Towards an Extensible Architecture for Refactoring Test Code

Rogério Marinke[1], Eduardo Martins Guerra[1], Fábio Fagundes Silveira[2](✉), Rafael Monico Azevedo[1], Wagner Nascimento[1], Rodrigo Simões de Almeida[1], Bruno Rodrigues Demboscki[1], and Tiago Silva da Silva[2]

[1] National Institute for Space Research, São José dos Campos, Brazil
rmarinke@gmail.com, guerraem@gmail.com, rmonico1@gmail.com, nascimentolwtn@gmail.com, rsalmeidafl@gmail.com, demboscki@gmail.com

[2] Federal University of São Paulo – UNIFESP, São José dos Campos, Brazil
{fsilveira,silvadasilva}@unifesp.br

Abstract. As the software evolves, new codes are written, and many other codes are refactored. Refactoring also involves the test code, to ensure that it continues performed, and adequately verifying the behavior of the software. This work proposes the creation of an extensible architecture named EARTC to perform refactoring safely in test code. The coding of a specific refactoring for the test code is hampered by coupling to the unit testing automation framework so that it becomes desirable to eliminate this coupling from the refactoring code. The architecture proposed in this work implements extension points, which allows refactoring to be performed regardless of the test framework to be used, that is, other refactorings can be added to the architecture without the need to change the test code. Additionally, the architecture enables other testing frameworks to be coupled without interfering with their internal structure. To validate the independence among architecture modules, a Design Structure Matrix was done, which shows that the objectives were achieved. Also, to analyze the results of the proposed architecture in an experiment, a tool called Neutrino was implemented. The results obtained with Neutrino are satisfactory and show that the architecture meets the objectives described for the accomplishment of this work.

Keywords: Neutrino · Refactoring · Test driven development

1 Introduction

One alternative to improve software product quality is to implement automated testing. The automated tests are easily reproducible, maintain the characteristic of being systematic, and control the execution of all the tests. Moreover, they store test artifacts, assist in verifying changes in software behavior and the test code itself after the refactorings [16].

According to Gerlec et al. [9], the changes from software maintenance are part of software evolution. However, it is important to emphasize that although

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11622, pp. 456–471, 2019.
https://doi.org/10.1007/978-3-030-24305-0_34

such maintenance and refactoring are unavoidable, the changes may lead to new defects and contribute to the decline of the software architecture initially designed. In this context, it is essential to know the types of changes requested and the impact of these changes in the structural aspects of architecture and quality.

During the activities of software development is natural to apply the code refactoring process. It happens due to constantly changes in environment as well as in the requirements. On the other side, when considering the development of software using Extreme Programming (XP), refactoring plays a fundamental role in the development process because of the interactions between test code and production code.

According to Fowler and Beck [6] and Beck [4], the code changes made in the internal structure of the software are intended to make the software easier to understand and cheaper to modify without altering its observable behavior. Still, according to these authors, there is a relationship between refactoring and Test-Driven Development (TDD) because, without the unit testing and acceptance activities, it is difficult to ensure that the changes were correctly made as expected.

XP is a software development method where the code is not the most important. Other activities as analysis, documentation, and planning became the XP itself relevant for the community of software development [4,24]. According to Williams et al. [24] and Beck [4], the primary objective of XP is to provide a set of good practices for software development. The most known XP techniques are testing-first programming, refactoring all the time, write short interactions and develop the simplest possible solutions [4,24].

Several XP practices involve pair programming, where usually programmers work together in only one requirement at a time. Beyond, programmers create a unit test for expected behavior. In the XP approach, the code is refactored to achieve the result, and the test previously written is executed again, until the functionality has all test passed with success [24].

TDD was proposed as a practice in agile method XP [12], not initially with this name, and according to Janzen and Saiedian [12] the main objectives in TDD are the relationship with design and analysis, not the testing itself. TDD recommend to write the simplest codes before the production code, and from this point are then made the refactor both in test and production code until all specified functionalities are implemented [12,24]. As a result of this approach, the programmer can execute the tests after they are written [24]. TDD is a practice to aid in developing software and to be used with other practices and methodologies such as agile, waterfall, spiral, incremental or evolutionary [12].

Software maintenance and implementation are crucial problems for the companies, mainly for those that use complex systems, considering that these have systems with many functionalities and many code refactoring are necessary, which should be tested to the maximum after each change to avoid including errors in the software. Thus, the study of new technologies and techniques that help in the minimization of software defects is relevant, once the aspects such

as reliability, efforts with maintenance and alterations in the source code may contribute to the organization and allocation of resources, minimization of costs and improvement of the quality and maturity of the software. In this context, it is necessary to improve the researches on the TDD, which have provided different approaches that impact directly on the costs and the quality of projects of development and maintenance of software.

The test code does not have tools for refactoring as the production code has. Although, a refactoring in the test code may improperly change the external behavior of the test. In this direction, this work is an attempt to bridge the gap of automation tools for refactoring of test code, being independent of the test framework adopted in the development project.

The remaining of this paper is structured as follows: Sect. 2 reports the background notions related to refactoring testing. Section 3 describes the *Extensible Architecture for Refactoring Test Code* (EARTC) proposed in this work. Section 4 describes the plugin Neutrino, discusses the results of our research and presents the threats to validity. Finally, Sect. 5 summarizes our findings.

2 Refactoring Code and Tools

2.1 Refactoring Test Code

The primary goal of refactoring the test code is to make it easier to understand, without duplication, and to collaborate with its maintenance [6]. According to Fowler [6], refactoring test code can occur mainly due to two factors: In the first form, refactoring test code occurs naturally as a consequence of the development activity – sometimes this refactoring is also performed to remove duplicates in the test code. The second factor that involves refactoring refers to the inclusion of new functionalities in the production code, so the test code needs to be refactored [6].

It is important to highlight that this approach is to guide the development of functionalities in the simplest possible way and maintains the focus on what needs to be implemented. Another relevant feature of this approach is that whenever there is a need for refactoring in the production code due to a possible change in the scope of functionality, you must first refactor the test code, only to later refactor the production code [6,19]. In addition to that, when refactoring is performed in the production code, the previously implemented test suite will signal if the behavior of the software has changed, helping to maintain the quality [6].

Although refactoring is necessary for software evolution, when codes are refactored, faults can be included during the refactoring. Some of these failures can be avoided or minimized with the detection of code smells. Code smells are symptoms of insufficient implementation, and that may lead to fault proneness [23]. At the same time, the code smells represent a design problem which is also an opportunity to improve the software through the refactoring, highlighting the importance of refactoring the test code [2,6].

According to Van Deursen et al. [23] and Meszaros [15], test code refactorings is different from production code refactoring because there is a distinct set of bad smells involved, and improving test code involves additional test-specific

Table 1. Refactorings techniques for test code [23]

Refactoring	Description
Inline resource	To remove the dependency between a test method and some external resource
Setup external resource	Creates or allocates resources as directories, databases, or files, before testing
Make resource unique	To avoid the use of overlapping resource names use unique identifiers
Reduce data	Minimize the data that is setup in fixtures, it contributes to the documentation, and your tests will be less sensitive to changes
Add assertion explanation	Use message to distinguish between different assertions that occur in the same test
Introduce equality method	Add an implementation for the equals method. Rewrite the tests that use string equality to use this method

refactorings. Moreover, according to these authors, the refactoring test code requires a set of other refactoring techniques that aim to ensure, among other characteristics, the quality and behavior of the test code, as shown in Table 1.

Mens and Tourwe [14] published a survey which described the refactoring activities that are supported and the effect of refactoring on the software process. They concluded the need for formalisms, processes, methods, and tools that address refactoring.

Guerra and Fernandes [10] presented three types of test code refactoring, considering the implementation scope: (i) Refactoring inside a test method – in general, has the source from the need to redefine the test scenario; (ii) Refactoring inside a test class – comes from the need to include/remove tests, fine-tuning the methods of finalizing and initializing the test suite; and (iii) Structural refactoring of test classes – some refactorings that occur in production code related to class structure (e.g., impact on the requirement that the structure of the test codes is also refactored.

2.2 Refactoring Tools

Although some of the works presented in this subsection examine refactoring tools in a wide context, also considering refactoring tools for production code, the selected works are relevant because they highlight some problems also found in refactoring tools for test code.

The use of automated testing suite has become more familiar with the use of tools such as the xUnit framework proposed by Meszaros [15], and this is important to support the TDD. These tests are gradually added to the project during the incremental development and are refactored as the project evolves. In this context, when a production code is refactored, the test suite is performed

again to ensure that the behavior has not changed. However, when refactoring is performed on the test code, we do not have a test suite for the tests themselves to ensure that the behavior has not changed. Considering this context, there is a need for refactoring test code tools that provide and verify whether the external behavior of the test has been maintained.

In the work presented by Murphy-Hill et al. [17], refactoring tools usage is observed and some forms of use and features of the tools are evaluated. The experimental method takes data from eight different sources, and their findings also suggest that refactoring tools need further improvements before programmers can use them frequently. The authors concluded that programmers do refactor frequently, but when committing code to version control systems, developers' messages appear not to indicate refactoring activity reliably.

According to Bavota et al. [3], refactoring activities induce faults, mainly as refactorings involving hierarchies. They carried out an empirical study on three Java software systems and automatically detected 15,008 refactoring operations. The authors suggest more accurate code inspection or testing activities when such specific refactorings are performed.

An empirical study was also presented by Gatrell et al. [8]. The work addresses two perspectives: the nature of classes to which refactorings have been applied and the overlap and correspondence between refactorings applied to each class type. In the study, it was used a bespoke tool for extracting a set of fifteen refactorings. They concluded that production and test classes evolve almost independently regarding refactoring.

Passier et al. [18] addressed the problem of breaking the link between the production code and the test code after a refactoring. Moreover, they proposed a tool for Eclipse IDE that maintains tracking of the modifications made to the production code and analyzes the impact of these modifications on the test suite. This way, it provides tips to the developer on how to update the test suite codes. However, the tool proposed by the authors, considering the project specifications, the refactoring track, and the analyzes, is performed after refactoring the production code. This is a different approach as suggested by TDD since in TDD the specification is used at first to write a test in the simplest possible way.

According to Silva et al. [22], when refactoring a production code we must pay attention to avoid inserting unwanted behavior into the software. One way to ensure that the behavior has not changed is the use of automatically generated regression test suites. Moreover, the authors addressed this problem by evaluating the tests cases generated automatically by the Randoop and EvoSuite tools. Authors concluded that the tools generated many obsolete test cases and that the automatically generated tests fail in more than 50% of all injected faults.

Kaur and Singh [13] divided the survey in code refactoring study and described an algorithm to improve the refactoring process. They concluded that the effectiveness of work by comparing the different code maintainability indexes generated by the tool.

In the work carried out by Rizzi et al. [20], the authors investigated the problem of the Cyclic Dependency and proposed a tool to detect smells at the

software architectural level. Furthermore, the tool is able to identify and suggest a possible refactoring approach to resolve Cyclic Dependency.

Bladel and Demeyer [5] introduced a tool to support the analysis of whether refactoring in test code does not change the behavior of the test. The tool uses a Test Behavior Tree to represents the behavior of the test code. Their approach considers to be stored the formula, variables, values, and objects detected in the test code. A similar technique is used in some compilers. The tree is constructed for both test codes, before and after refactoring. Although the tool can detect the change of behavior of the test suite, the tool is performed out of the development IDE, making their use more complicated by development teams.

3 The Extensible Architecture for Refactoring Test Code

Coding unit tests can provide higher quality and helps to maintain the production code functionality. However, a refactoring applied to a test code should have known about the framework for which this test code was written. This behavior occurs because there is information in the test code about the test framework, how tests should be implemented, suites, assertions, and how methods are implemented.

This coupling and dependency between the test framework and test code and, consequently, the refactorings intended for the test code, creates a difficulty in the creation of these refactorings, since the test frameworks, such as JUnit, require different implementations depending on the version used. Besides, it is a challenge to implement a general tool that provides specific refactorings for the test code independently of the framework being used. Another difficulty is the implementation and use of specific refactorings for test code, which may involve the detection of bad smells.

In this context, this work proposes the creation of an extensible architecture for refactoring test code in order to uncouple the unit testing framework from IDE and development project. It is contemplated a process and an abstract structure of code which represents tests. The process manipulates this structure, and both are decoupled from the test code framework. This way, it allows test refactorings to be performed independently of the test framework, and those new frameworks like JUnit 5 and TestNG can be added to make use of the existing refactorings in the EARTC. Besides, the EARTC allows that new refactoring methods can be added without any explicit modification in the architecture.

3.1 Overview

The EARTC is considered an intermediate structure and abstract syntax tree, aiming to represent the elements of the unit tests. This structure is called the Abstract Test Code Representation (ATCR) and was based on the model proposed by Guerra [11]. ATCR performs the parser of the test code for the class structure that represents the elements and operations of a test suite. This activity is performed by reading the test code to be refactored and, after that,

the parser is transformed into a structure of classes according to the model detailed in Sect. 3.3. Moreover, ATCR is also responsible for writing the new refactored test code file.

Since the proposal goal is to have an extensible architecture in the EARTC, which allows another test frameworks to be added, the ATCR has an abstract interface. That is, ATCR has a part where the operations that can be performed on the elements of a test are abstracted. However, the most important architectural feature is its decoupling of any unit testing framework. Thus, its concrete implementations have details on how these operations will affect each test framework.

It is important to emphasize that refactoring is conceptual, that is, the definition of refactoring does not depend on the programming language to which it was implemented, but only on the elements that are manipulated by it. Therefore, the EARTC may be used as a basis for the construction of others refactoring of the same type, including the use of other programming languages besides Java.

3.2 Functionalities

Guerra and Fernandes [10] developed a specific test code catalog for refactoring. This work uses such a catalog as a basis for deriving the functionalities that make up the EARTC. A list of the architecture functionalities are displayed in Table 2.

In addition to the architectural functionalities described above, the EARTC also are able to detect bad smells [23] according to the catalog proposed by Fowler [6] and adapted by Guerra [11]. The list of test bad smells detected by the EARTC are listed below:

- Eager test - when many behaviors are verified in a single test;
- Assertion Roullete - same behavior is verified, and for all verifications, the expected result is the same. When the test fails it is difficult to detect which exact line of assertion has failed;
- Assertion without explanation - lack of messages that explain to the programmer the reason for the error;
- Composite Assertion - more than one assertion in a single line of code, usually joined by the boolean operators AND and OR;
- Many assertions in the sequence - induce duplication of code and generally do not clearly express the intent of the test;
- Duplication of code at startup or termination - testing methods have the same initialization or termination code; and
- Branching in code - insertion of conditional codes to verify states of other objects. The problem could be solved by using the initialization or finalization methods.

It is important to highlight that the EARTC can perform refactorings in test code written for the Unit3 and Unit4 testing frameworks. Furthermore, as mentioned in the Sect. 3.1, the EARTC is extensible, and any other testing automation framework can be added.

3.3 Internal Architectural

ATCR defines the elements in the test code so that its manipulation by the refactoring implementation code can be done in a decoupled way from the unit testing framework for which the test code was written. The ATCR implementation was made considering the test definitions proposed by Guerra [11]. Thus, the ATCR in its internal organization has as a higher level element: the battery of tests. It contains the list of test suites that is composed of one more test classes. Below the test suite is the test method, which is responsible for running the tests of a functionality individually. A test suite must have at least one test method, i.e., empty test suites are not allowed. Below the test method is the test element, and this element represents a line of code belonging to a testing method. The test element can exist in two forms: actions and assertions. An action is a line of code where a test is prepared or exercised; this way, it is not a test properly, but an action necessary for the execution of it – can be methods or functions. The assertion, in turn, is the verification that the result obtained with action are addressed with the expected behavior. During the verification process

Table 2. Architecture functionalities of the EARTC (adapted from [10])

Refactoring	Description
Add assertion explanation	It adds a string parameter in the assertion method with its explanation
Create template test	It creates a template test to abstract functions like the template method design pattern [7]
Decompose assertion	It substitutes one composite assertion for similar ones. Refactoring inside a test class o Add Fixture - It creates a fixture and uses it as a test target
Extract setup method	It creates a setup() method and moves initialization code to it
Extract tear down method	It creates a teardown() method and moves finalization code to it
Introduce equality method	It substitutes many comparisons in the same object with a comparison method [23]
Join incremental tests	It joins tests that are created incrementally
Join similar tests with different data	It joins all similar tests that use different data in only one. Structural refactoring of test classes
Mirror hierarchy to tests	It mirrors to test classes the same hierarchy of the application classes
Pull down test	It moves a test method down in the hierarchy
Pull up test	It moves a test method up in the hierarchy
Simplify use setting	It substitutes the actions to simpler ones with the same effect on the test target
Split action from assertion	It substitutes an assertion that makes modifications on one action and one assertion
Split test from aggregated class	It separates the verification of the responsibility of an aggregated class into another unit test

(a set of assertions), in this context, an object called fixture will have some of its properties checked against an expected value [11].

In order to reach these requirements, we decided to divide the EARTC into four components:

- Preliminary Test Code Parser (PTCP);
- Preliminary Test Code Representation (PTCR);
- Test Code Parser (TCP); and
- Abstract Test Code Representation (ATCR).

The aiming of PTCP is to provide the source code received for the parsing in a way that is easy to be used by the other EARTC components. For this purpose, the PTCP uses an External Source Code Parser (ESCP) that deals directly with the source code. Thus, the PTCP ends up functioning as an intermediate layer that separates the complexity of the ESCP from the other components of the EARTC.

At the end of the parsing performed by the PTCP, the PTCR is generated, which is where the elements of the source code are represented for their later separation in the elements of the test code. After the encapsulation of the source code elements in the PTCR, it is then used by the TCP to separate the test code elements, which forms the ATCR. Finally, ATCR is then available for use by refactoring, thereby decoupling the code from those of the testing framework for which the test code was written. Figure 1 illustrated the architecture of the EARTC proposed in this paper.

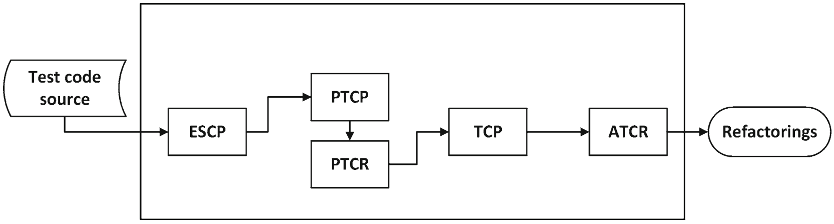

Fig. 1. Architecture of the EARTC proposed in this work.

The parsing process is started with the source code by the PTCP. This component sends the source code for the ESCP, where it will be separated into objects coupled to a specific ESCP. Once the ESCP, its elements, and especially the complexity of its specific functionalities are encapsulated in the PTCP, the resulting PTCR will be available for use in the next step of the process, the TCP. The work of TCP is to separate the components found in the first parsing, in the form of PTCR, in actions and assertions, which form the basis of the ATCR.

It is important to notice that all four components of the EARTC have implementations with abstract and concrete classes and are decoupled from the specific implementations of each component, which allows the architecture to be extended and used by other proposals individually.

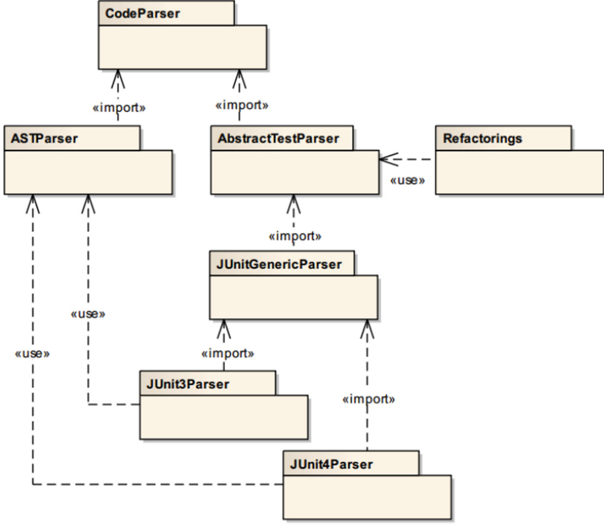

Fig. 2. Parsing classes structure.

3.4 Extension Points

The PTCP module is responsible for the creation of the tree structure form the test source code. However, since the EARTC should be extensible for other test frameworks, PTCP was designed containing abstract classes that should have a reference implementation for the testing framework to be used. In this way, the EARTC contains a package called ASTParser, which makes use of the AST framework contained in the PTCP. Thus, the ASTParser contains the concrete implementations of AST. AST is an API for transforming test source code into a tree structure of AST nodes. These nodes are subclasses of ASTNode, and each subclass is specialized in a Java language element, such as methods, variables, assignments, and so on [1].

Therefore, to allow other testing automation frameworks to be added to the EARTC, the ASTParser module must be extended. Figure 2 depicts the classes structure for parsing.

4 Neutrino: A Tool for Test Code Refactoring

In order to provide a proof of concept for the EARTC, we developed a plugin for the Eclipse IDE, called Neutrino[1]. This plugin is a non-intrusive tool, which means to use Neutrino it is not necessary to modify previously the annotation inserted in the test code, nor performing inheritance or interface implementations.

[1] https://github.com/nascimentolwtn/Neutrino.

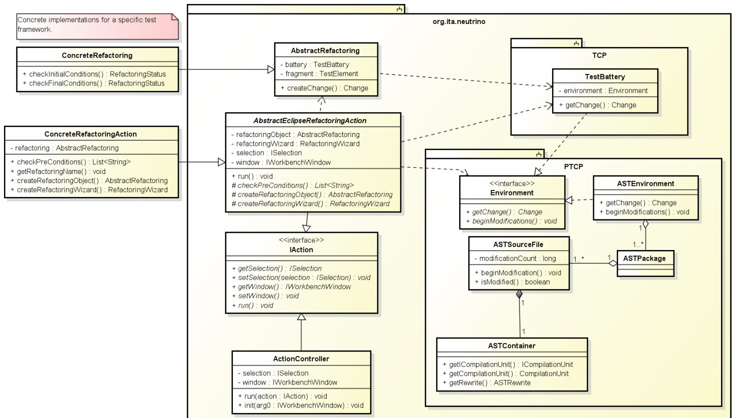

Fig. 3. Neutrino's class diagram.

To refactor the test code, Neutrino executes a set of activities exchanging data between the EARTC modules and the Eclipse IDE. These activities are detailed below. For a better understanding of the activities described here and performed by Neutrino, the class diagram with the main EARTC modules is shown in Fig. 3.

The first activity carried out by Neutrino is to perform a previous check of the IDE conditions, done by the specific container. In this check, it is verified whether the test code refactoring is possible or not. Nonetheless, this check is performed before parsing the test code, which makes this check quite superficial since no element of the test code is available. However, a negative result in this check ends the refactoring job immediately. Thus, this preliminary checking of the IDE conditions is intended to avoid unnecessary parsing execution.

After the first check is completed, the source code parsing must be performed. Since it does not have details related to refactoring, it is executed by the abstract container. After the execution of the PTCP, a PTCR object is then returned. This object will be used as a parameter in the next step, which is the parsing of the test code. In the same way parsing source code, the test code parsing is not dependent on the refactoring that was requested by the user. Therefore, this parsing is executed by the abstract container. This way, there is no coupling with IDE classes, as opposed to parsing the source code. After the test code parsing, an object represented by the ATCR is then generated.

Once the test code has been separated into its component elements, the class responsible for applying the test code refactoring is instantiated. With the instantiated class, it is configured with the ATCR object generated in the test code parsing. Since this task requires knowledge about the requested refactoring, it is executed in the specific refactoring container.

At this moment, the second and final check of the initial conditions is made. This second check of initial conditions is done after parsing the test code so

that the elements identified in this step can be used by this check, which does not occur with the previous one. Thus, the second initial condition check is independent of the test code automation framework.

Before the actual application of the changes to the test code, the final preparation of the refactoring object is still done. This step is used to ask the developer for the parameters needed to apply the refactoring, if necessary. Since the purpose of this step is to interact with the user, it is necessary that the user can cancel the application of the refactoring. This step is carried out by the specific container since it varies according to the refactoring and has a coupling to the development IDE. After requesting possible parameters to the user, the refactoring object is finally ready. At this point, the changes are applied to the ATCR, and the changes propagate from refactoring to the source code.

4.1 The Neutrino Refactoring Process

To use the tool is necessary to open the test class to perform refactoring, and choose *(re) run smell detection* from the menu. In this way, Neutrino will parse the code and display the detected bad smells, as depicted in Fig. 4.

The next step is to perform the refactorings in the test code as suggested by Neutrino. This process is performed automatically by the tool, but the developer must choose one kind of refactoring at a time. Figure 5 shows the execution of the *add explanation to assertion* refactoring, from the moment the developer chooses the line of code with a bad smell to be solved until the refactoring of the test code. Other refactorings have similar behavior for their execution. In most cases should be selected the line of code with bad smell, or even positioning the cursor within the test class and choose the desired option in the Eclipse menu.

Problems | Javadoc | Declaration | Console | Coverage

0 errors, 7 warnings, 0 others

Description	Type	Location	Resource
Warnings (7 items)			
Assertion is missing explanation	Test code smell	line 15	ScoreTests.java
Assertion is missing explanation	Test code smell	line 16	ScoreTests.java
Assertion is missing explanation	Test code smell	line 26	ScoreTests.java
Assertion is missing explanation	Test code smell	line 27	ScoreTests.java
Assertion is missing explanation	Test code smell	line 28	ScoreTests.java
composite assertion	Test code smell	line 16	ScoreTests.java
Repeated initialization code	Test code smell	line 7	ScoreTests.java

Fig. 4. Neutrino bad smells detection.

4.2 Design Structure Matrix

DSM is a strategy for managing dependencies between software modules. Using DSM makes it easy to view the architecture, identify structural problems, and

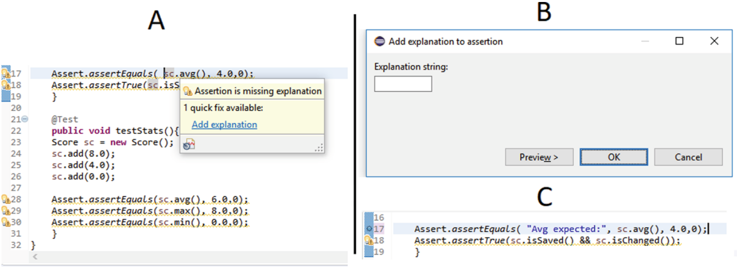

Fig. 5. An example of using Neutrino in eclipse.

check for refactoring results. The DSM is an adjacency matrix with the software modules labeling the horizontal and vertical axes, and indices in the columns and rows to represent the dependency between modules. In the matrix not considered self-dependencies, therefore there are no diagonal markings of the matrix [21].

In the context of this work, to validate the independence between the modules of the EARTC, a DSM was derivated, as described in Figs. 6 and 7.

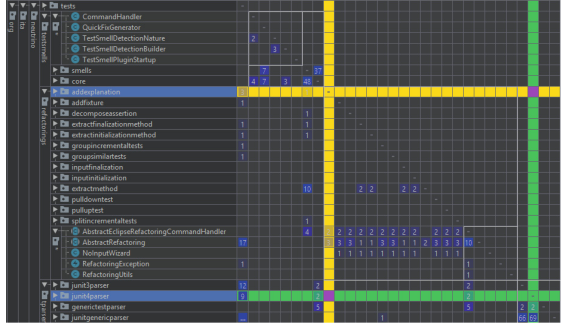

Fig. 6. DSM - Dependency between packages.

The derived DSM guarantees that the refactorings, in the test source code, performed by the EARTC implemented by Neutrino are fully independent, that is, they do not reference the *parser* packages of the test frameworks. If there were a direct reference between the refactoring packages and the *parser* package of the frameworks, refactoring would not work in other test frameworks, as is possible to see in Fig. 6.

The decoupling between test frameworks and EARTC modules can also be observed in the following references:

– there are no references from refactoring methods directly to the test frameworks, such as *junit3parser* and *junit4parser* (Figs. 6 and 7);

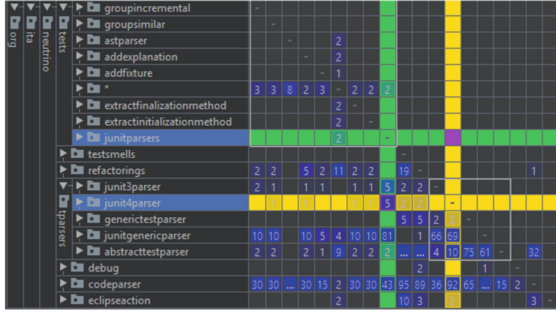

Fig. 7. DSM - Dependency with unit tests.

– Figure 7 indicates that there are 5 references to the *junit4parser* package, which come from the *tests* package. In this package are the parser unit tests. Therefore, this dependence does not influence the decoupling of frameworks.

Thus, these analyses of DSM allow concluding that the modules of the EARTC are wholly decoupled from the test frameworks. This feature makes the EARTC extensible, allowing new test frameworks to be incorporated into the Neutrino framework without the need for architectural change. Besides, as shown in Fig. 6, new refactorings for test code can be added to the EARTC without impacting architectural changes.

5 Conclusion and Future Work

The use of XP methodology has been receiving efforts from the academic community to investigate new methods and techniques to improve software development. In this context, the use of TDD has a relevant role, because it is a technique adopted for many industries with the aim to improve the quality of software products. The use of TDD has a relationship with the refactoring of the test code because the test is written first rather than of production code. However, it is essential to notes the impact of refactoring activity. In other words, it is necessary to check if the behavior of the test code is not changed after the refactoring. It is possible that the behavior of the test code has been changed. Besides, some refactorings might have added bad smells in the source test code impacting to fall of quality of the software.

The use of test automation can shorten software development and testing time by helping to save resources and improve quality. There are many test frameworks used for the industry with the objective the test automation, like JUnit and TestNG. However, these frameworks and other tools available in the literature do not make the detection of bad smells, and also not suggest automatic refactoring in the test code without change the original test behavior.

This work addresses the relevance of refactoring test code during software evolution. Refactoring is a repeated activity in many cycles of software development, and in this context, test codes are also refactored in order to meet changes

or needs for the implementation of new functionalities. As cited in Sect. 3, this work proposed an extensible architecture named EARTC for refactoring test code. The EARTC allows refactorings to be performed in the test source code in order to remove any test problems, such as assertion without explanation, many comparisons in the same object, composite assertions, and problems in the hierarchy of class. The EARTC allows new refactorings to be written without needing to change the internal architecture module. Besides, it allows the use of several frameworks for performing a unit test, such as JUnit. The incorporation of other testing frameworks can also be performed using the original architecture of the EARTC. Besides, this work implements a plugin (Neutrino) for the Eclipse IDE in order to validate the EARTC. To use Neutrino it is not necessary to make changes in the test classes, this non-intrusive feature facilitates the use of the tool. The results obtained by Neutrino were satisfactory and are shown in Sect. 4. Moreover, a DSM was derived from the Neutrino project to verify and prove the independence between the internal modules of the EARTC.

As future work, we plan to address the refactoring of test code and the use of mutation test. In this context, to improve the quality of software in the project, an alternative is to analyze the behavior of a test class before and after the insertion of mutation test classes. These analyses will allow verifying the external behavior of the test code after the refactoring.

Acknowledgments. The authors would like to thank CNPq (grant 455080/2014-3) and FAPESP (grant 2014/16236-6) for financial support.

References

1. Arthorne, J., Laffra, C.: Official Eclipse 3.0 FAQ (Eclipse Series). Addison-Wesley Professional, Boston (2004)
2. Bavota, G., Carluccio, B.D., Lucia, A.D., Penta, M.D., Oliveto, R., Strollo, O.: When does a refactoring induce bugs? An empirical study. In: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, pp. 104–113, September 2012
3. Bavota, G., Qusef, A., Oliveto, R., Lucia, A.D., Binkley, D.: An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 56–65, September 2012
4. Beck, K.: Test-Driven Development: By Example. Addison-Wesley Professional, Boston (2003)
5. Bladel, B.v., Demeyer, S.: Test behaviour detection as a test refactoring safety. In: Proceedings of the 2nd International Workshop on Refactoring, IWoR 2018, pp. 22–25. ACM, New York, NY, USA (2018). https://doi.org/10.1145/3242163.3242168
6. Fowler, M., Beck, K.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (1999)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Boston (1994)

8. Gatrell, M., Counsell, S., Swift, S., Hierons, R.M., Liu, X.: Test and production classes of an industrial c# system: a refactoring and fault perspective. In: 2015 41st Euromicro Conference on Software Engineering and Advanced Applications, pp. 35–38, August 2015
9. Gerlec, Č., Rakić, G., Budimac, Z., Heričko, M.: A programming language independent framework for metrics-based software evolution and analysis. Comput. Sci. Inf. Syst. **9**(3), 1155–1186 (2012)
10. Guerra, E.M., Fernandes, C.T.: Refactoring test code safely. In: International Conference on Software Engineering Advances, p. 44, August 2007
11. Guerra, E.M.: Um Estudo sobre Refatoração de Código de Teste. Ph.D. thesis, Instituto Tecnológico de Aeronáutica, São José dos Campos (2005)
12. Janzen, D., Saiedian, H.: Test-driven development concepts, taxonomy, and future direction. Computer **38**(9), 43–50 (2005). https://doi.org/10.1109/MC.2005.314
13. Kaur, G., Singh, B.: Improving the quality of software by refactoring. In: 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), pp. 185–191, June 2017. https://doi.org/10.1109/ICCONS.2017.8250707
14. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. **30**(2), 126–139 (2004)
15. Meszaros, G.: XUnit Test Patterns: Refactoring Test Code. Pearson Education, London (2007)
16. Mosley, D.J., Posey, B.A.: Just Enough Software Test Automation. Prentice Hall Professional, Upper Saddle River (2002)
17. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2012)
18. Passier, H., Bijlsma, L., Bockisch, C.: Maintaining unit tests during refactoring. In: Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, pp. 18:1–18:6. ACM (2016)
19. Pipka, J.U.: Refactoring in a "test first"-world. In: Proceedings of Third International Conference eXtreme Programming and Flexible Processes in Software Engineering (2002)
20. Rizzi, L., Fontana, F.A., Roveda, R.: Support for architectural smell refactoring. In: Proceedings of the 2nd International Workshop on Refactoring, IWoR 2018, pp. 7–10. ACM, New York, NY, USA (2018)
21. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In: Proceedings of the 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, vol. 40, pp. 167–176 (2005)
22. Silva, I.P.S.C., Alves, E.L.G., Andrade, W.L.: Analyzing automatic test generation tools for refactoring validation. In: Proceedings of the 12th International Workshop on Automation of Software Testing, AST 2017, pp. 38–44. IEEE Press, Piscataway, NJ, USA (2017)
23. Van Deursen, A., Moonen, L., Van Den Bergh, A., Kok, G.: Refactoring test code. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering, pp. 92–95 (2001)
24. Williams, L., Kessler, R.R., Cunningham, W., Jeffries, R.: Strengthening the case for pair programming. IEEE Softw. **17**(4), 19–25 (2000). https://doi.org/10.1109/52.854064