Test Case Prioritization Using PriorJ

Everton L. G. Alves[1], Samuel T. C. Santos[1], Patrícia D. L. Machado[1], Tiago Massoni[1]

[1]SPLab/ Federal University of Campina Grande, Brazil
everton@copin.ufcg.edu.br, samuel.santos@ccc.ufcg.edu.br,
{patricia, massoni}@computacao.ufcg.edu.br

Abstract. *Test case prioritization is aimed at reordering test cases during regression testing with the goal of achieving some testing objective more quickly. While potential benefits are widely recognized, only a few tools to support the prioritization process have been reported in the literature. PriorJ is a tool devoted to test case prioritization of JUnit test cases. The main goal is to provide support to the use of different techniques of prioritization during regression testing as well as the investigation and evaluation of new techniques. This paper presents an overview of the challenges to be faced and requirements to be met when automating test case prioritization, the main features of PriorJ and also discusses 3 scenarios in which the tool can be applied in practice. A case study illustrates the applicability of the tool.*

1. Introduction

Software modifications are very common during development and maintenance. These modifications are performed in order to improve some software internal aspects (refactoring changes), or to adding or removing functionalities (evolutionary changes). In agile or less strict software development processes (e.g. XP [Beck and Andres 2004]) those changes are even more common due to the lacking of well-founded planning and designing phases. Independently of the scope of application, the tasks evolved in a software modification edit can be very hard to manage [Alves et al. 2013].

In practice, developers usually combine the execution of software changes with the use of regression testing [Leung and White 1989]. A regression test suite is a set of test cases that reflects the stable behavior of a system under test (SUT). More formally we can say that given two programs P and P' (the program P after a modification), the regression suite can reveal behavior differences between P and P'. This practice has been extensively used with success in the industry [Onoma et al. 1998, Harrold and Orso 2008]. However, as the regression suite evolves, regression testing activities can be very costly and time consuming. Studies (e.g. [Leung and White 1989, Harrold and Orso 2008]) indicate that regression testing activities can spend almost half of the maintenance budget.

Test case prioritization [Rothermel et al. 1999] is a strategy that deals with the problem of reducing the costs related to regression testing by reordering the regression test cases aiming to achieve a certain testing goal more quickly (e.g. the improvement of the fault detection rate). In this sense, a good prioritized suite would place the test cases that detect the system faults in the prioritized suite top positions.

Several test case prioritization techniques have been proposed in the literature. Singh at al. [Singh et al. 2012], for example, performed a literature review in order to map the state-of-art about test case prioritization. They localized 106 prioritization techniques (e.g. [Rothermel et al. 1999, Srikanth et al. 2005, Kim and Porter 2002]) which

they classified among eight categories: *Coverage Based*, *Modification-based*, *Fault-based*, *Requirement-based*, *History-based*, *Genetic-based*, *Composite approaches*, and *Other approaches*.

Due to the great variability of techniques available, in practice, several challenges need to be faced for incorporating prioritization practices into real projects:
1) *Automation*. To make prioritization a cost-effective activity, automation is required. Most techniques make use of data such as coverage and object binding. Moreover, test cases need to parsed and analyzed along with all infra-structure information related to each test case execution that need to be extracted from code.
2) *Choice of a suitable technique*. Prioritization techniques may behave differently depending on aspects they are dealing with (e.g. SUT characteristics, suite characteristics, type of code modification, etc) [Alves et al. 2013]. It is usually difficult to decide which technique is more suitable, particularly due to the lack of information on which specific aspects the success/failure of a technique relies on.
3) *Data management*. After running each prioritization, several artifacts and data are usually generated (e.g. prioritized test suites, coverage data, suites' rate of fault detection, etc). Those elements, if stored correctly, could be useful to help the tester to perform a better evaluation about the quality of the suites, and also to allow him to take more informed decisions. But, usually, taking control of these artifacts is a very hard and tedious task [Do et al. 2005]. This fact tends to get worst after every prioritization running as the number of artifacts to be managed increases.

Rocha *et al.* [Rocha et al. 2012] present the initial efforts to address those problems by proposing the PriorJ tool. In its initial version, PriorJ enabled the running of a set of prioritization techniques for Java systems that have their regression test cases implemented according to the JUnit framework. The main contributions of this paper are: i) the definition of different situations in which we summarize the main activities that a prioritization tool/environment has to implement in order to support the main needs related to test case prioritization; and ii) a set of extensions for PriorJ developed in order to transform it into a more complete prioritization environment for the Java/JUnit world. A case study demonstrates the use of some of the new features of PriorJ.

The paper is structured as follows. Section 2 describes the basic concepts needed for making the paper self-contained. In Section 3 we describe activies that can be applied in 3 scenarios that prioritization tools have to deal. Section 4 describes the new architecture of PriorJ. In Section 5 we present an illustrative case study. Section 6 comments on related work. Finally, Section 7 presents concluding remarks along with points for further research.

2. Background

2.1. Test Case Prioritization

The goal of the prioritization techniques is to reschedule regression test cases in a new execution order that will allow the satisfaction of a given test objective earlier (e.g. the early finding of faults). The test case prioritization problem was formally defined by Rothermel [Rothermel et al. 1999] as follows:

Given: T, a test suite; PT, the set of permutations of T, and f, a function from PT to real numbers.

Problem: Find $T' \in PT$ such as $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

where PT represents the set of possible orderings of T, and f is a function that gets the best possible values when applied to any ordering.

In general, prioritization techniques differ from each other by the prioritization heuristics used, and by the context they work. But, due to their simplicity and also to the good results that they produce in general, coverage-based techniques are the most used in practice [Rothermel et al. 1999]. Among the most important coverage-based technique we can list the most traditional ones [Rothermel et al. 1999]: *Total Statement Coverage* (TSC), *Total Method Coverage* (TMC), *Additional Statement Coverage* (ASC), *Additional Method Coverage* (AMC). The TSC, for example, prioritizes the test cases according to the following process: the first test case to be chosen for composing the prioritized suite is the one which covers the major amount of statements from the SUT code; the second test case to be chosen will be the one which covers the second major amount of statements of the SUT code, and so on.

2.2. PriorJ

The PriorJ tool [Rocha et al. 2012] is an open-source project developed in order to allow the execution of different prioritization techniques for Java/JUnit systems. PriorJ automatizes the traditional prioritization techniques by providing an integrated coverage analyzer strategy. This strategy uses AspectJ files in order to run the JUnit regression test suite and collect the coverage traces in both statement and method level [Rocha et al. 2012]. Those traces are recorded following a certain pattern that can be directly used for prioritization purposes. The main features provided by PriorJ are: i) test coverage execution; ii) test case prioritization according to traditional techniques (TSC, TMC, ASC, AMC, Random); and iii) generation of directly executable Java artifacts related to the prioritized test suites. PriorJ was designed in a way that it can be easily extendable. Its module and interfaces-based design imposes that for including a new prioritization technique the developer must program a single new Java class that will be responsible for implementing the new prioritization algorithm.

3. The Prioritization Activities

In this section we present different scenarios/situations where test case prioritization can be applied and discuss issues to be addressed related to the running and management of prioritization activities in practice. We believe that every prioritization tool should implement at least a subset of those activities.

Each scenario is related to one of the challenges described in Section 1: i) when the tester is interested in running a single prioritization (Figure 1); ii) when the tester need to decide which prioritization technique is more adequate for his context (Figure 2); and iii) when the tester needs to manage the artifacts from previous prioritizations (Figure 3). All scenarios are modeled by using UML activity diagrams.

The original PriorJ tool [Rocha et al. 2012] implemented only part of the first prioritization scenario (Figure 1), activities like control version of prioritization artifacts, and the calculation of evaluative metrics were not available. A new version of PriorJ was created covering all scenarios (details in Section 4). Thus, we can see the new PriorJ as a proof of concept that the scenarios to be presented are implementable and that they

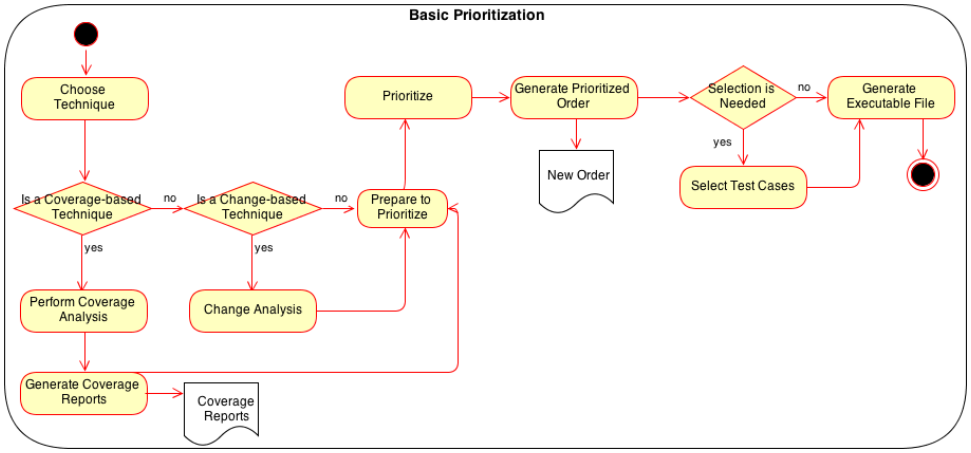

Figure 1. Basic prioritization.

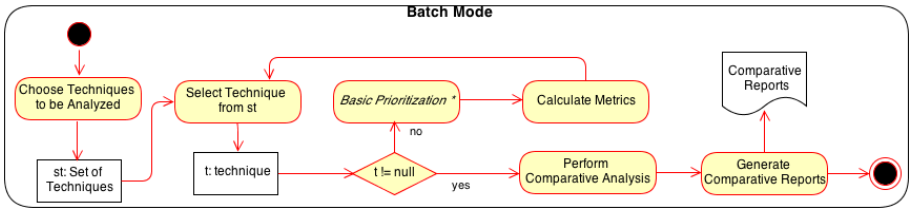

Figure 2. Batch prioritization.

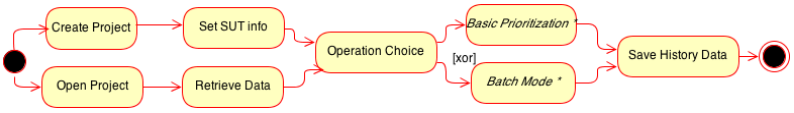

Figure 3. Data management.

help the development of useful prioritization tools. The new version of PriorJ is already available[1].

Figure 1 gives an overview of the basic activities needed as part of a basic prioritization process. The first feature enables the user/developer/tester to choose which prioritization technique is going to be applied (*Choose Technique* activity). Depending on the type of the chosen technique a different set of actions needs to be activated. If the user choose a coverage-based prioritization technique, a coverage analysis needs to be performed (*Perform Coverage Analysis* activity). This analysis can be done by using either a static or dynamic strategy. In order to automatically perform the coverage analysis, the prioritization tool may: i) implement its own coverage analysis. PriorJ, for example, uses execution traces in this sense; or ii) reuse an external coverage analysis tool (e.g. Emma[2]). As coverage data is one of the most used aspects for evaluating the quality of a test suite [Zhu et al. 1997], a good prioritization tool would also produce reports that could be used for this purpose. This coverage report can also give to the testers the first evidences whether a coverage-based prioritization would be adequate or not in a specific project.

[1]https://sites.google.com/a/computacao.ufcg.edu.br/priorj/
[2]http://emma.sourceforge.net/

If a change-based prioritization technique is chosen, the prioritization tool must perform a code change analysis (*Change Analysis* activity). Most of the change-based prioritization techniques (e.g. [Srivastava and Thiagarajan 2002]) requires information about which parts of the code were changed in order to perform their prioritization. PriorJ, for example, have a dedicate module that discovers the change methods and statements, considering two sequential versions of a Java system.

After performing the basic analysis, the data from this analysis must be summarized in a way that it can be directly manipulated by the prioritization algorithms (*Prepare to Prioritize* activity). The *Prepare to Prioritize* activity is also responsible for collecting any data needed for prioritization, if a different type of technique is chosen (not coverage nor change-based). This data can be collected directly from the project and/or from interactions with the user.

The *Prioritize* activity is responsible for running the prioritization algorithms. As most of the needed data were manipulated in the previous activities, the running of the prioritization algorithms should be a simple task. After the prioritization, a set of output artifacts must be generated: i) *the prioritization order (Generate Prioritized Order* activity). In some situations the user might want just to know the new test case order aiming at performing a manual inspection/selection; ii) *a selected suite*. In projects in which the regression test suite size is huge, developers usually are able to run just part of the prioritized suite. In the *Select Test Cases* activity the user informs the percentage of the prioritized test suite that he needs, and the tool selects the top test cases from the prioritized suite according to this value; and iii) *the executable prioritized suite* (*Generate Executable File* activity). In order to give to the user an artifact that can be directly used in his daily work, in this activity an executable file must be created. This file contains the code that calls directly the test cases following the new order proposed by the prioritization algorithm. PriorJ, for example, creates a Java class that calls the test cases by using Java reflection commands and have the JUnit infrastructure as testing arbiter.

When the tester does not have enough information about the project or the regression suite (e.g. legacy systems) it is hard to decide which prioritization technique is more suitable for his context. In this situation, a good prioritization tool would help him in this sense. For that, we defined the second prioritization scenario (Figure 2). In this scenario several prioritization algorithms are run in a batch mode and metrics from each prioritization are collected and presented to the user. This strategy can be applied by using artifacts from the real project (e.g. SUT and regression suite) or by using a small set of test cases and emulators, as a trial test. We can summarize this scenario as follows: first, the user must choose which prioritization techniques he wants to test (e.g. the coverage-based ones) – *Choose Techniques to be Analyzed* activity. For each technique the prioritization is performed independently (*Select Technique from st* activity and *Basic Prioritization* – Scenario 1) and metrics are calculated. The main metrics for evaluating the quality of a prioritized test suite are the *F-Measure* [Jiang et al. 2009] (the position of the first test case that reveals a fault) and the *APFD* [Rothermel et al. 1999] (the suite rate of fault detection). After that, the tool must be able to perform a comparative analysis that will help the user to visualize the most suitable technique for his context (*Perform Comparative Analysis* and *Generate Comparative Reports* activities). PriorJ generates reports that give numeric (tables) and visual (charts) comparison of the prioritization results.

Finally, in order to address the challenge related to the management of the prioriti-

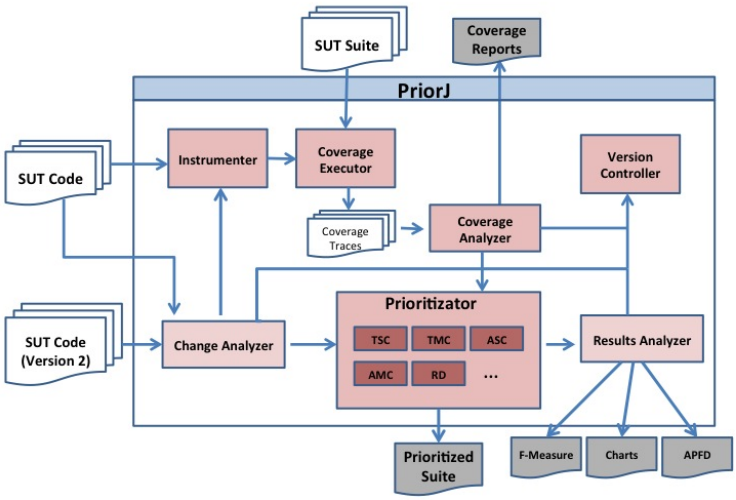

Figure 4. PriorJ Architecture.

zation data, we designed the third prioritization scenario – Figure 3. For that, we used the idea of version control and history management. Before allow any task, the prioritization tool must attach the session in use to a specific prioritization project (*Create Project* activity). In this project the information related to the SUT, test cases, faults, and versions are collected and stored (*Set SUT info* activity). After any prioritization those information are updated and properly stored as a new version (*Save History Data* activity). Similarly to code version control tools, the prioritization tools must be able to retrieve the data related to each prioritization performed in the past (*Open Project* and *Retrieve Data* activities). There are several prioritization techniques that use history data as prioritization factor (e.g. [Kim and Porter 2002]).

4. PriorJ Extension

In order to transform the PriorJ tool in a prioritization environment that fulfills all requirements established by the scenarios defined in Section 3, we had to implemented several extensions. The previous modules were evolved, new modules were developed, and prioritization techniques were included. The original PriorJ tool [Rocha et al. 2012] enabled only the automation of a few prioritization techniques (mainly coverage-based). The new PriorJ implements by default seven different prioritization techniques from different categories (TSC, TMC, ASC, AMC, Random, Change Blocks [Srivastava and Thiagarajan 2002], and RBA [Alves et al. 2013]). Beyond that, PriorJ now has a built-in support for several new features: i) project management; ii) prioritization experimentation; and iii) generation of several output artifacts (e.g. coverage report, prioritized suite, metrics, etc).

Figure 4 depicts, in a high level way, an overview of PriorJ's new architecture. The rectangular elements represent PrioJ modules, the arrows represent the relationship between the modules, and the grey elements represent PriorJ external outputs. In a few words, we can describe the main functionality of each module as follows:

Instrumenter. Module responsible for performing the SUT instrumentation and make it ready for the coverage analysis. For that, a temporary copy version of the SUT project is

created and marks are added in the temporary code. Those marks are patterns that are only known by the *Coverage Analyzer* module and are used during the test running. The *Instrumenter* module is accessed only when the user choose a coverage-based prioritization technique to be run.

Coverage Executor. Responsible for executing the test running and for generating the data to be addressed during the coverage analysis. For that, it uses the instrumented code and a set of AspectJ files that enables the storage of data related to any possible coverage of statements and/or methods. This module produces as output artifacts a set of log files that contains the data related to the coverage traces.

Coverage Analyzer. In this module, the files produced by the *Coverage Executor* module are read and manipulated in order to generate the coverage reports files and the coverage information needed for prioritization.

Change Analyzer. Responsible for identifying, from two versions of a Java program, which code parts are different. For that, this module uses an investigation based on the comparison of the model representation of Java classes. The output of this module is the set of methods and statements that were modified. This module will be run only when a change-based prioritization technique is selected by the user.

Prioritizator. This is the core module of PriorJ. This module contains the implementation of the prioritization algorithms/techniques/approaches. Each prioritization technique is implemented by a single Java class that follows a basic interface of methods. This interface of methods was designed in order to establish a pattern to be followed and consequently make it easier the adding of new prioritization techniques to PriorJ. As output of this module we have: i) a file containing the prioritized sequence of test cases; ii) an executable Java class that enables the prioritized suite execution; and iii) the data to be used for calculating the evaluation metrics (module *Results Analyzer*).

Results Analyzer. Responsible for calculating the metrics related to the quality of the prioritized suites and also to show them to the user in a easier to analyze way (e.g. charts). This module is also responsible for selecting the test cases, if requested by the user.

Version Controller. Responsible for implementing the version control mechanism. For that, it manages the creation of a prioritization project, and data storage and update.

5. Case Study

In order to demonstrate the applicability of the concepts proposed in Section 3, and to glimpse the potential that PriorJ's new features can bring on helping the conduction of prioritization activities, a case study was performed. For this case study, we used a real open-source project – JMock[3] – a library developed for helping the building of unit tests with Mock objects. The JMock project has about 5 KLOC, 504 JUnit test cases, and a test coverage of 92%.

For our study, we manually generate a faulty version of JMock's code. This version contains three faults related to the wrong application of refactoring edits from the catalog proposed by Fowler[4] (*Add Parameter*, *Pull Up Field* and *Move Method*). Those edits ended up by making 12 JMock test cases fail. Then, we used PriorJ, in the batch

[3]http://jmock.org/

[4]http://www.refactoring.com/catalog/

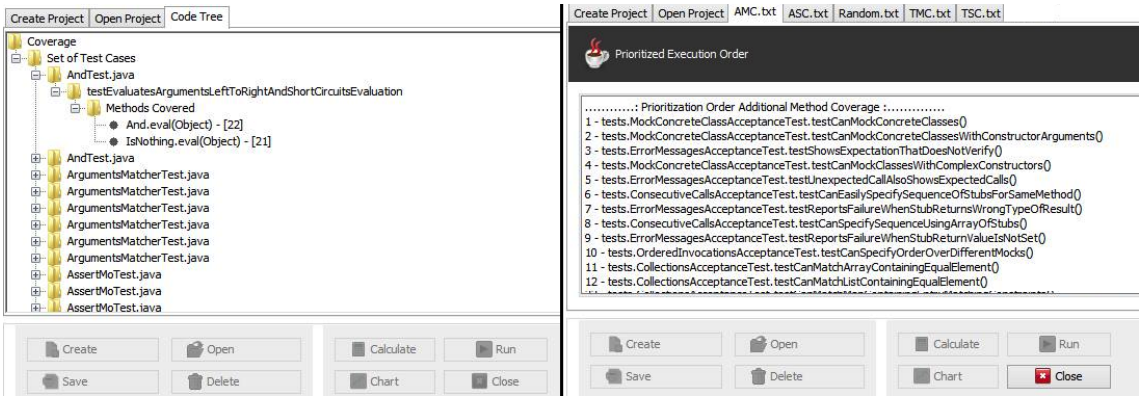

Figure 5. Examples of PriorJ reports.

mode, in order to help us to decide whether the coverage-based technique (TSC, TMC, ASC, AMC) and the *Random*) are effective to discover those faults.

Figure 5 (left side) shows part of one of the coverage reports generated by PriorJ. We can see the coverage analysis code tree representation in which, for each test case, it is shown the covered SUT methods and statements. This report can help the fault debugging in the sense that the tester can see which elements the failed test cases exercised. Those elements are usually more likely to be the source of the faults. In Figure 5 (right side) we can see the prioritization order proposed by techniques run in the batch mode.

Figure 6 shows the comparative APFD chart in which we can see the evolution of the suites fault localization according to each technique. As we can see, for this specific scenario and project, all techniques produced very similar results (similar APFD areas). In fact, the *Random* technique produced very similar results when compared with the coverage-based ones. As the Random prioritization is less expensive, this is probably the most suitable one for this context. Nevertheless, chart results also point to the need for more effective techniques.

It is important to add that tools like PriorJ may also help the conduction of scientific researches. PriorJ, for example, has been already used in evaluative prioritization empirical studies (e.g. [Alves et al. 2013]), helping to decrease the difficulties related to conduct comparative evaluations in this field.

6. Related Work

Although test case prioritization is a widely studied topic, to the best of our knowledge, there is no academic or commercial tool that covers all three challenges related to prioritization automation (Section 1). The available prioritization tools usually implement only a single prioritization technique and for very strict environments. They are usually proof of concept tools (e.g. [Srivastava and Thiagarajan 2002, Ma and Zhao 2008] that were developed in order to present a new prioritization algorithm. Thus, the PriorJ tool, and the ideas related to the scenarios proposed in Section 3, are unique in the sense that: i) they cover the key problems that a tester deals when he want to incorporate prioritization in his project; ii) they deal with practical issues that are usually neglect by academic papers; iii) they can be used as a guidance for the development of other prioritization tools.

We can list some important prioritization tools. Echelon

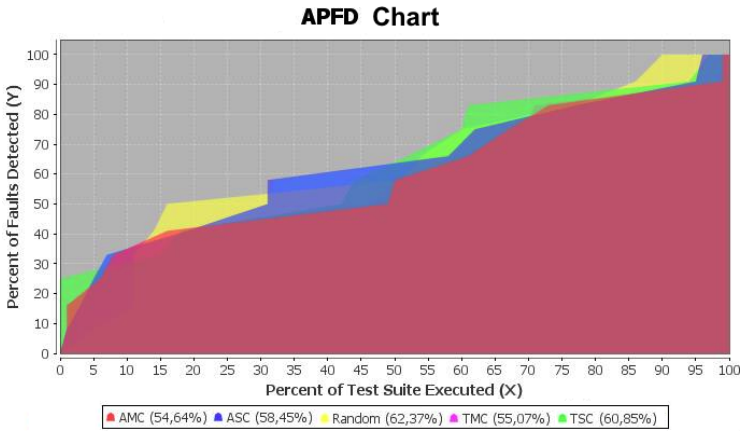

Figure 6. PriorJ APFD chart.

[Srivastava and Thiagarajan 2002], for example, is a tool that automatizes the *Change Blocks* prioritization technique. This tool implements a change-based analysis for Java systems by using binary comparison. Apros [Ma and Zhao 2008] is a tool that prioritizes Java test cases according to their testing-importance for each module (e.g. method) covered. Comparing those tools to PriorJ we can see the great potential that a more complete prioritization environment can bring to the development/testing context. Those tools generate just new test case execution orders. Then, the tester has to manually reorder the test cases and interpret their results, which are error prone tasks. PriorJ makes the tester's work easier and safer by automatizing the generation of execution files and helpful reports. Besides PriorJ, none of the tools give any help to decide which prioritization technique is more suitable for a specific context.

7. Concluding Remarks

This paper discuss the challenges related to the automation of prioritization activities in practice and define scenarios in order to help the development of prioritization tools/environments that aim to addressing those challenges. The scenarios are related to the main activities that a good prioritization tool have to deal in real projects during the development/maintenance phase. The PriorJ tool was extended in order to make it implement all activities defined in the scenarios. This tool has helped the conduction of both software development and prioritization researches. A case study was conducted to illustrate the applicability of the ideas proposed. As further work, we plan to conduct an empirical study investigating the effort needed for applying the main ideas addressed in this paper. Also, we intend to create an Eclipse plug-in version of PriorJ and build its benchmark.

8. Acknowledgments

This work was supported by CNPq grants 484643/2011-8 and 560014/2010-4. Also, this work was partially supported by the National Institute of Science and Technology for Software Engineering[5], funded by CNPq/Brasil, grant 573964/2008-4. First author was also supported by CNPq.

[5]www.ines.org.br

References

Alves, E. L. G., Machado, P. D. L., Massoni, T., and Santos, S. T. (2013). A refactoring-based approach for test case selection and prioritization. pages 93–99.

Beck, K. and Andres, C. (2004). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435.

Harrold, M. J. and Orso, A. (2008). Retesting software during development and maintenance. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 99–108. IEEE.

Jiang, B., Zhang, Z., Chan, W., and Tse, T. (2009). Adaptive random test case prioritization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 233–244. IEEE.

Kim, J. and Porter, A. (2002). A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 119–129. IEEE.

Leung, H. K. and White, L. (1989). Insights into regression testing [software testing]. In *Software Maintenance, 1989., Proceedings., Conference on*, pages 60–69. IEEE.

Ma, Z. and Zhao, J. (2008). Test case prioritization based on analysis of program structure. In *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*, pages 471–478. IEEE.

Onoma, A. K., Tsai, W.-T., Poonawala, M., and Suganuma, H. (1998). Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86.

Rocha, J., E.L.G, A., and Machado, P. (2012). Priorj - priorizacao automatica de casos de teste junit. *Proceedings of Third Brazilian Conference on Software: Theory and Practice (CBSoft) - Tools Section*, 4:43–50.

Rothermel, G., Untch, R., Chu, C., and Harrold, M. (1999). Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE.

Singh, Y., Kaur, A., Suri, B., and Singhal, S. (2012). Systematic literature review on regression test prioritization techniques. *Special Issue: Advances in Network Systems Guest Editors: Andrzej Chojnacki*, page 379.

Srikanth, H., Williams, L., and Osborne, J. (2005). System test case prioritization of new and regression test cases. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10 pp.–.

Srivastava, A. and Thiagarajan, J. (2002). Effectively prioritizing tests in development environment. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 97–106. ACM.

Zhu, H., Hall, P. A., and May, J. H. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427.