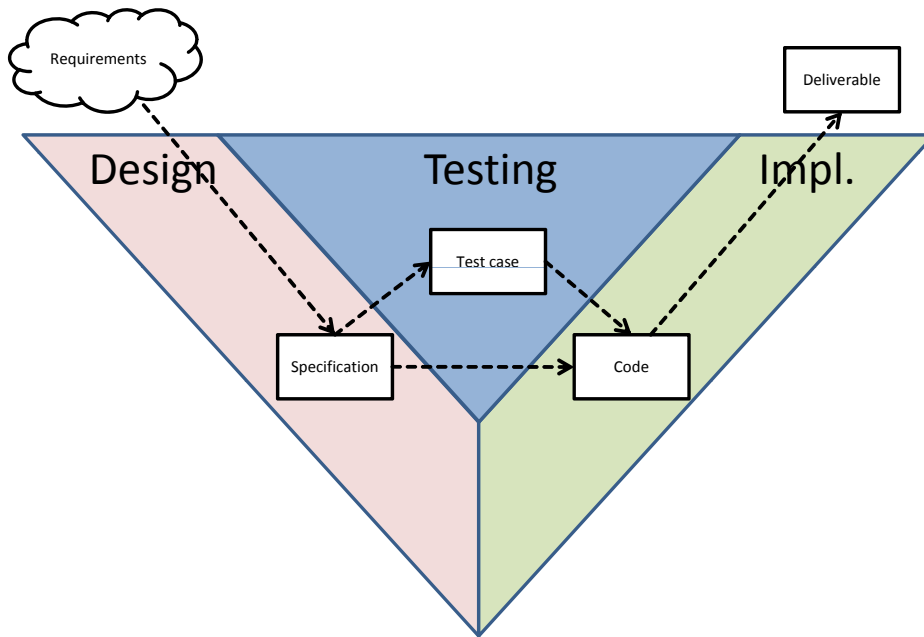


Figure 1.1: Activities and artifacts in V-model based on [BP06]

ation. In the context of model-based software development (MBSD), also testing activities are supported by models. Model-based testing (MBT) uses models for automating the manual tasks of testing, especially the test case design [UL07]. Thus, models play a central role in MBSD and MBT.

Dias-Neto et al. have identified in [DNSVT07] over 400 MBT approaches which address different kind of notations (e.g UML or FSM) and paradigms (e.g. state-based or event-based) for creating models. It is a challenge for developers and testers to select appropriate modeling notations and paradigms depending on many factors, e.g. the characteristics of the software under consideration, the skills of the developers and testers, compatibility with existing tool and notations [ESS08].

At the University of Paderborn, we also deal with various modeling notations and paradigms in the context of our research on MBSD. At the software engineering group of Prof. Engels, one of the main research topics is the MBSD with UML. Some of the research topics are: defining more formal semantics for UML [Küs04, Hau05, Sch09], defining UML profiles for domain-specific modeling [Mlynarski, Sauer] and defining new UML-based languages which extend UML [Loh06, För09]. In this context, Lohmann has developed in 2006 a new language *Visual Contracts* [Loh06] which extends the UML by the concept of Design-by-Contract (DbC) [Mey92]. He has shown that Visual Contracts are useful for specifying component interfaces by using pre-

and postconditions. He has also shown, how Visual Contracts can be used for code generation and run-time monitoring which are means for assuring software quality.

In this dissertation thesis, we carry forward the MBSD research and show how MBT can be integrated into the MBSD process using Visual Contracts. Thereby, we show how Visual Contracts can be used for test case design, test execution and evaluating test result. However, the idea of using contract based notations for testing is not new. There are already approaches, which use the DbC paradigm for supporting testing tasks. In this thesis, we report on a literature survey on contract-based testing (CBT) and show how a novel testing technique using Visual Contracts can help in improving the flaws of existing approaches.

In the next sections of this introductory chapter, we give more insight into MBSD and MBT and discuss the role of DbC for constructive and analytical activities. Then, we identify the requirements on CBT for integration into MBSD. After addressing the flaws of existing techniques, we motivate our MBT approach with Visual Contracts.

## 1.2 Model-based Software Development

In order to handle the complexity of software, model-based software development (MBSD) proposes using abstract models of software during the development process. Models are used to describe the software itself or its environment. Thereby, they focus on most important aspects of software thus hiding the irrelevant details for a particular point of time during the development process. Software developers use models (a) for communication within the development team, (b) for documenting the customer requirements and the specifications, (c) for analysis of software quality and finally (d) for code generation. Especially, the need for a model-based code generation is supported by the following citation of Bill Gates [Ude05]:

“There’s only really one metric to me for future software development, which is - do you write less code to get the same thing done?” Bill Gates 2005

Following the vision of writing less code, the Object Management Group (OMG) launched its Model-Driven Architecture (MDA) [Gro03a] initiative as well as modeling standards such as the Unified Modeling Language (UML) [Gro03b] that provides the foundation for MDA. The main target of MDA is to drive the software development by specifying models instead of writing code. In MDA first the requirements on software are translated into abstract

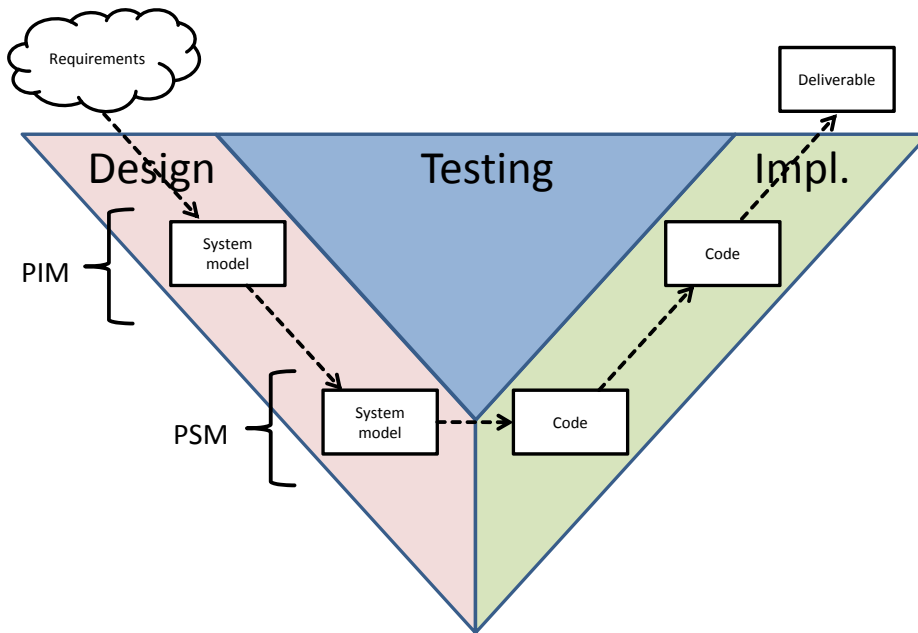


Figure 1.2: V-model with MDA concepts

models. Then, these models are stepwise refined with technical details until executable code arise.

Figure 1.2 shows the V-model with MDA concepts. The design phase begins with the creation of design models (also called *system models*) with respect to the requirements. These models arise in early phases of the development process and they do not address any technical details on the implementation or on the execution of software. That is why these models are called platform-independent models (PIM) and focus on only domain specific aspects. Then, the PIM's are stepwise refined to more specialized system models by adding technical details resulting in platform-specific models (PSM). After having fed PSM's with sufficient technical details, they can be transformed into program code. The refinement steps are conducted by Model Transformations (MT) [MG06]. OMG also defines a standard Query/View/Transformations (QVT) [Gro05b] for specifying model transformations. Using QVT and supporting tools the refinement steps can be automated. The program code can be further transformed to more complex code resulting in the fully-fledged deliverable software at the end.

However, Engels et. al formulates the problems of MDA as follows [ELSH06]: MDA is still in its infancy compared to its ambitious goals of having a (semi-)automatic, tool-supported stepwise refinement process from informal requirements specifications to complete running software . A lot of unre-

solved questions exist for modeling tasks as well as for automated model transformations. For instance, a complete understanding of the appropriate level of detail and abstraction of models is still missing. The notion of abstraction naturally conflicts with the desired automatic code generation from models as proposed by the MDA. To enable the latter, fairly complete and low-level models are needed.

Thus, in today's software development processes, manual programming by developers for realizing the requirements specified in abstract models is still indispensable. However, manual programming is error-prone. The more manual programming is conducted, the more errors can be inserted into the software. Before delivery of software, these errors must be detected by dynamic testing and be eliminated.

### 1.3 Model-based Testing

Traditional testing is mostly a manual activity and it is *expensive*. Thereby, an appropriate set of test cases has to be selected and the software under test has to be executed using these test cases. During the execution of each test case, the observed behavior of software is compared to the expected behavior which is defined in the specification. Moreover, often manual testing is *not systematic* and depends on the skills and experience of testers; such that the selection and the execution of test cases, and the evaluation of test results can suffer from the subjectiveness of testers. This hinders the reproducibility of the selection of test cases and their execution.

In order to handle high efforts caused by the manual testing, numerous automation techniques are proposed in the literature, e.g. capture-replay testing, script-based testing, keyword-driven testing. Depending on the needs and the maturity of the development and the testing process one or more testing techniques can be applied. Using these techniques, the execution of test cases can be easily automated. However, automating the test execution only does not solve all problems of manual testing. Fewster said in [FG99]:

“It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing. Automating chaos just gives faster chaos.” Fewster 1999

Thus, before the test execution is automated, the test case design should be conducted in a systematic way. Also the evaluation of the test results remains a challenge. These activities require the intelligence of tester to decide which test cases are important and whether the software behaves correctly or not.



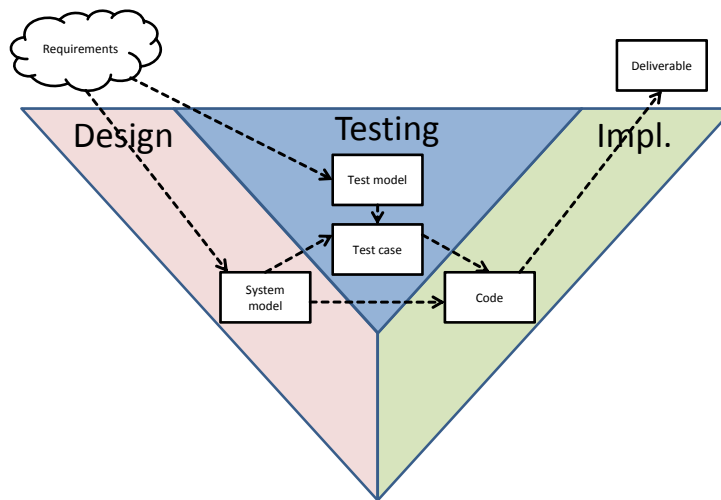


Figure 1.3: V-model with MBT concepts

In order to cope with these challenges, MBT utilizes formal models of the software under test for systematically deriving test cases and evaluating test results. Thereby, either system models, which are created during the design phase, can be used for MBT purposes, or testers can explicitly create their own models with respect to the initial requirements. We will call the later ones as *test models*. Figure 1.3 shows both scenarios on the V-model (cf. Figure 1.2). The derivation can be either manual or automated, however it must be systematic, i.e. a formal *test selection criteria* has to be defined. For automating the test case derivation, the models should be specified in a machine readable notation. In MBSD, many notations (e.g. UML, FSM, Petri nets, B or Z) are used for modeling different aspects of the software [UPL06].

## 1.4 Problem Statement

In MBSD and MBT, the selection of a suitable modeling notation depends strongly on the characteristics of the software under test and also on the context of software development. For example, in [UL07] Utting and Legeard propose using transition-based notations if control-oriented software is modeled and using pre/post-based notations if data-oriented software is modeled. If separate test models are to be created, testers can decide on the modeling notation by themselves, however, if system models are to be used for testing, testers and developers have to agree on common notations. For the selection of suitable notations, it is also important to clarify other aspects, e.g. which

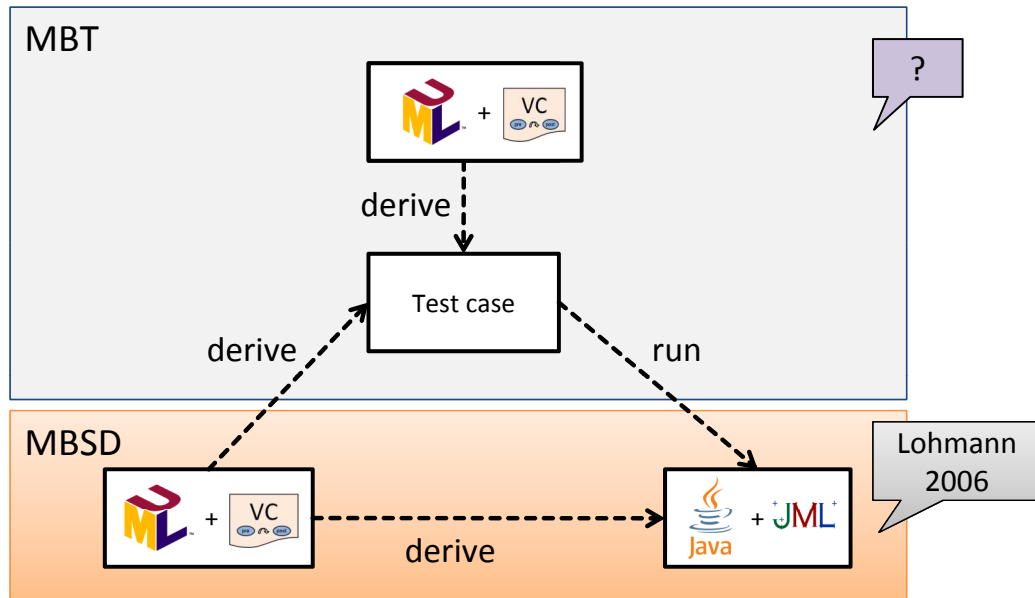


Figure 1.4: MBSD and MBT process using Visual Contracts

skills do team members have, which modeling notations and tools are already in use [ESS08]. Thus, integrating the MBT process into the MBSD process has to be planned carefully.

At the research group of Prof. Engels, we develop languages and methods for UML-based software engineering. In the previous years “Visual Contracts” (VC) has been developed by Lohmann [Loh06], which integrates the notion of Design-by-Contract (DbC) [Mey92] into UML-based modeling. Using VC the behavior of software is modeled in terms of pre- and postconditions on system’s state which is typed over a UML Class diagram. Thereby, VC specify conditions which must be fulfilled by any client of the software (*preconditions*) and conditions that must be fulfilled by the software if it has been executed (*postconditions*).

Lohmann has shown that VC is a useful and light-weight notation for specifying software interfaces [Loh06]. He has also shown how these specifications can be used for MBSD (see Figure 1.4, lower box) where parts of the Java program code are automatically generated. Missing parts of software are programmed manually which can insert errors into the software. For checking the correctness of software at runtime, he has developed the technique model-driven monitoring (MDM) where software behavior is monitored by assertions in Java Modeling Language (JML) [LC03] which are generated from VC and check the conformance of software behavior with the VC specification at runtime. If inconformities are detected, the software reports these

to the caller which is then responsible for further actions. However, the target of the quality assurance should be that errors should be detected during the test process before the software is delivered. For that, the software has to be executed with controlled test cases which systematically cover different scenarios for execution of software under test. The design of such test cases requires a systematic and efficient test process, which is integrated into the VC-based MBSD process.

In this dissertation, we extend the VC-based MBSD depicted in Figure 1.4 by a MBT process which uses VC as a source for test cases. Thereby, we generate test cases from VC, where test inputs are derived from preconditions and postconditions represent the expected behavior of software. The test inputs are object constellations as proposed by Winter in [Win99]. Testing with VC combines the DbC paradigm with MBT techniques.

However, the idea of combining DbC and testing is not new. There are already some approaches (e.g [Ciu08, MSM<sup>+</sup>07, Gro05a, Ave10]), which use the DbC paradigm and pre/post notations (e.g. Eiffel, JML, OCL, Spec#, B) for supporting testing tasks like test case generation or test case execution. Therefore, we have conducted a literature survey on the so called “contract-based testing (CBT)” and studied these approaches for conformance with the MBSD process depicted in Figure 1.4. We have identified the following flaws in these approaches:

- Most of the proposed notations are very low-level, such that a seamless integration into UML is not possible.
- Testers must have programming skills for creating low-level contracts. However, testers need light-weight notations for creating and maintaining test models.
- Most of the approaches address the unit testing level, however, during the development process also higher test levels, i.e. integration testing and system testing, must be supported by MBT.
- Most of the approaches use artificial states for invoking the software under test which hinders realistic test scenarios.

## 1.5 Solution

In this thesis, we show how a novel MBT approach using the Visual Contracts can help in handling the flaws of current CBT approaches for integrating DbC into MBT. We use VC for CBT on three testing levels: unit

testing, integration testing and system testing. We call our approach Visual Contracts-based Testing (VCBT).

In unit testing (cf. Figure 1.5), we generate test cases from the VC which are used by the developers as a specification as explained by Lohmann [Loh06]. We generate test cases from the preconditions which have two parts: test inputs and prestate. The prestate is used for setting the object of the class under test into a controlled state which fulfills the precondition of the operation under test. After setting the prestate, the operation is invoked using the test inputs. During the test execution, the JML assertions generated from the VC and embedded into the Java code check the conformance of the poststate to the postcondition of VC. If this is operation the case, then test has been passed. For automating this process, we generate JUnit test scripts which implement the scenario explained above [Ell08, EEG08].

The above mentioned test technique extracts an artificial prestate directly from the precondition of the operation under test. Since the precondition is partial, the computed prestate can be incomplete for a realistic test. In order to test with a more realistic prestate, the prestate should be set by a sequence of class operations. For computing an operation sequence, we consider the VC of all operations in the subsystem which is tested on this test level. Computing a state transition system, which represents all possible interactions within the operations, and using reachability analysis, we compute an operation sequence which should set a subsystem into a controlled prestate [EGL06a]. After setting the prestate in a more realistic way, the operation under test can be invoked similar to the unit testing with the artificial prestate.

During unit testing, the classes are tested in isolation. In integration testing (cf. Figure 1.6), we are interested in the interaction of many classes and their operations which constitute a subsystem. Thereby the caller and callee classes must be tested and stepwise integrated to build the subsystem. During the test process of a caller class, if a callee class is not implemented yet, it must be simulated using stubs. Because the simulation is a costly activity, the integration test must be planned well by considering the dependencies between the classes. In order to set up an efficient integration test process, we use topological sorting for planning the test execution and integration sequence of classes of a subsystem under test. A specialty in integration testing is the monitoring of subsystem interfaces for checking the the correctness of data interchange. For that we instrument the embedded assertions generated by Visual contracts.

During the unit testing and the integration testing, we use the VC which was created by developers as a system specification. Thereby, errors in the manual programming can be found with respect to the VC. However, there

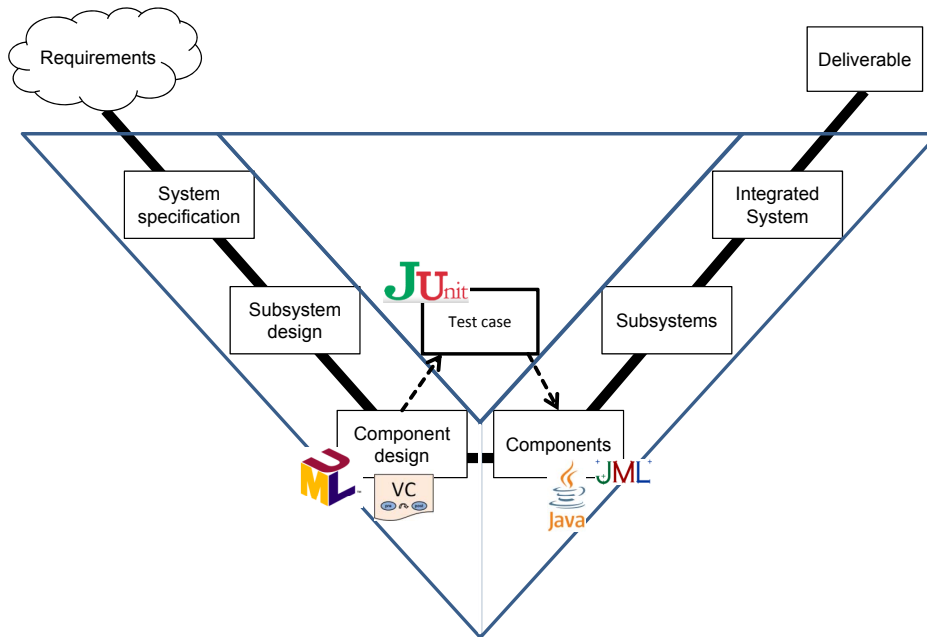


Figure 1.5: Unit testing using visual contracts

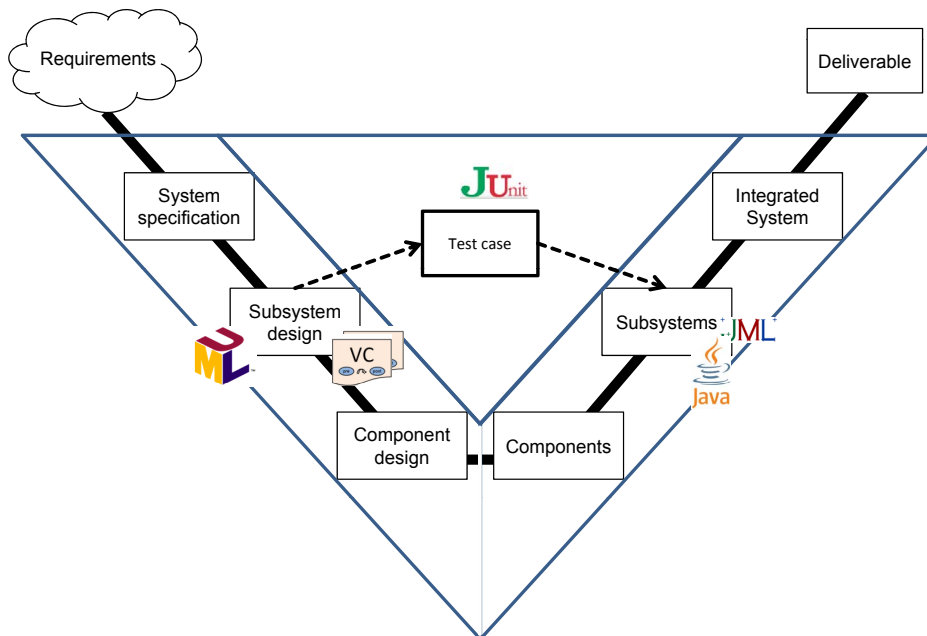


Figure 1.6: Integration testing using visual contracts

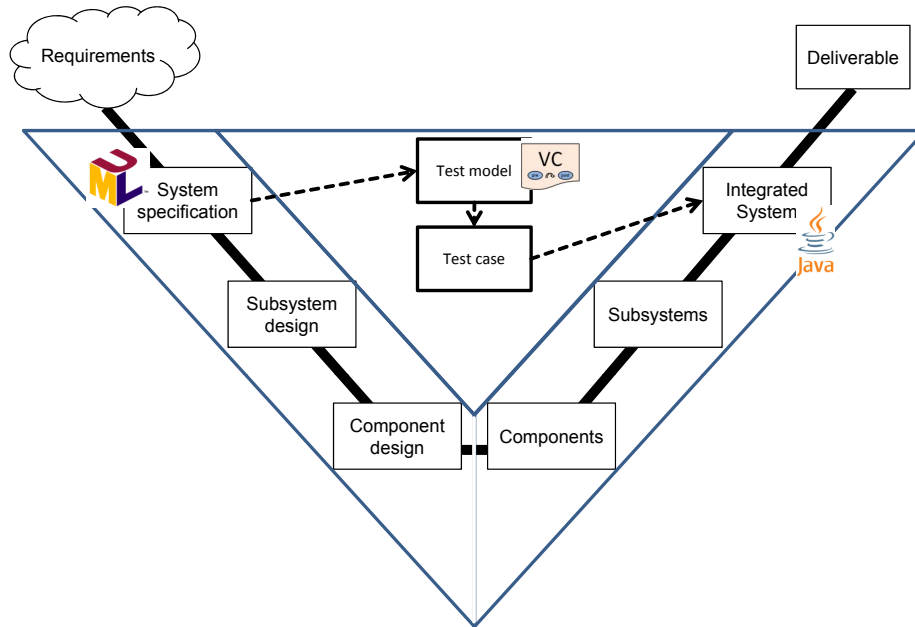


Figure 1.7: System testing using visual contracts

can also be errors in the VC which stem from erroneous interpretations of initial requirements on software. In order to prevent such errors, testers can directly derive test models from the initial requirements. In the level of system testing, we show how this can be done using the VC (cf. Figure 1.7). In this level, use-case specifications are partially formalized by VC, where the pre- and postconditions of the use-case are formalized by the pre- and postconditions of VC respectively. From the VC, test cases are generated using similar techniques as in unit testing and integration testing. However, the test cases have a higher abstraction than the test cases from the unit testing and integration testing. Because the use-cases, which are the basis for test cases, emerge in early phases of the development process, they contain abstract requirements on the software and thus the test cases are also abstract. Developers refine these use-cases stepwise until executable software evolves. In order to be able to execute the abstract test cases on executable code, test cases must also be concretized. For that, we use the documentation of design decisions of developers for transforming the abstract test cases into concrete test cases [GMWE09].

Using the techniques sketched above, we aim at completing the VC-based MBSD process and resolving the flaws of the existing CBT approaches. In the next chapters, we give more details on the context of our approach and on testing activities on each level. The rest of this thesis is structured as

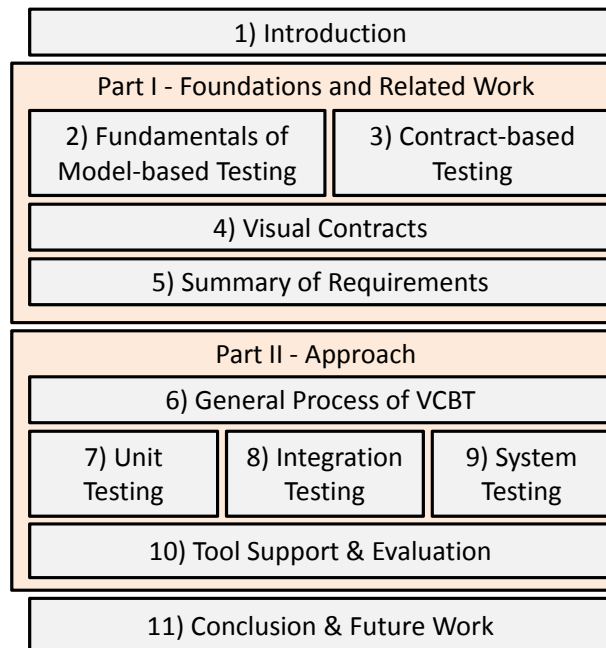


Figure 1.8: Outline of the dissertation

follows (Figure 1.8):

- Part I gives an overview on the foundations of MBT, CBT and VC and identifies the requirements for a novel testing approach.
  - Chapter 2 summarizes existing work on test automation and discusses the need for MBT. It also characterizes different approaches in MBT.
  - Chapter 3 introduces CBT as a sub-category of MBT. The related work on CBT is studied and their appropriateness for MBSD is discussed. Then, the need for a novel CBT approach using VC is motivated.
  - Chapter 4 summarizes the work of Lohmann on VC which is the modeling language for our approach. The intuitive and the formal semantics of VC are defined. An overview on the applications of VC is given.
  - Chapter 5 summarizes the requirements on a novel CBT approach for being integrated into MBSD.
- Part II introduces the Visual Contracts-based Testing (VCBT) approach for different test levels, the tool support prototypically devel-

oped during the dissertation and an evaluation which serves as a proof-of-concept.

- Chapter 6 describes the general MBT process using VC. Thereby, the MBT scenario, the fault model and the test selection criteria are presented in general.
- Chapters 7-9 introduces the different test levels for which the general test process is instantiated. The software under test on these levels are characterized and the challenges to be handled are addressed.
- Chapter 10 introduces the prototype tool support for the approach. An evaluation is presented which serves as a proof-of-concept for the approach.
- Chapter 11 concludes the thesis by discussing the fulfillment of the requirements and showing future directions for further research.



# Part I

## Foundations and Related Work



# Chapter 2

## Fundamentals of Model-based Testing

In Chapter 1 we have motivated the need for a novel approach of Contract-based testing embedded into the UML-based software development process. The basic idea is to use Visual Contracts not only for development purposes but also for testing purposes. Thus, our research bases on three main fundamentals: (1) Model-based software development and model-based testing, (2) Contract-based testing as a specialization of MBT, and (3) Visual contracts as modeling language. Part I of this thesis explains these fundamentals.

In this first chapter, we explain the role of model-based testing in the model-based development process. First, we give an overview on how testing activities are embedded into the development process and what are the challenges for testers in general. Section 2.4 explains then what is new in model-based testing (MBT) in comparison to classical testing techniques. Finally, we introduce a systematic comparison criterion in order to be able to characterize our new testing approach.

### 2.1 Software Development Methodology

Engels et al. state in [ESS08] that typically software developers have a good understanding of development tools they are using every day. However, a common understanding of the development methodology is mostly lacking among the development team. In order to establish a well-understood development methodology, the authors propose a step-by-step approach to define the elements of the development process (see Figure 2.1). Thereby, method engineers incl. development leaders and quality assurance leaders, should first define the software engineering concepts they want to realize. Then

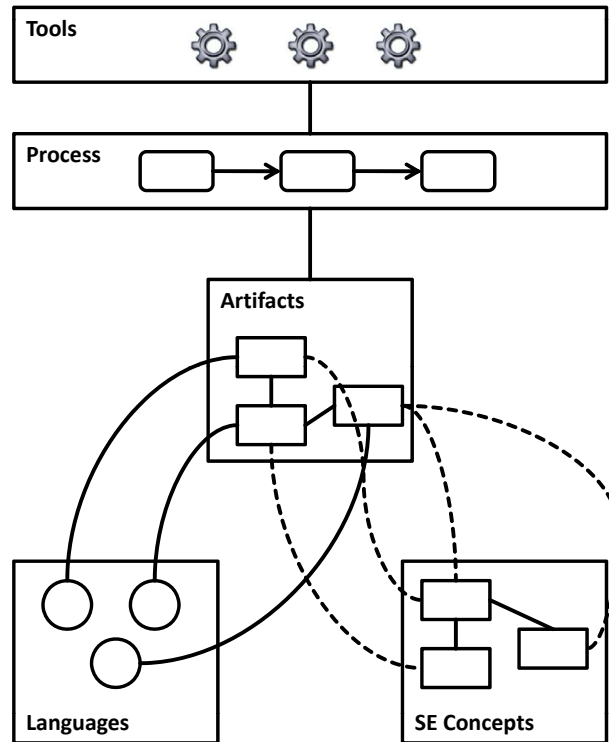


Figure 2.1: Step-by-step definition of a development method [ESS08]

they should select languages for describing these concepts.

The third step consists of organizing the concepts and their specification in the selected languages in development artifacts (e.g. design documents) and defining the relations between the development artifacts. In the next step, the processes are defined during which the development artifacts are produced. As last step the tools are selected which enable the defined processes and support the production of the development artifacts.

## 2.2 Model-based Software Development

In this section, we will briefly explain the concepts and techniques of model-based software development (MBSD). Figure 2.2 illustrates an exemplary MBSD scenario based on the V-model by Basilli and Pezze [BP06]. Compared with the abstract V-model from Figure 1.2, the design and the implementation artifacts are defined in this figure more concretely. During the design phase, the initial requirements are stepwise refined into system specifications, subsystem specifications and to component specifications. The

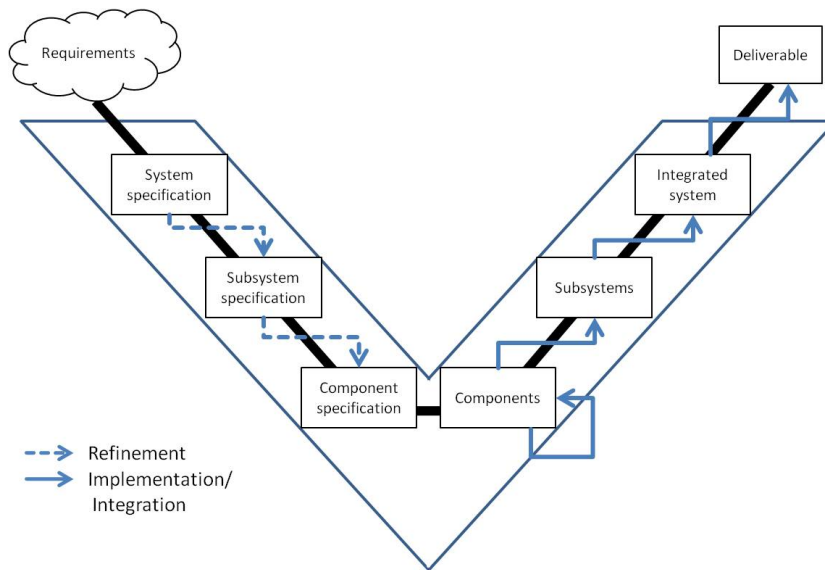


Figure 2.2: Model-based software development for V-model

implementation begins with the components, which are the building blocks of the software. The components are stepwise integrated to subsystems and subsystems to integrated system. The implementation phase results with the deliverable software.

MBSD aims at using *models* as central artifacts through the development process, especially for specifying requirements on software during the design phase. The purpose of using models is to handle complexity of the software. Models allow to abstract from irrelevant details and thus to focus on relevant aspects of software under development at a certain point of time during the development process. The abstraction improves the communication between team members and customers. Furthermore, they are used for documenting the customer requirements and design specifications. If models are captured in a formal and machine readable format, these can be automatically analyzed for certain quality attributes. Also program code can be generated from models by using model transformations.

In order to benefit from models, however, they first have to be created. In the next section, we give an overview on the state of the art modeling languages. After that, we will introduce model transformations as a tool for realizing the notion of MBSD to automatically translate models into other models or into program code.

### 2.2.1 Modeling

Models help us to better understand our environment by simplifying and generalizing our experiences from daily life [Lud03]. Stachowiak states three characteristics for models: (1) they have to map to properties of real world objects (mapping criterion), however, (2) they should not map all of the properties of real world objects (reduction criterion) and (3) they should fulfill useful purposes (pragmatic criterion) [Sta73, Lud03]. While we mostly deal with implicit mental models in our daily life, in MBSD mental models have to be made explicit, such that they can be discussed or exchanged within team members. If the criteria of Stachowiak are applied, models in MBSD are used for describing the software or its environment, they focus on certain aspects of the software while hiding other aspects, and they support a concrete development activity.

In MBSD, models are used for describing the structure and the behavior of software systems. *Structural models* describe the concepts or *entities* of software and how these are related to each other, which is the *architecture* of software. *Behavioral models* describe the *functions* of software, its *interaction* with the human users or with other software and how software's *states* are changing depending on internal or external events.

In the literature of software engineering, many modeling languages are addressed depending on the aspects to be modeled and on the requirements of the application domain. Unified Modeling Language (*UML*) [Gro03b] is the state of the art modeling language for software development. UML contains many notations for behavioral and structural modeling. Figure 2.4 summarizes the UML notations for describing the behavioral and structural aspects.

The UML notations are defined for general use. However, in a concrete software development process, not all of the UML notations have to be used. A subset of notations can be selected for the concrete needs of the development process. If further language constructs or totally new notations are needed, the extension mechanism of UML enables to extend the existing notations or to define new notations based on the UML meta-modeling. This mechanism is called *UML profiles* [Gro03b]. By using UML profiles, domain specific languages (*DSL*) can be defined.

### 2.2.2 Model Transformations

As discussed above, models are used in MBSD not only for documenting and communicating software requirements, they are also used for automated analysis. Thereby, certain quality properties of models can be automatically

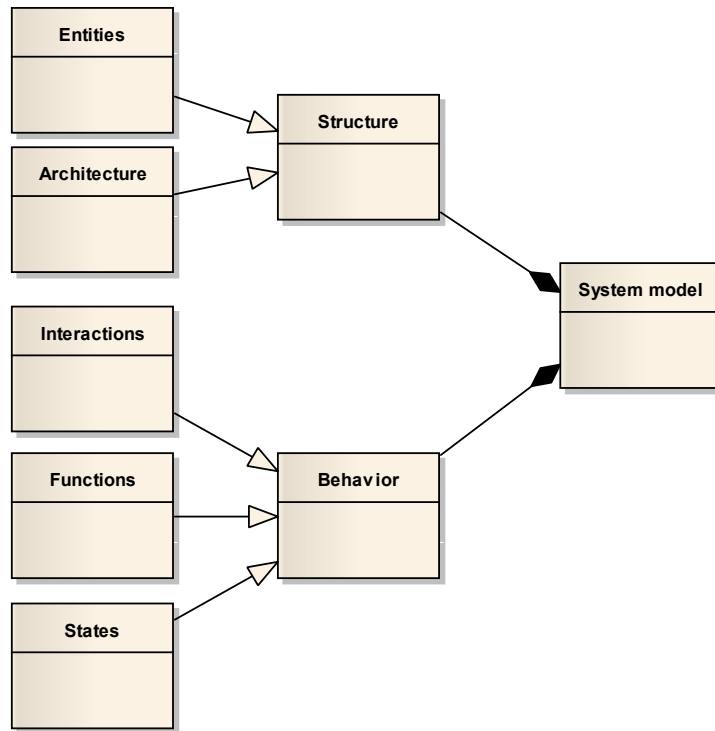


Figure 2.3: Design concepts in Model-based software development


				
Structure		Behavior		
Entities	Architecture	Functions	Interactions	States
Class diagrams	Package diagrams	Activity diagrams		State Charts
Object diagrams	Distribution diagrams	Use case diagrams	Sequence diagrams	
Component diagrams			Communication diagrams	
Composite structure diagrams			Interaction overview diagrams	
			...	

Figure 2.4: UML diagrams for modeling structure and behavior [Ros09]

analyzed or models can be automatically manipulated or transformed into different formats. For automation purposes, models must be defined in a formal way. In this section, we explain how models can be automatically transformed into other models or program code for fulfilling the notion of MBSB to use models as the central artifacts in the development process.

In *MDA* [Gro03a], which is the concrete MBSB technology of OMG, model transformations are used as an enabling technology for stepwise refinement of design specifications and for code generations. As shown in Figure 2.5, the MDA process starts with a manual derivation of platform independent models (PIM) from customers requirements. PIM's are abstract models which focus on logical and domain-related aspects of software under development. In the next step of MDA process, platform specific models (PSM) are derived from PIM's, where the elements of PIM are fed with technical details. The derivation can be conducted either manually or automatically. If PIM's and PSM's are defined formally, the derivation can automatically be done by model-to-model (M2M) transformations. Then, the PSM's are extended with further structural and behavioral details, such that they can be transformed into program code. If PSM's contain enough details, the transformations again can be done automatically. Thereby, the model elements in PSM's are translated into code fragments by model-to-text (M2T) transformations. Even if the notion of MDA is the fully automatic code generation from PSM's, in practice, the models do not contain to much implementation details such that manual programming is still needed for completing the generated program code for achieving executable software.

After having informally introduced the notion of model transformations, we want to give some details on the theoretical background of this technique. Figure 2.6 illustrates the concepts and the process of model transformations based on Czarnecki and Helsen [CH06]. A model transformation addresses three key areas which are the source models, the target models and the transformation itself. The source and target models must conform to corresponding metamodels which define their abstract syntax. The definition of the transformation contains transformation rules which refer to the syntax elements of the source and target metamodels. These rules specify how the syntactical elements from the source model have to be transformed into the syntax of the target metamodel resulting in the target model. The transformation engine then reads the source model and applies the transformations rules. Resulting model elements are written into the target model.

In the literature, various approaches and tools for model transformations are presented both from academia and industry. Czarnecki and Helsen [CH06] and Mens and Van Gorp [MG06] give detailed overviews on different categories of approaches and tools for model transformations. Parallel



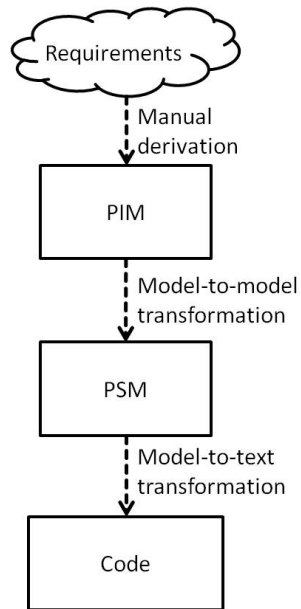


Figure 2.5: Model driven architecture (MDA) by OMG

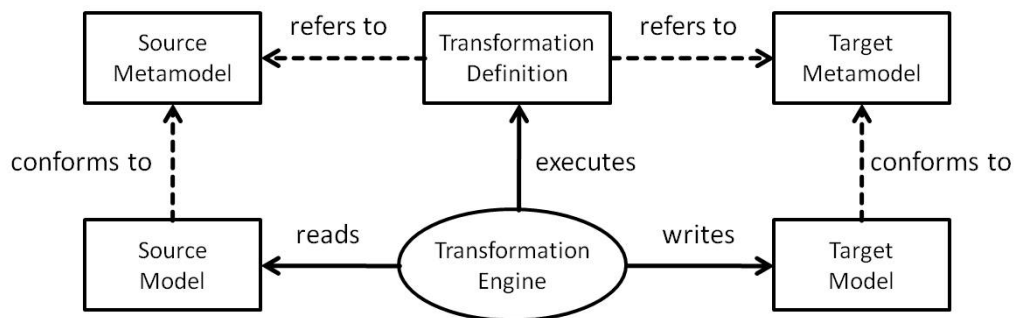


Figure 2.6: Model transformations based on [CH06]

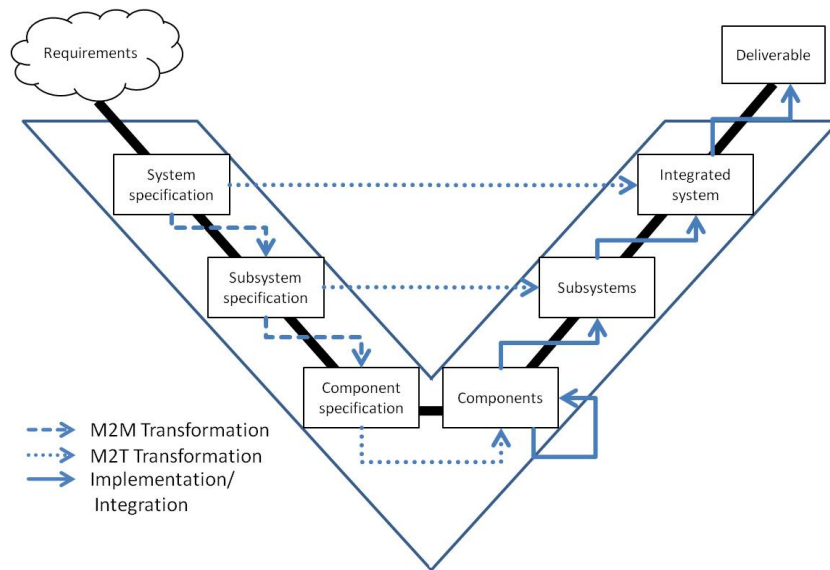


Figure 2.7: Model transformations in V-model

to UML and MDA, OMG also offers a standard for model transformations, which is the Query/View/Transformations (QVT) language [Gro05b]. QVT standard describes how the transformation definition in Figure 2.6 can be created. Concrete implementation of this standard (e.g. ATL, VIATRA, Fujaba [SEG]) can be then used as transformation engine, as shown in Figure 2.6.

Figure 2.7 shows how the concepts of MDA, UML and QVT can be integrated into the V-model. After initially deriving abstract system specification using UML, these can be stepwise refined into more detailed specifications by M2M transformations. Then, the specifications can be used for generating parts of software code (e.g. in Java) by using M2T transformations. From structural models in the specifications, code frames or interfaces or assertions can be generated from structural models. Programmers can manually extend the generated code by behavioral code which is specified in the behavioral models. Then, the resulting code parts can be stepwise integrated to more sophisticated software. Figure 2.8 illustrates the program code concepts in MBSD.

During the manual coding and integration activities, programmers can make errors which results in erroneous software. Also the automated transformation steps can cause errors in software, because of several possible reasons: the source models can be erroneous, or the transformation rules can be faulty, or the implementation of the transformation engine can contain errors. In any case, the resulting software must be tested after each imple-

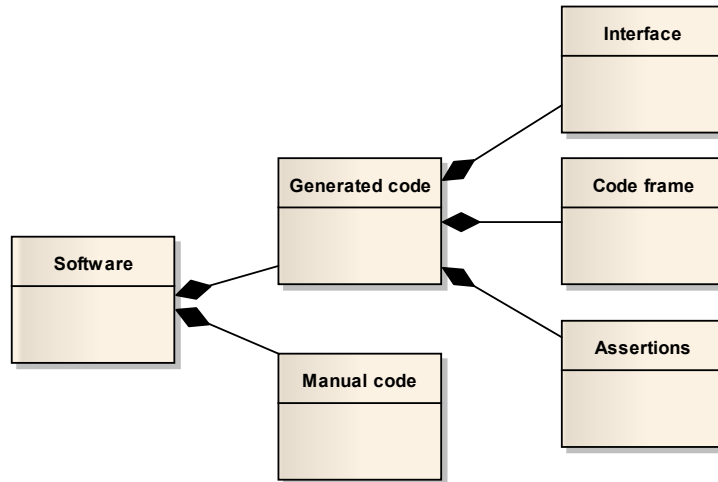


Figure 2.8: Implementation concepts in Model-based software development

mentation step in order to assure the correct functioning of software. In the next sections, we will explain as first the fundamentals of software testing in general and finally its integration into the MBSD process.

## 2.3 Testing in Development Process

We have already discussed, how design and implementation activities in V-model are organized in order to cope with the complexity of software development. Similarly, testing is also a complex task and should be considered as a distinct process integrated into the development process. In this thesis, we mainly consider the V-model as the development scenario, since this is the most traditional software development process.

As already shown in Figures 2.2 and 2.7, the design and the implementation activities produce specifications and software of different characteristics. While, the specifications have different abstraction levels, the software at each development level are of different size and complexity. The task of tester is to check the conformance of the implemented software to its specification on each development level. The different development levels require appropriate test activities and test techniques, which we explain in the next subsections.

### 2.3.1 Test Levels

The implementation activities first start with small size of software (e.g. methods, classes). Then, the small size software are stepwise integrated

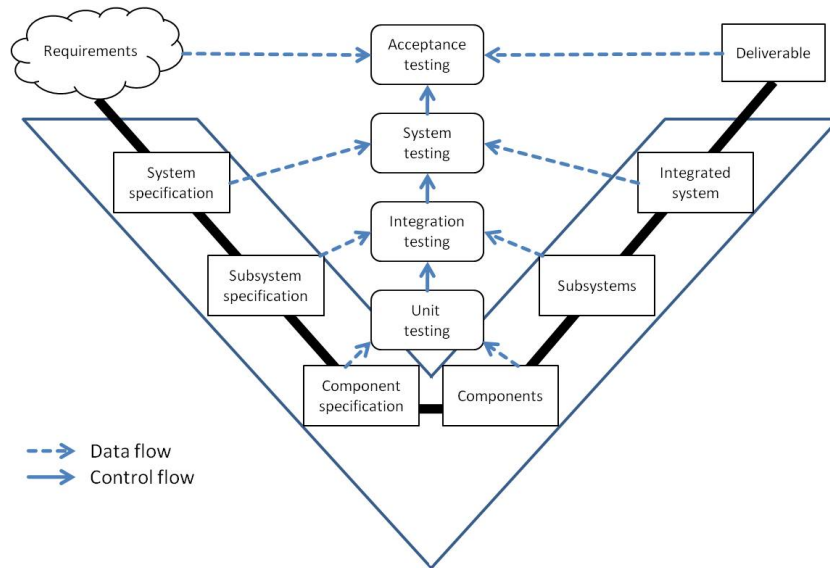


Figure 2.9: V-model with test levels

into bigger size of software (e.g. subsystems, systems) giving at the end of the development process the deliverable software. Software of different size and complexity must be tested differently. One reason for that is the different *types of defects*, which may have been inserted during the manual implementation and integration steps; an other reason is the different kinds of *interfaces* of software under test, which we call as *test objects*. A systematic test process must consider these differences in development levels.

Figure 2.9 shows four **test levels** in V-model which correspond to the development levels: unit testing, integration testing, system testing and acceptance testing. The dashed lines represent the data flow, where for each level of test process the test object and the corresponding specification are used as inputs. For simplicity, the test results, which are the outputs of the test process, are not illustrated in the Figure. The solid lines represent the control flow of the overall test process, where the testing activities start with the unit testing and ends with acceptance testing.

As next, we will characterize these test levels based on Spillner and Linz [AS05]. Thereby, we will address the following aspects for each test level: test object, test environment, test target, defect type, test strategy.

## Unit Testing

Unit testing is the most low-level testing activity which is mostly conducted by developers. That is why unit testing is sometimes called as *development*

*testing* [MH09]. Test objects are software components, classes in components and their methods. Unit testing aims at validating the correct implementation of components with respect to the component specification. Thereby the functional correctness of components is checked by input/output behavior of the class methods. The test environment is very similar to the programming environment, where a *test driver* calls the public interfaces of the components under test. Typical types of defects for components are faulty or missing program code. Besides functional correctness, also non-functional properties of components are in focus, e.g. robustness of components in case of invalid inputs (*negative testing*) or their efficiency. If errors are detected, components must be debugged, the errors have to be corrected and the components must be re-tested.

### Integration Testing

After single components are tested and corrected, the components can be integrated to subsystems. Thereby, not only the self developed components can be integrated, but also third-party components can be a part of the subsystem. The resulting subsystem constitute a self-contained domain-relevant functionality. The test target in integration testing is to validate the correct interaction of the integrated components to realize the intended functionality. Defect types are incompatible interfaces between the integrated components, that is, missing or erroneous interface parameters, inconsistent interpretation of interface data. In addition to these functional errors, also non-functional errors can occur, e.g. timing problems between components, if *caller* components and *callee* components cannot be synchronize the data transfer. The test environment is still low-level as in unit testing, i.e. component interfaces can be invoked by the test driver. An important challenge in integration testing is to define an order of testing activities, in which the components should be integrated and tested. *Top-down* integration requires that caller components are used as test driver for testing the interaction of the caller with the callee components. Thereby the callee components, which are not implemented yet, must be simulated. In bottom-up integration the callee components are assumed to be available. The caller components are simulated by the test driver. Besides top-down and bottom-up strategies, there are also some other strategies, e.g. ad-hoc integration (random order for testing components) or big-bang integration (testing all components at once).

### **System Testing**

System testing aims at testing the complete system which results from the integration of the subsystems. Test target is to validate the fulfillment of the customer requirements, which are specified in system specification and which address both the functional and the non-functional needs. Therefore, the testers require an test environment, which must be similar to the customer's environment as much as possible. It is a challenge for testers to set-up an environment involving the platform software (operating system and drivers) and the application software (office software, proprietary applications, etc.) and the hardware comparable to the customer's productive environment. If the test environment is set-up appropriately, errors can be found, which were not testable during the low-level test activities. For example, inconsistencies in the system specification, or missing requirements, or bad design-decisions can be detected after the complete system is constructed and executed on a realistic environment. For testing functional and non-functional properties, various test strategies can be applied. Important is, that the test activities involve end-user scenarios, e.g. testing use cases, performance, user-friendliness, etc. As test interface, end-user interfaces are used, e.g. the graphical user interface (*GUI*).

### **Acceptance Testing**

Even if the developers of the software system assure the quality of software by testing it before the delivery, the customer also wants to test the system by himself to make sure that the requirements are fulfilled. This activity is called as acceptance testing. The main target thereby is to check the contractual issues for the complete system. Important for acceptance testing is, that it should be conducted in the productive environment by end-users, which is also the main difference to the system testing. Otherwise the test cases can be similar to the system testing. Defect types relevant for acceptance testing are missing requirements, unexpected influences of other software or hardware components in the productive environment, which could not be simulated during system testing. There are different strategies for acceptance testing, e.g. alpha-tests, which involve both the developers and the end-users, and beta-tests, which is conducted by the end-users alone.

### **2.3.2 Test Activities**

In the last section, we have characterized the four test levels in V-model. Even if these test levels differ in many aspects, like defect types or test en-

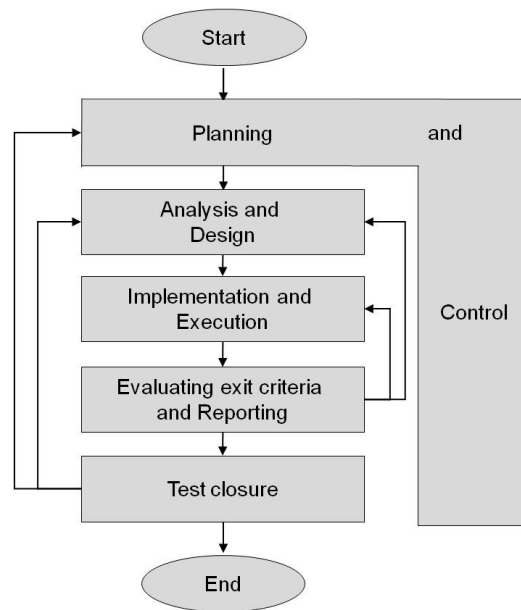


Figure 2.10: Fundamental test process by [ISTQB]

vironment, they involve similar activities. The International Software Testing Qualifications Board (*ISTQB*) has defined the fundamental test process (*FTP*) [ISTQB] which proposes a distinct and unified process for testing, which should be applied for each test level [Win09]. As illustrated in Figure 2.10, FTP contains the following activities [MBE<sup>+</sup>07]:

- **Planning and controlling:** Test planning aims at defining the *test objects* (i.e. software under test) and *test targets* of the particular test level and at planning the activities for meeting these targets. Also the *exit criteria* for testing phase, which refine the test targets, are decided in this phase. Controlling aims at a continuous monitoring of the test activities and checking whether testing activities process as planned. If not changes in the *test plans* have to be made.
- **Analysis and design:** During test analysis the specification documents are identified which are relevant for test targets. These documents serve as *test basis* for test design. From the test basis a set of *logical test cases* are derived by using some *selection criteria* resulting in a *test suite*. The logical test cases aim at defining the test relevant scenarios from an abstract point of view.
- **Implementation and execution:** In this phase the logical test cases are concretized with concrete test data, resulting in *concrete test cases*.

Then, concrete test cases are enhanced with technical details such that they can be executed. As we aim at automated test execution, we speak of *test scripts* at this stage, which are executable test cases. For test execution, testers set up a *test environment* involving the test objects and further software and hardware used by the test objects. If the setup of the test environment is not possible, this will be simulated. After test execution *test results* are verified with respect to the expected results.

- Evaluation and reporting: The fulfillment of the exit criteria is checked in this phase. If these criteria are not fulfilled, new test cases have to be defined. Test reporting aims at documenting the test results and the fulfillment of the exit criteria.
- Closure: Test closure is about collecting experiences, best practices, lessons learned from the completed test process and to consolidate these for future tests.

Testers face many challenges during the above mentioned activities. Main challenges in testing are hidden in test selection and the test execution. Because these activities are mostly conducted manually, they are inefficient and error-prone. Manual activity can also lead to subjective decisions of testers which makes the test process unsystematic and not replicable. As next, we want to focus on the test activities “Analysis and Design” and “Implementation and execution” which involve these challenges.

### Analysis and Design

The aim of testing is to validate the correct implementation of the software with respect to its specification. For doing this, testers think of exemplary usage scenarios based on the specification. Thereby, depending on the test level, testers handle different test objects, which emerge to different the development levels. Therefore, it is an important task for the testers during “analysis and design” to identify the test objects and the corresponding parts in the specification. This specification and all other documents related for testing the software are called *test basis*. From the test basis, a set of test cases (*test suite*) is derived. Figure 2.11 illustrates the concepts in analysis and design and their relations.

In traditional testing, testers define test cases based on their domain knowledge and experience. However, this way of defining test cases is not systematic and not comprehensible by other team members. In order to achieve a systematic and comprehensible test process, the test suite should



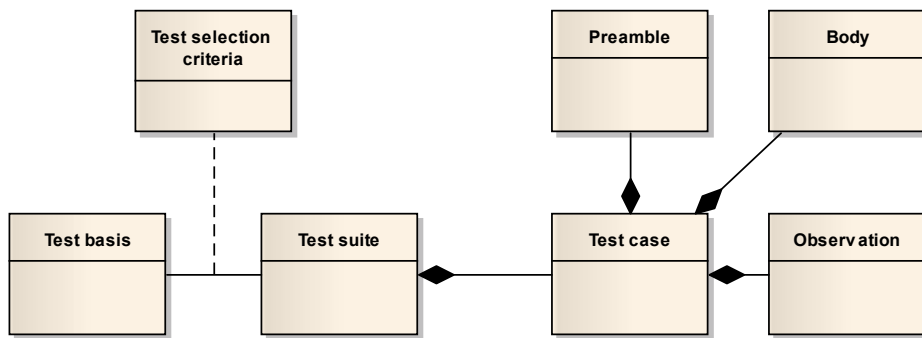


Figure 2.11: Testing concepts and abstract test case

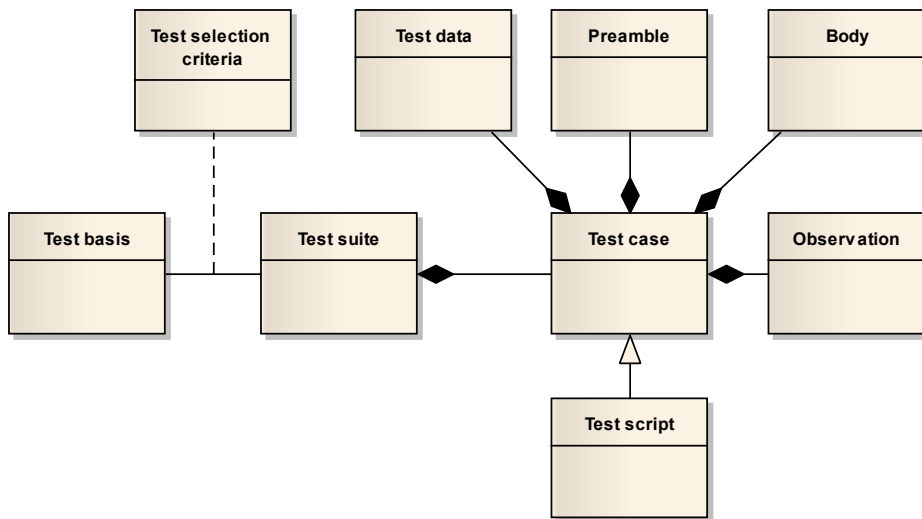


Figure 2.12: Concrete test case and test script

be selected from the test basis by using some predefined test selection criteria. Thereby, the selection criteria can base on some coverage metrics or on prioritization of requirements. A test suite contains a set of test cases. At this stage of test process, test cases describe the test scenarios from a logical point of view and do not concrete test data. However, they have to address which activities have to be conducted in order to prepare the test execution (*preamble*), the execution of the test objects itself (*body*) and the activities to decide on the correctness of the test results (*observation*) [ISO94].

### Implementation and Execution

After the logical test cases are derived from the test basis, during “implementation and execution” these are concretized with concrete *test data* and

technical details for being executed (see Figure 2.12). The resulting concrete test cases contain different combinations of test data. In general, it is not possible to test all combinations of test data for a logical test case. The reason for that is the (almost) infinite number of possible combinations of data, if the test objects deal with usual data types, e.g. integer, float or strings. Thus, testers have to select a subset of the all possible test data. For doing this, testers need a systematic way how to go through the specification and derive exemplary data combinations. In the literature, many techniques are proposed for test data selection. Some of these techniques are defined in [ISTQB] as follows:

- **Equivalence partitioning:** A design technique in which the test data space is divided into partitions with the assumption, that the test cases in a partition will result in the same or similar behavior of the test object. Thus, selecting a representative from a partition will be enough for covering the behavior under test. Testers are done, if from each partition at least one representative is selected and executed.
- **Boundary value analysis:** A test design technique which uses the boundary values of equivalence partitions as test input parameters. This technique is motivated by the assumption that test input parameters in the near of the edges (minimum and maximum values) of the equivalence partitions are likely to detect errors in software.
- **Random testing:** A test design technique where test data are selected by using a random generation algorithm.
- **Statistical testing:** A test design technique in which a model of the statistical distribution of the test data is used to select representatives.

After having concretized the test cases with concrete test data, testers execute them on the test object. During the test execution, the responses of the test object have to be analyzed for conformance to the expected behavior as defined in the specification. Similar to the test case selection during “analysis and design”, also the execution of test cases and the evaluation of test results in this stage of test process are tedious and error-prone tasks if they are manually conducted. Therefore, these test activities are tried to be conducted automatically.

For executing test cases automatically, they have to be translated into a computable format, which we call *test scripts* (cf. Figure 2.12). Test scripts are computer programs which implement the test scenarios described by test cases. Thus, they inherit the elements of a test case, which are the preamble,

body, observation and test data. The automation of test execution using test scripts makes the test process efficient, precise and repeatable. However, similar to software development, also the development of test scripts must be conducted carefully and systematically. The next section gives an overview on test automation techniques and its challenges.

### 2.3.3 Test Automation Techniques

The activities defined by the fundamental test process can be very expensive and error-prone if they are manually done. Especially the core activities “analysis and design” and “implementation and execution” contain the most tedious work for testers. During “analysis and design”, testers manually analyze the test basis and design the logical test cases. During “implementation and execution”, they manually select test input parameters, execute the test cases and evaluate the test results. Each of these individual tasks require creativity and concentration of testers. In order to reduce the error-proneness and to improve the efficiency of these tasks, test automation tools are used.

In classical test automation, the focus lies on the automated execution of test scenarios using *test scripts*, which are the implementation of test cases. The automatic execution of test scripts makes testing faster, more precise and repeatable. Test scripts are executed by *test drivers* (i.e. test tools), which are software able to interpret and execute the test logic implemented in the test scripts. Depending on the test level and the characteristics of test object, test drivers can invoke the operations of test objects using their interfaces (cf. Figure 2.8).

#### Script-based Test Automation

In the literature of software test automation, various test tools are proposed for different test levels. For unit testing, the JUnit was developed by Gamma and Beck [EG14], which is a programming environment for testers. Test scripts are programmed as Java classes and have standardized interfaces for preparing and executing the test cases. Test cases can be assigned to hierarchical test suites, which constitute complex test scenarios. The results after executing the test scripts and the test suites are collected and shown to the testers textually or graphically. Even if the repeatable test scripts improve the efficiency of test execution, ***script-based test automation*** requires the test scripts to be implemented manually, which is a costly task. Additionally, the correctness of the manually implemented test scripts has to be assured, which again increases the costs of the test automation.

### Capture/Replay Testing

As an alternative to the script-based test automation, *capture/replay testing* was introduced. In capture/replay (C/R) testing, the test scripts are not implemented manually, instead they are automatically recorded (or captured) by the *recording engine* during the interactions of the testers or users with the test object. The recorded test scripts can be then executed repeatedly (replay). Typically, capture/replay tools are applied on higher test levels (e.g. system testing or acceptance testing), where the test scripts are recorded on the graphical user interface (GUI) of the test object. Thereby, the recorded test scripts contain sequences of GUI interactions of the user including the test inputs used during these interactions. For the replay of the test scripts, the test inputs can be parameterized and bound to data sources in order to use different test data for each test iteration. This helps to simulate different test scenarios and to increase the test coverage. Besides of its advantages at creating the test script, capture/replay testing also has some disadvantages, i.e. the recorded test scripts must be parametrized and maintained. Depending on the techniques applied during the capturing, the recorded test script can be useless if changes on the GUI are made. That is, test scripts captured by C/R tools are not flexible and not maintainable.

### Keyword-driven Testing

As an answer to the disadvantages of capture/replay testing, *keyword-driven testing* was introduced, which proposes to define the test cases on a higher abstraction level, such that they do not contain any technical details. The technical details are handled during the test execution by an *adapter*, which translates the abstract definitions of the test steps into specific invocations on test object's interface. In this technique, the abstract test logic is separated from the technical test execution, which makes the test scripts more maintainable and flexible.

### Summary

So far, we have seen some test automation techniques which handle the problems of manual testing for creating, executing and maintaining test scripts. However, the definition of the test cases is in these techniques ad-hoc and not systematic. They are not explicitly linked to the test basis. Thus, a *coverage measurement* on the test basis during test execution is not directly possible. However, in order to be able to control the test process and to decide on test exit, systematic and coverage-oriented testing is important.

Another weakness of the above mentioned test automation techniques is the missing *test oracle*. Test oracle is “a source to determine expected results to compare with the actual result of the software under test” [MH09]. In each of the presented automation techniques, the expected results have to be manually integrated into the test scripts or the actual test results have to be manually checked after the test execution. Both of these approaches increase the costs of the test automation.

The flaws of the above mentioned test automation techniques show the need for a more systematic way of test automation. In the next section we will introduce model-based testing and show its role in model-based software development (MBSD) which introduced in section 2.2.

## 2.4 Model-based Testing

There are various definitions for model-based testing (*MBT*) in the literature which differ in their focus. Utting and Leguard define MBT in [UL07] as “the automation of black-box test design”. With this definition, they set the focus of MBT on the testing activity “analysis and design” (cf. Figure 2.10). Heckel and Lohmann focus in [HL03] on both “analysis and design” and “implementation and execution” (cf. section 2.3.2). They see the following tasks to be solved by MBT:

1. ”the generation of test cases from models according to a given coverage criterion,
2. the generation of a test oracle to determine the expected results of a test,
3. the execution of tests in test environments, possibly also generated from models.” [HL03]

Thus, models play a central role in MBT as it is also the case in MBSD (cf. section 2.2). The so-called *test models* are used as test basis from which test artifacts, like test cases, test oracles or test scripts, are generated. Figure 2.13 shows the relation of test models to other testing artifacts. Thereby, the test selection criteria must address coverage metrics for model elements or for possible test data.

The idea for using models for supporting testing activities is not new. Binder stated 1999 that “testing is always model-based” [Bin99]. He claims that testers always have an *implicit mental model* when they create test cases, implement test scripts or evaluate test results. Pretschner and Philipps

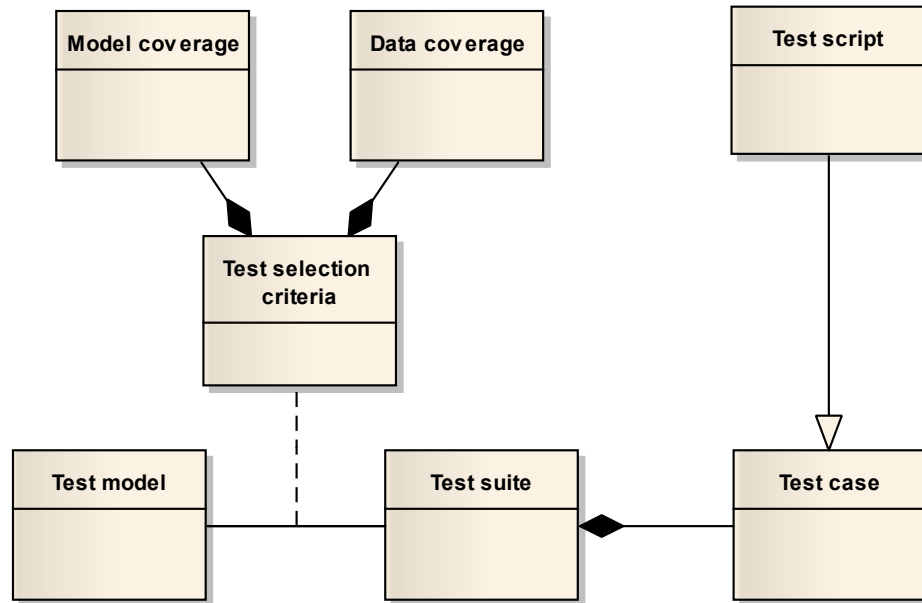


Figure 2.13: Artifacts in model-based testing

state in [PP04] that MBT makes the mental models of testers *explicit*. Similarly, Winter compares in [Win09] traditional specification-based testing techniques with model-based testing. He concludes that utilization of models in software testing is not new, however it becomes more important at times of MBSD.

The definitions of MBT in the literature are all agree that MBT is about using models for supporting test activities. However, MBT approaches differ in *to what extent* they use models. Rossner differentiates in a recent article [Ros09] between three maturity levels for MBT approaches:

- In *model-oriented testing*, models are used for documentation and communication purposes only. The test case design and the further testing activities, like the test execution and the evaluation of the test results are done manually.
- *Model-driven testing* contains the automatic generation of test artifacts, especially of the test scripts from models. However, the execution of the test scripts and the evaluation of the test results are handled similar to classical test automation. The test results are not linked to the models.
- In *model-centric testing* however, the models are the most central test artifacts, which are not only used for the generation of test scripts, they

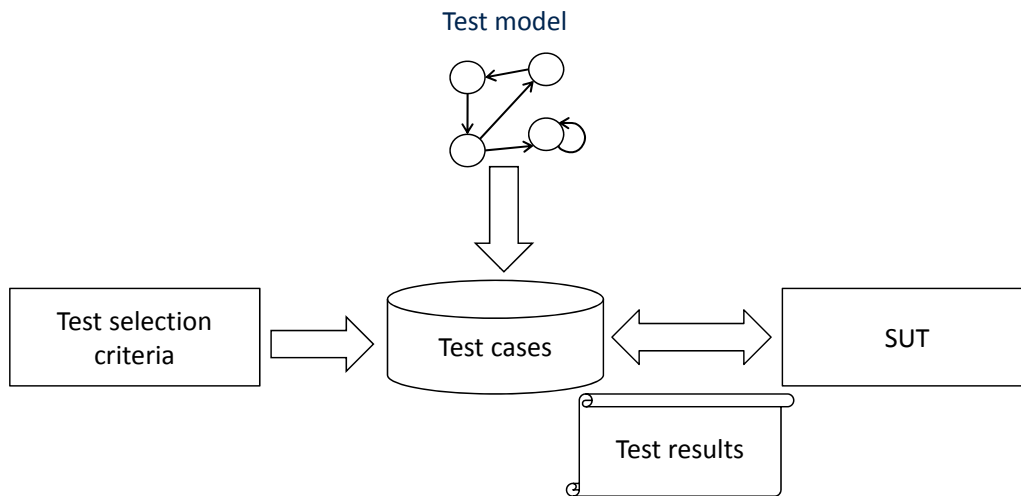


Figure 2.14: General process for MBT based on [PP04]

are also fed with the informations gathered after the evaluation of test results. Thus, during all phases of the test process, the informations (e.g. test coverage, found error) are hold and archived by means of models.

These definitions show that, MBT can help testers to resolve the flaws of test automation addressed in section 2.3.3. That is, MBT makes the test design systematic, efficient and comprehensible and it makes the test execution efficient, precise and repeatable. In the rest of this section, we will introduce the general MBT process and some MBT scenarios.

### 2.4.1 MBT Process

The general MBT process is illustrated by Pretschner and Philipps in [PP04] as in Figure 2.14. First, test models are created. Then, test cases are generated from test models with respect to some test selection criteria. The generated test cases are executed on the test object, which is the software under test (SUT). Then the test results are evaluated with respect to the expected behavior specified in the test models.

Being an automated technique, MBT aims at improving the test process by making it more efficient and more precise. Different than the classical test automation techniques explained in section 2.3.3, MBT also aims at making the test cases and the test scripts more flexible. The changes in the software specification are edited directly into the test models from which then the test cases or the test scripts are just re-generated. The test selection criteria make

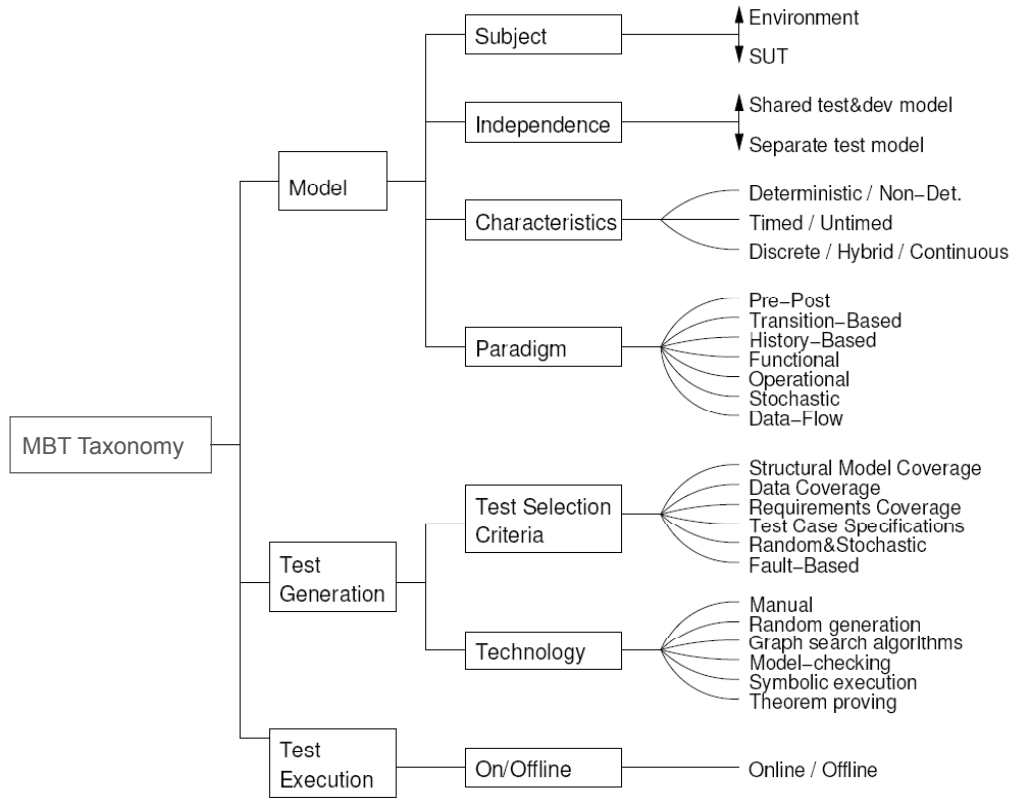


Figure 2.15: Taxonomy for model-based testing by [UPL06]

the test case generation more systematic, such that a navigability between the test cases and the test models is assured. Last but not least, MBT handles the oracle problem by embedding the specification of the SUT’s expected behavior into the models, which is used for the evaluation of the test results.

Even if MBT can help in coping with the flaws of classical test automation, it dissembles also some challenges: (1) creating the test models can be an expensive and error-prone work, and (2) MBT requires additional *skills* of testers for handling with models, and (3) MBT process requires *tools* which integrate into the development environment.

In the literature of MBT, various approaches are proposed, which differ in the languages, techniques and tools they utilize. Utting et al. created in [UPL06] a taxonomy for characterizing the MBT approaches, which is illustrated in Figure 2.15. They define seven dimensions regarding the MBT-activities *modeling*, *test generation* and *test execution*.

Regarding the *modeling*, four dimensions are defined: subject, independence, characteristics and paradigm. Subject is about what is modeled. The



modeling subject can be the SUT itself, but it can also be its environment. Schiefedecker also addresses in [Sch07] approaches which model the testware, e.g. the test driver, the adapter etc. Independence is about the *reuse* of the system models, which are created by developers during design phase, as test models. The more testers create their own separate models, the more independent are these two models. Pretschner and Philipps discuss in [PP04] the importance of independence for fault detecting capability of MBT. Characteristics describes the *properties* of the models, e.g. timed models, deterministic models or discrete models. These characteristics mainly depend on the application domain, e.g. in the automotive domain, timed and continuous models are required. Paradigm is about the modeling *notation* used for creating the models, e.g. pre/post-based or transition-based notation.

The dimensions for the activity *test generation* are *test selection criteria* and *technology*. Test selection criteria addresses some techniques for specifying which elements of the test models and which test data are relevant for the test case generation. These techniques are mostly adopted from the classical black-box and white-box test selection techniques mentioned in the Section 2.3.2. The technology dimension is about how test cases are technically generated. The tools and the algorithms used there depend on the notation and the characteristics of the test models.

The dimension *on/offline* for the *test execution* is about when the test cases are generated. They can be either generated prior to or during the test execution. In the later case the informations gathered from the interim test results can be used for generating new test cases.

In the next sections, we want to explain the dimensions “independence” and “paradigm” in detail, because these dimensions are the most important characterizing aspects for MBT approaches.

### 2.4.2 Different Approaches

As explained in the MBT taxonomy above, the relation between the system models and test models plays an important role in MBT. Figure 2.16 shows the structural dependencies between the artifacts in the MBSD process (cf. Figures 2.3 and 2.8). Structurally, test models can be similar to system models, such that they entail also behavioral and structural parts. However, while system models are used as a specification for the software under development, test models are used as a specification for the testing artifacts. By selecting exemplary scenarios from the test model and enriching them with test data, realistic application situations for the SUT can be established.

Even if system models and test models can be structurally similar, concerning the contents, they can (or should) be very different. The more testers

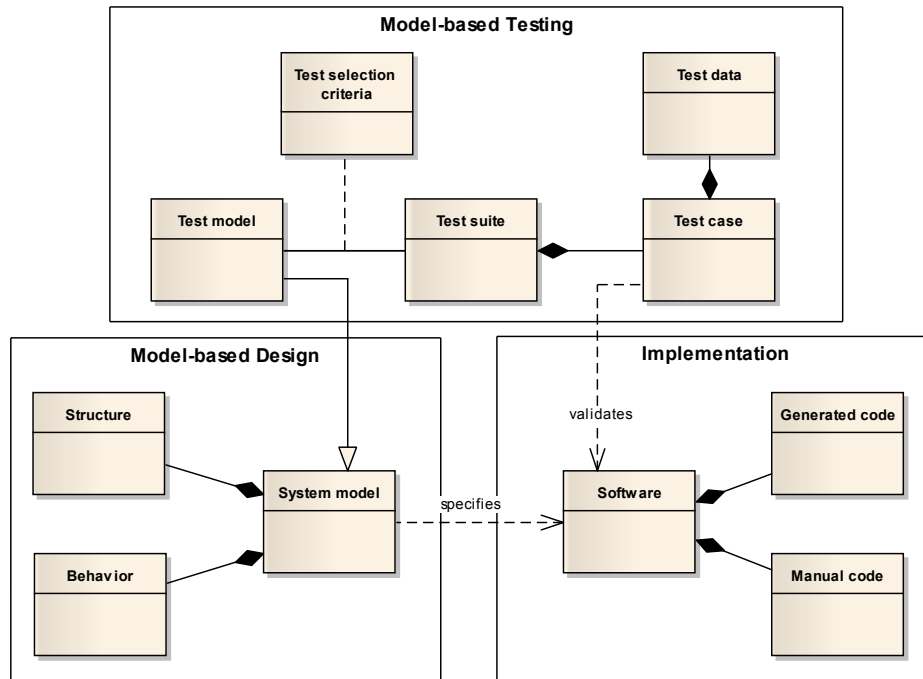


Figure 2.16: The relation between design, implementation and testing artifacts

reuse developers models for testing, the lower are their chance to find errors in the SUT. The reason for that is the fact that, developers models are used as a source for the implementation of the SUT. If there are errors in the developers models, in case of a high reuse, these will be also existent in the test models. Thus, the errors will be hidden from the testers. The more testers create their own separate models, the more independent are testers from the developers. Also Pretschner and Philipps discuss in [PP04] the importance of *independence* of test models for a better fault detecting capability of MBT. They differentiate between four approaches:

1. common model,
2. automatic model extraction from code,
3. manual modeling and
4. separate models.

Common model approach handles the development scenario, where developers and testers use the same models for their activities. In the second

approach, models are extracted from the SUT, i.e. from SUT's code or from its execution traces, by using re-engineering techniques. For the first two approaches, the authors see high dependency between the developers models and test models. The third approach is about creating the test models manually by looking at the developers models and the initial software requirements. Finally, the fourth approach creates test models totally separately from the developers activities. Thus, the last two approaches enable an independent development of test models, which increases the chances to detect errors in the SUT.

In addition to the four approaches in [PP04], we have identified in [BG10] two further approaches in the literature. The first one extracts test models from already existing test cases. The second one proposes using model transformations for creating test models from other test models or from developers models. In [BG10], we have also discussed the efforts needed by each of the totally six approaches.

In the rest of this section, we will give some details on the common model and the separate model. We will discuss the *pros* and *cons* for these approaches. Then, we will show which notations are used for creating the test models.

### Common Model

In the common model scenario, a single model is used for the development of the SUT and for test case generation as shown in Figure 2.17. The model is created by developers according to the customer requirements.

As this model can be used by the testers and the developers, no additional effort for test modeling is needed. This scenario has a high automation level, because the code and test cases can be automatically generated from a single model.

However, this scenario induce high dependency between the SUT and the test cases. Thus requirement-related errors cannot be found by using this scenario, because the source for the SUT and the test cases are the same. However, the manual evaluation of test results can detect errors in code generators and in the assumptions on the environment specified in the test models.

### Separate Models

In the separate model scenario, both testers and the developers create their own models as shown in Figure 2.18. The test model in this scenario is only used for automatic test case generation. The generated test cases can

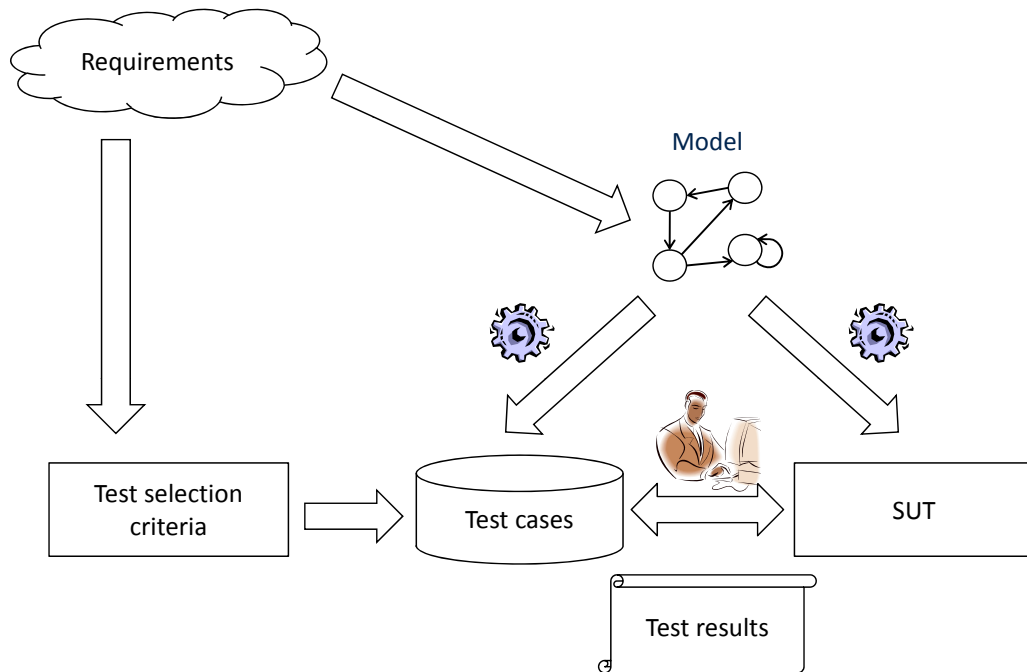


Figure 2.17: Common model based on [PP04]

be executed manually or automatically. As the needed redundancy between both models is given by this separation, the test model can be used as a test oracle. In this case, the test evaluation can be fully automated.

However, in order to profit from the advantages of this scenario, both models have to be manually created, which results in high efforts. An important factor is the need for high modeling skills of the test team, because they have to handle the modeling by their owns.

On the other side the overall efficiency is high, because the test case generation and test evaluation step can be better automated. Moreover, the separate modeling activity enables early start of testing activities thus reducing the dependency between the test team and the development team.

### 2.4.3 Different Paradigms

Having discussed two different approaches for using models, in this section we want to give an overview on the different modeling paradigms and notations which can be used for creating test models. The taxonomy of MBT in Figure 2.15 shows seven paradigms for modeling notations which are used in the literature of MBT ([vL00] as cited in [UPL06] and [UL07]): Pre/post-based (or state-based), transition-based, history-based, functional, operational, statis-

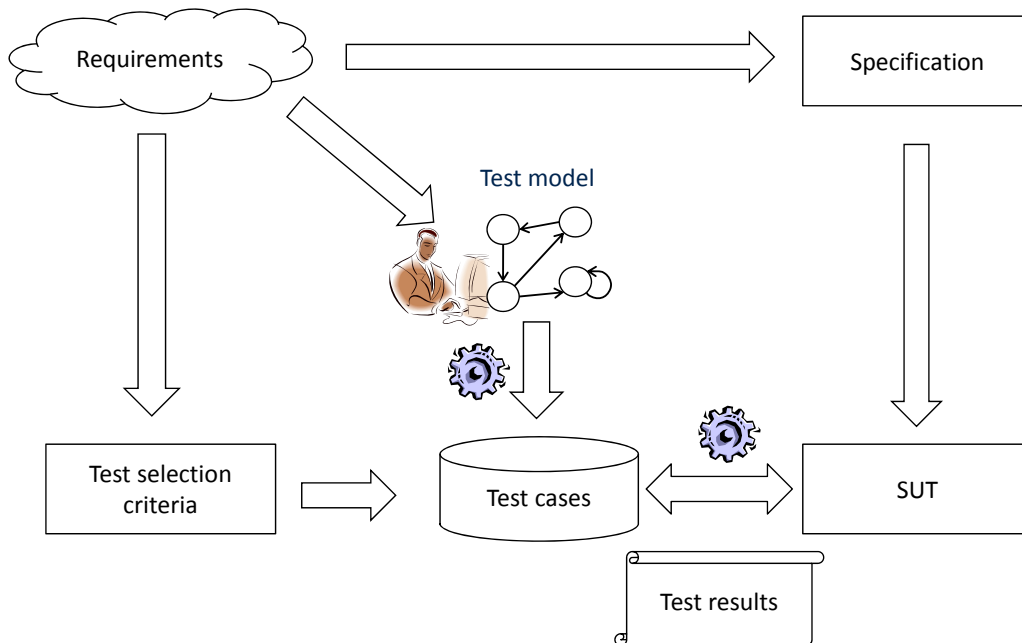


Figure 2.18: Separate model based on [PP04]

tical, data-flow-based.

For determining the most suitable paradigm for a project, the project context should be analyzed. Many factors play a role in this decision. First, the MBT approach as explained in Section 2.4.2 must be considered. If developers' models are to be re-used for testing purposes, testers should be familiar with the developers' notations. This has the advantage that existing know-how and tools in a development team can also be re-used. However, if separate test models are to be created by testers, also other modeling notations can be considered. In choosing a modeling notation, some general qualities of the test models should be considered. Lindland et al. differentiate in [LO94] as cited in [ECFGP10] between three types of model qualities:

- Syntactic quality indicates to which extent the model satisfies the rules of the modeling language. Syntactic errors and violations of the rules reduce the syntactic quality.
- Semantic quality points out the degree to which the model represents the problem domain. The more related the model and the problem domain, the better the semantic quality.
- Pragmatic quality represents the degree to which the model is correctly interpreted by its users. The less misunderstanding, the better the

pragmatic quality.

Aiming at the *semantic quality* of test models, the characteristics of the SUT must be considered for choosing a modeling notation. In [UL07], primarily two types of systems are addressed: control oriented systems and data-oriented systems. In *control-oriented* systems, the behavior description focuses on which operations are activated or invoked depending on the current state of the system or on the external or internal events. In *data-oriented* systems, the behavior is defined by the way how system's state, which comprises variables, objects and set of those, is modified by the operations. Thus, for specifying control-oriented systems, one should focus on the *reactions* of the system on the current state or on the events, whereas for specifying data-oriented systems, the focus lies on the *changes* in system's state. Of course, many systems have both of these characteristics and it is not always possible to assign a system to only one of these categories. However, for abstraction purposes in testing, the most important aspects have to be identified and focused during particular testing activities.

As next, we will explain which modeling paradigms are suitable for modeling control-oriented and data-oriented systems.

### Transition-based Paradigm

Utting and Legeard state that the *transition*-based paradigm offers the most suitable notations for modeling control-oriented systems. Transition-based modeling is the most traditional technique used in software testing [Cho78, FvBK<sup>+</sup>91, Pet01]. These approaches use finite state machines (FSM) as the modeling notation for test models. Thereby, the system states are symbolically modeled as nodes of FSM and the state transitions are modeled as edges. The transitions between the states are triggered by operations, which are modeled as labels at the edges.

In MBT with FSM's, test cases are represented by sequences of operations which trigger the state transitions. For selecting test cases, various *selection criteria* can be used as shown in the taxonomy in Figure 2.15. For example, the taxonomy defines the structural *model coverage* criterion, which exploits the structural elements of the model, such as the *nodes* and *edges* of a transition-based test model [UPL06]. Using this criterion, one can select test cases which cover all of the edges or nodes of the test model. In the first case, we speak of *edge coverage* (or transition coverage), in the later case we speak of *node coverage* (or state coverage). During the test execution, the operations in the test case are executed on the SUT, and the executability of the operations and the correct state transitions are checked.

### Pre/Post-based Paradigm

For data-oriented systems, the pre/post-based paradigm enables modeling the changes in the system's state [UL07]. Thereby, the operations of the SUT are described by means of the state changes they cause. The state change is specified by the difference between the state properties before (*preconditions*) and after (*postconditions*) the execution of the operations. This paradigm is also widely used in MBT [CL05, BKM02, HM05].

Pre/post-based MBT approaches generate test inputs from the preconditions and check the fulfillment of the postconditions after executing the operation under test with the generated test inputs. For selecting the test inputs, *data-coverage* criterion can be applied which systematically cover the value ranges of pre- and the postconditions. Data-coverage criterion selects concrete values from a number of possible input data by using the boundary analysis or equivalence classes (cf. section 2.3.2). One of the challenges in pre/post-based approaches is that they mostly use very low-level modeling notations, e.g. OCL, JML, Spec# [CL05, BKM02]. Thus, they require programming skills by testers for creating test models.

In the next chapter, we will focus in detail on the pre/post-based modeling paradigm and discuss the existing approaches and motivate the need for a novel UML-based MBT approach.





# Chapter 3

## Contract-based Testing

In the last chapter, we have addressed the foundations and the related work for model-based testing (MBT) in general. Due to the taxonomy of MBT, we have explained the different approaches and the modeling paradigms in MBT. As one of the modeling paradigms, we have shortly characterized the pre/post-based paradigm for modeling data-oriented systems and addressed some of its challenges. In this chapter, we give more details on the pre/post-based paradigm for MBT and discuss the pros and cons of the existing approaches, which we group under the term *contract-based testing*. Finally, we summarize the shortcomings of existing approaches, which motivated us to research on contract-based testing using Visual Contracts and its integration into the model-driven software development process.

### 3.1 Characteristics

Contract-based testing (CBT) addresses testing approaches which use behavioral contracts for testing activities. In the taxonomy of MBT [UPL06], CBT can be assigned to the pre/post modeling paradigm, as contracts are represented by pre- and postconditions. In this thesis we will use the terms *contracts* and *pre/postconditions* as synonyms.

Ciupa defines CBT in [Ciu08] as “the automatically testing effort of contracted object-oriented software, which is software built according to the principles of Design by Contract (*DbC*) by Meyer” [Mey92]. Thereby, *contracts* are mechanisms for specifying conditions for an operation (*callee*) that must be fulfilled by any client (*caller*) upon calling the operation (*preconditions*) and conditions that must be fulfilled by the operation after it has been executed (*postconditions*). Contracts also specify class *invariants*, which are conditions on the states of the class instances, which must be fulfilled during

the execution of class operations at all observable states [LBR06].

Ciupa states that CBT enables automated testing that uses operation's preconditions for constraining test inputs and postconditions and invariants as automated oracles [Ciu08].

Collet et al. state in [CDRT04] that contracts make testing very easy to conduct because once the requirements on the callee and the caller have been specified in terms of contracts, testing is equivalent to invoke the callee with contracts activated. They also describe how reasons of detected failures should be investigated. These can be either remain in the erroneous contract specification, or in the faulty implementation of the callee which violates the postcondition, or the caller is not conforming to the preconditions.

In summary, CBT involves the activities creating contracts, generating test inputs, invoking the software under test and analyzing the test results. In the following subsections, we will group these activities in two phases, which are the modeling phase and the testing phase.

### 3.1.1 Reference Model for Contract Modeling

Figure 3.1 shows a generic structure of a contract for a class operation. A class operation comprises software functionality which can be invoked over its interface by using some input parameters. Depending on the size of software under consideration, an operation can be a simple class method or a more complex functionality, e.g. a web service or a software feature executable on the graphical user interface. After the execution of the operation, it returns output parameters. The contract of the operation includes a precondition, a postcondition and an invariant as described by Meyer [Mey92]. The precondition and the postcondition constrain the input parameters and the output parameters respectively, i.e. the input parameters must fulfill the precondition and output parameters must fulfill the postcondition.

Models created using contracts represent “a system as a collection of variables, which represent a snapshot of the internal state of the system, plus some operations that modify those variables. Rather than defining the operations using code as with programming languages, each operation is usually defined by a precondition and a postcondition” [UPL06]. Thus, contracts can be used for defining the behavior of software functions. Thereby, the contracts can serve as a specification for the programmers or for testers.

For using contracts for testing, Utting and Legeard propose a reference process for modeling contracts [UL07]. The reference model contains the following four steps as given in [UL07]:

1. Choose a high-level test objective.

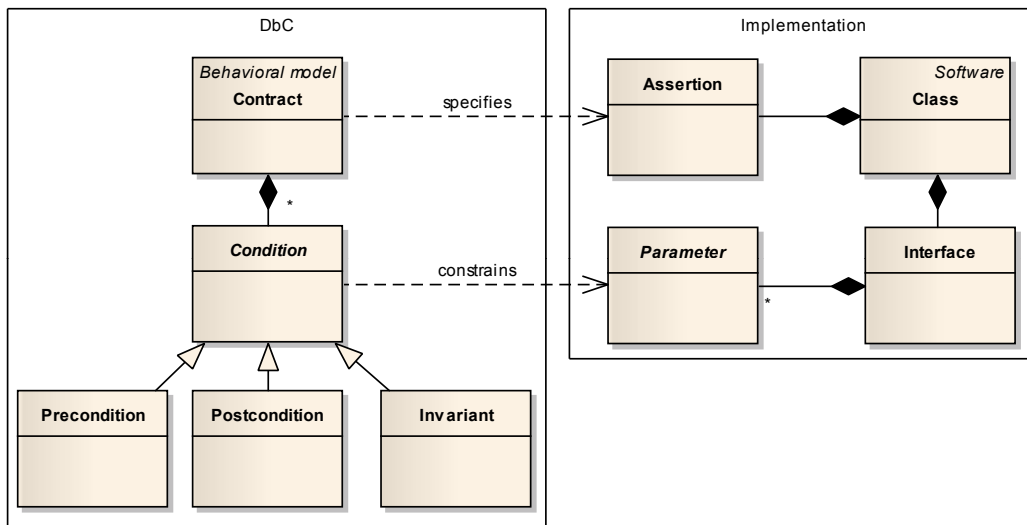


Figure 3.1: Structure of a contract for an operation

2. Design the signatures of the operations in the model.
3. Design the state variables of the model and choose their types.
4. Write the preconditions and postconditions of each operation.

Utting and Legeard point to the importance of how to choose a good level of abstraction for pre/post notations. They state that the goal of the contract modeling is to have just enough detail in the model to be able to generate test inputs and check the correctness of outputs while avoiding unnecessary detail that would make the model larger and more complex [UL07]. The authors mainly address contracts for low level software functions, like class methods. However, this process can also be applied to high level software functions if a suitable abstraction can be found.

### 3.1.2 Reference Model for Testing with Contracts

Utting and Legeard state in [UL07] that “pre/post notations allow you to develop behavior models that are good for generating functional test cases. The precondition of an operation specifies when and how the environment can call that operation. The postcondition describes how the operation changes the state of the model, which is an abstraction of how it changes the internal state of the SUT”. Thus, preconditions can be used for generating test inputs and postconditions can be used as test oracles, which show whether the outputs of the operation are as expected [UL07].

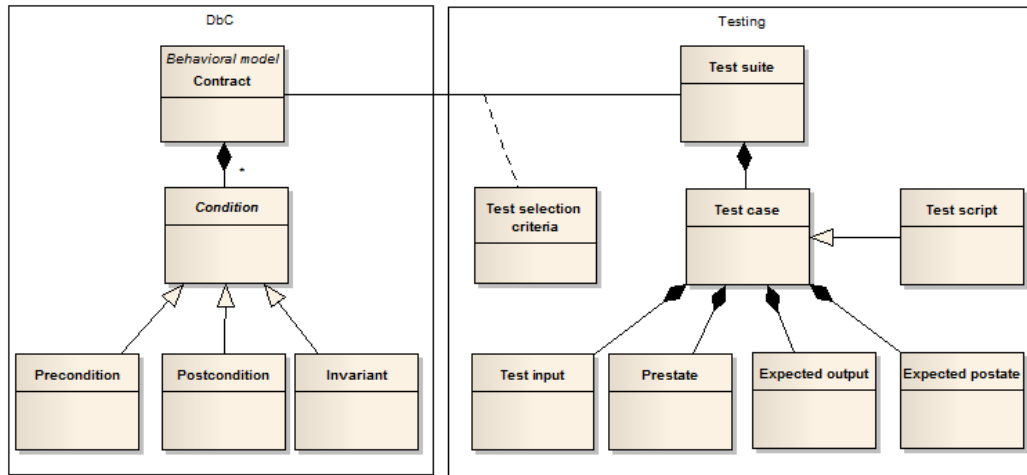


Figure 3.2: Test case in contract-based testing

Figure 3.2 shows the testing concepts related to the DbC concepts from Figure 3.1. A test case comprises test inputs which are typed over the input parameters of the operation under test and which conform to the precondition. Test inputs define the collection of input parameters, which are needed for the invocation of the operation under test. Test script defines, how the test case is applied on the operation under test, i.e. how the operation under test is invoked using the test case.

Having a test definition, which is specific for the contract-based testing, we can define as next the contract-based test process. The following steps are required for generating test cases from contracts and to execute them:

1. Define a test selection criteria
2. Generate test inputs from precondition
3. Invoke the operation with the test inputs
4. Check the fulfillment of the postconditions

The first step is about specifying how to select test cases from the contract. For that well-known selection techniques from testing literature are used, e.g data coverage criteria [AS05]. Secondly, using the test selection criteria, test inputs are generated from the precondition. The operation under test is invoked using the test inputs and the output parameters are checked for conformance with the postconditions. Thus postcondition of the contract is used as test oracle. In case of inconsistencies, the error can be in the contract itself or in operations's implementation, or something has gone false during the test execution.

## 3.2 Approaches in the Literature

Testing using contracts is studied extensively in the literature. In this section, we will give an overview on the approaches in the literature and explain how these approaches use contract specifications for testing. We will also discuss their strengths and shortcomings. Thereby, we will characterize and compare the approaches using the following dimensions: general characteristics (based on [VB05]), MBT-specific characteristics (based on [DNT09]), CBT-specific characteristics (based on own literature survey).

### General characteristics

Vegas and Basili have defined in [VB05] a characterization schema for selecting testing techniques. We have adopted some of their characterization attributes for CBT and grouped them logically as shown below.

- People
  - Knowledge
  - Experience
- Process
  - Application domain
  - Development method
  - Relation to a programming language
  - Test level
  - Test object
- Quality
  - Testing type
  - Test objective
  - Defect type (fault model)
  - Test selection criteria
  - Number of generated cases
  - Effectiveness
  - Completeness
  - Type of evaluation

The group *people* is about the human aspect in characterizing testing techniques. Testers should have some knowledge of special techniques and experience in certain development activities to use a technique. The group *process* contains characterization attributes for testing techniques which are related to the integration of testing into the development context. Thereby, the application domain (e.g. embedded software, business information systems) and the development method (e.g. RUP, V-model) has a major influence on test techniques and test organization. The development method can include different test levels (e.g. unit testing, system testing) handling different test objects (e.g. classes, subsystems). Especially in low level tests, the technologies used for testing can be directly related to the programming language used for the development of test objects.

The group *quality* is about qualitative and quantitative characteristics of test techniques. Test type addresses whether testing is conducted in a black-box manner (functional) or in a white-box manner (structural). Another characterization issue is the test objective which addresses the quality to be tested (functional or non-functional quality). Some test techniques are meant for detecting certain defect types (e.g. defects in control flow or data flow). In order to find these certain defects, the testing technique should define how a set of test cases is to be selected from the test basis (e.g. data coverage, requirements coverage). Choosing a formal selection criteria has the advantage that the test suite is generated in a replicable way: the number of test cases can be predicated in early stages of the test process which improves the cost estimation for test process. The test selection criteria influences also the the effectiveness and completeness of test process, i.e. how many defects can be found and how much coverage on the test basis or on program code can be achieved. For characterizing the quality of a test technique, the experiences with this technique in former projects can help. Dias-Neto et al. differentiate in [NSV<sup>+</sup>08] between test techniques which are evaluated in different context (e.g. simple example, industrial projects).

### **MBT-specific characteristics**

In addition to the general characteristics listed above, we will consider also MBT-specific characteristics for explaining and comparing the existing test techniques related to our approach. For that, we adapt the characterization schema of Dias-Neto et al. given in [DNT09] where they conducted a detailed survey on more than 400 publications. The schema of Dias-Neto bases on the schema of Vegas and Basili, that is why there are some overlapping between the characterization attributes of both approaches. We have grouped the characteristics of Dias-Neto again in four logical groups as shown below:

- Artifact
  - Notation
  - Inputs
  - Outputs
- Redundancy
  - Shared models
  - MBT scenario
- Automation
  - Tool support
  - Test case generation (logical)
  - Test case generation (executable)
  - Test execution
  - Test evaluation (oracle)
- Tool
  - Environment
  - External tools
  - Technology used
  - Extensible

As we have seen in section 1.3, main artifacts in MBT are test models. MBT-techniques differ in the notation (e.g. UML, OCL) they use to edit test models. Different techniques also require various input artifacts (e.g. behavioral models, environment models, test data) and produces various outputs (logical test cases, executable test scripts). In different MBT techniques, different scenarios (common models, separate models) are realized, where test models can either be shared between developers and testers, or testers create their own models. MBT techniques also differ in the automated tool support. Different testing activities as explained in section 2.3 can be supported by tools. In the literature there are various tools and technologies used in MBT. Tool characterization is important if MBT techniques are to be integrated in existing tool landscape.

### CBT-specific characteristics

In addition to the general and MBT-specific characteristics, CBT approaches exhibit other characteristics regarding the usage of contracts and the structure of test cases. The quantitative relation (1:1, 1:many, many:1) of contracts and software addresses how many contracts specify a test object. Sometimes a contract can specify many test objects, e.g. in case of inheritance. CBT techniques can also differ regarding the relation and the usage of pre- and postcondition and invariants for testing activities. The modus of contract addresses whether it is used as a declarative or an operative description. Depending on the characteristics of contracts, the generated test artifacts can also differ. Some approaches just produce test inputs, while some others additionally generate prestates and expected values.

- Contract
  - Relation contract/software
  - Relation pre/post
  - Usage of invariants
  - Modus
- Test case
  - Input parameter computation
  - Prestate computation
  - Preamble computation
  - Expected output computation

In the following sections we will characterize existing CBT approaches using the characterization scheme given above. A summarizing tabular comparison follows in Figure 3.3. For the sake of simplicity and if applicable, we title the sections with the name of the tools, which are implemented by these approaches.

#### 3.2.1 AutoTest

The first approach we want to report on is developed by Ciupa during her dissertation [Ciu08] at the chair of Bertrand Meyer at ETH Zurich. Meyer already addressed the role of software contracts in testing in his invited talk at RISE 2005 [Mey05]. Ciupa developed an automatic testing approach based on the Design by Contract for Eiffel programs. This approach uses embedded



contracts for testing classes' routines. Thus, this approach can be assigned to the level of unit testing. The contracts are used for two purposes by this approach: for generating test inputs from preconditions, and for validating test results by postconditions and invariants.

For test case generation, Ciupa differentiates between a *constructive* way and a *brute force* way. Constructive way is about generating test input objects in a natural way by using the routines of the class under test for setting the attributes of these objects. The brute force way is about setting the objects and their attributes directly without using the natural functions of the class under test. Ciupa applies a constructive approach for test case generation. For generating test input objects, she randomly invokes constructors of the classes and stores the generated objects in an object pool. The attribute values are either assigned predefined values or they are generated randomly. The objects in the object pool can be reused for different test cases, or new objects are generated if needed. Using an *object distance function*, the *diversification* of test input objects is guaranteed, which means that the test cases are dissimilar to each other such that more coverage can be assured by test execution. This approach also involves a *minimization* step for the number of test cases [Ciu08].

For the validation, the contracts can be seen as an executable specification. Any failed test case can signal an error in the implementation of the class' routines. However, the error can also be in the contracts it selves or in the execution environment of the class' routines.

Their tool AutoTest, which was developed together with Andreas Leitner, is fully functional including the test case generation, execution and evaluation of the test results. Additionally, it has an extensible architecture, such that other test selection strategies can be added to AutoTest.

As stated above, this approach supports unit testing activities where programmers can specify the contracts of class' routines and use them for testing the routines' implementation. Creating the contracts, which are specified in Eiffel language, requires low level programming skills, which makes this approach not suitable for model-driven development. The constructive way for test case generation as explained above can be good for unit testing, where the particular behavior of a single class' routine is examined. However, for higher test levels, e.g. integration testing or system testing, more sophisticated test generation techniques are needed. For example, the generation of test input objects may require more computation by class' routines and also more interaction of classes such that useful objects can be generated for testing.

Testing activity	JUnit	JMLUnit	Korat+JMLUnit
generating test inputs			+
generating test oracle		+	+
running tests	+	+	+

Table 3.1: Functions of related tools to Korat (based on [Mar04])

### 3.2.2 Korat

The Korat tool [BKM02, Mar04, MSM<sup>+</sup>07] uses JML specifications for test case generation for Java classes. Thus again, this approach is about unit testing. This tool assumes that imperative predicates specifying the class invariants and the pre- and postconditions of the method under test are given. It generates all non-isomorphic inputs up to a given bound from the precondition. Korat analyzes attribute values which can influence the evaluation of the precondition and only looks for alternative input values which do not violate the precondition. Because it sets attribute values directly (rather than building test inputs through sequences of creation procedure and routine calls), Korat must check that every generated input object fulfills its class invariant. If this is not the case, the input is invalid and it is discarded.

In [Mar04] Marinov explains the relation of the Korat tool to JUnit and JMLUnit (see table 3.1). JUnit is the widely used test driver which runs test scripts programmed in Java using the JUnit library. JMLUnit additionally can handle JUnit test scripts including JML assertions. The JML assertions can be used as test oracles. If the assertions are not fulfilled, JMLUnit catches the JML exceptions and reports them as failed test runs. On top of these two tools, Korat supports the test input generation, thus enabling a full scale test automation.

A major disadvantage of this approach is that the test inputs are only set artificially (also called *brute force* generation of test inputs by [Ciu08]). Generation of test inputs in a natural way by invoking a sequence of operations is not handled. Another disadvantage is that the invalid test inputs are just ignored. However these test inputs could be used for *robustness testing*. Also the low-level JML language and the usage of first-order logic for creating the contracts require high skills of testers in programming and in theoretical computer science.

### 3.2.3 WeSUF

Averstege and Winter show in [AW05] how predicate testing techniques known from white-box testing can be adapted to black-box functional test-

ing. They apply predicate coverage techniques on OCL contracts for both structural and functional testing for Java classes. This work is also inspired by design-by-contract and addresses invariants, preconditions and postconditions of Java classes for test input generation.

The authors motivate their work with the similarity between the techniques for predicate coverage and for contract coverage. They apply classical techniques known from white-box testing like branch coverage (C1), simple condition coverage, multiple condition coverage (MC) and modified condition/decision coverage (MC/DC) techniques on contracts and exploit their possibilities for functional testing.

For the sake of simplicity this approach only focuses on testing class operations. The authors define a testing strategy with the following testing order for operations: (1) constructors, (2) simple observers, (3) simple modifiers, (4) complex observers, (5) complex modifiers, (6) destructors. The test inputs for the operators are divided into positive tests, which are generated in conformance with the preconditions, and into negative tests, which are inconsistent with the preconditions. For each test input, the expected fulfillment of the postconditions must be computed manually.

The authors have implemented a tool support called WeSUF. It has a multi layer architecture and supports test generation for both structural and functional testing. A specialty of this tool is that it optimizes the set of test input data by checking the tautologies and computing a minimal set test inputs.

### 3.2.4 LTG/B

Utting and Legiard propose to use the B language for contract-based testing. Traditionally, the B language is used in B method for refinements and proofs to generate a correct implementation. In [UL07], they use the B abstract machine notation for model-based testing. Like the complete B method, the B notation is used to define an abstract model of the system under test. This model is a partial model, written just for testing purposes, rather than a complete model of the desired system. They assume that the implementation has been coded manually in some programming language and is likely to contain errors. They use model-based testing tools to automatically generate a systematic set of test cases from the model. When these tests are executed on the implementation, they expose differences between the behavior of the implementation and the behavior of the model. Each difference can address either an error in the implementation or in the test model. This can be the case because of an unclarity in the original informal requirements of the system. [UL07]

Even if the authors find the usage of B method lightweight, it requires low-level skills of testers to create the contracts in B language.

### 3.2.5 WSTVC

The most related work to the underlying dissertation is the body of work of many researchers from the Universities of Paderborn and Leicester on Web Service Testing using Visual Contracts (WSTVC).

The basic idea of testing with Visual Contracts was first presented by Reiko Heckel and Marc Lohmann at the University of Paderborn in [HM05]. The authors use graph transformation rules for test case generation for testing web services. Test cases are generated from the behavioral specifications and executed using a dedicated testing interface to ensure the correct functioning of the web service [HM05]. In a later work [LMH07] Lohmann, who has developed Visual Contracts, contributes to the work presented in [HM05] by using Visual Contracts. In this contribution (1) the authors extend and apply domain-based testing and data-flow techniques to the case of graph transformations, (2) they generate executable test oracles from graph transformation rules and (3) they automatically test and validate web services. However some differences still exists. For example this approach misses a precise definition and implementation of finding invocation sequences for setting the system state in a black-box system.

The most recent work related to this thesis is the dissertation of Tamim Ahmed Khan advised by Reiko Heckel from the University of Leicester [Kha12]. He has carried forward the research on testing of web services using Visual Contracts. Thereby Visual Contracts are used as a specification for web service operations and thus as the test basis. Even if this work addresses systematic test input generation from Visual Contracts, the focus lies in using them as test oracles. For that the Visual Contracts are used as a formal executable specification for determining the expected outputs for a given test input. Besides test oracles, this thesis also describes how Visual Contracts can be used for dependency analysis and for regression testing.

The main difference between the addressed thesis and the underlying thesis is that we focus more on test input generation for different test levels in the development process. The focus of [Kha12] is the usage of Visual Contracts as test oracles. We also different scenarios in creating Visual Contracts. While [Kha12] reuses existing Visual Contracts for testing, we also describe a scenario where Visual Contracts are created by testers separately.

### 3.3 Tabular Comparison of Approaches

In this section, we summarize the characteristics of CBT approaches addressed in last section and put them into a table. In this way the various approaches can be compared. From the comparison, we want to identify the problems of the existing approaches. Table 3.3 shows in the rows the characteristics introduced in the beginning of this chapter and the CBT approaches from last section in the columns.

Regarding the characteristics *People*, all of the approaches require high skills from testers in programming-like activities and a very good knowledge on using predicates for specifying software. The approaches are designed for very different *process* context. However, many of them are related to OOP (object-oriented programming). Most of the approaches are designed for low test levels like unit testing. As a consequence, test objects are mainly classes and their methods. Regarding the *quality* characteristics, the approaches differ very much. Besides classical use of CBT for black-box testing, some of the approaches also use program code additionally to the contracts for test case generation, thus we speak of white-box testing. Most of the approaches use well-established test selection criteria for test generation, e.g. random testing, MC/DC. The evaluations of the approaches mostly depend on proof-of-concept and only some of them have industrial evaluations.

Regarding the MBT-specific characteristics, many approaches use the process model common models, where the developers models are used for test models. This has the disadvantage that some of the specification failures cannot be found, because the developers and testers use the same information as basis for their activities. The notations used for creating contracts are mainly low-level notations, e.g. Eiffel, JML, OCL, which demands low level skills from testers. Many of the approaches supply automated support. However, not all of the MBT activities are supported. Especially, the generation of logical test cases is mostly not supported by the approaches. They mostly generate directly executable test scripts. The technologies used in tools vary from search algorithms to optimization algorithms. Most of the tools are extensible.

Regarding the CBT-specific characteristics, in most of the approaches there is a 1-to-1 relation between the contract and the test object. The contracts are mostly used as declarative specification of the test object's behavior. Testers generate test inputs from these contracts for invoking the test object. However, these test inputs are artificial ones. There is no guarantee, that using these test inputs will test realistic scenarios.

		AutoTest [Ciu08]	Korat [MSM+07]	WeSUF [AW05]	MBTVC [Khan2012]
<b>General characteristics</b>					
People	Knowledge	DbC	First-order logic		DbC
	Experience	Prog.	High Prog.	Prog.	Modeling, testing
Process	Application domain	n.A.	Complex data structures (linked data structures with complex invariants)	Web Services	Web Services
	Development method	OOP	OOP	OOP	OOP
	Rel. program. language	Eiffel	Java	Java	Java
	Test level	UT	UT, ST	UT, ST	CT, IT
	Test object	Class methods	Class methods	Class methods	Web Service operations
	Quality	Testing type	WB	BB, WB	BB
	Test objective	Functional	Functional	Functional	Functional
	Effectiveness				n.A.
	Completeness		high statement, branch, mutation coverage	statement, branch, predicate coverage	?
	Defect type (fault model)	Specification faults (...), implementation faults (...)		Wrong or missing or superfluous operators, wrong variables	
	Test selection criteria	Random, object distance	bounded non-isomorphic inputs	CI, MC, MC/DC	
	Number of generated cases	n.A.	depends on the given bound in finitization method		
	Type of evaluation	III-IV	III-IV	II	
<b>MBT-specific char.</b>					
Artifact	Notation	Eiffel	JML	OCL	
	Inputs	Contracts, classes, manual tests	Imperative predicate, finitization	Predicates	
	Outputs	Optimized set of test input objects	set of non-isomoprhoc test inputs	Optimal set of logical values	
Redundancy	Shared models	No	No	Yes	
	MBT scenario	Common model	Common model	Common model	Common model
Automation	Test case generation (logical)	No	No	Yes	(Yes)
	Test case generation (executable)	Yes	Yes	No	Yes
	Test execution	Yes	Yes	No	Yes
	Test evaluation (oracle)	Yes	Yes	Yes (manual)	Yes
Tool	Tool support	Yes	Yes	Yes	Yes
	Environment	See <a href="http://se.inf.ethz.ch/people/leitner/auto_test/">http://se.inf.ethz.ch/people/leitner/auto_test/</a>	Java		
	External tools	??	Alloy analyzer for vuzualization of test inputs, JMLUnit as test driver	SableCC for OCL parsing, Barat for Java parsing, DoIT for code instrumentation	AGG
	Technology used	Adaptive random testing for OO, diversification	Bounded-exhaustive testing by constraint solving	tautology checking, normalization of predicates	Graph tnasformations
	Extensible	Yes	Yes (pure Java)	No	?
<b>CBT-specific char.</b>					
Contract	Rel. contract/software	1--1	1--1	1--1	1--1
	Rel. pre/post		Yes ("old" operator)	Yes ("@pre" operator)	Yes
	Usage of invariants	Yes	Yes	Yes	No
	Modus	declarative	declarative	declarative	declarative, operative
Test case	Input parameter computation	Yes	Yes	Yes	Yes
	Prestate computation	Yes	Yes?	Yes?	Yes
	Preamble computation	No	Yes? (no evidence)	No (however a	
	Expected output computation	No	No	No	Yes

Figure 3.3: Tabular comparison of CBT approaches

## 3.4 Summary

Our comparison shows that the current approaches for CBT have some shortcomings. First, all of them address low-level testing activities and use low-level notations for modeling contracts, which are not conform to the languages used by testers, which makes them hard to learn. The notations used by these approaches are very specific for their application domain and they cannot be very well integrated into a standard UML-based development process. In most of the approaches, testers use the same contracts as the developers have specified for implementation purposes. These contracts contain low redundancy for detecting errors and may miss test relevant aspects. Finally, the test inputs generated from the contracts are artificial test inputs, which makes realistic test scenarios impossible.

In order to handle these shortcomings, we propose a novel contract-based testing approach using Visual Contracts [Loh06]. Before this approach is explained in Part II in detail, we briefly explain the language of Visual Contracts in the next chapter.





# Chapter 4

## Visual Contracts

In chapter 3 we have addressed some contract-based modeling approaches and their usage for contract-based testing. In the summary, we have seen that one of the shortcomings of existing approaches is the usage of low level contract notations. These notations are not conform to the languages used by testers, which makes them hard to learn (cf. section 3.4). Another shortcoming of these notations is that they can hardly be embedded into the UML-based development process.

In this chapter we will introduce Visual Contracts [Loh06], a visual modeling technique based on the idea of Design-by-Contract [Mey92], and explain how the shortcomings addressed above can be eliminated. Developed by Marc Lohmann [Loh06], Visual Contracts allow functional modeling of software by specifying pre and postconditions. The novelty of Visual Contracts is that they are UML-based and have a formal semantics. In this chapter, we will explain the characteristics of Visual Contracts on a simple example and address the application areas.

### 4.1 Modeling with Visual Contracts

In MBSO many modeling languages are used for describing the structure and the behavior of software. As shown in section 2.2.1, UML serves with many notations for describing structure by means of entities and architecture and for describing behavior by means of functions, interactions and states [Ros09]. The vision of MBSO is just to deal with models for specifying software characteristics and to automatically generate program code from models. In the past years, many techniques and tools are developed for fulfilling this vision [KBJV06], e.g. MDA by OMG [Gro03a], DSL by Microsoft, EMF/GEF by Eclipse community. However, all these techniques

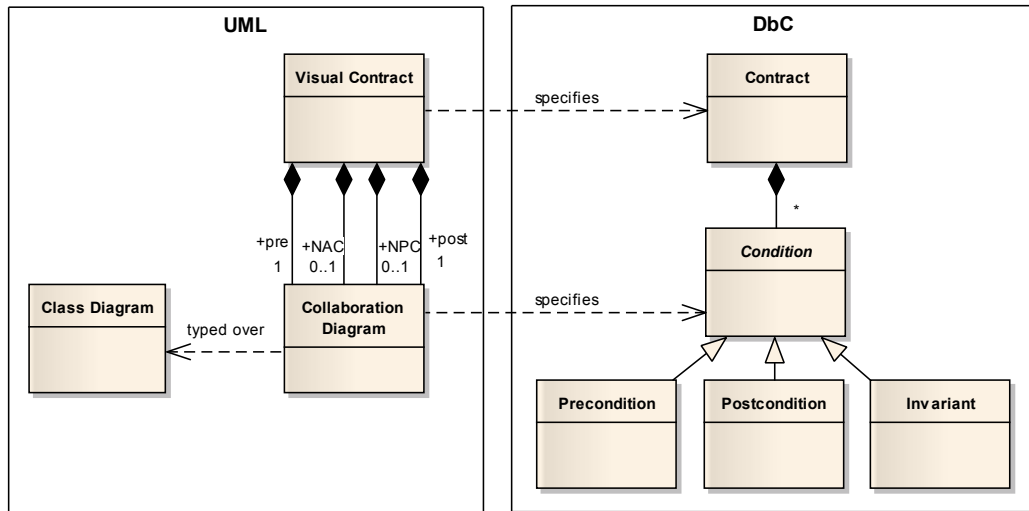


Figure 4.1: Visual Contracts embed DbC into UML

and tools require complete behavioral models in order to generate functional program code. Creating complete behavioral models can be a very tedious and error-prone work such that the efforts for modeling could be as much as the efforts needed for manual programming.

In our research, we prefer a partial modeling technique, where we do not model the behavior completely, instead we model the effects of single functions on the system state. For modeling we use Visual Contracts which are developed by Lohmann [Loh06] based on the Design-by-Contract (DbC) [Mey92] paradigm. Visual Contracts allow for modeling the requirements (*preconditions*) of a software function on the system state and its changes on the systems state (*postconditions*) partially. Thereby, preconditions and postconditions are represented by object structures which are modeled by UML Collaboration Diagrams which are typed over a UML Class Diagram (see Figure 4.1). The pre- and postconditions may also specify object structures, which are not allowed to exist before or after the invocation of the function. These are called *negative application conditions (NAC)* or *negative postconditions (NPC)*, respectively. For simplicity, Visual Contracts only model preconditions and the postconditions, but not the invariants as defined by DbC. By using Visual Contracts, we extend UML by the notions of DbC.

Figure 4.2 shows the concrete syntax of Visual Contracts. The outer frame of a Visual Contract contains on the top the *header* which shows the name of the function specified. If there are inputs and outputs of the function, they are also specified in the header. The body of the Visual

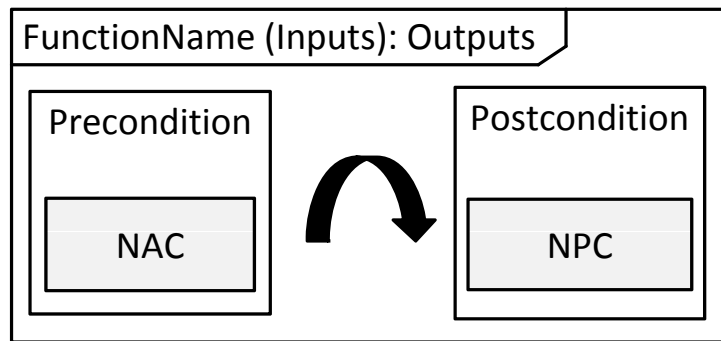


Figure 4.2: Concrete syntax of Visual Contracts

Contract contains two areas separated with an arrow. The left hand side of the arrow contains the precondition and the right hand side of the arrow contains the postcondition. While, the precondition can optionally include a NAC, the postcondition can optionally include a NPC [Loh06].

As next, we will define the semantics of Visual Contracts by means of a running example, which is simple enough to introduce the concepts intuitively. Then, the semantics of Visual Contracts will be defined formally.

#### 4.1.1 Running Example: Online Shop

The running example is a simple online shop enabling customers to start a shopping session, to add some products into the shopping cart and to clear the shopping cart. Figure 4.3 represents the required functionalities of the online shop using a UML Use Case Diagram. These functionalities shall be triggered from a GUI of the online shop which is not described here.

During requirements analysis, a domain model is created which shows the concepts of an online shop and their relations to each other. This step is followed by the design of the implementation model which refines the concepts from the domain model and maps them to an implementation model. In our simple example we jump over the domain model step and show as next the implementation model of the online shop.

The UML Class Diagram in Figure 4.4 shows the implementation classes of the online shop, where we differentiate between *controller* classes and *entity* classes. Controller classes are active classes which operate on and change the system state. The entity classes represent the real life objects in the application and are mostly responsible for storing the process relevant information. The functionalities of the applications are realized by controller classes by operating on the entity classes. In Figure 4.4 the class `OnlineShop` is the controller class and other classes are entity classes. The customer starts a

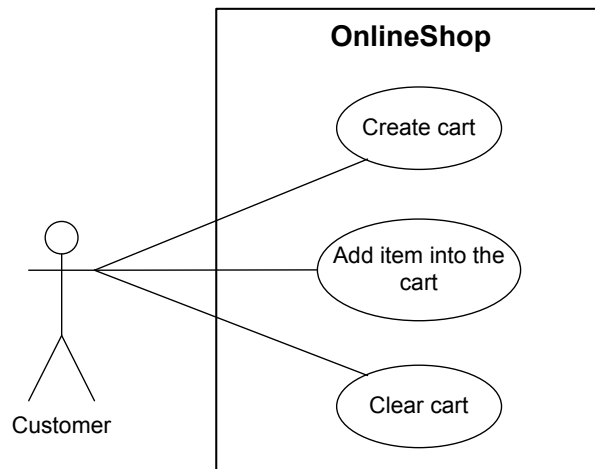


Figure 4.3: Use Case diagram for OnlineShop

shopping session by invoking the operation `cartCreate` of `OnlineShop` which then creates an instance of `Cart` class. The entity `Product` is an abstract class which is concretized by classes `Book` or `DVD`. A book can have one or many authors which are instances of class `Person`. A dvd can have one or many actors also instances of `Person` class. After the session is started the user can add products into the shopping cart. This functionality is implemented by the operation `cartAdd` of `OnlineShop`, where for each added product into the cart an instance of class `CartItem` is created and linked to corresponding `Cart` and `Product` objects.

If we were the programmers of the online shop and would be asked to implement the scenario described above, we would have the problem that the Class Diagram above only describes the structure of the implementation classes. However, the functionality of the operations in a class are not specified using models, they are just described textually which can lead to misunderstandings and inconsistencies. Therefore, also the functional requirements should be described in a more formal way. As we have seen so far, UML also offers notations for functional descriptions, e.g. UML State Charts, Activity Diagrams or Sequence Diagrams (cf. section 2.2.1). As we have discussed in chapter 2 different modeling paradigms have their strengths and weaknesses. In our running example, we want to demonstrate how functional descriptions can be made using pre/post notations, particularly using Visual Contracts.

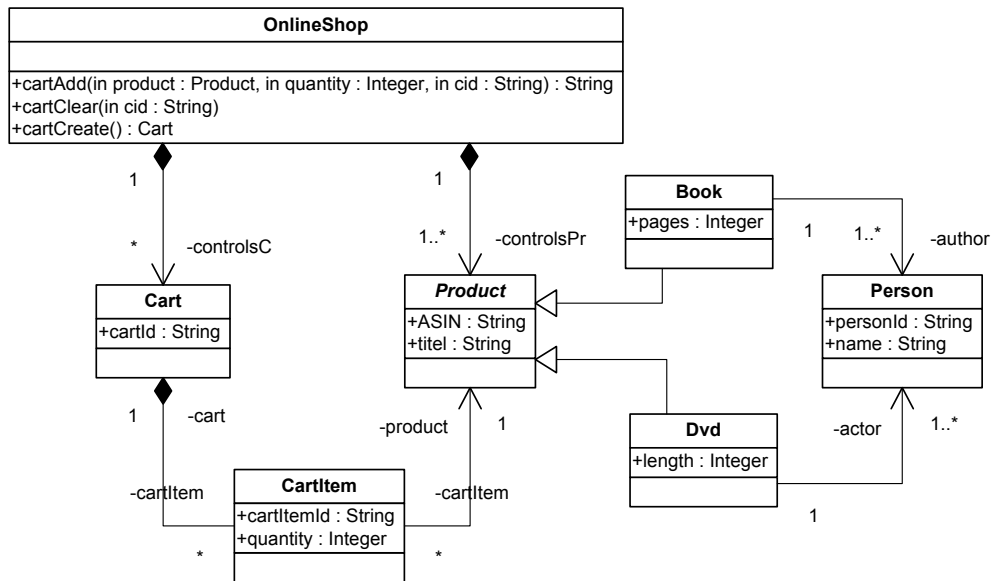
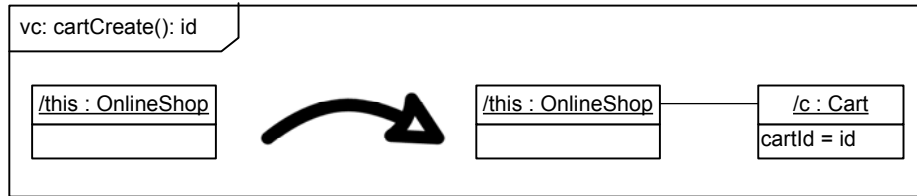
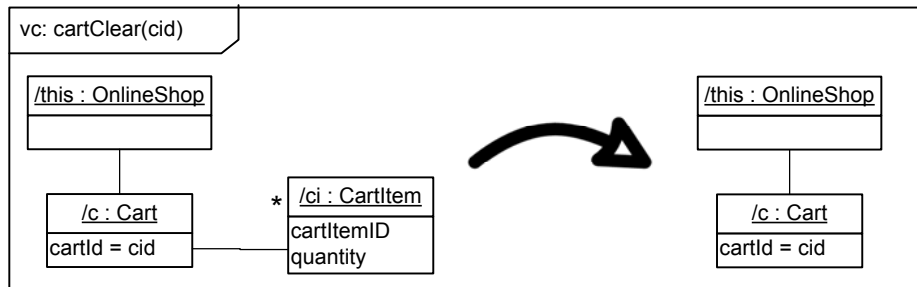


Figure 4.4: UML Class Diagram for implementation classes

### 4.1.2 Visual Contracts

First we demonstrate how the functionality of operation `cartCreate` can be described using Visual Contracts. This operation is responsible for creating an instance of `Cart`. The Visual Contract in Figure 4.5 illustrates this behavior using two object structures separated by an arrow (cf. Figure 4.2). The arrow represents a transition from a state to a new state. However Visual Contracts do not aim at describing system states in full detail. They describe just some parts of the system state which are of most interest for the programmers. The object structure on the left hand side of the arrow represents the conditions on the system state before the execution of the operation to be described [Loh06]. We call this condition *precondition* of the operation. The object structure on the right hand side of the arrow represents the *postcondition* of the operation, respectively.

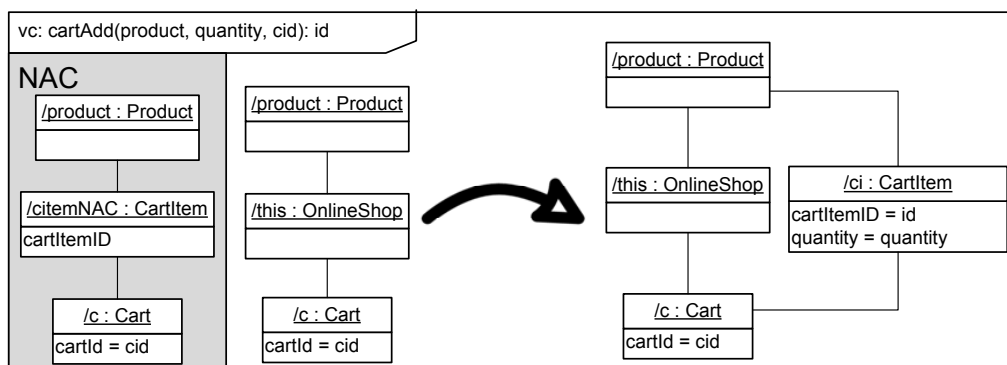
The precondition in Figure 4.5 shows a Visual Contract, where the only precondition of the operation `cartCreate` is the existence of an instance (`this`) of the class `OnlineShop`. This is a trivial case because every non-static operation requires an instance of the owner class. The postcondition includes additionally an instance of class `Cart`. The new object has an attribute `cartId` with a value `id`. The intuitive semantics of this Visual Contract is that during the execution of operation `cartCreate`, a new instance of the class `Cart` including a new `cartId` must be created. The new value of the attribute `cartID` is returned by the operation.

Figure 4.5: Visual contract for operation `cartCreate`Figure 4.6: Visual contract for operation `cartClear`

Next example of Visual Contract specifies the operation `cartClear`. Intuitively this operation should remove all items from the shopping cart. It makes just sense to invoke this functionality, if a shopping cart already exists and some items are already added into it. Figure 4.6 shows, how this scenario can be described using Visual Contracts in a more formal way. The precondition shows `this` object and an instance of `Cart` which is linked to a set of instances of type `CartItem`. These objects are to be removed during the execution of the operation. This behavior is specified by the postcondition which looks similar to the precondition; however all instances of `CartItem` are deleted.

The last example demonstrates how the operation `cartAdd` can be described using Visual Contracts. This functionality has to assure that a product selected by the user is put into the shopping cart, if it was not already in the shopping cart. Figure 4.7 shows, how this informal functional description can be formalized. The objects with the white background to the left of the arrow specify that in addition to the `this` object, an instance of `Cart` and an instance of `Book` or `DVD`, which are subclasses of `Product`, must exist.

The precondition also contains objects with a gray background. This part of the precondition is called *negative application condition (NAC)* and **prohibits** the existence of objects in the system state before the execution of the operation `cartAdd`. The NAC in Figure 4.7 specifies that no instance of `CartItem` should be already in the shopping cart.

Figure 4.7: Visual Contract for operation `cartAdd`

As shown in the postcondition, after the execution of `cartAdd` a new instance of `CartItem` must be generated and linked to the instances of `Cart` and `Book` or `DVD`.

Until now we explained how the operations can be specified using Visual Contracts in a more formal way than just with textual descriptions. As next we want to introduce the semantics of Visual Contracts in detail.

### 4.1.3 Semantics of Visual Contracts

As we have seen in the examples above, the pre- and postconditions in Visual Contracts are made up of object structures. The object structure in the precondition describes the requirements on the system state before the execution of a function. However this is not a complete description of the systems state, it is a partial description, which contains the most relevant objects. After the invocation of the specified function, the object structure in the postcondition is required in the systems state. Thereby, the objects which are specified both in the precondition and the postcondition must exist in the system state after and before the invocation of the function. The objects specified only in the precondition, but not in the postcondition have to be deleted. The objects specified not in the precondition, which are however specified in the postcondition, have to be generated by the function.

The above description is an intuitive description of the semantics of Visual Contracts. Lohmann has also defined a formal semantics for the Visual Contracts [Loh06], which makes Visual Contracts to a more powerful modeling language with many possibilities for analysis. Since the preconditions and the postconditions of Visual Contracts are represented by UML Collaboration Diagrams, which are also graphs [Hec06], the semantics of Visual Contracts is defined by using graph transformations [CMR<sup>+</sup>97]. Heckel states

in [Hec06] the following about the graph theoretical interpretation of UML notations:

“These notations produce models that can be easily seen as graphs and thus graph transformations are involved, either explicitly or behind the scenes, when specifying how these models should be built and interpreted, and how they evolve over time and are mapped to implementations. At the same time, graphs provide a universally adopted data structure, as well as a model for the topology of object-oriented, component-based and distributed systems. Computations in such systems are therefore naturally modeled as graph transformations, too.”

Using graph transformations, the changes in the objects structure specified between preconditions and postconditions can be defined in a formal way. In the following we give some formal definitions of graph transformations based on [Hec06, BH02, Loh06, Che06, Han08] which will help to define the semantics of Visual Contracts in the next section.

A Visual Contract contains two object structures which can formally be defined by a graph.

**Definition 1 (*Directed graph*)** *A directed graph is a tuple  $G = \langle G_V, G_E, src_G, tar_G \rangle$  where  $G_V$  is a set of vertices and  $G_E$  is a set of edges.  $src_G : G_V \rightarrow G_E$  and  $tar_G : G_V \rightarrow G_E$  are relations which assign to each edge a source and a target vertex [Hec06].*

Since the objects in Visual Contracts are typed over a Class Diagram we need to differentiate between *type graph* (TG) and *instance graph*. TG specifies the concepts (type) and the instance graph specifies the concrete occurrences of these concepts. This relation is similar to the relations between XML schema and XML documents, data bases and data base schema [Hec06].

In addition to vertices and edges a graph may also contain attributes  $a : T$  of type T to store values  $a = v$ . Based on these terms, we can define the relation between an object  $o : C$  of type class C in the following manner [Hec06]:

- for each vertex  $o : C$  in the instance graph  $G$  there must be a vertex type  $C$  in the type graph  $TG$
- for each edge between objects  $o_1 : C_1$  and  $o_2 : C_2$  there must be a corresponding edge type in  $TG$  between vertex types  $C_1$  and  $C_2$



- for each attribute value  $a = v$  associated with a vertex  $o : C$  in an instance graph, there must be a corresponding deceleration  $a : T$  in vertex type  $C$  such that  $v$  is of data type  $T$ .

**Definition 2 (Graph transformation rule)** A graph transformation rule  $r : L \rightarrow R$  consists of a name  $r$  and a pair of instance graphs  $L$  and  $R$  typed over a type graph  $TG$ . Thus, the structures of  $L$  and  $R$  are compatible, i.e. vertices with the same identity in  $L$  and  $R$  have the same type and attributes, and edges with the same identity have the same type, source, and target. The left-hand side  $L$  represents the pre-conditions of the rule while the right-hand side  $R$  describes the post-conditions [Hec06].

**Definition 3 (Graph transformation)** A modification of a pre-state  $G$  resulting in a post-state  $H$  is called a graph transformation  $G \Rightarrow_{r(o)} H$  where the following steps must be performed [Hec06]:

- 1) Find an occurrence  $o_L$  of the pre-condition  $L$  in the graph  $G$ .
- 2) Delete from  $G$  all vertices and edges matched by  $L \setminus R$ .
- 3) Add to the resulting graph a copy of  $R \setminus L$ , giving the graph  $H$ .

The graph transformation given in definition 3 is based on the standard interpretation which is called double-pushout (DPO) [Hec06]. However, this interpretation is very strict such that the changes in the graph must follow the defined steps. However, the formalization of Visual Contracts requires a more flexible interpretation of graph transformations [Loh06]. That's why we use another interpretation of graph transformation, which is called *graph transition*.

**Definition 4 (Graph transition)** A graph transition  $G \rightsquigarrow_{r(o)} H$  allows deleting or adding more vertices and edges than specified by  $L \setminus R$  or  $R \setminus L$  [Loh06].

A graph transition generalizes the definition of a graph transformation given in 3 such that the deletion and addition steps (2) and (3) are specified in a more flexible way. This interpretation is called double-pullback (DPB) approach [Hec98]. Different to other graph transformation approaches, DPB uses a loose or flexible interpretation of graph transformations, which means that also other objects may be changed by the graph transformation which are not specified in the pre- and postconditions. This interpretation allows the programmers, who uses Visual Contracts as a specification for the functions under development, to implement more behavior than specified in the Visual Contracts.

After having introduced the syntax and the semantics of Visual Contracts in this section, as next we will give an overview on the application areas of Visual Contracts as defined by Lohmann [Loh06].

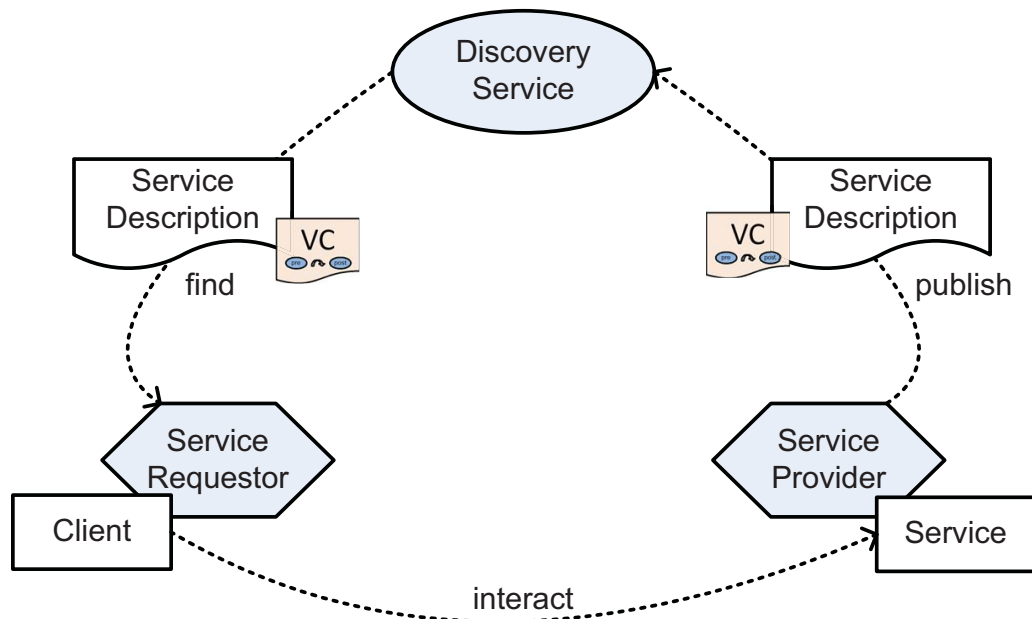


Figure 4.8: SOA triangle for service matching (based on [Loh06])

## 4.2 Application Areas

Visual Contracts are a visual language which integrates the Design-by-Contract paradigm into the UML-based modeling. Their formal semantics allow to conduct automated analysis on system which are specified by using Visual Contracts. In this section, we will give some examples on the different application scenarios of Visual Contracts in Model-based Software Development.

Visual Contracts are initially developed for modeling, implementing and searching web services in service oriented architectures (SOA) [Loh06]. Figure 4.8 shows the main concepts of a SOA. The main motivation was the lack of appropriate definition mechanisms for *service description* for service behavior. Even if the syntactical descriptions of services can be done formally using languages like WSDL [W3C14], the semantics of web service cannot be described by these languages in a formal way. Thus an automated finding and interacting of services is not possible. *A model-driven matching* approach developed by Lohmann solves this problem by specifying the service behavior using Visual Contracts. Thereby, Visual Contracts are embedded into the service descriptions of both providers and requesters. In this way, the searching and binding of services can be automated.

Having specified the behavior of services using Visual Contracts, they can also be used by programmers as a behavioral specification during the

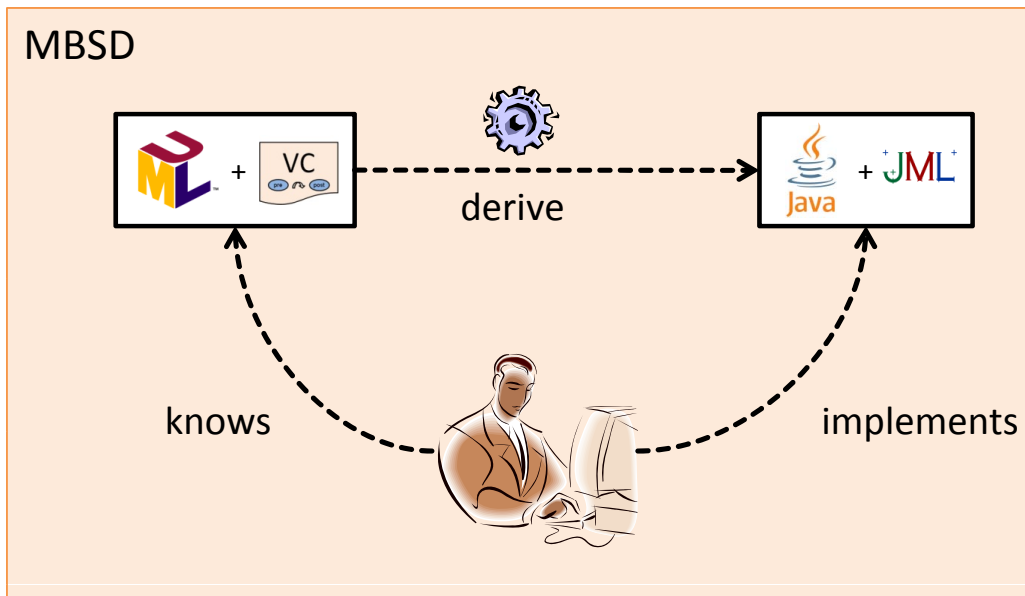


Figure 4.9: Model-based software development using Visual Contracts [Loh06]

development of the intended web service functions. As Figure 4.9 illustrates, Visual Contracts can extend the structural and behavioral specifications in an UML model by pre/post notations for individual functions. Lohmann proposed a *model-based software development* (MBSD) approach using Visual Contracts [Loh06], where program code in Java is partially generated from the UML Class Diagrams. Programmers extend these partial code manually by considering the functional specification given by Visual Contracts. Because of the loose semantics of Visual Contracts, the programmer can also implement other effects which are not specified by the Visual Contract (cf. section 4.1.3). In order to assure the correctness of the manually implemented functions, the Visual Contracts are converted to JML assertions and embedded into the Java code. After compiling them to an executable program, the embedded assertions can monitor the system state before and after the function invocation and give errors, if the state changes are not conform to the specification of Visual Contracts.

### 4.3 Experiences with Visual Contracts

Having introduced the language characteristics of Visual Contracts and their integration into the software development process, in this section we report

on some experiences with Visual Contracts from industrial projects and academic work.

Together with the company Capgemini (formerly sd&m AG) We have conducted a case study for modeling service behavior with Visual Contracts [EGL<sup>+</sup>06b]. Thereby, we have specified the behavior of over 40 services from the insurance domain. Before specifying the services, we have created an ontology for the German insurance domain (based on the [dDVe01]) using UML Class Diagrams. Together with colleagues from sd&m AG, we have analyzed the textual descriptions of services in this domain and created Visual Contracts for both conceptual and technical levels.

Our experience from the industrial projects has shown that

- Visual Contracts are very intuitive to use and they can be learned very quickly by newcomers.
- Given a proper type definition, Visual Contracts can be used for specifying the behavior on different abstraction levels. Thus, they can be used in different phases in the software development process.
- The loose semantics of Visual Contracts is helpful if designers do not want to specify the intended behavior completely, either because there are some lack of clarity in the requirements or because designers want to give programmers flexibility in making decision.

In our academic work we have also collected experiences with Visual Contracts and with different application scenarios explained in the last section. Lohmann has developed the Visual Contract Workbench (VCW) - an eclipse-based tool support for modeling and code generation with Visual Contracts [Loh06]. Using VCW, Visual Contracts and UML Class Diagrams can easily be edited and transformed into Java and JML code. After compiling these code fragments with a JML compiler, the runtime behavior of the software under consideration can be monitored. If the inputs used by invoking the software functions or the outputs of the software violate the pre- and post-conditions, VCW warns the user about an inconformity. Even this technique is helpful for assuring the functional correctness of software at runtime, it is a passive mean for quality assurance. Without a systematic and exhaustive trial of different input and output combinations, the errors can remain undetected and be delivered to the end-users. Even if these errors can be detected at runtime, the erroneous functionality can lead to dissatisfaction if the software misses its intended purpose.

“Neither behavioral nor timing contract testing can guarantee that all behavioral and timing failures are identified before the system is deployed. No

quality assurance technique is capable of doing that. Therefore, an additional quality assurance measure can be constantly carried out during runtime of the final system, and this is the assertion checking mechanisms that I briefly introduced in Chap. 4.”

These experiences motivated us to continue the research on Visual Contracts and to extend the MBSD process by a systematic testing process. We have also recognized that some of the challenges of contract-based testing as listed in chapter 3 can be solved by using Visual Contracts. Namely, they can be easily learned by testers and enable a high level UML-based communication between testers and other team members for various phases in the development process. Thus, we aimed to extend the MBSD process by a model-based testing (MBT) process using Visual Contracts (see Figure 4.10). The research question we want to deal with in the rest of this thesis is:

“How can we extend the MBSD process and enable a model-based testing process using Visual Contracts resolving the shortcomings of the existing contract-based approaches?”

In the next part will address our conceptual and technical solutions for resolving the shortcomings of existing approaches.

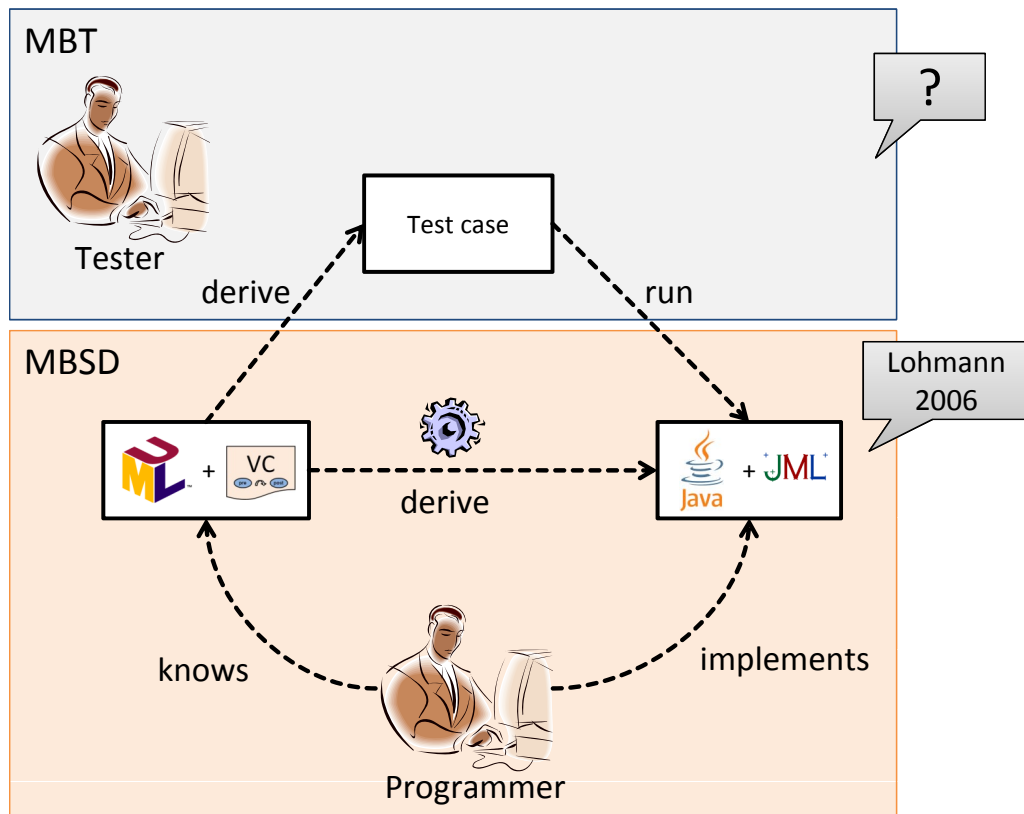


Figure 4.10: Integration of MBT into MBSD using Visual Contracts

# Chapter 5

## Summary of Part I

As illustrated in Figure 5.1, in the previous chapters of Part I, we have reported on the fundamentals of *model-based testing* (MBT) and on its special case *contract-based testing* (CBT). In the end of the chapter 3, we have compared the existing CBT approaches and identified some potential for further improvement. In chapter 4, we have described the Visual Contracts language which we will use to establish a novel testing approach called Visual Contracts-based Testing (VCBT) to reach the required improvements. In this chapter, we will summarize the challenges of MBT and CBT approaches as described in chapters 2 and 3 and define the requirements on an VCBT approach.

### 5.1 Improvement Potential in CBT Approaches

In chapter 3, we have motivated the use of contracts for testing and addressed various CBT approaches. The tabular comparison in Figure 3.3 shows a detailed comparison between the CBT approaches based on a big set of comparison criteria, which we summarize in the following sections. In this comparison, we have identified improvement potential regarding the following criteria:

- Relation to the development process
- Skills of testers and the selection of the modeling techniques
- Test levels
- Quality of test cases

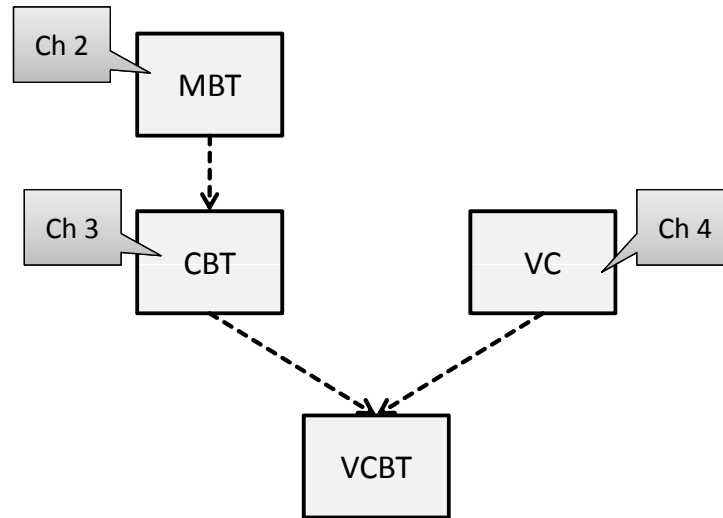


Figure 5.1: Overview on the building blocks of the requirements

### Relation to the development process

As testing must be an integral part in the development process, it is important to define the relations between the development and testing activities and the traceability between the development artifacts and testing artifacts. In section 2.4.2, we have addressed the *common model* and *separate model* approaches for MBT, which are totally different regarding the organization of the activities and the content of models [UPL06] in the following manner:

**Common model** (also *shared model*) proposes sharing the models between the developers and the testers, such that no additional efforts must be invested by testers for creating models as test basis. However, this approach lacks the redundancy which is required by test process to be able to find errors. As an alternative, **separate model** proposes totally independent development and testing activities, where testers create their own separate test models.

As shown in Figure 5.2, most of the CBT approaches we have studied in chapter 3 use the common model, such that the contracts created by developers for development purposes are reused by testers for testing purposes. However, the following problem may arise in the reuse of development contracts: The development contracts are created very late in the development process and so they may lack relevance to the initial customer requirements. If these contracts are used as a source for test cases, there may be a gap between the test results and the initial requirements. Such contracts are only suitable as a test basis for low level tests. Thus, if high level tests are



	AutoTest [Ciu08]	Korat [MSM <sup>+</sup> 07]	WeSUF [AW05]	MBTVC [Khan2012]
General characteristics → People → Test level → Process integration Quality	Low level UT No WB	Low level UT, ST No WB	Low level UT No BB	Low level UT,IT No WB
MBT-specific characteristics → Artifact → Redundancy Automation Tool	Eiffel Shared Yes Yes	JML Shared Yes Yes	OCL Separate Yes Yes	VC Shared Yes (Oracle) Yes (Oracle)
CBT-specific characteristics Contract → Test case	Declarative Artific. states	Declarative Artific. states	Declarative Artific. states	Decl./Imp.

Figure 5.2: Summary of tabular comparison of CBT approaches

needed, there is a need for an approach which allows testers creating their own contracts also earlier in the development process. In this way testers can decide on the abstraction level and on the desired content of test models for their testing purposes.

### Skills of testers and the selection of the modeling techniques

If testers should be able to create their own models as stated lastly, testers should also have the skills required for reading, creating or updating these models. The required skills include the knowledge on different *modeling paradigms* and on *modeling languages* suited to the characteristics of the software to be modeled. CBT approaches from chapter 3 all use the pre/post-based paradigm (cf. section 2.4.3) for specifying the changes in the data variables of a software function.

As shown in Figure 5.2, CBT approaches use different modeling languages for specifying contracts, e.g. Eiffel, JML, OCL. These languages are very low level ones which are perfectly suited for embedding them into the program code and to compile them together with the code in order to check the fulfillment of the contracts at run-time. However, these languages are too formal and not suitable for documenting and communicating the test specifications among the testers and also the customers. For that, more abstract and diagrammatic modeling languages are required.

UML, which contains general purpose modeling languages, have been

used for a long time for modeling test specifications. In CBT, there is a need to lift the contracts to the UML level and to integrate them into the UML-based development process.

### Test levels

The software development process based on the V-model as depicted in section 2.3 goes through different abstraction and detail levels. Accordingly, the test process defines mainly four test levels (unit testing, integration testing, system testing, acceptance testing) handling the different development stage and different granularity of software (cf. Figure 2.9).

As shown in Figure 5.2, most of the CBT approaches from chapter 3 address low level testing activities like Unit testing where contracts are specified for components or subsystems. However, there is also a need for CBT of data-oriented software on system level. Using similar modeling paradigms and modeling notations throughout the whole test process enables a uniform process and keeps the efforts for training testers low.

### Quality of test cases

The central activity in testing is the definition of test cases based on the test basis. In MBT, test cases are created from models either manually or automatically, but in both cases systematically by applying a model coverage criterion. Ideally, test cases should both include test inputs and expected behavior. Thus, models should specify both the inputs of the software under test and its expected behavior. Various modeling paradigms and modeling languages have different abilities and language elements to describe these aspects.

In CBT, the requirements of a software on its inputs and the guarantees of the software regarding its outputs are modeled using pre- and postconditions. The pre- and postconditions are ideally suited for generating test inputs and expected outputs. As shown in Figure 5.2, many CBT approaches from chapter 3 use contracts for generating test inputs. The postconditions are used as a test oracle at runtime for checking the expected outputs. However, there are two problems with most of the existing approaches regarding the quality of test cases:

- There is no guarantee that the test inputs which are generated from the preconditions are realistic enough. They may lack further objects and states which would have been a part of the execution environment, because other operations invoked previously could create them. In order to test with realistic inputs, a realistic *test preamble* is required,

which invokes operations in a realistic order such that required test inputs are created before the invocation of operation under test.

- The postconditions are used as a generic *test oracle* for each test input generated from the precondition, which may cause unsound or insignificant test verdicts. Thus, further investigations for test evaluation would be needed. For decreasing the efforts of test evaluation, there is a need for computing individual expected outputs for each test input.

## 5.2 Requirements on a novel Testing Approach

Having summarized the improvement potential in existing CBT approaches, we impose now some requirements on a novel approach which should realize these improvements.

The chapters 2 and 3 addressed the challenges of MBT and CBT as summarized in the last section. Some of these challenges are directly handled by using the Visual Contracts language described in chapter 4. However, the novel testing approach has to address all the challenges pretended by MBT and CBT. In this section, we explain which requirements we impose on the novel testing approach for handling these challenges.

Based on the challenges described in the last section, we define the following requirements on the novel testing approach:

1. Requirement: The novel testing approach should use a light weight and intuitive notation for testers. Using this notation, testers should be able to document and communicate the test models among the test team and if needed with the customers.
2. Requirement: UML is the de facto standard in software engineering, thus the novel testing approach should be compatible with the UML based software development process.
3. Requirement: The novel testing approach should enable using both MBT scenarios: shared models or separate models. The novel approach should enable both to reuse the contracts created by developers or to create separate contracts only for test purposes.
4. Requirement: The novel testing approach should be applicable for different test levels. Thereby, the contract language should enable to specify the characteristics of software under test on different abstraction levels.

5. Requirement: The novel testing approach should enable to automate typical testing activities. The most central activities are the test design, test execution and the test evaluation.
6. Requirement: The novel testing approach should describe a procedure how test inputs which are generated from the preconditions can be set in a realistic way.

By using Visual Contracts (cf. chapter 4) as a modeling language and introducing proper testing techniques, we contribute to the field of CBT in the following way:

1. **Contribution to Requirements 1 and 2:** Visual Contracts are an UML based notation which uses two UML Collaboration Diagrams for specifying the pre- and postconditions. The Collaboration Diagrams are typed over an UML Class Diagram. Both of these notations are intuitive and easy-to-learn notations, which contributes to the Requirements 1 and 2.

However, the questions how to define the test inputs from the preconditions systematically and how to check the correctness of the test outputs compared to the postconditions remain unsolved. VCBT defines selection criteria and selection algorithms for generating test inputs from preconditions systematically. Furthermore, VCBT defines how postconditions can be used as test oracles.

2. **Contribution to Requirement 3 and 4:** The development scenario Model-driven monitoring introduced in chapter 4 describes the usage of Visual Contracts for specifying the behavior of the software and using them for runtime monitoring. The same contracts can be used by the testers for test case generation. This usage of Visual Contracts corresponds to the MBT scenario *shared models*. Model-driven matching, which is also introduced in chapter 4, shows the creation of Visual Contracts by different parties in the development process. Thus, also testers can create Visual Contracts for their own purposes enabling the MBT scenario *separate model*.

However, the testing scenarios using shared models and or separate models require informations of different detail levels. The current development scenarios with Visual Contracts do not address the separation of concerns between developer models and test models. VCBT shows how developers contacts can be used for testing purposes and how testers can create their own contracts for upper testing levels.

3. **Contribution to Requirement 5:** Model-driven monitoring enables automatic evaluation of test results at run-time. Thereby, the Visual Contracts are translated to low-level assertion code (e.g. JML or Spec#) and embedded into the program code (e.g. Java or C# respectively). After compiling the assertions together with the program code, the software under test can be invoked with arbitrary inputs. The conformance of the inputs with the preconditions and the conformance of the outputs with the postconditions can be checked by the embedded assertions automatically.

However, the questions still remain how test case selection, test execution and test evaluation can be implemented by tools and how these tools can be integrated into the development environment. VCBT introduces new tools and techniques in order to able the test case selection, test script generation and test execution and integrates these tools into the development environment for Visual Contracts.

4. **Contribution to Requirement 6:** In order to set the test inputs in a realistic way, a realistic execution context of the software under test must be defined. A realistic execution context is characterized as a realistic execution order of the operations of the SUT.

VCBT uses the the formal semantics of Visual Contracts based on graph transformations (cf. subsection 4.1.3) for computing the dependencies between operations specified by VCs. Then, these dependencies are used for defining a realistic execution order of operations resulting realistic test inputs.

In Part II, we will explain in detail how Visual-Contract-based Testing (VCBT) approach enables the above mentioned contributions.



**Part II**  
**Approach**





# Chapter 6

## General Approach

Having introduced the related work and motivated the need for a novel contract-based testing approach in Part I, in Part II we introduce our contract-based testing approach using Visual Contracts and its integration into the UML-based development process. The contribution of our approach lies in its generic treatment of all test levels (cf. section 2.3.1) and of the main test activities (cf. section 2.3.2) as required by the ISO/IEC 29119 [ISO]. Whereas other contract-based testing approaches address mainly low level tests (cf. table 3.3), we show how contracts created at different phases of the development process can be used for different testing levels. While other approaches mainly concentrate on how test cases are derived from contracts, we also show how these test cases are transformed into executable test scripts and how test execution and test evaluation can be supported by contracts. For assuring the generality of our approach, we adapt the notion of *component* by Gross [Gro05a] and develop uniform development and testing techniques for components with different functional granularity (cf. [Cri11, Blo11] and section 2.2).

Before introducing our contract-based approach for each test level in detail, in this chapter we characterize the general properties of our approach. Section 6.1 gives an overview on our UML-based development process with Visual Contracts and explains the *component* notion which enables a uniformed treatment of software with different granularity during both development and testing activities. Section 6.2 summarizes the semi-automated implementation process using Visual Contracts defined by [Loh06] (cf. chapter 4) and extends this process for better testability. Each one of the sections 6.3-6.5 explains a particular testing activity based on the fundamental testing process [ISTQB]: First, we describe how abstract test cases can be derived from Visual Contracts using a formal selection criteria. Section 6.4 explains how the abstract test cases are transformed into executable test scripts. Fi-

nally in section 6.5, we show how the test scripts are executed and how embedded assertions act as test oracles for evaluating the test results. Section 6.6 summarizes the general concepts and techniques introduced in this chapter.

## 6.1 Development Process Overview

Our development process is based on the modified V-model by Basili and Pezze [BP06], where implementation artifacts are explicitly named and associated with design and test levels (see Figure 6.1). Vertical and horizontal dimensions, as described in [Gro05a] and [Cri11, Blo11], represent the *abstraction*, *decomposition* and *granularity* relations between the development artifacts (cf. section 2.2).

In the left branch of the V-model, design activities begin with creating abstract specifications of the software system and its business-level functionalities. These specifications are decomposed step-by-step into more concrete specifications for the subsystems and components with fine-grained functionalities. On the right branch, the components are implemented and integrated into subsystems and finally into the deliverable system with coarse-grained and business-level functionality. In order to assure the correct functionality of each component, subsystem and the deliverable system with respect to their functional specifications, these are tested using test cases derived from these specifications. For automation purposes, test cases are transformed to test scripts, which can be automatically executed by test drivers.

During the design, we use behavioral and structural notations of UML for creating the functional specifications. A system specification contains Use Case models describing the business-level system functionalities. The terms and concepts of the desired system are specified by a Class Diagram. The standard and alternative steps of a Use Case are refined by Activity Diagrams. The system specification is refined into subsystems by decomposing the coarse-grained business-level system functionalities into fine-grained functional units. We use Component Diagrams and Class Diagrams for specifying the structure of the functional units and their interfaces. Finally, the subsystem specifications are decomposed into component specifications which make up the most low-level functional specifications. Component specifications contain Component Diagrams and Class Diagrams for specifying the structure of the functional units and their interfaces. In UML, the behavior of the implementation artifacts are typically specified by Statecharts or Activity Diagrams. However, these notations require mostly a complete specification of the system states and functional steps, thus making the specification

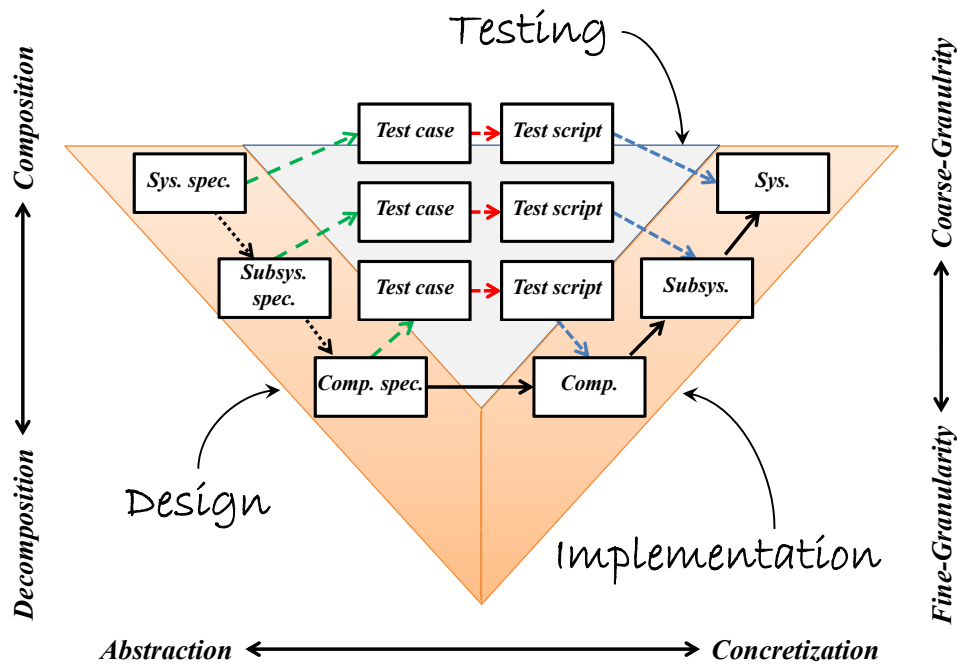


Figure 6.1: Extended V-model based on [BP06]

activity complex. Thereby, designers can forget modeling some states and functions due the component behavior, thus the completeness of the models cannot be assured. As an alternative to traditional notations for behavioral specification, we use Visual Contracts (VC) [Loh06], which enable specifying the behavior partially following the Design-by-Contract paradigm (cf. chapter 3). Thereby pre- and postconditions are used for specifying the required system state changes when system functions are executed. The programmer has total freedom for coding the system functions, as long as he or she ensures that the pre- and postconditions are fulfilled by the implementation. Besides traditional UML notations, we propose to use Visual Contracts for behavioral specification for components, subsystems and system. Thereby, we specify the provided and required interfaces of the software artifacts using Visual Contracts (see Figure 6.2). The components are implemented with respect to the Visual Contracts specifications and then stepwise integrated giving the subsystems and systems which also have to fulfill their Visual Contracts specifications.

The manual coding activities are error-prone which can induce software errors during the implementation and integration of the software components. Winter et al. classify errors in three groups (see Figure 6.3): component

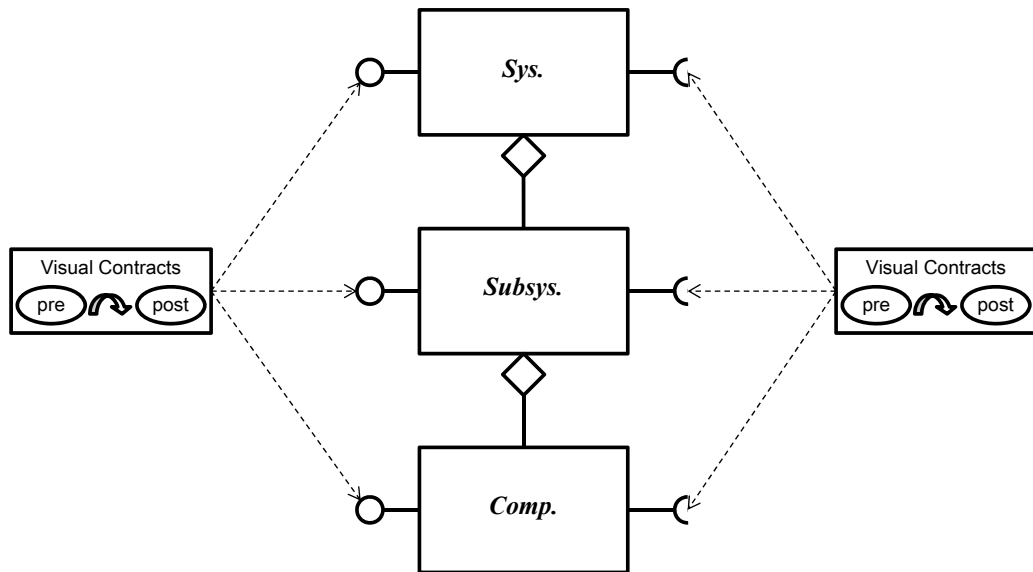


Figure 6.2: Implementation artifacts and their provided and required interfaces specified by Visual Contracts

errors, integration errors and system errors [WEMS<sup>+</sup>12]. In our test process, we are able to detect these kinds of errors by instrumenting Visual Contracts for test case generation, test execution and for test evaluation. Thus, after each implementation and composition step, our test process assures that the Visual Contracts are fulfilled by the implementation artifacts.

Figure 6.4 focuses on the central artifacts and activities of an implementation and testing phase during the V-model (cf. Figure 6.1), which we want to explain in detail in the next sections. First, we explain, how the implementation artifacts are developed based on the *Spec* including the Visual Contracts. Then, we explain our test design techniques for deriving *Test Cases* from the *Spec* systematically using formal selection criteria. For executing the test cases automatically, we define a transformation from test cases to executable *Test Scripts*. The implementation under test, which is called (*Test object*), is then automatically invoked by test scripts and its conformance to the *Spec* is checked. While explaining these activities in the next sections, we keep these as generic as possible in order to fulfill the requirements of ISO-29119 [ISO].

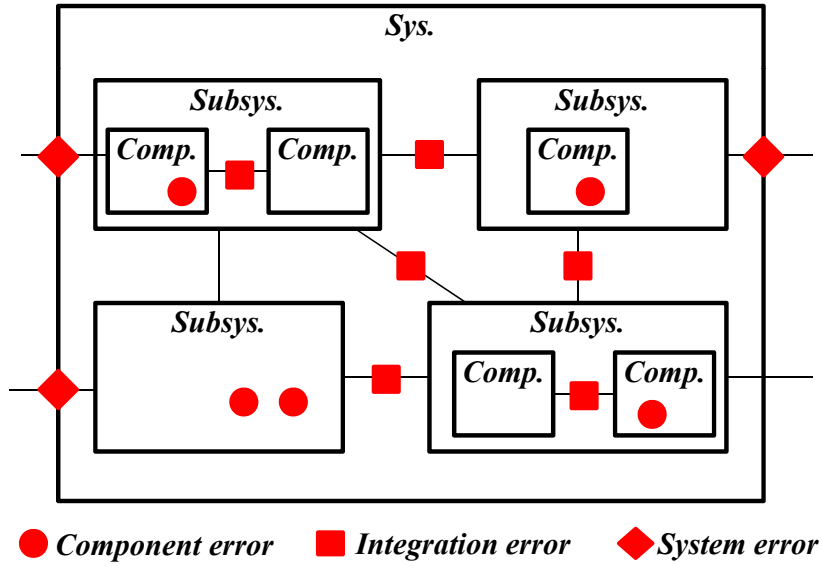


Figure 6.3: Classification of errors (based on [WEMS<sup>+</sup>12])

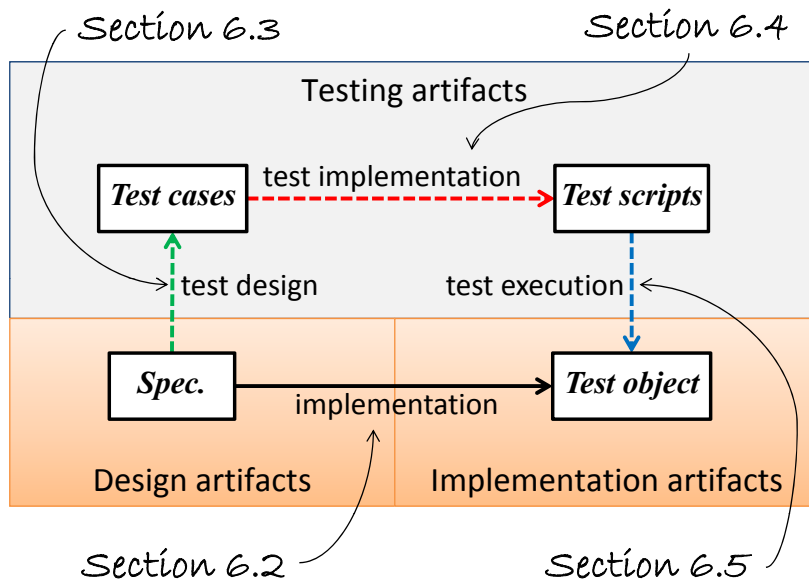


Figure 6.4: Overview on the activities of the general process

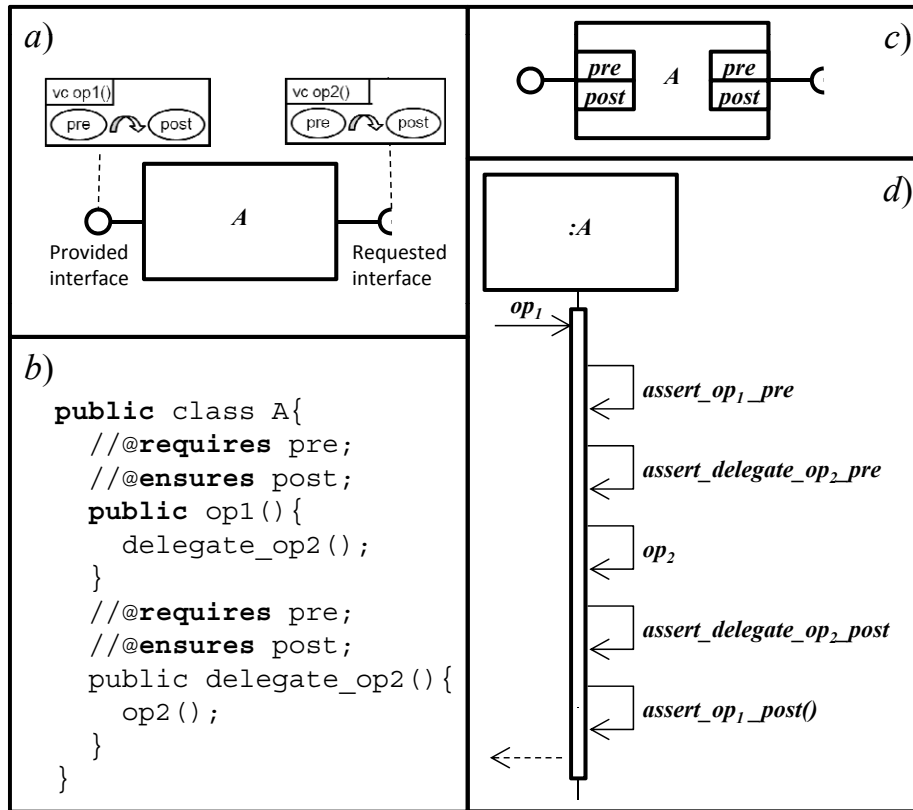


Figure 6.5: Implementation life-cycle

## 6.2 Implementation

During the implementation, we use the design specification incl. Visual Contracts for a semi-automated development of the *components* as described by Lohmann in [Loh06]. Thereby, we automatically derive class frames and method frames from the Class Diagrams. We complete these code frames with the functional code, for which we use the Visual Contracts as a specification. That is, we implement a functional code which should fulfill the pre- and postconditions in a Visual Contract. In parallel, the pre- and postconditions in the Visual Contracts are automatically translated into embedded assertions. If the generated assertions and the program code are compiled together into an executable binary, the embedded assertions act as runtime monitors during the execution and check the fulfillment of the contracts by the executed program. For our development process, we adopted and extended the implementation activities of Lohmann for better testability as illustrated in Figure 6.5.

First, we have extended the code and assertion generation by Lohmann (cf. Chapter 4) using the required interfaces (see Figure 6.5 box *a*). Besides provided interfaces, which specify the interaction with the *callers*, required interfaces specify the interaction with the *callees*. Thus, we propose the generation of additional embedded assertions for required interfaces. Box *b* in the Figure shows the generated code for component *A* including the generated Java operation frames and JML (Java Modeling Language [LBR06]) assertions for the required interface. Thereby, we use the *delegation pattern*, such that a *delegate operation* is generated, which forwards the calls to the actual operation. The Visual Contracts specify which conditions the outgoing invocations and incoming responses to/from the callee must fulfill. Box *c* illustrates the executable component and the embedded assertions for provided and required interfaces after the compilation of program code and assertion code. Box *d* shows the execution of the assertions, while the  $op_1$  calls the required operation  $op_2$  over the delegation operation. Thereby, when the provided operation  $op_1$  is called, the preconditions of  $op_1$  are checked by *assert\_op1\_pre*. If the condition hold, the required operation  $op_2$  is invoked. Thereby, first the preconditions of the delegation operation are checked by *assert\_delegate\_op2\_pre*, before the real operation  $op_2$  is called. The response of the operation  $op_2$  is first checked by the required assertions *assert\_delegate\_op2\_post*, before the delegation operation forwards these to the operation  $op_1$ . Before  $op_1$  response to its callers, the postconditions of  $op_1$  are checked by *assert\_op1\_post*. If during these checks of embedded assertions, the conditions do not hold, the execution is aborted and an exception is thrown, which will be explained in section 6.5 in detail.

In order to formalize the concepts explained above, we have created a domain model, based on the component meta-model by Gross [Gro05a], which shows the relations between the design and implementation artifacts (see Figure 6.6). The source for the automated code generation contain the Class Diagrams and Visual Contracts. A Visual Contract contains two Conditions, a Precondition and a Postcondition. The objects specified in conditions are typed over the Class Diagram, which means that these objects must be valid instances of a class in the Class Diagram. One or many operations in a Class Diagram are specified by a Visual Contract. From the design artifacts, implementation artifacts are generated semi-automatically. Thereby, class and operation frames for components, subsystems and systems are generated automatically from the Class Diagram, and the behavioral code for the operations are implemented manually with respect to the pre- and postconditions of Visual Contracts. Each software artifact has a Provided Interface and a Required Interface which contain Operations specified by Visual Contracts. An Operations have Input Parameter and Output Parameter. Operation has

two Assertions one for the Precondition of the Visual Contract and one for the Postcondition of Visual Contract.

The embedded assertions act as runtime monitors and validate the properties of the system state during the execution and throw exceptions, if these properties do not hold. This technique is called by Lohmann Model-driven Monitoring (MDM) [Loh06] and is an effective quality assurance technique for software at use. With the extension of the embedded assertions for the required interfaces, this monitoring technique becomes more powerful.

Even if MDM is able to detect errors during the execution of the system and to prevent system failures by exception handling, errors in the system during the usage, i.e. after the delivery of the software, may lead to dissatisfaction of users. In other words, it is desired that no or at least fewer errors exist in the software while delivery. Using testing techniques, errors can be detected before delivery. In the next section, we explain how Visual Contracts can be utilized for designing test cases.

### 6.3 Test Design

Each implementation artifact must fulfill its specification given by the design artifacts. In order to check the conformance to the specification, the implementation must be tested systematically against the design artifacts. The focus of our test process lies in testing of the state-changing behavior of the implementation artifacts, specified by Visual Contracts. How an implementation artifact should change the system state, is specified by the pre- and postconditions in Visual Contracts. If invoked in a state which conforms to the precondition, an implementation artifact must transform the system state to a new one, which conforms to the postcondition. Checking this conformity is the main target of our testing approach.

During the test design, test cases are derived from the Visual Contracts and Class Diagrams (see Figure 6.7). A test case contains an *initial state* representing an exemplary object constellations conforming to the precondition and *input parameter values* for the invocation of the *operation* under test. The test target is to check whether the operation under test changes the given initial state after the invocation in a way, such that the state changes conform to the postcondition of the Visual Contract. The objects in the initial state are derived from the precondition using a formal test selection criteria. Depending on the constructs in the precondition, different objects constellations can be considered. The objects have attribute values which have to be consistent with the input parameter values. The object constellations must also conform to the associations and attributes given in the Class



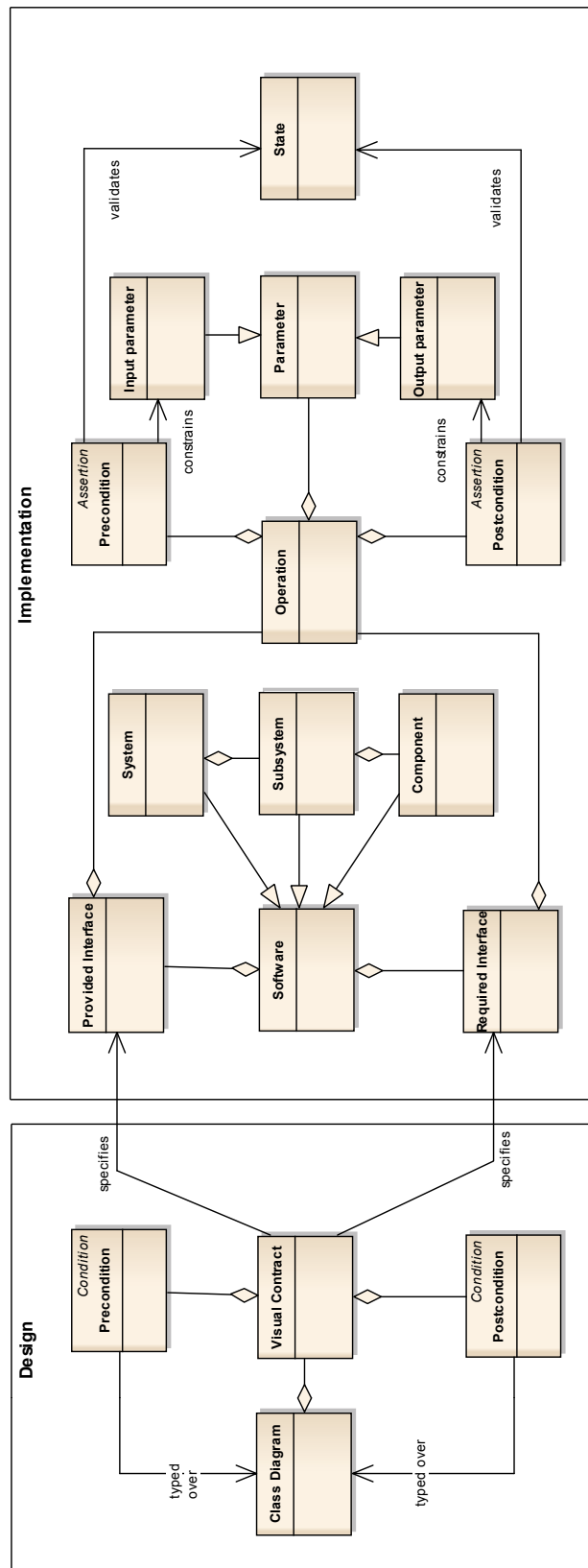


Figure 6.6: Meta-model for design and implementation artifacts (based on [Gro05a])

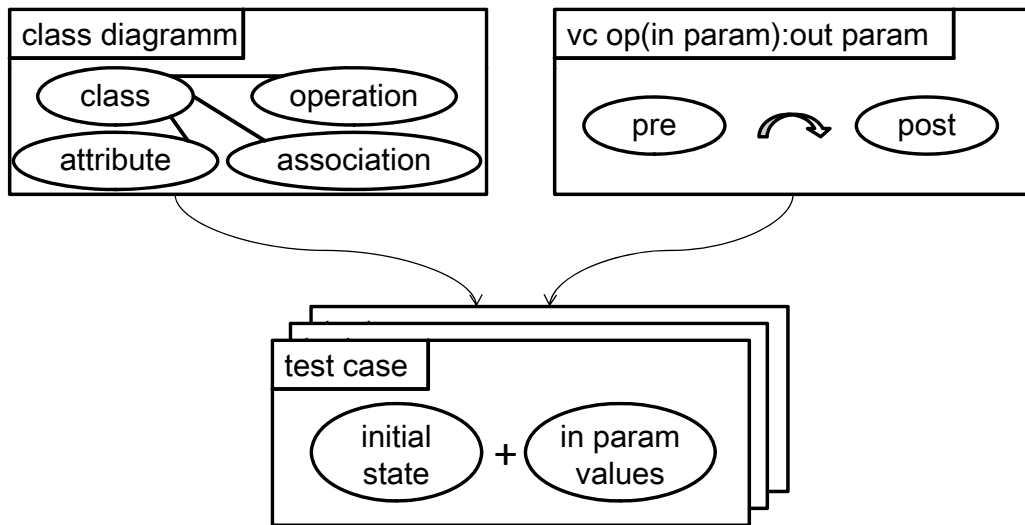


Figure 6.7: Test design artifacts

Diagram. For that the objects in the initial state are completed by further objects and their attribute values so that the cardinalities given in the Class Diagram are fulfilled.

Formally defined, our test design activity addresses the following design and testing artifacts as illustrated in Figure 6.8. A *Test case* is composed of *Initial states* and *Input parameter values* which are derived from the *Precondition* of the *Visual Contract* and from the *Class Diagram*. For testing with different object constellations and different parameter values, a *Test selection criteria* is used which results in a *Test suite* containing a set of test cases.

The standard definition of a test case also contains the information of expected return values and expected states after the invocation of the system under test [MH09], however, in our approach, we not necessarily compute the expected states during the test design. The embedded assertions derived from the Visual Contracts (cf. chapter 4) checks the conformance of the post object constellation with the postcondition at runtime. This will be explained in section 6.5 in detail.

## 6.4 Test Implementation

As illustrated in the overview of our development and testing activities in Figure 6.4, the test cases are then transformed into executable test scripts for the automated test execution. The test script should operationalize the test target, which includes the following steps:

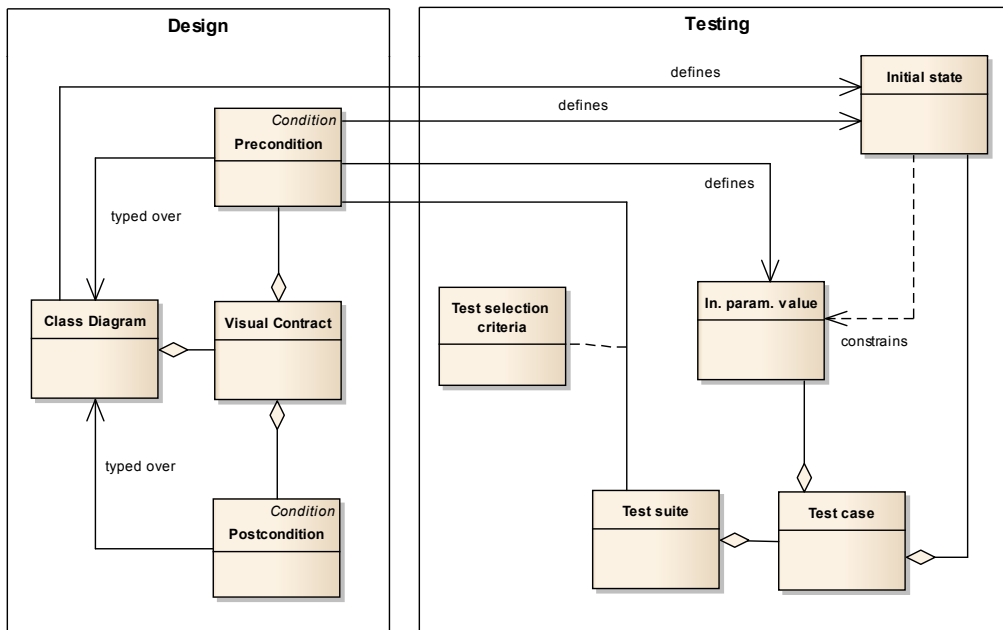


Figure 6.8: Meta-model for design and testing

1. preamble for setting the initial state,
2. invoking the implementation under test,
3. setting verdict for the test execution.

Figure 6.9 shows a pseudo code for a test script which implements these steps. Thereby, first the initial state is created as specified by the test case. Then, the operation under test is invoked using the input parameters. During the execution of the operation, the embedded assertions (cf. Figure 6.5) check the fulfillment of the pre- and postconditions. If the initial state is conform to the precondition and if the resulting state is conform to the postcondition, the invocation returns normally and the verdicts is set to pass.

If any inconformities are detected during the invocation, exceptions are thrown. If a preconditional exception is thrown, the initial state did not conform to the precondition. Thus the requirements of the operation under test are not fulfilled and it will not executed. Since the operation under test is not executed, the verdict is set to *inconclusive*, which means that the test case cannot decide on the correctness of the operation under test. If a postconditional exception is thrown, that means that the precondition is fulfilled and the operation under test is executed using the input parameters, however, the postcondition is not fulfilled by the state after the invocation,

```

testscript() {
  try {
    set initial_state;
    call operation_under_test(in_param);
    //execute embedded assertions
    //if normal_return
    verdict = pass;
  } catch (precondition_exception) {
    verdict = inconclusive;
  } catch (postcondition_exception) {
    verdict = fail;
  } catch (other) {
    verdict = error;
  }
}

```

Figure 6.9: Abstract test script

thus the operation under test *fails* the test case. If any other exception occurs during the test execution, the verdict is set to *error*.

Figure 6.10 formalizes the concepts of test scripts and their relation to the implementation. Thereby, the testing meta-model from Figure 6.8 is extended by a *Test script* and a *Verdict*. Test script invokes a operation under test. The assertions generated from the pre- and postconditions of a Visual Contract define the value of the verdicts as described above.

## 6.5 Test Execution

The abstract and generic test script in Figure 6.9 can be implemented in various test script languages, e.g. JUnit [EG14], TTCN-3 [TTC]. Then, the test scripts are executed by a test driver which can parse and interpret the corresponding test script language. Thereby, the test driver executes each script statement one by one as illustrated in Figure 6.11.

First the *initial\_state* is set by the test driver, then the operation under test  $op_1$  of class instance  $:A$  of class  $A$  is invoked by using the input parameters  $param$ . During the execution of the operation  $op_1$ , the embedded assertions for precondition  $assert_{op_1\_pre}()$  and postcondition  $assert_{op_1\_post}()$  are invoked. The sequence diagram shows two alternative fragments (*alt*) each for one assertion. If  $assert_{op_1\_pre}()$  returns *true* ( $[op_1.pre==true]$ ),

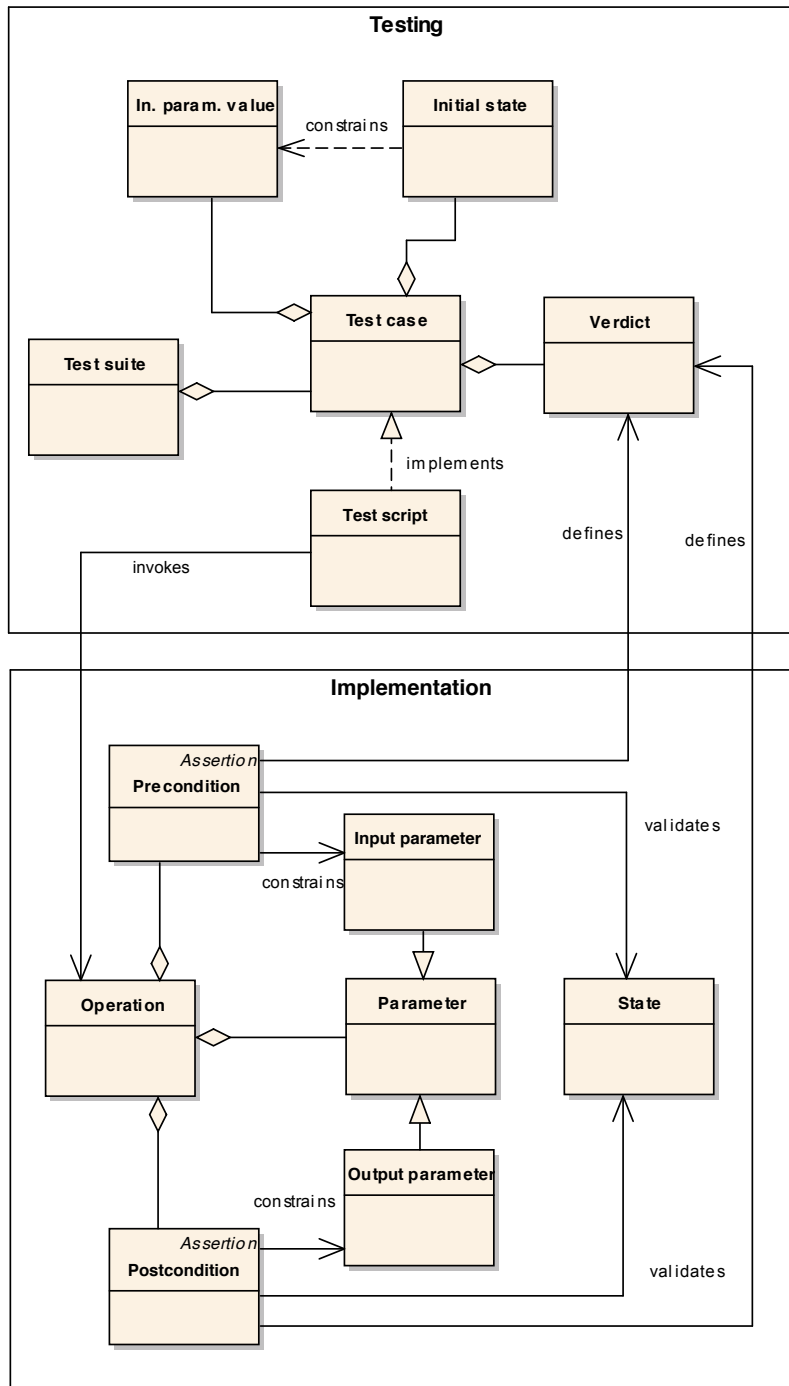


Figure 6.10: Meta-model for test execution

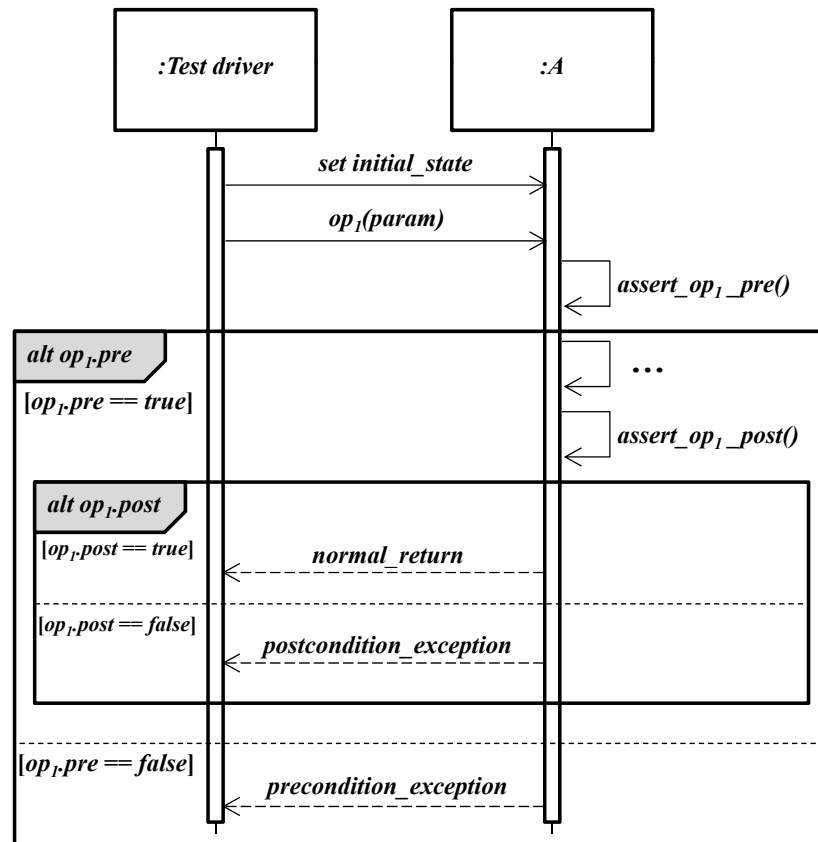


Figure 6.11: Test execution by a test driver

which means that the precondition is fulfilled, the actual behavior of the operation (...) is executed and the assertion for postcondition  $assert\_op_1\_post()$  is executed. If the precondition is not fulfilled ( $[op_1.pre == false]$ ), then the class instance  $:A$  throws a *precondition\_exception*.

If the precondition is fulfilled, a second alternative fragment (*alt op<sub>1</sub>.post*) is executed for the two possibilities of fulfillment of the postcondition. If the postcondition is also fulfilled ( $[op_1.post == true]$ ), then both assertions succeeded and the class instance returns a *normal\_return*. If the postcondition fails ( $[op_1.post == false]$ ), then the class instance  $:A$  return with a *postcondition\_exception*. Depending on thrown exceptions or the normal return, the verdict of the test case is defined as explained in the last section.

Compared to the implementation life-cycle in Figure 6.5, the sequence diagram shows only the execution flow of the embedded assertions for the required interface. The required interface is invoked by the test driver. In the next chapters, we will see that the operation under test can also call other

functions on its requested interface. As shown in Figure 6.5, the requested interface can also be specified by Visual Contracts, for which delegation assertions are generated. In a test scenario, where the requested interface is also of importance, further assertion blocks will be required, which we have omitted for simplicity in this chapter. However, the chapter 8 *Integration testing* explicitly addresses the use of delegation assertions during the test execution with more than one participating classes.

## 6.6 Summary

In this chapter, we have given an overview on the development and testing activities of our general approach. The usage of *component* term enables us to characterize the granularity of design and implementation activities. Depending on that, we also characterize the test objects and test interfaces. We have shown how the Visual Contracts can be used for generating embedded assertions and test cases. For automation purposes, we have explained how test scripts can be implemented and executed. Using meta-models we have formalized the relation between design, implementation and test concepts.

Depending on the individual requirements and characteristics of each test level, the general approach explained above has to be concretized accordingly. Figure 6.12 shows the V-model from Figure 6.1 in a flattened way, where the “V” has been pulled down on both ends. This enables us to get a uniform illustration of the processes of each test level as 6.4.

In the next chapters, we will explain how the general approach can be concretized for each test level. Thereby, we will explain which characteristics Visual Contracts must fulfill in order to be used as a test basis in a specific test level. Then, we will explain how the test cases are computed from the test basis and transformed into test scripts. Finally, we will explain for each test level, how the test scripts can be executed and the test results be evaluated.

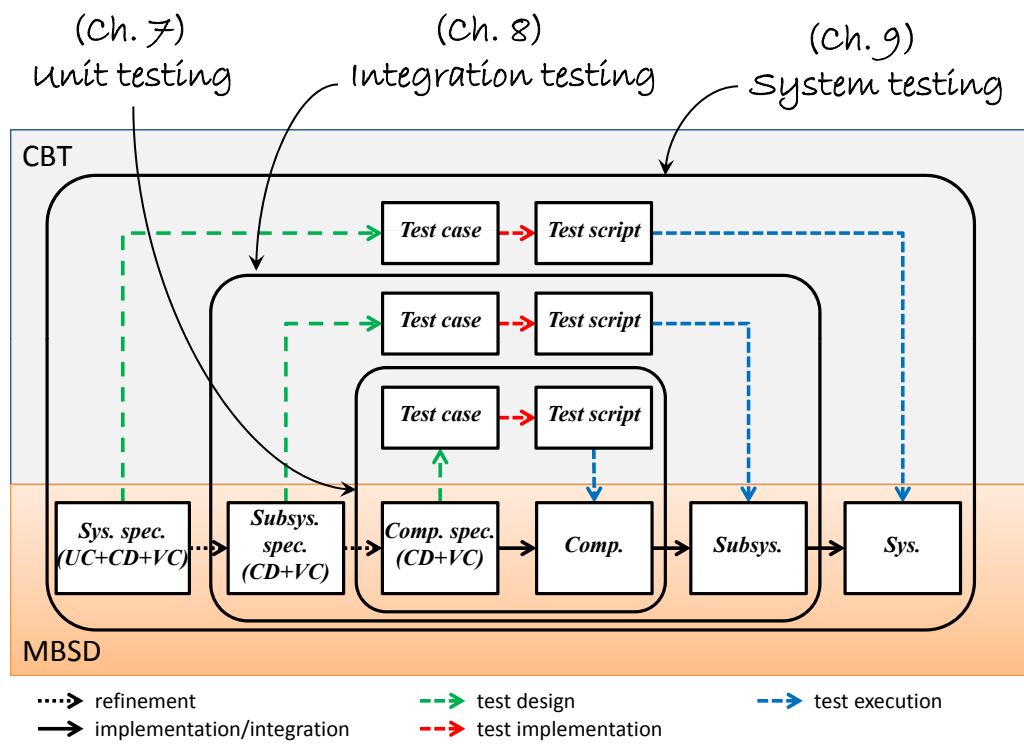


Figure 6.12: Flattened V-model showing the parallel activities of MBSD and MBT



# Chapter 7

## Unit Testing

In the last chapter, we have explained our Visual Contract-based development and testing approach in a generic manner as required by the ISO/IEC 29119 [ISO]. This chapter concretizes this approach for low-level software components (cf. Figure 6.2). The contribution of our Unit Testing approach lies in the systematic selection of test preambles for state-based testing. First, we propose a light-weight selection technique which produces *artificial* but *comprehensive* prestates. Second, we propose a more sophisticated technique which produces longer preambles for setting *natural* prestates.

By creating prestates, we are interested in checking the conformance of behavior of low-level software components such as classes and their operations in changing the system state with respect to their contract specifications: if the *component under test* is executed outgoing from a created *prestate* which fulfills the preconditions of the contract, it must change the prestate during its execution in such a way, that the *poststate* after the execution is conform to the postcondition of the contract. If the component under test passes the checks for preconditions and postconditions given in the contract, it is assumed to be functional correct.

We explain in the next sections, how components' state changing behavior can be specified by UML notations extended by Visual Contracts and how these specifications can be used for a systematic selection of test cases with artificial and natural prestates. For selecting artificial test cases, we adapt

<i>test objects</i>	classes and operations
<i>test target</i>	conformance of operation implementations to Visual contracts
<i>test strategy</i>	artificial prestates, natural prestates

Table 7.1: Overview Unit Testing

objects coverage criteria for UML and classical test data selection criteria. For selecting natural test cases, we use model checking techniques for computing *preambles* which set the components under test in a state, which is reachable from a given initial state. We also show how test execution and test evaluation are conducted based on Visual Contracts.

## 7.1 Development Scenario

Component-based development aims at developing low-level, fine-grained and reusable software components using which functional subsystems are implemented [Gro05a]. Thereby, component interface descriptions specify the intended behavior of the components (cf. Figure 6.2). Unit Testing aims at checking the correct implementation of components with respect to their interface specifications [ISTQB]. Figure 7.1 shows the contract-based Unit Testing process (CBT) combined with the development process (MBSD) using Visual Contracts. Before explaining the test process in detail, we explain first how the components are developed.

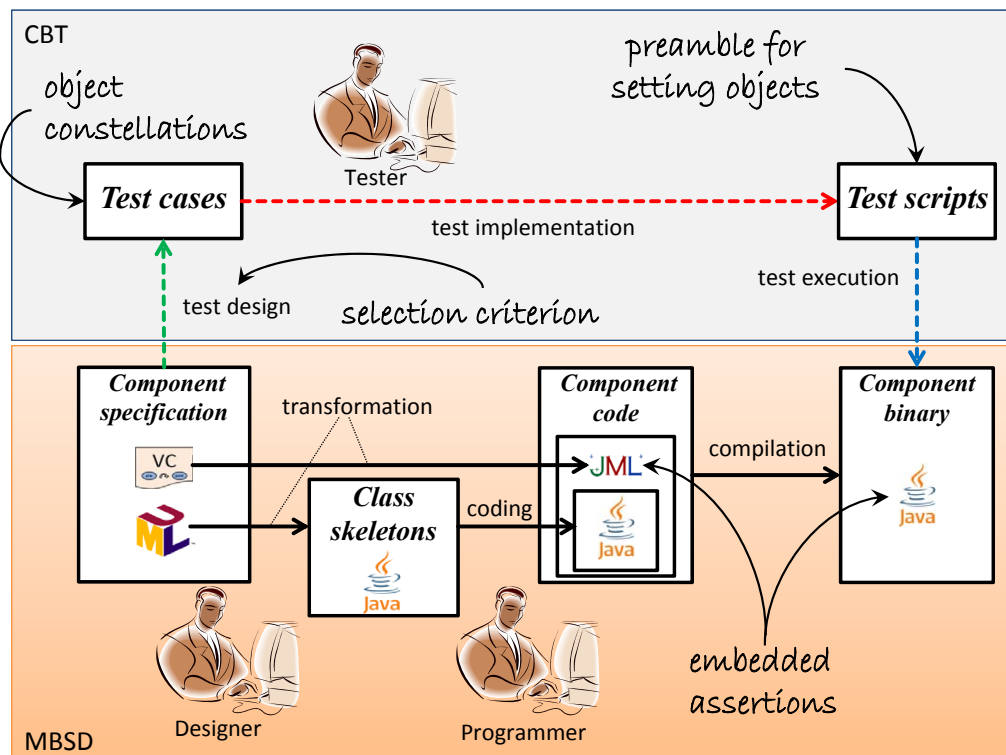


Figure 7.1: Process of Unit Testing

Our development process for components as illustrated in the lower box of Figure 7.1 follows the *model-driven development* approach based on Visual Contracts introduced by Lohmann [Loh06]. This approach is different than the classical model-driven development approaches whose target is to generate the full functional code automatically. Lohmann's approach proposes the generation of class skeletons from the component specification, the manual completion of class skeletons with behavioral code and the generation of embedded JML assertions by model transformations to monitor the components behavior at runtime (cf. Chapter 4).

During the component design, a software designer specifies a model of the component under development. This model consists of *UML Class Diagrams* and *Visual Contracts*. The Class Diagrams describe the static aspects of the system, like the class attributes, the syntax of class operations and the relations between classes. Each Visual Contract specifies the behavior of a class operation. The behavior of the operation is given in terms of state changes by pre- and postconditions, which are modeled by a pair of UML Composite Structure Diagrams [ELSH06] (cf. Chapter 4).

After the component design, we generate code fragments from the design models using the *transformation rules* illustrated in Figure 7.2. With the first rule  $r_1$ , we generate Java class skeletons from the design Class Diagrams. The second rule  $r_2$  generates a frame for each provided operation consisting of a return type and if given input parameters and annotates them with JML assertions generated from the pre- and postconditions of the Visual Contract. The third rule  $r_3$  generates for a requested operation a delegation pattern which frames the actual operation call with JML assertions for pre- and postconditions. This pattern enables a better testability of state variables by client-side monitoring. By applying the algorithm in Figure 7.3, starting with an empty code, the transformation rules create step by step code fragments containing placeholders for the programmers to extend.

After the code generation by model transformations, a programmer *manually completing* the generated Java fragments with behavioral code resulting in a functional component. Thereby, the programmer uses the Visual Contracts in the component design as a behavioral specification for the operations. The programmer codes the operation bodies and is also allowed to add new operations or new classes, but the programmer is not allowed to change the JML assertions generated by the model transformations [LSE05].

When the programmer has completed the behavioral code, he uses a JML compiler to build the executable component binary. The binary component consists of the programmer's behavioral code and executable runtime assertions which are generated by the JML compiler from the JML assertions. During the execution, the behavioral code leads to changes in the system

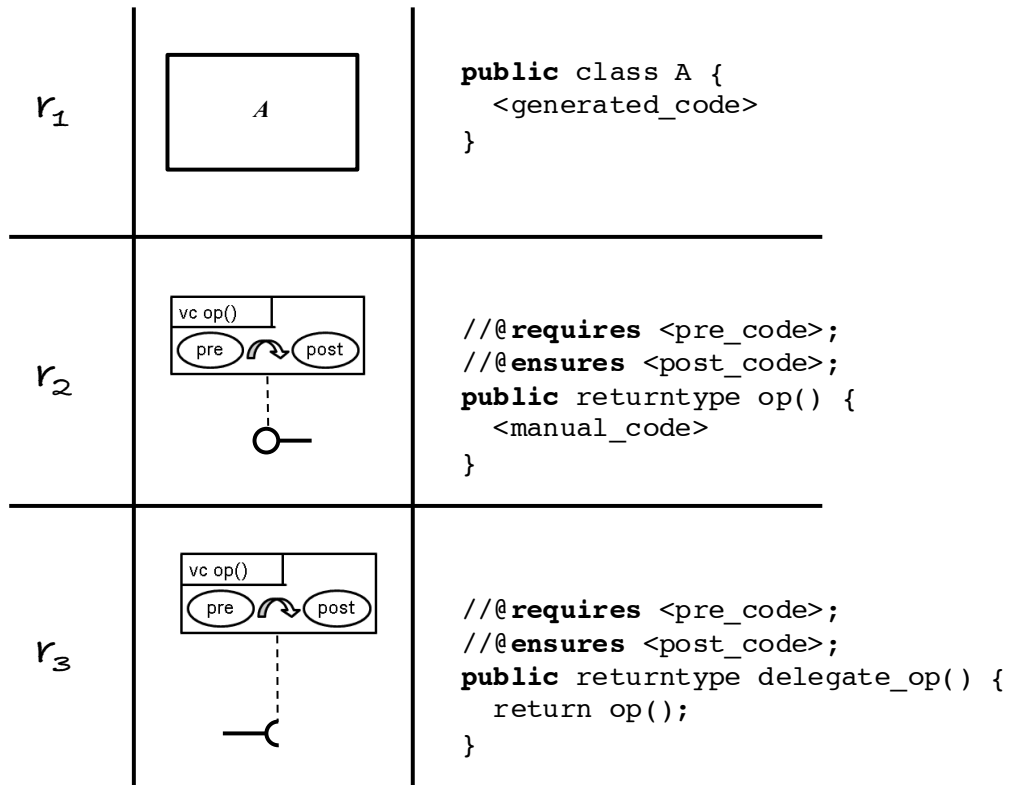


Figure 7.2: Transformation rules for code generation

---

**Algorithm** Code generation from Visual Contracts
 

---

1. `component_code = empty string`
  2. **for all** `class_c` **do**
  3.   `component_code = invoke`  $r_1$ (`class_c`)
  4.   **for all** `provided_op` **do**
  5.     `op_code = invoke`  $r_2$ (`provided_op`)
  6.     **extend** `component_code` **with** `op_code`
  7.   **for all** `required_op` **do**
  8.     `op_code = invoke`  $r_3$ (`required_op`)
  9.     **extend** `component_code` **with** `op_code`
- 

Figure 7.3: Algorithm for code generation using transformation rules

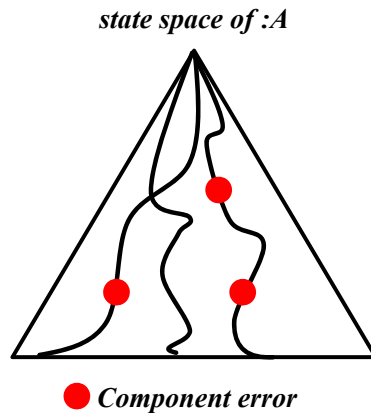


Figure 7.4: Component errors

state. Thereby, the generated runtime assertions *monitor* the conformance of the system state with the pre- and postconditions [Loh06].

The manual coding activity of programmers are error-prone, which can lead to programming errors in the component. These can be exposed during various execution paths of the component as illustrated in Figure 7.4. Even if such errors can be detected by runtime monitoring during the operation of the component as explained above, existence of errors after the delivery can lead to dissatisfaction of users. That is why a component must be *tested* thoroughly to find and remove the errors before delivery. Since we deal with state-based errors, during Unit Testing, the test cases have to set the component under test to a prestate in which it can be invoked in a controlled way. Only if the component under test is executed under controlled conditions, its results can be evaluated to pass or to fail. In the next section, we will explain how such *test cases with controlled prestates* can be specified systematically.

## 7.2 Test Design

In Unit Testing, we use Visual Contracts as the test basis which are part of the component specification. On one hand, we derive controlled prestates from the Visual Contracts as test inputs for state-based testing. On the other hand, we use the Visual Contracts as test oracles for evaluating the conformance of the state changes during the test execution with the embedded assertions. During our research, we have developed two approaches for specifying controlled prestates for state-based testing.

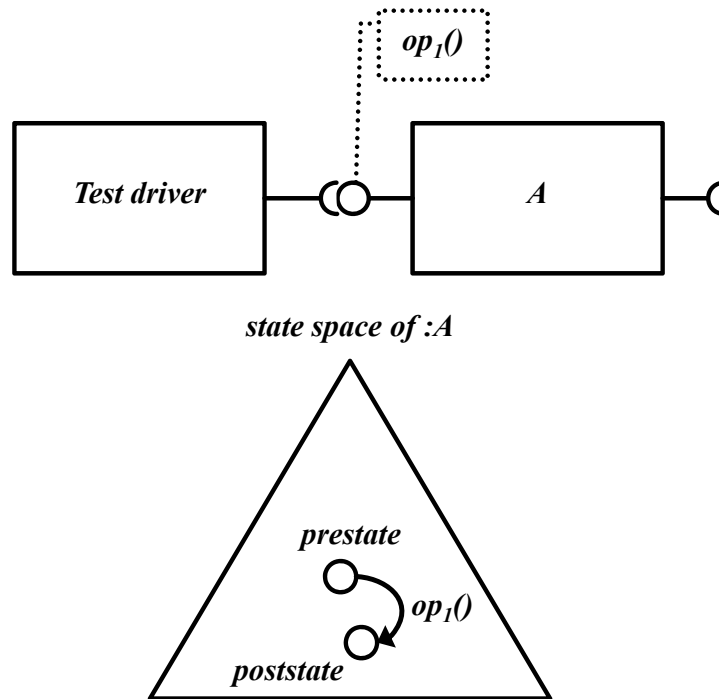


Figure 7.5: Setting the prestate for testing  $op_1$  artificially

In the first approach, a minimal prestate is computed from the specification, such that the precondition of a method under test is fulfilled. Since this prestate is created from scratch, we call this kind of state an *artificial* prestate. In the second approach, the prestate for a method under test is computed by invoking other methods. We call this kind of state a *natural* prestate, since it represents a system state which could have been created by real method interactions. The more realistic the prestate is, the more reliable are the test results. As next, we explain these two approaches in detail.

### 7.2.1 Approach 1: Artificial Prestate

As illustrated in Figure 7.5, the prestate is a member of the set of all possible states (*state space*) of an object of class *A*. A *Test driver* is responsible for setting the test object to a *prestate* and invoke a class operation  $op_1$  using test inputs and to evaluate the test results. The prestate is crucial for the invocation and for the evaluation of the test results. In this scenario we will explain, how a prestate can be set artificially and how the resulting *poststate* is evaluated.

The overall test design procedure of this approach is illustrated in Figure

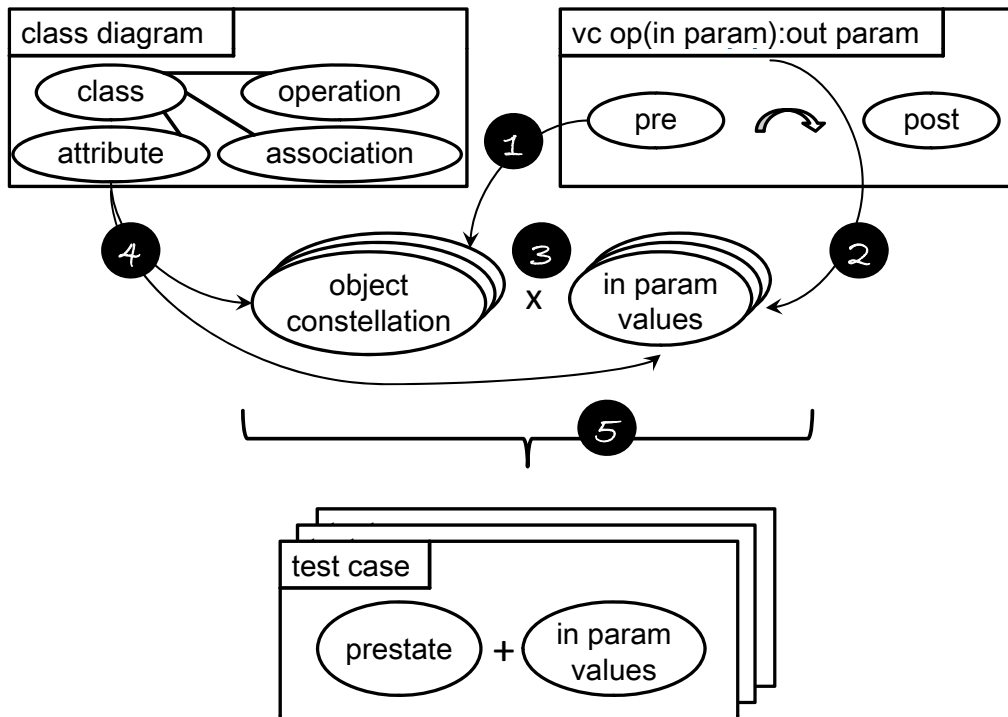


Figure 7.6: Procedure for generating test cases

7.6. In step 1 *object constellations* are derived from the precondition of the Visual Contract of the operation under test *op*. After concrete input parameter values are derived from the signature of the *op* in step 2, these are assigned in step 3 to the derived objects. In step 4, the object constellations and their variables are completed based on the class specifications resulting in logical *test cases* including a prestate and input parameter values for the test invocation (step 5). As next we explain these steps in detail using the Visual Contract for the operation `cartAdd` as shown in Figure 7.7.

**Step 1** of the overall procedure is the generation of object constellations which build the basis for the prestates. The algorithm *generateobjects* for this step as shown in Figure 7.8 starts with the `this`-object of the class under (line 1-2) test and computes possible object constellations conforming to the prestate (line 3-12). If there are no multiple objects in the precondition, the object constellation will have similar number objects like in the precondition. However, if there are multiple objects, various object constellations are possible, which contain different number of instances for each multiple objects. To keep it simple, we derive 0, 1 and a random number of objects for each multiple object. The result of the algorithm is a set of object constellations,

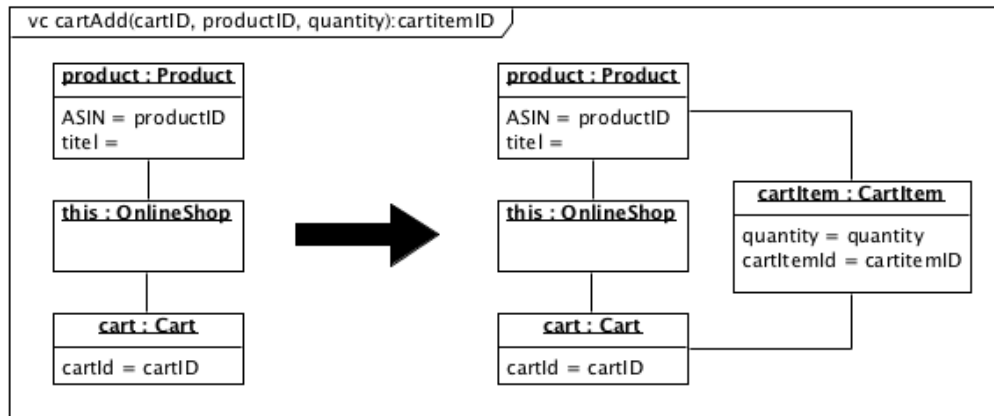


Figure 7.7: Example for Visual Contract

**Algorithm 1.** generateobjects(startobject):objectset

Generation of object constellation from precondition

---

```

1. objectset = {startobject}
2. currentobject = startobject
3. foreach association of currentobject.type do
4.   if association not already covered do
5.     newobject = new association.targetobject.type
6.     newlink = new association
7.     newlink.source = currentobject
8.     newlink.target = newobject
9.     currentobject.links.add(newlink)
10.    objectset.add(newobject)
11.    objectset.join(generateobjects(newobject))
12.  end
13. end
14. return objectset

```

---

Figure 7.8: Algorithm for step 1



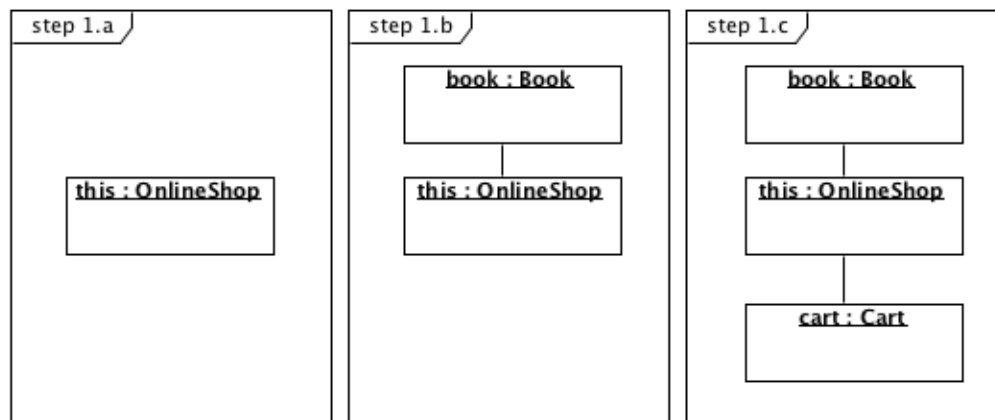


Figure 7.9: Example for step 1

each of which represents a test case.

Figure 7.9 shows an example for step 1 using the Visual Contract `cartAdd` (cf. Figure 4.7 in chapter 4). From the precondition, empty object constellations for classes `Book`, `OnlineShop` and `Cart` are created. Because, the objects in the precondition are single objects (not multiple objects), single instances of the classes are generated and linked to each other obeying the class associations.

Besides object constellations which build the basis for the prestates, a test case also contains concrete values for input parameters for invoking an operation under test. For that, we derive in **step 2** concrete values for each input parameter. As shown in Figure 7.10, the derivation can follow various test data generation techniques [AS05]. In our approach, we apply *random* test data generation, *boundary value* analysis and *equivalence partitioning* method. Using these techniques, huge number of combinations of parameter values are possible. However, former research has shown that high test data coverage helps in increasing the fault detecting capability of a testing approach [ABLN06]. In order to cope with the big number of test cases, corresponding configuration mechanism must be available in tool support, such that each tester can define its own way for test data coverage.

Figure 7.11 shows an example for step 2 where the parameters in the operation signature of `cartAdd` are instantiated. Our example applies the random selection for strings and the boundary value technique for integers. Thereby, for the data types integer the boundary values  $0(\pm 1)$ ,  $\text{MIN\_INT}(\pm 1)$  and  $\text{MAX\_INT}(\pm 1)$  are used for instantiating the input parameters. In this way, many possible combinations of input parameter values can be computed

---

**Algorithm 2. generateparameters(paramset, signature)**  
 Generation of parameter values from the signature

---

```

1. paramset = {}
2. foreach parameter in signature do
3.   case selectioncriteria do
4.     random:
5.       paramvalue = random(paramtype)
6.       paramset.add(<paramname, paramvalue>)
7.     boundaryvalue:
8.       foreach boundaryvalue of paramtype do
9.         paramset.add(<paramname, boundaryvalue >)
10.    equivalenceclasses:
11.      foreach equivalenceclass of paramtype do
12.        paramvalue = random(equivalenceclass)
13.        paramset.add(<paramname, paramvalue>)
14. return paramset

```

---

Figure 7.10: Algorithm for step 2

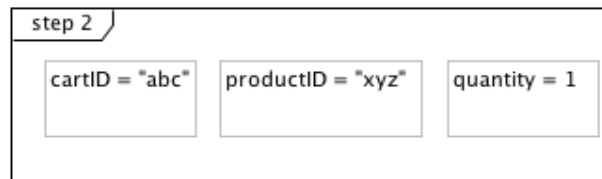


Figure 7.11: Example for step 2

as in the example. Ellerweg has studied in [EEG08] the boundary values of various data types for test case generation.

For defining prestates, the object constellations from step 1 and the input parameter values from step 2 need to be combined in **step 3** because of two reasons: The object constellations contain objects and object links, however, their attribute values (object variables) are undefined. The input parameters can address the attribute values of objects, so that these can be directly assigned to object variables. Furthermore, the input parameter values can also contain objects which are already part of the object constellation computed in step 1. That is why the results of step 1 and 2 are consolidated (see Figure 7.12) resulting in object constellations with concrete attribute values.

Figure 7.13 shows an example for the initialization of object variables in step 3. Thereby, the parameter values generated in example in Figure 7.11

---

**Algorithm 3. initializeobjects(objectset, paramset)**  
 Initialize object constelleations using parameter values

---

1. **foreach** object **in** objectset **do**
2.     **foreach** variable **of** object **do**
3.         **if** variable **in** paramset **then**
4.             variablevalue = paramvalue
5. **return** objectset

---

Figure 7.12: Algorithm for step 3

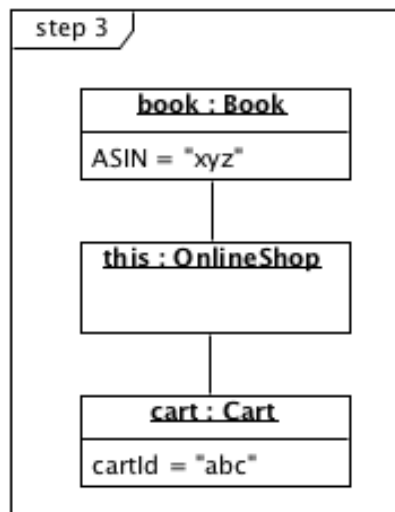


Figure 7.13: Example for step 3

are combined with the generated objects from example in Figure 7.9 resulting in realistic objects.

As stated at the beginning of this section, the more realistic the prestate, the more reliable are the test results. The prestates in approach 1 are artificial because these are directly derived from the precondition of an operation. That means, the prestates are created to fulfill the preconditions. It may happen, that these prestates would never be created in real deployment of the class under test. However, this approach enables a controlled invocation of the operation under test and the evaluation of the state changing behavior of the operation. To make the artificial prestate more realistic, in **step 4**, we extent it based on the specifications in the class diagram. If an object in the object constellation is specified in the class diagram to have further links to other objects, these are completed in the object constellation (see Figure

**Algorithm 4. completeobjectsconstellations(objectset, classdiagram)**

Initialize object constelleations using parameter values

- 
1. **foreach** object in objectset **do**
  2.     class = type of object;
  3.     **foreach** association **of** class **do**
  4.         **if** association **not in** object.links **then**
  5.             newobject = **create** association.targetClass
  6.             objectset.**add**(newobject)
  7. **return** objectset
- 

Figure 7.14: Algorithm for step 4

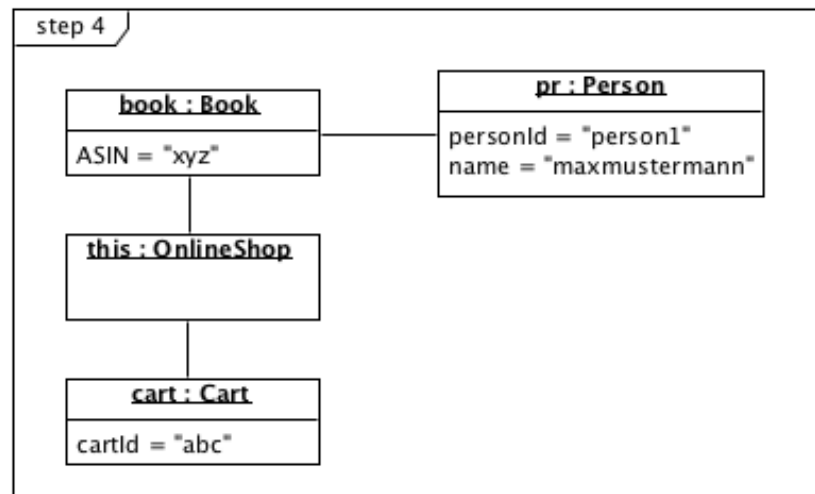


Figure 7.15: Example for step 4

7.14).

Figure 7.15 shows an example for the extension of object constellation in step 4. Since the corresponding Visual Contract is typed by the class diagram shown in Figure 4.4, the objects are checked for further links. Thereby, the object `book` of class `Book` is extended by an object `pr` of class `Person`.

Having derived object constellations from preconditions and completed them based on the class diagram and using the generated input parameter values, in **step 5** we combine these to give individual test cases each of which contains a prestate and a list of parameter values for the test invocation (see Figure 7.16). The prestates fulfill the specifications given by the preconditions and by the class diagram. Figure 7.17 shows an example joining the prestate from Figure 7.15 and the input parameters from Figure 7.11.

**Algorithm 5. generatetestcases(objectset, paramset)**

Combine objects and parameters giving test cases

---

```

1. testcases = {}
2. foreach object in objectset do
3.   foreach param in paramset do
4.     objectstate = combine(object, param)
5.     testcases.add(objectstate)
6. return testcases

```

---

Figure 7.16: Algorithm for step 5

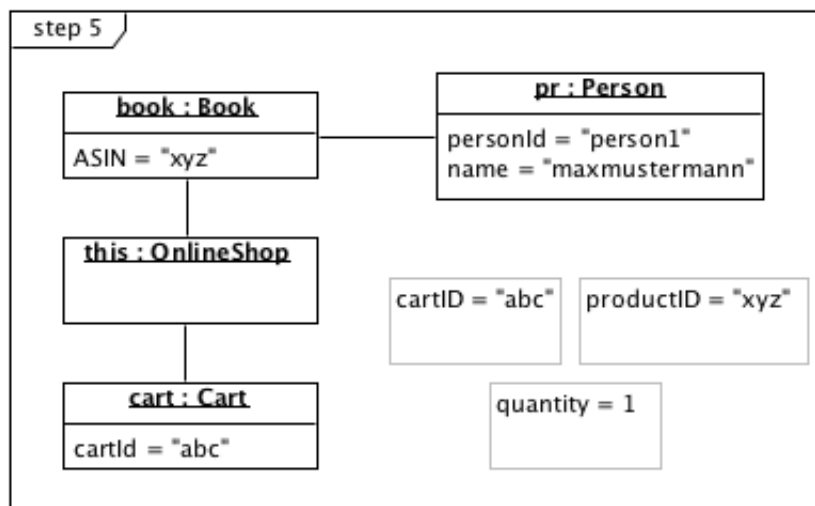


Figure 7.17: Example for step 5

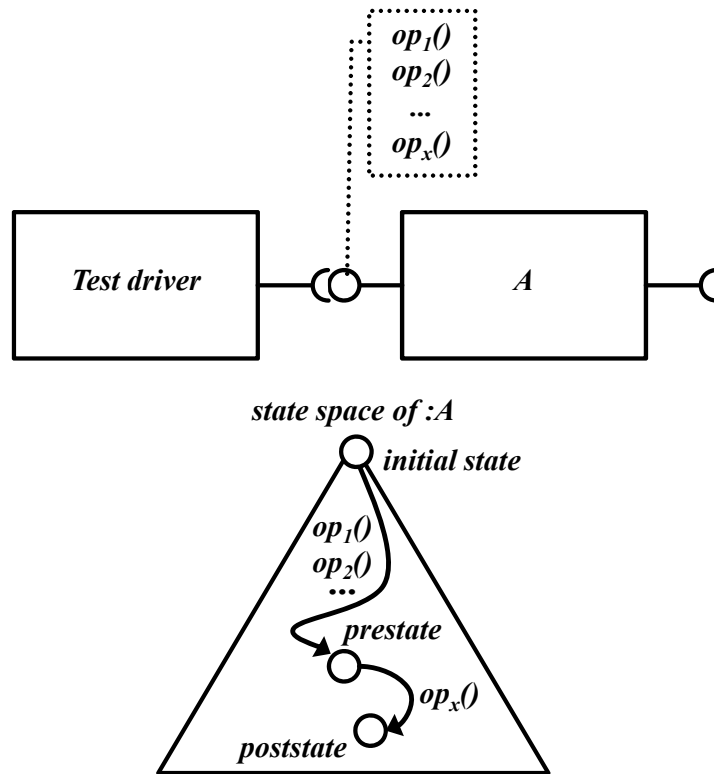


Figure 7.18: Setting the prestate for testing  $op_x$  naturally

### 7.2.2 Approach 2: Natural Prestate

In approach 1, the prestate is generated artificially based on the preconditions of a Visual Contract in order to set the class under test into a testable state. This can be a first validation step of the quality assurance where the implemented classes are tested just after their implementation using controlled prestates. However, in real deployment, objects of classes always interact with other objects and constitute more complex behavior. During this interaction, system states changes as an effect of invocation of various operations. Thus, in order to test class operations in a realistic manner, their prestates must be set also in a natural way.

For this purpose, in this section we introduce a second approach where the prestate is computed by the invocation of various operations. Thereby, the order in which the operations are invoked is crucial for setting a required prestate. Figure 7.18 illustrates the *state space* of the class *A* and the *invocation* of its operations in a particular sequence starting from an *initial state*, such that the class under test is set step-by-step into a required

**prestate.** From the prestate on, the operation under test  $op_x$  can be invoked and the **poststate** can be checked for conformance with the **Visual Contract**, which specifies the operation under test (cf. Chapter 6). We call the particular sequence of operations for setting the prestate a **preamble**. The computation of the preamble requires a formal analysis of the state space of the class under test and the reachability of the prestate.

For computing the state space and the preamble, we use techniques of **graph transformations** [Hec06] and **model checking** [KR06]. The theory of graph transformations enables a formal definition of semantics of Visual Contracts for specifying the state changing behavior of class operations (cf. Chapter 4). Having these specifications, we use model checking techniques to explore the state space of a class and look for a path from the initial state to the prestate. If such a path exists, it represents the preamble for setting the prestate.

In the terminology of graph transformations, the preamble computation can be explained as follows (see also Table 7.2): A **graph transition rule** transforms a **source graph** into a **target graph**. Given an **initial graph** and a set of graph transition rules, a **graph transition system** can be computed which comprises a possibly infinite set of graphs. cpunterexample reachability analysis, a **graph transition sequence** can be computed which transforms an initial graph step-by-step into a required source graph. The graph transition sequence represents the order of operations to be invoked for setting the prestate. We call this sequence of operations the preamble.

Having roughly explained how graph transformations can map the concepts of Unit testing and how preambles can be computed, in Figure 7.19 we illustrate the overall approach which we explain now in detail. In **step 1**, starting with an initial state and a set of operation specifications given by Visual Contracts, a state space is computed. Every state in this state

Unit testing	Graph transformations
state space	graph transition system
initial state	initial graph
Visual contract specification	graph transition rule
operation invocation	graph transition
preamble	graph transition sequence
prestate	source graph
poststate	target graph

Table 7.2: Mapping between concepts of Unit testing and Graph transformations

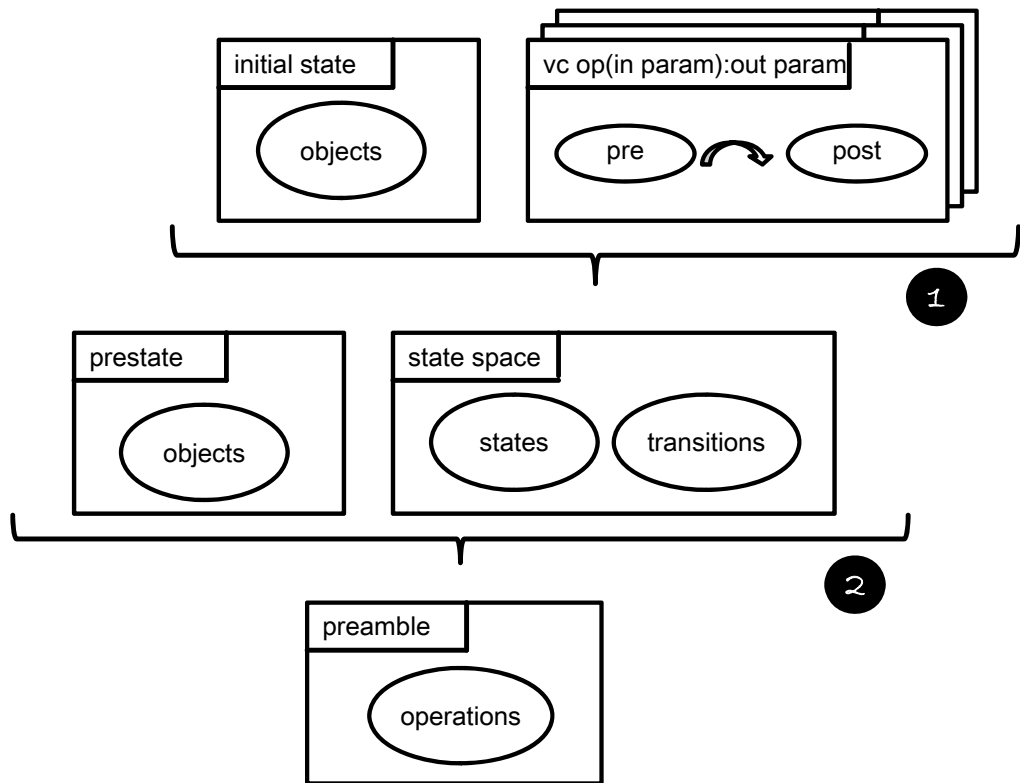


Figure 7.19: Procedure for generating test cases

space, which is reachable from the initial state by state transitions as a result of invocation of class operations, is a natural state. In **step 2**, we try to compute a preamble for setting the class under test into a controlled state. First, a test case containing a prestate and input parameter values for test invocation are generated using the techniques introduced in approach 1 (cf. Section 7.2.1). Then, by checking the reachability of a prestate in the state space, we try to compute a preamble which transforms the initial state to the required prestate. If such a preamble cannot be computed, the required prestate is not reachable, thus not feasible for testing purposes. In this case, another prestate should be computed using the techniques of approach 1.

### Step 1: Computation of state space

Since the objects states are *graphs*, the state space means a set of graphs which results from graph transformations. Having defined the *graph transformation rules* and *graph transitions* in Chapter 4, we introduce in this section further definitions based on [Roz97, KR06, Han08].



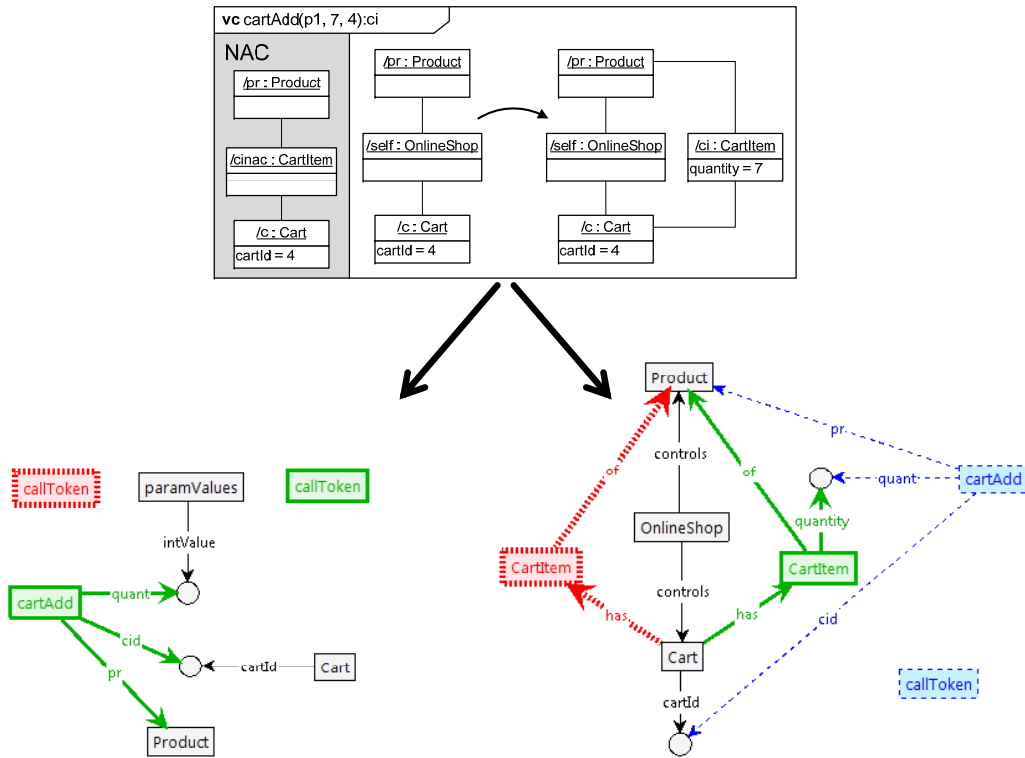


Figure 7.20: Example for transformation rule 1: cartAdd

**Definition 5 (Graph production system)** A graph production system is a tuple  $GPS = \langle I, R \rangle$  where  $I$  is a graph representing an initial state and  $R$  is a set of graph transformation rules.

Figures 7.20 and 7.21 illustrate examples for a graph transformation rule and for an initial state. Figure 7.20 shows how a Visual Contract can be formalized as a visual graph transformation rule for model checker GROOVE [KR06]. Thereby, the objects to be created in postcondition are colored *green*, the objects to be deleted are colored *blue*. Objects which are not allowed to exist (Negative application conditions - *NAC*) are colored *red*. Figure 7.21 illustrates an initial state  $s_0$  containing an instance of class *OnlineShop* and concrete parameter values for the input parameters of operation *cartAdd*

**Definition 6 (Graph transition sequence)** A sequence of graph transformations which begins with a start state and ends with a concrete state from  $S$  is represented by  $G_0 \Rightarrow_{r_0} G_1 \Rightarrow_{r_1} \dots \Rightarrow_{r_{n-1}} G_n$  where the transformation rules are  $r_0 \dots r_{n-1}$ .

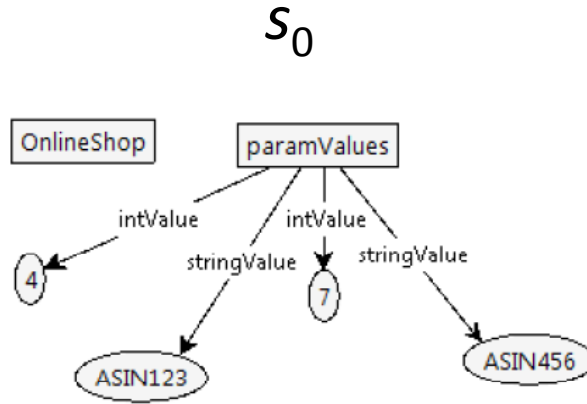


Figure 7.21: Example for initial state

**Definition 7 (State space)** Given a GPS, the state space  $S$  containing all reachable states from an initial state  $I$  over the graph transitions by a set of transformation rules  $R$  is represented by  $S = \{G \mid G \in \mathcal{G} : I \Rightarrow_R^* G\}$ .

**Definition 8 (Graph transition system)** The graph transition system  $GTS = \langle S, \rightarrow, I \rangle$  generated by a graph production system  $GPS = \langle I, R \rangle$  consists of a set  $S$  of states, which are actually graphs; a transition relation  $\rightarrow \subseteq S \times R \times [\mathcal{G} \rightarrow \mathcal{G}] \times S$ , such that  $\langle G, r, m, H \rangle \in \rightarrow$  iff there is a rule application  $G \xrightarrow{r, m} H'$  with  $H'$  isomorphic to  $H$ ; and an initial state  $I \in S$  [KR06].

Figure 7.22 illustrates how the state space of the class under test is explored and the graph transition system is computed. Thereby, starting from a initial state  $s_0$ , we apply the Visual contracts of class operations as graph transformation rules. The initials state  $s_0$  is set by any constructor operation. Starting from the  $s_0$ , in each step, a set of Visual contracts  $(vc_1 \dots vc_n)$  are applicable on the current state. The Visual contracts, which are applicable in step  $k$ , are defined as  $vc_{n_k}^k$ .

Figure 7.23 shows an illustrated example how a graph transition system can be computed by the GROOVE tool for the OnlineShop example. Starting with an initial state  $s_0$  (see Figure 7.21), by applying graph transformation rules (see Figure 7.20), possible states and and state transition are computed.

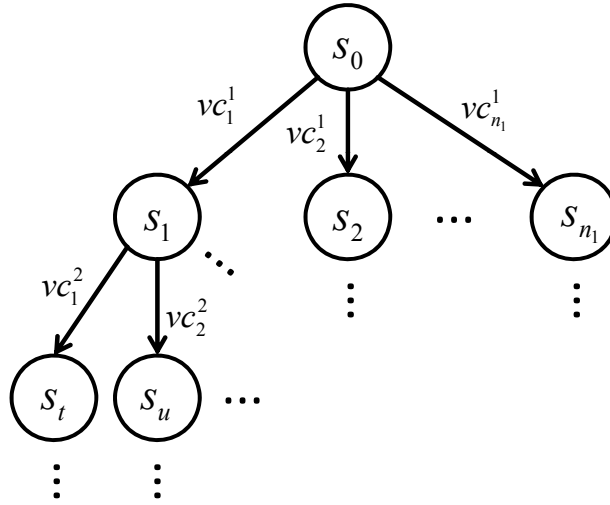


Figure 7.22: GTS generated by Visual Contracts as production rules

## Step 2: Computation of preamble

In order to compute the preamble for setting a controlled prestate in a GTS, we apply model checking for graph transformations as defined in [KR06, Han08]. This model checking technique is integrated into our testing approach using the corresponding GROOVE tool [GMR<sup>+</sup>12] which will be explained later in chapter 10. In this section, we explain how the preamble found in the GTS can be described formally and how it maps to our testing concepts.

In [KR06], the GTS is transformed into a *Kripke structure*, which is comparable to a finite state machine, however without an input and an output alphabet. The states of the Kripke structure are assigned a set of graph transformation rules which are applicable on that state [KR06]. For the Kripke structure, system properties are formulated in temporal logic which are then checked by exploring the state space using specific graph traversal algorithms [KR06].

In our model checking approach, we define the system properties using Computation Tree Logic (CTL) [BBF<sup>+</sup>10]. We differentiate between the following three classes of properties regarding their art of formulating the property: *Reachability properties* define system properties, saying that some state of the system with a desired property is reachable. A reachability property holds, if there is some execution of a system including a state where the property holds. *Liveness properties* assure that a system executes as expected or that “something good will eventually happen”. *Safety properties*

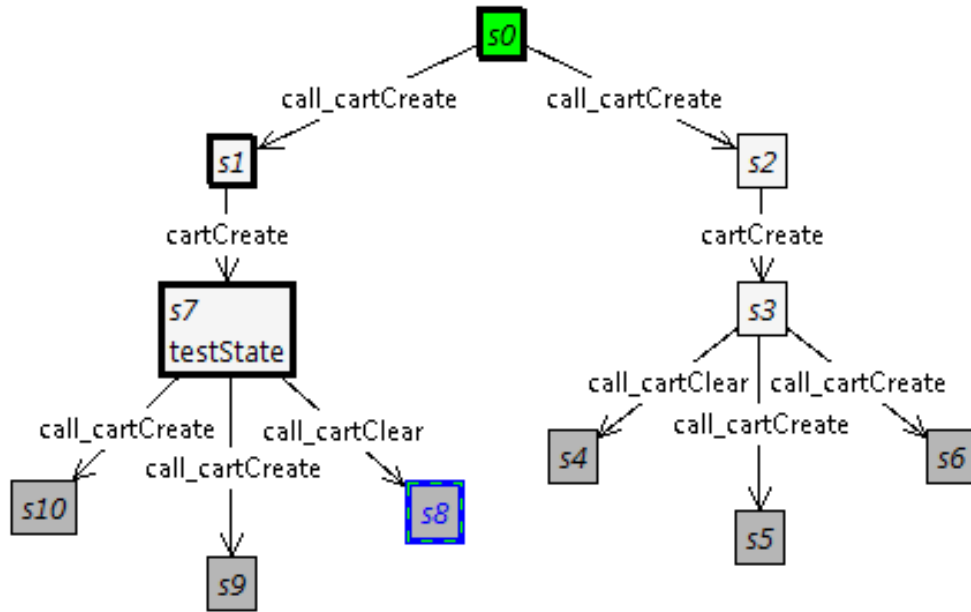


Figure 7.23: Example for GTS for OnlineShop

on the other hand ensure that the system does not enter an undesired state or “that something bad will not happen” [BBF<sup>+</sup>10, Gül05].

In order to find a sequence of graph transformation rules for transforming an initial state into a required prestate, we formulate the following reachability property: “Does any path of graph transitions starting at a given initial state exist, which ends with a state of GTS where the required prestate is a subset of that state?” This property can be formulated in CTL as  $\mathbf{EF}\phi$ . Thereby,  $\phi$  is a state property which represents the required prestate. The temporal operator  $\mathbf{F}$  states that the state property  $\phi$  will *eventually* hold on a given path at some state. The path quantifier  $\mathbf{E}$  states that there must exist at least one path starting from the initial state, on which the path property  $\mathbf{F}\phi$  holds. This property is checked on the Kripke structure of GTS  $K_{GTS}$ . The model checking question is represented formally as  $K_{GTS} \models \neg\mathbf{EF}\phi$ .

If such a path exists in GTS, the model checking will return a *yes* as a result. However, we are not only interested in finding out whether such a path exists or not, we are also interested to know how this path looks like, i.e. we want to know which transitions are contained in this path. For that purpose, we use the facility of model checking to produce *counterexamples*, in case of a not fulfilled property. Therefore, we *negate* our reachability

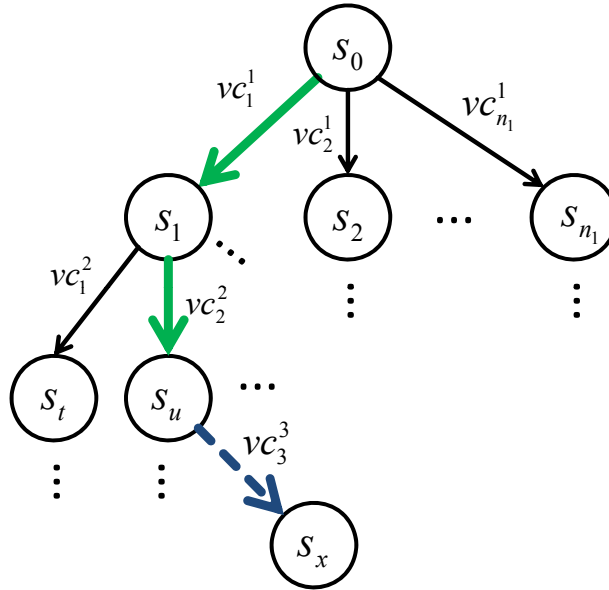


Figure 7.24: Operation invocation sequence representing the preamble

property resulting in a safety property  $\neg \mathbf{EF}\phi$  which is equivalent to  $\mathbf{AG}\neg\phi$ . This property includes the temporal operator  $\mathbf{G}$  and the path quantifier  $\mathbf{A}$  and has the following semantic: “Is it true, that there is no state which is the same as the prestate represented as  $\phi$  in all possible paths of GTS?” The model checking question  $K_{GTS} \models \mathbf{AG}\neg\phi$  can either return a *yes*, stating that the required prestate does not exist or it is not reachable. Or it can answer *no*, i.e. that the prestate does exist. In the later case, the model checking returns a counterexample which represents a path where this property does not hold. In other words, model checking returns a path which contains transitions from the initial state to the required prestate. Using this trick, we can formally define the preamble as follows.

**Definition 9 (Preamble)** *A preamble is the result of model checking of the reachability property. It is represented by a sequence of operation invocations  $G_0 \Rightarrow_{op_0} G_1 \Rightarrow_{op_1} \dots \Rightarrow_{op_{n-1}} G_n$  where the transformation rules  $op_0 \dots op_{n-1}$  represent the operations.*

Figure 7.24 illustrates the exploration of the preamble for the following testing scenario: We are looking for a preamble for testing the operation  $op_3$  which is specified by the Visual contract  $vc_3$ . In order to be able to invoke  $op_3$ , we have to set the object under test into a controlled state where the precondition of  $op_3$  is fulfilled. Therefore, we derive an object constellation

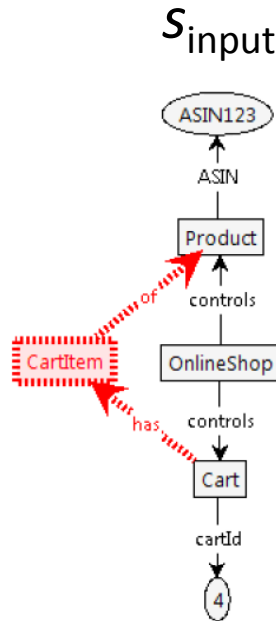


Figure 7.25: Target graph

$s_u$  from the precondition of  $vc_3$  using test case generation techniques from approach 1 (cf. section 7.2.1). This object constellation is the prestate we are looking for. For that, we formulate the safety property is  $K_{GTS} \models \mathbf{AG} \neg s_u$ . Since this state is contained in the state space of GTS and it is reachable from the initial state, the model checker produces a counterexample containing the transitions  $s_0 \Rightarrow_{vc_1^1} s_1 \Rightarrow_{vc_2^2} s_u$  (see the solid green line).

Having found the preamble the object under test can be set into a prestate from where the operation under test  $op_3$  can be invoked (see the dashed blue line). This invocation will also lead to a new state  $s_x$ . Our test target in this approach is to check this new state for conformance with the postcondition of  $vc_3$ .

Figure 7.25 shows an example for a target graph  $s_{input}$  which can be used as a state property. If this state property is model checked on the GTS from Figure 7.23, a counterexample is computed which represents the path on which the state property is not fulfilled. This counterexample can then be used as a preamble for testing as shown in Figure 7.26.

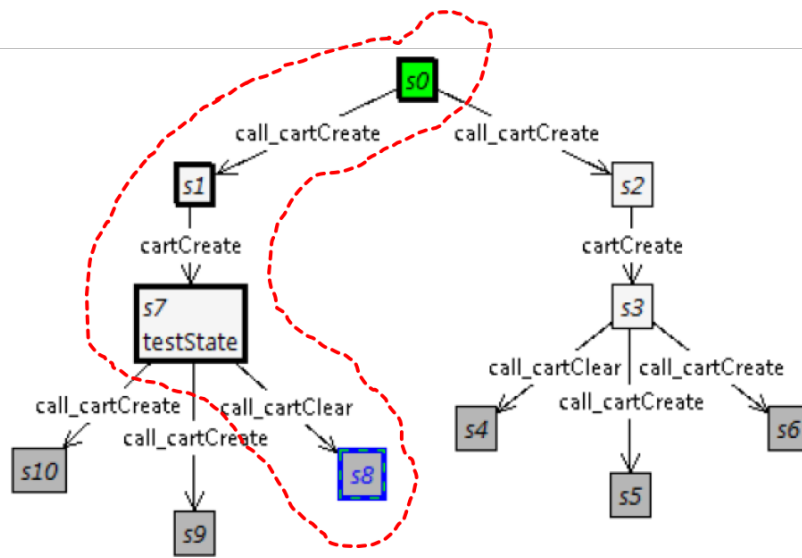


Figure 7.26: Example for preamble

### 7.3 Test Implementation

In the last section Test Design, we have described our approach to systematically derive test cases for state-based testing for classes. The test target is to check the conformance of a class operation with its Visual contract specification, i.e. to validate that the implementation of an operation changes the state of the object as specified by the pre- and postconditions. For that test target, the object is set into a controlled state which conforms to the precondition of the Visual contract and is executed with input parameters. After the execution, if the return values and the state changes conform to the postcondition of the Visual contract, the implementation succeeds the test.

As we aim at the automation of our testing approach, we define in this section how the above described test scenario can be implemented using test scripts. In the abstract test script shown in Figure 6.9, we have defined three main parts: (1) setting the prestate, (2) invoking the operation under test, (3) setting test verdict after checking poststate. In the last section, we have described how the prestate can be computed. Here we describe how a test script can be implemented such that the computed prestate is set during the test execution. The implementations differ according to the test design approaches for artificial and natural prestates as described in sections 7.2.1 and 7.2.2.

Figure 7.27 shows in pseudo code how the abstract step for setting the

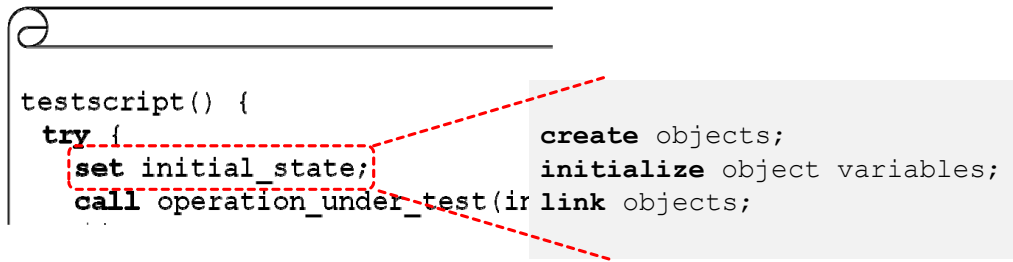


Figure 7.27: Test implementation for artificial prestate

prestate can be refined for artificial prestates. Thereby, the objects of an artificial prestate are directly generated by object constructors, their variables are assigned with concrete values and they are linked together resulting in a prestate from where on the operation under test can be invoked.

In the second approach, setting the natural prestate requires a more comprehensive way of creating and linking the objects. The preamble computed in the last section is transformed into script code. As defined in Definition 9, the preamble is a sequence of operation calls  $G_0 \Rightarrow_{op_0} G_1 \Rightarrow_{op_1} \dots \Rightarrow_{op_{n-1}} G_n$  which starts with an initial state  $G_0$  and transforms it at each operation call  $op_x$  into a new state until the end state  $G_n$  is reached. As shown in Figure 7.28 these operation calls are implemented in pseudo code.

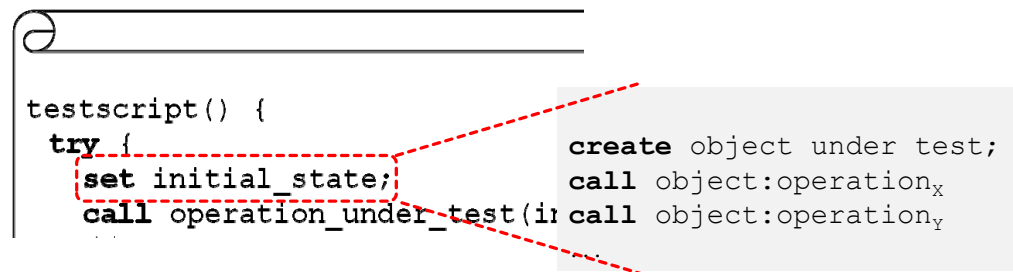


Figure 7.28: Test implementation for natural prestate

## 7.4 Test Execution

Having conceptually sketched the implementation of a test script, in this section we explain the execution of test scripts. This activity contains the assertion checks for the pre- and postconditions. In Figure 6.11 of chapter 6, using a sequence diagram we have shown the interaction of a test driver and the object under test during the execution of the test script. This interaction



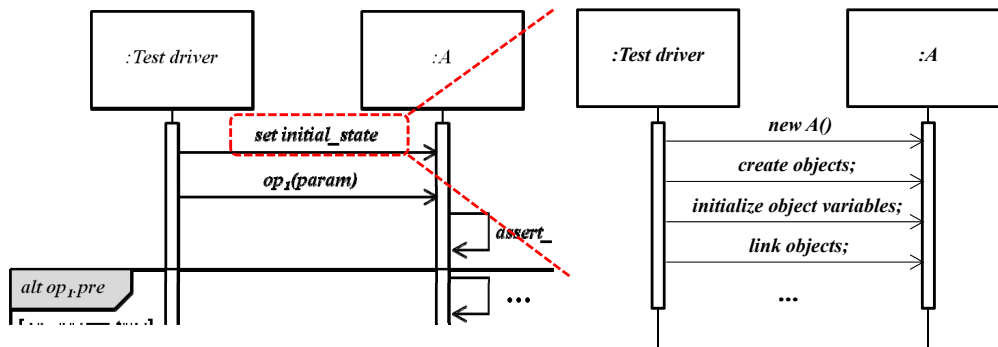


Figure 7.29: Test execution using artificial prestate

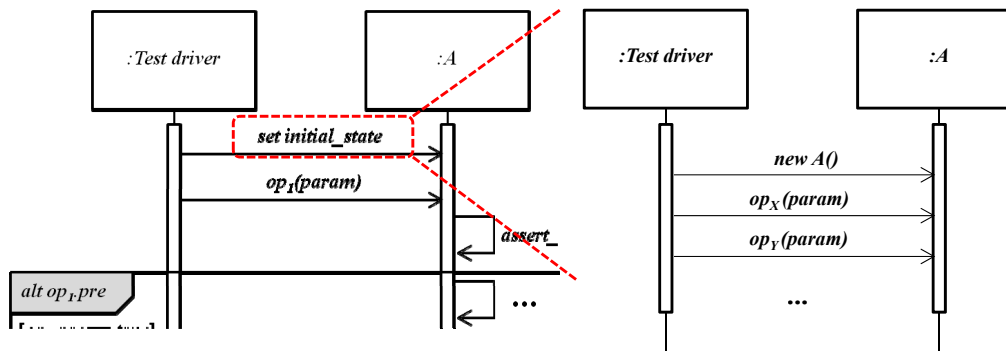


Figure 7.30: Test execution using natural prestate

comprises two main steps: First, the operation sequence determined by the test case generation must be executed in order to set prestate is set. Second, the operation under test is called with the test input parameters generated during the test case design. The sequence diagram shows also the invocation of embedded assertions for pre- and postcondition during the test execution.

Figures 7.27 and 7.28 shows the refinement of setting the prestate according to our two approaches. While the test driver one by one creates and links the artificial objects computed during test design in the first approach, for the second approach the test driver invokes the operations from the computed preamble one after another which should set the object state to the required prestate.

The embedded assertions lead to the runtime behavior as shown in Figure 7.31. When the operation under test is called, a precondition check method evaluates the method's precondition and throws a precondition violation exception if it does not hold. If the precondition holds, then the original, manually implemented operation is invoked. After the execution of

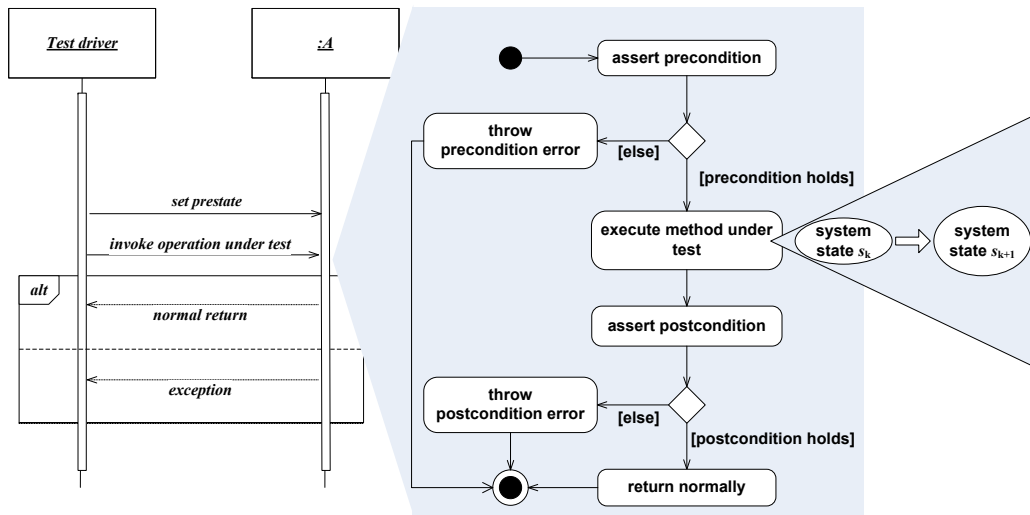


Figure 7.31: Assertion checking during test execution

the original operation, a postcondition check method evaluates the postcondition and throws a postcondition violation exception if it does not hold. If the embedded assertions throw an exception then the implementation does not behave according to its specification. Thus, we have found an error.

# Chapter 8

## Integration Testing

The last chapter dealt with Visual Contract-based unit testing, where classes and their operations are tested against their specification given by Visual Contracts. Thereby, we assumed that the classes under tests are 1) either *independent* and do not call other classes, or 2) they call classes, which are already quality assured, or 3) the classes to call are simulated by mock objects. After all classes have been tested during unit testing, in the next test level, the classes have to be tested in integration, which means that the mock objects are replaced with real classes and real interactions.

In this chapter, we show how subsystem specifications with Visual Contracts can support planning and conducting integration testing. Thereby, we analyze the Visual Contract specifications of subsystems and their classes in order to compute the dependencies between them and to compute appropriate *test plans*. In section 8.2, the test target is not to test individual components as in unit testing. The new test target is to test the interaction of components and their interfaces. For the test implementation and execution, test scripts must be derived from test cases with respect to the test plan which aims at reducing the need for mock objects or simulators. Therefore, we introduce an algorithm to compute the test order. In section 8.3, we describe the test environment and the configuration of embedded assertions for adapting the degree of monitoring of subsystem interfaces, which is the main contribution of our integration testing approach.

<i>test objects</i>	subsystems and classes
<i>test target</i>	conformance of interfaces to Visual contracts
<i>test strategy</i>	topology sorting, monitoring levels

Table 8.1: Overview Integration Testing

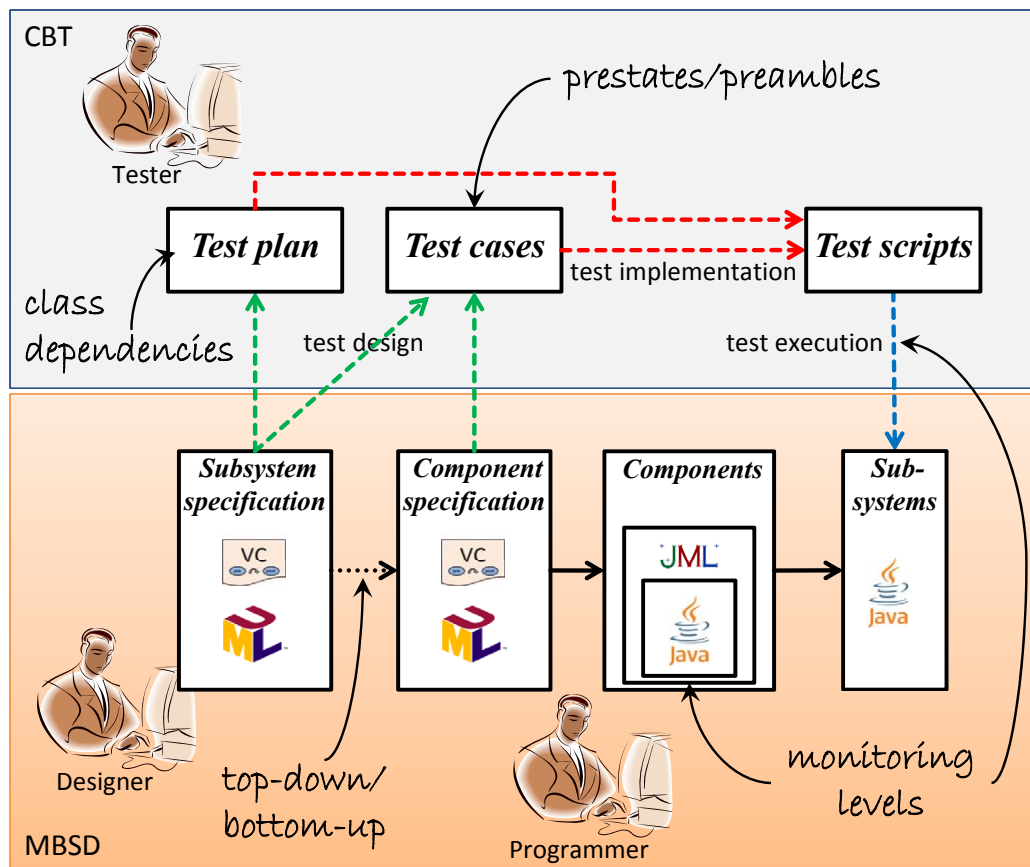


Figure 8.1: Integration Testing Process

## 8.1 Development Scenario

After the quality assurance of classes and components during Unit testing, these can be integrated into subsystems to build more complex computational units. As we have explained in chapter 6, the assembly of subsystems and the subsystem interfaces are specified during the left branch of the V-model. Visual Contracts are used for specifying the required and provided interfaces of subsystems (cf. Figure 6.2). The task of developers is to integrate the implemented classes to subsystems such that the Visual Contract specifications of subsystems are fulfilled. In order to assure that the integrated subsystems will be tested against the Visual Contracts.

Figure 8.1 extends the Unit testing process from last chapter by subsystems and their specifications in the lower box (MBSD). First, the designer specifies the architecture of the components assembly and defines interfaces between the components. As exemplarily illustrated in Figure 8.2, a subsystem

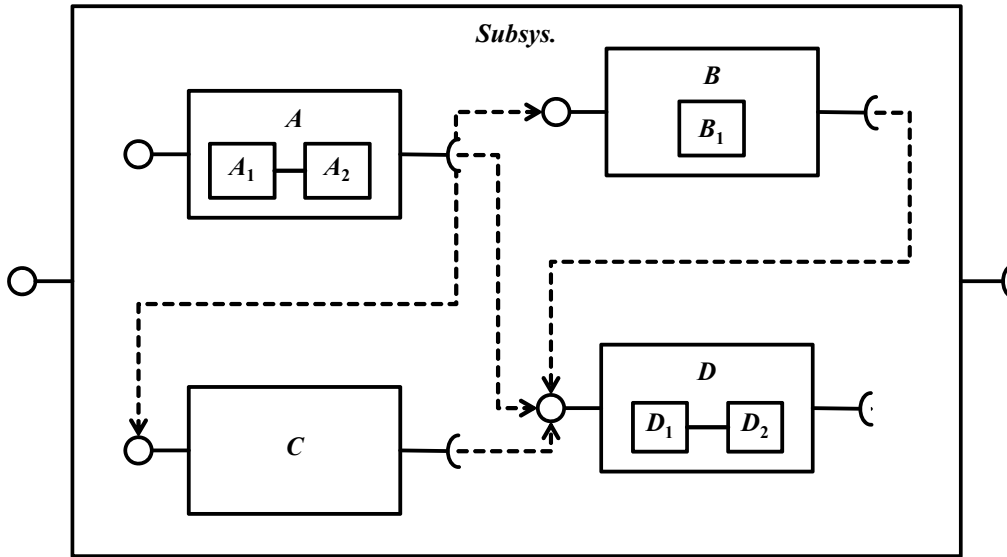


Figure 8.2: Exemplary Subsystem model

can contain other components which again are subsystems containing various components (e.g.  $A$ ,  $B$  and  $D$ ). Also third party components can be involved where the Designer or Programmer have no insight into the component (e.g.  $C$ ).

During the assembled execution, data is interchanged between classes of subsystems over their interfaces. As illustrated in Figure 8.3, a *caller* class  $A$  invokes an operation from a *callee* class  $B$  over the provided interface of  $B$ . Dependent on the state of the callee class and the input parameters of the caller class, it conducts computations and makes changes in its state and returns output values. In our design phase for subsystems and their interfaces, we specify only state changes which are visible at the interface of the components. These are specified using Visual Contracts both at the provided and required interfaces.

The semantics of a Visual Contract specification for a provided interface is, that the operations at that interface can be invoked, if the objects and their variables in input parameters of caller fulfill the preconditions of the provided interface of callee. The postcondition specifies that the callee guarantees that the objects and their variables in the output parameters fulfill the postconditions. The semantics of a Visual Contract for a required interface is, that the caller components assure the objects and their variables as input parameters as specified in the precondition by invoking an operation from a callee component. As a result of the operation invocation, it requires at

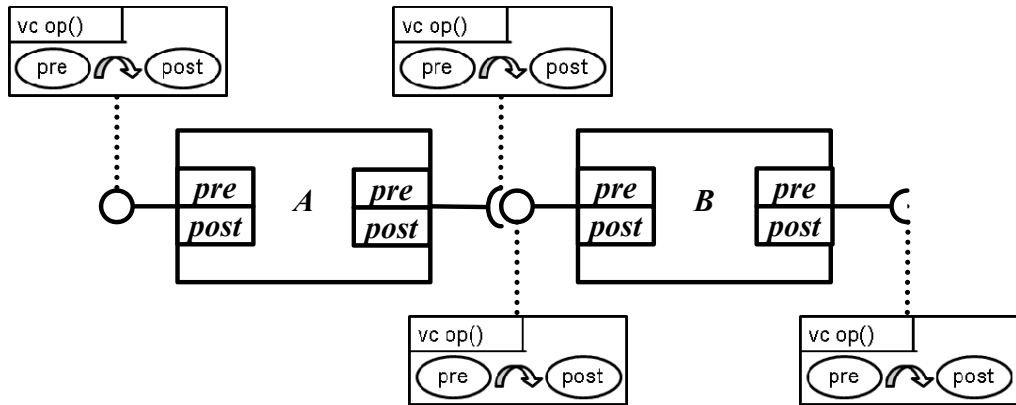


Figure 8.3: Interface specification using Visual Contracts

least the objects in the return values as specified in the postcondition. If we consider two Visual Contract specifications for a required interface of component  $A$  and for a provided interface of component  $B$  which are connected together, the following relation must hold between these two specifications (see also [Loh06]): The preconditions of the callee component  $B$  must be a subset of the preconditions of the caller component  $A$ , and the postconditions of the caller component  $A$  must be subset of the postconditions of the callee component  $B$ . In other words, the callee component can get more inputs that it requires and it can return more outputs than the caller component expects.

For the integration phase, besides the architecture of the subsystems and components and their interface specifications, also the order of their integration is of big importance. The order of integration is relevant in case of availability of components to be integrated and in case of dependencies between components. Thereby, two strategies can be followed for the integration of components to subsystems: *bottom-up* integration and *top-down* integration.

Bottom-up integration starts with components which have less dependencies and thus reside on the bottom of a call hierarchy. These components are functionally self-contained, which means that they do not require operations from other classes. If such a component is implemented and tested successfully, it can be used during the integration of other components which reside higher on the call hierarchy. The disadvantage of bottom-up integration is that the overall functionality of the subsystem will not be available until all other components on the top of the call hierarchy are also integrated.

Top-down integration means, that integration activity starts with compo-

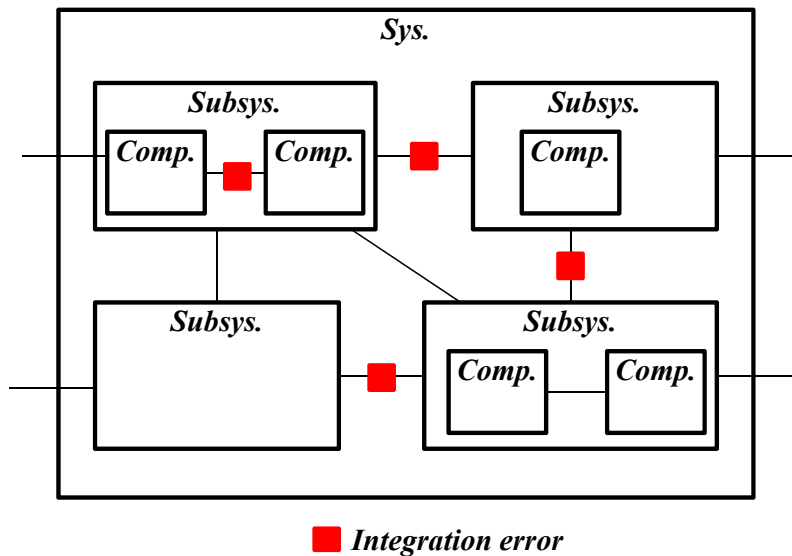


Figure 8.4: Fault model component integration (based on [WEMS<sup>+</sup>12])

nents which reside on the top of the call hierarchy and which have the most dependencies to other components. In this strategy, the overall functionality of the subsystem can be invoked over the interface of the already existing components on the top of the call hierarchy. However, if lower components do not exist yet or if these are not quality assured yet, the subsystem will require some mocks and stubs to simulate the functionality of the missing components.

During the assembly of components, integration errors can be induced as illustrated in Figure 8.4. Since we assume that the individual components are already tested during Unit testing, now we focus on the communication and data interchange between the components to be integrated. Thereby, the exchange of information among the operation invocations are checked for *completeness* and for *semantic* and *syntactical* correctness.

## 8.2 Test Design

Chapter 7 dealt with testing of components and classes using unit testing techniques. Thereby test cases are derived from Visual Contracts and the state changing behavior of low level software components are tested. If errors are detected, they have to be fixed before routing them to the next development phase. However, if no errors are detected, the question remains, whether the tested components also function correctly if they are executed

in integration with other components.

The issue in integration testing is to validate the conformance of data interchange between components for conformance with the Visual Contracts for provided and required interfaces. As illustrated in the upper box (CBT) of Figure 8.1, three main concepts are crucial for our integration testing process: 1) the order in which the components are tested has to be defined, 2) test cases for invoking interface operations have to be derived, 3) during test execution, the data interchange has to be monitored by the embedded assertions. In this section, we will describe how the test order can be defined by considering the component dependencies and how test cases can be derived from Visual Contracts. The test execution will be the topic of next section.

Since we deal with components with different size, *integration testing* becomes different meanings. Winter et al. define four types of integration testing in [WEMS<sup>+</sup>12] with respect to the granularity of software components to be integrated:

- Member integration testing
- Class and module integration testing
- Components and subsystem integration testing
- System integration testing

Applying to our exemplary components assembly in Figure 8.2, the member integration testing addresses the integration of the member classes  $A_1 - A_2$  and  $D_1 - D_2$ . Class and module integration addresses the integration of classes  $A, B, C$  and  $D$ . Components and subsystem integration would address the integration of *Subsys.* with other subsystem. Even if the techniques described in this chapter can also be applied to system integration testing, to keep it simple, we will not go further into that type.

Besides the granularity of software artifacts to be integrated, also the order of their integration is of big relevance. As explained in the last section, the two integration strategies bottom-up and top-down address the availability of components and the dependency between them. While the availability is an issue of test management where the components can be integrated and tested according a time schedule, the dependency between components must be detected by formal analysis. This can be done by *topology sorting* as proposed by many testing experts [Bin99, Bor09, WEMS<sup>+</sup>12].

Borner summarizes in his thesis [Bor09] many techniques for defining the order of integration. We have adapted a graph based technique to our context, where we consider each integrable component as a graph node. Starting



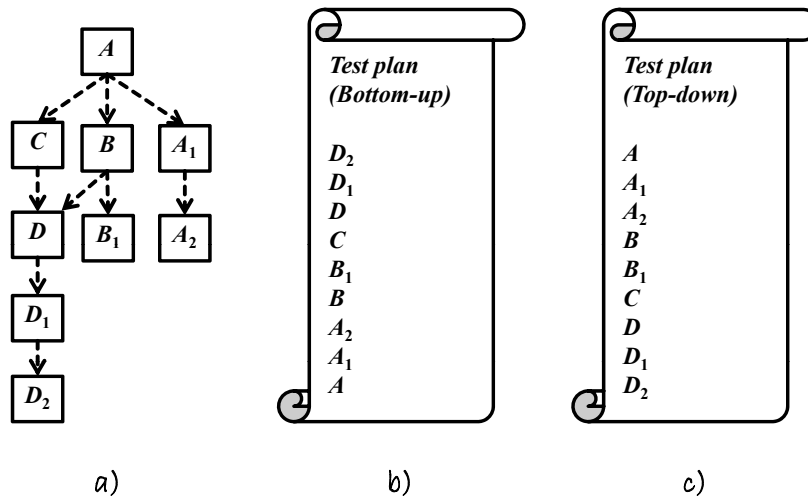


Figure 8.5: a) Topology sorting, Test plan for b) bottom-up and c) top-down integration

with callee components with no required interfaces, we stepwise identify the caller components and sort these hierarchically such that we end up with a topology sorting [Kah62]. Thereby, we assume that the components do not have cycles in their dependency relations. Figure 8.5-a show the topology sorting of the components in the Figure 8.2. Thereby, the components  $A_2$ ,  $B_1$  and  $D_2$  are callee classes with only provided interface. Based on this sorting, we can now define concrete *test plans* for integration strategies *top-down* (Figure 8.5-c) and *bottom-up* (Figure 8.5-b).

In the bottom-up strategy, testing activities begin with the components which have no required interfaces, i.e. which do not depend on other components. In our example, the bottom-up integration testing can start either with  $A_2$ ,  $B_1$  or  $D_2$ . Then, the next component in the topology order will be integrated and tested. One concrete order beginning with  $D_2$  and ending with  $A$  is shown in Figure 8.5-b. The advantage of bottom-up strategy is that the components under test requires no mocks or stubs during testing, because the topology sorting assures that the required functionalities of a component are available and already tested, thus these can directly be involved in the test environment. Besides its advantages, bottom-up integration always needs to implement a test driver for each new component to be integrated.

Top-down integration starts with testing top-level caller components which invoke other component operations over their provided interfaces (Figure 8.5-c). Thereby, at each integration step, the top-level component acts as a test

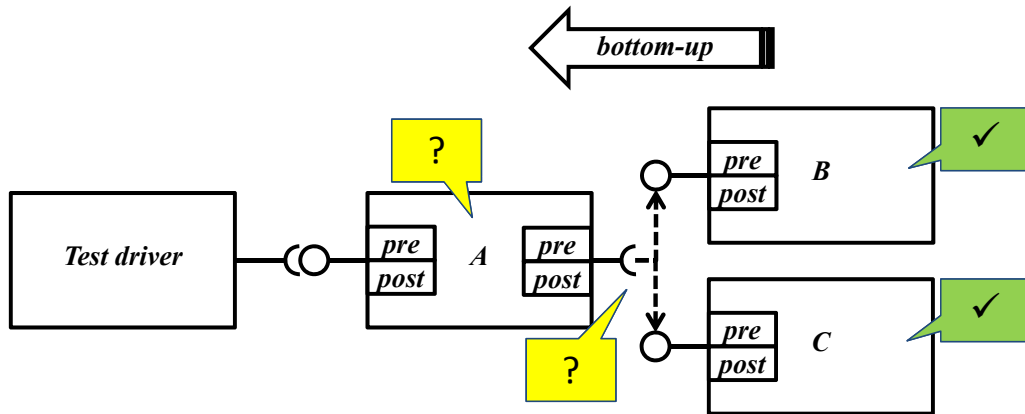


Figure 8.6: Bottom-up integration testing

driver for other components. This makes top-down integration more efficient compared with bottom-up integration, because no additional test drivers are required. However, if some lower components are missing in the call hierarchy, these have to be simulated by mock objects or simulators, which is the main disadvantage of the top-down integration.

### 8.3 Test Implementation and Execution

Independent of the integration strategy, concrete test cases for the invocation of the interface operations are derived using the same techniques as in Unit Testing. We rather prefer artificial prestates for integration testing in order to strictly control the prestate for the operation under test. In the natural prestate approach, preambles could be required which contain operation invocations which are not part of the test plan according to the integration strategy.

The test scripts for integration testing are similar to those of unit testing, where first a prestate is set, an operation under test is invoked and according to the assertions checks the test verdict is set. However, the difference of integration testing lies in the test environment and the test execution. The test environment does not only consist of the component under test and a test driver as in unit testing, it contains more than one component, if required the test driver and mock objects or stubs for simulating callee components.

Figure 8.6 shows the test environment for bottom-up integration testing. Thereby, we assume that the most low-level components in the topology sorting *B* and *C* are already tested during unit testing. Thus the test target here is first to test the functions of component *A* in assembly with the other

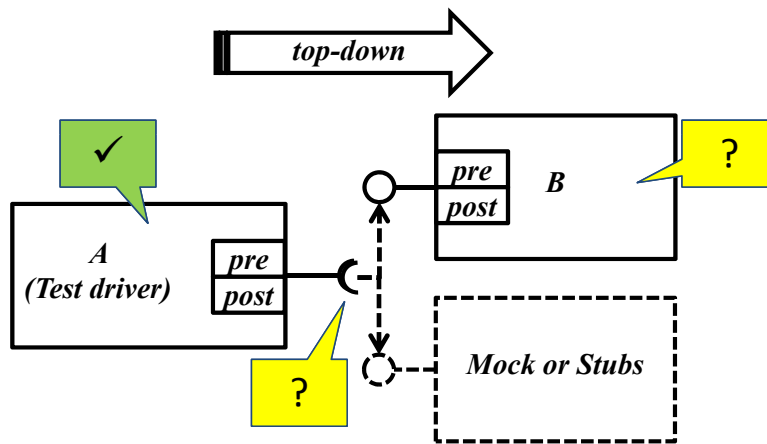


Figure 8.7: Top-down integration testing

two components and second to test the data interchange between *A* and *B*, and between *A* and *C*. In order to test *A*, a test driver invokes operation at the provided interface of *A*. Test cases for those invocations can be created based on the provided Visual Contracts by using techniques from unit testing.

In top-down integration, testing starts with top-level components. As explained in the last section, the top-down integration requires the simulation of low-level components in the call hierarchy if these are missing. As shown in Figure 8.7, we assume that the top-level components are already tested roughly during unit testing, however only by simulating all component dependencies. This can be done by traditional testing artifacts stubs and mock objects. Fowler defines these two artifacts in [Fow07] as follows:

“Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what’s programmed in for the test.

Mocks are ... objects pre-programmed with expectations which form a specification of the calls they are expected to receive.”

By using Stubs or Mocks, we can test the components *A* and *B* in assembly. Thereby, we can instrument the component *A* as a test driver for the component *B*. Test cases are derived on the basis of provided Visual Contracts of component *B*. Also at this integration strategy, we are interested in testing the data interchange between the components.

In order to monitor the data interchange, we instrument the embedded assertions for interaction monitoring. Even if we cannot monitor the whole transfer of data between two components, we can monitor the interfaces

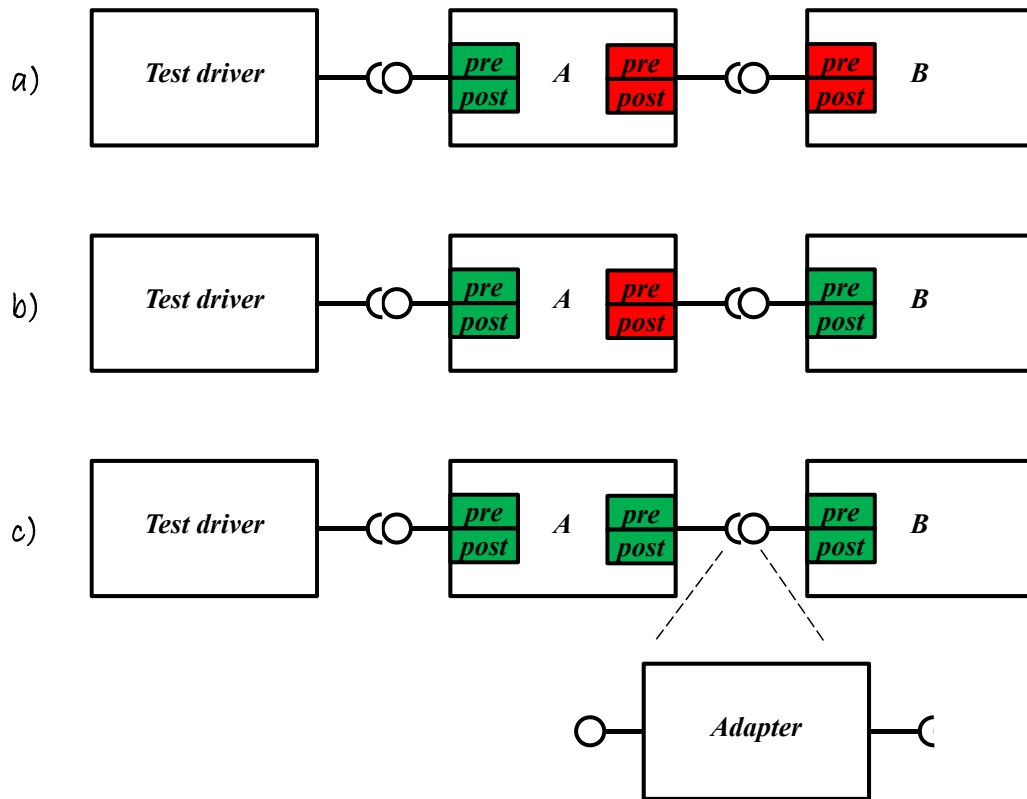


Figure 8.8: Levels of monitoring

locally, such that the incoming and outgoing data can be checked for conformance with Visual Contract specifications. For that, we adapt the *model driven monitoring* technique proposed by Lohmann [LES06] (cf. chapter 4).

Figure 8.8 shows different levels of monitoring which can be realized by the embedded assertions generated from Visual Contracts. Since monitoring can negatively effect the performance of the test execution, the embedded assertions can be switched *on* and *off*. If testers are interested only in the functional correctness of a component *A* as observed at the provided interfaces of *A*, the embedded assertions for the provided contracts of *A* can be switched on, while other assertions of interfaces of *A* and *B* can be switched off (see *a* in the Figure). The most powerful monitoring level is illustrated in scenario *c* where all embedded assertions are activated for monitoring every provided and required interface. This scenario is especially relevant for checking the correctness of the interaction if *Adapters* are used between the two components, because the conversion of adapters can be error-prone.

Figures 8.9 and 8.10 illustrate the monitoring activity of embedded assertions in case of monitoring levels 2 and 3 correspondingly. Both interactions

begin with setting the prestate. After the execution of operation under test  $op_1$  with input parameters, first the embedded assertions for preconditions of  $op_1$  activated. The most outer alternate box specifies the further progress in case of the fulfillment or not fulfillment of the preconditions. To begin with the the simple case, if the precondition is not fulfilled, the object  $:A$  returns with a *precondition\_exception* as shown in the lower half and the test execution ends with an *erroneous* verdict (cf. chapter 6). If however, the preconditions are fulfilled, the test execution continues with further operation invocations of object  $:A$  on object  $:B$ . If object  $:B$  returns with an exceptional case, the exception will directly forwarded to the test driver by object  $:A$  which again leads to an erroneous verdict. Only if the object  $:B$  returns normally, the postcondition assertions of object  $:A$  becomes relevant for defining the test result. If the postcondition of  $:A$  is fulfilled, it returns normally and the verdict is set to *pass*. In case of a *postcondition\_exception*, that means that the return data of  $:B$  does not conform to the expectations of  $:A$ . Thus the verdict is set to *fail*.

In monitoring level 3, in addition to the assertion checks for provided Visual Contracts, also the assertion for required Visual Contracts are executed during testing. As explained in Figure 6.5 in chapter 6, for the required Visual Contracts, delegation operations are generated. Thus, in this monitoring level, also the assertion for delegated operations are executed.

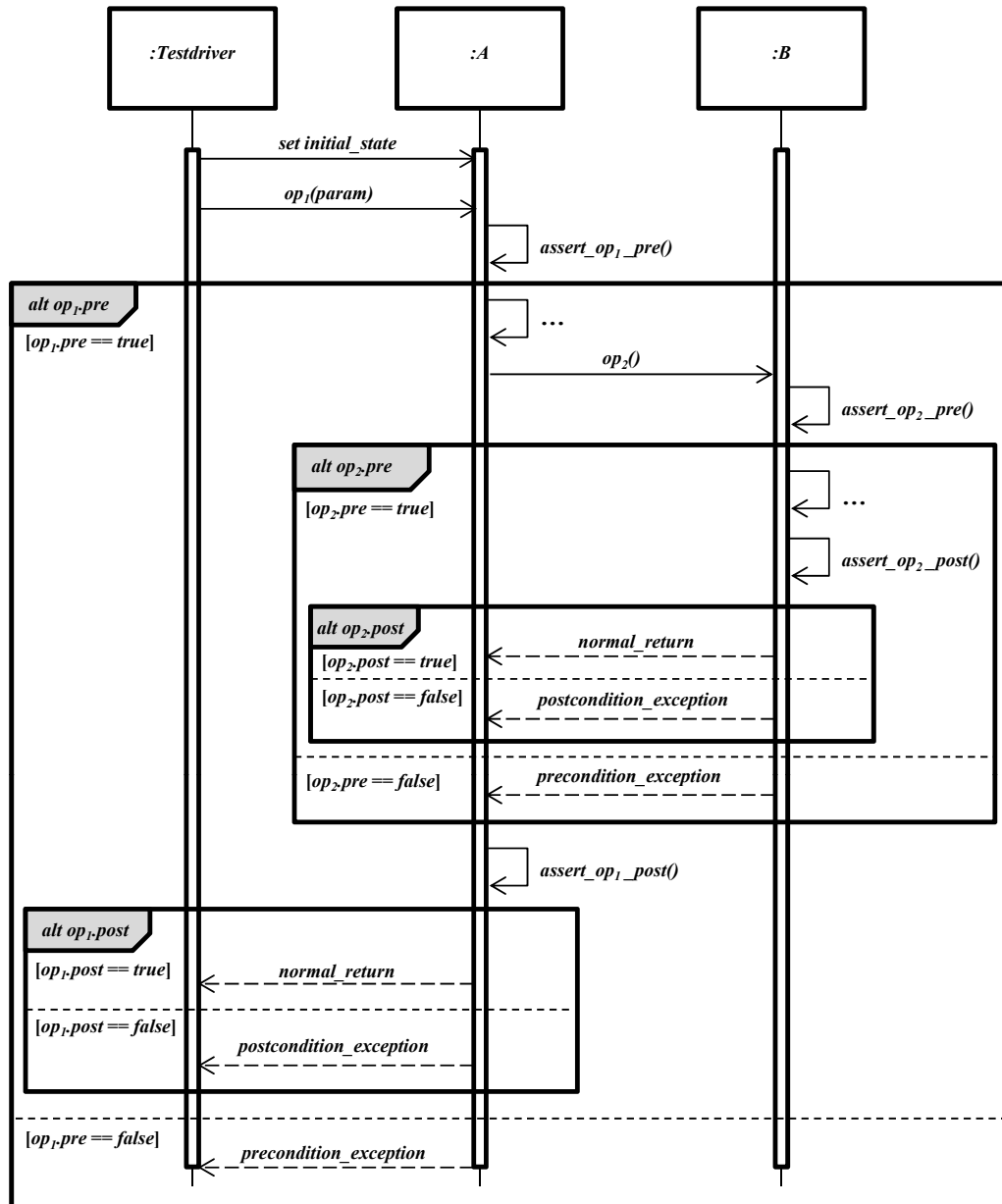


Figure 8.9: Level 2 monitoring

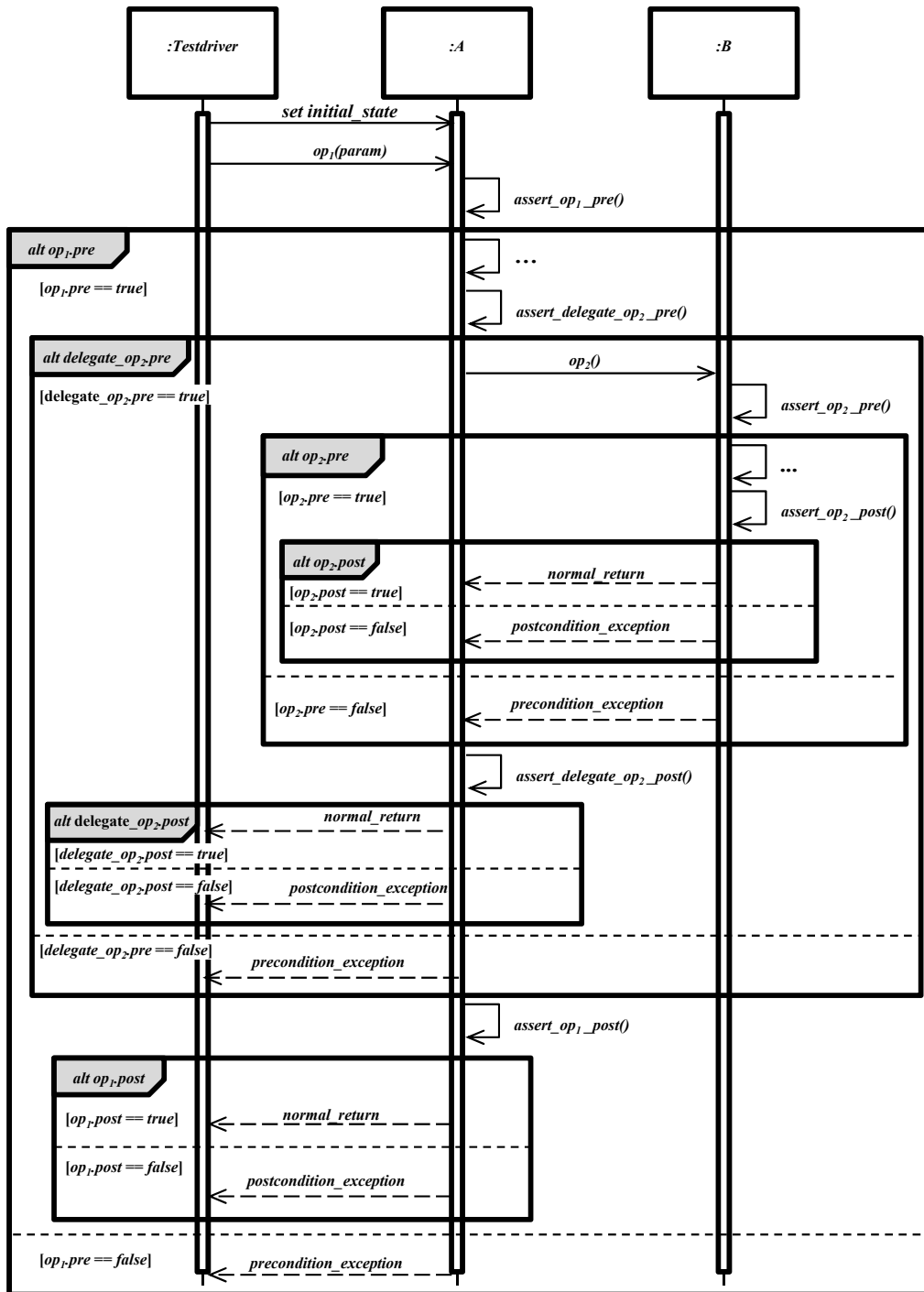


Figure 8.10: Level 3 monitoring





# Chapter 9

## System Testing

The chapters 7 and 8 have dealt with contract-based testing of low-level software components, e.g. classes, modules and subsystems. Thereby, test cases are derived from low-level Visual Contracts. In these test levels, because testing activities start directly after the development activities, the semantic distance between the Visual Contracts and the implemented software is low. Thus, the object states specified by the test cases can directly be transformed into program code and used for testing.

On the test level *system testing*, the software under test is a composition of the tested subsystems. The composite system is tested against the Use case specifications to show that it fulfills the functional requirements. However, the Use cases are created generally very early in the development process and they do not cover technical details about how the abstract concepts are to be implemented later in the development process. Thus, the semantic gap between the Use cases and the implemented software is big. Furthermore, the informal Use case descriptions must be formalized for a systematic and automated test design.

This chapter shows at first, how the informal Use case specifications can be formalized using Visual Contracts and used for a systematic and automated derivation of test cases. Then, we show how abstract test cases can be transformed into executable test scripts by using model transformations in order to bridge the gap between abstract requirements and the implementation.

<i>test objects</i>	composite system
<i>test target</i>	conformance of composite system to Use cases
<i>test strategy</i>	refinement of test cases by model transformations

Table 9.1: Overview Integration Testing

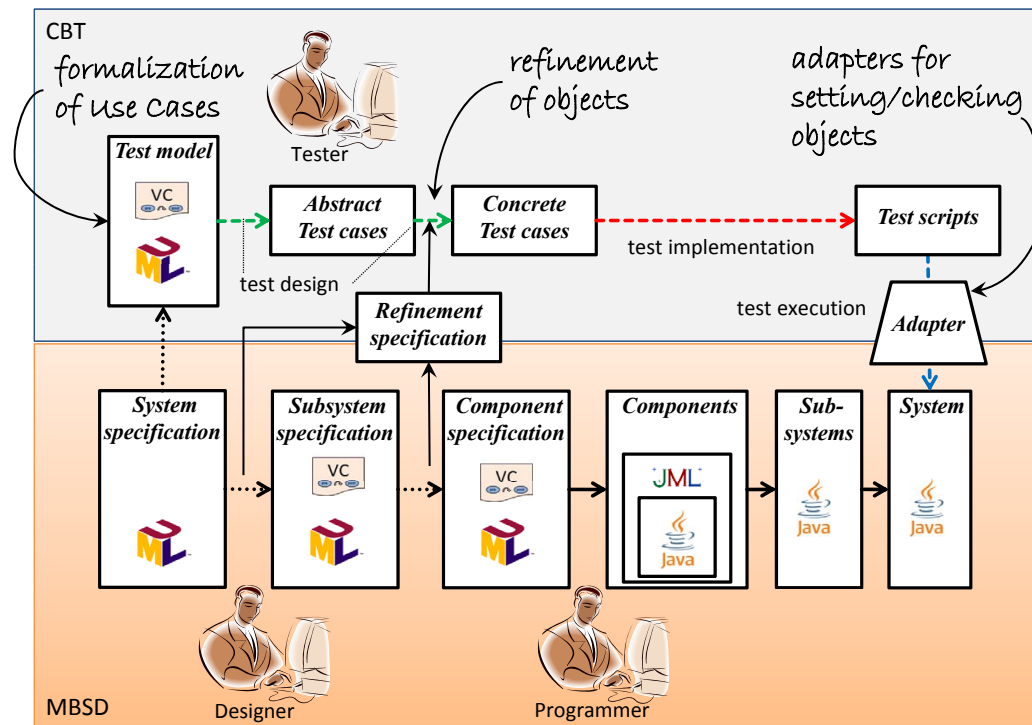


Figure 9.1: System Testing Process

## 9.1 Development Scenario

Every software development process begins with specifying the system requirements from the end users point of view. In UML-based development processes, requirements specification is typically done by using the Use Case modeling. The system analyst captures the functional requirements using a Use Case template [Coc01] including e.g. the preconditions, Use Case steps and postconditions. These elements in a Use Case specification are typically formulated in natural language, such that all stakeholders incl. customer, designer, developers and testers can understand them. From these Use Cases, team members derive their own artifacts, e.g. design documents, test cases as shown in our general approach in Figure 6.1.

The usage of natural language for Use Case specification has both advantages and disadvantages. The biggest advantages are the readability, understandability and editability of Use Cases by all stakeholders. No special skills or comprehensive trainings are required for reading and editing Use Cases. However, the disadvantage of Use Cases in natural language is, that they can be ambiguous and thus lead to misunderstandings between the requirements analysts and the members of design team and test team. By formalizing

Use Cases, system requirements can be specified in a more consistent and clear way. The following citation from Hsia et al. [HKS97] addresses the role of formalism for requirement specifications, not only for creating better specifications, but also enabling automation of some activities:

“Researchers have shown that using formal specification languages in the requirements phase can reduce the effort in acceptance testing because it helps one to generate test cases automatically [Ferguson and Korel 1996] [FK96]. However, formalisms are often used at the expense of communication between customers and developers, since most customers are not well versed in formal specification languages. This approach may be useful and effective in certain domain specific applications as it can facilitate the automation of many activities required in the software development process. However, the prospect of popularizing this approach is probably not very bright.” [HKS97]

In our model-based development process as illustrated in Figure 9.1, we require modeling languages which are both *understandable* for communication purposes and *formal* for automation purposes. The central development artifact we deal with in this chapter is the *System specification* (see lower box) which contains Use Cases for functional specification. The following information must be specified to capture functionalities using Use Cases [Coc01]:

- The activation of the functionality (*trigger*)
- The requirements of the functionality (*preconditions*)
- Standard and alternative scenarios for the functionality (*steps*)

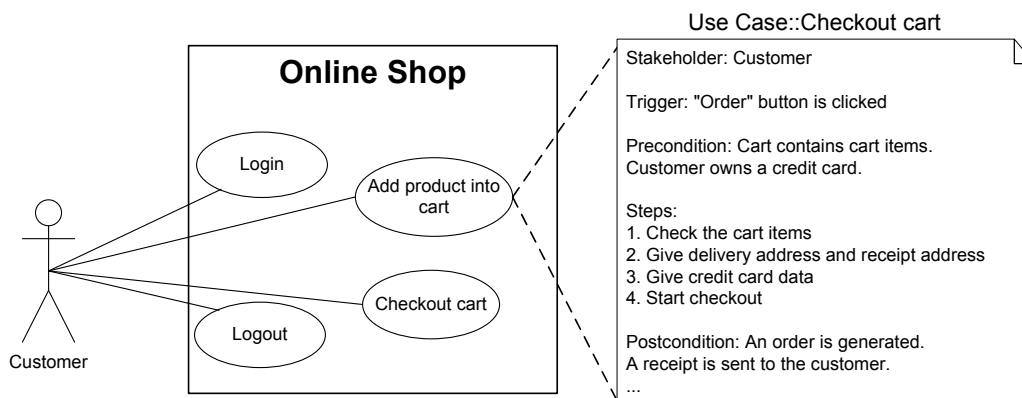


Figure 9.2: Test basis for system testing

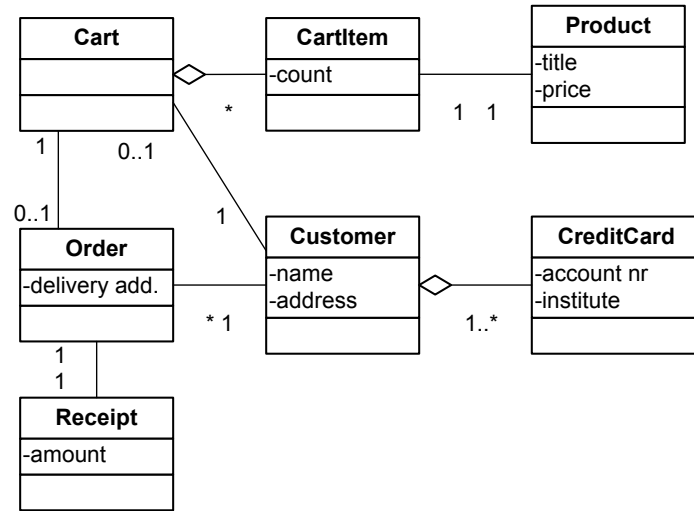


Figure 9.3: Domain model for Online Shop

- The expected results after execution of the functionality (*postconditions*)

Having this structure and described in natural language, Use cases ensure an understandable specification of the required system functionalities. Based on the Use Cases in the System specification, developers stepwise create Subsystem specifications and Component specifications as the basis for the implementation as explained in chapters Integration Testing and Unit Testing.

Figure 9.2 shows an exemplary Use Case description of an Online Shop: the customer can add product items into a shopping cart and order these by checking out. The functionality Checkout Cart is described in the figure. In order to checkout the cart the customer has to click the Order button (trigger). However, before the button is clicked, some products have to be added to the cart and customers' payment method must be known (preconditions). If the preconditions are fulfilled and the trigger is activated, the scenario is stepwise conducted. This scenario must result in a (generated) order and a receipt which is sent to the customer (postcondition).

Besides Use Cases, the System specification also contains a *Domain model* for capturing the important concepts and the relations of these concepts to each other. It is created by using *Class diagrams*. Figure 9.3 shows an exemplary domain model for an Online Shop. This model is created by analyzing the requirements descriptions in the Use Cases and identifying the important concepts and their relations. Since these concepts are described in Use Cases in an abstract manner, also the resulting Domain model is abstract. In

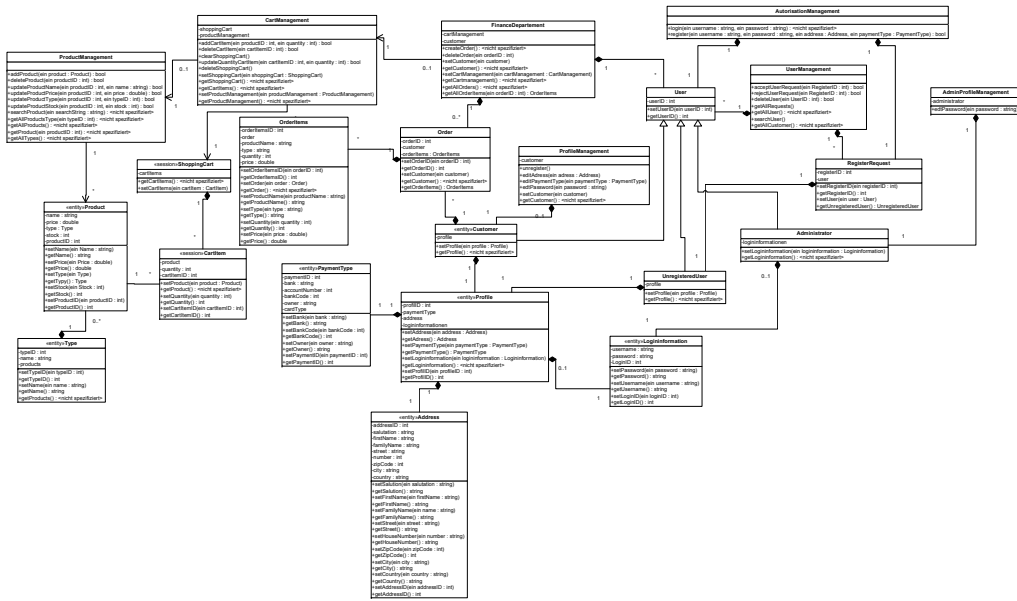


Figure 9.4: Implementation model for Online Shop

MDA, models of this level of abstraction are referred to *Platform independent models* (PIM), since they do not handle platform specific technical issues.

During the stepwise refinement of the System specification into subsystem specifications and component specifications, developers make decisions on technical issues, e.g. which subsystems are required, which software architecture is to be used, which third-party libraries are to be used. The Domain model evolves with these technical decisions into *Platform specific models* (PSM). Figure 9.4 shows a fragment of the *Implementation model* for the Online Shop, which specifies implementation details for the concepts from Domain model. In this example we see that the developers have decided to use the model-view-controller (MVC) architecture and they have specified which concepts from the Domain model are to be implemented as Controller classes, Entity classes and View classes. These classes have also been assigned Attributes and Operations.

In our development scenario, we assume that such design and implementation decisions are documented in a new development artifact *Refinement specification* in a formal way, such that these can be re-used by other members of the development team [Kön10] (cf. Figure 9.1). For the formal documentation of refinement specifications, a formal language is required which is compatible with the UML-based specification techniques. The OMG proposes in MDA guide [Gro03a] model-to-model transformations (M2M) for

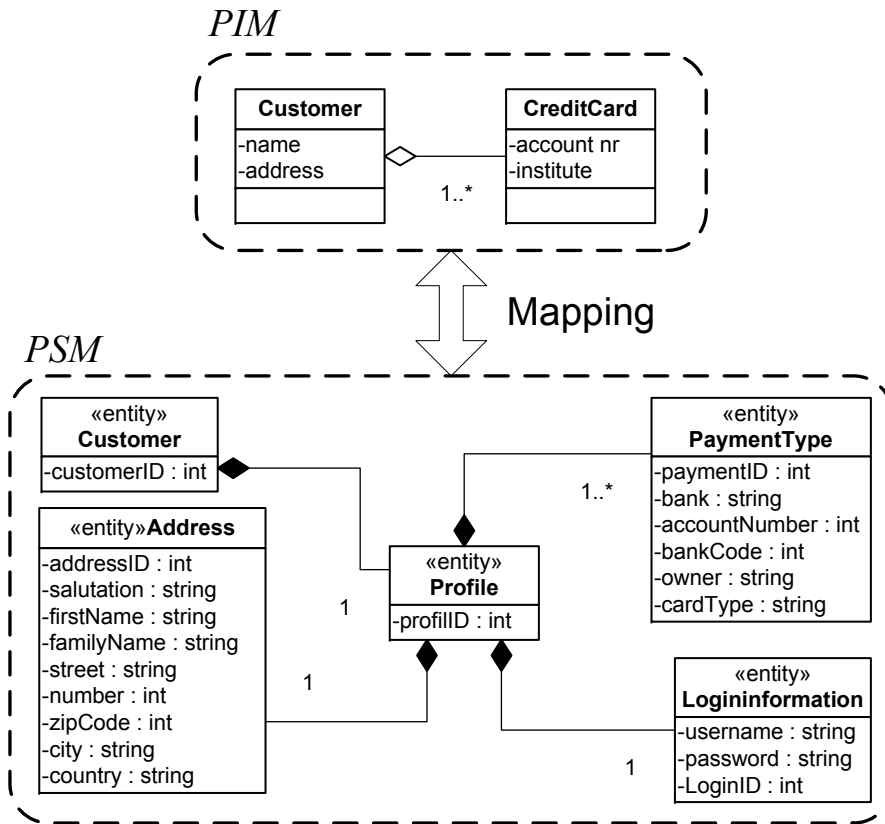


Figure 9.5: Mapping between PIM and PSM

specifying transformations from a PIM to a PSM. M2M transformations can be specified by languages Queries/Views/Transformations (QVT) [Gro05b] and Triple Graph Grammars (TGG) [KS06]. The transformation specification, usually a set of transformation rules, defines how particular elements from a source model (e.g. a PIM) are mapped to a target model (e.g. a PSM) [MG06].

To give an example, the mapping in Figure 9.5 sketches the decision of how the classes *Customer* with *CreditCards* from PIM are refined in the PSM. Several technical details were added to the classes: a *Profile* connects further information to the *Customer*; namely a detailed *Address*, a *LoginInformation*, and a more detailed representation of credit cards as *PaymentType*. Moreover, each class in the PSM-part of the mapping is stereotyped as an *entity* and has an attribute holding a unique identifier. Figure 9.6 shows the formalization of the mapping as a transformation rule in QVT.

Lines 2-5 define the transformation rule *Logical2Executable* with the source

```

1  ...
2  transformation Logical2Executable(
3      in src:PIM,
4      out tgt:PSM
5  )
6
7  main() {
8      src.objectsOfType(Customer)->map cust2cust ();
9  }
10
11 mapping Customer::cust2cust():Customer {
12     customerID := random();
13     ...
14     self.objectsOfType(CreditCard)->map card2pay();
15 }
16
17 mapping CreditCard::card2pay():PaymentType {
18     paymentID := randomInteger();
19     accountNumber := self.accountnr;
20     cardType := self.institute;
21     ...
22 }

```

Figure 9.6: Transformation rule from PIM to PSM

model of type PIM and the target model of type PSM. Lines 7-9 define the starting point of the transformation where for all instances of the *Customer* class the subtransformation *cust2cust* is invoked. Lines 11-15 define that the subtransformation *cust2cust* which first generates a random *ID* number. Then for all instances of the class *CreditCard* which are linked to the *Customer* object another subtransformation *card2pay* is invoked where attributes of the *CreditCard* object are used to instantiate attributes of *PaymentType* object (Lines 17-22). For simplicity, further details of the transformation concerning the classes *Address*, *Profile* and *Logininformation* are not shown here.

Having a set of of such Refinement specifications, other members of the development team can reuse them for own purposes [Kön10]. In our approach, we propose to use them for testing purposes [GMWE09]. As shown in the upper box of Figure 9.1, our testing process begins with the formalization of Use Cases by using Visual Contracts and storing them as separate *Test models*. Then *Abstract test cases* can be automatically derived from *Test models* and transformed stepwise into executable *Test scripts*. Thereby, the test objects in the abstract test cases are refined and concretized due to the *Refinement specifications*. The *Concrete test cases* can then be transformed into *Test scripts* which contain enough technical details for being executed

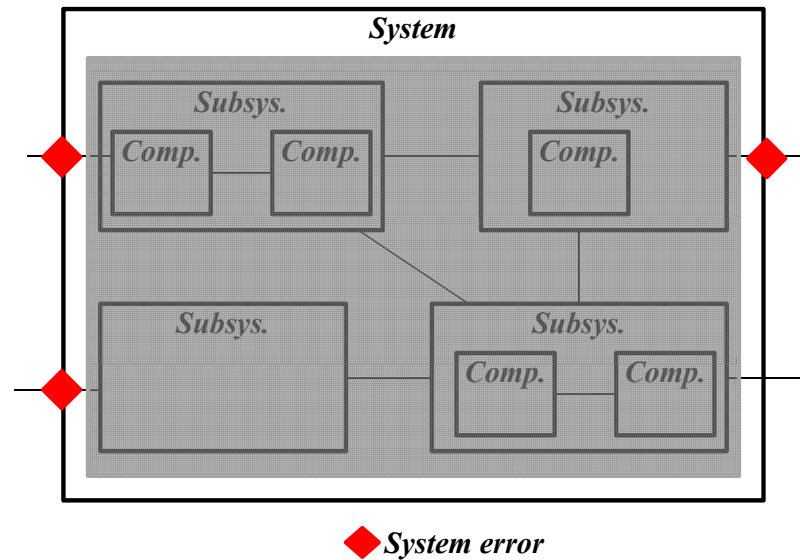


Figure 9.7: Fault model for system testing

by the test driver. An *Adapter* helps in setting test objects in the system state and checking the changes on the system state after the execution of system functionalities.

The contribution of the above sketched testing approach lies in the systematic refinement of abstract test cases into executable test scripts. For the refinement, developers' design decisions are reused. In this way, the development and testing activities evolve in parallel and the semantic gap between abstract and concrete testing artifacts can be filled. In the following, we explain how the test models are created, how abstract test cases can be derived from test models and refined to concrete test cases and finally how test scripts can be implemented and executed.

## 9.2 Test Design

While integrating subsystems to the composite software system, errors can be induced into the software, or errors which were not detected during Unit testing and Integration testing can be detected after the integration of the subsystems. The target of system testing is to detect such errors by invoking the system features on the system interfaces (see Figure 9.7). In other words, during system testing, we are trying to detect system errors which can be triggered using the system interfaces.

For detecting system errors, Use case specifications are used as test basis.



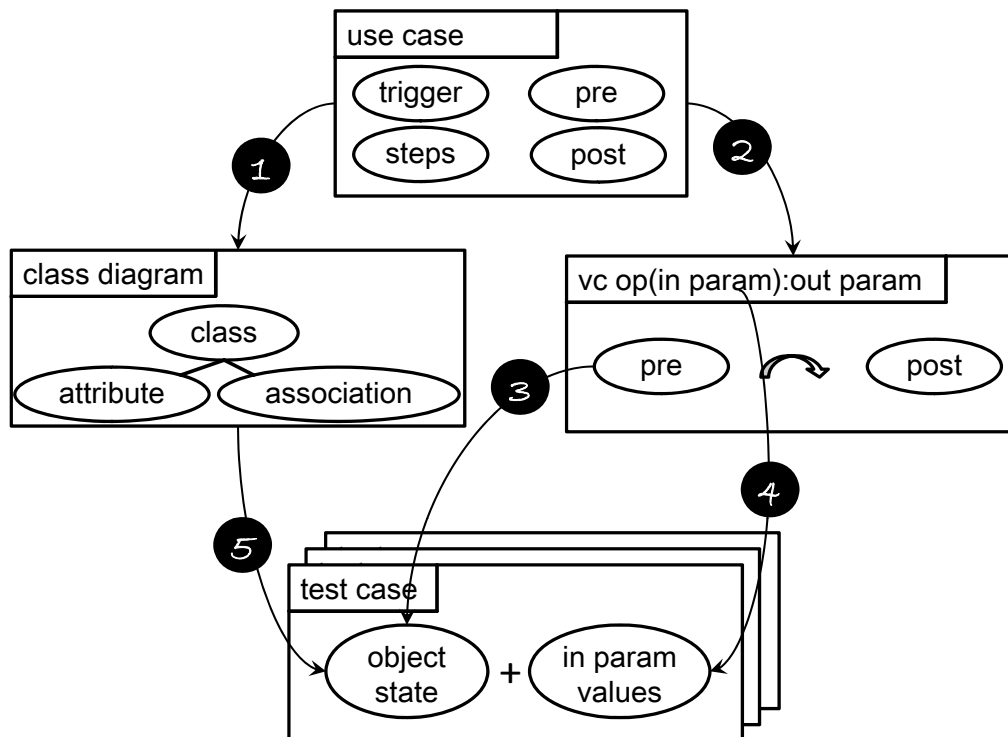


Figure 9.8: Process of test design

Since we are mainly interested in testing the state changing behavior specified in a Use Case, we create formal models of the textual descriptions of pre- and postcondition by using Visual Contracts. Figure 9.8 shows the process of deriving test cases from the use cases in five steps. Having already explained the derivation of a domain model (**step 1**) in form of a class diagram in the last section, we explain how to derive Visual contracts in **step 2** from the Use case specifications.

Figure 9.9 shows an example for modeling the pre- and postconditions of the Use case in Figure 9.2 which are modeled as two object diagrams. The object structure in the left hand side show the formalization of the textual precondition in the Use case as given in Figure 9.2: “Cart contains cart items. Customer own a credit card.”. Instances of classes *Cart*, *CartItem*, *Product*, *Customer* and *CreditCard* are specified and linked to each other conforming to the domain model in Figure 9.3. This object structure must be fulfilled in order to invoke the system functionality *Checkout Cart*. After the invocation, the following postcondition must hold: “An order is generated. A receipt is sent to the customer.” Thus the object structure on the right hand side show newly created instances of classes *Order* and *Receipt*. The delivery

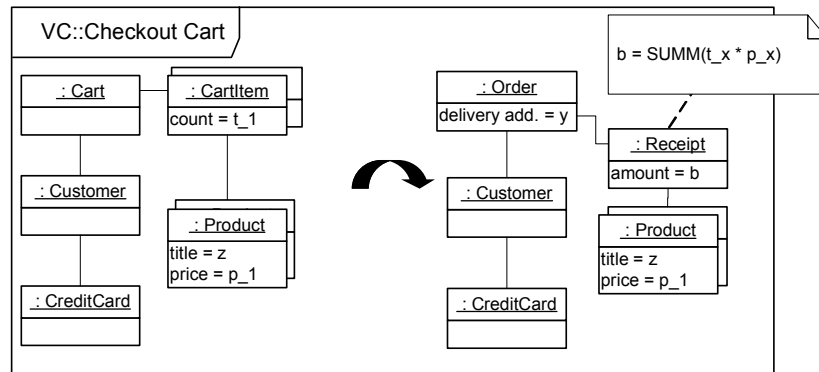


Figure 9.9: Visual Contract for formalizing the Use case *Checkout Cart*

address variable of instance *Order* gets the value of the address variable of instance *Customer*, and the amount variable of instance *Receipt* gets the summed value of the product prices multiplied by the count of *CartItems*. As explained in chapter 4, Visual Contracts are designed to specify objects and objects relations, however, in this chapter, we also add some *constraints* to Visual contracts that specify variable values depending on other object variables.

After the formalization of the pre- and postcondition of the Use cases in form of a Visual contract, this can be used for a systematic test case design. In **step 3**, object constellations are selected using the same techniques like in Unit Testing, which are then completed in **step 5** with further objects, if the domain model specifies further object relations. Input parameters for the test invocation are derived from the input parameters of the system features in **step 4**. After these steps, test cases are derived which have the same abstraction level as the domain model and the Visual contacts.

As explained in chapter General Approach, our definition of a test case comprises a *prestate* and *input parameter values*. During Unit Testing and Integration Testing we have not computed the expected poststate explicitly, but checked the real poststate for conformance with the embedded assertions. In this way, we could decide on the success of test case. In system testing, we propose another approach, where we define the expected poststate in an abstract manner at design time. However, it is not generated directly from the postcondition, which is the case in prestate generation from the precondition. Instead, the expected poststate for a prestate, which is derived from the precondition using the test case selection techniques described in section Approach 1: Artificial Prestate, is computed using the formal graph transformation semantics of Visual contracts. Therefore, the Visual contract

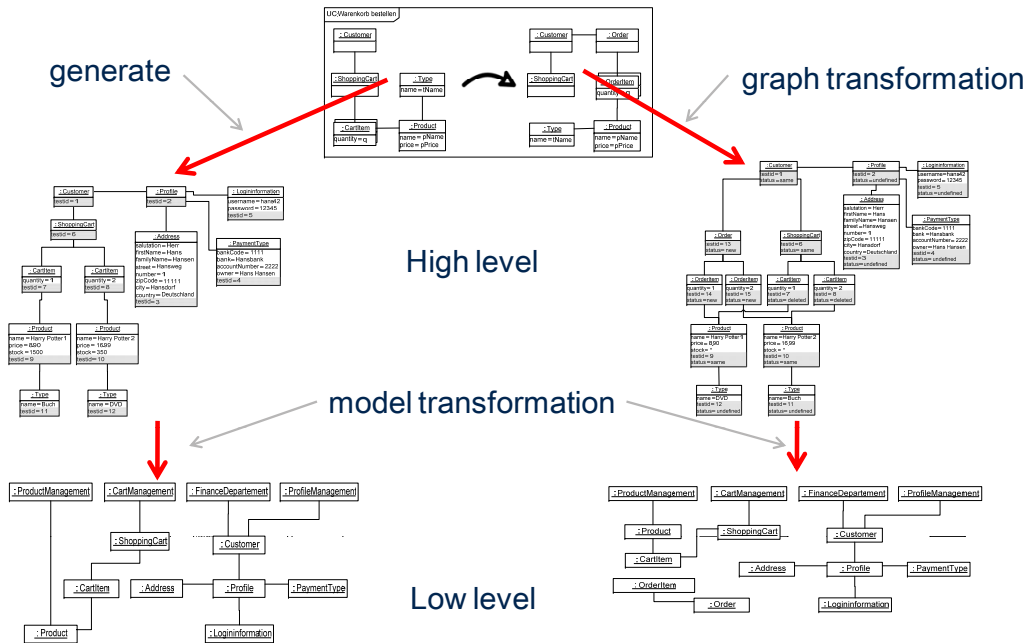


Figure 9.10: Transformation of the test inputs and expected outputs

is applied as a *graph production rule* on the prestate as described in section Approach 2: Natural Prestate [Kla08, GMWE09]. In this case, the prestate is handled as the *source graph* of a graph transformation (see Figure 9.10). The object structure resulting from the graph transformation comprises the *target graph*. Thus graph transformation semantics of Visual contracts acts as a test oracle. After the derivation of the prestate and the computation of the expected poststate, these high-level object structures have to be transformed into low-level object structures for the test execution.

For transforming high-level objects structures to low-level objects structures, we propose to reuse the Refinement specifications of developers (see last section) as transformation specification for test cases as shown in Figure 9.11. The upper part depicts the transformation from System specification (PIM in Figure 9.5) to a Component specification (PSM in Figure 9.5) in the MDA process. The Refinement specification represents the design decisions made by the developers during transforming PIM to PSM. The lower part is a new transformation of the high-level test cases (conforming to the PIM) to low-level test cases (conforming to the PSM).

The goal here is to perform the test case transformation as automated as possible, having the Refinement specification available. The instantiation of PSM objects is straightforward. However, there are several non-trivial issues

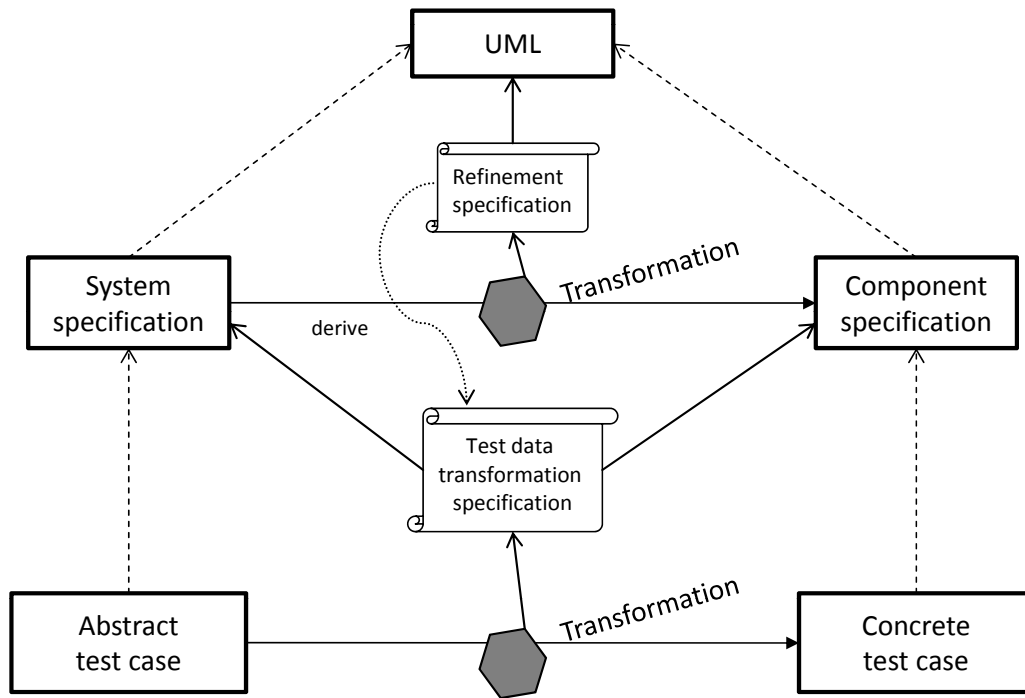


Figure 9.11: Transformation of test cases using the design decisions

concerning attributes: how are the attributes in the PIM related to the ones in the PSM? Is an automatic derivation even possible or is user-interaction required here? After these issues are solved (which needs to be done only once), the transformation can be performed. As a remark, even though the derivation of the test data transformation specification could be performed automatically, a revision of it is strongly recommended to ensure correct test data propagation.

### 9.3 Test Implementation

As we have explained in the last section, two important requirements on model-based specification techniques are 1) understandability by all stakeholders and 2) enabling the automation. Since the model-based specifications are used as a source for the test cases, these requirements also apply to them, i.e. test cases must be understandable by all stakeholders, especially by the end users, and must enable the automation of testing activities. Having the same abstraction as the high-level domain model and Visual contracts, the derived test cases can be assumed to be understandable. However, ab-

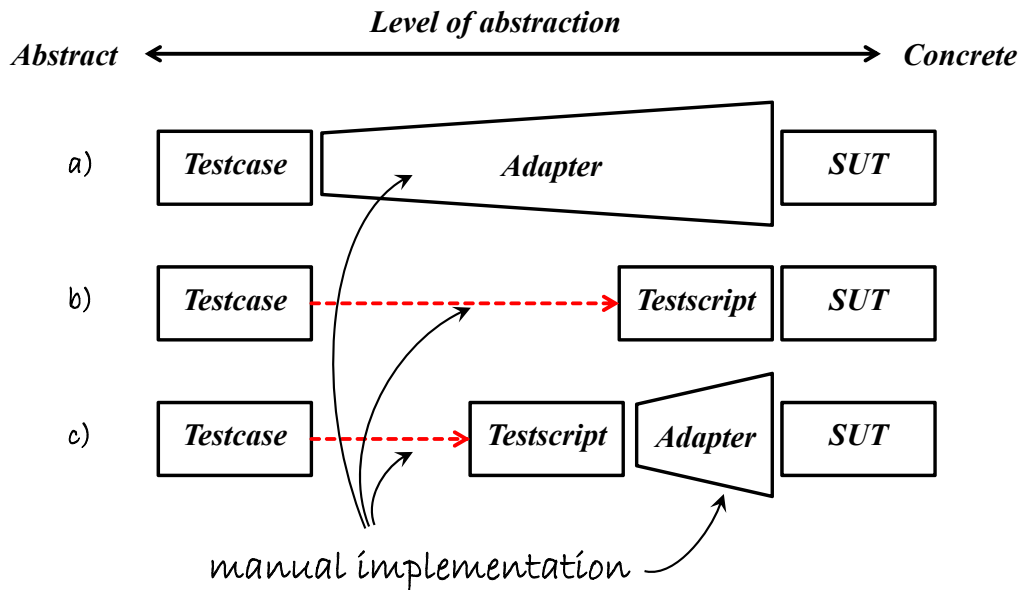


Figure 9.12: Level of abstraction (based on [UL07])

straction means also that the test cases should not involve technical details, which are required for the execution of test cases. In other words, the technical details are relevant and important for invoking system functionalities and observing and validating system behavior. In case of automated test execution with test scripts, the need for technical details arise, so that the test driver can conduct test steps without much manual intervention.

Utting and Legiard describe in [UL07] three scenarios for test automation (see Figure 9.12), how the gap between abstract test cases (*TC*) and test scripts can be filled. If an abstract *TC* is to be executed on a software under test *SUT*, the semantic gap can be filled in the following way:

- a) An *Adapter* is implemented, which can read and understand the elements in an abstract *TC* and directly invoke the *SUT* using the inputs given in *TC*. The *Adapter* is a proprietary software which is specialized for the corresponding *TC*s and the *SUT*.
- b) The *TC*s are transformed manually into test scripts (*TS*) in a standardized scripting language and can be executed with an appropriate test driver. In order to assure the executability of the *TS* by the test driver, the transformation must address all the required elements of the scripting language.

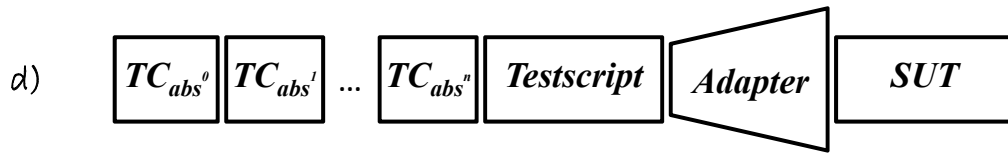


Figure 9.13: Level of abstraction for Visual Contract-based system testing

- c) In order to spread the efforts for transforming the *TS* and the implementation of the *Adapter*, the third scenario follows a mixed approach. Thereby, *TS* make up a more technical view of *TC*, however these are not overcrowded with technical details which are only relevant for the test driver. The technical details are implemented by the *Adapter*.

These three scenarios help in bridging the gap between abstract *TCs* and the *SUT*, however, the transformation steps and the implementation of the *Adapter* cause additional efforts and may lead to new errors, since these are all manual activities. In our approach we aim at reducing the efforts and errors in test automation using abstract *TCs*, while the understandability and automation requirements are still fulfilled. Figure 9.13 shows a fourth scenario we propose where the test process begins with an abstract  $TC_{abs^0}$  which is then transformed stepwise into a more concrete  $TC_{abs^n}$  using the model transformation approach described in the last section until a *Testscript* is created which contains enough technical details. Then, the testscript can be executed by a test driver using an *Adapter*. In our system testing approach, we integrate this fourth scenario described above.

Figure 9.14 shows an example, how an abstract test case can be stepwise transformed into a test script. Thereby, we differentiate between the test steps and the test data. The test steps have to be derived from the specification of Use case steps as shown in Figure 9.15. Also for this activity there are model-based test case generation techniques like [KKBK07, LJX<sup>+</sup>04], which

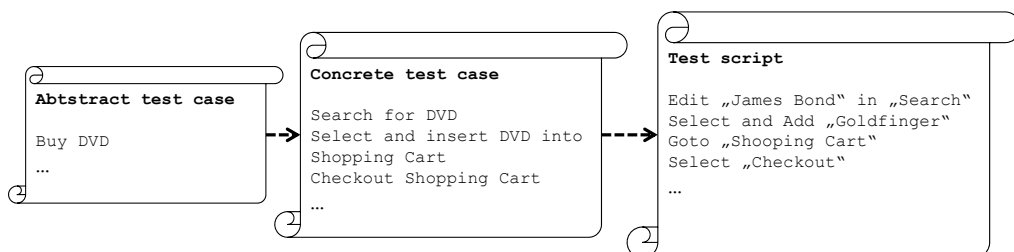


Figure 9.14: Example for different levels of abstraction of test cases

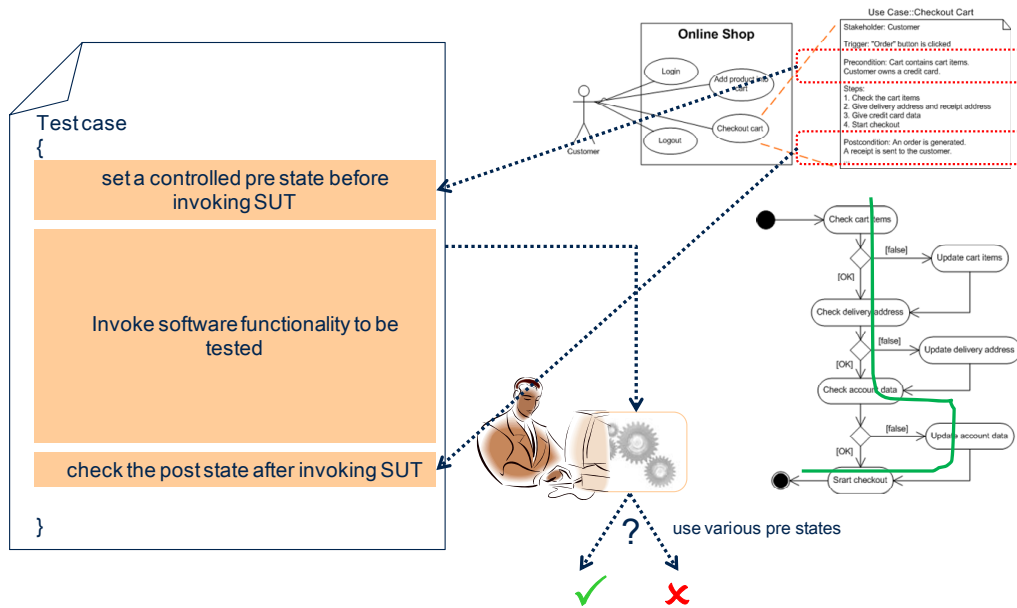


Figure 9.15: Test script for system testing

we not explain in detail. Our focus is on the test data which are required for these steps.

The abstract test case in Figure 9.14 shows a test step for testing the *Buying DVD* functionality at an *Online shop*. If refined further, buying a DVD at an online shop includes the following concrete activities: first *search* for the desired DVD, then *select* and *insert* into the shopping cart, finally *checkout* the shopping cart. The abstract test step is refined into the corresponding concrete test steps. For test automation purposes, a test script is derived from the concrete test case by adding one or more technical test steps for each concrete test step. For example, “Searching for a product” is realized by the test script as “editing the name of the product in the search field”.

For automated execution of the test script, the test driver must map the technical test steps to the application interface of the SUT. Figure 9.16 shows an example for mapping some of the technical steps from the test script in Figure 9.14 to the interface of `HTMLPage` class. Editing a search term `X` and conducting a search is mapped by the adapter to the public operation `search()` after setting the edit field to `X`. Selecting a search result `X` is mapped to setting the selected-Attribute of a checkbox to `true`.

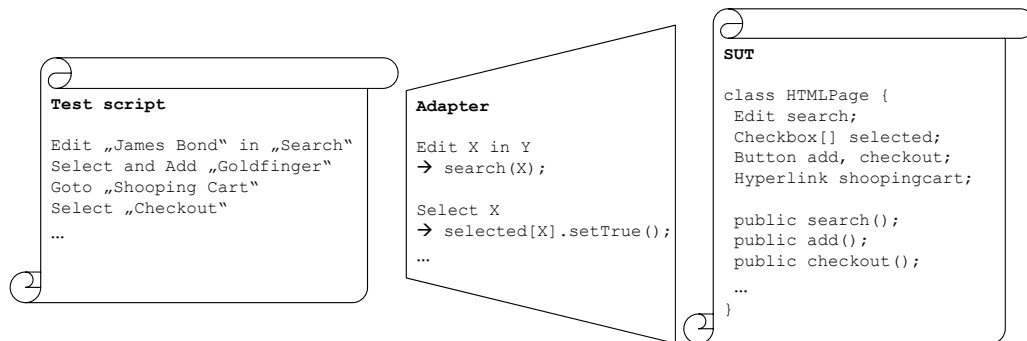


Figure 9.16: Example for the adapter between test script and SUT

## 9.4 Test Execution

Having generated the test cases and transformed them into test scripts, the third step of our test process is to execute them. Figure 9.17 shows the architecture of the test environment for system testing. Thereby, a test driver accesses various interfaces of the SUT: interfaces  $ti_1$  and  $ti_2$  for exchanging the prestate and the expected poststate, and an interface  $f$  for functional invocations.

The test execution realized by the test driver includes the following three steps as shown in Figure 9.18:

- (1) **setup** The prestate is sent to the SUT by the test driver using a dedicated test interface  $ti_1$ . The system under test sets the prestate in order to enable the execution of functionalities under test and sends an acknowledgment back to the test driver.
- (2) **run** After the required prestate is set, the test steps defined in last section are invoked using the test parameters. These steps address end user actions interacting with the external functional interface  $f$  of the SUT (e.g. clicking buttons or editing text fields on the graphical user

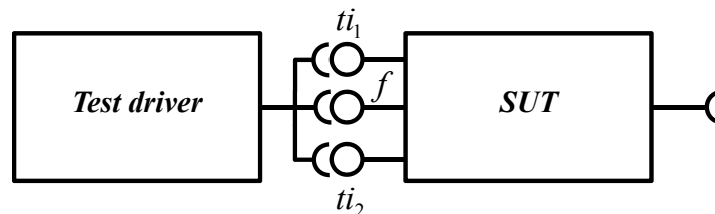


Figure 9.17: Test architecture for system testing



interface). After each test step, the test driver gets an acknowledge that the action is done.

- (3) **evaluate** During step (2) some changes occur in the system state of the SUT which are to be checked for compliance with the postcondition of the Use case specification. The expected poststate is sent to the SUT via a dedicated test interface  $ti_2$ . SUT invokes an assertion which checks the system state after the execution of test steps with the expected poststate. If the actual system state conforms to the expected poststate, it returns normally which sets the test verdict on *pass*. If this is not the case, it returns with an postcondition, which sets the *test verdict* on *erroneous*.

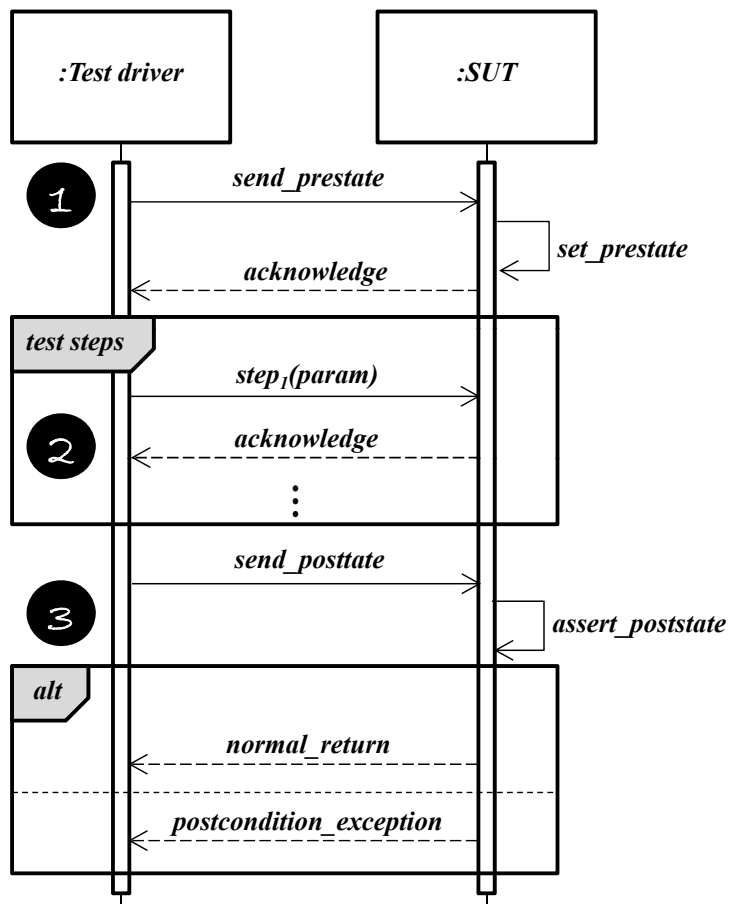


Figure 9.18: Test execution in system testing

# Chapter 10

## Tool Support and Evaluation

The chapters 6-9 have addressed conceptual aspects of the Visual Contract-based Testing approach. In order to show the applicability of the approach, we have developed tool support for various algorithms and execution procedures [Ell08, Kla08, Beu09, Han08]. We have published our results in various conferences and workshops [EEG08, EGL06a, GMWE09]. Furthermore, various case studies are conducted in order to study the effectiveness and the scalability of the approach [SG10]. In this chapter, we will summarize these tools and the evaluation results.

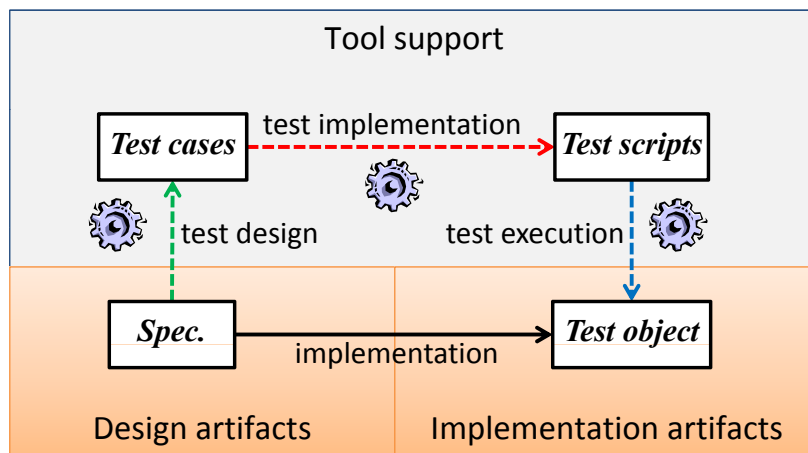


Figure 10.1: Overview of tool support

The tool support covers mainly three activities of our approach as illustrated in Figure 10.1: First test cases are generated from the Visual contracts. Second, test cases are transformed into executable test scripts. Finally, the system under test is executed using the test scripts resulting in a test verdict.

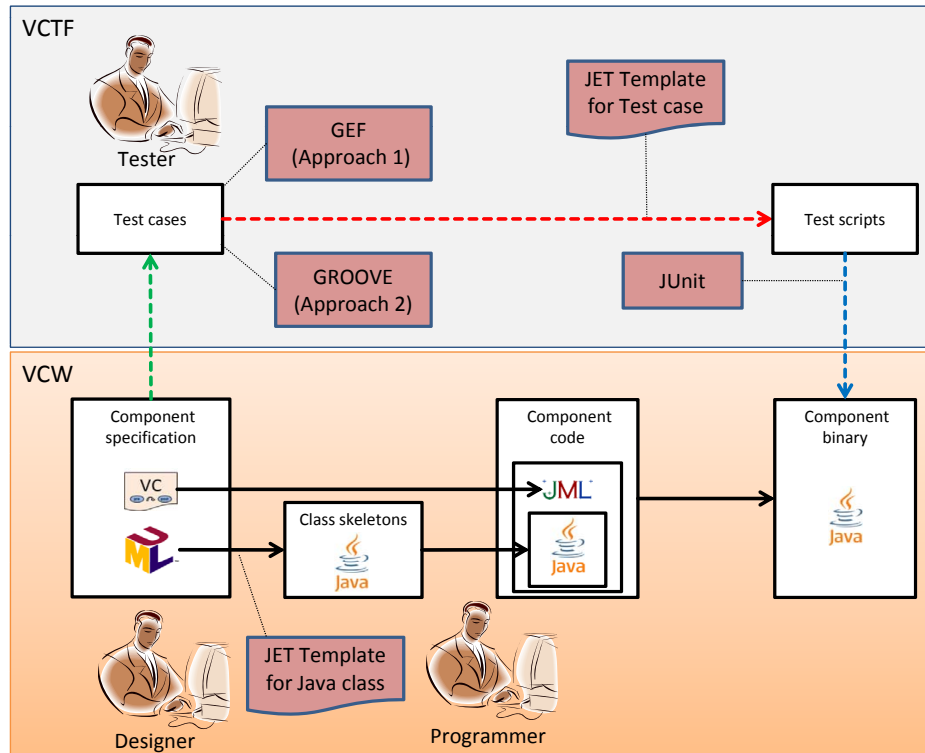


Figure 10.2: Tool support for Unit testing

The contribution of our tool support is that we could show the applicability of various software engineering techniques on contract-based testing, e.g. graph transformations, model checking, model transformations, code generation and model-driven monitoring. Not every test level makes use of each technique, but many techniques we have implemented are transferable to further test levels. Also, not every testing activity is completely automated, some manual interventions are required.

The next sections 10.1-10.3 describe the implementations of some these techniques for various test levels. The section 10.4 reports on the results of some evaluations.

## 10.1 Unit Testing

The Visual Contract-based Unit Testing is characterized in chapter 7 as a low-level testing activity where the tester deals with fine-grained software components like classes and their operations. As a consequence, the tool support for testers is similar to the development environment. Figure 10.2

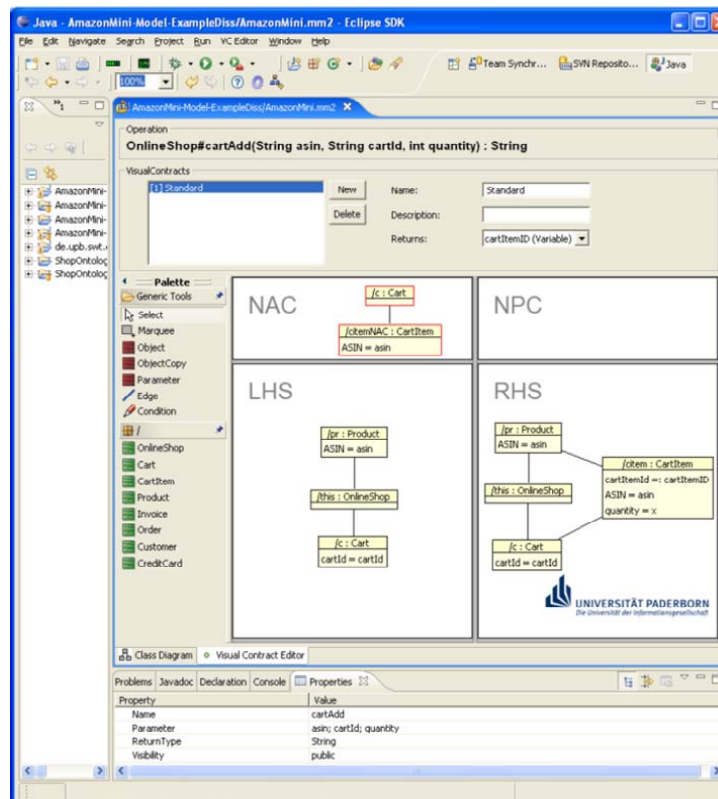


Figure 10.3: Modeling visual contracts with VCW

shows the tools which we have used for implementing the concepts described in chapter Unit Testing.

The basis for our contract-based testing approach is the design of Visual Contracts which is illustrated in the lower box of Figure 10.2. Visual contracts are created using the Visual Contract Workbench (VCW) developed by Lohmann [Loh06]. Also the activities of Java code frame generation, JML assertion generation and the compilation of JML assertions together with the manually extended code are conducted by the VCW-Tool.

As shown in Figure 10.3 VCW allows modeling Visual Contracts by specifying their preconditions (LHS) and postconditions (RHS) and negative application condition (NAC) and negative postconditions (NPC). The objects in Visual Contract are typed over a class diagram, which can also be modeled by VCW (see Figure 10.4). Lohmann has developed this tool based on the Eclipse platform such it can easily be extended using the Plug-In mechanism.

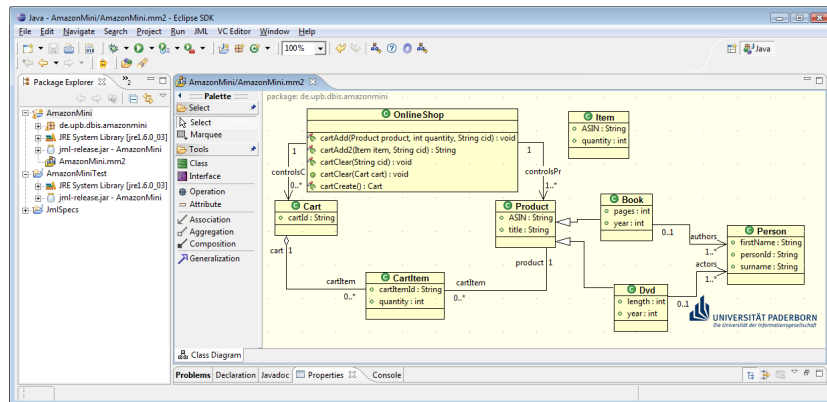


Figure 10.4: Modeling class diagrams with VCW

### 10.1.1 Approach 1: Artificial Prestate

For the purposes of Unit testing with artificial prestates, Jens Ellerweg have extended the VCW by some feature of generating testing artifacts, naming it Visual Contract Test Framework (VCTF) [Ell08] (see upper box in Figure 10.2).

VCTF first generates logical test cases from Visual Contracts. Logical test cases are objects structures and input parameters generated from the precondition of a Visual contract. The objects represent a prestate which is required for the execution of the system under test. Afterwards, executable test scripts are generated from the test cases and they are executed. This procedure is illustrated in the Figure 10.2.

These techniques are implemented in VCTF using a Plug-in mechanism, so that they can easily be modified or extended. Figure 10.5 shows the flexible component architecture of VCTF. The component `de.upb.swt.ggsu.test` is the interface to the VCW of Lohmann [Loh06]. The Visual Contracts created by using the VCW are further processed by the `vctest`-components. The three components at the bottom of Figure 10.5 implement the algorithms for prestate generation and for test input generation as explained in chapter 7. If further selection algorithms should be implemented, new components can be plugged into the `vctest.defaulttestdriver` component. Details to the implementation of these components can be found in the master thesis of Jens Ellerweg [Ell08].

The main technological element in VCTF is Java Emitter Templates (JET) [Con06c]. JET enables programmers to create code templates which are then translated into template java classes. VCW uses JET technology for generating java classes from the source model with embedded assertions

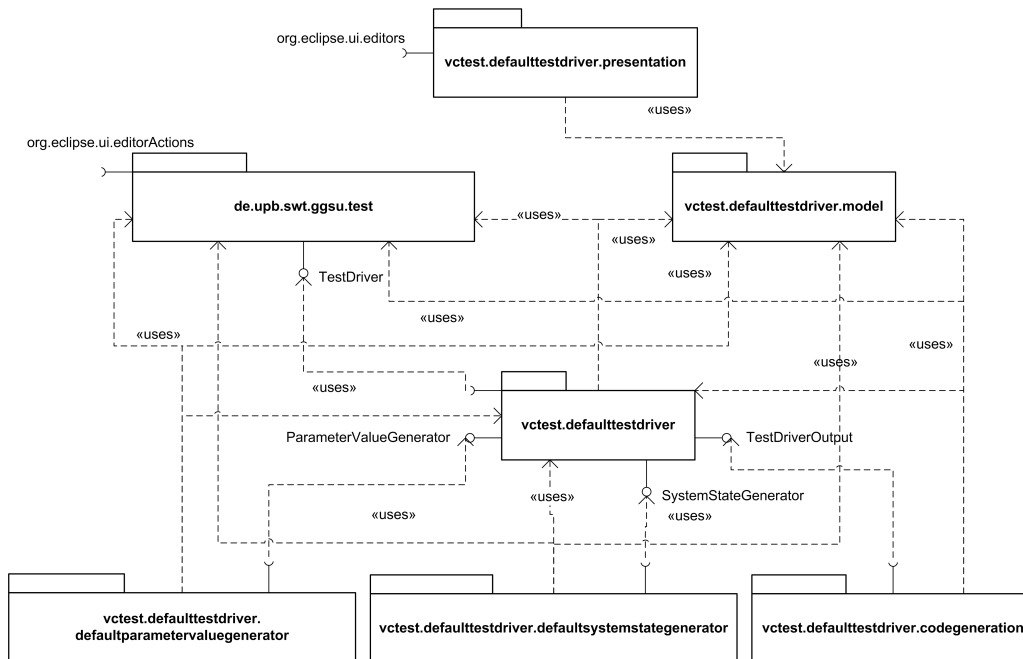


Figure 10.5: Components of VCTF [Ell08]

derived from the pre- and postconditions. Based on this idea, VCTF defines test case templates which are filled with object structures derived from Visual Contracts as logical test cases. These can be viewed and edited by GEF (Graphical editing Framework) editors [Con06b]. The logical test cases are then transformed into executable JUnit test scripts.

Figure 10.6 shows an exemplary test script generated by VCTF on the basis of Visual Contract for `cartAdd` operation of class `OnlineShop`. For the class under test, a new class (`TestOnlineShop`) extending the `TestCase` is generated with test operations for each logical test case (e.g. `testCardAdd_0()`). Each test operation is composed of following four steps as conceptually explained in chapter 7:

- Step 1: For each input parameter and the self-object, a local variable is declared. The variables are initialized with values defined by the selection criteria.
- Step 2: Set up system state by invoking a helper `setUp`-function. As shown in Figure 10.7 this function creates instances for the objects and links given in the precondition of the corresponding Visual Contract. The resulting object state is linked with the `self`-object.

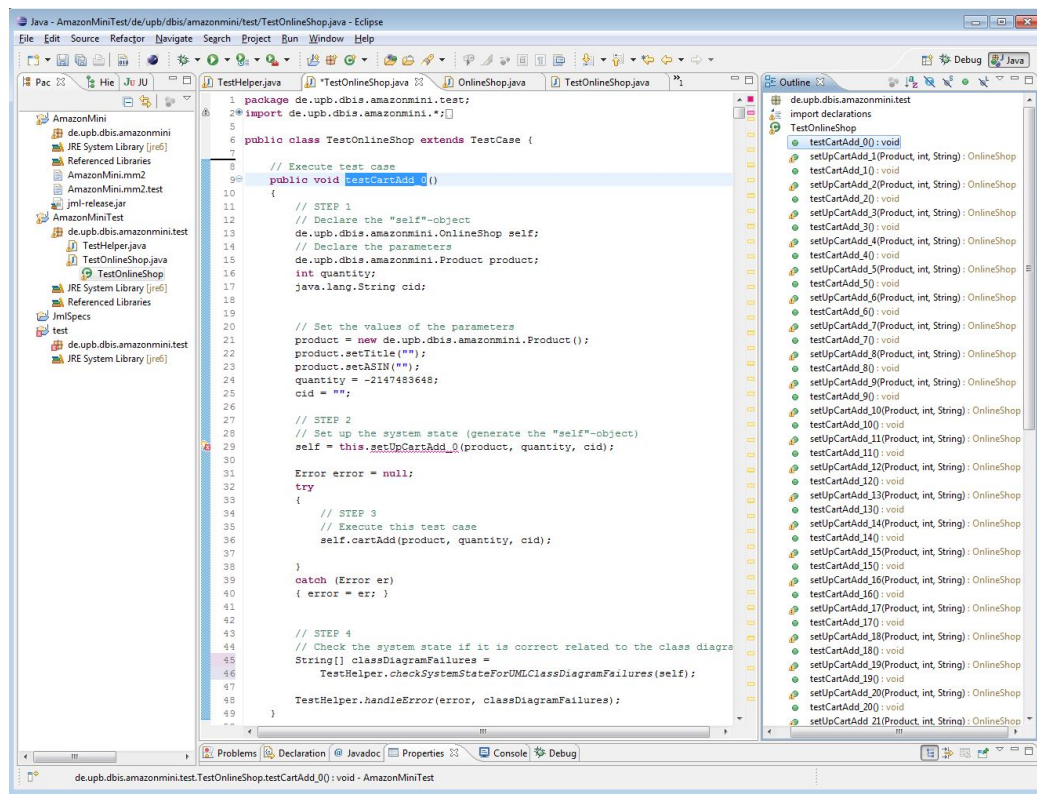


Figure 10.6: JUnit test script generated from Visual Contract for `cartAdd`

- Step 3: Having created a prestate fulfilling the precondition of the Visual Contract, the operation under test can be invoked by the test operation using the input parameters and objects in steps 1 and 2. During the invocation, the JML assertions are automatically checked by the Java runtime environment (JRE) for conformance with the pre- and postconditions of the corresponding Visual Contract. If the system state is generated correctly, the assertion for precondition will pass. Otherwise the JRE returns with an exception. After the assertion for precondition, the operation under test is invoked. The resulting poststate is checked then by the assertions for postcondition. Again here, if the system state after the invocation of operation under test conforms to the postcondition of the Visual Contract, JRE returns with pass. Otherwise an exception will be thrown.
- Step 4: Even if, the assertions for the pre- and postconditions return with pass, i.e if the the system state was conform with the pre- and postconditions of Visual Contract, the resulting system state may not



```

2056 // Test of OnlineShop.cartAdd
2057 // Set up test case
2058 public de.upb.dbis.amazonmini.OnlineShop setUpCartAdd_0(
2059     de.upb.dbis.amazonmini.Product product,
2060     int quantity,
2061     java.lang.String cid)
2062 {
2063     // STEP 2.a
2064     // Declare the variables
2065     de.upb.dbis.amazonmini.OnlineShop self;
2066     java.lang.String producttitle;
2067     java.lang.String productASIN;
2068     de.upb.dbis.amazonmini.Cart c;
2069
2070     // Initialization of the variables
2071     self = new de.upb.dbis.amazonmini.OnlineShop();
2072     producttitle = "";
2073     productASIN = "";
2074     c = new de.upb.dbis.amazonmini.Cart();
2075
2076     // STEP 2.b
2077     // Assignment of the attributes
2078     c.setCartId(cid);
2079
2080     // STEP 2.c
2081     // Set the links between the instances
2082     self.addControlsPr(product);
2083     self.addControlsC(c);
2084
2085
2086     // return the "self"-object (class under test)
2087     return self;
2088 }

```

Figure 10.7: Helper operation for setting up the system state

be conform with the associations specified in the class diagram. This is because the Visual Contracts may incomplete with respect to the class diagram. That is why a final check for conformance with class diagram is implemented in the test operation as the last step.

As shown in Figure 10.1, the third tool supported step is the execution of the test scripts. The test driver of JUnit can started from the context menu of VCTF as shown in Figure 10.8. This leads to the invocation of all test operations in the `Test`-class.

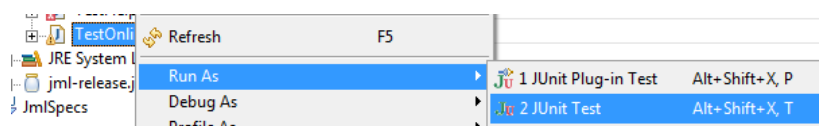


Figure 10.8: Running the test script from the context menu

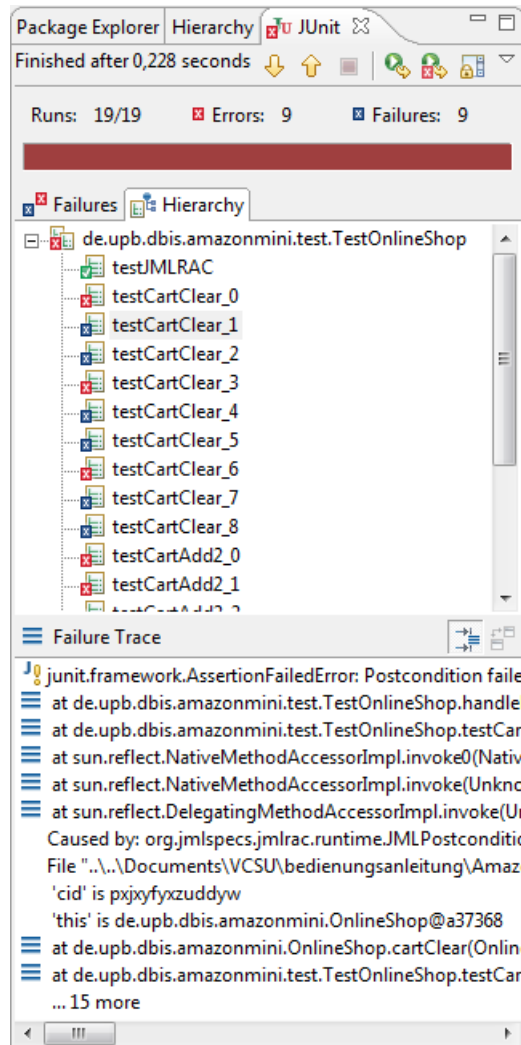


Figure 10.9: JUnit test report indicating the status of test cases

During the test invocation, the test script steps are executed and a test report is generated as shown in Figure 10.9. Depending on the results of JML assertions, the test operations are assigned a test verdict **pass**, **fail** or **error**. Test verdict *pass* means that both the assertions for precondition and postcondition are fulfilled by the operation under test. *Fail* means that either the assertions for the postcondition or the final conformance check with the class associations have failed. *Error* indicates that either the assertions of precondition are not fulfilled or during the execution of the assertions an error has occurred. For determining the actual fault, manual inspections are required. Unfortunately, the VCTF does not supply with detailed informations indicating the exact position of faulty code [Ell08].

### 10.1.2 Approach 2: Natural Prestate

In the last section, we have presented the tool support for generating artificial prestates from Visual contracts and translating them into JUnit test scripts for an automated execution. As described in chapter Unit Testing, a second approach enables the computation of natural prestates for a more realistic testing the state changing behavior of the system under test.

In section Approach 2: Natural Prestate of chapter Unit Testing, we have described how preambles for test cases can be computed which will set the system under test into a required prestate in a natural way. For that, we have defined a mapping between the test concepts and graph transformation concepts (cf. Table 7.2). This mapping is illustrated in Figure 10.10 visually.

The aim in this approach is to interpret the Visual contracts as graph production rules and compute a graph transition system starting from an initial graph. Having computed the graph transition system, analysis on the state transitions becomes possible. Then, we instrument a reachability analysis using model checking techniques for searching for a path of operation invocations which brings the system from the initial state to a required state. This path represents the invocation sequence of class operations.

In order to these computations, we have used the Groove-Tool [GMR<sup>+</sup>12] which is a model checker for graph transformation systems. Hannwacker described in his master thesis [Han08], how the Visual contracts can be translated into graph transformation rules in Groove and how the preamble can be computed using reachability analysis. First step in adapting the Groove tool for our purposes is to translate the Visual contracts into the Groove language.

In this section, we present a small example, where the operation *cartAdd* of a class *OnlineShop* is tested. For computing a preamble for this operation, we need further operation calls for setting the instance of class into

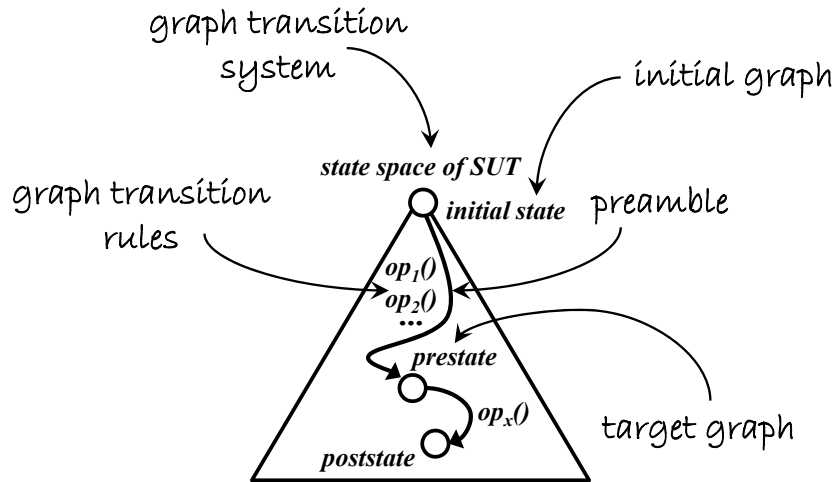


Figure 10.10: Mapping between concepts of Unit testing and Groove-Tool

a required prestate. Figures 10.11 and 10.12 illustrate two translations of Visual contracts for the class operations *cartAdd* and *cartCreate*. We will demonstrate how these contracts can be used for computing a preamble.

Figure 10.11 illustrates the translation of three main concepts in Visual contracts:

- preconditions and postcondition
- negative application conditions
- operation signature with input parameters

The objects and their links are translated to graph objects and links in Groove. Thereby, the precondition and the postcondition of a Visual contract are represented in one graph (right hand side of Figure 10.11). Thereby, the objects and links which remain as unchanged between the pre- and postcondition are colored **grey** (e.g. instances of *Product*, *OnlineShop* and *Cart*). Objects which exist only in the postcondition, i.e. which are created newly, are colored **green** (e.g. instance of *CartItem*, its attribute *quantity* and corresponding links to *Product* and *Cart*). Objects which exist only in the precondition, i.e. which are deleted, are colored **blue** (e.g. objects *cartAdd* and *callToken* and corresponding links). Finally, the objects are colored red, which are part of the negative application condition, i.e. which are not allowed to exist before the execution of the specified operation (e.g. instance of *CartItem* and corresponding links).

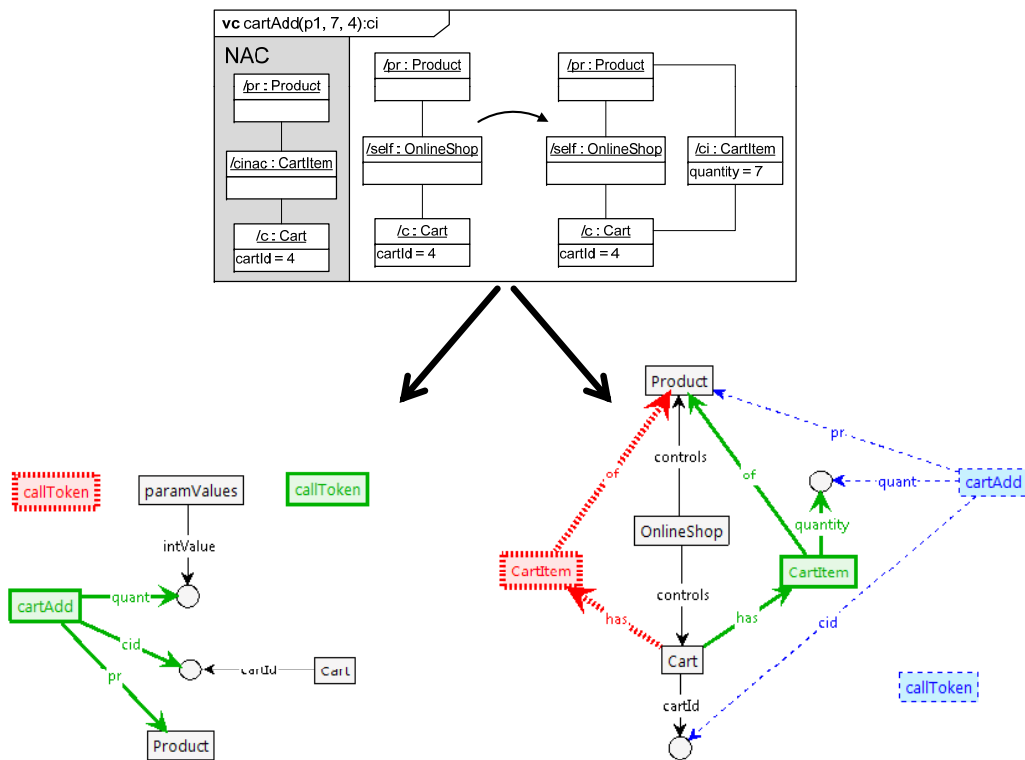


Figure 10.11: Transformation of a Visual Contract into a Graph Transformation Rule for Groove

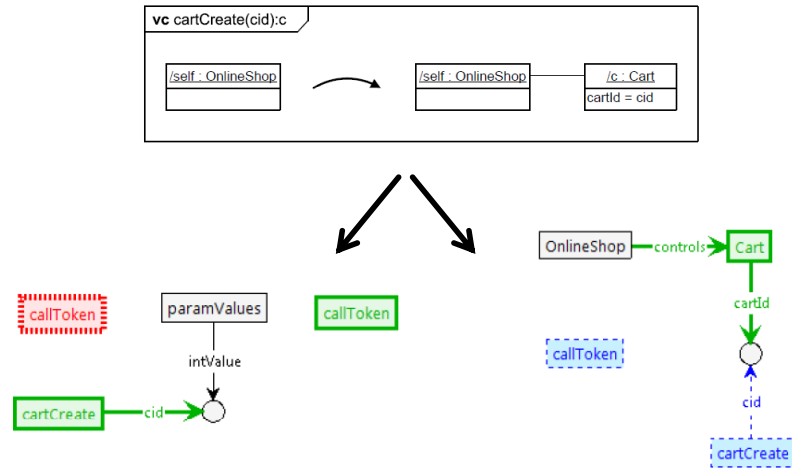


Figure 10.12: Transformation rule 2

A specialty in our translation is the definition of a new type or rule, which we call *invocation rules* (left hand side of Figure 10.11). These rules ensure the application of the graph transformation rule for a Visual contract with a given set of concrete input parameters [Han08]. Before matching during the state space exploration, these rules create *callToken* objects together with the input parameters which are then deleted by the actual rule after its application.

Figure 10.12 shows a second example for a rule translation. Thereby, the instance of class *OnlineShop* is preserved in the postcondition, whereas the instance of class *Cart* is created and bound to the input parameter *cid*. After the required changes in the system state is done, the objects of invocation rule are deleted.

Having translated the Visual contracts of operations into the Groove language, we need a *start graph* for starting the state space exploration. Figure 10.13 illustrates an example of a start graph  $s_0$  in Groove. Since we aim at testing the operation *cartAdd* of class *OnlineShop* in our running example, we require an instance of this class and the input parameters for the operation under test. These input parameters will be matched by the rule for

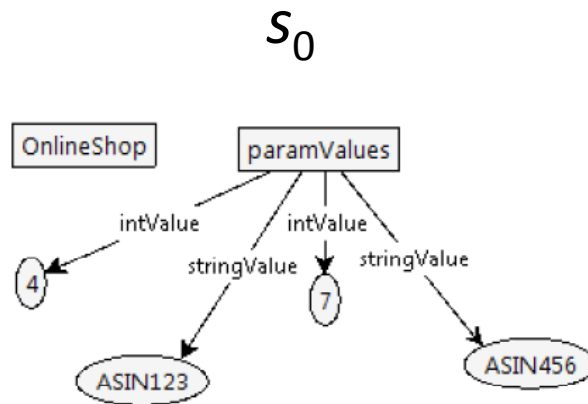


Figure 10.13: Start graph

operation under test, if the state space exploration finds a path from the start graph to the required prestate.

As we have already mentioned, we aim at adapting reachability analysis via model checking for computing preambles for `cartAdd`. As described in section 7.2.2, we use the model checking functionality of the Groove tool. Thereby, we define a *state property*  $\phi$  which represents the required prestate. This prestate will act as a *target graph* for the state space exploration. Thus, we derive a state property from the precondition of the Visual contract of operation under test. In our example, we derive an object structure  $s_{input}$  from the precondition of `cartAdd` and extend it with input parameters defined in the start graph. As we explained in section 7.2.2, we formulate the CTL formula  $\mathbf{AG} \neg s_{input}$  for triggering the counter example finding feature of the model checker.

Given a start graph and a set of graph production rules, Groove tool explores the reachable states and creates a state space [KR06]). Figure 10.15 visualizes a typical state space computed by Groove (example from [KR06]). Since our example is small, our state space exploration ends up with a smaller one as illustrated in Figure 10.16. Starting from the  $s_0$  as shown in Figure 10.13, the graph transition rules for operations `cartCreate` and `cartClear` have resulted in the given state space and the a counter example which is shown with the thick framed black vertices.

In order to the integrate this functionality of Groove tool in our VCTF, Hannwacker has defined in [Han08] how the extension mechanism of VCTW can be used. Thereby, a new *test driver*, which has access on the functionalities of Groove, can replace extend the abstract test driver (see Figure 10.17).

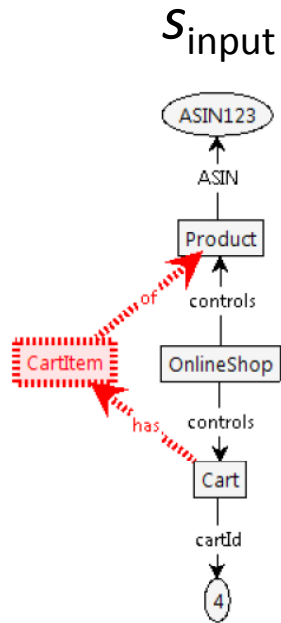


Figure 10.14: Target graph

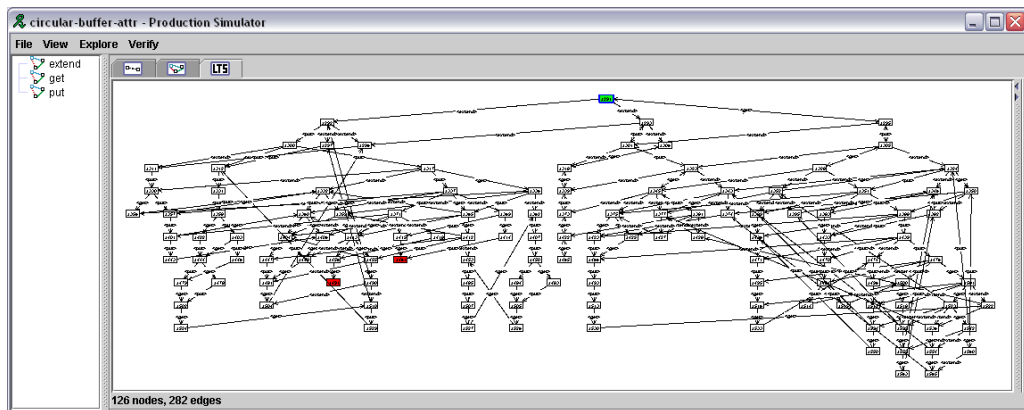


Figure 10.15: State space visualitaion



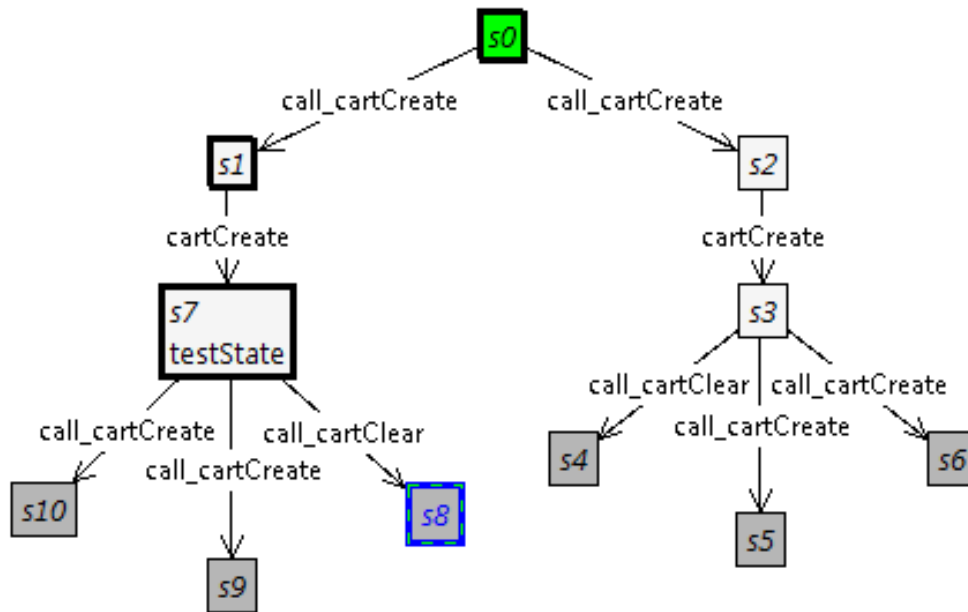


Figure 10.16: State space for Online Shop

The user interface of VCTF enables the selection of the new test drivers as shown in Figure 10.18.

## 10.2 Integration Testing

Since the integration testing is very similar to unit testing, there are no dedicated tool support for this level. The test cases for integration can be generated using the same tooling as for Unit testing. However, there are two specialities in integration testing, which we presented in chapter Integration Testing: (1) topology sorting and (2) configurable monitoring levels.

Since topology sorting is a well known traditional testing in integration testing, there are many tools which can compute the hierarchical dependencies. We have also implemented a small Java program which computes the top-down or bottom-up integration orders of components given a connected graph of subsystems. A challenge in computing the topology sorting are circular dependencies [WEMS<sup>+</sup>12]. To keep it simple, we assume that we have not such kind of dependencies.

Also some UML modeling tools have features for computing the topol-

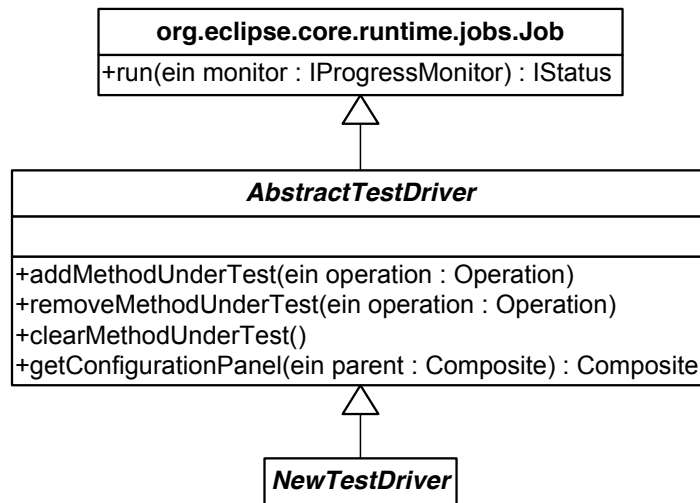


Figure 10.17: Extension of Plug-Ins for new test driver

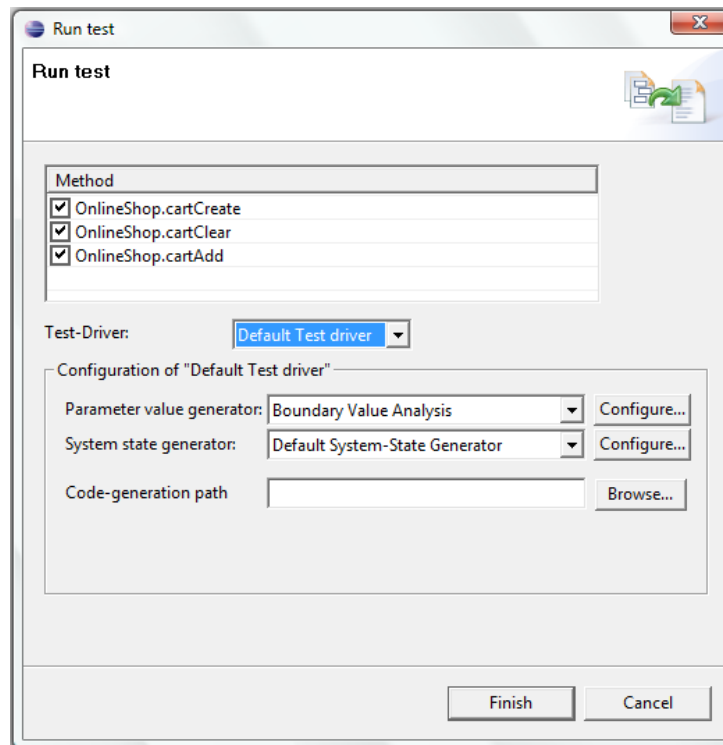


Figure 10.18: Selection of Plug-Ins

ogy sorting. For example, the Enterprise Architect of Sparx Systems<sup>1</sup> can compute and visualize the hierarchical dependencies of components.

## 10.3 System Testing

As described in chapter System Testing, the system testing process contains the following steps:

- Formalizing the preconditions and postconditions of Uses cases using visual contracts which are used as test models.
- Documenting and formalizing the design decisions of developers during transforming PIM to PSM by using model transformations.
- Generating logical test cases from test models.
- Transforming the logical test cases to executable test scripts by using model transformations derived from developers' activities.
- Executing the test scripts using Adapters and automatically evaluating test results.

For automating these steps, we aim at using various tools for particular steps. Figure 10.19 shows the development and testing paths in the lower and upper boxed correspondingly. While the VCW by Lohmann can be used for modeling the Visual contracts and implementing the code, the VCTF requires many extensions for automating the testing activities.

For documenting and reusing the Refinement specifications, there are many model-to-model transformation techniques available, both in commercial and non-commercial tools and environments. We require rule- or mapping-based transformation engines, which almost applies to any engine. Hence, we focus on freely available tools, namely the QVT implementations in the Eclipse framework and an implementation of TGGs [KS06] based on the Eclipse Modeling Framework; still, other technique would work as well, e.g. the Fujaba tool<sup>2</sup>, IBM Rational Software Architect<sup>3</sup>, openArchitectureWare<sup>4</sup>, or mediniQVT<sup>5</sup>. For showing the applicability of the approach, we have created some ATL rules as described in chapter System Testing. We have also created some higher order transformation using the Henshin tool<sup>6</sup>.

---

<sup>1</sup><http://www.sparxsystems.de/uml/neweditions/>

<sup>2</sup><http://www.fujaba.de/>

<sup>3</sup><http://www-03.ibm.com/software/products/de/ratisoftarch>

<sup>4</sup><http://www.openarchitectureware.org/>

<sup>5</sup><http://projects.ikv.de/qvt>

<sup>6</sup><https://www.eclipse.org/henshin/>

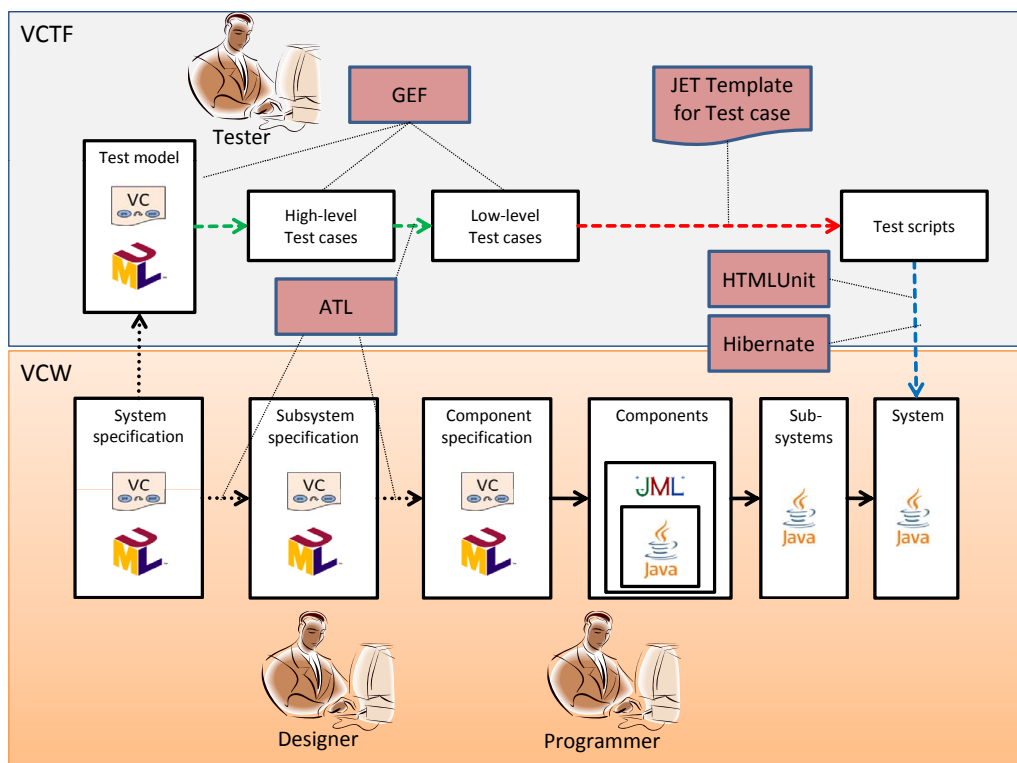


Figure 10.19: Technology overview of the VCW extension for system testing

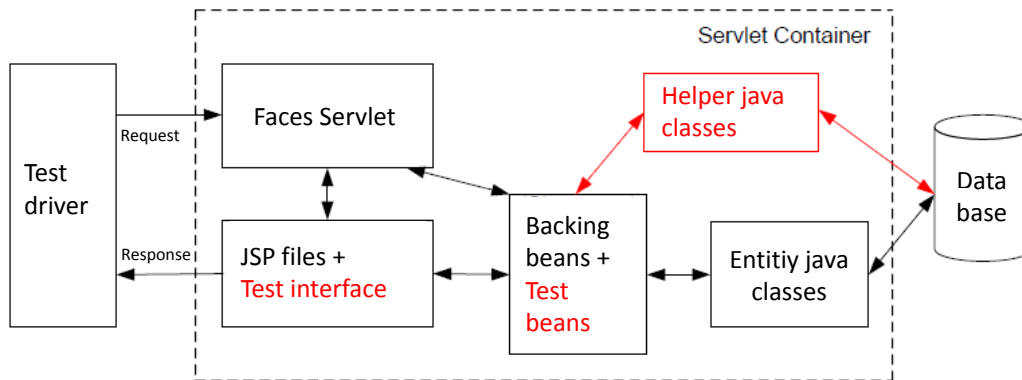


Figure 10.20: Components of VCTF for system testing ([Kla08])

Having transformed abstract test cases into test scripts, we require test interfaces and functional interfaces at the system under test for setting the prestates for Use cases and for triggering a check procedure for the poststate. Klaholt have implemented in his thesis [Kla08] a prototypical Online shop fulfilling the requirements defined in chapter System Testing. As illustrated in Figure 10.20, the Online shop has EJB architecture where entity classes and functional beans are connected to a database and to user interfaces. For setting and checking the instances of entity classes, which comprise the system state, dedicated test interfaces and test beans are implemented.

Figure 10.21 shows a HTMLUnit<sup>7</sup> test script which is composed of three part:

- **set pre-state:** This step accesses the test interface *setTest.jsp* and uploads the prestate and the expected poststate which are stored in XML files *preState.xml* and *postState.xml* respectively.
- **execute test script:** This part contains functional calls to the Online shop as specified in the Use case description. If the input parameters are relevant to the precondition, then objects and variable values from the *preState.xml* can be used here.
- **check poststate:** If the functional steps are complete, the test driver invokes the evaluation step for the poststate.

At the end of the test script, a test report is generated documenting the *test verdict* and an explanation if test case has failed.

<sup>7</sup><http://htmlunit.sourceforge.net/>

```

1   ...
2   final WebClient webClient = new WebClient();
3
4   // set pre-state
5   HtmlPage page = (HtmlPage) webClient.getPage("...setTest.jsp");
6   HtmlForm form = page.getFormByName("upload_form");
7   HtmlFileInput testfile = (HtmlFileInput) form.getInputByName("
      upload_form:testFile");
8   HtmlFileInput evalfile = (HtmlFileInput) form.getInputByName("
      upload_form:evalFile");
9   testfile.setValueAttribute("...preState.xml");
10  evalfile.setValueAttribute("...postState.xml");
11  HtmlInput input = form.getInputByName("upload_form:submit");
12  input.click();
13
14  /*****
15   * execute test script
16   */
17  page = (HtmlPage) webClient.getPage("...shoppingCart.jsp");
18  form = page.getFormByName("pageform");
19  input = form.getInputByName("pageform:submit");
20
21  page = (HtmlPage) input.click();
22  form = page.getFormByName("pageform");
23  input = form.getInputByName("pageform:next");
24
25  page = (HtmlPage) input.click();
26  ...
27  /*****/
28
29
30  // check post-state
31  page = (HtmlPage) webClient.getPage("...evaluateTest.jsp");
32
33  // print test protocol to the console
34
35  HtmlDivision resultDiv = (HtmlDivision) page.getElementById("
      result");
36  String result = resultDiv.getTextContent();
37  System.out.print(result);

```

Figure 10.21: HtmlUnit test script generated from Visual Contract for use case Checkout Cart

## 10.4 Evaluation

In order to assess the the applicability, the adequacy and the reliability of scientific methods, the following evaluation methods can be applied as explained in [LG87]:

- ***peer review***: Experts in a research field read and evaluate the documented results of a research and give feedback about the flaws and strengths of the research results and suggest improvements. Typically, submissions to scientific conferences, workshops and journals are reviewed by at least 2-3 experts.
- ***interviews and questionnaires***: While peer review aims at getting detailed feedback from few experts, interviews and questionnaires aim at involving a bigger set of experts in evaluating research results. Thereby, standardized questions are prepared for making answers of participants short, comparable and countable.
- ***quantitative methods***: The quantitative methods aim at evaluating research results systematically by measurements specially to prove the effectiveness of the developed techniques. Mostly cost-benefit analysis is done to show the efforts needed for applying a technique and the advantages gained by applying the technique.
- ***case studies***: Case studies are empirical investigations of the quality of research results in realistic context by applying them on realistic problems, e.g. in the industry. The applicability and the usefulness of the research results from the industry point of view suffice to evaluate them to be adequate and reliable.

We have evaluated our approach by applying all four evaluation methods. We have submitted our research results in many international and national conferences and workshops, where they are peer reviewed. Together industrial partners we have conducted interviews and case studies. Within the scope of student works, we have evaluated the effectiveness of our approach using quantitative methods.

### 10.4.1 Evaluation by interviews and case studies

Our evaluation together with the industrial partner had two parts: First, we interviewed the partner for assessing the principal applicability of Visual Contracts for modeling component interfaces. Second, we have applied the

modeling technique with Visual Contracts in a case study to evaluate practical applicability of the modeling technique by software engineers from the industry [EGL<sup>+</sup>06b].

In the first part of the industrial evaluation, we have evaluated the principal abilities of Visual Contracts and their applicability by software engineers from the industry, who did not have any experience with Visual Contracts.

The industrial project together with Capgemini sd&m aimed at evaluating the usability of Visual Contracts by software engineers from the industry, who do not have any former experience with this modeling language. Even if the context in this project was modeling of web services in the context of a service oriented architecture (SOA), the results can be transferred to the domain of software testing. Also here, software testers have to deal with a new language and a new modeling paradigm. The project has shown that, Visual Contracts and the pre/post-based modeling paradigm can be easily learned by software engineers. The colleagues from sd&m could create a domain model for insurance domain and many Visual Contracts in short time. However, the project has also shown that, for a more detailed modeling, further language constructs are required. From these experiences, I infer that software testers also would learn Visual Contracts easily and use them especially for creating separate models for system testing purposes.

### 10.4.2 Evaluation by quantitative methods

In order to experiment with our approach, to evaluate the applicability and to make some measurements on the effectiveness of our approach, we have conducted student works. Bachelor of science and master of science students at the University of Paderborn have implemented the algorithms explained in this thesis, conducted experiments, made measurements and documented their results. Following four theses are subject of the evaluation by quantitative methods:

- Master thesis by Ellerweg, J.: Komponententest mit visuellen Kontrakten, University of Paderborn, 2008
- Bachelor of science thesis by Klaholt, D.: Einsatz visueller Kontrakte für modellbasierten Systemtest am Beispiel einer Web-Anwendung, University of Paderborn, 2008
- Master thesis by Hannwacker, D.: Kontraktbasierte Generierung von Methodensequenzen für Testfile mittels Model-Checking, University of Paderborn, 2008



- Bachelor of science thesis by Beulen, D.: Evaluierung eines Ansatzes zum kontraktbasierten Komponententest fr eine datenintensive Java-Anwendung, University of Paderborn, 2009

The student works aimed at assessing the realizability of the described algorithms and at finding the fault detecting capability of our algorithms. Thereby, we have identified that our approach is capable to detect typical state-based errors [Ell08, EEG08, Beu09], e.g.

- initialization failures
- missing links, missing objects
- reachability errors

In detail, we have identified nine types of state-based errors which are listed in the table 10.1. This table shows the fault-detecting capability of different test data generation techniques, i.e. boundary value testing (BVT) and random testing (RT).

Table 10.1: Fault detecting capability of applied techniques

Nr.	Fault description	BVT	RT
1	Generated object with faulty variables	+	+
2	Generated object with faulty links	+	+
3	Missing object to be generated	+	+
4	Changed object with faulty links	+	+
5	Object not deleted	+	+
6	Not-changeable Object does not exist	+	+
7	Inconformance with cardinality	+	+
8	Faulty decision node	+	-
9	Object not generated	-	-

Furthermore, we have measured some performance attributes of thge implemented algorithms for boundary value testing and for random testing. The results are shown in table 10.2

Table 10.2: Time spent by generator

Criteria	BVT	RT
Number of generated test cases	134	34
Time for test case generation	90 ms	50 ms
Time for coxode generation	500 ms	500 ms
Time for test execution	200 ms	100 ms

We have conducted a further quantitative evaluation together with Matthias Schnelte [SG10] in order to assess the performance and scalability properties of our approach. The question of interest was which memory consumption do the state space exploration techniques by Visual Contracts require in comparison to the planning algorithms. The results are shown in 10.3.

Table 10.3: Memory consumption

Number of objects	Preamble length	States generated (LAMA/GROOVE)
4	4	14/10
7	5	27/152
10	6	44/3248
13	7	65/20000
16	8	90/n.a.

### 10.4.3 Evaluation by peer reviews

The research results which are the subject of this evaluation chapter are submitted to the international and national conferences to get feedback from the scientific community. Here is a chronological list of the peer reviewed and accepted papers which emerged in the context of our research:

- Engels, G.; Güldali, B., Lohmann, M. Hearnden, D.; S, J.; Rapin, N., Baudry, B. (ed.) Towards Model-Driven Unit Testing Proceedings of the workshop on Model Design and Validation (MoDeVa 2006), Toulouse (France), Le Commissariat l’Energie Atomique - CEA, 2006, 16-29

- Gregor Engels, Baris Güldali, Marc Lohmann, Oliver Juwig, and Jan-Peter Richter. Industrielle fallstudie: Einsatz visueller kontrakte in serviceorientierten architekturen. In Bettina Biel, Matthias Book, and Volker Gruhn, editors, Software Engineering, volume 79 of LNI, pages 111-122. GI, 2006
- Ellerweg, J.; Engels, G., Güldali, B. Hegering, H.-G.; Lehmann, A.; Ohlbach, H. J., Scheideler, C. (ed.) Modellbasierter Komponententest mit visuellen Kontrakten INFORMATIK 2008, Beherrschbare Systeme - dank Informatik, Band 1, Beitrge der 38. Jahrestagung der Gesellschaft fr Informatik e.V. (GI), Gesellschaft fr Informatik (GI), 2008, 133, 211-214
- Güldali, B.; Mlynarski, M.; Wübbecke, A., Engels, G. Model-Based System Testing Using Visual Contracts EUROMICRO-SEAA, 2009, 121-124
- B. Güldali, M. Mlynarski, Y. Sancar. Effort Comparison for Model-based Testing Scenarios Proc. of Quombat Workshop at ICST, 2010
- Schnelte, M., Güldali, B. Test Case Generation for Visual Contracts Using AI Planning INFORMATIK 2010, Beitrge der 40. Jahrestagung der Gesellschaft fr Informatik e.V. (GI), Gesellschaft fr Informatik (GI), 2010, 176, 369-374
- Mlynarski, M.; Güldali, B.; Weileder, S., Engels, G. Model-Based Testing: Achievements and Future Challenges Advances in Computers, 2012, 86, 1-39



# Part III

## Closure



# Chapter 11

## Conclusion and Future Work

In this thesis, we have proposed the Visual Contract-based Testing approach, which uses Visual Contracts as test basis and derives test artifacts from them using automated techniques.

As shown in Figure 11.1, we assume that Visual Contracts are used by designers as an extension to UML for specifying system behavior at different development phases, i.e. beginning from high level system specification ending with implementation of executable components. Each implementation and integration activity of programmers is followed by a testing activity, where the components are tested against Visual Contracts specifying the behavior of components. Then, components are integrated resulting in subsystems. The subsystem operations are tested against the Visual Contracts specifying the subsystem interfaces and the component interactions. Finally, subsystems are integrated into complete system, which is tested against the high level system specifications describing the use cases. We have showed in this thesis, how testers can reuse the Visual Contracts of designers for low-level test levels, or create their own test models at higher test levels.

We have used various technologies in order to implement the algorithms for test case generation and for test execution. Thereby we have used the model checking tool Groove for computing operation sequences setting artificial prestates, the model transformation tool ATL for transforming high level test cases into implementation level test cases, and JUnit for executing test cases.

In this concluding chapter, we address our contributions, give a personal comment and show future directions for further research.

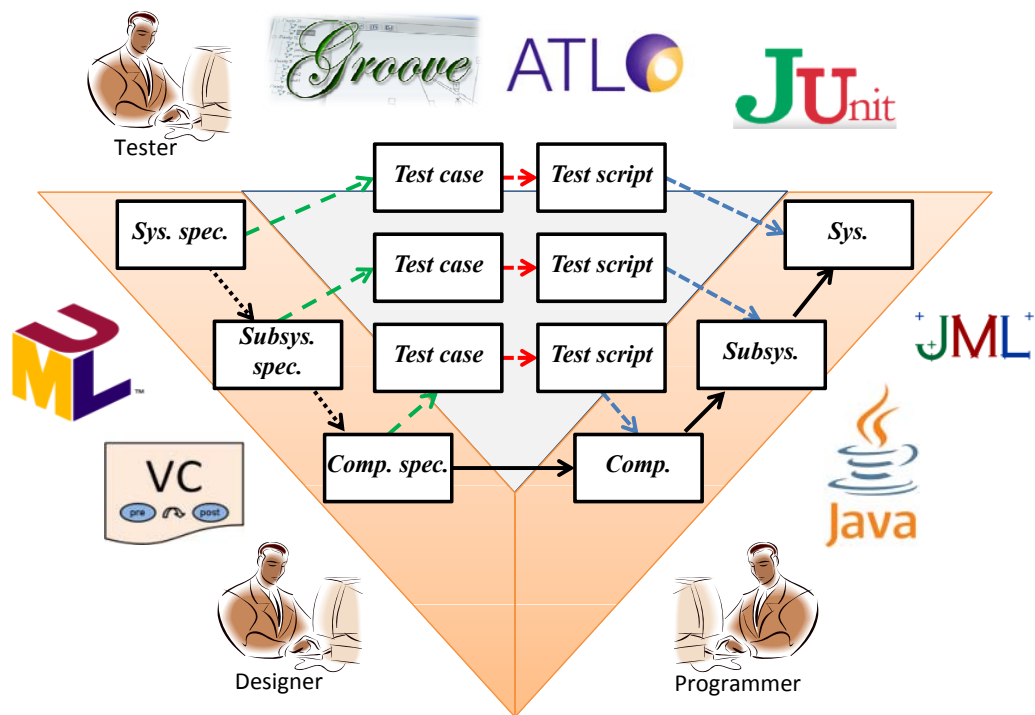


Figure 11.1: Process overview of Visual Contract-based Testing



	AutoTest [Ciu08]	Korat [MSM <sup>+</sup> 07]	WeSUF [AW05]	MBTVC [Khan2012]	VCBT
<b>General characteristics</b>					
People	Low level	Low level	Low level	Low level	High level
Test level	UT	UT, ST	UT	UT,IT	UT, IT, ST
Process integration	No	No	No	No	Yes
Quality	Industrial c.s.	Industrial c.s.	Small c.s.	WB	Small c.s.
<b>MBT-specific characteristics</b>					
Artifact	Eiffel	JML	OC	VC	UML
Redundancy	Separate	Separate	Separate	Shared	Sep./shared
Automation	Yes	Yes	Yes	Yes (Oracle)	Yes
Tool	Yes	Yes	Yes	Yes (Oracle)	Yes
<b>CBT-specific characteristics</b>					
Contract	Declarative	Declarative	Declarative	Decl./Imp.	Decl./Oper.
Test case	Artificial	Artificial	Artificial		Artif./Nat.

Figure 11.2: Comparison of the contract-based testing approaches

## 11.1 Contributions

The VCBT approach, as summarized above, lead to improvements compared with other CBT approaches (see the Figure 11.2) and fulfills the requirements defined in section 5.2 in the following way.

### Relation to the development process

Aiming at the integration of contract-based testing into the model-driven process, the VCBT approach seamlessly integrates into to the UML-based development process, fulfilling the requirements 2 and 3 (cf. section 5.2)

**Contribution to Requirement 2:** VCBT approach is compatible with the UML-based software development process, because it uses or extends the concepts and languages of the UML-based process. We have shown, how concepts of design-by-contract (section 3.1) are related to the concepts of UML for behavioral and structural modeling (section 2.2) and to the concepts of model-based testing (section 2.4). The pre- and postconditions of a Visual Contract constrains the states of the software under test before and after the execution of the SUT. By doing that, Visual Contracts both specify structures using the pre- and postconditions and also behavior by defining a transition relation between the pre- and postconditions.

For specifying pre- and postconditions of a Visual Contracts, we have used UML collaboration diagrams which are typed over UML class diagrams (cf. chapter 4). The collaboration diagrams are grouped into four compartments including two positive application conditions for the pre- and postconditions

particularly, one negative application condition and one negative postcondition.

Using these concepts and languages, the VCBT approach defines how the Visual-Contract based software development process defined by Lohmann [Loh06] can be extended by testing activities (section 6). In order to do that, this thesis presents algorithms and tools for deriving test artifacts from the Visual Contracts and conducting test execution and evaluation for different test levels (see chapters 7-10).

**Contribution to Requirement 3:** Both MBT scenarios shared models and separate models (cf. section 2.4.2) are supported by VCBT approach. The difference between these scenarios lies in the fault detecting capability of the testing approach with respect to the initial requirements. If testers reuse the developers model as test basis, they will test the software under test against the developers models, however, they will not be able to find any errors related to the initial requirements.

The VCBT approach extends the artifacts of UML-based development process, such that Visual Contracts can either be a part of the specification artifacts of the system under development as shown in chapters 7 or 8, or they can be separately created by testers as explicit test models, as shown in chapter 9. Unit testing and integration testing against the Visual Contracts by developers can detect errors where the callee operations do not fulfill the postconditions after being executed or the callers functions do not fulfill the preconditions for executing a callee operation. These errors may result either from the manual coding activities where the Visual Contracts are not transformed to code correctly or from wrong specifications of the Visual Contracts during the refinement of design artifacts. In each case, testers and developers have to look for the reasons for the errors during the test evaluation phase.

The separate models scenario for system testing shows that testers can go a totally independent way from the developers, where they derive the test basis directly from the initial system requirements. In this way, testers are also able to find unconformities between the Visual Contracts for system design and the initial system requirements.

### Test levels and Quality of Test Cases

VCBT approach fulfills the requirement 4 in section 5.2 that it should support different test levels in the development process as follows.

**Contribution to Requirement 4 and 6:** Model-based software development process contains various refinement steps where software under development is specified on different abstraction and detail level. These

specification levels also define the test levels, where test cases have similar abstraction as the test basis. Considering the typical three refinement steps in software design, i.e. system specification, subsystem specification and component specification (cf. section 2.2), we defined the three test levels *system testing*, *integration testing* and *unit testing* (chapters 7-9).

Unit testing (chapter 7) aims at checking the conformance between the the implemented class operations with their specifications by Visual Contracts. The test target here is to validate that the class operations fulfill the postconditions if they are invoked with test cases which fulfill their preconditions. The test cases are object constellations as defined by Winter [Win99] which we derive from the preconditions by applying well-known selection criteria on object variables and on multi-objects. We have developed an Eclipse-based test generator implementing the test case generation algorithms and transforming the test cases into executable test scripts [Ell08]. The execution of the test scripts is done by the JUnit tool. The fulfillment of the pre- and postconditions are checked at runtime automatically by embedded JML assertions [Loh06]. Visual Contract-based Unit testing can find errors due to missing objects, missing or wrong object relations, or wrong object variables.

The object constellations used as test cases for unit testing are generated directly from the precondition of a Visual Contract specifying a single operation. Thus these object constellations are generated in isolation from the real execution context of the operation under test. In a realistic environment, the class operations should be tested in context with other class operations where the object constellations are created naturally in interaction with other operations. For that reason, we have proposed a second approach for unit testing 8 to setup a more natural test environment for testing the class interactions. For a natural test environment, the object constellations required for testing a class operation is set by a sequence of other class operations. Thereby we compute operation sequences based on the contract specifications of the class operations and model checking techniques (chapter 7). Here, given a start state and a set of Visual Contracts, a state transition system is computed by the model checker Groove [Ren03]. Reachability analysis is conducted in order to compute a path from the *start state* to the *prestate* of the operation under test. If a path can be found on the specification, during the test execution, the corresponding invocation sequence of operations should 1) be executable in the computed order and 2) lead to a prestate which fulfills the precondition of the operation under test. If this is not the case, either the Visual Contracts are not complete missing some required objects, or the class operations are not correctly implemented due to the Visual Contracts.

If the low-level components are tested during unit testing, these are inte-

grated to subsystems. The task of integration testing is to define an integration strategy and to test the components in interaction. For computing the order in which the components are to be integrated and tested, we proposed to use topology sorting. Due to the chosen strategy top-down or bottom-up integration, the test environment requires test drivers and stubs in order to simulate the caller and callee components which are either not implemented yet or which are not tested yet. We have proposed for these strategies the usage of Visual contracts as a medium for monitoring the data interchange between components. By activating or deactivating embedded assertions, which are generated from Visual Contracts, we can define various monitoring levels. The higher the monitoring level, the more insight the testers have into the inner life of the integrated system.

Both unit testing and integration testing dealt with testing low-level software artifacts, like components and subsystems which are collections of components. These are tested against Visual Contracts which are created by developers as an implementation specification. These test activities can verify the conformance of implemented code with their specification, however in order to validate the fulfillment of initial requirements, the system specifications from earlier development phases must be used as test basis. This is exactly the target of Visual Contract-based system testing (cf. chapter 9), where the integrated system is tested against customer requirements. In UML-based software development process, customer requirements are documented as Use Case models where required functionalities are described by using preconditions, operation steps and postconditions [Coc01]. Testers can use these information to specify system test cases. In our approach, we have shown how pre- and postconditions of Use Cases can be formalized by using Visual Contracts. Similar to test case generation techniques for unit testing and integration testing, pre- and postconditions of Visual Contracts are used for generation of object constellations. However, the objects created on the basis of Use Cases must be translated into the detail level of the implemented software. For that, we reuse the formalized design decisions [Kön10] as model transformations rules for transforming high level test objects into implementation level test objects. Our tool support includes the ATL model transformation [Con06a] technology for transforming and serializing high level test objects into implementation level objects.

### **Skills of testers and the selection of the modeling techniques**

One of the reasons, why model-based software development does not gain acceptance by industry, is the usage of too formal modeling languages and the attempt to create complete models [Spe12]. That is why notations for

contract-based testing must be more user-friendly and easily adaptable for testers. Another reason for the missing acceptance is the tool support. Our approach fulfills these requirements as formulated in section 5.2.

**Contribution to Requirements 1:** VCBT uses Visual Contracts as modeling notation, which is a easy-to-learn contract language. Furthermore, Visual Contracts use UML Object diagrams for specifying pre- and post-conditions which makes this approach fully compatible with the UML based development process.

**Contribution to Requirement 5:** We have implemented the algorithms and procedures for test case generation, test script generation and test execution using Open Source technologies. We have used Eclipse as the platform for the testing framework which is extensible for further test selection criteria and for further test drivers. We have used JUnit as test driver for unit testing. JML assertions are used as as embedded test oracles. Groove model checker is used for computing state setting operation sequences. ATL is used for transformations of abstract test cases into implementation level test cases.

## 11.2 Epilogue

Even if developers have longtime experiences with these technologies and tools, testers are relatively new to the model-driven development techniques. Due to the Gartner Hype Cycle of application development from 2007, model-based testing (see *scriptless testing* [Gar07]) was seen as a rising hype topic “generating over-enthusiasm and unrealistic expectations” [Wik]. Since then, many case studies have been conducted with varying results. In [MGWE12] and [WGM<sup>+</sup>11], we have reported on the most influential case studies and their results. The main finding in these case studies was that MBT reduces efforts in repeating activities like test case design and test implementation. However, relatively big efforts must be invested for creating and maintaining the models thus many testers hesitate to introduce MBT and prefer waiting until more practical and light-weight approaches are invented. Also the Gartner hype cycle from 2010 acknowledges this fact, where MBT has reached the *trough of disillusionment* where it does not “meet expectations and quickly becomes unfashionable” [Wik].

In order to build acceptance in MBT among the test organizations, suitable modeling notations must be selected which are easy to learn and use by testers. Also the integration of development and testing must be assured by enabling model interchange between developers and testers. In some cases, testers should reuse developer models as test basis. This is mostly the case

if low level development and testing activities are considered. In other cases, testers should be able to create their own test models which are independent of development artifacts. In summary, the tight integration of testing and development activities through many test levels and the usage of similar light-weight modeling notations and corresponding tools will improve the acceptance of MBT among the test teams.

Traditionally, MDD approaches uses UML notations for behavioral and structural modeling. Thereby, developers tend to create complete specifications in order to be able to generate program code from the models automatically. However, because of many unsolved issues, the automatic code generation for an executable program code is not feasible. Also for testers, using behavioral and structural UML notations lead to complete test models, which require too high efforts compared with the benefit they bring and are heavily maintainable.

An alternative to the classical UML notations is the *contract-based* modeling paradigm where system behavior is described partially by specifying preconditions and postconditions for system functions. Thus, contracts do not aim at describing complete behavior but some aspects of it which the modeler wants to focus on. Developers can use then contracts as minimal specifications and implement program code using a combination of automatic code generation and manual coding fulfilling the conditions of the contract. In this way developers create more light-weight models which are more maintainable and developers can handle the flaws of the automatic code generation manually.

The research community has proposed the usage of many contract modeling notations for development and testing so far (e.g. Eiffel, JML, OCL [Ciu08, Mar04, Gro05a, AW05]). However, a comparison of these notations (cf. chapter 5, Figure 5.2) have shown that no one of the proposed notations fully fulfill the requirements of developers and testers in the context of model-driven software development. As a result, a new UML-based contract notations was developed by Lohmann [Loh06], which is called *Visual Contracts*. Visual Contracts specify preconditions and postconditions using two object diagrams which are typed over a class diagram. The modeler is free to choose the detail level for the preconditions and postconditions, which enables partial specifications. Lohmann has also defined a development process using Visual Contracts, where parts of program code are generated from the Visual Contracts automatically and completed manually. Even if automatic code generation helps in assuring the correctness of the generated code, because of the manual completion, errors can be inserted into the program code. That is why, the Visual Contract-based development process has to be extended by a suitable testing approach.

## 11.3 Roadmap for further Research

In our research, we have dealt with a wide spectrum of topics of software engineering. We have shown that Visual contracts can be seamlessly integrated into the Model-driven software development process. However, during our research, we have also identified further possibilities, how this spectrum can be extended. In this section we describe some ideas, how Visual contracts can be used for further software engineering topics.

### 11.3.1 Further selection criteria for test data

As described in chapters 7-9, we require concrete test data for input parameters of the functionalities under test. To keep it simple we have dealt with some basic techniques *Boundary analysis* and *Random generation*, how concrete test data can be selected.

For increasing the test data coverage and to bring more alternation in to the test data, also further test data selection techniques can be integrated, e.g.

- Equivalence classes
- Statistical data coverage
- Pairwise testing

### 11.3.2 Binding real test data

Instead of generating random or boundary values, we could use real data from customer databases, which contain valuable information for testers. Visual Contracts can be used as a visual SQL language to formulate SQL queries. As shown in the Figures 11.3 and 11.4, two strategies are possible. Either the objects derived from the pre- or postconditions of a Visual contract can be independently be selected, or the data dependencies between the objects can also be considered. In the first case, simple *select* queries in SQL are formulated for each test object and filled with real data from a database. In the second case, the select queries must be refined with *where* clauses addressing the data relations between the objects as given in the pre- and postconditions.

An important issue while the usage fo real data is, that the customer data is not allowed to be directly used for testing purposes [Osw11], thus

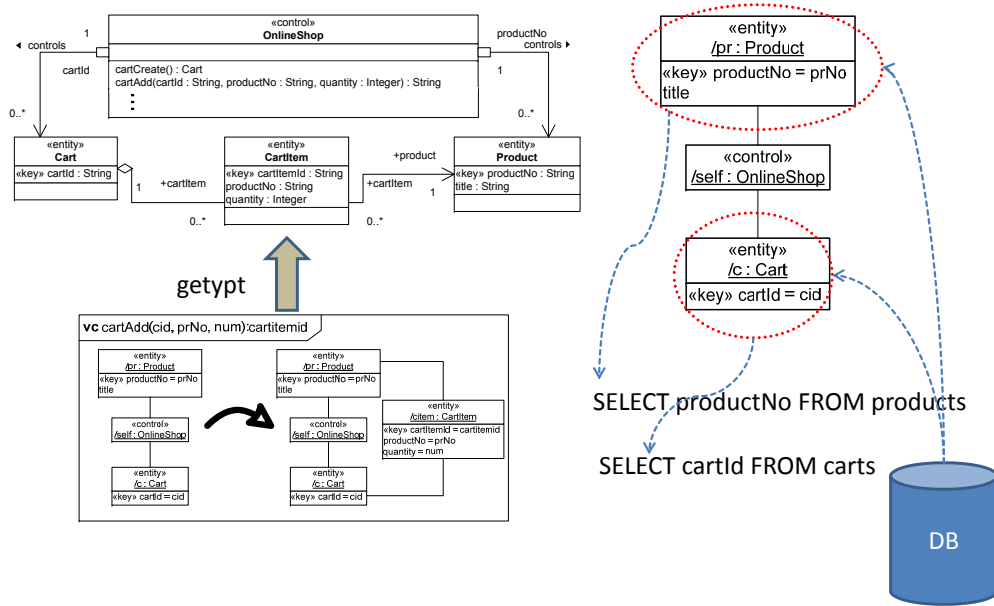


Figure 11.3: Extracting independent test data from DB

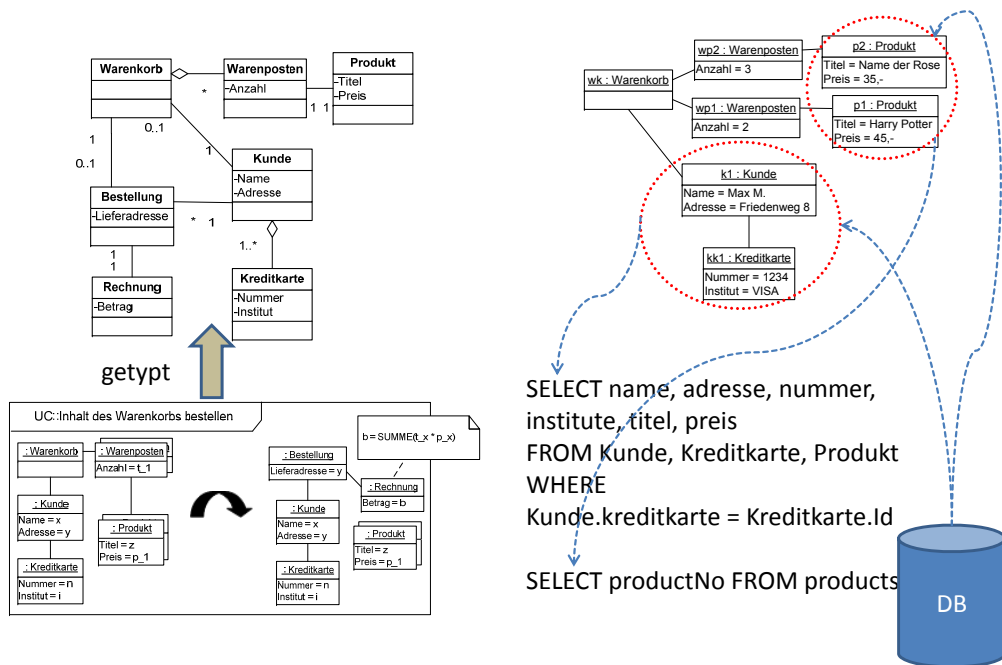


Figure 11.4: Extracting dependent test data from DB



it must be anonymized before using as test data. However, there are some approaches for anonymizing productive data, e.g. Q-up<sup>1</sup>.

### 11.3.3 Agile development

We have used Visual contracts in a plan-driven development process based on the V-model so far. Thereby, we have strictly distinguished between the test levels and the abstraction levels of the software specifications and the generated test cases. However, many software development companies apply agile development processes where the development phases and test levels are not strictly isolated. In Scrum [SB01], test activities of different levels are conducted in daily sprints.

In order to improve the effectiveness of test design and test automation, we have already proposed adapting model-based testing techniques into Scrum [GR11]. Thereby we have proposed to use light weight modeling techniques based on UML. However, even if the models are light weight, the consistency requirements between the models increases the complexity of this approach. Therefore, Visual contracts could support the agile testing activities as a light weight modeling notation since it does not require much relations to other UML notations other than Class diagrams.

---

<sup>1</sup><https://www.q-up-data.com/>



# Bibliography

- [ABLN06] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, Aug 2006.
- [AS05] T. Linz A. Spillner. *Basiswissen Softwaretest*. dpunkt.verlag, 3 edition, 2005.
- [Ave10] Michael Averstegge. Generisches testen von web services: Automatisierung ohne programmierung. *OBJEKTSpektrum, SIGSDATACOM*, 4:69–74, 2010.
- [AW05] Michael Averstegge and Mario Winter. Structural and functional predicate coverage testing. In *Proc. of 3rd World Congress for Software Quality (WCSQ) 2005*, pages 1–12, 2005.
- [BBF<sup>+</sup>10] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools*. Springer Publishing Company, Incorporated, 2010.
- [Beu09] Dominik Beulen. Evaluierung eines ansatzes zum kontraktbasierten komponententest für eine datenintensive java-anwendung. Master’s thesis, University of Paderborn, 2009.
- [BG10] Y. Sancar B. Güldali, M. Mlynarski. Effort comparison for model-based testing scenarios. In *Proc. of Quombat Workshop at ICST*, 2010.
- [BH02] Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *Graph Transformation*, pages 402–429. Springer Berlin Heidelberg, 2002.

- [Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proc. of International Symposium on Software Testing and Analysis (ISSTA) 2002*, pages 123–133, 2002.
- [Blo11] Jason Bloomberg. Functional vs. interface granularity: Still powerful ideas, February 2011.
- [Boe84] B. W. Boehm. Verifying and validating software requirements and design specifications. *IEEE Softw.*, 1(1):75–88, 1984.
- [Bor09] Lars Borner. *Integrationstest - Testprozess, Testfokus und Integrationsreihenfolge*. PhD thesis, Ruprecht – Karls – Universität, Heidelberg, 2009.
- [BP06] Luciano Baresi and Mauro Pezzè. An introduction to software testing. *Electr. Notes Theor. Comput. Sci.*, 148(1):89–111, 2006.
- [CDRT04] P. Collet, D. Deveaux, R. Rousseau, and Y.L. Traon. Contract-based testing: from objects to components. In *Proc. of first International Workshop on Testability Assessment, 2004. IWoTA 2004.*, pages 5 – 14, 2 2004.
- [CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM SYSTEMS JOURNAL*, 45(3):621–645, 2006.
- [Che06] Alexey Cherkhago. *Service specification and matching based on graph transformation*. PhD thesis, University of Paderborn, <http://ubdata.uni-paderborn.de/ediss/17/2006/cherchag/disserta.pdf>, 2006.
- [Cho78] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.
- [Ciu08] Ilinca Ciupa. *Strategies for Random Contract-Based Testing*. PhD thesis, ETH Zürich, 2008.

- [CL05] I. Ciupa and A. Leitner. Automatic testing based on design by contract. In *Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World)*, pages 545–557, 2005.
- [CMR<sup>+</sup>97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997. Chapter Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach.
- [Coc01] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.
- [Con06a] Eclipse Consortium. Atl - a model transformation technology, 2006.
- [Con06b] Eclipse Consortium. Eclipse modeling framework (emf) - version 2.1.2., 2006.
- [Con06c] Eclipse Consortium. Java emitter templates (jet). eclipse modeling framework (emf) - version 2.1.1, 2006.
- [Cri11] Peter Cripps. Architectural granularity, April 2011.
- [dDVe01] Gesamtverband der Deutschen Versicherungswirtschaft e.V. Versicherungsanwendungsarchitektur (vaa), 2001.
- [DNSVT07] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASEL Tech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 31–36, New York, NY, USA, 2007. ACM.
- [DNT09] Arilo Claudio Dias-Neto and Guilherme Horta Travassos. Model-based testing approaches selection for software projects. *Inf. Softw. Technol.*, 51(11):1487–1504, 2009.
- [ECFGP10] Sergio España, Nelly Condori-Fernandez, Arturo González, and Óscar Pastor. An empirical comparative evaluation of requirements engineering methods. *Journal of Brazilian Computer Society*, 16:3–19, 2010.

- [EEG08] Jens Ellerweg, Gregor Engels, and Baris Güldali. Modellbasierter komponententest mit visuellen kontrakten. In H.-G. Hegering, A. Lehmann, H. J. Ohlbach, and C. Scheideler, editors, *INFORMATIK 2008, Beherrschbare Systeme - dank Informatik, Band 1, Beiträge der 38. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, volume 133 of *Lecture Notes in Informatics*, pages 211–214, Bonn, 2008. Gesellschaft für Informatik (GI).
- [EG14] Kent Beck Erich Gamma. Junit, 2014.
- [EGL06a] Gregor Engels, Baris Güldali, and Marc Lohmann. Towards model-driven unit testing. In D. Hearnden, J.G. Süß, N. Rapin, and B. Baudry, editors, *Proceedings of the workshop on Model Design and Validation (MoDeVa 2006), Toulouse (France)*, pages 16–29, Berlin / Heidelberg, October 2006. Le Commissariat à l’Energie Atomique - CEA.
- [EGL<sup>+</sup>06b] Gregor Engels, Baris Güldali, Marc Lohmann, Oliver Juwig, and Jan-Peter Richter. Industrielle fallstudie: Einsatz visueller kontrakte in serviceorientierten architekturen. In Bettina Biel, Matthias Book, and Volker Gruhn, editors, *Software Engineering*, volume 79 of *LNI*, pages 111–122. GI, 2006.
- [Ell08] Jens Ellerweg. Komponententest mit visuellen kontrakten. Master’s thesis, University of Paderborn, 2008.
- [ELSH06] Gregor Engels, Marc Lohmann, Stefan Sauer, and Reiko Heckel. Model-driven monitoring: An application of graph transformation for design by contract. In *International Conference on Graph Transformation (ICGT) 2006*, 2006.
- [ESS08] Gregor Engels, Stefan Sauer, and Christian Soltenborn. Unternehmensweit verstehen – unternehmensweit entwickeln: Von der modellierungssprache zur softwareentwicklungsmethode. *Informatik-Spektrum*, 31:451–459, 2008. 10.1007/s00287-008-0274-9.
- [FG99] M. Fewster and D. Graham. *Software Test Automation*. Addison Wesley, 1999.
- [FK96] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Trans. Softw. Eng. Methodol.*, 5(1):63–86, January 1996.

- [För09] Alexander Förster. *Pattern based business process design and verification*. PhD thesis, University of Paderborn, 2009.
- [Fow07] Martin Fowler. Mocks aren't stubs, February 2007.
- [FvBK<sup>+</sup>91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Softw. Eng.*, 17(6):591–603, 1991.
- [Gar07] Gartner. Gartner's hype cycle, 2007.
- [GMR<sup>+</sup>12] AmirHossein Ghamarian, Maarten Mol, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using groove. *International Journal on Software Tools for Technology Transfer*, 14(1):15–40, 2012.
- [GMWE09] Baris Güldali, Michael Mlynarski, Andreas Wübbeke, and Gregor Engels. Model-based system testing using visual contracts. In *EUROMICRO-SEAA*, pages 121–124, 2009.
- [GR11] Baris Güldali and Thomas Rossner. Integration von modellbasiertem testen in den agilen entwicklungsprozess. In *German Testing Day*, 2011.
- [Gro03a] Object Management Group. Mda guide, 2003.
- [Gro03b] Object Management Group. Unified modeling language specification, 2003.
- [Gro05a] Hans-Gerhard Gross. *Component-Based Software Testing with UML*. Springer Verlag, 2005.
- [Gro05b] Object Management Group. Mof qvt final adopted specification, omg adopted specification ptc/05-11-01, 2005.
- [Gül05] Baris Güldali. Model testing – combining model checking and coverage testing. Master's thesis, University of Paderborn, 2005.
- [Han08] Dennis Hannwacker. Kontraktbasierte generierung von methodensequenzen für testfälle mittels model-checking. Master's thesis, University of Paderborn, 2008.

- [Hau05] Jan Hendrik Hausmann. *Dynamic meta modeling : a semantic description technique for visual modeling languages*. PhD thesis, University of Paderborn, 2005.
- [Hec98] Reiko Heckel. *Open Graph Transformation Systems*. PhD thesis, Technical University of Berlin, 1998.
- [Hec06] Reiko Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148:187–198, 2006.
- [HKS97] Pei Hsia, David Kung, and Chris Sell. Software requirements and acceptance testing. *Annals of Software Engineering*, 3(1):291–317, 1997.
- [HL03] Reiko Heckel and Marc Lohmann. Towards model-driven testing. *Electronic Notes in Theoretical Computer Science*, 82(6):33 – 43, 2003. TACoS’03, International Workshop on Test and Analysis of Component-Based Systems (Satellite Event of ETAPS 2003).
- [HM05] Reiko Heckel and Leonardo Mariani. Automatic conformance testing of web services. In *Fundamental Approaches to Software Engineering (FASE) 2005*, pages 34–48, 2005.
- [ISO] Iso/iec/ieee 29119 the international software testing standard.
- [ISO94] Iso/iec 9646-1. information technology—open systems interconnection—conformance testing methodology and framework, part 1: General concepts., 1994.
- [ISO01] Iso/iec 9126-1 software engineering - product quality, 2001.
- [ISTQB] ISTQB International Software Testing Qualifications Board. Certified tester - foundation level.
- [Kah62] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.
- [KBJV06] I. Kurtev, J. Bezivin, F. Jouault, and P. Valduriez. Model-based dsl frameworks. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 602–616, New York, 2006. ACM Press.
- [Kha12] Tamim Ahmed Khan. *MODEL-BASED TESTING USING VISUAL CONTRACTS*. PhD thesis, Tamim Ahmed Khan, 2012.



- [KKBK07] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, and Inyong Ko. Test cases generation from uml activity diagrams. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 3, pages 556–561, July 2007.
- [Kla08] Dominik Klaholt. Einsatz visueller kontrakte für modellbasierten systemtest am beispiel einer web-anwendung. Master’s thesis, University of Paderborn, 2008.
- [Kön10] Patrick Könemann. Design decisions in model-driven software development. In *Software Engineering (Workshops)*, pages 531–536, 2010.
- [KR06] Harmen Kastenbergh and Arend Rensink. Model checking dynamic states in groove. In *13th International SPIN Workshop*, pages 299–305, 2006.
- [Kru99] P. Kruchten. *Der Rational Unified Process - Eine Einführung*. Addison-Wesley, 2 edition, 1999.
- [KS06] Alexander Königs and Andy Schürr. Tool integration with triple graph grammars - a survey. *Electronic Notes in Theoretical Computer Science*, 148(1):113 – 150, 2006. `};ce:title;Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004);/ce:title; ;xocs:full-name;Foundations of Visual Modelling Techniques 2004;/xocs:full-name;.`
- [Küs04] Jochen Malte Küster. *Consistency management of object-oriented behavioral models*. PhD thesis, University of Paderborn, 2004.
- [LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.
- [LC03] G. Leavens and Y. Cheon. Design by contract with jml, 2003.
- [LES06] Marc Lohmann, Gregor Engels, and Stefan Sauer. Model-driven monitoring: Generating assertions from visual contracts. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE) 2006 Demonstration Session*, 2006.

- [LG87] Terttu Luukkonen-Gronow. Scientific research evaluation: a review of methods and various contexts of their application. *R&D Management*, 17:207—221, 1987.
- [LJX<sup>+</sup>04] Wang Linzhang, Yuan Jiesong, Yu Xiaofeng, Hu Jun, Li Xuan-dong, and Zheng Guoliang. Generating test cases from uml activity diagram based on gray-box method. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 284–291, Nov 2004.
- [LMH07] Marc Lohmann, Leonardo Mariani, and Reiko Heckel. A model-driven approach to discovery, testing and monitoring of web services. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 173–204. Springer, 2007.
- [LO94] Sølvsberg A Lindland OI, Sindre G. Understanding quality in conceptual modeling. *IEEE Software*, 11(2):42–49, 1994.
- [Loh06] Marc Lohmann. *Kontraktbasierte Modellierung, Implementierung und Suche von Komponenten in serviceorientierten Architekturen*. PhD thesis, University of Paderborn, 2006.
- [LSE05] Marc Lohmann, Stefan Sauer, and Gregor Engels. Executable visual contracts. In Martin Erwig and Andy Schürr, editors, *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 63–70, 2005.
- [Lud03] Jochen Ludewig. Models in software engineering – an introduction. *Softw Syst Model*, 2:5—14, 2003. cite this!
- [Mar04] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004.
- [MBE<sup>+</sup>07] T. Müller, R. Black, S. Eldh, D. Graham, K. Olsen, M. Pyhäjärvi, G. Thompson, and E. van Veendendal. Certified tester - foundation level syllabus - version 2007, 2007. International Software Testing Qualifications Board (ISTQB), Möhrendorf, Germany.
- [Mey92] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

- [Mey05] Bertrand Meyer. Doing more with contracts: Towards automatic tests and proofs. In *RISE*, page 1, 2005.
- [MG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.
- [MGWE12] Michael Mlynarski, Baris Güldali, Stephan Weißleder, and Gregor Engels. Model-based testing: Achievements and future challenges. *Advances in Computers*, 86:1–39, 2012.
- [MH09] Horst Pohlmann Matthias Hamburg, Uwe Hehn. Istqb/gtb standard glossar der testbegriffe. Technical report, German Testing Board (GTB), 2009. Version 2.0.
- [MSM<sup>+</sup>07] Aleksandar Milicevic, Sasa, Misailovi, Darko Marinov, and Sarfraz Khurshid. Korat: A tool for generating structurally complex test inputs. In *Proc. of 29th International Conference on Software Engineering (ICSE'07)*, pages 771–774. IEEE Computer Society Press, 2007.
- [NSV<sup>+</sup>08] Arilo Dias Neto, Rajesh Subramanyan, Marlon Vieira, Guilherme Horta Travassos, and Forrest Shull. Improving evidence about software technologies: A look at model-based testing. *IEEE Software*, 25:10–13, 2008.
- [Osw11] Stephan Oswald. Software testen mit rechtskonformen daten. *OBJEKTSpektrum*, Quality Managament:21–22, 2011.
- [Pet01] Alexandre Petrenko. Fault model-driven test derivation from finite state models: annotated bibliography. pages 196–205, 2001.
- [PP04] Alexander Pretschner and Jan Philipps. Methodological issues in model-based testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 281–291. Springer, 2004.
- [Ren03] Arend Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE) 2003*, pages 479–485, 2003.

- [Ros09] Thomas Rossner. Modellbasiertes testen – eine einföhrung. *iX - heise Verlag*, 11:125–130, 2009.
- [Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
- [SB01] K. Schwaber and M. Beedle. *Agile Software Development with SCRUM*. Prentice-Hall, 2001.
- [Sch07] Ina Schieferdecker. Modellbasiertes testen. *OBJEKTSpektrum, SIGSDATACOM*, 03:39–45, 2007.
- [Sch09] Tim Schattkowsky. *Platform independent modeling of synthesizable software systems using UML 2*. PhD thesis, University of Paderborn, 2009.
- [SEG] University of Paderborn. Software Engineering Group. Fujaba tool suite.
- [SG10] Matthias Schnelte and Baris Güldali. Test case generation for visual contracts using ai planning. In *INFORMATIK 2010, Beiträge der 40. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, volume 176 of *Lecture Notes in Informatics*, pages 369–374. Gesellschaft für Informatik (GI), 2010.
- [Spe12] Various Speakers. Closing panel: Code generation - how far have we come in 5 years?, 2012.
- [SS10] Harry M. Sneed and Richard Seidl. Einflussfaktoren bei der schätzung von testprojekten. Online Ressource, 21. January 2010. Presentaion at Software Quality Days 2010.
- [Sta73] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien, 1973.
- [TTC] Testing and test control notation version 3 (ttn-3).
- [Ude05] Jon Udell. An interview with bill gates at pdc 2005, September 2005.
- [UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

- [UPL06] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Technical Report 04/2006, Department of Computer Science, The University of Waikato (New Zealand), April 2006.
- [VB05] Sira Vegas and Victor Basili. A characterisation schema for software testing techniques. *Empirical Softw. Engg.*, 10(4):437–466, 2005.
- [vL00] A. van Lamsweerde. Formal specification: a roadmap. In *Proc. ICSE'00*, pages 147–159, 2000.
- [W3C14] W3C. Web services description language (wsdl) version 2.0 part 1: Core language, 2014.
- [WEMS<sup>+</sup>12] Mario Winter, Mohsen Ekssir-Monfared, Harry M. Sneed, Richard Seidla, and Lars Borner. *Der Integrationstest: Von Entwurf und Architektur zur Komponenten- und Systemintegration*. Carl Hanser Verlag GmbH & CO. KG, 2012.
- [WGM<sup>+</sup>11] Stephan Weißleder, Baris Güldali, Michael Mlynarski, Arne-Michael Törsel, David Faragó, Florian Prester, and Mario Winter. Modellbasiertes testen: Hype oder realität? *OBJEKTSpektrum*, 06/2011:60–66, 2011.
- [Wik] Wikipedia. Hype cycle. Online. Last visited at 6.2.2013.
- [Win99] Mario Winter. *Qualitätssicherung für objektorientierte Software: Anforderungsermittlung und Test gegen die Anforderungsspezifikation*. PhD thesis, Fachbereich Informatik der FernUniversität - Gesamthochschule - in Hagen, September 1999.
- [Win09] Mario Winter. Modellbasierter test ? alter wein in neuen schläuchen? Presentation, 2009. Software & Systems Quality Conference 2009.

