Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2006-44

**Masterarbeit**

im Studiengang "Angewandte Informatik"

# Pattern-based Smell Detection in TTCN-3 Test Suites

Martin Bisanz

am Institut für

Informatik

Gruppe Softwaretechnik für Verteilte Systeme

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

19. Dezember 2006

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

|  |  |
|---|---|
| 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 19. Dezember 2006

Master's thesis

# Pattern-based Smell Detection in TTCN-3 Test Suites

Martin Bisanz

December 19, 2006

Supervised by Dr. Helmut Neukirchen
Software Engineering for Distributed Systems Group
Institute for Informatics
Georg-August-University Göttingen

## Abstract

TTCN-3 is a specification and implementation language for software tests. Just like any ordinary software, software tests can suffer from many quality problems. Software Engineering provides means and techniques to counteract these problems and to increase the quality of software in many respects. Refactoring is such a technique which is used to enhance the internal structure of code and to increase its readability, maintainability and comprehensiveness. Code parts in need of refactoring are referred to as "code smells". Refactorings for TTCN-3 test suites have already been investigated. In this thesis the concept of code smells is applied to TTCN-3. A catalog of code smells for TTCN-3 is presented, and each smell is connected to a corresponding counter measure. In addition, a tool for automated code smell detection is introduced together with results of applying this tool on existing TTCN-3 test suites.

## Kurzbeschreibung

TTCN-3 ist eine Spezifikations- und Implementierungssprache für Softwaretests. Tests können genau wie herkömmliche Software unter zahlreichen Qualitätsproblemen leiden. Software Engineering stellt Mittel und Techniken zur Verfügung, die diesen Problemen entgegenwirken und durch die die Qualität in vielerlei Hinsicht verbessert werden kann. Refactoring ist eine solche Technik, die zur Verbesserung der internen Struktur des Codes und zur Steigerung von Lesbarkeit, Änderbarkeit und Verständlichkeit eingesetzt wird. Codeabschnitte, die ein Refactoring benötigen, werden auch als "Code Smells" bezeichnet. Refactorings für TTCN-3 Testsuiten wurden bereits untersucht. In dieser Arbeit wird das Konzept der Code Smells auf TTCN-3 übertragen. Ein Code-Smell-Katalog wird präsentiert und jeder Code Smell mit einer entsprechenden Gegenmaßnahme verknüpft. Außerdem wird ein Softwareprogramm zum automatisierten Auffinden von Code Smells vorgestellt und Ergebnisse der Anwendung des Softwareprogramms auf existierende Testsuiten gezeigt.

# Contents

# 1 Introduction

*Software testing* is a well established approach for the investigation of software systems. It plays an important role in quality assurance for software. Testing can be defined as the execution of a *System Under Test* (SUT) in order to reveal defects. This is usually achieved by comparing the actual behavior of the SUT to the expected behavior, which, for example, is extracted from a specification. Often specifications for a software system are not fixed, but evolve over time. This can necessitate changes in the tests as well.

Frequent changes together with growing size and complexity can make test suites suffer from similar quality problems that affect any other software. If changes are performed without thorough adaptation of the internal code structure, they can result in a decay of code quality and – as a worst case – in unreadable and unmaintainable code. Especially for large software systems, tests can involve huge investments. Hence, quality assurance for tests is as important as quality assurance for the software that is tested.

The *Testing and Test Control Notation Version 3* (TTCN-3) is a language for specification and implementation of distributed test systems. It is applicable for different types of black-box testing, and its wide acceptance has led to the development of large and often complex test suites used in standardization and industry. The core notation [19] defines a textual syntax similar to modern programming languages.

*Refactoring* [26] is an established means for improving the quality of existing code, especially in terms of readability, maintainability and reuseability. It denotes an improvement to the internal structure of the code without changing its behavior. Refactorings for TTCN-3 have been investigated [11, 59, 61].

The term *code smell* (or *bad smell in code*) was introduced by Martin Fowler and Kent Beck [26]. A code smell is considered a certain area in the code that indicates the need for a refactoring. It can help localizing possibly problematic parts in the code. While code smells for regular programming languages have been studied thoroughly, code smells in test code have not been researched well. In this thesis the concept of code smells is applied to TTCN-3.

Chapter 2 gives a brief introduction to groundwork needed for further understanding: TTCN-3 as a language for test description and implementation, the *Eclipse Platform* and the *Static Analysis Framework* from the *Test & Performance Tools Platform* (TPTP) as a foundation for the implementation. Chapter 3 explains the notion of "code smell" and delimits it from similar concepts. It discusses the advantages of automated code smell detection and points out approaches. Similar works and tools are mentioned as well. In chapter 4, a catalog of 39 code smells for TTCN-3 is presented. The smells are organized in categories and arranged according to a fixed format. An implementation for automated code smell detection is introduced in chapter 5. It is realized as plug-in for the *TTCN-3 Refactoring and Metrics Tool* (TRex) and builds on the TPTP Static Analysis Framework. Some rules for code smell detection are presented in detail. As a sample for the applicability of the tool, code smells found in existing test suites are presented in chapter 6. Furthermore origins of the smells are discussed. Finally, chapter 7 gives an overall conclusion and an outlook on future prospects.

# 2 Foundations

This chapter gives a brief introduction to foundations of this thesis. First, an overview of TTCN-3 as test specification and implementation language is given. Afterwards, the Eclipse platform and the *Test & Performance Tools Platform* (TPTP) are introduced. Finally, a glance is thrown at the *TTCN-3 Refactoring and Metrics Tool* (TRex) as base for the automated smell detection.

## 2.1 TTCN-3

This section focuses on giving an overview and introducing those parts of TTCN-3 which are important for the understanding of subsequent chapters. More detailed information about TTCN-3 can be found in [19, 30, 47, 58].

### 2.1.1 Overview

The *Testing and Test Control Notation Version 3* (TTCN-3) [19] is a language for test specification and implementation standardized by the *European Telecommunications Standard Institute* (ETSI). It has its roots in functional black box testing for telecommunication systems like GSM or DECT, but it is applicable for other domains as well, e.g. internet protocol implementations or CORBA-based systems. TTCN-3 has its focus on system test, but can also be used on lower levels, i.e. for integration or even unit tests.

Its predecessor, the *Tree and Tabular Combined Notation* (TTCN) [37], was written using a tabular notation. TTCN-3 introduces a textual representation, the TTCN-3 core notation [19]. The syntax is similar to other programming languages. Other than the core language there exist various presentation formats like a tabular and a graphical notation [20, 21]. However, for the scope of this thesis only the core notation is taken into account.

Usually TTCN-3 tests are black box tests based on a specification without any knowledge of the underlying implementation. To design such a test, valid and invalid input and the expected output are extracted from the specification. At run-time, the actual output is compared to the expected output.

A TTCN-3 test system is connected to a *System Under Test* (SUT) via a *Test System Interface* (TSI). The test itself consists of test components: a *Main Test Component* (MTC) and any number of *Parallel Test Components* (PTCs). Components and TSI are connected

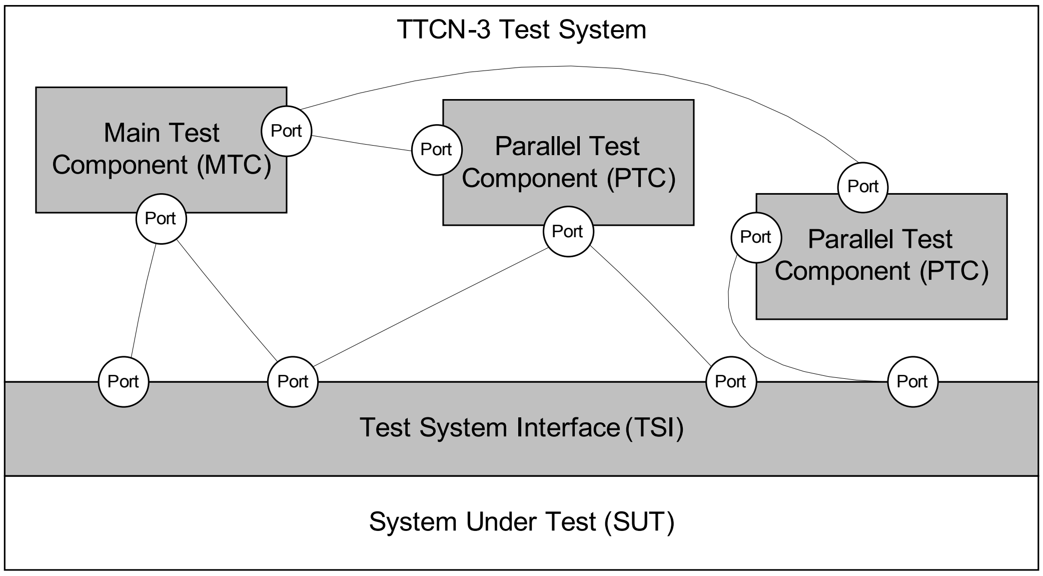

TTCN-3 Test System
Main Test Component (MTC)
Parallel Test Component (PTC)
Parallel Test Component (PTC)
Port
Test System Interface (TSI)
System Under Test (SUT)

*Figure 2.1: A test configuration*

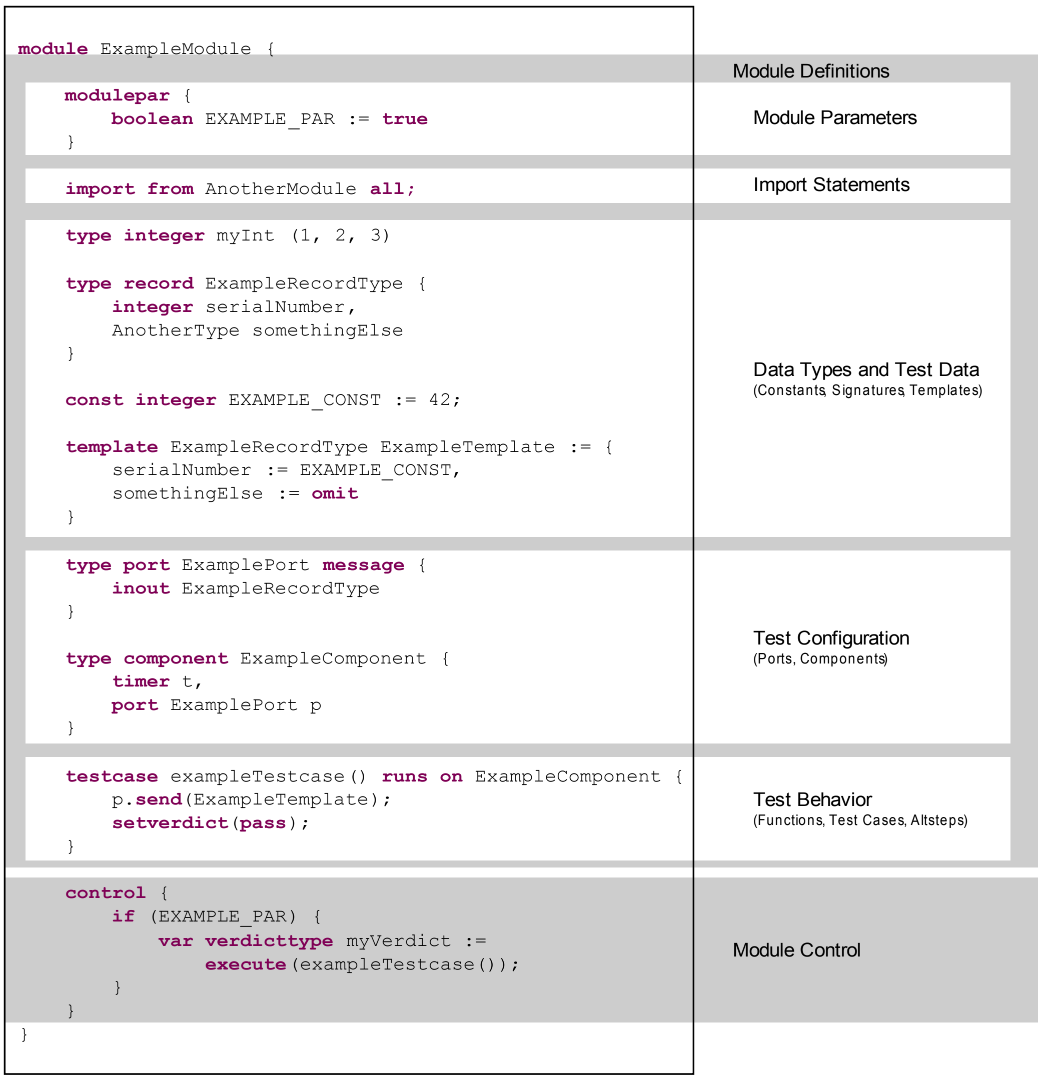
to each other via ports. Ports are modeled as infinite FIFO queue and have a direction (**in**, **out** or **inout**). A test case runs on an MTC and is able to create PTCs as needed. Figure 2.1 shows a test configuration with an MTC and two PTCs.

The outcome of the test case run is captured by a *test verdict*. Each component has its own local verdict. All local verdicts make up the global test case verdict. A verdict can have one of the following values, listed with increasing priority: **none**, **pass**, **inconc** (for inconclusive), **fail**, **error**. Both local verdicts and the global verdict are made up by the **setverdict** operation setting the highest priority. For example, if a test case utilizes three components and the local verdicts are **pass** for two components and **fail** for the third, the global verdict can only be **fail** or **error**.

### 2.1.2 Language elements

TTCN-3 source code is organized in *modules*. Modules are the building blocks of all TTCN-3 test specifications. They can be parameterized, and they can import definitions from other modules. The import mechanism allows to import single definitions, all definitions of a certain kind (e.g. all functions) and groups of definitions. All imports have to be declared explicitly; implicit imports are not allowed.

Figure 2.2 shows the structure of a module. It consists of an optional definitions part, followed by an optional control part. The control part is the entry point of a module which calls the test cases and controls their execution. The definitions part contains all



```
module ExampleModule {
    modulepar {
        boolean EXAMPLE_PAR := true
    }

    import from AnotherModule all;

    type integer myInt (1, 2, 3)

    type record ExampleRecordType {
        integer serialNumber,
        AnotherType somethingElse
    }

    const integer EXAMPLE_CONST := 42;

    template ExampleRecordType ExampleTemplate := {
        serialNumber := EXAMPLE_CONST,
        somethingElse := omit
    }

    type port ExamplePort message {
        inout ExampleRecordType
    }

    type component ExampleComponent {
        timer t,
        port ExamplePort p
    }

    testcase exampleTestcase() runs on ExampleComponent {
        p.send(ExampleTemplate);
        setverdict(pass);
    }

    control {
        if (EXAMPLE_PAR) {
            var verdicttype myVerdict :=
                execute(exampleTestcase());
        }
    }
}
```

*Figure 2.2: Structure of a TTCN-3 module*

definitions of the module; they are global to the entire module. A module may declare data types, constants, templates, ports, components, signatures, functions, test cases and altsteps. TTCN-3 does not support the declaration of global variables. Module definitions can be collected in named groups. A group of declarations is allowed wherever a single declaration can be specified.

### Types and Values

Data type definitions are based on predefined types. TTCN-3 contains a large number of basic types like **integer**, **float**, **boolean**, **char** and string types. These basic types can be sub-typed to restrict their values. Additionally, structured types like **record**, **set**, **union**, **enumerated** and arrays can be constructed from other types. Special types like **address**, **port** and **component** are associated with test configuration.

Test data is defined by *constants* and *templates*. Templates are a means to organize and re-use test data. They can either be used to transmit a set of values or to test whether a set of received values matches the template specification. In the simplest case a template is just a concrete instance of a type with a certain value. Furthermore, templates provide a matching mechanism and a simple form of inheritance, and templates can be parameterized. Beside being defined globally in the module scope they can be defined locally in behavioral entities or in-line in a communication operation within the test behavior.

*Ports* and *components* specify the structural elements of a test configuration. Ports are used for the communication among components and the TSI. They can be either message-based or procedure-based. Their direction can be specified as **in**, **out** or **inout**. Component type definitions define the ports that are associated with a component. In addition they can declare local variables, constants and timers. Components make up the elements for test distribution. They are defined separately from behavior.

### Behavior Specification

Test behavior is specified in *functions*, *test cases* and *altsteps*. These behavioral entities can be associated with components by a **runs on** clause. They all may declare local variables, constants, timers and templates. Functions can be "pure" functions, which do not have a communication interface and normally do data manipulation of some kind, or declare a **runs on** clause and contain communication operations such as **send** and **receive** statements. An altstep is a special kind of function and is used to structure alternative behavior. A test case is another special kind of function which must declare a component it is running on, and is executed in the control part of a module.

Usually the test configuration is set up in a test case using certain configuration operations. PTCs are created using the **create** operation. Ports of components can be connected to other components using the **connect** operation and to the TSI using the **map** operation.

Behavior on components can be started (**start**) and stopped (**stop**), and termination of components can be checked (**running**) and waited for (**done**).

The statement blocks in functions, test cases and altsteps support basic program statements like assignments, **if-else** constructs, **for**, **while** and **do-while** loops, labels and **goto** statements. An extraordinary construct in TTCN-3 is the **alt** statement for specifying alternative behavior.

The handling of alternative behavior is quite uncommon. Alternative behavior has to be specified whenever a behavioral entity waits for a response from the SUT or a timeout. It is typically defined by an **alt** construct with a number of alternatives which are guarded by expressions. Alternatives can be realized as altstep which can be called explicitly from the **alt** construct or activated as default beforehand. The alternatives are evaluated according to their order of appearance. Defaults are evaluated after all other alternatives have been tried. Furthermore, an else branch can be specified which is executed if no other branches is taken. Listing 2.1 shows a code snipped illustrating an **alt** statement with five branches of alternative behavior.

```
alt {
  [] P1.receive(MyMessage1) {
    // do something...
  }
  [x > 0] P2.receive(MyMessage2a) {}
  [x <= 0] P2.receive(MyMessage2b) {}
  [] myAltstep() // call of altstep
  [else] {      // else branch
    stop
  }
}
```

*Listing 2.1: Alternative behavior*

## 2.2 Eclipse Platform

The name *Eclipse* is often used as synonym for the Eclipse *Software Development Kit* (SDK), which includes a leading Java *Integrated Development Environment* (IDE), the *Java Development Tools* (JDT). Actually Eclipse is an open source community managed through the *Eclipse Foundation* [13] which oversees a number of projects centering around an open development platform called the *Eclipse Platform.* The JDT is only one among many projects building on the Eclipse Platform and extending it with further functionality. Just as well, TRex, which is introduced in section 2.4, builds on the Eclipse Platform.

The Eclipse Platform itself is a universal tool platform. It defines a set of frameworks and common services, including a standard user interface model (called *workbench*) and a portable native widget toolkit, a project model for managing resources (the *workspace*), automatic resource delta management for incremental compilers and builders, language-

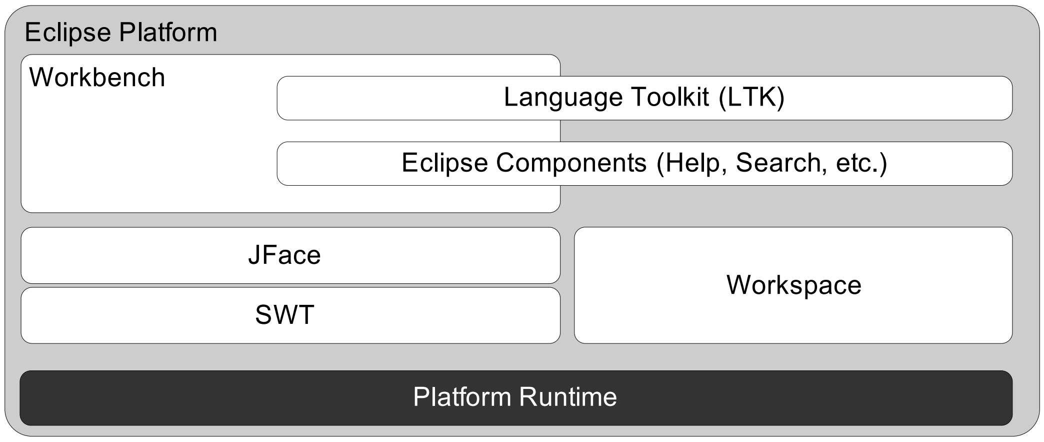


*Figure 2.3: The Eclipse Platform [61]*

independent debug infrastructure and infrastructure for distributed multi-user versioned resource management like the *Concurrent Versions System* (CVS) [10].

Except for a small kernel known as the *Platform Runtime*, almost the whole functionality is located in *plug-ins*. A plug-in is the smallest unit of Eclipse Platform functionality. It typically consists of Java code and other resources, although plug-ins may not even include code at all. Each plug-in has a manifest declaring its interconnections and dependencies on other plug-ins. It may declare any number of *extension points*, which can be extended by other plug-ins, and any number of *extensions* to one or more extension points in other plug-ins. An extension point may have a corresponding *Application Programming Interface* (API). Plug-ins providing extensions to this extension point contribute implementations for this interface. Plug-ins belonging together can be bundled as *feature*.

Figure 2.3 gives an architectural overview over the Eclipse Platform. One of its core functionalities is the workspace. It consists of one or more top-level projects, where each project maps to a folder in an abstract file system layer (e.g. the local file system). Projects contain files and folders which are created and manipulated by the user. Projects, files and folders are summarized as *resources*. Workspace resources can be annotated by a marker mechanism. Markers are used to record diverse annotations such as compiler errors, bookmarks, debugger breakpoints and search hits. Plug-ins can declare custom marker subtypes.

The Eclipse Platform *User Interface* (UI) is build on top of two toolkits: The *Standard Widget Toolkit* (SWT) provides a common OS-independent API for widgets and graphics which is tightly integrated with the underlying native window system; *JFace* builds on SWT and simplifies common UI programming tasks. The *workbench* is the actual UI of the Eclipse Platform and supplies the structure in which tools interact with the user.

Fundamental concepts of the workbench are editors, views and perspectives. Editors provide means for opening, editing and saving objects. The Platform includes a standard text editor for text resources; plug-ins can supply more specialized editors. Views show information about any object the user is working on in the workbench. For example, a view may support other views or editors by providing additional information about the currently selected object. The Platform includes several standard views (e.g. Package Explorer, Outline, Problems, Console, Tasks). Plug-ins may contribute custom views. Arrangements of views can be stored in perspectives. A workbench window can have several perspectives, but only one can be visible at a given time. Among perspectives provided by the Platform are those for general resource navigation, online help and team support tasks. Again, additional perspectives can be provided by plug-ins (e.g. the Java perspective provided by JDT plug-ins).

The Platform includes several other components like search functionality or online help. The *Language Toolkit* (LTK) [27] is another component included which provides a framework for refactorings. It is exceptionally important for TRex, the TTCN-3 tool introduced in section 2.4.

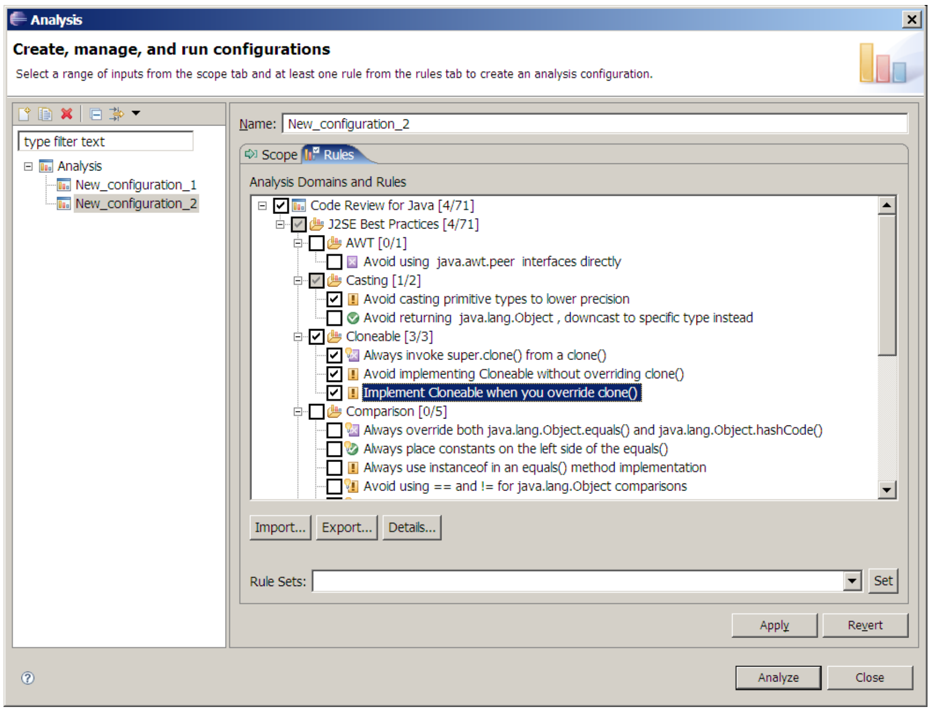
## 2.3 TPTP and Static Resource Analysis Framework

The Eclipse *Test & Performance Tools Platform* (TPTP) Project [15] is divided into four sub-projects: TPTP Platform, Monitoring Tools, Testing Tools and Tracing and Profiling Tools. It is a top level project of the Eclipse Foundation, just like the Eclipse Project which cares for the Eclipse Platform, Equinox, the JDT and the *Plug-In Development Environment* (PDE). TPTP provides an open platform supplying frameworks and services for building test and performance tools.

The TPTP Platform Project is a conglomeration of subsystems as common infrastructure for the other sub-projects. Among them is the Static Analysis Framework [31, 32, 33] for running analysis rules against resources. It basically consists of two plug-ins, a core and a UI plug-in. Analysis rules for Java are provided in separate plug-ins.

The sets of rules and resources for an analysis run are controlled by an *analysis configuration*. The Static Analysis Framework adds a “Run > Analysis” menu option to the workbench menu which behaves similar to the “Run > Run” and “Run > Debug” options shipped with the Eclipse SDK. In fact it uses the same launch mechanism. The menu option opens a dialog for managing and running analysis configurations. Figure 2.4 shows an analysis configuration for the Code Review for Java, which is provided as a reference implementation for the Static Analysis Framework.

An analysis configuration determines the range of resources on which the analysis is performed. The available options for this range are the entire workspace, a working set of resources or a set of projects. These options can be selected on a “Scope” tab of the



Analysis

Create, manage, and run configurations

Select a range of inputs from the scope tab and at least one rule from the rules tab to create an analysis configuration.

```
type filter text
⊟ Analysis
    New_configuration_1
    New_configuration_2
```

Name: New_configuration_2

Scope | Rules

Analysis Domains and Rules

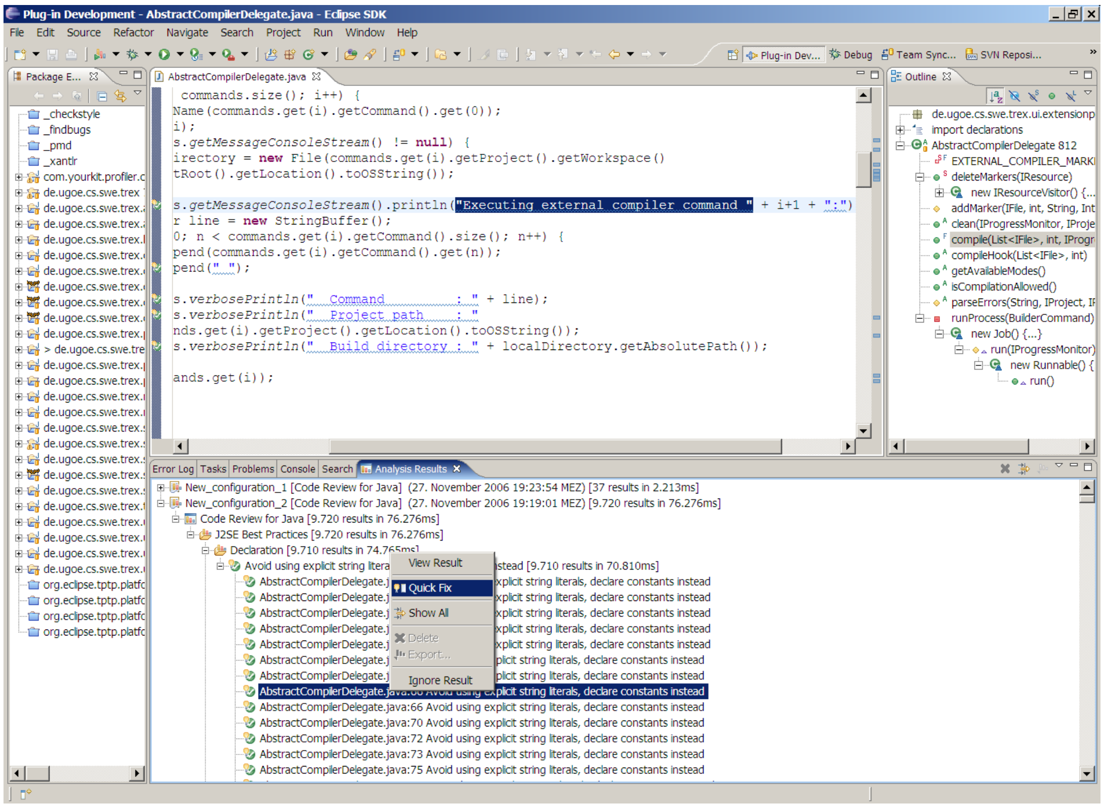
```
⊟ [✓] Code Review for Java [4/71]
  ⊟ [✓] J2SE Best Practices [4/71]
    ⊟ [ ] AWT [0/1]
        [ ] Avoid using java.awt.peer interfaces directly
    ⊟ [✓] Casting [1/2]
        [✓] Avoid casting primitive types to lower precision
        [ ] Avoid returning java.lang.Object , downcast to specific type instead
    ⊟ [✓] Cloneable [3/3]
        [✓] Always invoke super.clone() from a clone()
        [✓] Avoid implementing Cloneable without overriding clone()
        [✓] Implement Cloneable when you override clone()
    ⊟ [ ] Comparison [0/5]
        [ ] Always override both java.lang.Object.equals() and java.lang.Object.hashCode()
        [ ] Always place constants on the left side of the equals()
        [ ] Always use instanceof in an equals() method implementation
        [ ] Avoid using == and != for java.lang.Object comparisons
```

*Figure 2.4: Analysis run configuration*

dialog (which is not active in figure 2.4). Additionally, the configuration includes a set of elements for the analysis. The element set can be edited on the “Rules” tab of the dialog. It is presented as a tree allowing selection and deselection of elements.

The topmost nodes of the tree are *analysis providers*. They represent the type of analysis tools known to the Static Analysis Framework. The child nodes of providers are *categories*, which are used to organize analysis elements. Categories may contain any number of *rules* and other categories. Rules perform the actual work during the analysis process by checking conditions and generating results. A rule is classified by a severity level (*recommendation*, *warning* or *error*). Furthermore rules can be parameterized to make them customizable. If a rule is selected in the configuration dialog, both severity setting and rule parameters can be edited via the “Details” button. In addition to the rules supplied by an analysis provider it is also possible for end users to create rules from templates. A template is an abstract rule that has to be concretized before it can be used.

A run of an analysis configuration creates a new entry in the Analysis Results view, called *analysis history*. A history is a list of all providers, categories and rules that were launched together with a list of results produced. Figure 2.5 shows the workbench with an active Analysis Results view containing two history elements. The Results view supports filters


Plug-in Development - AbstractCompilerDelegate.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
AbstractCompilerDelegate.java
Error Log Tasks Problems Console Search Analysis Results
View Result
Quick Fix
Show All
Delete
Export...
Ignore Result

```
 commands.size(); i++) {
Name(commands.get(i).getCommand().get(0));
i);
s.getMessageConsoleStream() != null) {
irectory = new File(commands.get(i).getProject().getWorkspace()
tRoot().getLocation().toOSString());

s.getMessageConsoleStream().println("Executing external compiler command " + i+1 + ":")
r line = new StringBuffer();
0; n < commands.get(i).getCommand().size(); n++) {
pend(commands.get(i).getCommand().get(n));
pend(" ");

s.verbosePrintln("  Command          : " + line);
s.verbosePrintln("  Project path     : "
nds.get(i).getProject().getLocation().toOSString());
s.verbosePrintln("  Build directory : " + localDirectory.getAbsolutePath());

ands.get(i));
```

```
New_configuration_1 [Code Review for Java] (27. November 2006 19:23:54 MEZ) [37 results in 2.213ms]
New_configuration_2 [Code Review for Java] (27. November 2006 19:19:01 MEZ) [9.720 results in 76.276ms]
  Code Review for Java [9.720 results in 76.276ms]
    J2SE Best Practices [9.720 results in 76.276ms]
      Declaration [9.710 results in 74.765ms]
        Avoid using explicit string litera                  nstead [9.710 results in 70.810ms]
          AbstractCompilerDelegate.j                  xplicit string literals, declare constants instead
          AbstractCompilerDelegate.j                  xplicit string literals, declare constants instead
          AbstractCompilerDelegate.j                  xplicit string literals, declare constants instead
          AbstractCompilerDelegate.j                  xplicit string literals, declare constants instead
          AbstractCompilerDelegate.j                  xplicit string literals, declare constants instead
          AbstractCompilerDelegate.j                  plicit string literals, declare constants instead
          AbstractCompilerDelegate.j                  plicit string literals, declare constants instead
          AbstractCompilerDelegate.j                  plicit string literals, declare constants instead
          AbstractCompilerDelegate.java:66 Avoid using explicit string literals, declare constants instead
          AbstractCompilerDelegate.java:70 Avoid using explicit string literals, declare constants instead
          AbstractCompilerDelegate.java:72 Avoid using explicit string literals, declare constants instead
          AbstractCompilerDelegate.java:73 Avoid using explicit string literals, declare constants instead
          AbstractCompilerDelegate.java:75 Avoid using explicit string literals, declare constants instead
```

*Figure 2.5: Analysis Results view*

(e.g. to disable rules without results being displayed), and some rules supply a *quick-fix* feature which is a quick solution for the problem found. It can be used to modify the code part matched by the rule. The quick-fix feature is accessible via the context menu of a result.

## 2.4 TRex

The *TTCN-3 Refactoring and Metrics Tool* (TRex) [53] is an open-source tool for TTCN-3 written as plug-in for the Eclipse Platform. It provides IDE functionality for the TTCN-3 core notation and can be used for the assessment and restructuring of TTCN-3 test suites by means of metrics and refactoring. It is developed and maintained by the Software Engineering for Distributed Systems Group at the Institute for Informatics, University of Göttingen. Major parts of it originate from Benjamin Zeiss' master's thesis [61], which is also a good reference on TRex.

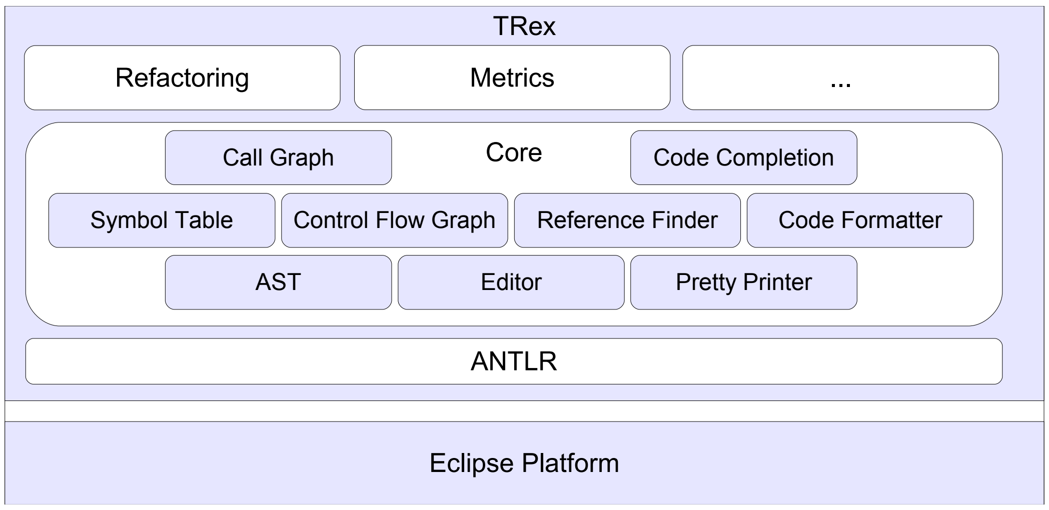


*Figure 2.6: TRex architecture*

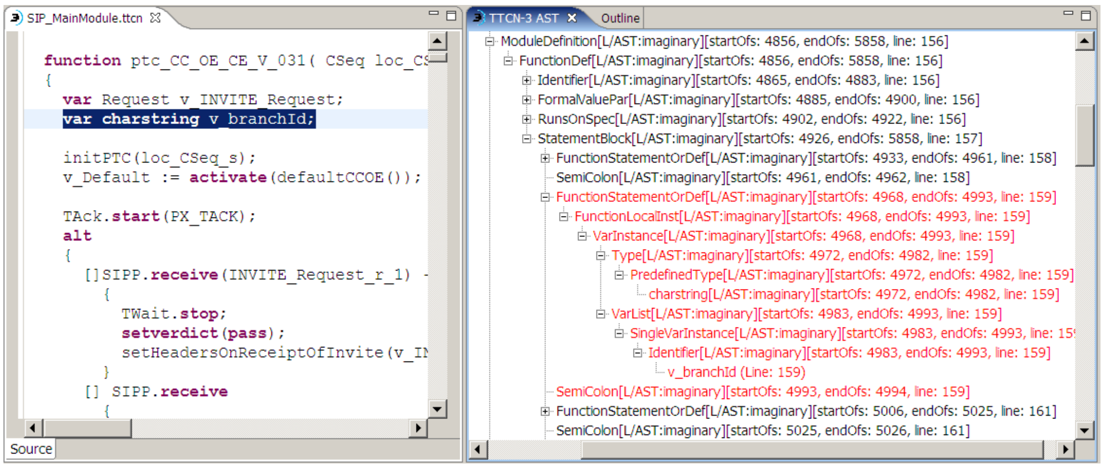
Figure 2.6 shows an architectural overview of TRex. The core plug-in provides the key functionalities of an IDE for TTCN-3. One of them is the editor, which is capable of syntax highlighting. The source code is parsed using a parser generated by *Another Tool for Language Recognition* (ANTLR), which is a tool that generates code for lexical and syntactical analysis. Grammatical descriptions for ANTLR are provided in a language similar to the *Extended Backus-Naur Form* (EBNF).

The result of lexing and parsing a TTCN-3 source file is an *Abstract Syntax Tree* (AST)[1], which is a tree representation of the source code. It holds most of the input symbols and reflects the relationship between those tokens in the structure of the tree.

Figure 2.7 shows an example of an AST as part of a screenshot of TRex. The editor on the left hand side shows TTCN-3 source code in which a variable declaration is selected. On the right hand side, the AST view shows the corresponding nodes in the AST (marked red). The variable declaration is commensurate with the *FunctionStatementOrDef* node, which contains a *FunctionLocalInst* that holds a *VarInstance*. The *VarInstance* consists of a *Type* and a *VarList* node. The *VarList* has only one *SingleVarInstance* and contains the *Identifier* of the variable. The *FunctionStatementOrDef* is followed by a *SemiColon*.

The symbol table is build on top of the AST. It contains information about symbols in the source code, like its type or scope. The code completion makes use of this information and suggests keywords and identifiers during the editing process. The reference finder locates all references to a symbol. The pretty printer is a component for reconstructing TTCN-3

[1]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.



```
function ptc_CC_OE_CE_V_031( CSeq loc_CS
{
  var Request v_INVITE_Request;
  var charstring v_branchId;

  initPTC(loc_CSeq_s);
  v_Default := activate(defaultCCOE());

  TAck.start(PX_TACK);
  alt
  {
    []SIPP.receive(INVITE_Request_r_1) -
      {
        TWait.stop;
        setverdict(pass);
        setHeadersOnReceiptOfInvite(v_IN
      }
    [] SIPP.receive
      {
```

```
ModuleDefinition[L/AST:imaginary][startOfs: 4856, endOfs: 5858, line: 156]
  FunctionDef[L/AST:imaginary][startOfs: 4856, endOfs: 5858, line: 156]
    Identifier[L/AST:imaginary][startOfs: 4865, endOfs: 4883, line: 156]
    FormalValuePar[L/AST:imaginary][startOfs: 4885, endOfs: 4900, line: 156]
    RunsOnSpec[L/AST:imaginary][startOfs: 4902, endOfs: 4922, line: 156]
    StatementBlock[L/AST:imaginary][startOfs: 4926, endOfs: 5858, line: 157]
      FunctionStatementOrDef[L/AST:imaginary][startOfs: 4933, endOfs: 4961, line: 158]
      SemiColon[L/AST:imaginary][startOfs: 4961, endOfs: 4962, line: 158]
      FunctionStatementOrDef[L/AST:imaginary][startOfs: 4968, endOfs: 4993, line: 159]
        FunctionLocalInst[L/AST:imaginary][startOfs: 4968, endOfs: 4993, line: 159]
          VarInstance[L/AST:imaginary][startOfs: 4968, endOfs: 4993, line: 159]
            Type[L/AST:imaginary][startOfs: 4972, endOfs: 4982, line: 159]
              PredefinedType[L/AST:imaginary][startOfs: 4972, endOfs: 4982, line: 159]
                charstring[L/AST:imaginary][startOfs: 4972, endOfs: 4982, line: 159]
            VarList[L/AST:imaginary][startOfs: 4983, endOfs: 4993, line: 159]
              SingleVarInstance[L/AST:imaginary][startOfs: 4983, endOfs: 4993, line: 15
                Identifier[L/AST:imaginary][startOfs: 4983, endOfs: 4993, line: 159]
                  v_branchId (Line: 159)
      SemiColon[L/AST:imaginary][startOfs: 4993, endOfs: 4994, line: 159]
      FunctionStatementOrDef[L/AST:imaginary][startOfs: 5006, endOfs: 5025, line: 161]
      SemiColon[L/AST:imaginary][startOfs: 5025, endOfs: 5026, line: 161]
```

Figure 2.7: Abstract Syntax Tree

source code from the AST. It is used by the code formatter, which reformats code according to customizable rules. Control and data flow analyses build graphs from the AST as basis for corresponding views as well as for other components.

Further functionality is build on top of this core functionality, like refactorings [61] and metrics [62]. TRex is the base for the implementation of automated code smell detection presented in chapter 5.

# 3 Quality Assurance for Software and Tests

Like ordinary software, test suites can suffer from many quality problems. Changes to test suites may be necessary in order to improve test coverage or to reflect changes in the specification and the SUT. Additionally, TTCN-3 test suites are often tool-generated, which can make their core notation representation hard to read, maintain and reuse.

If such a generated or overly complex test suite needs to be changed on core notation level, there is a risk of changes being carried out without changing the internal structure accordingly, especially if time is short. The result is a decay of the internal structure of the code. In the worst case this can lead to unmaintainable or unreliable code.

## 3.1 Refactoring

Software Engineering provides techniques to counteract the mentioned quality problems. *Refactoring* is such a technique. The concept of refactoring have been existing for several years, but the breakthrough came as key concept of agile development processes such as *Extreme Programming* (XP) [3] and with the book "Refactoring" by Martin Fowler [26].

Philipps et al. [48] describe the conceptual roots and ideas behind Refactoring. It aims at enhancing the internal structure of code while preserving its semantics. Refactorings often consist of small steps which are then combined to sequences like "copy a to b, then rename c, then delete a and check d." It is dependent on certain situations in which it can be applied.

Fowler defines refactoring 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" [26, p. 53]. He presents a catalog of 72 refactorings for Java. Well-known examples from this catalog are *Rename Method*, *Extract Method*, *Inline Method*, *Replace Temp with Query* and others. Additional refactorings have been added on the book's web site [25].

Each refactoring is following the same fixed format:

- a *name* used to build a common vocabulary for developers
- a *summary* to outline the situation in which the refactoring is carried out in one or two sentences
- a *motivation* to illustrate when and why a refactoring should be accomplished
- *mechanics* as step-by-step instructions about how to carry out the refactoring
- an *example* to illustrate how the refactoring works

Zeiss [61] picks up the concept of refactoring and applies it to TTCN-3 test suites. He presents a list of 28 refactorings from the Refactoring book which can be applied to TTCN-3 and 21 TTCN-3-specific refactorings. He subdivides them into refactorings for test behavior, for data descriptions and for improving the overall structure of a test suite.

The motivation part gives an idea of the reasons for carrying out a refactoring in a condensed and informal way. However, before being able to perform a refactoring the problematic areas in the code must be located by means of suitable indicators (see section 3.2).

Listings 3.1 and 3.2 give an impression of the *Extract Altstep* refactoring. Listing 3.1 shows the unrefactored version. The **alt** construct in test case *tc_exampleTestCase* (lines 5–17) contains three branches. The third branch (lines 13–16) is selected for an extraction.

```
 1  testcase tc_exampleTestCase() runs on ExampleComponent {
 2    timer t_guard;
 3    //...
 4    t_guard.start(10.0);
 5    alt {
 6      [] pt.receive(a_MessageOne) {
 7        pt.send(a_MessageTwo);
 8      }
 9      [] any port.receive {
10        setverdict(fail);
11        stop;
12      }
13      [] t_guard.timeout {
14        setverdict(fail);
15        stop;
16      }
17    }
18  }
```

*Listing 3.1: Extract Altstep (unrefactored)*

The refactored version is shown in listing 3.2. The branch has been moved into the new altstep *a_exampleAltstep* (lines 16–21). The local timer instance *t_guard* is passed into the altstep as parameter (line 13).

```
 1  testcase tc_exampleTestCase() runs on ExampleComponent {
 2    timer t_guard;
 3    //...
 4    t_guard.start(10.0);
 5    alt {
 6      [] pt.receive(a_MessageOne) {
 7        pt.send(a_MessageTwo);
 8      }
 9      [] any port.receive {
10        setverdict(fail);
11        stop;
12      }
13      [] a_exampleAltstep(t_guard);
14    }
15  }
```

```
altstep a_exampleAltstep(timer t) runs on ExampleComponent {
  [] t_guard.timeout {
    setverdict(fail);
    stop;
  }
}
```

*Listing 3.2: Extract Altstep (refactored)*

## 3.2 Code Smells

Before a refactoring can be performed, the relevant code parts have to be determined. Beck and Fowler introduce the metaphor of *bad smells*[1] for indicators that lead to starting points for refactorings. They describe smells as "certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring" [26, p. 75].

The smells are presented in a single flat list and described in a textual, rather informal way. They are connected to common refactorings via a table with a row for each smell. Some examples are ***Duplicated Code***, ***Long Method*** (containing too many statements), ***Large Class*** (containing too much functionality), ***Long Parameter List*** and ***Data Clumbs*** (data items appearing together regularly).

### 3.2.1 Terminology

As well as refactorings, the concept of code smells can be applied to TTCN-3. Before smells can be identified in the context of TTCN-3, the meaning of the term has to be encircled. The following statements are to narrow down the notion of code smells:

- Code smells affect the internal structure of the code. They are located on code level rather than on architectural level.
- Code smells are code parts whose quality is considered "bad". They are usually associated with bad program design and bad programming practices. In many cases code smells affect quality aspects like readability and maintainability rather than correctness.
- Code smells can be used to decide when and what to refactor. Van Emden et al. [16] state that the idea behind code smells is not necessarily to disallow any smells, but rather to indicate beneficial refactorings.

[1] The full title of the chapter reads "Bad Smells in Code". For this thesis, also the term *code smell* (or just *smell*) is used synonymously.

- As explained in section 3.1, refactoring is a behavior-preserving transformation. Hence, defects whose elimination changes the behavior of the code are not code smells in the strictest sense.
- A broader definition allows potential bugs as well, in contrast to bugs which can be safely identified as such. However, the elimination of these smells might not be behavior-preserving. (Some of the smells presented in chapter 4 require this broader approach).
- Defects concerning violation of syntax or static semantics are not among code smells. Therefore code smell analysis is only reasonable on code devoid of syntax errors and violations of static semantics, i.e. code that is free from defects usually recognized by compilers.

Most of the code smells presented in chapter 4 satisfy the above statements; however, there are some entries whose elimination may change the behavior. For example, an ***Idle PTC*** (4.7.1) can be fixed in two ways: On the one hand a PTC which is created but never started can be removed behavior-preservingly. On the other hand, a **start** statement can be inserted, which is not behavior-preserving but might reflect the intention.

### 3.2.2 Finding Code Smells

The smells descriptions presented by Fowler and Beck are intended to be utilized by developers for manual analysis. They state that when finding out when and what to refactor, "no set of metrics rivals informed human intuition" [26, p. 75]. However, it seems feasible to automate code smell detection to a certain degree, and there are a number of reasons why automated smell detection is favorable [42]:

- Automated analysis is time- and cost-saving. Although it does not redundantize manual code review, it certainly can help to localize problematic code parts and reduce expenditure of time for manual reviews.
- Developers working on the same task for a period of time usually become blind to the shortcomings of their own work and lose the ability to objectively judge the quality of their own code. A tool that automatically analyzes the code does not have this handicap.
- Code smells are more a matter of taste than syntax errors or functional defects. One developer might consider a code part a smell while another might not. Once a set of code smells rules has been agreed on, automated code smell analysis can objectively match the code against the rules.

- Together with automated refactorings, the whole process of analyzing and improving test suites can be automated — or at least semi-automated, leaving the confirmation and configuration of refactorings to the test developer. As a result of this thesis a tool for automated smell detection is presented in chapter 5.

There are different ways to locate bad smells. The most important ones other than human intuition are the use of *software metrics* or *pattern matching.*

Metrics can be used to assess the quality of code. A metric is a measure of some property of a piece of software. Well-know examples are *Lines of Code*, size metrics like *Program Volume* (from the Halstead metrics [34]) and structural metrics like McCabe's *Cyclomatic Number* [44]. Metrics for TTCN-3 have been presented [56, 62].

A metric is a quantification of certain characteristics of the underlying code. Together with boundary values, metrics can be utilized to locate problematic code parts. For example, a maximum value of 30 could be identified for the Lines of Code of a function. If this boundary is exceeded by a function, the presence of the smell ***Long Statement Block*** (4.4.1) is indicated.

This thesis follows up a more pattern-based approach. Smells are described as patterns of certain code structures. These patterns are matched against an AST as a representation of the source code. If an instance of a pattern can be found, it indicates the presence of a smell.

Although metrics are suitable for detecting certain smells (e.g. ***Unused Definition*** (4.2.3)), they seem to be inappropriate for finding others (e.g. ***Duplicate Statements*** (4.1.1)). However, the same shortcoming can often be expressed both by a pattern and by a metric. For example, the pattern "A template is never referenced" is equivalent to a metric "number of references to template" with a lower boundary of 1.

Automated pattern matching varies depending on the search domain (figure 3.1). The simplest variant works on textual level without any preceding analysis, which is too basic for the recognition of more sophisticated smells and impacted by formatting issues. A lexeme-based search works on the token stream and is formatting-independent, but is still not capable of finding advanced smells. The smell detection presented in chapter 5 works mainly on the AST. Some smells require additional semantic meta-information, like symbol table or control and data flow information.

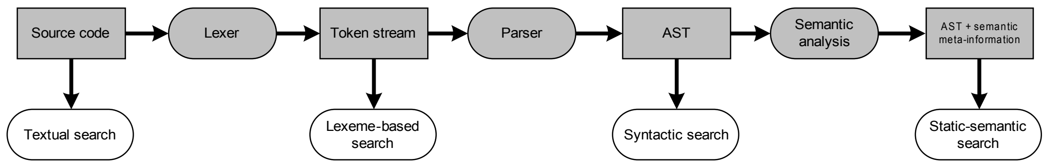


*Figure 3.1: Search domains for pattern matching*

The approach followed by this thesis is by static analysis, which seams feasible for most of the identified smells. Dynamic techniques rely on the runtime-behavior of the program (i.e. the test) and are limited to finding issues in the paths that are actually executed. In contrast, static techniques are able to examine abstractions of all possible program behaviors.

## 3.3 Related Work

Most of the existing work and tools about code smells and pattern matching was written with implementation languages in mind, often Java. Nevertheless, there are some works which present ideas and concepts valuable for smell detection in TTCN-3 source code.

### 3.3.1 Taxonomy

Fowler does not arrange smells in groups, but presents them in a single flat list. This is the starting point for Mäntylä [42]. He introduces a taxonomy for Fowler's code smells and subdivides them into categories named Bloaters, OO-Abusers, Change Preventers, Dispensables, Encapsulators, Couplers and Others. Although this categorization is not transferable to TTCN-3, the idea of grouping smells is followed in this thesis as well.

### 3.3.2 Similar Concepts

Code smells are often referred to as *anti-patterns* [6]. An anti-pattern is a description of a solution to a reoccurring problem. In contrast to *design patterns*, anti-patterns describe a "bad" solution as an example of how *not* to solve a problem. The most famous anti-pattern is probably spaghetti code, which describes a code structure that is barely comprehensible due to an overly complex control flow.

An anti-pattern is a more general concept than a smell: Beside anti-patterns for programming or design problems, there are also anti-patterns for scopes like project management, methodology, configuration management and others. Additionally, anti-patterns have a different position in the error/fault/failure chain [52, pp. 7]: Code smells describe internal manifestations of bad solutions (the "faults") while anti-patterns focus more on the bad solutions themselves (the "errors").

A *design defect* is another code-smell-related term [45]. It describes the absence or the bad use of design patterns. Design defects are more situated on a micro-architectural level while code smells are more located on code level. Just as well, *architecture smells* refer to the architectural level and are connected to extensive architectural refactorings [50].

### 3.3.3 Automated Smell Detection

Most of the existing work regarding code smells and automated pattern detection is tailored to programming languages rather than tests.

Van Emden et al. [16] propose automated code inspection for finding code smells in Java code. They include adherence to coding standards and conformance checks in their notion of code smells. Another important aspect of their approach is that the set of smells examined is configurable and extensible. Smell detection is applied on a source model which is constructed from the AST using static analysis to extract primitive properties and smell aspects. A smell aspect is a building block of a smell which can be observed directly in the code.

Hovemeyer et al. [36] use static analysis to find instances of bug patterns similar to the smell patterns used in this thesis. A bug pattern is defined as code idiom that is likely to be an error. It is originating from the use of erroneous design patterns, misunderstanding of language semantics or simple and common mistakes. Hence, the focus lies on detecting bugs rather than code smells.

### 3.3.4 Clone Detection

The detection of duplicated code (also know as *clone detection*) adopts an exceptional position among smell detection techniques. Clone detection has been studied thoroughly and emerged as field of research on its on. Bellon [4] gives a good overview and compares different techniques to each other [1, 2, 12, 35, 41, 43]. He distinguishes between three different types of clones: Exact clones (except for white spaces and comments), clones with renamed identifiers and clones with further modifications.

Clone detection is not the main focus of this thesis, and the implementation presented in chapter 5 currently uses a very basic approach for finding duplicated code which only detects exact clones. However, a more sophisticated technique could be used instead. Especially the Baxter method [2] seems feasible. It is based on sub-tree matching on the AST and is able to find exact clones as well as clones with modifications.

### 3.3.5 Tools

There are a number of tools for automated code checking and code smell detection, again most of them for programming languages, especially Java.

PMD [49] is an open-source tool which scans Java source code for problems like possible bugs, dead code, suboptimal code, overcomplicated expressions and duplicated code. Rules can be contributed either programmatically by implementing a Java interface, or declaratively by specifing an XPath expression [8] on the AST. PMD builds on JavaCC [39]. The build-in clone detection, named Copy and Paste Detector, is applicable to Java, C, C++

and PHP source code and uses the Karp-Rabin string matching algorithm [40]. PMD can be integrated with Eclipse.

A similar tool is Checkstyle [7], which focusses more on conformance checking. Like TRex it builds on ANTLR. It has a build-in duplicated code checker which detects exact clones only and and interface to Simian [51], a comercial clone detector. Checkstyle uses the visitor pattern to implement checks for certain types of syntax tree nodes. There exists a plug-in for the integration with Eclipse.

FindBugs [24] looks for bugs in Java bytecode. It is realized both as command line tool and Eclipse plug-in. As the name suggests it focuses on defects concerning correctness rather than code smells as defined in this chapter. It is based on the concept of bug patterns, which is defined as “code idiom that is often an error”. A bug pattern is implemented as Java class using the visitor pattern.

Jackpot [38] is another tool for enhancing Java source code. It is integrated with the Netbeans IDE [46]. It uses queries to explore the code, report found patterns and (optionally) refactor the found patterns. Its domain is therefore not only the analysis, but also the transformation of source code. It has a simple build-in rule language which can be used to write own queries. More sophisticated queries can be contributed by implementing a Java class.

# 4 A TTCN-3 Code Smell Catalog

The concept of code smells can be utilized not only for implementation languages, but for TTCN-3 as well. This chapter presents a catalog of 39 code smells for TTCN-3 which can be used for quality assessment of test suites and localization of possibly problematic code parts. It contains both general smells applicable to TTCN-3 and TTCN-3-specific smells. The list is not exhaustive, however, it is the first attempt known to gather and organize smells for TTCN-3. Most smells are intended to be detectable by static analysis, although some of them (especially those concerning test behavior) require the aid of dynamic analysis to be fully detectable.

A main source for the following catalog was Fowler's refactoring book [26]. Smells from this book were designed with the Java programming language in mind. Some of them are not applicable for TTCN-3, to some extend because they deal with object-oriented concepts. Others could be transferred more easily. Further sources were the "motivation" parts from Zeiss' TTCN-3 refactoring catalog [61, pp. 27] and rules from existing Java smell detection tools [7, 24, 38, 49].

Each smell is listed in accordance with the following format:

- *Name*: The name of each smell is used consistently throughout the whole thesis in order to build a common vocabulary for developers.
- *Derived from* (optional): The source(s) for this smell entry are named (if any).
- *Description*: A depiction of how an instance of this smell looks like is given.
- *Motivation*: The question why the structure described is classified a smell is met in this part.
- *Options* (optional): If there are feasible variations for the smell, they are pointed out here.
- *Related action(s)*: Appropriate actions (usually refactorings) for the removal of this smell are presented. All references to names of refactorings from Zeiss' refactoring catalog are printed in *slanted* type.
- *Example*: The smell is illustrated with the aid of a short code excerpt.

References to names of code smells both from the following catalog and from the Refactoring book are printed in ***bold and slanted*** type. Additionally, references to smells from the catalog include the section number in parentheses.

The 39 code smells are organized in categories according to common characteristics. The categories are certainly not the only reasonable organization for the smells. Their intention is rather to add structure and clarity to the list. The structure is as follows:

### Duplicated Code

- Duplicate Statements
- Duplicate Alt Branches
- Duplicated Code in Conditional
- Duplicate In-Line Templates
- Duplicate Template Fields
- Duplicate Component Definition
- Duplicate Local Variable/Constant/Timer

### References

- Singular Template Reference
- Singular Component Variable/Constant/Timer Reference
- Unused Definition
- Unused Imports
- Unrestricted Imports

### Parameters

- Unused Parameter
- Constant Actual Parameter Value
- Fully-Parameterized Template

**Complexity**

- Long Statement Block
- Long Parameter List
- Complex Conditional
- Nested Conditional
- Short Template

**Default Anomalies**

- Activation Asymmetry
- Unreachable Default

**Test Behavior**

- Missing Verdict
- Missing Log
- Stop in Function

**Test Configuration**

- Idle PTC
- Isolated PTC

**Coding Standards**

- Magic Values
- Bad Naming
- Disorder
- Insufficient Grouping
- Bad Comment Rate
- Bad Documentation Comment

#### Data Flow Anomalies

- Missing Variable Definition
- Unused Variable Definition
- Wasted Variable Definition

#### Miscellaneous

- Name-clashing Import
- Over-specific Runs On
- Goto

## 4.1 Duplicated Code

Duplicated code is considered the most frequent code smell [26, p. 76]. It is common practice to copy and paste code blocks if similar behavior is needed. The problem with this is that code duplication blows up code size and affects maintainability in a massive way.

Fowler focuses on duplicated statements in methods. For TTCN-3 it makes sense to consider other kinds of duplicated code, too, in behavioral code as well as data description and test configuration.

### 4.1.1 Duplicate Statements

**Derived from:** [26] (***Duplicated Code***)

**Description:** A duplicate sequence of statements in the statement block of one or multiple behavioral entities (functions, test cases and altsteps). Special cases like code duplication in **alt** constructs (section 4.1.2) and conditionals (section 4.1.3) are listed separately.

**Motivation:** Code duplication should be avoided. Especially large sequences of duplicated statements can often be extracted into a common function.

**Related action(s):** *Extract Function*, *Parameterize Function*

**Example:** Function *f_sendMessages* contains a code sequence (listing 4.1, lines 3–5) which has a duplicate (lines 6–8). The sequence could be extracted into a separate (parameterized) function.

```
function f_sendMessages(in float p_duration) runs on ExampleComponent {
  timer t;
  t.start(p_duration);
  t.timeout;
  pt.send("first timeout");
  t.start(p_duration);
  t.timeout;
  pt.send("second timeout");
}
```

*Listing 4.1: Duplicate Statements*

### 4.1.2 Duplicate Alt Branches

**Derived from:** [61] (Motivations for *Extract Altstep*, *Split Altstep* and *Replace Altstep with Default*)

**Description:** Different **alt** constructs contain duplicate branches.

**Motivation:** Code duplication in branches of **alt** constructs should be avoided just as well as any other duplicated code. Especially common branches for error handling can often be handled by default altsteps if extracted into an own altstep beforehand.

**Related action(s):** *Extract Altstep* to separate the duplicate branches into an own altstep. Consider *Split Altstep* if the extracted altstep contains branches which are not closely related to each other and *Replace Altstep with Default* if the duplicate branches are invariably used at the end of the **alt** construct as default branches.

**Example:** In listing 4.2, both test cases contain **alt** constructs with three alternatives. The last two alternatives in both **alt** constructs (lines 9–16 and lines 27–34)) are identical and could be extracted into a separate altstep.

```
1  testcase tc_exampleTestCase1() runs on ExampleComponent {
2    timer t_guard;
3    //...
4    t_guard.start(10.0);
5    alt {
6      [] pt.receive(a_MessageOne) {
7        pt.send(a_MessageTwo);
8      }
9      [] any port.receive {
10       setverdict(fail);
11       stop;
12     }
13     [] t_guard.timeout {
14       setverdict(fail);
15       stop;
16     }
17   }
18 }
```

```
testcase tc_exampleTestCase2() runs on ExampleComponent {
  timer t_guard;
  //...
  t_guard.start(10.0);
  alt {
    [] pt.receive(a_MessageThree) {
      pt.send(a_MessageFour);
    }
    [] any port.receive {
      setverdict(fail);
      stop;
    }
    [] t_guard.timeout {
      setverdict(fail);
      stop;
    }
  }
}
```

*Listing 4.2: Duplicate Alt Branches*

### 4.1.3 Duplicated Code in Conditional

**Derived from:** [26] (Motivations for *Consolidate Conditional Expression* and *Consolidate Duplicate Conditional Fragments*)

**Description:** The duplicated code can appear in a series of conditionals (with different conditions and the same action in each check) or in all legs of a conditional.

**Motivation:** Duplicated code should be avoided. Additionally, conditionals should be kept as simple as possible.

**Related action(s):** *Consolidate Conditional Expression*, *Consolidate Duplicate Conditional Fragments*

**Example:** Function *f_checkSomething* contains identical **return** statements in the first three legs of the conditional construct (listing 4.3, lines 3, 6, 9). Instead, the three conditions could be combined into a single expression. Function *f_checkSomethingElse* contains duplicate **send** statements in different legs of the conditional construct (lines 18, 21). These statements could be moved out of the conditional.

```
1  function f_checkSomething(in float p1, in float p2) return boolean {
2    if (p1 < 0.0) {
3      return false;
4    }
5    if (p2 >= 7.0) {
6      return false;
7    }
8    if (p2 < p1) {
    return false;
  }
  return true;
}

function f_checkSomethingElse(in float p1) runs on ExampleComponent {
  var charstring result;
  if (p1 > 0) {
    result := "foo";
    pt.send(result);
  } else {
    result := "bar";
    pt.send(result);
  }
}
```

*Listing 4.3: Duplicated Code in Conditional*

### 4.1.4 Duplicate In-Line Templates

**Derived from:** [61] (Motivation for *Extract Template*)

**Description:** Two or more similar or identical in-line templates.

**Motivation:** To maximize maintainability, duplicate in-line templates should be extracted. Note that extracting templates can lead to a ***Short Template*** (4.4.5). If the templates are not identical but very similar, the different fields can be parameterized or a common base template can be introduced (see also ***Duplicate Template Fields*** (4.1.5)).

**Related action(s):** *Extract Template*

**Example:** In listing 4.4, test case *tc_exampleTestCase* contains two identical templates in the first two **send** statements (lines 17, 19) and another similar template in the third **send** statement (line 21).

```
 1 module DuplicateInlineTemplates {
 2   type record ExampleRecordType {
 3     boolean exampleField1,
 4     integer exampleField2,
 5     charstring exampleField3
 6   }
 7
 8   type port ExamplePort message {
 9     out ExampleRecordType;
10   }
11
12   type component ExampleComponent {
13     port ExamplePort pt;
14   }
```

```
  testcase tc_exampleTestCase() runs on ExampleComponent {
    //...
    pt.send(ExampleRecordType:{true, omit, "foo"});
    //...
    pt.send(ExampleRecordType:{true, omit, "foo"});
    //...
    pt.send(ExampleRecordType:{true, omit, "bar"});
  }
}
```

*Listing 4.4: Duplicate In-Line Templates*

### 4.1.5 Duplicate Template Fields

**Derived from:** [61] (Motivations for *Replace Template with Modified Template*, *Parameterize Template* and *Decompose Template*)

**Description:** The fields of two or more templates are identical or very similar.

**Motivation:** Two or more very similar or identical templates should be merged into one to avoid code duplication. If similar templates differ on the same fields, they can be parameterized on these fields. If the templates differ on varying fields, a common base template can be extracted which can be modified by the other templates on the varying fields. If the templates have a common record type field, they should be decomposed.

**Related action(s):** *Replace Template with Modified Template*, *Parameterize Template*, *Decompose Template*

**Example:** In listing 4.5, templates *t1* (lines 1–5)and *t2* (lines 7–11) are identical and could be merged without parameterization. Template *t3* (lines 13–17) has a different value for *field1* (line 14) and could be merged with the other two templates into a parameterized template.

```
1  template MyRecordType t1 := {
2    field1 := "foo",
3    field2 := 1,
4    field3 := true
5  }
6
7  template MyRecordType t2 := {
8    field1 := "foo",
9    field2 := 1,
10   field3 := true
11 }
12
13 template MyRecordType t3 := {
14   field1 := "bar",
15   field2 := 1,
  field3 := true
}
```

*Listing 4.5: Duplicate Template Fields*

### 4.1.6 Duplicate Component Definition

**Derived from:** [61] (Motivation for *Extract Parent Component*)

**Description:** Two or more components declare identical variables, constants, timers or ports.

**Motivation:** If multiple components contain the same definition, it should be moved to a parent component to avoid code duplication.

**Related action(s):** *Extract Parent Component*

**Example:** Components *c1* and *c2* both declare a timer *t* (listing 4.6, lines 4, 10) and a port of type *ExamplePort* (lines 5, 11). These elements could be moved to a common parent component.

```
1  type component c1 {
2    var integer i;
3    const integer id := 1;
4    timer t;
5    port ExamplePort p1;
6  }
7
8  type component c2 {
9    const integer id := 2;
10   timer t;
11   port ExamplePort p2;
12 }
```

*Listing 4.6: Duplicate Component Definition*

### 4.1.7 Duplicate Local Variable/Constant/Timer

**Derived from:** [61] (Motivation for *Move Local Variable/Constant/Timer to Component*)

**Description:** The same local variable, constant or timer is defined in two or more functions, test cases or altsteps running on the same component.

**Motivation:** If multiple functions, test cases or altsteps are running on the same component and define an identical local variable, constant or timer, the local definition should be moved to component scope.

**Related action(s):** *Move Local Variable/Constant/Timer to Component*

**Example:** Both *tc1* and *tc2* run on *c* and both define a local timer *t* (listing 4.7, lines 6, 23). Hence *t* should be defined in *c*.

```
1  type component c {
2    port ExamplePort p;
3  }
4
5  testcase tc1() runs on c {
6    timer t;
7    p.send("foo1");
8    t.start(10.0);
9    alt {
10     [] p.receive("bar1") {
11     // do something
12     }
13     [] any port.receive {
14     // error handling
15     }
16     [] t.timeout {
17     // error handling
18     }
19   }
20 }
21
22 testcase tc2() runs on c {
23   timer t;
24   p.send("foo2");
25   t.start(20.0);
26   alt {
27     [] p.receive("bar2") {
28     // do something
29     }
30     [] any port.receive {
31     // error handling
32     }
33     [] t.timeout {
34     // error handling
35     }
36   }
37 }
```

*Listing 4.7: Duplicate Local Variable/Constant/Timer*

## 4.2 References

The code smells from this section concern the number and quality of references to a definition. Note that only for local elements the number of references can be safely determined. For global elements, modules out of the scope of the smell analysis may import definitions from analyzed modules. Hence, there might exist additional references to global definitions.

### 4.2.1 Singular Template Reference

**Derived from:** [61] (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 4.8, assume the only reference to template *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 4.8: Singular Template Reference*

### 4.2.2 Singular Component Variable/Constant/Timer Reference

**Derived from:** [61] (Motivation for *Move Component Variable/Constant/Timer to Local Scope*)

**Description:** A component variable, constant or timer is referenced by one single function, test case or altstep only, although other behavioral entities run on the component as well.

**Motivation:** To increase reuse potential of components, variables, constants and timers that are specific to a behavioral entity should not be defined in the component. However, sometimes it will make sense to leave the definition in the component even if it is referenced only once.

**Related action(s):** *Move Component Variable/Constant/Timer to Local Scope*

**Example:** Component *c* is used by function *f* and test case *tc*. Only *tc* references timer *t* (listing 4.9, line 17), hence the definition of *t* could be moved to *tc*.

```
1  module SingularComponentVCTReference {
2    type port ExamplePort message {
3      inout charstring;
4    }
5
6    type component c {
7      timer t;
8      port ExamplePort p;
9    }
10
11   function f() runs on c {
12     p.send("bar");
13     p.send("baz");
14   }
15
16   testcase tc() runs on c {
17     t.start(10.0);
18     alt {
19       [] p.receive("foo") {
20         p.send("bar");
21       }
22       [] any port.receive {
23       // error handling
24       }
25       [] t.timeout {
26       // error handling
27       }
28     }
29   }
30 }
```

*Listing 4.9: Singular Component Variable/Constant/Timer Reference*

### 4.2.3 Unused Definition

**Description:** A definition is never referenced (also known as *dead code*).

**Motivation:** Unused code should be removed. Note that only local definitions can be removed safely because they cannot be accessed from outside the defining unit. For global definitions there might exist references in modules which have not been considered.

**Related action(s):** Remove the unused definition.

**Example:** Function *f* defines local variables *j* and *k* (listing 4.10, line 2). Only *j* is used in *f*, hence *k* can be safely removed.

```
function f(in integer i) return integer {
  var integer j := 42, k;
  return i + j;
}
```

*Listing 4.10: Unused Decfinition*

### 4.2.4 Unused Imports

**Description:** An import from another module is never used.

**Description:** Unused **import** statements should be removed, because they increase complexity unnecessarily.

**Related action(s):** Remove the unused imports.

**Example:** In listing 4.11, module *Baz* imports all definitions from *Foo* (line 10) and *Bar* (line 11). Only the import from *Foo* is used in *Baz* (line 14), hence the import from *Bar* can be removed.

```
 1  module Foo {
 2    const charstring FOO_CONST := "foo";
 3  }
 4
 5  module Bar {
 6    const charstring BAR_CONST := "bar";
 7  }
 8
 9  module Baz {
10    import from Foo all;
11    import from Bar all;
12
13    function f(in charstring s) return boolean {
14      if (FOO_CONST == s) {
15        return true;
16      }
17      return false;
18    }
19  }
```

*Listing 4.11: Unused Imports*

### 4.2.5 Unrestricted Imports

**Derived from:** [61] (Motivation for *Restrict Imports*)

**Description:** A module imports more from another module than needed.

**Motivation:** In general, only required elements/groups of other modules should be imported to clarify the dependencies between modules. On the other hand, a too detailed **import** statement can affect readability. A good compromise seems to organize jointly used definitions in groups and import the group.

**Related action(s):** *Restrict Imports*

**Example:** In listing 4.12, module *Bar* imports all definitions from *Foo* (line 18). However, only constant *FOO_CONST* is used in *Bar* (line 21), hence the import could be restricted to importing only the constant or at least only group *groupConstants*.

```
1  module Foo {
2    group groupConstants {
3      const charstring FOO_CONST := "foo";
4      // some other constants...
5    }
6
7    group groupTypes {
8      // type definitions...
9    }
10
11   group groupComponents {
12     // component definitions...
13   }
14   // further definions...
15 }
16
17 module Bar {
18   import from Foo all;
19
20   function f(in charstring s) return boolean {
21     if (FOO_CONST == s) {
22       return true;
23     }
24     return false;
25   }
26 }
```

*Listing 4.12: Unrestricted Imports*

## 4.3 Parameters

These code smells deal with anomalies concerning parameterization. In TTCN-3 templates, functions, altsteps, test cases and signatures can be parameterized. Types and modules support static parameterization.

### 4.3.1 Unused Parameter

**Derived from:** [26] (Motivation for *Remove Parameter*)

**Description:** A parameter is never used within the declaring unit. For **in**-parameters, the parameter is never read, for **out**-parameters never defined, for **inout**-parameters never accessed at all.

**Motivation:** Parameterization increases complexity, hence unused parameters should be removed (from the declaration and all references).

**Related action(s):** *Remove Parameter*

**Example:** Function *f* declares parameters *i* and *j* (listing 4.13, line 1). Only *i* is in use (line 3), hence *j* can be removed.

```
1 function f(in integer i, in integer j) return integer {
2   var integer k := 1;
3   return i + k;
4 }
```

*Listing 4.13: Unused Parameter*

### 4.3.2 Constant Actual Parameter Value

**Derived from:** [61] (Motivation for *Inline Template Parameter*)

**Description:** The value of an actual parameter is the same for all occurances. In contrast to ***Unused Parameter*** (4.3.1), the parameter is in use within the declaring entity and must not simply be removed. The declaring entity could be a template or a behavioral entity (function, test case or altstep).

**Motivation:** Parameterization increases complexity, hence unneeded parameters should be removed.

**Related action(s):** *Inline Template Parameter*[1]

**Example:** Assume template *t* is referenced only by function *f* (listing 4.14, lines 9, 11, 13). In all references, parameter *p1* has the same value (`foo`), so this parameter could be inlined.

```
1 template myType t(charstring p1, integer p2) := {
2   field1 := true,
3   field2 := p2,
4   field3 := p1
5 }
6
```

[1]The refactoring focuses on inlining template parameters. Inlining parameters of functions, test cases and altsteps works similarly.

```
function f() runs on myComponent {
  //...
  p.send(templateA("foo", 42));
  //...
  p.send(templateA("foo", 42));
  //...
  p.send(templateA("foo", 43));
}
```

*Listing 4.14: Constant Actual Parameter Value*

### 4.3.3 Fully-Parameterized Template

**Derived from:** [62] (Rule 5)

**Description:** All fields of a template are defined by formal parameters.

**Motivation:** A template with all fields defined by formal parameters has no information on its own and can be replaced by in-line templates.

**Related action(s):** *Inline Template*

**Example:** In listing 4.15, all template fields of *exampleTemplate* are defined by formal parameters (lines 8–10). Hence an in-line template could be used just as well.

```
1  type record MyMessageType {
2    integer field1,
3    charstring field2,
4    boolean field3
5  }
6
7  template MyMessageType exampleTemplate(integer i, charstring c, boolean b) := {
8    field1 := i,
9    field2 := c,
10   field3 := b
11 }
12
13 function f() runs on MyComponent {
14   // ...
15   p.send(exampleTemplate(42, "dent", true));
16   // ...
17 }
```

*Listing 4.15: Fully-Parameterized Template*

## 4.4 Complexity

Code smells from this category increase complexity unnecessarily.

### 4.4.1 Long Statement Block

**Derived from:** [26] (***Long Method***)

**Description:** Long statement block in function, test case or altstep.

**Motivation:** A long function is more difficult to understand than a short one. Although the use of short functions (i.e. methods) is especially important for modern object-oriented languages, short functions have a certain importance for TTCN-3 as well. Long statement blocks in functions, test cases and altsteps should be decomposed into short functions with meaningful names.

**Related action(s):** *Extract Function*, *Parameterize Function*

**Example:** Listing 4.16 shows a function with a long statement block (taken from [22]). Especially the nested **if** construct (lines 20–34) could be simplified and branches could be extracted from the **alt** construct or be replaced by default behavior. (lines 8–54).

```
1  function ptc_CC_PR_MP_RQ_V_030(CSeq loc_CSeq_s) runs on SipComponent {
2    var Request v_INVITE_Request;
3    var Request v_BYE_Request;
4    var Request v_ACK_Request;
5    var charstring v_branch := "";
6    initPTC(loc_CSeq_s);
7    v_Default := activate(defaultCCPRPTC());
8    alt {
9      [] SIPP.receive(INVITE_Request_r_1) ->
10         value v_INVITE_Request sender sent_label {
11       TGuard.stop;
12       setHeadersOnReceiptOfInvite(v_INVITE_Request);
13       sendPTC200OKInvite();
14       setverdict(pass);
15       repeat;
16     }
17     [] SIPP.receive(ACK_Request_r_1(v_CallId)) ->
18         value v_ACK_Request sender sent_label {
19       v_Via := v_ACK_Request.msgHeader.via;
20       if (ispresent(v_Via.viaBody[0].viaParams)) {
21         var SemicolonParam_List tmp_params :=
22             v_Via.viaBody[0].viaParams;
23         if (checkBranchPresent(tmp_params, v_branch)) {
24           if (match(v_branch, ValidBranch)) {
25             setverdict(pass);
26           } else {
27             setverdict(fail);
28           };
29         } else {
30           setverdict(fail)
31         }
32       } else {
33         setverdict(fail)
34       };
```

```
      cpA.send(CM_Check_Done);
      repeat;
    }
    [] SIPP.receive(BYE_Request_r_1(v_CallId)) ->
        value v_BYE_Request sender sent_label {
      setHeadersOnReceiptOfBye(v_BYE_Request);
      send200OK();
    }
    [] cpA.receive(CM_Stop) {
      all timer.stop;
      stop;
    }
    [] SIPP.receive {
      repeat;
    }
    [] TGuard.timeout {
      setverdict(fail);
      stop;
    }
  }
}
```

*Listing 4.16: Long Statement Block*

### 4.4.2 Long Parameter List

**Derived from:** [26] (*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 behavioral 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*, *Introduce Record Type Parameter*

**Example:** Listing 4.17 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 behavior...
4 }
```

*Listing 4.17: Long Parameter List*

### 4.4.3 Complex Conditional

**Derived from:** [26] (Motivation for *Decompose Conditional*)

**Description:** A conditional expression is composed of many boolean conjunctions.

**Motivation:** A complex conditional statement is hard to understand. Often it can be replaced by a call of a function with a meaningful name.

**Related action(s):** *Decompose Conditional*

**Example:** The condition of the **if** construct (listing 4.18, lines 3–4) could be simplified by extracting the boolean expression into a meaningful function (e.g. *isLeapYear*).

```
 1 function calculateAmount(integer year) return float {
 2   var float amount;
 3   if (((year mod 4) == 0 and not (year mod 100) == 0)
 4       or (year mod 400) == 0) {
 5     amount := BASE_AMOUNT * 366;
 6   } else {
 7     amount := BASE_AMOUNT * 365;
 8   }
 9   return amount;
10 }
```

*Listing 4.18: Complex Conditional*

### 4.4.4 Nested Conditional

**Derived from:** [26] (Motivation for *Replace Nested Conditional With Guard Clause*)

**Description:** Nested conditional expression

**Motivation:** Use **if** and **else** leg of a conditional only if both paths are part of the normal behavior; if one leg is an unusual condition, use a separate exit point (guard clause) instead.

**Related action(s):** *Replace Nested Conditional With Guard Clause*

**Example:** Listing 4.16 from smell ***Long Statement Block*** (4.4.1) contains an example for a nested conditional (lines 20–34) whose **else** branches could be replaced by guard clauses.

### 4.4.5 Short Template

**Derived from:** [62] (Rule 3)

**Description:** A template definition is very short (in terms of characters or number of fields).

**Motivation:** To maximize readability, short template definition can be inlined even if they are referenced more than once (see ***Singular Template Reference*** (4.2.1)). Note that inlining templates with multiple references leads to ***Duplicate In-Line Templates*** (4.1.4). Hence, this smell is important only for readability.

**Related action(s):** *Inline Template*

**Example:** In listing 4.19, an example is given for a rather short template definition (line 1). Even if the template is referenced more than once, an in-line template would shorten code length and increase readability.

```
1 template integer exampleTemplate := 1
2
3 testcase exampleTestCase() runs on ExampleComponent {
4   // ...
5   pt.send(exampleTemplate);
6   // ...
7 }
```

*Listing 4.19: Short Template*

## 4.5 Default Anomalies

The TTCN-3 default mechanism is a powerful means for handling default behavior, but for a reader of the source code it can be difficult to determine all active defaults at a certain position of an execution path. The smells presented in this section are detectable by static analysis, although determination of active defaults could best be achieved by dynamic analysis.

### 4.5.1 Activation Asymmetry

**Description:** A default activation has no matching subsequent deactivation in the same statement block, or a deactivation has no matching previous activation.

**Motivation:** For improved readability it is recommended that default activation and deactivation is done on the same "level", i.e. at the very beginning and end of the same statement block.

**Options:** Because defaults are deactivated at the end of a testcase run, statement blocks in test cases can be excluded optionally.

**Related action(s):** Default activation or deactivation should be added if missing, and matching default activation and deactivation should be moved to the same statement block.

**Example:** Function *deactivateDefault* only deactivates default *d* (listing 4.20, line 64) without any previous activation. Test case *myTestcase1* activates and deactivates default *myDefaultVar* in the same statement block (lines 19, 29), whereas *myTestcase2* only deactivates default *myDefaultVar* by itself (line 45) and uses function *activateDefault* for activation (line 35). Test case *myTestcase3* activates a default (line 52), but deactivates it in the statement block of an **if** construct (line 64). Hence, function *deactivate-Default*, test case *myTestcase2* and test case *myTestcase3* have an asymmetric default activation.

```
1   altstep myAltstep(timer t) runs on MyComponent {
2     [] any port.receive {
3       setverdict(fail)
4       log("unexpected message")
5     }
6     [] t.timeout {
7       setverdict(fail)
8       log("timeout")
9     }
10  }
11
12  function activateDefault(timer t) return default {
13    // no deactivation in this function!
14    return activate(myAltstep(t))
15  }
16
17  testcase myTestcase1() runs on MyComponent {
18    timer t
19    var default myDefaultVar := activate(myAltstep(t))
20    t.start(10.0)
21    alt {
22      [] p.receive(charstring:("foo1")) {
23        p.send("ack")
24      }
25      [] p.receive(charstring:("bar1")) {
26        p.send("nack")
27      }
28    }
29    deactivate(myDefaultVar)
30  }
31
32  testcase myTestcase2() runs on MyComponent {
33    timer t
34    // activation in function call
35    var default myDefaultVar := activateDefault(t)
```

```
  t.start(10.0)
  alt {
    [] p.receive(charstring:("foo2")) {
      p.send("ack")
    }
    [] p.receive(charstring:("bar2")) {
      p.send("nack")
    }
  }
  deactivate(myDefaultVar)
}

testcase myTestcase3() runs on MyComponent {
  // de-/activation in different statement blocks
  timer t
  var default myDefaultVar
  myDefaultVar := activate(myAltstep(t))
  t.start(10.0)

  if (2 > 1) {
    alt {
      [] p.receive(charstring:("foo5")) {
        p.send("ack")
      }
      [] p.receive(charstring:("bar5")) {
        p.send("nack")
      }
    }
    deactivate(myDefaultVar)
  }
}
```

*Listing 4.20: Activation Asymmetry*

### 4.5.2 Unreachable Default

**Description:** An **alt** statement contains an **else** branch while a default is active.

**Motivation:** The **else** branch of an **alt** statement is taken if no other branch is applicable. If a default is active at the same time, its branches come after all branches of the **alt** statement. Hence the default altstep can never be executed if an **else** branch is present.

**Related action(s):** The intended behavior should be clarified by either deactivating the default or moving the **else** branch to the default altstep.

**Example:** In listing 4.21, the **alt** construct in test case *myTestcase* has an active default (line 2) *and* contains an **else** branch (line 10).

```
testcase myTestcase() runs on MyComponent {
  var default myDefaultVar := activate(myAltstep(t))
  alt {
    [] p.receive(charstring:("foo1")) {
      p.send("ack")
    }
    [] p.receive(charstring:("bar1")) {
      p.send("nack")
    }
    [else] {
      setverdict(fail)
      log("unexpected behavior")
    }
  }
  deactivate(myDefaultVar)
}
```

*Listing 4.21: Unreachable Default*

## 4.6 Test Behavior

The following smells concern the flow of a test. They are mostly connected with TTCN-3 specific constructs. Some of them are not smells in the strictest sense, because the cannot necessarily be removed without changing the behavior. However, they might point to possibly erroneous code.

### 4.6.1 Missing Verdict

**Description:** A test case does not set a verdict.

**Motivation:** Normally a test case should set a verdict before terminating.

**Options:** The **setverdict** statement can be considered essential at further locations, e.g. before **stop** statements.

**Related action(s):** Insert a **setverdict** statement (if missing) or move it to the testcase if the statement is part of a subfunction call.

**Example:** Test case *exampleTestCase* does not set a verdict if the **timeout** branch of the **alt** statement is taken (listing 4.22, lines 14–16).

```
1  testcase exampleTestCase() runs on ExampleComponent {
2    timer t_guard;
3    //...
4    t_guard.start(10.0);
5    alt {
6      [] pt.receive(a_MessageOne) {
7        t_guard.stop
8        setverdict(pass)
9        pt.send(a_MessageTwo);
10     }
11     [] any port.receive {
12       repeat;
13     }
14     [] t_guard.timeout {
15       stop;
16     }
17   }
18 }
```

*Listing 4.22: Missing Verdict*

### 4.6.2 Missing Log

**Derived from:** [61] (Motivation for *Add Explaining Log*)

**Description:** **setverdict** is used with verdict **inconc** or **fail**, but without calling **log**.

**Motivation:** Inconclusive or unsuccessful test verdicts should be logged, because this helps discovering the reasons for the failure. However, this smell should be classified weak compared to other smells.

**Related action(s):** *Add Explaining Log*

**Example:** Test case *exampleTestCase* does not log the reason for the **fail** verdict in the **timeout** branch of the **alt** statement (listing 4.22, line 15).

```
1  testcase exampleTestCase() runs on ExampleComponent {
2    timer t_guard;
3    //...
4    t_guard.start(10.0);
5    alt {
6      [] pt.receive(a_MessageOne) {
7        t_guard.stop
8        setverdict(pass)
9        pt.send(a_MessageTwo);
10     }
11     [] any port.receive {
12       repeat;
13     }
14     [] t_guard.timeout {
15       setverdict(fail)
16       stop;
17     }
18   }
19 }
```

*Listing 4.23: Missing Log*

### 4.6.3 Stop in Function

**Description:** A function contains a **stop** statement.

**Motivation:** If possible, functions should not contain any **stop** statement, because this can prevent the execution of postambles (e.g. code that has to be executed after each test case). Instead, functions should use return values. However, this smell should be classified weak compared to other smells.

**Options:** As a restriction, only functions without a **runs on** clause could be examined for **stop** statements.

**Related action(s):** Use return value instead of **stop** statement.

**Example:** In listing 4.24, function *f* contains a **stop** statement (line 11).

```
1  function f() {
2    timer t := 50;
3    t.start;
4    alt {
5      [] p.receive("foo") {
6        t.stop;
7        setverdict(pass);
8      }
9      [] t.timeout {
10       setverdict(inconc);
11       stop;
12     }
13   }
14 }
```

*Listing 4.24: Stop in Function*

## 4.7 Test Configuration

These smells are related to anomalies concerning creation and connection of test components which make up the architecture for distributed tests.

### 4.7.1 Idle PTC

**Description:** A PTC is created but never started.

**Motivation:** A PTC which is not started is of no use for the test case.

**Related action(s):** Insert a **start** statement or remove the PTC.

**Example:** Test case *exampleTestCase* creates a PTC *exampleComponent* (line 3) and connects it to the MTC (line 5), but fails to start any behavior on the PTC.

```
1  testcase exampleTestCase() runs on MainComponentType system SystemType {
2    // ...
3    var ParallelComponentType exampleComponent := ParallelComponentType.create;
4    map(self:aPort, system:aPort);
5    connect(self:anotherPort, exampleComponent:aPort);
6    // no start here...
7  }
```

*Listing 4.25: PTC not started*

### 4.7.2 Isolated PTC

**Description:** A PTC is created and started, but neither connected to another component nor mapped to the TSI.

**Motivation:** A PTC which is not connected or mapped is isolated from all other components, especially the MTC, and is of no use for the test.

**Related action(s):** Insert a **connect/map** statement or remove the PTC.

**Example:** In listing 4.26, test case *exampleTestCase* creates a PTC *exampleComponent* (line 3), but fails to connect or map its ports to other ports before it is started.

```
1  testcase exampleTestCase() runs on MainComponentType system SystemType {
2    // ...
3    var ParallelComponentType exampleComponent := ParallelComponentType.create;
4    // no map or connect statements here!
5    exampleComponent.start(exampleBehavior())
6    exampleComponent.done
7    // ...
8  }
```

*Listing 4.26: PTC isolated*

## 4.8 Coding Standards

Coding standards ensure that code is written according to given programming guidelines, formatting rules, commenting, naming and ordering conventions. Note that to be able to check conformance to coding standards, a set of guidelines has to be agreed on beforehand. Nevertheless, it makes sense to treat violations of coding standards as smells — or at least as smell-templates instantiated by concrete (project-specific) definitions.

### 4.8.1 Magic Values

**Derived from:** [26] (Motivation for *Replace Magic Number with Symbolic Constant*)

**Description:** Magic Values are literals not defined as constant. Numeric literals are called Magic Numbers, string literals are called Magic Strings.

**Motivation:** The use of Magic Values should be avoided. Instead, constants should be used with a meaningful name. This can improve readability and make the code easier to understand. Additionally maintainability is improved, because if the value changes, there will only be one point of change.

**Options:** There are a number of cases in which Magic Values could be excluded from being considered a code smell:

- Certain values (e.g. `0` and `1` for numbers) could be excluded from being recognized as smell.
- Certain code areas could be excluded, e.g. literals in user-defined types or template fields or the initialization and increment statements of **for**-loops.
- Literals could be allowed in variable initialization (e.g. `var integer i := 42`) and disallowed in variable assignments (`i := 42`).
- Literals could be considered a code smell only if they appear in the code more than once.
- Strings with a length below a threshold value could be excluded.

**Related action(s):** *Replace Magic Number with Symbolic Constant*. Note that values which are specific to the test environment should be stored in a module parameter rather than a module constant (consider *Parameterize Module*).

**Example:** In listing 4.27 (taken from [22]), test case *SIP_CC_PR_TR_SE_TI_004* uses some Magic Numbers for the definition of a local variable (lines 19, 21, 24) that is used in calls to *repeatRespInTime*. By the use of well-named constants, the meaning of these numbers could be made clearer.

```
1  testcase SIP_CC_PR_TR_SE_TI_004(inout CSeq loc_CSeq_s, CSeq loc_CSeq_ptc_s)
2     runs on SipComponent system SipInterfaces {
3    var SipComponent v_ptc;
4    var Response v_Response;
5    var float v_delay;
6    v_Default := activate(defaultCCPR());
7    v_ptc := SipComponent.create;
8    initConfig1(mtc, v_ptc, system);
9    initMTCphase1(loc_CSeq_s);
10   setHeadersPtcInvite(loc_CSeq_s);
```

```
  v_ptc.start(ptc_Wait_Check_Invite_Completed_State(loc_CSeq_ptc_s));
  initMTCphase2();
  SIPP.send(INVITE_Request_s_2(v_RequestUri, v_CallId, loc_CSeq_s, v_From,
      v_To, v_Via)) to sent_label;
  v_CSeq := loc_CSeq_s;
  awaitingFirstAnyFinalResp(v_Response, loc_CSeq_s);
  setHeadersOnReceiptOfResponse(loc_CSeq_s, v_Response);
  // First Repetition
  repeatRespInTime(v_Response, loc_CSeq_s, PX_T1 * 1.5);
  // Second Repetition
  v_delay := minValue(2.0 * PX_T1, PX_T2) * 1.5;
  repeatRespInTime(v_Response, loc_CSeq_s, v_delay);
  // Third repetition
  v_delay := minValue(4.0 * PX_T1, PX_T2) * 1.1;
  repeatRespInTime(v_Response, loc_CSeq_s, v_delay);
  sendACK(loc_CSeq_s);
  synchroniseCheckDone();
  wait_end_ptc(v_ptc);
} // end testcase SIP_CC_PR_TR_SE_TI_004
```

*Listing 4.27: Magic Values*

### 4.8.2 Bad Naming

**Derived from:** [18]

**Description:** An identifier does not conform to a given naming convention. Before this smell can be detected, the naming conventions have to be agreed on. Proposed naming conventions for TTCN-3 can be found on the official TTCN-3 home page [18].

**Motivation:** Naming Conventions can make code more readable and comprehensible.

**Related action(s):** *Rename*

**Example:** Function *calculateSomething* (listing 4.28, line 1) does not conform to the ETSI naming conventions [18]; function *f_calculateSomethingElse* (line 5) start with "f_" and does conform to the conventions.

```
1 function calculateSomething() {
2   //...
3 }
4
5 function f_calculateSomethingElse() {
6   //...
7 }
```

*Listing 4.28: Bad Naming*

### 4.8.3 Disorder

**Description:** The sequence of elements within a module does not conform to a given order. A preferred ordering could be:

- imports
- module parameters
- data types
- port types
- component types
- templates
- functions
- altsteps
- test cases
- control part

**Motivation:** Consistent ordering can improve readability.

**Related action(s):** *Reorganize Fragments*

**Example:** In listing 4.29, function *f* is declared atop (line 1), followed by a type and a template definition lines 5, 9). If the above ordering was valid, the function declarations would be in the wrong order.

```
 1  function f() {
 2    //...
 3  }
 4
 5  type record exampleRecordType {
 6    //...
 7  }
 8
 9  template exampleRecordType t := {
10    //...
11  }
```

*Listing 4.29: Disorder*

### 4.8.4 Insufficient Grouping

**Derived from:** [61] (Motivations for *Group Fragments*)

**Description:** A module or group contains too many elements.

**Motivation:** Especially for large modules, groups should be used to add logical structure to the module and enhance readability. If a group reaches a critical size, it can be structured further by subgroups.

**Options:** A maximum allowable number of elements per module and per group could be specified. Additionally, a group for each kind of element (message types, port types, components, templates, functions, altsteps, test cases) could be demanded.

**Related action(s):** *Group Fragments*

**Example:** In listing 4.30, there are at least 11 module definitions. If the maximum number of elements for this module is exceeded, it makes sense to group the type and template definitions.

```
module UnsufficientGrouping {
  type record myRecordType1 { //...
  }

  type record myRecordType2 { //...
  }

  type record myRecordType3 { //...
  }

  type record myRecordType4 { //...
  }

  type record myRecordType5 { //...
  }

  template myRecordType1 myTemplate1 := { //...
  }

  template myRecordType1 myTemplate2 := { //...
  }

  template myRecordType1 myTemplate3 := { //...
  }

  template myRecordType1 myTemplate4 := { //...
  }

  template myRecordType1 myTemplate5 := { //...
  }

  function f1() { //...
  }

  // more declarations here...
}
```

*Listing 4.30: Insufficient Grouping*

### 4.8.5 Bad Comment Rate

**Description:** The comment rate (number of comments per line) is too high or too low.

**Motivation:** One the one hand, a low comment rate could be considered a smell, because the code might not be easy to understand. On the other hand, a too high comment rate could indicate a high code complexity and therefore be considered a smell, too.

**Related action(s):** If missing, a comment should be added. If comment rate is too high, the code readability should be improved (e.g. by using *Rename*, *Extract Function* and others).

**Example:** The example from ***Long Statement Block*** (4.4.1), shown in listing 4.16 does not contain any comments, although the code is not quite self-explanatory.

### 4.8.6 Bad Documentation Comment

**Description:** A documentation comment does not conform to its format.

**Motivation:** Documentation comments like T3Doc [17] need to conform to certain formatting rules and appear at certain positions in the source code.

**Related action(s):** Modify the documentation comment to conform to its format.

**Example:** Listing 4.31 shows a bad documentation comment (lines 1–5). The identifier of the parameter is misspelled and does not correspond to the formal parameter declared by the function.

```
1 /**
2  * This function does something.
3  * @param imParam
4  *   a very important parameter
5  */
6 function exampleFunction(integer inParam) {
7   //...
8 }
```

*Listing 4.31: Bad Documentation Comment*

## 4.9 Data Flow Anomalies

Data flow anomalies are located on the edge between bugs and code smells. They deal with definition and usage of variables of a behavioral entity. Such variables can be defined (D), read (R) and undefined (U). There is not necessarily a behavior-preserving refactoring to remove a data flow anomaly. For example, a DD data flow anomaly (i.e. a variable that is defined and defined again without being read in the meantime) can be removed without changing the behavior simply by removing the first definition. On the other hand, the smell might indicate that a read access is missing and therefor might require a change in the behavior. Nevertheless this section has been included in the catalog for the sake of completeness.

The smells ***Unused Definition*** (4.2.3), if referring to local variables, and ***Unused Parameter*** (4.3.1) can be interpreted as data flow anomalies as well.

### 4.9.1 Missing Variable Definition

**Derived from:** [52, p. 88]

**Description:** A variable or out parameter is read before its value has been defined. (This smell is also known as UR data flow anomaly.)

**Motivation:** Access to undefined variables might result in unpredictable behavior.

**Related action(s):** Make sure that the variable gets a value before it is used or remove the usage of the variable.

**Example:** In listing 4.32, function *f* has two parameters, *i* and *j*, and two local variables, *k* and *l*. Both **out**-parameter *j* and variable *l* are undefined in the assignment expression at the end of *f* (line 3).

```
1 function f(in integer i, out integer j) {
2   var integer k := 1, l;
3   j := i + j + k + l;
4 }
```

*Listing 4.32: Missing Variable Definition*

### 4.9.2 Unused Variable Definition

**Derived from:** [52, p. 88]

**Description:** A defined variable or in parameter is not read before it becomes undefined. (This smell is also known as DU data flow anomaly.)

**Motivation:** An assignment which is not used later on only increases code complexity unnecessarily.

**Related action(s):** Remove the assignment or insert a statement which makes use of the variable.

**Example:** In listing 4.33, function *f* has two parameters, *i* and *j*, and a local variable *k*. Both *i* and *k* are defined (lines 1, 2) but never read.

```
1 function f(in integer i, out integer j) {
2   var integer k := 1;
3   j := 1;
4 }
```

*Listing 4.33: Unused Variable Definition*

### 4.9.3 Wasted Variable Definition

**Derived from:** [52, p. 88]

**Description:** A variable is defined and defined again before it is read. (This smell is also known as DD data flow anomaly.)

**Motivation:** An assignment which is not used later on only increases code complexity unnecessarily.

**Related action(s):** Remove the first assignment or insert a statement which makes use of the variable after the first assignment.

**Example:** In listing 4.34, a value is assigned to local variable *k* (line 2). Before *k* is read, another value from parameter *i* is assigned to *k* (line 3).

```
1 function f(in integer i, out integer j) {
2   var integer k := 1;
3   k := i;
4   j := k;
5 }
```

*Listing 4.34: Wasted Variable Definition*

## 4.10 Miscellaneous

This category contains smells which simply do not fit into any other category.

### 4.10.1 Name-clashing Import

**Derived from:** [61] (Motivation for *Prefix Imported Declarations*)

**Description:** An imported element causes a name clash with a declaration in the importing module or another imported element.

**Motivation:** To avoid name clashes, references to imported declarations should be prefixed.

**Related action(s):** *Prefix Imported Declarations*

**Example:** In listing 4.35, module *Bar* imports a constant *MY_CONST* from a module *Foo* (line 6) and declares a constant with the same name (line 8). Function *f* has a reference to *MY_CONST* (line 11). This reference should be clarified to either *Bar.MY_CONST* (which would be behavior-preserving) or to *Foo.MY_CONST*.

```
 1 module Foo {
 2   const charstring MY_CONST := "foo";
 3 }
 4
 5 module Bar {
 6   import from Foo all;
 7
 8   const charstring MY_CONST := "bar";
 9
10   function f(in charstring s) return boolean {
11     if (MY_CONST == s) {
12       return true;
13     }
14     return false;
15   }
16 }
```

*Listing 4.35: Name-clashing Import*

### 4.10.2 Over-specific Runs On

**Derived from:** [61] (Motivation for *Generalize Runs On*)

**Description:** A behavioral entity (function, test case or altstep) is declared to run on a component, but uses only elements of this component's super-component or no elements of the component at all.

**Motivation:** To increase reuse potential and decrease complexity, behavioral entities should be kept as universal as possible. They should only run on components whose elements they make use of.

**Related action(s):** *Generalize Runs On*

**Example:** Component *ExtendedComponentType* extends another component *BaseComponentType* (listing 4.36, line 5). Function *f* runs on *ExtendedComponentType* (line 10), but uses only port *pOut* (line 12) which is declared in *BaseComponentType* (line 2). Hence, *f* should be generalized to running on *BaseComponentType*.

```
type component BaseComponentType {
  port OutPort pOut;
}

type component ExtendedComponentType extends BaseComponentType {
  port InPort pIn;
  timer t;
}

function f(charstring aMessage) runs on ExtendedComponentType {
  if (checkSomething()) {
    pOut.sent(aMessage);
  }
}
```

*Listing 4.36: Over-specific Runs On*

### 4.10.3 Goto

**Description:** A **goto** statement is used.

**Motivation:** The use of **goto** statements is inadvisable and should be avoided.

**Related action(s):** Change the structure of the code in a way that gotos are superfluous and remove them. (Note that this transformation can be complex and can vary from case to case.)

**Example:** Function *f* contains two **goto** statements, one of them jumping backwards to label *L1* (line 6), the other one jumping forwards to label *L2* (line 10). The first statement could be replaced by a **while** loop, the second statement could be replaced by an **if** construct with inverted condition (listing 4.37).

```
1  function f(integer i) runs on ExampleComponent {
2    var integer MyVar := i;
3    label L1;
4    MyVar := 2 * MyVar;
5    if (MyVar < 2000) {
6      goto L1;
7    }
8    MyVar2 := f2(MyVar);
9    if (MyVar2 > MyVar) {
10     goto L2;
11   }
12   p.send(MyVar);
13   p.receive -> value MyVar2;
14   label L2;
15 }
```

*Listing 4.37: Goto*

# 5 Code Smell Detection for TRex

As discussed in section 3.2.2, automated code smell detection has many advantages. An automated detection for a subset of smells presented in chapter 4 has been implemented as plug-in for TRex. It builds on the TPTP Static Analysis Framework and implements the code smell detections as analysis rules. With this automation it is possible to detect code smells in TTCN-3 code, display the code parts in a TTCN-3 editor and – if a quick-fix is supplied for the rule – remove the code smell by means of refactoring.

This chapter describes the key features of the code smell detection for TRex. It gives a general overview of how the analysis works and describes a library of methods used by the smell detection rules. Additionally, some of the smell detection rules are explained in detail. Finally, the quick-fix functionality and tests are explained.

## 5.1 General Overview

As plug-in for TRex, the smell detection has the same requirements (Eclipse version 3.2 or greater, Java SE version 5 or greater) plus the dependency on the Static Analysis Framework. It was developed and tested with TPTP version 4.2.0. The *Eclipse Modeling Framework* (EMF) [14] is specified as prerequisite for TPTP, but is not required by the Static Analysis Framework plug-ins.

Figure 5.1 shows the architecture of the code smell detection for TRex. Like other parts of TRex it builds on core functionality provided by the core plug-in, especially the AST and the symbol table. In addition it extends the TPTP Static Analysis Framework by adding an analysis provider and rules for code smell detection.

Code smell detection for TRex consists of 3 plug-ins:

- The *de.ugoe.cs.swe.trex.patterndetection.core* plug-in bundles the actual analysis. It includes general infrastructure like the analysis provider and a rule class for each smell. Classes from this plug-in are separated into two packages:
  - Package *de.ugoe.cs.swe.trex.patterndetection* contains classes which extend the Static Analysis Framework and provides a framework for the actual smell detection, which is done by rules.

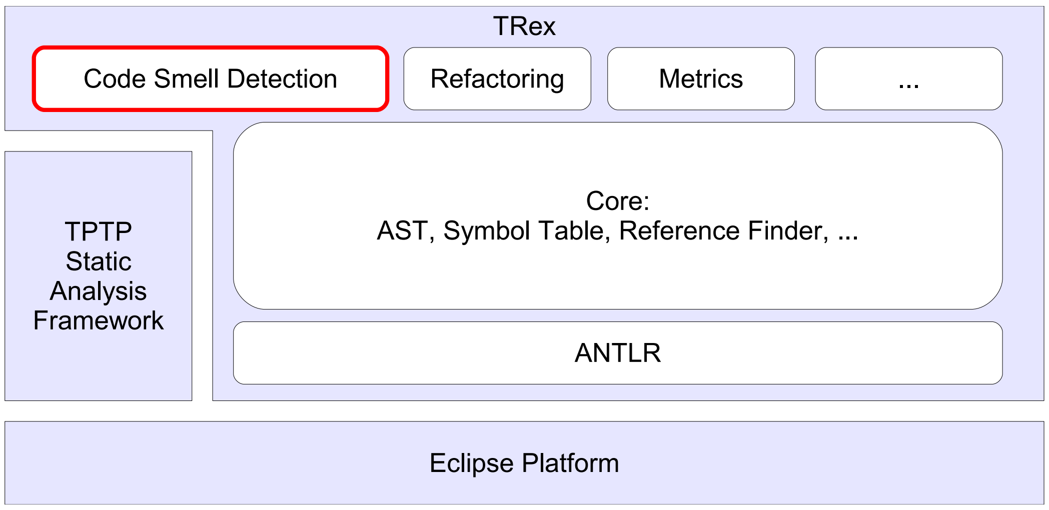

TRex
Code Smell Detection
Refactoring
Metrics
...
TPTP Static Analysis Framework
Core: AST, Symbol Table, Reference Finder, ...
ANTLR
Eclipse Platform

*Figure 5.1: Architecture of the smell detection for TRex*

- Package *de.ugoe.cs.swe.trex.patterndetection.rules* contains the actual rules together with an abstract base class for the rules and a class with static helper methods.

- The *de.ugoe.cs.swe.trex.patterndetection.ui* plug-in provides contributions to the UI like support for annotating and displaying code smells in the editor and quick-fix support. Its classes are divided into two packages:
  - Package *de.ugoe.cs.swe.trex.patterndetection.ui* contains an annotation provider which is responsible for the appearance of markers generated for each detected smell.
  - Package *de.ugoe.cs.swe.trex.patterndetection.ui.quickfix* contains some quick-fix implementations together with an abstract base class.

- The *de.ugoe.cs.swe.trex.patterndetection.test* plug-in provides JUnit plug-in tests for the smell detection rules. Classes belong to the following packages:
  - Package *de.ugoe.cs.swe.trex.patterndetection.tests* contains a test project class and a JUnit test suite class.
  - Package *de.ugoe.cs.swe.trex.patterndetection.rules.tests* contains JUnit test cases for all smell detection rules.

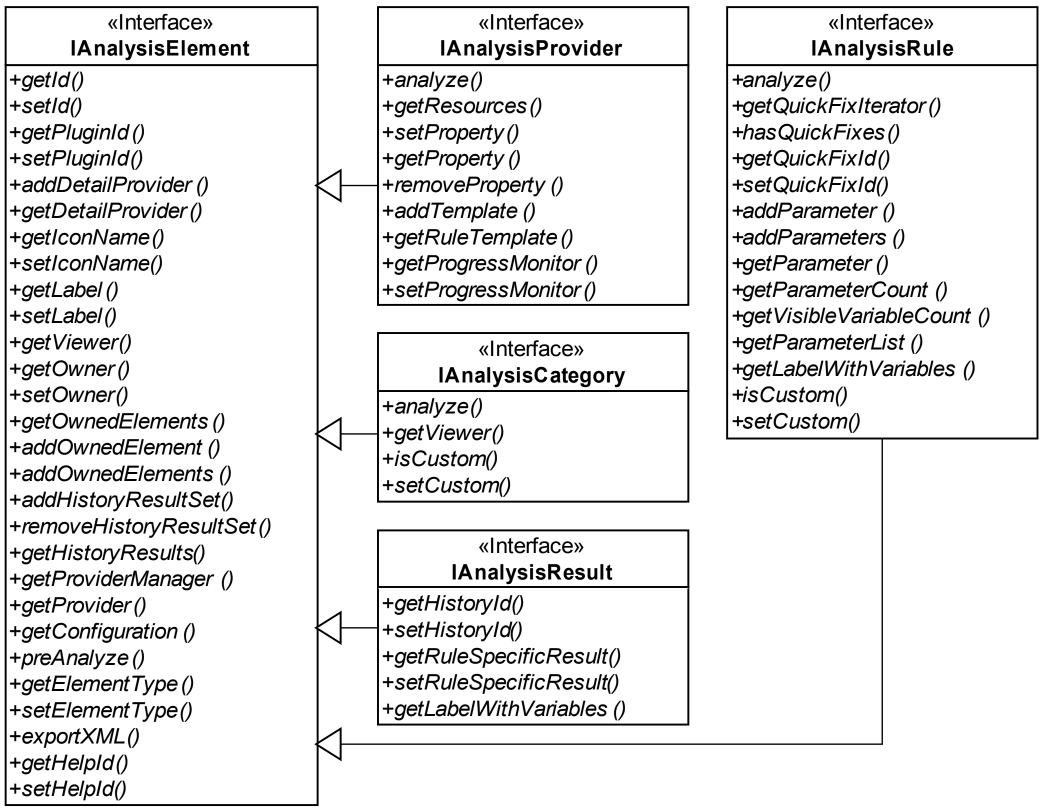


*Figure 5.2: Static Analysis Framework interface hierarchy*

### 5.1.1 Static Structure

Figure 5.2 shows the hierarchy of interfaces for elements that participate in the analysis process. *IAnalysisElement* is the base interface for all analysis elements. It defines characteristics that all elements share, for example the ability to own other analysis elements. *IAnalysisProvider* is the interface for providers. Providers own one or more categories, which implement *IAnalysisCategory*. *IAnalysisRule* defines the interface for rules which perform the actual analysis work. *IAnalysisResult* is the interface for analysis results. For all interfaces there exist abstract classes which can be used as base classes for own implementations; for *IAnalysisCategory* a default implementation is provided.

The static structure of the code smell detection core plug-in is illustrated in figure 5.3. Class *PatternDetectionProvider* implements the *IAnalysisProvider* interface by extending *AbstractAnalysisProvider*, which is included in the Static Analysis Framework as abstract base class for analysis providers. *PatternDetectionResource* is a wrapper for TTCN-3 files.

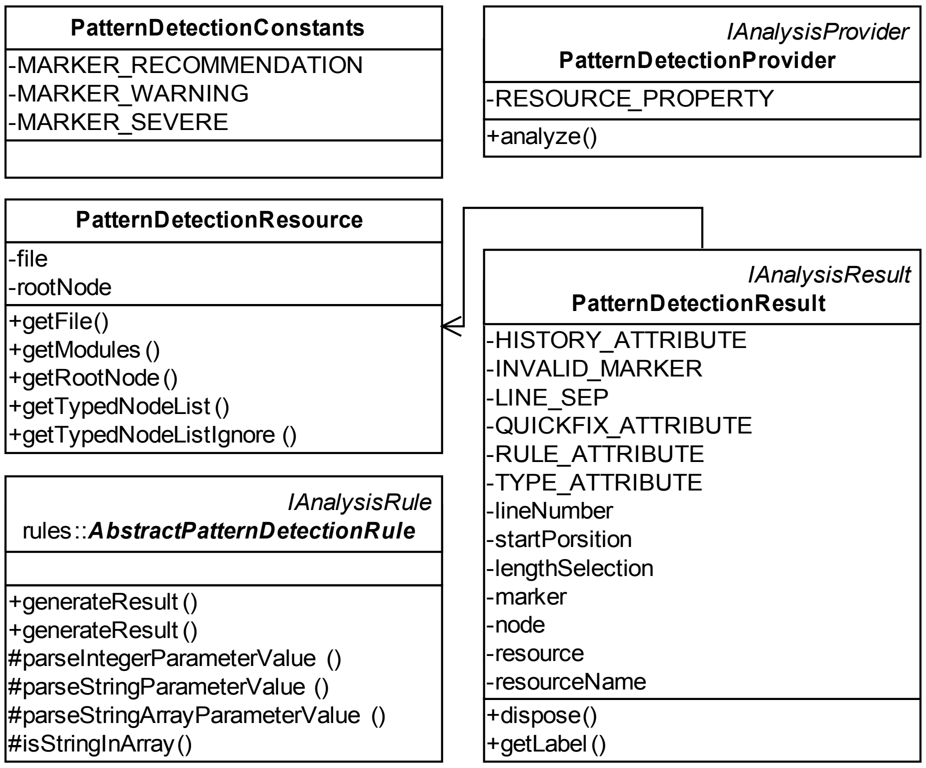


*Figure 5.3: Static structure of the smell detection*

It simplifies access to the corresponding AST and provides a method for extracting all nodes of a given type (e.g. all template definitions). *PatternDetectionResult* implements *IAnalysisResult* by extending *AbstractAnalysisResult*. An instance of this class is created each time a rule detects a smell. It is connected to the resource in which the smell was detected. The result stores all information about the smell instance, like offset and line number in the corresponding file, label (for the results view) and the affected AST node. Additionally it manages the creation and removal of markers in the corresponding resource. *PatternDetectionConstants* contains identifiers for different marker types. *AbstractPattern-DetectionRule* is included in the sub-package *rules* as base class for concrete rules. Rule implementations are explained in detail in section 5.3.

### 5.1.2 Analysis Sequence

Internally the Static Analysis Framework controlles the execution of analyses by an *Analysis-ProviderManager* class that manages all analysis providers in the system. When an analysis

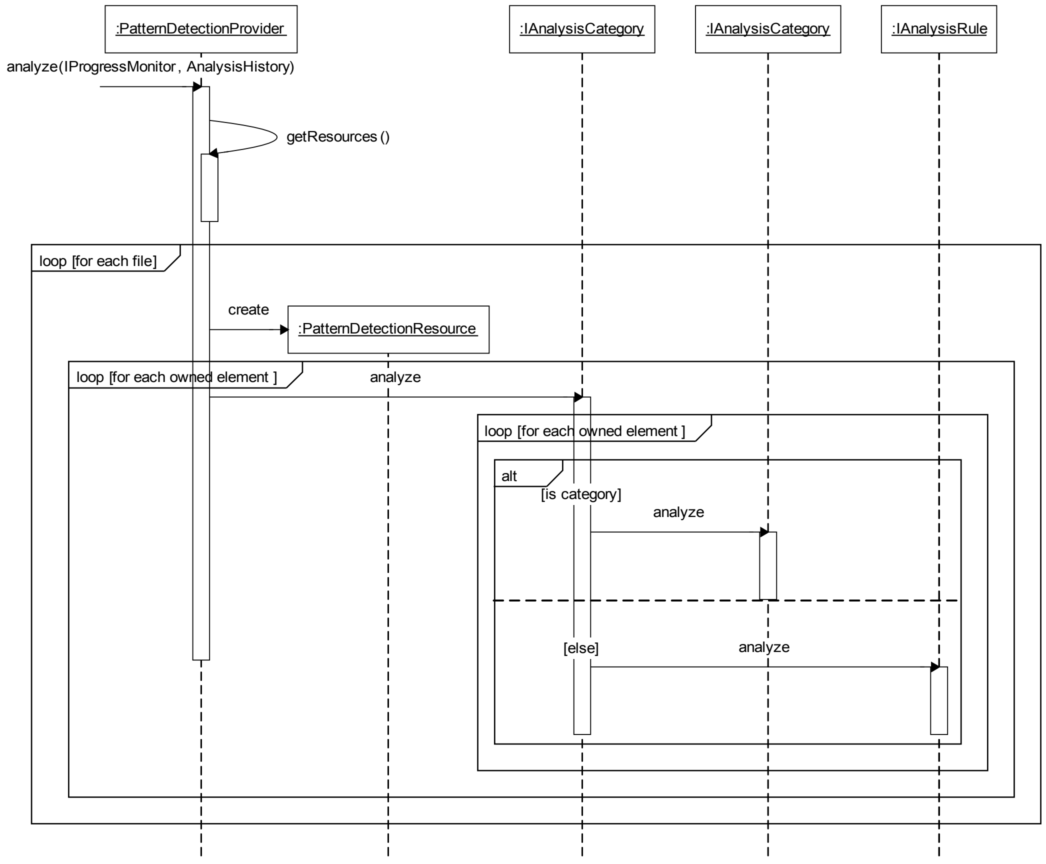

:PatternDetectionProvider
:IAnalysisCategory
:IAnalysisCategory
:IAnalysisRule
analyze(IProgressMonitor, AnalysisHistory)
getResources ()
loop [for each file]
create
:PatternDetectionResource
loop [for each owned element ]
analyze
loop [for each owned element ]
alt
[is category]
analyze
[else]
analyze

*Figure 5.4: Sequence diagram for the smell detection*

run is performed, a new analysis history instance is created. The provider manager starts an asynchronous job for each enabled provider and calls its *analyze* method. Each provider typically calls the *analyze* method of any of its enabled categories. In turn a category typically invokes the *analyze* method of any of its enabled rules. Rules then do some work and generate results. Results are not directly associated with a rule, but are stored in the analysis history.

Figure 5.4 shows the sequence for a call to the *analyze* method of the smell detection provider. The analysis history is passed in as parameter. The provider retrieves a list of workspace resources (i.e. TTCN-3 files) which are to be analyzed and iterates over this list. For each file a *PatternDetectionResource* instance is created and the *analyze* method for each active category owned by the provider is called. The code smell detection uses the default implementation for categories, which simply calls *analyze* on all active categories and rules owned by this category. Normally a rule then fetches the current *PatternDetectionResource* from the provider and does some sort of analysis on it.

### 5.1.3 Framework Extensions

Contributions made to the Static Analysis Framework (e.g. providers, categories and rules) are made public via extensions defined in the *plugin.xml* file of one of the smell detection plug-ins. Listing 5.1 shows a snippet from the file that defines the extension point for the provider. It specifies the provider class which implements the *IAnalysisProvider* interface, a unique identifier for the provider, a label that is used for the UI and the identifier of a viewer which is invoked when a result of this provider is double-clicked.

```xml
<extension point="org.eclipse.tptp.platform.analysis.core.analysisProvider">
  <analysisProvider
    class="de.ugoe.cs.swe.trex.patterndetection.PatternDetectionProvider"
    id="de.ugoe.cs.swe.trex.patterndetection.provider"
    label="TTCN-3 Code Review Provider"
    viewer="de.ugoe.cs.swe.trex.patterndetection.ui.PatternDetectionViewer"/>
</extension>
```

*Listing 5.1: Provider extension*

Categories are specified in a similar way. Listing 5.2 shows a shortened extract from the *plugin.xml* file. Each top-level category is connected to the provider by a *provider* attribute, sub-categories are connected to other categories (not shown in the listing). Again an identifier and a label are specified. The *DefaultAnalysisCategory* class used within the code smell detection as implementation for categories is a default implementation provided by the Static Analysis Framework.

```xml
<extension point="org.eclipse.tptp.platform.analysis.core.analysisCategory">
  <analysisCategory
    class="org.eclipse.tptp.platform.analysis.core.category.DefaultAnalysisCategory"
    id="de.ugoe.cs.swe.trex.patterndetection.category.duplicatedCode"
    label="Duplicated Code"
    provider="de.ugoe.cs.swe.trex.patterndetection.provider"/>
  <analysisCategory
    class="org.eclipse.tptp.platform.analysis.core.category.DefaultAnalysisCategory"
    id="de.ugoe.cs.swe.trex.patterndetection.category.references"
    label="References"
    provider="de.ugoe.cs.swe.trex.patterndetection.provider"/>
  <!-- all other categories defined here similarly -->
</extension>
```

*Listing 5.2: Category extensions*

Rules are defined alike. Listing 5.3 shows another shortened excerpt. Each rule shown is specified by the category it belongs to, the class that implements the *IAnalysisRule* interface, a unique identifier, a label used for the UI and a severity level. Rule parameters may be specified in a *ruleParameter* element. Additionally, each rule may provide a quick-fix identifier. Quick-fixes for this rule extend a separate extension point and refer to this identifier (not shown in the listing).

```
<extension
    point="org.eclipse.tptp.platform.analysis.core.analysisRule">
  <analysisRule
    category="de.ugoe.cs.swe.trex.patterndetection.category.magicValue"
    class="de.ugoe.cs.swe.trex.patterndetection.rules.MagicNumberRule"
    id="de.ugoe.cs.swe.trex.patterndetection.rules.MagicNumberRule"
    label="Magic Number"
    severity="1">
    <ruleParameter
      label="Exclude (comma-separated):"
      name="de.ugoe.cs.swe.trex.patterndetection.rules.MagicNumberRule.exclude"
      style="text"
      type="string"
      value="0,1"/>
  </analysisRule>
  <analysisRule
    category="de.ugoe.cs.swe.trex.patterndetection.category.singularReference"
    class="de.ugoe.cs.swe.trex.patterndetection.rules.SingularTemplateReferenceRule"
    id="de.ugoe.cs.swe.trex.patterndetection.rules.SingularTemplateReferenceRule"
    label="Singular Template Reference"
    severity="0">
    <quickfix
      id="de.ugoe.cs.swe.trex.patterndetection.rules.SingularTemplateReferenceRule.
          quickfix"/>
  </analysisRule>
  <!-- all other rules defined here similarly -->
```

*Listing 5.3: Rule extensions*

## 5.2 A library for analyzing ASTs

All rules investigate AST nodes in order to find a smell instance. Class *PatternDetection-Resource* offers methods to access the root node of the AST of a file and to obtain all nodes of a given type. In addition, there are a number of methods which are used by the rules while processing the AST. These methods partially belong to the *AST* class provided by ANTLR, to *ASTUtil*, a helper class from the TRex core plug-in and to a *Util* class included in the smell detection core plug-in. Together they form a library for examining the AST.

The TRex AST is a tree representation of the syntactic structure of a TTCN-3 file. Figure 5.5 shows a schematic view of such a tree. Logically each node can have multiple children (shown as light-grey arrows in the figure); physically the tree is realized as binary tree (black arrows). The root node of the tree represents the TTCN-3 file. A node can be connected to another node as child or sibling. The child axis can be navigated in both directions, the sibling axis only in forward direction. Each node is of a specific type and has a text attribute. Furthermore, nodes contain their start and end position as offset from the top of the file. This is why the class for nodes in the TRex AST is named *LocationAST*.

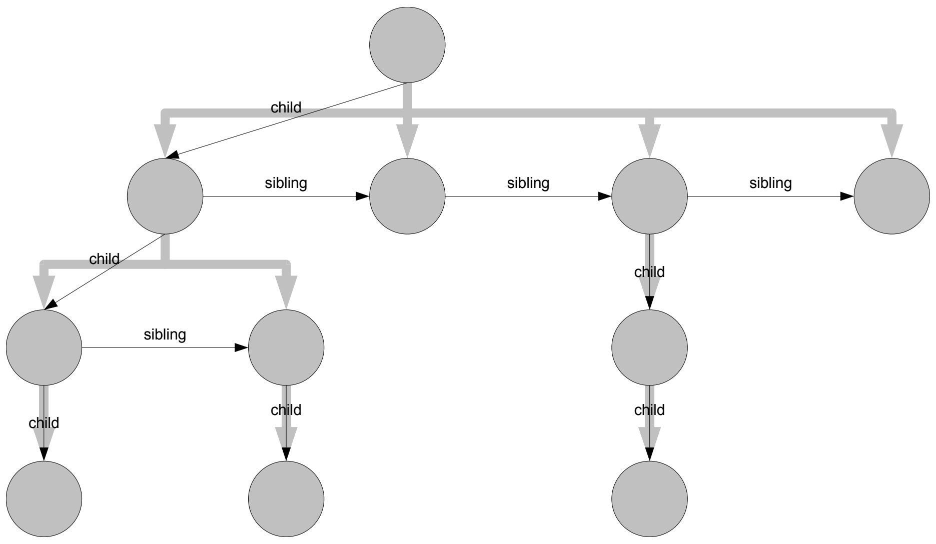


*Figure 5.5: Structure of the TRex AST*

The library contains methods for basic navigation in the AST along the child and sibling axes:

- *getFirstChild* returns the first, *getNthChild* the $n$th node along the child axis.
- *getParent* returns the first, *getNthParent* the $n$th node backwards along the child axis.
- *getNextSibling* returns the first node along the sibling axis.
- *getDescendantNode* descents the child axis until a leaf node (i.e. a node without children and siblings) is reached and returns this node.

The following methods are used for finding nodes of a given type:

- *findChild* returns the first node of a given type along the child axis.
- *resolveParentsUntilType* returns the first node of a given type, but navigates backwards along the child axis.
- *findSibling* returns the first node of a given type along the sibling axis.
- *findFirstDescendant* starts from a given node and traverses the subtree (first child axis, then sibling axis). It returns the first node of a given type.
- *findTypeNodes* starts from a given node and traverses the subtree, returning all nodes of a given type.

- *findTypeNodesIgnore* starts from a given node and traverses the subtree, returning all nodes of a given type. If a node of a given ignore-type is found, this node and all its children are ignored.

These methods are used for comparing nodes, subtrees and lists of subtrees:

- *equals* compares single nodes to each other. Two nodes are equal if they are of the same type and contain the same text. (Two *LocationAST* nodes are equal if they additionally have the same start and end position.)
- *equalsTree* and *equalsTreePartial* compare (sub)trees to each other using the *equals* method. *equalsTree* returns true if the two trees are equal, *equalsTreePartial* returns true if one tree is equal to a subtree of the second tree.
- *equalsList* and *equalsListPartial* does the same for lists of trees.
- *findAll* walks the tree looking for all exact subtree matches and returns them.
- *compareTo* compares the text of two nodes to each other.

Other than these basic methods there is a huge number of more specific methods for navigating to a target node from a source node (like finding the template body from a template definition) or finding a set of nodes from a source node (like finding all parameter definitions from a template definition).

The smell detection rules are implemented using the library methods summarized in this section. Another approach for investigating ANTLR generated ASTs is by the use of *tree walkers*. A tree walker is implemented by a tree grammar, which is basically a tree specification enriched with actions. These actions are executed on tree traversal. Other parts of TRex use the tree waker approach, e.g. the pretty printer, the symbol table generation and the control and data flow analyses. However, a tree grammar for each smell detection rule would be overly intricate. Most rules only work on small fragments of the AST anyway, and some are simple enough to be implemented in a few lines of Java code. Additionally, changes in the AST structure created by the parser would result in extensive changes in all of the tree walkers. Another popular alternative for processing trees is the *visitor pattern* [29]. Yet it is not easily applicable for the TRex ASTs, because all nodes of the tree are of the same type (i.e. Java class) while the syntactic type of nodes is stored as integer attribute.

## 5.3 Rules for Code Smell Detection

Rules are the "workhorses" of the smell detection. They perform the analysis on the AST and generate results if an instance of a smell is found. A total number of 11 smell detection rules for TTCN-3 has been implemented, 5 of them reusing code developed out of the scope of this thesis as analysis for template-related issues [62]. Their class names are as follows:

- *ActivationAsymmetryRule* detects instances of ***Activation Asymmetry*** (4.5.1). Test cases can optionally be excluded from the analysis.
- *ConstantActualTemplateParameterRule*[1] finds template parameter instances of ***Constant Actual Parameter Value*** (4.3.2).
- *DuplicateAltBranchesRule* searches for ***Duplicate Alt Branches*** (4.1.2).
- *FullyParameterizedTemplateRule*[1] detects instances of ***Fully-Parameterized Template*** (4.3.3).
- *MagicNumberRule* finds numeric ***Magic Values*** (4.8.1). It can be parameterized by a list of values which are to be excluded.
- *MagicStringRule* searches for ***Magic Values*** (4.8.1) of any string type.
- *ShortTemplateRule*[1] detects instances of ***Short Template*** (4.4.5). A boundary value for the length of the template (in characters) can be specified.
- *SingularComponentVCTReferenceRule* traces instances of ***Singular Component Variable/Constant/Timer Reference*** (4.2.2).
- *SingularTemplateReferenceRule*[1] finds instances of ***Singular Template Reference*** (4.2.1).
- *UnusedLocalDefinitionRule* detects instances of ***Unused Definition*** (4.2.3) for local variable, constant, timer and template definitions.
- *UnusedTemplateDefinitionRule*[1] finds all global template instances of ***Unused Definition*** (4.2.3).

Below, three of the rules are explained more detailled. They were chosen to work as differently as possible and should give a good impression of how the smell detection is realized.

### 5.3.1 The Magic Number Rule

The Magic Number rule is part of the implementation for the smell ***Magic Values*** (4.8.1), together with a Magic String rule. It searches for numeric literals used outside of constant definitions. Values can be excluded from the search by a rule parameter. By default, `0` and `1` are ignored.

[1]This rule utilizes code not developed as part of this thesis.

Listing 5.4 shows the rule class. It extends *AbstractPatternDetectionRule* which defines some attributes and methods common to all rules. The Magic Number rule obtains the *PatternDetectionResource* instance from the provider (lines 11–13). Values that are not to be detected as Magic Numbers are parsed from a string parameter that is made available by the Static Analysis Framework (line 14). The rule then obtains a list of all AST nodes that represent numbers, i.e. integer and float values (line 15). For each of these nodes the value is determined and compared to the exclude values (line 20). If the value is different, the rule checks whether the number is part of a constant definition (lines 24–25). If not, a result is generated which marks the number as Magic Number (line 29). For this purpose, the affected node is passed into the result.

```
1  public class MagicNumberRule extends AbstractPatternDetectionRule {
2
3      private static final String PARAMETER_EXCLUDE =
4          "de.ugoe.cs.swe.trex.patterndetection.rules.MagicNumberRule.exclude";
5
6      private final int[] NODE_TYPES = new int[] {
7              TTCN3LexerTokenTypes.IntegerValue,
8              TTCN3LexerTokenTypes.FloatValue };
9
10     public void analyze(AnalysisHistory history) {
11         PatternDetectionResource resource = (PatternDetectionResource) getProvider()
12                 .getProperty(history.getHistoryId(),
13                         PatternDetectionProvider.RESOURCE_PROPERTY);
14         String[] exclude = parseStringArrayParameterValue(PARAMETER_EXCLUDE);
15         List<LocationAST> list = resource.getTypedNodeList(NODE_TYPES);
16         for (LocationAST node : list) {
17             // ignore excluded values
18             LocationAST valueNode = node.getFirstChild();
19             String value = valueNode.getText();
20             if (isStringInArray(value, exclude)) {
21                 continue;
22             }
23             // ignore const definitions
24             if (LocationAST.resolveParentsUntilType(node,
25                     TTCN3LexerTokenTypes.ConstDef) != null) {
26                 continue;
27             }
28             // not a const definition? report smell!
29             generateResult(resource, history.getHistoryId(), node);
30         }
31     }
32 }
```

*Listing 5.4: Magic Number rule*

### 5.3.2 The Duplicate Alt Branches Rule

Detection of the smell ***Duplicate Alt Branches*** (4.1.2) is implemented by a Duplicate Alt Branches rule. It uses a simple subtree comparison algorithm between branches of all alternative behavior constructs. More sophisticated approaches (like the ones presented in section 3.3.4) could be used instead. Furthermore the current implementation only detects duplicates within the same file. However, it would be possible to extend the search on all files of the project or the whole workspace.

A branch in an alternative behavior construct consists of three parts: the guard expression in square brackets, the guard operation (e.g. a **timeout** or **receive** statement) and a statement block in curly braces. Together they form a guard statement. For duplication only guard operation and statement block are considered, because they can be extracted into a common altstep, and the invocation of this altstep can still be guarded.

The rule is presented in listing 5.5. It obtains all guard statements from the AST (lines 11–12). A list with possible clones is filled with all guard statements, and an empty list with all clones found is prepared (lines 13–16). For each guard statement it is first checked whether the statement has already been identified as clone (line 20). If not, the list of all possible clones is investigated for clones of this guard statement (line 24). The *findClones* method checks for duplicates of guard operation and statement block (lines 37–49). Each duplicate found is added to the list of duplicates and excluded from further analysis (line 27). If there was at least one duplicate, a result is generated and the first branch from the set of duplicates is marked (lines 29–32).

```
1  public class DuplicateAltBranchesRule extends AbstractPatternDetectionRule {
2
3      private final int[] NODE_TYPES = new int[] {
4          TTCN3LexerTokenTypes.GuardStatement
5    };
6
7      public void analyze(AnalysisHistory history) {
8          PatternDetectionResource resource = (PatternDetectionResource) getProvider()
9                  .getProperty(history.getHistoryId(),
10                         PatternDetectionProvider.RESOURCE_PROPERTY);
11         List<LocationAST> guardStatements = resource
12                 .getTypedNodeList(NODE_TYPES);
13         List<LocationAST> foundClones = new ArrayList<LocationAST>();
14         List<LocationAST> possibleClones = new ArrayList<LocationAST>();
15         // start with all guard statements
16         possibleClones.addAll(guardStatements);
17         for (LocationAST guardStatement : guardStatements) {
18             possibleClones.remove(guardStatement);
19             // ignore guard statements that have already been identified as clones
20             if (foundClones.contains(guardStatement))
21                 continue;
22             // navigate to guard operation
23           LocationAST guardOp = guardStatement.getFirstChild().getNextSibling();
24             List<LocationAST> clones = findClones(guardOp, possibleClones);
25             if (!clones.isEmpty()) {
26         // found clones? add them to the list!
27                 foundClones.addAll(clones);
28                 // report smell!
29                 generateResult(resource, history.getHistoryId(),
30                         guardStatement, guardOp.getLine(), guardOp
31                                 .getOffset(), guardStatement
```

```
                                        .getEndOffset());
            }
        }
    }

    private List<LocationAST> findClones(LocationAST guardOp,
            List<LocationAST> possibleClones) {
        List<LocationAST> clones = new ArrayList<LocationAST>();
        for (LocationAST guardStatement : possibleClones) {
            LocationAST possibleClonedGuardOp = guardStatement
                    .getFirstChild().getNextSibling();
            if (possibleClonedGuardOp.getType() == TTCN3LexerTokenTypes.GuardOp) {
                if (guardOp.equalsList(possibleClonedGuardOp))
                    clones.add(guardStatement);
            }
        }
        return clones;
    }
}
```

*Listing 5.5: Duplicate Alt Branches rule*

## 5.3.3 The Unused Local Definition Rule

The Unused Local Definition rule detects local ***Unused Definition*** (4.2.3) instances. A local definition which is not used can be safely removed, because it cannot be referenced from outside the declaring unit. In contrast, a global definition can have no known references, but be in use by a module which is not included in the analysis. The Unused Local Definition rule is connected to a quick-fix, which simply removes the local definition.

In listing 5.6 the rule class is shown. For each local variable, constant, timer and template definition the identifier node is determined (lines 14–15). If there are no references to this identifier, a result is generated (lines 18–22). The *isUnreferenced* method (line 18) makes use of the reference finder provided by the TRex core plug-in.

```
1  public class UnusedLocalDefinitionRule extends AbstractPatternDetectionRule {
2
3      private final int[] NODE_TYPES = new int[] {
4              TTCN3LexerTokenTypes.FunctionLocalDef,
5              TTCN3LexerTokenTypes.FunctionLocalInst };
6
7      public void analyze(AnalysisHistory history) {
8          PatternDetectionResource resource = (PatternDetectionResource) getProvider()
9                  .getProperty(history.getHistoryId(),
10                         PatternDetectionProvider.RESOURCE_PROPERTY);
11         List<LocationAST> list = resource.getTypedNodeList(NODE_TYPES);
12         for (LocationAST node : list) {
13             // find IDENTIFIER(s) of this variable/constant/timer/template
14             List<LocationAST> identifiers = Util
15                     .findIdentifiersFromLocalDefOrInst(node);
16             for (LocationAST identifier : identifiers) {
17                 // find all references
18                 if (Util.isUnreferenced(identifier)) {
```

```
                        // no references found? report smell!
                        generateResult(resource, history
                                .getHistoryId(), identifier);
                    }
                }
            }
        }
}
```

*Listing 5.6: Unused Local Definition rule*

## 5.4 Quick Fix Support

Some of the rules are connected to a corresponding quick-fix. The quick-fixes are part of the code smell detection UI plug-in, because they build on other UI components like the editor or refactoring wizards.

Figure 5.6 shows the classes and interfaces that make up the static structure of the quick-fixes. The abstract base class *AbstractPatternDetectionQuickFix* implements the *quickfix* method of interface *IAnalysisQuickFix*. It makes sure that affected resources can be edited and cares for removal of fixed results from the analysis history and for re-execution of analysis on modified resources. The actual quick-fix is delegated to the protected method *fixPatternDetectionResult* which is implemented by sub-classes.

The rules currently supporting the quick-fix feature are *SingularTemplateReference-Rule*, *UnusedLocalDefinitionRule* and *UnusedTemplateDefinitionRule*. Both *UnusedLocal-DefinitionQuickFix* and *UnusedTemplateDefinitionQuickFix* currently simply remove the unused definition.

*SingularTemplateDefinitionQuickFix* makes use of an existing refactoring implementation, the *Inline Template* refactoring. Listing 5.7 shows the quick-fix class. The template definition node is retrieved from the result and passed into a new refactoring processor instance (lines 6–7). The processor is used to create a refactoring (lines 8–9) and a refactoring wizard

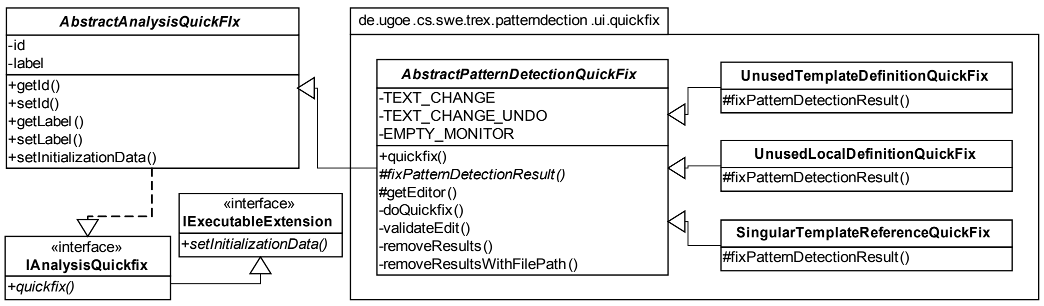

AbstractAnalysisQuickFIx
-id
-label
+getId()
+setId()
+getLabel ()
+setLabel()
+setInitializationData()
«interface»
IAnalysisQuickfix
+quickfix()
«interface»
IExecutableExtension
+setInitializationData()
de.ugoe .cs.swe.trex.patterndection .ui.quickfix
AbstractPatternDetectionQuickFix
-TEXT_CHANGE
-TEXT_CHANGE_UNDO
-EMPTY_MONITOR
+quickfix()
#fixPatternDetectionResult ()
#getEditor ()
-doQuickfix ()
-validateEdit ()
-removeResults ()
-removeResultsWithFilePath ()
UnusedTemplateDefinitionQuickFix
#fixPatternDetectionResult()
UnusedLocalDefinitionQuickFix
#fixPatternDetectionResult()
SingularTemplateReferenceQuickFix
#fixPatternDetectionResult()

*Figure 5.6: Static structure of the quick-fix feature*

(lines 10–13). The wizard is started (lines 14–16) and presents the suggested refactoring to the user.

```
1  public class SingularTemplateReferenceQuickFix extends
2          AbstractPatternDetectionQuickFix {
3
4      protected void fixPatternDetectionResult(PatternDetectionResult result,
5              IFile file) throws Exception {
6          TTCN3InlineTemplateProcessor processor =
7                  new TTCN3InlineTemplateProcessor(result.getNode());
8          TTCN3InlineTemplateRefactoring refactoring =
9                  new TTCN3InlineTemplateRefactoring(processor);
10         TTCN3InlineTemplateRefactoringWizard wizard =
11                 new TTCN3InlineTemplateRefactoringWizard(
12                         refactoring,
13                                 RefactoringWizard.WIZARD_BASED_USER_INTERFACE);
14         RefactoringWizardOpenOperation openOperation =
15                 new RefactoringWizardOpenOperation(wizard);
16         openOperation.run(Display.getCurrent().getActiveShell(), "");
17     }
18 }
```

*Listing 5.7: Singular Template Reference quick-fix*

## 5.5 JUnit Tests

For all of the rules implemented there exist JUnit tests [28]. They reside in a separate tests plug-in. Like other TRex tests, the tests are run in another Eclipse instance that is usually started from the instance which is used to develop TRex by selecting "Run As..." and "JUnit Plug-In Test".

The test instance workspace is created from a resource directory that is included in the test plug-in. It contains a set of TTCN-3 files which are analyzed by the rule that is tested. Additionally the plug-in contains a launch folder with a launch configuration for each smell rule.

A launch configuration is a data structure created by the Eclipse Platform launch mechanism each time a launch is performed. Because the Static Analysis Framework uses the launch mechanism, a launch is created each time an analysis run is performed. Launches are stored in files in a metadata subfolder of the workspace folder. The tests make use of these launch configuration files. For each rule the corresponding test simply uses a configuration that launches an analysis in which only the rule that is about to be tested is selected. The launch file is copied to the test instance workspace from where it can be loaded and executed.

Figure 5.7 shows the main classes involved in the testing. A concrete test case creates a *PatternDetectionTestProject*. It is a subclass of *TTCN3TestProject* and therefore inherits its functionality to setup the test instance workspace. When created by the test case, it copies the relevant launch configuration and TTCN-3 files to the test instance workspace

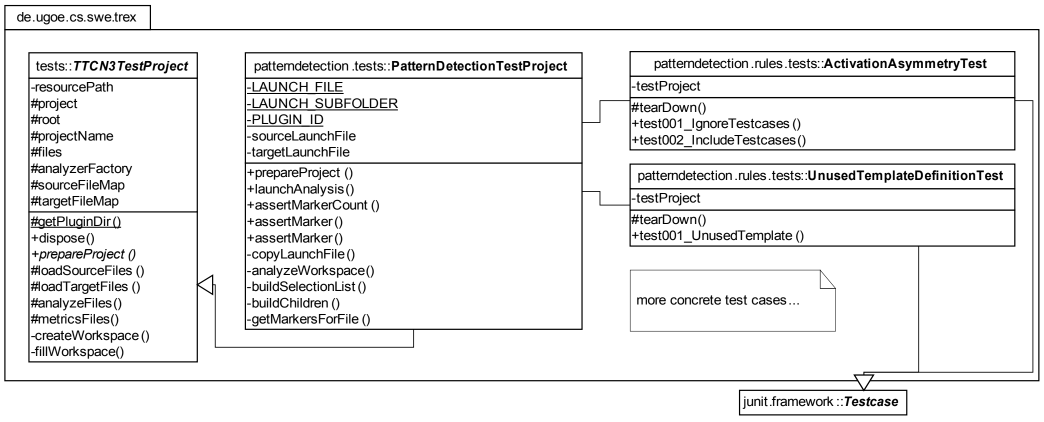


*Figure 5.7: Static structure of the tests*

and invokes parser and analyzer for creation of AST and symbol table. Afterwards the code smell detection is started by launching the copied analysis configuration. The test case then usually asserts the number of markers generated by the rule and their type and position in the documents.

Analysis results are checked by markers that have been generated by the rule. The smell detection core plug-in declares three different types of markers for the three different severity levels *recommendation*, *warning* and *error*. The tests are able to check the type of a marker as well as its position in the document.

Listing 5.8 shows the *assertMarker* method of *PatternDetectionTestProject*. It checks for the presence of a marker in the given line number of the specified file. Optionally the type and a message of the marker can be specified as well. First of all, an array of markers is retrieved from the file (line 3). For each marker the line number is compared to the given line number, and if they are equal and no type is specified, the method returns without failing (lines 11–16). Only if a type is specified, it is compared to the type of the marker, and if they are equal and no message is specified, the method returns (lines 17–21). Again, if a message is specified, it is compared to the marker's message, and the method returns if they are equal (lines 22–25). If all markers have been checked and the method did not return, no appropriate marker could be found, and the assertion fails (lines 30–32).

```
1     public void assertMarker(String markerFile, int lineNumber, String type,
2             String message) throws Exception {
3         IMarker[] markers = getMarkersForFile(markerFile);
4         for (int i = 0; i < markers.length; i++) {
5             Integer attributeLineNumber = (Integer) markers[i]
6                     .getAttribute(IMarker.LINE_NUMBER);
7             String attributeType = markers[i].getType();
8             String attributeMessage = (String) markers[i]
9                     .getAttribute(IMarker.MESSAGE);
```

```
            // check line number
            if (lineNumber == attributeLineNumber.intValue()) {
                // correct line number!
                if (type == null) {
                    // ignore type and message
                    return;
                }
                if (type.equals(attributeType)) {
                    if (message == null) {
                        // ignore message
                        return;
                    }
                    if (message.equals(attributeMessage)) {
                        // correct message!
                        return;
                    }
                }
            }
        }
        // no marker found
        Assert.fail("There is no marker in file "   + markerFile
            + ", line " + lineNumber + ", of type " + type
            + ", with message '" + message  + "'."));
    }
```

*Listing 5.8: Marker assertion*

# 6 Code Smells in Existing Test Suites

As an indication for their relevance, some of the smell detection implementations presented in chapter 5 have been applied to existing test suites, namely a test suite for the *Session Initiation Protocol* (SIP) protocol [22] and a preliminary version of a test suite for the *Internet Protocol Version 6* (IPv6) protocol [23]. Both test suites are published by the ETSI and may serve as reference for TTCN-3 tools.

The following smell detections have been run on the test suites:

- ***Magic Values*** (4.8.1): Only numeric literals (Magic Numbers) have been included, values `0` and `1` excluded. (See section 5.3.1 for details about this rule.)
- ***Activation Asymmetry*** (4.5.1): This rule has been applied both with and without consideration of test cases.
- ***Duplicate Alt Branches*** (4.1.2): Only duplicates in the same file have been considered. (See section 5.3.2 for details about this rule.)
- ***Singular Component Variable/Constant/Timer Reference*** (4.2.2): References in all analyzed files have been included.
- ***Unused Definition*** (4.2.3): Only local definitions have been considered. (See section 5.3.3 for details about this rule.)

The upper part of table 6.1 shows some size metrics for the SIP and IPv6 test suites in order to give a notion of their size. The lower part lists the number of smells found in the test suite. Especially numeric ***Magic Values*** (4.8.1) and instances of ***Activation Asymmetry*** (4.5.1) appear in great quantities. A fair number of ***Duplicate Alt Branches*** (4.1.2) and instances of ***Unused Definition*** (4.2.3) in local scope are found, too. In contrast, ***Singular Component Variable/Constant/Timer Reference*** (4.2.2) seems to be of little importance for these test suites.

In the remainder of this chapter, smells found in the SIP are investigated further. Results for the IPv6 test suite seem to have similar origins as the ones for the SIP test suites.

The high number of Magic Numbers results mainly from a frequent usage of numeric literals in expressions regarding timer settings. Further Magic Numbers are part of template definitions. Figure 6.1 shows examples for Magic Numbers defining timer values in the SIP test main module. Without further documentation the meaning of these values is not

| | SIP | IPv6 |
|---|---|---|
| Lines of code | 42397 | 46163 |
| Number of control parts | 1 | 3 |
| Number of functions | 785 | 643 |
| Number of test cases | 528 | 295 |
| Number of altsteps | 10 | 11 |
| Number of components | 2 | 10 |
| Magic Numbers (0,1 excluded) | 543 | 368 |
| Activation Asymmetries (test cases included) | 602 | 801 |
| Activation Asymmetries (test cases excluded) | 73 | 317 |
| Duplicate Alt Branches | 119 | 48 |
| Singular Component Variable/Constant/Timer References | 2 | 15 |
| Unused Local Definition | 50 | 156 |

*Table 6.1: Code Smells in existing test suites*

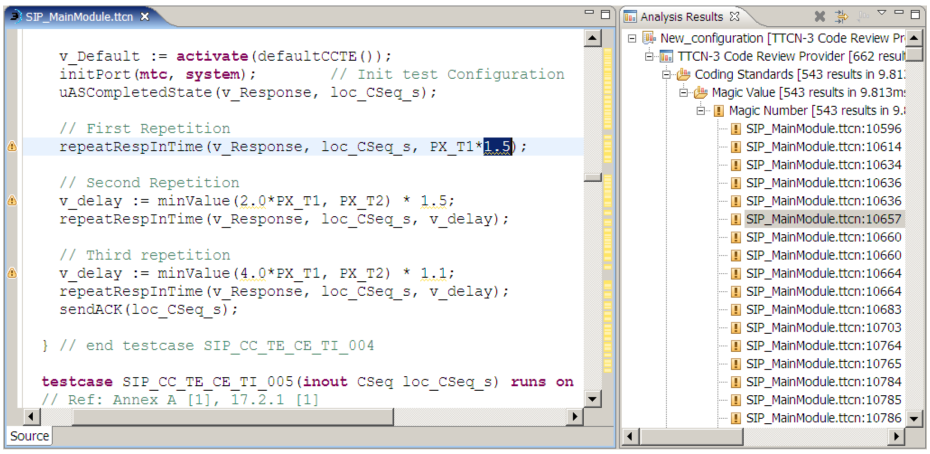
comprehensible. The use of constants with meaningful names could improve readability significantly. Hence, ***Magic Values*** (4.8.1) seem to be relevant.

The high number of Activation Asymmetry in the SIP test suite is caused by the fact that in many test cases a default is activated but not deactived. With termination of a test case all active defaults are deactivated anyway, so obviously the explicit deactivation is regarded to be superfluous. If test cases are excluded from the analysis, there still remains number of functions that activate defaults previous to an alternative behavior construct and miss deactivating it. Figure 6.2 shows an example of such a function. Determination of active defaults is made difficult by this kind of default handling. However, the smell ***Activation Asymmetry*** (4.5.1) remains partly a matter of taste.

Most of the Duplicate Alt Branches are duplicates of only a few specific branches. These branches handle timeout and unexpected communication behavior and are perfect candidates for default altsteps. Figure 6.3 shows an example. The branch handles the reception of unexpected messages. It could be perfectly extracted into an own altstep and activated as default behavior. Hence, ***Duplicate Alt Branches*** (4.1.2) seem to be meaningful for the SIP test suite.

There is also a remarkably great number of Unused Local Definitions in the SIP test suite. Figure 6.4 shows an unused variable which is declared locally and never used. Maybe these definitions originate from changes made during the evolution of the test suites. However, local instances of ***Unused Definition*** (4.2.3) only increase code size unnecessarily and are certainly relevant.

Altogether, the application of the smell detection to existing reference test suites shows that the identified code smells have a relevance for real-life test suites.



SIP_MainModule.ttcn

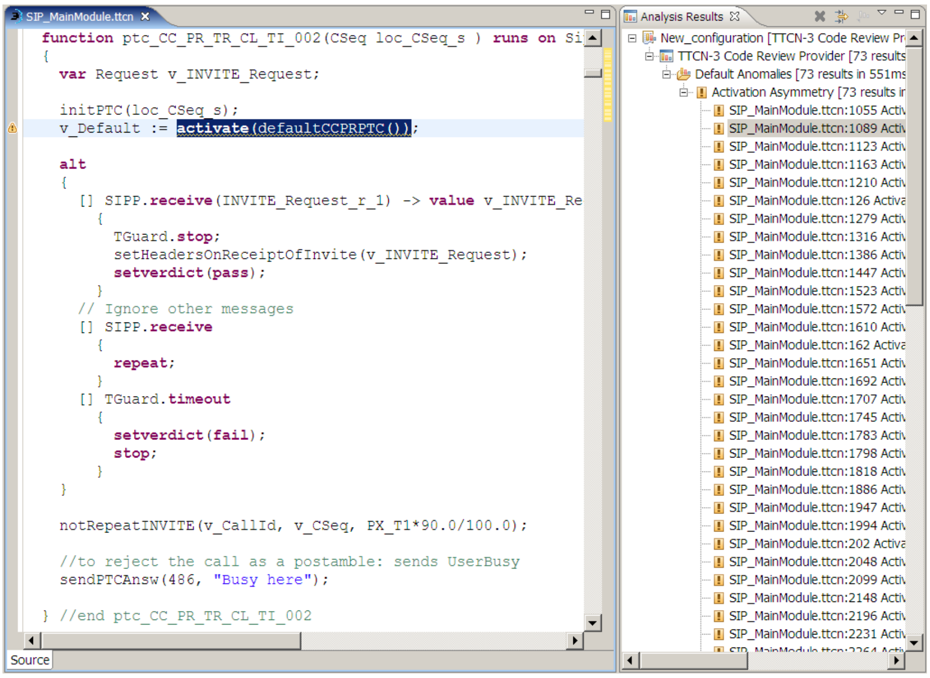
```
    v_Default := activate(defaultCCTE());
    initPort(mtc, system);          // Init test Configuration
    uASCompletedState(v_Response, loc_CSeq_s);

    // First Repetition
    repeatRespInTime(v_Response, loc_CSeq_s, PX_T1*1.5);

    // Second Repetition
    v_delay := minValue(2.0*PX_T1, PX_T2) * 1.5;
    repeatRespInTime(v_Response, loc_CSeq_s, v_delay);

    // Third repetition
    v_delay := minValue(4.0*PX_T1, PX_T2) * 1.1;
    repeatRespInTime(v_Response, loc_CSeq_s, v_delay);
    sendACK(loc_CSeq_s);

  } // end testcase SIP_CC_TE_CE_TI_004

  testcase SIP_CC_TE_CE_TI_005(inout CSeq loc_CSeq_s) runs on
  // Ref: Annex A [1], 17.2.1 [1]
```

Source

Analysis Results

```
New_configuration [TTCN-3 Code Review Pr
  TTCN-3 Code Review Provider [662 result
    Coding Standards [543 results in 9.81
      Magic Value [543 results in 9.813m
        Magic Number [543 results in 9.
          SIP_MainModule.ttcn:10596
          SIP_MainModule.ttcn:10614
          SIP_MainModule.ttcn:10634
          SIP_MainModule.ttcn:10636
          SIP_MainModule.ttcn:10636
          SIP_MainModule.ttcn:10657
          SIP_MainModule.ttcn:10660
          SIP_MainModule.ttcn:10660
          SIP_MainModule.ttcn:10664
          SIP_MainModule.ttcn:10664
          SIP_MainModule.ttcn:10683
          SIP_MainModule.ttcn:10703
          SIP_MainModule.ttcn:10764
          SIP_MainModule.ttcn:10765
          SIP_MainModule.ttcn:10784
          SIP_MainModule.ttcn:10785
          SIP_MainModule.ttcn:10786
```

*Figure 6.1: Magic Numbers in the SIP test suite*



SIP_MainModule.ttcn

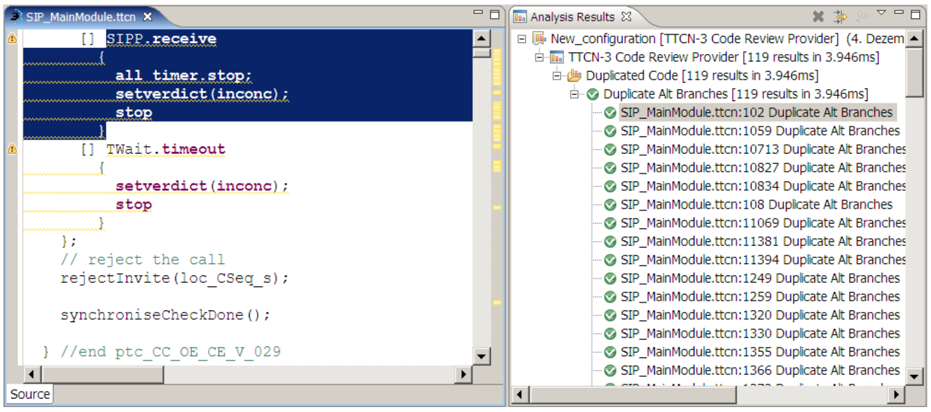
```
  function ptc_CC_PR_TR_CL_TI_002(CSeq loc_CSeq_s ) runs on Si
  {
    var Request v_INVITE_Request;

    initPTC(loc_CSeq_s);
    v_Default := activate(defaultCCPRPTC());

    alt
    {
      [] SIPP.receive(INVITE_Request_r_1) -> value v_INVITE_Re
        {
          TGuard.stop;
          setHeadersOnReceiptOfInvite(v_INVITE_Request);
          setverdict(pass);
        }
      // Ignore other messages
      [] SIPP.receive
        {
          repeat;
        }
      [] TGuard.timeout
        {
          setverdict(fail);
          stop;
        }
    }

    notRepeatINVITE(v_CallId, v_CSeq, PX_T1*90.0/100.0);

    //to reject the call as a postamble: sends UserBusy
    sendPTCAnsw(486, "Busy here");

  } //end ptc_CC_PR_TR_CL_TI_002
```

Source

Analysis Results

```
New_configuration [TTCN-3 Code Review Pr
  TTCN-3 Code Review Provider [73 results
    Default Anomalies [73 results in 551ms
      Activation Asymmetry [73 results ir
        SIP_MainModule.ttcn:1055 Activ
        SIP_MainModule.ttcn:1089 Activ
        SIP_MainModule.ttcn:1123 Activ
        SIP_MainModule.ttcn:1163 Activ
        SIP_MainModule.ttcn:1210 Activ
        SIP_MainModule.ttcn:126 Activa
        SIP_MainModule.ttcn:1279 Activ
        SIP_MainModule.ttcn:1316 Activ
        SIP_MainModule.ttcn:1386 Activ
        SIP_MainModule.ttcn:1447 Activ
        SIP_MainModule.ttcn:1523 Activ
        SIP_MainModule.ttcn:1572 Activ
        SIP_MainModule.ttcn:1610 Activ
        SIP_MainModule.ttcn:162 Activa
        SIP_MainModule.ttcn:1651 Activ
        SIP_MainModule.ttcn:1692 Activ
        SIP_MainModule.ttcn:1707 Activ
        SIP_MainModule.ttcn:1745 Activ
        SIP_MainModule.ttcn:1783 Activ
        SIP_MainModule.ttcn:1798 Activ
        SIP_MainModule.ttcn:1818 Activ
        SIP_MainModule.ttcn:1886 Activ
        SIP_MainModule.ttcn:1947 Activ
        SIP_MainModule.ttcn:1994 Activ
        SIP_MainModule.ttcn:202 Activa
        SIP_MainModule.ttcn:2048 Activ
        SIP_MainModule.ttcn:2099 Activ
        SIP_MainModule.ttcn:2148 Activ
        SIP_MainModule.ttcn:2196 Activ
        SIP_MainModule.ttcn:2231 Activ
```

*Figure 6.2: Activation Asymmetries in the SIP test suite*



SIP_MainModule.ttcn

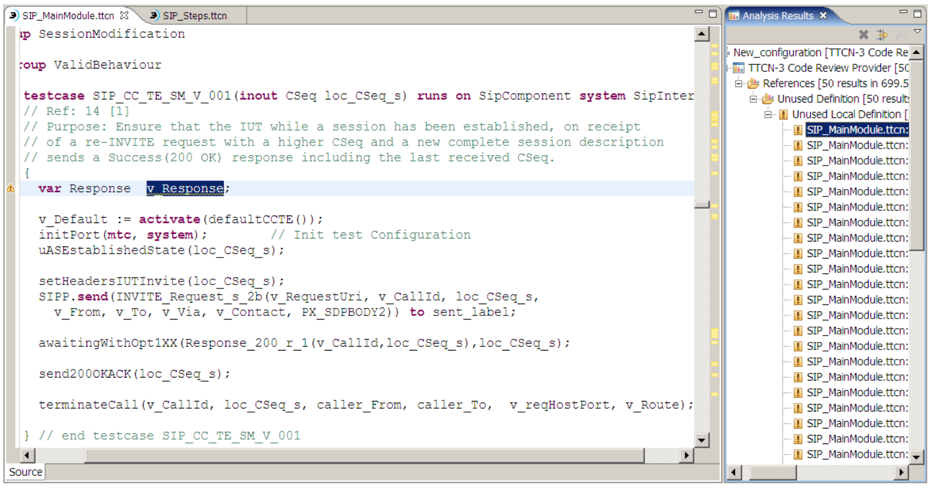
```
            [] SIPP.receive
               {
                 all timer.stop;
                 setverdict(inconc);
                 stop
               }
            [] TWait.timeout
               {
                 setverdict(inconc);
                 stop
               }
          };
          // reject the call
          rejectInvite(loc_CSeq_s);

          synchroniseCheckDone();

      } //end ptc_CC_OE_CE_V_029
```

Source

Analysis Results

```
⊟ New_configuration [TTCN-3 Code Review Provider] (4. Dezem
  ⊟ TTCN-3 Code Review Provider [119 results in 3.946ms]
    ⊟ Duplicated Code [119 results in 3.946ms]
      ⊟ Duplicate Alt Branches [119 results in 3.946ms]
          SIP_MainModule.ttcn:102 Duplicate Alt Branches
          SIP_MainModule.ttcn:1059 Duplicate Alt Branches
          SIP_MainModule.ttcn:10713 Duplicate Alt Branches
          SIP_MainModule.ttcn:10827 Duplicate Alt Branches
          SIP_MainModule.ttcn:10834 Duplicate Alt Branches
          SIP_MainModule.ttcn:108 Duplicate Alt Branches
          SIP_MainModule.ttcn:11069 Duplicate Alt Branches
          SIP_MainModule.ttcn:11381 Duplicate Alt Branches
          SIP_MainModule.ttcn:11394 Duplicate Alt Branches
          SIP_MainModule.ttcn:1249 Duplicate Alt Branches
          SIP_MainModule.ttcn:1259 Duplicate Alt Branches
          SIP_MainModule.ttcn:1320 Duplicate Alt Branches
          SIP_MainModule.ttcn:1330 Duplicate Alt Branches
          SIP_MainModule.ttcn:1355 Duplicate Alt Branches
          SIP_MainModule.ttcn:1366 Duplicate Alt Branches
```

*Figure 6.3: Duplicate Alt Branches in the SIP test suite*



SIP_MainModule.ttcn   SIP_Steps.ttcn

```
ıp SessionModification

:oup ValidBehaviour

testcase SIP_CC_TE_SM_V_001(inout CSeq loc_CSeq_s) runs on SipComponent system SipInter
// Ref: 14 [1]
// Purpose: Ensure that the IUT while a session has been established, on receipt
// of a re-INVITE request with a higher CSeq and a new complete session description
// sends a Success(200 OK) response including the last received CSeq.
{
  var Response  v_Response;

  v_Default := activate(defaultCCTE());
  initPort(mtc, system);          // Init test Configuration
  uASEstablishedState(loc_CSeq_s);

  setHeadersIUTInvite(loc_CSeq_s);
  SIPP.send(INVITE_Request_s_2b(v_RequestUri, v_CallId, loc_CSeq_s,
    v_From, v_To, v_Via, v_Contact, PX_SDPBODY2)) to sent_label;

  awaitingWithOpt1XX(Response_200_r_1(v_CallId,loc_CSeq_s),loc_CSeq_s);

  send200OKACK(loc_CSeq_s);

  terminateCall(v_CallId, loc_CSeq_s, caller_From, caller_To,  v_reqHostPort, v_Route);

} // end testcase SIP_CC_TE_SM_V_001
```

Source

Analysis Results

```
New_configuration [TTCN-3 Code Re
TTCN-3 Code Review Provider [50
  ⊟ References [50 results in 699.5
    ⊟ Unused Definition [50 result
      ⊟ Unused Local Definition [
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
          SIP_MainModule.ttcn:
```

*Figure 6.4: Unused Local Definitions in the SIP test suite*

# 7 Summary and Outlook

This chapter summarizes the previous chapters. Furthermore, possible future prospects are discussed.

## 7.1 Summary

Just like any other piece of software, tests written in TTCN-3 can suffer from serious quality problems. *Refactoring* is a means to improve the quality by enhancing the internal structure without changing the observable behavior. Code parts in need of refactoring are characterized by the metaphor of *code smells*. Although smells can be located by human intuition, the use of automated techniques is preferable. Automated smell detection can be realized either by defining threshold values for *metrics* or by *pattern matching*. Together with automated refactoring, automated smell detection allows a semi-automated process for improving test suites.

Motivated by refactorings for TTCN-3 and by well-known code smells for Java, a catalog of 39 code smells for TTCN-3 has been compiled. It contains both general and TTCN-3 specific code smells. For better clearness the smells are organized in categories. Each smell is presented with a name, a description, a motivation, related refactorings and an example. The catalog can be used for both manual inspection and as groundwork for automated smell detection.

Based on the TRex tool and thus on the Eclipse Platform, a tool for code smell detection has been developed. It uses static analysis to match smell patterns on the AST in order to find code smell instances. Smell detections are implemented as analysis rules for the TPTP Static Analysis Framework. A total number of 11 analysis rules have been implemented, three of them connected to a quick-fix. Furthermore, unit tests for all smell detection rules have been written. As an indication for its relevance, the code smell detection has been applied to existing TTCN-3 test suites.

## 7.2 Outlook

There are some issues remaining which could not be covered by the scope of this thesis. These issues are summarized by the following subsections.

### 7.2.1 Declarative Approaches

The current implementation relies on rules written in Java. The description of the smell that is detected by a rule is implied in the code. Hence the rule class can be seen as imperative description of the smell it detects. Further smells can be supported by providing further rules. However, it would be desirable to allow some kind of declarative smell description. This would ease adding new smells.

Some of the tools introduced in section 3.3.5 offer declarative techniques. For example, PMD [49] uses XPath [8] for the specification of some of its rules. XPath is a language for addressing parts of an XML document. XML documents can thereby be seen as a tree-based structure – the *Document Object Model* (DOM), the abstract data type defined for XML documents, is basically a tree.

If a syntax tree can be transformed into an XML document, XPath can be used to navigate through this document and select sets of nodes. Predicates in square brackets constrict the selection. For example, ***Magic Values*** (4.8.1) of type **integer** correspond to the following XPath expression, which selects all **integer** values which are not part of a constant definition (a similar expression could be used for other types):

```
//IntegerValue[not(./ancestor::ConstDef)]
```

While XPath is suitable for simple selection of nodes, it falls short for more complex smells. For example, there is no straightforward construct to match sub tree clones (which is needed for all smells regarding duplicated code). For these cases XQuery [5] could be used instead. XQuery is a full-grown query language using a SQL-style syntax which includes XPath as a subset. To use XPath or XQuery for examining syntax trees an engine for the languages has to be integrated into TRex, and the AST has to be transformed to a DOM. For the latter purpose, Widemann et al. [60] present an interesting approach for the creation of a DOM from an ANTLR-generated AST.

Of course there are many other approaches than XPath for matching patterns in syntax trees [9, 38, 54, 55, 57]. Many of them do not only support pattern matching, but also offer transformation mechanisms.

### 7.2.2 Dynamic analysis

The smells collected for the catalog presented in chapter 4 were collected with static analysis in mind. Accordingly, the smell detection tool introduced in chapter 5 is based on static analysis. Dynamic techniques could possibly enhance the detection of some smells and allow the expansion on "dynamic smells". For example, default activation and deactivation could be analyzed dynamically.

### 7.2.3 Implementational Enhancements

In its current form, the smell detection analyzes each source file in isolation. For some smells (like Duplicated Code) it would be preferrable to consider all files of a project or of the whole workspace. This could be achieved by linking each *PatternDetectionResource* with all other resources in the same project or workspace.

Furthermore a connection between the TRex metrics plug-in and the code smell detection could be established. This would offer the possibility to use metrics values as part of smell detection rules. For example, an ***Unused Definition*** (4.2.3) detection could be implemented by retrieving all definition whose number of references is zero.

Finally the tests could be enhanced by using mock objects which are connected to rules that are to be tested. By this approach the test would interact directly with the rule instead of using the launch framework, and result generation could be controlled firsthand instead of using the marker mechanism.

# Acknowledgments

First of all I would like to thank Dr. Helmut Neukirchen for his guidance and excellent supervision of this thesis. Further thanks go to Prof. Dr. Jens Grabowski and Benjamin Zeiss for their ideas and advice. I would also like to thank Patrick Schiel for proof-reading and Robert Burdick for keeping me from rusting in. Finally I would like to thank my fiancée for her thorough encouragement, and my parents for their unrestricted support throughout all years of study.

# Abbreviations and Acronyms

**ANTLR** *Another Tool for Language Recognition*

**API** *Application Programming Interface*

**AST** *Abstract Syntax Tree*

**CORBA** *Common Object Request Broker Architecture*

**CVS** *Concurrent Versions System*

**DECT** *Digital Enhanced Cordless Telecommunications*

**DOM** *Document Object Model*

**EBNF** *Extended Backus-Naur Form*

**EMF** *Eclipse Modeling Framework*

**ETSI** *European Telecommunications Standard Institute*

**GSM** *Global System for Mobile Communications*

**IDE** *Integrated Development Environment*

**IPv6** *Internet Protocol Version 6*

**JDT** *Java Development Tools*

**LTK** *Language Toolkit*

**MTC** *Main Test Component*

**OS** *Operating System*

**PDE** *Plug-In Development Environment*

**PTC** *Parallel Test Component*

**SDK** *Software Development Kit*

**SIP** *Session Initiation Protocol*

**SUT** *System Under Test*

**SWT** *Standard Widget Toolkit*

**TPTP** *Test & Performance Tools Platform*

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

**TSI** *Test System Interface*

**TTCN** *Tree and Tabular Combined Notation*

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

**UI** *User Interface*

**XML** *Extensible Markup Language*

**XP** *Extreme Programming*

# Bibliography

[1] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, 1997.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the 1998 International Conference on Software Maintenance*, pages 368–377. IEEE Computer Society, 1998.

[3] K. Beck. *Extreme Programming Explained.* Addison Wesley, 2000.

[4] S. Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. Diploma thesis, University of Stuttgart, Institute for Software Technology, Germany, 2004.

[5] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3C Proposed Recommendation, `http://www.w3.org/TR/xquery`, 2006.

[6] W. J. Brown, R. C. Malveau, I. Hays W. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[7] Checkstyle. `http://checkstyle.sourceforge.net`.

[8] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, `http://www.w3.org/TR/xpath`, 1999.

[9] J. R. Cordy, I. H. Carmichael, and R. Halliday. *The TXL Programming Language Version 10.4*, 2005.

[10] CVS — Concurrent Versions System. `http://www.nongnu.org/cvs`.

[11] T. Deiß. Refactoring and Converting a TTCN-2 Test Suite. Presentation at the TTCN-3 User Conference 2005, June 6-8, 2005, Sophia-Antipolis, France, May 2005.

[12] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM '99: Proceedings of the 1999 International Conference on Software Maintenance*, pages 109–118. IEEE Computer Society, sep 1999.

[13] Eclipse Foundation. Eclipse. http://www.eclipse.org.

[14] Eclipse Foundation. Eclipse Modelling Framework. http://www.eclipse.org/emf.

[15] Eclipse Foundation. Eclipse Test & Performance Tools Platform Project (TPTP). http://www.eclipse.org/tptp.

[16] E. V. Emden and L. Moonen. Java quality assurance by detecting code smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 97, Washington, DC, USA, 2002. IEEE Computer Society.

[17] ETSI. ETSI Standard (ES) 201 873-10 V3.2.1 (to appear): TTCN-3 Documentation Comment Specification. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France.

[18] ETSI. TTCN-3 naming conventions. http://www.ttcn-3.org/NamingConventions.htm.

[19] ETSI. ETSI Standard (ES) 201 873-1 V3.1.1 (2005-06): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.140, 2005.

[20] ETSI. ETSI Standard (ES) 201 873-2 V3.1.1 (2005-06): The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular Presentation Format (TFT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.141, 2005.

[21] ETSI. ETSI Standard (ES) 201 873-3 V3.1.1 (2005-06): The Tree and Tabular Combined Notation version 3; Part 3: Graphical Presentation Format for TTCN-3 (GFT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.142, 2005.

[22] ETSI. Technical Specification TS 102 027-3 V3.2.1 (2005-07): 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, 2005.

[23] 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.

[24] FindBugs. http://findbugs.sourceforge.net.

[25] M. Fowler. Refactoring Home Page. `http://www.refactoring.com`.

[26] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[27] L. Frenzel. Neutral im Sinne der Qualität. *Eclipse Magazin*, 5, 2005.

[28] E. Gamma and K. Beck. JUnit. `http://junit.sourceforge.net`.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, Massachusetts, 2000.

[30] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks, Volume 42, Issue 3*, pages 375–403, June 2003.

[31] S. Gutz and O. 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.

[32] S. Gutz and O. 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.

[33] S. Gutz and O. 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.

[34] M. H. Halstead. *Elements of Software Science.* Elsevier, New York, 1977.

[35] Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On software maintenance process improvement based on code clone analysis. In *PROFES '02: Proceedings of the 4th International Conference on Product Focused Software Process Improvement*, pages 185–197, London, UK, 2002. Springer.

[36] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[37] ISO/IEC. International standard ISO/IEC 9646-3:1998: Information Technology – Open Systems Interconnection – Conformance testing methodology and framework – Part 3: The Tree and Tabular Combined Notation (TTCN). International Organization for Standardization/International Electrotechnical Commission, 1998.

[38] The Jackpot Project. `http://jackpot.netbeans.org`.

[39] JavaCC Project. `https://javacc.dev.java.net`.

[40] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[41] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 301, Washington, DC, USA, 2001. IEEE Computer Society.

[42] M. Mäntylä. Bad smells in software - a taxonomy and an empirical study. Master's thesis, Helsinki University of Technology, 2003.

[43] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 244, Washington, DC, USA, 1996. IEEE Computer Society.

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

[45] N. Moha and Y.-G. Gueheneuc. On the Automatic Detection and Correction of Design Defects. In S. Demeyer, K. Mens, R. Wuyts, and S. Ducasse, editors, *proceedings of the* $6^{th}$ *ECOOP Workshop on Object-Oriented Reengineering*, July 2005.

[46] Netbeans. `http://www.netbeans.org`.

[47] H. Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, Georg-August-Universität Göttingen, 2004.

[48] J. Philipps and B. Rumpe. Roots of Refactoring. In K. Baclavski and H. Kilov, editors, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15, 2001.* Northeastern University, 2001.

[49] PMD. `http://pmd.sourceforge.net`.

[50] S. Roock and M. Lippert. *Refactorings in grossen Softwareprojekten. Komplexe Restrukturierungen erfolgreich durchführen.* dpunkt.verlag, Heidelberg, 2004.

[51] Simian – Similarity Analyser. `http://www.redhillconsulting.com.au/products/simian/index.html`.

[52] A. Spillner and T. Linz. *Basiswissen Softwaretest.* dpunkt.verlag, Heildelberg, 2004.

[53] TRex – the TTCN-3 Refactoring and Metrics Tool. `http://www.trex.informatik.uni-goettingen.de`.

[54] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, London, UK, 2001. Springer.

[55] P. van Eijk, A. Belinfante, H. Eertink, and H. Alblas. The term processor generator kimwitu. In *TACAS '97: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 96–111, London, UK, 1997. Springer.

[56] D. E. Vega and I. Schieferdecker. Towards Quality of TTCN-3 Tests. In *Proceedings of SAM'06: Fifth Workshop on System Analysis and Modelling (formerly SDL and MSC Workshop), May 31st-June 2nd 2006, University of Kaiserslautern, Kaiserslautern, Germany*, 2006.

[57] E. Visser. Program transformation with stratego/xt. rules, strategies, tools, and systems in stratego/xt 0.9. Technical Report UU-CS-2004-011, Institute of Information and Computing Sciences, Utrecht University, 2004.

[58] C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. John Wiley & Sons, Ltd, 2005.

[59] A. Wu-Hen-Chang, D. L. Viet, G. Batori, R. Gecse, and G. Csopaki. High-Level Restructuring of TTCN-3 Test Data. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing: 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science (LNCS)*, pages 180–194. Springer, 2005.

[60] B. T. y Widemann, M. Lepper, and J. Wieland. Automatic construction of XML-based tools seen as meta-programming. *Automated Software Engineering*, 10(1):23–38, 2003.

[61] B. 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, Mar. 2006.

[62] B. Zeiss, H. Neukirchen, J. Grabowski, D. Evans, and P. 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 (LNCS) 4320. DOI: 10.1007/11951148_10*, pages 148–165. Springer, Dec. 2006.

All URLs have been verified on December 18, 2006.