Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites

Helmut Neukirchen[1] and Martin Bisanz[2]

[1] Software Engineering for Distributed Systems Group,
Institute for Informatics, University of Göttingen,
Lotzestr. 16–18, 37083 Göttingen, Germany
neukirchen@cs.uni-goettingen.de
[2] PRODYNA GmbH, Eschborner Landstr. 42–50, 60489 Frankfurt, Germany
martin.bisanz@prodyna.de

Abstract. Today, test suites of several ten thousand lines of code are specified using the *Testing and Test Control Notation* (TTCN-3). Experience shows that the resulting test suites suffer from quality problems with respect to internal quality aspects like usability, maintainability, or reusability. Therefore, a quality assessment of TTCN-3 test suites is desirable. A powerful approach to detect quality problems in source code is the identification of *code smells*. Code smells are patterns of inappropriate language usage that is error-prone or may lead to quality problems. This paper presents a quality assessment approach for TTCN-3 test suites which is based on TTCN-3 code smells: To this aim, various TTCN-3 code smells have been identified and collected in a catalogue; the detection of instances of TTCN-3 code smells in test suites has been automated by a tool. The applicability of this approach is demonstrated by providing results from the quality assessment of several standardised TTCN-3 test suites.

1 Introduction

Current test suites from industry and standardisation that are specified using the *Testing and Test Control Notation* (TTCN-3) [1,2] reach sizes of around 40–60 thousand lines of code [3,4,5]. These test suites are either generated or respectively migrated automatically [6] or they are created manually [4,5]. In both cases, the resulting test suites need to be maintained afterwards. The maintenance of test suites is an important issue for industry [6] and standardisation [7,8]. A burden is put on the maintainers if the test suites have a low internal quality resulting from badly generated code or from inexperienced developers [6]. Hence, it is desirable to assess the quality of TTCN-3 test specifications.

According to the ISO/IEC standard 9126 [9], a software product can be evaluated with respect to three different types of quality: *internal quality* is assessed using static analysis of source code. *External quality* refers to properties of software interacting with its environment. In contrast, *quality in use* refers to the quality perceived by an end user who executes a software product in a specific context. In the remainder, we will focus on internal quality problems.

A simple approach for the quality assessment of source code are metrics [10]. In earlier work, we have experienced that metrics are suitable to assess either very local [3] or very global [11] internal quality aspects of TTCN-3 test suites. However, properties of language constructs which are, for example, related but distributed all over the source code are hard to assess using simple metrics. Instead, a more powerful pattern-based approach is required to detect patterns of inappropriate language usage that is error-prone or may lead to quality problems. These patterns in source code are described by so called *code smells*.

This paper introduces TTCN-3 code smells and utilises them to detect internal quality problems in TTCN-3 test suites. The located quality problems can be used as input for the plain quality assessment of test suites and as well as a starting point for the quality improvement of test suites.

The structure of this paper is as follows: subsequent to this introduction, foundations and work related to smells in software are presented in Section 2. A survey of work concerning smells in tests is given in Section 3. As the main contribution, a catalogue of TTCN-3 code smells is introduced in Section 4. Then, in Section 5, a tool is described which is able to automatically detect instances of TTCN-3 code smells in test suites. Section 6 provides results from applying this tool to several huge standardised test suites. Finally, this paper concludes with a summary and an outlook.

2 Foundations

The metaphor of "*bad smells in code*" has been coined by Beck and Fowler in the context of refactoring [12]. Refactoring is a technique to improve the internal quality of software by restructuring it without changing its observable behaviour. As an aid to decide where the application of a refactoring is worthwhile, Beck and Fowler introduce the notion of smell: they define smells in source code as "*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*" [12]. According to this definition, defects with respect to program logic, syntax, or static semantics are not smells, because these defects cannot be removed by a behaviour-preserving refactoring. This means, smells are indicators of bad internal quality with respect to (re-)usability, maintainability, efficiency, and portability.

Smells provide only hints: whether the occurrence of an instance of a certain smell in a source code is considered as a sign of low quality may be a matter that depends on preferences and experiences. For the same reason, a list of code structures which are considered as smell is never complete, but may vary from project to project and from domain to domain [13].

Beck and Fowler provide a list of 22 smells which may occur in Java source code. They describe their smells using unstructured English text. The most prominent smell is *Duplicated Code*. Code duplication deteriorates in particular the changeability of a source code: if code that is duplicated needs to be modified, it usually needs to be changed in all duplicated locations as well. Another example from Beck's and Fowler's list of smells is *Long Method* which relates to the fact that short methods are easier to understand and to reuse, because they do exactly one thing. A further example is the smell called *Data Class* which characterises classes that only have attributes and accessor methods, but the actual algorithms working on these data are wrongly located in methods of other classes.

Most of the smells from Beck and Fowler relate to pathological structures in the source code. Thus, to detect such structures, a pattern-based approach is required: for example, to identify duplicated code, pattern-matching is required; to detect data classes, it has to be identified whether the methods of a class are only simple get and set methods and whether methods in other classes do excessively manipulate data from that particular class. Such patterns cannot be detected by metrics — however, the notion of metrics and smells is not disjoint: each smell can be turned into a metric by counting the occurrences of a smell, and sometimes, a metric can be used to detect and locate an instance of a smell. The latter is, for example, the case for the *Long Method* smell which can be expressed by a metric which counts the lines of code of a method.[1]

Bad smells are also related to *anti-patterns* [14]. Anti-patterns describe solutions to recurring problems that are wrong and bad practice and shall thus be avoided. A well-known anti-pattern is the one called *Spaghetti Code*, i.e. software with little structure. Even though this and other anti-patterns relate to source code, anti-patterns do not refer to low-level code details as code smells do. In fact, the majority of the anti-patterns do not relate to source code at all, but to common mistakes in project management.

The awareness of problematic source code structures is older than the notion of smells in source code. For example, patterns of data flow anomalies which can be detected by static analysis have been known for a long time [15]. However, these older works mainly relate to erroneous, inconsistent, inefficient, and wasteful code constructs. The added value of smells is to consider also more abstract source code quality problems, for example those which lead to maintenance problems.

3 Smells in Tests

As stated in the previous section, the perception of what is considered as a smell may vary from domain to domain. Hence, for the testing domain, a separate investigation of smells is required. Van Deursen et al. and Meszaros studied smells in the context of tests that are based on the Java unit test framework *JUnit* [16].

Van Deursen et al. [17] introduce the term *test smell* for smells that are specific to the usage of *JUnit* as well as for more general JUnit-independent issues in test

[1] It has to be noted that Beck and Fowler state that for detecting instances of a smell "*no set of metrics rivals informed human intuition*". This is obviously true for those smells where no corresponding metric exists. However, in the cases, where such a metric exists, this statement does in our opinion rather relate to the fact that reasonable boundary values for such a metric may vary from case to case and thus it is hard to provide a universally valid boundary value for that metric.

behaviour that can be removed by a refactoring. An example for a JUnit-specific test smell is *General Fixture* which refers to test cases that share unnecessarily the same fixture (i.e. test preamble), just because the test cases are collected in the same JUnit testcase class. A more general test smell is, for example, *Test Run War* which relates to the fact that test cases may behave non-deterministic due to shared test resources when several test campaigns run in parallel.

Meszaros [18] refines the notion of test smell by distinguishing between three kinds of smells that concern tests: *code smells* relate to test issues that can be detected when looking at source code, *behaviour smells* affect the outcome of tests as they execute, and *project smells* are indicators of the overall health of a project which do not involve looking at code or executing tests. Within this classification, smells of different kinds may affect each other; for example, the root cause of a behaviour smell may be a problem in the code. We regard this classification of test smells as reasonable and adopt this terminology as well.

Those test smells from Van Deursen et al. that are JUnit-specific (e.g. *General Fixture*) can be considered as code smells while others are more general (e.g. *Test Run War*) and can thus be regarded as behaviour smells. Meszaros does not only refine the notion of test smells, but also extends the list of test smells from Van Deursen et al. by further smells. An example for an additional code smell is *Conditional Test Logic* which refers to tests which are error-prone because they use complex algorithms to calculate test data and to steer test behaviour. A behaviour smell identified by Meszaros is, for example, *Fragile Tests*, which are tests that fail after non-relevant changes of the *System Under Test* (SUT). An example of a project smell is *Developers Not Writing Tests*.

4 A TTCN-3 Code Smell Catalogue

While code smells have been identified for tests written using the JUnit framework, smells have not yet been investigated in the context of TTCN-3. The project smells identified by Meszaros [18] are independent from any test language and can thus be used as well in projects that involve TTCN-3. Most of Meszaros' behaviour smells apply to TTCN-3 tests without change, however those behaviour smells whose root cause is a JUnit related code smell are only applicable after a reinterpretation. Only a subset of the JUnit related code smells can be reinterpreted in a way that they are applicable to TTCN-3. Hence, code smells related to TTCN-3 need further investigation.

We have started to identify TTCN-3 code smells which we use to assess the internal quality of TTCN-3 test specifications. When investigating possible smell candidates we have relaxed Beck's and Fowler's definition of smells in source code: We include not only internal quality problems in TTCN-3 source code that can be improved by a behaviour preserving refactoring, but we consider as well quality problems which obviously require a change of the behaviour. One example is a test case which never sets a test verdict. In this case, a statement that sets a verdict needs to be added. This cannot be achieved by applying a refactoring, since this is a change that would not be behaviour-preserving.

Though, we still adhere to the definition of code smell, in that we do not consider errors in TTCN-3 source code with respect to syntax or static semantics as a smell.

As a starting point for our investigations, we examined those code smells that were already known for implementation and testing languages. Even though the smells listed by Beck and Fowler [12] are intended for Java code, some of them proved to be suitable for TTCN-3 code. A further source was the TTCN-3 refactoring catalogue [3,19,20] which was in turn inspired by the JUnit refactorings and JUnit code smells published by Van Deursen et al. [17]. The refactorings collected in the TTCN-3 refactoring catalogue already refer briefly to code smell-like quality issues as a motivation for each refactoring.

In contrast to the plain listing of unstructured smell descriptions that is used by Beck and Fowler or by Van Deursen et al., we have catalogued our TTCN-3 code smells in a structured way. This structured presentation allows a more systematic and faster access to the smell descriptions. The entries in our TTCN-3 code smell catalogue are listed in the following format: each smell has a *name*; those smells which are derived from other sources have a *derived from* section which lists the corresponding references; a *description* provides a prose summary of the issue described by the smell; the *motivation* part explains why the described code structure is considered to have low quality; if several variants of a smell are possible (e.g. by relaxing or tightening certain requirements on a code structure), this is mentioned in an *options* section; one or more actions (typically refactorings) which are applicable to remove a smell are listed in the *related actions* section; finally, a TTCN-3 source code snippet is provided for each smell in the *example* section.

In our smell catalogue, the names of TTCN-3 code smells are emphasised using *slanted* type and TTCN-3 keywords are printed using **bold** type. The following overview on our TTCN-3 code smell catalogue gives an impression of the so far identified 38 TTCN-3 code smells. The overview provides the name and the summary of each smell and uses the same division into 10 sections as our TTCN-3 code smell catalogue:

Duplicated Code

- *Duplicate Statements:* A duplicate sequence of statements occurs in the statement block of one or multiple behavioural entities (functions, test cases, and altsteps).
- *Duplicate Alt Branches:* Different **alt** constructs contain duplicate branches.
- *Duplicated Code in Conditional:* Duplicated code is found in the branches of a series of conditionals.
- *Duplicate In-Line Templates:* Two or more in-line templates are very similar or identical.
- *Duplicate Template Fields:* The fields of two or more templates are identical or very similar.
- *Duplicate Component Definition:* Two or more test components declare identical variables, constants, timers, or ports.

– *Duplicate Local Variable/Constant/Timer:* The same local variable, constant, or timer is defined in two or more functions, test cases, or altsteps running on the same test component.

References

– *Singular Template Reference:* A template definition is referenced only once.
– *Singular Component Variable/Constant/Timer Reference:* A component variable, constant, or timer is referenced by one single function, test case, or altstep only, although other behaviour runs on the component as well.
– *Unused Definition:* A definition is never referenced.
– *Unused Imports:* An import from another module is never used.
– *Unrestricted Imports:* A module imports more than needed.

Parameters

– *Unused Parameter:* A parameter is never used within the declaring unit: **in**-parameters are never read, **out**-parameters are never assigned, **inout**-parameters are never accessed at all.
– *Constant Actual Parameter Value:* The actual parameter values for a formal parameter are the same for all references.
– *Fully-Parametrised Template:* All fields of a template are defined by formal parameters.

Complexity

– *Long Statement Block:* A function, test case, or altstep has a long statement block.
– *Long Parameter List:* The number of formal parameters is high.
– *Complex Conditional:* A conditional expression is composed of many Boolean conjunctions.
– *Nested Conditional:* A conditional expression is unnecessarily nested.
– *Short Template:* A template definition is very short.

Default Anomalies

– *Activation Asymmetry:* A default activation has no matching subsequent deactivation in the same statement block, or a deactivation has no matching previous activation.
– *Unreachable Default:* An **alt** statement contains an **else** branch while a default is active.

Test Behaviour

– *Missing Verdict:* A test case does not set a verdict.
– *Missing Log:* **setverdict** sets the verdict **inconc** or **fail** without calling **log**.
– *Stop in Function:* A function contains a **stop** statement.

Test Configuration

- *Idle PTC:* A *Parallel Test Component* (PTC) is created, but never started.
- *Isolated PTC:* A PTC is created and started, but its ports are not connected to other ports.

Coding Standards

- *Magic Values:* A literal is not defined as a TTCN-3 constant.
- *Bad Naming:* An identifier does not conform to a given naming convention.
- *Disorder:* The sequence of elements within a module does not conform to a given order.
- *Insufficient Grouping:* A module or group contains too many elements.
- *Bad Comment Rate:* The comment rate is too high or too low.
- *Bad Documentation Comment:* A documentation comment does not conform to a given format, e.g. T3Doc [21].

Data Flow Anomalies

- *Missing Variable Definition:* A variable or **out** parameter is read before a value has been assigned.
- *Unused Variable Definition:* An assigned variable or **in**-parameter is not read before it becomes undefined.
- *Wasted Variable Definition:* A variable is assigned and assigned again before it is read.

Miscellaneous

- *Over-specific Runs On:* A behavioural entity runs on a component but uses only elements of the super-component or no component elements at all.
- *Goto:* A **goto** statement is used.

To give an impression of how the entries in our TTCN-3 code smell catalogue look like, the smells *Duplicate Alt Branches* and *Activation Asymmetry* are subsequently presented in detail. In addition to the already mentioned style of typesetting TTCN-3 keywords and names of smells, references to refactorings from the TTCN-3 refactoring catalogue [3,19,20] are printed in *slanted* type as well.[2] Please refer to our complete TTCN-3 code smell catalogue [22] for a detailed description of all so far identified TTCN-3 code smells.

4.1 TTCN-3 Code Smell: *Duplicate Alt Branches*

Derived from: TTCN-3 refactoring catalogue [3,19,20].
Description: Different **alt** constructs contain duplicate branches.

[2] References to refactorings and to smells can still be distinguished, because the names of refactorings usually start with a verb followed by a noun, whereas the names of smells usually consist of an adjective and a noun.

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

Options: Since analysability is increased if the path leading to a **pass** verdict is explicitly visible in a test case, **alt** branches leading to **pass** can be excluded optionally.

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

Example: In Listing 1.1, both test cases contain an **alt** construct where the last branch (lines 6–10 and lines 19–23) can be found as well in the other **alt** construct.

```
 1  testcase myTestcase1() runs on myComponent {
 2      alt {
 3          [ ] pt.receive(messageOne) {
 4              pt.send(messageTwo);
 5          }
 6          [ ] any port.receive {
 7              log("unexpected message");
 8              setverdict(inconc);
 9              stop;
10          }
11      }
12  }
13
14  testcase myTestcase2() runs on myComponent {
15      alt {
16          [ ] pt.receive(messageThree) {
17              pt.send(messageFour);
18          }
19          [ ] any port.receive {
20              log("unexpected message");
21              setverdict(inconc);
22              stop;
23          }
24      }
25  }
```

Listing 1.1. Duplicate Alt Branches

4.2 TTCN-3 Code Smell: *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: The analysability with respect to active defaults is improved if default activation and deactivation is done on the same "level", usually at the very beginning and end of the same statement block. Furthermore, this enables a static analysis of matching activation and deactivation.

Options: Because defaults are implicitly deactivated at the end of a test case 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: In Listing 1.2, the altstep "myAltstep" (lines 1–6) is used as default. Function "myFunction" (lines 8–10) activates this altstep as default, but no **deactivate** statement is contained in the statement block of this function. Even though it might be reasonable in some situations to move activation and deactivation of defaults into separate functions, this has to be considered as an asymmetric default activation. A further asymmetry can be found in the test case "myTestcase": the statement block of the **deactivate** statement in Line 20 consists of lines 13–15 and Line 20. This statement block contains no **activate** statement, since the activation of the default is performed within the statement block of the function "myFunction" that is called in Line 14.

```
 1  altstep myAltstep() runs on myComponent {
 2      [ ] any port.receive {
 3          log("unexpected message");
 4          setverdict(inconc);
 5      }
 6  }
 7
 8  function myFunction() return default {
 9      return activate(myAltstep());
10  }
11
12  testcase myTestcase() runs on myComponent {
13      var default myDefaultVar := null;
14      myDefaultVar := myFunction();
15      alt {
16          [ ] pt.receive(messageOne) {
17              pt.send(messageTwo);
18          }
19      }
20      deactivate(myDefaultVar);
21  }
```

Listing 1.2. Activation Asymmetry

5 A Tool for Detecting TTCN-3 Code Smell Instances

Our TTCN-3 code smell catalogue can be utilised for the quality assessment of TTCN-3 test suites. One possibility is to use it as part of a checklist in a manual inspection of TTCN-3 code. However, the efficiency of such a code inspection can be significantly improved if the detection of instances of TTCN-3 code smells is automated by a tool.[3] This allows the code reviewers to focus

[3] All of our TTCN-3 code smells are intended to be detected by static analysis; however, those analyses required for smells related to test behaviour and data flow anomalies are —in the general case— undecidable and can thus only solved by static analysis heuristics (in the simplest case by neglecting any branching and assuming a linear control flow instead).

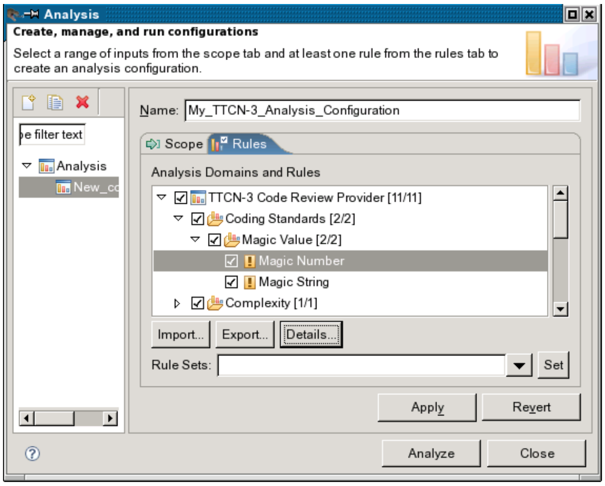
on high-level logical errors in the test suite, since instances of low-level code smells have already been detected automatically. However, in our experience an everyday usage of an automated issue detection outside of a formal inspection is even more beneficial: the push-button detection of smell instances allows test engineers to easily obtain feedback on the internal quality of the TTCN-3 test suites that they are currently developing.

We have implemented the automated detection of instances of TTCN-3 code smells into our open-source TTCN-3 Refactoring and Metrics tool *TRex* [20]. The initial version of TRex [23] has been developed in collaboration with the Motorola Labs, UK, to provide an *Integrated Development Environment* (IDE) for the quality assessment and improvement of TTCN-3 test suites. In that version, the quality assessment was based on metrics; for the quality improvement, refactoring is used [3]. Since then, we have extended the quality assessment capabilities of TRex by an additional automated detection of TTCN-3 code smell instances. So far, TRex provides rules to detect by static analysis instances of the following 11 TTCN-3 code smells:

– *Activation Asymmetry*,
– *Constant Actual Parameter Value* smells for templates,
– *Duplicate Alt Branches*,
– *Fully-Parametrised Template*,
– *Magic Values* of numeric or string types with configurable tolerable magic numbers,
– *Short Template* smells with configurable character lengths,
– *Singular Component Variable/Constant/Timer Reference*,
– *Singular Template Reference*,
– *Duplicate Template Fields*,
– instances of any local *Unused Definition*,
– an *Unused Definition* of a global template instance.

As stated in Section 2, whether a certain code structure is considered as a smell or not, may vary from project to project. Therefore, TRex supports enabling and disabling individual TTCN-3 code smell detection rules and to store these preferences as customised analysis configurations (Figure 1). Furthermore, it is possible to parametrise some smell detection rules. For example, for detecting instances of the *Magic Values* smell, a *Magic Number* detection rule and a *Magic String* detection rule are available; the *Magic Number* detection rule can be parametrised to exclude user defined values (e.g. 0 and 1 which are usually considered to be tolerable magic numbers) from the smell instance detection.

The results of the smell analysis are displayed as a tree in the *Analysis Results* view (Figure 2). The results are collected in a history, which allows to compare analysis results. Clicking on an entry of the analysis result jumps to the corresponding location in the TTCN-3 source code to allow a further manual inspection. Some rules, for example *Unused Definitions*, offer the possibility of invoking so called *Quick Fixes*. Quick Fixes automatically suggest the invocation of TTCN-3 refactoring to remove a detected instance of a smell. Since a couple of refactorings are implemented in TRex [23], this does not only allow an


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.

Name: My_TTCN-3_Analysis_Configuration

Analysis Domains and Rules

```
▽ ☑ TTCN-3 Code Review Provider [11/11]
  ▽ ☑ Coding Standards [2/2]
    ▽ ☑ Magic Value [2/2]
        ☑ Magic Number
        ☑ Magic String
  ▷ ☑ Complexity [1/1]
```

Rule Sets:

Fig. 1. TRex Smell Analysis Configuration

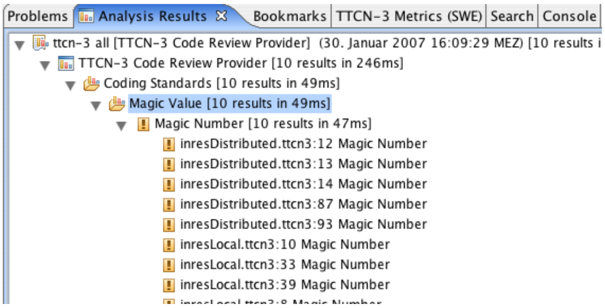

```
▼ ttcn-3 all [TTCN-3 Code Review Provider] (30. Januar 2007 16:09:29 MEZ) [10 results i
  ▼ TTCN-3 Code Review Provider [10 results in 246ms]
    ▼ Coding Standards [10 results in 49ms]
      ▼ Magic Value [10 results in 49ms]
        ▼ Magic Number [10 results in 47ms]
            inresDistributed.ttcn3:12 Magic Number
            inresDistributed.ttcn3:13 Magic Number
            inresDistributed.ttcn3:14 Magic Number
            inresDistributed.ttcn3:87 Magic Number
            inresDistributed.ttcn3:93 Magic Number
            inresLocal.ttcn3:10 Magic Number
            inresLocal.ttcn3:33 Magic Number
            inresLocal.ttcn3:39 Magic Number
```

Fig. 2. TRex Smell Analysis Results View

automated quality assessment, but as well an automated quality improvement of TTCN-3 test suites.

5.1 Implementation

The implementation of the TRex tool is based on the Eclipse platform [24] as shown in Figure 3. Eclipse provides generic user interface and text editor components as well as a language toolkit for behaviour preserving source code transformation. As an infrastructure for the automated quality assessment and quality improvement functionality of TRex (blocks (2) and (3) of Figure 3), TRex creates a syntax tree and a symbol table of the currently opened test suites (Block (1) of Figure 3). For lexing and parsing the TTCN-3 core notation, '*ANother Tool for Language Recognition*' (ANTLR) [25] is used. A further description of the implementation of the quality improvement based on refactor-

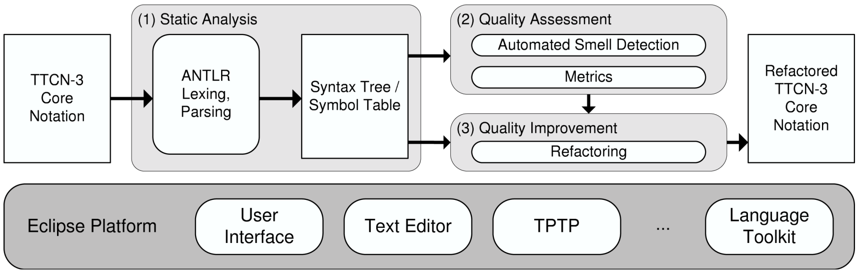


Fig. 3. The TRex Toolchain

ings and of the implementation of the quality assessment based on metrics can be found in earlier papers [3,23].

The smell analysis configuration dialogue and the smell analysis results view are provided by the *static analysis framework* which is part of the *Eclipse Test & Performance Tools Platform* (TPTP) [26]. In the context of the TPTP static analysis framework, each smell detection capability is represented by a rule. TPTP provides the underlying programming interface to add and implement rules and to call and apply the rules to files according to the user-defined analysis configuration. The actual smell instance detection is based on syntax tree traversals and symbol table lookups. For example, to detect *Duplicate Alt Branches*, the sub-syntaxtrees of all branches of **alt** and **altstep** constructs of a TTCN-3 module are compared. Currently, only exact sub-tree matches are detected; however, since the syntax tree does not contain tokens which do not have any semantical meaning, the detection of duplicates is tolerant with respect to formatting and comments. The *Unused Definition* rules make intensively use of the symbol table to check for every symbol whether it is referenced at least once or not. To ease the implementation of future smell detection rules, we have extracted frequently used helper algorithms into methods of a smell detection library.

5.2 Related Work

Approaches for the automatic detection of source code issues that are detectable by static analysis and go beyond the application of metrics have been known for a long time. The most prominent example is probably the *Lint* tool [27]. Even though Lint is older than the notion of smells, it detects issues which are nowadays considered as code smell. Current research shows that automatic detection of instances of a smell is still relevant [13,28]. In addition to this research, mature tools for detecting instances of smells in Java programs do already exist. Examples are tools like *FindBugs* [29] or *PMD* [30]. All the mentioned work deals with the detection of instances of smells in source code written in implementation languages like C or Java. Hence, this work does neither consider TTCN-3 related smells nor more general test specific smells at all. The only known work on the automated detection of instances of test smells is restricted to the detection of JUnit code smells [31].

6 Application

To evaluate the practicability of our approach, we applied TRex to several huge test suites that have been standardised by the *European Telecommunications Standards Institute* (ETSI). The first considered test suite is Version 3.2.1 of the test suite for the *Session Initiation Protocol* (SIP) [4], the second is a preliminary version of a test suite for the *Internet Protocol Version 6* (IPv6) [5]. Table 1 shows the number of detected instances of TTCN-3 code smells and provides as well some simple size metrics to give an impression of the size of these test suites.

Both test suites are comparable in size and in both, the same types of smells can be found. Magic numbers can be found quite often in both test suites. An excerpt from the SIP test suite is shown in Listing 1.3: the magic number "65.0" used in Line 10 occurs several times throughout the test suite. If that number must be changed during maintenance, it must probably changed at all other places as well which is very tedious.

The number of detected instances of the *Activation Asymmetry* smell is as well very high in both test suites. However, the number drops, if test cases are excluded from the detection. Even though the SIP test suite has less *Activation Asymmetry* smell instances, they still deteriorate the analysability of this test suite as shown in Listing 1.3: the altstep "defaultCCPRPTC" is activated in Line 6 and remains activated after leaving this function. Hence, calling this function leads to side effects that are difficult to analyse.

Finally, Listing 1.3 can be used to demonstrate occurrences of the *Unused Definition* smell in the SIP test suite: the local variable "v_BYE_Request" defined in Line 3 is never used in the function and thus just bloats the code, making it harder to analyse.

The instances of *Singular Component Variable/Constant/Timer Reference* smells can be neglected in both test suites. However, the high number of *Duplicate Alt Branches* in both test suites indicates that the introduction of further altsteps is worthwhile. For example, the branch in lines 9–11 of Listing 1.4 can be found as duplicate in several **alt** statements of the IPv6 tests suite.

Table 1. Instances of TTCN-3 Code Smells Found in ETSI Test Suites

Metric/TTCN-3 Code Smell	**SIP**	**IPv6**
Lines of code	42397	46163
Number of functions	785	643
Number of test cases	528	295
Number of altsteps	10	11
Number of components	2	10
Instances of *Magic Values* (Magic numbers only, 0 and 1 excluded)	543	368
Instances of *Activation Asymmetry* (Test cases included)	602	801
Instances of *Activation Asymmetry* (Test cases excluded)	73	317
Instances of *Duplicate Alt Branches* (Inside the same module only)	938	224
Instances of *Singular Component Variable/Constant/Timer Reference*	2	15
Instances of *Unused Definition* (Local definitions only)	50	156

This and our further analysis [22] of the detected smell instances give evidence that these instances are correctly considered as issues and can thus be used for quality assessment and as starting point to improve the internal quality of the respective test suites.

```
function ptc_CC_PR_TR_CL_TI_015(CSeq loc_CSeq_s ) runs on SipComponent
{
    var Request v_BYE_Request;

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

    tryingPTCBYE();

    waitForTimeout(65.0*PX_T1);

    notRepeatBYE(PX_TACK);

} //end ptc_CC_PR_TR_CL_TI_015
```

Listing 1.3. *Magic Values*, *Activation Asymmetry*, *Unused Definition* (SIP)

```
tc_ac.start;
alt {
    [ ] ipPort.receive ( mw_nbrAdv_noExtHdr (
                            p_paramsIut.lla,
                            p_paramsRt01.lla ) ) {
        tc_ac.stop;
        v_ret := e_success;
    }
    [ ] tc_ac.timeout{
        v_ret := e_timeout;
    }
} // end alt
```

Listing 1.4. *Duplicate Alt Branches* (IPv6)

7 Conclusion

We presented a catalogue of 38 TTCN-3 code smells that can be utilised to detect code-level problems in TTCN-3 test suites with respect to internal quality characteristics like usability, maintainability, or reusability. Each of our entries in the TTCN-3 code smell catalogue provides a description of the considered code issue, a motivation why it is considered to have low quality, an action to remove the smell (typically using a TTCN-3 refactoring [3]), and an example. In this paper, we gave an overview of our TTCN-3 code smell catalogue and presented excerpts from the full version [22]. We have implemented the automated detection of instances of TTCN-3 code smells in our TRex tool and demonstrated the applicability of our approach by assessing the internal quality of standardised test suites.

In future, we intend to extend our TTCN-3 smell catalogue by further code smells and also by more sophisticated high-level smells (e.g. smells related to issues in a test architecture). In parallel, we will implement further smell detection rules in TRex and evaluate their validity. The current smell detection rules are implemented in an imperative style in Java. To ease the implementation of further smell detection rules it is desirable to specify the code pattern that is described by a smell in a declarative way like the PMD tool [30] supports for Java-specific smells. Finally, we believe that it is worthwhile to investigate smells for other test specification languages, for example the *UML 2.0 Testing Profile* (U2TP) [32].

Acknowledgements. The authors like to thank Jens Grabowski and the anonymous reviewers for valuable comments on improving this paper.

References

1. ETSI: ETSI Standard (ES) 201 873-1 V3.2.1 (2007-02): 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 (February 2007)
2. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Computer Networks 42(3), 375–403 (2003)
3. Zeiss, B., Neukirchen, H., Grabowski, J., Evans, D., Baker, P.: Refactoring and Metrics for TTCN-3 Test Suites. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, pp. 148–165. Springer, Heidelberg (2006)
4. 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 (July 2005)
5. ETSI: Technical Specification (TS) 102 516 V1.1 (2006-04): IPv6 Core Protocol; Conformance Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (April 2006)
6. Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)
7. ETSI: Specialist Task Force 296: Maintenance of SIP Test Specifications. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2007)
8. ETSI: Specialist Task Force 320: Upgrading and maintenance of IPv6 test specifications. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2007)
9. ISO/IEC: ISO/IEC Standard No. 9126: Software engineering – Product quality; Parts 1–4. International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland (2001-2004)
10. Fenton, N.E., Pfleeger, S.L.: Software Metrics. PWS Publishing, Boston (1997)
11. Zeiss, B., Vega, D., Schieferdecker, I., Neukirchen, H., Grabowski, J.: Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications. In: Bleek, W.G., Raasch, J., Züllighoven, H. (eds.) Proceedings of Software Engineering 2007 (SE 2007), Bonn, Gesellschaft für Informatik. Lecture Notes in Informatics, vol. 105, pp. 231–242. Köllen Verlag (2007)

12. Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
13. van Emden, E., Moonen, L.: Java Quality Assurance by Detecting Code Smells. In: Proceedings of the 9th Working Conference on Reverse Engineering, pp. 97–106. IEEE Computer Society Press, Los Alamitos (2002)
14. Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: Anti-Patterns. Wiley, New York (1998)
15. Fosdick, L.D., Osterweil, L.J.: Data Flow Analysis in Software Reliability. ACM Computing Surveys 8(3), 305–330 (1976)
16. Gamma, E., Beck, K.: JUnit (February 2007) http://junit.sourceforge.net
17. van Deursen, A., Moonen, L., van den Bergh, A., Kok, G.: Refactoring Test Code. In: Extreme Programming Perspectives, pp. 141–152. Addison-Wesley, Boston (2002)
18. Meszaros, G.: XUnit Test Patterns. Addison-Wesley, Boston (to appear, 2007)
19. Zeiss, B.: A Refactoring Tool for TTCN-3. Master's thesis, Institute for Informatics, University of Göttingen, Germany (March 2006) ZFI-BM-2006-05
20. TRex Team: TRex Website (February 2007) http://www.trex.informatik.uni-goettingen.de
21. ETSI: ETSI Standard (ES) 201 873-10 V3.2.1: TTCN-3 Documentation Comment Specification. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (to appear, 2007)
22. Bisanz, M.: Pattern-based Smell Detection in TTCN-3 Test Suites. Master's thesis, Institute for Informatics, University of Göttingen, Germany (December 2006) ZFI-BM-2006-44
23. Baker, P., Evans, D., Grabowski, J., Neukirchen, H., Zeiss, B.: TRex – The Refactoring and Metrics Tool for TTCN-3 Test Specifications. In: Proceedings of TAIC PART 2006 (Testing: Academic & Industrial Conference – Practice And Research Techniques), Cumberland Lodge, Windsor Great Park, UK, pp. 90–94. IEEE Computer Society, Los Alamitos (29th–31st August 2006)
24. Eclipse Foundation: Eclipse (February 2007) http://www.eclipse.org
25. Parr, T.: ANTLR parser generator (February 2007) http://www.antlr.org
26. Eclipse Foundation: Eclipse Test & Performance Tools Platform Project (TPTP) (February 2007) http://www.eclipse.org/tptp
27. Johnson, S.: Lint, a C Program Checker. Unix Programmer's Manual, AT&T Bell Laboratories (1978)
28. Moha, N., Gueheneuc, Y.G.: On the Automatic Detection and Correction of Design Defects. In: Demeyer, S., Mens, K., Wuyts, R., Ducasse, S. (eds.) Proceedings of the 6^{th} ECOOP Workshop on Object-Oriented Reengineering. LNCS, Springer, Heidelberg (to appear)
29. Pugh, B.: FindBugs (February 2007) http://findbugs.sourceforge.net
30. Dixon-Peugh, D.: PMD (February 2007) http://pmd.sourceforge.net
31. van Rompaey, B., du Bois, B., Demeyer, S.: Characterizing the Relative Significance of a Test Smell. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006), Philadelphia, Pennsylvania, pp. 391–400. IEEE Computer Society, Los Alamitos (September 25–27, 2006)
32. OMG: UML Testing Profile (Version 1.0 formal/05-07-07). Object Management Group (OMG) (July 2005)