UNIVERSIDAD POLITÉCNICA
DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE –
EUROPEAN MASTER IN SOFTWARE ENGINEERING

Identification and Correction of Besouro's Faults: Impact on Test-Driven Development Experiments

Master Thesis

David Millan Novell

Madrid, July 2015

This thesis is submitted to the ETSI Informáticos at Universidad Politécnica de Madrid in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering.

Master Thesis
Master Universitario en Ingeniería del Software – European Master in Software Engineering
Thesis Title: Identification and Correction of Besouro's Faults: Impact on Test-Driven Development Experiments
Thesis no: EMSE-2015-04
July 2015

Author: David Millan Novell
European Master in Software Engineering
Universidad Politécnica de Madrid

Supervisor:
Name Oscar Dieste
Academic title: Fhd Computing
University of the presented title: UPM (Universidad Politécnica de Madrid)
Department: DLSIIS (Departamento Lenguanjes Sistemas Informáticos en Ingenieria del Software)
School: ETSII (Escuela Técnica Superior Ingenieria Informática)
University: UPM (Universidad Politécnica de Madrid)

ETSI Informáticos
Universidad Politécnica de Madrid
Campus de Montegancedo, s/n
28660 Boadilla del Monte (Madrid)
Spain

Abstract

Context. This thesis is framed in experimental software engineering. More concretely, it addresses the problems arisen when assessing process conformance in test-driven development experiments conducted by UPM's Experimental Software Engineering group. Process conformance was studied using the Eclipse's plug-in tool Besouro. It has been observed that Besouro does not work correctly in some circumstances. It creates doubts about the correction of the existing experimental data which render it useless. ***Aim***. The main objective of this work is the identification and correction of Besouro's faults. A secondary goal is fixing the datasets already obtained in past experiments to the maximum possible extent. This way, existing experimental results could be used with confidence. ***Method***. (1) Testing Besouro using different sequences of events (creation methods, assertions etc..) to identify the underlying faults. (2) Fix the code and (3) fix the datasets using code specially created for this purpose. ***Results***. (1) We confirmed the existence of several fault in Besouro's code that affected to Test-First and Test-Last episode identification. These faults caused the incorrect identification of 20% of episodes. (2) We were able to fix Besouro's code. (3) The correction of existing datasets was possible, subjected to some restrictions (such us the impossibility of tracing code size increase to programming time. ***Conclusion***. The results of past experiments dependent upon Besouro's data could no be trustable. We have the suspicion that more faults remain in Besouro's code, whose identification requires further analysis.

Table of Contents:

1. Introduction

1.1 Problem description

1.1.1 Research area

Nowadays, there is no evidence to support the majority of assumptions related to the creation of software. We do not know exactly the suitability, limits, qualities, costs and risks about the available technologies of software developing. The use of experimentations helps us to contrast these beliefs and opinions against facts.

The objective of making experiments is to identify the reasons why certain results occur. Experiments are done in laboratories (under controlled conditions), activating the main targeted characteristics and/or properties with the aim at understanding better them. Experimentation helps in the identification and understanding of the variables used in software construction and the connections between them. If we perform experiments with software construction we can obtain more comprehension to make better software.

An experiment is a formal, rigorous and controlled research, where variables are manipulated (methodology of develop, techniques of the proves, experience of developers etc..) to know in mode of response variables as for example the efficiency, effectiveness, productivity, order etc.. (depending on what we are investigating in that experiment).

1.1.2 Test-Driven Development (TDD)

Test-Driven Development (TDD) is a method of programming for developers to develop software. Basically, the idea is to make a code based on a test. In this method it is important that developers follow the instructions and make the code based on the tests. The developers could better understand what the code needs to do, test the preconceptions of how the code will work, and improve the overall design of the code iteratively. TDD forces developers to think in smaller bits, which could be testable, and redesign the code after implementing each subtask.

Basically TDD follows the sequence of actions: decompose the specification into small, manageable programming tasks, write a low-level functional test associated with each task before producing code, get instant feedback from tests to decide whether the task has been fully implemented as intended or it interferes with the old code, maintain up-to-date and complete list of tests in place that are frequently run and focus on low-level design and incrementally refactor the production code.

1.1.3 TDD experiments

There are a lot of experiments that use TDD in a lot of program languages but the target of this work is to evaluate the methodology that the developers use to make the code. The use of these experiments is for achieving different objectives, for example it can be to see which is the level of the developers when they make a software, the correctness of the program according to the tests, the time that the developers need for applying some tests, the variability of the developers to make the same code etc... Shull summarizes empirical studies on TDD regarding four main aspects. These aspects are the variants of TDD, the effort spent on TDD, type of study(pilot, industrial use or controlled experiment) and subjects who apply it.[1]

Another type of experiments in TDD, that is important to highlight, were done by Munir who reported a systematic review that identified 41 empirical studies on TDD and analysed them with respect to rigour and relevance score using an assessment rubric. They classified 41 empirical studies with respect to the research methodology reported in these studies. Results on seven case studies and two surveys with high rigor and high relevance scores indicate that the developers perceived an improvement in quality, further supporting the measured external quality improvement. The majority of empirical studies, among which there are 19 studies classified as experiments and one that is classified as a case study, with high rigor and low relevance scores stipulated that there is no difference between TDD and test-last development for a number of variables (productivity, external quality, internal code quality, number of tests, effort/size spent for TDD and developers opinion). [2]

1.1.4 Conformance

We can understand conformance as the degree of agreement between the software process that is really carried out and the process that is believed to be carried out. There are four components to measure process conformance in software development, measurement based on the model, an alternative measurement and guidelines to modify a process. Some authors highlighted the absence of adequate process conformance measurements in software engineering. In particular, the authors are concerned about the obtrusiveness of the existing methods to gauge operational conformity.

Different studies have tackled the problem of quantifying process conformance, specifically for TDD. Through several heuristics, they could identify 22 different events that usually took place during the development flow and categorized them into eight coarse-grained classes. Each sequence of episodes was then marked as TDD compliant or not by a rules-based system.

1.1.5 Instrumentation (Zorro & Besouro)

The process of conformance in TDD is evaluated typically by using tools that analyse the behaviour of the developers. Two of these tools are Zorro and Besouro.

Zorro is a Hackystat-based application that is designed to help development groups gain insight into their use of test driven design practices and the impact of this use on their software development process and products. Zorro works by analysing process data gathered by a Hackystat. Is, an open source framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data. Hackystat is used as a sensor attached to an interactive development environment such as Eclipse. Examples of such process data includes the editing of a test case, the compilation and building of the software, the running of a test case, and so on.

Zorro implements a set of rules that first partitions the stream of observed developer behaviours gathered by the sensor into a sequence of “episodes”, each of which may or may not correspond to a TDD cycle. A second set of rules are then applied to each episode in order to classify it as an instance of TDD or not. A pilot validation study conducted in the Spring of 2006 indicated that Zorro recognized episodes of TDD with almost 90% accuracy.[4]

Besouro is an adaptation of Zorro, an automated TDD behaviour recognizer. The main difference between them, so far, is that Zorro relies on Hackystat and Besouro is a stand-alone Eclipse Plugin. Besouro was initially built to make the Zorro's rule system available for the general public.

1.1.6 Types of behaviour

Zorro and therefore Besouro classify the behaviour in different types. These types are:

- Test-First: First creating a test and next code editing to pass the test.
- Refactoring: Code editing by refactoring and pass a test.
- Test addition: Creation of a test and pass it.
- Regression: Without edit activities pass a test.
- Code Production: Code editing where size is increased and pass the test.
- Test Last: Code editing then test editing and pass the test.
- Long: Behaviour with many activities and then test pass.
- Unknown: None of the above.

1.2 Problem description

1.2.1 ACME² experiment

ACME^2 carried out an experiment to evaluate the difference between the results obtained by ITL (Iterative Test Last) versus the results obtained by TDD (Test-Driven Development). To perform this study, ACME^2 carried out the experiment split in two different days: the first day was on the 8th of April of 2014, and the second was on the 9th of April of 2014. The experiment consisted in making two different softwares, one per day. During the first day, developers were asked to develop the software by using ITL while on the second day they were asked to develop the same software by using TDD technique. This experiment was done with Besouro tool. In our work, we collected 18 datasets from this experiment in order to analyse them.

1.2.2 Besouro wrong behaviour

During ITL (Iterative Test Last), the developers should make the tests after having coded the functionality. But in the analysis data of the datasets that are given by Besouro we found first strange behaviours in these results, when we tried to make a test-last.

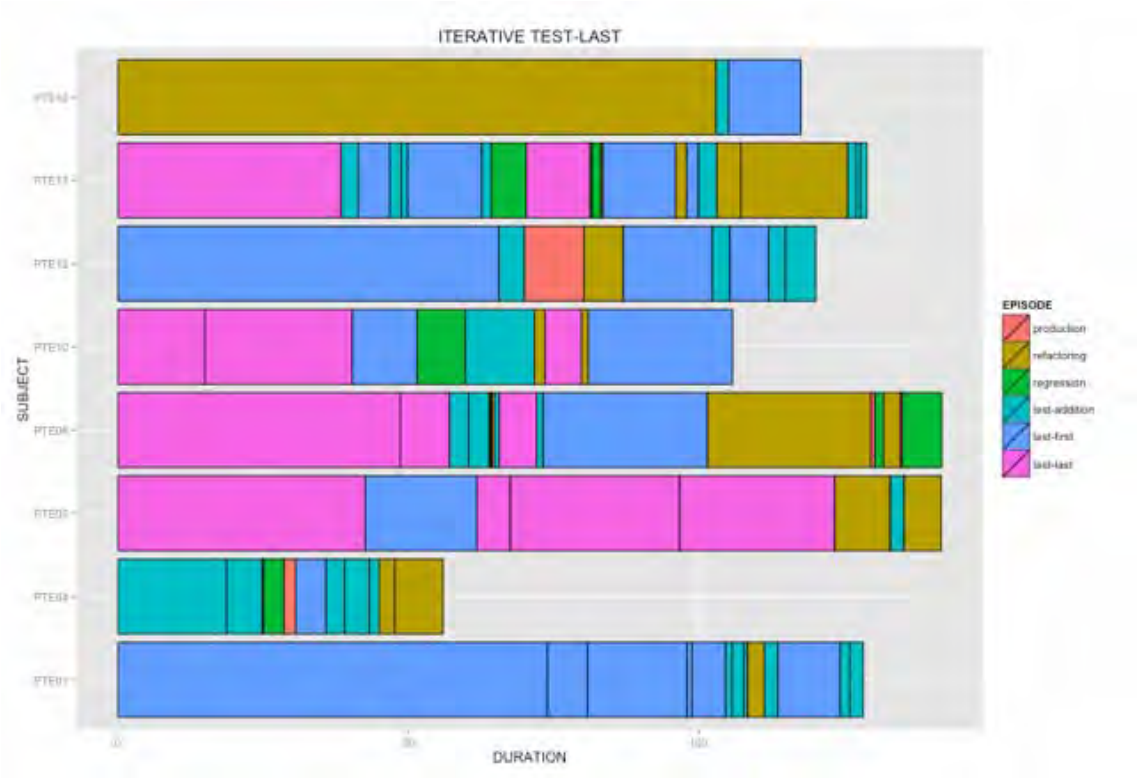

Figure 1: Results Besouro Iterative Test-Last

We were surprised when we tested Besouro over a code in the method of ITL, since Besouro returned the classification of the episodes in TDD. This made us realize that Besouro classified the episodes incorrectly.

1.3 Objectives

After presenting the previous sections the scenario where this work is developed, the main objectives of this thesis can be defined as focused on:

- Faithfully assess whether the Besouro classifies the wrong behaviour of developers.
- Identify the origin of the incorrect identification of episodes.
- Fix the errors of Besouro to make it a good tool for analysing the developers behaviours working with TDD method.
- Fix the existing datasets of the ACME[2] experiment to obtain results more accurate than the ones we have now.

- Evaluate the impact of Besouro's incorrect classification, as to have an idea of the seventy behaviours faults.

1.4 Relevance of the problem

Besouro can be a good tool in Eclipse to verify if the methodology that developers use to generate a code is a TDD method or not. The tool is based on the actions that the developer have done and the classification of the episodes. If we have a correct order of the actions and a correct meaning of the episodes we can evaluate if one project of developing software is made using TDD or not and how it is done. And this is necessary in the experiments of ACME[2] to ensure that it ITL is evaluated versus TDD. On the contrary, we can commit errors in our investigations.

There are a lot of previous experiments on TDD and there are some experiments using the Besouro tool. So, the relevance of the problem is important in this thesis. We not only want to fix Besouro. We want to make a program able to solve these wrong results given by Besouro. We need to fix the errors given by Besouro because the experiments were done, the actions were classified and the classification is incorrect. We cannot repeat the experiments with Besouro fixed. We have take advantage of the datasets previously done.

1.5 Thesis outline

This section presents how this document is structured in order to understand the different chapters of this document. The rest of the document is organized according to the following chapter:

- Background: Consists in all the theoretical concepts needed to jusify the importance of this work. These theoretical concepts will help to understand the next steps that this document have. Basically, the concepts presented in this section are TDD (method to develop software), Zorro (how we obtain the actions of the developers and how to analyse them), Besouro (plug-in of Eclipse with which we work in that

thesis), ACME^2 experiment (datasets that we obtained in one experiment that used Besouro) and Jess (class of Java to evaluate rules in logical programming).

- Research question: This part is intended to clarify what are the most important parts of this work and gives a description of each one explaining why it is important in our work the research that we have done.

- Methodology: That is an explanation of all the steps that we followed to make that thesis. This part is important to know what steps we followed to arrive at our solution. This part can be read first of all to understand all the process of this thesis.

- Execution: This part illustrates with every possible detail what we have done step by step. In particular, we talk about our solution in Besouro and the datasets of ACME^2.

- Related works: This part pretends to evaluate the impact of our work.

- Conclusions: Final conclusions that we have obtained during this thesis are presented.

2. Background

2.1 Test-Driven Development (TDD)

Test-Driven Development (TDD) is a type of software development practice. Basically, it consists in two rules:

1- Write a new code only if an automated test has failed.

2- Eliminate duplication (Refactor).

The first principle is fundamental for the TDD method because this principle introduces technique where a developer first writes a test and then implements the code. Another important consequence of this rule is that test development is

driving implementation. The second principle, which is called Refactoring, implies the improving of the design of existing code.

We can see these principles like a traffic light (see figure 2), first red that means a test that do not pass, second green that means that the code has been modified to pass the test, and finally yellow (in the figure black) when the developers refactors the code.

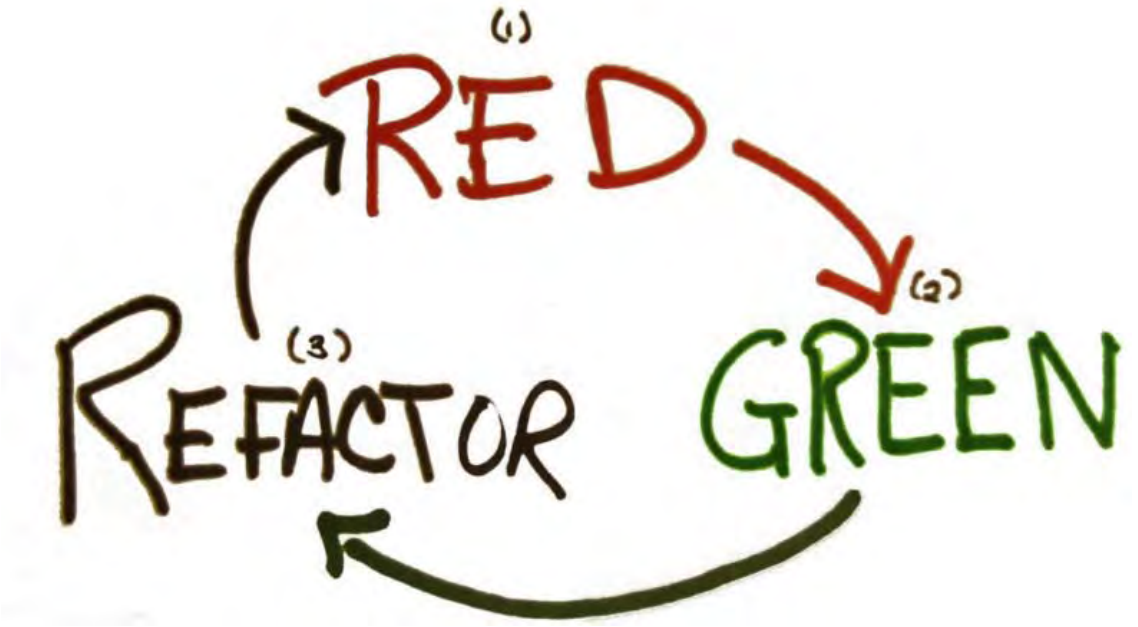

Figure 2: Steps of TDD

There is no need to write production code if there are no broken test cases. It is easy to see that if the test cases are correct for a TDD, if you need a lot of time to make the code, then the TDD is not correct, the developer had to divide the programming task into smaller subtasks that could be solved in a shorter period of time. When the developer passes the test then have to make the refactoring step, consisting in refactoring the code to arrive to create the simplest possible design. In TDD, all the developers have to execute all the tests at the end of each iteration. The revision should be easy because each iteration has to consist in a small quantity of code. We can understand much better this sequence, by looking at the next figure[17]:

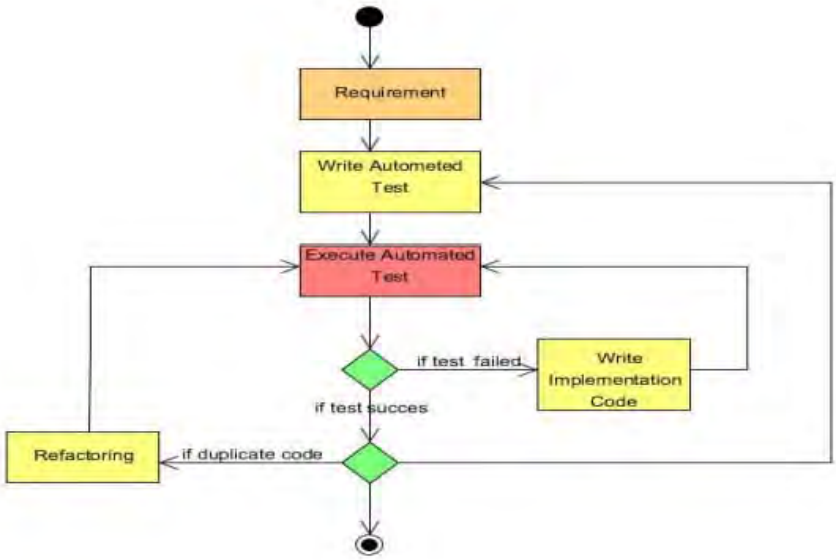

Figure 3: Test-Driven Development diagram.

The figure 3 shows a diagram to explain the steps followed by TDD. First, a test in based on the requirements that the software needs is created. Then, this step is followed by the execution of the tests and the writing of the implementation code to pass the test and re-execute the test to pass it. Finally, if there are duplications of the code, refactoring of the code production is done.

Nowadays, the method TDD is not well used at the time of creating some software. A lot of companies decide not to use this method since they prefer to create the code following other methods more traditional to develop software. The main problems of using TDD to create software are [5]:

- User interface: TDD is difficult to be implemented in the user interface layer (presentation), because this activity contains elements containing lengthen the test cycle and development. Instead, TDD fits easily on developments in the business logic layer (domain objects) and data access.

- Database: One of the main problems of using TDD applications with databases, is that once a test has been run, the database may be in a different state than the one needed to make the next test.

- Errors unidentified: Passing all tests in the tool used (JUNIT for example) does not mean that there are no errors, it just means that while the tests were running no errors were encountered. The use of TDD could lead to

a false sense of security, therefore, we need to focus on where the evidence is detailed and cover all possible scenarios.

- Losing overview: TDD is a bottom-up approach, and you should be aware of that it could lose overall visibility of the project and of the application.

- Steep learning curve: As stated in other deliveries, TDD is hard to be implemented, so a decrease in productivity can be expected during the first two months of implementation.

However the use of TDD can be more advantageous than traditional methods if we look at these advantages[5]:

- Higher quality: It ensures that tests run, preventing, then, applications to fail the first time the user executes them or find errors, instead of being found by the development team. Also, the testing in early stages of development is a way to incorporate quality into the process, resulting in fewer bugs in the final stages of the project.

- Design focused on the needs: The fact of writing test code first, requires that the customer needs are considered at the very beginning, and this fact forces to analyse first what does it really takes to make the code and no other considerations. The advantage of it is a reduction in a later rework.

- Greater simplicity in design: instead of focusing on making large, complex designs, the team will focus on customer needs or requirements, adding only the functionality that the customer needs.

- The design will be adapted to the understanding of the problem: Every time the developers makes iterations to test and program, understanding the problem increases, thus successive iterations have a better understanding, reducing misunderstandings functionality to the end of development.

- Increased productivity: Using a traditional method usually causes at first a high production, however, that productivity falls towards the end of the

project, when developers begin to find errors everywhere, there appear misunderstandings on what the client wanted, or when the client makes a couple of destabilizing changes. With the use of TDD the situation is the opposite because TDD have immediately feedback on the software developed, at first something unproductive, because a series of test cases that failed the first attempt need to be written. However, the benefits are evident when the application is constantly tested, errors have been corrected early and clarified functionality doubts.

- Less time spent on debugging error: The code is being developed by small parts, therefore, when an error occurs, efforts focus on the small piece of code that was changed, so that problems can be identified more directly.

So, it seems that traditional methods can be more productive initially but looking into the advantages about the use of TDD, we can expect that changing to TDD method can be more advantageous than using traditional methods. It can be more complex at initial use of it but in the long term this complexity changes to an easier use than the traditional methods.

2.2 Research on TDD: experiments

There are a lot of experiments based on TDD in literature, to look for different information, for example the viability of TDD, the effectiveness of the developers to solve some problems, how developers learn to use TDD etc.. TDD is a programming method relatively new and we need more experiments to develop this method as well as determine the possible dependence on the needs.

Shull, summarizes empirical studies on TDD regarding four main aspects. These aspects are the variants of TDD, the effort spent on TDD, type of study(pilot, industrial use or controlled experiment) and subjects who apply it. TDD experiments were observed in terms of developers productivity, test quality, code quality and external quality. According to Shull's work eight out of 32 empirical studies are classified as controlled experiments, in which four of them were conducted with graduate students or professionals. With these

empirical studies Shull says that there is a trend of improvement in terms of external quality using TDD, whereas the effects on productivity is inconclusive.[1]

Munir, who reports a systematic review that identifies 41 empirical studies on TDD and analyse them with respect to rigour and relevance score using an assessment rubric. According Munir classified 41 empirical studies with respect to the research methodology reported in these studies. The results indicate that the developers perceived an improvement in quality, further supporting the measured external quality improvement. The majority of empirical studies show that there is no difference between TDD and test-last development for a number of variables (productivity, external quality, internal code quality, number of tests, effort/size spent for TDD and developers opinion).[2]

It was found that both the quality improvement and productivity drop are much larger in industrial studies compared to academic studies. It was also observed that improvements in quality are significantly larger in cases where the difference in test effort using TDD and task size are substantial. 10 out of 37 emprirical studies were classified as industry experiments. 3 out of 10 studies are classified as controlled experiments, while the rest of the studies classified their research as "industrial case studies" which reported quantitative analysis on the effects of TDD.[3]

2.3 ACME² experiment

This company made an experiment to see how the developers develop some software. This experiment consists in creating two softwares where the two softwares follow the same rules. The starting point is a template with some tests and some of these tests, have to be passed. When all the finished tests have been passed, the work has been finished. This experiment is to check if the developers created the software following the rules of TDD or not.

As previously presented, the experiment consists in making two softwares. One of these software is called Bowling Score (BSK), which consists in generating different functions to make a software to be used in a bowling game. The

second software is called Mars Rover which is used in a game consisting in exploring Mars by movements in a grid, trying to simulate the Mars Exploration Rover Mission from the NASA.

MarsRover and Bowling Score Keeper have little in common from a high-level point of view, however structurally and functionally they are homologous. MarsRover's data structure is an NxN matrix of cells. Each cell may store an obstacle. Obstacles have not behaviour and can be modelled with simple data types (e.g. Boolean). There are six main operations to be implemented. In all cases, the matrix (or, more precisely speaking, the class that embodies it) is responsible for execution. The task can be thus solved by using just one.

The goal of the experiment is to compare TDD and ITL effectiveness with respect to the quantification of the work done, external quality and productivity of developers. There are two factors in the study: the development approach and the task. Both factors have three levels (or treatments). The development approach can be: TDD and ITL. The task can be: MarsRover and BSK

There are three response variables in the study, namely the amount of work done to implement the required functionality (measured as the number of tackled user stories), the external quality (measured as conformance to requirements by means of acceptance tests), and the productivity (measured as the amount of work successfully delivered).

The aim of this experiment is to evaluate the impact of Test-Driven Development (TDD) on internal code quality, external code quality and productivity compared to Test-Last Development (TLD). Test-Last Development is another method of developing software more similar than to traditional methods where the code is done first and the test is applied last. The results of this experiment reveals that TLD give better results than TDD but this is one experiment and the majority of developers in the experiment prefer TLD over TDD. Is it possible that the developers have more experience using TLD than TDD? there are other experiences that show that the TDD have better results than TLD. For this reason, it is important to carry out more experiments and develop tools to help in making more reliable results like Besouro offers.

2.4 Conformance

Conformance is a key concept in the present work. In this section, a review of the state of the art on conformance is presented. Some of the relevant works on the field are commented.

Sørumgard defined process conformance in the context of software development, informally as “the degree of agreement between the software development process that is really carried out and the process that is believed to be carried out” [6]. The author aimed to uncover the importance of process conformance in the software engineering field. The study helped to uncover four key components to measure process conformance in software development: definition of the model, measurement based on the model, an alternative measurement and guidelines to modify a process [6]. On the other hand, Cook and Jonathan [7], provided deeper insights into the relationship between a formal model and its execution for high level processes in software development. In particular, they realized that if the model and executions diverge, something crucial is occurring. They proposed numerous metrics to aid the validation of a process, along with the type of data to be collected or ignored. These metrics were demonstrated by using the Test Unit task from [8]. Silva and Travassos [9] highlighted the absence of adequate process conformance measurements in software engineering. In particular, the authors are concerned about the obtrusiveness of the existing methods to gauge operational conformity. They proposed and validated a tool in industrial settings to support the adherence to the prospective-based reading for requirements elicitation. The use of a tool to ensure the subjects to conform to the process resulted in a significant reduction in the time that was needed for elicitation [9]. Notably, Silva and Travassos reflected on the importance of measuring process conformance to have a more adequate understanding of the process itself. Zazworka et al.[10] developed a new approach to identify non-conformities in the life cycle of software development. They defined a template for the completion of a process and searched for patterns of non-conformity in the collected data. Furthermore, they used data visualization techniques to present

and evaluate the risk that is associated with non-conformity behaviours. The authors claim that, with this approach, they can identify deviations from the model from the start and throughout the whole process [10]. The method that Zazworka et al. presented was subsequently tested to gauge the process conformance during an XP course, and to identify process conformance violations as a principal obstruction in instructing developers about new processes [11].

Different studies have tackled the problem of quantifying process conformance, specifically for TDD. In particular, Wang and Erdogmus [12] measured the process with the goal of improving it by analysing data that was extracted from the developers' IDE at a micro-level. They developed a tool that mines Hackystat 1 data looking for patterns that are typical of the TDD cycle (i.e.; lines of code to lines of test ratio, testing effort versus production effort and time-stamp). Madeyski and Szala [13] conducted an experiment on TDD effectiveness. To improve the internal validity of their study, they measured process conformance. Their approach relied on the duration of development events that were subsequently categorized into passive and active. They could infer TDD process conformance from the passive-to-active time ratio, concluding that TDD subjects attained higher active time than test-last developers. In their studies, Johnson and Kou [14, 15], used low level development episodes to calculate process conformance. Through several heuristics, they could identify 22 different events that usually took place during the development flow (e.g.; adding a new class, compiling, and executing the test suite), and categorized them into eight coarse-grained classes. Each sequence of episodes was then marked as TDD compliant or not by a rules-based system. They implemented a tool, Zorro, and validated it by comparing the results of automated analysis to the manual analysis of a video recording of the developers' activity. Zorro classified correctly 89% of the development episodes during the first run of the experiment in university settings and 85% during the second run in industrial settings.

As reviewed in this section, the existing body of knowledge on conformance is focused on the construction of tools and frameworks to gauge TDD process conformance, nevertheless it is never used to quantify its relationship with the

effects of applying TDD. This fact is one of the motivation points of the present work.

2.5 Conformance measurement: Zorro & Besouro

Zorro is a system created to identify automatically if a developer is complying an operational definition of Test-Driven Development(TDD) practice or not. This procedure can benefit the software development community in different aspects (investigation of TDD uses, make a pedagogical form of TDD, accuracy of efficiency of that TDD etc...).

2.5.1 Zorro

To better understand Zorro, let's see the architectural components of Zorro. The Zorro architecture consists of three subsystems, as we can see in the next figure and explained next[16]:

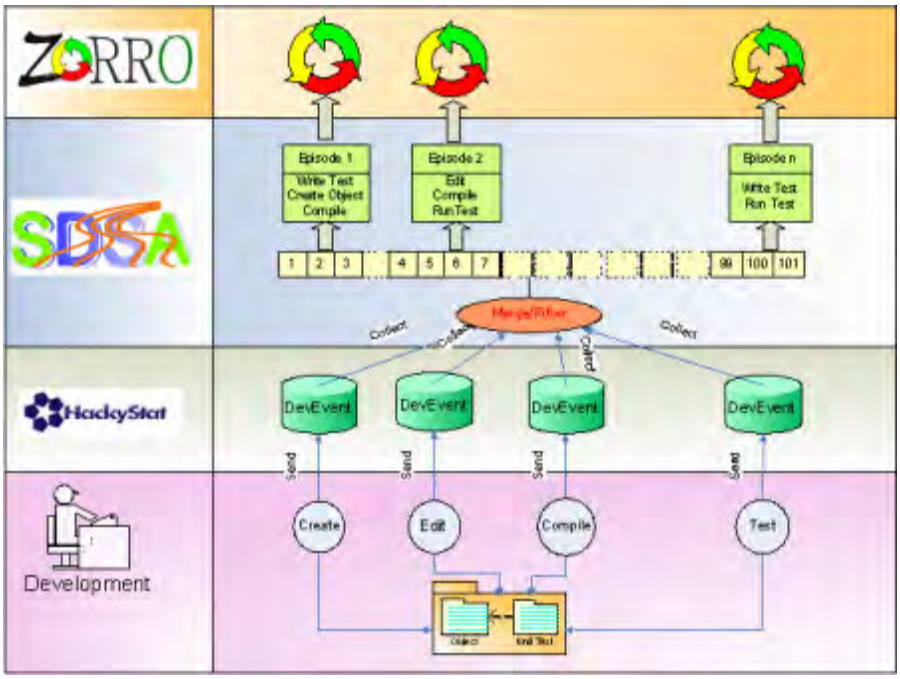

Figure 4: The Zorro Architecture

- Hackystat: Hackystat is an open source for automated collection and analysis of software engineering process and product data. Hackystat collects the data via "sensors" that are attached to development

environment tools. Hackystat can collect this type of events: unit test invocation and their results, compilation events and their results, refactoring events and edition.

- SDSA: Software Development Stream Analysis (SDSA) is a Hackystatbased application, that organizes and analyses the different kinds of data received by Hackystat. This input that Hackystat has done, is an input to a rule-based, time-series analysis. SDSA receives the same information received by Hackystat into a single sequence called "development stream".

Then SDSA creates a sequence at higher level called "episodes" and in this stage makes the elaboration of the episodes that the developer had done during development. This part is called *tokenizing.* The development stream of the sensors in Hackystatbased is returned by tokenizing which returns all the actions that the developer had done, as for example a unit test invocation, a file compilation, a configuration management commit, or refactoring operation.

Other kinds of events considered are the limits of the episode. These events are the final point of an episode to generate the next episode. Examples of this are a configuration management system check-in, test pass event, or a buffer transition.

Once we have obtained a sequence of episodes the next step in SDSA is to classify each episode in a process. SDSA uses an interface to the JESS rule and this allows obtaining a set of episodes to specify all or part of the classification process.

- Zorro: The layer of Zorro analyses in order to recognize whether the behaviours of the developer are a Test-Driven Development (TDD) or not. Also, it classifies if one episode is TDD conformance or not.

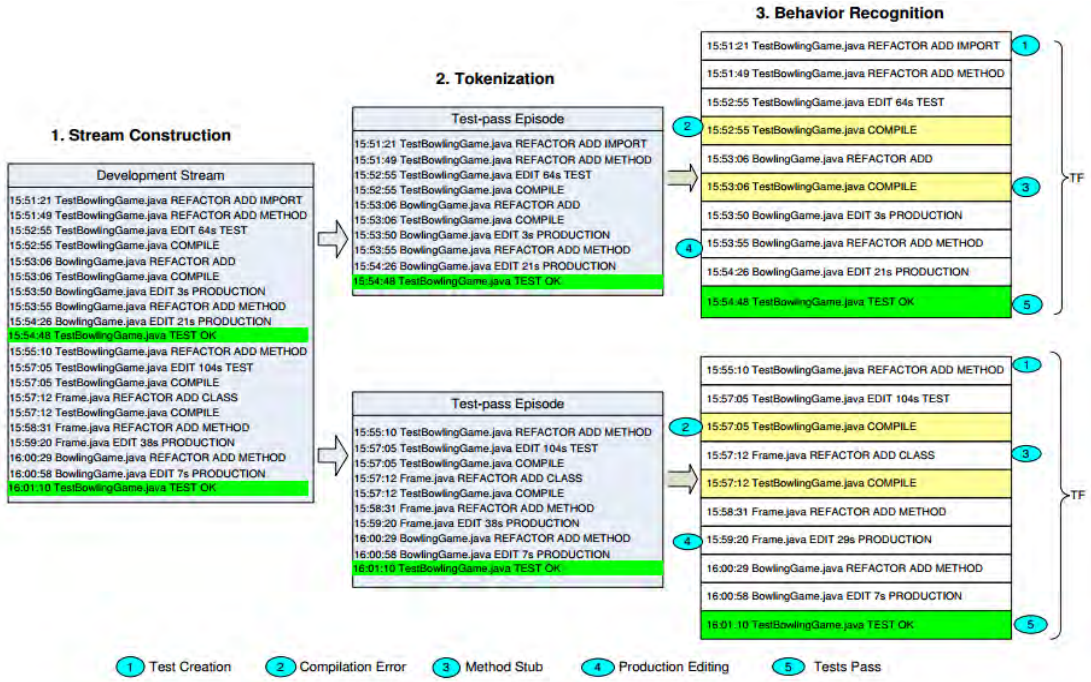

Figure 5: Example of the last three steps in this process

As presented before, first of all we collect data at low-level developer sensors by using Hackystat. Next, we extract the data of the sensors into a developer event stream and then, we partition the event stream into episodes. Finally, we analyse if each episode is TDD or not. Zorro uses that process to analyse if one episode is TDD or not. All the actions made by developers have associated metrics which give information about that action. As an example, if the metrics detects an increment in the number of tests, assertions and size of the unit test file, Zorro detects that it is an action to create a test.

Zorro defines the partitions of the streams of developer actions in episodes when one test passes. So, each passing test determines the end of an episode. To make the decision if one episode is a TDD conformance or not, Zorro uses a system to recognize it based on rules. One example illustrating this fact consists in reception of this sequence of rules: (a) a test method is created, (b) a compilation error, (c) a method is created in production code or production code is edited with a successful compilation,(d) all the tests pass. These actions conform a TDD episode.

2.5.2 Besouro

As presented in the introduction of this document, Besouro is a plug-in of Eclipse. So, we need to have Eclipse installed to use it. The Besouro plug-in is an open source that can be downloaded from https://github.com/brunopedroso/besouro. In order to use the Besouro plug-in Eclipse SDK must be downloaded, containing the Eclipse Plug-in Development Environment (PDE). After done the installation, the main Besouro project will present an error because of a missing library. It corresponds to the Jess Rule engine, which is derived from Zorro, until the time, it could not be redistributed due to its non open source license, but it is possible to obtain it from the web page *www.jessrules.com*.[18]

Once we have installed the plug-in, the plug-in is ready for use as we can see in the next figure:

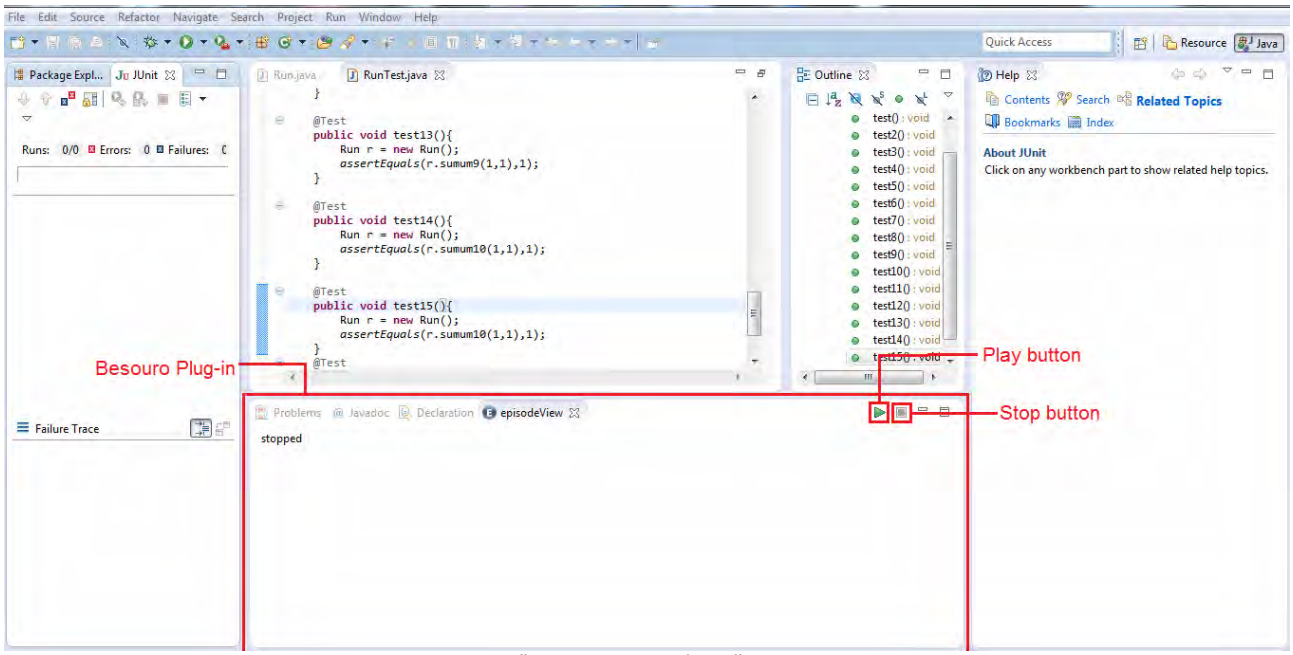

figure 6: Besouro in Eclipse

The functionality of this plug-in is easy to understand. There are two buttons labelled play and stop. Before starting to write the code, the play button must be clicked. Next, when the developer or developers make an event (for example test-first is an event where the test is done first and the code later on (TDD) and

if it works as the test requires, this is an event), the steps of this event followed to arrive at the solution can be seen. These actions are ruled by Zorro as we will see in the section 2.6 of this document.

Besouro makes a folder in the workspace of Eclipse, which is generated to obtain all the actions created during the development time. These actions are created by different text files. These text files contain the actions done by the developer. Have the same but in greater amount depending on what you want to see. In our case we take special care about two text files, namely *actions* which contains all the actions that developer made and *zorroEpisodes* which contains all the events that the developer have made related to the actions. These events are ruled according to the Zorro episodes as shown in section 2.6 of this document.

It is important to analyse the file of actions because this file contains all the actions that the user had done during the development of the software. Let us see an example of the contents in the file *actions.txt*:

```
FileOpenedAction 1396960088888 MarsRover.java 1083 2 1 0

RefactoringAction 1396960105966 MarsRover.java ADD void mo FIELD

RefactoringAction 1396960110471 MarsRover.java RENAME mo=>void move FIELD

RefactoringAction 1396960118976 MarsRover.java RENAME move=>void moveFor FIELD

RefactoringAction 1396960120980 MarsRover.java RENAME moveFor=>void moveForward FIELD

RefactoringAction 1396960122984 MarsRover.java RENAME moveForward=>void moveForward() METHOD

RefactoringAction 1396960133505 MarsRover.java ADD void move FIELD

RefactoringAction 1396960136509 MarsRover.java RENAME move=>void moveBackw FIELD

RefactoringAction 1396960139013 MarsRover.java RENAME moveBackw=>void moveBackward FIELD

RefactoringAction 1396960140517 MarsRover.java RENAME

...
```

To understand this document, we need to take into account the order of presentation of the stored information. In the first place, the action done by the user is presented. Second, the time when the user had done the action is listed

and next the document affected by this action is presented. Finally the parameters affected by the type of the action are shown.

Another file takes these actions in groups of episodes (the separation of the episodes are made when there are actions that indicates that one test pass, this is the end of an episode) and classifies, using the classification of Zorro, which kind of episode is. Let us see an example of the contents in the file zorroEpisodes*.txt*:

```
1397046664435 test-last 1 612 false
1397046727883 test-addition 1 63 true
1397047732823 test-first 2 1004 true
1397047920389 test-first 3 187 true
1397048365513 test-first 3 445 true
1397048615493 test-first 3 249 true
1397048764502 test-addition 1 149 true
1397049101968 test-first 3 337 true
1397049783642 test-first 3 681 true
1397049791477 regression 1 7 true
1397049792611 regression 1 1 true
1397051590239 test-first 3 1797 true
1397051656662 test-addition 1 66 true
1397051791343 test-addition 1 134 true
...
```

To understand this document, first we have the time of the last action in this episode done by the user, next the name of the episode classified, next the type of the episode, next the total time dedicated in this episode and, finally, the conformance of the episode in TDD.

About the implementation of Besouro, we can see that the architecture of Besouro follows the next figure:

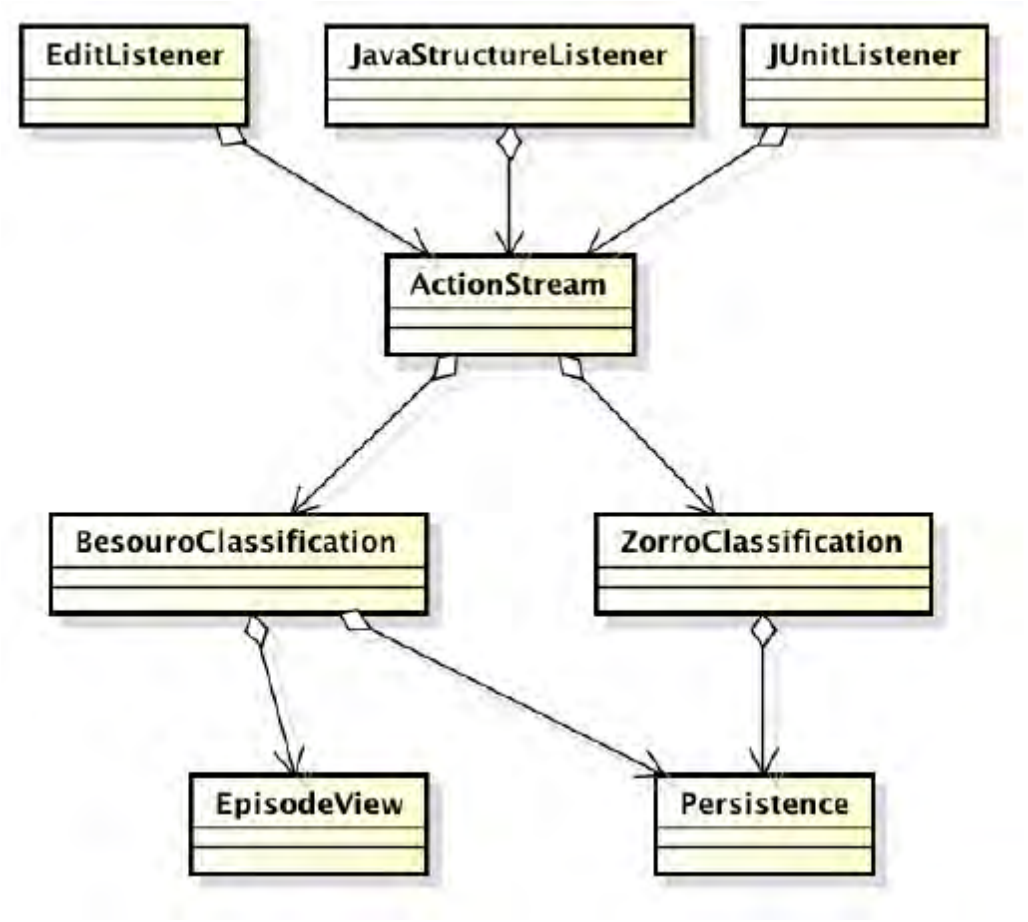

Figure 7: Besouro architecture

This architecture can be seen as a layer architecture, the first layer is the layer of listeners of Besouro, to obtain the events occurred in the interface of Eclipse, registered in the standard IDE for develop plugins in Eclipse. These events are the same events collected by Zorro (open and close files, edit files, changes in program structure, compilation errors and execution of the tests). Some of this events are not used because are not important to classify episodes.

The calculated metrics, considered about the rules to classify episodes follow the original system: name and size of the file, number of bytes, name of the class of the file, identification of the class as a test or production. The system admits the application of the rules in parallel. Some components called *EpisodeListeners* are notified for the streams every time that one episode is recognized, and save the classification in disk.

This architecture allows to performing the classification of the episodes during the task of programming and showed by the console of the Eclipse.

Before going on to see how Besouro classifies the episodes, we need to know some details about Besouro. First of all, Besouro not only uses the classification derived from Zorro. This thesis is focuses on this part which is directly related to Zorro. However, it should be noted that there are other additional types of classification. This additional classification is not the main to be considered because the results given by Besouro in Eclipse console are derived primarily from the classification of Zorro. The results of Besouro related to the additional classifications are stored in files separated from the results dericed from the classification from Zorro.

Furhtermore, a created class or method can not be named test (a method or a class are different ways of identifying parts of Java code; methods are functions and classes come from the fact of being object oriented programming) because Besouro uses this name as a reference to determine where is a test located (if test besouro is written in production code, a wrong classification will be done since it will be identified as a test). It is also important, to note the need of start Besouro at the time when the developer starting his activity, because activating Besouro after something has been done may cause the result of this episode to be wrong.

2.6 Episode classification

Let's see which are the rules used by Zorro to identify whether one episode is conformance with TDD or not. This is based on the behaviour followed by developers when they are programming in TDD. For example, if the developer only makes a test case but he does not change the source code, it is considered as a test addition. Test addition is another kind of rule that Zorro follows to classify the episodes. This case is more complex because this episode can be conformant with TDD or not. The key to solve this is to look at the previous episode to check if it is conformant with TDD or not.

The rules that Zorro uses to analyse it are 22. Here you can see the types of the episodes of Zorro:

ID	Definition	TDD Conformant
Test First		
TF-1	Test creation -> Test compilation error -> Code editing -> Test failure -> Code editing -> Test pass	Yes
TF-2	Test creation -> Test compilation error -> Code editing -> Test pass	Yes
TF-3	Test creation -> Code editing -> Test failure -> Code editing -> Test pass	Yes
TF-4	Test creation -> Code editing -> Test pass	Yes
Refactoring		
RF-1	Test editing -> Test pass	Context sensitive
RF-2	Test refactoring operation -> Test pass	Context sensitive
RF-3	Code editing (number of methods or statements decrease) -> Test pass	Context sensitive
RF-4	Code refactoring operation -> Test pass	Context sensitive
RF-5	[Test Editing && Code editing (number of methods or statements decrease)]+ -> Test pass	Context sensitive
Test Addition		
TA-1	Test creation -> Test pass	Context sensitive
TA-2	Test creation -> Test failure -> Test editing -> Test pass	Context sensitive
Regression		
RG-1	Non-editing activities -> Test pass	Context sensitive
RG-2	Test failure -> Non-editing activities -> Test pass	Context sensitive
Code Production		
CP-1	Code editing (number methods unchanged, statements increase) -> Test pass	Context sensitive
CP-2	Code editing (number methods /statements increase slightly (source code size increase <= 100 bytes) -> Test pass	Context sensitive
CP-3	Code editing (number methods /statements increase significantly (source code size increase > 100 bytes) -> Test pass	No
Test Last		
TL-1	Code editing -> Test editing -> Test pass	No
TL-2	Code editing -> Test editing -> Test failure -> Test pass	No
Long		
LN-1	Episode with many activities (> 200) -> Test pass	No
LN-2	Episode with a long duration (> 30 minutes) -> Test pass	No
Unknown		
UN-1	None of the above -> Test pass	No
UN-2	None of the above	No

Table 1: Zorro episodes types, definitions and TDD conformance.

To understand the classification, let us see the different types of episodes shown in this table:

- Test First: Test first have four possible types. The idea of this episode is first test creation and next application of that test. The four types come from the different ways that we can arrive at this episode.

- Refactoring: This part has, also, different types. The type one is in case that there is an edition of a test and next the test passes. The second type is when developer makes a refactorization of his test and then passes the test. The third type is a code editing and then pass a test previously created. The fourth type is when developer makes a refactorization of his code and then passes the test and, the fifth type is a combination of the types two and four and then the test passes too.

- Test addition: This classification is when developer adds a new test and, without changes in the production code, the test passes.

- Regression: Is in case that one test that passes without changes in the production code (this can be, for example, adding some libraries).

- Code production: Is in case that the developer makes a production code and then passes a test.

- Test last: This case is the opposite of the test first, that is. first edit the code and then apply the test.

- Long: These cases are when the code production is too long to be analysed.

- Unknown: This case is when we do not know what happens exactly, is the default case when it does not follow any of the other cases.

Once all the episodes are classified, the final step consists in the conformance of this TDD. In the previous table, you can see that the half part of the rules defines if one episode is TDD conformance or not. The rest needs to see the previous episode in order to define if it is TDD or not. These cases are the cases classified as a context sensitive.

2.7 Jess

Jess is a rule engine and scripting environment written entirely in Oracle's Java language by Ernest Friedman-Hill. Using Jess, you can build Java software that has the capacity to "reason", based on declarative rules Jess finds the rule that makes the accomplishment of the conditions. Jess is made to work in Eclipse platform like a library in the project or in the Eclipse depending on what developer needs to make.

Jess uses an enhanced version of the Rete algorithm to process rules. Rete is used for solving the difficult matching problem. Some advantages of using Jess is that you can manipulate and reason about Java objects, create Java objects, call Java methods, and implement Java interfaces without compiling any Java code[3].

The main problem of using Jess is that Jess is not an open source code. You need to pay to use it for commercial purposes, but you can download a free trial version for 30 days.

Besouro uses this Jess library. The moment when Besouro needs to use it is when the actions of the developer have to be analysed to see what kind of episode is. Besouro implements the rules of Zorro using Jess.

To better understand why Besouro uses Jess to classify episodes, let us see an example of one particular rule:

```
;; Type 4 test-first iteration

(defrule Test-Driven-Classifier-Type4
    (declare (salience -100))
    (there-is-test-creation ?a)
    (ProductionEditAction (index ?p))
    (not (test-compilation-error-then-production ?b ?c))
    (not (unittest-then-production ?d ?e))
    (test (< ?a ?p))
    =>
    (assert (episode
               (category test-first)
               (type 4)
               (explanation (str-cat ?a "," ?p))
             )
     )
)
```

This case corresponds to the rule Test-First type 4 which first says that there are a test creation `(there-is-test-creation ?a)`, next a production code `(ProductionEditAction (index ?p))`, then the condition that the test cannot be compiled and then production code the test have to pass `(not (test-compilation-error-then-production ?b ?c))`, `(not (unittest-then-production ?d ?e))`, the next condition is about the index of the positions of the action in that case the test creation (a) have to be less than the code production (p). Finally, in the assert there is the result of that classification. If an episode follows these conditions, this result is obtained.

3. Research questions

In this chapter the three principal research questions are presented. This research questions are:

- RQ1: Assess whether Besouro identifies correctly episodes or not.

We want that Besouro have a good conformance to see if the episodes of TDD are conformant or not conformant. If Besouro classifies episodes incorrectly then the datasets of the experiment ACME² (and other experiments) will be incorrect. This situation is a luxury in an experimental point of view that we cannot pay.

- RQ2: Identify the origin of the incorrect identification of episodes.

We need to see the internal working of Besouro by making different trials and experiments. These experiments have to be evaluated by looking in the code and see how Jess makes the classification of the episodes. We do this to know which errors have Besouro, with the intention to fix it and fix the datasets of the existing experiments.

- RQ3: Evaluate the impact of Besouro's wrong behaviour in existing datasets.

We want to know the grade that Besouro fail, this is the difference between detected episodes Test-First and then the ITL and TDD episodes. Looking into the datasets (in our case ACME²) in order to find the correct and incorrect actions in the result of Besouro and find a pattern that can fix the result of actions as far as possible.

And the actions that we have done are:

- A1: Create a software to display Besouro's data.

We want to create a software that show the results of the actions to analyse different things. In our case we the production code size, test code size, test methods, test/production time ratio, test/production code size ratio, assertions/test method ratio, production versus test code size and production versus test time. This software is made in language R, you can download this code in the ANNEX III.

- A2: Fix the Besouro's code for future experiments.

Once we find the Besouro errors, change or add the part of the code that is wrong and make proves to see if the result of the Besouro is correct or not. This research question is not easy, because there are a lot of possibilities of classification in the 22 episodes and there are different possibilities for each episode, so it is difficult to prove all. To download the Besouros code, you can do it in the ANNEX I.

- A3: Fix the existing datasets whenever possible.

We want to fix the datasets approximately to the reality. It is possible to find a pattern to redistribute the actions in a correct order as far as possible. But each action has some parameters that are impossible to know exactly. In our case principally there are two values. One is the time value and the other is the quantity of the size that has changed. To download the software, you can do ir in the ANNEX II.

4. Methodology

The methodology applied in this work to answer the main questions addressed in this investigation, follows this diagram:

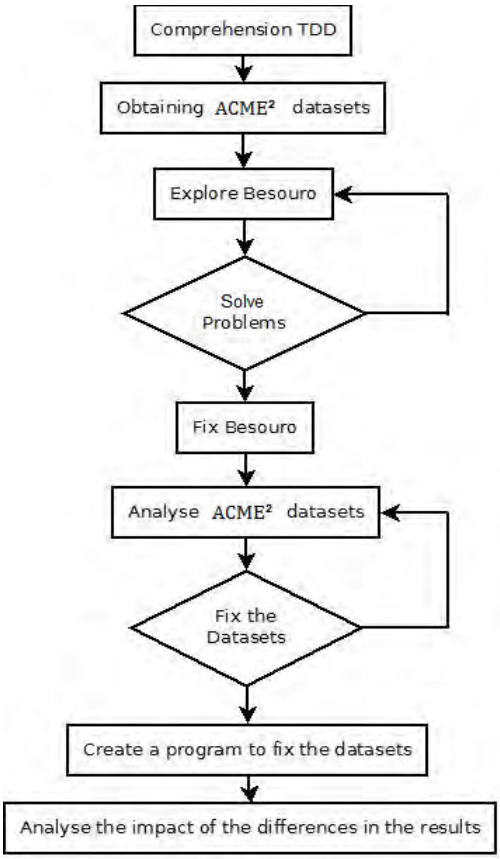

Figure 8: Diagram the steps of this thesis

The different steps of the process are described next:

Comprehension of TDD: A good comprehension of what is TDD is needed. This step is reflected in this document by briefly explaining the basic concepts about TDD by using theory of TDD, (we made a project about TDD in the third semester of Master EMSE).

Obtaining ACME² datasets: The motivation of this thesis started when we saw that the results of the ACME² experiment based on Besouro were wrong.

Explore Besouro: Explore Besouro code to see which problems provoke these wrong results and, then, fix them. We made a software using code R to see it.

Solver problems: Once the problems are fixed, we update that problem in the repository of Besouro. This repository is in *https://github.com/brunopedroso/besouro,* is an open source and we made a fork to the creator that can see our proposition of solution of the Besouro.

Analyse ACME² datasets: Analyse the datasets to see what kind of patterns we can to follow to solve the mistakes that Besouro had done. This is for actions and for the episodes classification

Fix the datasets: Once found the patterns able to solve the mistakes, we check if the fixed datasets correspond to a real type of episodes.

Fix the datasets: Create a program to fixes it, as far as possible. The new results must be more real than that we had when we started this thesis.

Analyse the impact of the differences in the results: And finally, we have to analyse the new results and evaluate the impact percentatge between this results with the previous ones.

5. Execution

5.1 Creation of a software to display Besouro's datasets

During the experiment of ACME², the developers had instructions about the methodology to make the software. These instructions were defined to see the difference between the conduct of the developers at the time when they were developing. The objective of the instructions was to check if the TDD was well done or not. Once the results were obtained, we can see that big part of the results were wrong. All the episodes in that way were wrong and do not started in the time they should. For example, in case of test first the developers do not start by the fails of the test but there are first time using the source code and then the fail of the test. It seems strange that in all cases it was the same. We created a software in code R in order to look deeper into this situation. We illustrate this problem by showing the following sample of the analysis of the results:

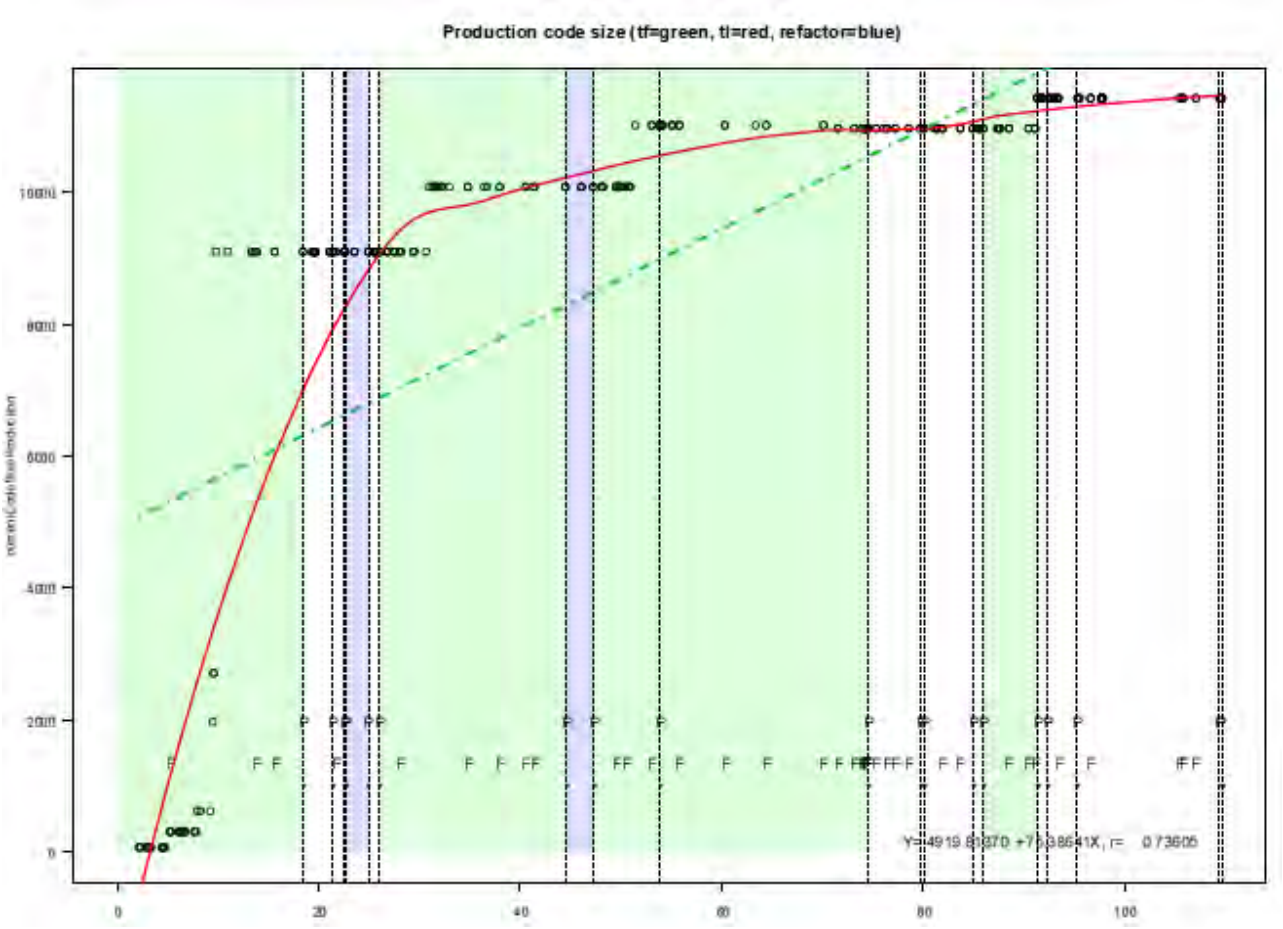

Figure 9: Analysis of Besouro results by using software coded R

This graph shows the production of the code by the size. We will focus in the episodes present in this case have. Every episode is marked by dashed lines and every episode ends when one test passes (indicated by letter P (pass) in the graph). The problem is that the majority of letters F (fail) are no located at the beginning of the episode but in the middle. It can be accepted tha some letters F are located in the middle of an episode, but at least one should be at the beginning. So, these results are wrong because in the case of a test-first episode the results indicated that it is test-last.

5.2 Assessment about correct identification of episodes

To better understand this problem, we make a function called add. That function adds two numbers, then we create a test to add 1 plus 1 and assert if the result is 2. This is the code of the functions:

```
public int add(int i, int j){
            return(i+j);
```

```
	}
```

And the test to prove it is:

```
@Test
	public void test(){
		Besouro a = new Besouro();
		assertEquals(2, a.add(1, 1));
	}
```

The result Besouro gives in this case is Test-First. So, the results of Besouro are not correct in some cases. The problem is due to the order of the actions that the developer had done which are not exactly identical to the reality. This plug-in evaluates the actions and then obtains which kind of episode (test-first, test-last, refactoring etc..) had they done. But the actions are not well ordered so the results of the episodes are wrong.

To verify if Besouro identifies correctly episodes or not, we started to make proofs by using Besouro and we saw that all the actions that we did in one episode were returned by the Besouro in an incorrect order. One of the problems that can have Besouro is not in the way that Besouro makes the episodes but in the order that the actions arrive to Besouro. You can see that problem in action in the next video:

-Video: Besouro fail incorrect order episodes

If you watched the video, you can understand what exactly that problem is. First, we make a function that adds two numbers. Next, this addition is tested by creating a test. So this episode using Zorro is Test-Last type 1, but Besouro gives that this episode is Test-First type 4. The results of the sequence of actions are:

```
FileOpenedAction 1434670028205 Besouro.java 47 0 0 0

RefactoringAction 1434670034274 Besouro.java ADD in add FIELD

RefactoringAction 1434670036297 Besouro.java RENAME add => int add(int) METHOD

RefactoringAction 1434670038854 Besouro.java RENAME add(int) => int add(int,int) METHOD

RefactoringAction 1434670064390 BesouroTest.java ADD void test() METHOD

RefactoringAction 1434670086081 BesouroTest.java ADD import Besouro.Besouro IMPORT

EditAction 1434670096553 Besouro.java 102 1 1 0

EditAction 1434670100732 BesouroTest.java 244 1 2 1

UnitTestCaseAction 1434670106636 BesouroTest.java OK

UnitTestCaseAction 1434670106651 BesouroTest OK
```

Also, we prove exactly the same as before but with the only difference of saving the change to the code in to the test, In this case, we obtain that the classification of the episode is correctly Test-Last type 1. So, looking into the actions in the results that the developer has done, we see that the order of the actions in both cases are different. The results of the sequence of actions are:

```
RefactoringAction 1434670120984 Besouro.java ADD int minus() METHOD

RefactoringAction 1434670125007 Besouro.java RENAME minus() => int minus(int,int) METHOD

EditAction 1434670141999 Besouro.java 159 2 2 0

RefactoringAction 1434670157978 BesouroTest.java ADD void test() FIELD

RefactoringAction 1434670158997 BesouroTest.java RENAME test => void test()/2 METHOD

EditAction 1434670189134 BesouroTest.java 345 2 4 2

CompilationAction 1434670196082 BesouroTest.java

RefactoringAction 1434670208350 BesouroTest.java RENAME test()/2 => void test2() METHOD

CompilationAction 1434670211453 BesouroTest.java

UnitTestCaseAction 1434670212850 BesouroTest.java OK
```

```
UnitTestCaseAction 1434670212852 BesouroTest.test2 OK
```

Performing other experiments we saw another problem, particulary in episode Test-Last type 2. In that case, if we have a correct order of the actions, the result of the episode is wrong. You can see that fail in the next video:

-Video: besouro fail Test-Last type 2

Watching that video, we prove in that case the episode Test-Last type 2 as we can see in the rules of Zorro. In this video, we can see first of all the same error as we found before, when we included the function add, we can see that the order of the actions are wrong. The results of the sequence of actions are:

```
FileOpenedAction 1434670856443 Besouro.java 47 0 0 0

RefactoringAction 1434670867507 Besouro.java ADD int add(int,int) METHOD

RefactoringAction 1434670888978 BesouroTest.java ADD void test FIELD

RefactoringAction 1434670889990 BesouroTest.java RENAME test => void test() METHOD

EditAction 1434670922168 Besouro.java 104 1 1 0

EditAction 1434670924049 BesouroTest.java 242 1 2 1

UnitTestCaseAction 1434670928999 BesouroTest.java FAIL

UnitTestCaseAction 1434670929001 BesouroTest FAIL

EditAction 1434670938386 Besouro.java 102 1 1 0
```

```
UnitTestCaseAction 1434670941976 BesouroTest.java OK

UnitTestCaseAction 1434670941992 Besouro OK
```

But using the saving operation between the changes we can see that the order of the actions is well done, but if we look at the classification of the episode it is Test-First type 4. So, this is another problem that we find in the Besouro plug-in at the moment of classifying episodes. The results of the sequence of actions are:

```
RefactoringAction 1434670951851 Besouro.java ADD int m FIELD

RefactoringAction 1434670953367 Besouro.java RENAME m => int minus FIELD

RefactoringAction 1434670956882 Besouro.java RENAME minus => int minus(int,int) METHOD

EditAction 1434670970086 Besouro.java 161 2 2 0

RefactoringAction 1434670985887 BesouroTest.java ADD void test2 FIELD

RefactoringAction 1434670986904 BesouroTest.java RENAME test2 => void test2() METHOD

EditAction 1434671020340 BesouroTest.java 341 2 4 2

UnitTestCaseAction 1434671023829 BesouroTest.java FAIL

UnitTestCaseAction 1434671023830 BesouroTest FAIL

EditAction 1434671023830 Besouro.java 159 2 2 0

UnitTestCaseAction 1434671037638 BesouroTest.java OK

UnitTestCaseAction 1434671037640 Besouro OK
```

While we were doing these activities we found other problems of Besouro. These problems are:

- There are inconsistencies between the rules. Using Jess, this evaluates the actions based on rules. We find that depending on what kind of rule we want to activate, Jess returns two or more rules that accomplish the actions. So, the rules are not well defined. The system of Besouro only takes the first rule that arrives. In our case, we find that Test-First type 3 and Test-Last type 1 have some inconsistencies that have to be solved for a well classification of the episodes.

- There are more inconsistencies in the definition of the rules in different cases of episodes, Jess activates different rules for each episode and Besouro classifies the episode with the first rule that is activated. So, we

think that is possible that there are a more inconsistencies in the classification of the episodes than that we found. Is difficult to test all the rules and obtain a perfect definition of each rule without inconsistencies.

5.3 Identify the origin of the incorrect identification of episodes

5.3.1 Origin of the incorrect order

To identify the origin of the incorrect classification of the episodes given by Besouro, we have to look at the code. To see how Besouro receives the actions by Eclipse, we have to look at this piece of the code:

```
	public void resourceChanged(IResourceChangeEvent event) {

		if ((event.getType() & IResourceChangeEvent.POST_CHANGE) != 0) {
			try {
					IResourceDelta rootDelta = event.getDelta();
					delta = rootDelta;
				// Accepts the class instance to let the instance
be able to visit resource delta.
				buffer = new ArrayList<EditAction>();
				delta.accept(this);
				reorganizeSensor();
			} catch (CoreException e) {
				e.printStackTrace();
			}
		}
	}
```

This piece of code coincides with the moment that Besouro receives the actions. This function uses the parameter `IResourceChangeEvent event` to, activate the start of Besouro to collect the actions. The problem that appears is that this parameter is only received when the changes are saved in Eclipse.

The first problem found was the results about the order of the actions. We also saw in the video that if we previously save in Eclipse, the order of the actions become correct. Looking in the code we can see that the actions given are

activated when a developer saves or compiles his project. This is because these actions are given by internal functions of Eclipse in the moment of saving the project. We don't know the order in which Eclipse saves the information, so the order of the actions does not necessarily correspond exactly with the reality. This problem appears in each case that developer applies TDD because two classes are needed at minimun to make it, namely, one for code and another for tests. So, the order of these cases is wrong each time.

We found, by making some trials, that the actions are not given in the exact time when the developer actually makes them. We receive nothing from the plug-in until the developer saves or compiles the project. Unless these actions are given, Besouro cannot start to classify the episode and the developer can make any change that will be lost by Besouro. At the end, when the developer selects save or compile the project all the actions are given by the event of Eclipse. However, these actions are given by a different order than the real one followed by the developer and these actions do not reflect the real actions that the developer had done. If we look at the two examples that we show in section 6.2, we can see that the EditActions do not have difference of time between them, and the order of the RefactoringActions are mixed between production code and test.

The impact of the previous presented wrong operation in existing datasets is very important. In order to evaluate this impact, we looked deeper in the experiment of ACME² and we detected two different types of documents. One of the documents is called *actions* and contains, all the actions that the developer had done during the time that Besouro has been activated. The other document is *zorroEpisodes.* These two files are the most important to take into account in order to analyse the results given by Besouro. The impact is that, in the first document named *actions* (as shown in the video) the indicated actions of *EditAction* are not correct and the order of the actions is incorrect too. In *EditAction,* the order is not correct, but not only this, as shown in this example of wrong classified episodes:

```
FileOpenedAction 1434670028205 Besouro.java 47 0 0 0
RefactoringAction 1434670034274 Besouro.java ADD int add FIELD
```

```
RefactoringAction 1434670036297 Besouro.java RENAME add => int add(int) METHOD

RefactoringAction 1434670038854 Besouro.java RENAME add(int) => int add(int,int) METHOD

RefactoringAction 1434670064390 BesouroTest.java ADD void test() METHOD

RefactoringAction 1434670086081 BesouroTest.java ADD import Besouro.Besouro IMPORT

EditAction 1434670096553 Besouro.java 102 1 1 0

EditAction 1434670100732 BesouroTest.java 244 1 2 1

UnitTestCaseAction 143467016636 BesouroTest.java OK

UnitTestSessionAction 14367016651 Besouro Test OK
```

If we focus on the actions *EditAction,* we can see more errors, as for example, the first value *1434670096553* indicates the time when the file has been changed, but in our case, we received it when Besouro compiled it. As a consequence, all the files have the same time and not the real time. Another important question is that we do not know how many times the user has been changing the code. The value of size, in this case *102,* does not need to be real depending on the actions that the developer had done. For example, an error will appear in the case that the developer, without saving his process, edits one Java file and then changes another Java file and next returns to the first Java file to edit it again. In that case, we have in the first Java file two real edit actions with two different sizes. However, without having save this steps, Besouro returns these actions as if there was only one which has the time when it, has been compiled (it can even pass more time until the developer decides to compile or save) and we only can see the last size corresponding to the last edition and not the intermediate steps.

It is necessary to create a software able to change these documents and make the results as real as possible. Unfortunately it is very complex to fix completely the results because it is impossible to know exactly the times and the sizes done in each step of the developer. In chapter 5.4 of this document, you can see our proposal to solve the results that Besouro generated in the ACME² experiment and other possible experiments that used Besouro. The basic idea that we want to apply is to change the last episode with piece of code:

```
FileOpenedAction 1434670028205 Besouro.java 47 0 0 0
```

```
RefactoringAction 1434670034274 Besouro.java ADD int add FIELD

RefactoringAction 1434670036297 Besouro.java RENAME add => int
add(int) METHOD

RefactoringAction 1434670038854 Besouro.java RENAME add(int) => int
add(int,int) METHOD

EditAction 1434670038855 Besouro.java 102 1 1 0

RefactoringAction 1434670064390 BesouroTest.java ADD void test()
METHOD

RefactoringAction 1434670086081 BesouroTest.java ADD import
Besouro.Besouro IMPORT

EditAction 1434670086082 BesouroTest.java 244 1 2 1

UnitTestCaseAction 143467016636 BesouroTest.java OK

UnitTestSessionAction 14367016651 Besouro Test OK
```

5.3.2 Origin of the incorrect definition of the Zorro rules

The second video case presented before, illustrates the situation where the order of the actions is well done but the classification of the episode is wrong. In particular, it corresponds to the case of Test-Last type 2. We saw the problem when we looked into the code. The problem is that the Test-Last type 2 is not defined in the code. There are all the other rules except the Test-Last type 2. We need to define the rule that looks for episode Test-Last type 2 without inconsistencies between the rest of rules.

With relationship with the last error that we found, the rules are too much complex to link the table of Zorro rules with the Jess rules. This is, because the rules defined by Zorro are conceptual, but in Jess the rules are operative. Because there are 22 rules and some rules follow a similar pattern and depending on the actions that developer had done, it is possible that Jess activates more than one rule. To understand better this problem, we present the example of rule, Test-Last type 1. The definition of this rule is:

```
(defrule Test-Last-1-Classifier
   (declare (salience -100))
   (ProductionEditAction (index ?p) (file ?pfile))
   (there-is-test-creation ?a)
   (test (< ?p ?a))
```

```
   (UnitTestAction (file ?ufile) (errmsg ?msg&:(eq ?msg nil)))
  =>
   (assert (episode
               (category test-last)
               (type 1)
               (explanation (str-cat ?a "," ?p))
           )
    )
)
```

This rule defines a Test-Last type 1, by Zorro. This case starts with a first production code, and next follows with the creation of a test, `(test (< ?p ?a))`. This situation implies that there is first a production code, and next the creation of the test, to finally end up with the application of the test. But the problem of this rule is found in cases as for example when the developer does the Test-Last type 2, this rule can pass to because there are not a condition that one test have to fail and then pass. So, is inconsistent and can be activated in other cases than the cases of Test-Last type 1.

5.4 Fix Besouro's code for future experiments

5.4.1 Fix Besouro's code for the actions

When we saw the problem in Besouro, we started to think what kind of solution could we propose for this plug-in. The problem is that the actions that Besouro analyses arrive by eclipse using an event. This event is received in the plug-in every time that the developer saves or compiles (when you compile in Eclipse the files are also saved). So, the solution was that each time the developer changes the tab of the document that were he is saving if the progress must be done. This seems an effective idea because Eclipse gives, by using the event step by step, all the steps that the developer made in a good order. Also. The proposed solution does not affect the episodes because the episodes finish only when the test passes.

To understand better what we made to solve that problem, we insert this part of code in the source code of Besouro in the part of window listeners:

```
public void partDeactivated(IWorkbenchPart part) {
	if (part instanceof ITextEditor) {
		if(part.getTitle().contains(".java")){
			PlatformUI.getWorkbench().saveAllEditors(false);
		}
	}
}
```

Let's see in detail each line of this change. The first conditional is part of workbench library and this part listens all that the developer are using in the Eclipse. In this case, the conditional is used when the developer disables one part of the Eclipse. The next conditional ensures that this part is a text file (the other parts of Eclipse are not important for this case). The last conditional is to ensure that the document is a Java file because we do not need to pay attention to other types of files like UML or notes or others. Finally, the action consists in saving the changes in the project which is the same action as seen in the video of the demonstration that Besouro fails when we save it, in the step when we change the source code to test code.

-Video: Besouro work saving

As we can see in the video, when we change the source code to the test part, the asterisk of the tab disappears. This is because when developer changes the Besouro plug-in save all the changes. This is not the only process that is making. On the other hand, Eclipse gives the event to Besouro as well as all the actions that the developer has done. So, now we receive in a good order all the actions and Besouro can analyse them with both, a correct order of the actions and which type of episode is.

5.4.2 Fix Besouro's code for the rules

The order of the actions is not the only problem that we found. Previously, we found that the rule of Test-Last type 2 is not created in the code. So, we created the Test-Last 2 coordinated between the other rules that are codified, in order to not disturb the other rules. The code of that rule is as follows:

```
(defrule Test-Last-2-Classifier
   (declare (salience -100))
   (ProductionEditAction (index ?p) (file ?pfile))
   (there-is-test-creation ?a)
   (test (< ?p ?a))
   (unittest-then-production ?d ?e)
   (UnitTestAction (file ?ufile) (errmsg ?msg&:(eq ?msg nil)))
  =>
   (assert (episode
              (category test-last)
            (type 2)
            (explanation (str-cat ?p "," ?a "," ?d "," ?e))
            )
    )
)
```

This rule defines the Test-Last type 2. This rule defines the option as first edit code, next creation of the test, then, the test do not pass and, finally, modify the code and pass the test.

This step was not easy, because, we had to learn the syntax of Jess and these rules of production are a paradigm of programming different than the imperative programming. For this reason, we had some difficulties to be adapted to this language. We had some problems, also, to interpret why the rules use indexes to link the actions, which is the order that the actions follow to be agreed with the conditions. We had some problems too, to understand the previous definitions to identify some aspects as that for example code production or test compile error because this rules are defined depending on the cases.

With the previously presented changes, we can see in the next video that Besouro is now working as should to be working. That is, returning well the order of the actions and the rule of Test-Last type 2:

-Video: Besouro work rule of Test-Last type 2

In the second case, we made the same but we fail in the code, so we can check that when is Test-Last type 2 it is true too. When we were making the trials, we saw that some inconsistencies appeared between the rule and the definition of this rule. More particularly, we found inconsistencies in Test-Last type 1 and Test-First type 3. This was because we found that, for example, if we test other

episodes like Test-Last, Besouro then recognizes it as a Test-First type 3. The reason for that is found in the definition of the rule. Now we see the changes:

- In case of Test-First type 3, this is the definition of the rule before our changes:

```
(defrule Test-Driven-Classifier-Type3
    (declare (salience -100))
    (there-is-test-creation ?a)
    (not (test-compilation-error-then-production ?b ?c))
    (unittest-then-production ?d ?e)
    (test (< ?a ?d))
    =>
    (assert (episode
                (category test-first)
                (type 3)
                (explanation (str-cat ?a "," ?d "," ?e))
             )
    )
)
```

- This is our definition rule of Test-First type 3:

```
(defrule Test-First-3-Classifier
   (declare (salience -100))
   (ProductionEditAction (index ?p) (file ?pfile))
   (there-is-test-creation ?a)
   (test (> ?p ?a))
   (unittest-then-production ?d ?e)
  =>
   (assert (episode
              (category test-first)
            (type 3)
            (explanation (str-cat ?p "," ?a "," ?d "," ?e))
            )
    )
```

```
)
```

The rule of Zorro that define Test-First type 3 is Test creation → Code editing → Test failure → code editing → Test pass. We found this incongruence when we were checking Test-Last type 2. The main problem of that rule that we found is about the code editing. This is not defined in the first rule and this code editing can be the repeat the same production of the code. When we put this rule Test First type 3 it ceased to have inconsistencies in our experiments.

- In case of Test-Last type 1, this is the definition of the rule before our changes:

```
(defrule Test-Last-1-Classifier
   (declare (salience -100))
   (ProductionEditAction (index ?p) (file ?pfile))
   (there-is-test-creation ?a)
   (test (< ?p ?a))
   (UnitTestAction (file ?ufile) (errmsg ?msg&:(eq ?msg nil)))
  =>
   (assert (episode
               (category test-last)
               (type 1)
               (explanation (str-cat ?a "," ?p))
            )
    )
)
```

- This is our definition rule of Test-Last type 1:

```
(defrule Test-Last-1-Classifier
   (declare (salience -100))
   (ProductionEditAction (index ?p) (file ?pfile))
   (there-is-test-creation ?a)
   (test (< ?p ?a))
   (UnitTestAction (file ?ufile) (errmsg ?msg&:(eq ?msg nil)))
   (not (UnitTestAction (file ?ufile) (errmsg ?msg2&:(not (eq ?msg2 nil)))))
  =>
   (assert (episode
```

```
                (category test-last)
                (type 1)
                (explanation (str-cat ?a "," ?p))
            )
    )
)
```

This inconsistency, similar to that in Test-First type 3, appears when we check Test-Last type 2. If we look into the definition of Test-Last type 1 in Zorro, it is: Code editing --> text editing --> test pass. The inconsistency is because we need to assure that it cannot be any error in the test. For these reason, we add this (not (UnitTestAction (file ?ufile) (errmsg ?msg2&:(not (eq ?msg2 nil))))) in the rule. With this edition the inconsistency of the Test-Last type 1 when we check Test-Last type 2 disappears.

With the changes presented so far, do inconsistencies disappear? We cannot confirm it. We fear that there are more inconsistencies. But we cannot devote more time to ensure it. So it will be a work for the future.

5.5 Creation of a software to fix the existing datasets whenever possible

We created a software to fix the two text files, namely, *actions* and *zorroEpisodes.* The software, first of all, makes a mapping based on one folder given by the user. This mapping searches all the folders called *.besouro* because, in this folder, is where there are all the files that *Besouro* has made. This software collects all the actions that *actions* file indicate. Then, the software orders the actions, the way that the software orders it are done according to the next steps:

1- Create two lists one with all the actions and another with the actions called *EditAction.*

2- Introduce in correct order the *EditAction* actions in the list of all the actions with changed parameters trying to be the most faithful possible to the reality. To achieve this, first of all, we separate the actions into episodes. This operation is

easy because one episode finishes when one test passes. Next of it, we notice that the actions of *Refactoring* that are given (the major part) have correct time, so, when the actions of Refactoring change the name of the file we add the *EditAction* with the name of the file equal to the name of the file in the last *Refactoring* action. To put the parameters the most similar to the reality, we extract the time of the last *Refactoring* and put it in the action of *EditAction* with the time of the last *Refactoring* plus 1. In relationship with, the size of the *EditAction,* we put it first with the size of the action, but when this action is put another time we add 1 in that value to indicate that there were changes.

3- Classify each episode with the new order that we made. To make this, we extract to *Besouro* the part of the classification episodes and add in this software (notice that we need the Jess library), when one episode is ordered next of it we pass this episode to this classification to obtain what kind of episode is.

4- Finally, we save the actions ordered in a text file called *actions-adapted* and save the classification episodes in another text file called *zorroEpisodes-fixed.* They are saved in the same folder that we found the files (*.besouro*).

The diagram of the functionality is:

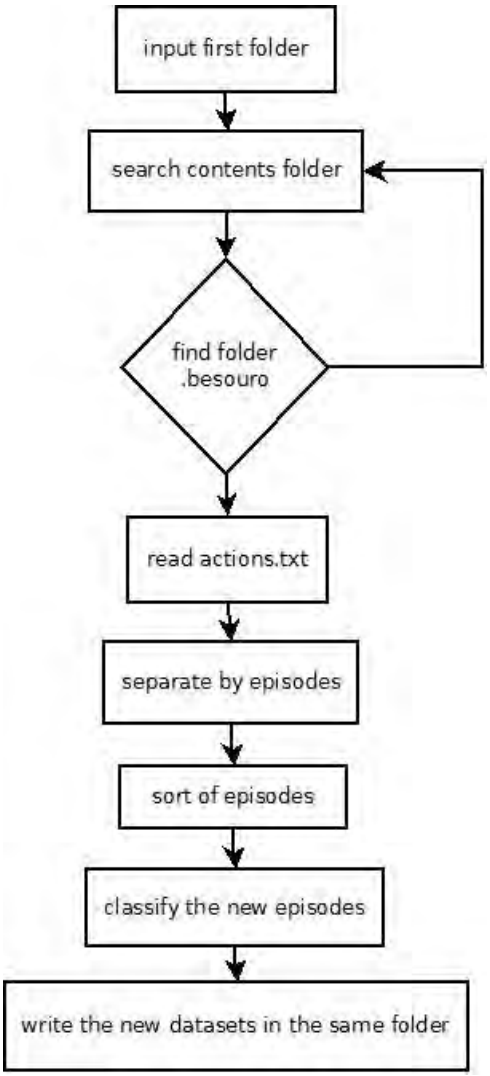

Figure 10: Diagram of sequence of the software.

We add the classes that Besouro uses to classify the episodes as well as all the models that Besouro uses to define the actions. These models are used to identify the different types of actions (*EditActions, CompileActions, RefactoringActions* etc...). We had some problems at the time of classifying the episodes because Besouro defines an internal queue to connect the *FileOpenedActions* and *EditActions* to identify the previous actions. At the same time, we had problems to add *EditActions* in the list of the actions because we had to modify the time depending on the last *RefactoringAction* that has used the same file and the value of the size that we had to increment by 1 to say that there are changes. Finally, when we classified the episodes we obtained the episode and type of episode but we had to take the time of the last action, we had to make a subtraction between the first time of the episode and the second

time. Finally about the conformance we put the n/a because this step is not developed is more easy find it, looking it. The reason to make it is because we want exactly the same structure that the files are done by Besouro.

5.6 Evaluation of the impact of Besouro's wrong behaviour on existing datasets

To evaluate the impact of our work on existing datasets, we created a software codified in language R as shown in annex III. In this chapter present the comparison between the new results versus the old results. In the 18 cases of study we obtained differences in practically all the cases but we show the most significance to understand the impact to fixing datasets.

First of all, we do not know exactly what the reason is but there are some cases where some episodes are missing as illustrated in the next graphs:

- Old results:

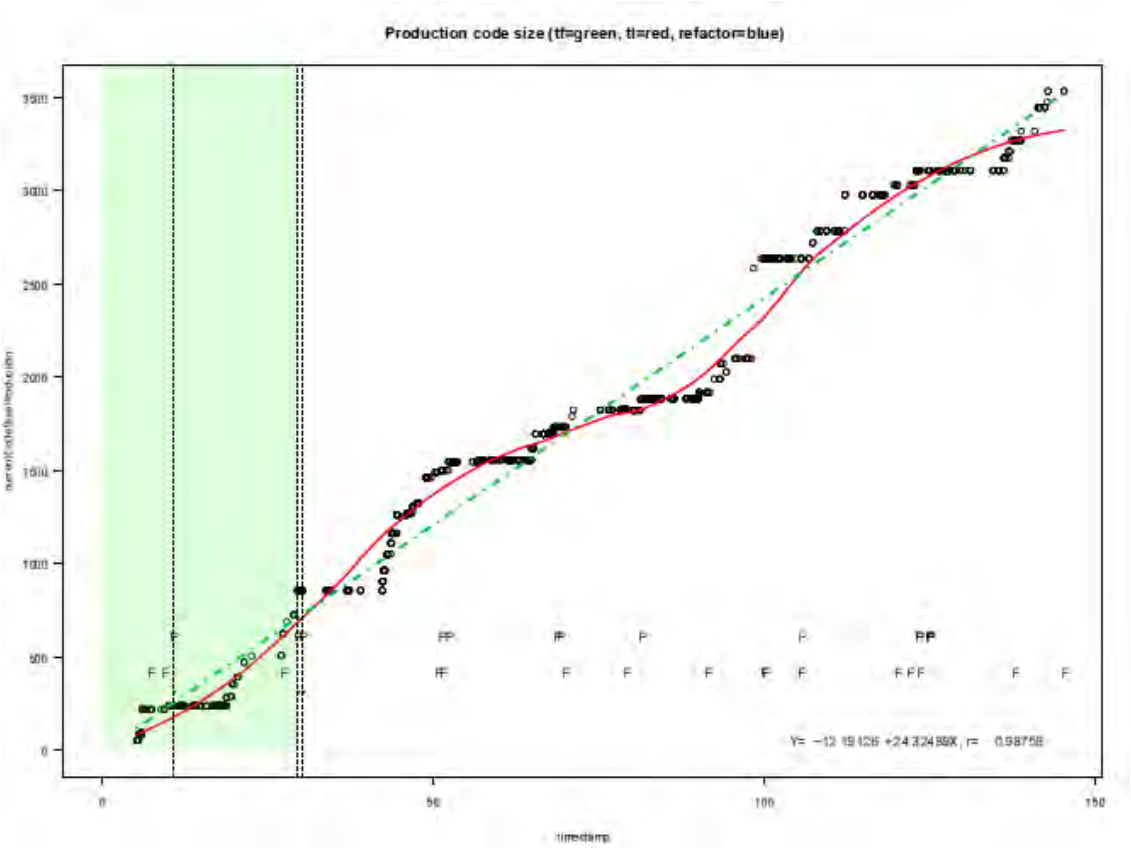

Figure 11: Analysis of Production code size in one case of ACME² experiment (older results)

As we can see in the graph corresponding to old datasets, the episodes are divided by vertical broken lines. In this graph, there are 4 episodes. But if we look the fixed datasets we can see some relevant changes:

- New results:

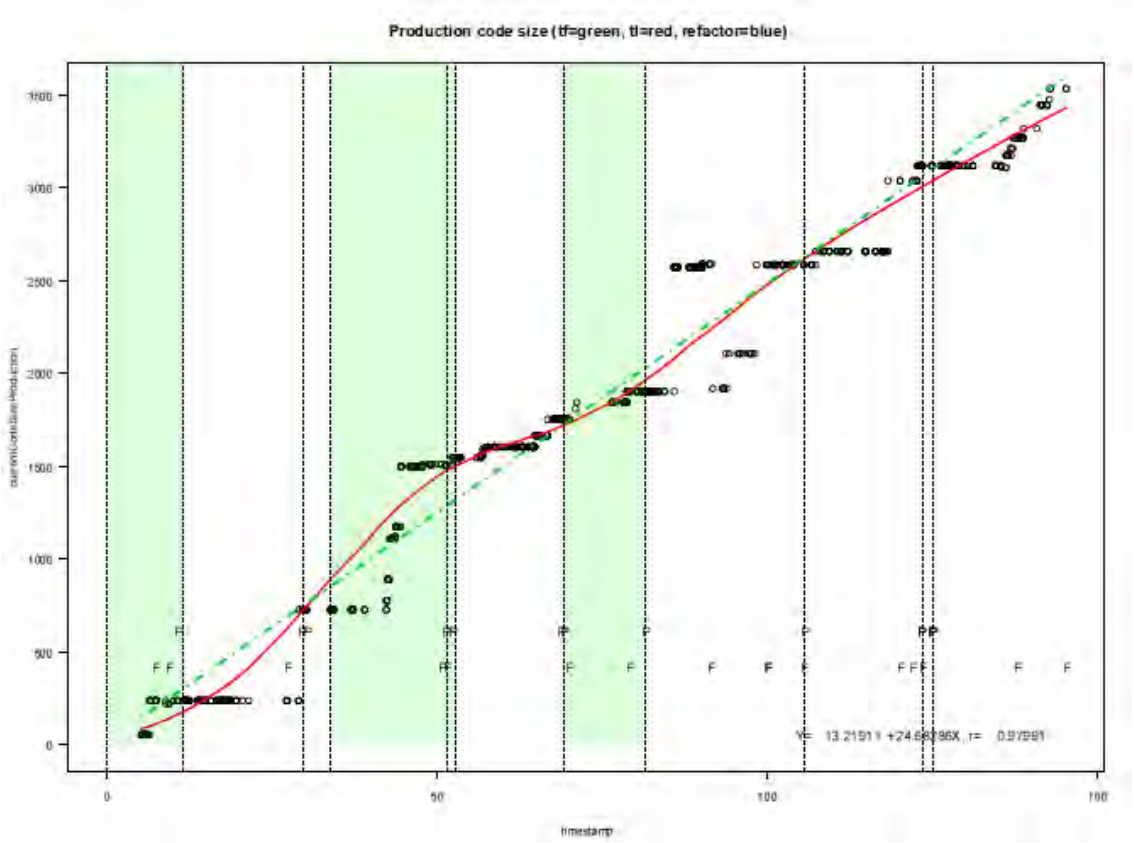

Figure 12: Analysis of Production code size in one case of ACME² experiment (fixed results)

As shown this graph corresponding to the new results, there are actually 12 episodes. We do not know exactly the reason why the older actions have the 12 episodes, but the file *zorroEpisodes* generated by Besouro gives only 4 episodes.

The next case that we want to show is the case done in the first day of the experiment with the indication of generating the code according to ITL. The comparison between the results derived from old and new datasets can be done through the next graphs:

- Old results:

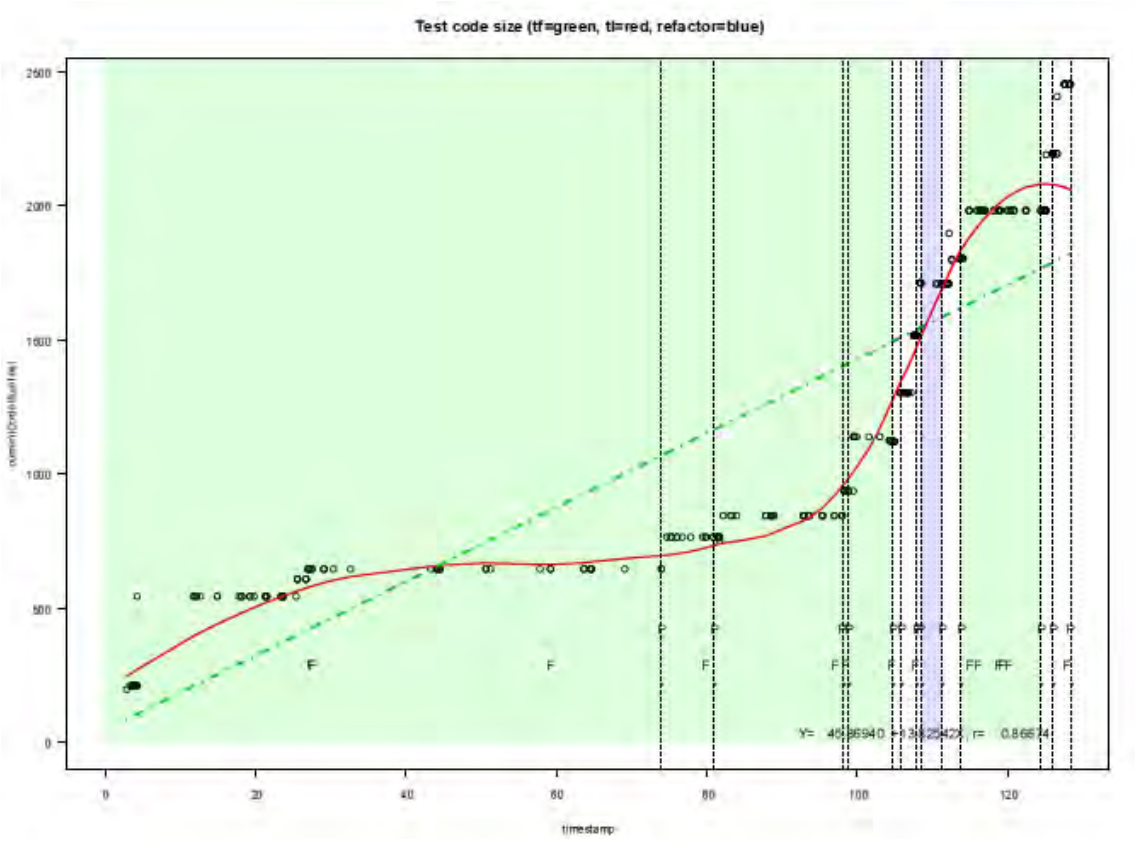

Figure 13: Analysis of Test code size in one case of ACME² experiment (older results)

As we can see, in this experiment the majority of the episodes are Test-First (colour green). This fact is not expected because the instructions was based on ITL. If we take a look into the new results we can see:

- New results:

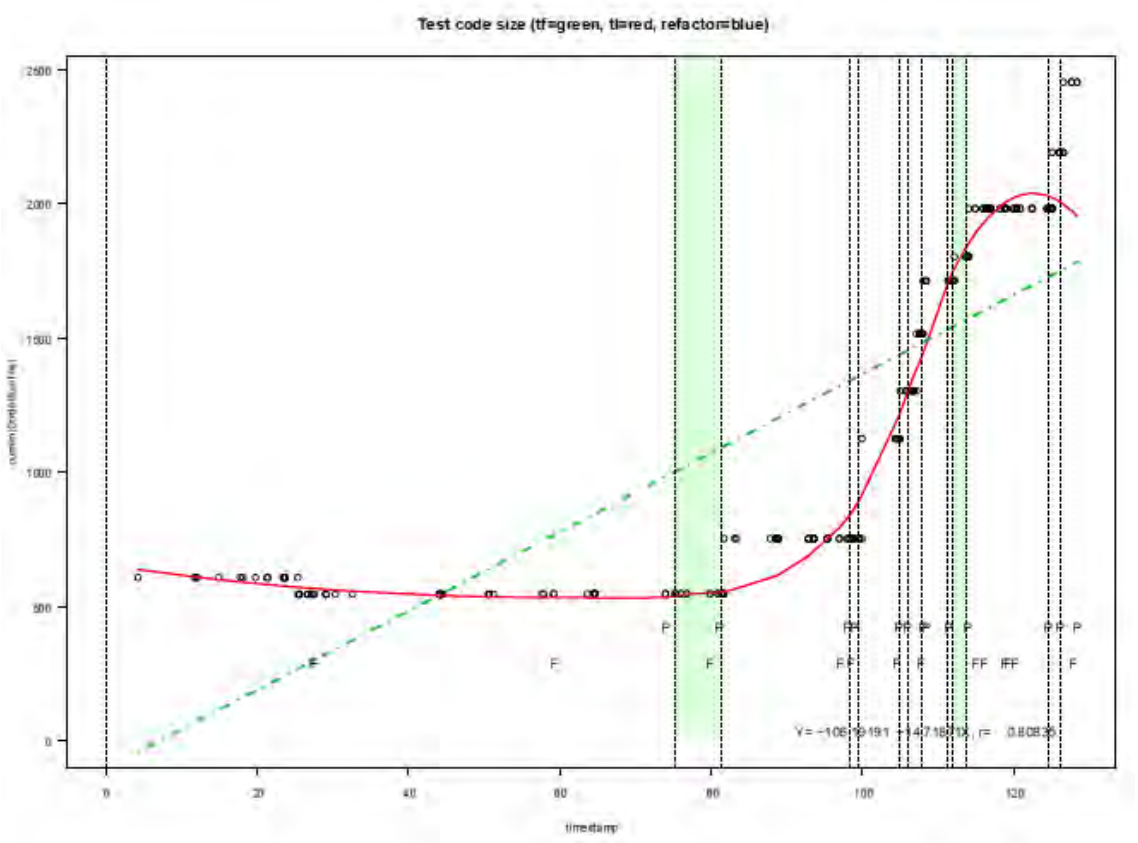

Figure 14: Analysis of Test code size in one case of ACME² experiment (new results)

As illustrated in the graph, there are two episodes of Test-First. It is normal that there are some episodes of Test-First if the developers made some depuration, but not the major part of the episodes. And in that case, if we take a look at the time durations of these episodes it is short. This is in concordance the idea of depuration.

Another case that we want to show corresponds the second day of the experiment. During this day, the instructions of the experiment were creating the code using TDD. So we expect more Test-First than the first day. If we take a look these next in these graphs:

- Old results:

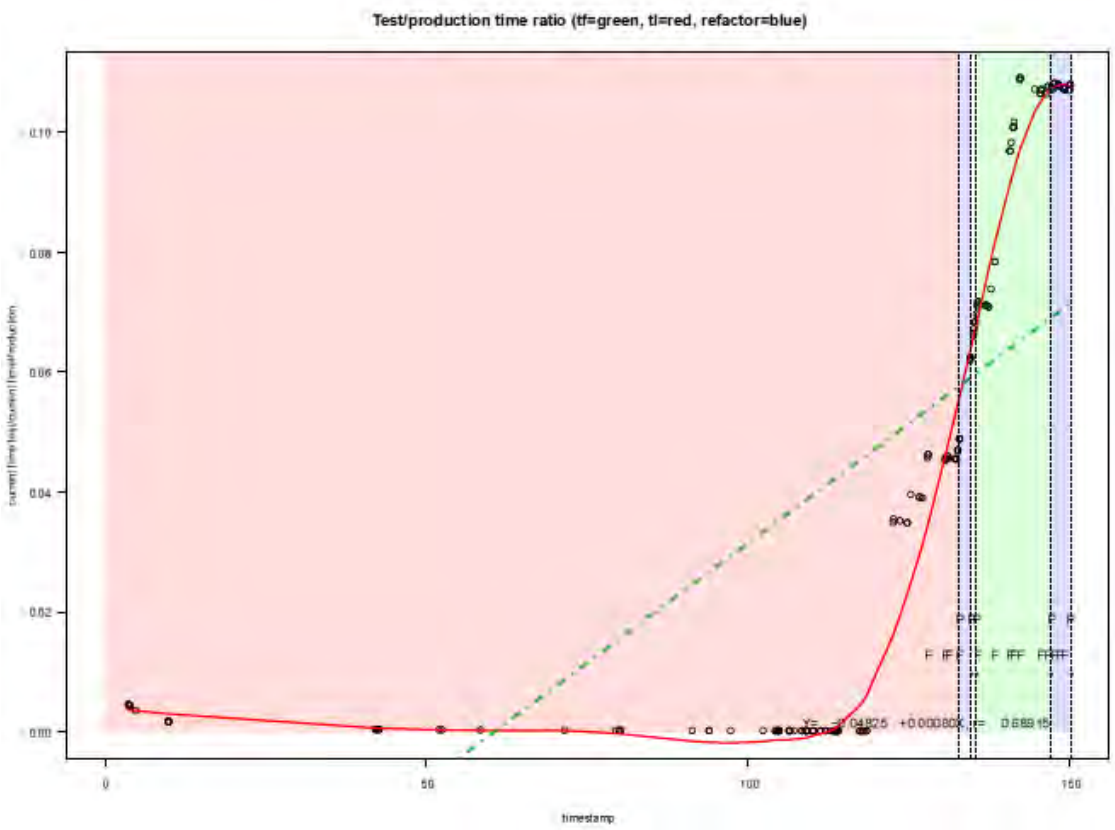

Figure 15: Analysis of Test/production time ratio in one case of ACME² experiment (old results)

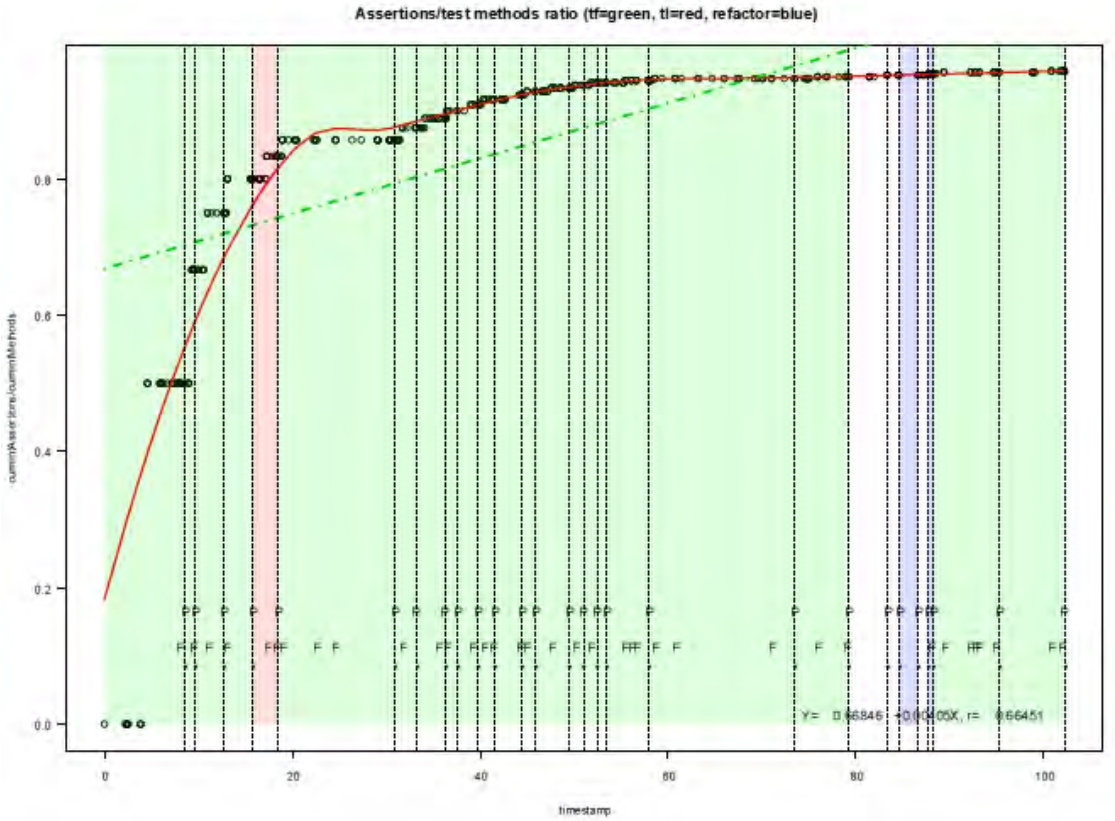

Figure 16: Analysis of Assertions/test methods ratio in one case of ACME² experiment (old results)

As we can see in that graphs, there are two episodes of Test-Last (colour red) which is not normal because the instructions were based on TDD. But we can assume that maybe the developers were wrong and not followed the instructions in these episodes. In order to clarify this, let us, take a look on the fixed graphs:

- New results:

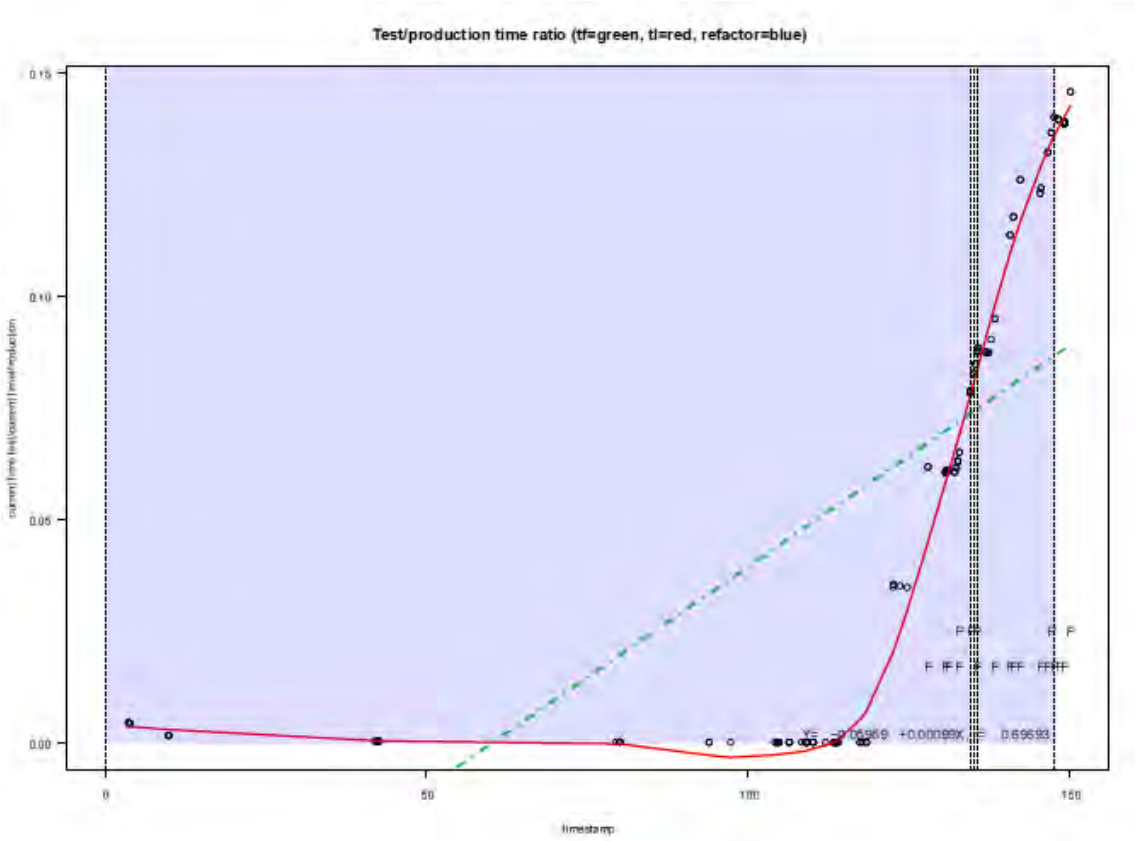

Figure 17: Analysis of Test/production time ratio in one case of ACME² experiment (new results)

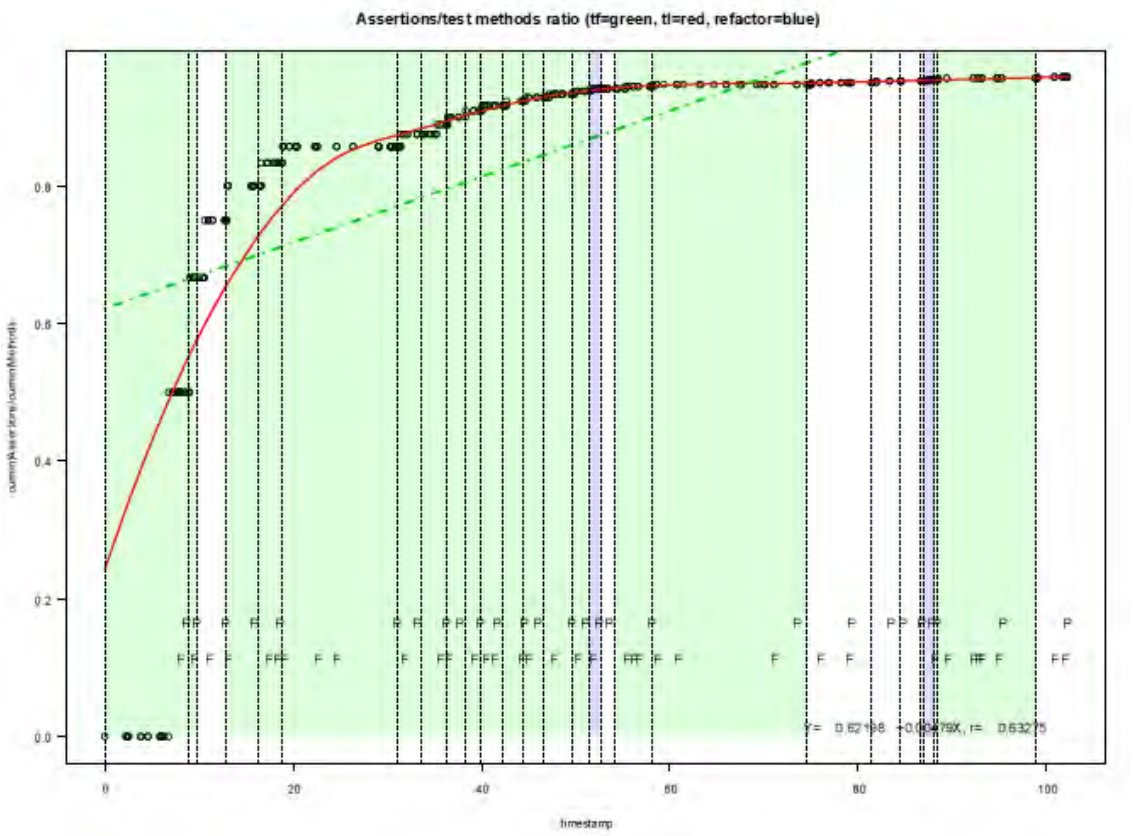

Figure 18: Analysis of Assertions/test methods ratio in one case of ACME² experiment (new results)

We can see in these two cases the episodes that were classified as a Test-Last. Looking into the fixed datasets, these cases changed in different ways, namely, in the first case as a refactor (colour blue) and in the second case as a Test-First (colour green). Is possible that the reason of these cases was the order of the actions done in each case.

We showed the main differences found in the 18 cases that we analysed. The cases not shown are cases where the data are similar, or cases with small differences or cases equivalent to the showed ones. There are differences too in the resulting line of all the graphs, comparing old cases with new cases. This is because in the fixed datasets we added some *EditActions* that were lost when Besouro returned the actions. These differences are notable in all cases but always follow a similar tendency.

6. Related works: Impact of Besouro's episode classification mistakes

As illustrated in this document, the output datasets of ACME² experiment have a lot of errors caused by the functionality Besouro problems shown in previous sections. In fact, Besouro can be used for different types of experiments related to conformance of TDD (test quality, developer's productivity, correctness of TDD process etc...). In the case of ACME², the experiment was done to analyse how developers create some softwares in the steps of TDD. Besouro is a tool that has been used for other experiments of TDD.

In Google Scholar we find another experiment that uses Besouro carried out by Karin Becker, et al. This experiment aims at exploring the features of Besouro for comparing variations of a TDD operational definition, for the purpose of evaluating with respect to TDD conformance criteria from the point of view of developers in the context of programming activities. This experiment was conducted by developers belonging to the Brazilian Agile community.

The questions arisen in this experiment are:

1) Can the conformance of episode categories production, refactoring, test addition and regression be established in isolation, regardless the neighbour episodes?

2) Which types of episodes may result in multiple, conflicting category classification?

3) Is code coverage variation a good metric to distinguish between refactoring and production episodes?

The first question involves comparing the context-sensitive conformance classification of Zorro with another conformance interpretation. The second question is motivated by the category classification component, which relies on the Jess engine. Recall that Jess may output a set of categories for a same episode, based on all rules that are satisfied. Finally, the third question focuses on understanding whether additional criteria could improve the automatic recognition of production and refactoring episodes. To distinguish between

them, Zorro uses rules that, in addition to a specific sequence of actions, take into account code size variation.

The metrics to solve these questions are:

- To solve the first question:

1) Number of times users actively disagreed with the presented assessment (category and/or conformance).

2) Number of times users actively disagreed with Zorro's assessment (category and/or conformance).

3) Number of times users actively disagreed with Besouro's assessment (category and/or conformance).

4) Episode conformance assessment according to users feedback (active disagreement and passive agreement).

5) Episode conformance assessment according to users active disagreement.

To solve the second question:

6) number of episodes classified according to multiple, conflicting categories.

7) number of episodes per conflict found.

Finally, to solve the third question:

8) difference of line/block code coverage, by comparing coverage in the beginning and in the end of an episode.

The results that they obtain are represented in the following tables:

Kata	Nb. of Episodes	Kata Duration	Episode Average Duration (std dev)	M1: Nb. of Active User Disagreements	M2: Nb (%) of Active Disagreements wrt Zorro	M3: Nb (%) of Active Disagreements wrt Besouro
1	30	71m0s	2m22s (3m41s)	11	9 (30%)	9 (30%)
2	39	94m48s	2m24s (3m49s)	16	5 (12,8%)	5 (12,8%)
3	21	76m58s	4m37s (5m21s)	6	0 (0%)	0 (0%)
4	19	72m0s	4m47s (3m25s)	6	6 (31,6%)	6 (31,6%)
5	17	19m35s	1m6s (1m59s)	10	5 (29,4%)	5 (29,4%)
6	10	14m14s	1m25s (1m24s)	0	0 (0%)	0 (0%)
7	28	76m59s	3m43s (3m1s)	0	0 (0%)	0 (0%)
8	8	44m19s	6m32s (8m52s)	2	2 (25%)	2 (25%)
9	14	64m23s	5m36s (8m56s)	3	3 (21,4%)	3 (21,4%)
10	15	21m3s	1m24s (1m49s)	1	1 (6,7%)	1 (6,7%)
11	13	28m37s	2m7s(1m26s)	2	2 (15,4%)	2 (15,4%)
12	9	64m46s	7m5s(7m40s)	3	3 (33,3%)	2 (22,2%)
13	4	16m57s	4m14s (2m58s)	2	2 (50%)	2 (50%)
14	35	87m20s	2m30s (6m54s)	13	3 (8,6)%)	2 (5,7%)
15	4	13m31s	3m23s (1m8s)	0	0 (0%)	0 (0%)
16	9	12m25s	1m23s (1m22s)	3	3 (33,3%)	2 (22,2%)
17	24	41m21s	2m43s (2m38s)	13	8 (33,3%)	6 (25%)
18	10	41m34s	4m9s (2m25s)	1	1 (10%)	1 (10%)
19	31	52m3s	2m41s (2m4s)	17	9 (29%)	9 (29%)
20	7	21m33s	3m5s (2m37s)	3	2 (28,6%)	2 (28,6%)
21	13	45m42s	3m26s (3m52s)	2	2 (15,4%)	2 (15,4%)
Total	**360**	**16h13m**		**114**	**66 (46%)**	**61 (42,2%)**

Table 2: Results to see metrics 1,2,3,4 and 5

Episode category	Nb. episodes	M4: active/passive feedback	
		Conformant	Non-conformant
Production	14	0	14
Test addition	74	63	11
Refactoring	54	40	14
Regression	16	10	6
Total	158	113	45

Table 3: Results to check metric 6

Episode category	Nb. episodes	M5: Active disagreement feedback	
		Conformant	Non-conformant
Production	14	0	5
Test addition	74	0	1
Refactoring	54	12	4
Regression	16	0	1
Total	158	12	11

Table 4: Results to check metric 7

Comparison of measured conformance adherence.

Kata	Programmer (%)	Zorro (%)	Besouro (%)
10	100	100	100
15	100	100	100
13	100	48.3	48.7
2	99.5	100	99.1
3	99.2	100	100
14	95.8	99.4	94.4
8	95.1	100	98.7
5	93.4	89	77.6
12	88.9	100	89
18	88.8	35.3	63.5
17	85.2	72.9	91.9
21	84.5	100	100
1	79.7	100	88.8
19	78.8	100	100
9	72.8	73.4	74.1
16	62.9	49.3	87
4	55.8	87.8	77.6
11	53.5	87.6	59.5
20	39.6	37.9	70.9

Table 5: Results to check metric 8

The results of this experiment are more addressed to evaluate the working functionality of Besouro than to check of TDD method. More precisely, it is focused on evaluating the functionality of the program about different types of developers in different contexts.

They analyse the results of the experiment concluding that the documentation was written informally and most subjects performed the experiment completely unsupervised. It may have influencing aspects such as conformance interpretation, as well as how subject interpreted. They should introduce non-conformance practices during the experiment development.

As derived from previous paragraphs, the authors did not expect their obtained results and they justified them by some possible reasons without knowing that Besouro generates the set of faults presented in this thesis. Our proposed solution can have an important impact on experiments that use Besouro. Our proposal also offers the possibility of repairing the datasets and re-evaluate the results to obtain more accurate conclusions.

7. Conclusions

We started this thesis motivated by the misclassification of episodes made by Besouro and with the aim at fixing it or, at least, improving it. The initial lack of

knowledge about the reason of this misclassification did not assure the success in the solving solution. Even though, we were encouraged with the possibility of, at least, being able to report more details about the problem.

We found that the problem was caused by the fact that the events utilised by Besouro arrive when the developer saves or compiles the code developed. So, we decided to force this saving each time the developer changes the tabs. With this change the actions, are assured to arrive to Besouro in a correct order.

Later on in the work, we found that this was not the only problem. We found that there were inconsistences with the rules of Zorro made. We analysed with special attention all the rules Test-First and Test-Last and then subsequently developed new declaration of the rules that have inconsistencies.

At the last stage of the work, we decided to create a software to fix an available set of results made in a previous experiment and fix these results to analyse well the experiment made. And, finally, we analysed the difference between this new results and the previous ones and see evaluated the difference between them. We found some differences between them, the most important the quantity of conformance in TDD in base on an experiment that you can see in the annex IV of this document.

8. References

[1] F. Shull, G. Melnik, B. Turhan, L. Layman, M. Diep, H. Erdogmus, What do we know about test-driven development?, IEEE Software (Voice of Ev- idence) 27 (6) (2010) 16–19.

[2] H. Munir, M. Moayyed, K. Petersen, Considering rigor and relevance when evaluating test driven development: A systematic review, Information and Software Technology 56 (2014) 375–394.

[3] http://http://www.jessrules.com

[4] http://csdl.ics.hawaii.edu/research/zorro/

[5] http://www.pmoinformatica.com/2012/12/test-driven-development-ventajas-y.html

[6] S. Sørumg˚ard. Verification of process conformance in empirical studies of software development. PhD thesis, Ph. D. thesis, Norwegian University of Science and Technology, 1997.

[7] J. E. Cook and A. Wolf. Toward metrics for process validation. In Software Process, In Proceedings of the Third International Conference on, pages 33–44, 1994.

[8] M. I. Kellner, P. H. Feiler, A. Finkelstein, T. Katayama, L. J. Osterweil, M. H. Penedo, and H. D. Rombach. Software Process Modeling Example Problem. In Software Process Workshop, Proceedings of the 6th International, pages 19–29. IEEE, 1990.

[9] L. Silva and G. Travassos. Tool-supported unobtrusive evaluation of software engineering process conformance. In Empirical Software Engineering, International Symposium on, pages 127–135, 2004.

[10] N. Zazworka, V. Basili, and F. Shull. Tool Supported Detection and Judgment of Nonconformance in Process Execution. In Proceedings of the 3rd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pages 312–323, 2009.

[11] N. Zazworka, K. Stapel, E. Knauss, F. Shull, V. R. Basili, and K. Schneider. Are Developers Complying with the Process: an XP Study. In Proceedings of the 4th ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pages 1–10, 2010.

[12] Y. Wang and H. Erdogmus. The Role Of Process Measurement In Test-Driven Development. In 4th Conference on Extreme Programming and Agile Methods, 2004.

[13] L. Madeyski and L. Sza la. The Impact Of Test-Driven Development On Software Development Productivity — Empirical Study. In Software Process Improvement, pages 200–211. Springer, 2007.

[14] P. Johnson and H. Kou. Automated Recognition of Test-Driven Development with Zorro. In AGILE 2007, pages 15–25. IEEE, 2007.

[15] H. Kou, P. M. Johnson, and H. Erdogmus. Operational Definition And Automated Inference Of Test-Driven Development With Zorro. Automated Software Engineering, 17(1):57–85, 2010.

[16] https://csdl-techreports.googlecode.com/svn/trunk/techreports/2009/09-01/09-01.pdf

[17] http://proceedings.informingscience.org/InSITE2012/InSITE12p165-187Bulajic0052.pdf

[18] https://github.com/brunopedroso/besouro

ANNEX I: Besouro code fixed.

- Besouro code fixed in this document. Attachment: Besouro.zip

ANNEX II: Software to fix datasets code.

- Software to fix datasets code in this document. Attachment: ACME.zip

ANNEX III: Code R to analyse results.

- Code R to analyse results in this document. Attachement: ProcessesBesouro.R

ANNEX IV: Comparison datasets ACME².

In ACME² experiment we received 18 results of the experiment. In this annex, we compare the results between the original results and the fixed results and evaluate the conformance of these results in TDD. The experiment was separated in two days. The first day the developers have to develop the code using the methodology ITL. In the second day the developers have to develop the code using TDD.

The results of the experiment that we are interested in comparing are shown in the next tables:

-First day (ITL)

Nº exp	Name	Nº episodes	Test-First	Non-Test-First	Average TDD
1	MR – Old	13	9	4	70%
	MR – New	13	2	11	15%
2	MR – Old	-	-	-	-
	MR – New	-	-	-	-
3	MR – Old	-	-	-	-
	MR – New	-	-	-	-
4	MR – Old	11	1	10	9%
	MR – New	11	2	9	18%
5	BSK – Old	9	1	4	11%

	BSK – New	9	2	1	0%
6	BSK – Old	-	-	-	-
	BSK – New	22	1	21	4%
7	BSK – Old	-	-	-	-
	BSK – New	-	-	-	-
8	MR – Old	17	1	16	6%
	MR – New	17	1	16	6%
9	BSK – Old	-	-	-	-
	BSK – New	-	-	-	-
10	BSK – Old	9	2	7	22%
	BSK – New	9	0	9	0%
11	BSK – Old	-	-	-	-
	BSK – New	-	-	-	-
12	MR – Old	9	3	6	33%
	MR – New	9	1	8	11%
13	MR – Old	21	4	17	19%
	MR – New	21	2	19	10%
14	BSK – Old	-	-	-	-
	BSK – New	-	-	-	-
15	BSK – Old	-	-	-	-
	BSK – New	31	3	28	0%
16	MR – Old	3	0	3	0%
	MR – New	3	1	2	33%

17	BSK – Old	-	-	-	-
	BSK – New	-	-	-	-
18	MR – Old	2	0	2	0%
	MR – New	7	0	7	0%

Table 6: datasets of Test-First the first day experiment

- Second day (TDD)

Nº exp	**Name**	**Nº episodes**	**Conformance TDD**	**Non-Conformance TDD**	**Average TDD**
1	BSK – Old	31	12	19	38%
	BSK – New	31	10	21	0%
2	BSK – Old	19	5	14	26%
	BSK – New	19	5	14	26%
3	BSK – Old	-	-	-	-
	BSK – New	-	-	-	-
4	BSK – Old	16	4	12	25%
	BSK – New	16	3	13	19%
5	MR – Old	3	2	0	67%
	MR – New	14	5	11	36%
6	MR – Old	6	3	3	50%
	MR – New	20	7	13	35%
7	MR – Old	34	11	23	32%
	MR – New	34	8	26	23%
8	BSK – Old	45	14	31	31%

	BSK – New	45	10	35	22%
9	MR – Old	13	9	4	70%
	MR – New	13	9	4	70%
10	MR – Old	23	12	11	52%
	MR – New	23	6	17	26%
11	MR – Old	5	1	4	20%
	MR – New	5	2	3	40%
12	BSK – Old	23	12	11	52%
	BSK – New	88	5	83	5%
13	BSK – Old	22	8	14	36%
	BSK – New	22	6	16	27%
14	MR – Old	27	22	6	81%
	MR – New	27	19	8	70%
15	MR – Old	19	8	11	42%
	MR – New	19	6	13	31%
16	BSK – Old	17	7	10	41%
	BSK – New	17	7	10	41%
17	MR – Old	15	6	9	40%
	MR – New	15	5	10	33%
18	BSK – Old	12	9	3	75%
	BSK – New	12	6	6	50%

Table 6: datasets of Test-First the first day experiment

We observe some differences between the results new and old as expected. The first day corresponds to an experiment of ITL so, the quantity of Test-First has to be a smaller value than in the second day. This is true for the two cases,

old and new. But, with the datasets fixed, the majority of cases are less than the old databases. It is normal that there are some Test-First in this experiment because is possible to use Test-First to depurate the code. If we look into the second day, it consists in an experiment of TDD so, the quantity of Test-First has to be bigger, and it is true. But just like the first experiment, the fixed results are smaller than old datasets. This is because we made changes in the rules of Test-First and these results in limiting the episodes that meet the results of the rules of Test-First.