Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BM-2006-05

Masterarbeit

im Studiengang "Angewandte Informatik"

A Refactoring Tool for TTCN-3

Benjamin Zeiß

am Institut für

Informatik

Gruppe Softwaretechnik für Verteilte Systeme

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

15. März 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 15. März 2006

Master's thesis

A Refactoring Tool for TTCN-3

Benjamin Zeiß

March 15, 2006

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

Abstract

Software is continuously growing in size and getting more and more complex while development cycles are getting shorter. The resulting software aging effect leads to a decay in software quality causing maintainability problems and bugs. Software engineering techniques like modularization, object-orientation, design patterns, aspect-oriented programming or refactoring have been developed to slow down the software aging effects, increase flexibility and support reuse.

Software tests written in the TTCN-3 core notation also suffer from the effects of software aging. Huge efforts are spent to maintain TTCN-3 test suites, but even standardized test suites often consist of few files with a length of several ten-thousand lines each. *Refactoring* is a proven technique to systematically restructure code to improve its quality and maintainability while preserving the semantics which can also be applied to TTCN-3.

In this thesis, existing well known refactorings have been investigated for their applicability and a catalog with 20 specific refactorings for TTCN-3 is presented. In addition, a tool called TRex based on the Eclipse Platform has been implemented which provides the infrastructure for automating TTCN-3 refactorings and corresponding sample implementations of TTCN-3 refactorings.

Contents

1 Introduction

Software is critical for everybody's daily life and is continuously becoming more and more complex. In software development, techniques have been developed and proven to increase the development speed, improve flexibility, reduce costs and decrease the number of errors. Concepts related to these techniques are modularization, object-orientation, design patterns [40] or refactoring [35]. In addition, software tests (e.g. unit tests) are used to verify the correctness of software modules in an automated way. A tremendous amount of money is spent on quality assurance which includes such software tests.

The automated test suites must not only keep up with the development speed of the product itself, but often, they need to be improved as well in terms of test coverage. However, they suffer from the same effects that is visible in ordinary software. This effect is called *software aging* and describes the decay of quality within the code. In the worst case, this decay of quality makes the code unmaintainable. Even automatically generated test suites are affected when the actually generated code must be analyzed, e.g. when a test failed. In addition, maintenance of automatically generated test suites can be hard when they need to be manually modified, e.g. because the corresponding model is incomplete or too abstract to derive complete test suites.

The *Testing and Test Control Notation version 3* (TTCN-3) [31, 43] is widely used for test specification and implementation of distributed systems. While its predecessor TTCN-2 [44] has been primarily used for OSI conformance testing and the testing of other protocols in the telecommunications sector, the successor TTCN-3 is more flexible and not limited to this domain any more. In addition, the newly introduced core notation in TTCN-3 provides the flexibility of modern programming languages that are known to suffer from aging. Fortunately, the same or at least similar techniques from the classical software engineering can be used to fight these effects.

Refactoring [35] is such a technique. It provides a way to systematically restructure software in a behavior preserving way to improve its quality and therefore counteract the software aging process. In this thesis, refactoring is suggested as a reasonable method to improve the quality of TTCN-3 test suites and is comprised of the following contributions:

- Research on the applicability of well known refactorings for Java and a catalog presenting 21 refactorings specific to TTCN-3.
- An infrastructure for the realization of automated refactorings in TTCN-3 named TRex(TTCN-3 Refactoring and Metrics Tool).
- Two refactorings implemented on the basis of this infrastructure.

The structure of this thesis is as follows: following this introduction, the foundations for this thesis are given in chapter 2. This includes a short introduction to the TTCN-3 core notation as well as the technical base for the implementation of the refactoring infrastructure. ANTLR and the Eclipse Platform are the two of its key components.

In chapter 3, the process of software and test aging is explained. It is discussed how problematic code pieces can be identified and to what extent refactoring is a solution to these problems in TTCN-3.

A refactoring catalog for TTCN-3 is presented in chapter 4. A total of 72 refactorings for Java [35] have been studied for their applicability to TTCN-3. In addition, 21 refactorings specific for TTCN-3 have been developed and are presented in this chapter. These descriptions can be used for manual application of the refactorings, but also as a guideline for the implementation of automated refactorings which reduce errors and save time in comparison to manually applied refactorings.

The implementation of the infrastructure for automated TTCN-3 refactorings as well as the implementation of two concrete refactorings (*Rename* and *Inline Template*) is described in chapter 5. In addition, several functionalities have been implemented on top of the same infrastructure to enhance the programming and development experience with the TRex tool.

Finally, an overall summary and conclusion is given in chapter 6.

2 Foundations

Before TTCN-3 refactoring is dealt with in detail, the thesis foundations are presented in this chapter. It provides a brief introduction to TTCN-3 as a language for test specification and implementation and introduces the tools used for the implementation of TRex (i.e. ANTLR and the Eclipse Platform).

2.1 TTCN-3

TTCN-3 (Testing and Test Control Notation Version 3) is a test specification and implementation language standardized at the European Telecommunications Standards Institute (ETSI). It is the successor of TTCN-2 (Tree and Tabular Combined Notation) which was originally designed for the conformance testing of OSI (Open Systems Interconnection) protocol implementations and was widely used for the testing of telecommunication protocols like GSM, UMTS or DECT. These tests are black-box tests based on the protocol specification (i.e. no internal details about the implementation of the system under test (SUT) are known). TTCN-3 is more flexible than TTCN-2 and can also be used for other purposes such as the testing of internet protocols (e.g. IPv6 or SIP) or APIs while it retains the proven features. Although *system test* is still the main field of interest for TTCN-3, it is basically capable of handling lower level tests such as *integration tests* and *unit tests* as well. Test cases in TTCN-2 were written using a tabular notation. TTCN-3 introduces the core notation [31] which is a textual syntax similar to other modern programming languages. It is therefore easy to learn and easy to use for any programmer.

While the core notation may look similar to other modern programming languages, it is designed with testing in mind and contains concepts not part of other languages (e.g. build-in data matching, extended type system, integrated timer support or concurrent test component execution). The TTCN-3 core language can be represented in different formats besides the core notation. Two other presentation formats, the tabular presentation format [33] and the graphical presentation format [30], are standardized by the ETSI as well. The core notation is the only presentation format discussed in the following sections and chapters.

2.1.1 Language Basics

The most basic construct in TTCN-3 is the module. A module can contain a whole test suite or it may contain library code. Such library code can be used in other modules through

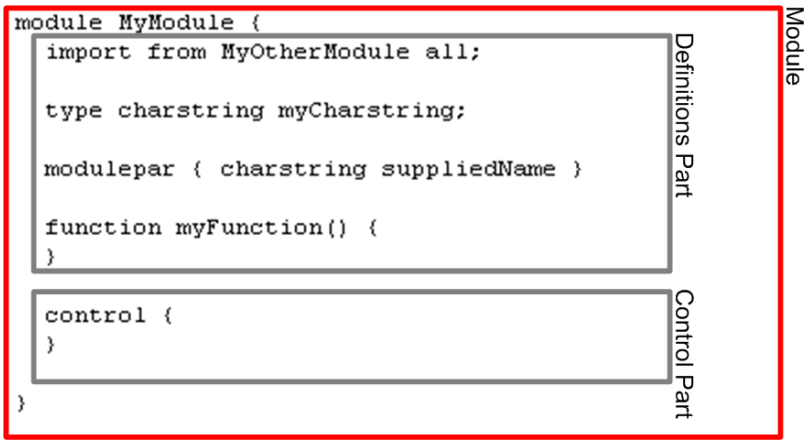


Figure 2.1: TTCN-3 Module Structure

the **import** statement. The **import** statement can be specified finely grained and therefore allows minimal interfaces between modules. Modules consist of a definitions part and a control part (of which both are optional). The definitions part contains declarations[1] for constants, types, templates, functions, altsteps, testcases, signatures and module parameters. Figure 2.1 illustrates the module structure.

TTCN-3 contains a very large number of built-in types. The most basic data types and most widely used ones are the types *integer*, *boolean*, *charstring*, *float* and *record*. The built-in data types can be used to declare user-defined types. Both, built-in data types as well as user-defined types, can be used to declare *templates* (data descriptions) which are subsequently used for the message-based communication using the *send* and *receive* operations on the ports of a test component.

Constants, types, functions and basic statements (such as conditionals, assignments or loops) are similar to other well known programming languages and are hence not discussed in this thesis (details can be found in [74] or [31]).

2.1.2 Concepts

Unlike a regular modern programming language, TTCN-3 contains functionalities and concepts that are specific for testing. The refactorings presented in chapter 4 are based on these specific concepts. Therefore, a subset of these concepts is introduced in this section.

[1]The difference of the terms ”definition” and ”declaration” in the context of TTCN-3 is not entirely clear as it is primarily a specification language rather than a programming language that allocates memory. Therefore, the term ”declaration” is used exclusively throughout this thesis to avoid confusion.

Subtypes

Subtypes are used to restrict the allowed range of an existing type declaration. This can avoid manual value verifications as subtype violations automatically lead to an error either at compile time or runtime. Listing 2.1 shows a few subtype examples. In line 1, a bit is defined as a range of integer values between zero and one. A second possibility is the use of value list as found in line 3. In this type declaration, the value of charstring MyStringList can only be one of the specified character strings. In line 5, the length of the string is limited to a length between 2 and 8 characters. There are several other subtyping mechanisms available (such as matching character patterns).

```
1 type integer MyBit ( 0 .. 1 );
2
3 type charstring MyStringList ("abcd", "rgy", "xyz");
4
5 type charstring MyString length( 2 .. 8 );
```

Listing 2.1: Subtypes

Components

A test configuration in TTCN-3 consists of one or more test components. Each component contains one or more communication ports which describe its interface. In the more simple test scenarios, the test system interface (TSI) of the system under test (SUT) is defined within a single component which is then called the main test component (MTC). Every test configuration has one MTC and may have any number of parallel test components (PTC). Initially, a test case is executed on the MTC. This test case may dynamically create PTCs using the **create** operation. In addition, components can have local constants, timers, templates and variables.

```
1  type port MyPort message {
2    in charstring
3  }
4
5  type component MyComponent {
6    port MyPort msgInPort;
7  }
8
9  type component MyExtendedComponent extends MyComponent {
10   var integer myValue;
11 }
```

Listing 2.2: Components

Listing 2.2 shows a component declaration. First, in lines 1–3, a port is declared. This port is able to receive character string messages. Secondly, the component is declared in lines 5–7. The previously defined port is used in this component and therefore declaring its interface. Finally, components can be extended, i.e. the extended component implicitly contains all declarations of the parent type. This concept is similar to inheritance of classes as known from Java. However, components do not contain any behavior and therefore there is no such thing as polymorphism in TTCN-3. Lines 9–11 show the extended component declaration. The extended component contains an additional integer variable called *myValue*.

Test Cases

Test cases are behavioral descriptions much like functions (in TTCN-3) or methods in other languages. However, they are specifically run on a test component (through the **runs on** clause). If only the MTC is instantiated, this test component provides the interface towards the SUT implicitly. Otherwise, the test system interface must be specified through the **system** clause. Every component contains an implicit local test verdict which can be used with the **setverdict** and **getverdict** operations. Test cases are executed from the control part of a TTCN-3 module. The control part is the entry point of the execution of a TTCN-3 module much like the *main* method in Java or C++.

```
1 testcase myTestcase() runs on myComponent {
2   setverdict(fail);
3 }
4
5 control {
6   var verdicttype v := execute(myTestcase);
7 }
```

Listing 2.3: Test Case

Listing 2.3 demonstrates the use of a test case using its local verdict. In line 1, a test case *myTestcase* is declared which uses the interface of the component *myComponent*. The use of the **setverdict** operation is shown in line 2. The local verdict of this test case is set to *fail*. Of course, real test cases would set the local verdict depending on the result of the test case behavior. The control part starting at line 5 is then used to execute the test case (line 6). The **execute** operation returns the test case verdict which can be stored in a variable and used for subsequent control part behavior.

Templates

On the one hand, templates are data descriptions for the messages that are send to the SUT or parallel components and on the other hand, they are used to test whether a received

message corresponds or matches to a template specification. In the simplest case, a template is just a concrete instance of a type with values assigned to it. However, templates can contain more complicated values such as patterns for matching a set of messages.

```
1  type record PersonType {
2    charstring firstName,
3    charstring middleName optional,
4    charstring lastName
5  }
6
7  template PersonType Turing {
8    firstName := "Alan",
9    middleName := omit,
10   lastName := "Turing"
11 }
12
13 template PersonType TuringFull modifies Turing {
14   middleName := Mathison
15 }
16
17 template PersonType MyPerson(charstring p_firstName, charstring p_lastName) {
18   firstName := p_firstName,
19   middleName := omit,
20   lastName := p_lastName
21 }
```

Listing 2.4: Templates

The declaration of templates is demonstrated in listing 2.4. The type declaration used by the template can be found in lines 1–5. There are several possibilities to declare templates. The most simple one is to just assign values to a template of a given type. This is shown in lines 7–11. The second way to declare templates is to use another template definition and only change the values that differ. These declaration are called modified templates (lines 13–15). Finally, templates can be parameterized as shown in lines 17–21 and inlined (i.e. an in-place value list notation directly within the test behavior). In the behavior, templates are typically sent and received using the **send** and **receive** statements.

Alt Statements

The **receive** operation or the **timeout** operation are blocking operations, e.g. when they are used as normal statement, they would block the execution of the remaining behavior until the specified message is received or a specified timer expires. The **alt** statement provides the possibility to provide alternatives for such blocking statements allowing fine grained error handling (e.g. when an unexpected message arrives) and alternative behavior for different messages received.

```
1  testcase myTestcase() runs on myComponent {
2    alt {
3      [] pt.receive(expectedMessage);
4        pt.send(answerMessage);
5      }
6      [] any port.receive {
7        setverdict(fail);
8      }
9    }
10 }
```

Listing 2.5: Alt Statements

Listing 2.5 demonstrates such an **alt** statement (lines 2–9). It is located within a test case *myTestcase* (line 1). Two alternatives are provided: either the message *expectedMessage* is received on port *pt* (line 3) and as a result a message *answerMessage* is sent back or an unexpected message is received on any port (line 6). In this case, the verdict is set to false (line 7).

Altsteps

Quite often **alt** statements resemble each other. For this reason, altsteps can be used to combine several **alt** statements. The duplicate alternatives are replaced with a reference to the name of the altstep. Actually, altsteps can be comprehended as a function especially for alternatives of an **alt** statement.

```
1  altstep myErrorHandler() {
2    [] any port.receive {
3      setverdict(fail);
4    }
5  }
6
7  testcase myTestcase() runs on myComponent {
8    alt {
9      [] pt.receive(expectedMessage);
10       pt.send(answerMessage);
11     }
12     [] myErrorHandler() { }
13   }
14 }
```

Listing 2.6: Altsteps

An example is given in listing 2.6. In line 1, an altstep with the name *myErrorHandler* is declared. The sole purpose of this altstep is to handle all message received on an unexpected port (line 2). As in the previous example (listing 2.5), the test case verdict is set to *fail*

(line 3) if such a message is received. This altstep is used in the test case *myTestcase* (line 7). The **alt** statement handles expected messages (lines 9–10) and then calls the altstep *myErrorHandler* to do the error handling (line 12).

As altsteps are often called at the end of **alt** statements, there is a concept called *default altsteps*. The concept is similar to the concepts of aspect-oriented programming. Altsteps are activated as default altstep using the *activate* statement and deactivated using the *deactivate* statement. Once, an altstep is activated, it is implicitly attached to the end of each **alt** statement.

2.2 ANTLR

ANTLR (Another Tool for Language Recognition) [1] is a tool using grammatical descriptions (similar to EBNF) of languages to generate code for lexical analysis (lexer) and syntactical analysis (parser) in Java, C++, Python or C#. Tools like ANTLR are used since typical hand-written lexers and parsers involve a lot of similar code making it a laborious task to write them. Code generators like ANTLR take this burden off the programmer and reduce the amount of work to specifying a grammar enhanced with *semantic action routines*. Semantic actions are code pieces that are associated to rules within the grammar and executed when the rules are applied by the parser. These actions, for example, support creating parse trees. In addition to the generation of lexers and parsers, ANTLR supports the generation of code to traverse syntax trees. This code is again generated from a grammatical description called *tree grammar* which essentially is a tree specification. Tree grammars can be enriched with semantic actions as well. For example, symbol tables, translators or pretty printers can be implemented using tree grammar actions. The resulting code is called *tree parser* or *tree walker*.

Figure 2.2 shows the processing order. First, text is read and tokenized by the lexer according to the rules specified in the lexer grammar. The result is a stream of tokens which is used as input of the parser. The parser matches the specified parser grammar rules on the token stream and uses either built-in functions or semantic actions to create an abstract syntax tree (AST) or a parse tree. The resulting tree is walked in a depth-first manner using the tree grammar specification. An example tree grammar is shown in listing 2.7. This grammar is used to evaluate a typical arithmetic expression syntax tree. In line 1, the *ExprTreeParser* is declared and extended from an abstract *TreeParser* class. The node specification for the expression node *expr* is located in lines 3–9. The tree parser

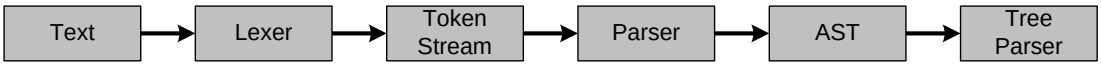


Figure 2.2: ANTLR Processing Order

evaluates only expressions using integer values. Therefore, the return value is an integer and is initialized with zero (line 3). *a* and *b* are declared as local integer variables in line 4 and are assigned return values in lines 5–7 or the token value in line 8. In lines 5–7, the expressions are evaluated within the curly brackets depending on the subnode (i.e. the return values are added when the subnode is a *PLUS* etc.). The syntax within the curly brackets is inlined into the generated code and hence is written in the target language.

```
1 class ExprTreeParser extends TreeParser;
2
3 expr returns [int r=0]
4 { int a,b; }
5     : #(PLUS  a=expr b=expr) {r = a+b;}
6     | #(MINUS a=expr b=expr) {r = a-b;}
7     | #(STAR  a=expr b=expr) {r = a*b;}
8     | i:INT                  {r = (int)Integer.parseInt(i.getText());}
9     ;
```

Listing 2.7: Simple Calculator Evaluation Tree Parser

Unlike other popular code generators for language recognition such as Lex and Yacc [14] which generate code on the basis of an LALR algorithm (bottom-up parsers), ANTLR uses a predicated LL(k) algorithm (top-down parsers). As a result, the generated code is easy to understand and similar to what a hand-written parser would look like.

ANTLR is developed by Terence Parr and colleagues since 1989 and was formerly known as *PCCTS*. It is available under the BSD license [13].

2.3 Eclipse

The term *Eclipse* is often used as synonym for the *Eclipse Platform* or a Java IDE, but in fact, *Eclipse* is essentially an open source community managed through the *Eclipse Foundation* that builds Java based tools with the Eclipse Platform being the most important and most popular subproject. The most important Eclipse project is the *Eclipse Project* which includes the *Eclipse Platform*, the *Eclipse Rich Client Platform* (RCP) and the *Java Development Tools* (JDT). The Eclipse nomenclature is therefore slightly confusing. Other important Eclipse projects include *AspectJ* (an aspect-oriented language extension for Java), the *Web Tools Platform* (WTP) which extends the Eclipse Platform with tools for the development of J2EE web applications or the *Eclipse Modeling Framework* (EMF) which is a modeling framework and code generation facility for building tools and applications based on a structured data model.

Eclipse was originally developed by IBM, but is now independent. In fact, many known tool vendors such as Borland or BEA joined the Eclipse foundation to support the Eclipse development. In TRex, only the *Eclipse Platform* is used.

Eclipse Platform

The Eclipse Platform is a generic tooling platform and software framework which can be conceived as the generic foundation for an integrated development environment (IDE). The popular JDT are a prime example for a successful IDE on the basis of the Eclipse Platform.

The architecture of the Eclipse Platform is shown in figure 2.3. At its core is the *Platform Runtime* which provides the plug-in infrastructure and is responsible for booting Eclipse as well as discovering, loading and managing the installed plug-ins. The plug-in infrastructure is based on the OSGi framework R4.0 specification [10]. It forms a framework for defining, composing and executing components or *bundles*. Bundles can be thought of as plug-in implementations. In fact, every Eclipse plug-in is a bundle and the term *plug-in* in the context of Eclipse is merely used for historical reasons as the Eclipse vocabulary (e.g. in the documentation) was not adjusted when the underlying plug-in infrastructure was switched to an OSGi framework implementation (project *Equinox*) in Eclipse 3.0.

One key characteristic of the Eclipse Platform is the fact that almost every functionality is implemented as plug-in (also called *extension*). Plug-ins are loaded lazily, i.e. they are only loaded when they are really needed. As a result, the number of installed plug-ins has only little effect on the startup time of Eclipse. Instead of loading all classes belonging to a plug-in at startup, only a plug-in XML specification located in *plugin.xml* files are loaded. In this specification, there is a definition of extensions the plug-in provides for certain extension points. These extension points are either part of the Eclipse Platform or of other plug-ins. Hence, plug-ins can in turn be extended by other plug-ins if they provide their own extension points. Eclipse features are a set of plug-ins or other features belonging together. They are mainly used for distribution and branding. Distribution of a product's Eclipse plug-ins is typically realized through an *update site* which is a directory on a web server with a fixed layout and an XML specification file *site.xml* containing information about the contents of the update site. These update sites are feature driven, i.e. update sites distribute features and not plug-ins directly. Feature-based branding is concerned with customizing the appearance of an Eclipse product, e.g. using custom icons. Features are also specified through an XML file called *feature.xml*. Thus, a lot of Eclipse configuration is driven through XML. Fortunately, an Eclipse Developer does not need to deal with the corresponding XML Schemas directly in most cases as Eclipse provides plug-ins especially designed for Eclipse development. These plug-ins are called PDE (Plug-In Development Environment). It adds views and wizards for creating, maintaining and publishing Eclipse plug-ins and takes care of the XML generation.

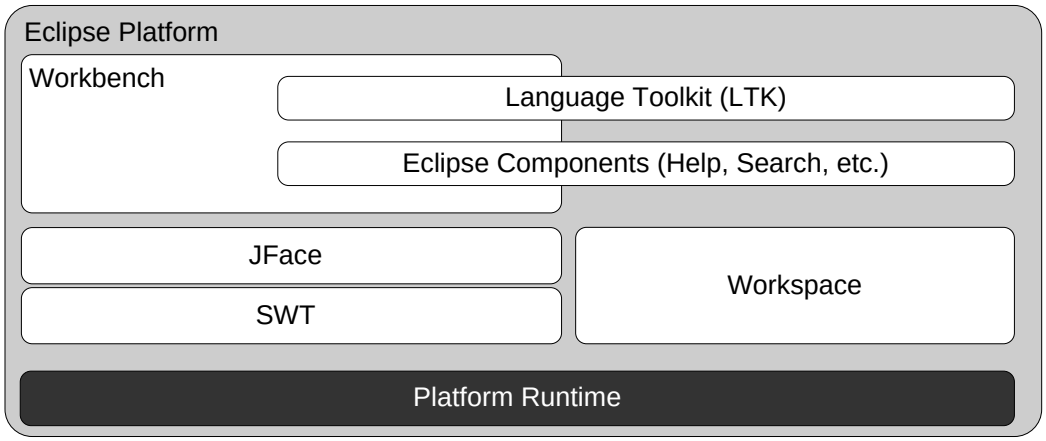


Figure 2.3: The Eclipse Platform

On the right of figure 2.3 is the workspace which manages one or more projects. A project is comprised of files and folders which (in Eclipse 3.2) map to an abstract file system layer (e.g. the local file system). When developing plug-ins using PDE, Eclipse differentiates between the *host workbench* (the one which is used for working on the plug-in sources) and the *runtime workbench* which is used when a second eclipse instance is started for testing the plug-in.

The Standard Widget Toolkit (SWT) provides a standard widget set for the user interface based on the native elements of the underlying operating system. JFace is built on top of SWT and supports common UI tasks, e.g. by offering classes for wizard generation. In comparison to Swing which is part of the Java API, SWT and JFace provide a native look and feel (i.e. they use the widgets offered by the operating system) for each platform and the UI typically feels a little more responsive than Swing UIs. On the downside, SWT uses JNI (Java Native Interface) for platform specific binary code on each platform.

The Workbench is the actual user interface which is the base for any IDE developed on top of the Eclipse platform. The UI paradigm centers around editors, views and perspectives. A view is the part that makes up a window in the Workbench. Editors are actually a similar concept as views. The main difference is that editors can only appear in one region of the Workbench, they have different content states (e.g. dirty state for an editor with modified content), editors can be associated with a filename and multiple instances of an editor can be opened. In addition, several abstract editors are provided making it easy to create source editors with syntax highlighting for example. Figure 2.4 shows a screenshot of a typical Eclipse workbench. On the left is the *Package Explorer* view which displays projects and the classes/files belonging to each project, but based on packages instead of folders. In the center, there is a Java source editor, on the right the *Outline* displaying

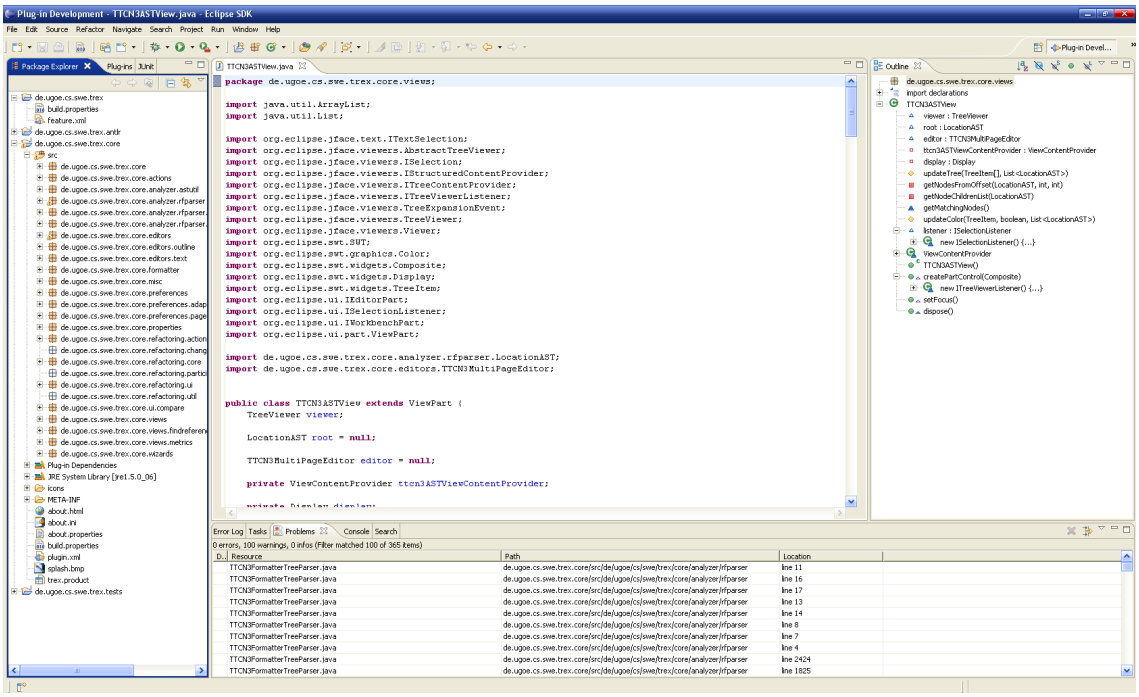


Figure 2.4: Eclipse Workbench

clickable declared elements within the opened Java source. On the bottom are typically views like the *problem view* displaying the results of on-the-fly syntactical and semantical analysis. View arrangements as just described can be stored and are called *perspectives*. For example, when debugging a Java program, another more appropriate debugging perspective is used which contains views with the variable contents or breakpoints.

Several other components are part of the Eclipse platform. This includes typical IDE functionality such as a search dialog or an online help. These components are partially based only on the core or core-based extensions and partially depended on the workbench. For TRex, the Language Toolkit (LTK) component is of particular interest as it provides a framework for the implementation of semantic preserving workspace transformations.

While the Eclipse Platform was originally designed to be a tooling platform, it has also become a generic software platform for rich-client applications (as opposed to thin-client applications). The Eclipse Platform is released under the Eclipse Public License (EPL) [5].

3 Evolution of Software and TTCN-3 Tests

Software aging [63] is a common cause for software failures and system outages. The degradation of software over time is exhibited in reduced extensibility, reusability, maintainability, efficiency and increased complexity. This process is inevitable as software is complex and never free of errors. Verification that a software system is completely bug free is usually impossible. Even successful and well designed software is affected by this process. It is expensive to change software properly and new customer-demanded functionality is therefore often added without adapting the architecture accordingly to reach short-term goals. The code loses its structure in a cumulative way and maintenance of such a software becomes a burden, changes are difficult and the software becomes unreliable.

TTCN-3 test suites are also complex pieces of software. They only have the special purpose to black-box test other software or hardware systems. However, while it is true that regression tests, for example, are usually run with the same test suites for comparative reasons, test suites are nevertheless under constant development to improve the test coverage on the one hand and on the other hand to reflect the evolution of the SUT. TTCN-3 test suites may also be generated from specifications on model level (e.g. UML models) and they are high quality in terms of correctness. Unfortunately, they are hard to understand and reuse. In practice, changes to test suites are rarely implemented at model level, but directly in the generated core notation. As a result, test engineers often deal with generated or outgrown hand-written test suites that must be extended and maintained on core notation level. The same problems apply as in ordinary software: to reach short-term goals, test suite maintenance is disregarded and the test suites lose structure. Similar to ordinary software, the test suite verdicts may become unreliable. While this may not be instantaneously impact the product (the SUT), there are still subtle consequences:

- Defects may be reported that are actually working properly and as a result unnecessarily consume time of the development team.
- Real defects covered through the test suite may not be detected and the SUT quality decreases.
- Unmaintainable test suites can not be extended or changed and the resulting effects are:
  - The test coverage can not be improved. Uncovered bugs stay undetected.
  - The test suite can not be adapted to the evolving SUT.

Hence, software aging and disregarded maintenance of test suites also directly affects the quality of the tested product. This is especially the case for TTCN-3 test suites. Unlike its predecessor TTCN-2 which guided test development and structure due to its predefined tables, the TTCN-3 core notation has the look and feel of a general purpose programming language. Hence, TTCN-3 gains the flexibility of modern programming languages and offers the test engineer a lot of freedom, but also inherits the associated maintenance challenges.

There are techniques to slow down the aging process though. *Refactoring* has been a successful technique for continuous systematic restructuring of software on code level to improve its quality.

3.1 Bad Smells

Before systematic software restructuring is possible, the problematic code parts must be identified first. The indicators that lead to finding those parts are called *bad smells* [35]. There are various ways to find *bad smells*. Currently, the most important techniques are the usage of software metrics, pattern recognition and human intuition. Fowler [35] claims that no set of metrics rivals the informed human. Still, informal code smell descriptions are useful as starting point for creating formalized metrics and pattern recognition algorithms and therefore a few useful ones are presented subsequently.

The most important bad smell is **Duplicated Code**. Any expressions which are exactly the same and occur at several different locations are better when they are unified as a change is only needed in one place. The **Long Function** [Long Method][1] smell states that long functions are hard to understand and indirection can be better supported by small functions. As a guideline, a subroutine should be moved when there is a need to write a comment otherwise. Actually, **Comments** is a smell by itself. When something needs to be explained, it is probably better to move into a method with a descriptive name. The **Long Parameter List** smell criticizes that long parameter lists are hard to understand, become inconsistent and difficult to use as they are always changed when more data is needed. A **Large Module** [Large Class] indicates that the module may be trying to do too much. It becomes hard to read and it likely combines too many distinct code parts which are better off in multiple modules. Several other smells in [35] concern object-oriented concepts and paradigms and cannot be used in TTCN-3.

Bad smells are closely related to antipatterns [24]. Antipatterns are organized and written similarly to design patterns, but actually describe the opposite: solutions to reoccurring problems that are wrong and bad practice. The intention is to show by example what should not be done. The most famous antipattern is probably *Spaghetti Code* which is described as software with little structure due to ignorance and sloth as root causes. Unlike

[1] The bad smells from [35] were described with Java in mind, but are in parts applicable to TTCN-3– either directly or slightly reinterpreted. However, it is necessary to rename some smell names to reflect their meaning in the context of TTCN-3. The original names are given in square brackets in these cases.

bad smells, antipatterns are also applied on processes outside software engineering such as project management. They are therefore more general and not as code centric as bad smells.

```
1  testcase tc_firstExampleTestCase() 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 }
19
20 testcase tc_secondExampleTestCase() runs on ExampleComponent {
21   timer t_guard;
22   // ...
23   t_guard.start ( 10.0 );
24   alt {
25     [ ] pt.receive( a_MessageThree ) {
26       pt.send( a_MessageFour );
27     }
28     [ ] any port.receive {
29       setverdict( fail );
30       stop;
31     }
32     [ ] t_guard.timeout {
33       setverdict( fail );
34       stop;
35     }
36   }
37 }
```

Listing 3.1: Bad Smell: Duplicated Code

In Figure 3.1, a typical bad smell in TTCN-3 is shown. In this example, there are two test cases *tc_firstExampleTestCase* and *tc_secondExampleTestCase* which are very similar[2]. Each test case contains an **alt** statement where it communicates with the SUT. At the end of each **alt** statement, there are two branches treating unexpected behavior (lines 9–35 and

[2]The comments in lines 3 and 22 indicate there might be additional behavior or code in general at these locations that is irrelevant for the example. This notation is also used in the subsequent sections and chapters.

lines 28–35) which is exactly the same in each **alt** statement and therefore duplicated. A programmer would intuitively want to move such code segments into an altstep.

Although [35] suggests to find bad smells using the programmers intuition, there are several reasons why automatic bad smell detection is favorable [49]. First, there must be developers who spend their time on code analysis. As a result, either the actual development speed and productivity slows down or there are additional costs for more personnel. In commercial situations, both alternatives are hardly acceptable. In addition, the longer a developer participates in a project, the more he loses the ability to objectively judge the quality of its code. Hence, there is ongoing research into the automatic detection of bad smells and defects [56, 57].

3.2 Refactoring

Refactoring is the term used for the systematic way of restructuring source code in order to improve its quality. A concrete definition of refactoring is given in [35]: "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". Hence, another important aspect of refactoring is the preservation of the semantic context in the original code. As opposed to simple code cleanup, refactoring provides a more efficient and controlled way for the source transformation by providing a mechanic which can be executed step by step either manually or automatically. The changes in each step of the mechanics are predictable. In addition, refactorings are always applied with the intention to make the code clearer and better to understand. Thus, it mitigates the effects of software aging and helps in keeping the software maintainable. This does not necessarily mean that the resulting code is more efficient performance wise. On the contrary, some refactorings arguably deteriorate the code performance. On the other hand, performance optimizations often result in code that is hard to understand. As a result, there is sometimes a trade-off between performance and code quality. This is not always the case though as [75] proves. Also, to actually optimize the performance of software which is considered too slow, it must be understood well and in this case, refactoring the code will help.

Some roots of refactoring [64] are actually found in compiler optimization and therefore code transformations can be actually designed to improve performance (e.g. inlining a function call). Other origins are program transformation (e.g. stepwise refinement), maintenance of object-oriented databases (e.g. schema evolution). The first detailed work on refactoring was written in 1992 by William Opdyke [61] and treats refactoring of C++ code. The technique has been successfully applied and automated [65] in the *Smalltalk* community [19]. Popularity was finally achieved through the book "Refactoring" by Martin Fowler in 1999. This book presented a catalog containing 72 refactorings for Java (with additions on its website [36]).

Most refactorings are independent from the language actually used. However, each language has its own paradigms and concepts which result in specific refactorings. For example, TTCN-3 is not an object-oriented language and therefore a lot of refactorings from [35] that concern class relationships are not applicable to TTCN-3. TTCN-3 on the other hand includes concepts (section 2.1.2) specific for writing test cases and test suites which are not found in any other language. Hence, there are several refactorings for TTCN-3 (e.g. refactorings operating on templates) that are unique.

A simple example for a refactoring is the *Rename* refactoring which provides a simple way to rename identifiers such as variable names, function names or module names while taking the scoping rules of the underlying programming language into consideration. The *Encapsulate Field* refactoring replaces direct field accesses by calls to corresponding getter and setter methods or other refactorings are concerning with the extraction of a group of statements into a function. More complex refactorings are sometimes composites of smaller refactorings. For example, the conversion of a procedual design into an object-oriented design involves encapsulating fields, conversion of record types into classes or move and extracting behavior into class methods. The bad smell shown in figure 3.1 can be removed by applying the *Extract Altstep* refactoring (section 4.2.1).

3.3 Refactoring Formalisms

One problem often recognized in the context of refactoring is the fact that semantical preservation cannot be taken for granted. In fact, [52] identifies missing formal proofs as one of the key problems in the current research concerning refactorings. The initial work by [61] and [66] provide only a formal way to express conditions written in predicate calculus that must be satisfied to guarantee that a change is behavior preserving. However, they fail to proof formally that the refactoring itself is in fact behavior preserving. It is merely argued that the provided conditions satisfy seven properties which are related to inheritance, scoping, type compatibility and semantic equivalence. In fact, Tokuda [72] recognizes that the invariants introduced by Opdyke are not sufficient, argues that a mature refactoring implementation should be treated as trusted tool and compares the refactoring transformations to compiler transformations to assembly where also typically no mathematical proof is provided. Compilers are still used despite this fact.

The authors of [66] and [35] take the position that a program is correct when it meets its specification. Such as specification can be provided by a test suite, e.g. unit tests. Therefore, a refactoring preserves the behavior of a program when it still meets its specification after the application of a refactoring. Especially in agile development processes such as *Extreme Programming* (XP) [20], the unit testing and refactoring interaction is a key concept. For refactoring test cases, this poses a problem as unit testing against a test suite is not practical or even possible. Van Deursen et al. [73] suggest to run the test suite which is subject of a

refactoring against the same implementation before and after the refactoring. Afterwards, the same verdict should be assigned to the test suite. This solution, however, is not sufficient since not all paths of the test suite may be executed. *Bisimulation* [55, 62] of the refactored and the original test suite may be a possibility to prove the equivalence of both. Whether this solution is practical remains to be seen.

Other more suitable techniques for proving behavior preservation are based on concept analysis [69] and graph rewriting [51]. In addition, there is one recent work [27] that deals with behavior preservation of refactorings using algebraic refinement rules. However, there is no single genuinely accepted method by all researchers. Therefore, proofing behavioral preservation itself is still a topic for ongoing research and out of the scope for this thesis. Nevertheless, 20 refactorings specific for TTCN-3 have been identified and informally described in chapter 4.

3.4 Refactoring Automation

Refactoring provides a disciplined way for code restructuring with the goal to improve the design while the meaning stays the same. However, the process has problems when it is applied manually. Although the mechanic provides small predictable steps, it is still easy to make mistakes in the execution. Therefore, a tool that automates these steps is desirable:

- Automated refactorings are time-saving as the programmer does not have to apply all steps from the concerned mechanic by hand. Especially refactorings like *Rename* can be very time consuming by hand.
- They are supposed to guarantee a certain level of correctness in the transformation.

The latter reason for automated refactorings raises a critical concern. The programmer must have confidence that the automated refactorings really do what he expects. Automated refactorings can automatically verify conditions like the ones described in [61] and therefore improve the trust in the transformation. As a result, running unit tests or other verifications is not necessarily needed after each refactoring. As tool support makes refactoring inexpensive with respect to the time spent on the restructuring (as opposed to manually applied refactorings), it becomes an activity that is used during programming rather than a separate activity.

There are various well working refactoring tools available. The Java Development Tools (JDT) [29] and IntelliJ IDEA [45] are two prime examples for refactoring tools for Java. However, current refactoring tools are only semi-automatic. Bad smells are not automatically detected and refactoring configuration (e.g. the name of the target identifier in the *Rename* refactoring) has to be specified by the programmer. Particularly automatic bad smell detection is likely to be a part of next-generation refactoring tools.

3.5 Related Work

Existing work on refactoring deals almost exclusively with the refactoring of production source code rather than tests (e.g. [35, 61, 66]). The refactoring of unit tests is often informally recommended in publications dealing with agile development processes such as *Extreme Programming* (XP) [20] where the tests are written and maintained continually. The context is different in most cases though: the unit tests verify that a refactoring did not change the behavior of the implementation.

The article "JUnit Best Practices" [8] deals with best practices for testing with JUnit [39] and does not specifically cover refactoring of test codes. The article, nevertheless, identifies problems in test code that can be considered as bad smells and gives advice how to avoid the associated problems. The identified problems are again specific for unit tests and cannot be applied to TTCN-3.

There is only one single publication dealing with the refactoring of unit test code [73]. The authors find refactoring test code different from refactoring production code and present a distinct set of 11 bad smells and 6 specific refactorings that apply to unit tests. The refactorings are concerned with the handling of external test resources (e.g. external files used for testing), the minimization of needed test data, explanatory log messages for assertions and the implementation and usage of equality methods rather than *toString* methods. The *Add Assertion Explanation* refactoring presented in this paper is the only refactoring which can be applied to TTCN-3 test code. All other refactorings deal with problems specific for unit tests.

A few publications, however, deal with transformations in TTCN-3 and its predecessor TTCN-2 that can be regarded as refactoring. Schmitt [68] involves the automatic structuring of TTCN-2 constraint descriptions and presents an algorithm of 4 steps in which each step is comprised of a transformation that can effectively be considered as a refactoring. Although his thesis deals with constraints instead of TTCN-3 templates as data descriptions, some of these steps can be reformulated into refactorings and are therefore part of section 4.2.

The paper "High-Level Restructuring of TTCN-3 Test Data" [75] presents an algorithm to automatically restructure TTCN-3 test data definitions (i.e. templates) to reduce redundancy and their length to improve run-time inefficiencies. According to their empirical experiences, test data definitions occupy at least 60-70 percent of a complete test specification and are highly redundant. Due to this fact and the situation that current TTCN-3 compilers only insufficiently perform code optimization, the time for the compilation of a TTCN-3 test suite dramatically sets back the development process. Therefore, a model for TTCN-3 templates is introduced and a graph-based algorithm operating on this model which removes the redundancies. Therefore, the paper is more concerned with optimizing the compilation process rather than improve the quality of the test code itself. The work nevertheless identifies that there are problems involved in the way TTCN-3 test suites are currently written or generated which can be improved through restructuring of the test code. Both [68] and

[75] are based on concepts that are available in TTCN-2 as well as TTCN-3. These involve specialization, parametrization and referencing of constraints and templates.

A presentation from Deiss [28] is concerned with the automated conversion of a TTCN-2 test suite to TTCN-3. The resulting generated code contains awkward constructs such as altsteps only containing a single else branch with a send statement. Such constructs are improved by applying refactoring-like transformations (e.g. transforming such an altstep into a function). The improvements presented are specialized for automatically converted test suites are not generally useful or too general. The conversion of test suites from TTCN-2 to TTCN-3 is motivated by the fact that TTCN-3 offers better modularization and flexibility and therefore reduces the maintenance.

Mäki-Asiala [48] investigates the reusability of TTCN-3 code. He differentiates between vertical reuse (reuse between different testing levels, e.g. reusing tests in functional testing and scalability testing), horizontal reuse (reuse of tests between various products) and historical reuse (reuse of tests between product generations). He proposes 10 non-systematic guidelines to improve TTCN-3 code reusability.

Neukirchen [59] suggests to apply reuse techniques from classical engineering to test development as well to improve productivity, quality, development speed, preservation of knowledge and to reduce costs. The reuse concepts *Modules*, *Object-Orientation*, *Aspect-Orientation*, *Refactoring*, *Applications*, *Components*, *Libraries*, *Frameworks* and *Patterns* from classical engineering are analyzed for their relevance to test development with TTCN-3, the UML Testing Profile [60] and JUnit. Refactoring is considered a reasonable technique for test development as large test suites suffer from aging just like ordinary software and huge efforts are spent for the maintenance of standardized test suites. To ensure the behavior preservation, bi-simulation and tool supported or formally proven transformation steps are proposed. It reveals that refactoring for TTCN-3 has not yet been studied while the refactoring of JUnit tests and UML models is well known and supported by tools.

4 A TTCN-3 Refactoring Catalog

To counteract or at least slow down the effects of software aging within TTCN-3 test suites, refactoring can be used for TTCN-3 tests as well. With the TTCN-3 refactorings presented in this chapter, test suite maintenance and readability is improved and their application is reasonable for both hand-written tests or generated tests. The refactorings are presented in the form of a catalog which is inspired by the catalog in Fowler's book [35]. Hence, the same fixed format is used for the refactoring descriptions: each refactoring is described by its *name*, a *summary*, a *motivation*, *mechanics* and an example. The name of a referenced refactoring is always written in *slanted* type and is used to build a common vocabulary for developers. The summary is a short description of the refactoring in one or two sentences. The motivation explains when and why a refactoring can be applied. In addition, the motivation may refer to possible subsequent refactorings. The mechanic contains precise step by step instructions on how to perform the refactoring. In the mechanics section, the term "source" is used to refer to the original code which is addressed by a refactoring or a location or elements in the original code which are used for the change. The term "target" refers to the resulting code after applying a refactoring to the source or other elements that are changed due to the refactoring. The terms "source" and "target" may also be used to refer to unchanged declarations whose references are replaced in a refactoring. The mechanics can be used as a manual step-by-step instruction for predictable restructuring or as a guide for tool implementation. Since manual refactorings are error-prone, the mechanics contain instructions "compile" and "validate". The compile step is used to check whether syntax and static semantics of the test cases are still valid. The validate step means to start a bisimulation process (i.e. validation of test equivalence using all possible execution paths) to validate that original and refactored test suite still behave equivalently. The compile and validate steps are suggested as soon and as often as they are applicable. The example section illustrates the refactoring by showing TTCN-3 core notation excerpts before and after the refactoring is applied.

The refactorings presented in this chapter can be classified into refactorings for test behavior, refactorings for data descriptions and refactorings improving the overall structure of a test suite. Refactorings for test behavior simplify behavioral statements or reduce behavioral duplication for example. Refactorings for data descriptions concern primarily test data in TTCN-3 such as templates, but also subtyping. Refactorings for improving the overall structure of a test suite are concerned with modularization, proper naming or the correct locations of declarations for example.

4.1 General Refactorings Applied to TTCN-3

In this section a list of classical refactorings is presented which can be applied to TTCN-3. For this purpose, all 72 refactorings from [35] have been investigated for their relevance. Even though they were intended for Java, some of them are language independent or can be reinterpreted in a way that they are applicable to TTCN-3. For their reinterpretation, it is necessary to replace the notion of Java *methods* by TTCN-3 *functions* or test cases. As TTCN-3 lacks most important object-oriented features in the conventional sense (e.g. classes, polymorphism), many well known refactorings cannot be applied. However, some of the Java refactorings are nevertheless applicable if the notion of Java *classes* and *fields* is replaced by TTCN-3 *component types* and *variables*, *constants*, *timers* and *ports* local to the component respectively.

There are some refactorings which can be applied in the TTCN-3 language, but lack a comparable code improving effect. The *Encapsulate Collection* refactoring, for example, could surely be applied by wrapping access to an array, for instance, in functions on a module level (instead of classes). However, the use would be limited as array sizes can only be assigned at declaration and if there is really a demand to hide internal data structures, it is likely the language is used in the wrong context. Therefore, such borderline cases are omitted within this list of reinterpreted refactorings.

As a complete catalog would be a book in itself, the extent of the chapter is limited to one exemplary refactoring (section 4.1.2) and a list of the ones working (4.1.1).

4.1.1 TTCN-3 Compatible Classical Refactorings

In the following, a list of 28 refactorings from [35] is presented which can be applied to TTCN-3 with only little or no modification apart from syntactical differences.

If a reinterpretation as described is needed, the original name from [35] is given in square brackets:

Refactoring for Test Behavior

- Consolidate Conditional Expression,
- Consolidate Duplicate Conditional Fragments,
- Decompose Conditional,
- Extract Function [Extract Method],
- Introduce Assertion,
- Introduce Explaining Variable,

- Inline Function [Inline Method],
- Inline Temp,
- Remove Assignments to Parameters,
- Remove Control Flag,
- Replace Nested Conditional with Guard Clauses,
- Replace Temp with Query,
- Separate Query From Modifier,
- Split Temporary Variable,
- Substitute Algorithm.

Refactorings for Improving the Overall Structure of a Test Suite

- Add Parameter,
- Extract Extended Component [Extract Subclass],
- Extract Parent Component [Extract Superclass],
- Introduce Local Port/Variable/Constant/Timer [Introduce Local Extension],
- Introduce Record Type Parameter [Introduce Parameter Object],
- Parameterize Testcase/Function/Altstep [Parameterize Method],
- Pull Up Port/Variable/Constant/Timer [Pull Up Field],
- Push Down Port/Variable/Constant/Timer [Push Down Field],
- Replace Magic Number with Symbolic Constant,
- Remove Parameter,
- Rename [Rename Method][1],
- Replace Parameter with Explicit Functions [Replace Parameter with Explicit Methods],
- Replace Parameter with Function [Replace Parameter with Method].

[1] Fowler [35] refers only to renaming a method. However, not only corresponding altsteps, test cases and functions qualify for renaming, but also variables, types, constants, ports, etc.

Note that some refactorings like *Move Method* or *Move Field* don't work in the sense they are presented in the book. However, they do with modifications in their interpretation. Instead of diffusing and widening their meaning, they are reintroduced with a more precise name (e.g. *Move Method/Field to Module*) in chapter 4.2.

Unsurprisingly, there are no refactorings from [35] that concern data descriptions. Refactorings for data descriptions include concepts which are unique to TTCN-3 (e.g. templates) and not part of Java. However, some of Fowler's refactorings like *Inline Method* or *Add* and *Remove Parameter* are quite generic and may also be reinterpreted for TTCN-3 templates. Where the mechanics of these refactorings differs significantly when applied to templates, they are considered as TTCN-3 specific refactorings and described in section 4.2.

An example for additional complexity in TTCN-3 and therefore changed mechanics is that functions can be run globally or on components. In section 4.1.2, a classical refactoring with such a change in mechanics is presented. Altogether, these changes are manageable though and this single example is sufficient as a demonstration.

4.1.2 Extract Function

A group of statements can be moved into its own function.

Motivation

Typical smells for this refactoring are long functions or test cases and repeated code fragments. Extracting parts from a long function or test case increases the possibility for reuse or on the other hand may reduce code duplication. In either case it improves reusability, readability and reduces maintenance effort. Method names should be chosen with care as they should exactly represent the semantics of the method.

Usually, the concerned statements are extracted into a TTCN-3 function. Only in rare cases, however, the statements may be extracted into a test case. Hence, the refactoring name reflects only extraction into a function.

Mechanics

- Create a new function and name it by what it does.
- Copy the extracted code from the source location into the new target function.
- Examine the extracted code for references to any variables, constants and timers that are local in scope of the source location, i.e. either locally defined temporary elements or elements passed as parameters to the source function. Port cannot be locally defined, but passed as parameters; in this case, they have also local scope.

  - Locally defined temporary elements which are only used in the extracted code are declared in the new target function again as local temporary elements. In a later step, their declaration can also be removed from the source location.
  - Temporary local-scope elements which are used before and after the extracted code must be passed to the target function as follows:
    * Timers and ports are always passed as reference, i.e. as **inout**.
    * Elements which are only read but not modified by the extracted code or read and modified by the extracted code but not read afterwards at the source location are passed as **in** parameter.
    * Elements that are modified but not read in the extracted code and read afterwards at the source location can be treated in two different ways: either they are passed as **out** parameters or they are returned by the function. The latter solution is only possible if only a single elements needs to be returned though. Other possibilities to improve the code for refactorings in these cases are the *Split Temporary Variable* and *Replace Temp with Query* refactorings. Mixing the return statement and out parameters is not recommended as it makes the function harder to understand and has to functional advantage.
    * Elements that are both read and modified in the extracted code and also read afterwards need to be passed as **inout** parameters or using **in** parameters and a return value if applicable.

- Check whether the source method runs on a component. If so, find out whether any component specific variables are used in the extracted code. In this case, the extracted method needs to run on this component as well. Hence, the **runs on** specification from the source behavior needs to be added. Afterwards, *Generalize Runs On* may be appropriate or the **runs on** specification may even be removed by applying *Move Component Variable/Constant/Timer to Local Scope* (section 4.2.20).

- Compile.

- Replace the extracted code in the source behavior with a call to the new target function and provide any required parameters.

  - If any declarations of local temporary elements have been copied to the target function, remove their declaration from the source compound.
  - If the extracted code is duplicated at other locations, repeat the previous step for these alternate source locations. If these additional duplicated locations are not in the same module as the extracted target function, a corresponding import statement must be added or adjusted in the module of the additional duplicated code locations to import the target function.

- Compile and validate.

Example

Sometimes, there needs to be a delay between two successive events in a TTCN-3. In listing 4.1, it is easy to detect the code duplication in the timer code of the *f_sendMessages* method (lines 15–16 and 20–21). The *Extract Function* refactoring is used remove this duplication.

```
1  module ExtractFunctionExample {
2    // ...
3
4    type component ExampleComponent {
5      timer t;
6      port ExamplePort pt;
7    }
8
9    // ...
10
11   function f_sendMessages(in float p_duration)
12     runs on ExampleComponent {
13
14     timer t;
15     t.start( p_duration );
16     t.timeout;
17
18     pt.send( a_MessageOne );
19
20     t.start( p_duration );
21     t.timeout;
22
23     pt.send( a_MessageTwo );
24   }
25 }
```

Listing 4.1: Extract function example (unrefactored)

The result of the applied *Extract Function* refactoring to remove the code duplication is shown in listing 4.2. The new function *f_wait* contains the extracted code. According to the mechanic, the *duration* element is passed as **in** parameter as it is read, but not modified within the extracted code. As the timer accessed within the extracted code is a local element of the component, the *f_wait* function needs to run on *ExampleComponent* as well.

```
1  module ExtractFunctionExample {
2    // ...
3
4    type component ExampleComponent {
5      timer t;
6      port ExamplePort pt;
7    }
8
9    // ...
```

```
10    function f_wait(in float p_duration)
11      runs on ExampleComponent{
12      t.start( p_duration );
13      t.timeout;
14    }
15
16    function f_sendMessages(in float p_duration)
17      runs on ExampleComponent {
18      timer t;
19
20      f_wait(p_duration);
21      pt.send( a_MessageOne );
22
23      f_wait(p_duration);
24      pt.send( a_MessageTwo );
25    }
26  }
```

Listing 4.2: Extract function example (refactored)

4.2 TTCN-3 Specific Refactorings

In the following, refactorings are introduced which are specific to TTCN-3. While the refactorings presented in the previous section were essentially language independent, the ones presented in this section consider language constructs specific to TTCN-3 such as altsteps, templates, grouping, modules and importing from modules, components, restricted sub-types, logging and concurrent test cases. Some of the techniques presented in the following are not new and have been introduced in [68] or [74] for example. However, these code optimizations have never been presented as refactorings providing a mechanical step by step instruction to clean up the code. Until now, 21 refactorings specific to TTCN-3 have been identified. The 21 refactorings are grouped as follows (using the categories introduced in section 4) and are explained in detail in the following sections:

Refactoring for Test Behavior

- Extract Altstep,
- Split Altstep,
- Replace Altstep with Default,
- Add Explaining Log,
- Distribute Test.

Refactorings for Data Descriptions

- Inline Template,
- Inline Template Parameter[2],
- Extract Template,
- Replace Template with Modified Template,
- Parameterize Template,
- Decompose Template,
- Subtype Basic Types.

Refactorings for Improving the Overall Structure of a Test Suite

- Extract Module / Move Declarations to Another Module,
- Group Fragments,
- Restrict Imports,
- Prefix Imported Declarations,
- Parameterize Module,
- Move Module Constants to Component,
- Move Local Variable/Constant/Timer to Component,
- Move Component Variable/Constant/Timer to Local Scope,
- Generalize Runs On.

[2]The necessity for this refactoring became clear through a collaboration with Motorola UK.

4.2.1 Extract Altstep

Move alternative branches of an **alt** statement into an altstep when used more than once.

Motivation

Quite often, identical alternative branches in an **alt** statement are used more than once in a test suite. To avoid code duplication, reduce maintenance load and improve readability, identical alternative branches should be extracted into their own altstep. An altstep invocation inside an **alt** statement can provide code as "tail" for the altstep. Hence, the identical branches may even differ in their end-piece which would then be left in the calling **alt** statement. Note that altsteps need to be properly named to be self-explanatory. Depending on the content of the extracted alternative branches, it may be necessary to add parameters to the altstep. A refactoring that may be used after extracting altsteps is the *Replace Altstep with Default* (section 4.2.3) refactoring.

Mechanics

- Create a new target altstep and name it by what the alternative branch to be extracted does. If the alternative is so simple that it is hard to come up with a good name, usage of this refactoring should possibly be reconsidered.
- Copy the source branches to be extracted from the **alt** statement into the target altstep.
- Check the extracted alternative branch for references to any variables, constants and timers that were local to the scope of the source **alt** statement or its surrounding compound respectively, i.e. either locally defined temporary elements or elements passed as parameters to the surrounding compound. Ports cannot be locally defined, but passed as parameters; in this case, they have also local scope.
  - Locally defined temporary elements which are only used in the extracted branches are declared in the new target altstep again as temporary elements.
  - All other local-scope elements must be passed in or out of the altstep as follows:
    - Timers and ports are always passed as reference. i.e. as **inout**.
    - Elements which are only read but not modified by the extracted branches or read and modified by the extracted branches but not read afterwards at the source location are passed as **in** parameter.
    - Elements that are modified but not read before modification in the extracted branches and read afterwards at the source location must be passed as **out** parameter.
    - Elements that are both read and modified in the extracted code and also read afterwards need to be passed as **inout** parameters.

- Check if any elements declared in the component on which the **alt** statement runs on are used in the extracted branches. If so, the target altstep must have a **runs on** specifying this component as well. Afterwards, *Generalize Runs On* may be appropriate or the **runs on** specification may even be removed if *Move Component Variable/Constant/Timer to Local Scope* (section 4.2.20) is applicable.
- Compile.
- Replace the extracted code in the source **alt** statement with a call to the new target altstep and provide any required parameters.
  - If any declarations of local temporary elements have been copied to the target altstep, remove their declaration from the source compound.
  - If the extracted branches are duplicated at other locations, repeat the previous step for those locations as well. If the additional duplicated branches to be replaced are located in different modules than the target altstep, a corresponding import statement must be added or adjusted in the module of the additional duplicated branches to import the altstep.
- Compile and validate.

Example

alt statements can easily block when no alternative matches as expected. Therefore, it is a good idea to use a timer that can stop the **alt** statement when it takes unexpectedly long on the one hand and on the other hand unexpected messages should be handled as well. Such error handling distracts from the core behavior, is very generic and can therefore be reused. The concerned branches are located in lines 9–16 of listing 4.3.

```
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 4.3: Extract Altstep (unrefactored)

By applying the *Extract Altstep* refactoring, the identified branches are moved from the **alt** statement into the new altstep *alt_otherwiseFail* as shown in lines 14–24 of listing 4.4. Because the local timer *t_guard* needs to be initialized before the **alt** statement, it cannot be moved to the altstep and cannot be declared as temporary variable. Instead, it is passed as parameter. Since the altstep uses ports from the *ExampleComponent*, the **runs on** specification is required as well.

```
1  testcase tc_exampleTestCase() runs on ExampleComponent {
2    timer t_guard;
3    // [...]
4    t_guard.start ( 10.0 );
5
6    alt {
7      [] pt.receive( a_MessageOne ) {
8        pt.send( a_MessageTwo );
9      }
10     [] alt_otherwiseFail( t_guard ) { }
11   }
12 }
13
14 altstep alt_otherwiseFail( inout timer p_t )
15   runs on ExampleComponent {
16   [] any port.receive {
17     setverdict( fail );
18     stop;
19   }
20   [] p_t.timeout {
21     setverdict( fail );
22     stop;
23   }
24 }
```

Listing 4.4: Extract Altstep (refactored)

4.2.2 Split Altstep

Altsteps that contain branches which are not closely related to each other are split to maximize reuse potential.

Motivation

Altsteps sometimes contain branches that are not really related to each other, e.g. processing of expected messages paired with generic error handling. To maximize the reuse of branches which are possibly more generic, the altstep should be split into two separate altsteps. The generic parts can then be reused and are not coupled to branches that may be more specific for a certain test case. A typical smell leading to this refactoring is duplicate code, i.e. a

number of subsequent branches in various **alt** statements or altsteps that are equal. The altstep names must be adjusted to reflect the new separate meaning of both altsteps and the unrelated branches in the source altstep must not be mixed in their order. Unlike the *Extract Altstep* refactoring, the *Split Altstep* refactoring does not extract any selected branches, but is a real split between two subsequent unrelated sections. As a result, the newly created altstep is not called from a branch within the source altstep, but the behavior calling the source altstep.

Mechanics

- Identify two subsequent semantically unrelated sections of **alt** branches in the source altstep.
- Create a new target altstep and give it a temporary name. The new altstep should also have the same parameters as specified in the source altstep. If the source altstep runs on a component, the target altstep should run on this component as well.
- Copy one section (source section) to the target altstep.
- Compile.
- Remove the the source section from the source altstep.
- Find all **alt** statements with calls to the source altstep. Add a new **alt** branch before or after this call to the source altstep containing a call to the target altstep with the same actual parameters. The position of this new branch depends on the order of the original source altstep: if the branch section moved to the target altstep came first in the source altstep, then a call to the target altstep is inserted before the existing call. Otherwise, it is inserted after the existing call.
- Find all standalone calls to the source altstep. Add a new standalone call statement to the target altstep with the same actual parameters. The position of this statement depends on the order of the original altstep (see previous step).
- Compile and validate.
- Apply the *Rename* refactoring [35] on both the source and target altstep. The new names should reflect semantics of the separated branches.
- Verify if the parameters of both the source and target altstep are still referenced within the altstep. If there are unused parameters within an altstep signature, apply the *Remove Parameter* [35] refactoring.
- Compile and validate.

- Apply the *Generalize Runs On* (section 4.2.21) refactoring on the source and target altstep to exploit further reuse possibilities due to this restructuring.
- Find duplicate branches which are completely equal to either of the split altsteps and replace these branches with their according altstep.

Example

In listing 4.5, the test case *tc_exampleTestCase* (lines 1–14) contains an **alt** statement which calls the altstep *alt_handleExpectedMessages* (line 9). Within this altstep (lines 16–32), there are two different sections: in the first section (lines 18–23) expected messages are handled and in the second section, there is typical and generic TTCN-3 error handling (lines 24–31).

```
1  testcase tc_exampleTestCase() runs on ExampleComponent {
2    timer t_guard;
3
4    // [...]
5
6    t_guard.start ( 10.0 );
7
8    alt {
9      [] alt_handleExpectedMessage( t_guard ) { }
10     [] pt.receive( a_expectedMessage ) {
11       pt.send( a_answerMessage );
12     }
13   }
14 }
15
16 altstep alt_handleExpectedMessages( inout timer p_t )
17   runs on ExampleComponent {
18   [] pt.receive( a_MessageOne ) {
19     pt.send( a_MessageTwo );
20   }
21   [] pt.receive( a_MessageThree ) {
22     pt.send( a_MessageFour );
23   }
24   [] any port.receive {
25     setverdict( fail );
26     stop;
27   }
28   [] p_t.timeout {
29     setverdict( fail );
30     stop;
31   }
32 }
```

Listing 4.5: Split Altstep (unrefactored)

As these two sections are unrelated and the generic error handling can definately be reused, the *Split Altstep* refactoring is applied (listing 4.6). The error handling is now

located in the altstep *alt_otherwiseFail* (lines 15–24) while the other branches remain in *alt_handleExpectedMessages*. The altstep *alt_otherwiseFail* was called from the test case *tc_exampleTestCase* and was located in the second section of the source altstep. Hence, another alternative branch is inserted after the original call to *alt_handleExpectedMessage* in line 7 with a call to *alt_otherwiseFail*.

As no parameters are necessary anymore in *alt_handleExpectedMessages*, the parameter *p_t* was removed by the *Remove Parameter* refactoring. The *Generalize Runs On* refactoring caused the removal of the **runs on** clause in *alt_otherwiseFail* (line 15).

```
1  testcase tc_exampleTestCase() runs on ExampleComponent {
2    timer t_guard;
3    // [...]
4    t_guard.start ( 10.0 );
5
6    alt {
7      [] alt_handleExpectedMessage() { }
8      [] alt_otherwiseFail( t_guard ) { }
9      [] pt.receive( a_expectedMessage ) {
10       pt.send( a_answerMessage );
11     }
12   }
13 }
14
15 altstep alt_otherwiseFail( inout timer p_t ) {
16   [] any port.receive {
17     setverdict( fail );
18     stop;
19   }
20   [] p_t.timeout {
21     setverdict( fail );
22     stop;
23   }
24 }
25
26 altstep alt_handleExpectedMessages()
27   runs on ExampleComponent {
28   [] pt.receive( a_MessageOne ) {
29     pt.send( a_MessageTwo );
30   }
31   [] pt.receive( a_MessageThree ) {
32     pt.send( a_MessageFour );
33   }
34 }
```

Listing 4.6: Split Altstep (refactored)

4.2.3 Replace Altstep with Default

When subsequent **alt** statements have an equal branch at the end that is calling an altstep, the altstep can be activated as default and hence reduce code clutter.

Motivation

Subsequent **alt** statements often contain an duplicate branch at the end that is calling an altstep. Such a branch is typically used for error handling (e.g. handling of unexpected messages or timeouts). Such error handling is a typical cross-cutting concern in TTCN-3 and can be further simplified by using default altsteps. By using a default altstep, the equal alternative branches can be removed, but need to be activated prior to the first **alt** statement and deactivated after the last. While it may simplify the **alt** statements, it can also make code hard to understand when used too much as tracking of the activated altsteps can be hard. Activating and deactivating altsteps at different nesting levels is strongly discouraged as it makes the code very hard to understand. Also note that semantic preservation can only be guaranteed when the equal branches calling an altstep are at the end of the **alt** statement since default altsteps are automatically attached to these positions. Reordering of branches may be possible to achieve equal ends, but this depends on the semantics of the **alt** statement.

Mechanics

- Find the first and the last source **alt** statement referencing a duplicate tail **altstep**. If the **alt** statement tails have more than one altstep call in common and they all should be activated, it is easier to apply the following steps at once.
- Activate the target **altstep** before the first source **alt** statement and store the returned reference in a newly declared variable of type **default**. Typically, this variable is declared in the local scope of the test case or function.
  - If there are other default **altstep** activations before this first source **alt** statement, place the new target default **altstep** activation before the first existing default altstep activation (TTCN-3 default altsteps are executed in the reverse order of their activation, i.e. the first **altstep** activated is executed last).
- Deactivate the target altstep using the stored reference after the last source **alt** statement. If the last source **alt** statement is nested one or more levels below the activation level, deactivation should be delayed until returned to the activation level.
- Remove the last branch of each source **alt** statement.
- Compile and validate.

Example

In listing 4.7, there are two subsequent **alt** statements (lines 7–10 and 15–18) with equal branches at their end (lines 9 and 17). These branches are calling the same altstep *alt_timeGuard* (lines 22–26) which makes sure that the local test case verdict of *tc_exampleTestCase* fails when no expected message is received in time. The *Replace Altstep with Default* refactoring can be applied on these **alt** statements.

```
1  testcase tc_exampleTestCase() runs on ExampleComponent {
2    timer t_guard;
3
4    // [...]
5
6    t_guard.start ( 10.0 );
7    alt {
8      [] // [...]
9      [] alt_timeGuard( t_guard ) { }
10   }
11
12   // [...]
13
14   t_guard.start ( 10.0 );
15   alt {
16     [] // [...]
17     [] alt_timeGuard( t_guard ) { }
18   }
19
20 }
21
22 altstep alt_timeGuard( inout timer p_t ) {
23   [] p_t.timeout {
24     setverdict( fail );
25   }
26 }
```

Listing 4.7: Replace Alternatives with Default Altstep (unrefactored)

After the refactoring is applied (listing 4.8), the *alt_timeGuard* altstep is activated before the first **alt** statement (lines 5–6) and deactivated after the last (line 20). As a timer was started immediately before the first **alt** statement, the altstep activation is placed before the timer launch.

```
1  testcase tc_exampleTestCase() runs on ExampleComponent {
2    timer t_guard;
3
4    // [...]
5    var default v_defaultRef;
6    v_defaultRef := activate( alt_timeGuard( t_guard ) );
7
```

```
8     t_guard.start ( 10.0 );
9     alt {
10      [] // [...]
11    }
12
13    // [...]
14
15    t_guard.start ( 10.0 );
16    alt {
17      [] // [...]
18    }
19
20    deactivate( v_defaultRef );
21  }
22
23  altstep alt_timeGuard( inout timer p_t ) {
24    [] p_t.timeout {
25      setverdict( fail );
26    }
27  }
```

Listing 4.8: Replace Alternatives with Default Altstep (refactored)

4.2.4 Add Explaining Log

Add explanatory log statements where verdicts are changed.

Motivation

A problem often occurring is that the tester does not know what went wrong when a test case fails. This is especially the case when a test case can fail due to several reasons (e.g. an unexpected message arrived, the message content is unexpected, a timeout occurs) and the implicit logging is not sufficient. Therefore, it is recommended to add logging statements to the code that make it easier to comprehend why a test case failed. As a result, less time is spent for analyzing test case verdicts.

Mechanics

- Find code locations in the source (source locations) where the verdict is changed to *fail* or *inconc*.
- Add an explanatory log statement before the verdict change is performed which gives the tester sufficient information on why the verdict is changed. Repeat this step for all source locations.
- Compile and validate.

Example

In listing 4.9, the **alt** statement contains typical error handling where unexpected messages or timeouts lead to a failed test case verdict (lines 9–12 and 13–16). However, when this test suite with this test case is executed and the test case fails, there is no possibility to know if the reason was an expected message or a timeout. Hence, log statements are added.

```
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 4.9: Add Explaining Log Altstep (unrefactored)

After applying the *Add Explaining Log* refactoring (listing 4.10), the **setverdict** statements are explained through a log statement before them (lines 10 and 14)

```
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       log("tc_exampleTestCase: unexpected message received.");
11       setverdict( fail ); stop;
12     }
13     [] t_guard.timeout {
14       log("tc_exampleTestCase: no messages received in time.");
15       setverdict( fail ); stop;
16     }
17   }
18 }
```

Listing 4.10: Add Explaining Log (refactored)

4.2.5 Distribute Test

Transform a non-concurrent test case into a distributed concurrent test case.

Motivation

Non-concurrent TTCN-3 test cases using one main test component (MTC) are called *local tests*. However, in a distributed architecture, the test system may have to control several interfaces of the SUT and each interface has a different role towards the SUT. In a *local test*, the behavior is strictly sequential. Hence, several different roles in a *local test* result in a behavior mixture which is hard to understand and extend. In addition, the concurrent behavior is not adequately modeled. Therefore, it makes sense to distribute a test with different roles across multiple parallel test components (PTCs). Role-specific behavior can be separated through the test components, the behavior mixture of code concerning different roles is reduced.

However, distributed tests are more complicated since they are concurrent. For behavior preservation, it is necessary to synchronize the behavior between the parallel test components. Otherwise, messages may be sent or received when the other test components or the SUT do not expect them due to the missing sequential behavior. For this purpose, each parallel test component must have one or more *coordination points* where the parallel test components exchange synchronization messages. Due to these messages, there is some overhead involved in the communication though. There are arbitrary different ways to realize this synchronization and the choice highly depends on the test architecture. Therefore, the synchronization is handled universally in the mechanics. The example shows a concrete way to synchronize the PTCs.

This refactoring can be considered a ”big“ refactoring. Hence, it involves more work and time than the other refactorings presented in this chapter and involves careful considerations and decisions on behalf of the test engineer.

Mechanics

- Identify the different roles of the *local test*. Find out to which roles each port of the test system interface belongs to.
- For each role, create a component declaration with an appropriate name. Copy the ports belonging to this role into this component declaration. In a local architecture, different roles may share ports. In this case, the shared ports are duplicated in the PTCs.
- Declare synchronization ports for the MTC and the PTCs and include them in the MTC and PTC component declarations accordingly. Make sure the communication directions are supported as needed.

- Each PTC needs to have a startup function from where the behavior is controlled. This function is used later-on as parameter when the MTC starts the behavior on the concurrent PTCs.
- The MTC behavior is described by using the **testcase** construct which must now contain the test system interface in the **system** clause.
- In the MTC, add variable declarations (or a set of variables) for each PTC. Add a TTCN-3 statement calling the create operation on the corresponding test component instances and add statements to map their ports to the test system interface. Also, insert statements to connect the synchronization ports as needed.
- Add a TTCN-3 statement to start the PTCs in the MTC.
- Rewrite the test behavior and split it across the PTCs according to their roles.
  - Where necessary to maintain the correct ordering, add synchronization points to each PTCs behavior in order to preserve the message exchange order.
  - **Interleave** constructs which were used in the local architecture to simplify the notation of parallel behavior where the order of the received messages did not matter can be simplified in some cases: an **alt** statement can then be used instead of the **interleave** construct which is then distributed across the components according to their roles without synchronization.
    * If there are multiple branches in the **interleave** statement using ports in its receiving operation of the matching criteria belonging to the same role, the resulting split behavior belonging to this role must be written in an **interleave** statement and not in an **alt** statement. The remaining branches must be handled as described in the following steps.
    * If the behavior of a branch in the **interleave** statement uses ports belonging to different roles, the distribution is not mechanical and needs additional synchronization and an individual behavior distribution.
    * If the distributed **alt** statements contain only a single branch and there is no default altstep activated, the **alt** statement can be rewritten using standalone statements.

Example

Listing 4.11 shows a non-current local test. The test system interface *MyTestSystemInterface* contains two ports *p1* and *p2* (lines 1–4). It is used in the test case *tc_myTestCase* (lines 6–15). In this test case, the behavior of two roles is mixed.

```
1  type component MyTestSystemInterface {
2    port MyPort1 p1;
3    port MyPort2 p2;
4  }
5
6  testcase tc_myTestCase() runs on MyTestSystemInterface {
7    p1.send( a_m1 );
8    p1.receive( a_m2 );
9    p2.send( a_m3 );
10   p1.receive( a_m4 );
11   interleave {
12     [] p1.receive( a_m5 ) { p1.send( a_m6 ); }
13     [] p2.receive( a_m7 )  { p2.send( a_m8 ); }
14   }
15 }
```

Listing 4.11: Distribute Test (unrefactored)

After applying the *Distribute Test* refactoring, the behavior is distributed across the parallel components *FirstEntity* (lines 10–13) and *SecondEntity* (lines 15–18). Both contain the port *p* which they use for communication with the SUT. In addition, the PTCs have a synchronization port which is used for the communication with the other PTC (lines 12 and 17). The PTC ports to the SUT are mapped to the test system interface (lines 44–45) and the synchronization ports are connected to each other (line 47).

The actual behavior is distributed to the corresponding components. When the components are started (lines 49–50), they are given functions which are then executed on the respective parallel component. These functions are *f_firstRoleStartComponent* (lines 20–30) and *f_secondRoleStartComponent* (lines 32–38). The behavior is split according to the ports of the test system interface. In addition, the behavior is synchronized (e.g. lines 24 and 33). That way, messages are received and send when they are expected. The synchronization messages contain simple boolean values. In more realistic scenarios, the component synchronization would probably need to send and receive messages containing more information than a boolean value. The **interleave** construct is rewritten as an **alt** statement which is split across the PTC behavior. As only a single branch would be left and no default altsteps are activated, the **alt** statements are written standalone **receive** operations (lines 28–29 and 36–37).

```
1  type port SyncPort message {
2    inout syncmsg;
3  }
4
5  type component MyTestSystemInterface {
6    port MyPort1 p1;
7    port MyPort2 p2;
8  }
9
```

```
10 type component FirstEntity {
11   port MyPort1 p;
12   port SyncPort secondEntitySyncPort;
13 }
14
15 type component SecondEntity {
16   port MyPort2 p;
17   port SyncPort firstEntitySyncPort;
18 }
19
20 function f_firstRoleStartComponent() runs on FirstEntity {
21   p.send( a_m1 );
22   p.receive( a_m2 );
23
24   secondEntitySyncPort.send( boolean:true );
25
26   p.receive( a_m4 );
27
28   p.receive( a_m5 );
29   p.send( a_m6 );
30 }
31
32 function f_secondRoleStartComponent() runs on SecondEntity {
33   firstEntitySyncPort.receive( boolean:true );
34   p.send( a_m3 );
35
36   p.receive( a_m7 );
37   p.send( a_m8 );
38 }
39
40 testcase tc_myTestCase() runs on MasterComponent system MyTestSystemInterface {
41   var FirstEntity v_firstRole := FirstEntity.create;
42   var SecondEntity v_secondRole := SecondEntity.create;
43
44   map( v_firstRole:p, system:p1 );
45   map( v_secondRole:p, system:p2 );
46
47   connect( v_firstRole:secondEntitySyncPort, v_secondRole:firstEntitySyncPort);
48
49   v_firstRole.start( f_firstRoleStartComponent() );
50   v_secondRole.start( f_secondRoleStartComponent() );
51 }
```

Listing 4.12: Distribute Test (refactored)

4.2.6 Inline Template

A template that is used only once can be inlined for improved readability.

Motivation

In some cases, a template declaration is referenced only once throughout the whole TTCN-3 code. Using the value list notation instead of a reference for this template may improve readability as the programmer does not have to search through the code to find the corresponding declaration and it may shorten the code length. Candidates for this refactoring are only simple templates as inlined templates are typically written in a single line.

Mechanics

- Identify the source template declaration. It should be used only once throughout the TTCN-3 code and it should be simple. Otherwise, the use of this refactoring may not be appropriate.
- Find the code location where the source template declaration is referenced. This is the source reference.
- Replace the source reference with the value list notation of the source template.
  - When a normal template is inlined, its notation has the standard notation of an inlined template.
  - When a modified template is inlined, its notation must be adjusted to reflect the notation of modified inlined templates.
  - When a parameterized template is inlined, the actual parameter values of the reference must be inlined into this value list notation.
- Compile and validate.
- Remove the source template declaration. If the source template declaration is located in a different module than the source reference, the corresponding import statement in the module of the source reference should be adjusted to exclude the source template declaration.
- Compile and validate.

Example

Listing 4.13 shows the unrefactored version. The module contains the record declaration *MyMessageType* (lines 2–6) and the source template declaration *a_myMessageTemplate* (lines 8–12). In the test case (line 15), a message is sent by using a reference to the source template declaration *a_myMessageTemplate* (line 17). As this is the only reference to the template, it is the source reference.

```
1  // ...
2  type record MyMessageType {
3    integer field1 optional,
4    charstring field2,
5    boolean field3
6  }
7
8  template MyMessageType a_myMessageTemplate := {
9    field1 := omit,
10   field2 := "My string",
11   field3 := true
12 }
13
14 // ...
15 testcase tc_exampleTestCase() runs on ExampleComponent {
16   // ...
17   pt.send( a_myMessageTemplate );
18   // ...
19 }
```

Listing 4.13: Inline Template (unrefactored)

Listing 4.14 demonstrates the result of applying *Inline Template*. The source template declaration is inlined into the send statement (line 10) and replaces the source reference. No imports must be adjusted as the the source template declaration and the source reference are both in the same module.

```
1  // ...
2  type record MyMessageType {
3    integer field1 optional,
4    charstring field2,
5    boolean field3
6  }
7  // ...
8  testcase tc_exampleTestCase() runs on ExampleComponent {
9    // ...
10   pt.send( MyMessageType:{omit, "My string", true} );
11   // ...
12 }
```

Listing 4.14: Inline Template (refactored)

4.2.7 Inline Template Parameter

Inline a template parameter when all its references use a common actual parameter value.

Motivation

Templates are typically parameterized to avoid multiple template declarations that differ only in few values. However, as test suites grow and change over time, the usage of its templates may change as well. As a result, there may be situations when all references to a parameterized template have one or more actual parameters with the same values. This can also happen when the test engineer is overly eager: he parameterizes templates as he thinks it might be useful, but it turns out to be unnecessary after all. In any case, there are template references with unneeded parameters creating code clutter and more complexity than useful. Thus, the template parameter should be inlined and removed from all references.

The concept of inlining parameterized templates is used within this refactoring. Therefore, it is partially related to the *Inline Template*.

Mechanics

- Verify that all template references to the parameterized source template declaration have a common actual parameter value. The parameter with the common actual parameter values is the source parameter. Note down the common value.
  - If you have more than one common actual parameter value in all references, it is easier to inline them together. Therefore, perform each step that concerns the source parameter for each source parameter at once.
- Copy the source template declaration and give the copied declaration a temporary name. It is the target template declaration.
- In the target template declaration body, replace each reference to the source parameter with the noted value from step 1. In the target template declaration signature, remove the parameter corresponding to the source parameter.
- Compile.
- Remove the source template declaration.
- Rename the name of the target template declaration using the name of the source template declaration.
- Find all references to the target template declaration. Remove the source parameter from the actual parameter list of each reference.

- Compile and validate.
- Consider usage of the *Rename* refactoring to improve the target template declaration name.

Example

Listing 4.15, contains the parameterized template *exampleTemplate* in lines 6–9. All references to this template use the same actual parameter value (lines 12 and 13). Hence, the corresponding parameter *addressParameter* in Line 6 is inlined.

```
1  type record ExampleType {
2    boolean ipv6,
3    charstring ipAddress
4  }
5
6  template ExampleType exampleTemplate( charstring addressParameter ) := {
7    ipv6 := false,
8    ipAddress := addressParameter
9  }
10
11 testcase exampleTestCase() runs on ExampleComponent {
12   pt.send( exampleTemplate( "127.0.0.1" ) );
13   pt.receive( exampleTemplate( "127.0.0.1" ) );
14 }
```

Listing 4.15: Inline Template Parameter (Unrefactored)

After applying the *Inline Template Parameter* refactoring (Listing 4.16), the string value *"127.0.0.1"* is inlined into the template body of *exampleTemplate* (Line 8), the corresponding formal parameter of the template (Line 6) and the corresponding actual parameter of each reference to *exampleTemplate* (lines 12 and 13) are removed.

```
1  type record ExampleType {
2    boolean ipv6,
3    charstring ipAddress
4  }
5
6  template ExampleType exampleTemplate := {
7    ipv6 := false,
8    ipAddress :=  "127.0.0.1"
9  }
10
11 testcase exampleTestCase() runs on ExampleComponent {
12   pt.send( exampleTemplate );
13   pt.receive( exampleTemplate );
14 }
```

Listing 4.16: Inline Template Parameter (Refactored)

4.2.8 Extract Template

A template with the same values inlined more than once should be extracted into a template on its own to support code reuse and maintenance.

Motivation

Inlined templates of simple structured types are nice as they can improve readability. In fact, the *Inline Template* refactoring (section 4.2.6) even recommends the use of inlined templates when a simple template declaration is referenced only once. However, when a template with the same values is inlined more than once, there are reduced possibilities for code reuse due to code duplication and the maintainability is bad. Hence, the duplicate inlined templates (sources) should be extracted into a single target template declaration. The duplicate inlined source templates are replaced with a reference to the target template declaration and the code duplication is thus removed.

Mechanics

- Identify duplicate inlined source templates having the same value list notation.
- Create a new target template declaration containing the values of the duplicate inlined source templates. Give it a meaningful name describing the content of the values.
- Compile.
- Replace the duplicate inlined source templates with a reference to the new target template declaration. If the template declaration is located in a different module than the inlined source template, an import statement must be added or adjusted to include the target template declaration.
- Compile and validate.

Example

The example illustrates the *Extract Template* refactoring as a reversed *Inline Template* refactoring (section 4.2.6). In listing 4.17, two send statements (line 10 and 12) use inlined source templates with the exactly the same values.

```
1 // ...
2 type record MyMessageType {
3   integer field1 optional,
4   charstring field2,
5   boolean field3
6 }
```

```
7  // ...
8  testcase tc_exampleTestCase() runs on ExampleComponent {
9    // ...
10   pt.send( MyMessageType:{omit, "My string", true} );
11   // ...
12   pt.send( MyMessageType:{omit, "My string", true} );
13 }
```

Listing 4.17: Extract Template (unrefactored)

After applying the *Extract Template* refactoring (listing 4.18), this code duplication is removed. A new target template declaration *a_myMessageTemplate* is created (line 9). The duplicate inlined source templates are replaced with a reference to the target template declaration (lines 19 and 21). As the duplicate inlined source templates are all within the same module, the import statement is not changed.

```
1  // ...
2
3  type record MyMessageType {
4    integer field1 optional,
5    charstring field2,
6    boolean field3
7  }
8
9  template MyMessageType a_myMessageTemplate := {
10   field1 := omit,
11   field2 := "My string",
12   field3 := true
13 }
14
15 // ...
16
17 testcase tc_exampleTestCase() runs on ExampleComponent {
18   // ...
19   pt.send( a_myMessageTemplate );
20   // ...
21   pt.send( a_myMessageTemplate );
22 }
```

Listing 4.18: Extract Template (refactored)

4.2.9 Replace Template with Modified Template

Templates of structured or list types of the same type with similar content values that differ in few different fields can be simplified by using modified templates.

Motivation

When several templates of the same structured or list types have mostly the same content values and differ only in few fields, these templates (targets) can be simplified by using modified templates. A modified template declaration uses another template declaration as base (source), but redefines only the fields where it differs. All other values are inherited from the base template.

While this may sound similar to what the *Parameterize Template* refactoring does (section 4.2.10), the difference is that a parameterized template can combine only templates where one or more of the same fields differ (a template parameter is referenced in those fields) while a modified template simplifies the declaration of templates where the values are different mostly in varying fields. Replacing a target template declaration with a modified template improves maintenance and reusability as value changes in the source base template can be made at a single location (i.e. the source base template).

Mechanics

- Find template declarations of the same structured or list types that differ only in few values.
- Choose the source base template declaration from the template declarations found in the previous step. The source base template declaration is the template declaration which has the most non-differing values if compared with all other remaining templates. All other template declarations found in the previous step are the target templates.
- For each target template, repeat the following steps:
  - Modify the target template declaration to be a modified template of the source base template, i.e. remove all duplicate value entries and add a *modifies* specification.
  - Compile and validate.

Example

In the unrefactored example (listing 4.19), three templates with varying values are declared which are very similar (lines 7–12, 13–7 and 19–24). After careful inspection, the template

a_firstTemplate is chosen as the source base template as it is necessary to redefine only one single value for each target template.

```
1  type record ExampleType {
2    boolean ipv6,
3    integer examplePort,
4    charstring ipAddress,
5    charstring id
6  }
7  template ExampleType a_firstTemplate := {
8    ipv6        := false,
9    examplePort := 80,
10   ipAddress   := "127.0.0.1",
11   id          := 1
12 }
13 template ExampleType a_secondTemplate := {
14   ipv6        := false,
15   examplePort := 80,
16   ipAddress   := "134.72.13.2",
17   id          := 1
18 }
19 template ExampleType a_thirdTemplate := {
20   ipv6        := false,
21   examplePort := 80,
22   ipAddress   := "127.0.0.1",
23   id          := 2
24 }
```

Listing 4.19: Replace Template with Modified Template (unrefactored)

After applying the refactoring *Replace Template with Modified Template* (listing 4.20), the target templates *a_secondTemplate* (lines 7–9) and *a_thirdTemplate* (lines 10–12) are now declared with *a_firstTemplate* as base template (lines 1–6) using the **modifies** keyword. The common values from *a_secondTemplate* and *a_thirdTemplate* are removed.

```
1  template ExampleType a_firstTemplate := {
2    ipv6        := false,
3    examplePort := 80,
4    ipAddress   := "127.0.0.1",
5    id          := 1
6  }
7  template ExampleType a_secondTemplate modifies a_firstTemplate := {
8    ipAddress   := "134.72.13.2"
9  }
10 template ExampleType a_thirdTemplate modifies a_firstTemplate := {
11   id          := 2
12 }
```

Listing 4.20: Replace Template with Modified Template (refactored)

4.2.10 Parameterize Template

Replace several template declarations of the same type using different values for the same fields with one single parameterized template.

Motivation

Occasionally, there are several template declarations of the same type which are basically similar, but vary in values at the same fields (source template declarations). These template declarations are candidates for parametrization. Instead of keeping all of them, they are replaced with one single target template declaration where the variations are handled by template parameters. Such a change removes code duplication, improves maintainability and increases flexibility. If the template declarations are similar, but the values vary in differing fields, the *Replace Template with Modified Template* refactoring (section 4.2.9) may be a better choice.

This refactoring is similar to *Add Parameter* from [35] when reinterpreted for the use of templates. However, the mechanic of *Parameterize Template* is improved while the mechanic of *Remove Parameter* does not need such a change.

Mechanics

- Create the parameterized target template signature. It is of the same type as the source templates. Introduce a parameter for each field in which the source template values differ. The target template declaration's name should additionally reflect the meaning of the non-parameterized values.
- Copy one source template body to the parameterized target template declaration and replace the varying parts with their newly introduced template parameters.
- Compile.
- Repeat the following steps for all references to the source template declarations:
  - Replace the reference with a reference to the parameterized target template. As parameter values, use the values of the originally referenced template declaration corresponding to the parameterized values in the target template.
  - Compile and validate.
- Remove the source template declarations from the code. They should not be referenced anymore.
- Compile and validate.

Example

Listing 4.21 shows the unrefactored example. The source template declarations *a_firstTemplate* (lines 6–9) and *a_secondTemplate* (lines 11–14) differ only in the values of *ipAddress*.

```
1  type record ExampleType {
2    boolean ipv6,
3    charstring ipAddress
4  }
5
6  template ExampleType a_firstTemplate := {
7    ipv6        := false,
8    ipAddress   := "127.0.0.1"
9  }
10
11 template ExampleType a_secondTemplate := {
12   ipv6        := false,
13   ipAddress   := "134.72.13.2"
14 }
15
16 testcase tc_exampleTestCase() runs on ExampleComponent {
17   pt.send( a_firstTemplate );
18   pt.send( a_secondTemplate );
19 }
```

Listing 4.21: Parameterize Template (unrefactored)

The resulting code after applying *Parameterize Template* is shown in listing 4.22. A new target template declaration *a_parameterizedTemplate* (lines 6–9) is created which has a parameter for the varying *ipAddress* field in the source template declarations. The references to *a_firstTemplate* (line 12) and *a_secondTemplate* (line 13) are replaced with *a_parameterizedTemplate* and their corresponding IP addresses as parameters.

```
1  type record ExampleType {
2    boolean ipv6,
3    charstring ipAddress
4  }
5
6  template ExampleType a_parameterizedTemplate( charstring p_ipAddress ) := {
7    ipv6        := false,
8    ipAddress   := p_ipAddress
9  }
10
11 testcase tc_exampleTestCase() runs on ExampleComponent {
12   pt.send( a_parameterizedTemplate( "127.0.0.1" ) );
13   pt.send( a_parameterizedTemplate( "134.72.13.2" ) );
14 }
```

Listing 4.22: Parameterize Template (refactored)

4.2.11 Decompose Template

Decompose complex template declarations into smaller templates using references.

Motivation

Template declarations for complex types can easily be very long. They become hard to read and maintain. In addition to this, the longer a single template declaration is, the more likely that it is highly specialized and cannot be reused in other places. It may be possible though to reuse single parts of the complex template. A template may be decomposed into smaller templates in TTCN-3. The originally complex template then references the small subtemplates instead of defining every value within itself. Application of this refactoring should not be exaggerated. Too many subtemplates or too many small templates are confusing as well. Refactorings that may be used after decomposing a template are *Parameterize Template* or *Replace Template with Modified Template*.

Mechanics

- Create a new target template for each field of the source template which contains a record or list type. The type of the target template is determined by the type of the corresponding field of the source template. The name usually relates to the name of the corresponding field of the source template.
- Copy each value specification from the source template into its corresponding smaller target template.
- Compile.
- Repeat the following steps until all record or list type fields of the source template are replaced with references to the corresponding target template:
  - Replace one value specification from the source template with a reference to its corresponding target template.
  - Compile and validate.

Example

The example is derived from an ETSI SIP test suite [32]. Listing 4.23 shows the rather complex template *a_contactAddress* which specifies all fields right in the template declaration (lines 7–18).

```
1  type record ContactAddress
2  {
3    Addr_Union             addressField,
4    SemicolonParam_List    contactParams optional
5  }
6
7  template ContactAddress a_contactAddress := {
8    addressField := {
9      nameAddr := {
10       displayName := "ETSI Tester",
11       addrSpec := omit
12     }
13   },
14   contactParams := {{
15     id := "transport",
16     paramValue := PX_TRANSPORT
17   }}
18 }
```

Listing 4.23: Decompose Template (unrefactored)

Listing 4.24 shows the refactored version. The values of *addressField* and *contactParams* have been moved to their own templates (lines 7–12 and 14–19) and are then referenced within the originally complex template (lines 21–24).

```
1  type record ContactAddress
2  {
3    Addr_Union             addressField,
4    SemicolonParam_List    contactParams optional
5  }
6
7  template Addr_Union a_specificAddressField := {
8    nameAddr := {
9      displayName := "ETSI Tester",
10     addrSpec := omit
11   }
12 }
13
14 template SemicolonParam_List a_specificContactParams := {
15   {
16     id := "transport",
17     paramValue := PX_TRANSPORT
18   }
19 }
20
21 template ContactAddress a_contactAddress := {
22   addressField := a_specificAddressField,
23   contactParams := a_specificContactParams
24 }
```

Listing 4.24: Decompose Template (refactored)

4.2.12 Subtype Basic Types

Use subtypes on basic type declarations to improve code flaw detection.

Motivation

It is possible to restrict types in TTCN-3 by using subtypes (such as lists, ranges and length restrictions) on basic types. This is good for two reasons. Firstly, it forces the test engineer to think about what he wants to achieve and what values he expects the type instantiations to have. Secondly, it allows early detection of code flaws either in the test suite or the SUT. A tool making a thorough semantic analysis of the code can partially detect when an assignment is outside its declared range and report this error for example. Other violations (e.g. range violations) can only be detected at test execution, but entirely undetected range violations can possibly lead to unexpected behavior that may lead to false test verdicts which is, needless to say, a bad thing for a test.

Mechanics

- Copy the source type declaration and give it a carefully decided subtype. Give it a temporary name. This is the target type declaration.
- Compile.
- Remove the source type declaration and rename the name of the subtyped target type declaration to the name of the source type declaration.
- Compile and validate.
- Consider if applying the *Rename* refactoring to the target type declaration may improve the meaningfulness of its name now that it is subtyped.

Example

The example uses subtyping for early detection when values are out of range. In listing 4.25, a variable *v_byte* of type integer is declared (line 4) and then assigned 1024 (line 5). A byte is of course only 8 bits long and its range is between 0 and 255. Hence, assigning 1024 to a variable that is supposed to be within this range is illegal and will probably result in unexpected behavior when not detected early by the compiler.

```
1 // ...
2 control {
3   // ...
4   var integer v_byte;
5   v_byte := 1024;
6 }
```

Listing 4.25: Subtype Basic Types (unrefactored)

Listing 4.26 improves this weak point by introducing a type *byte* (line 2) with a range restriction which is then used instead of type **integer** (line 6). The compiler is now able to detect this flaw at compilation.

```
1 // ...
2 type integer byte ( 0 .. 255 );
3
4 control {
5   // ...
6   var byte v_byte;
7   v_byte := 1024;
8 }
```

Listing 4.26: Subtype Basic Types (refactored)

4.2.13 Extract Module / Move Declarations to Another Module

Move parts of a module into a newly created module or into another existing module to improve structure and reusability.

Motivation

Modules are essential and necessary for flexibility and structure of TTCN-3 test suites. Structuring code is essential for readability on the one hand and on the other hand allows to apply other refactorings that can improve the reuse of certain parts of code (e.g. using *Parameterize Module* (section 4.2.17)). Therefore, it should be considered to move parts that share a meaningful relationship of a long test suite into its own module (i.e. *Extract Module*) or into another more suitable module that already exists (i.e. *Move Declarations to Module*). Except for the different target modules, both refactorings are the same and thus described together in this section. Good candidates for extracting/moving may already be collected in a **group**. If not, a possible preceding refactoring is *Group Fragments* (section 4.2.14).

Mechanics

- If the target module does not exist already, create a new module as target using a name describing the identified code part in a good way.
- Copy the identified code into the target module. Copy also the complete **import** declarations from the source module to the target module. Dependencies between modules should generally be kept to a minimum. This should be kept in mind when extracting a module or moving declarations. Furthermore, make sure that the result of moving the chosen declarations does not create a cyclic dependency between the source module and the target module. If such a dependency occurs, this refactoring is not applicable.
- Compile.
- Remove the identified code from the source module and put an **import from ... all** statement at the top of the source module to import all declarations from the new module.
- Compile and validate.
- Use the *Restrict Imports* refactoring on the **import** statement in both, source and target, modules.

Example

The Example module from listing 4.27 contains type (lines 2–17) and template declarations (lines 18–21). It is desirable to separate type and templates declarations for a better structure.

```
1  module Example {
2    group types {
3      type charstring Answer;
4      type integer IdentificationCode;
5    }
6
7    group structured {
8      type record PersonType {
9        charstring firstName,
10       charstring middleName,
11       charstring lastName
12     }
13     type record ExampleMessageType {
14       PersonType  Person,
15       IdentificationCode idCode
16     }
17   }
```

```
18    template ExampleMessageType a_testMessage := {
19      Person := { "Arthur", "Dent" },
20      idCode := 42
21    }
22  }
```

Listing 4.27: Extract Module (unrefactored)

A new module with the name *ExampleTypes* is created (lisiting 4.28) which is the target module. The type declarations from module *Example* are copied to the target module *ExampleTypes* (lines 1–19) and the corresponding declarations from the source module are removed while the template declarations reside in module *Example*. No **import** is necessary as the extracted code in module *ExampleTypes* does not refer to any external declarations. The source module, however, needs needs to import from *ExampleTypes* to have access to the required type declarations (lines 22–24). The imports have been minimized by applying the *Restrict Imports* (section 4.2.15) refactoring afterwards.

```
1   module ExampleTypes {
2     group types {
3       type charstring Answer;
4       type integer IdentificationCode;
5     }
6
7     group structured {
8       type record PersonType {
9         charstring firstName,
10        charstring middleName,
11        charstring lastName
12      }
13
14      type record ExampleMessageType {
15        PersonType Person,
16        IdentificationCode idCode
17      }
18    }
19  }
20
21  module Example {
22    import from ExampleTypes {
23      type PersonType, IdentificationCode, ExampleMessageType;
24    }
25
26    template ExampleMessageType a_testMessage := {
27      Person := { "Arthur", "Dent" },
28      idCode := 42
29    }
30  }
```

Listing 4.28: Extract Module (refactored)

4.2.14 Group Fragments

Add additional structure to a module by putting logically related declarations into groups.

Motivation

Although TTCN-3 groups have only little semantical meaning, they can help to improve the code in two ways. Firstly, the general structure of a module may be improved as the grouping collects elements that logically belong together. The benefit is good readability and easy navigation if the used TTCN-3 development environment supports navigation using groups. Secondly, proper grouping can be helpful for fine grained **import** statements since groups can be used in the import restrictions. Therefore, *Group Fragments* is related to the *Restrict Imports* (section 4.2.15) refactoring which can make use of groups. Groups are also often candidates to move into another module using the *Extract Module / Move Declarations to Another Module* refactoring (section 4.2.13).

Mechanics

- Find declarations which can be logically grouped and find a name for this group reflecting its contents. These are the source declarations. All declarations must be located in the same module. Otherwise, *Extract Module / Move Declarations to Another Module* should be used beforehand (section 4.2.13).
- Check whether a target group with the same name or with a corresponding meaning already exists. If not, create the target group with the chosen name.
- Move the source declarations to the target group.
- Compile and validate.
- Consider using the *Restrict Imports* (section 4.2.15) refactoring afterwards.

Example

Listing 4.29 shows a module containing declarations which can be logically grouped by constants (lines 2–3), types (lines 5–6) and structured types (lines 8–10).

```
1 module ExampleModule {
2   const integer c_defaultSmtpPort := 25;
3   const integer c_defaultSmtpSSLPort := 465;
4
5   type charstring Answer;
6   type charstring Question;
7
```

```
8     type record SmtpMessage {
9       // ...
10    }
11
12  }
```

Listing 4.29: Group Fragments (unrefactored)

In the refactored version (Listing 4.30), these declarations are surrounded by groups with names corresponding to the declaration types (lines 2–5, 7–10 and 12–16).

```
1   module ExampleModule {
2     group consts {
3       const integer c_defaultSmtpPort := 25;
4       const integer c_defaultSmtpSSLPort := 465;
5     }
6
7     group types {
8       type charstring Answer;
9       type charstring Question;
10    }
11
12    group structured {
13      type record SmtpMessage {
14        // ...
15      }
16    }
17
18  }
```

Listing 4.30: Group Fragments (refactored)

4.2.15 Restrict Imports

Restrict **import** statements for smaller inter-module interfaces and less processing load for TTCN-3 tools.

Motivation

TTCN-3 offers the possibility to restrict **import** statements. The inter-module interfaces are minimized when only declarations are imported that are actually used. As a result, modularity and reuse possibilities are improved and the possibility for name clashes is reduced. Furthermore, the module dependencies and relationships become more clear. Depending on the implementation, the processing load of TTCN-3 tools may be reduced when the inter-module dependencies are minimized. *Restrict Import* can be applied whenever module

dependencies or the internal structure of type declarations change (e.g. through application of *Group Fragments* (section 4.2.14). Note that the provided mechanics do not exploit all restriction possibilities of TTCN-3. Specifically group exceptions can make an import statement hard to understand and should be used only when absolutely necessary.

Mechanics

- Find the declarations for all references which are imported from other modules and used in the source module.
- Replace the existing import statements in the source module with new import statements (one import statement for each module on which the source module depends on) which are restricted by listing the concrete declarations names found in the previous step.
- Compile.
- If all declarations of a kind are listed within a restriction, remove all concretely listed declaration names of this kind and add a kind import for all elements of this kind (e.g. **type all**, **altstep all**, etc.).
- Compile.
- If all declarations of a group are listed within a restriction, remove all concretely listed declaration names of this group and add a group import.
- Compile.
- Compare the number of all concretely listed elements of a kind or a group to the total number of elements of their corresponding kind or group within their module. As a loose rule, if the difference is equal or less than three and the number of concretely listed elements is greater than three, this concrete kind or group import list should be replaced by a statement importing all elements of a kind or a group with these few elements as exceptions (e.g. **type all except concreteDeclaration**)
- Compile and validate.

Example

Listing 4.31 contains two modules. *ExampleModule* (lines 1–17) contains declarations which are grouped according to their types (lines 2–5, 7–10 and 12–16). *DependantModule* (lines 19–27) uses only the constants from *ExampleModule*, but imports all declarations (line 20). Hence, the inter-module interface is bigger than it needs to be.

```
1  module ExampleModule {
2    group consts {
3      const integer c_defaultSmtpPort := 25;
4      const integer c_defaultSmtpSSLPort := 465;
5    }
6
7    group types {
8      type charstring Answer;
9      type charstring Question;
10   }
11
12   group structured {
13     type record SmtpMessage {
14       // ...
15     }
16   }
17 }
18
19 module DependantModule {
20   import from ExampleModule all;
21
22   testcase tc_exampleTestCase() runs on ExampleComponent {
23     // ...
24     theAddress.portField := c_defaultSmtpPort;
25     theAddress.portFieldSSL := c_defaultSmtpSSLPort;
26   }
27 }
```

Listing 4.31: Restrict Imports (unrefactored)

After applying the *Restrict Imports* refactoring, the situation is improved by adding a restriction to the import statement. In the first step, the restriction contains the concrete variable names *c_defaultSmtpPort* and *c_defaultSmtpSSLPort*. In the second step (listing 4.32), this is optimized by replacing the two concrete imports with a group import (line 19).

```
1  module ExampleModule {
2    group consts {
3      const integer c_defaultSmtpPort := 25;
4      const integer c_defaultSmtpSSLPort := 465;
5    }
6
7    group types {
8      type charstring Answer;
9      type charstring Question;
10   }
11
12   group structured {
13     type record SmtpMessage {
14       // ...
15     }
16   }
17 }
```

```
18 module DependantModule {
19   import from ExampleModule { group consts };
20
21   // ...
22
23   testcase tc_exampleTestCase() runs on ExampleComponent {
24     // ...
25     theAddress.portField := c_defaultSmtpPort;
26     theAddress.portFieldSSL := c_defaultSmtpSSLPort;
27   }
28 }
```

Listing 4.32: Restrict Imports (refactored)

4.2.16 Prefix Imported Declarations

References to imported declarations are prefixed to avoid possible name clashes.

Motivation

References to imported declarations can be prefixed with the module name where they originate from. Usage of this refactoring should only be considered when name clashes are inevitable (e.g. when an imported module cannot be changed) and should not be applied generally as prefixed references are typically harder to read. Also, in the case of name clashes, it should be verified if the *Rename* refactoring applied to the local clashing declaration would not yield a better result.

Mechanics

- In the source module, find the module from which the declaration to be prefixed is imported from and find the module's name.
- For each reference to this imported declaration, add a prefix containing the module name.
- Compile and validate.

Example

Listing 4.33 shows the unrefactored version. In the *ExampleModule* (line 1–6), a type *SipMethod* (lines 3–4) is declared which is then imported by the *DependantModule* (lines 8–18). In line 11, there is a second type declaration with the name *SipMethod* which causes a name clash. According to the scoping rules, *v_method* (line 15) uses *SipMethod* as a **charstring**, but the actual value hints that the imported *SipMethod* is meant.

```
1  module ExampleModule {
2    group types {
3      type charstring SipMethod ("REGISTER", "INVITE", "ACK", "BYE",
4        "CANCEL", "OPTIONS");
5    }
6  }
7
8  module DependantModule {
9    import from ExampleModule all;
10
11   type charstring SipMethod;
12
13   testcase tc_exampleTestCase() runs on ExampleComponent {
14     // ...
15     var SipMethod v_method := "QUIT";
16   }
17
18 }
```

Listing 4.33: Prefix Imported Declarations (unrefactored)

Listing 4.34 solves this problem by prefixing *SipMethod* with *ExampleModule* on declaration of *v_method* (line 15). It is clear now that *v_method* should use the imported type declaration.

```
1  module ExampleModule {
2    group types {
3      type charstring SipMethod ("REGISTER", "INVITE", "ACK", "BYE",
4        "CANCEL", "OPTIONS");
5    }
6  }
7
8  module DependantModule {
9    import from ExampleModule all;
10
11   type charstring SipMethod;
12
13   testcase tc_exampleTestCase() runs on ExampleComponent {
14     // ...
15     var ExampleModule.SipMethod v_method := "QUIT";
16   }
17
18 }
```

Listing 4.34: Prefix Imported Declarations (refactored)

4.2.17 Parameterize Module

Parameterize modules to specify environment specific parameters at tool level.

Motivation

TTCN-3 test cases are executed in an automated way and there is no possibility to interact with the user through a GUI at TTCN-3 level for example. Therefore, information that is specific to a local test environment (such as IP addresses of SUTs) may sometimes be found in constants. However, through module parameters, this task can be deferred to the TTCN-3 tooling. As a result, TTCN-3 code does not have to be changed when used in different test environments in the ideal case. Hence, reusability is improved and maintenance work reduced.

Mechanics

- Find constants and magic numbers that are specific to the test environment in the source module. For magic numbers, apply the *Replace Magic Number with Symbolic Constant* refactoring [35]. The resulting constant declarations are the source declarations.
- Move all source constant declarations into the **modulepar** section. If there is no **modulepar** section yet, it must be created. Remove the **const** keyword from each declaration.
- Specify custom environment-specific actual parameters for the module parameters at tool level.
- Compile and validate using the original magic numbers from the source module as module parameters at tool level.

Example

Listing 4.35 shows the unrefactored version. There are two templates where custom IP addresses are specified (lines 6–9 and 11–14). Apparently, this is not optimal as the specified IP addresses (lines 8 and 13) would have to be modified for each test environment.

```
1  type record ExampleType {
2    boolean ipv6,
3    charstring ipAddress
4  }
5
6  template ExampleType a_firstRemoteAddressTemplate := {
7    ipv6        := false,
8    ipAddress   := "64.233.187.99"
9  }
10
```

```
11 template ExampleType a_secondRemoteAddressTemplate := {
12   ipv6        := false,
13   ipAddress   := "134.72.13.2"
14 }
15 testcase tc_exampleTestCase() runs on ExampleComponent {
16   pt.send( a_firstRemoteAddressTemplate );
17   pt.send( a_secondRemoteAddressTemplate );
18 }
```

Listing 4.35: Parameterize Module (unrefactored)

After application of the *Replace Magic Number with Symbolic Constant* refactoring, two constants are introduced for the local (line 1) and remote IP address (line 2). The templates *a_localAddressTemplate* and *a_remoteAddressTemplate* now reference these constants (lines 6 and 10).

```
1  const charstring c_firstRemoteIPAddress := "64.233.187.99";
2  const charstring c_secondRemoteIPAddress := "134.72.13.2";
3
4  template ExampleType a_firstRemoteAddressTemplate := {
5    ipv6        := false,
6    ipAddress   := c_firstRemoteIPAddress
7  }
8  template ExampleType a_secondRemoteAddressTemplate := {
9    ipv6        := false,
10   ipAddress   := c_secondRemoteIPAddress
11 }
```

Listing 4.36: Parameterize Module (intermediate step)

In the next and final step, the constants are converted to module parameters by putting them into the **modulepar** section (lines 1–4) and removing the **const** keyword from the declaration. Now, they can be overwritten externally by TTCN-3 tools.

```
1  modulepar {
2    charstring mp_firstRemoteIPAddress := "64.233.187.99";
3    charstring mp_secondRemoteIPAddress := "134.72.13.2";
4  }
5
6  template ExampleType a_firstRemoteAddressTemplate := {
7    ipv6        := false,
8    ipAddress   := mp_firstRemoteIPAddress
9  }
10 template ExampleType a_secondRemoteAddressTemplate := {
11   ipv6        := false,
12   ipAddress   := mp_secondRemoteIPAddress
13 }
```

Listing 4.37: Parameterize Module (refactored)

4.2.18 Move Module Constants to Component

When the declaration of a constant at module level is used exclusively by a module component or functions/altsteps/test cases running on this component, it should be moved into the component declaration.

Motivation

Constant declarations at module level can introduce problems when constants with the same name from other modules are imported. This is especially the case when unspecifically all declarations from a module are imported. The programmer may not be aware of the name clash and in the worst case, his IDE or compiler does not provide useful error messages which makes this implicit mistake hard to find. So the constant declarations that are used only within a single component anyway should always be moved into the component declaration to limit its scope. Another method to avoid these kinds of problems is the *Prefix Imported Declarations* refactoring (section 4.2.16). When constants are moved to the component, their tie to the component is fortified. Therefore, the decision whether to use *Prefix Imported Declarations* or *Move Module Constants to Component* when name clashes happen, depends on the overall context.

Mechanics

- Copy the source constant declaration to the target component.
- Compile.
- Remove the source constant declaration.
- Compile and validate.

Example

Listing 4.38 shows the module *ExampleModule* which has a constant declaration *v_exampleVariable* (line 2) at module scope. This variable is only used by the test case *tc_exampleTestCase* (lines 8–10). Hence, it can be moved from the module scope into the component.

```
1 module ExampleModule {
2   const charstring v_exampleConstant := "teststring";
3
4   type component TestComponent {
5     // ...
6   }
7   // ...
```

```
8     testcase tc_exampleTestCase() runs on TestComponent {
9       // ...
10      f_doSomething(v_exampleConstant);
11    }
12  }
```

Listing 4.38: Move Module Constants to Component (unrefactored)

Listing 4.39 shows the refactored version. Instead of a declaration at module level, the constant declaration is now part of the component (line 4).

```
1   module ExampleModule {
2     type component TestComponent {
3       // ...
4       const charstring v_exampleConstant := "teststring";
5     }
6     // ...
7     testcase tc_exampleTestCase() runs on TestComponent {
8       // ...
9       f_doSomething(v_exampleConstant);
10    }
11  }
```

Listing 4.39: Move Module Constants to Component (refactored)

4.2.19 Move Local Variables/Constants/Timer to Component

Local declarations of variables, constants and timers can be moved to a component when used in different functions, test cases or altsteps running on the same component to reduce code clutter.

Motivation

When a local variable, constant or timer in a function, test case or altstep that runs on a component is used in more than one function, test case or altstep running on the same component by parameter passing, it might be a good idea to move it to the component. This way, accessibility of this variable and its value is improved within the component scope. The test engineer avoids unnecessary passing of parameters and therefore avoids code clutter. Note that usage of this refactoring should always be used with care as moving variables to the component level can introduce problems similar to global variables depending on the complexity of the component and the number of component variables. Using component variables within functions and test cases fortifies its tie to the component. Hence, detaching such a function, test case or altstep from a component using component variables can be hard.

Mechanics

- Find the declaration of the local source variable/constant/timer in its function, altstep or test case.
- Copy this local source variable/constant/timer declaration to the target component as component declaration.
- If a source variable is initialized with a value at its local declaration, choose whether the initialization must be moved to the target component declaration as well or whether it should be converted into an assignment at the location of the source declaration. This depends on the semantics of the component behavior.
- Compile.
- Remove the local source declaration from the function, altstep or test case.
- Compile and validate.
- Find functions, altsteps and test cases running on the same target component and called from the scope or subscope of the source declaration where a reference to the source declaration was passed as parameter. For each of these functions, altstep or test cases, remove the corresponding parameter and adjust its body to use references to the target component variable, constant or timer instead.
- Compile and validate.

Example

In this example (Listing 4.40), there are two functions *f_exampleFunction* (line 6) and *f_anotherExampleFunction* (line 12). A local variable *v_exampleVariable* is declared in *f_exampleFunction* and then assigned the value *16* (line 8). This variable passed as an **in** parameter to the function *f_anotherExampleFunction* (line 10) where it is used in a conditional statement (line 15).

```
1  // ...
2  type component ExampleComponent {
3    // ...
4  }
5
6  function f_exampleFunction() runs on ExampleComponent {
7    // ...
8    var integer v_exampleVariable := 16;
9    // ...
10   f_anotherExampleFunction(v_exampleVariable);
11 }
```

```
12 function f_anotherExampleFunction(in integer p_exampleParam)
13   runs on ExampleComponent {
14   // ...
15   if (p_exampleParam > 0) {
16     // ...
17   }
18 }
19
20 // ...
```

Listing 4.40: Move Local Variables/Constants/Timer to Component (unrefactored)

The refactored version (listing 4.41) moved the local variable declaration including its initialization to the component (line 4) and adjusts the signature and body of the function *f_anotherExampleFunction* (line 12) to use the component variable instead of a parameter.

```
1  // ...
2  type component ExampleComponent {
3    // ...
4    var integer v_exampleVariable := 16;
5  }
6
7  function f_exampleFunction() runs on ExampleComponent {
8    // ...
9    f_anotherExampleFunction();
10 }
11
12 function f_anotherExampleFunction() runs on ExampleComponent {
13   // ...
14   if (v_exampleVariable > 0) {
15     // ...
16   }
17 }
18 // ...
```

Listing 4.41: Move Local Variables/Constants/Timer to Component (refactored)

4.2.20 Move Component Variable/Constant/Timer to Local Scope

A component variable, constant or timer is moved to a local scope when only used in a single function, altstep or test case.

Motivation

When a local variable, constant or timer declaration of a component is referenced only by a single function, altstep or test case, the reusability of this function, altstep or test case can be improved when the component declaration is moved to the local scope of the function, altstep or test case. This way, its coupling to the component is reduced or removed. As a result, the **runs on** clause can possibly be generalized with the *Generalize Runs On* refactoring (section 4.2.21) to allow its use on other components or it may even become superfluous when no other component declaration is referenced. Hence, component declarations should always be used by multiple functions, altsteps or test cases.

Mechanics

- Inspect all functions, altsteps and test cases running on the source component to ensure the target function, altstep or test case is indeed the only one referencing the concerned declaration in the source component.
- Copy the declaration (including a possible initialization value) from the source component to the target function, altstep or test case. The copied declaration should be the first statement within the target.
- Compile
- Remove the declaration from the source component.
- Compile and validate.
- Consider the application on *Generalize Runs On* (section 4.2.21) or remove the **runs on** clause when no more declarations from the source component are referenced within the target.

Example

The component *MyComponent* (lines 1–4) in listing 4.42 contains a local declaration *myInt* (line 3) which is used in the function *f_myFunction* (lines 6–8).

```
1 type component MyComponent {
2   // ...
3   var integer myInt;
4 }
5
6 function f_myFunction() runs on MyComponent {
7   myInt := 255;
8 }
```

Listing 4.42: Move Component Variable/Constant/Timer to Local Scope (unrefactored)

As *f_myFunction* is the only function, altstep or test case referencing this component declaration, it is moved as local declaration to the function itself (listing 4.43). As a result, *f_myFunction* no longer depends on *MyComponent* as no ports or other declaration local to the component are used. Therefore, the **runs on** clause is removed from line 5.

```
1 type component MyComponent {
2   // ...
3 }
4
5 function f_myFunction() {
6   var integer myInt;
7   myInt := 255;
8 }
```

Listing 4.43: Move Component Variable/Constant/Timer to Local Scope (refactored)

4.2.21 Generalize Runs On

Change the component in the **runs on** clause of a function, test case or altstep to a more general component to improve reusability.

Motivation

Functions, test cases and altsteps running on a component typically use ports, constants, variables or timers which are part of this component (source component). When another component (target component) is available that contains exactly the ports, constants, variables and timers that are used in this function, test case or altstep and the target component is a subset of the source component (in terms of the declarations local to the component), the **runs on** clause of the function, test case or altstep may be generalized and hence changed to the target component. As a result, the reuse is improved since it may be used with any component which is an extension or superset of the target component. In the extreme case, the **runs on** clause can be removed entirely when no component element is referenced at all.

Note that the source component is not necessarily an extension of the target component. Type compatibility rules in TTCN-3 allow the use of a function, test case or altstep running on component *a* to be used with any component *b* that contains all declarations of *a* with the same identifier names, types and initialization values. In addition, note that the **runs on** clause in a local test architecture implicitly defines the interface towards the SUT and should not simply be changed.

Mechanics

- If no element of the source component is referenced in a function or altstep, the **runs on** clause can be removed entirely. Otherwise, follow the subsequent steps. Test cases must always have a **runs on** clause.
- Find all components containing a subset of the declarations in the source component. These components must have the same identifier names, types and initializations values for its elements.
- From these found components, find the most minimal component which is still compatible with the concerned function, test case or altstep, i.e. it must have all declarations which are referenced in the function, test case or altstep and it must have the fewest declarations. This is the target component.
- Change the **runs on** specification in concerned function, test case or altstep to use the target component instead of the source component.
- Compile and validate.

Example

In listing 4.44, there are three components *MySuperComponent* (lines 1–3), *mySecondSuperComponent* (lines 5–7) and *MySubComponent* (lines 8–10). *MySuperComponent* and *mySecondSuperComponent* are extensions of *MySubComponent* and hence the declaration *myIntegerVar* (line 9) is also part of *MySuperComponent* and *MySecondSuperComponent*. The function *f_myFunction* runs on *MySuperComponent*, but actually uses only *myIntegerVar* (line 13).

```
1 type component MySuperComponent extends MySubComponent {
2   var charstring myCharstring;
3 }
4
5 type component MySecondSuperComponent extends MySubComponent {
6   var boolean myBoolean;
7 }
```

```
8  type component MySubComponent {
9    var integer myIntegerVar;
10 }
11
12 function f_myFunction() runs on MySuperComponent {
13   myIntegerVar := 255;
14 }
```

Listing 4.44: Generalize Runs On (unrefactored)

In order to use *f_myFunction* with *MySecondSuperComponent* as well, the **runs on** clause must be generalized. The only component which is a subset of *MySuperComponent* and is compatible with *f_myFunction* is *MySubComponent*. Hence, after applying the *Generalize Runs On* refactoring (listing 4.45), the **runs on** clause of *f_myFunction* is changed to *MySubComponent* (line 13) and *f_myFunction* can be used with the components *MySuperComponent*, *MySecondSuperComponent* and *MySubComponent* now.

```
1  type component MySuperComponent extends MySubComponent {
2    var charstring myCharstring;
3  }
4
5  type component MySecondSuperComponent extends MySubComponent {
6    var boolean myBoolean;
7  }
8
9  type component MySubComponent {
10   var integer myIntegerVar;
11 }
12
13 function f_myFunction() runs on MySubComponent {
14   myIntegerVar := 255;
15 }
```

Listing 4.45: Generalize Runs On (refactored)

5 The TRex Refactoring Tool

The refactoring mechanics presented in chapter 4 can be applied manually. However, automated refactorings are significantly less error-prone and can be time saving. For example, many refactorings involve source changes which are spread across multiple files and many different positions within the source code making them error-prone and time consuming. Therefore, the idea to automate or partially automate this process is obvious. To prove that refactoring automation is possible with TTCN-3, such a tool has been developed within the context of this thesis. The result is TRex, a refactoring tool based on the Eclipse Platform [29].

Using the Eclipse Platform is advantageous as it provides many ready to use components necessary when writing an IDE and related functionalities. Such components are a user interface specifically designed for software development (Workbench) or project and file management (workspace) for example. Other integrated development environments like IDEA or Netbeans [9] are similarly useful. Eclipse however, is well designed, open source and above all very well documented. Books like [25], [38], [18] or [50] are very useful when implementing support for new languages in Eclipse. The Eclipse Platform and TRex are both written in Java (TRex requires a Java 5.0 runtime). It is not only a refactoring tool, but also the foundation for an Eclipse based integrated development environment (IDE) for TTCN-3. It already provides several functionalities of advanced development tools like the Eclipse Java Development Tools (JDT) [29] or IntelliJ IDEA [45]. The concrete implementation provided valuable insights into the practical realization of semantic preserving program transformations of the TTCN-3 core notation and will be the basis for further research.

This chapter describes the key components of the refactoring tool and the implemented refactorings in detail. The key components for the infrastructure are the pretty printer (section 5.2) and the symbol table (section 5.3). Section 5.4 describes the implemented refactorings. Other TRex features mostly unrelated to the refactorings can be found in section 5.5. Finally, information on the unit test infrastructure and the build system is provided in section 5.6

5.1 The TRex Architecture

Figure 5.1 shows the architecture of the tool. The TRex feature bundles two plug-ins: The TRex ANTLR plug-in and the TRex plug-in. The TRex ANTLR plug-in merely provides the ANTLR runtime bundled within an Eclipse plug-in. This runtime provides classes that

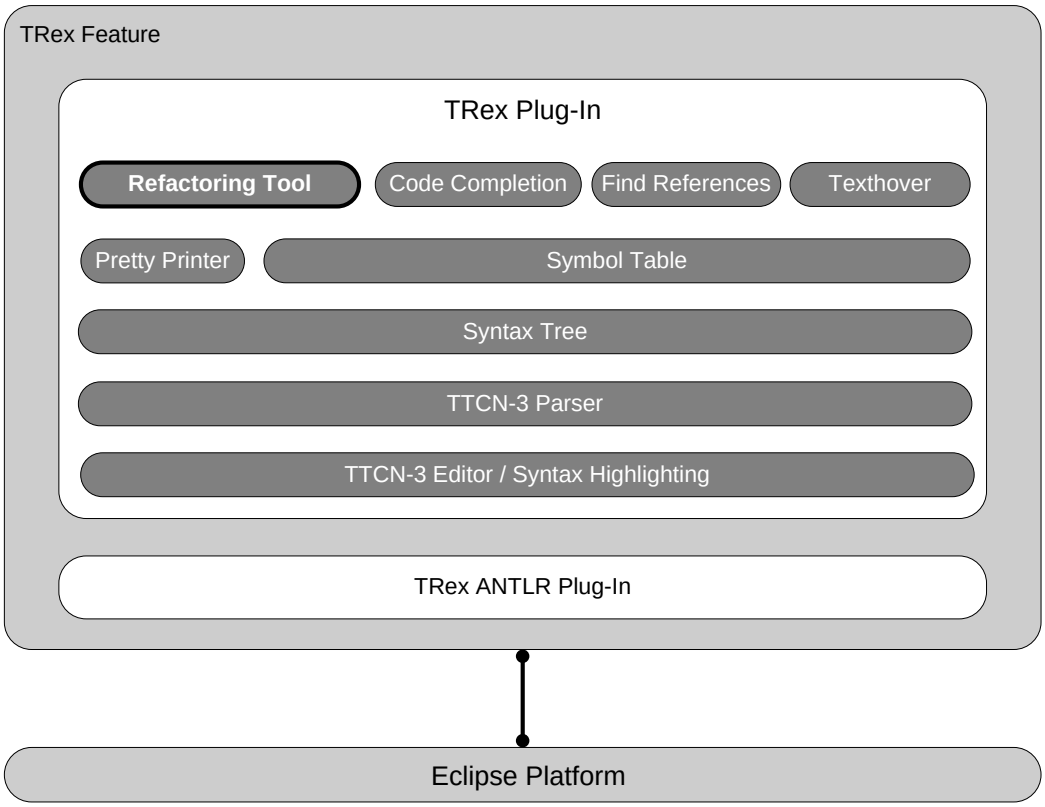


Figure 5.1: TRex Extension Diagram

are necessary for the usage of the syntax tree. The TRex core plug-in provides the key functionality of TRex. The very essential functionality for any IDE is the editor. The TTCN-3 editor is based on an abstract editor of the Eclipse Platform and therefore has the look and feel known from JDT for example. Naturally, like any other proper editor designed to edit source code, the editor provides syntax highlighting for TTCN-3.

The TRex project consists of 6 subprojects:

- The *de.ugoe.cs.swe.trex* project contains an Eclipse feature. Eclipse features primarily collect sets of Eclipse plug-ins that form a unit. They are essential for update sites and product branding.
- The *de.ugoe.cs.swe.trex.antlr* project contains the ANTLR runtime as an Eclipse plug-in. It is necessary for the usage of the syntax tree.
- The *de.ugoe.cs.swe.trex.build* project contains the build system.

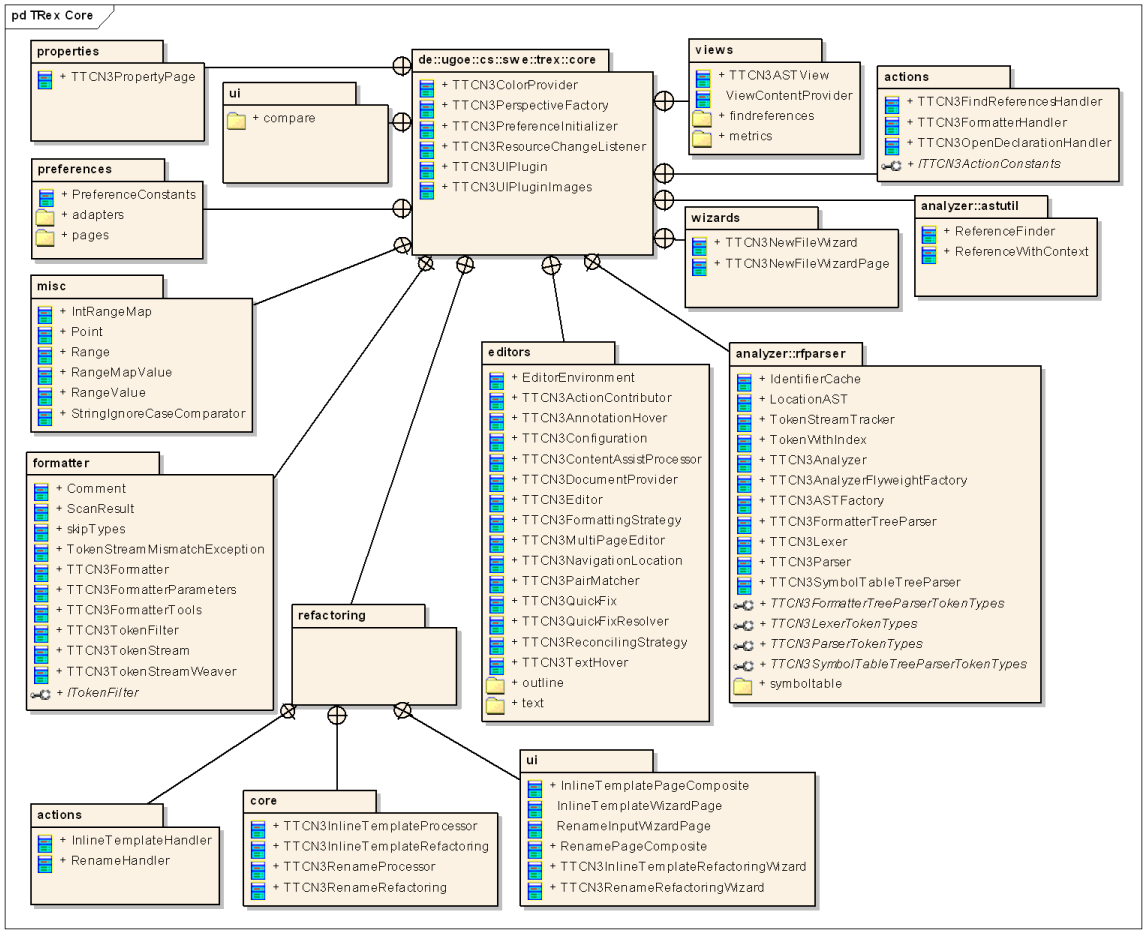


Figure 5.2: TRex Core Package Diagram

- The *de.ugoe.cs.swe.trex.core* project contains the actual TRex extension. It includes UI code, data structures and the algorithms for TRex.
- The *de.ugoe.cs.swe.trex.tests* project contains unit tests.
- The *sitexmlTask* is a task for Apache ANT which was written to generate an update site specification file *site.xml* directly from ANT [2].

The overall package structure of TRex is shown in figure 5.2. The first package with source code is *de.ugoe.cs.swe.trex.core*. Only general *UI* (User Interface) classes for the workbench extension are located here. The packages *editors*, *preferences*, *ui*, *properties*,

views, *actions* are also concerned with the UI provided by the plug-in. The *analyzer.rfparser*, *analyzer.astutil* and *misc* packages provide the parser, the symbol table and the necessary data structures for the implementation of refactorings. Finally, the refactoring package has three subpackages: *actions*, *core* and *ui*. The *actions* and *ui* packages are concerned with the user-interface of the refactorings while the *core* packages implements the source transformations. The UI code for the refactoring implementations is maintained separately as the refactoring implementation might be moved into its own Eclipse plug-in in the future.

The edited source code is parsed using ANTLR. The parser is based on the work of Wei Zhao [76]. However, the grammar files were heavily modified to create a syntax tree that is syntactically non-ambiguous. This change was imperative as refactorings often contain tree transformations and such transformed subtrees need to be converted into core notation again with the same syntactical notation as before. The syntactical analysis provided by the TTCN-3 parser is integrated into the user interface through problem markers. These markers interpret exceptions thrown by the parser and mark syntactically incorrect lines using a special problem view and editor annotations. The basic Eclipse editor with syntactical problem markers is based on the bachelor thesis of Jochen Kemnade [46]. His work also provides a basic semantical analysis and a symbol table. A symbol table is a data structure used in source analysis where each symbol in a source code is associated with information belonging to it, e.g. its type or scope. However, the semantical analysis and symbol table from this work was not used in TRex. The symbol table from [46] was incomplete and too simple in its design for the purpose of refactoring and further work. Therefore, the symbol table has been completely reimplemented. As a result the semantical analysis from [46] became completely incompatible and was therefore removed.

The symbol table and the pretty printer are implemented on top of the syntax tree which is created by the ANTLR parser. Although a metal-model as presented in [67] could have been used for the implementation of the refactorings, such an additional layer was not used. There is no complete documentation of a TTCN-3 meta-model and the extra effort would have been enormous compared to the benefits in the context of refactoring. The TRex syntax tree is syntactically non-ambiguous, i.e. it contains more information than a regular abstract syntax tree which only contains semantically complete information. Each node in this tree is an instance of the *LocationAST* class. The *LocationAST* class contains attributes for the parent node, corresponding start and end offsets as well as the line in the parsed source file, the associated token in the token stream and the associated scope.

This tree is used for the implementation of a pretty printer which traverses the tree for reconstruction of the TTCN-3 core notation. The pretty printer is used for two functionalities in TRex: it can be used for formatting the TTCN-3 source code according to rules specified in the IDE preferences and it is used to transform syntax subtrees into TTCN-3 core notation. The latter use is part of the infrastructure which can be used for the implementation of TTCN-3 refactorings.

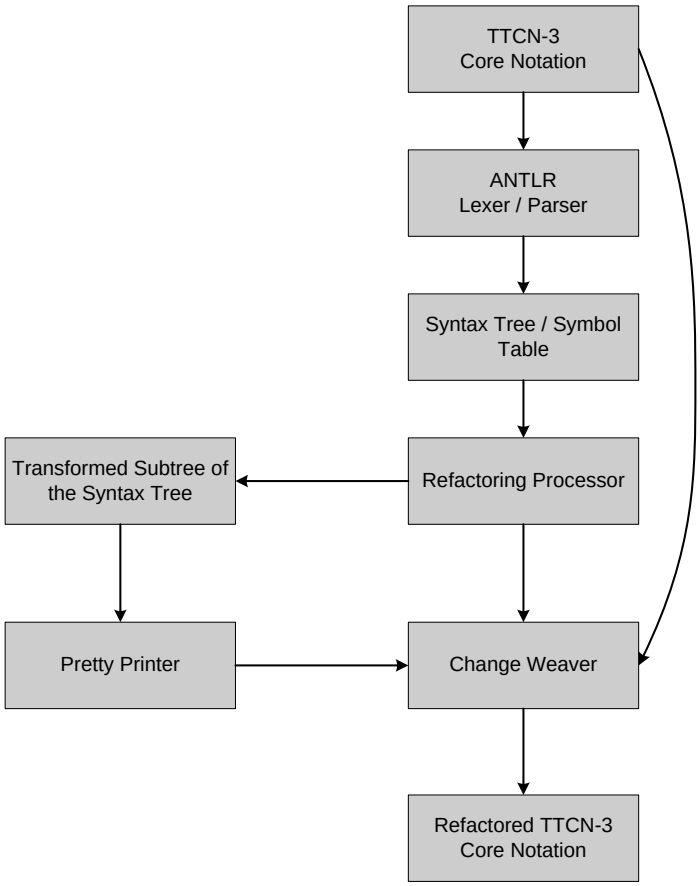


Figure 5.3: The Refactoring Tool

The TTCN-3 refactoring implementations are based on the syntax tree, the symbol table and the pretty printer. Other typical IDE features like the content assist use either the syntax tree, the symbol table or both.

Figure 5.3 shows the essential steps involved in a refactoring implementation. As stated before, source code in the TTCN-3 core notation is lexed and parsed using an ANTLR grammar. The resulting syntax tree is used to generate a symbol table. The refactoring implementations subsequently use the information provided by the syntax tree and the symbol table to generate the changes needed to transform the source code. These changes are generated in the refactoring processor. The changes can either be created directly from the symbol table and the syntax tree or they involve a tree transformation as well. The transformed subtree uses the pretty printer to create TTCN-3 core notation. The generated changes are then weaved into the original source code using a programmatic text editor to obtain the refactored core notation.

Refactorings as implemented in the current version of TRex are not completely automated. Like other popular refactoring tools, the developer still has to decide which refactoring he wants to apply and he has to specify where the affected code regions are located. Most refactorings need some additional information, e.g. how some identifier in the refactored code should be named.

5.2 The Pretty Printer

One part of the refactoring infrastructure in TRex is the TTCN-3 pretty printer. While the pretty printer can also be used on its own for source code formatting, its main purpose in the context of the refactoring tool is to recreate core notation from subtrees of the syntax tree. The pretty printer implementation consists of two core steps. The first step reconstructs core notation from a syntax tree (section 5.2.1) and the second step post processes the reconstructed core notation, e.g. for reintegrating single and multi-line comments (section 5.2.2).

5.2.1 The Tree Walker

The TRex pretty printer is primarily implemented through an ANTLR tree grammar used for structured tree traversal. ANTLR tree grammars are basically tree specifications which can be enriched with actions that are executed on tree traversal. In the case of TRex, such actions always contain Java code. The pretty printer is implemented in a special tree grammar file which contains actions. These actions have the sole purpose to reconstruct the TTCN-3 core notation from the information that is gathered while walking the tree. Certain aspects of the way how the code should be formatted can be configured through a Java Bean. Such aspects are the spacing type (tabs or spaces) or the placement or the curly braces (e.g. Kernighan and Richie style indentation [23]).

The *Visitor* design pattern is a popular alternative to the tree grammar approach. In the case of the available TTCN-3 parser, this would result in several hundred classes though. The tree grammar approach also has its drawbacks though. Concretely, having multiple tree walker grammars entails a maintenance problem as a change in the parser always results in a manual change every single tree grammar file. Unfortunately, there is no such thing as a tree grammar merge tool.

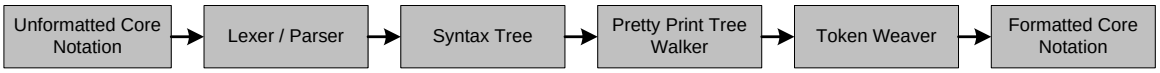


Figure 5.4: The Pretty Printer

Figure 5.4 shows the steps involved in pretty printing. A syntax tree from a source core notation is created through lexing and parsing. This tree is traversed using a special tree grammar for pretty printing. Finally, some post processing is done using a token weaver (see section 5.2.2) before finally the formatted core notation is ready.

```
1  pr_FunctionDef returns [StringBuffer s]
2  {   s=new StringBuffer();
3      StringBuffer a,b,c,d,e; } :
4  #(FunctionDef
5      { s.append("function "); }
6      a=pr_Identifier
7      { s.append(a).append("("); }
8      (   b=pr_FunctionFormalParList
9          { s.append(b); }
10     )?
11     { s.append(")"); }
12     (   c=pr_RunsOnSpec
13         { s.append(" runs on " + c); }
14     )?
15     (   d=pr_ReturnType
16         { s.append(" " + d);}
17     )?
18     {   s.append(getOpeningBrace());
19         currentNesting++;
20     }
21     e=pr_StatementBlock
22     {   currentNesting--;
23         s.append(e).append(getSpacing() + "}");
24     }
25 );
```

Listing 5.1: Pretty Printer Tree Grammar Example

Listing 5.1 shows a typical tree grammar rule with pretty printer actions. In line 1, the rule *pr_FunctionDef* is declared with a *StringBuffer* return value. This return value is used by rules which call *pr_FunctionDef*. Lines 2–3 contain rule initialization code where the variables needed for the actions are initialized. The subtree specification starts in line 4 where the rule tries to match a *FunctionDef* node. This specification is enriched with Java actions. For example, the action in line 5 makes sure that any matched *FunctionDef* node appends a string containing "function" to the result buffer. Lines 6–7 show how return values of rules are used for the core notation reconstruction. Finally, special Java methods and variables are sometimes used when the formatting should be configurable. The use of the *getSpacing* method in line 23 demonstrates this case. Depending on the pretty printer configuration, the spacing can use tabs or spaces. The number of tabs or spaces per nesting level is also configurable. Therefore, the spacing must be obtained by a configuration-aware function.

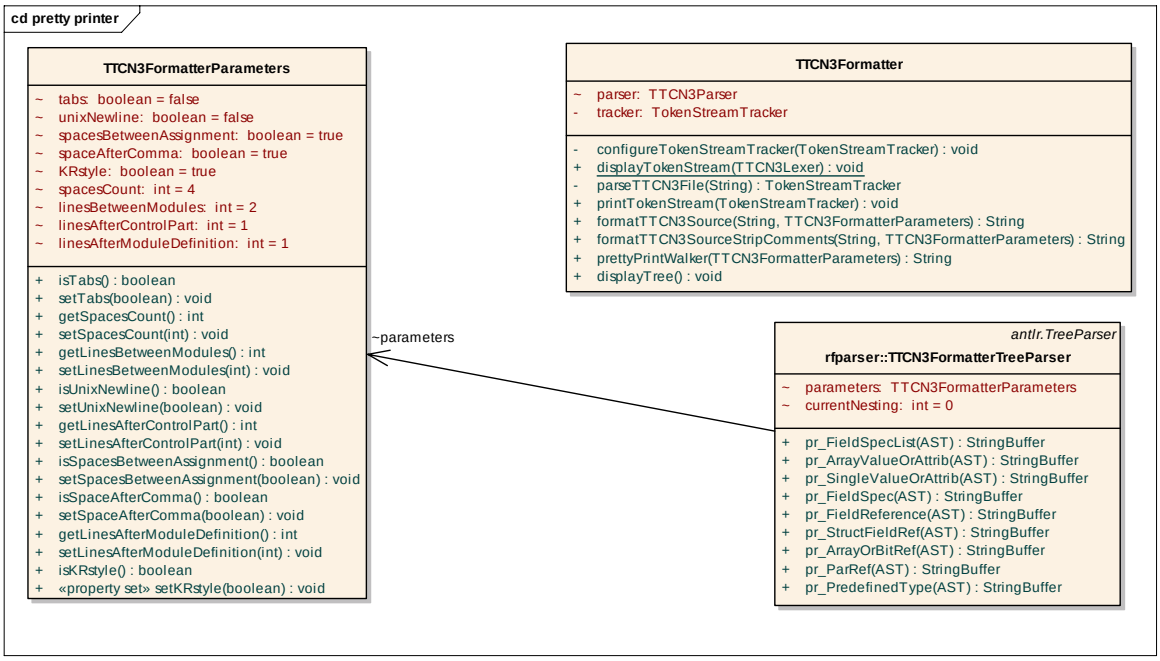


Figure 5.5: Pretty Printer Class Diagram

Figure 5.5 shows a high-level class diagram. The *TTCN3FormatterTreeParser* class is generated from the ANTLR tree grammar file (attributes and methods of this class have been shortened). This class is configured with a Java Bean called *TTCN3FormatterParameters* which carries information about specific formatting aspects. The *TTCN3Formatter* class is a facade providing a small, unified and easy to use interface.

5.2.2 Token Weaving for Comments

There is one major problem involved when writing a pretty printer on syntax tree level. This problem is related to the typical single and multi-line comments that are found in almost every source code. Scanners and parsers usually skip comment tokens, because they don't have any semantical meaning. At best, the parser can be modified to attach comment tokens to the nodes found in the syntax tree. However, normally syntax trees don't store every token, but only the ones needed to preserve the semantics (e.g. normally, semicolons are not stored in a syntax tree[1]). Therefore, comment tokens can't always be directly attached to

[1]In TRex, semicolons are actually stored in the syntax tree as they are often optional and must be reconstructed the way they are in the source notation. However, it is also possible to reconstruct the semicolon tokens using the token weaving technique.

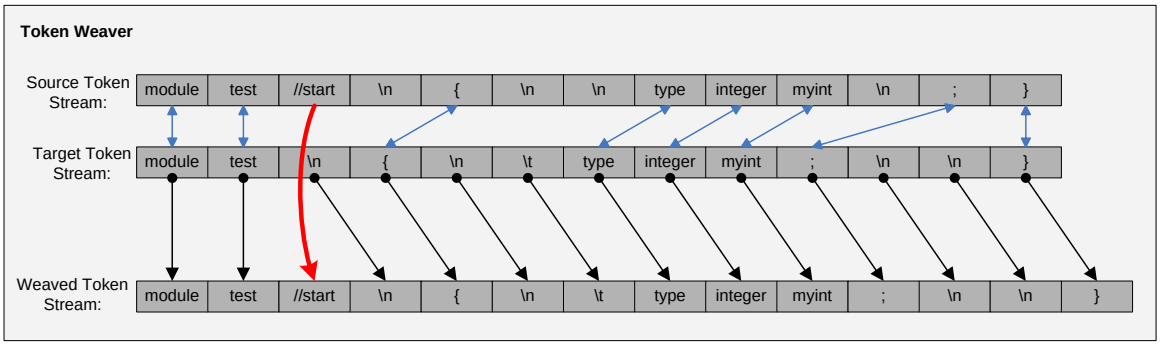


Figure 5.6: The Token Weaver

the tokens that follow or precede them. As a result, pretty printers based completely on syntax trees can't always restore comment tokens at their proper positions.

Another possible solution to overcome this problem is to write the pretty printer scanner-based. In this approach, the token stream of the unformatted source code would be analyzed and transformed into an formatted token stream using rules. Working completely on token streams can be complicated though as formatting decisions often have to be made from the context of a token which results in forward and backward token inspections per rule.

The TRex approach is different. It uses the syntax tree to generate pretty printed source code without comments. In a post-processing step, the missing comments tokens from the unformatted source code are weaved into the formatted source code. To do this, two token streams are needed and unfortunately, the formatted source code must be lexed again for this approach. Using this technique, the pretty printer benefits from the context information provided by the syntax tree and is able to place the comments at their correct positions. Figure 5.6 illustrates the token weaving process. The source token stream is the lexed token stream of the unformatted source code. The target token stream is obtained by lexing the output of the pretty printer tree walker which contains no comments. The tokens streams are then traversed token by token skipping whitespaces. Whitespaces must be skipped as they differ between the two token streams due to different formatting. Whenever a comment token is found in the source token stream, it is inserted into the target token stream. In practice, various special cases need to be handled within the weaving process to ensure correct formatting. For example, multiple subsequent multi-line comments should not generally be separated with a newline as they typically belong together.

5.3 Symbol Table

Generally, a symbol table can be defined as a data structure that stores information associated with each symbol found in the analyzed source code. This information is most importantly the symbol type, its scope and its location. A symbol table is needed in nearly every task that involves semantics simply because the type and scope information of an identifier is necessary to assess whether a statement in the analyzed source makes any sense. A simple example is an assignment to an undeclared variable. Such a variable must be declared and visible within the statement's scope. Otherwise, the statement would be illegal. Another common verification is called *type checking* where usage consistency between identifiers and their declarations is validated. For example, a string should not be assigned to a variable that is declared as integer. The step that involves such verifications is called *semantic analysis*.

5.3.1 Data Structure

Refactorings are transformations that need to maintain the semantics of the underlying source code. Hence, a symbol table is needed. A common data structure used for symbol tables are hash tables. A hash table looks up a key to find an associated value by transforming the key into a *hash*. This hash is then used to directly access the value's position. Thus, values are always stored unsorted. Hash tables provide a constant-time $O(1)$ lookup on average. In the worst case, lookup can be $O(n)$.

The TRex symbol table, however, is implemented using a red-black tree which essentially is a balanced binary tree (using the *TreeMap* class from the Java API). While averagely slower than hash tables, red-black trees have a good worst-case performance $O(log\ n)$ and maintain a sorted structure. The primary reason for choosing the red-black tree over the hash table was that the performance of the code-completion would benefit of a sorted structure (section 5.5.3).

Simpler data structures like arrays or linked-lists are inappropriate for symbol tables as the number of symbols within a source code can be very high. The performance would suffer.

Symbol tables can be implemented dynamically or statically. Dynamic symbol tables destroy symbol information as they walk along the tree. Therefore, dynamic symbol tables only make sense in the context of a single-pass analysis. In TRex, complete symbol information is needed as the source is edited or processed at all time. In addition, TRex is designed to work in multiple passes. Therefore, the symbol table is stored statically.

5.3.2 Design

Figure 5.7 shows the key classes participating in in the symbol lookup. The *SymbolTable* class encapsulates the underlying data structure of the symbol table making it easy to switch to a hash table for example. The scope class represents a single scope in a TTCN-3 source.

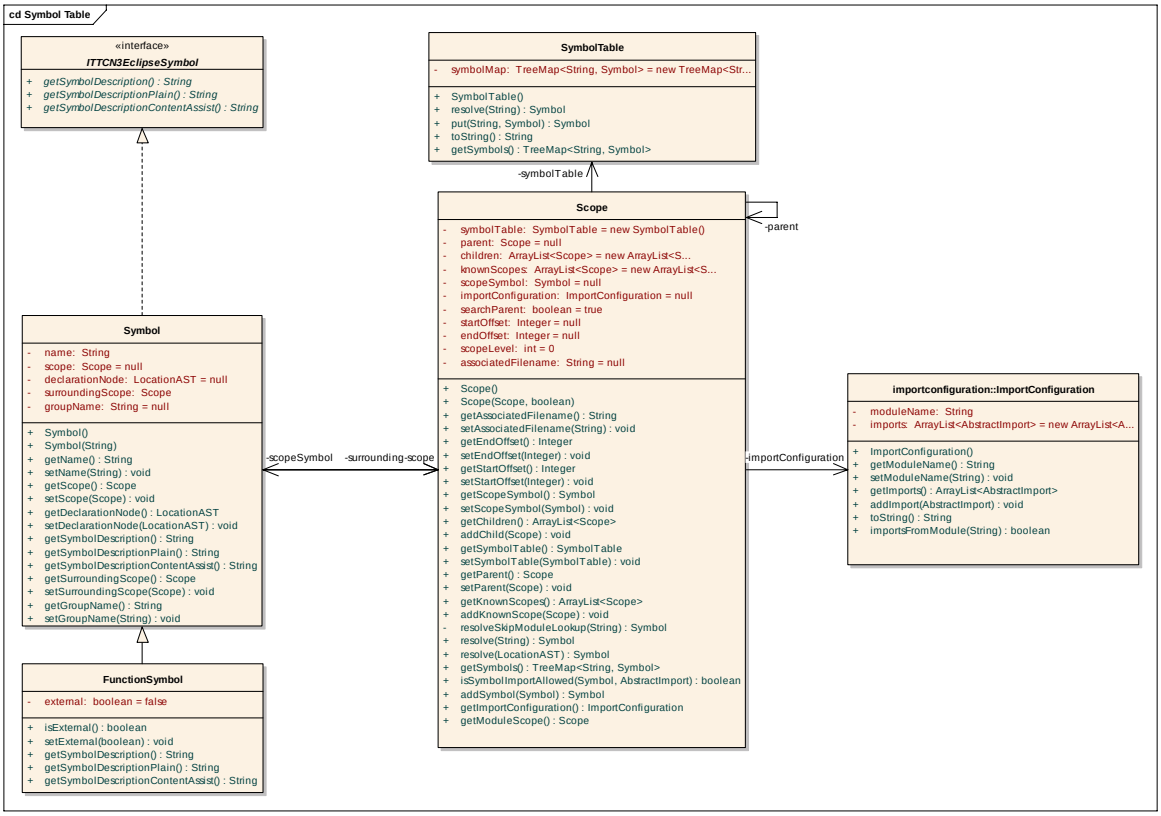


Figure 5.7: Symbol Table Class Diagram

Every scope contains a symbol table object. In addition, the scope provides several functions which are necessary or useful while working with symbols and scopes. For example, the scope contains attributes specifying the start and end offset within the source code file or its parent source filename.

A scope may have known scopes. Known scopes are used when the symbol lookup must also be performed in a different scope than the parent scopes. For example, when a function runs on a component, lookup through the parent scopes is not sufficient. The function scope must inspect the component scope as well. Similarly, super components are also known scopes to their sub components that extend them.

The scopes are organized within an n-ary tree[2]. While traversing the tree to build the symbol table, the scopes are tracked with a stack (as suggested in [22] for example). When-

[2]There are different interpretations about the meaning of n-ary trees in the literature. In some interpretations, n-ary implies that each node has n fixed children. Here, n-ary means that each node can have any number of children.

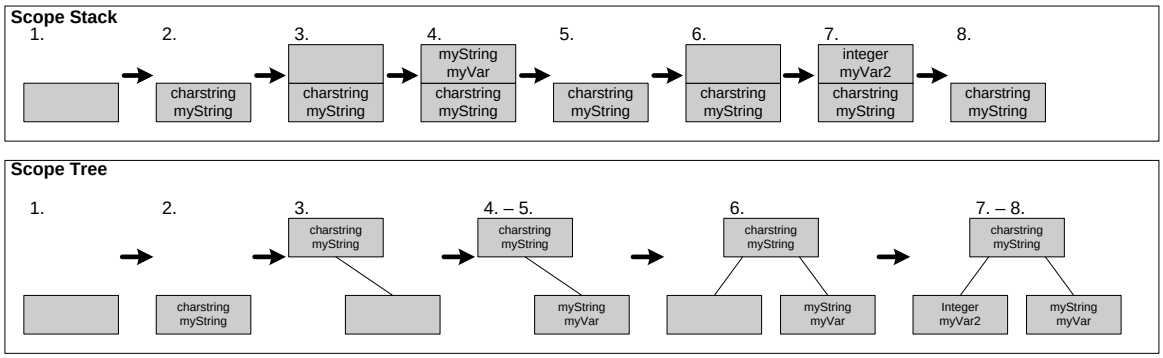


Figure 5.8: Scope Stack

ever a new scope is created, it is linked with the stack's top element and then pushed. If the scope end is reached, the stack is popped to get the last element again. Due to the tree linking on scope creation, the popped element is not lost though. A scope can be associated to a scope symbol, i.e. a symbol introducing a new scope. This can be a function symbol for example.

```
1  module scopeStackExample {
2
3    type charstring myString;
4
5    function myFunc1() {
6      var myString myVar;
7    }
8
9    function myFunc2() {
10     var integer myVar2;
11   }
12 }
```

Listing 5.2: Scope Stack Example

Listing 5.2 and figure 5.8 illustrate the way the scope stack works. The initial scope stack is empty. In line 1, a new module is encountered and hence a new scope is created (stage 1). Within the module scope, there is a type declaration *myString* (line 3) which is written into the scope's symbol table (stage 2). A function *myFunc1* (line 5) introduces a new scope which is pushed on the stack (stage 3) and linked as child to the module scope. The variable declaration in function *myFunc1* (line 6) is written into the symbol table of the stack's top scope (stage 4). When the end of *myFunc1* is encountered (line 7), the topmost element of the scope stack is popped (stage 5). However, it is still accessible through the scope

tree which is static. The handling of *myFunc2* (line 9) in stages 6-8 is analogous to stages 3-5. In line 12, the module ends and the module scope is therefore popped from the stack as well. Subsequently, the scope stack is empty, but the scope tree still exists. The static scope tree is created once along with lexing and parsing. The analysis of files within Eclipse projects is handled by the class *TTCN3ReconcilingStrategy* which is called when new files are opened, saved or after files have been changed and some time has passed. When the first file is opened, all files within the project are analyzed[3]. Once the project has been fully analyzed, only changed files are reanalyzed.

The *Symbol* class represents an abstract symbol within the symbol table. All concrete symbols, e.g. the *FunctionSymbol* shown in the diagram, are subclasses. In addition, each Symbol is associated with its surrounding scope and its corresponding node in the syntax tree. The *Symbol* class implements the *ITTCN3EclipseSymbol* interface which merely requires each symbol to implement some methods used for displaying symbol information within the IDE.

Each module scope contains an *ImportConfiguration* instance. This class (and its associated classes not shown in the diagram) handles the various import configuration possibilities and is used when resolving symbols across different modules.

5.4 TTCN-3 Refactorings in Eclipse

This section discusses the concrete implementation of TTCN-3 refactorings in TRex. First, the Eclipse refactoring component LTK (Language Toolkit) is introduced (section 5.4.1). Then, the identifier range map data structure used for identifier lookups is explained (section 5.4.2) and the *Rename* and *Inline Template* implementations are described (sections 5.4.3 and 5.4.4).

5.4.1 The Language Toolkit (LTK)

The language toolkit is a relatively new Eclipse API component introduced in Eclipse 3.1. Reoccurring language neutral parts of the JDT refactorings have been pushed down into this new layer with the aim to ease refactoring development for other languages. Due to its novelty, the API is not covered in any books and simple examples are rare [37]. Fortunately, the API is easy to use and the provided Javadoc documentation is detailed enough for actual usage.

The LTK consists of two library plug-ins: the *org.eclipse.ltk.core.refactoring* plug-in and the *org.eclipse.ltk.ui.refactoring* plug-in. The first plug-in contains classes necessary for the

[3]This process could be optimized by building a lightweight TTCN-3 parser which is only used to analyze the dependencies between the different files in the project through the import statements. Then, only dependent files need to be fully analyzed.

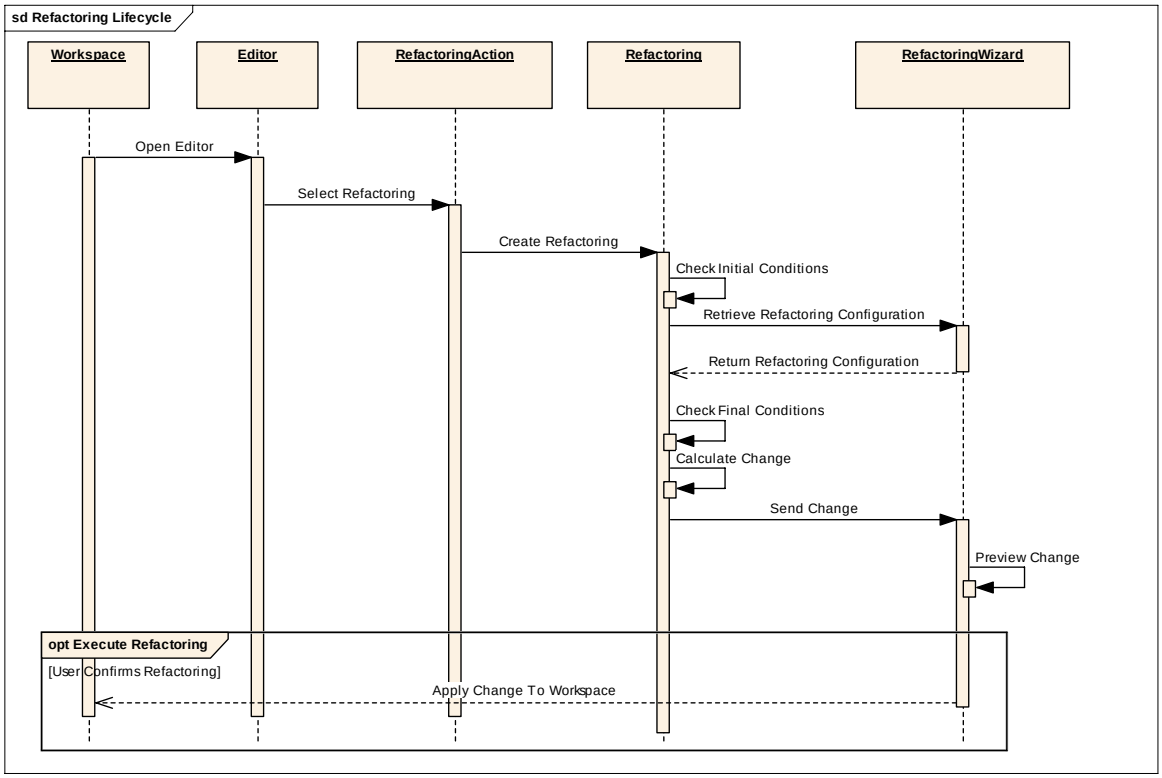


Figure 5.9: The LTK Refactoring Lifecycle

implementation of refactorings independently from any user interface. The second plug-in integrates refactorings which are implemented using the LTK core with the Eclipse workbench providing abstract wizard classes specifically for refactorings that can be subclassed. One key benefit of an LTK refactoring wizard is the integrated preview view where the original version and the refactored version of the source code can be compared side by side.

Figure 5.9 illustrates the typical life cycle of a refactoring in Eclipse. Starting with the workspace which manages projects, a file is selected and the corresponding editor is opened. Within the editor, an identifier or a region is selected and a refactoring is chosen (e.g. through a context menu). The refactoring can be selected since a *RefactoringAction* has been linked to an extension point within Eclipse. Selecting the refactoring causes the *RefactoringAction* to execute and a new refactoring instance is created. Now, the first validation *Check Initial Conditions* is started to assess whether the refactoring is actually possible within the specified context. If these conditions are met, a *RefactoringWizard* is opened and retrieves the information needed to successfully execute the refactoring (e.g. the

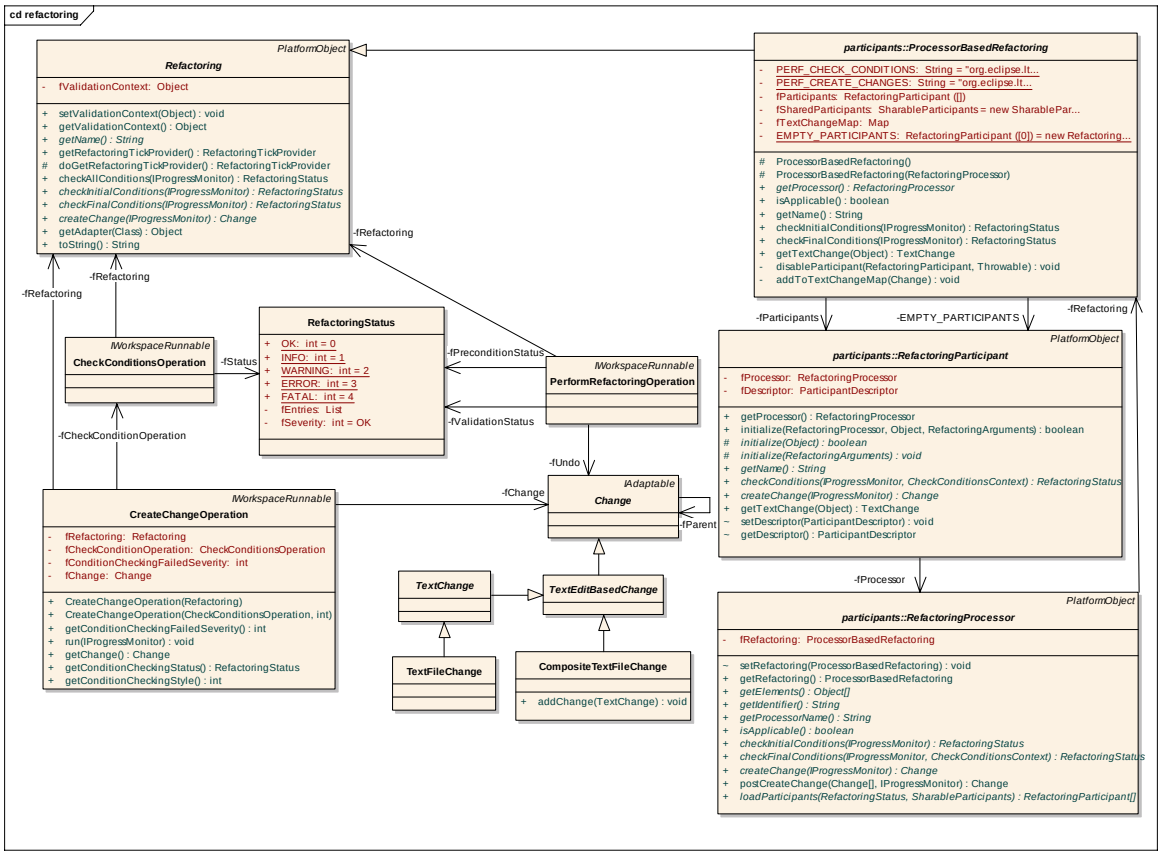


Figure 5.10: Language Toolkit Core Class Diagram

new identifier name in case of a rename refactoring). This information is then used for a last applicability verifcation *Check Final Conditions*. On success, the change for the refactoring is calculated and handed over to the wizard. There, the change is previewed to the user. If he accepts, the change is applied to the workspace. Otherwise, the workspace remains untouched.

Figure 5.10 shows a class diagram containing the key classes within the LTK core. The most important class is the *Refactoring* class. This class always needs to be subclassed by any refactoring implementation. This can be done either by direct subclassing or by subclassing *ProcessorBasedRefactoring*. The latter class supports refactoring participants.

Participants are objects that are supplied to the refactoring through an extension point and take part in the refactoring. This is useful when a refactoring may not only affect the resources that are changed by it directly, but also parts that are indirectly affected. A simple example for a participant is the *plugin.xml* in a PDE (Plug-In Development Environment)

project. This XML file refers to Java classes in the PDE project. When a Java class is renamed, the reference in the XML file becomes obsolete. However, the rename refactoring is implemented within the JDT and the PDE is a plug-in on its own. Therefore, PDE supplies a refactoring participant to the JDT rename refactoring that makes sure the *plugin.xml* is changed as well. Participants need to subclass *RefactoringParticipant*.

The class *PerformRefactoringOperation* is normally called by the LTK wizard. Conditions are checked through the class *CheckConditionsOperation* and created using the class *CreateChangeOperation*. These classes implement the *IWorkspaceRunnable* interface which provides an interface for running batch operations on the workspace. Finally, the changes that are created, are part of the *Change* class hierarchy. Normally, the changes used in source code refactorings are based on the class *TextEditBasedChange*. Specifically, subclasses can be created from *TextFileChange* for example. Should the refactoring affect more than a single file, changes can be collected in a *CompositeTextFileChange* class.

Implementing the user interface part involved in a refactoring is simple when using the classes provided in the *org.eclipse.ltk.ui.refactoring* package. Any custom wizard pages are subclasses of *UserInputWizardPage* and are added to a newly created *RefactoringWizard* which is associated with the concrete refactoring object. This wizard object is handed over to a *RefactoringWizardOpenOperation* class which is running on the workspace.

5.4.2 The Identifier Range Map

The one problem left before TTCN-3 refactorings can be implemented is finding the right nodes in the syntax tree depending on the cursor position or the marked text in the source code editor. The refactoring implementations need these nodes as input in order to calculate the necessary changes. For example, in the case of the *Rename* refactoring, the identifier which is marked by the cursor position should be renamed. However, the corresponding node must be found somehow and traversing the whole syntax tree for this particular node is not very efficient as such node lookups are not only needed by the refactorings, but also other commonly used functionalities like the text hover (section 5.5.1). An efficient and sorted data structure is needed storing integer ranges as key.

In TRex, every interaction between the syntax tree, the symbol table and the editor is based on text offset positions rather than lines and columns. This is the case, because Eclipse interprets tab stops differently in offsets and columns. The columns that Eclipse calculates uses the tab setting from the user preferences. Therefore, the columns can't be calculated consistently if user settings are disregarded. Text offsets on the other hand have a consistent tab stop interpretation. Some changes in the ANTLR TTCN-3 lexer and the *LocationAST* class were needed in order to track offsets though. Internet research revealed that this is a well known problem for ANTLR users. Unfortunately, all approaches found involved changing the ANTLR runtime. Yet the TRex offset tracking solution is implemented without changing it. The following changes were necessary:

- The custom *LocationAST* class needs attributes for the start offset and end offset.
- The *initialize* method in *LocationAST* must be adjusted to calculate start and end offsets for the node from a token.
- The *setLocation* method in *LocationAST* must be adjusted to set the offsets from start and end tokens.
- The TTCN-3 lexer needs an offset attribute.
- The TTCN-3 lexer *consume* and *makeToken* methods must be overridden to track offsets.

This way, the ANTLR runtime can be exchanged easily when new versions are available. Once the offsets are tracked, the tab stop size, i.e. the character length of a tab stop, of the ANTLR TTCN-3 lexer must be changed to match the tab size of Eclipse offsets. Afterwards, the Eclipse editor offsets are consistent with the syntax tree offsets.

Now that each node contains start and end offset attributes, they can be stored for efficient lookup in a data structure. Again, a red-black tree (implemented through a Java *TreeMap*) was chosen. The storage keys are not typical though as they must contain information about the start as well as the end offset (an offset range). A node query should be performed with single offsets and the lookup ought to find all nodes with a matching range.

Figure 5.11 shows the involved classes. *IntRangeMap* is a generic class that can store any value object *K* using a range key in a red-black tree where *K* is a *LocationAST* for the identifier map. The key is implemented in the class *Range* which must implement the *Comparable* interface. This key class needs customized *compareTo* and *equals* methods where ranges can be compared to ranges and points for example. The *Point* class is merely representing a single offset in this matter. The result of the *get(int, int)* method is not directly a *K* element, but an object of the type *RangeMapValue*. This class was introduced to represent multiple values matching a key when ranges overlap. Support for overlapping ranges is slightly more complicated and also implemented, but not actually needed in TRex (due to a change in its usage afterwards). The *get(int)* method is the method mainly used by TRex. It simplifies the usage for TRex purposes and only returns a list with the matching *K*s while neglecting the associated (possibly differing) ranges.

The described data structure is not only useful for storing identifiers with ranges, but also for storing scope information with offset ranges. For the content assist (section 5.5.3), it is necessary to know the corresponding scope for each position within a source file. Therefore, this data structure is used to store every scope with its start and end offsets. Once the right scope could be determined, finding all visible symbols was just a matter of adding a *getSymbols* method with the *Scope* class. In this case, the *Scope* class is used for *K*.

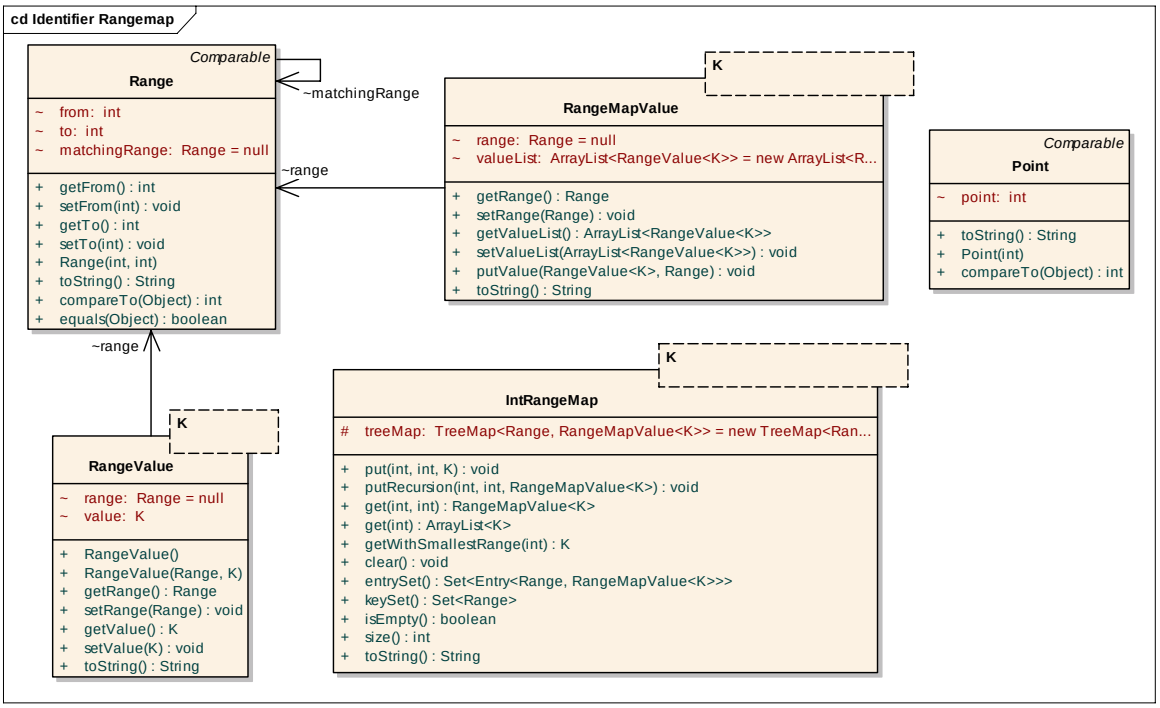


Figure 5.11: Identifier Range Map Class Diagram

5.4.3 The Rename Refactoring

The first TTCN-3 refactoring implemented in TRex is the *Rename* refactoring. It provides an easy way to rename identifiers (i.e. declarations and references) such as variable names, function names or module names. Unlike a simple search and replace function provided by nearly every text editor, the *Rename* refactoring regards the TTCN-3 scoping rules. Typical problem cases like shadowed variables or imported declarations are respected by the refactoring through symbol table lookups. A normal search and replace operation cannot guarantee this kind of correctness.

At its core is a reference finder algorithm which is very generic and thus already reused in several other functionalities of TRex such as the *Inline Template* Refactoring (section 4.2.6) and the Find References View (section 5.5.4).

Figure 5.12 shows the participating classes. The *Rename* refactoring is implemented as *RefactoringProcessor* and therefore allows participants if they should become necessary. *TTCN3RenameProcessor* is a subclass of *RenameProcessor*. The *RenameProcessor* class from the LTK API is supposed to be subclassed by *Rename* refactorings, but it actually does not provide any additional functionality over the *RefactoringProcessor* class.

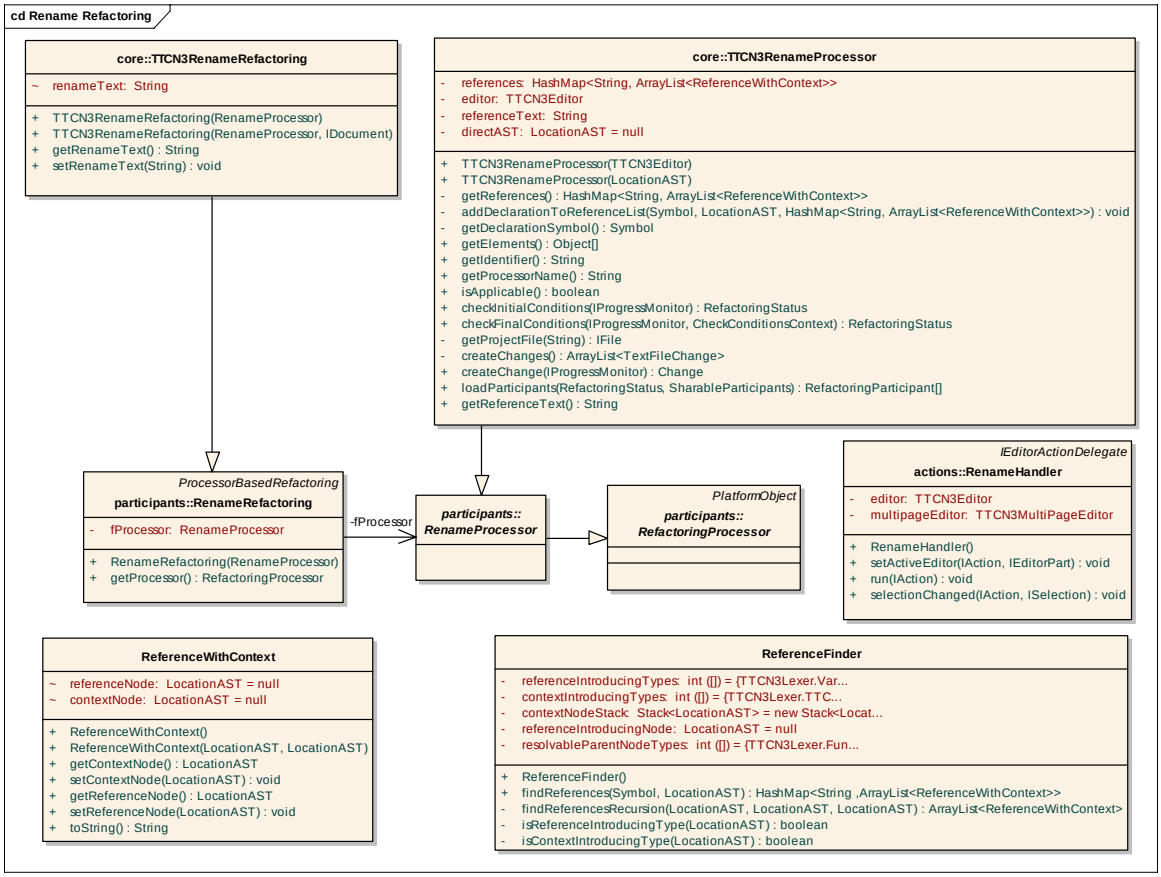


Figure 5.12: Rename Refactoring Class Diagram

All implemented TRex refactorings support two execution modes. A UI-based mode for direct use within the editor and a headless mode for refactorings that automatically find bad smells and for testing automated refactorings. For each mode, there is a different constructor in *TTCN3RenameProcessor*. In the UI-based mode, the active editor instance is passed as parameter. The refactoring then uses this editor instance to locate all necessary information. The headless mode requires the concerned identifier node as parameter. Due to the use of the refactoring processor, the complete refactoring logic is located in *TTCN3RenameProcessor* while the actual refactoring class *TTCN3RenameRefactoring* does not contain any behavioral code related to the rename refactoring. It merely carries the single attribute necessary for this refactoring: the target identifier name. This name is retrieved through the refactoring wizard (not shown in the diagram). The refactoring behavior is executed through the *fProcessor* delegate in the *RenameRefactoring* class.

```
1  public void run(IAction action) {
2      TTCN3RenameProcessor processor = new TTCN3RenameProcessor(editor);
3      TTCN3RenameRefactoring refactoring = new TTCN3RenameRefactoring(
4              processor);
5      TTCN3RenameRefactoringWizard wizard = new TTCN3RenameRefactoringWizard(
6              refactoring, RefactoringWizard.WIZARD_BASED_USER_INTERFACE);
7      wizard.setRenameText(processor.getReferenceText());
8      RefactoringWizardOpenOperation openOperation =
9          new RefactoringWizardOpenOperation( wizard);
10     try {
11         openOperation.run(Display.getCurrent().getActiveShell(),
12             "Refactoring not possible!");
13         IFile ff = (IFile) editor.getEditorInput().getAdapter(IFile.class);
14         editor.getReconcilingStrategy().analyzeAll(ff);
15     } catch (InterruptedException e) {
16         MessageDialog.openInformation(
17                 Display.getDefault().getActiveShell(),
18                 "Rename Refactoring",
19                 "Error while applying refactoring to workbench/wizard: "
20                         + e.getMessage());
21         e.printStackTrace();
22     }
23
24 }
```

Listing 5.3: run method of the Rename Refactoring Action

The refactoring itself is triggered by a context menu extension to the extension point *org.eclipse.ui.popupMenus*. The associated action class is *RenameHandler* which basically creates the refactoring object and runs the refactoring wizard on the workspace. Listing 5.3 shows the *run* method of the refactoring action. At first, a refactoring processor is created (line 2) which is used as delegate in the concrete refactoring (line 3). The refactoring object is passed to the wizard (line 6) and the wizard is initialized with necessary initial data (line 7). Finally, a wizard open operation is created and run on the workspace (lines 8–11). In the last step, the syntax trees are reanalyzed (lines 11–14). The action implementation is similar for other refactorings.

When the wizard is run on the work space, the initial conditions are checked. The refactoring cannot be applied when one of the following conditions is true:

- The TTCN-3 source is syntactically incorrect.
- The reference finder has a fatal error.

The final conditions are more interesting as they can incorporate semantical verifications concerning the new name:

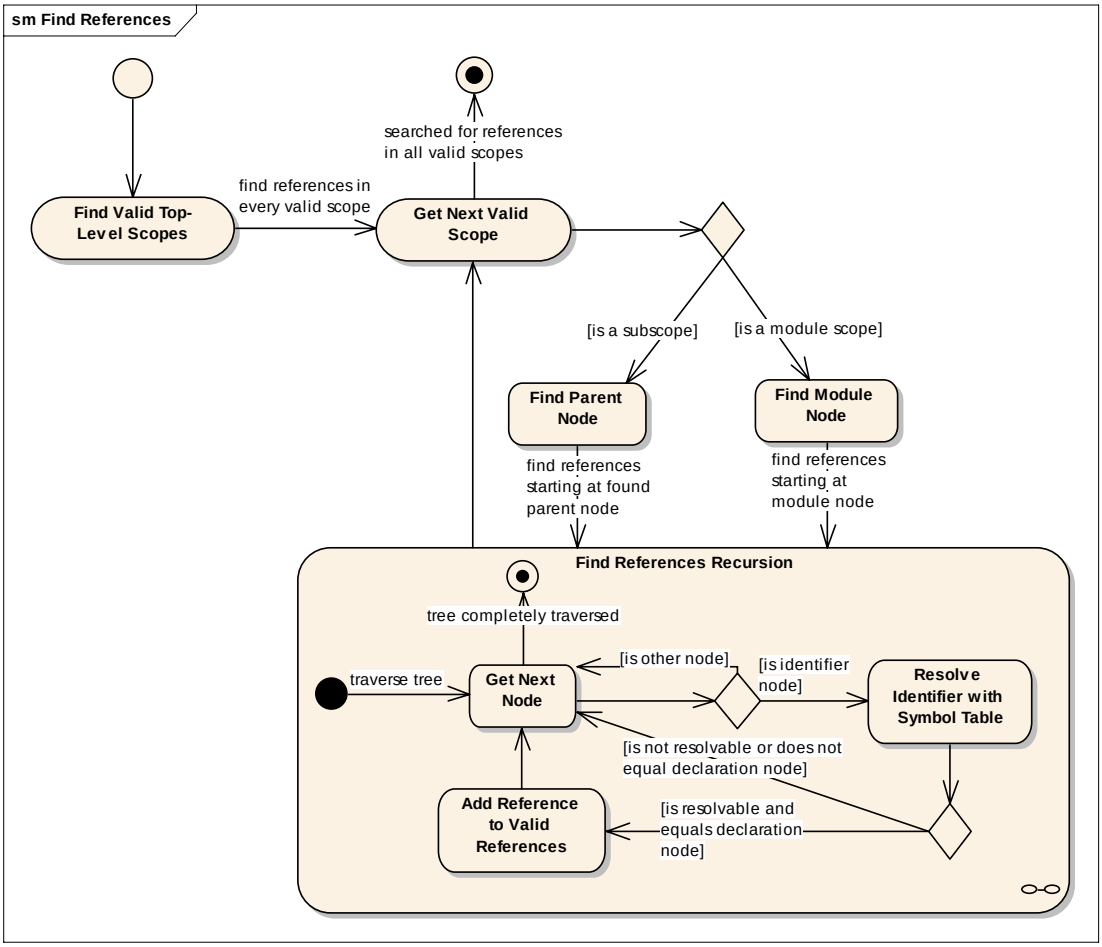


Figure 5.13: Find References State Machine Diagram

- The target name must not be empty.
- The target name must be different from the original identifier name.

The implemented conditions are the most simple ones. They can be extended with various additional verifications such as checking for name clashes in the same scope or subscopes. A formal description of these additionally possible conditions can be found in [61].

The *TTCN3RenameProcessor* instance creates a *ReferenceFinder* object used to retrieve references of the currently selected identifier's declaration.

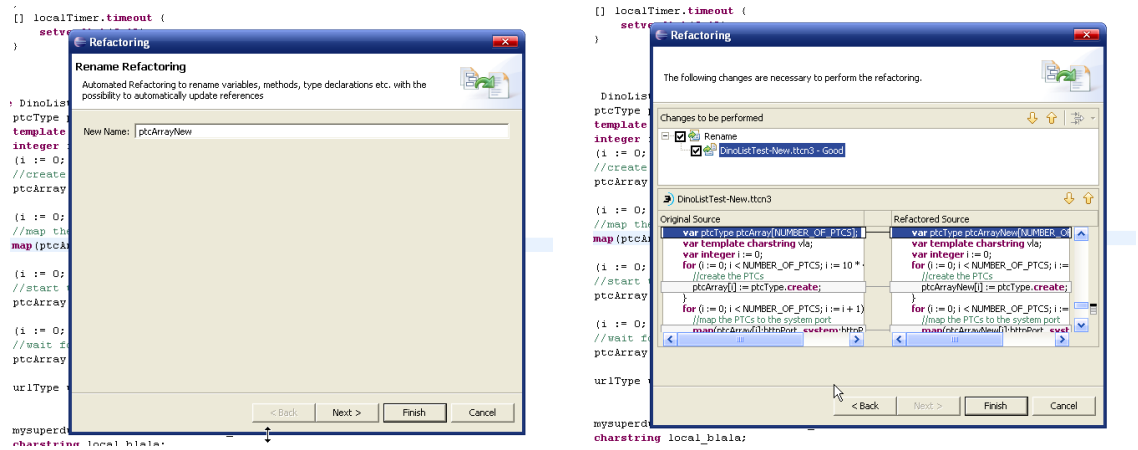
Figure 5.13 shows a state machine diagram with the vital behavior of the reference finder algorithm. The first step is to find the declaration of the selected reference. The declaration

node is passed to the reference finder algorithm. The algorithm starts with finding all scopes that are candidates for reference inspection. For symbols declared on module scope, every module scope that imports from the module where the concerned symbol is declared and where the symbol is actually resolvable is such a candidate. For declarations in subscopes (e.g. variable declarations in a function), the declaration scope is the candidate directly. In the next step, every valid scope is inspected for references. For every scope inspection, a parent node has to be found where the tree traversal starts. In module scopes, this parent node is a *TTCN3Module* node while subscopes can have various parent nodes such as *FunctionStatementOrDef* or *AltstepLocalDefList* nodes. Therefore, this parent node lookup depends on the scope type. Once this node is found, a recursive tree traversal is started. In this traversal, every identifier node found needs a symbol table lookup. The identifiers that can be resolved and are equal to their declaration symbol, are references to their declarations. In any other case, they are not references.

Once all references for the declaration are obtained, renaming is just a matter of invoking a programmatical editor which is offered through the Eclipse API. The package *org.eclipse.text.edits* provides the classes needed. Concretely, the *ReplaceEdit* and *MultiTextEdit* classes are used. As the name suggests, the *ReplaceEdit* class is used to replace text in a document. The *MultiEdit* class is really useful as it lets you apply multiple edit operations on a document using the original offset positions of the file to be edited. Concretely, this means that it is not necessary to account changing offset positions after each replacement in the source document. The *MultiEdit* class performs this task already.

The edit operations are added to the change object which is then returned by the refactoring in the *createChange* method. As refactorings may affect multiple files in the workspace, this change is a *CompositeChange*.

Figures 5.14(a) and 5.14(b) show the wizard pages presented to the user. In figure 5.14(a), the user is presented a page containing a single text field where the concerned identifier name is inserted as default value. On this page, the user modifies the identified name to reflect the target identifier name. When finished, the "next" button must be pressed. If the final conditions are fulfilled and the refactoring change can be generated successfully, the preview page is presented (figure 5.14(b)). The upper part of this wizard page displays the workspace files which are affected by this refactoring. In addition, it is possible to deselect the changes for every single file using a check box next to the file name. In this case, the refactoring will omit changes to the deselected files when the refactoring is applied to the workspace. The lower part of the page shows the original source on the left and the refactored source on the right. Every affected line is marked and connected between the source viewers. This way, the changes to each line can be carefully reviewed manually. The source code viewers on the preview page are provided through the classes *TTCN3ContentMergeViewer* and *TTCN3ContentViewerCreator* and reuse the syntax highlighting support from the editor infrastructure.



(a) Custom Page for the Refactoring Configuration (b) Preview Page

Figure 5.14: Refactoring Wizard Pages for Rename

5.4.4 The Inline Template Refactoring

The *Inline Template* refactoring (section 4.2.6) is the second implemented refactoring in TRex. It is able to replace a template reference with its inline notation and may help to improve readability of the source code.

The overall class structure (figure 5.15) of the refactoring is very similar to the *Rename* refactoring. The refactoring behavior is located in *TTCN3InlineTemplateProcessor* which enables the use of participants again. As in the *Rename* refactoring, there is a headless and a UI-based mode for the refactoring. *TTCN3InlineTemplateProcessor* therefore has two different constructors. The *TTCN3InlineTemplateRefactoring* uses the processor as delegate through *ProcessorBasedRefactoring* and contains attributes for the refactoring configuration. Two options are possible for inlining templates:

- Deletion of the declaration if the declaration is referenced only once.
- Apply the pretty printer formatting rules on the inlined template.

The corresponding attributes in *TTCN3InlineTemplateRefactoring* are *deleteDeclaration* and *prettyPrintInlinedTemplates*. The attributes are initialized on refactoring execution by a custom wizard page containing a checkbox for each attribute. The action is handled by the *InlineTemplateHandler* class and works similar to listing 5.3. There are three main cases that need to be distinguished in the refactoring behavior:

- The template to be inlined is a normal template.

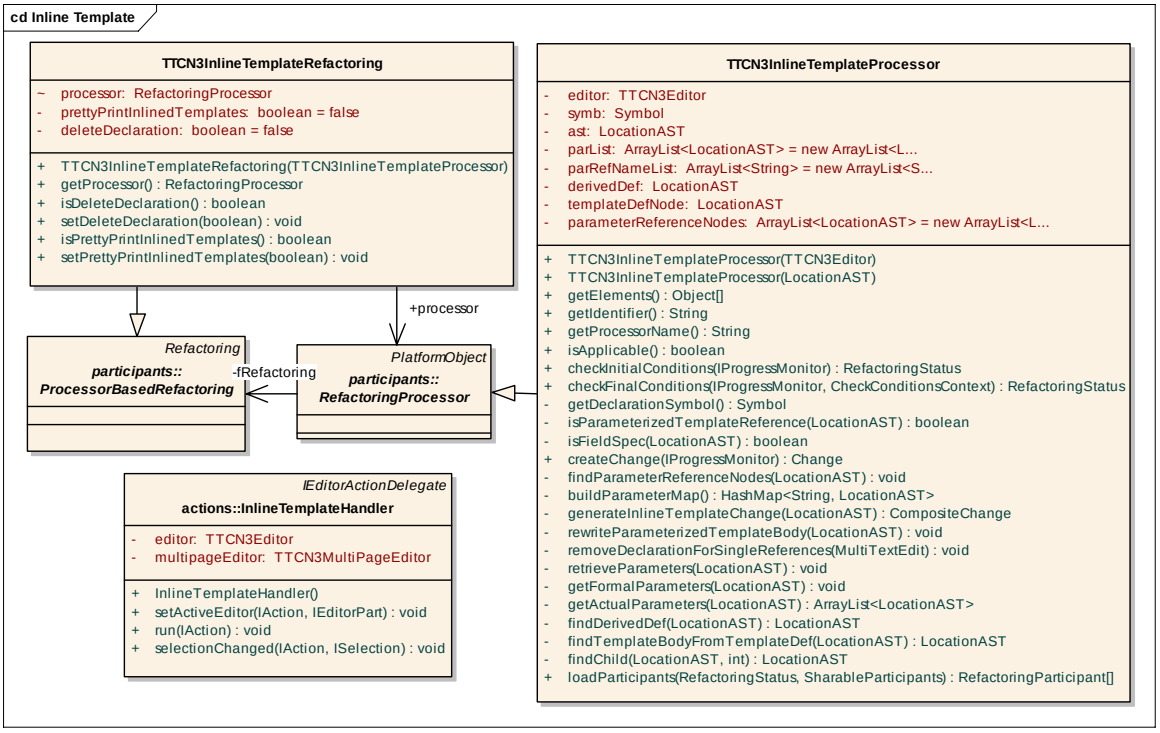


Figure 5.15: Inline Template Class Diagram

- The template to be inlined is a modified template.
- The template to be inlined is a parameterized template.

The first step in the refactoring behavior (figure 5.16) is to resolve the identifier using the symbol table to get the template declaration. If this template is a parameterized template, the template body subtree must be copied and altered to inline the parameters of the reference. Otherwise, the template body stays the way it is (the differences between normal templates and modified templates are handled in the rewrite step). In the next step, the find references algorithm from section 5.4.3 is used to find all references to this declaration. If only a single reference is found, the selected reference is inevitably the found one. In this case, the declaration can therefore be removed if the option was chosen in the refactoring configuration. In the last two steps, the pretty printer is applied on the template body subtree and rewritten to form a syntactically correct inlined template. The concrete rewrite step depends on the template declaration and is slightly different for modified templates.

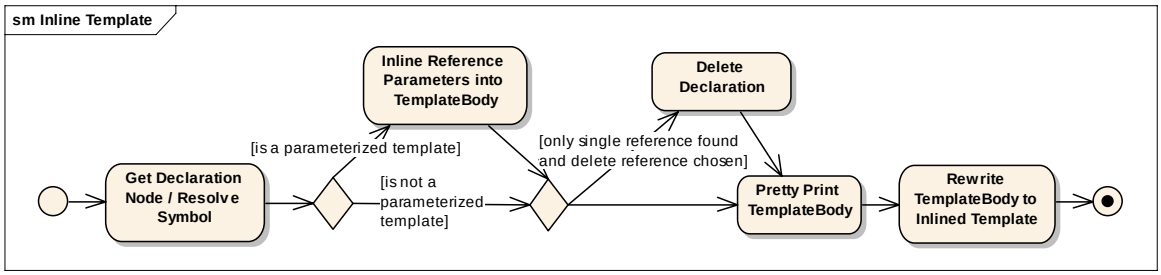


Figure 5.16: Inline Template State Machine Diagram

The initial conditions for *Inline Template* are:

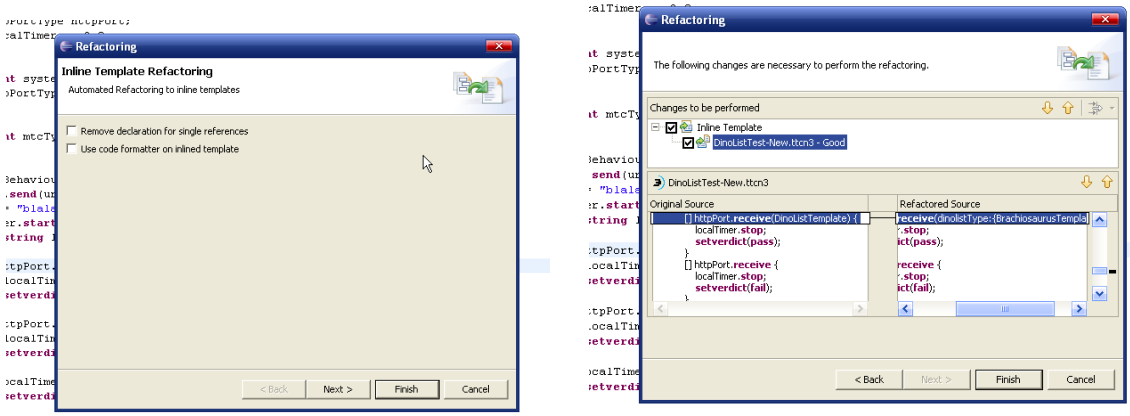
- The TTCN-3 source must be syntactically correct.
- The identifier must be a template reference.
- The parameter count of the template declaration and the template reference must match.

Again, these conditions can be extended. Such a possible extension is parameter type verification. There are currently no implemented final conditions. The actual edit operations for both replacing template reference with inlined template and declaration removal are realized with the *ReplaceEdit* class.

Figures 5.17(a) and 5.17(b) show the wizard pages presented to the user. Similar to the *Rename* refactoring implementation, the first page is used for the refactoring configuration (figure 5.17(a)). The user can choose two options realized through check boxes. If the first option is checked, the refactoring will remove the template declaration if the inlined template was the only reference to this declaration. The second options uses the pretty printer configuration on the inlined template. The default setting is to write the inlined template in a single line. The preview page on figure 5.17(b) is analogous to the preview page for the rename refactoring (figure 5.14(b)).

5.5 Further Functionality in TRex

While building the infrastructure necessary for realization of the refactorings, the implementation of several other useful features known from advanced IDE's became easy. Therefore, they have been implemented in TRex as well. Most of this work is based on the information obtained from the symbol table. An abstract text hover and content assist implementation is provided by Eclipse.



(a) Custom Page for the Refactoring Configuration

(b) Preview Page

Figure 5.17: Refactoring Wizard Pages for Inline Template

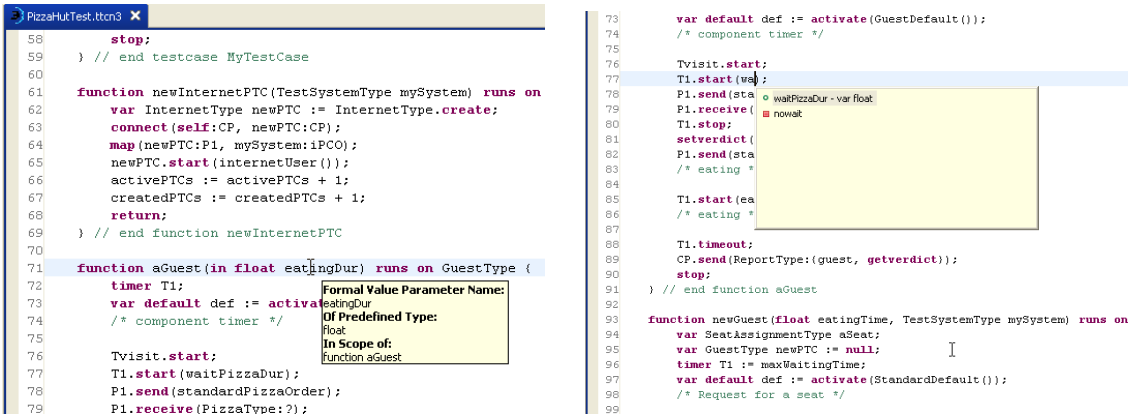
5.5.1 Text hover

The text hover is used to display information on specific parts of the source code in a pop up window depending on where the mouse cursor is located. In TRex, this functionality is used to display symbol information when the mouse cursor is moved over an identifier. This symbol information displayed is the symbol name, its type and the symbol declaration's scope. Figure 5.18(a) shows such a text hover pop up window.

In the text hover implementation, the identifier range map and the symbol table are used. The range map is used to obtain the identifier node located at the mouse cursor position. Using the identifier's scope and the identifier name, the symbol information can be found easily with the symbol table. The symbol information displayed is formatted using *HTML* [15] and is provided by every symbol implementing the *ITTCN3EclipseSymbol* interface through the *getSymbolDescription* method.

5.5.2 Open Declaration

The *Open Declaration* feature is triggered by the corresponding entry in the context menu or by using the hot key F3. Its purpose is to directly jump to the declaration of a reference within the editor. This way, a possibly time consuming search for the declaration is avoided. If the declaration is located in a file not currently open, the relevant editor window is made the active window and opened if necessary. The implementation is very similar to the text hover implementation. First, the identifier node is found through the identifier range map. Then the identifier symbol is resolved using the symbol table. The obtained symbol information contains the associated workspace filename and file position necessary to implement the workbench behavior.



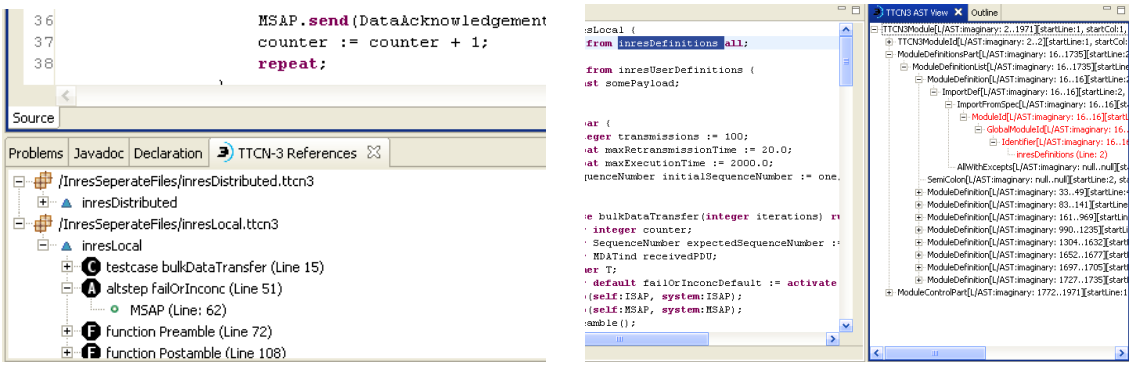
(a) Text Hover (b) Content Assist

Figure 5.18: Further Functionality

5.5.3 Content Assist

The content assist functionality is arguably one of the more important functionalities in modern IDE's. Being human, programmers don't remember every detail of their work. This is especially true when they deal with complex TTCN-3 test suites with thousands of lines and hundreds of different type and template declarations. Luckily, the content assist feature can take some guesswork off the programmer by making completion proposals depending on what is already typed into the editor. For instance, when the programmer starts to type the name of a variable name he wants to use, but then cannot remember exactly the rest of the word, he can press CTRL+Space and a pop up window with completion proposals is shown (figure 5.18(b)). Continued typing narrows down the displayed proposal list on the fly. Hence, the programmer need not know every single name within his code and he can easily use long (and therefore possibly more descriptive) identifier names. In the latter case, content assist mitigates the negative effects such as increased typing effort.

Naturally, these proposals need to be context aware. While simply displaying all declared identifiers in this list might help to some degree, it is not exactly helpful when the proposal list possibly has several hundred entries. Therefore, the information from the symbol table is used once again. However, instead of resolving a single symbol, all symbols visible in the scope of the current cursor position are collected. That way, all symbols that are not within the current scope are omitted and not proposed. The scope is obtained by a scope range map which is created at the same time as the symbol table. This map is very similar to the identifier range map, but it stores scope information instead of identifier information. As the symbol table uses a red-black tree, the data-structure is sorted and collecting symbols depending on the partial identifier already typed is efficient ($O(log\ n)$).



(a) Find References (b) AST View

Figure 5.19: Further Functionality

The semantical intelligence of the TRex content assist is currently limited to the use of scope information. It can be extended by using more semantical information. For example, when typing the right side of an assignment, the proposal list can be narrowed down further to the symbols compatible with the left side of the assignment.

5.5.4 Find References

Find References is located in the editor context menu and can be used to find all references to a declaration within the whole workspace project. It can be triggered either by positioning the editor's cursor over the identifier of a declaration or a reference to it. On execution, the *Find References View* is opened in the workbench displaying the results in a tree structure (figure 5.19(a)). The nodes in this tree can be double clicked to jump to the corresponding line in the editor. *Find References* can improve the programmer's understanding of the source code he edits and also provide useful information for the realization of manual refactorings. This is the case as declarations and its references are often part of refactorings.

The implementation of this view once more uses the reference finder algorithm presented in section 5.4.3 and directly uses the obtained information to create the result tree in the *Find References View*.

5.5.5 AST View

The *AST View*[4] displays the syntax tree of the source code in the active editor window. In addition, the tree is sensitive to the text marked in the active editor, i.e. the corresponding

[4]Strictly speaking, the TRex syntax tree is not abstract as it contains syntactical elements without semantical meaning. The view is still uses the term *AST* as this abbreviation is used throughout the whole

nodes are expanded and marked with red color. The view is primarily useful for the development of TRex as the tree structure of certain source code parts can be inspected easily. Figure 5.19(b) shows the *AST View* coloring and expanding the marked text in the source editor on the left.

5.6 Testing and Building TRex

Quality assurance is an issue for every software product. Unit tests are an important part of quality assurance and their purpose is to get confidence that the tested software modules work as expected. Section 5.6.1 describes the unit test infrastructure of TRex which is based on JUnit.

Also important and often neglected is a build system which can be executed on a regular basis (e.g. nightly builds) and are reproducible in their behavior. Section 5.6.2 explains the build technique used.

5.6.1 Unit Tests

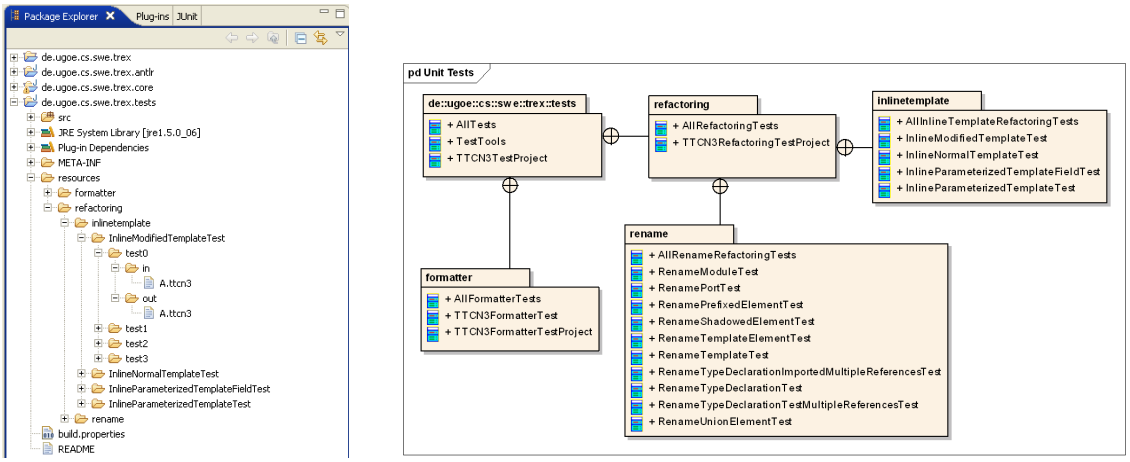
In TRex, there are unit tests for the *Rename* refactoring (13 Tests) and the *Inline Template* refactoring (12 Tests). In addition, there are also unit tests for the pretty printer (6 Tests). The realization of the tests is similar to the refactoring tests of the JDT [29]. These tests run on hand-picked scenarios and they should be able to find any crude defects.

The unit tests are located in a separate Eclipse project. While the first intuitive idea may be to put the unit tests directly into the concerned plug-in, this turns out to be a bad choice as this results in a dependency to the JUnit plug-in. Obviously, the dependencies of the TRex plug-in should be kept to a minimum. Therefore, the tests are located in a separate plug-in called *trex.tests*.

In this plug-in, there is a folder *src* containing the JUnit tests and a folder *resources* containing the resource files necessary for the tests. The resources are structured using directories. For example, the tests for inlining modified templates is located in the subdirectory *resources/refactoring/inlinetemplate/InlineModifiedTemplateTest*. Starting from this directory, the resources are consecutively numbered (e.g. *test0*, *test1*, *test2* etc.). Each numbered directory contains the subdirectories *in* and *out*. The *in* directory contains the files in which the refactorings should be applied. The *out* folder contains the files with the refactoring correctly applied. Figure 5.20(a) shows this directory structure in the Eclipse package explorer.

In each unit test, a workspace is created from an *in* subdirectory. The unit tests then execute a refactoring on the workspace in headless mode (i.e. the refactoring does not require

ANTLR and TRex API.



(a) Unit Tests Resources Directory Structure

(b) Unit Tests Package Diagram

Figure 5.20: Unit Tests Structure

any user input). Finally, the refactored workspace is compared against the *out* subdirectory. If the concerned files are not equal, the refactoring was not successful and the unit test fails.

The folder containing the JUnit source code is structured analogously (figure 5.20(b)). Each package containing unit test classes contains an additional class where these unit tests are combined (e.g. *AllInlineTemplateRefactoringTests*).

Furthermore, the classes *TTCN3TestProject* and *TTCN3RefactoringTestProject* are used for setting up the testing workspace and providing a common infrastructure for the workspace comparison (figure 5.21). The *TTCN3RefactoringTestProject* class is a subclass of *TTCN3TestProject* and therefore inherits its functionality to setup a new testing workspace. The *TTCN3RefactoringTestProject* provides additional functionality for application of refactorings to the workspace and the comparison afterwards. These classes are used as delegate rather than superclasses by the testcases (e.g. *InlineNormalTemplateTest*, *InlineParameterizedTemplateTest* or *InlineModifiedTemplateTest*) to allow multiple workspaces and tests within a single JUnit test case class. Each testing method in a test case (e.g. *testWithoutImport*, *TestWithImport* etc.) needs to setup its workspace using the *TTCN3RefactoringTestProject* instance and when finished, it needs to call the *dispose* method to destruct the workspace. Due to the use of the *TTCN3TestProject* and *TTCN3RefactoringTestProject* classes, writing the test cases is very easy and code repetition is minimized. The tests are executed by selecting "Run As" and "JUnit Plug-In Test" with the *AllTests* class opened in the Eclipse workbench.

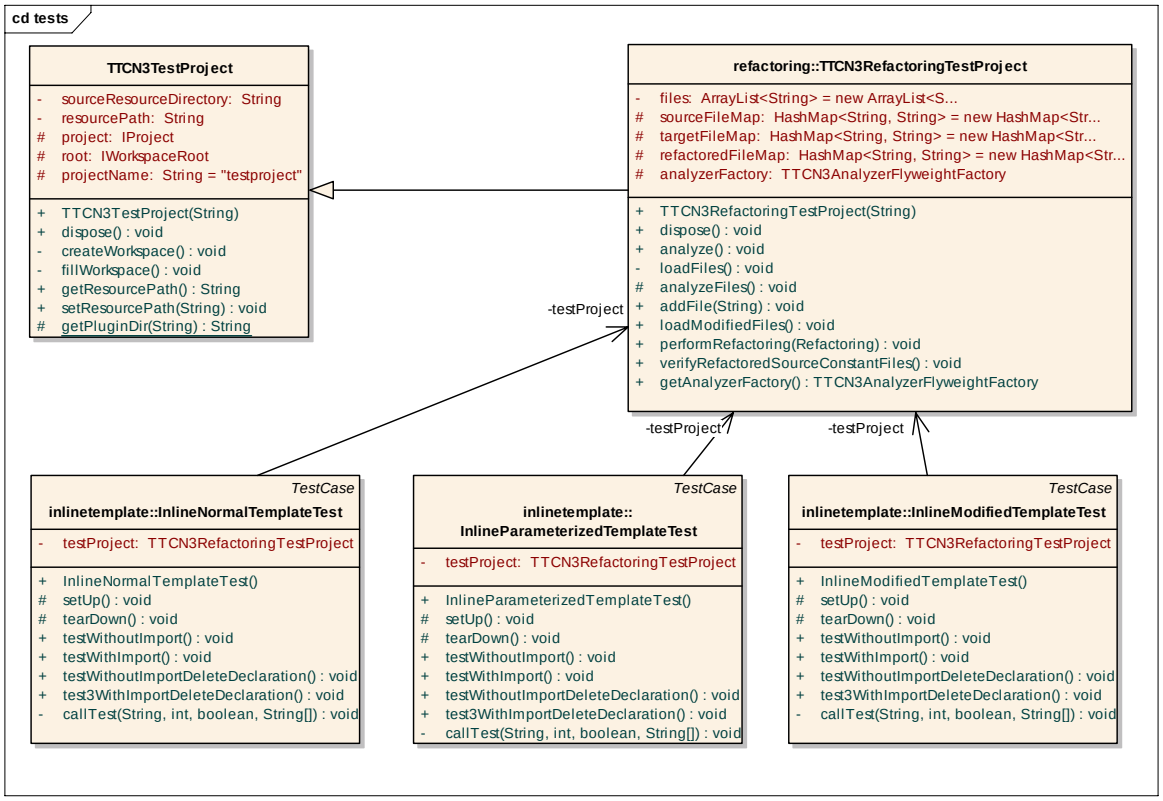


Figure 5.21: Refactoring Unit Tests Class Diagram

5.6.2 The Build System

The easiest way to build deliverable Eclipse plug-ins is to use the integrated export wizard. Exporting can become tedious when more than a single target platform is involved (e.g. plug-in builds for different Eclipse versions). At some point, every growing software product needs a consistent, automated and reproducible build system. TRex offers such a build system. It is based on the same infrastructure as the PDE export wizard and therefore makes use of all necessary details stored within the project configuration. This infrastructure is called *PDE Build*.

The TRex build system is located in its own Eclipse plug-in called *de.ugoe.cs.swe.trex.build*. This is common practice for Eclipse plug-ins although they do not actually extend the Eclipse workbench in any visible way. To build TRex, this single directory is the only needed. The TRex build system is based on Apache ANT [2] and is actually consisting of two separate parts: a custom front-end and *PDE Build*. The custom front-end is found in the file *build.xml* and invokes *PDE Build*. Although *PDE Build* itself is also based on Apache ANT,

this custom front-end is necessary for two reasons: calling *PDE Build* is complicated as it needs to execute Eclipse in headless mode and secondly, *PDE Build* does not support Subversion [11] which is used for version control in the TRex development. In the normal case, *PDE Build* uses CVS [3] checkouts and updates to create a directory structure with up to date plug-in source codes. Due to the use of Subversion, this internal feature could not be used. Luckily, there is an ANT task available called SVNANT [12]. Using *SVNANT*, the source directory creation and the Subversion based update handling is part of the custom front-end. For ease of use, SVNANT libraries are distributed with the build plug-in and have been modified to use JAVASVN [7] to avoid dependencies on native external binaries. In addition, the ANTLR library is included which is used by *PDE Build* to generate Java source from the grammar files. So the *build.xml* basically creates or updates the necessary source directories for the plug-in and then invokes *PDE Build* by calling Eclipse in headless mode.

However, there is some configuration needed to build TRex. After all, the build system is flexible and can build against various targets for example. The only file that must be configured is *build.properties*. In most cases, only the fields *updateSiteLocation*, *base*, *baseLocation*, *pdeBuildPluginVersion* and *buildDirectory* are relevant for the build configuration. *base* and *baseLocation* point to an installed Eclipse SDK which is the build target. *pdeBuildPluginVersion* is the directory name of the PDE Build plug-in in the base Eclipse SDK. This directory varies depending on the Eclipse version used. The *updateSiteLocation* points to the location of the update site. This directory should either point to a directory directly within the access of a web server or it should synchronized to a web server regularly. Finally, the *buildDirectory* is the directory where the TRex plug-ins are built. The content of this directory is automatically created through the ANT front-end through subversion checkouts and updates. The TRex build system not only creates archives containing a distributable versions of the plug-in. It also automatically creates an update site. Update sites consist of a directory structure with a special layout and a file called *site.xml* containing the specification for the update site. The update site directory can be uploaded to a web server. Eclipse clients use the URL to the update site directory to download the latest versions of the plug-in. These updates can happen automatically which makes update distribution very easy. Unfortunately, *PDE Build* does not provide any functionality for directly creating the update site specification file *site.xml*. Therefore, a small ANT task called *sitexmlTask* has been written to create a valid update site from the directory created by *PDE Build*.

The *PDE Build* process itself was customized as well. Luckily, *PDE Build* provides hooks in its targets which can be used for extension. These hook targets are located in *customTargets.xml*. Using these hooks, the build system is able to generate the TTCN-3 Java Parser from the ANTLR grammar files before compiling the plug-in and creates the update site in the very last step. The whole build process is basically started as usual by running ANT on the *build.xml* file at the command line.

6 Conclusion

Software aging is a problem that concerns not only ordinary software, but also tests written in TTCN-3. They also need to be maintained and hence must be easy to understand on the one hand and reusable on the other. Typical pieces of code with quality problems are called *bad smells* and are found either by human intuition or techniques that can be automated (e.g. metrics). A proven technique to systematically restructure such problematic code pieces is *refactoring*. Refactorings can be applied either in a disciplined way by hand or through tools. The success of refactoring is tremendous even though it is still lacking an established way to prove its behavior preservation. Refactoring automation through mature tools guarantee reproducible transformations even though they may not be formally proven. In addition, they save time and reduce errors due to the automated restructuring.

Based on the already existing work for improving structure of TTCN-2 and TTCN-3 test suites and the TTCN-3 experience gathered while working on this thesis, a refactoring catalog has been developed. In its first part, 72 well known refactorings have been investigated for their applicability to TTCN-3. As a result, 28 refactorings are essentially language independent and can be either directly applied to TTCN-3 or they must be slightly reinterpreted. In the second part of the catalog, 21 new refactorings specifically designed for TTCN-3 are presented in detail. Each refactoring is presented with a name, a motivation, mechanics and an example.

Finally, the TRex implementation is presented. Based on the Eclipse Platform, an infrastructure for the implementation of TTCN-3 refactorings has been developed. For the infrastructure, a new symbol table (including complete import support) and a pretty printer for TTCN-3 have been written. On top of this infrastructure, two refactorings have been implemented: *Rename* and *Inline Template*. To improve the usage experience of TRex and ease the refactoring development, several other functionalities have been realized on base of this infrastructure. For the TRex usage, the content assist, text hover, find references and open declaration functionalities and for the refactoring development, a syntax tree view have been implemented. A build system based on *PDE Build* and ANT has been developed allowing regular automated and reproducible builds of TRex. Furthermore, unit tests for the verification of the pretty printer and the implemented refactorings have been written.

Outlook

The TTCN-3 refactorings presented are informal and based on experience. A formal proof would be desirable, but there is currently no widely accepted method to realize this although several recent approaches seem reasonable. Therefore, this is still a research topic in itself and independent from TTCN-3. The alternative to a formal proof is the development and usage of a bisimulation tool for TTCN-3 which could prove the preservation of behavior through execution before and after the refactoring.

The bad smells presented in [35] are partially language independent. However, the language independent smells that apply to TTCN-3 still have to be identified in detail and it is probable that TTCN-3 will have a set of specific smells due to its unique concepts.

Refactoring tools today are mostly semi-automated. The programmer still has to detect the bad smells by hand. However, finding quality problems in code is time consuming and hard to do for programmers working on the code for a long time. Therefore, the automation of bad smell detection is desirable. This can be achieved through the development and implementation of metrics specific for TTCN-3 for example.

The suggested refactorings (chapter 4) should also be evaluated for their practicability in a larger case study. This could be, for example, the refactoring of a larger standardized test suite where the maintainability is evaluated before and after refactoring. Automatic code smell detection would certainly support this evaluation. However, given the already existing industry interest in the results of this thesis due to the general deficiency of work concerning the refactoring of test specifications, there is little doubt that refactoring of TTCN-3 and the problematic maintainability of TTCN-3 test suites is an existing and important topic.

Finally, the implemented refactorings in TRex are currently rather few. However, the overall implementation of the infrastructure was the demanding and more challenging part in the TRex development due to the complexity of the TTCN-3 grammar and the manifold concepts which are part of TTCN-3. Nevertheless, based on the result of this thesis, the implementation of more refactorings in TRex is certainly worthwhile.

Acknowledgments

This thesis would not have been possible without the expertise, guidance and encouragement of Dr. Helmut Neukirchen. His vast knowledge and attention for important details contributed a lot to the experience and i would like to thank him very much for the tremendous efforts he put into this project. I am positive I have learned a lot during this time. I would also like to thank the other members of my research group, Prof. Dr. Jens Grabowski, Edith Werner and Wafi Dahman for proof reading papers, for listening to my presentations, giving me important advice when needed and supporting my plans for the upcoming time. Dr. Daniel Zeiß and Remko Ricanek must be thanked for proof reading my thesis and finally but most importantly, I would like to thank my parents for their dedication and unconditional support.

Abbreviations and Acronyms

ANTLR	Another Tool for Language Recognition
AST	Abstract Syntax Tree
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
ETSI	European Telecommunications Standard Institute
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
ISO	International Organization for Standardization
JDT	Java Development Tools
LALR	Look-Ahead LR
LTK	Eclipse Language Toolkit
MTC	Main Test Component
PDE	Plug-In Development Environment
PTC	Parallel Test Component
RCP	Rich Client Platform
SUT	System Under Test
SWT	Standard Widget Toolkit
TC	Test Component
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
UML	Unified Modeling Language
URL	Uniform Resource Locator
WTP	Web Tools Platform
XP	Extreme Programming

Bibliography

[1] ANTLR. http://www.antlr.org.

[2] Apache ANT. http://ant.apache.org/.

[3] CVS. http://www.nongnu.org/cvs//.

[4] Eclipse Platform API Specification. http://www.eclipse.org/documentation/html/plugins/org.eclipse.platform.doc.isv/doc/reference/api/.

[5] Eclipse Public License - Version 1.0. www.eclipse.org/legal/epl-v10.html.

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

[7] JAVASVN. http://tmate.org/svn/.

[8] JUnit Best Practices. http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html.

[9] Netbeans. http://www.netbeans.org.

[10] OSGi. http://www.osgi.org.

[11] Subversion. http://subversion.tigris.org/.

[12] SVNANT. http://subclipse.tigris.org/.

[13] The BSD License. http://www.opensource.org/licenses/bsd-license.php.

[14] The Lex & Yacc Page. http://dinosaur.compilertools.net.

[15] W3C HTML 4.01 Specification. http://www.w3.org/TR/html401/.

[16] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles and Techniques and Tools*. Addison-Wesley, 1986.

[17] ANSI/INCITS. Information Technology - Programming Languages - Smalltalk. American National Standards Institute / InterNational Committee for Information Technology Standards 319-1998, 1998.

[18] John Arthorne and Crhis Laffra. *Official Eclipse 3.0 FAQs*. The Eclipse Series. Addison Wesley, 2004.

[19] Kent Beck. Make it Run, Make it Right: Design Through Refactoring. *Smalltalk Report*, 6(4):19–24, 1997.

[20] Kent Beck. *Extreme Programming Explained*. Addison Wesley, 2000.

[21] Boris Beizer. *Black-Box Testing*. Wiley, 1995.

[22] Frank Benders, Jan-Willem Haaring, Thijs Janssen, Dennis Meffert, and Alex van Oostenrijk. Compiler Construction: A Practical Approach. January 2003.

[23] Brian W. Kernigham and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[24] William J. Brown, Raphael C. Malveau, III Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA, 1998.

[25] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-Ins*. The Eclipse Series. Addison-Wesley, 2004.

[26] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[27] Márcio Lopes Cornélio. *Refactorings as Formal Refinements*. PhD thesis, Universidade Federal de Pernambuco, Brasil, March 2004.

[28] Thomas 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.

[29] Eclipse Foundation. Eclipse. `http://www.eclipse.org`, 2006.

[30] ETSI European 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.

[31] ETSI. European 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.

[32] ETSI. TS 102 027-3: SIP ATS & PIXIT; Part 3: Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT), 07 2005.

[33] ETSI European 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.

[34] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

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

[36] Martin Fowler. Refactoring Home Page. `http://www.refactoring.com/`, 2005.

[37] Leif Frenzel. Neutral im Sinne der Qualität. *Eclipse Magazin*, 5, 2005.

[38] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns and Plug-Ins*. The Eclipse Series. Addison-Wesley, 2003.

[39] Erich Gamma and Kent Beck. JUnit. `http://junit.sourceforge.net/`, 2006.

[40] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[41] Erich Gamma, Eric Meade, and Kent Beck. JUnit. `http://junit.sourceforge.net/`, 2004.

[42] James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. Java series. Addison-Wesley, second edition, 2000.

[43] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). *Computer Networks, Volume 42, Issue 3*, pages 375–403, June 2003.

[44] 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.

[45] Jetbrains. IntelliJ IDEA. `http://www.jetbrains.com`.

[46] Jochen Kemnade. Development of a Semantics-aware Editor for TTCN-3 as an Eclipse Plug-in. Bachelor's thesis, Georg-August-Universität Göttingen, 09 2005.

[47] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science (LNCS)*. Springer, June 1997.

[48] Pekka Mäki-Asiala. Reuse of TTCN-3 Code. Master's thesis, University of Oulu, Department of Electrical and Information Engineering, Finland, 2004.

[49] Mika Mäntylä. *Bad Smells in Software - a Taxonomy and an Empirical Study*. Master's thesis, Helsinki University of Technology, 2003.

[50] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform*. The Eclipse Series. Addison-Wesley, 2005.

[51] Tom Mens. A Formal Foundation for Object-Oriented Software Evolution. In *Proc. Int. Conf. Software Maintenance*, pages 549–552. IEEE Computer Society Press, 2001.

[52] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Electr. Notes Theor. Comput. Sci*, 82(3), 2003.

[53] Tom Mens and Tom Tourwe. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[54] Gerard Meszaros. Patterns of XUnit Test Automation: Refactorings. `http://tap.testautomationpatterns.com:8080/Refactorings.html`, 2005.

[55] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1980.

[56] Matthew J. Munro. Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *IEEE METRICS*, page 15, 2005.

[57] Naouel Moha and Yann-Gael Gueheneuc. On the Automatic Detection and Correction of Design Defects. In Roel Wuyts Serge Demeyer, Kim Mens and Stéphane Ducasse, editors, *proceedings of the* 6^{th} *ECOOP Workshop on Object-Oriented Reengineering*, July 2005.

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

[59] Helmut Neukirchen. Re-Usability in Testing. Presentation, TAROT Summer School 2005, June 2005.

[60] UML 2.0 Testing Profile Specification (ptc/04-04-02). Object Management Group (OMG), April 2004.

[61] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[62] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science (LNCS)*, pages 167–183. Springer, 1981.

[63] David L. Parnas. Software Aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[64] Jan Philipps and Bernhard Rumpe. Roots of Refactoring. In Kenneth Baclavski and Haim Kilov, editors, *Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15, 2001.* Northeastern University, 2001.

[65] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of ObjectSystems (TAPOS)*, 3(4):253–263, 1997.

[66] Donald B. Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.

[67] Ina Schieferdecker and George Din. A Meta-model for TTCN-3. In Manuel Núñez, Zakaria Maamar, Fernando L. Pelayo, Key Pousttchi, and Fernando Rubio, editors, *Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops The FormEMC, EPEW, ITM, Toledo, Spain, October 1-2, 2004*, volume 3236 of *Lecture Notes in Computer Science (LNCS)*, pages 366–379. Springer, 2004.

[68] Michael Schmitt. *Automatic Test Generation: Practical Procedures for Efficient State Space Exploration and Improved Representation of Test Cases*. PhD thesis, Georg-August-Universität Göttingen, 2003.

[69] Gregor Snelting and Frank Tip. Reengineering Class Hierarchies Using Concept Analysis. Technical Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA, 1997.

[70] Terence Parr. Preserving Original Token Sequence In ASTs. `http://www.antlr.org/article/preserving.token.order/preserving.token.order.tml`.

[71] Testing Technologies TT Tool Series product information. `http://www.testingtech.de/products/TTToolSeries.html`, 2004.

[72] Lance A. Tokuda. *Evolving Object-Oriented Designs with Refactorings*. PhD thesis, University of Texas at Austin, 1999.

[73] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring Test Code. In Michele Marchesi and Giancarlo Succi, editors, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)*, May 2001.

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

[75] Antal Wu-Hen-Chang, Dung Le Viet, Gabor Batori, Roland Gecse, and Gyula Csopaki. High-Level Restructuring of TTCN-3 Test Data. In Jens Grabowski and Brian 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.

[76] Wei Zhao. Entwicklung eines Parsers für TTCN-3 Version 3 unter Verwendung des Parsergenerators ANTLR. Bachelor's thesis, Georg-August-Universität Göttingen, 2005.

All URLs have been verified on March 8, 2006.