Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2007-36

Masterarbeit

im Studiengang Angewandte Informatik

An XML-based Approach for Software Analysis

Applied to Detect Bad Smells in TTCN-3 Test Suites

Jens M. Nödler

Institut für Informatik
Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

26.10.2007

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Deutschland

Tel. +49 (5 51) 39-1 44 14
Fax +49 (5 51) 39-1 44 15
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 26.10.2007

Master's Thesis

An XML-based Approach for Software Analysis

Applied to Detect Bad Smells in TTCN-3 Test Suites

Jens M. Nödler

2007-10-26

Supervised by Dr. Helmut Neukirchen
Co-supervised by Prof. Dr. Jens Grabowski
Software Engineering for Distributed Systems Group
Institute for Computer Science
University of Göttingen, Germany

To everybody who contributed to this thesis,
supported my work, and shared time with me.

I appreciate it.

Abstract

This thesis presents an XML-based approach for software analysis. Patterns in software artefacts which should be found by the analysis are described using the declarative XML query language XQuery. Implementation and design of a software analysis framework are presented. The design of the framework allows patterns to be described in a generic, abstract, and reusable way. The framework is customised for the detection of places in source codes which should be refactored, so-called bad smells. As a case study, the framework is used to detect bad smells in test suites written in the testing language TTCN-3.

Zusammenfassung

Diese Arbeit präsentiert einen XML-basierten Ansatz zur Software-Analyse. Software-Muster, die von der Analyse gefunden werden sollen, sind mittels der deklarativen XML-Abfragesprache XQuery beschrieben. Das Design des implementierten Software-Analyse-Frameworks ermöglichen eine generische, abstrakte und wiederverwendbare Musterbeschreibung. Das Framework ist angepasst für die Erkennung von Stellen in Software-Quelltexten, die mittels Refactoring überarbeitet werden sollten. Diese Stellen werden als „Bad Smells“ bezeichnet. Als Fallbeispiel wird das Framework benutzt, um solche „Bad Smells“ in Testfällen zu finden, die in der Test-Sprache TTCN-3 beschrieben sind.

Contents

Chapter 1

Introduction

Software is continuously changing as the customer's requirements are growing and varying. This makes software more complex and can lead to so-called *software ageing*. The cost of maintaining and enhancing existing software is a deciding factor in a software's lifetime [MMFA04]. About 50% of the software developer's time is consumed by browsing and comprehending existing source code [vMV95]. Maintenance activities took over 60% of all software development costs in the last three decades [Pre05]. These numbers reveal the importance and usefulness of analysis technologies as the starting point for the enhancement of software quality, reusability, and maintainability [EKW99].

Software analysis can be seen as the counterpart of software engineering. While the latter describes processes for the development of new software, the analysis is an elementary part in examining and understanding existing software. The analysis of software concerns internal properties like source code and external properties like runtime behaviour. Software can be analysed regarding different qualities like functionality, reliability, and maintainability for instance [ISO04]. The analysis of internal structures of software is not restricted to reverse engineering or reengineering and can be used amongst others for the following tasks: conformance checking regarding coding styles, extraction of software facts, calculation of source code metrics, control and data flow analysis, and detection of source code anomalies and security-related bugs [Som04, Chap. 22].

Refactoring enhances the internal structure of source code without changing the observable behaviour. The purpose is to increase the software's maintainability and comprehensibility. Refactoring started as a manual process separated from the actual software development. Nowadays refactoring is widely adopted, automated, and tightly integrated with the software development [Fow99, Chap. 14]. Still it is up to the developer's experience and intuition to know *where* to apply refactorings. Locations in the source code which should be refactored are called *bad smells*. Hence, deploying software

analysis for the automated detection of bad smells is the next consequent step towards higher software quality.

Bad smells can be understood as patterns in source code. Describing patterns using a declarative language provides good readability, reusability, and extensibility. Hence, the main goal of this thesis is to develop a declarative way for the description of patterns like bad smells. The XML query language XQuery provides a declarative syntax and is used for the description of software patterns. Therefore, software that is to be analysed, needs to be represented using XML.

The usage of XML and XQuery is bundled in a software analysis framework developed as a part of this thesis. The framework provides the possibility to describe patterns in a generic and reusable way. This is achieved by abstracting from the underlying source code and using XQuery for the pattern description and detection. The framework's design allows arbitrary kinds of analyses: pattern-based detection of anomalies such as bad smells and interactive source code queries for instance.

An application of the framework is the detection of bad smells in TTCN-3 source code. The testing language TTCN-3 is a standardised language for the implementation of test suites for black-box testing. The syntax of the TTCN-3 core notation is very close to those of other common programming languages. Hence, bad smells can also occur in TTCN-3 test suites. The framework is integrated in the TTCN-3 tool TRex which already has refactoring capabilities. The automated smell detection helps bridging the gap towards a smoother refactoring process.

1.1 Structure of this Thesis

This thesis is structured in seven chapters. Following to this introduction, chapter 2 provides foundations which reoccur throughout this thesis. This includes patterns in software artefacts like bad smells, an introduction to the testing language TTCN-3, a preface to XML and related technologies, and an excursion concerning the representation of software artefacts. Related work regarding strategies for the detection of software patterns like bad smells is discussed in chapter 3. As a part of this thesis, a software analysis framework is developed whose requirements and design are presented in chapter 4. In the subsequent chapter 5, the implementation of the XML-based analysis framework and its adaption for the detection of bad smells in TTCN-3 test suites is discussed in depth. The penultimate chapter 6 shows applications of the framework and discusses the extension and customisation of the framework to new fields of application. Finally, chapter 7 provides a summary of this thesis and an outlook towards future work.

Chapter 2

Foundations

This chapter introduces foundations which occur throughout this thesis. Section 2.1 gives an overview of different terms related to patterns in software artefacts. The main topic of this thesis is software analysis which is applied to the testing language TTCN-3. The language is introduced in section 2.2 by describing its features and syntax. Section 2.3 gives an introduction to the *Extensible Markup Language* (XML) and the XML-related technologies XPath and XQuery. These XML technologies are used as the backbone of the software analysis. As the representation of software is elementary for the analysis of software, different representation formats for software artefacts are introduced in section 2.4. Readers who are already familiar with the presented foundations may skip single sections or the complete chapter 2.

2.1 Patterns in Software Artefacts

This section introduces terms describing patterns in software artefacts. As these terms do not only apply to patterns which are restricted to the source code level but also apply to the architectural level of software, the all-embracing term *software artefacts* is used for both levels. No strict definition is available for most of the terms to be introduced. Therefore, they are explained on a descriptive level and differentiations between them are made. By contrast, a few terms are defined by the *International Software Testing Qualification Board* (ISTQB) in the *Standard glossary of terms used in Software Testing* [Int06] and are used in this thesis wherever applicable.

Patterns were first used by the architect Christopher Alexander [AIS+77] in the late 70s as a formal way of documenting successful solutions to problems regarding urban planning and building architecture. They provide high-level architectural descriptions in a generic and reuseable way instead of addressing individual cases on a low level. Each pattern consists of four essential elements: a unique name to establish a common vocabulary, a context in which the pattern might be applied, a description of the problem that the pattern is capable to solve, and the description how to solve this

problem. These four elements apply to all kinds of patterns, not only to those for buildings, as described by Alexander.

In the mid 90s the term *design patterns* was coined by a group of four researchers called "The Gang of Four" [GHJV95] in the field of software architecture. Instead of talking about patterns in buildings and towns as Alexander did, they used the term design patterns to describe reuseable elements of object-oriented software. As the term *design* in design pattern states, they provide solutions on an architectural level. The design patterns were identified by their common usage to solve recurring problems. A design pattern catalogue, introduced in [GHJV95], uses a detailed format to describe each design pattern including motivation, applicability, graphical representation, implementation, examples, and related patterns. The patterns of the catalogue are grouped by their main purpose: creational patterns (like *Singleton* and *Factory Method*), structural patterns (like *Adapter* and *Facade*), and behavioural patterns (like *Iterator* and *Observer*).

The idea of patterns was widely adopted and is nowadays used in many more topics than the design of object-oriented software. The term *bug patterns* for example is at least used twice: to describe recurring relationships between signalled errors and underlying bugs in programs regarding debugging [All02] and also as "error-prone coding practices that arise from the use of erroneous design patterns, misunderstanding of language semantics, or simple and common mistakes" [HP04]. *Test patterns* [Mes07] provide common solutions for problems in nearly all phases of the test process. For example, the patterns *Mock Object* and *Fresh Fixture*. Patterns are however not limited to the technical level and can also be used at the organisational and management level [CH04] for instance.

Anti-patterns [BMMM98] are the counterpart of patterns and describe commonly occurring solutions to problems leading to negative consequences. The idea of anti-patterns (which are sometimes also called *pitfalls*) is to show how *not* to solve a problem. Like patterns, anti-patterns are not limited to the design of software: design anti-patterns are just one category of anti-patterns. Others are project management, programming, methodological, and organisational anti-patterns. The programming anti-pattern *Spaghetti Code* for example describes a program with a software structure that lacks clarity and is therefore hard to maintain and extend because of its complexity [MG05].

Design defects are related to design patterns. While design patterns propose proven solutions to recurring design problems in object-oriented architectures, design defects describe occurring errors in the design of software [GAA01, MG05]. These errors might originate from the absence or the wrong usage of design patterns, which is "the most common mistake in using design patterns" [BMMM98, page 8].

Once a design defect, design anti-pattern or programming anti-pattern is identified, a method called *refactoring* can be applied to transform the

software towards the usage of design patterns [Ker05]. Refactoring is defined by Fowler as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [Fow99]. Analogous to the mentioned catalogue of design patterns, Fowler created a refactoring catalogue consisting of seven categories where each refactoring is presented using a unique name, summary, motivation, mechanics, and examples. Common refactorings are *Rename*, *Encapsulate Field*, and *Extract Method*.

Bad smells are "certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring" [Fow99, Chap. 3]. According to this definition, bad smells are located at the source code level and are therefore often called *code smells*. The term *refactoring opportunities* is also used as a synonym for bad smells. Since the term bad smells was introduced, different other kinds of smells were identified: project smells [AMS02] (that relate to project management anti-patterns), architectural and design smells (that are located at the design level and relate to design defects), and test smells identified by Meszaros [Mes07]. According to Meszaros, three kinds of test smells can be found: smells related to the source code level of tests, behavioural smells that affect the outcome of a test execution, and project smells as indicators of the overall health of a test project which do not involve looking at test code or execution of tests.

The original list of bad smells was presented by Fowler and Beck [Fow99, Chap. 3] additionally to the refactoring catalogue in the same book. Each smell consists of a unique name, a textual description and proposed refactorings to remove the smell. The list of smells is less well structured compared to the catalogues of design patterns and refactorings where each pattern/refactoring follows a strict format. Another drawback of the list of bad smells is the missing topical grouping into categories like the design patterns/refactorings catalogues. The reasons to remove bad smells (as well as design/programming anti-patterns and design defects) by refactoring are various. The most import reasons are: improve the design of software, make software easier to understand, find bugs, and to programme more efficiently [Fow99, Chap. 2]. Altogether, this helps to reduce the maintenance costs and to deliver faster and more reliable software.

Exemplary, the common smells *Duplicated Code* and *Long Parameter List* are presented in a summarised way as they appeared in the original list of bad smells [Fow99, Chap. 3]:

- *Duplicated Code* is the self-describing name for the same code structure found in more than one places. Fowler's advice is to unify the places where the duplicates occur, depending on the their relation to each other. "The simplest duplicated code problem is when you have the same expression in two methods of the same class" [Fow99, page 63]. The recommended refactoring in this case is *Extract Method* whereas

Extract Class is recommended if the duplicated code occurs in different classes.

- The smell *Long Parameter List* applies to methods that have many (for example more than six) parameters. In object-oriented systems long parameter lists should be avoided and instead the method's class should provide most of the required data. Fowler suggests to apply the refactoring *Introduce Parameter Object* to use one object as parameter that contains all required data or *Replace Parameter with Method* to let the called method collect the required data itself by calling all necessary methods.

Software metrics are strongly related to bad smells as they can be used to assess the quality of code according to the ISO/IEC 9126 standard [ISO04]. The ISTQB defines metrics as an overloaded term: "a measurement scale and the method used for measurement" [Int06]. In addition to this definition, the measurement scale must be refined to suit the requirements of software metrics. Hence, the scale must be at least an interval, better a ratio or absolute scale to be able to compare and rank the values of the scale [Pan03]. Applying this definition to the metric *Lines of Code* means that the method used for the measurement is counting all non-empty lines of the source code. The scale of the measurement is the result of the method used for the measurement. In this case the absolute scale would be the number of lines of code, allowing it to compare the results of this metric applied to different entities.

Another definition of metrics originates from Fenton and Pfleeger [FP97] whereby software metrics embraces all activities which involve software measurement. This definition is much wider compared to the one of the ISTQB and includes measures for properties and attributes of entities like processes, resources, and products. Furthermore, it can be distinguished between external and internal attributes, whereat external attributes can be measured only with respect to other entities in their environment. Internal attributes can be measured by examining the products, processes, or resources themselves, and can be separated from their environments. An internal attribute is the concept of internal quality of the ISO/IEC 9126 standard [ISO04]. An example for an internal attribute is the size of the source code of a product as it can be measured in isolation. In contrast, productivity is an external attribute of an entity. For the measurement of the external attributes of a product, execution of the product is required, whereas for measuring internal attributes, static code analysis is sufficient in most cases [ZNG+06].

Regardless of their definitions, metrics can be grouped into two categories: *size metrics* (like *Program Volume* [Hal77]) are based on the number of occurrences of certain language constructs and *structural metrics* (like McCabe's *Cyclomatic Number* [McC76]) are calculated with respect to the structural layout of a program, for instance its branching characteristics.

Bad smells and metrics are related to the source code level while design anti-patterns and especially design defects cover defects on an architectural level. Design patterns, refactorings, and bad smells refer to the object-oriented programming in general and in particular to the programming language *Java*. This means using terms such as *class* and *method* in the context of Java and describing patterns that apply to features specific to Java. However, most of the underlying ideas can be applied for other programming languages as well—even procedural and functional ones.

2.2 TTCN-3 Testing Language

This section introduces the testing language TTCN-3 by describing its features and syntax—exemplified by a test case written in TTCN-3. Afterwards, in section 2.2.2 the *TTCN-3 Code Smell Catalogue* is presented which is an adaption of the list of bad smells for TTCN-3.

2.2.1 The TTCN-3 Core Language

TTCN-3 is a standardised language for the specification and implementation of test suites for black-box testing. The acronym stands for *Testing and Test Control Notation Version 3*. TTCN-3 is widely used for the testing of the implementations of telecommunication and Internet protocols. TTCN-3 test suites reflect the requirements a *System Under Test* (SUT) has to fulfil. Hence, the test execution is the verification of the SUT [Som04]. The TTCN-3 standard [ETS07a] is maintained by the *European Telecommunications Standards Institute* (ETSI) and the first edition of the standard was published in May 2001 [ETS01]. It is also approved by the ITU Telecommunication Standardization Sector (ITU-T) [IT06a] as part of the *Z-series*, which is a collection of standards about languages and general software aspects for telecommunication systems.

"TTCN-3's main feature is the separation of concern between Abstract Test Suites and the Adapter Layer, which allows full portability of test suites and thus make them independent of any platform implementation."[1] For instance, a TTCN-3 test suite for a specific protocol can be used to test different implementations without the need to change the test suite itself—only the adaptor layer might need an update. Hence, test suites written in TTCN-3 provide a high level of reusability. This adaptor layer is called *Real Test System Interface* and is located between the TTCN-3 test system and the SUT as illustrated in figure 2.1. Other major capabilities of TTCN-3 are the support of different presentation formats, dynamic concurrent test configurations, operations for synchronous and asynchronous communications, and data and signature templates with powerful matching mechanisms [IT06b].

[1] `http://www.site.uottawa.ca/~bernard/ttcn3_in_a_nutshell.html`

The latest edition of the TTCN-3 standard is "Edition 3 Version 2" (v3.2.1) published in February 2007, which consists of 10 documents whereat the first one specifies the core notation [ETS07a] of the language—a textual syntax similar to other procedural programming languages but aware of test-specific extensions [WDT+05, Chap. 1]. The remaining documents describe two alternative presentation formats of TTCN-3 (Tabular Presentation Format [ETS07b] and Graphical Presentation Format [ETS07c]), the operational semantics of TTCN-3, components to connect the test suite to the SUT, components to control and log the test execution, several language bindings, and data exchange mappings. Altogether, TTCN-3 is more than just a language for the specification of test cases—it is a framework supporting a broad range of the testing process. Every time TTCN-3 is mentioned in this thesis, the core language of TTCN-3 [ETS07a] is meant—not TTCN-3 as a testing framework—because the software analysis is applied to the core language.

Therefore a closer look at the TTCN-3 core language is required. The building blocks of TTCN-3 are modules, which contain all TTCN-3 code and consist of a definition and a control part whereat both are optional. Modules may import definitions from other modules and can be parametrised to provide more flexibility. Declarations such as data structures and functions are made within the definition part—the control part in turn specifies in which order, under which preconditions, and with which parameters test cases are executed [WDT+05, Chap. 3]. The term *test suite* is synonymous with a complete TTCN-3 module containing test cases and a control part. The body of a simple TTCN-3 module as shown in listing 2.1 consists of a declaration (line 2) and an empty control part only containing a comment (line 3).

```
1 module MyModule {
2   const integer c_integer := 23;
3   control {  /* [...] */  }
4 }
```

Listing 2.1: The body of a TTCN-3 module

TTCN-3 is a typed language with a large number of built-in types namely integer, float, boolean, different string types and a verdict type which reflects the result of a test case. Allowed verdict values are `pass`, `fail`, `inconc`, `none`, and `error`. Common programming constructs such as conditions and loops are also supported as specialised language constructs related to testing such as timers, default behaviour, and alternative behaviour based on the reactions of the SUT [ETS07a].

The communication inside the TTCN-3 test system and the communication with the SUT is abstracted by user-defined ports. They facilitate communication between test components and the SUT and also among the

test components itself. Figure 2.1 illustrates that a test suite consists of at least one *Main Test Component* (MTC) and optionally additional *Parallel Test Components* (PTC). Ports are used for the communication between MTC and PTCs and for the communication between MTC/PTCs and the SUT. The operations on ports provide message-based and procedure-based communication capabilities.

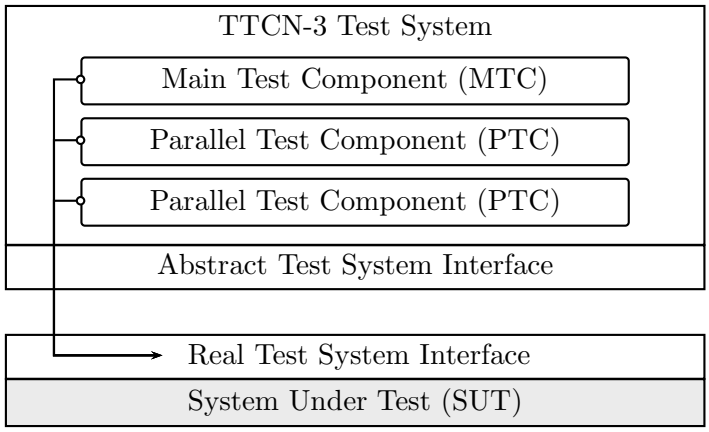

Figure 2.1: Overview of the TTCN-3 architecture: ports are used for the communication between test components and the SUT

For instance, the system under test is an *Hypertext Transfer Protocol* (HTTP) server. An according module must be defined which comprises the port definitions for sending and receiving requests and responses as shown in listing 2.2 in lines 2 and 3. These ports must be assigned to a component that reflects the interface of an HTTP *client* (lines 5–8) as the test suite plays the role of the client. Indeed, these abstract ports need to be mapped to the real interfaces of the SUT what invokes the adaption layer of TTCN-3. This layer is not described in detail here, as it is not relevant for the understanding of the TTCN-3 core language.

```
1 module HTTP {
2   type port PortTypeOutput message { out charstring }
3   type port PortTypeInput  message { in  charstring }
4
5   type component HTTPClientComponentType {
6     port PortTypeInput  InputPort;
7     port PortTypeOutput OutputPort;
8   }
```

Listing 2.2: HTTP module, part 1: definition of ports and a component

Templates are a special kind of data structure providing parametrisation and matching mechanisms for specifying test data to be sent or received over

test ports. Templates are used to either transmit a set of distinct values or to test whether a set of received values matches the template specification [WDT+05, Chap. 10].

Dynamic test behaviour is expressed using test cases. TTCN-3 statements include powerful behavioural description mechanisms: functions and testcases are used to specify and structure test behaviour, to define default behaviour and to structure modules. The alt statement denotes branching of test behaviour due to the reception and handling of communication and/or timer events and/or the termination of parallel test components. Each alt statement consists of at least one branch including a condition and an action to be executed. Branches may be extracted into own function-like statements (called *altstep*) for their reuse or the activation of an altstep as a default behaviour in an alt statement [ETS07a].

In the next step the TTCN-3 module `HTTP` from listing 2.2 is extended to test an HTTP request/response cycle. Therefore a template that matches a response is defined in line 9 of listing 2.3 using a regular expression pattern. A test case should test the SUT which is abstracted using components. Therefore the test case `HTTPHeadRequest` is defined to run on the HTTP client component (line 11) as defined in listing 2.2. That allows to access the ports of this component and to send an HTTP request through the output port to the server (line 14). The following alt statement consists of three steps that cover different possible responses of the server. In the first case (lines 17–19) an HTTP response that matches the template `HTTPResponse` is received via the input port and the test verdict is set to `pass`. The other steps cover potential error conditions. The second alternative step (lines 20–22) sets the verdict to `fail` if a response is received that does *not* match the template `HTTPResponse`. The third altstep is triggered if no response at all is received within 5 seconds (lines 23–25) based on the timer that was started (in line 13) right before the request was send.

2.2.2 The TTCN-3 Code Smell Catalogue

The *TTCN-3 Code Smell Catalogue* is the first approach to adopt the bad smells of Fowler [Fow99] for the core language of TTCN-3. It was first presented in the master's thesis of Bisanz [Bis06] and is now maintained by the Software Engineering for Distributed Systems Group at the University of Göttingen [NB07, NZG08].

The catalogue for TTCN-3 differs in many points from the original list of bad smells:

- The catalogue contains not only bad smells that can be found in (more or less) any programming language but also additionally smells that only apply to TTCN-3.
- Even though removing bad smells by refactoring the source code means by definition to preserve the behaviour of the program, the catalogue

```
 9   template charstring HTTPResponse := pattern "HTTP/1.1 \d\d\d *";
10
11   testcase HTTPHeadRequest() runs on HTTPClientComponentType {
12     timer t;
13     t.start(5.0);
14     OutputPort.send("HEAD /index.html HTTP/1.1");
15
16     alt {
17       [] InputPort.receive(HTTPResponse) {
18         setverdict(pass);
19       }
20       [] InputPort.receive {
21         setverdict(fail);
22       }
23       [] t.timeout {
24         setverdict(fail);
25       }
26     }
27   }
28 }
```

Listing 2.3: HTTP module, part 2: testing a request/response cycle

contains some smells that cannot be removed without changing the observable behaviour.

- The bad smells of the catalogue are grouped in ten different categories to add structure and clarity. The following categories contain general as well as TTCN-3-specific smells: *Duplicated Code*, *References*, *Parameters*, *Complexity*, *Coding Standards*, *Data Flow Anomalies*, and *Miscellaneous*. Additionally, these categories only contain smells related to TTCN-3: *Default Anomalies*, *Test Behaviour*, and *Test Configuration*.

- Each bad smell follows a strict format describing the smell in depth: *Name*, *Derived from*, *Description*, *Motivation*, *Options* (optional), *Related action(s)*, and *Example*.

- Following the categorisation of test smells by Meszaros [Mes07] the code smell catalogue contains source code smells and behaviour smells.

As one goal of this thesis is to detect the bad smells of this catalogue in TTCN-3 test suites, two smell examples are quoted [Bis06]:

Name: Singular Template Reference

Derived from: [Zei06] (Motivation for *Inline Template*)

Description: A template definition is referenced only once.

Motivation: If a template definition is referenced only once, it can be inlined without duplicating code. This can improve readability if the

template definition is not too complex; in case of a very complex template a separate template definition can still be preferable.

Related action(s): *Inline Template*

Example: In listing 2.4, assume the only reference to the template named `exampleTemplate` (lines 1–5) is the one in test case `exampleTestCase` (line 9). In this case, the template could be inlined to reduce code length and improve readability.

```
1  template MyMessageType exampleTemplate := {
2    field1 := omit,
3    field2 := "foo",
4    field3 := true
5  }
6
7  testcase exampleTestCase() runs on ExampleComponent {
8    // ...
9    pt.send(exampleTemplate);
10   // ...
11 }
```

Listing 2.4: The bad smell *Singular Template Reference*

Name: Long Parameter List

Derived from: [Fow99] (*Long Parameter List*)

Description: High number of formal parameters.

Motivation: Long parameter lists are hard to read and should be avoided. Although this smell is more relevant for object-oriented languages (because method parameters can be replaced by attributes within a class), a group of single parameters can be replaced by a record type parameter. If the calling behavioural entity (function, test case or altstep) gets a parameter by calling another function or altstep and does not need the parameter by itself, *Replace Parameter with Function* can be applied.

Related action(s): *Replace Parameter with Function* or *Introduce Record Type Parameter*

Example: Listing 2.5 contains the signature of a function with six integer parameters (line 2). Instead a set or record type could be used.

```
1 function f1(integer i1, integer i2, integer i3,
2     integer i4, integer i5, integer i6) {
3   // some behaviour...
4 }
```

Listing 2.5: The bad smell *Long Parameter List*

2.3 XML Technologies

This section introduces the *Extensible Markup Language* (XML) and the XML query languages XPath and XQuery. All technologies are used for the software analysis framework presented in chapter 5: XML as representation format for TTCN-3 source code and XPath/XQuery to analyse the XML-encoded TTCN-3 source code for specific characteristics and patterns.

2.3.1 Extensible Markup Language

The "*Extensible Markup Language* (XML) is a simple, very flexible text format [...] playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere."[2] XML is a free, platform independent, open standard [BSMY+06] provided by the *World Wide Web Consortium* (W3C) and a subset of the Standard Generalized Markup Language (SGML, ISO 8879). As the name Extensible Markup Language already implies, the language addresses the markup of data by adding structural information to it. XML is extensible because it does not provide a set of predefined markup tags as HTML [JRH99] or LaTeX. In contrast, XML is a meta markup language that defines a syntax to create new domain-specific markup languages [Har01, Chap. 1]. To illustrate the XML syntax an example of a new defined language is given in listing 2.6 that structures a few seasons, episodes, and titles of the TV sitcom "The Simpsons" as an XML document.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <simpsons>
3    <season number="1">
4      <episode number="1" title="Simpsons Roasting on an Open Fire"/>
5      <episode number="2" title="Bart the Genius"/>
6      <!-- [...] -->
7    </season>
8    <season number="19">
9      <episode number="401" title="He Loves to Fly and He D'oh's"/>
10     <episode number="402" title="The Homer of Seville"/>
11     <!-- [...] -->
12   </season>
13 </simpsons>
```

Listing 2.6: XML example: "The Simpsons" seasons and episodes

[2] http://www.w3.org/XML/

Listing 2.6 illustrates the most important properties of an XML document. First of all it should be human readable and self describing—but that is up to the author of the XML language. According to the XML specification each document consists of an optional prolog (line 1, also called "XML declaration") and exactly *one* root element (in the example `simpsons`, line 2–13). The logical structure is based on elements whereas each element has a start tag (like `<simpsons>`, line 2) and end tag (like `</simpsons>`, line 13). Each element may contain content between its start and end tag (for instance the content "D'oh!" of an element `homer`: `<homer>D'oh!</homer>`), attributes (like the `number`, and `title` attributes in the example), an arbitrary number of nested elements or may be empty (for example `<no-content/>`). XML comments (lines 6 and 11) start with the character sequence `<!--` and end with `-->`.

An XML document following these rules is called "well-formed". This is the minimal requirement for a document to be referred to as XML document at all [BSMY+06, Section 2.1]. More sophisticated XML documents are called "valid" if they have an associated *Document Type Definition* (DTD) and if the document complies with the constraints expressed in the DTD. The property "valid" includes the property "well-formed" [BSMY+06, Section 2.8]. DTDs define the structure of an XML document in terms of production rules similar to the *Backus–Naur Form* (BNF). DTDs are an integral part of the XML specifications [BSMY+06, BYC+06] and can be inlined in the XML document or included from another file. Adding the *Document Type Declaration* statement `<!DOCTYPE simpsons SYSTEM "simpsons.dtd">` to the example behind the XML prolog would include the DTD shown in listing 2.7.

```
1 <!ELEMENT simpsons (season*)>
2 <!ELEMENT season (episode*)>
3 <!ELEMENT episode EMPTY>
4 <!ATTLIST season number CDATA #REQUIRED>
5 <!ATTLIST episode
6   number CDATA #REQUIRED
7   title  CDATA #REQUIRED>
```

Listing 2.7: Document Type Definition for the Simpsons example

The first three lines of the DTD define the allowed elements, their relation among each other, and the frequency they may occur. The root element `simpsons` may contain zero or more `season` elements (line 1), which may contain any number of `episode` elements (line 2). The `episode` element itself is an empty element (line 3) that must contain two attributes named `number` and `title` (lines 5–7). Line 4 specifies an obligatory attribute `number` for all `season` elements. Using a validating XML processor—a software that is capable of checking an XML document against its DTD—permits to find missing or needless elements and attributes and to ensure that the document is valid.

With the knowledge of DTDs in mind an XML document is said to be an instance of a DTD [Har01, Chap. 8]. Hence, the recommended way to create new XML languages is by designing the Document Type Definition first—comparable to relational databases where the entity-relationship model must be set up first [Teo98, Chap. 2]. Instead of DTDs, that are part of the XML specifications [BSMY+06, BYC+06], more powerful and expressive approaches such as XML Schema [WF04] or RELAX NG [CM01] can be used for modelling and validating XML languages.

If an XML document makes use of more than one XML language definition, name collisions might occur as different languages may use elements with the same name for instance. Additionally, the problem of recognising which elements belong to which XML language emerge in such XML documents. To avoid these problems, XML namespaces [LTBH06] can be used to provide uniquely named elements and attributes. Each XML element is bound to a unique identifier—its XML namespace. The result is an *expanded name* which consists of the namespace name and the local name (like the element name). This allows to identify each node and avoids name collisions as XML names only need to be unique it their own namespace.

Currently two versions of XML co-exist: XML version 1.0 [BSMY+06] and version 1.1 [BYC+06]. Although the latter seems to supersede the first version, the usage of XML 1.0 is still recommended, because it serves most use cases [HM04, Chap. 2] and the tool support for XML 1.0 is considerable better than for version 1.1. The main difference between both specifications is that the "overall philosophy of names [of elements, attributes etc.] has changed since XML 1.0. Whereas XML 1.0 provided a rigid definition of names, wherein everything that was not permitted was forbidden, XML 1.1 names are designed so that everything that is not forbidden (for a specific reason) is permitted" [BYC+06]. Names in XML 1.0 are only allowed to used characters defined by the Unicode 2.0 standard [Con96]. But as Unicode is being enhanced to version 5.0 [Con06] and beyond, it would have required new versions of XML everytime new characters were included in the Unicode standard—or to allow all characters, as done in version 1.1. For reasons of clarity: this only effects XML names and not the content of an XML document. Hence, it is possible to use Unicode 5.0 characters for the content of an XML 1.0 document but not for its names.

2.3.2 XML Path Language

The main purpose of the *XML Path Language* (XPath) [DC99, CBF+07] is to navigate through the tree structure of an XML document, to address certain nodes and to select parts of the document using a simple declarative syntax. For example, the path expression `//season` would evaluate to a sequence containing all `season` elements (including their child elements) from the XML document shown in listing 2.6. The expression `//episode[@number > 400]`

would return a sequence of all `episode` elements with an attribute named `number` containing a value larger than 400.

Two versions of the XPath standard have been released as recommendations by the W3C: XPath 1.0 [DC99] in 1999 and XPath 2.0 [CBF+07] in 2007. Even though both version's main purpose stays the same, they differ significantly from each other. Version 1.0 was defined to "provide a common syntax and semantics for functionality shared between *XSL Transformations* (XSLT) [Cla99] and XPointer [DMJ01]" [DC99, Har01]. XPath 2.0 in turn is designed to be embedded in host languages like XSLT 2.0 [Kay07] and XQuery 1.0 [RSF+07]. In contrast to the specification of XPath 1.0, that includes not only the syntax and semantics of the language but also the data model, built-in library functions, and operators, XPath 2.0 is defined in a set of specialised specifications. The main document is called "XML Path Language (XPath) 2.0" [CBF+07] and mostly describes the concepts and syntax of XPath, meanwhile the semantics, data model, built-in functions and operators are shared with XQuery 1.0 and therefore specified in combined recommendations: "XQuery 1.0 and XPath 2.0 Formal Semantics" [MRS+07], "XQuery 1.0 and XPath 2.0 Data Model (XDM)" [WMN+07] and "XQuery 1.0 and XPath 2.0 Functions and Operators" [MWM07].

Compared to its successor, XPath 1.0 is a simple "one-line language" being less expressive and flexible as it is mainly capable of path expressions (`//season` for instance) whereas XPath 2.0 supports more sophisticated expressions like conditional and loop constructs. Hence, XPath 2.0 is a superset of XPath 1.0 including more language expressions and a lot of more built-in functions. Additionally, XPath 2.0 supports a richer set of data types, to take advantage of type information when documents are validated using XML Schema. From here on, the term XPath always refers to version 2.0.

The XPath data model [WMN+07] features two elementary properties making the handling of nodes more efficient. The first is called *document order*, which is defined as the order of all nodes in which they appear in the XML document. When nodes are selected from the document, they are output in the document order by default. The document order also allows to compare the position of nodes in a before/behind relation. The second elementary property is called *node identity*, which assigns a unique identity to each node what allows to differentiate between comparing nodes based on their value and their identity.

As the name XPath already indicates, path expressions are elementary constructs of the language. A path expression can be used to locate nodes within XML trees and consists of a series of one or more steps, separated by `/` or `//`, and optionally beginning with `/` or `//`. Steps are defined as axis steps or filter expressions. Axis steps define the direction of movement in the tree and filter expressions test if a selected node holds a specified condition. An axis step returns all nodes that are reachable from the context node via a specified axis and that fulfil the filter expression [HM04].

Axis	*Meaning*
`child::`	Children of the context node.
`descendant::`	All children of the context node.
`attribute::`	Attributes of the context node.
`self::`	The context node itself.
`descendant-or-self::`	The context node and its descendants.
`following-sibling::`	All siblings of the context node that follow it.
`following::`	All nodes that follow the context node in the document.
`parent::`	The parent of the context node.
`ancestor::`	All ancestors of the context node.
`ancestor-or-self::`	The context node and all its ancestors.
`preceding-sibling::`	All the siblings of the context node that precede it.
`preceding::`	All nodes that precede the context node in the document.

Table 2.1: All axes provided by XPath and their meanings

XPath supports different forward and backward axes which reflect the direction of movement in the document order of the XML tree. Forward means nodes after or beneath the current context node; backward means nodes before or above the current context node [Bru04, Chap. 3]. Because the axes are important for XPath all axes and their meanings are presented in table 2.1.

For the most frequently used axes an abbreviated syntax is defined: `descendant-or-self::` is shortened to "`//`", `parent::` to "`..`", `self::` to "`.`", `attribute::` to "`@`", and `child::` can be omitted completely. Using the abbreviated syntax—which is favoured in this thesis wherever possible—means writing `episode/@number` instead of `child::episode/attribute::number` for example.

Looking again at the example `//episode[@number > 400]` that was already given at the introduction to XPath, it is now possible to disassemble the path expression step by step. Path expressions are evaluated from the left to the right side: the slashes `//` traverse the descendant-or-self axis of the XML tree (listing 2.6) starting from the root node, searching element nodes named `episode`, and selecting each as the current context node. A filter expression `[condition]` is now applied to the context node and the attribute axis is inspected for an attribute named `number` holding a value larger than 400. If this condition is true, the context node is included in the resulting sequence. All matching nodes are returned in document order as shown in listing 2.8.

Note that the result of an XPath expression is not needed to be well-formed or valid XML nor XML at all—it is just a sequence of items. An item is either an atomic value (of a primitive simple type defined by the

```
1 <episode number="401" title="He Loves to Fly and He D'oh's"/>
2 <episode number="402" title="The Homer of Seville"/>
```

Listing 2.8: Result of the XPath expression `//episode[@number > 400]`

XPath data model) or a node [WMN+07]. Conforming with the XPath data model, seven nodes types are differentiated. The most important ones are: document nodes, element nodes, attribute nodes, text nodes, and comment nodes. An XPath result example is the sequence `(23, "foo", <bar/>)` which contains two atomic values and one element node. Another example is the path expression `//episode/@title` which returns a sequence of all `title` attribute nodes of `episode` elements.

Beside path expressions, sequence expressions are relevant for XPath. First of all, to construct a new sequence a pair of parentheses `(···)` is used to surround all elements of a sequence. The elements themselves are delimited by commas. For example, the sequence of the first seven Fibonacci numbers `(0, 1, 1, 2, 3, 5, 8)` or the sequence of all Simpsons season and episode numbers `(//season/@number, //episode/@number)`.

As already seen in the example above, filter expressions can be applied to any sequence to cherry-pick the desired nodes. The most simple filter expression uses the position of a node in its sequence. Hence, `(//episode)[2]` returns the second `episode` element node (whereas the filter expression `[2]` is an abbreviation of `[position() = 2]`). To get the last element of a sequence the function `last()` needs to be called. For instance, the filter expression `[last() - 1]` returns the penultimate element of the context sequence. It is also possible to evaluate the content of the context node (or its descendant respectively its ancestor) in a filter expression. For example, `(1 to 100)[. mod 5 eq 0]` lists all integers from 1 to 100 that are divisible by 5 without remainder based on the context node's content. It is possible to stack as many filters as required: `(//episode)[2][@foo]` would return the second episode element if it would have an attribute called `foo`.

Using name tests and node type tests, XPath permits to select nodes by their type and name. Using the wildcard `*` it is possible to select elements and attributes regardless of their names. The path expression `//@*` selects all attribute nodes for instance. The language provides three built-in operators for combining sequences: `union`, `intersect`, and `except`. All these operators eliminate duplicate nodes from their result sequences based on the node identities. Additionally, XPath supports basic arithmetic expressions (`+`, `-`, `*`, `div`, and `mod`) and comparison expressions for values and nodes.

XPath shares predefined functions and operators with XQuery [MWM07]. They cover a wide area of application like numerics, strings, booleans, dates and times, sequences, and casting. The specification [MWM07] defines 68 operations and 122 functions. Function calls looks like this: `data(//@number)` to

get the value of all `number` attributes or `doc("http://noedler.de/index.rss")` to load an XML document.

As mentioned before, XPath 2.0 supports more advanced constructs than version 1.0 does, namely loop, conditional, and quantified expressions. Using the `for` expression as shown in listing 2.9 allows to loop through the XML tree in a more flexible way than using path expressions. The example returns tuples of all Simpsons season numbers and their appropriate episode numbers which would not have been possible using XPath 1.0.

```
1 for $s in //season,
2     $e in $s/episode
3 return ( data($s/@number), data($e/@number) )
```

Listing 2.9: XPath expression using the `for` construct

The conditional expression is an if-then-else construct that allows to react appropriately based on values in the XML tree. Quantified expressions support existential (keyword `some`) and universal (keyword `every`) quantification whose result is true or false, which allows to check that a condition is true/false for some or all nodes. The expression in listing 2.10 returns `true`, because every `episode` element satisfies the condition to own a `number` attribute whose value is larger than zero. Quantified expressions are often used in conjunction with conditional expressions.

```
1 every $episode in //episode
2 satisfies $episode/@number > 0
```

Listing 2.10: XPath quantified expression

2.3.3 XML Query Language

The *XML Query Language* (XQuery) is described from a high-level view as "a standardized language for combining documents, databases, Web pages, and almost anything else. It is very widely implemented. It is powerful and easy to learn."[3] The language is standardised by the W3C [RSF+07] and its aim is to use the structure of XML intelligently to express queries across all kinds of XML. It allows to select XML elements of interest, reorganise and transform them, and return them in a user-defined order and structure [Wal07, Chap. 1].

Seen from a technical point of view, XQuery is for XML what SQL is for the relational databases. "XQuery will serve as a unifying interface for access to XML data, much as SQL has done for relational data."[4] XQuery is derived from an XML query language called Quilt, which borrowed features

[3] `http://www.w3.org/XML/Query/`

[4] `http://electronics.ihs.com/news/w3c-xml-approves.htm`

from several other languages, including SQL, XQL, and XPath 1.0. In fact XQuery 1.0 is an extension of XPath 2.0 or—defined vice versa—XPath is a complete subset of XQuery [CBF+07].

XQuery (and therefore also XPath) is a functional and typed language, which means it is built from expressions rather than statements. "Every construct in the language [...] is an expression and expressions can be composed arbitrarily. The result of one expression can be used as the input to any other expression. [...] Another characteristic of a functional language is that variables are always passed by value, and a variable's value cannot be modified through side effects" [MRS+07]. XQuery is a typed language, because it is capable of importing types from an XML Schema definition and use them to perform operations based on these types, such as early detection of type errors.

The basic structure of many (but not all) queries is the FLWOR (pronounced "flower") expression which stands for the keywords `for`, `let`, `where`, `order by`, and `return` used in the expression. FLWORs, unlike path expressions, allow to manipulate, transform, and sort results. This is the main difference and advantage of XQuery compared to XPath. The latter is only capable of retrieving nodes from XML documents whereat XQuery allows to rearrange nodes and to create new ones. The following listing 2.11 uses a FLWOR expression to restructure the Simpsons XML tree (listing 2.6) and to return a sequence (listing 2.12) containing all Simpsons episodes sorted by their title.

```
1 for $episode in doc("simpsons.xml")//episode
2 let $title := data($episode/@title)
3 order by $title
4 return element simpsons-episode { $title }
```

Listing 2.11: XQuery expression to restructure the Simpsons example

The query first iterates over all `episode` elements (line 1) of the input XML document `simpsons.xml` and binds each element to a variable called `$episode`, reads the data value of the `title` attribute of the current context node `$episode`, binds the value to the variable `$title` (line 2), and returns each title as the content of a new `simpsons-episode` element (line 4) whereas the result set is ordered by the title (line 3).

```
1 <simpsons-episode>Bart the Genius</simpsons-episode>
2 <simpsons-episode>He Loves to Fly and He D'oh's</simpsons-episode>
3 <simpsons-episode>Simpsons Roasting on an Open Fire</simpsons-episode>
4 <simpsons-episode>The Homer of Seville</simpsons-episode>
```

Listing 2.12: Resulting sequence of query 2.11

XQuery is not a pure query language as it allows the definition of user-defined function, which can also be called recursively [Wal07, Chap. 8]. Mod-

ules allow the grouping of functions in a reuseable way and the language itself is proven to be Turing complete [Kep04].

The XQueryX 1.0 specification [MM07] provides an XML syntax for XQuery. Converters such as XQ2XML [XQ206] permit to transform XQuery to XQueryX and an XSLT stylesheet provided by the XQueryX 1.0 specification can be used to transform XQueryX back to XQuery. XQueryX allows to use the XML representation of an XQuery expression as input for XQuery or XSLT to apply transformations and queries on itself.

One drawback of XQuery is the missing support for updating parts of an XML document without re-generating the complete document. The W3C has identified this problem and is working on a draft called "XQuery Update Facility" [FRC06]. Another approach into this direction is the XML update language XUpdate [XUp00].

2.4 Representations of Software Artefacts

Software needs to be represented in varying ways to fulfil different requirements. Software architects for example require a global view of all components of a system to be able to design extensible and maintainable software. Therefore they require software representations at an abstract level. In contrast, source code analysis tools need to investigate all details of a software to find design flaws or potential bugs and require a software representation that reflects the fully detailed source code.

This section introduces facilities for software representation such as metamodels (section 2.4.1), the *Unified Modeling Language* (UML) (section 2.4.2), XML-based formats (section 2.4.3), and other common approaches (section 2.4.4) like intermediate formats and tree-based source code representations. For each representation format it is discussed whether it is capable to serve as a high-level software representation and/or full detail software representation format.

2.4.1 Metamodels

A model is an abstraction of the real world and the according metamodel is the abstraction of its model. Metamodelling is the process of constructing a collection of concepts within a certain domain, for example the software domain in the field of computer science. Metamodels define languages for the specification of models [Pen02]. Hence, metamodels are playing the same role for models as—for example—grammars for programming languages. Vice versa, a model is said to conform to its metamodel such as source code conforms to the grammar of the language it is written in.

Metamodels use a four-layer metamodel architecture [Pen02] that can be explained by the definition of an XML language using a DTD[5] (see section 2.3 for details on XML and DTD):

Level M0 The *User Object Layer* defines specific subject domain information such as the real world data to be described. In the case of a new XML language this is the content that should be contained in the XML documents.

Level M1 The *Model Layer* defines the language to use to describe a subject domain. For this example of an XML language, that means which elements and attributes are required to model the content and at large the complete XML document.

Level M2 The *Metamodel Layer* defines the language for specifying models. In case of XML, using a DTD is one way to describe the model as the DTD defines the logical structure of an XML document. Also XML Schema and RELAX NG are located at the metamodel layer as they can be used to describe models of XML documents.

Level M3 The *Meta-Metamodel Layer* defines the language for specifying metamodels. For this XML example, that means this layer defines the language a DTD is written in. In the case of XML Schema it is XML again as XML Schema uses an XML syntax.

The advantage of this layered structure is—assuming that the meta-metamodel is rich enough—the four-layer architecture can support and model most, if not all kinds of information and meta-information imaginable.

These metamodelling foundations are provided by the *Object Management Group* (OMG)[6] because the OMG was in need of a metamodel architecture for the definition of the *Unified Modeling Language* (UML) [UML07]. The *Meta-Object Facility* (MOF) [MOF07] serves as the basis for UML and makes use of the four-layer architecture discussed above. It is defined as a closed metamodelling architecture as MOF defines a M3 model, which conforms to itself. The most prominent example of a M2 model is the UML metamodel, the model that describes UML. Hence, models written in UML are located at the M1 layer.

MOF and UML provide a "key foundation for OMG's *Model-Driven Architecture* (MDA), which unifies every step of development and integration from business modeling, through architectural and application modeling, to development, deployment, maintenance, and evolution."[7] Models expressed in a well-defined notation are a cornerstone of systems for enterprise-scale so-

[5] `http://www.devhood.com/tutorials/tutorial_details.aspx?tutorial_id=698`
[6] `http://www.omg.org`
[7] `http://www.uml.org`

lutions.[8] The development of systems can be organised around a set of models, transformations between models, and derivations from models [ST03].

Metamodels are mainly used for the representation of software artefacts at an architectural level. Different metamodel approaches for software representation emerged and can be differentiated by their granularity of the representation of the underlying software. Some, for instance the FAMIX metamodel [TSDN00], concentrate on the most important object-oriented entities (classes, attributes, and methods) and their relationships (access and invocations). A metamodel with about the same degree of granularity is presented by Guéhéneuc [Gué02] to display program architectures and behavioural information. Limiting the granularity reduces the complexity of the metamodel and provides a high-level view of the underlying software with the restriction that the source code is not completely reflected by the model. A more detailed approach for the representation of Java source code is presented by van Emden and Moonen [vEM02]: the metamodel "contains information regarding program entities like packages, classes, [...] methods, constructors, static blocks, and fields. Furthermore, it describes the relations between these entities such as composition, inheritance, [and] method calls" [vEM02].

But metamodels are not limited to the representation of software on a high-level view, but can also be used to represent source code in depth. That allows for example to apply refactorings at the model level as a model transformation and to generate the refactored source code from the transformed model [GEJ03]. The range of metamodels representing source code in depth varies from ones that try to cover different object-oriented languages [GEJ03] to specialised ones, like a metamodel for TTCN-3 [SD04].

2.4.2 Unified Modeling Language

The *Unified Modeling Language* (UML) is a standardised general-purpose modelling language that includes a graphical notation. UML is used to create an abstract model of a system, called *UML model*. The language is specified by the Object Management Group (OMG) and the latest version of UML is 2.1.1 [UML07]. UML provides a generic extension mechanism to build UML models for particular domains called *UML profiles* that tailor the language to specific areas.

As already described in section 2.4.1, UML is formally defined using a Meta-Object Facility metamodel that defines all constructs of UML. The metamodel itself is expressed in UML. This is an example of a metacircular definition as the language is defined in terms of itself. But things are not completely circular, only a small subset of UML is used to define the metamodel [QP06]. But these internals are only important to those who use

[8] `http://www-128.ibm.com/developerworks/rational/library/3100.html`

UML as a programming language, as it defines the abstract syntax of that language [Fow03].

More important is the overall impact of UML in the field of software engineering regarding modelling software. While analysis and design phases are known by nearly all software development processes, using UML allows to perform these phases on a unified basis. UML can be used in nearly all phases of a software development process: use case and activity diagrams in the analysis and requirements phase and class diagrams during the design phase are just a few possible applications [Fox06, Som04]. A couple of development processes are based even completely on UML respectively models, for example the model-driven development that is part of OMG's model-driven architecture. The extensive usage of models starts with the automatic generation of source code stubs and can lead to the use of executable models [MBMB02]. In most cases, UML is used for designing and representing software on an architectural level and not as a representation format for full detailed source code.

The development process is one part of a software engineering method. Other parts are a common vocabulary and a set of guidelines. The vocabulary is used to describe the process and the products created during the application of the process. The part of all software engineering methods that can be standardised is the vocabulary, often expressed in a notation. UML is a common notation that may be applied to many different types of software projects using different methods [Pen02].

After having a look at the definition and the possible applications of UML, the different diagram types are briefly introduced. UML provides a set of thirteen predefined types of diagrams for different purposes. They are grouped into three categories[9]:

1. Structure diagrams represent static application structure and include the class diagram, object diagram, component diagram, composite structure diagram, package diagram, and deployment diagram.

2. Behaviour diagrams represent general types of behaviour and include the use case diagram, activity diagram, and state machine diagram.

3. Interaction diagrams represent different aspects of interactions. They are all derived from the more general behaviour diagram and include the sequence diagram, communication diagram, timing diagram, and interaction overview diagram.

To give an example of how UML looks, figure 2.2 shows a UML class diagram containing two classes. Class `Person` is the base class containing the attribute `name` and the according operation `getName()` to access the private class attribute. The class `Employee` inherits from the base class `Person` and adds the attribute `salary` and the according getter operation.

[9] `http://www.omg.org/gettingstarted/what_is_uml.htm`

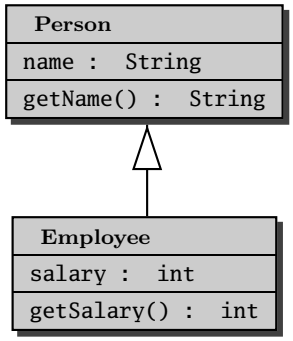

Figure 2.2: UML class diagram: class `Employee` inherits from class `Person`

What makes UML special among other representation formats for software artefacts (presented in this section) are two properties: First, UML is a general-purpose modelling language that can be used for business modelling and modelling of other non-software systems. But UML is mainly designed for modelling software what makes it different from other representations formats that are generic approaches reused for the representation of software. Second, UML is a graphical approach. Even though it is important to distinguish between the UML model and the set of diagrams as graphical representations of the models, UML is mostly used on a graphical level. While all other representation formats can also be visualised, the visualisation is not an integral part of these formats.

2.4.3 XML-based Representations

The Extensible Markup Language (XML, see section 2.3 for details) is not only used for exchanging structured information but serves also as a language for representation formats. This section shows how XML is used to represent software on an architectural level and also how to represent fully detailed source code. The first part covers the standardised language *XML Metadata Interchange* (XMI) that is mainly used for the representation of UML models as XML, while the second part covers the usage of XML to encode source code.

The reasons to use XML for the representation of software artefacts are various: XML is a free and open standard that is platform-independent and commonly used. But even more important is availability of a wide range of XML processing languages such as XQuery and XSLT and XML-aware tools.

2.4.3.1 XML Metadata Interchange

The *XML Metadata Interchange* (XMI) [XMI05] is an XML language standardised by the Object Management Group for serialising and exchanging

metadata information such as metamodels across platforms and systems. The most common usage of XMI is as an exchange format for UML models and metamodels, although XMI can also be used for serialisation of non-UML models and metamodels. In general XMI can be used for any metadata whose metamodel can be expressed using the Meta-Object Facility (MOF) on the M3, M2, or M1 layer.

XMI bridges the gap between generic objects (for example models/metamodels) and XML by defining a new domain-specific XML language to improve the usage of XML for the interchange of objects [GDB02]. The flexibility of XML allows to map objects to XML in nearly infinite possible ways. XMI addresses this property of XML and acts as a gateway by defining standards *how* the mapping from an object to XML elements/attributes should be done. XMI also defines the reverse mapping to restore the original object from an XMI document. XMI specifies how to create XML Schema and Document Type Definitions (DTD) from a model and vice versa how to create a model from a given DTD or XML Schema [GDB02].

Listing 2.13 shows the XMI representation of the class `Employee` of the UML class diagram shown in figure 2.2. It was created using the open-source UML modelling tool ArgoUML 0.24 [Arg07] that uses XMI as its default serialisation format for UML models. A lot of details (XMI elements and attributes) are left out, to concentrate on the main structure of the XMI document. Additionally, the values of the `xmi.id` and `xmi.idref` attributes have been changed from automatically generated identifiers to better readable ones. The first part of this snippet of the XMI document represents the class `Employee` (lines 1–7), its attribute `salary`, and operation `getSalary` (lines 4 and 5). Line 2 references the generalisation of the class `Employee` which is represented in the second parts of the XMI snippet (lines 8–15). The inheritance relationship of the classes `Employee` and `Person` is expressed as a child-parent relation in which the class `Employee` is the child (lines 9–11) of the parent class `Person` (lines 12–14).

```
1  <UML:Class xmi.id='ClassEmployee' name='Employee' visibility='public'>
2    <UML:Generalization xmi.idref='GeneralizationPersonEmployee'/>
3    <UML:Classifier.feature>
4      <UML:Attribute name='salary' visibility='private'/>
5      <UML:Operation name='getSalary' visibility='public'/>
6    </UML:Classifier.feature>
7  </UML:Class>
8  <UML:Generalization xmi.id='GeneralizationPersonEmployee'>
9    <UML:Generalization.child>
10     <UML:Class xmi.idref='ClassEmployee'/>
11   </UML:Generalization.child>
12   <UML:Generalization.parent>
13     <UML:Class xmi.idref='ClassPerson'/>
14   </UML:Generalization.parent>
15 </UML:Generalization>
```

Listing 2.13: XMI representation of the UML class diagram figure 2.2

Applications using XMI as their primary serialisation format are for example the UML modelling tool ArgoUML [Arg07] as seen above and the *Eclipse Modeling Framework* (EMF) [Ecl07b]. EMF extends the usage of XMI by not only storing metamodels or models but also the data of these models (the M0 layer in the four-layer metamodel architecture, see section 2.4.1 for details) what is not covered by the XMI specification.

One often raised point of critique regarding XMI is that the XMI standard is not detailed enough and leaves to much space for implementation-specific variations [AP05]. That causes problems when it comes to exchanging metadata using XMI between products of different vendors. In practice this means exchanging XMI files between UML modelling tools from different vendors using XMI is rarely possible [AP05].

2.4.3.2 Encoding Source Code as XML

This section introduces the usage of XML for the fully detailed representation of source code. As the software analysis of this thesis is XML-based, the two most important varieties of XML representation formats are discussed in depth in this section.

Even though a lot of approaches exist to encode source code as XML, two basic concepts can be distinguished. The first concept maps all constructs of the original programming language to according XML elements/attributes. User-defined data such as variable names is stored as the content of these XML elements/attributes. This approach is called *representation concept* as it provides a new representation of the source code. The second concept is called *annotation concept* as it literally marks-up the original source code using XML elements without losing the original source code formatting.

In particular for the encoding of Java source code as XML a lot of approaches are available. Two of these approaches are presented in depth to show the main differences between the representation and annotation concept. The following listing 2.14 contains the original source code of a Java class that will be encoded as XML.

```
1 public class Person {
2   private String name;
3   public String getName() {
4     return name;
5   }
6 }
```

Listing 2.14: Java source code to be encoded as XML

Listing 2.15 shows the encoded Java source code in an XML representation called *JavaML* (Java Markup Language) that was developed by Badros [Bad00]. It uses the representation concept to map all language constructs of Java to XML elements. All constructs of the source code

are mapped to XML elements or attributes. For example, the source code `public class Person` (listing 2.14, line 1) is mapped to the XML element `class` (listing 2.15, line 1) with the attributes `name="Person"` and `visibility="public"`. Listing 2.15 is simplified by leaving out a few XML attributes for reasons of clarity. JavaML typically encodes also information about the location of each statement in the original source code (using line, end-line, column, and end-column attributes), adds an identifier to each statement that allows to reference it using the `idref` attribute, and is also capable of representing Java comments.

```
1  <class name="Person" visibility="public">
2    <superclass name="Object"/>
3    <field name="name" visibility="private">
4      <type name="String"/>
5    </field>
6    <method name="getName" visibility="public">
7      <type name="String"/>
8      <formal-arguments/>
9      <block>
10       <return><var-ref name="name"/></return>
11     </block>
12   </method>
13 </class>
```

Listing 2.15: JavaML representation of listing 2.14

An instance of the annotation concept is *srcML* (Source Code Markup Language) [MCK02, MCK04] which allows to markup Java, C/C++, and AspectJ source codes using XML. The following listing 2.16 contains the result of the annotation of the example Java source code. All constructs and also the formatting of the original source code are preserved and XML elements are put around the Java constructs. In line 1 of listing 2.16 the start of the Java class is marked-up using a `class` element, the visibility of the class is marked-up with a `specifier` element and the name of the class with a `name` element.

```
1  <class><specifier>public</specifier> class <name>Person</name> <block>{
2    <decl_stmt><decl><type>private <name>String</name></type> <name>name</
         name></decl>;</decl_stmt>
3    <function><type>public <name>String</name></type> <name>getName</name><
         parameter_list>()</parameter_list> <block>{
4      <return>return <expr><name>name</name></expr>;</return>
5    }</block></function>
6  }</block></class>
```

Listing 2.16: srcML representation of listing 2.14

As srcML preserves the formatting of the original source code, the resulting XML looks not as structured as the JavaML representation in listing 2.15. This might be seen as a drawback, but as most of the time the XML data is processed by XML tools and not by humans, this formatting

issue does not matter. It is an advantage of the annotation concept that the original source code can be recovered with small effort by removing all XML elements. While it requires parsing the XML file and mapping it back to source code when using representation concepts like JavaML. Because of the additional effort, many instances of the representation concept do not support a regeneration of the original source code from the XML representation at all.

One property is often missing in both concepts: the line numbers of the original source code elements are often not mapped to the XML. That makes it nearly impossible to reference from XML elements to the according elements in the original source code. This feature is missing in srcML and even though the original formatting of the source code is preserved, it is impossible to reconstruct the exact position of an element (line number and offset) in the original source code using standard XML technologies like XQuery or XSLT. JavaML in contrast has an optimal support for this feature encoding the starting and ending line and offset for each element of the source as XML attributes.

While the users of JavaML cannot influence the granularity of the mapping to an XML document, srcML offers options to mark-up literals and operators which are not be marked-up by default. Altogether, it can be asserted that the overall granularity of the resulting XML documents is higher for representation concepts like JavaML than for annotation concepts like srcML.

As mentioned at the beginning of this section, there are various approaches for the representation of Java source code as XML. Unfortunately, most of them are named *JavaML*. JavaML by Badros [Bad00] presented in detail above and is superseeded by an enhanced version *JavaML 2.0* also by Badros [ADB04]. JavaML by Mamas and Kontogiannis [MK00] follows the representation concept and is very likewise compared to JavaML by Badros. The separability of JavaML by McArthur, Mylopoulos, and Keith Ng [MMN02] is the usage of a multi-weight parser which allows to remain some syntactic constructs unparsed. This relates to the mark-up options of srcML but based on the representation and not the annotation concept. The *Extensible Software Document Markup Language* (XSDML) by Maruyama and Yamamoto [MY04] is conceptually not bound to, but currently implemented for Java. XSDML uses the annotation concept as srcML and is in contrast to srcML capable to encode line and offset information. Also additional semantical information such as fully-qualified names and references can be encoded. These properties make XSDML an intersection of srcML and JavaML.

Most XML formats for source code representation are often designed for *one* programming language, like JavaML for Java. A few support multiple source languages, like srcML (Java, C/C++, and AspectJ). Other approaches such as *OOML* (Object Oriented Markup Language) [MK00] use

a higher level of abstraction and allow to map different object-oriented languages to one XML representation. The *Extensible Common Intermediate Language* (XCIL) [KBSD04] is also developed to provide a common semantic representation that is independent of the target language. Therefore all structural elements are based on corresponding UML definitions and are mapped to XML. Because of that "XCIL and XMI are virtually identical except with respect to their representation of elements from the action semantics" [KBSD04].

Another approach to encode source code as XML—that relates to the representation concept—is to map an abstract syntax tree (AST, see section 2.4.4.3 for details) to XML [ZK01]. As ASTs mostly contain the line numbers and offsets for all statements of the underlying programming language, these information can be mapped to XML attributes. Only semantically relevant parts of the source code (like statements and identifiers) are represented in the AST. Comments and blank lines are not part of it and can therefore not be mapped to the XML what prohibits reconstruction of the original source code. An advantage of this approach is the very high granularity of the AST and the resulting XML, allowing to perform detailed analyses.

XML is not only used for the representation of static software artefacts (like source code or architectural models) but also for the representation of graphs like control flow and data flow graphs. One approach for the representation of graphs as XML is presented by Al-Ekram and Kontogiannis [AEK05]. They propose different XML formats for each kind of graphs: *Control Flow Graph Markup Language* (CFGML), *Program Dependence Graph Markup Language* (PDGML), and *Call Graph Markup Language* (CGML). All of them are based on an intermediate XML format called FactML that is used to unify different input formats like JavaML and OOML.

A general approach for the representation of any kinds of graphs as XML is the *Graph Exchange Language* (GXL) [HWS00] that is developed to enable interoperability between software reengineering tools and components. The tool XOgastan [APMV03] permits to transform the AST created by the C/C++ compilers of the *GNU Compiler Collection* (GCC) to a GXL document. XOgastan also includes special features like the extraction of control flow graphs. Hence, one application of GXL is the representation of ASTs and control flow graphs. GXL is also used as backend for an "infrastructure that supports interoperability among reverse engineering tools" [KMP07] what is important as interoperability is a "contributing factor to the lack of adoption of available [reengineering] infrastructures" [KMP07]. Using approaches like XML in general and GXL in particular, it is also possible to encode dynamic software behaviour as traces of program executions [MW03].

2.4.4 Other Common Approaches

The representation of software artefacts is not limited to high-level formats such as UML and metamodels or to XML-based formats which are described in the preceding sections. Other common approaches for the representation of software artefacts are presented in the following sections: the usage of plain source code (section 2.4.4.1), intermediate formats (section 2.4.4.2), and tree structures (section 2.4.4.3).

2.4.4.1 Plain Source Code

Source code is not directly executable—it always requires a compiler or at least an interpreter to execute it. Therefore, source code is strictly spoken nothing but the serialisation of software or a representation format. Hence, the most obvious representation format for software is source code itself. To differentiate between source code as a target of software analysis and source code as representation format, the term *plain source code* is used for the latter.

Plain source code can be used for similar purposes like other representation formats that are capable of reflecting source code in depth. Plain source code is also used to query it for anomalies using regular expressions or other lexical and syntactical approaches. See section 3.2 for details regarding this.

2.4.4.2 Intermediate Formats

Source code is not always directly compiled to executables or executed using an interpreter. Intermediate formats are located between the source code level and the actual execution of a program. One prominent example of an intermediate format is the Java bytecode that is the result of the compilation of Java source code. The Java bytecode (stored in class files) cannot be directly executed and needs to be executed by a *Java Virtual Machine* (JVM) instead [LY99]. The same concept is used by Microsoft's .NET platform which is a pendant to the Java runtime environment. Microsoft's intermediate format is called *Common Intermediate Language* (CIL) and fulfils the same role as the Java bytecode: it is a platform-independent representation format of the underlying source code that can be executed by any virtual machine that is capable of the intermediate format.

Intermediate formats can serve as a basis for multiple programming languages. Tool vendors only need to build compilers that transform the particular programming language to the intermediate format instead of providing more complex compilers that generates executables. Microsoft's .NET platform extensively uses this approach and transforms all .NET programming languages (for instance C# and VB.NET) to the Common Intermediate Language (CIL). As the specifications of CIL and of the Java bytecode are publicly available, third-party programming languages are able to make use

of the intermediate formats and their runtime environments. For example, IronPython[10] allows Python code to run on the .NET platform and JRuby[11] allows to run Ruby code using a JVM[12]. This property of intermediate formats can be of interest regarding software analysis. As the intermediate formats provide an abstraction from the underlying programming language an analysis that is based on an intermediate format is not bound to a concrete language.

For example, the *Byte Code Engineering Library* (BCEL) [BCE06] allows to access Java bytecode and acts as "a convenient possibility to analyze, create, and manipulate (binary) Java class files [...]. Classes are represented by [Java] objects which contain all the symbolic information of the given class: methods, fields, and byte code instructions, in particular." Using BCEL, it becomes easily possible to use Java bytecode as a starting point for the software analysis [HP04, MY04]. Particularly with regard to closed-source software—where only the Java bytecode is available—the usage of intermediate formats gets interesting.

2.4.4.3 Tree Structures

The structure of a programming language is predefined by its grammar that defines the allowed language constructs and their combinations. The recognition of the grammar in a piece of source code is called parsing. The result of the parsing process is a *Parse Tree* which is a hierarchical representation of the derivations of the source code from its grammar. An *Abstract Syntax Tree* (AST) is a more economical representation of the source code, abstracting out the redundant grammar productions of the parse tree. These are for instance comments or grouping parentheses that are already expressed by the tree structure. Parse trees and abstract syntax trees are the most important tree-based data structures for the representation of source code. These trees are integral parts of nearly all compilers and interpreters as intermediate representations of the program. The trees are the abstraction of the source code in terms of the language's grammar and hence strongly dependent on the underlying programming language. An example of an AST (encoded using XML) is provided in listing 5.4.

The transformation process from the source code to an AST consists mainly of a lexical, syntactical, and semantical analysis. In the first step, the lexical analysis is performed and the source code is converted into a token stream. Each token (also called symbol) represents a single expression of the underlying grammar and gets classified (operators, identifiers, ...). In the second step the token stream is converted into a parse tree that represents

[10] `http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython`

[11] `http://jruby.codehaus.org`

[12] `http://www.robert-tolksdorf.de/vmlanguages.html` provides a list of about 200 different projects using Java bytecode respectively the JVM as their runtime environment.

the structure of the program. During this convertion, the syntax of the program gets validated and a differentiation between declarations, statements, expressions, etc. is carried out. Finally, the parse tree is converted into an AST that is a finite, labeled, and directed tree, where the interior nodes of the tree represent the non-terminals and the leaves terminal symbols of the grammar. During the semantical analysis a symbol table is created in which each identifier of the source code is associated with information relating to its declaration or appearance in the source, like its type and scope.

Chapter 3

Related Work

This chapter covers related work regarding software analysis in the area of pattern detection in general and approaches specific to the detection of bad smell in particular. Approaches based on source code metrics (section 3.1), regular expressions (section 3.2), and logic programming (section 3.3) are presented followed by domain-specific languages (section 3.4), XML-based approaches (section 3.5), and other common approaches (section 3.6). Finally, a summary of this chapter is given and the results are discussed (section 3.7).

Each of the sections is made up of a few representative approaches which are presented by giving examples and discussing their properties for example regarding matching power, robustness, reusability, and limitations. Their technical realisations are not discussed in depth but mentioned briefly. The main focus is set to the detection of patterns which can be discovered using static code analysis. This is done as the software analysis framework presented and discussed in the chapters 4 and 5 is also focused on static analysis.

3.1 Source Code Metrics

This section presents approaches for the detection of pattern using source code metrics. (The terms bad smell and metric have been introduced in section 2.1.) Even though metrics cover software measurement while bad smells are related to the structure of source code, useful intersections between both patterns exist and are described in the following. The result of a metric is by default a descriptive value that should be linked to a boundary value to become prescriptive [FY94]. Applying the metric *Lines of Code* to a Java class can lead to a descriptive result of 1500 lines of code. If a Java class is considered to be too long if it contains for example more than 1000 lines of code, this boundary value is linked to the metric which can now be used to find instances of too large classes.

The connection between metrics and bad smells is that a certain boundary value can indicate the instances of bad smells. The example in the preceding paragraph shows that the metric *Lines of Code* can be used to find instances of the bad smell *Large Class*. Hence, metrics with a linked boundary value might be seen as a subset of bad smells as they also allow to detect structures in the code that should be refactored. The other way around, also bad smells might be seen as a subset of metrics as the number of found smells often equals the result of a metric.

An immense number of metric tools[1] for all kinds of target languages but *without* automatic smell detection is available. Only a small number of approaches make use of source code metrics for the *automatic* detection of bad smells [AK05, CMM04, SSL01]. But the scope of metrics regarding smell detection is limited as they "do not comprise the full spectrum of possible smell symptoms and still are uncertain" [WP05]. Hence, more advanced approaches—like those discussed in the following sections—are required for a powerful smell detection. These approaches are focused on the internal structure of the source code and are not limited to the measurement results of metrics.

3.2 Regular Expressions

Regular Expressions (RE) are expressions describing sets of strings and are used for pattern matching. REs feature their own syntaxes to express the patterns to match. The most known regular expression syntax is the one of the programming language Perl [Per07]. The basic concepts of regular expressions—wildcards, alternation, grouping, quantification, and metavariables—are extended by Perl with character classes, modifiers, backreferences, and replacements amongst others.[2] Perl-compatible regular expressions are nowadays widely adopted and implemented by nearly all newsworthy programming languages like Java, Python, Ruby, and the .NET platform.

The term "regular expression" has different meanings for the theory of formal languages and for pattern matching. Regarding formal languages, regular expressions describe the according language to the grammar the formal language is based on. Most programming languages are based on context-free grammars which are located at the type-2 level in the Chomsky–Schützenberger hierarchy [CS63]. Additionally to type-2 grammars, most definitions of programming languages make use of static semantics which provide additional contextual constraints for the language. REs regarding pattern matching are type-3 languages according the Chomsky–Schützenberger hierarchy and therefore less expressive. Hence, the expressive power of regular expres-

[1] For example: `http://metrics.sourceforge.net`, `http://www.clarkware.com/software/JDepend.html`, and `http://www.mmsindia.com/jstyle.html`

[2] `http://search.cpan.org/dist/perl/pod/perlre.pod`

sions is not sufficient to match patterns of programming languages which are based on type-2 grammars.

The query power of RE-based tools like `grep` or `AWK` and strict lexical approaches in general is too limited regarding the source code domain. "General string-searching tools can handle only trivial queries in the context of source code. Based on regular expressions, these tools do not exploit the rich syntactic structure of the programming language" [BMG+94]. Therefore, the core concepts regarding pattern matching of regular expressions have been adopted by special-purpose languages which allow pattern detection in source code of programming languages based on context-free grammars.

One of these languages is part of the source code search system SCRUPLE (Source Code Retrieval Using Pattern Languages) [PP94a]. Its pattern-based query language can be used to specify complex structural patterns of source code as it works on a syntactical instead of a lexical representation of the source code. The pattern language provides flexibility regarding the degree of precision to which a code structure is specified.

The query language is an extension of the programming language under investigation—currently C. The extensions include regular expression-like meta-variables which can be used as substitutes for syntactic entities in the programming language. Such as statements (`@`), declarations (`$d`), expressions (`#`), functions (`$f`), and variables (`$v`). The pattern-matching engine of SCRUPLE searches the source code for fragments that match the patterns. Listing 3.1 shows two examples of the pattern language. The pattern in line 1 matches three `if` statements that follow one after another. The pattern in line 2 finds all functions that have references to the identifier `foo`.

```
1 if @ #; if @ #; if @ #;
2 $t $f_x <foo> ($v*) { @* }
```

Listing 3.1: Example of the pattern-based query language of SCRUPLE

While the query language is quite powerful, it is specific to the underlying programming language. There is no abstraction between queries and the source code to query. Hence, applying existing patterns to a new programming language requires additional effort.

Another language making use of the core concepts of regular expressions is *TAWK* [AG06] which is based on the `AWK` language. TAWK uses a language-independent pattern syntax which combines the lexical power of `AWK` with matching support for abstract syntax trees. "Retargeting to a new language requires [...] no special effort [...] to the pattern matcher itself, since it is language independent" [AG06]. This paper also provides a comparison of different further lexical and syntactical languages and software analysis systems like LSME (Lightweight Source Model Extractor) [MN95],

SCRUPLE [PP94a], and GENOA [Dev92] regarding their expressive power, programming power, robustness, and speed.

3.3 Logic Programming

This sections covers approaches for the pattern detection that make use of logic programming and *Logic Meta-Programming* (LMP). While the first is the usage of logic programming languages (typically variants of Prolog) for analysing and/or transforming software artefacts [AK07], LMP is a combination of logic programming and the analysis target itself. Both approaches require fact-based representations of the underlying programming languages to express analyses using logic rules and predicates. As this thesis attends software analysis, the transformation capabilities of logic programming are not taken into account.

An example for the usage of logic programming is *JQuery*—a query-based browser for Java source code [Vol06]. Queries are formulated using a modified version of the Prolog-like logic programming language *TyRuBa* [TyR06]. "A set of TyRuBa predicates and rules have been defined that make up the JQuery query language. These predicates operate on a fact database that is generated from [...] abstract syntax trees (for .java files) and parsing Java bytecode (for .class files)."[3]

Listing 3.2 shows a query that makes use of the JQuery predicates to find all public getter methods of all Java classes. First all classes are selected and bound to the variable `?C`, afterwards all methods of the current class are bound to `?M`. Finally, all methods whose name matches the regular expression `/^get/` and whose modifier is `public` are returned.

```
1 class(?C), method(?C, ?M), re_name(?M, /^get/), modifier(?M, public)
```

Listing 3.2: JQuery query to find all public Java getter methods

An approach for software analysis based on logic-meta programming is the *Generic Transformation Language* (GenTL) [SAK07, Gen07]. Although its name states that GenTL is a transformation language, it also provides analysis capabilities. GenTL make use of logic-based programming and extends it by so called *Concrete Syntax Patterns* (CSP)[4]. CSPs are snippets of the programming language to analyse which contain meta-variables. Those meta-variable are logic predicates and act as placeholders for expressions of the underlying programming language. Currently, GenTL is adapted for Java but its core concepts are language-independent [AK07].

Listing 3.3 provides two GenTL examples. In line 1 a pattern is used to find all `if` statements. The pattern in line 2 finds all classes and binds them

[3] `http://jquery.cs.ubc.ca/documentation/overview.html`
[4] `https://sewiki.iai.uni-bonn.de/research/gentl/concepts/predicates`

to the user-defined meta-variable `?allClasses`. "Since CSPs match at the AST level, matching is not restricted to lexical structures" [AK07]. Hence, the pattern in line 1 matches also the source code `if (done) return;` which does not include the curly braces of the pattern. As CSPs do not need to specify all elements of the matched element, the pattern in line 2 also matches abstract classes even though this keyword is not contained in the pattern.

```
1 if (?expr) { ??statements }
2 ?allClasses is [[ class ?classname { ??class_members } ]]
```

Listing 3.3: Two GenTL examples: combination of logic predicates and syntax patterns

The concept of LMP combines logic predicates with the syntax of the analysis target. The resulting queries look much more like the analysis target itself. Hence, they are easier to understand than pure logic-based queries like those of JQuery. "Thus the concept of meta-variables is all that programmers have to learn in addition to mastering the analysed language" [SAK07].

A lot more approaches than the presented ones are based on logic programming: SOUL [TM03] is a variant of Prolog and used for the detection of bad smells like *Obsolete Parameter* and *Inappropriate Interfaces*. ASTLOG [Cre97] is a logic-based language for examining abstract syntax trees. Even though ASTLOG is a Prolog variant, it avoids the overhead of translating the source code into the Prolog database and works directly on abstract syntax trees. CodeQuest [HVdM06] combines the usage of logic programming and database systems. This is done by mapping Datalog queries (also a Prolog-like language) to SQL. A comprehensive overview of the available logic-based approaches regarding source code analysis and transformation (amongst others CodeQuest and JQuery) provides [KHR07].

3.4 Domain-Specific Languages

Domain-Specific Languages (DSL) are programming languages which perform specific kinds of tasks. In contrast to general-purpose programming languages the scope of DSLs is limited by design. For just that reason, DSLs are superior in their domain compared to general-purpose languages. This sections presents a couple of languages specific to the domain of pattern detection in software artefacts.

Jackpot [Jac07] is an extension of the open-source development environment *NetBeans* [Net07] which adds extensible refactoring capabilities for Java source code to the IDE. User-defined refactorings can be contributed by writing new Java classes according to the Jackpot API or by making

use of Jackpot's rule language[5]. This language is not intended to be a complete language, but it serves the need to define pattern matching and replacement for Java source code. The basic construct of the language is `<pattern> => <replacement>;`. This read as: if the pattern matches, substitute it with the replacement. The pattern consists of Java constructs and meta-variables to provide wildcard-like matching of the source code. Listing 3.4 shows two examples: the pattern in line 1 shows how boolean expressions can be simplified by applying DeMorgan's Theorem. The example in line 2 uses the optional guard expression `::` which allows to formulate conditions under which a replacement should happen. In this case: if the object bound to the meta-variable `$object` is an instance of the type `$type`, the casting of this object can be removed as it is not necessary.

```
1 !($a <= $b) => $a > $b;
2 ($type)$object => $object :: $object instanceof $type;
```

Listing 3.4: Two examples using the Jackpot rule language

Even though the Jackpot rule language is a transformation language and no pure query language (beside the workaround to replace the pattern by itself) it is a demonstrative example for a domain-specific language related to software analysis.

The *Program Query Language* (PQL) [PQL07, MLL05] allows to express queries regarding sequences of events associated with a set of related objects in an application-specific context. The focus of PQL is software behaviour instead of software structure as it allows to track method invocations and accesses in related objects. Therefore PQL abstracts the program execution as a trace of events. PQL queries are patterns to be matched on the execution trace and actions to be performed upon matches. A matching query is a set of objects and a subsequence of the trace that together satisfy the pattern.

PQL is currently adapted for the analysis of Java systems and therefore PQL queries partially look like pieces of Java source code. Listing 3.5 shows a query which finds all occurrences where a Java OutputStream object is not shut done correctly before the surrounding method ends (lines 2–5). This is expressed as an object that is instantiated (line 3) but not terminated using its `close` method (expressed by the sign ~ in line 4). PQL is not a pure query language and can also be used for transformations. In this example the missing call to the method `close` can be added at the end of the method under investigation (line 6).

To enable those high-level queries, the PQL system hides a lot of complexity: PQL queries are represented by state machines and all inputs and results for the static analysis are stored in a relational database. Inter-

[5] `http://jackpot.netbeans.org/docs/rule-language.html`

```
1 uses object OutputStream o;
2 matches {
3   o = new OutputStream();
4   ~o.close();
5 }
6 executes o.close();
```

Listing 3.5: Simplified PQL query to detect potential resource leaks

nally, static analyses are performed by translating PQL queries into Datalog (a subset of Prolog) and using the database to resolve the queries.

While the most other software analysis systems permit patterns to be matched only against source code, a key contribution of PQL is its pattern matcher which "combines object-based parametric matching across widely-spaced events" [MLL05].

The originators of SCRUPLE (see section 3.2) found out that "one of the fundamental problems designers of source code querying systems face is the lack of good underlying models to represent source code information and to express queries. [...] We found that no satisfactory choice for the underlying model to represent program information was available" [PP94a]. The desired source code query system should be able to answer queries based on global structural information, on syntactical structure, and on program flow information. Most approaches lack of either a powerful query language or adequate modeling power due to "the absence of clean formalisms for modeling and querying source code" [PP94b].

To address these issues a completely different approach is chosen: modeling source code using an algebra. The *Source Code Algebra* (SCA) plays the same role for source code as the relational algebra plays for relational databases. The algebra is developed as a theoretical foundation for a powerful source code query system [PP94b, PP96]. The benefits of using SCA include the integration of structural and flow information and the ability to process high-level source code queries using SCA expressions. Two sample queries are shown in the following.

- Find all functions defined in the file `analyzer.c`:
  $funcs(pick_{name='analyzer.c}(FILE))$
- Find all functions directly or indirectly called by the function `sort`:
  $closure_{calls}(pick_{name='sort}(FUNCTION))$

Even though the domain model of SCA depends on the underlying source code and must be adapted to the specifics of each programming language, the SCA query language itself is domain-independent. This is a valuable feature and essentially means that an implementation of a SCA query processor works unchanged across different SCA domain models [PP94b].

3.5 XML-based Approaches

As seen in section 2.4.3 a lot of approaches for the representation of software as XML exist. All levels are covered: high-level and full-detail XML representations are available. The range of XML-based approaches to process XML documents and to locate patterns in these documents is also wide. This section first introduces the most important APIs for programmatical access to XML documents. The application of these APIs to the software analysis is only briefly described as they provide no declarative query possibilities. Hence, the focus is set to declarative processing languages for XML. First the languages TQL and XDuce are presented, followed by approaches that make use of the standardised XML processing languages XPath and XQuery for the analysis of software.

The *Document Object Model* (DOM) "is a platform- and language-neutral interface that [allows] to dynamically access and update the content, structure and style of [XML] documents" [DOM05]. The DOM tree is a representation of the underlying XML document and can be traversed and transformed arbitrarily. Implementations are available for nearly all programming languages, but the drawback of DOM is its high memory requirement as the complete DOM tree must be kept in memory at once.

The opposite approach is the *Simple API for XML* (SAX) [SAX04] which provides a mechanism for read-only access to the data of XML documents. SAX traverses XML documents once from the start to the end and calls user-defined callback methods on occurring events. This event-driven approach makes SAX faster than DOM and features a small memory footprint. On the other hand programming with low-level API of SAX is non-trivial and it is not possible to change the XML document.

The *Streaming API for XML* (StAX) [StA04] is a Java-only API which "gives parsing control to the programmer by exposing a simple iterator based API and an underlying stream of events. [It allows a] developer to ask for the next event (pull the event) rather than handling the event in a callback" [StA04]. This design is a mixture of DOM and SAX as it allows to navigate in the XML document comparable to DOM and features adequate memory consumption and good speed like SAX.

There is no known approach directly deploying one of the APIs for the task of finding patterns in software artefacts, even though their usage is often recommended for this purpose [APMV03, ZK01, AEK05, Bad00]. If these APIs are used, they are wrapped into specialised high-level APIs for software analysis [MY04]. The situation is summarised by [MY04]: "The standard APIs (e.g. DOM and SAX) are of course convenient for writing code, independent to a specific programming language but too primitive for most developers when they build tools in practice."

Therefore the focus is set to declarative XML processing languages. First, two academic approaches are presented whose main targets and applications are not software analysis, but which provide properties to be used for software analysis tasks.

The *Tree Query Language* (TQL) [CG04] is a language to query semistructured data like unordered forests with labelled nodes. TQL can also be used to query XML trees and is therefore discussed in this section. Papers regarding TQL and a reference implementation can be found online [TQL03]. TQL does not follow an approach limited to path-based pattern matching, but also incorporates logic-based elements. This makes TQL queries more declarative and less operational than queries in comparable languages like XPath. Listing 3.6 shows an example to clarify this feature. The query returns all elements which are located inside of a `book` element (`.bib.book.$elements`) and which contain a `firstname` element whose content matches the string `Homer` (`.firstname[Homer]`). Even though the equivalent XPath expression `$bib/book/*[.//firstname[. = "Homer"]]` is a little bit shorter, it lacks of readability compared to the TQL one.

```
1 from $bib |= .bib.book.$elements.firstname[Homer]
2 select $elements
```

Listing 3.6: TQL query which combines path-based and logic elements

The expressive power of TQL is comparable with XPath but lesser then XQuery which is Turing-complete. Nevertheless TQL could be used to detect patterns in XML representations of software artefacts. The crucial reason *not* to use TQL, is its unordered data model—while the data model of XPath respectively XQuery keeps all nodes in document order. This feature is important especially regarding software analysis when expressing queries which compare the location and/or order of software constructs. This limitation is known and future versions of TQL will resolve it [CG04]. Another limitation of TQL is its missing support for user-defined functions and modules.

While TQL links elements of logic programming with tree data structures like XML, the following approach joins XML and regular expressions. *XDuce* is a statically typed XML processing language with support for regular expression pattern matching [HP03b, HP03a, XDu05]. XDuce is based on so-called regular expression types which correspond to the schema of the XML document. Hence, XDuce can make use of existing schema definitions in the DTD format. It is also possible to define own types like shown in listing 3.7. The type `Addrbook` in line 1 consists of a name, an address, and an optional telephone number which are defined in lines 2–4. Type definitions of XDuce are at about the same level of expressiveness as DTDs but not as powerful as XML schema definitions.

XDuce supports user-definition functions and pattern matching with a single-match semantic. This is demonstrated in listing 3.8 in which the

```
1 type Addrbook = addrbook[(Name,Addr,Tel?)*]
2 type Name     = name[String]
3 type Addr     = addr[String]
4 type Tel      = tel[String]
```

Listing 3.7: XDuce: definition of a regular expression type

function `create-telephone-list` is defined. The function makes use of the types defined in the preceding listing and takes a sequence of one or more `Addrbook` items as input and returns a sequence of name and telephone items. Each item of the input sequence is bound to the variable `item` and its content is evaluated in lines 2–6. If the `Addrbook` type consists of name, address, and the optional telephone number (line 3), the values of the name and telephone fields are returned (line 4). In this case and also in the case that the current address contains no telephone number (lines 5 and 6) the function is called recursively with the residual items as argument. This is the practical consequence of XDuce's single-match semantic: the first matching item is taken and further items are processed recursively.

```
1 fun create-telephone-list (val item as Addrbook+) : (Name,Tel)+ =
2   match item with
3     name[val n], addr[val a], tel[val t], val rest
4       -> name[n], tel[t], create-telephone-list(rest)
5   | name[val n], addr[val a], val rest
6       -> create-telephone-list(rest)
```

Listing 3.8: XDuce function for the transformation of XML data

Even though XDuce supports user-defined functions, what is an advantage compared to TQL, and its query power might be adequate for most software analysis purposes, it lacks all those additional features standardised languages can provide: an active user community, multiple reference implementations, and ongoing development of tools and language concepts.

This is where standardised and declarative XML processing languages, mainly XPath, XQuery, and XSLT, come into the field. The first two of these languages have been introduced in detail in the sections 2.3.2 and 2.3.3. As this thesis is focused on software analysis and not on the transformation of software, the presentation of approaches making use of XPath and/or XQuery is preferred over those using the transformation language XSLT. For more considerations on XQuery vs. XSLT regarding software analysis see section 4.4.

PMD [PMD07] is a tool for scanning Java source code for potential problems. It provides a set of predefined rules for the detection of source code anomalies and two ways to define new rules. This can be done by providing new Java classes that traverse the AST or by writing custom rules using XPath [Cop05]. Listing 3.9 shows two example XPath expressions that op-

erate on an XML representation of the AST provided by PMD. The pattern in line 1 finds all `while` loops that omit the optional braces. This is expressed by searching all `WhileStatement` elements whose `Statement` element does not contain a `Block` element. The expression in line 2 finds occurrences of empty if statements by searching `IfStatement` elements without braces and without any contained statements (`EmptyStatement`) and by searching empty blocks (`Block[count(*) = 0]`) contained by `IfStatement` elements.

```
1 //WhileStatement[not(Statement/Block)]
2 //IfStatement/Statement[EmptyStatement or Block[count(*) = 0]]
```

Listing 3.9: PMD XPath expressions for the analysis of Java source code

PMD currently supports only XPath 1.0 expressions which are less powerful then XPath 2.0 ones. The reason is the XPath engine *jaxen* used by PMD which only supports XPath 1.0. Support for version 2.0 of XPath is not planned.[6] PMD is not only restricted to anomalies in Java source code, but the XPath expressions are additionally tightly coupled to the AST of PMD. If the underlying grammar changes, for example due to new features in upcoming Java releases, the XPath expressions must reflect these changes as they would also be visible in the XML representation of the AST.

More powerful analyses can be achieved by using XQuery instead of XPath as query language. The framework *XIRC* (XML-based Information Retrieval and Conversion) [EMOS04] is used by the open static analysis platform *Magellan* [Eic07] to support XQuery for application-specific analyses. As a case study, the analysis of *Enterprise JavaBean* (EJB) classes is used and XQuery expression are applied for checking structural properties of EJBs.

According to the EJB specification, all entity bean classes and contained methods must not be declared as final. Listing 3.10 shows an XQuery expression which find classes and methods that violate this EJB constraint. In the first line all entity bean classes are located using a path expression and bound to the variable `$ebs`. In the second line those classes and methods are returned that are misleadingly declared as final.

```
1 let $ebs := //class[./annotations//@type = "javax.ejb.Entity"]
2 return $ebs[@final = "true"] union $ebs/method[@final = "true"]
```

Listing 3.10: XQuery expression searching final EJB classes and methods

Even though a central concern of Magellan is the decoupling of analyses as "the same functionality for parsing and analyzing code is developed over and over again" [Eic07], this is not done consequently as the XQuery analysis

[6] `http://jaxen.org/faq.html`

functions are bound to the underlying XML data structure what limits their reusability.

The same critique applies to the framework *XCARE* [MFM04] which stands for XML-based Code Analysis and Reverse Engineering. XCARE is also based on XQuery and implements the following analysis operations: metrics, design critiques (a mixture of design defects and bad smells, the term was introduced by *RevJava*[7]), and reverse engineering operations. Listing 3.11 shows an XQuery expression to find public Java classes (represented as XML using the JavaML version of Badros [Bad00]) that are lacking a constructor. As seen with PMD and Magellan, the query directly accesses the underlying XML document which leads to the described drawbacks.

```
1 for $class in $document//class
2 where (($class/@visibility = "public")
3   and not(exists($class/constructor)))
4 return $class/@name
```

Listing 3.11: XCARE pattern of the design critique *Uninstantiable Public*

The *RefaX* refactoring framework [MMFA04] makes use of XQuery to check pre- and post-conditions for refactorings while the actual refactorings are implemented using the XML update language XUpdate [XUp00]. Even though RefaX is about software transformations and the detection of bad smells is explicitly not in the scope of RefaX, the design of the framework is also of interest regarding software analysis as it addresses the problems described with PMD, Magellan, and XCARE.

The main point of critique is that most of the available approaches "rely on their own, closed mechanisms for representing and manipulating source code information, which makes them difficult to customize, extend and reuse" [MMFA04]. Hence, the primary design goals of RefaX are scheme-independence (support different XML data models for the same programming language) and language-independence (refactorings should be applicable to different programming languages). This is achieved by varying levels of abstraction between the XML representation of the programming language and the refactorings respectively the pre- and post-condition checks. Access to the XML data is never performed directly, but each XML data access is encapsulated so that consumer of the data needs no further information regarding the underlying structures. This approach can also be adopted for software analysis.

3.6 Other Common Approaches

The methods presented above to find patterns in software artefacts cover a wide range of the available approaches. Other classes of common meth-

[7] http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm

ods are not presented in depth, but mentioned for reasons of completeness. Approximate matching of patterns can be achieved by approaches based on fuzzy logic [NWW01]. The detection of duplicate code (so-called clone detection) is also a common task for software analyses [BYM+98, KdMMB96] which can be combined with fuzzy logic [Wei05]. Clone detection can also take information into account contained in repositories of version control systems for source code. This allows for example to detect divergent changes over time [She05] and so-called change patterns [BGA06].

Abstract syntax trees and parse trees are important data structures regarding software analysis as nearly all approaches—also those presented in the preceding sections—require parsing the source code at first. Instead of using abstract syntax trees only as intermediate format to fill XML or fact databases for instance, the syntax tree can be used as primary data structure for the software analysis. Nearly all known approaches for the detection of bad smells are based on the direct use of abstract syntax trees. For instance, the TTCN-3 tool TRex [TRe07] which is used for the integration of the software analysis framework presented in this thesis, already features a pattern detection engine. It is based on traversing the AST and making use of related data structures like the symbol table and reference finder provided by TRex [Bis06]. The *Static Analysis Framework* of the Eclipse *Test & Performance Tools Platform* (TPTP) Project [Ecl07c] on which the pattern detection of TRex is built, also provides a set of 70 predefined code review rules for Java. The technical foundations are very close to the pattern detection of TRex: it makes use of an abstract syntax tree—in this case provided by the Eclipse *Java Development Tools* (JDT)—and detects anomalies in the source code by traversing this AST. Built on the same infrastructure is *CodeNose* [Sli05], an Eclipse plug-in specialised for the detection of bad smells in Java source code. Additionally to the access to the AST, a fact database is used to detect "more complex smells like Refused Bequest and Feature Envy" [Sli05].

FindBugs is a tool which uses "static analysis to look for bugs in Java code" [Fin07]. It is based on the concept of bug patterns—code idioms that are often errors—which are described more detailed in section 2.1 and in the according FindBugs paper [HP04]. As the tool's name already states, *Checkstyle* [Che07] is focused on the review of coding standards for Java. Checkstyle is included here, as it is extended to "find class design problems, duplicate code, or bug patterns" [Che07]. FindBugs and Checkstyle provide APIs to write new analysis function but only by plugging-in Java classes. While FindBugs and Checkstyle directly work on abstract syntax trees, a metamodel is used to represent the source code for the code smell browser *jCOSMO* [vEM02]. It is capable to detect and visualise smells in Java source code but "it is possible to generalize our approach to other object-oriented languages" [vEM02].

3.7 Summary and Discussion

Table 3.1 summarises some of the query technologies for the pattern detection presented in the preceding sections. It includes languages specific to software analysis and/or transformation of the sections 3.2–3.4 but no general purpose languages like those for XML processing. This is done as the concepts of specialised languages differ significantly from those of general purpose languages. Additionally, general purpose languages feature no specifics regarding pattern detection and software analysis.

The table columns consist of the names of the approach in the first one, the class of the approach in the second column (regular expression, logic programming, or domain-specific language) and a short summary of the provided functionality. The last two columns state if the query language is independent regarding the underlying programming language and which language is the current main analysis target.

Name	*Class*	*Summary*	*Lang. Indep.*	*Analysis Target*
SCRUPLE	RE	Lexical approach using RE-based patterns to match ASTs	No	C
TAWK	RE	Syntactical approach using syntax patterns to match ASTs	Yes	C
JQuery	Logic	Predicates to query a fact database	Mostly	Java
GenTL	Logic	Predicates and syntax pattern to query a logic representation of ASTs	Mostly	Java
Jackpot	DSL	Meta-variables and Java constructs	No	Java
PQL	DSL	Based on objects and sequences of events	Hardly	Java
SCA	DSL	Algebraic source code model, high-level queries	Yes	C

Table 3.1: Different approaches for pattern detection in software

Table 3.1 points up that the presented analysis languages base on very different approaches and are therefore hardly comparable. Only a small subset of properties like the independence of the analysed programming language or the query power can be compared. But none of these properties can be mapped to a comparable scale and therefore the results would be unsatisfying.

A conclusion that can be drawn is that domain-specific language provide well ease of use as their scope is clearly laid out and the syntax is straightforward. Another remarkable outcome is that most of the query languages

specialised to one analysis target, make use of the underlying programming language itself. The Jackpot rule language for example uses Java statements like `instanceof` as shown in listing 3.4 what limits the language reusability by design.

Even though only a few approaches address the language-independence regarding the analysis target, most of the approaches provide flexible and powerful query capabilities. Support for user-defined queries without making programmatic use of provided APIs is also not common. Just a few approaches allow on-the-fly integration of queries like those based on XPath or XQuery, logic programming, or domain-specific languages. The experiences and issues of this chapter flow into the requirements and design of the software analysis framework which is described in the remainder of this thesis.

Chapter 4

Requirements and Design

This chapter describes the requirements and the design of a software analysis framework. In section 4.1 the requirements of the framework are presented. The transformation of the requirements into a concrete framework design and architecture is described in sections 4.2, 4.3, and 4.4. Finally, the overall design of the framework is discussed in section 4.5. The concrete implementation of a software analysis framework that fulfils the requirements and realises the design is presented in chapter 5.

4.1 Requirements for a Software Analysis Framework

Before going into the details of the requirements for a software analysis framework, definitions of the terms *software*, *analysis*, and *framework* are given and it is clarified how they apply in the context of this thesis.

According to the ISTQB, *software* is defined as "computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system" [Int06]. Regarding this thesis, the term software is used for software artefacts to be analysed. These software artefacts can represent source code or software on an abstract level as described in section 2.4.

Analysis in this context is the generic term for *dynamic analysis* and *static analysis* of software. The first is "the process of evaluating behavior, e.g. memory performance, CPU usage, of a system or component *during execution*" while the latter is an "analysis of software artifacts, e.g. requirements or code, carried out *without execution* of these software artifacts" [Int06]. Another difference between both kinds of analyses is that the information obtained from a static analysis is valid for all possible executions of the software while the result of a dynamic analysis is typically valid for the run in question [JR00].

Frameworks provide conceptual structures to solve complex problems. Software frameworks can be defined as a "set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations" [GHJV95]. Thus, frameworks provide reusable and extensible structures for application-specific problems. Customising frameworks is mostly done by inheriting from predefined structures or by writing plug-ins for extension points defined by the framework. Even though frameworks often make use of design patterns both concepts must be differentiated. While design patterns provide abstract solutions for common problems, frameworks provide concrete extensible implementations that address domain-specific problems.

Combining these three definitions results in a definition of a software analysis framework: it is a tool providing an infrastructure for the static and/or dynamic analysis of software artefacts. It can include predefined analysis plug-ins and adoptions for specific software artefacts. It allows to integrate new analysis plug-ins as well as adoptions for new software artefacts. As the focus of this thesis is the static analysis, the main task of a software analysis framework is to carry out static code analysis. "The tool checks source code, for certain properties such as conformance to coding standards, quality metrics or data flow anomalies" [Int06].

The following enumeration lists general requirements for a software analysis framework, sorted by relevance, and tagged with corresponding short names (like R5 for the fifth requirement) to be able to refer to each requirement:

R1 The framework must provide a way to describe patterns that occur in software artefacts (such as design patterns, bad smells, and metrics as described in section 2.1).

R2 The patterns must be describable in a generic way on a high level of abstraction.

R3 The language which is used for the description of the patterns must assure well comprehensibility of the patterns. Functional and declarative languages should be favoured over procedural ones.

R4 The description of patterns must be independent of specific analysis targets (for example a concrete programming language like Java or C) and specific representations of the analysis targets (for example a concrete representation format like an abstract syntax tree).

R5 The description of patterns must be human readable and directly executable by the framework at the same time.

R6 The framework must be able to execute the patterns to detect instances of these patterns in software artefacts. (Such an instance is called a match.)

R7 The framework must be capable to detect patterns in result of static analyses. It should also be possible to detect patterns in result of dynamic analyses.

R8 The pattern detection must work accurate. It should generate as few as possible false positive and false negative matches.

R9 The framework must be extensible regarding new types of patterns and also regarding new analysis targets (like new kinds of software artefacts or representation formats).

R10 It must be possible to add user-defined patterns to the framework to enrich the set of predefined patterns. Writing user-defined patterns should be possible without profound knowledge of the underlying representation format.

Some requirements are not explicitly mentioned in this list as they arise from the presented requirements. For example the need to perform the analysis in an acceptable space of time to encourage developers to make use of it. Other requirements are the possibility to group related patterns and to execute a subset of the available patterns and/or plug-ins instead of executing all patterns.

The requirements R1–R10 concern the framework core of describing and executing patterns and ways of extension. Requirements regarding user interfaces (UI) and usability are *not* in the scope of the requirements R1–R10. Instead, the application embedding the framework should provide an appropriate user interface which permits at least to select patterns to be executed and to display the result of an analysis in a suitable, graphical way.

The result of an analysis is composed of zero or more matches. Matches are instances of patterns detected by the analysis in the representation format. The result of the analysis should be presented by the embedding application in the analysis target and not in its representation format. Hence, representation formats needs to fulfil the following requirements to allow the mapping of matches from the representation format to the underlying analysis target where they originate from. The following requirements for representation formats assume that the analysis target is stored in one or more files whereat each file contains one or more lines.

For matches that encompass *at most one line*, representation formats must contain the following information for each differentiable entity (such as variable names or statements) of the analysis target:

- minimal set of information: filename and line number
- optimal set of information: filename, line number, start and end[1] offset (based on the start of the current line)

For matches that encompass *multiple lines*, representation formats must contain the following information for each differentiable entity of the analysis target:

- minimal set of information: filename and start and end line numbers or alternatively: filename and start and end offsets (based on the start of the current file)
- optimal set of information: filename, start and end line numbers, and start and end offsets (based on the start of the current file)

The subtitle of this thesis states that the software analysis is applied to detect bad smells in TTCN-3 test suites (see section 2.2.1 for details on TTCN-3). This leads to the following list of requirements for a software analysis framework regarding TTCN-3 (sorted by relevance):

T1 The framework must be customised for the static analysis of TTCN-3 source code.

T2 More specific, it must be adopted for the detection of bad smells in TTCN-3 source code.

T3 It must be capable to detect those bad smells described in the TTCN-3 code smell catalogue (see section 2.2.2 for details) that can be discovered in result of static analyses.

T4 The software analysis framework must be integrated into the TTCN-3 tool TRex [TRe07].

T5 It should be possible to customise the framework for the detection of test purposes in TTCN-3 source code or in behavioural representations of TTCN-3 source code. See section 6.3 for details regarding this.

Requirement R10 demands the possibility to add new user-defined patterns to the framework and consists of a number of subordinate requirements. They do not directly concern the framework core itself but the user interface of the embedding application. As the framework must be integrated into the TTCN-3 tool TRex (according to requirement T4) the following requirements regarding user-defined patterns must be considered.

[1] Equivalent to the *start and end* information is the *start and length* information.

U1 The application embedding the software analysis framework must allow to enter and execute user-defined patterns. Errors (like syntax errors) must be handled.

U2 It must present the result of the pattern detection in an appropriate way.

U3 It must be able to store user-defined patterns including pattern names.

U4 It must allow to select a subset of user-defined and predefined patterns for analysis runs.

U5 It should be possible to view, edit, delete, and rename stored user-defined patterns.

While this section lists abstract requirements for a software analysis framework and its customisation towards TTCN-3, the following sections transfer these requirements into a concrete design for the software analysis framework.

4.2 Design: A Layered and Extensible Architecture

This section presents the global architecture of the software analysis framework and addresses the requirements R4, R7, and R9. The most important requirement of these is requirement R4 which demands patterns to be independent of the analysis targets and their representation formats. Requirements R7 and R9 mainly demand an infrastructure which supports adding new analysis targets and new types of patterns.

The overall design of the framework is described by differentiating between the vertical and horizontal design. The vertical design makes use of the *facade* design pattern which provides a "unified interface to a set of interfaces in a subsystem. [The] facade defines a higher-level interface that makes the subsystem easier to use" [GHJV95]. A positive side effect of this design pattern is a loose coupling between the layers below the facade layer and upper layers using the facade.

Figure 4.1 gives an overview of the vertical, layered design of the framework. The boxes on the left side use abstract terms while the right side shows a concrete instance of the framework customised for the detection of bad smells in TTCN-3 source code. The first layer consists of an analysis target, TTCN-3 source code in this example. The second layer is a representation of the analysis target that acts as the primary data structure for the analysis.[2] The exemplified instance on the right side uses an abstract

[2]The representation format and the underlying analysis target might also be the same. For example, when analysing source code and using plain source code as representation format.

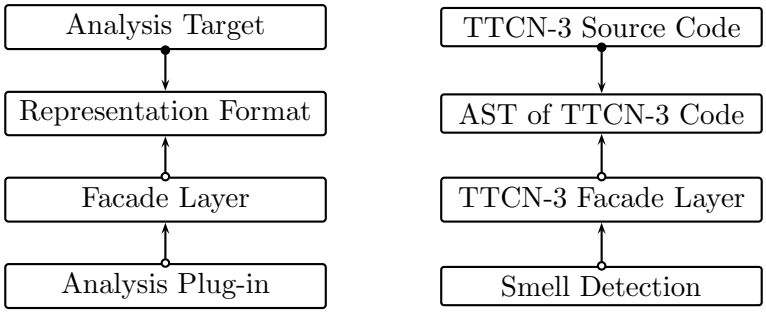

Figure 4.1: Vertical framework design: abstract on the left side, examplified on the right side

syntax tree for the representation of TTCN-3 source code. The third layer is an instance of the facade design pattern which is used for a unified access to the representation format. Towards the fourth layer the facade layer provides stable interfaces to ensure the reusability of plug-ins. Towards the layer of the representation format the facade layer must be customised for each format to actually be able to access it. The fourth layer consists of plug-ins that perform analyses. Plug-ins make use of the facade layer to access the analysis target. This layered architecture permits plug-ins to depend only on the facade layer and to be independent of the analysis target and its representation—and therewith fulfil requirement R4.

Plug-ins consist of a set of analysis functions and facade layers consists of a set of stable interfaces—respectively a set of functions implementing these interfaces. Both, the plug-in and the facade layer, can be topically grouped to improve the overview and reusability. Given a topical intersection of different analysis targets (for instance commonness between different programming languages), these similarities can be reflected by these layers. The smell detection plug-in for example can be grouped into generic smell detection functions and specific smell detection ones (for instance groups of functions for TTCN-3 specific and UML specific smell detection). This can also be done for the facade layer by grouping the interfaces into general interfaces and interfaces for specific analysis targets.

Note that in figures (like figure 4.1 and 4.2) regarding the framework architecture the arrow type $\circ\!\!\longrightarrow$ is used for any kinds of access and the arrow type $\bullet\!\!\longrightarrow$ for conversions (for instance conversions of software artefacts to representation formats).

The horizontal design of the software analysis framework is shown in figure 4.2. Extending the framework is possible by adding plug-ins such as the metrics and smell detection plug-ins in this example. As plug-ins are only bound to the stable facade interfaces, all plug-ins can be used for the analysis of all targets. Customising the framework is possible by integrating new representation formats respectively analysis targets (UML models and

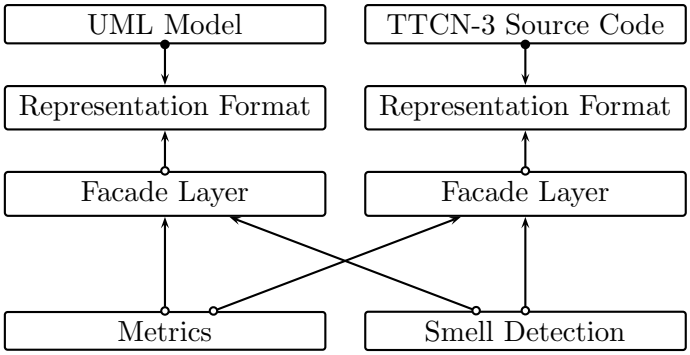

Figure 4.2: Horizontal framework design: plug-ins extend the framework and can be used independently of analysis targets

TTCN-3 source code in this example). This requires a customisation of the facade layer towards the representation layer to adapt the new representation format. These properties of the framework fulfil requirements R7 and R9.

The core of the software analysis framework is the application of the facade design pattern to create an abstraction layer between the representation layer and the analysis layer. Plug-ins are only coupled with the facade layer what results in a decoupling of the plug-ins from the underlying representation formats respectively analysis targets. If the analysis target and/or its representation format changes or new targets/representations are added, only an adoption of the facade layer is required—the stable interfaces of the facade layer and even more important the plug-ins remain untouched.

4.3 XML for the Representation of Software Artefacts

As discussed in section 2.4.3 the Extensible Markup Language (XML) can serve as representation format for different kinds of software artefacts. This includes abstract high-level software representations, fully detailed source code representations and even ones for data and control flow graphs. "This makes XML [...] a natural choice to be used as [...] representation format for program representations" [AEK05].

The software analysis framework also builds upon XML as a universal representation format. Reasons for this decision are the availability of many XML-based software representation formats and XML-aware languages like XQuery and XSLT [MMFA04]. Additionally, it is possible to transform nearly every data structure into XML without major effort. Examples regarding those transformations are given in section 2.4.3 by the transformation of ASTs, models, and graphs to XML.

XML also fulfils the requirements to serve as representation format for a software analysis framework (see section 4.1), as all information required for the mapping of matches from the representation format to the analysis target can be stored using XML elements or attributes—as long as these information are provided by the underlying representation format respectively analysis target.

4.4 XQuery for the Facade Layer and Pattern Description

This section presents a concretion regarding the design of the software analysis framework concerning the plug-in and facade layers. In the first step the requirements for both layers are summarised, followed by a discussion of technical realisation possibilities, and a concrete recommendation for the implementation.

Concerning the plug-in layer, the main focus is to fulfil the requirements regarding the description of the patterns (R1–R3, R5, R6, R8, and R10). The core requirements for the pattern description (R1–R3) mainly demand the possibility to describe patterns in a general, abstract, and comprehensive way. According to requirement R3, this could be achieved by using functional and declarative languages for the pattern description. Additionally, the patterns should be human readable on the one hand and directly executable by the framework on the other hand (R5). By executing the patterns, the framework must be able to detect instances of these patterns in the representation format of the analysis target (R6).

The requirements for the facade layer arise from the global architecture of the framework (section 4.2) and the design decision to use XML as representation format for the framework (section 4.3). Hence, the facade layer must fulfil two major requirements: it must provide stable interfaces towards the plug-in layer and it must be able to access the XML of the representation layer. The facade layer must be able to handle XML and therefore the usage of an XML-aware language is self-evident.

The most well-known, widely implemented, and standardised XML-aware programming languages are XSLT 2.0 and XQuery 1.0. The latter is described in detail in section 2.3.3 and its application regarding software analysis is discussed in chapter 3. XQuery is focused on querying XML while XSLT's domain is the transformation of XML. Both languages make use of XPath 2.0 but their concepts of doing this differ significantly. While XQuery extends the non-XML syntax of XPath with powerful constructs, XSLT uses an own XML-based syntax and incorporates XPath expressions. XSLT and XQuery are functional and declarative programming languages and support the concept of user-defined functions and modules which allows to group related functions into independent packages.

To demonstrate the different approaches and syntaxes of XSLT and XQuery an example is given. The goal is to transform the Simpsons example XML document of listing 2.6 from section 2.3 to the output shown in listing 4.1—once using XSLT, once using XQuery.

```
1 <episode number="1" season="1"/>
2 <episode number="2" season="1"/>
3 <episode number="401" season="19"/>
4 <episode number="402" season="19"/>
```

Listing 4.1: Simpsons episodes including episode and season numbers

Listing 4.2 shows the XSLT stylesheet for the desired transformation of the Simpsons example. For each `episode` element which is matched in line 1, a new `episode` element is created in line 2. The attributes are added in line 3–8 and the according episode and season numbers are selected in lines 4 and 7 using incorporated XPath expressions. The used XSLT syntax could be abbreviated, but to demonstrate the different syntaxes of XSLT and XQuery this elaborate syntax is chosen.

```
1  <xsl:template match="//episode">
2    <episode>
3      <xsl:attribute name="number">
4        <xsl:value-of select="@number"/>
5      </xsl:attribute>
6      <xsl:attribute name="season">
7        <xsl:value-of select="../@number"/>
8      </xsl:attribute>
9    </episode>
10 </xsl:template>
```

Listing 4.2: Using XSLT to create the XML output of listing 4.1

Listing 4.3 shows the equivalent XQuery expression which needs no further explanations as the language was already introduced in section 2.3.3. The usage of a non-XML syntax eases the readability and also the effort to write new queries.

```
1 for $episode in $xml//episode
2 return element episode {
3   attribute number { $episode/@number    },
4   attribute season { $episode/../@number }
5 }
```

Listing 4.3: Using XQuery to create the XML output of listing 4.1

Although it would be possible to combine XSLT and XQuery, the usage of one language for the facade and plug-in layer is favoured for reasons of simplicity. The functional capabilities of both languages are at the same level. But as software analysis is more about querying than about transforming

and as XQuery's non-XML syntax is better human readable than the XML syntax of XSLT, XQuery is the preferred language for the implementation of the facade and plug-in layers.

XQuery functions (respectively the function's signatures) serve as stable interfaces of the facade layer and the function's bodies perform the access to the XML of representation layer at the same time. All functions that belong to a concrete facade (such as a facade for the access to a concrete representation format) are grouped into one XQuery module. Listing 4.4 shows an example of a complete XQuery module providing the desired functionality of the facade layer. In the first line the namespace prefix `example-facade` and the according namespace are declared. Line 2 contains a declaration of an `external` XQuery variable which means that the content of this variable is provided by the XQuery processor prior to the execution of the query. The variable `$example-facade:root` contains the input XML data in the chosen representation format. In lines 4–6 the function `example-facade:get-classes()` is declared and covers two different purposes. First, the signature of the function serves as a stable interface. Hence, calling this facade function always returns all class definitions regardless of the underlying representation format. The second requirement is the actual access to the XML of the representation format. XQuery functions fulfilling both requirements are called *facade functions*. In this example, class definitions can be accessed through the module variable `$example-facade:root` and XML elements named `ClassDef` in line 5. Adopting the facade layer for a new representation format would only require renaming this XML element name.

```
1 module namespace example-facade = "unique-facade-namespace";
2 declare variable $example-facade:root external;
3
4 declare function example-facade:get-classes() {
5   $example-facade:root//ClassDef
6 };
```

Listing 4.4: Example XQuery module of the facade layer

XQuery modules are also used for the plug-in layer as containers for related analysis functions. These analysis functions are XQuery functions describing patterns in software artefacts. Listing 4.5 contains a sample module of the plug-in layer with one analysis function. Line 1 of listing 4.5 contains the standard header of every XQuery module: the declaration of the namespace and its prefix. In lines 3 and 4 the facade module of listing 4.4 is imported into the current module to be able to access the facade layer functions. Lines 6–9 contain the declaration of an exemplary analysis function which returns the number of functions for each class. The access to the analysis target respectively its representation format is fully channelled through the interfaces of the facade layer.

```
1 module namespace example-plugin = "unique-plugin-namespace";
2
3 import module namespace example-facade =
4   "unique-facade-namespace" at "facade.example.xquery";
5
6 declare function example-plugin:functions-per-class() {
7   for $class in example-facade:get-classes()
8   return count(example-facade:get-functions($class))
9 };
```

Listing 4.5: Example XQuery module of the plug-in layer

The requirements R1 and R2 demand the possibility to describe patterns in a generic way on a high level of abstraction. The usage of XQuery for the pattern description fulfils these requirements as the Turing complete [Kep04] language allows to express all kinds of patterns. A high level of abstraction is achieved through the facade layer which also permits the patterns to be as generic as possible by being independent of the underlying analysis target. XQuery is a functional and declarative programming language which improves the comprehensibility of the patterns and fulfils requirement R3. The syntax of XQuery is human readable on the one hand and also directly executable by an XQuery processor on the other hand—as demanded by requirement R5. The detection of pattern instances is done by executing XQuery functions which describe patterns. Hence, using XQuery for the pattern description also fulfils requirement R6.

The requirements that the detection must work accurate (R8) and the possibility to add user-defined patterns to a set of predefined patterns (R10) is also addressed. The latter can be achieved by an XQuery module for user-defined patterns and the possibility to add functions to existing modules. The accuracy of the pattern detection is mainly up to the author's care when creating new patterns.

After having solved the main technical issues by deciding to use XQuery and the introduction of XQuery-related vocabularies, a few words about the terminology are required. The term *pattern* is used as an abstract term for patterns in software artefacts referring to the requirements and the design of the analysis framework. In contrast, the term *function* stands for concrete XQuery functions describing such patterns. The same relation holds for the terms *plug-in* and *module*. While the first is an abstract term used mainly for the design and requirements, the term *module* stands for XQuery modules used for the facade layer and plug-in layer. The term *query* describes any XQuery expression that can be executed.

Plug-ins realised as XQuery modules consist of XQuery analysis functions. They appear in modules as flat and unstructured lists of functions. The user of a module could manually look into it and select the required functions. However, that would not work for a program as it cannot recognise the semantics of the functions. But as frameworks should smoothly interact

with other programs, a way must be provided to describe the structure of modules.

This is done by an XML document for each XQuery plug-in module. The XML document represents each analysis function of the module and the hierarchical structure of the module. The structure of the XML document itself is defined by an XML schema (`hierarchy.xsd`). An exemplary XML document conforming to the `hierarchy.xsd` schema is presented in listing 4.6. The root element `hierarchy` (lines 1–6) features the attribute `filename` to find the according XQuery module file. The attributes `namespace-prefix` and `namespace` allow to access the module with correct namespace information. The hierarchy itself is made up of arbitrarily nested `category` elements (lines 7 and 10) which contain `function` elements (lines 8 and 11) for each XQuery analysis function. Each `function` element features the attributes `name` used for a description and `xquery-name` which is the function name used in the XQuery module. The `function` elements can contain `parameter` elements (line 12) whereat each one represents one XQuery parameter of the analysis function.

```
1  <hierarchy
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:noNamespaceSchemaLocation="hierarchy.xsd"
4    filename="plug-in.example.xquery"
5    namespace-prefix="example-plugin"
6    namespace="unique-plug-in-namespace">
7    <category name="Root Category">
8      <function name="Number of Functions per Class"
9        xquery-name="functions-per-class"/>
10     <category name="Nested Category">
11       <function name="Another Analysis Function" xquery-name="function">
12         <parameter name="Limit" xquery-name="limit" type="integer"/>
13       </function>
14     </category>
15   </category>
16 </hierarchy>
```

Listing 4.6: Describing the structure of plug-ins using `hierarchy.xsd`

With the information provided by a hierarchy XML document it is possible to access all functions of the described module. An application can parse the hierarchy XML file and extract all required information to access the according XQuery module including its structure and the analysis functions provided by the module including their parameters and parameter types. The `hierarchy.xsd` schema is not bound to the software analysis focus of the framework and can be used to describe the structure of any kinds of plug-ins. The software analysis tool PMD [PMD07] presented in section 3.5 uses a similar approach to manage its rules: XML files[3] contain related rules including descriptions, examples, and XPath expressions. This strategy permits a completely declarative way to extend the frameworks with new rules.

[3] An example XML file of PMD can be found in this article: `http://www.onjava.com/pub/a/onjava/2003/04/09/pmd_rules.html`

As mentioned in section 4.1 the result of an analysis consists of matches whereat a match is an instance of a detected pattern. Analysis plug-ins return matches in order to allow the embedding application to display the matches in the underlying analysis target. Each analysis function is an XQuery function which returns XML nodes. These nodes constitute matches and reflect instances of patterns in terms of selections of the representation format (subtrees of the XML input document). Instead of returning complete matches only the information required for locating the matches in the analysis target needs to be returned. To enhance the reusability of the software analysis framework the way of returning matches is standardised by an XML Schema definition (`matches.xsd`). This schema reflects the requirements any representation formats needs to fulfil for the mapping of matches to the analysis target (these requirements are defined in section 4.1).

Instead of presenting the schema itself in detail, listing 4.7 shows an instance of an XML document which conforms to the schema. The root element (line 1) is called `matches` and holds an arbitrary number of `match` elements. In this example two `match` elements are part of the XML document (lines 4–7), each featuring a set of attributes allowing to locate match in the analysis target. These are the attributes `filename`, `offset`, and `offset-end`. Optionally the schema allows the additional attributes `line` and `line-end` to specify additional location information. The attribute `found-by` of a `match` element permits to recognise which match originates from which analysis function. For a unique mapping of the matches to the analysis functions the value of the `found-by` attribute should be the namespace prefix followed by the function's name (for example `example-plugin:functions-per-class`).

```
1 <matches
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:noNamespaceSchemaLocation="matches.xsd">
4   <match filename="foo.c" offset="17" offset-end="33" found-by="pattern1"/>
5   <match filename="bar.c" offset="44" offset-end="68" found-by="pattern2">
6     <caused-by offset="23" offset-end="27"/>
7   </match>
8 </matches>
```

Listing 4.7: The schema `match.xsd` defines how matches are returned

Each `match` element may have a child element `caused-by` to refer to a place in the analysis target which caused the current match. An example is shown in line 6 in listing 4.7. The usefulness of the caused by information can be explained with the bad smell *Unreachable Default* (see section 5.5). The actual smell is a default which is unreachable due to an else branch of an alt construct. This else branch can be reported as the place that caused the smell. It can be useful to always report such places when the actual smell and its root cause are different. Each `match` element may feature an optional attribute `return-value` which is not shown in the example and allows

to pass additional information from the analysis function to the subsequent application.

The detection of duplicates is a common task for any software analysis framework. Hence, this is reflected by the schema as it allows `duplicates` elements which contain two or more `match` elements representing the duplicates. See listing 4.8 for an example of three duplicates.

```
1 <matches
2   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:noNamespaceSchemaLocation="matches.xsd">
4   <duplicates>
5     <match filename="foo" offset="11" offset-end="17" found-by="pattern1"/>
6     <match filename="foo" offset="23" offset-end="29" found-by="pattern1"/>
7     <match filename="foo" offset="80" offset-end="86" found-by="pattern1"/>
8   </duplicates>
9 </matches>
```

Listing 4.8: Returning duplicate matches conforming to `match.xsd`

The `matches.xsd` schema reflects requirements for analysing software but is kept as general as possible and could be used by other plug-ins. As this schema is only bound to the plug-ins using it and not to the framework itself, other plug-ins might use own schemata to define their ways of returning matches.

4.5 Discussion of this Approach

As seen in the previous sections the general requirements R1–R10 are covered by the design and architecture of the software analysis framework. The requirements T1–T5 regarding TTCN-3 and those concerning user-defined pattern (U1–U5) are not yet discussed as they must be addressed by the application embedding the framework. Therefore these requirements are picked up again in the discussion of the framework implementation in section 5.9.

Reasoning about the pros and cons of the design and architecture of the software analysis framework reveals one main point of critique: too few specialisation respectively too much generalisation due to the usage of common technologies like XML and XQuery. Languages specialised for software analysis as presented in chapter 3 can be more powerful and intuitive to use. But the high degree of specialisation is also a drawback of these languages at the same time as they are tightly coupled with their underlying data structures and analysis targets. Exactly this is addressed by the presented framework architecture which permits a decoupling of analysis targets from the actual analyses. This allows to write analyses on a high level of abstraction what improves the reusability of analyses and permits eminent extensibility of the framework. Additionally, the readability of the patterns benefits from the abstraction.

The usage of XML as representation format permits to analysis any kinds of data as it is the "lingua franca" for information markup and exchange. But XML tends to bloat up the encoded information by being verbose (for instance element names are repeated in the closing tag) and this can slow down the analysis process. As a counter strategy, the storing of XML information in an XML-aware database can be considered. But the utilisability of this approach depends on the concrete analysis target. If the content of the analysis target changes often, the additional effort to keep an XML database synchronous with the analysis target might be considered as too high.

Chapter 5

Implementation and Test

The concrete implementation of a software analysis framework discussed in this chapter is called *XAF* which stands for "XQuery-based Analysis Framework [for Software]". To point out that the design of the framework and central parts of the implementation are not limited to the analysis of software, the appendix *for Software* is written in squared braces. The concrete implementation is focused on the detection of bad smells in TTCN-3 test suites and is therefore named *XAF for TTCN-3*. For reasons of simplicity the implementation is referred to as *XAF* instead of its full name.

This chapter is split up in the following sections: in section 5.1 the software technologies used for the implementation of XAF are introduced, section 5.2 describes possibilities of interaction between Java and XQuery also used for the implementation, section 5.3 covers the concept and implementation of the conversion of TTCN-3 source code to XML. The main focus of this chapter are sections 5.4 and 5.5 which cover the usage of XQuery for the TTCN-3 facade layer and XQuery for the pattern description of bad smells. The integration of the framework into the TTCN-3 tool TRex is presented in section 5.6, followed by the implementation of user-defined queries and according use cases in section 5.7. Section 5.8 covers testing XAF and a discussion of the implementation is given in section 5.9.

The source code of XAF is located at the CD-ROM added to the last page of the print version of this thesis. For a brief description of the contents of the CD-ROM, see also appendix B. Alternatively the latest version of the source code can be found in the Subversion repository of TRex which is located at `http://www.trex.informatik.uni-goettingen.de/svn/trex/`.

5.1 Underlying Software Technologies

This section gives an overview of the software used for the implementation of the XQuery-based Analysis Framework for TTCN-3. According to requirement T4 the framework should be integrated into the TTCN-3 refactoring and metrics tool *TRex* [TRe07]. TRex was initially developed as a

part of the master's thesis of Zeiss [Zei06] and provides "IDE functionality for the TTCN-3 core notation, and supports the assessment and automatic restructuring of TTCN-3 test suites by providing suitable metrics and refactorings"[1]. TRex is built up on the Eclipse platform [Ecl07a] and makes use of foundations provided by the bachelor's thesis of Kemnade [Kem05]. XAF is also integrated into TRex respectively the Java-based Eclipse platform. The Eclipse way to extend and integrate is to create new plug-ins. Extension points define where and how plug-ins can extend existing functionality and add new features. Therefore XAF plug-ins for TRex are provided to couple the framework with TRex. For the implementation of XAF plug-ins and the integration into TRex see section 5.6.

ANTLR (ANother Tool for Language Recognition) "is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions" [ANT07]. The grammatical structure of the TTCN-3 core notation (provided by the TTCN-3 specification [ETS07a]) was translated to the ANTLR grammar syntax. This allows TRex to make use of ANTLR 2.7 to parse the TTCN-3 source code and to transform it to an abstract syntax tree. For details on the usage of ANTLR regarding TTCN-3 and its TRex integration see the master's thesis of Zeiss [Zei06].

TRex features not only plug-ins for refactoring and metrics but also for pattern-based smell detection in TTCN-3 test suites. The pattern-based smell detection plug-ins as well as the TTCN-3 code smell catalogue (see section 2.2.2) originate from the master's thesis of Bisanz [Bis06]. The pattern detection plug-ins make use of the *Static Analysis Framework* provided by the Eclipse *Test & Performance Tools Platform* (TPTP) Project [Ecl07c]. To make use of the static analysis framework an application needs to extend predefined abstract classes for analysis providers, categories, rules, and analysis results. Providers are the topmost of these classes. They contain all categories and rules and control the execution of analysis runs. Categories are used to structure analysis rules and can include arbitrary subcategories. Rules perform the actual analyses and can be parametrised using rule parameters. Result classes prepare the analysis results in a way the framework can display them using the predefined user interface (UI). The user interface of the static analysis framework is not only responsible for the graphical presentation of analysis results but also for the selection of categories and rules for analysis runs. For details on the usage and structure of the TPTP static analysis framework see [Bis06, GM05, GM06a, GM06b]. The framework is also used for the integration of XAF into TRex and acts as a link between both components. The latest available version 4.4 of TPTP which requires Eclipse version 3.3 is used for the implementation of XAF. This is no additional restriction, as the upcoming release 0.6 of TRex requires Eclipse 3.3 anyway.

[1] `http://www.trex.informatik.uni-goettingen.de`

The XQuery processor is a central part of the XQuery-based analysis framework as it is responsible for the execution of the queries and the detection of the patterns. The most important requirements regarding the processor are a complete and correct implementation of the XQuery 1.0 standard and a good performance. *Saxon* [Sax07] is an XSLT and XQuery processor developed by Michael Kay which fulfils these requirements as Saxon is the only processor that reached 100% pass rates[2] against the *XQuery Test Suite*[3] provided by the W3C. These results are due to the fact that the author of Saxon is the editor of the XSLT 2.0 standard [Kay07] and one of the editors of the XPath 2.0 standard [CBF+07]. Hence, Saxon is something like a reference implementation of an XSLT and XQuery processor.

Saxon is available in different versions: Saxon-B (for *b*asic) is the less powerful but freely available open-source version, Saxon-SA (for *s*chema-*a*ware) is a commercial product built on the same source code basis as Saxon-B. Both products are available for Java and Microsoft .NET platforms. Saxon-SA provides advanced features like schema-awareness (allows to import an XML Schema and to validate input and output trees) and advanced optimisers which recognise joins in XPath expressions and XQuery FLWOR expressions. Starting with version 8.9.0.4 Saxon-SA provides the ability to translate XQuery expressions "directly into Java source code, reducing execution time by anything from 25% to 80%."[4] The superior performance of Saxon-SA is also backed by third-party performance tests [Eic07].

XAF utilises the Java edition of Saxon-B (version 8.9.0.4) as Eclipse and TRex are also based on Java. To be able to use Saxon from inside of TRex is must be packaged as an Eclipse plug-in. Hence, the plug-in `de.ugoe.cs.swe.saxon` contains all Saxon Java archive (JAR) files and exports Saxon's packages which are located in the namespaces `net.sf.saxon.*` to allow other plug-ins to makes use of Saxon.

All underlying software technologies used for the implementation of XAF are open-source software: Eclipse, TPTP, and TRex are available under the terms of the Eclipse Public License 1.0 (EPL)[5] while Saxon-B is licensed under the terms of the Mozilla Public License 1.0 (MPL)[6]. Both licenses follow the Open Source Initiative (OSI)[7] definition of open source[8] and are open-source licenses approved by the OSI. All parts of XAF that relate to the TRex integration are also licensed using the EPL 1.0 while the XQuery core of XAF is licensed under the terms of the GNU Lesser General Public License 2.1 (LGPL)[9]. The reason for this decision is the stronger *copyleft*

[2] `http://www.w3.org/XML/Query/test-suite/XQTSReport.html`
[3] `http://www.w3.org/XML/Query/test-suite/`
[4] `http://www.saxonica.com`
[5] `http://www.opensource.org/licenses/eclipse-1.0.php`
[6] `http://www.opensource.org/licenses/mozilla1.0.php`
[7] `http://www.opensource.org`
[8] `http://www.opensource.org/docs/osd/`
[9] `http://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`

property of the LGPL compared to the Eclipse Public License. Hence, all contributions made to LGPL licensed source code must also be licensed using LGPL while EPL allows to license contributions using non-free licenses. The combination of the different licenses does not lead to problems regarding license compatibility as no source code was copied, shared, or linked together.

5.2 Alliances of Java and XQuery

This section describes ways to combine the usage of Java and XQuery. This is necessary as the patterns are described using XQuery. Indeed, the execution of the patterns is controlled by TRex which is Java-based. In the first part of this section the *XQuery API for Java* is introduced which provides a standardised way of executing XQuery expressions from Java. The second part covers the other way around and describes a possibility to access Java from XQuery.

The *XQuery API for Java* (XQJ) [XQJ07] is currently being standardised as Java Specification Request 225. The specification reached the status of a public review draft in May 2007 and the final standard is expected for the year 2008. XQJ is a "common API that allows an application to submit queries conforming to the W3C XQuery 1.0 specification and to process the results of such queries" [XQJ07]. Hence, XQJ is for XQuery what the *Java Database Connectivity* (JDBC) API is for relational databases. XQJ also provides a mapping of XQuery data types to Java data types and other advanced features like transactional operations. The current Java edition of Saxon implements the latest XQJ specification 0.9 and is used by XAF for the execution of the XQuery expressions.

The usage of XQJ is straightforward as listing 5.1 shows. The method `executeXQuery` (line 1) takes an XQuery expression as string parameter and returns an `XQResultSequence` object which represents a sequence of items. As mentioned in section 2.3.2 such a sequence consists of atomic values and/or nodes. It is up to the caller of the method `executeXQuery` to loop through the sequence and extract the desired information. In lines 2 and 3 an `XQExpression` object is instantiated and its method `executeQuery` is called in line 4. The query's result is stored in the `XQResultSequence` object which is returned in line 6 if it contains any items (test in line 5). Otherwise an `EmptySequenceException` exception is risen in line 8 to indicate that the sequence is empty.

Some XQuery tools offer Java bindings which permit creating Java objects and calling Java methods from within XQuery. These non-standardised Java bindings were introduced by Saxon[10] and adopted by other XQuery tools like the XML database *eXist* [eXi07]. Listing 5.2 shows an XQuery expression inspired by the eXist documentation[11] making use of the Java

[10] `http://www.saxonica.com/documentation/extensibility/functions.html`
[11] `http://exist.sourceforge.net/xquery.html#N10599`

```
1 public XQResultSequence executeXQuery(String query) throws Exception {
2   XQExpression expression =
3     new SaxonXQDataSource().getConnection().createExpression();
4   XQResultSequence result = expression.executeQuery(query);
5   if (result.next())
6     return result;
7   else
8     throw new EmptySequenceException();
9 }
```

Listing 5.1: Executing an XQuery expression from Java using XQJ

bindings. The query uses the `java.io.File` class to return a sequence of the XML elements `directory` and `file`. These feature according `name` attributes reflecting the contents of the current working directory.

```
1 declare namespace file="java:java.io.File";
2
3 for $current-file in file:listFiles(file:new("."))
4 let $name := file:getName($current-file)
5 return
6   if (file:isDirectory($current-file)) then
7     <directory name="{ $name }" />
8   else
9     <file name="{ $name }" />
```

Listing 5.2: Using Java objects and methods from within XQuery

In line 1 of listing 5.2 the XQuery namespace `file` is bound to the Java class `java.io.File`. In line 3 a file object is instantiated (`file:new(".")`) and used as argument of the method `listFiles`. This style differs from the classic Java way of calling member methods. Using the XQuery Java bindings the current object instance is always passed as first parameter to the called member method. This can also be seen in line 4 and 6 for the calls of the methods `getName` and `isDirectory`. To clarify the difference the equivalent Java source code snippet is shown in listing 5.3.

In line 3 of listing 5.2 each Java file object returned by the method `listFiles` is bound to the XQuery variable `$current-file`. This variable is used to get the filename (line 4) and to distinguish between directories and files (line 6). Finally, a `directory` respectively a `file` element is returned including the according `name` attribute (lines 7 and 9).

The possibility to access arbitrary Java classes from within XQuery is a powerful extension mechanism. It is also used by the implementation of XAF to make use of existing methods already implemented in Java by TRex.

5.3 Conversion of TTCN-3 Source Code to XML

As the framework is integrated into TRex—and this "dinosaur" is already capable of parsing TTCN-3 source code using ANTLR—it is self-evident

```
1 for (File currentFile : new File(".").listFiles()) {
2   String name = currentFile.getName();
3   if (currentFile.isDirectory())
4     System.out.println("directory: " + name);
5   else
6     System.out.println("file: " + name);
7 }
```

Listing 5.3: Java equivalent of listing 5.2

to build the XML representation of TTCN-3 on these foundations. Each TTCN-3 source code file is represented as an abstract syntax tree by TRex[12]. This tree can be accessed through the class `LocationAST` in the package `de.ugoe.cs.swe.trex.core.analyzer.rfparser`[13]. The class `LocationAST` extends the class `CommonAST` provided by ANTLR and adds "corresponding start and end offsets as well as the line in the parsed source file, the associated token in the token stream and the associated scope" [Zei06].

The information stored in `LocationAST` objects fulfil the requirements to serve as representation format for the software analysis framework. The location information of the TTCN-3 source code entities can be used to map matches found by the analysis framework to the original source code. Hence, a nearly complete one-to-one mapping of `LocationAST` objects to XML documents is required. TTCN-3 definitions, operators, statements, and blocks (respectively their AST representations) are mapped to XML elements because elements can contain other nested elements. TTCN-3 identifier and literals are mapped to XML text nodes and location information (like the filename, line numbers, and offset information) are mapped to XML attributes.

The implementation of the XML export is done by extending the class `LocationAST` by a new public method `xmlSerialize(Writer stream, String file.Name, boolean indentXML)` and some private helper methods. The AST is traversed recursively and all elements of the tree are mapped to according XML elements and attributes as described in the previous paragraph. The resulting XML document is written to the `Writer` object passed to the `xmlSerialize` method. The parameter `String fileName` must be passed as this information is not already part of the `LocationAST` class. The filename is added an an XML attribute to the root element `TTCN3File` to be able to relocate the according TTCN-3 source code file. This is required when the XML result of the analysis is mapping back to the TTCN-3 source code. The XML document can be created with indentation for better readability. This is controlled by the parameter `boolean indentXML`. Additionally, the attribute `endLine` is added to the class `LocationAST` which denotes the line in which the associated token ends.

[12] "In a strict sense the TRex syntax tree is not abstract as it contains elements without semantic relevance as well. Hence, it is similar to a parse tree/concrete syntax tree." [Bis06]

[13] For an overview of the TRex packages and plug-in structure see: `http://www.trex.informatik.uni-goettingen.de/trac/wiki/PluginStructure`

Listing 5.4 shows a part of the XML document of the serialised `LocationAST` object representing the TTCN-3 source code of listings 2.2 and 2.3 from section 2.2.1. The XML snippet in listing 5.4 represents this source code: `template charstring HTTPResponse := pattern "HTTP/1.1 \d\d\d *";` (it is taken from line 9 in listing 2.3 from section 2.2.1). Definitions, blocks, and types are represented as XML elements. Identifier and literals as XML text nodes, and location information as XML attributes. For reasons of clarity only the attributes of the `ModuleDefinition` element in line 1 are included. Normally each element features the attributes `line`, `offset`, and `offset-end`. The attribute `line-end` is only added if it is different from the attribute `line` to avoid information duplication. As the template in this example is defined in one line, the attribute `line-end` is not included.

```
1  <ModuleDefinition line="9" offset="234" offset-end="297">
2    <TemplateDef>
3      <BaseTemplate>
4        <Type><PredefinedType>charstring</PredefinedType></Type>
5        <Identifier>HTTPResponse</Identifier>
6      </BaseTemplate>
7      <TemplateBody>
8        <SingleValueOrAttrib>
9          <MatchingSymbol>
10           <CharStringMatch>HTTP/1.1 \d\d\d *</CharStringMatch>
11         </MatchingSymbol>
12       </SingleValueOrAttrib>
13     </TemplateBody>
14   </TemplateDef>
15 </ModuleDefinition>SemiColon
```

Listing 5.4: Partial XML representation of a `LocationAST` object

The method `xmlSerialize` of the `LocationAST` class is used by the main XAF plug-in `de.ugoe.cs.swe.trex.xaf` to create one global XML document representing all TTCN-3 files that should be analysed. This is performed by the method `createXML` of the class `Analysis`. The `createXML` method iterates over all files to be analysed and instantiates a `TTCN3Analyzer` object for each file. The analyser object permits access to the `LocationAST` object and on which finally the `xmlSerialize` method is called. The resulting global XML document is written to a file named `ttcn3.xml` which is used as input for the XQuery facade layer. Listing 5.5 shows the global structure of such an XML document. One XML file representing multiple TTCN-3 files allows analyses crossing the file borders but can also lead to large file sizes. The file sizes for the XML representation of the TRex AST can be estimated by 1.6 megabyte per thousand lines of TTCN-3 source code.

```
1 <TTCN3Files>
2   <TTCN3File filename="/folder/moduleFoo.ttcn3">
3     <!-- [...] -->
4   </TTCN3File>
5   <TTCN3File filename="/folder/moduleBar.ttcn3">
6     <!-- [...] -->
7   </TTCN3File>
8   <!-- [...] -->
9 </TTCN3Files>
```

Listing 5.5: Structure of the global XML document

5.4 XQuery Facade Layer for TTCN-3

This section describes the implementation of the XQuery-based facade layer according to its design presented in section 4.4. The purpose of the facade layer is to provide an abstraction from the underlying representation format respectively analysis target and a decoupling of the analysis plug-ins from the underlying layers.

The XQuery module `facade.ttcn3.trex.xquery` and all other XQuery and XML files belonging to XAF are contained in the Eclipse plug-in `de.ugoe.cs.swe.xaf`. The naming of the plug-in already states that XAF is neither part of TRex nor depends on it. Indeed, the facade module is customised to provide access to TTCN-3 source code provided by TRex. This is reflected in the naming of the XQuery module file (`facade.ttcn3.trex.xquery`[14]) and the module namespace (`de.ugoe.cs.swe.xaf.facade.ttcn3.trex`, see listing 5.6, line 1).

Listing 5.6 shows the header of the facade layer module for the access to the XML-encoded TTCN-3 source code provided by TRex. Line 1 contains the mentioned namespace declaration, followed by the declaration of the external variable `$xfacade:path` in line 3. The value of this variable needs to be provided by the XQuery processor and is the absolute path to the input XML file `ttcn3.xml`. The XML structure contained in this file is loaded into the global module variable `$xfacade:root` in line 4 using the `doc` function. This variable is used by the facade functions to provide a channelled access to the underlying representation format.

Instead of passing the path to the XML file as an external variable to the XQuery module, the XML document itself could have been passed directly as an external variable. The presented solution is chosen because performance tests revealed that passing the document directly is considerably slower than passing the path and reading the file using the `doc` function.

[14] XAF facade namespaces and module files should be named according to the pattern `facade.analysis-target.representation-format[.xquery]`. For instance, `facade.java.srcml.xquery` for a facade module file providing access to Java source code using the representation format srcML.

```
1  module namespace xfacade = "de.ugoe.cs.swe.xaf.facade.ttcn3.trex";
2
3  declare variable $xfacade:path external;
4  declare variable $xfacade:root := doc($xfacade:path);
```

Listing 5.6: Header of the XQuery facade layer module for TTCN-3

Listing 5.7 is the continuation of the preceding listing 5.6 and contains two exemplary facade functions (lines 1–7) and the helper function `get-node` (lines 9–14). The two facade functions `get-files` and `get-functions` provide access to the XML element `TTCN3File` which represents a TTCN-3 source code file respectively to the element `FunctionDef` which represents a TTCN-3 function definition. Both functions feature a parameter `$n` of the type `node()*` (zero or more items of the type node) and do not directly access the global module variable `$xfacade:root` but instead pass the parameter `$n` to the helper function `get-node` (lines 2 and 6) to access the underlying XML document.

This strategy allows to control the scope of facade functions. If for instance `get-functions` is called and the parameter `$n` is the empty sequence (`get-functions(())`), all function definitions are returned. On the other hand, if parameter `$n` is for instance bound to a `TTCN3File` element, only the function definitions of this TTCN-3 file are returned. This behaviour is implemented in the helper method `get-node` which returns either the variable `$xfacade:root` (line 11) if `$n` is empty or otherwise `$n` itself (line 13).

```
1  declare function xfacade:get-files($n as node()*) {
2    xfacade:get-node($n)//TTCN3File
3  };
4
5  declare function xfacade:get-functions($n as node()*) {
6    xfacade:get-node($n)//FunctionDef
7  };
8
9  declare function xfacade:get-node($n as node()*) {
10   if (empty($n)) then
11     $xfacade:root
12   else
13     $n
14 };
```

Listing 5.7: Facade functions of the TTCN-3 facade module

The complete facade module currently consists of about 70 facade functions and their source code is included as appendix A.1. For a better overview, the function are structured in groups. The first group contains facade functions like `get-files`, `get-functions`, and `get-identifiers` which are common for most programming languages. The second group consists of functions specific to TTCN-3 such as `get-testcases` and `get-alt-constructs`. The facade functions are not complete in the way that all kinds of information of the underlying XML tree could be accessed through them. The currently

implemented functions providing access to the most important parts of the TTCN-3 representation. But as seen above, the functions of the facade layer are quite simple and therefore missing ones could simply be added as the framework grows.

The third group of facade functions is a special case and contains functions calling AST-related Java methods of TRex. This is realised using the Java bindings provided by Saxon which are described in section 5.2. The direct access to the AST using the TRex APIs is preferred over the access to the XML structure, as TRex already implements AST-related analysis methods like a scope-aware reference finder. Repeating these implementations using XQuery would have required additional effort without significant benefits. The use of the direct access to the AST is currently only used by two smell detection functions to discover single references. All other analysis functions are written using pure XQuery expressions.

The following details regarding the usage of Java methods of TRex are strictly related to the facade layer. Hence, they are presented here and not in section 5.6 which discusses the integration of XAF into TRex. The Java methods called by the facade module are located in the class `TRexAST` of the main XAF plug-in `de.ugoe.cs.swe.trex.xaf`. This main XAF plug-in requires the Saxon plug-in `de.ugoe.cs.swe.saxon` to make use of Saxon to execute XQuery expressions. Java calls from XQuery modules are executed by Saxon and therefore Saxon needs to find the according classes. Normally the Java classpath is used for this purpose, but the Eclipse plug-in concept differs from the classpath approach. To permit Saxon to find the class `TRexAST` in the plug-in `de.ugoe.cs.swe.trex.xaf` it is *not* possible to let the Saxon plug-in depend on the XAF plug-in because this would lead to a plug-in dependency cycle. The solution is the *Buddy Class Loading*[15] concept of Eclipse. For instance, this allows plug-in `A` to register at plug-in `B`. If plug-in `A` cannot find a class in its own scope, it looks for the class at all registered plug-ins, in this example plug-in `B`. This behaviour is controlled by entries in the `MANIFEST.MF` files. Hence, the file `MANIFEST.MF` of the Saxon plug-in is modified to permit other plug-ins to register themselves at the Saxon plug-in by adding the line `Eclipse-BuddyPolicy: registered`. The manifest file of the XAF plug-in is also modified to register itself at the Saxon plug-in by adding the line `Eclipse-RegisterBuddy: de.ugoe.cs.swe.saxon`. The Saxon plug-in is now able to find the `TRexAST` class in the XAF plug-in.

Figure 5.1 summarises the tasks of the XQuery facade layer module and shows its central position in the framework. The facade layer is used by the smell detection module to access the XML representation of the TRex AST and additionally to directly access the TRex AST by utilising the TRex API.

[15] `http://wiki.eclipse.org/index.php/Context_Class_Loader_Enhancements#Buddy_Class_Loading`, `http://www.eclipsezone.com/articles/eclipse-vms/`

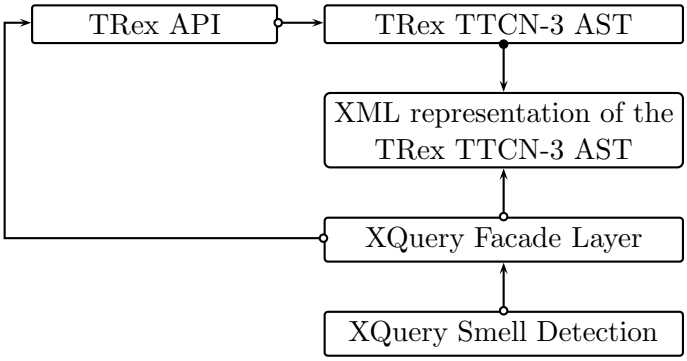

Figure 5.1: Central position of the XQuery facade layer module

5.5 XQuery Pattern Description of Bad Smells

The description of patterns using XQuery is the actual core of XAF. As the framework is customised for the detection of bad smells, this section shows how XQuery is used to describe and detect bad smells in TTCN-3 source code. The section is structured as follows: the main focus is set to the detailed presentation of a few sample XQuery functions for the detection of bad smells. This is succeeded by XQuery implementation-related features, a list of all currently available XQuery smell detection functions, and the presentation of the XML file `smells.ttcn3.xml` describing the structure of the bad smell detection module.

Listing 5.8 shows the XQuery function to detect instances of the bad smell *Long Parameter List* (see section 2.2.2 for details on this smell). The function `long-parameter-list` is parametrised with the `integer` parameter `$floor`. The value of `$floor` is the minimal number of parameters required to mark a parameter list as too long. TTCN-3 allows not only functions to have a parameter list but also templates, testcases, typedefs, external functions, signatures, and altsteps. Therefore the parameter lists of those constructs are also included in the analysis. For reasons of clarity the presented XQuery function only takes functions and templates into account. Hence, in line 4 all function and template definitions are joined into one sequence. Using a `for` loop, each item of the sequence is bound to the variable `$parametrizable-construct` one after another in line 3. Its parameters are stored in the variable `$parameters` in line 5, and the number of parameters is compared to the number of required parameters to match. This is done using the `where` condition in line 6 which checks if the number of parameters (`count($parameters)`) is greater or equal to the value of the variable `$floor`. If this condition is true, an instance of the bad smell is found and returned in line 7.

Listing 5.9 shows the second sample XQuery function which is called `missing-log`. It describes the bad smell *Missing Log* which is a TTCN-3-specific smell that occurs if "setverdict is used with verdict inconc or fail,

```
1  declare function xsmell:long-parameter-list($floor as xs:integer)
2  {
3    for $parametrizable-construct in
4      (xfacade:get-functions(()), xfacade:get-templates(()))
5    let $parameters := xfacade:get-parameters($parametrizable-construct)
6    where count($parameters) >= $floor
7    return $parametrizable-construct
8  };
```

Listing 5.8: XQuery function to detect the bad smell *Long Parameter List*

but without calling log" [Bis06]. The function `missing-log` iterates over all `setverdict` statements (line 3), stores the according type of the verdict in the variable `$verdict-type` (line 4), and the parent block of the `setverdict` statement in the variable `$parent-block` (line 5). The `where` condition in lines 6–8 ensures that in the block of the `setverdict` statement no log operation is contained and that the verdict type is fail or inconc (lines 7–8). If both conditions are true (no log operation and verdict type fail or inconc) an instance of the bad smell *Missing Log* is found and the verdict is returned in line 9.

```
1  declare function xsmell:missing-log()
2  {
3    for $verdict in xfacade:get-setverdicts(())
4    let $verdict-type := xfacade:get-verdict-type($verdict)
5    let $parent-block := xfacade:get-parent-block($verdict)
6    where not(xfacade:get-log-ops($parent-block))
7      and (xfacade:verdict-is-fail($verdict-type)
8       or xfacade:verdict-is-inconc($verdict-type))
9    return $verdict
10 };
```

Listing 5.9: XQuery function to detect the bad smell *Missing Log*

Listing 5.10 shows slightly simplified version of the smell detection function `unreachable-default`. It describes the TTCN-3-related bad smell *Unreachable Default* which occurs if an alt statement contains an else branch while a default is active. This functions makes use of the XQuery data model which keeps all nodes in the document order. This allows the function to detect if a default is activated when an else branch is executed. The function loops over all block which contain default declarations in line 3. In lines 6–8 the else branch and the activation/deactivation statements of the default are bound to corresponding variables. If the activate statement is followed by the else branch which is followed by the deactivation of the default, an instance of this bad smell is found. The according check is performed in line 10.

XAF also implements an infrastructure for the clone detection of duplicated code. The actual comparison functions are located in a library module and their application is presented to detect instances of the bad smell *Duplicate Alt Branches*. Listing 5.11 shows a slightly simplified version of the

```
1  declare function xsmell:unreachable-default()
2  {
3    for $block in xfacade:get-blocks(())[xfacade:get-defaults(.)]
4    return
5      let $alt  := xfacade:get-alt-constructs($block)
6      let $else := (xfacade:get-else-statements($alt))[1]
7      for $activate   in xfacade:get-activate-ops($block)
8      for $deactivate in xfacade:get-deactivate-ops($block)
9      (: else statement must be surrounded by activate and deactivate :)
10     where $activate << $else and $else << $deactivate
11     return $activate
12 };
```

Listing 5.10: XQuery function to detect the bad smell *Unreachable Default*

XQuery function describing this bad smell. The workflow for each duplicate search is related: first the scope of the comparison is estimated. In this case duplicate branches contained in altsteps and alt constructs must be compared. Hence, the for loop in line 3 interates over all altsteps and alt constructs and selects the alt branches of the current scope. The actual comparison is performed by the function `find-duplicates` in line 5 which returns a sequence of duplicates. As this sequence may contain multiple kinds of duplicates[16] the duplicates belonging together are grouped by the function `group-duplicates` and returned in line 6.

```
1  declare function xsmell:duplicate-alt-branches()
2  {
3    for $scope in (xfacade:get-altsteps(()), xfacade:get-alt-constructs(()))
4    let $to-compare := xfacade:get-alt-branches($scope)
5    let $duplicates := xlib:find-duplicates($to-compare)
6    return xlib:group-duplicates($duplicates)
7  };
```

Listing 5.11: XQuery function to detect the bad smell *Duplicate Alt Branches*

The duplicate search can simply be adapted to find other types of duplicate code by changing the scope and the elements to compare in the lines 3 and 4. It is also possible to omit the scope completely and to compare elements in the complete XML document what allows to find duplicates that are located in different TTCN-3 source code files. But these kinds of duplicate searches are quite expensive regarding running time as each element must be compared to each other.

The precise number of comparisons that are performed by the function `find-duplicates` is $\sum_{i=1}^{n} i + \ldots + (n-i) \Rightarrow \frac{n*(n-1)}{2}$ whereat n is the number of elements to compare. Using the O-notation the maximal effort can be estimated by an upper boundary of $O(\frac{n^2}{2}) \Rightarrow O(n^2)$ [Knu97].

[16] Using a non-XML syntax to describe this: the sequence might look like (dup1, dup1, dup2, dup2, dup2). To ease the subsequent processing this sequence is transformed to look like ((dup1, dup1), (dup2, dup2, dup2)).

The implemented XQuery functions for the detection of duplicated code currently support two comparison modes. The default mode returns only strictly matching duplicates while the second mode returns those elements as duplicates whose XML subtrees contain the same elements. Text nodes and attributes are not taken into account for the comparison using this mode. As only identifiers and literals are stored as text nodes, this strategy allows to find duplicates with differing arguments. For example, the two alt branches shown in listing 5.12 are considered as duplicates—even though the arguments of the functions `receive` and `log` are different. The desired comparison behaviour is controlled via a parameter of the clone detection functions which is left out in listing 5.11.

```
1 [] InputPort.receive(ErrorFoo) {
2   setverdict(fail);
3   log("received error foo");
4 }
5 [] InputPort.receive(ErrorBar) {
6   setverdict(fail);
7   log("received error bar");
8 }
```

Listing 5.12: TTCN-3 alt branches considered as duplicates

The presented sample XQuery smell detections functions clarify the advantages of using XQuery for the pattern description. Easy readability of the patterns is achieved by using descriptive variable names and avoiding deep nesting of function calls. The usage of library functions and facade functions instead of XML element and attribute names also improves the comprehensibility of the patterns.

The described XQuery functions return their matches as XML subtrees of the global XML document `ttcn3.xml`. As discussed in section 4.4 only the information required for locating the matches in the analysis target need to be returned. Thus, for each analysis function a corresponding function is declared which returns the match formated according to the XML schema `matches.xsd`. Listing 5.13 shows a pair of two functions called `missing-log`. The first one is the analysis function, the second function formats the match as desired. In lines 1–4 the function of listing 5.9 is repeated but the function body is left out as it is not necessary for this context. In lines 6–9 the second function named `missing-log` is located. The only purpose of this function is to take the additional parameter `$name` and call the function `format-match` with the result of the first `missing-log` function and the parameter `$name`. The `format-match` function extracts all required information from the complete match(es) and returns a sequence of `match` elements conforming to the `matches.xsd` schema. The value of the parameter `$name` should be set by the caller to the function name itself (for instance `xsmell:missing-log`) to include this value in the `match` element as `found-by` attribute. This allows the subse-

quent application to distinguish which match element was created by which function. Unfortunately, neither XQuery itself nor a Saxon extension supports reflection mechanisms which would have allowed to get the current function name from inside of XQuery.

```
1 declare function xsmell:missing-log()
2 {
3   [...]
4 };
5
6 declare function xsmell:missing-log($name as xs:string)
7 {
8   xfacade:format-match(xsmell:missing-log(), $name)
9 };
```

Listing 5.13: Two XQuery functions per analysis: One returning complete matches, one returning `match` elements conforming to the `matches.xsd` schema

Currently 20 XQuery analysis functions for the detection of bad smells are implemented. All of them are described in the TTCN-3 code smell catalogue. The implemented analysis function cover all categories of the code smell catalogue but the category data flow anomalies. The following analysis functions are not TTCN-3-specific as the bad smells can also occur in other programming languages (ordered by name): `duplicate-code-in-conditionals`, `goto`, `long-parameter-list`, `long-statement-block`, `magic-number`, and `nested-conditional`. The following smell detection functions are specific to TTCN-3 (ordered by name): `deactivation-other-level`, `duplicate-alt-branches`, `fully-parameterized-template`, `idle-ptc`, `missing-activation`, `missing-deactivation`, `missing-log`, `missing-verdict`, `short-template` (based on the number of characters), `short-template2` (based on the number of template fields), `singular-component-element-reference`, `singular-template-reference`, `stop-in-function`, and `unreachable-default`.

For details on the bad smells and their implementation, see the comments in the XQuery module `smells.ttcn3.xquery`, the according XML file `smells.ttcn3.xml`, and the TTCN-3 code smell catalogue. Additionally, the source codes of all XQuery analysis functions are included as appendix A.2 (generic smell detection functions) and appendix A.3 (smell detection functions for TTCN-3).

As discussed in section 4.4 XQuery modules are unstructured lists of XQuery functions which is not suitable for a framework as the embedding application cannot easily make use of the modules. Hence, the structure of modules is described in according XML documents conforming to the XML schema definition `hierarchy.xsd`. Listing 5.14 contains a snippet of the file `smells.ttcn3.xml` which describes the hierarchical structure of the smell detection module `smells.ttcn3.xquery`. As the allowed XML elements and attributes of hierarchy XML documents are presented in detail in section 4.4, listing 5.14 is only briefly described. Lines 1–6 contain the root element

hierarchy and its attributes to find the according XQuery module file and to access it with correct namespace information. The remainder of the document (lines 7–15) consists of a category element containing two function elements. The second function element features a parameter element and according attributes.

```
1  <hierarchy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2    xsi:noNamespaceSchemaLocation="hierarchy.xsd"
3    name="Bad Smell Detection for TTCN-3 Source Code"
4    filename="smells.ttcn3.xquery"
5    namespace-prefix="xsmell"
6    namespace="de.ugoe.cs.swe.xaf.smells.ttcn3">
7    <category name="Test Behavior">
8      <function name="Missing Log" xquery-name="missing-log"/>
9      <function name="Stop In Function" xquery-name="stop-in-function">
10       <parameter
11         name="Exclude functions running on a component"
12         xquery-name="skip-runs-on" type="boolean"/>
13     </function>
14   </category>
15 </hierarchy>
```

Listing 5.14: File smells.ttcn3.xml describing the structure of the smell detection module smells.ttcn3.xquery

5.6 Integration into TRex

Before going into the details of the integration, an overview of the complete framework architecture including its embedding into TRex is given. Figure 5.2 shows the global structure of the XQuery-based analysis framework customised for the detection of bad smells in TTCN-3 test suites and integrated into TRex. While figure 5.1 is focused on the position of the facade layer, figure 5.2 includes all relevant parts of the framework. Only details regarding the integration into the TPTP static analysis framework are left out. The three boxes in figure 5.2 at the right bottom of the figure illustrate the core framework: the XQuery facade layer, XQuery smell detection module, and XQuery library modules. The latter contain common functions which are not related to software analysis and therefore no further described. The round box at the left bottom is the starting point for the usage of XAF and the interface where the framework is coupled with TPTP. The three boxes on the right top show, starting with the topmost box, the analysis target (TTCN-3 source code) and its representation formats (AST of the TTCN-3 source code and the XML representation of the AST). The box on the left side shows the direct access from the facade layer through TRex APIs to the AST passing by the XML representation format which is described in detail at the end of section 5.4.

The integration of XAF into TRex is described in this section. First a narrow description of preparations regarding updating existing TRex plug-

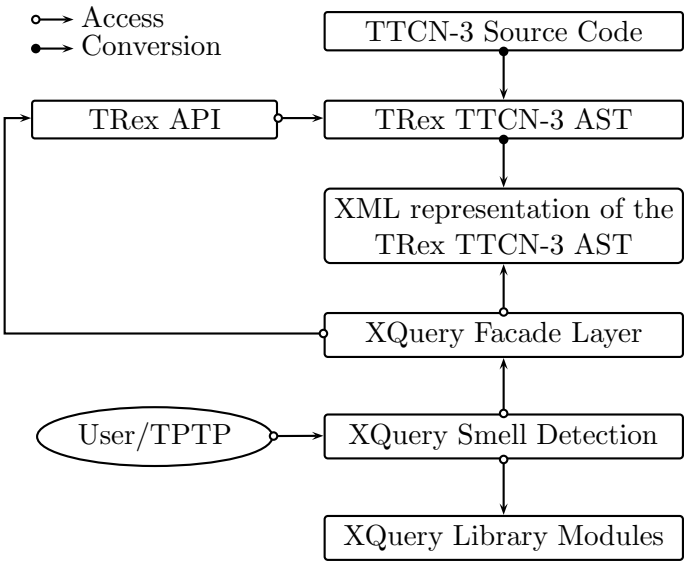

Figure 5.2: Overview of the framework customised for the detection of bad smells in TTCN-3 test suites

ins is given. Afterwards the actual integration of XAF into TRex using the static analysis framework of TPTP is discussed.

The Eclipse plug-ins `de.ugoe.cs.swe.trex.patterndetection.*` created by Bisanz [Bis06] in 2006 serve as a starting point for the integration of XAF into TRex as they are already built upon the TPTP static analysis framework. These pattern detection plug-ins shall be preserved and it should be possible to use them in conjunction with the newly created XAF plug-ins. Additionally, as much as possible TPTP-related functionality should be shared between the pattern detection and XAF plug-ins to avoid code duplication.

The fist step was an update of the existing pattern detection plug-ins to the current TPTP 4.4.0 branch which required mostly adoptions of TPTP interface changes. In the second step, functionality that should be shared between the XAF and pattern detection plug-ins was identified. As a result of this, the plug-in `de.ugoe.cs.swe.trex.tptp` was created which contains a common base class for analysis results and declarations of shared constants. Additionally, the plug-in `de.ugoe.cs.swe.trex.patterndetection.ui` no longer depends on the core plug-in of the pattern detection and can now serve as the user interface for the pattern detection and XAF plug-ins. These changes allow to combine both analysis plug-ins and to perform analyses including rules from both plug-ins at the same time.

The XAF integration into TRex is located in the plug-in `de.ugoe.cs.swe.trex.xaf` which contains all classes required for the usage of the TPTP static analysis framework. The most important classes are briefly described in

Class Name	*Functionality*
`AnalysisProvider`	Creates TPTP categories and rules and controls the execution of analysis runs.
`AnalysisRule`	Represents XQuery functions as TPTP rules.
`Analysis`	Represents analysis runs.
`XafResults`	Represents XML results of XAF as Java object.

Table 5.1: Important classes of the `de.ugoe.cs.swe.trex.xaf` plug-in

table 5.1. The entry point to the static analysis framework is the XAF class `AnalysisProvider` which extends the class `AbstractAnalysisProvider` of TPTP. This relationship is reflected by the `plugin.xml` file shown in listing 5.15. The TPTP extension point in line 2 is extended by the XAF analysis provider in lines 3–7. A link to the `AnalysisProvider` class is provided in line 4, a unique identifier and a human readable label for the provider are stored in lines 5 and 6. In line 7 the link to user interface implementation of a viewer class is provided. As described above, XAF and the pattern detection plug-ins by Bisanz make use of the shared viewer class `de.ugoe.cs.swe.trex.patterndetection.ui.PatternDetectionViewer`.

```
1 <plugin>
2   <extension point="org.eclipse.tptp.platform.analysis.core.
        analysisProvider">
3     <analysisProvider
4       class="de.ugoe.cs.swe.trex.xaf.AnalysisProvider"
5       id="de.ugoe.cs.swe.trex.xaf.provider"
6       label="XQuery-based Analysis Framework for TTCN-3"
7       viewer="de.ugoe.cs.swe.trex.patterndetection.ui.
            PatternDetectionViewer"/>
8   </extension>
9 </plugin>
```

Listing 5.15: File `plugin.xml` of the XAF plug-in for TRex

The `AnalysisProvider` class fulfils the following tasks: it is responsible for the creation of the TPTP categories and rules based on the contents of the file `smells.ttcn3.xml`. Additionally, the provider controls the workflow of analysis runs. The implementation of both tasks is described in the following.

The XQuery-based analysis framework describes the structure of XQuery analysis modules in XML files conforming to the `hierarchy.xsd` schema. This allows the structure of modules to be described independent of any third-party standards and dependencies. Hence, the class `AnalysisProvider` needs to parse the file `smells.ttcn3.xml` to create a TPTP category for each category defined in the XML file and to create a TPTP rule for each function defined in the XML file. This is done on-the-fly the first time the TPTP analysis dialogue is opened inside a TRex instance.

The TPTP static analysis framework is not designed for adding categories and rules at runtime. Normally they are statically declared as TPTP extensions in the file `plugin.xml`. To work around this limitation, the method `getOwnedElements` of the abstract provider class is overwritten and used to insert the categories and rules on-the-fly. The first time the `getOwnedElements` method is called, the file `smells.ttcn3.xml` is parsed and all TPTP categories and rules are created according to contents of the XML file.

Without going into details, the process of creating TPTP categories and rules on-the-fly is described in this paragraph. First the file `smells.ttcn3.xml` is parsed, made available as DOM tree, and validated against its XML schema definition. Afterwards the DOM tree is traversed recursively and TPTP categories and rules are created. For each `category` element of the XML file an according `DefaultAnalysisCategory` object is instantiated. Each XML `function` element—which describes an XQuery analysis function—is mapped to an `AnalysisRule` object which is provided by XAF. The `AnalysisRule` object stores the function name of its corresponding XQuery to enable a reverse mapping from the analysis rule to the corresponding XQuery function. To complete the mapping of the XML file to the static analysis framework of TPTP, `parameter` elements of the XML file are mapped to TPTP `AnalysisParameter` objects.

The control of the workflow of an analysis run—the execution of multiple analysis rules—is also located in the `AnalysisProvider` class and consists of the following steps.

Step 1: The scope of the analysis run is determined. All TTCN-3 source code files respectively projects or a subsets of these can be selected for an analysis run. The scope is set by the user through an interface provided by TPTP.

Step 2: The XML representation is created which reflects the abstract syntax trees of all files of the analysis scope.

Step 3: The hierarchy of the TPTP categories and rules is traversed to collect the information which rules are selected by the user and need therefore to be included in the analysis run. Based on these information an according XQuery expression is built.

Step 4: The XQuery expression created in the preceding step is executed, the resulting XML is parsed and stored in an `XafResults` object.

Step 5: Finally, the matches contained in the `XafResults` object are transformed into TPTP results to allow to display them using the TPTP user interface.

The starting point of each analysis run is the method `analyze` located in the class `AnalysisProvider` which is called by the TPTP static analysis

framework. The implementation of the five steps is briefly described in the following.

Step 1 is completely done by the TPTP static analysis framework: the method `getFilteredResources` returns a list of all required TTCN-3 `IFile` objects. This list reflects the scope of the analysis run. Step 2, the generation of the XML representation, is realised by the `createXML` method of the XAF `Analysis` class. It covers all required subordinate steps like parsing the TTCN-3 source code files and calling the `xmlSerialize` method of each `LocationAST` object. In the end an XML file named `ttcn3.xml` is created which is the input for the software analysis.

Step 3—building the XQuery expression based on the selected TPTP rules—is quite complex. As motivation an example of the desired result is given in listing 5.16. It shows an XQuery expression calling two analysis functions from the module `smells.ttcn3.xquery`. The execution of this expression by the XQuery processor is the actual XQuery-based analysis. The first four lines of the expression import the required smell detection and library modules. This is followed by the calls of the analysis functions `missing-log` and `long-parameter-list` in lines 7 and 8. Both function calls are wrapped into a call to the XQuery helper function `finalize-query` (line 6) which ensures that the returned results conform to the XML schema `matches.xsd`.

```
1 import module namespace xsmell =
2   'de.ugoe.cs.swe.xaf.smells.ttcn3' at 'smells.ttcn3.xquery';
3 import module namespace xlib =
4   'de.ugoe.cs.swe.xaf.library' at 'library.xquery';
5
6 xlib:finalize-query((
7   xsmell:missing-log('xsmell:missing-log'),
8   xsmell:long-parameter-list(4, 'xsmell:long-parameter-list')
9 ))
```

Listing 5.16: XQuery expression calling two analysis functions

To build such an XQuery expression it must first be determined which TPTP rules are selected by the user and should be included in the expression. This is done by the helper method `traverseCategories` which recursively traverses the hierarchy of all TPTP categories/rules and calls the method `generateXQueryFunctionCall` for each object `AnalysisRule` selected by the user. This generates a list of function calls—like the list of two calls in lines 7 and 8 in listing 5.16—which is stored in the class `Analysis`.

The mapping of TPTP rule parameter values to according XQuery parameter values is also required. The first parameter of the function call `long-parameter-list` in line 8 has the value `4`. This value originates from the TPTP analysis dialogue in which the user entered the value. Hence, these parameter values need to be mapped from Java to XQuery values. For instance, a checkbox of the TPTP dialogue is represented by the Java type `boolean`. If the value of such a Java `boolean` object is `true` it is mapped to the

XQuery type `xs:boolean` and value `true()`. Parameter mapping is currently supported from the Java types `boolean`, `String`, and `Integer` to the according XQuery types.

Step 4 of each analysis run is the actual execution of the XQuery expression created in the preceding step. The analysis provider thus calls the method `executeXQuery` of the `Analysis` class. This method adds all required namespace import statements as shown in lines 1–4 of listing 5.16 and also adds the call the XQuery function `finalize-query` (line 6). Additionally, it sets the value of the external XQuery variable `$xfacade:path` (listing 5.6, line 3) of the facade layer module. The content of this variable is the absolute path to the XML input file `ttcn3.xml` generated in step 2. This allows the facade layer module to access and read the XML file and to provide access to this representation format through its functions. Finally, the XQuery expression is executed using the XQuery API for Java (XQJ) provided by Saxon (see sections 5.1 and 5.2 for details on both). The resulting XML which conforms to the `matches.xsd` schema is parsed and transformed to a Java object of the class `XafResults` which provides simplified access to all matches and their attributes.

In the final step 5, the matches contained in the `XafResults` object are mapped to TPTP-based `AnalysisResult` objects. This is required to make use of the user interface provided by TPTP to present the results to the user. This again requires recursively traversing the hierarchy of the TPTP categories/rules using the helper method `traverseCategories` and calling the method `generateResult` for each rule. The reason for these traversals lays in the way XAF makes use of the static analysis framework. Normally each rule object features an `analyze` method which is called once for each analysis run, performs the analysis, and immediately returns its results. For example, this workflow is used by the pattern detection plug-ins by Bisanz [Bis06]. XAF could have adopted this, but for reasons of performance the described process of five steps was chosen. Otherwise each `AnalysisRule` object would have executed its own XQuery. This would have required parsing the XML input file for each analysis function. This would have slowed down the analysis obviously and needlessly—especially for large XML input files.

While figure 5.2 is organised XAF-centric, the following figure 5.3 is TRex-centric and shows integration of XAF into TRex. On the top, the different TRex plug-ins are listed like the XAF for TTCN-3 plug-in and the Java-based pattern detection plug-in. Both make use of the static analysis framework of TPTP which in turn is based on Eclipse. On the right side, some of the TRex core functionalities are listed. On the left side, the XQuery-based analysis framework itself and the XQuery processor Saxon are displayed.

The following Eclipse plug-ins are modified respectively created during the implementation of XAF and the integration into TRex.

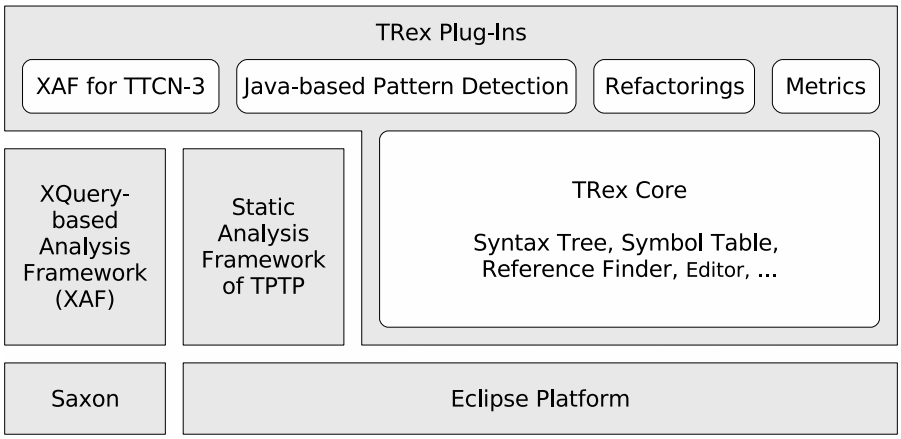

Figure 5.3: TRex overall architecture including the XQuery-based analysis framework

- The plug-in `de.ugoe.cs.swe.trex.patterndetection.ui` is modified to act as user interface component for the the pattern detection plug-in and the XAF TRex plug-in at the same time.
- The plug-in `de.ugoe.cs.swe.trex.patterndetection.tests` is modified to be extensible and act as basis for the XAF tests. See section 5.8 for details on this.
- The plug-in `de.ugoe.cs.swe.trex.tptp` is introduced to share TPTP-related code between the pattern detection plug-in and the XAF plug-in using TPTP.
- The plug-in `de.ugoe.cs.swe.xaf` is created to contain the XAF core of XQuery modules and related XML files.
- The plug-in `de.ugoe.cs.swe.trex.xaf` makes use of the preceding plug-in and is created to separate the XAF core from its integration into TRex. The TPTP-related classes described in this section 5.6 are contained in this plug-in.
- The plug-in `de.ugoe.cs.swe.trex.xaf.ui` contains the implementation of user-defined queries for TRex which are based on XAF. The implementation is described in detail in the following section 5.7.
- The plug-in `de.ugoe.cs.swe.saxon` wraps the Saxon packages into an Eclipse plug-in to be able to make use of Saxon from within of Eclipse.

5.7 User-Defined Queries

This section covers the implementation of XAF user-defined queries for TRex. Extending XAF itself with new analysis functions is possible by

adding new XQuery modules or by adding additional functions to existing XQuery modules. As this is not very comfortable for end-users, a user interface (UI) for creating and executing user-definition queries is provided. The plug-in `de.ugoe.cs.swe.trex.xaf.ui` contains the implementation which allows TRex users to enter and execute custom XQuery expression and to search for patterns in TTCN-3 source code using the XAF infrastructure. The search dialogue itself is located in the search menu of TRex and can alternatively be opened using the keyboard shortcut `Ctrl+H`. For displaying results of user-defined queries, a result view is implemented which displays matches grouped per file. Additionally, it is possible to store user-defined queries and to include these queries as TPTP rules in analysis runs.

As most of the Java code for the implementation of user-defined queries is related to the user interface, it is separated into the UI plug-in `de.ugoe.cs.swe.trex.xaf.ui`. Functionalities like executing XQuery expressions and parsing results are already implemented in the plug-in `de.ugoe.cs.swe.trex.xaf` and are reused by the UI plug-in. Sharing features is also the reason to store results of XQuery expressions in `XafResults` objects instead of directly mapping the XQuery results to TPTP result objects: `XafResults` objects are also used by the implementation of user-defined queries.

The search dialogue and the result view are implemented by extending the Eclipse extension point `org.eclipse.search.searchPages` with the class `SearchPage` and the extension point `org.eclipse.search.searchResultViewPages` with the class `SearchResultPage`. The `SearchPage` class is mainly responsible for filling the search dialogue with *Standard Widget Toolkit* (SWT) controls like a text box to enter the XQuery expression and other controls like a button to store the current query.

The button to execute the search is already predefined by Eclipse. Clicking it invokes the `run` method of the XAF class `SearchQuery` which first calls the method `createXML` of the `Analysis` class to generate the `ttnc3.xml` input file. Afterwards the user-defined XQuery is executed using the method `executeXQuery` which is also located in the `Analysis` class and the result is displayed in the result view. If no matches are found or the XQuery expression contains syntax errors, the user is informed using a pop-up message box including an appropriate advice.

Displaying and handling the results of user-defined queries is taken over by the XAF classes `SearchResult` and `SearchResultPage`. First, the returned `XafResults` object is transformed into `FileMatches` objects (one per file) as the result view displays the results grouped by files. The `FileMatches` objects in turn are encapsulated in `Match` objects which are provided by Eclipse and required to make use of the given interfaces of the `SearchResult` classes. Finally, the class `SearchResultPage` is responsible to deliver the matches in the desired format to predefined interfaces to allow the user to navigate through a tree of matches, to click on each match, and to open a TRex editor window with the selected match.

If the user stores an XQuery expression, the query and its name are stored using the XAF class `PersistentPreferences`. It makes use of the Eclipse class `PreferenceStore` which allows to persistently store arbitrary data. The serialisation of the according object is stored in the root of the TRex workspace directory as file `xaf-user-defined-queries.store`. It is a plain text file containing key-value pairs for each user-defined query respectively its name. Beside the purpose of storing queries to enrich the set of predefined analysis functions, additionally the last executed user-defined query is stored in this file. The query is loaded from the store when the search dialogue is opened again. This allows to refine queries step-by-step without starting with an empty search dialogue every time.

Figure 5.4 shows a screenshot of the TRex user interface for the user-defined queries. On the right side, the search dialogue is shown. On the left side, the result view is shown at the bottom of the screenshot. On the upper left side, a TTCN-3 file is opened in a TRex editor window and the found matches are highlighted using Eclipse search markers.

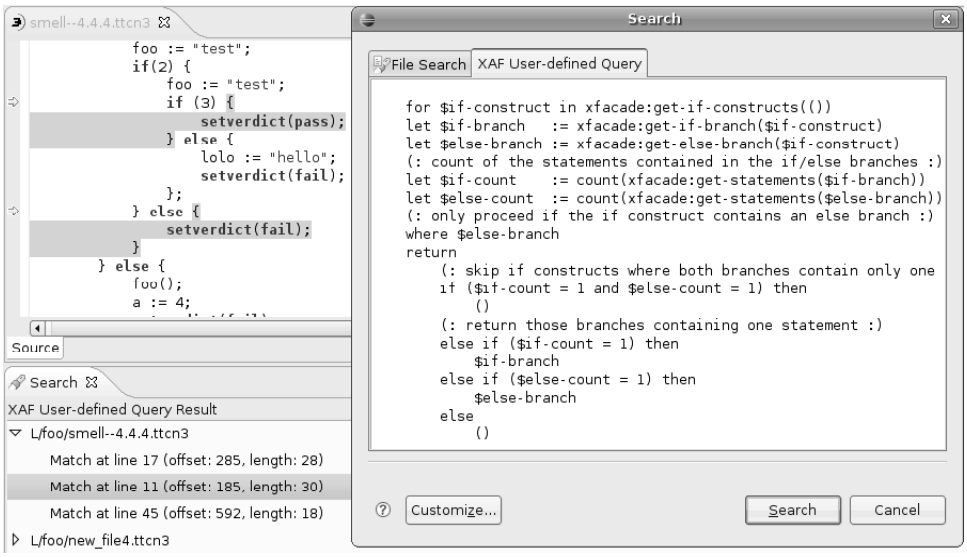

Figure 5.4: TRex user-defined query: search dialogue, results, and markers

To be able to run analyses including a mixture of predefined and user-defined queries, it is required to map user-defined queries from the store to TPTP rules. This is done analogous to the run-time creation of predefined queries in the method `getOwnedElements` of the class `AnalysisProvider` presented in section 5.6. The method `addUserDefinedCategoryAndRules` creates a category for all user-defined queries and iterates over the queries in the store. For each query an `AnalysisRule` object is created and added to the category. At the same time, an XQuery module containing all user-defined queries is created on-the-fly and stored as file `user.ttcn3.trex.xquery` with

the other XAF modules. This allows analysis runs to call functions from both modules—the module containing user-defined queries and the smell detection module.

User-defined queries themselves can access the facade layer module and also the smell detection module. This allows to create new smell detection functions—either based on existing ones or completely new ones. The implementation of user-defined queries is in closer contact to TRex and TTCN-3 compared to the abstracted smell detection module. Hence, it is possible to give up the policy that queries should be independent of their representation formats and analysis targets. Bypassing the facade layer permits user-defined queries to directly interact with the XML representation of the AST what affords writing compact and highly specialised queries.

In the following a few sample user-defined queries are presented. Listing 5.17 shows a query which checks function names for their conformance to the TTCN-3 Naming Conventions [ETS07d] proposed by the ETSI. It is recommended that function names should start with `f_` and therefore the query in listing 5.17 discovers those functions whose name does not start accordingly.

```
1 for $function in xfacade:get-functions(())
2 let $name := xfacade:get-identifier-up($function)
3 where not(starts-with($name, "f_"))
4 return $name
```

Listing 5.17: User-defined query using facade functions

The second user-defined query in listing 5.18 fulfils the same task as the first one, but while the first query in listing 5.17 consequently makes use of the facade layer functions to access the XML document, the query in listing 5.18 directly accesses the XML document. Direct access to the XML document permits to write queries which are much shorter than the once making use of the facade layer. Additionally, it is possible to access parts of the XML document for which no corresponding facade functions are available. (See appendix A.1 for all currently available facade functions.) The drawback of such queries is evident: they completely depend on the structure of the XML document and hence on TTCN-3. Additionally, their readability is worse than those making use of the facade functions.

```
1 $xfacade:root//(FunctionDef//Identifier)[1][not(starts-with(., 'f_'))]
```

Listing 5.18: User-defined query directly accessing the XML document

"Querying source code interactively [...] is a critical task" [PP94b] and a base technology regarding reverse engineering of software [EKW99]. Hence, the workflow of creating efficiently new user-defined queries for TTCN-3 source code using TRex is presented in the following.

The internal representation of the TTCN-3 source code as abstract syntax trees is based on the BNF grammar definition provided by the TTCN-3 specification [ETS07a]. For example, the naming and nesting of the nonterminal symbols of the TTCN-3 grammar is reflected by the abstract syntax tree. The *AST View* provided by TRex allows to visualise the abstract syntax tree of the currently displayed file. Marking places in the source code, highlights the according passage in the AST. Combining this TRex view and the information about the TTCN-3 grammar in the specification, helps to improve the understanding of the structure of the TTCN-3 core language and its tree representation.

The XML document which is used as data structure for the user-defined queries, is nearly a one-to-one mapping of the abstract syntax tree. This eases the creation of new queries as the names in the AST view and those of the XML elements and hence those used for the XQuery path expressions are all the same. The first step in the workflow towards a new user-defined query is that the user needs to know what to query for—in terms of source code. Marking the source code of interest and looking at the AST view, reveals those elements of the syntax tree which represent this currently marked piece of source code. As just mentioned, these names are also used by the XML document and can therefore be used to formulate user-defined queries.

For instance, marking the source code `myTimer.start(5);` would let the user know that the corresponding XML element is called `StartTimerStatement` and that the query `xfacade:get-root()//StartTimerStatement` finds all timers on which the `start` function is called. Some further investigations later, the user might be able to write a user-defined function which detects all started timers that are never stopped as shown in listing 5.19.

```
1 for $start in xfacade:get-root()//StartTimerStatement
2 let $block := xfacade:get-parent-block($start)
3 where not($block//StopTCStatement)
4 return $start
```

Listing 5.19: User-defined query to find never stopped timers

Still some improvements regarding user-defined queries might be reasonable. A tighter interaction between the AST view and the search dialogue of the user-defined queries is imaginable. For instance, each element of the AST view could provide a context menu with the option to start a query based on this node. This could open the search dialogue with an XQuery path expression based on the element that was clicked in the AST view. Another desireable improvement would be a live search: while typing an

XQuery expression, the matched source code would be highlight on-the-fly in the currently displayed file. Both possible improvements would permit an even faster and efficient creation of user-defined queries.

5.8 Tests for the Implementation

The integration of XAF into TRex is tested using JUnit 3.8 [JUn07]. Even though testing Eclipse plug-ins using JUnit is similar to the standard workflow of JUnit, the testcases must be executed as *JUnit Plug-In Test* (also called *Plug-In Development Environment* (PDE) JUnit Test). The main difference is that instead of "using the standard JUnit class `TestRunner`, PDE JUnit tests are executed by a special test runner that launches another Eclipse instance [...] and executes the test methods within that workbench."[17]

The JUnit testsuite for the XAF plug-ins is located in the plug-in `de.ugoe.cs.swe.trex.xaf.tests` and based on the testsuite for the pattern detection plug-ins [Bis06, Chap. 5.5]. This is done as both projects use the UI of the TPTP static analysis framework to interact with the user and therefore the pattern detection tests are already adapted to TPTP. To make use of the existing test infrastructure, the plug-in `de.ugoe.cs.swe.trex.patterndetection.tests` was changed to be extensible and the XAF test plug-in was created on top of that.

For each testcase the following files must be provided: one or more TTCN-3 source code files, a launch configuration file which selects the TPTP rules to include in the analysis run, and an according Java `TestCase` subclass which controls the test execution. Every testcase makes use of the class `PatternDetectionTestProject` to copy the source code and launch configuration files to the expected locations, parse the source code files, create the abstract syntax trees, and launch the TPTP analysis run. Finally, it is checked that the desired matches are found by the analysis.

This testing workflow covers the following parts of TRex respectively XAF:

Step 1: Starting TRex, creating a workspace and a TTCN-3 project, importing TTCN-3 source code files, and creating abstract syntax trees.

Step 2: Dynamic creation of the TPTP categories and rules of XAF based on the XML file `smells.ttcn3.xml`.

Step 3: Transformation of the abstract syntax trees to the XML representation of the AST.

Step 4: Execution of TPTP analysis runs and displaying the analysis results. This includes the following step:

[17] `http://wiki.eclipse.org/index.php/FAQ_What_is_a_PDE_JUnit_test%3F`

- Generation of XQuery expressions based on the TPTP rule selections and execution of the XQuery expressions.
- Parsing of the matches returned from XQuery and transformation of the matches to TPTP results.

Four JUnit test cases for different XQuery analysis functions are currently implemented. They cover different kinds of XQuery functions: one that calls Java methods of TRex from XQuery, one that returns matches in different TTCN-3 files, and two with multiple matches. Test cases for the user interface of the user-defined queries are currently not provided.

5.9 Discussion

Before going into the discussion of the implemented XQuery-based analysis framework, it is checked if the requirements for the framework as formulated in chapter 4 are met. The general requirements R1–R10 are already covered by the design and architecture of the framework as discussed in section 4.5. The requirements T1–T4 are specific to TTCN-3 and mainly demand a customisation of the framework towards the detection of bad smells in TTCN-3 source code and the integration into TRex. These requirements are fulfilled as presented in the preceding sections 5.3–5.6 by converting TTCN-3 source code into XML, implementing the facade layer for TTCN-3 using XQuery, describing and detecting bad smells of the TTCN-3 code smell catalogue using XQuery, and integrating XAF into TRex and the static analysis framework of TPTP. The subordinate requirements U1–U5 of requirement R10 demand the possibility to add user-defined pattern and mainly concern the user interface of TRex. The according implementation is described in section 5.7 and fulfils all but the optional requirement U5 which requests support to view, edit, delete, and rename stored user-defined queries.

This leads directly to the limitations of the current implementation regarding the integration of XAF into TRex and the implementation of user-defined queries. No limitation concerns the XQuery core of XAF. Most of the following items are due to the limitations of the TPTP static analysis framework respectively the way XAF makes use of it.

- TPTP measures the overall time of an analysis run as sum of the times each rule takes. As all XQuery analysis functions are executed in one step, only the overall time the execution takes can be presented by TPTP instead of the time information for each rule.

- Normally, each TPTP rule features a severity property. For dynamically added rules, like those of XAF, this severity property is not available.

- Displaying duplicates using TPTP is not very well arranged as all duplicates of one rule appear in a flat list of matches. It is desireable to extend TPTP to allow displaying matches belonging together in separate groups.
- TPTP allows to link so-called quick fixes to TPTP rules. These quick fixes suggest refactorings to remove the bad smell and make therefore use of the refactoring capabilities of TRex. Currently, XAF does not implement any quick fixes.
- There is no possibility to persistently skip false positives or undesired matches for future analysis runs. This could be implemented for instance using source code comments containing special instructions to skip a match or using a database which reminds undesired matches by their position in the source code.

Regarding user-defined queries, the following limitations are known:

- As already mentioned, currently no user interface to view, edit, delete, and rename stored queries is implemented (requirement U5).
- Unlike TPTP analysis runs, the search dialogue of the user-defined queries does not support scope-aware searches. All TTCN-3 files of the currently opened TRex projects are included in the search. As soon as a query is stored and executed as TPTP rule it uses the scope of TPTP.
- The text box of the search dialogue where user-defined queries are entered, features no undo capability or other desireable features like syntax highlighting for XQuery.

Removing these limitations would make the framework more user-friendly and powerful, but the currently implemented functions already provide a smooth integration of the predefined XQuery functions. Additionally, user-defined queries provide a powerful way to extend the framework with new smell detection functions and other patterns. Regarding the application and performance of the framework it is referred to section 6.1.

Chapter 6

Application and Extension

The current chapter presents results of the application of the XQuery-based analysis framework for the detection of bad smells in existing, commercially used TTCN-3 test suites (section 6.1). Additionally, this chapter reveals possibilities for the extension of the framework towards new analysis targets (section 6.2) and towards new types of patterns (section 6.3) beside this thesis' focus on TTCN-3 and bad smells.

6.1 Detection of Bad Smells in TTCN-3 Test Suites

In this section, the results of the bad smell detection in existing TTCN-3 test suites using XAF are discussed. This is done mainly for two reasons: first to check the implementation of XAF for correctness and usability. The second reason is to reveal the need for automated bad smell detection even for commercially used TTCN-3 test suites.

The following TTCN-3 test suites[1] are publicly available and used to perform the tests: Session Initiation Protocol (SIP) [ETS06a], Internet Protocol version 6 (IPv6) [ETS06c], and Digital Mobile Radio (DMR) [ETS06b]. All tests are performed using the following 32-bit software versions: Debian GNU/Linux version 4.0, Sun Java 6 SE Runtime Environment (1.6.0_03-b05), Eclipse SDK 3.3.1 (M20070921-1145), Eclipse TPTP 4.4.0.3, Saxon-B 8.9.0.4, XAF 1.0.0, and the currently latest available version of TRex[2]. The test system was equipped with 2 GiB of RAM and an AMD Athlon 64 single-core processor with 2 GHz.

The results of the test runs are presented in table 6.1. The first column contains the names of the analysis functions and their parameter settings. The other three columns contain the number of smell instances found for each function. Additionally, the time consumption for the complete analysis run is given in the last row. The overall time consumption includes the

[1] `http://www.ttcn-3.org/PublicTTCN3TestSuites.htm`
[2] Subversion branch `trunk`, timestamp 2007-10-17

Test Suite	SIP	IPv6	DMR
Version	v4.1.1	v1.1.1	v1.2.1
Lines of Code	55510	24462	15485
Avoid Goto Statement	–	–	–
Magic Numbers (not 0,0.0,1,1.0)	427	331	109
Long Statement Block (120+ lines)	58	4	–
Nested Conditional	178	58	58
Short Template (2 or less fields)	62	39	88
Short Template (50 or less chars)	31	21	65
Deactivation On Another Level	–	–	–
Missing Activation	–	–	–
Missing Deactivation	686	367	326
Unreachable Default	–	–	–
Duplicate Alt Branches	5	26	48
Duplicate Code in Conditionals	192	111	90
Fully Parameterized Template	–	–	–
Long Parameter List (6+ parameters)	108	8	2
Missing Log	692	13	389
Missing Verdict (in alt branches)	3	–	–
Stop In Function	313	5	5
Idle PTC	–	–	7
Duration seconds (XQuery/total)	142/170	34/43	30/36

Table 6.1: Bad smells found by XAF in TTCN-3 test suites

following steps: creation of the XML representation of the abstract syntax tree (the time required for building the AST is not included), execution of the XQuery expressions (this includes reading the XML file `ttcn3.xml`), and displaying the matches in the TPTP view inclusive Eclipse markers. Also the time the execution of the XQuery expressions takes (without creating the XML file and displaying the results) is given.

First of all, the duration of the analysis runs shows that the execution of the XQuery expressions took between a half minute for the DMR test suite and about two and a half minutes for the SIP test suite. As the latter consists of about 55000 lines of code while the DMR suite consists of about 15000 lines of code, it can be assessed that the framework scales well. Additional to the time the XQuery executions take, about 20% extra time is consumed by the TRex for the creation of the input XML file and the parsing and displaying of the results. The performance can be improved by using a further optimised XQuery processor like Saxon-SA instead of Saxon-B (see section 5.1 for details on this).

The detection results can be parted in two groups. The first one contains those analysis functions which found no smell instances at all. The reasons

for this result can be various: the smell detection functions might be written in a way to miss existing smells or there are no instances of these smells in the investigated test suites. For some smell detection functions like *Avoid Goto Statement* it can be assessed that they work as intended and miss no smell instances. Another possibility is that the functions are working correct and the smell descriptions which originate from an academic background are not existing in real-world test suites.

The second group contains those smell detection functions that returned matches. The bad smell *Missing Deactivation* seem to be very common. But as defaults are automatically deactivated at the end of their lifetimes, the consequences of this smell are not drastic. The current implementation of the detection function to find instances of the smell *Missing Log* might return false positive matches. This is due to the fact that log operations called in subordinate functions are currently not taken into account. The majority of the functions like *Magic Numbers*, *Long Statement Block*, and *Duplicate Alt Branches* finds smell instances without false positives and negatives and indicate a lot of refactoring opportunities.

Only the 18 pure XQuery analysis functions of the currently 20 implemented ones are included in the test runs discussed above. The two functions calling Java methods of the TRex reference finder, are left out due to the fact that the TRex implementation of the reference finder is slow and would have adulterated the performance results. To show how large the effect of the invocations of these TRex methods is, two additional analysis runs are performed using the Java-based pattern detection of Bisanz [Bis06] with the DMR test suite [ETS06b]: the first run consists of all eleven implemented rules of the Java-based pattern detection and took 476 seconds. The second run includes only those seven rules not using the reference finder of TRex and took only 43 seconds. A similar performance loss can be observed with XAF when the reference finder of TRex is invoked from within XQuery functions. To show that the invocation of Java methods from within XQuery is not the bottleneck—but the referencing counting itself is slow—the Java-only implementation of Bisanz is chosen for this two test runs.

To compare the performance of TRex' Java-based pattern detection and XAF, four smell detection functions which are implemented by both approaches (*Magic Number*, *Short Template*, *Duplicate Alt Branches*, and *Fully Parameterized Template*) are used for analysis runs with the DMR test suite [ETS06b]. The analysis run with the Java-based approach takes 2.4 seconds while the XAF run takes 22.3 seconds. The reasons for this obvious difference are various: while the Java-based approach starts analysing the AST which was created before, XAF first needs to convert the AST to XML (what takes about three seconds for the DMR test suite) and parse the XML document before starting the actual analysis. Additional test runs using an XQuery debugger[3] revealed that most of the time is spent by the

[3] `http://www.oxygenxml.com/xquery_debugger.html`

duplicate search and highlighted some potential optimisation opportunities. Altogether, the performance loss seams to be the price that needs to be paid for a declarative approach.

6.2 Extension of the Framework to New Analysis Targets

To prove the extensibility of the XQuery-based analysis framework, two new analysis targets and according representation formats are integrated as experimental implementations. As additional analysis targets UML models (created with ArgoUML [Arg07]) and Java source code are used. They are represented using XMI (see section 2.4.3.1, also created using ArgoUML) respectively srcML (see section 2.4.3.2). The XQuery analysis function `long-parameter-list` (see section 5.5) is applied to TTCN-3, UML, and Java to detect instances of the bad smell *Long Parameter List* in these different analysis targets. Therefore according facade layer modules for XMI and srcML are implemented which only consist of those facade functions required for the detection of this bad smell. All required new facade functions are shown in listing 6.1 (UML/XMI) respectively listing 6.2 (Java/srcML). Compared to those for TTCN-3 (see appendix A.1) only the XML element names had to be changed to the according names of the XMI respectively srcML format.

```
1 declare namespace UML = "org.omg.xmi.namespace.UML";
2 declare function xfacade:get-functions($n as node()*) {
3   xfacade:get-node($n)//UML:Operation
4 };
5 declare function xfacade:get-parameters($n as node()*) {
6   xfacade:get-node($n)//UML:BehavioralFeature.parameter/UML:Parameter
7 };
```

Listing 6.1: Facade functions for UML in XMI/ArgoUML flavour

```
1 declare function xfacade:get-functions($n as node()*) {
2   xfacade:get-node($n)//function
3 };
4 declare function xfacade:get-parameters($n as node()*) {
5   xfacade:get-node($n)//parameter_list/param
6 };
```

Listing 6.2: Facade functions for Java in srcML flavour

The positive results of this prototypical integration of UML models and Java source code demonstrates that the usage of XML as representation format allows a smooth extension of the framework towards new analysis targets. The application of the facade layer provides a level of abstraction

that allows to write analysis functions to be independent of the underlying analysis target and representation format. This allows to implement analysis functions once and make use of them to discover pattern in different software artefacts.

The analysis targets presented up to now reflect static structures of source code respectively software architectures. The core framework is also adoptable for the analysis of software behaviour. Given an XML representation of software behaviour, it is possible to adopt the framework to this scenario. Potential XML formats capable of representing software behaviour are introduced in section 2.4.3.2.

Patterns regarding software behaviour are different from those that can be found in static software artefacts. While switching from one representation format of static software artefacts to another requires relatively small changes, the initial creation of a facade layer for the access to a behavioural representation format requires a higher effort. This is due to the fact that the underlying representation formats differ significantly from those representing static software artefacts. Still the process is the same as for the presented facade layer for TTCN-3: identify information required by the analysis layer and provide abstract access to these information using XQuery facade functions.

6.3 Extension of the Framework to New Types of Patterns

The framework is designed to be extensible regarding new analysis target and also regarding new types of patterns. This section covers the possibility to make use of XAF for the integration of new pattern types. It provides a concrete example how the framework could be used for the calculation of source code metrics. Additionally, a possibility is discussed to adopt the framework for the recognition of test purposes in traces of test case executions.

As discussed in section 3.1, source code metrics which exceed their boundary value, can indicate instances of bad smells and can therefore be used for the smell detection. But the original purpose of source code metrics is software measurement which is a stand-alone type of pattern. The XQuery-based analysis framework can be extended to the calculation of source code metrics as listing 6.3 shows. The provided example calculates the simple metric *Lines of Code* which counts all non-empty and non-comment lines of a source code file. Using XQuery and the facade layer of TTCN-3 is is done by iterating over all files (line 1) and returning for each file an XML element `lines-of-code` (line 2). Each of these elements contains the according filename as attribute (line 3) and the number of lines of code as content. The calculation of the lines of code is based on the location information contained

in the XML document: the line information where each entity of the source code starts and ends are joined in one sequence (line 5) and the total number of the distinct values equals the lines of code.

```
1 for $file in xfacade:get-files(())
2 return element lines-of-code {
3   xfacade:get-filename($file),
4   count(distinct-values(
5     (xfacade:get-lines($file), xfacade:get-line-ends($file))
6   ))
7 }
```

Listing 6.3: XQuery implementation of the metric *Lines of Code*

As the implementation indicates, XQuery can be used to express arbitrary metrics. More metrics based on XQuery, like *Number of Methods* and *Lack of Cohesion in Methods*, are discussed in [Eic07].

One desireable goal regarding testing is to examine if the implemented tests fulfil their purposes. Test purposes are prose descriptions of testing objectives regarding conformance testing.[4] They can be formulated using natural language, but also using more formal, structured approaches. Test purposes can be seen as patterns in the test behaviour. The goal is to retarget test purposes in the according behaviour representations of test cases. This is only possible, if the purposes are specified using a formal or semi-formal approach like the *Testing Purpose Language* (TPLan) [SWR07]. This increases the possibility to recover the purposes from the test cases. "TPLan has been designed to make test purpose specification more formal without inhibiting the expressive power of prose" [SWR07].

Listing 6.4 shows the basic structure of TPLan test purpose specifications. Each test purpose needs an unique identifier (line 1) and a `with` condition (line 2) which contains specific initial condition required for the test purpose to be valid. The `ensure that` part (lines 3–6) contains the actual critical verdict criteria for a test in form of a stimulus and response to ensure that the requirements are met [SWR07]. The parts written between "<" and ">" need to be filled with predefined or user-defined entities, events, values, units, conditions, and words describing the actual purpose. The degree of prose used for this is up to the author of the test purposes.

The optional requirement T5 demands possibilities for the detection of test purposes in TTCN-3 source code or in behavioural representations of it. As currently no tool is available which allows to extract behavioural representations of TTCN-3 test suite executions (for instance traces of the execution), this requirement could not be investigated in depth. Even though it is currently not implemented, it should be possible to make use of the software analysis framework to perform analyses for the detection of test

[4] `http://portal.etsi.org/mbs/Testing/conformance/conformance.htm#TP`

```
1 TP id: <string>
2 with { <pre-conditions> }
3 ensure that {
4   when { <stimulus> }
5   then { <response> }
6 }
```

Listing 6.4: Basic structure of TPLan test purpose specification

purposes, given the following prerequisites: an XML representation reflecting the behaviour of TTCN-3 test suite executions and the availability of an accordingly adopted facade layer module as described in section 6.3. Additionally, the test purposes must be specified in formal way, for instance using TPLan. This would allow a transformation of the purposes into XQuery expressions and to check if they can be found in the traces of the test case execution.

Chapter 7

Summary and Outlook

This thesis presented a generic XML-based approach for software analysis and an accordingly implemented framework. The most important goal is the possibility to describe software patterns like bad smells using the declarative XML query language XQuery. Another major achievement is the independence of the patterns regarding the underlying programming languages respectively software artefacts. This combination allows to describe patterns in a generic way and on a high level of abstraction. Additionally, it enhances the readability and reusability of the patterns.

The software analysis framework is adopted towards the testing language TTCN-3 and the detection of bad smells in TTCN-3 test suites. Currently 20 functions for the detection of bad smells are implemented and the framework is integrated in the TTCN-3 tool TRex. Additionally to the smell detection, user-defined queries for TRex are implemented which permit to interactively query TTCN-3 source code. The results of the smell detection in existing test suites reveal the need for automated smell detection and highlight the usability of the framework.

The prototypical adoption of the framework towards new types of patterns and the integration of new analysis targets gives an outlook towards additional fields of application. The presented solution is mainly focused on structural issues like the detection of bad smells and can be advanced towards related patterns like bug patterns, design defects, and design patterns. Additional research can also be done into the direction of data and control flow anomalies and the analysis of XML-based traces of dynamic software behaviour.

Abbreviations and Acronyms

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
BNF	Backus–Naur Form
DOM	Document Object Model
DSL	Domain-Specific Languages
DTD	Document Type Definition
EJB	Enterprise JavaBean
EPL	Eclipse Public License
ETSI	European Telecommunications Standards Institute
GCC	GNU Compiler Collection
GenTL	Generic Transformation Language
GNU	GNU is not Unix
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
ISO	International Organization for Standardization
ISTQB	International Software Testing Qualification Board
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
JavaML	Java Markup Language
LGPL	GNU Lesser General Public License
LMP	Logic Meta-Programming

MOF ..... Meta-Object Facility

MPL ..... Mozilla Public License

MTC ..... Main Test Component

OMG ..... Object Management Group

OSI ..... Open Source Initiative

PDE ..... Plug-In Development Environment

PQL ..... Program Query Language

RE ..... Regular Expression

SAX ..... Simple API for XML

SCA ..... Source Code Algebra

SCRUPLE ..... Source Code Retrieval Using Pattern Languages

SGML ..... Standard Generalized Markup Language

SQL ..... Structured Query Language

srcML ..... Source Code Markup Language

StAX ..... Streaming API for XML

SUT ..... System Under Test

SWT ..... Standard Widget Toolkit

TPLan ..... Testing Purpose Language

TPTP ..... Test & Performance Tools Platform

TQL ..... Tree Query Language

TRex ..... TTCN-3 Refactoring and Metrics Tool

TTCN-3 ..... Testing and Test Control Notation Version 3

UI ..... User Interface

UML ..... Unified Modeling Language

W3C ..... World Wide Web Consortium

XAF ..... XQuery-based Analysis Framework [for Software]

XMI ..... XML Metadata Interchange

XML ..... Extensible Markup Language

XPath ..... XML Path Language

XQJ ..... XQuery API for Java

XQuery ..... XML Query Language

XSL ..... Extensible Stylesheet Language

XSLT ..... XSL Transformations

Bibliography

[ADB04] Ademar Aguiar, Gabriel David, and Greg Badros. JavaML 2.0: Enriching the Markup Language for Java Source Code. *XML: Aplicações e Tecnologias Associadas (XATA 2004), Porto, Portugal*, 2004. `http://www.di.uminho.pt/~jcr/XML/conferencias/xata2004/artigos/AdemarAguiar/JavaML2.pdf`.

[AEK05] Raihan Al-Ekram and Kostas Kontogiannis. An XML-Based Framework for Language Neutral Program Representation and Generic Analysis. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 42–51, Washington, DC, USA, 2005. IEEE Computer Society.

[AG06] Darren Atkinson and William Griswold. Effective Pattern Matching of Source Code Using Abstract Syntax Patterns. *Software – Practice & Experience*, 36(4):413–447, 2006.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York, USA, 1977.

[AK05] Darren Atkinson and Todd King. Lightweight Detection of Program Refactorings. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 663–670, Washington, DC, USA, 2005. IEEE Computer Society.

[AK07] Malte Appeltauer and Günter Kniesel. Towards Concrete Syntax Patterns for Logic-Based Transformation Rules. In *Proceedings of the Eighth International Workshop on Rule-Based Programming*, 2007. Electronic Notes in Theoretical Computer Science, Elsevier, Amsterdam, Netherlands.

[All02] Eric Allen. *Bug patterns in Java.* Apress, New York, USA, 2002.

[AMS02] Jennitta Andrea, Gerard Meszaros, and Shaun Smith. Catalog of XP Project Smells. *The Third International Conference on eXtreme Programming and Agile Processes in Software Engineering*, 2002.

[ANT07] ANTLR website. `http://www.antlr.org`, 2007.

[AP05] Marcus Alanen and Ivan Porres. Model Interchange Using OMG Standards. In *EUROMICRO '05: Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 450–459, Washington, DC, USA, 2005. IEEE Computer Society.

[APMV03] Giuliano Antoniol, Massimiliano Di Penta, Gianluca Masone, and Umberto Villano. XOgastan: XML-oriented GCC AST Analysis and Transformations. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 173–182, Washington, DC, USA, 2003. IEEE Computer Society. `http://xogastan.sourceforge.net`.

[Arg07] ArgoUML website. `http://argouml.tigris.org`, 2007.

[Bad00] Greg Badros. JavaML: A Markup Language for Java Source Code. *Computer Networks*, 33(1-6):159–177, 2000.

[BCE06] Byte Code Engineering Library (BCEL) website. `http://jakarta.apache.org/bcel/`, 2006.

[BGA06] Salah Bouktif1, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Extracting Change-patterns from CVS Repositories. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[Bis06] Martin Bisanz. Pattern-based Smell Detection in TTCN-3 Test Suites. Master's thesis, Institute for Informatics, ZFI-BM-2006-44, ISSN 1612-6793, Center for Informatics, University of Göttingen, 2006.

[BMG+94] Erich Buss, Renato De Mori, Morven Gentleman, John Henshaw, Howard Johnson, Kostas Kontogiannis, Ettore Merlo, Hausi Muller, John Mylopoulos, Santanu Paul, Atul Prakash, Martin Stanley, Scott Tilley, Joel Troster, and Kenny Wong. Investigating Reverse Engineering Technologies for the CAS Program Understanding Project. *IBM Systems Journal*, 33(3):477–500, 1994.

[BMMM98] William Brown, Raphael Malveau, Hays McCormick, and Thomas Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, USA, 1998.

[Bru04] Michael Brundage. *XQuery: The XML Query Language*. Pearson Higher Education, Boston, MA, USA, 2004.

[BSMY+06] Tim Bray, Michael Sperberg-McQueen, François Yergeau, Eve Maler, and Jean Paoli. Extensible Markup Language (XML) 1.0 (Fourth Edition). World Wide Web Consortium Recommendation, August 2006. http://www.w3.org/TR/2006/REC-xml-20060816.

[BYC+06] Tim Bray, François Yergeau, John Cowan, Eve Maler, Jean Paoli, and Michael Sperberg-McQueen. Extensible Markup Language (XML) 1.1 (Second Edition). World Wide Web Consortium Recommendation, August 2006. http://www.w3.org/TR/2006/REC-xml11-20060816.

[BYM+98] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE Computer Society, 1998.

[CBF+07] Don Chamberlin, Anders Berglund, Mary Fernández, Jonathan Robie, Jérôme Siméon, Scott Boag, and Michael Kay. XML Path Language (XPath) 2.0. World Wide Web Consortium Recommendation, January 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123.

[CG04] Luca Cardelli and Giorgio Ghelli. TQL: A Query Language for Semistructured Data Based on the Ambient Logic. *Mathematical Structures in Computer Science*, 14(3):285–327, 2004.

[CH04] James Coplien and Neil Harrison. *Organizational Patterns of Agile Software Development*. Prentice-Hall, Upper Saddle River, NJ, USA, 2004.

[Che07] Checkstyle website. http://checkstyle.sourceforge.net, 2007.

[Cla99] James Clark. XSL Transformations (XSLT) Version 1.0. World Wide Web Consortium Recommendation, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[CM01] James Clark and MURATA Makoto. RELAX NG Specification. OASIS Committee Specification, December 2001. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

[CMM04] Glauco Carneiro, Manoel Mendonça, and José Carlos Maldonado. Automatic Detection of Refactoring Opportunities. *Transactions on Software Engineering*, 2004. http://lens.cos.ufrj.br:8080/eselaw/papers/interestedareas/eselaw19.

[Con96] The Unicode Consortium. *The Unicode standard, Version 2.0*. Addison-Wesley, Boston, MA, USA, 1996.

[Con06] The Unicode Consortium. *The Unicode Standard, Version 5.0*. Addison-Wesley, Boston, MA, USA, 2006.

[Cop05] Tom Copeland. *PMD Applied*. Centennial Books, Colorado Springs, Colorado, USA, 2005.

[Cre97] Roger Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages*, pages 229–242, Berkeley, CA, USA, 1997. USENIX Association.

[CS63] Noam Chomsky and Marcel Paul Schützenberger. The algebraic theory of context-free languages. In *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing, 1963.

[DC99] Steven DeRose and James Clark. XML Path Language (XPath) Version 1.0. World Wide Web Consortium Recommendation, November 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[Dev92] Premkumar Devanbu. GENOA: A Customizable Language- and Front-end Independent Code Analyzer. In *ICSE '92: Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, New York, NY, USA, 1992. ACM Press.

[DMJ01] Steve DeRose, Eve Maler, and Ron Daniel Jr. XML Pointer Language (XPointer) Version 1.0. World Wide Web Consortium, Last Call Working Draft, January 2001. http://www.w3.org/TR/2001/WD-xptr-20010108.

[DOM05] Document Object Model (DOM) website of the World Wide Web Consortium. http://www.w3.org/DOM/, 2005.

[Ecl07a] Eclipse website. http://www.eclipse.org, 2007.

[Ecl07b] Eclipse Modeling Framework Project (EMF) website. http://www.eclipse.org/emf/, 2007.

[Ecl07c] Eclipse Test & Performance Tools Platform Project (TPTP) website. http://www.eclipse.org/tptp/, 2007.

[Eic07] Michael Eichberg. *Open Integrated Development and Analysis Environments*. PhD thesis, Technical University Darmstadt, Department of Computer Science, 2007.

[EKW99] Juergen Ebert, Bernt Kullbach, and Andreas Winter. Querying as an Enabling Technology in Software Reengineering. In *CSMR '99: Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 42–50, Washington, DC, USA, 1999. IEEE Computer Society.

[EMOS04] Michael Eichberg, Mira Mezini, Klaus Ostermann, and Thorsten Schafer. XIRC: A Kernel for Cross-Artifact Information Engineering in Software Development Environments. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 182–191, Washington, DC, USA, 2004. IEEE Computer Society.

[ETS01] ETSI. Standard ES 201 873-1 V1.1.2 (2001-06), Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, June 2001.

[ETS06a] ETSI. Technical Specification TS 102 027-3 V4.1.1 (2006-07): SIP ATS & PIXIT; Part 3: Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2006.

[ETS06b] ETSI. Technical Specification TS 102 362-3 V1.2.1 (2006-06): Conformance testing for the Digital Mobile Radio (DMR), Part 3: Abstract Test Suite (ATS). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2006.

[ETS06c] ETSI. Technical Specification TS 102 516 V1.1.1 (2006-04): IPv6 Core Protocol; Conformance Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT) proforma. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, 2006.

[ETS07a] ETSI. Standard ES 201 873-1 V3.2.1 (2007-02), Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, February 2007.

[ETS07b] ETSI. Standard ES 201 873-2 V3.2.1 (2007-02), Part 2: TTCN-3 Tabular Presentation Format (TFT). European

Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, February 2007.

[ETS07c] ETSI. Standard ES 201 873-3 V3.2.1 (2007-02), Part 3: Graphical Presentation Format for TTCN-3 (GFT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, February 2007.

[ETS07d] ETSI. TTCN-3 Naming Conventions website. `http://www.ttcn-3.org/NamingConventions.htm`, 2007.

[eXi07] eXist – Open Source native XML Database website. `http://exist.sourceforge.net`, 2007.

[Fin07] FindBugs website. `http://findbugs.sourceforge.net`, 2007.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing, Boston, MA, USA, 1999.

[Fow03] Martin Fowler. *UML Distilled*. Addison-Wesley, Boston, MA, USA, 3rd edition, 2003.

[Fox06] John Fox. *Introduction to Software Engineering Design: Processes, Principles, and Patterns with UML2*. Pearson Education, Boston, MA, USA, 2006.

[FP97] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co. Boston, MA, USA, 1997.

[FRC06] Daniela Florescu, Jonathan Robie, and Don Chamberlin. XQuery Update Facility. World Wide Web Consortium Working Draft, July 2006. `http://www.w3.org/TR/2006/WD-xqupdate-20060711`.

[FY94] Chin-Feng Fan and Swu Yih. Prescriptive Metrics for Software Quality Assurance. In *Proceedings of the First Asia-Pacific Software Engineering Conference*, pages 430–438. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

[GAA01] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects. In *TOOLS '01: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, pages 296–305, Washington, DC, USA, 2001. IEEE Computer Society.

[GDB02] Timothy Grose, Gary Doney, and Stephen Brodsky. *Mastering XMI: Java Programming with XMI, XML, and UML.* John Wiley & Sons, New York, USA, 2002.

[GEJ03] Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Metamodel. *Proceedings of the 1st International FUJABA Days*, pages 17–24, 2003. http://wwwcs.uni-paderborn.de/cs/fujaba/.

[Gen07] Generic Transformation Language GenTL website. https://sewiki.iai.uni-bonn.de/research/gentl/start, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, MA, USA, 1995.

[GM05] Steve Gutz and Orlando Marquez. TPTP Static Analysis Tutorial Part 1 – A Consistent Analysis Interface. http://www.eclipse.org/tptp/home/documents/process/development/static_analysis/TPTP_static_analysis_tutorial_part1.html, 2005.

[GM06a] Steve Gutz and Orlando Marquez. TPTP Static Analysis Tutorial Part 2 – Enhancing Java Code Review. http://www.eclipse.org/tptp/home/documents/process/development/static_analysis/TPTP_static_analysis_tutorial_part2.html, 2006.

[GM06b] Steve Gutz and Orlando Marquez. TPTP Static Analysis Tutorial Part 3 – Integrating Your Own Analysis. http://www.eclipse.org/tptp/home/documents/process/development/static_analysis/TPTP_static_analysis_tutorial_part3.html, 2006.

[Gué02] Yann-Gaël Guéhéneuc. Three Musketeers to the Rescue – Meta-modelling, Logic Programming, and Explanation-based Constraint Programming for Pattern Description and Detection. *Proceedings of the 1st ASE Workshop on Declarative Meta-Prorgramming, University of British Columbia, Computer Science Department*, 2002.

[Hal77] Maurice Halstead. *Elements of Software Science.* Elsevier, New York, USA, 1977.

[Har01] Elliotte Rusty Harold. *XML Bible.* John Wiley & Sons, New York, USA, 2nd edition, 2001.

[HM04] Elliotte Rusty Harold and Scott Means. *XML in a Nutshell.* O'Reilly Media, Sebastopol, CA, USA, 3rd edition, 2004.

[HP03a] Haruo Hosoya and Benjamin Pierce. Regular Expression Pattern Matching for XML. *Journal of Functional Programming*, 13(06):961–1004, 2003.

[HP03b] Haruo Hosoya and Benjamin Pierce. XDuce: A Statically Typed XML Processing Language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[HP04] David Hovemeyer and William Pugh. Finding Bugs Is Easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[HVdM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable Source Code Queries with Datalog. In Dave Thomas, editor, *ECOOP'06: Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, Berlin, Germany, 2006. Springer.

[HWS00] Richard Holt, Andreas Winter, and Andy Schürr. GXL: Toward a Standard Exchange Format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE 2000)*, pages 162–171. IEEE Computer Society, 2000. `http://www.gupro.de/GXL/`.

[Int06] International Software Testing Qualification Board (ISTQB). Standard Glossary of Terms Used in Software Testing, Version 1.2. June 2006.

[ISO04] ISO/IEC. ISO/IEC Standard No. 9126: Software Engineering, Product Quality, Parts 1–4. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland, 2001-2004.

[IT06a] ITU-T. Standard Z.140 (03/2006): Testing and Test Control Notation version 3 (TTCN-3): Core language. ITU Telecommunication Standardization Sector (ITU-T), Geneva, Switzerland, 2006.

[IT06b] ITU-T. The Evolution of TTCN. 2006. `http://www.itu.int/ITU-T/studygroups/com17/ttcn.html`.

[Jac07] The Jackpot Project website. `http://jackpot.netbeans.org`, 2007.

[JR00] Daniel Jackson and Martin Rinard. Software Analysis: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 133–145, New York, NY, USA, 2000. ACM Press.

[JRH99] Ian Jacobs, David Raggett, and Arnaud Le Hors. HTML 4.01 specification. World Wide Web Consortium Recommendation, December 1999. http://www.w3.org/TR/1999/REC-html401-19991224.

[JUn07] JUnit testing framework website. http://www.junit.org, http://junit.sourceforge.net, 2007.

[Kay07] Michael Kay. XSL Transformations (XSLT) Version 2.0. World Wide Web Consortium Recommendation, January 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123.

[KBSD04] Suraj Kothari, Luke Bishop, Jeremias Sauceda, and Gary Daugherty. A Pattern-Based Framework for Software Anomaly Detection. *Software Quality Journal*, 12(2):99–120, 2004.

[KdMMB96] Kostas Kontogiannis, Renato de Mori, Ettore Merlo, and Morris Bernstein. Pattern Matching for Clone and Concept Detection. *Applied Categorical Structures*, 3(1):77–108, 1996.

[Kem05] Jochen Kemnade. Development of a Semantics-aware Editor for TTCN-3 as an Eclipse Plug-in. Bachelor's thesis, Institute for Informatics, ZFI-BM-2005-19, ISSN 1612-6793, Center for Informatics, University of Göttingen, 2005.

[Kep04] Stephan Kepser. A Simple Proof for the Turing-Completeness of XSLT and XQuery. 2004. http://www.idealliance.org/papers/extreme/proceedings/html/2004/Kepser01/EML2004Kepser01.html.

[Ker05] Joshua Kerievsky. *Refactoring to patterns*. Addison-Wesley, Boston, MA, USA, 2005.

[KHR07] Günter Kniesel, Jan Hannemann, and Tobias Rho. A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction. In *LATE '07: Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution*, New York, NY, USA, 2007. ACM Press.

[KMP07] Nicholas Kraft, Brian Malloy, and James Power. An Infrastructure to Support Interoperability in Reverse Engineering. *Information and Software Technology*, 49(3):292–307, 2007.

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming: Volume 1, Fundamental Algorithms*. Addison-Wesley, Boston, MA, USA, 3 edition, 1997.

[LTBH06] Andrew Layman, Richard Tobin, Tim Bray, and Dave Hollander. Namespaces in XML 1.0 (Second Edition). World Wide

Web Consortium Recommendation, August 2006. http://www.w3.org/TR/2006/REC-xml-names-20060816.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2nd edition, 1999.

[MBMB02] Stephen Mellor, Marc Balcer, Stephen Mellor, and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, Boston, MA, USA, 2002.

[McC76] Thomas McCabe. A Complexity Measure. *IEEE Transactions of Software Engineering*, 2(4):308–320, 1976.

[MCK02] Jonathan Maletic, Michael Collard, and Huzefa Kagdi. Source Code Files as Structured Documents. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, pages 289–292, Washington, DC, USA, 2002. IEEE Computer Society. http://www.sdml.info/projects/srcml/.

[MCK04] Jonathan Maletic, Michael Collard, and Huzefa Kagdi. Leveraging XML Technologies in Developing Program Analysis Tools. In *Proceedings of the 4th International Workshop on Adoption-Centric Software Engineering (ACSE'04) - ICSE'04 Workshop*, 2004.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, Boston, MA, USA, 2007.

[MFM04] Nabor Mendonça, Leonardo Fonseca, and Paulo Henrique Maia. Towards Reusable Code Analysis Tools Using Standard XML Technologies. *Anais do I Workshop de Ciências da Computação e Sistemas da Informação da Região Sul (WORKCOM-SUL), Palhoça, SC, Maio*, 2004. http://inf.unisul.br/~ines/workcomp/cd/pdfs/2404.pdf.

[MG05] Naouel Moha and Yann-Gaël Guéhéneuc. On the Automatic Detection and Correction of Software Architectural Defects in Object-Oriented Designs. In *Proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*. Universities of Glasgow and Strathclyde, Glasgow, UK, July 2005.

[MK00] Even Mamas and Kostas Kontogiannis. Towards Portable Source Code Representations Using XML. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 172–182, Washington, DC, USA, 2000. IEEE Computer Society.

[MLL05] Michael Martin, Benjamin Livshits, and Monica Lam. Finding Application Errors and Security Flaws Using PQL: a Program Query Language. *ACM SIGPLAN Notices*, 40(10):365–383, 2005.

[MM07] Subramanian Muralidhar and Jim Melton. XML Syntax for XQuery 1.0 (XQueryX). World Wide Web Consortium Recommendation, January 2007. `http://www.w3.org/TR/2007/REC-xqueryx-20070123`.

[MMFA04] Nabor Mendonça, Paulo Maia, Leonardo Fonseca, and Rossana Andrade. Building Flexible Refactoring Tools with XML. *Proceedings of Simpósio Brasileiro de Engenharia de Software (SBES 2004)*, pages 20–22, 2004. `http://nabor.mendonca.googlepages.com/SBES2004-paper.pdf`.

[MMN02] Gregory McArthur, John Mylopoulos, and Siu Kee Keith Ng. An Extensible Tool for Source Code Representation Using XML. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 199–208, Washington, DC, USA, 2002. IEEE Computer Society.

[MN95] Gail Murphy and David Notkin. Lightweight source model extraction. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 116–127, New York, NY, USA, 1995. ACM Press.

[MOF07] Object Management Group, MetaObject Facility (MOF). `http://www.omg.org/mof/`, 2007.

[MRS+07] Ashok Malhotra, Kristoffer Rose, Jérôme Siméon, Mary Fernández, Peter Fankhauser, Denise Draper, Philip Wadler, and Michael Rys. XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium Recommendation, January 2007. `http://www.w3.org/TR/2007/REC-xquery-semantics-20070123`.

[MW03] André Marburger and Bernhard Westfechtel. Behavioral Analysis of Telecommunication Systems by Graph Transformations. In John Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *AGTIVE*, volume 3062 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 2003.

[MWM07] Ashok Malhotra, Norman Walsh, and Jim Melton. XQuery 1.0 and XPath 2.0 Functions and Operators. World Wide Web Consortium Recommendation, January 2007. `http://www.w3.org/TR/2007/REC-xpath-functions-20070123`.

[MY04] Katsuhisa Maruyama and Shinichiro Yamamoto. A CASE Tool Platform Using an XML Representation of Java Source Code. In *SCAM '04: Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[NB07] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007), June 26-29 2007, Tallinn, Estonia. Lecture Notes in Computer Science 4581*, pages 228–243. Springer, Heidelberg, Germany, June 2007.

[Net07] NetBeans IDE website. `http://www.netbeans.org`, 2007.

[NWW01] Jörg Niere, Jörg Wadsack, and Lothar Wendehals. Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic. Technical report, Technical Report tr-ri-01-222, University of Paderborn, Germany, 2001.

[NZG08] Helmut Neukirchen, Benjamin Zeiss, and Jens Grabowski. An Approach to Quality Engineering of TTCN-3 Test Specifications. *International Journal on Software Tools for Technology Transfer (STTT)*, 2008. to appear.

[Pan03] Ravindranath Pandian. *Software Metrics: A Guide to Planning, Analysis, and Application*. Auerbach, CRC Press LLC, Boca Raton, USA, 2003.

[Pen02] Thomas Pender. *UML Weekend Crash Course*. Wiley Publishing, Indianapolis, USA, 2002.

[Per07] Perl website. `http://perl.com`, 2007.

[PMD07] PMD website. `http://pmd.sourceforge.net`, 2007.

[PP94a] Santanu Paul and Atul Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[PP94b] Santanu Paul and Atul Prakash. Supporting Queries on Source Code: A Formal Framework. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):325–348, 1994.

[PP96] Santanu Paul and Ataul Prakash. A Query Algebra for Program Databases. *IEEE Transactions on Software Engineering*, 22(3):202–217, 1996.

[PQL07] PQL: Program Query Language website. `http://pql.sourceforge.net`, 2007.

[Pre05] Roger Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, New York, USA, 6th edition, 2005.

[QP06] Terry Quatrani and Jim Palistrant. *Visual Modeling with IBM Rational Software Architect and UML*. Pearson Education, Boston, MA, USA, 2006.

[RSF+07] Jonathan Robie, Jérôme Siméon, Mary Fernández, Don Chamberlin, Daniela Florescu, and Scott Boag. XQuery 1.0: An XML Query Language. World Wide Web Consortium Recommendation, January 2007. `http://www.w3.org/TR/2007/REC-xquery-20070123`.

[SAK07] Daniel Speicher, Malte Appeltauer, and Günter Kniesel. Code Analyses for Refactoring by Source Code Patterns and Logical Queries. In *Proceedings of the 1st Workshop on Refactoring Tools held in conjunction with 21st European Conference on Object-Oriented Programming (ECOOP 2007), Berlin. (Editors: Danny Dig, Michael Cebulla). Technical Report No 2007-8, Technical University Berlin, ISSN 1436-9915*, 2007.

[SAX04] Simple API for XML (SAX) website. `http://www.saxproject.org`, 2004.

[Sax07] Saxon – The XSLT and XQuery Processor website. `http://saxon.sourceforge.net`, 2007.

[SD04] Ina Schieferdecker and George Din. A Meta-Model for TTCN-3. In Manuel Núñez, Zakaria Maamar, Fernando Pelayo, Key Pousttchi, and Fernando Rubio, editors, *Applying Formal Methods: Testing, Performance, and M/E-Commerce: FORTE 2004 Workshops The FormEMC, EPEW, ITM*, volume 3236 of *Lecture Notes in Computer Science*, pages 366–379. Springer, 2004.

[She05] Saad Inaam Sheikh. Detecting Bad Code Smells for Refactoring by Using Historical Data of Source Control System. Master's thesis, National University of Computer and Emerging Sciences, Lahore, Pakistan, 2005.

[Sli05] Stefan Slinger. Code Smell Detection in Eclipse. Master's thesis, University of Technology, Delft, Netherlands, 2005.

[Som04] Ian Sommerville. *Software Engineering*. Pearson Education, Harlow, England, UK, 7th edition, 2004.

[SSL01] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. Metrics Based Refactoring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 30–38, Washington, DC, USA, 2001. IEEE Computer Society.

[ST03] Ed Seidewitz and InteliData Technologies. What Models Mean. *IEEE Software*, 20(5):26–32, 2003.

[StA04] Streaming API for XML (StAX) 1.0 Final Release. `http://jcp.org/en/jsr/detail?id=173`, 2004.

[SWR07] Stephan Schulz, Anthony Wiles, and Steve Randall. TPLan-A Notation for Expressing Test Purposes. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2007.

[Teo98] Toby Teorey. *Database Modeling and Design*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 3rd edition, 1998.

[TM03] Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 91–100, Washington, DC, USA, 2003. IEEE Computer Society.

[TQL03] TQL website. `http://www.di.unipi.it/~ghelli/tql/`, 2003.

[TRe07] TRex – The TTCN-3 Refactoring and Metrics Tool website. `http://www.trex.informatik.uni-goettingen.de`, 2007.

[TSDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A Meta-Model for Language-Independent Refactoring. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 157–169. IEEE Computer Society Press, 2000.

[TyR06] TyRuBa website. `http://tyruba.sourceforge.net`, 2006.

[UML07] Object Management Group, Unified Modeling Language (UML), version 2.1.1. `http://www.omg.org/technology/documents/formal/uml.htm`, 2007.

[vEM02] Eva van Emden and Leon Moonen. Java Quality Assurance by Detecting Code Smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering*, page 97, Washington, DC, USA, 2002. IEEE Computer Society.

[vMV95] Anneliese von Mayrhauser and Marie Vans. Program Comprehension During Software Maintenance and Evolution. *IEEE Computer*, 28(8):44–55, 1995.

[Vol06] Kris De Volder. JQuery: A Generic Code Browser with a Declarative Configuration Language. volume 3819 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2006.

[Wal07] Priscilla Walmsley. *XQuery*. O'Reilly Media, Sebastopol, CA, USA, 2007.

[WDT+05] Colin Willcock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3*. John Wiley & Sons, Ltd, Chichester, UK, 2005.

[Wei05] Melanie Weis. Fuzzy Duplicate Detection on XML Data. 2005. Humboldt University, Berlin, Germany, `http://www.hpi.uni-potsdam.de/fileadmin/hpi/FG_Naumann/publications/VLDB05Phd_xmloid.pdf`.

[WF04] Priscilla Walmsley and David Fallside. XML Schema Part 0: Primer Second Edition. World Wide Web Consortium Recommendation, October 2004. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028`.

[WMN+07] Norman Walsh, Ashok Malhotra, Marton Nagy, Jonathan Marsh, and Mary Fernández. XQuery 1.0 and XPath 2.0 Data Model (XDM). World Wide Web Consortium Recommendation, January 2007. `http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123`.

[WP05] Bartosz Walter and Blazej Pietrzak. Multi-criteria Detection of Bad Smells in Code with UTA Method. In Hubert Baumeister, Michele Marchesi, and Mike Holcombe, editors, *XP*, volume 3556 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2005.

[XDu05] XDuce website. `http://xduce.sourceforge.net`, 2005.

[XMI05] Object Management Group, MOF 2.0/XMI Mapping Specification, v2.1. `http://www.omg.org/technology/documents/formal/xmi.htm`, 2005.

[XQ206] XQ2XML website. http://monet.nag.co.uk/xq2xml/, 2006.

[XQJ07] XQuery API for Java (XQJ) 0.9 Public Review Draft. http://jcp.org/en/jsr/detail?id=225, 2007.

[XUp00] XUpdate Working Draft 2000-09-14. http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html, 2000.

[Zei06] Benjamin Zeiss. A Refactoring Tool for TTCN-3. Master's thesis, Institute for Informatics, ZFI-BM-2006-05, ISSN 1612-6793, Center for Informatics, University of Göttingen, March 2006.

[ZK01] Ying Zou and Kostas Kontogiannis. Towards a Portable XML-based Source Code Representation. In *Proceedings of the International Conference on Software Engineering (ICSE) 2001 Workshops of XML Technologies and Software Engineering (XSE)*, 2001. http://post.queensu.ca/~zouy/files/xse-2001.pdf.

[ZNG+06] Benjamin Zeiss, Helmut Neukirchen, Jens Grabowski, Dominic Evans, and Paul Baker. Refactoring and Metrics for TTCN-3 Test Suites. In *System Analysis and Modeling: Language Profiles. 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31–June 2, 2006, Revised Selected Papers, Lecture Notes in Computer Science 4320*, pages 148–165. Springer, Heidelberg, Germany, December 2006.

All URIs mentioned in this thesis have been verified on the 2007-10-24. Copies of the pages can be requested by sending an email to jens@noedler.de.

Appendix A

XQuery Source Codes

A.1 XQuery Facade Functions

```
(:::::::::::::::::::::::: generic facade functions ::::::::::::::::::::::::::)

declare function xfacade:get-root()
{
  $xfacade:root
};

declare function xfacade:get-files($n as node()*)
{
  xfacade:get-node($n)//TTCN3File
};

declare function xfacade:get-modules($n as node()*)
{
  xfacade:get-node($n)//TTCN3Module
};

declare function xfacade:get-blocks($n as node()*)
{
  xfacade:get-node($n)//StatementBlock
};

declare function xfacade:get-parent-block($n as node()*)
{
  ($n/ancestor::StatementBlock)[last()]
};

declare function xfacade:get-functions($n as node()*)
{
  xfacade:get-node($n)//FunctionDef
};

declare function xfacade:get-unions($n)
{
  xfacade:get-node($n)//UnionDef
};

declare function xfacade:get-typedefs($n)
{
  xfacade:get-node($n)//TypeDef
};
```

```
42
43  declare function xfacade:get-constdefs($n)
44  {
45    xfacade:get-node($n)//ConstDef
46  };
47
48  declare function xfacade:is-const-value($n)
49  {
50    $n/ancestor::ConstDef
51  };
52
53  declare function xfacade:get-parameters($n as node()*)
54  {
55    xfacade:get-node($n)//FormalValuePar
56  };
57
58  declare function xfacade:get-statements($n as node()*)
59  {
60    xfacade:get-node($n)//FunctionStatementOrDef
61  };
62
63  declare function xfacade:get-if-constructs($n as node()*)
64  {
65    xfacade:get-node($n)//ConditionalConstruct
66  };
67
68  declare function xfacade:get-if-branch($n as element(ConditionalConstruct)*)
69  {
70    $n/StatementBlock
71  };
72
73  declare function xfacade:get-else-branch($n as element(ConditionalConstruct)*)
74  {
75    $n/ElseClause/StatementBlock
76  };
77
78  declare function xfacade:get-elseif-branch($n as element(ConditionalConstruct)*)
79  {
80    $n/ElseIfClause/StatementBlock
81  };
82
83  declare function xfacade:get-gotos($n as node()*)
84  {
85    xfacade:get-node($n)//GotoStatement
86  };
87
88  (: gets the next identifier that is located on the descendant axis. hence, the
89     search for the identifier in the tree is towards the leaves. that's up! :)
90  declare function xfacade:get-identifier-up($n as node()*)
91  {
92    (xfacade:get-node($n)//Identifier)[1]
93  };
94
95  (: gets the next identifier that is located on the preceding axis. hence, the
96     search for the identifier in the tree is towards the root. that's down! :)
97  declare function xfacade:get-identifier-down($n as node()*)
98  {
99    (xfacade:get-node($n)/preceding::Identifier)[last()]
100 };
101
102
103
104
```

```
105 declare function xfacade:get-identifiers($n as node()*)
106 {
107   xfacade:get-node($n)//Identifier
108 };
109
110 declare function xfacade:get-repeat-ops($n as node()*)
111 {
112   xfacade:get-node($n)//RepeatStatement
113 };
114
115 declare function xfacade:get-log-ops($n as node()*)
116 {
117   xfacade:get-node($n)//LogStatement
118 };
119
120 declare function xfacade:get-integer-values($n as node()*)
121 {
122   xfacade:get-node($n)//IntegerValue
123 };
124
125 declare function xfacade:get-float-values($n as node()*)
126 {
127   xfacade:get-node($n)//FloatValue
128 };
129
130 declare function xfacade:get-variable-references($n as node()*)
131 {
132   xfacade:get-node($n)//VariableRef
133 };
134
135 declare function xfacade:get-filename($n as node()*)
136 {
137   (: try to get the attribute form the TTCN3File element :)
138   let $filename := $n/ancestor::TTCN3File/@filename
139   return
140     if (exists($filename)) then
141       $filename
142     else
143       (: if the attr was not found, search all elements for the attribute.
144          this is not done in the first place for reasons of speed. :)
145       ($n//@filename)[1]
146 };
147
148 declare function xfacade:get-line($n as node()*)
149 {
150   $n/@line
151 };
152
153 declare function xfacade:get-lines($n as node()*)
154 {
155   $n//@line
156 };
157
158 declare function xfacade:get-line-end($n as node()*)
159 {
160   $n/@line-end
161 };
162
163 declare function xfacade:get-line-ends($n as node()*)
164 {
165   $n//@line-end
166 };
167
```

```
168 declare function xfacade:get-offset($n as node()*)
169 {
170   $n/@offset
171 };
172
173 declare function xfacade:get-offset-end($n as node()*)
174 {
175   $n/@offset-end
176 };
177
178
179 (:::::::::::::::::::: TTCN-3-specific facade functions ::::::::::::::::::::::)
180
181 declare function xfacade:get-testcases($n)
182 {
183   xfacade:get-node($n)//TestcaseDef
184 };
185
186 declare function xfacade:get-external-functions($n)
187 {
188   xfacade:get-node($n)//ExtFunctionDef
189 };
190
191 declare function xfacade:get-components($n)
192 {
193   xfacade:get-node($n)//ComponentDef
194 };
195
196 declare function xfacade:get-signatures($n)
197 {
198   xfacade:get-node($n)//SignatureDef
199 };
200
201 declare function xfacade:get-altsteps($n)
202 {
203   xfacade:get-node($n)//AltstepDef
204 };
205
206 declare function xfacade:get-else-statements($n as node()*)
207 {
208   xfacade:get-node($n)//ElseStatement
209 };
210
211 declare function xfacade:get-defaults($n as node()*)
212 {
213   xfacade:get-node($n)[*//PredefinedType[. = "default"]]
214 };
215
216 declare function xfacade:get-activate-ops($n as node()*)
217 {
218   xfacade:get-node($n)//ActivateOp
219 };
220
221 declare function xfacade:get-deactivate-ops($n as node()*)
222 {
223   xfacade:get-node($n)//DeactivateStatement
224 };
225
226 declare function xfacade:get-activate-by-name($n, $name)
227 {
228   xfacade:get-identifier-down(xfacade:get-activate-ops($n))[. = $name]
229 };
230
```

```
231 declare function xfacade:get-deactivate-by-name($n, $name)
232 {
233   xfacade:get-node($n)//DeactivateStatement[VariableRef/Identifier = $name]
234 };
235
236 declare function xfacade:get-alt-constructs($n as node()*)
237 {
238   xfacade:get-node($n)//AltConstruct
239 };
240
241 declare function xfacade:get-alt-branches($n as node()*)
242 {
243   xfacade:get-node($n)//GuardStatement
244 };
245
246 declare function xfacade:get-templates($n as node()*)
247 {
248   xfacade:get-node($n)//TemplateDef
249 };
250
251 declare function xfacade:is-template-field-value($n)
252 {
253   $n/ancestor::TemplateDef
254 };
255
256 declare function xfacade:get-template-inner-bodies($n as node()*)
257 {
258   xfacade:get-node($n)/TemplateBody//TemplateBody
259 };
260
261 declare function xfacade:get-template-body($n as node()*)
262 {
263   (xfacade:get-node($n)//TemplateBody)[1]
264 };
265
266 declare function xfacade:get-template-fields($n as node()*)
267 {
268   xfacade:get-node($n)//FieldSpec
269 };
270
271 declare function xfacade:get-template-identifier($n as element(TemplateDef))
272 {
273   $n/BaseTemplate/Identifier
274 };
275
276 declare function xfacade:get-identifieres-of-deactivate-ops-same-level(
277   $n as node()*)
278 {
279   $n/following-sibling::FunctionStatementOrDef/
280   FunctionStatement/BehaviourStatements/DeactivateStatement//Identifier
281 };
282
283 declare function xfacade:get-setverdicts($n as node()*)
284 {
285   xfacade:get-node($n)//SetLocalVerdict
286 };
287
288 declare function xfacade:get-verdict-type($n as node()*)
289 {
290   xfacade:get-node($n)//VerdictTypeValue
291 };
292
293
```

```
294 declare function xfacade:verdict-is-fail($n as element(VerdictTypeValue)*)
295 {
296   $n = "Fail"
297 };
298
299 declare function xfacade:verdict-is-inconc($n as element(VerdictTypeValue)*)
300 {
301   $n = "Inconc"
302 };
303
304 declare function xfacade:get-runs-on($n as node()*)
305 {
306   xfacade:get-node($n)//RunsOnSpec
307 };
308
309 declare function xfacade:get-timers($n as node()*)
310 {
311   xfacade:get-node($n)//TimerInstance
312 };
313
314 (: returns complete variable declarations like "var integer i, j;" :)
315 declare function xfacade:get-vars($n as node()*)
316 {
317   xfacade:get-node($n)//VarInstance
318 };
319
320 (: returns single variables like "i" of a declarations "var integer i, j;" :)
321 declare function xfacade:get-single-vars($n as node()*)
322 {
323   xfacade:get-node($n)//SingleVarInstance
324 };
325
326 declare function xfacade:get-type($n as node()*)
327 {
328   xfacade:get-node($n)//Type
329 };
330
331 declare function xfacade:get-create-ops($n as node()*)
332 {
333   xfacade:get-node($n)//CreateOp
334 };
335
336 declare function xfacade:get-start-ops($n as node()*)
337 {
338   xfacade:get-node($n)//ConfigurationStatements[. = "StartTCStatement"]
339 };
340
341 declare function xfacade:get-stop-ops($n as node()*)
342 {
343   xfacade:get-node($n)//ConfigurationStatements[. = "StopTCStatement"]
344 };
345
346 declare function xfacade:is-start-op($n)
347 {
348   $n/ancestor::StartTCStatement
349 };
```

A.2 Generic Smell Detection Functions

```
declare function xsmell:duplicate-code-in-conditionals(
  $compare-mode as xs:integer) as node()*
{
  for $scope in xfacade:get-blocks(())
  let $to-compare := (
    let $if := xfacade:get-if-constructs($scope)
    return (
      xfacade:get-if-branch($if) |
      xfacade:get-elseif-branch($if) |
      xfacade:get-else-branch($if)
    )
  )

  let $duplicates := xlib:find-duplicates($to-compare, $compare-mode)
  let $duplicates := xfacade:add-filename-attributes($duplicates)
  return xlib:group-duplicates($duplicates, $compare-mode)
};


declare function xsmell:nested-conditional() as node()*
{
  for $if-construct in xfacade:get-if-constructs(())
  let $if-branch   := xfacade:get-if-branch($if-construct)
  let $else-branch := xfacade:get-else-branch($if-construct)
  (: count of the statements contained in the if/else branches :)
  let $if-count    := count(xfacade:get-statements($if-branch))
  let $else-count  := count(xfacade:get-statements($else-branch))
  (: only proceed if the if construct contains an else branch :)
  where $else-branch
  return
    (: skip if constructs where both branches contain only one statement :)
    if ($if-count = 1 and $else-count = 1) then
      ()
    (: return those branches containing one statement :)
    else if ($if-count = 1) then
      $if-branch
    else if ($else-count = 1) then
      $else-branch
    else
      ()
};


declare function xsmell:goto() as node()*
{
  xfacade:get-gotos(())
};


declare function xsmell:long-statement-block($floor as xs:integer) as node()*
{
  (: check not only statement blocks of functions, test cases or altsteps
     as the smell catalog demands, but any statement block :)
  for $block in xfacade:get-blocks(())
  let $block-length := xfacade:get-line-end($block) - xfacade:get-line($block)
  where $block-length + 1 >= $floor
  return $block
};

```

```
declare function xsmell:magic-number($ignore as item()*) as node()*
{
  for $value in (xfacade:get-integer-values(()), xfacade:get-float-values(()))
  (: skip constants and template field values :)
  where not(xfacade:is-const-value($value))
    and not(xfacade:is-template-field-value($value))
    (: skip values found in the ignore list :)
    and not(functx:is-value-in-sequence($value, $ignore))
  return $value
};


declare function xsmell:long-parameter-list($floor as xs:integer) as node()*
{
  (: not only functions can be parameterized but also many other constructs :)
  for $parametrizable-construct in
    (
      xfacade:get-functions(()),
      xfacade:get-templates(()),
      xfacade:get-testcases(()),
      xfacade:get-typedefs(()),
      xfacade:get-external-functions(()),
      xfacade:get-signatures(()),
      xfacade:get-altsteps(())
    )
  let $parameters := xfacade:get-parameters($parametrizable-construct)
  let $number-of-parameters := count($parameters)
  where $number-of-parameters >= $floor
  return ($parametrizable-construct, xlib:return-value($number-of-parameters))
};
```

A.3 Smell Detection Functions for TTCN-3

```
declare function xsmell:duplicate-alt-branches($compare-mode as xs:integer)
  as node()*
{
  for $scope in (xfacade:get-altsteps(()), xfacade:get-alt-constructs(()))
  let $to-compare := xfacade:get-alt-branches($scope)
  let $duplicates := xlib:find-duplicates($to-compare, $compare-mode)
  let $duplicates := xfacade:add-filename-attributes($duplicates)

  return xlib:group-duplicates($duplicates, $compare-mode)
};


declare function xsmell:unreachable-default() as node()*
{
  for $block in xfacade:get-blocks(())[xfacade:get-defaults(.)]
  return
    let $alt  := xfacade:get-alt-constructs($block)
    let $else := (xfacade:get-else-statements($alt))[1]
    for $activate   in xfacade:get-activate-ops($block)
    for $deactivate in xfacade:get-deactivate-ops($block)
    (: else statement must be surrounded by activate and deactivate :)
    where $activate << $else and $else << $deactivate
    (: identifieres of the activate and deactivate must be the same :)
      and xfacade:get-identifier-down($activate) eq
        xfacade:get-identifier-up($deactivate)
    return ($activate, xlib:caused-by($else))
};
```

```
28 declare function xsmell:fully-parameterized-template() as node()*
29 {
30   for $template in xfacade:get-templates(())
31   let $defined := data(xfacade:get-identifiers(
32     xfacade:get-parameters($template)))
33   let $used := data(xfacade:get-identifiers(
34     xfacade:get-template-inner-bodies($template)))
35   where count($defined) > 0
36     and count($defined) = count(xfacade:get-template-fields($template))
37     and deep-equal($defined, $used)
38   return $template
39 };
40
41
42 declare function xsmell:missing-deactivation() as node()*
43 {
44   for $block in xfacade:get-blocks(())
45   let $activate := xfacade:get-activate-ops($block)
46   where not(xfacade:get-deactivate-ops($block))
47   return $activate
48 };
49
50
51 declare function xsmell:missing-activation() as node()*
52 {
53   for $block in xfacade:get-blocks(())
54   let $deactivate := xfacade:get-deactivate-ops($block)
55   where not(xfacade:get-activate-ops($block))
56   return $deactivate
57 };
58
59
60 declare function xsmell:deactivation-other-level() as node()*
61 {
62   for $block in xfacade:get-blocks(())
63   for $statement in xfacade:get-statements($block)
64   for $default in xfacade:get-defaults($statement)
65   where $default
66   return
67     let $default-name := xfacade:get-identifier-up($default)
68     let $activate := xfacade:get-activate-by-name($block, $default-name)
69     let $deactivate := xfacade:get-deactivate-by-name($block, $default-name)
70     let $deactivate-names :=
71       xfacade:get-identifieres-of-deactivate-ops-same-level($default)
72     (: the default must be:
73        - activated and deactivated
74        - deactivated at another level as its the declaration   :)
75     where $activate
76       and $deactivate
77       and not($deactivate-names = $default-name)
78     return $deactivate
79 };
80
81
82 declare function xsmell:missing-verdict($skip-testcases as xs:boolean)
83   as node()*
84 {
85   for $testcase in xfacade:get-testcases(())
86   return
87     (: return test cases without any setverdict call :)
88     if (not(xfacade:get-setverdicts($testcase))) then
89       if ($skip-testcases) then
90         ()
```

```
91        else
92          $testcase
93      else
94        (: return alt guards without any setverdict call :)
95        for $altguard in xfacade:get-alt-branches($testcase)
96        where not(xfacade:get-setverdicts($altguard))
97        (: skip alt guards which call repeat() as no verdict is required :)
98          and not(xfacade:get-repeat-ops($altguard))
99        return $altguard
100 };
101
102
103 declare function xsmell:missing-log() as node()*
104 {
105   for $verdict in xfacade:get-setverdicts(())
106   let $verdict-type := xfacade:get-verdict-type($verdict)
107   let $parent-block := xfacade:get-parent-block($verdict)
108   where not(xfacade:get-log-ops($parent-block))
109     and (xfacade:verdict-is-fail($verdict-type)
110      or xfacade:verdict-is-inconc($verdict-type))
111   return $verdict
112 };
113
114
115 declare function xsmell:stop-in-function($skip-runs-on as xs:boolean)
116   as node()*
117 {
118   for $func in xfacade:get-functions(())
119   let $stop := xfacade:get-stop-ops($func)
120   where $stop
121   return
122     if ($skip-runs-on and xfacade:get-runs-on($func)) then
123       ()
124     else
125       $stop
126 };
127
128
129 declare function xsmell:short-template($floor as xs:integer)
130   as node()*
131 {
132   for $template in xfacade:get-templates(())
133   let $template-body := xfacade:get-template-body($template)
134   let $number-of-chars :=
135     xfacade:get-offset-end($template-body) - xfacade:get-offset($template-body)
136   where $number-of-chars <= $floor
137   return $template
138 };
139
140
141 declare function xsmell:short-template2($floor as xs:integer)
142   as node()*
143 {
144   for $template in xfacade:get-templates(())
145   where count(xfacade:get-template-fields($template)) <= $floor
146   return $template
147 };
148
149
150 declare function xsmell:idle-ptc() as node()*
151 {
152   (: check for each variable declaration if it's of the type component :)
153   for $var  in xfacade:get-vars(())
```

```
154   for $comp in xfacade:get-components(())
155   where xfacade:get-identifier-up($comp) =
156         xfacade:get-identifier-up(xfacade:get-type($var))
157   return
158     (: get component's variable name, surrounding block, and references :)
159     let $var-name := xfacade:get-identifier-up(xfacade:get-single-vars($var))
160     let $block    := xfacade:get-parent-block($var)
161     let $references :=
162       xfacade:get-variable-references($block)
163       [xfacade:get-identifier-up(.) = $var-name]
164     return
165       (: no references of this component found? that smells! :)
166       if (empty($references)) then
167         $var
168       (: report an idle ptc, if the create operation is called
169          but the start operation is never called  :)
170       else if (
171         xfacade:get-create-ops($block)
172         [xfacade:get-identifier-down(.) = $var-name]
173       and
174         (every $r in $references satisfies not(xfacade:is-start-op($r))))
175       then
176         $var
177       else
178         ()
179 };
180
181
182 declare function xsmell:singular-template-reference() as node()*
183 {
184   for $template in xfacade:get-templates(())
185   let $template-references := xfacade:get-template-references($template)
186   where $template-references = 1
187   return $template
188 };
189
190
191 declare function xsmell:singular-component-element-reference() as node()*
192 {
193   for $component in xfacade:get-components(())
194   where xfacade:get-component-references($component) > 1
195   (: check each timer, var, and const if it referenced only once :)
196   return (
197     for $timer in xfacade:get-timers($component)
198     where xfacade:get-timer-references($timer) = 1
199     return $timer,
200     for $var in xfacade:get-single-vars($component)
201     where xfacade:get-var-references($var) = 1
202     return $var,
203     for $const in xfacade:get-constdefs($component)
204     where xfacade:get-const-references($const) = 1
205     return $const
206   )
207 };
```

Appendix B

Contents of the CD-ROM

The CD-ROM which is attached to the print version of this thesis contains:

- a digital PDF version of this thesis and
- the source codes of TRex and the XQuery-based Analysis Framework (XAF) taken from the Subversion repository (branch `trunk`) of TRex which is located at `http://www.trex.informatik.uni-goettingen.de/svn/trex/`.