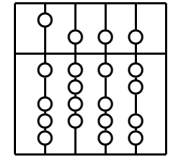


Technische Universität München  
Fakultät für Informatik

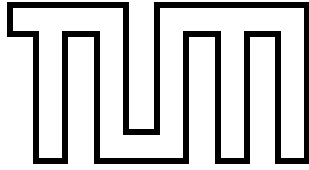


Diplomarbeit

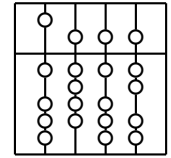
# **Component-Based Testing: Towards Continuous Integration in Software Engineering**

Andreas Schildbach





Technische Universität München  
Fakultät für Informatik



Diplomarbeit

# **Component-Based Testing: Towards Continuous Integration in Software Engineering**

Andreas Schildbach

Aufgabensteller: Prof. Bernd Brügge, Ph.D.

Betreuer: Dipl.-Inf. Univ. Oliver Creighton

Abgabedatum: 2001-11-15

# Component-Based Testing: Towards Continuous Integration in Software Engineering

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

All other trademarks and logos are trademarks or registered trademarks of their respective owners.

## Acknowledgements

I am thankful to *Bernd Brügge* for the opportunity to work on this project, his trust in my abilities, and his encouragements that kept me pushing my limits.

I would like to thank my supervisor *Oliver Creighton* for his invaluable help and advice for this thesis.

I am appreciative of *Michael*, *Mathias* and *Wolfgang* for their last-minute read-throughs.

I would like to thank *Violetta*, who gave me unlimited moral support.

## Erklärung

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

# Contents

<b>Abstract</b>	<b>7</b>
<b>I Introduction</b>	<b>9</b>
<b>1 Motivation</b>	<b>11</b>
1.1 Productivity and Reliability . . . . .	11
1.2 The Objective . . . . .	12
1.3 Structure of this Document . . . . .	13
<b>2 The Anatomy of a Component</b>	<b>15</b>
2.1 Basics . . . . .	15
2.2 Correctness . . . . .	16
<b>3 The Dynamic World</b>	<b>19</b>
3.1 Dependencies . . . . .	19
3.2 Versioning . . . . .	20
<b>4 Managing Change</b>	<b>23</b>
4.1 Dependencies between Components . . . . .	23
4.2 Transition between Versions . . . . .	24
4.3 Mission Statement . . . . .	25
<b>5 Related Work</b>	<b>27</b>
5.1 Tools . . . . .	27
5.2 Concepts . . . . .	28
<b>II The Testbench</b>	<b>29</b>
<b>6 Prerequisites</b>	<b>31</b>
6.1 Uniform Project Name . . . . .	31
6.2 Software . . . . .	31
6.3 The Software Engineering Approach . . . . .	32

<b>7 Problem Statement</b>	<b>33</b>
7.1 Filling The Gap . . . . .	33
<b>8 Requirements Elicitation</b>	<b>35</b>
8.1 Use Cases . . . . .	35
8.2 Requirements . . . . .	39
<b>9 Analysis</b>	<b>41</b>
9.1 Constraint Languages . . . . .	41
9.2 Analysis Model . . . . .	41
<b>10 System Design</b>	<b>43</b>
<b>11 Object Design</b>	<b>45</b>
11.1 Model . . . . .	45
11.2 Run-Time . . . . .	46
11.3 Language . . . . .	46
11.4 Action . . . . .	47
11.5 Ant . . . . .	48
<b>12 Implementation</b>	<b>49</b>
<b>13 Test</b>	<b>51</b>
13.1 T.R.A.M.P . . . . .	51
<b>III Results</b>	<b>53</b>
<b>14 Testing with Testbench</b>	<b>55</b>
14.1 Unit Tests . . . . .	55
14.2 Integration Tests . . . . .	56
14.3 System Tests . . . . .	56
<b>15 The OCL Interpreter</b>	<b>57</b>
<b>16 Split Personality: Inner and Outer Interface</b>	<b>59</b>
<b>17 Pattern Catalogue</b>	<b>61</b>
17.1 Pattern Template . . . . .	62
17.2 Uniform Project Name . . . . .	62
17.3 Project Structure and Version Control . . . . .	64
17.4 Design By Contract . . . . .	66
17.5 Component-Based Testing . . . . .	67
17.6 Automated Integration . . . . .	69
17.7 Continuous Integration . . . . .	71

<b>IV Discussion</b>	<b>73</b>
<b>18 Pros and Cons</b>	<b>75</b>
18.1 Code Instrumentation or External Checking? . . . . .	75
<b>19 Putting Constraints into Code?</b>	<b>77</b>
19.1 Roles . . . . .	77
19.2 Implementation Logic . . . . .	77
<b>20 Future Directions</b>	<b>79</b>
20.1 Test Stubs . . . . .	79
20.2 Enhanced Control Flow . . . . .	79
20.3 Extending the Process Pattern Catalogue . . . . .	80
20.4 Separate OCL Interpreter Project . . . . .	80
<b>V Appendix</b>	<b>81</b>
<b>A Project Structure and Version Control</b>	<b>83</b>
A.1 An Example . . . . .	83
<b>B Specification of the Model</b>	<b>85</b>
B.1 Document Type Definition . . . . .	85
B.2 Sample Document . . . . .	86
<b>C Specification of a Test Case</b>	<b>89</b>
C.1 Document Type Definition . . . . .	89
C.2 Sample Document . . . . .	91
<b>D Bibliography</b>	<b>95</b>
<b>E Index</b>	<b>99</b>

# List of Figures

2.1	The Anatomy of a Component . . . . .	16
3.1	Populating a Component . . . . .	20
3.2	Straightforward Development . . . . .	21
3.3	Development Branches . . . . .	21
4.1	An Example for Complicated Dependencies . . . . .	24
8.1	Use Case Diagram ( <i>Automated Integration</i> applied) . . . . .	36
8.2	Use Case Diagram ( <i>Continuous Integration</i> applied) . . . . .	37
9.1	The Analysis Model . . . . .	42
10.1	Decomposition into Subsystems . . . . .	44
11.1	The Class Model of the Model Subsystem . . . . .	45
11.2	The Language Subsystem . . . . .	47
11.3	Usage of the Command Pattern in the Action Subsystem . . . . .	48
12.1	Sequence Diagram of the Sample in Appendix C.2 . . . . .	50
16.1	Inner and Outer Interfaces . . . . .	59
16.2	The Extension to the Component Anatomy . . . . .	60
17.1	Process Patterns Relationships . . . . .	61

---

**ABSTRACT**

---

**Part I** talks about the problem domain and the motivation for the research. It provides the background for all following parts. Chapter 1 introduces the term reliability and how to control it. Chapters 2-4 deal with the trinity *static* aspects of a component, *dynamic* aspects of a component and the effect of *change*. The static aspects of a component are described as an anatomy, consolidating everything from source to executable in one coherent picture. The dynamic aspects are represented by dependencies between the anatomical parts of components and versions. An overview of related work in chapter 5 concludes part I.

**Part II** explains our approach of reaching the mission goal that was set in part I: extending the software engineering process by an omnipresent phase for component-based testing. Part II also describes our adoption of the software engineering process. As it goes through six phases (chapters 8-13), the problem statement is being refined. In the end, the final product is reached: *testbench*, a test harness framework for software components. The framework is extendable in every aspect: languages, runtime systems and actions can be added with little effort.

**Part III** presents the products and insights that result from the application of the methods described in part II. Chapter 14 explains the usage of testbench and chapter 15 talks about a byproduct, an interpreter for the Object Constraint Language. The concept of a component is extended in chapter 16 by examining not only the *outer interface*, most commonly known as the public interface, but also what we call the *inner interface*, which are all the used methods the component depends on. Chapter 17 concludes part III with a catalogue of process patterns for development of software.

**Part IV** discusses the results that were presented in part III. It weighs up pros and cons of constraint checking techniques (chapter 18). Code instrumentation has got greater access to the component's features and external checking is more flexible as the interface to the component can vary. Chapter 19 makes clear that putting constraints into source code is not a good idea. Finally, future directions are given in chapter 20. Enhancement ideas concern testbench, which could not only provide the test driver but also the test stubs. It could also be of more use to integration and system testing if it would provide methods for enhanced control flow. The OCL interpreter and the process pattern catalogue could be separated from this thesis and continued as projects of their own.

---



## Part I

# Introduction

## ABOUT THIS PART

---

This part talks about the problem domain. It introduces the important terms *reliability* and *correctness*.

This part then continues on to a definition of the concept of a *component*, their *artifacts* and *dependencies* that they comprise. The idea of a *version* is introduced.

The part concludes with a discussion of related work.

---

# Chapter 1

## Motivation

### 1.1 Productivity and Reliability

*When quality is pursued, productivity follows [GMJ91]*

When asked about new software development methods or tools, many people quickly bring up productivity as the major benefit. Quality only gets second places, if mentioned at all. People tend to forget that productivity without quality cannot exist, because a bad product will have to be redone at some time. Even more, productivity can appear without saying if quality is achieved.

One major component of quality is reliability, the ability to perform a job according to the specification and to handle abnormal situations. To put it more simply, it is the absence of bugs.

The reliability of components plays a very important role in the development of software. Should a component prove faulty, it has to be bug-fixed or even refactored. This costs the developers time and enterprises money. Even worse, if a component fails during operation, consequential damage can occur. For example, a faulty database component can mess up sensitive customer information.

There are several techniques to control the reliability of software components. They can all be put into three main categories:

1. **Fault Avoidance:** *The most harmless bug is the one that does not exist.*

Fault avoidance implements several methodologies that target at producing bug free components in the first place. For example, applying common-sense source code conventions helps developers in reading their own and source codes of others and thus avoids faults by misinterpreting.

2. **Fault Detection:** *Let's squish'em bugs!*

Fault detection deals with discovering defects that have crept in despite of all Fault Avoid-

ance techniques. Usually this is done by testing, either at source code level (also called debugging) or at run time. Testing is a very complex matter: There is a wealth of test methods and no combination of them will find all faults. Only formal verification can make guarantees, but can only be used in relatively few cases [BM79].

3. **Fault Tolerance:** *Who cares about that bug if the system can tolerate it?*

Fault tolerance is the last resort to prevent system failure due to faults that have hidden well enough to pass both Fault Avoidance and Fault Detection (or are caused by natural disasters).

Components should be made tolerant of defects in a way that the defect is in a completely controlled environment. The situation is detected, maybe the user is warned about the situation and measures are taken to either correct the situation or shut down gracefully. Never should a failure cause data corruption or unbearable system downtimes. A good example for Fault Tolerance is the transaction management that is a fundamental feature of today's middleware and integration tier servers: the program and information flow is split up into a number of parts that can be undone (*rolled back*) individually in case of failure.

The complexity of reliability often leads to some form of helplessness within the ranks of developers and administratives. As a human reaction, the issue of controlling reliability is neglected or even ignored completely. People only implement part of the techniques and concentrate on “getting the work done”. Unfortunately, this does not seem to work.

Unreliable software leads to immense maintenance costs, unhappy staff and in the end, no work is getting done at all. On the other hand, a reliable system's costs are calculable, the team is happy about “having done a good job” and they can address themselves to a new task.

Regardless of these positive side effects of quality, traditional techniques to control reliability are often seen as a tedious job. For example, most developers neglect documenting their work. This may be because they think that they have got everything in their mind and doing a “hard-copy” of their mind is futile.

## 1.2 The Objective

The surrounding environment of this thesis consists of techniques that

- are accepted by developers in a way that they like to apply them
- are efficient enough to raise the reliability by a good amount

The thesis itself focuses on building a fault detection tool. It is a *testbench* which a developer can use to put live components on and test at run time. This tool category is called “test harness”. The testbench supports the striving of developers for automation, which helps developers a great deal to perceive testing as more convenient.

### 1.3 Structure of this Document

This document is structured in the following way: Part I on page 10 introduces the problem domain and discusses related work. Part II on page 30 documents the development of the testbench. Part III on page 54 tells the reader about the results and how to apply them. Part IV on page 74 discusses the results. Finally, part V on page 82 contains the appendix, where sample documents, the bibliography, and an index can be found.

The document takes advantage of several font styles. Bold font in text indicates something the user types. A monospaced font is used for code output, URLs, and file and directory names.

Diagrams are in UML [omg00] where appropriate.



## Chapter 2

# The Anatomy of a Component

Components are interchangeable software parts. [Szy98] writes: Components are “deployment-units that can be dropped into a system and put to work”. In fact this covers only the aspects at deployment time. For developers, components consist of various artifacts that are described in this chapter. Figure 2 on the next page shows their relationships.

### 2.1 Basics

First of all, every developer knows the *source* files. They are created during the activity of implementation, where the object model is translated into actual code.

From the source, the compiler generates the *binary*<sup>1</sup>. It instructs the machine (be it virtual or real) how the component behaves. Depending on the component technology being used, the binary also contains the *interface*, which tells the client of the component (most likely other components) how to make use of the behaviour of the component. In cases where the interface is not contained in the binary, the interface is supplied by a different artifact. The component would now be ready for deployment in a perfect world.

Unfortunately, the world is not perfect. Time has told that without taking special precautions, components will not behave as expected. This brings up the question of how is a component expected to behave? This semantics is defined in the component *specification*, which consists of various models that result from the Software Engineering Process described in [BD00]. And how high is the component’s ability to behave exactly as defined in its specification? This ability is called *correctness* and is the topic of the next section.

---

<sup>1</sup>for the sake of clarity, we do not consider interpreted programming languages

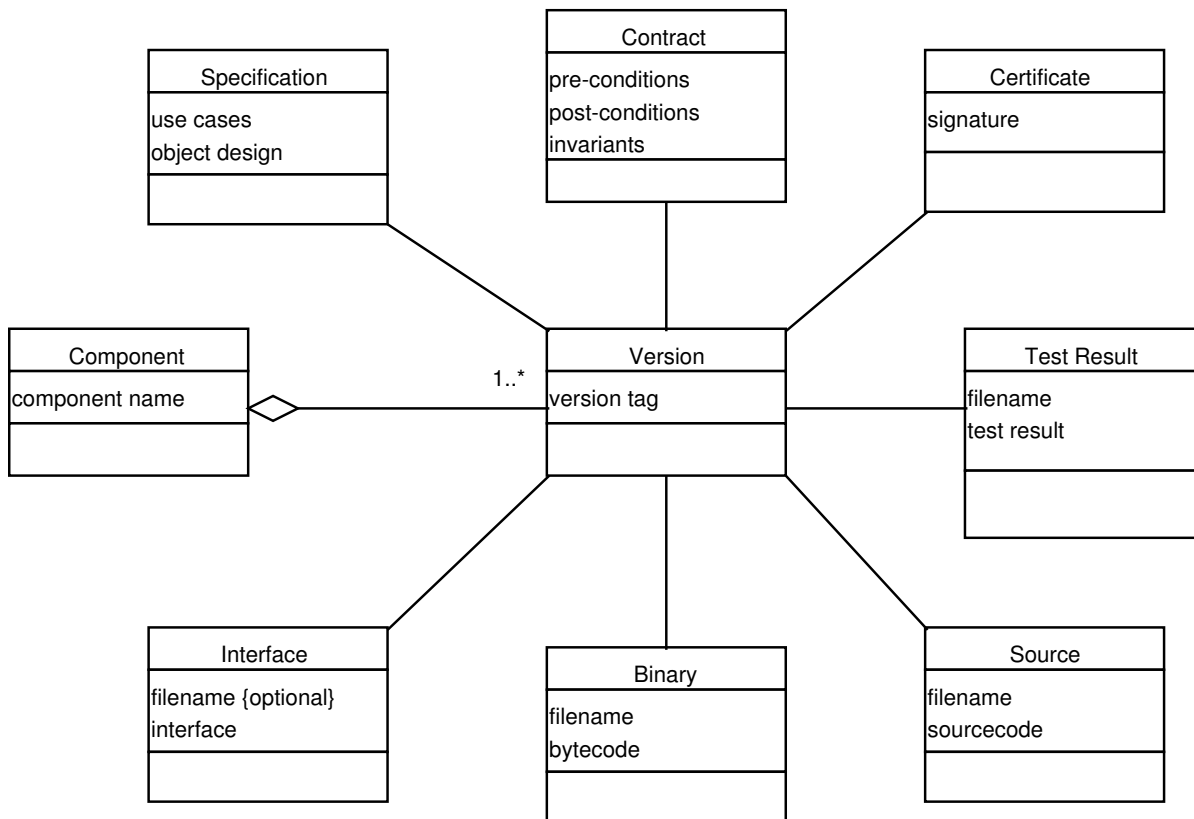


Figure 2.1: The Anatomy of a Component

## 2.2 Correctness

Judging the correctness of a software component is an activity that re-occurs frequently during development. There are several techniques that can be used for this task. Their result is always a statement about the confidence in the correctness.

For the following sections let us assume that this confidence is linearly scaled to a numeric interval. The smallest possible value is 0, meaning that it is absolutely unknown whether a component is correct. The highest value is 1 (or 100%) which means that the correctness is mathematically proven (verified). This is possible with the computational logic of Boyer and Moore [BM79], but not in every case. In every other case, developers fall back on testing.

Testing, on the other hand, is an activity that is based on proving the complement of correctness: incorrectness. Testing will never prove correctness, even a test that suggests that a component behaves as expected will not yield a correctness confidence value of 100%. It is not obvious how high the value would be in this case - this could be determined by statistical methods, but is out of this thesis' focus.

The higher the number of (different) negative tests<sup>2</sup>, the higher the correctness confidence will be. Confidence rises as the set of tests approaches the set of all possible tests (*completeness* of test space). At some point the correctness confidence is so high that the component is considered correct and is being certified. This *certificate* is the last artifact to be added to the component.

At this point the *Design by Contract* approach comes into play. Design by Contract has its origins in the Eiffel programming language [Mey92]. Having come a long way, it is now being used in more conventional languages like Java [ico] and C++. The central idea of Design by Contract is that the specification is transformed into a *contract*. It defines the relationship between the component and the client (user of the component), expressing each party's rights and obligations. The contract can be used as a foundation for writing test cases and also serves as documentation for the component.

---

<sup>2</sup>since a tester's job is to find bugs, we speak of a *positive test* or *success* if a bug was found and a *negative test* or *failure* if not.



## Chapter 3

# The Dynamic World

### 3.1 Dependencies

The preceding chapter described the static artifacts of a component. This model cannot be realistic. We live in a dynamic world. Requirements change, and so does software and its components. Components are added, modified or removed.

Components start out empty. Over their life span, they are being populated with artifacts. In *forward engineering* [BD00] the specification is added first. From the specification, the interface and the contract is derived. The implementor implements the interface, obeying the contract at the same time. The resulting source code is then compiled to a binary by a compiler. For testing the binary, the tester also needs the specification and the contract. Depending on the test result, the component can either be certified or the results are fed back to object design or implementation. These dependencies are visualized in an activity diagram (figure 3.1 on the following page).

A change of a requirement potentially outdates every artifact of the component. There are two strategies to handle this situation (both assuming forward engineering):

**clean build** Every artifact is just removed and then continued as if the component was just added.

**incremental build** As a beginning, only the specification is being outdated. For every artifact that is outdated, changes are applied and it is decided if that actually outdates the artifact that depends on the originating artifact. This step is repeated until no more artifacts need to change. This is called *ripple effect*, because it reminds of ripples originating from a stone thrown into a lake and running to every point of the surface.

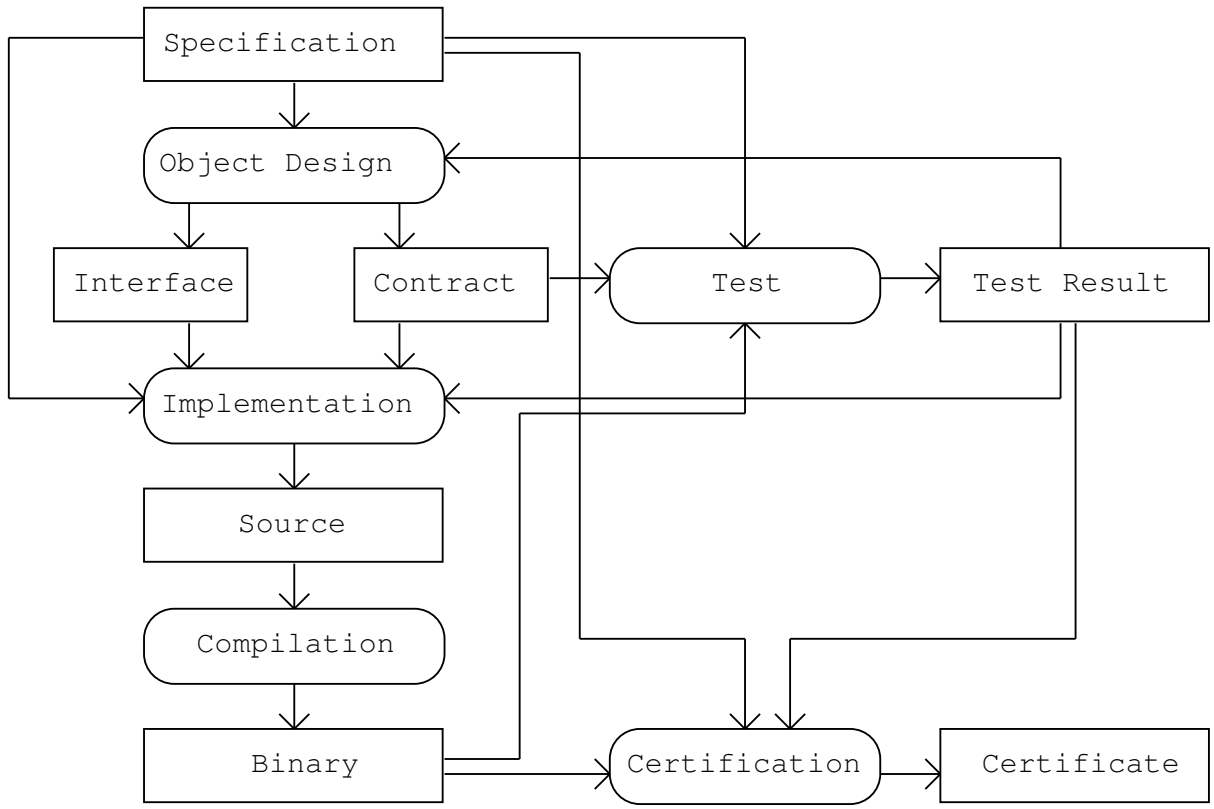


Figure 3.1: Populating a Component

### 3.2 Versioning

The concept of versions adds an organisation to the changes [Mun93]. Everytime a component changes, a new version is created. Usually, each version is tagged with a name (e.g. “1.1”) in order to uniquely identify a specific version.

Each version, with the exception of the very first one, is created from an already existing version. This is called the parent version. It is possible to draw a graph of all versions with unidirectional edges. If development goes straightforward, the graph will be very linear (figure 3.2 on the next page). As soon as any changes have to be made to an old version, the parent gets a second child and the development is branched (figure 3.2 on the facing page).

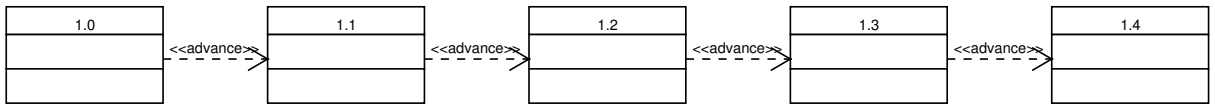


Figure 3.2: Straightforward Development

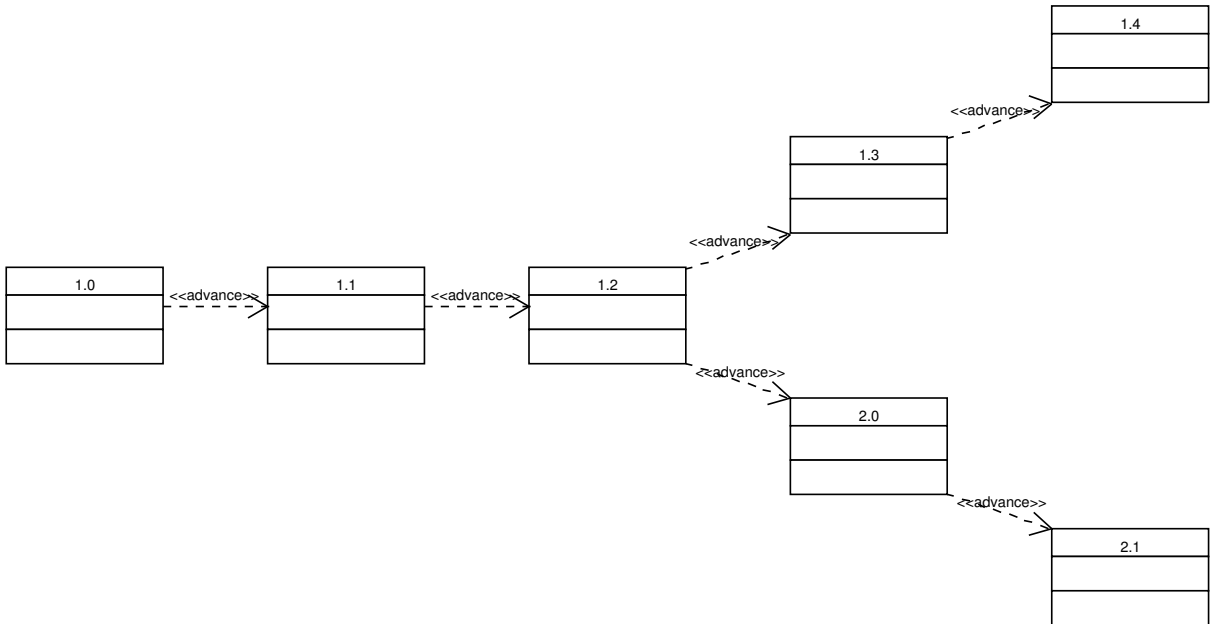


Figure 3.3: Development Branches



## Chapter 4

# Managing Change

### 4.1 Dependencies between Components

In the last chapter, we talked about *intra*-component dependencies, that is, the dependencies between the anatomical parts of a component. This section is about *inter*-component dependencies, which means the dependencies between multiple components.

In order to analyse what happens if a component changes, we have to distinguish the following cases:

- If the **interface** is modified (i.e. the signature of a method), all client components that use the changed parts will break. Problems are easy to find in this case: Either the compiler reports an error or the invocation of a non-existing method raises an exception,
- A change of the **semantics** is much more dangerous. In this case, nothing is preventing a client component to use the interface and getting a result of the expected type, albeit not of the expected value. This is because the expectation is still derived from the specification of the old version of the changed component, while the new version exposes the changed behaviour.

For this reason it is highly recommended to alter the interface due to modifications in semantics. A change of the method name is the least a developer can do in order to sail around this reef.

- A change of the **source** or **binary** is the typical case where no other components are affected, as long as the new implementation is still correct. For example, the code to detect a prime number can be replaced by a new one, if a faster algorithm has been found.

## 4.2 Transition between Versions

The last section made clear that if component B depends on A and A gets a new version, it is very likely that B will get a new version, too. The right point in the software engineering process to decide about that is *object design*, because in that phase all components are known and the dependencies are clear.

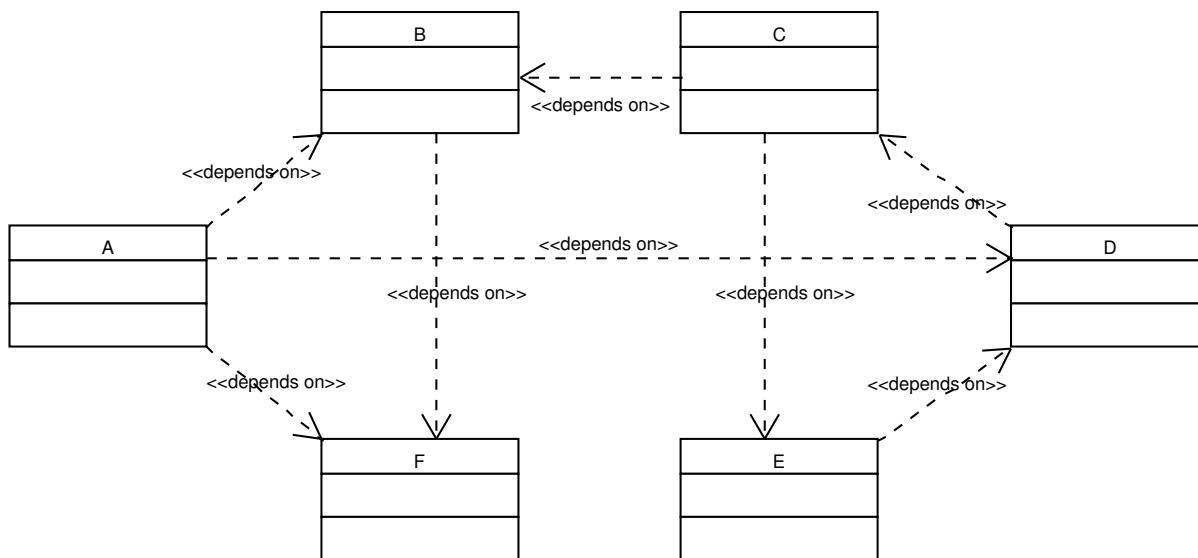


Figure 4.1: An Example for Complicated Dependencies

A problem is when A *and* B need to be changed. Regardless if you change A or B first, the system will be broken until you change both components. In real projects, there are many more components than just two. Figure 4.2 shows how complicated things can be. The change of mutually dependent components is usually handled in one of the following ways:

- All components are changed at the same time. For large projects, this will require a lot of effort. Unfortunately, the latency between the change and the point in time where the system is consistent again can be quite high.
- Components are changed one after another. In order for the system to stay consistent at any given moment, each component not only implements the new interface, but also the old interface. This can be done by adapter methods, that are marked deprecated.

## 4.3 Mission Statement

Chapter 2 on page 15 discussed the static foundation of components. Chapter 3 on page 19 went on with the dynamic aspects. This chapter added the time axis to the picture. These three chapters describe the reality of the software engineering process.

Reality is coined by approved methods. It is one goal of this thesis to formulate these methods. The result of this effort is a catalogue of process patterns (chapter 17 on page 61). Further on, testing has been identified as an omnipresent aspect of the process. Hence, we will extend the process at multiple ends instead of talking about a separate testing phase (see part II on page 30 for a description of our approach). The next chapter discusses work that have already been done in this realm.



## Chapter 5

# Related Work

There are several tools that focus on component-based testing. Furthermore, there are methods that make use of the concepts made possible by the tools.

### 5.1 Tools

**iContract** [ico] is described by its developers by the term “The Java Design by Contract Tool”. It is a freely available sourcecode pre-processor which takes expressions from special comment tags as an input and then instruments the code with checks for class invariants, pre- and postconditions. The expressions are roughly compatible with the Object Constraint Language (OCL) [omg00]. Due to the non-mandatory nature of the comment tags, source code that contains annotations remains fully compatible with Java and can be processed with standard Java compilers. The disadvantage of iContract lies in its inflexibility: Constraints can only be formulated by OCL and cannot be kept separate to Java sources.

**Dresden OCL Injector** is a subsystem of the OCL toolkit available from the Dresden University [tud]. It was developed as part of a diploma thesis [Fin00]. The term *Injector* stems from the ability of the tool to put code fragments which check OCL constraints into user Java code. Like iContract, the OCL constraints are read from comment tags. It is limited to OCL, too.

**JUnit** [jun] is a regression testing framework written by Erich Gamma and Kent Beck. It can be seen as the Java version of the more general XUnit framework. For each test, a Java class has to be written that implements a generic testing interface which is defined by the framework. Test classes can then be bundled into so-called test suites. Different clients for testing are possible, like special tasks for Ant or a GUI frontend. Since tests need to be implemented using Java code, the tester needs to know Java quite well. This is one of the

main disadvantages of JUnit because the specification of test cases can only be done by implementors.

**U.S.E. [use]** is a graphical and UML-based specification environment. The developer specifies a model which describes a system by using UML and OCL. He can then verify the system by manipulating and observing the system's state. However, this tool cannot be used for testing, as no code is being executed. Every manipulation must be done manually.

## 5.2 Concepts

**Extreme Programming (XP)** Beck [Bec00] describes some concepts that influenced this thesis. For example, the idea of Continuous Integration is sketched. In the patterns part of this thesis (chapter 17 on page 61), an extension to Continuous Integration is introduced. Beck also talks about testing code, and that testing is so important that he recommends to write the tests before actually implementing the code to be tested. He also sees implementation and testing as a whole, as each change has to be regression tested against the existing codebase.

**Assertions** Many programming languages offer assertions as a means for debugging. Mostly they use the keyword `assert` followed by a boolean expression as an argument. Should the expression evaluate to `false`, an exception or an error is raised, else execution continues. Special care must be taken that the assertions do not have any side-effects, i.e. the contained expression must not modify the state of the system.

The next part describes our approach of extending the ideas and achievements of these works, while we try to sail around the disadvantages that have been discussed.

## Part II

# The Testbench

## ABOUT THIS PART

---

This part explains the methods applied to achieve the results that are described in part III on page 54.

This part documents an implementation of the Component-Based Testing pattern described in chapter 17.5 on page 67.

---

# Chapter 6

## Prerequisites

### 6.1 Uniform Project Name

At first, an official project name is defined: *testbench*. This name is derived from the idea of lifting components to some workbench and testing them. It does also fit into the requirements for project names: it can be used as a name element in every language and development tool. Most importantly, it can be used as (a part of) file, directory, variable and package names.

This approach implements the Uniform Project Name process pattern, which is described in the appendix 17.2 on page 62.

### 6.2 Software

During the software engineering process, several software products are needed. We prefer tools from the open source community, as these tools often become a standard [RY99].

From the very first phases on, a program for drawing UML diagrams is needed. We chose to use ArgoUML, as it is not only open source, but also platform independent. ArgoUML is much more than just a UML diagram editor. It is “a domain-oriented design environment that provides cognitive support of object-oriented design” [arg].

The implementation phase usually contains the decision for a programming language. For various reasons that are explained in chapter 12 on page 49, the Java programming language was chosen. While not open source, but at least free to use, the SUN Java Development Kit (JDK) [sunb] was used. From all of the different JDK versions, v1.3.1 has been chosen because it is the latest yet stable release. In order to benefit from some of the JDK 1.4<sup>1</sup> APIs, support for XML processing and regular expressions was added by including optional packages.

---

<sup>1</sup> at the time of the writing of this thesis, JDK 1.4 was only available as a beta release

In the system design phase in chapter 10 on page 43, a build tool is searched for. Ant, which is rapidly gaining momentum, fulfills the needs. This is what the developers of the tool say: “Ant is a Java-based build tool. In theory it is kind of like ‘make’ without makes wrinkles and with the full portability of pure Java code” [jak01]. Basically, the developer writes a build file (`build.xml`) for a project in an XML dialect. Elements include targets (and its dependencies) and parameterized actions. We used Ant version 1.4.1.

### 6.3 The Software Engineering Approach

One of our goals was to follow a modern software development process. We adopted the process from [BD00] and assigned separate chapters to every phase:

1. Problem Statement (chapter 7 on the facing page)
2. Requirements Elicitation (chapter 8 on page 35)
3. Analysis (chapter 9 on page 41)
4. System Design (chapter 10 on page 43)
5. Object Design (chapter 11 on page 45)
6. Implementation (chapter 12 on page 49)
7. Test (chapter 13 on page 51)

For the sake of clarity, the actual development progress is described in the form of the classical waterfall model. However, in reality it was not linear. Many iterations were done and a lot of partial results were sacrificed on the way.

## Chapter 7

# Problem Statement

The problem statement tells about the problem to be solved in a very general way. Basically, just an idea is sketched.

### 7.1 Filling The Gap

There is a multitude of solutions for nearly every part of a software system integration cycle: Compilers transform source code into executable code. Document generators typeset electronic books by processing documentation source files [suna]. Build tools manage dependencies, prepare the directory structure, assemble all files and bundle them into deliverables [jak01].

However, one part is neglected in the integration cycle: component-based testing. Investigations reveal that in fact there are tools that allow developers to do component-based tests. Most of the tools mix up the component contract with the implementation itself (i.e. by writing JavaDoc comments into the implementing source code [ico]). This practise is not a good idea because it supports copying errors from the contract to the implementation by thinking the same way. Ideally, the contract should be separate from the implementation not only by different artifacts but also by varying developers.

The idea is to develop a testing environment that blends nicely with the integration cycle. Test definition will be part of object design and test execution part of every integration.



## Chapter 8

# Requirements Elicitation

Describing the purpose of the system to be developed during a project is the main point of the phase of requirement elicitation [BD00].

### 8.1 Use Cases

As a first step, the actors are identified. For the testbench, the actors can be developers or systems that interact with the testbench. A developer can act as a specification writer, implementor, integrator or tester. Actors can also be systems like the repository or the build system.

Use cases describe typical functionality provided by the future system. The overview can be derived from the problem statement: The *specification writer* specifies the components and their contracts. The *implementor* puts the component specification into actual code and a *tester* writes the unit test plan. An *integrator's* job is to come up with an integration strategy and instruct the build system how to implement that strategy. The artifacts resulting from all use cases are stored in the same repository (*single source point*, chapter 17.3 on page 64). Each actor can trigger the integration process, either explicitly (*automated integration*, chapter 17.6 on page 69) or implicitly by altering the contents of the repository (*continuous integration*, chapter 17.7 on page 71). At some point during integration, the test is run by the build system. Unsuccessful tests are presented to the developers by appropriate warnings or exceptions. The team has to make sure that no test fails (either an implementation has to be corrected or a test specification has to be refined [Bec00]).

Figures 8.1 on the following page and 8.1 on page 37 picture the dependencies between the actors and the use cases. Formal use case descriptions follow on the next pages.

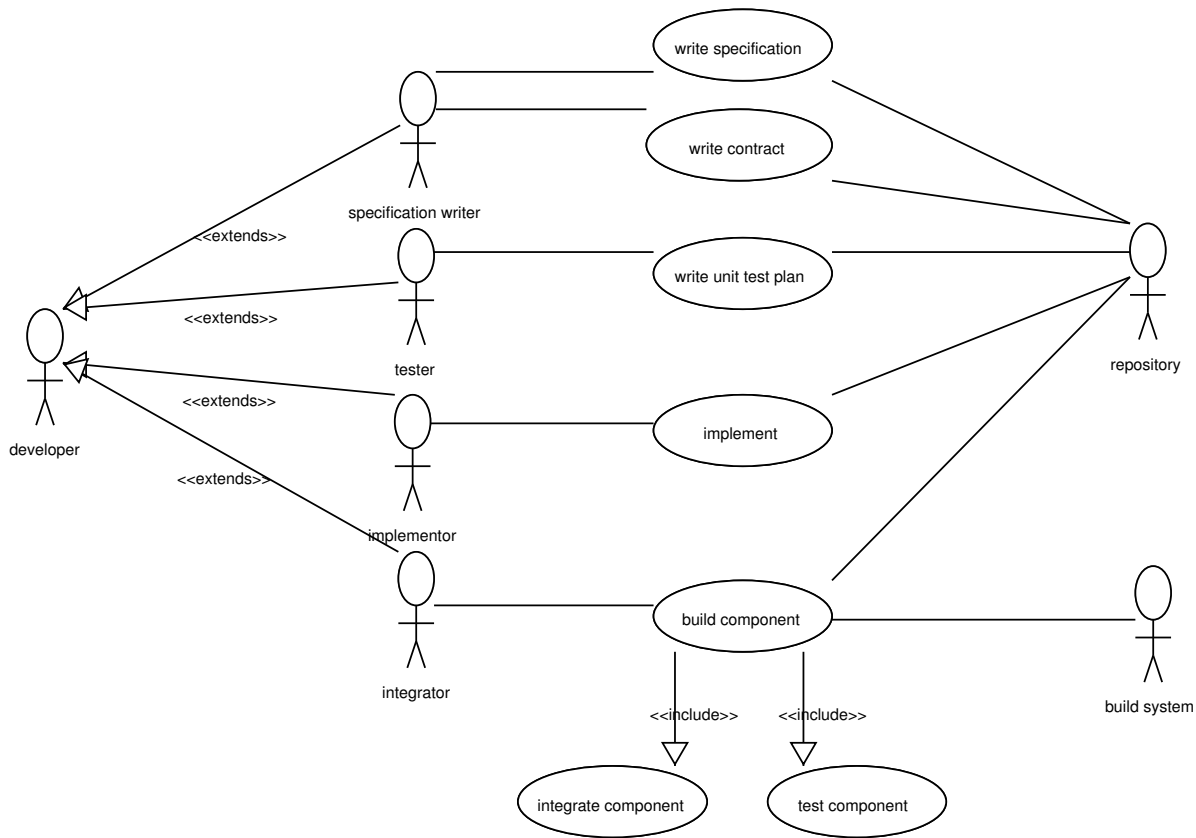
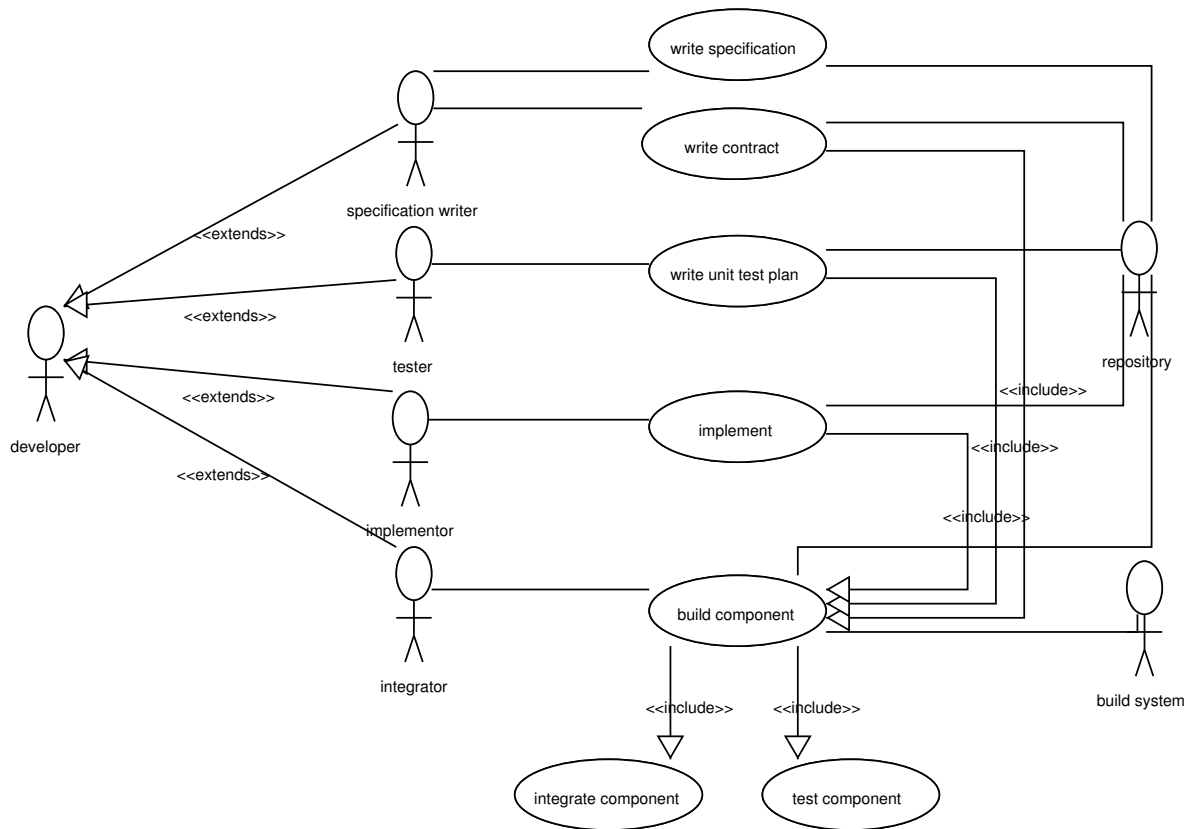


Figure 8.1: Use Case Diagram (*Automated Integration* applied)

<i>Use case name</i>	<b>write specification</b>
<i>Participating actors</i>	invoked by <b>specification writer</b> , communicates with <b>repository</b>
<i>Entry condition</i>	1. software engineering process has reached object design phase
<i>Flow of events</i>	2. <b>specification writer</b> derives the component specification from the artifacts resulting from the preceding phases of the software engineering process 3. <b>specification writer</b> submits specification to the <b>repository</b>
<i>Exit condition</i>	4. component specification in the <b>repository</b> is up to date

Figure 8.2: Use Case Diagram (*Continuous Integration* applied)

<i>Use case name</i>	<b>write contract</b>
<i>Participating actors</i>	invoked by <b>specification writer</b> , communicates with <b>repository</b>
<i>Entry condition</i>	1. component specification in the <b>repository</b> is up to date
<i>Flow of events</i>	2. <b>specification writer</b> derives the component contract from the specification 3. <b>specification writer</b> submits contract to the <b>repository</b> 4. if continuous integration is applied, the <b>build component</b> use case is triggered
<i>Exit condition</i>	5. component contract in the <b>repository</b> is up to date

---

<i>Use case name</i>	<b>write unit test plan</b>
<i>Participating actors</i>	invoked by <b>tester</b> , communicates with <b>repository</b>
<i>Entry condition</i>	1. component specification and contract in the <b>repository</b> are up to date
<i>Flow of events</i>	2. <b>tester</b> creates a test plan by taking into account the specification, the contract and the different test methods described in [BD00] 3. <b>tester</b> submits unit test plan to the <b>repository</b> 4. if continuous integration is applied, the <b>build component</b> use case is triggered
<i>Exit condition</i>	5. unit test plan in the <b>repository</b> is up to date

---



---

<i>Use case name</i>	<b>implement</b>
<i>Participating actors</i>	invoked by <b>implementor</b> , communicates with <b>repository</b>
<i>Entry condition</i>	1. component specification and contract in the <b>repository</b> are up to date
<i>Flow of events</i>	2. <b>implementor</b> implements the component by taking into account the specification and the contract 3. <b>implementor</b> submits the source code to the <b>repository</b> 4. if continuous integration is applied, the <b>build component</b> use case is triggered
<i>Exit condition</i>	5. source code in the <b>repository</b> is up to date

---



---

<i>Use case name</i>	<b>write integration strategy</b>
<i>Participating actors</i>	invoked by <b>integrator</b> , communicates with <b>repository</b>
<i>Entry condition</i>	1. component specification in the <b>repository</b> is up to date
<i>Flow of events</i>	2. <b>integrator</b> implements an integration strategy by taking into account the specification, resulting in a build script 3. <b>integrator</b> submits the build script to the <b>repository</b> 4. if continuous integration is applied, the <b>build component</b> use case is triggered
<i>Exit condition</i>	5. build script in the <b>repository</b> is up to date

---

<i>Use case name</i>	<b>build component</b>
<i>Participating actors</i>	invoked by <b>integrator</b> (if <i>automated integration</i> is applied), communicates with <b>repository</b> and <b>build system</b>
<i>Entry condition</i>	1. can be included from other use cases if <i>continuous integration</i> is applied
<i>Flow of events</i>	2. include <b>compile code</b> (omitted); triggers <b>implement</b> on failure 3. include <b>integrate component</b> ; triggers <b>write specification</b> or <b>integration strategy</b> on failure 4. include <b>test component</b> ; triggers <b>write specification</b> , <b>write contract</b> or <b>implement</b> when positive
<i>Exit condition</i>	5. component is consistent
<i>Use case name</i>	<b>integrate component</b> (included in <b>build component</b> use case)
<i>Participating actors</i>	inherited from <b>build component</b> use case
<i>Entry condition</i>	1. component source code is compiled
<i>Flow of events</i>	2. according to the build script, the component is integrated 3. if any instruction in the script fails, a failure is returned from this use case
<i>Exit condition</i>	4. success is returned
<i>Use case name</i>	<b>test component</b> (included in <b>build component</b> use case)
<i>Participating actors</i>	inherited from <b>build component</b> use case
<i>Entry condition</i>	1. component source code is compiled and component integrated
<i>Flow of events</i>	2. according to the test plan, the test is run (repeat steps 3 and 4) 3. on each call to a component the respective contract is checked 4. on any positive test, positive is returned from this use case
<i>Exit condition</i>	5. negative is returned

## 8.2 Requirements

The following sections describe the nonfunctional requirements that have been identified in the phase of requirement elicitation:

*Platform independence* is an implicit requirement. The problem statement does not make any further assumptions about the components to be tested, thus it should (potentially) be possible

to test on all operating systems and all hardware platforms. To also enable independence from the programming language for the components to be tested is beyond the scope of this thesis.

*Reusability* is the ability of software elements to serve for the construction of many different applications. Because the testbench will be deployed in the change-intensive environment of academic research, it should be possible for each subsystem to be used in a slightly different project. Designing for re-use is the key factor here.

Because the testbench is used in production environments, *performance* also plays a role. It should be possible to test several hundred components in one integration cycle, which should not take longer than 5 to 30 minutes (depending on the size of the project).

There are no *security* requirements for the first release of this project.

Since Ant has been chosen to be used for running the test plan, it is a *pseudo requirement* that several custom Ant tasks control the tests.

# Chapter 9

## Analysis

The activity of *Analysis* results in a correct and complete model of the system.

### 9.1 Constraint Languages

At this point, some statements should be made about the languages that can be used to express the constraints of the contract. The OMG has standardized the *Object Constraint Language* [omg00] for this purpose. OCL is very powerful: It is typed, features local variables (so called let expressions) and can navigate associations. Its weakness is the lack of string operations. To test if a string can represent a binary number is a nuisance, with only substring operations and relational operators.

For the reasons described above, it has been decided to include *regular expressions* together with OCL. They can be used to describe most characteristics of strings in an easy way. Regular expressions are well known and mostly standardized [Fri97].

Still not everything is covered by the languages of testbench. A third language, called *Simple Range Language* has been designed to fill the scarcity of date operations. While it would go beyond the focus of this thesis to implement full date arithmetics, it supports checking if dates and numeric values are within a specified range.

### 9.2 Analysis Model

The analysis model is shown in figure 9.2 on the following page. Compared to the use case model, there are some additional parts: The *runtime system* is responsible for all different runtime environments. It can be seen as a living room for the testee's *bytecode*. A *testbench task* has been

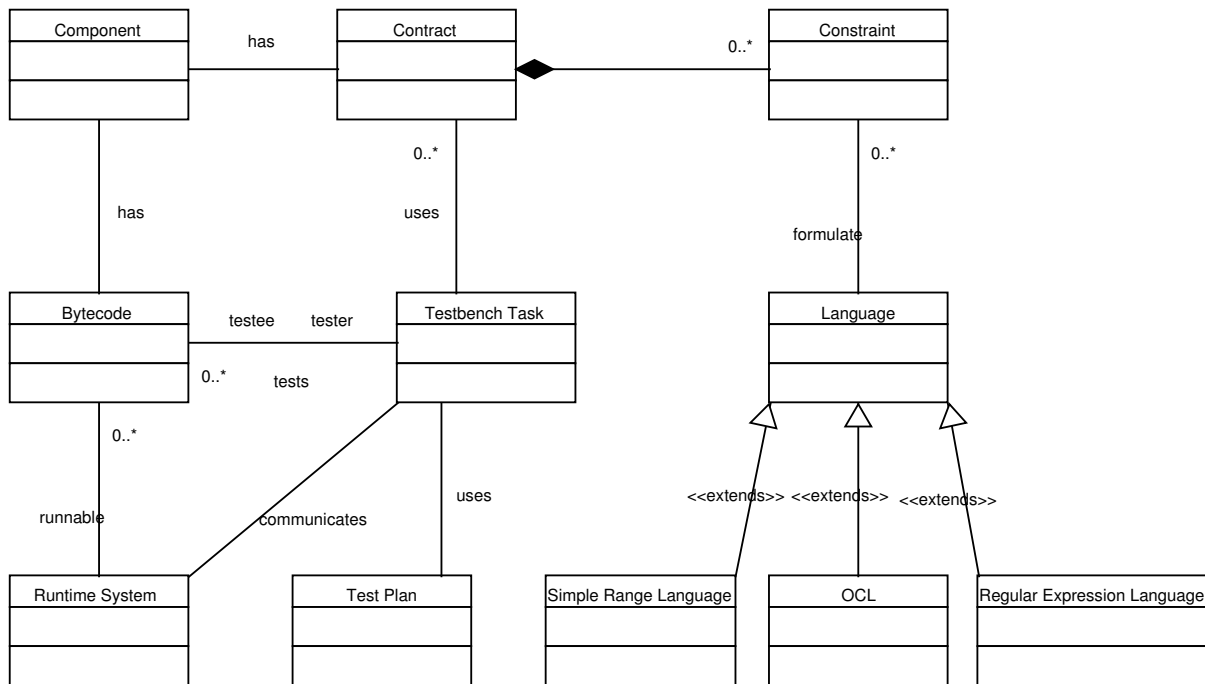


Figure 9.1: The Analysis Model

identified as a central component. It reads the *test plan* and the *contract* and communicates with the runtime. On the right side of the diagram the structure of a contract is shown: It consists of several *constraints* which are formulated by the three different *constraint languages*.

## Chapter 10

# System Design

Decomposing into subsystems is vital for the manageability of a system [BD00]. However, most of these subsystems need functionality of one or more other subsystems. Figure 10 on the next page shows how different functional parts of the testbench were put into different subsystems and how they depend on each other.

The several subsystems are as follows:

**model** is used to build up the object model of the classes to be tested. The classes in this subsystem are tightly coupled, as they have plenty of associations to each other.

**runtime** deals with all different runtime systems that can be used as a base for testing.

**language** contains the languages that can be used to describe the constraints of the contract.

**language.ocl** is a subsystem of language which is purely devoted to the object constraint language.

**action** needs to know all actions that have to be handled in order to test a software.

**ant** realizes the integration with Ant.

**test** contains the test related sources that were used during development.

Part of system design is to put the hardware and software mapping straight. The requirement of platform independence implies that every serious platform is targeted. To fulfil this goal, the Java programming language and its whole universe of standards is well suited.

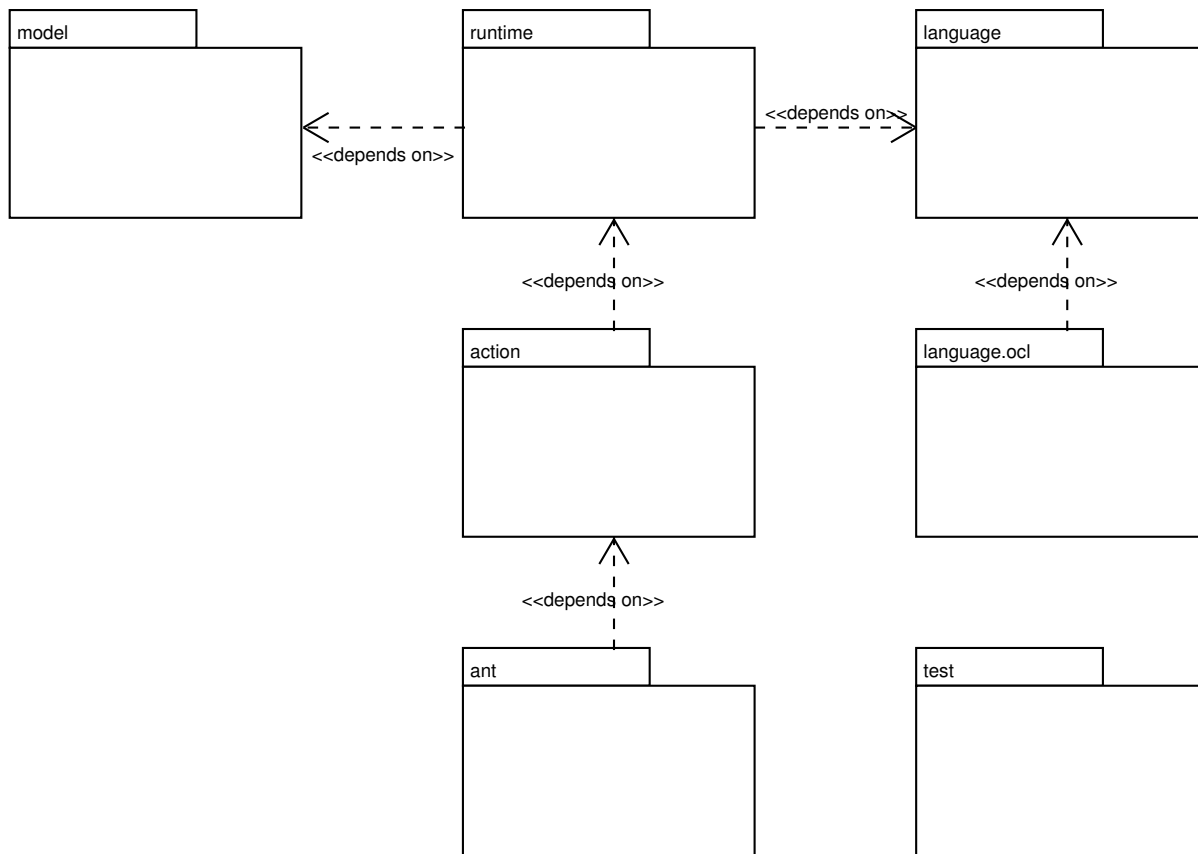


Figure 10.1: Decomposition into Subsystems

# Chapter 11

## Object Design

### 11.1 Model

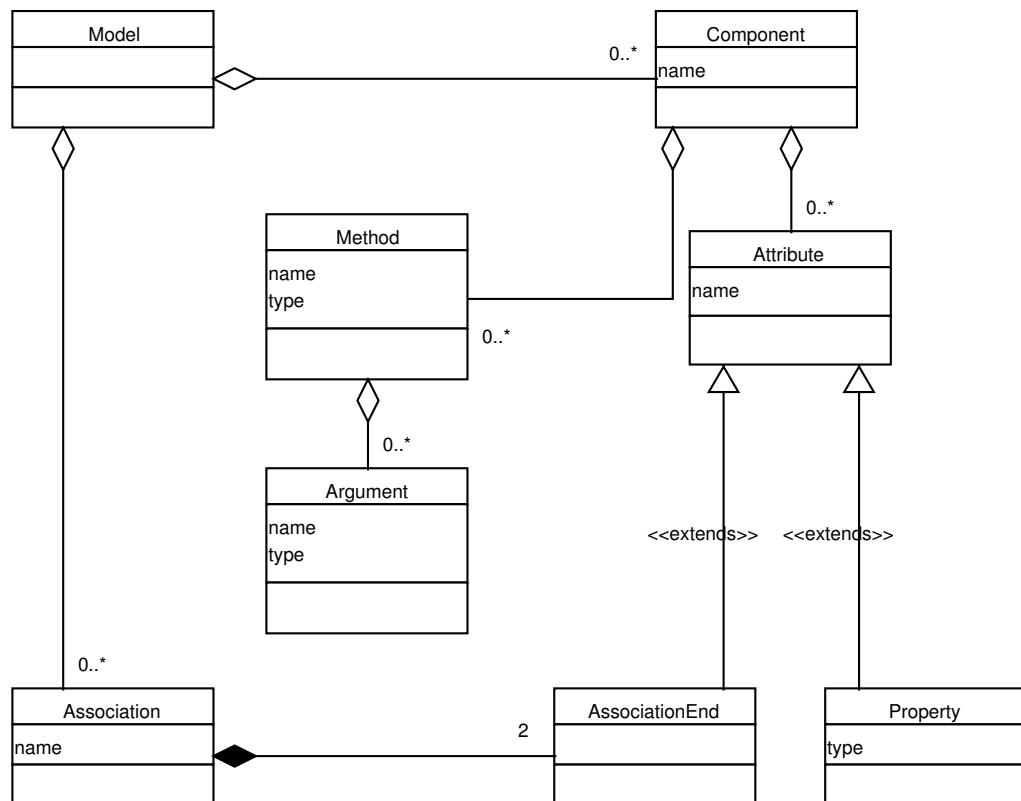


Figure 11.1: The Class Model of the Model Subsystem

All model classes are simple entity classes without any business logic. Their properties are exposed via standard accessor/mutator method pairs. The associations use accessors and mutators in the case of 1:1 relations and a simplified collection interface for 1:n relations. The simplified collection interface consists of an add/remove- pair for mutation and an iterator for access. n:n relations are not supported directly. As in relational database systems, an intermediate map object must be used to realize n:n relations.

Figure 11.1 on the preceding page shows how these interfaces form the has-a relationships. A model consists of components and its associations. Components have methods with their respective arguments. Components also do have attributes, which are a generalization of simple properties and the ends of associations. Each association has got exactly two association ends.

## 11.2 Run-Time

The runtime systems deal with creating instances of components (described by the model), calling methods on them and destroying them when they are no longer needed. In order to allow different runtime systems, a bridge pattern [GHJV95] is used. This way, the abstraction (runtime interface) is decoupled from its implementations, so that the implementations can vary dynamically. Since most runtime systems are already implemented elsewhere (for example Enterprise Java Beans with its home and remote interface [DYK01]), almost all implementations will actually be adapters to the interface of the implementation.

## 11.3 Language

The language system (figure 11.3 on the next page) uses a very similar concept to the runtime systems. Again, the central pattern that is used is a bridge pattern, allowing for pluggable languages. But in this case, some languages are implemented for real (simple range language) and some just use adapters to existing implementations (object constraint language).

### 11.3.1 Object Constraint Language

The Object Constraint Language is complex enough [omg00] that it is placed into a separate subsystem. By using an adapter pattern [GHJV95], it is laid out as a (rather big) adapter to an existing implementation. Writing an interpreter from scratch would need some serious effort and take a lot of time. Fortunately, there is an OCL toolkit available [tud, Fin00]. Using this toolkit, the following steps are needed to build an OCL interpreter:

1. *parsing*. An abstract syntax tree of the OCL statement is created.

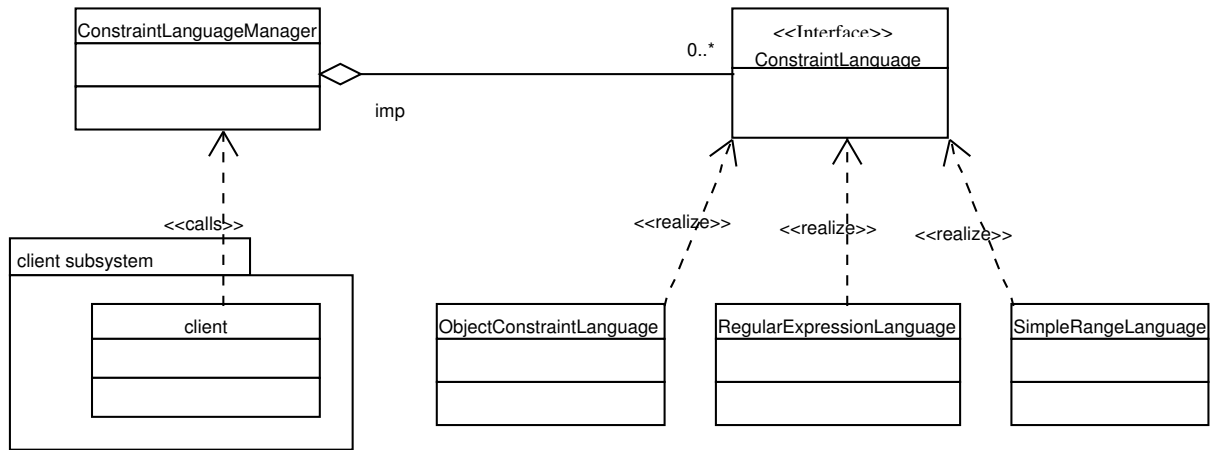


Figure 11.2: The Language Subsystem

2. *semantic analysis*. Simple consistency checks and type checks are done.
3. *normalization*. To pave the way for interpretation, normalization steps can be applied to the abstract syntax tree. This reduces interpreter complexity.
4. *interpretation*. The abstract syntax tree is walked and values of nodes assigned. The value of the root node (*start node* in Dresden OCL jargon) is the boolean value of the whole OCL expression.

[ASU88] discusses these steps in great detail.

## 11.4 Action

The classes contained in the action subsystem represent the different test actions that can be invoked. The command pattern [GHJV95] has been chosen to encapsulate an action as an object. This can be seen in figure 11.4 on the following page. Each action is instantiated, parameterized and then executed. Therefore, different clients can easily invoke actions on the framework. One of these clients is the ant subsystem. Other clients, like GUI frontends or a stand-alone script parser, are possible.

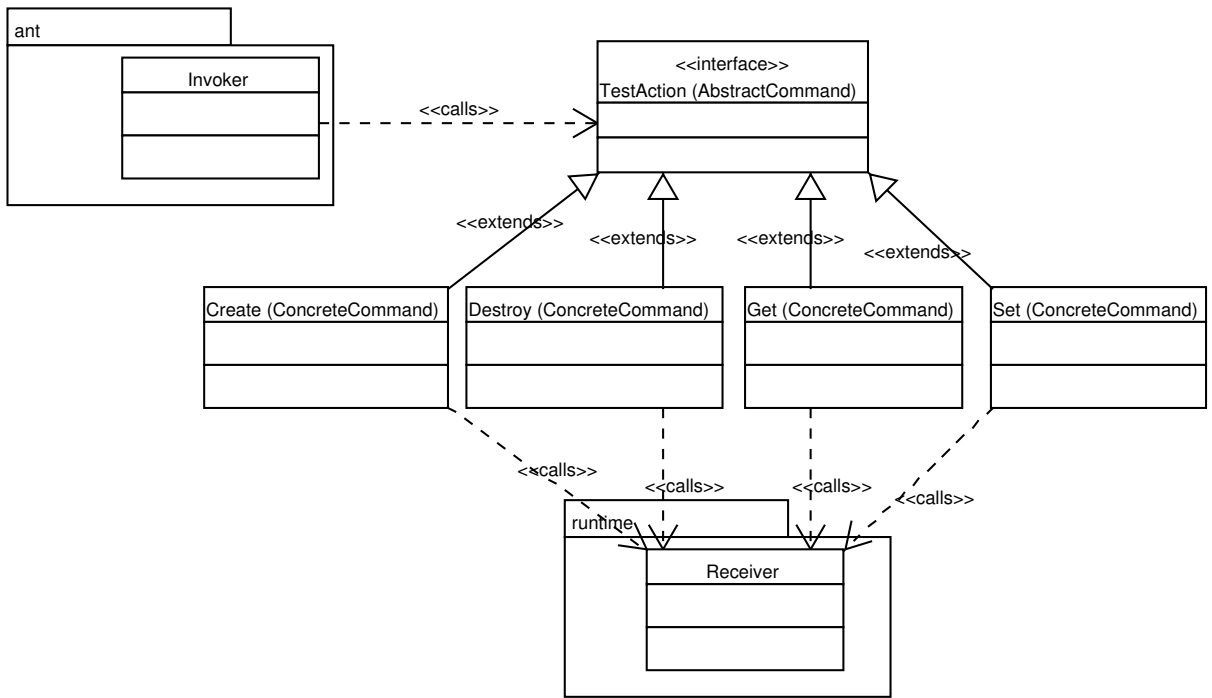


Figure 11.3: Usage of the Command Pattern in the Action Subsystem

## 11.5 Ant

The ant subsystem provides the adapters necessary between the “task-interface” of Ant [jak01] and the action interface exposed by the action subsystem.

## Chapter 12

# Implementation

Implementation deals with putting the abstract object specification, an artifact from the previous phases, into source code.

The question of the programming language being used is a very essential one. It has already been stated during system design that the Java programming language is appropriate for implementation of the testbench.

The Java programming language allows a set of classes to be structured into different packages. These packages are used to represent the different subsystems that were recognized during subsystem decomposition. A straight-forward naming scheme leads to the the following packages: **testbench.model**, **testbench.runtime**, **testbench.language**, **testbench.language.ocl**, **testbench.action**, **testbench.ant**, **testbench.test**.

For the activity of Implementation it was decided to apply the Automated Integration pattern described in section 17.6 on page 69. This allows for faster development, as bugs are easier to locate and conceptual flaws will show up early. As suggested by the pattern's strategy, Ant (version 1.4.1) is being used for automation of the different steps. An excerpt of the build file can be seen in appendix C.2 on page 91. Figure 12 on the next page illustrates the execution of the sample build file.

Testbench does not only use Ant for automating the integration process. Testbench is also integrated into Ant in a way that developers can specify the test cases with a set of Ant targets. This integration into Ant also uses version 1.4.1 of Ant. The process of developing Ant tasks is described by the Ant manual in section “developing with Ant” [jak01]. Basically, for each task the developer has to write a class that extends `org.apache.tools.ant.Task`. Every attribute must be represented with a setter-method. The method `public void execute()` implements the task itself.

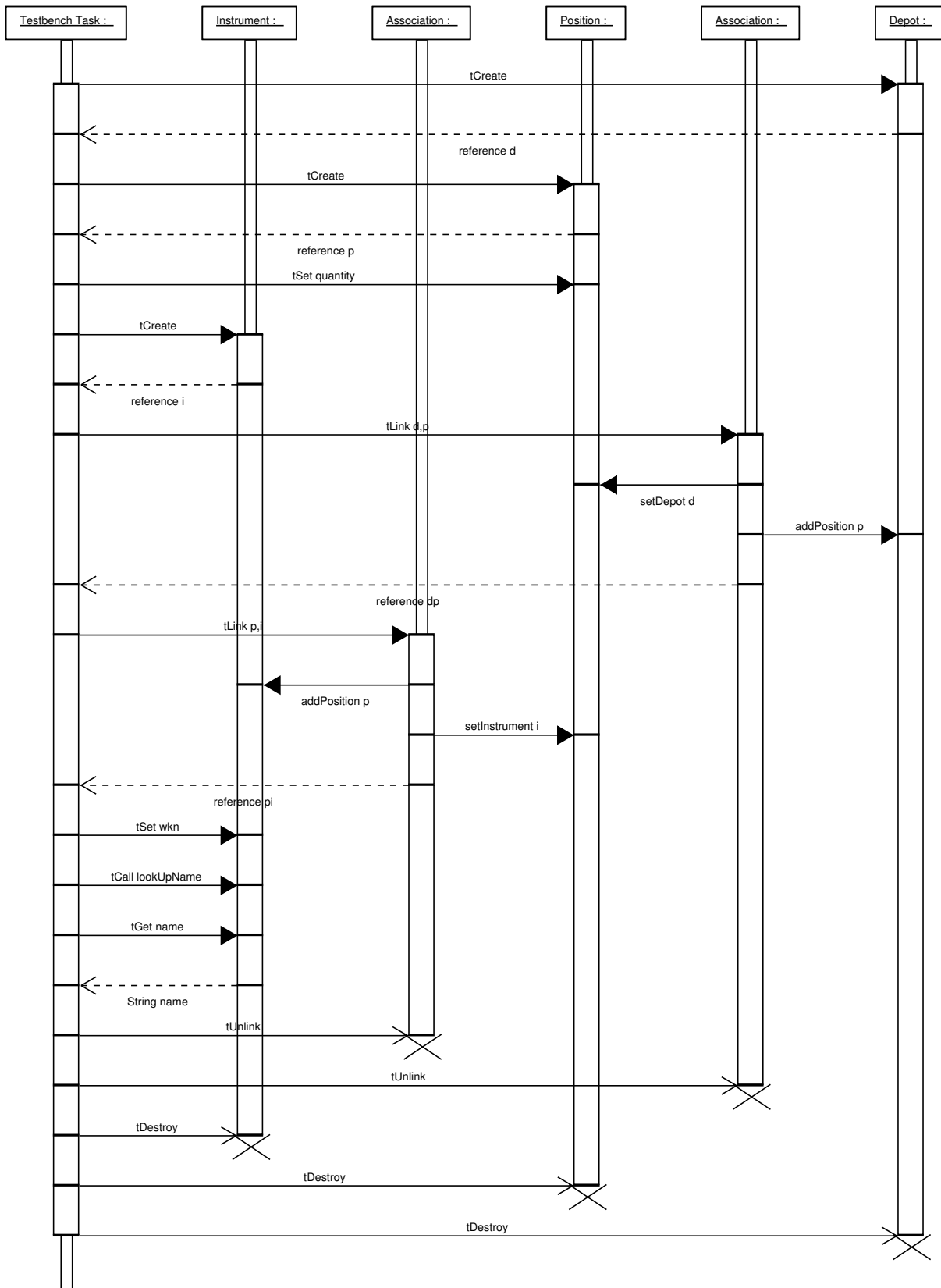


Figure 12.1: Sequence Diagram of the Sample in Appendix C.2

# Chapter 13

## Test

### 13.1 T.R.A.M.P

The testbench will be field-tested in the T.R.A.M.P praktikum [glob], one instance of the GlobalSE (Global Software Engineering) Project [gloa], which is a project that takes place in collaboration with the Carnegie Mellon University (CMU) in Pittsburgh, Pennsylvania.

T.R.A.M.P has got 50 participants who are studying computer science. They form eight specialized, but cooperating teams. The testing team will do the field-test of testbench.



## Part III

# Results

## ABOUT THIS PART

---

This part presents the products and insights that result from the application of the methods described in part II on page 30.

This part explains how the testbench can be employed for various test methods. It presents an extension to usual concept of a component. A byproduct, an interpreter for the Object Constraint Language, is presented.

This part concludes with a catalogue of process patterns for the development of software, another valuable result from the experiences gained during the work on this thesis.

---

## Chapter 14

# Testing with Testbench

### 14.1 Unit Tests

Unit testing finds faults by isolating an individual component [BD00]. Test drivers are used to simulate the parts of the system that call the component to be tested. If the component itself calls any other subsystems or components, test stubs are needed to simulate the behavior of these parts.

Since testbench is designed as a test harness, it serves as a test driver for the class to be tested. The tester is still reliable for building the needed test stubs.

Note that testbench always treats a component as a blackbox and has no access to the components private features. The tester has to decide himself when to use equivalence, boundary, path or state-based testing [BD00].

In order to use testbench for a unit test, the following steps have to be taken:

1. **Writing a model specification.** This is done with an XML file, which consists of **component**, **property**, **method** and **argument** elements. The contract is added with **constraint** elements. The complete DTD (Document Type Definition) is listed in appendix B on page 85.
2. **Writing a test case specification.** Due to the Ant subsystem, this can easily be specified by using Ant tasks. It is advisable to put all test tasks into a separate target called “test”. For an example, see appendix C on page 89. A brief description of the possible tasks follows:

**tContract** loads the model specification into memory. This is mandatory, because testbench needs to know about the model.

**tCreate** creates an instance of a component and generates a reference. The instance can then be used in preceding tasks.

**tDestroy** destroys an instance of a component.

**tLink** links two component instances by using an association. This instantiates an association and tells both ends of the association about the other component by calling the respective mutators.

**tUnlink** breaks a link between two instances.

**tGet** gets a value from a component property.

**tSet** sets a value of a component property.

**tCall** calls a business logic method on an instance. Full constraint checking is applied at this point.

**tInvariant** “manual” checking of a given invariant. This can be used for quick checking instead of defining a constraint in the contract.

3. **Running the test.** This is as easy as invoking `ant test` on the command line. In order to use Ant to its full potential, dependencies between targets can be defined. This way, the tests are implicitly called whenever it is necessary.

## 14.2 Integration Tests

Integration testing focuses on finding faults that have not been detected during unit testing because they only appear when two or more components are “plugged” together (this is called integration). Since it is too time consuming to test any possible combination of components, components are grouped into hierarchical layers. It is desired that each layer only calls components from the layers below.

There are several strategies for integration testing (big bang, bottom-up, top-down and sandwich, [BD00]). Regardless of what strategy the tester chooses, the test can be run by testbench by using the same procedure as for unit testing. However, test cases can get very complicated.

## 14.3 System Tests

In order to test if the complete system complies with the functional or nonfunctional requirements, testbench can only help a little. Constraints do not play a role at this level of testing.

*Stress tests* and *volume tests* can be done by using the control flow of certain Ant tasks, like loops. *timing tests* can use the built-in timer of Ant.

Generally speaking, testbench would need extensions to be an aid for system testing.

## Chapter 15

# The OCL Interpreter

As already addressed in Object Design (chapter 11 on page 45), an interpreter for the Object Constraint Language has been implemented. According to Frank Finger, author of the Dresden OCL Toolkit, this has not been done before (although several people tried).

The interpreter could be isolated as a stand alone product.



## Chapter 16

# Split Personality: Inner and Outer Interface

In the introduction, we discussed the concept of a component. This chapter extends this concept.

As it was already stated in the last chapter, testbench requires test stubs to be available. They have to be written by the tester. This is a bad practice, because this way the stubs are always tested together with the component that is on the bench. We would like to obtain maximum isolation of just one component. This needs a full harness, consisting of a test driver and test stubs.

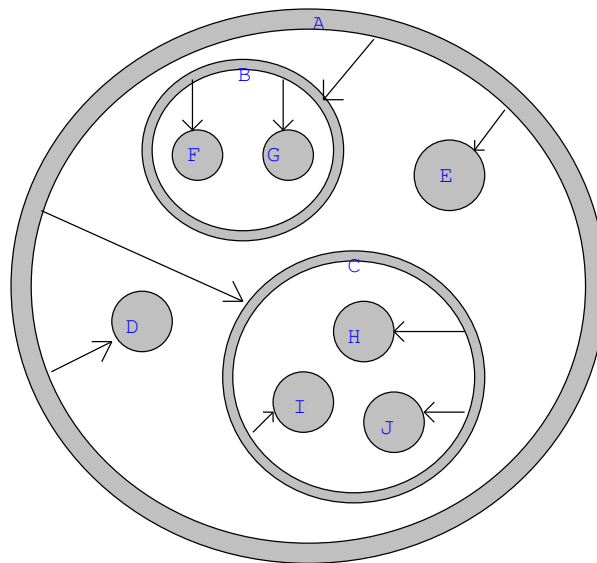


Figure 16.1: Inner and Outer Interfaces

One result of this thesis is the understanding that the classical interface specification, as introduced in chapter I on page 10, is not enough. Components do not only *get called*, they also *do call* other components. Figure 16 on the page before sketches this relationship. Components are displayed as rings, containing other components that get called.

Let us give names to these two interfaces: The *outer interface* is the classical one, which tells the clients how to use the services of the components. The *inner interface* is a specification of what services the component needs from other components. In the figure, the outer interface can be seen as the convex cover of a ring (component) and the inner interface is the concave part.

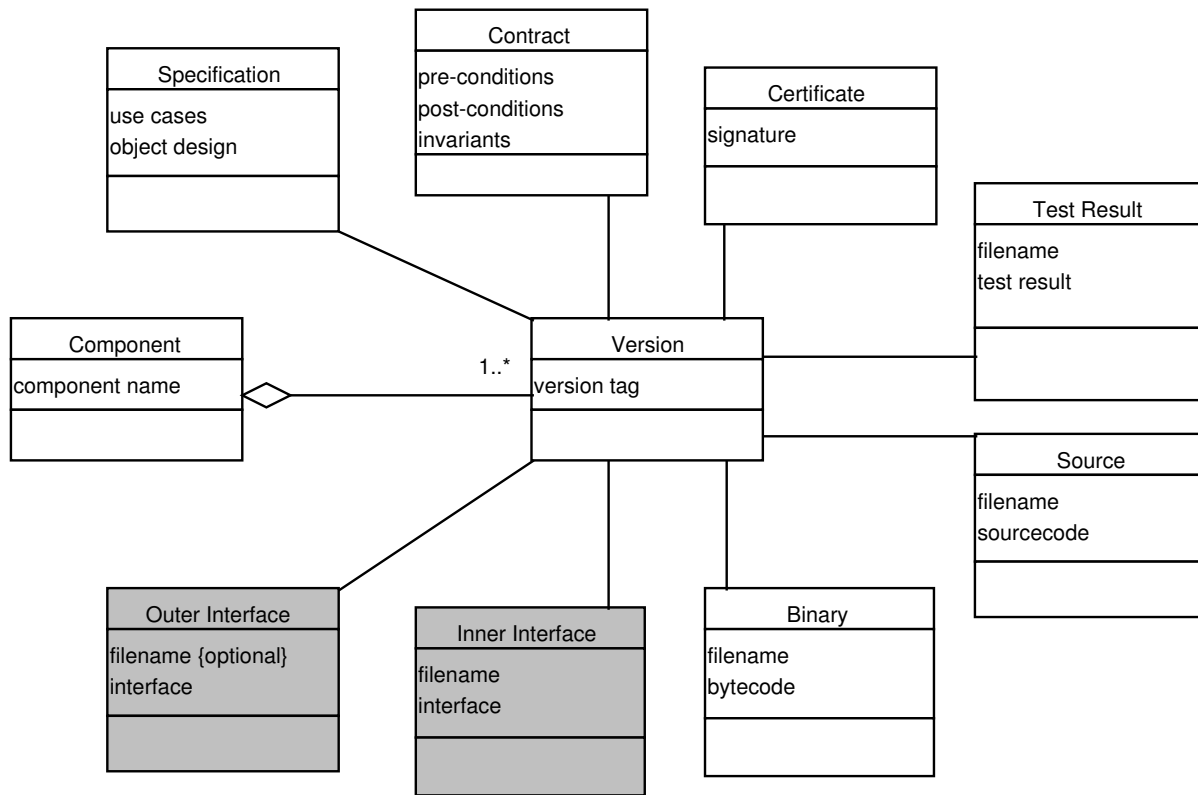


Figure 16.2: The Extension to the Component Anatomy

With the information on both interfaces, a full test harness can be constructed. Therefore we propose to extend the component specification by the inner interface (figure 16).

We spent a good amount of time on searching for publications on this matter. We found [Har] and [OHR00]. Although they mention the dependency of a component on other called components, neither of them went far enough to identify an *inner interface*.

# Chapter 17

## Pattern Catalogue

While developing the testbench, a lot of problems were recurring. We would like to share the solutions for these problems in the form of a pattern language [AIS77]: the *process patterns*. This enables the reader of this thesis to apply these patterns on their own projects, should they encounter the same situation.

The patterns are meant as an aid for developing reliable software systems. Most of the patterns can be used individually, but the biggest benefit is achieved by implementing combinations. Figure 17 shows the relationships between the patterns and how they can be combined. Detailed instructions of how to do this can be found in the individual pattern descriptions.

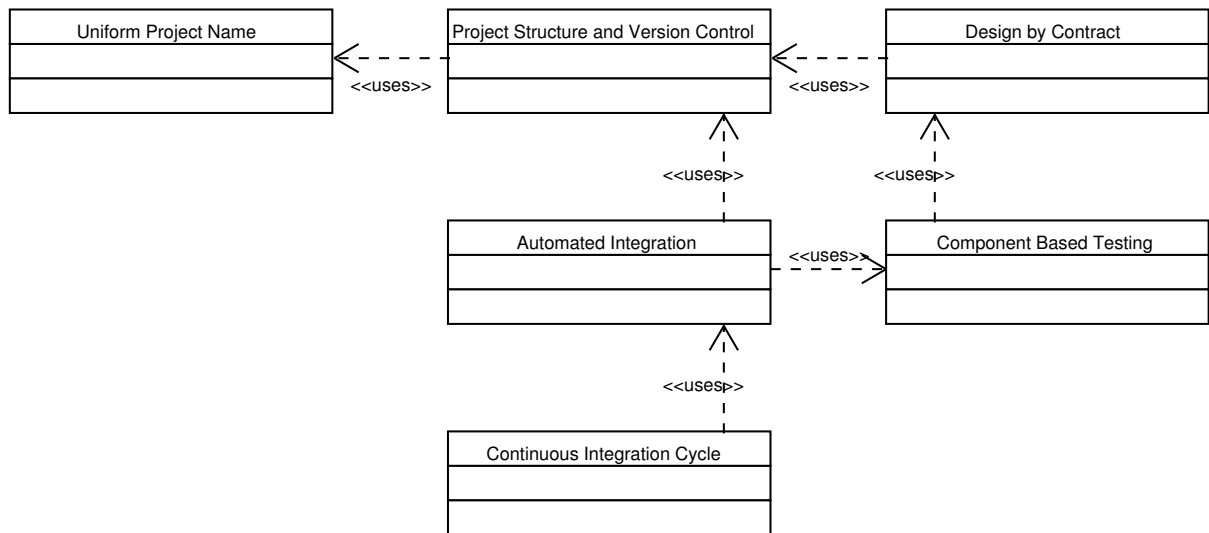


Figure 17.1: Process Patterns Relationships

## 17.1 Pattern Template

This thesis adopts a pattern template that consists of the following sections:

**Context** Sets the environment under which the pattern exists.

**Problem** Describes the design issues faced by the developer.

**Forces** Lists the reasons and motivations that affect the problem and the solution.

**Solution** Describes the solution approach briefly.

**Strategies** Describes one or more different ways a pattern may be implemented.

**Consequences** Here the pattern trade-offs are described. Generally, this section focuses on the results of using a particular pattern or its strategies, and notes the pros and cons that may result from the application of the pattern.

**Related Patterns** This section lists other relevant patterns. For each related pattern, there is a brief description of its relationship to the pattern being described.

## 17.2 Uniform Project Name

### 17.2.1 Context

Communication between parties involved plays a very important part during the whole life cycle of a project. Misunderstandings provoke errors and lead to frustration of team members.

### 17.2.2 Problem

Considering that many parties work on more than one project at a time, every bit of communication needs to be assigned to a specific project (interpersonal communication exempted).

Also, infrastructural resources (filesystems, repositories, and so on) are used for multiple projects in parallel. Information must be tagged with the project it belongs to because otherwise it would mix up with other information.

### 17.2.3 Forces

- In project-specific communication, it needs to be clear what project the information is specific to.
- When using infrastructural resources for project-specific information, again it must be clear what project the information is assigned to.

### 17.2.4 Solution

Define an official, uniform project name for every project and use it.

### 17.2.5 Strategies

**When?** The right point in time to define the project name is just before the first information originates or the first communication occurs.

**How?** Use this name for every communication and every storage of information regarding the project.

**separation of project and product name** Projects tend to shift its focus over time. This may tempt people to defer the definition of a project name until later phases of a project. The preferred strategy is to separate the project name from the final products name.

For example, Microsoft's successor to Windows 2000 had the project name "Whistler". Only a few months before Whistler went gold<sup>1</sup>, marketing decided to use the name "Windows XP" for the final product. Nevertheless, "Whistler" is still used for API's, symbols in configuration descriptors, and so on.

**universal name** Names can be used at numerous occasions. The more universal a name, the more places it can be used. It is a good idea to not use special characters that would limit its use. For example, to use a name as a file or directory name, it must not contain any slashes or an asterisk. To be used as part of a Java package name, it must be free of any dot. The Java Code Conventions [sun99] is a good place to start looking for a pattern of universal names.

### 17.2.6 Consequences

1. Team members can uniquely identify a project by its name.

---

<sup>1</sup> Going gold means being released to CD duplication. This stems from times where CD recordable media had a golden reflective layer and clear polycarbonate. Today's CD recordables use a silver reflective layer and blue/green dye.

2. Developers can use uniform project names as a name element in every language and development tool. Most importantly, it can be used as (part of) file, directory, variable and package names.

### 17.2.7 Related Patterns

**Project Structure and Version Control (section 17.3):** As version control uses infrastructural resources the project need to be uniquely identified. The projects structure is then created in its unique name space.

## 17.3 Project Structure and Version Control

### 17.3.1 Context

During the life-cycle of a project, a multitude of information will accumulate. Software developers write source files and build libraries and executables. Architects write specifications and documentation. Web designers deliver web content and templates. This is just to name a few.

### 17.3.2 Problem

- Without rules and conventions, storage of these bits of information will not follow any order and may get lost, making it very hard to retrieve it. Generally speaking, the entropy of information increases.
- Without any special precautions, new versions of files will overwrite all old versions. This makes it impossible to retrieve any old version whenever it may prove necessary.

### 17.3.3 Forces

In order to work efficiently and reliable, every developer should be able to easily answer the following questions:

- what information does already exist?
- where can the information be found?
- which versions of the information do exist and what is the latest version?

### 17.3.4 Solution

Plan the project structure. Establish a *single source point*[fow, Bec00]. Use a version control tool.

### 17.3.5 Strategies

This strategy describes one very common method of planning the project structure and versioning its files.

1. All files belonging to one project are kept in one single directory tree. It keeps everything for everyone at one designated place. Files are easier to locate and a versioning system can be used to take care of different versions of files in that directory.
2. A distinction is made between source files and build files. Source files are all files that were manually created by the developers or came from external sources. To be more precise, source files cannot be generated from other files. On the other hand, build files can be derived from source files. This makes it easy to separate the wheat from the chaff: Only source files need to be exchanged between developers, versioned and backed up.
3. All source files are checked into some form of repository. From this central point, developers can fetch files, apply changes and feed the changes back into the repository.
4. In order to be able to access old revisions of files, a versioning system assigns versions to files and keeps track of old versions. Developers can roll back to old versions and add tags to a specific combination of files so that everyone knows which revisions form a valid release.

### 17.3.6 Consequences

- After applying this pattern, developers know defined procedures where to find which files. The more projects this pattern is applied to, the easier it will be for developers to switch between projects.
- Most version control tools take care of backing up your work as a welcome side effect, so depending on your backup strategy, there may be no need for a separate backup tool.

### 17.3.7 Related Patterns

**Uniform Project Name (section 17.2 on page 62):** Before applying this pattern, it is very important to choose a Uniform Project Name. This is because repositories need a module or project name assigned to the files checked in.

**Automated Integration (section 17.6 on page 69):** The Automated Integration pattern needs this patterns project structure as a basis.

**Design By Contract (section 17.4):** Design By Contract uses this pattern, because contracts have their home in the project structure in the same way source codes do.

### 17.3.8 Samples

As an example, the directory structure of this thesis (which employs this pattern) is described in appendix A on page 83.

## 17.4 Design By Contract

### 17.4.1 Context

While Design by Contract plays a central role in our pattern structure, it is not a pattern per se. Detailed discussions can be found in chapter 2 on page 17 and [Mey97].

### 17.4.2 Problem

In order to decide if a system is able to perform a job according to the specification, a systematic approach is needed.

### 17.4.3 Forces

- All parties involved in the production of a software system want to know if the system is able to perform its job according to the specification.
- Developers need instruments to describe the semantics of components.
- Developers need instruments to isolate bugs in code.

### 17.4.4 Solution

The solution is delivered by Design by Contract. Each component gets a contract, which defines the relationship between the component and the client (user of the component). It expresses each party's rights and obligations.

### 17.4.5 Strategies

Contracts usually come in the form of constraints. There are three types of constraints:

**Precondition** is evaluated just before the invocation of a component. It can be used to describe the input data of a method (initial state of the component and arguments).

**Postcondition** is evaluated after the invocation of a component. It can be used to describe the output data of a method (altered state and return value).

**Invariant** should always evaluate to true.

All contracts should be created in the Object Design phase.

### 17.4.6 Consequences

- The contract has to be obeyed by the implementation.
- The contract can be used as a foundation for writing test cases.
- The contract also serves as documentation for the component.

### 17.4.7 Related Patterns

**Project Structure and Version Control (section 17.3 on page 64):** This pattern needs a project structure to store the contracts.

**Component-Based Testing (section 17.5):** As stated in this pattern, the contract can be used for writing test cases.

## 17.5 Component-Based Testing

### 17.5.1 Context

Let us assume a project which is about to enter its implementation phase. In the prior phase (object design), an object model and its contracts have been specified. Just when the first code is implemented, testers can start to test if the appropriate components observe their respective contract.

### 17.5.2 Problem

The contract testing should be done as often as possible. Ideally, each change to a source file should trigger a contract test.

### 17.5.3 Forces

- Developers want to make sure that their implemented code observes the contract.
- Developers want to find deviations from the contract as early as possible.

### 17.5.4 Solution

Add automation to the component tests.

### 17.5.5 Strategies

One half of the work for applying this strategy has already been done by using the Design By Contract pattern first: Contracts have been defined. The last piece of the puzzle is a testing tool. In the introduction, several testing tools have been mentioned. It is favourable to choose one tool that accepts contracts in the form in which they have been specified (e.g. the Object Constraint Language [omg00]).

### 17.5.6 Consequences

Adoption of this pattern will help to

- find bugs in the code early.
- find conceptual flaws early.

It will also mean that

- developers will have to write the component contracts in the object design phase.
- developers will have to execute the tests on a regular basis (the Automated Integration pattern helps with that).

### 17.5.7 Related Patterns

**Design By Contract (section 17.4 on page 66):** It is clear that component contracts have to be defined before applying this pattern.

**Automated Integration (section 17.6):** Evidently, automated component tests can be added to the automated integration process.

## 17.6 Automated Integration

### 17.6.1 Context

After components have been specified and implemented, they have to be put together in order to form the subsystems. Subsystems themselves have to be connected to each other, according to the system design document. This process is called *integration* and the role that is responsible for integration is called *integrator*.

### 17.6.2 Problem

The process of integration can hold many surprises for the unlucky integrators. Many conceptual flaws show up only when the interaction of components and the cooperation of subsystems is tested. As a consequence, integration should be tested as early as possible, and as often as possible.

### 17.6.3 Forces

- Problems with integration should show up as early as possible.

### 17.6.4 Solution

Automate the process of integration.

### 17.6.5 Strategies

The following strategy can be used to apply this pattern.

1. The first step is the decision about a build tool that aids in automating processes. For easy processes batch files are appropriate, and sometimes tools that detect changes to source files and manage changes are favoured. The classic “make” and the cleaner, cross-platform “Ant” are well known examples for the last category.
2. Then, all subprocesses that are needed to build the whole integration process have to be written into the source file for the build tool.
3. Tests for checking the success of the integration are added to the integration process, too.
4. Each time one of the components has changed, the integration process can be executed.

### 17.6.6 Consequences

- Conceptual flaws will show up early. This leaves more room for changes of specifications that resulted from system and object design.
- Bugs will be easier to locate: most likely they occur in the part of the code that has been changed since the last integration.
- The integration process grows with the project itself.
- The process of integration itself is tested, and when the “traditional integrations” (client acceptance testing, product delivery) come, every integrator will be familiar with the process.

### 17.6.7 Related Patterns

**Project Structure and Version Control (section 17.3 on page 64):** This pattern needs a project structure as a basis for automation.

**Component-Based Testing (section 17.5 on page 67):** As part of the build process, components can be unit tested using the Component-Based Testing pattern.

**Continuous Integration (section 17.7 on the facing page):** The execution of the integration process can be automated, too. This leads to a continuous process, an extension to this pattern.

## 17.7 Continuous Integration

### 17.7.1 Context

This pattern is an extension to the Automated Integration pattern and is described in [Bec00] and [fow]. After every set of changes to the source files of a project, developers execute the integration process.

### 17.7.2 Problem

When several developers are working in parallel, problems can occur when one or more shared resource is accessed by multiple processes at the same time.

### 17.7.3 Forces

- Only one integration process should run at a time.
- Every developer should be informed about the progress and results of the integration process.

### 17.7.4 Solution

Assign control of the integration process to one single instance.

### 17.7.5 Strategy

[fow] suggests the following strategy:

1. Choose a machine to run the master builds. This platform will be the reference for every member of the team. Everything has to compile on this machine and integrate on this machine.
2. Set up a build daemon. This is a process that waits for changes to the repository. If there is any new code, it executes the integration process. At the end of the integration, the build daemon sends an e-mail to all developers that had newly checked in code with that integration.

### 17.7.6 Consequences

- The integration process is more disciplined and thus projects can be controlled easier.
- Integration testing cannot be neglected by developers: as soon as code is committed, the integration process will start and report any bugs.
- This pattern supports the feeling for collective responsibility. Ideally, no developer leaves the house before the integration with his or her last changes succeeds.

### 17.7.7 Related Patterns

**Automated Integration (section 17.6 on page 69):** As described in the pattern, it executes the integration process.

## Part IV

# Discussion

## ABOUT THIS PART

---

This part discusses the results that were presented in part III on page 54.

It weighs up the pros and cons of the different techniques of checking constraints.

Concluding, future directions are given. This includes enhancement ideas to testbench and thoughts about the future of the process pattern catalogue (chapter 17 on page 61) and the OCL interpreter.

---

## Chapter 18

# Pros and Cons

### 18.1 Code Instrumentation or External Checking?

In the previous parts, we talked about two different methods of constraint checking.

The first alternative is *checking from inside of the methods*. iContract and the OCL Injector takes this route. The code is instrumented with statements that represent the constraint. Should a constraint evaluate to false, the mechanisms of the respective programming language have to be used to report the failure to the caller (e.g. exception in Java). Constraints are able to access all features of the component - regardless of its access modifiers.

The other alternative is *checking from outside of the methods*. Testbench implements this alternative. The precondition is evaluated at an external place (e.g. the caller), then the method is called. After the control flow returns, the postcondition is evaluated. Constraints can only use the public interface to access any features of the component in this case. The advantage lies in this method's flexibility: The type of interface to the component can easily be exchanged. It does not have to be a class level interface, it could also be a CORBA[omg01] interface or an internet protocol like SMTP. Support for other types of interfaces can be added to the runtime subsystem of testbench.



## Chapter 19

# Putting Constraints into Code?

### 19.1 Roles

Another interesting question: Is it a good thing to put constraints into source code (e.g. as JavaDoc comments)? When researching about it, the question of who writes the constraints and who writes the rest of the source code has to be answered first. Constraints form the component contract, which means that the specification writer who creates the contract also writes the constraints. Source code is written by an implementor.

### 19.2 Implementation Logic

Obviously two different actors (potentially two different developers) are working on the same artifact. In reality both of these roles can be assigned to the same developer. This is not a good practise for the following reason: the implementation logic will exactly follow the logic of the constraints. If the constraints are used as input for a test, the tests will not be able to detect any deviations in the code.

Additionally, separation supports efficiency because it adds a degree of parallelism to the software development process.



## Chapter 20

# Future Directions

### 20.1 Test Stubs

As already mentioned in the results section, testbench does not yet provide any test stubs. This is still the responsibility of the tester. A very useful extension to testbench would be to use the insights from chapter 16 on page 59 to assist the tester in building the test stubs. They could even be automatically derived from the contracts of the components referenced in the inner interface.

The last section rises an interesting question: Is it possible to take the responsibility for both test drivers and stubs out of the hands of developers? As a side effect, sources for stubs would disappear from the repository, leaving only sources for the final deliverable behind. The answer to this question could be the aim for follow-up research.

### 20.2 Enhanced Control Flow

In order to use testbench for performance testing or stress testing, more control flow would be advisable. For example, loop statements could repeat certain tests hundreds of times on the same instance of a component.

A straightforward way to implement control flow statements would be additional Ant tasks. The advantage of this approach lies in reusability: The tasks could be used for other purposes, too. They could even be part of a future Ant distribution.

### **20.3 Extending the Process Pattern Catalogue**

The process pattern catalogue is just a beginning. There is a richness of various processes that could be presented in the pattern language. The catalogue could be published and advanced on a stand alone basis.

### **20.4 Separate OCL Interpreter Project**

The interpreter for the Object Constraint Language is another subsystem that could be separated from the rest of the project in order to form a new project. More generic interfaces could extend the re-usability and flexibility of the interpreter.

Part V

Appendix



# Appendix A

## Project Structure and Version Control

### A.1 An Example

This section documents the project structure that is used for this thesis. Source files (all files in the `/thesis/src` directory and `/thesis/build.xml`) are committed into CVS, everything else can be built from the source files.

`/thesis/` This is the project's root directory.

`/thesis/src/` Contains any hand-crafted files and auxiliary files, that are needed to automatically build the whole project. And only those. All other files, to be exact, those who can be generated out of the source files (classes, javadocs, distribution packs, etc) will be found in `/build`.

The `/src` directory contains several sub-directories, which are described as follows.

`/thesis/src/classes/` Java source files go here. Since packages have been used to differentiate between the subsystems, a directory structure representing the package structure is created under this directory.

`/thesis/src/etc/` Generally, this directory contains additional support files, like component descriptors, servlet descriptors, en/decryption keyrings, resource bundles and so on. For this thesis, this directory only contains a few Java property style configuration files.

`/thesis/src/lib/` All libraries can be found here, in the form of jar-files. Examples are database drivers, external utility classes and so on.

`/thesis/src/tests` These are the test cases that were used during regression testing of the thesis.

**/thesis/src/tex** is the home of the thesis paper itself. All  $\text{\LaTeX}$  source files and imported graphics are contained in this directory and its subdirectories.

**/thesis/build/** All files that are created by the automated integration process will be created here.

**/thesis/build.xml** This file contains the instruction of how to automatically accomplish certain tasks that allow automation, like compiling code into classes, packaging up distribution archives, invoke tests, and so on.

It consists of several targets that can be invoked using Ant.

Note that this file is the essential link between the contents of the source and the build directories, as most of the actions described by it take files from the source and put the output in build.

# Appendix B

## Specification of the Model

### B.1 Document Type Definition

```
<?xml version="1.0" ?>

<!ELEMENT contract (component*)>
<!ATTLIST contract
    name CDATA #REQUIRED
>
<!ELEMENT component (property*,associationend*,method*)>
<!ATTLIST component
    name CDATA #REQUIRED
    class CDATA #IMPLIED
    runtime CDATA #IMPLIED
>
<!ELEMENT property EMPTY>
<!ATTLIST property
    name CDATA #REQUIRED
    type CDATA #REQUIRED
>
<!ELEMENT associationend EMPTY>
<!ATTLIST associationend
    association CDATA #IMPLIED
    name CDATA #REQUIRED
    target CDATA #REQUIRED
    multiplicity CDATA #REQUIRED
>
<!ELEMENT method (argument*,constraint*)>
```

```

<!ATTLIST method
    type CDATA #IMPLIED
    name CDATA #REQUIRED
>
<!ELEMENT argument EMPTY>
<!ATTLIST argument
    type CDATA #REQUIRED
    name CDATA #IMPLIED
    property CDATA #IMPLIED
>
<!ELEMENT constraint (#PCDATA)>
<!ATTLIST constraint
    property CDATA #IMPLIED
    type CDATA #REQUIRED
    language CDATA #IMPLIED
>

```

## B.2 Sample Document

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE component SYSTEM "system-contract.dtd">

<contract name="financial service">

    <component name="Depot" class="testbench.test.Depot" runtime="bean">

        <property type="string" name="name"/>
        <property type="int" name="numberOfPositions"/>

        <associationend
            association="dp"
            name="positions"
            target="Position"
            multiplicity="n"
        />

    </component>

    <component name="Position" class="testbench.test.Position" runtime="bean">

```

```
<property type="int" name="quantity"/>

<associationend
  association="dp"
  name="depot"
  target="Depot"
  multiplicity="1"
/>
<associationend
  association="pi"
  name="instrument"
  target="Instrument"
  multiplicity="1"
/>

</component>

<component name="Instrument" class="testbench.test.Instrument" runtime="bean">

  <property type="string" name="name"/>
  <property type="string" name="wkn"/>
  <property type="float" name="price"/>

  <method name="lookUpName">
    <constraint property="wkn" type="pre" language="regexp">
      [0-9]*6
    </constraint>
    <constraint property="name" type="post" language="regexp">
      [a-zA-Z0-9]*
    </constraint>
  </method>

  <associationend
    association="pi"
    name="positions"
    target="Position"
    multiplicity="n"
  />

</component>

</contract>
```



# Appendix C

## Specification of a Test Case

### C.1 Document Type Definition

This chapter contains an excerpt from the DTD (Document Type Definition) of the test case specification. As this specification is embedded into the Ant build file, the test case DTD is kind of embedded into the build file DTD, too.

```
<?xml version="1.0" ?>

<!ENTITY % boolean "(true|false|on|off|yes|no)">
<!ENTITY % tasks "[...] | tContract | tCreate | tDestroy | tLink | tUnlink | tSet |
                  tGet | tInvariant | tCall">
<!ENTITY % types "fileset | patternset | filterset | description | filelist |
                  path | mapper">

<!ELEMENT project (target | property | taskdef | %types;)*>
<!ATTLIST project
    name      CDATA #REQUIRED
    default   CDATA #REQUIRED
    basedir   CDATA #IMPLIED>

<!ELEMENT target (%tasks; | %types;)*>
<!ATTLIST target
    id         ID      #IMPLIED
    name       CDATA  #REQUIRED
    if         CDATA  #IMPLIED
    unless     CDATA  #IMPLIED
    depends    CDATA  #IMPLIED
```

```
description CDATA #IMPLIED>
```

```
[...]
```

```
<!ELEMENT tContract EMPTY>
```

```
<!ATTLIST tContract
```

```
id ID #IMPLIED
```

```
taskname CDATA #IMPLIED
```

```
file CDATA #IMPLIED
```

```
description CDATA #IMPLIED>
```

```
<!ELEMENT tCreate EMPTY>
```

```
<!ATTLIST tCreate
```

```
id ID #IMPLIED
```

```
component CDATA #IMPLIED
```

```
taskname CDATA #IMPLIED
```

```
ref CDATA #IMPLIED
```

```
description CDATA #IMPLIED>
```

```
<!ELEMENT tDestroy EMPTY>
```

```
<!ATTLIST tDestroy
```

```
id ID #IMPLIED
```

```
taskname CDATA #IMPLIED
```

```
ref CDATA #IMPLIED
```

```
description CDATA #IMPLIED>
```

```
<!ELEMENT tLink EMPTY>
```

```
<!ATTLIST tLink
```

```
id ID #IMPLIED
```

```
association CDATA #IMPLIED
```

```
ref1 CDATA #IMPLIED
```

```
ref2 CDATA #IMPLIED
```

```
taskname CDATA #IMPLIED
```

```
description CDATA #IMPLIED>
```

```
<!ELEMENT tUnlink EMPTY>
```

```
<!ATTLIST tUnlink
```

```
id ID #IMPLIED
```

```
taskname CDATA #IMPLIED
```

```
description CDATA #IMPLIED>
```

```
<!ELEMENT tSet EMPTY>
```

```
<!ATTLIST tSet
    id ID #IMPLIED
    taskname CDATA #IMPLIED
    ref CDATA #IMPLIED
    description CDATA #IMPLIED
    value CDATA #IMPLIED
    property CDATA #IMPLIED>

<!ELEMENT tGet (expect)*>
<!ATTLIST tGet
    id ID #IMPLIED
    taskname CDATA #IMPLIED
    ref CDATA #IMPLIED
    description CDATA #IMPLIED
    property CDATA #IMPLIED>

<!ELEMENT tInvariant (expect)*>
<!ATTLIST tInvariant
    id ID #IMPLIED
    taskname CDATA #IMPLIED
    context CDATA #IMPLIED
    description CDATA #IMPLIED>

<!ELEMENT tCall EMPTY>
<!ATTLIST tCall
    id ID #IMPLIED
    taskname CDATA #IMPLIED
    ref CDATA #IMPLIED
    method CDATA #IMPLIED
    description CDATA #IMPLIED>
```

## C.2 Sample Document

This is an example of a build file, which also contains a test case specification (again, only the relevant parts):

```
<?xml version="1.0" ?>

<project name="thesis" basedir="." default="compile">
```

```
<target name="all" depends="clean,compile,test,javadoc,tex"/>

<target name="clean">
  <delete dir="build"/>
</target>

<target name="prepare">
  <mkdir dir="build/classes"/>
  <mkdir dir="build/tex"/>
  <mkdir dir="build/doc"/>
</target>

<target name="compile" depends="prepare">
  <javac
    srcdir="src/classes"
    destdir="build/classes"
    includes="**/*.java"
    debug="on"
    optimize="off"
  >
    <classpath><fileset dir="src/lib"/></classpath>
  </javac>
</target>

<target name="javadoc" depends="prepare">
  <javadoc
    sourcepath="src/classes"
    destdir="build/doc"
    packagenames="thesis.*,tudresden.*"
    version="true"
    author="true"
  >
    <classpath><fileset dir="src/lib"/></classpath>
  </javadoc>
</target>

[...]

<target name="taskdef">
  <taskdef file="src/etc/ant-taskdefs.properties" classpath="build/classes"/>
</target>
```

```
<target name="test" depends="taskdef">
  <tContract file="src/tests/depot-contract.xml"/>
  <tCreate ref="d" component="Depot"/>
  <tCreate ref="p" component="Position"/>
  <tSet ref="p" property="quantity" value="2"/>
  <tCreate ref="i" component="Instrument"/>
  <tSet ref="i" property="wkn" value="723610"/>
  <tLink id="dp1" association="dp" ref1="d" ref2="p"/>
  <tLink id="pi1" association="pi" ref1="p" ref2="i"/>
  <tCall ref="i" method="lookUpName"/>
  <tGet ref="i" property="name"/>
  <tUnlink id="pi1"/>
  <tUnlink id="dp1"/>
  <tDestroy ref="i"/>
  <tDestroy ref="p"/>
  <tDestroy ref="d"/>
</target>

</project>
```

Figure 12 on page 50 visualizes the **test** target as a sequence diagram.



## Appendix D

# Bibliography

- [AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.
- [arg] *ArgoUML*  
<http://www.argouml.org/>.
- [ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley Longman Limited Harlow, England Reading, Massachusetts Menlo Park, California New York, 1988.
- [BD00] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*. Prentice Hall, 2000.
- [Bec00] Kent Beck. *Extreme Programming Explained. Embrace Change*. Addison Wesley, 2000.
- [BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.
- [DYK01] Linda G. DeMichiel, L. Umit Yalcinalp, and Sanjeev Krishnan. *Enterprise JavaBeans Specification, Version 2.0*  
<http://java.sun.com/products/ejb/docs.html>. Sun Microsystems, August 2001.
- [Fin00] Frank Finger. *Design and Implementation of a Modular OCL Compiler*  
<http://www-st.inf.tu-dresden.de/ocl/ff3/diplom.pdf>, March 2000.
- [fow] *Continuous Integration*  
<http://www.martinfowler.com/articles/continuousIntegration.html>.
- [Fri97] Jeffrey E. F. Friedl. *Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools*. O'Reilly and Associates, 1997.

- [GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides, editors. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman Limited Harlow, England Reading, Massachusetts Menlo Park, California New York, 1995.
- [gloa] *Global Software Engineering Organisation*  
<http://www.globalse.org/>.
- [glob] *T.R.A.M.P: Travelling Repair And Maintenance Platform*  
<http://tramp.globalse.org/>.
- [GMJ91] Carlo Ghezzi, Dino Mandrioli, and Mehdi Jazayeri. *Fundamentals of Software Engineering*. Prentice Hall, 1991.
- [Har] Mary Jean Harrold. Using component metadata to support the regression testing of component-based software.
- [ico] *iContract*  
<http://www.reliable-systems.com/tools/iContract/iContract.htm>.
- [jak01] Apache Software Foundation. *The Jakarta Project - Ant*  
<http://jakarta.apache.org/ant/>, 2001.
- [jun] *JUnit*  
<http://www.junit.org>.
- [Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 1997.
- [Mun93] B.P. Munch. *Versioning in Software Engineering Database: the Change Oriented Way (PhD dissertation)*, 1993.
- [OHR00] A. Orso, M. Harrold, and D. Rosenblum. Component metadata for software engineering tasks, 2000.
- [omg00] Object Management Group. *Unified Modeling Language Specification Version 1.3*  
<http://www.omg.org/technology/documents/formal/>, March 2000.
- [omg01] Object Management Group. *Common Object Request Broker Architecture Specification Version 2.5*  
<http://www.corba.org>, September 2001.
- [RY99] Eric S. Raymond and Bob Young. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly and Associates, Oct 1999.
- [suna] *Javadoc Tool Home Page*  
<http://java.sun.com/j2se/javadoc/>.

- [sunb] *SUN Java Development Kit*  
<http://java.sun.com/j2se/>.
- [sun99] Sun Microsystems. *Code Conventions for the Java<sup>TM</sup> Programming Language*  
<http://java.sun.com/docs/codeconv/>, 1999.
- [Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*.  
Addison-Wesley, Jan 1998.
- [tud] *Dresden OCL Toolkit*  
<http://dresden-ocl.sourceforge.net/index.html>.
- [use] *U.S.E: An UML-based Specification Environment*  
<http://www.db.informatik.uni-bremen.de/projects/USE/>.



# Appendix E

## Index

- actor, 35
- analysis model, 41
- Ant, 32
- ArgoUML, 31
- assertion, 28
  
- bibliography, 95–97
- binary, 15
- build process
  - clean, 19
  - incremental, 19
  
- certificate, 17
- component, 15
  - bytecode, 41
  - change, 23–24
  - contract, 42
  - populating a, 19
- constraint, 42
- constraint language, 42
- control flow, 79
- correctness, 15
  - confidence, 16
  
- Design by Contract, 17, 66–67
- Dresden OCL
  - Injector, 27
  - Toolkit, 46
  
- extreme programming, 28
  
- faults
  - avoidance, 11
  - detection, 11
  - tolerance, 12
  
- GlobalSE*, 51
  
- iContract, 27
- implementation, 49
- implementor, 35
- integration strategy, 35
- integrator, 35
- interface, 15
  - inner, 60
  - outer, 60
  
- JDK, 31
- JUnit, 27
  
- language, 42
  
- object constraint language, 41
  - interpreter, 46, 57
- object design, 45
- object-oriented software engineering, 32
  
- packages, 49
- patterns
  - design
    - adapter pattern, 46
    - bridge pattern, 46
    - command pattern, 47
  - process
    - Automated Integration, 49, 69–70
    - catalogue, 61–72

- Component-Based Testing, 67–69
- Continuous Integration, 71–72
- Project Structure and Version Control, 64–66
- relationships, 61
- template, 62
- Uniform Project Name, 31, 62–64
- performance, 40
- platform independence, 39
- problem statement, 33
- productivity, 11
- quality, 11
- regular expressions, 41
- reliability, 11
  - controlling, 11
- requirements
  - analysis, 41
  - elicitation, 35
- reusability, 40
- runtime system, 46
- security, 40
- simple range language, 41
- single source point, 35
- source, 15
- specification, 15
- specification writer, 35
- subsystem decomposition, 43
- system design, 43
- T.R.A.M.P, 51
- tCall, 56
- tContract, 55
- tCreate, 56
- tDestroy, 56
- test harness, 13, 55
- test plan, 35, 42
- testbench, 13
- testbench task, 41
- tester, 35
- testing, 16
  - integration tests, 56
  - system tests, 56
  - unit tests, 55
- tGet, 56
- tInvariant, 56
- tLink, 56
- tSet, 56
- tUnlink, 56
- U.S.E., 28
- verification, 16
- version, 20
  - tag, 20
- XP, 28