SELF: an Easy Way to Perform Acceptance Testing

Pedro Oliveira[1], **Denise Costa**[1], **Matheus Souza**[1], **Pedro Santos Neto**[1]

[1]EASII - DC - UFPI - Teresina - PI - Brazil

{petrus.cc, denise.alvesss, matheusmmcs}@gmail.com, pasn@ufpi.edu.br

Abstract. *Tests are useful in many aspects related to quality assurance. A special aspect is the software validation by customers. This task does not require knowledge in programming languages or special technologies, because the customers should verify only the correctness of business rules. In this paper, it is proposed a tool that allows the creation of tests that can be used by developers during the software construction and even by the customers during the acceptance of the developed work. The tool saves time during testing activities through the combination of preexisting tools adapted for operation in a new method for Automated Acceptance Testing.*

Resumo. *Testes de Software são úteis em muitos aspectos relacionados à garantia de qualidade dos sistemas desenvolvidos. Um aspecto especial de tais testes é a validação do software pelo cliente. Esta tarefa não requer conhecimentos em linguagens de programação ou tecnologias especiais, porque os clientes devem verificar apenas se as regras de negócio estão corretas. Este artigo apresenta uma ferramenta que permite a criação de testes que podem ser utilizados por programadores durante a construção do software e ainda por clientes durante a aceitação do trabalho desenvolvido. A ferramenta otimiza o tempo gasto durante as atividades de teste através da combinação de ferramentas pré-existentes, adaptadas para operação em uma nova abordagem para testes de aceitação automatizados.*

1. INTRODUCTION

Acceptance Test-Driven Development (ATDD), Behavior-driven software development (BDD), Specification by Example are different names for techniques which have in common the aim of helping software developers understand what the other stakeholders want and help the latter describe what they want [Hanssen and Haugset 2009, Sauvé et al. 2006]. Using these techniques all the stakeholders collaborate to shed clarity and gain shared understanding before software coding starts.

ATDD and the other approaches, in some sense, is an advanced practice in the domain of Test-Driven Development (TDD), and, similarly to unit tests, acceptance tests can also be automated. Despite all the advantages of acceptance testing, the execution of this kind of testing in a manual way is, in most cases, tedious, expensive and time consuming [Hanssen and Haugset 2009]

With Automated Acceptance Test-Driven Development (AAT) not only the first development steps are considered but also the test regression needed due to development interactions and requirement changes. Unfortunately a common scenario in software de-

velopment is the use of tools such as Selenium[1] in the context of testing web-based applications, after their development, and FitNesse[2] in the context of understanding the requirements, but not in a truly integrated way with testing.

This work demonstrates that these important contexts can be integrated. This integration brings in important benefits (e.g. reduction of effort). But since these practices are not yet in wide spread we believe that our integrated approach will foster their adoption and help organizations in enhancing their quality.

Despite the advantages of Acceptance Testing tools use, the generation of test cases in the suitable format is hard in terms of effort. However, the generation of a test case using a record & playback tool is simple. The purpose of this work is to join the both approaches generating an easier way to use Acceptance Testing Tools, like FitNesse, by reusing test scripts created using Record & Playback tools, like Selenium.

The integration proposed in this work is performed by SELF. SELF combines a method and a tool which attempt to integrate the utility of regression testing with the clarity of acceptance testing in order to create an environment for testing that reduces the effort in this activity and facilitates the monitoring by customers. This is possible due to the combination between Fit and Selenium in order to generate a new environment for AAT. This is the main contribution of this work, but presents as other contributions:

- A definition of SELF method to work with AAT that allows the use of acceptance testing both by developers and customers;
- A tool that implements the method prescriptions;
- An experimental study that shows a SELF evaluation in a controlled environment.

The remainder of this paper is structured as follows: Section 2 presents some concepts related to this work; Section 3 presents an overview of SELF; Section 4 presents the experimental study performed to evaluate the proposed approach; Section 5 describes some related works, and finally, Section 6 presents the conclusion and directions for future works.

2. BACKGROUND

Technically speaking, writing acceptance tests is usually simpler than writing unit or integration tests. This happens due to acceptance tests being written, or at least having to be understood, by customers themselves. As many others testing objectives, acceptance testing can be done manually. But the effort required in the setup and maintenance of acceptance testing is so hard that automation becomes mandatory. The automation of regression and acceptance testing, and their execution in a continuous integration cycle, is healthy to software development, but requires suitable tools to help in this task.

The most well-known implementation of a tool suitable for acceptance testing automation is Fit (or Framework for Integrated Test), introduced by Ward Cunningham [Mugridge and Cunningham 2005]. Fit is an open-source tool that allows customers to provide examples of how their software should work by writing them down in a table format. Fit tests are mapped to the target system, case-by-case, using a fixture, a special

[1] http://docs.seleniumhq.org
[2] http://www.fitnesse.org

class developed to link the test specification (table-based) and the target function in the Software Under Test (SUT).

In order to make these tests operate on the SUT without requiring knowledge in its inner workings (black box testing), it is common to associate Fit with Capture and Playback tools, such as Selenium. Selenium is an open-source testing tool for web applications that essentially follows the same style for test cases specification used by Fit: tests are defined in HTML tables. Test results are also visualized in the same way as in Fit, by coloring the table rows according to their outcome.

Despite the advantages of Capture and Replay tools, there are opportunities to improve their behavior. One example of this is the fact that this kind of tools does not follow the procedure and test case pattern [Binder 1999], where actions and test data are separated, and the test data is additionally split into input and expect outputs. These patterns allow test reuse and facilitate the understanding of testing goals.

Thus, the creation of acceptance tests using only FitNesse is hard, since the tables are time-consuming and the fixture must be closely linked to implementation of the SUT. The creation of testing using Selenium is simple, but the generated scripts are hard to maintain and to reuse. In order to solve this problem, we present in this article an approach that reuses testing scripts generated by Selenium to create acceptance tests for FitNesse.

3. SELF OVERVIEW

The main purpose of SELF is to provide a practical, simple, and effortless use of AAT. Besides this, another goal is to facilitate the creation of tests used by developers (regression testing) and by customers (acceptance testing). This is done by adapting two technologies associated with tools used in the industrial scenario (Record & Playback and Acceptance tools), to improve testing.

SELF was proposed because our experience in AAT (using the available testing tools) has required a lot of effort [Børge Haugset]. The creation of these tests using only FitNesse [Mugridge and Cunningham 2005] is very time consuming, since the required tables, detailing the software behavior, are repetitive and error prone, besides the construction of a fixture to link the testing specification pages and the target software be also costly [Hanssen and Haugset 2009]. However, we have noticed that the creation of testing using Selenium is simple, but the generated scripts are hard to maintain and to reuse. In order to solve this problem, we started some experiences to develop an approach that reuses testing scripts generated by Selenium to create acceptance tests for FitNesse. This action generates the SELF method.

It is important to emphasize that SELF method is not automated overall. There are activities executed by human beings and others related to some tools, including SELF tool. Our intention is to describe all the SELF activities, in order to create a complete guide to SELF use. Because of this we do not describe only the automated activities, we describe the whole process.

The SELF activities are presented in Figure 1. There are activities performed by human actors, and activities performed by tools. The first activity is the Test Planning. In this activity it is necessary to select the features that should be tested.The tester must specify the uses cases to be tested. This activity requires testing expertise and it is performed

by the tester.

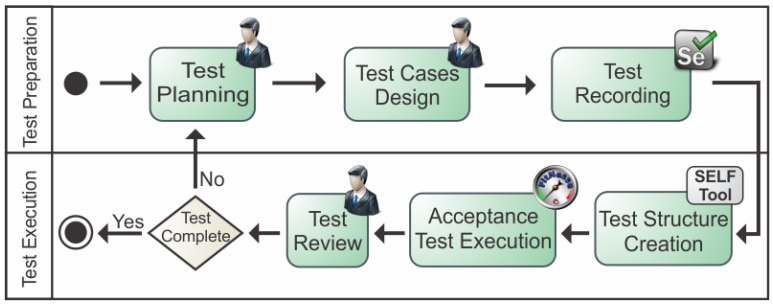


Figure 1. SELF Activities.

The second activity is called Test Case Design. In this activity the tester must think about at least one test related to the feature to be considered. The next step (Test Recording) is just to record the tests related to the behavior to be verified. The Test Case Design and Test Recording can be done at the same time. The tester can specify a test and immediately use a recording/playback tool to record this test. There is no need to write a document before this activity, since the structure to be created is itself a test documentation. The result of these activities is the test specification in a script format, containing the actions related to an execution, including the inputs, actions, and the expected result.

It is important to mention that the Test Case Design should not embrace all the conditions in the software under test (SUT). This test is used as basis for acceptance testing tools setup, since it is necessary to know the commands and fields related to a specific feature. After this setup, it is possible to extend the generated tests in whatever direction desired by the tester, in a simple way. Later, during the Test Review activity, at the end of a test cycle, it should be possible to extend the testing to other scenarios, but reusing the already created structure for test execution.

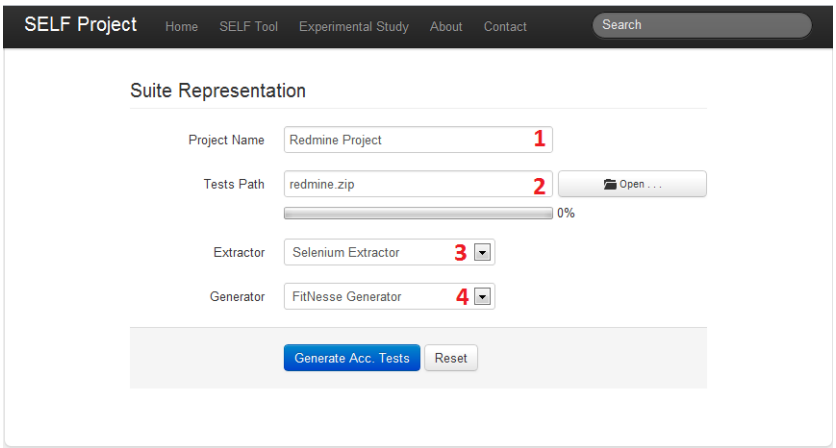


Figure 2. SELF main window.

The fourth activity is the use of the SELF tool to generate the environment for AAT. The tests generated by the SELF tool are dependent of the tool used for test execution. Thus, if the tester wants to use Fitnesse to execute the AAT, SELF tool must generate

tests following the structure required by Fitnesse. Nowadays, the SELF tool supports two test scripts as input (Selenium and Canoo[3]), as well as two acceptance testing tools as output (Fit[4] and FitNesse). However it is simple to extend SELF to support another formats, since its architecture was planned taking this fact into account. As shown in Figure 2, it is possible to select the extractor to be used for test input analysis and the generator to be used for AAT generation/execution.

The SELF tool generates AAT associated to every test present in the test scripts (path to the tests in Figure 2) supplied as input for environment setup. The algorithm to perform this task executes in three main phases: i) it extracts the test case and test procedures from the tests supplied as input; ii) it converts the test procedures in a fixture, mapping the commands in the test to actions in the target software, using Selenium or Canoo drivers; iii) it generates the test case specification (input and outputs) in a acceptance testing tool format, such as Fitnesse.

It is possible to observe that the activities performed by the SELF tool algorithm are simple. In fact, the most important thing associated to this work, for us, it is exactly the idea. This idea makes the AAT easier.

After the generation of the AAT, it is possible to continue the testing cycle, by creating new tests in the tables generated by SELF, once the infrastructure for testing is provided. This activity (Test Review) is associated to the execution of new testing cycles, in order to improve the testing.

The Figure 3 shows an example of a FitNesse test generated by SELF from a Selenium test. SELF uses as input a test recorded in Selenium and generates a test in FitNesse format. SELF is responsible to understand a test in Selenium format, identifying the inputs, the commands and splitting the test case of the test procedure related to a set of tests.

Figure 4 presents an example of a test generated by SELF. There are four test cases in the table related to a password change. The first three tests do not address a ”happy path” (successful scenario). They address exceptional conditions related to the SUT. The last one addresses a ”happy path” of the password change.

The previous example is interesting to show that a test can be built step by step. For instance, a tester could generate only the first test in the figure, by using SELF support. After this, the tester, together with the customers, could generate other test cases, since the environment is prepared to be extended.

3.1. Architecture

Figure 5 shows the SELF architecture. SELF architecture is composed by three main components: generator, model and extractor. The Extractor encapsulates TWork component, responsible to extract information from tests. The Generator component is responsible to create the link between the test case and the Software Under Test (SUT), in the suitable format (dependent of the Acceptance Testing tool used). In the FitNesse framework this link is called Fixture and it is very similar to a test procedure.

[3]http://docs.seleniumhq.org
[4]http://fit.c2.com

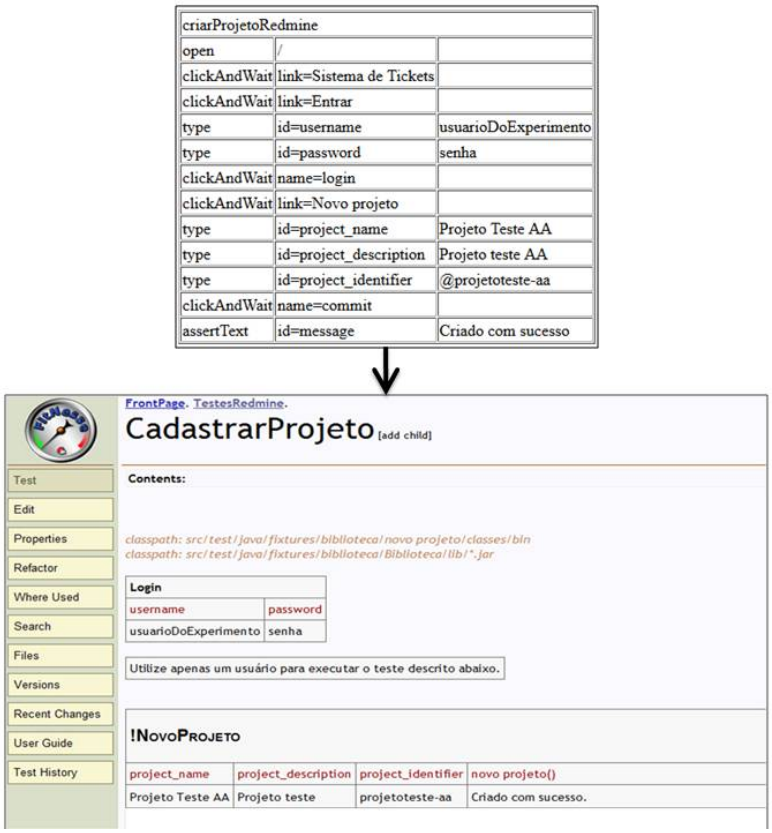


Figure 3. Example of SELF execution.

The GenericLink and GenericTable represents abstraction required to allow the use of several acceptance testing tools. For instance, to work with FitNesse, it is required to code the FitNesseLink and FitNesseTable by extending the base classes GenericLink and GenericTable. In the same way, it is necessary to create FitNesseGenerator by extending SELFGenerator. In these classes there are several definitions related to test generation but it is necessary to code some aspects directly related to a specific technology.

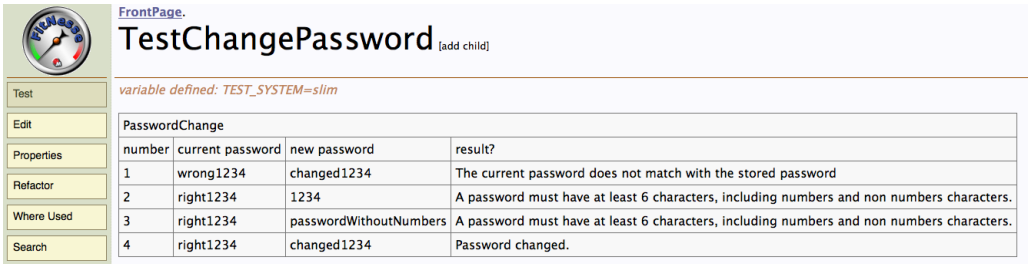
Currently SELF has one limitation: it is suitable only for web-based application. There are others possible improvements, like the generation of the data required, in the database, for test execution, but this is another work with several challenges. We have already done something in this direction [Santos et al. 2011] but the work performed is not integrated with SELF environment.

4. EXPERIMENTAL STUDY

This section presents an experimental study performed to evaluate SELF following the guidelines prescribed by experimental software engineering [Wohlin et al. 2000].

The purpose of the study was to evaluate if the use of SELF to support AAT can decrease the effort required for this task. The object of study is the SELF and the perspective of the study is from the point of view of the researchers.

The experiment was conducted in a controlled environment (in vitro), within Software Engineering (SE) and Object-Oriented Programming (OOP) classes of the Computer


FrontPage.

TestChangePassword [add child]

variable defined: TEST_SYSTEM=slim

Test | Edit | Properties | Refactor | Where Used | Search

PasswordChange			
number	current password	new password	result?
1	wrong1234	changed1234	The current password does not match with the stored password
2	right1234	1234	A password must have at least 6 characters, including numbers and non numbers characters.
3	right1234	passwordWithoutNumbers	A password must have at least 6 characters, including numbers and non numbers characters.
4	right1234	changed1234	Password changed.

Figure 4. AAT generated by SELF.

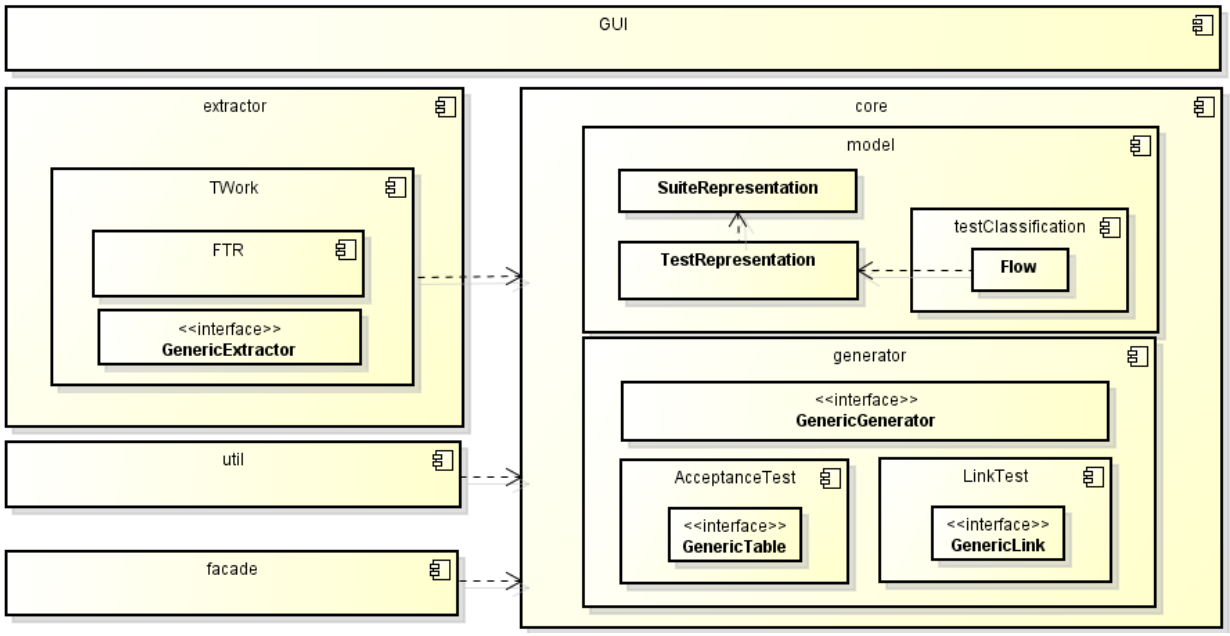


Figure 5. Components of the SELF Architecture.

Science (CS) course given at Universidade Federal do Piauí.

The study was performed using some use cases of a management system for medical association. This application is used in several states from Brazil. Basically, this system is responsible to manage all the services performed by a group of physicians to health insurance companies and hospitals. The participants had to create AAT for some use cases of this system with and without SELF support.

Tests created with SELF support are presented into the Fitnesse environment. SELF also generates fixtures to link the test cases and the target system using Selenium Drivers. Tests created without SELF support must be hard coded and must use JUnit and Selenium to allow their execution. This is a very common scenario in software development companies and due to this fact it was selected as a usual scenario to be compared with SELF support.

The experiment was carried out to answer the following research questions:

- Can SELF decrease the effort required to create AAT? We have used the time in minutes to evaluate this question.

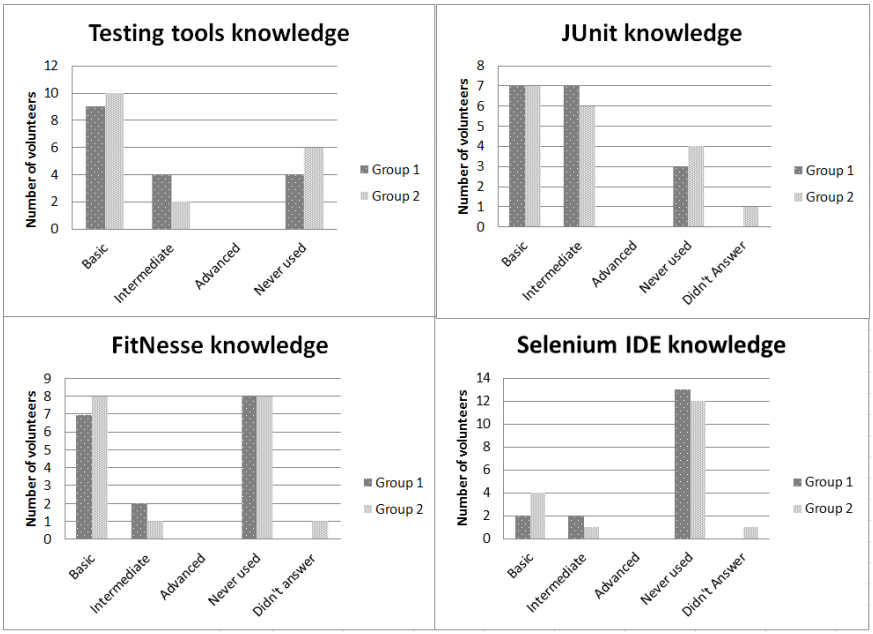


Figure 6. Volunteers profile.

- Is the clarity of AAT created with SELF support higher than the clarity of AAT created without SELF support? We have used a questionary to answer this question.

In the study we have two factors that could influence the results: the process of usual AAT development and the AAT development supported by SELF method (independent variables). The time required to perform the tasks represents the evidences of the results related to the two factors used (dependent variables).

Thirty-five students have participated in the experimental study. The volunteers were students from the Computer Science course, some from the Object Oriented Programming class and others from the Software Engineering class. The experiment had two groups of volunteers. Each class was divided in these groups, randomly.

Before the start of the experiment, the students answered a questionnaire for their characterization. We have not found a significantly difference among the groups, as can be shown in the Figure 6.

At least a basic knowledge in Java was required to participate in the study, since a testing creation in a usual way requires Java knowledge. This was a study requirement.

Since the characterization revealed a superficial knowledge about software testing, we have planned a training about this theme. We have given a class during one hour focusing on Selenium and JUnit use. As mentioned before, these tools are directly related to the usual way of creating AAT.

4.1. Experiment Design

The experiment was performed in two phases, each phase with two stages (training and execution) according to Table 1. We have split the participants into two groups randomly

selected. As mentioned before, we have chosen several use cases from an industrial medical management system. The use cases selected for the study were:

- Set of Functionalities 1 (SF1): Create New User, Medical Service Registry;
- Set of Functionalities 2 (SF2): Create New Holiday, Contract Registry.

It is important to emphasize that both groups of functionalities have the same characteristics: there is a simple use case (User and Holiday) and a more complex and specific use case (Medical Service and Contract). The use cases are similar in terms of size and complexity.

Table 1. Experimental study phases.

Phase 1		
Group	Training	Execution
1	Usual AAT	Usual AAT
2	Usual AAT	Usual AAT
Phase 2		
Group	Training	Execution
1	Usual AAT	Usual AAT
2	AAT with SELF support	AAT with SELF support

We have defined Usual AAT as the test creation using only JUnit and Selenium tools. This is a well known approach to automate acceptance testing in worldwide. This format for AAT is intensively based on programming. AAT with SELF support is the use of SELF to automate a part of the work related to the usual way, since SELF ool tgenerates tables and fixtures based on the test script example used as input.

Group 1 has created AAT for SF1 during the Phase 1 and for SF2 during Phase 2 in a usual way. Group 2 has created AAT, in a usual way, for SF1 during Phase 1, but in Phase 2 it has created AAT with SELF support for SF2. This allows the use of Group 1 as control group.

After the AAT creation, the study authors have executed the tests in order to evaluate their coverage in terms of business rules and identifying possible faults into the tests. The participants had to create a minimal set of tests for each funcionality. This set was analyzed by the authors in order to assure the minimal quality for each test.

4.2. Operation

The participants followed the same procedure presented in training session for the observation session, but without guidance by the study authors. In the observation session, the participants used only their knowledge and the available material. At the end of each phase the volunteers filled an electronic form containing their names and the time spent on AAT development. At the end of the second phase the Group 2 participants answered a post experiment questionnaire to collect as much data as possible about the session. They have answered the following questions:

- What is your opinion about the SELF tool?
- What are the strong and weak points of SELF?
- What is the best AAT presentation: with Fitnesse support or in Java code style?
- Would you use SELF again for AAT development?

4.3. Result Analysis

Figure 7 shows the summary of the data collected in the experimental study. Figure 8 shows a comparison of time spent by both groups. At the first phase, both groups created tests using the usual way. At the second phase, Group 1 (blue) created AAT by usual way and Group 2 (red) created AAT with SELF support.

Descriptive Statistics						
Phase	Group	Variable	N	Minimum	Maximum	Mean
		Score	17	4,7	9,2	7,7
		Errors	17	0	6	2,47
	1	Time	17	20	56	34,47
		Score	18	5,4	8,9	6,9
		Errors	18	0	3	1,56
1	2	Time	18	20	55	37,39
		Score	17	4,7	9,2	7,7
		Errors	17	0	4	1,29
	1	Time	17	12	83	45,18
		Score	18	5,4	8,9	6,9
		Errors	18	0	2	0,22
2	2	Time	18	16	42	27

Figure 7. Data summary in the experimental study.

The results showed the potential of SELF applied to AAT development. SELF support decreased the effort required for AAT development, maintaining the same degree of quality, since all the tests have covered the minimal set required in the study.

We have performed several statistical tests in order to evaluate our hypothesis. The collected data followed the normal distribution, so we have used *student t test* for all the comparisons, except the comparison between the participants grade scores ("score" rows in Figure 7), since they did not follow a normal distribution. In this case, we have used the Mann Whitney test [Wohlin et al. 2000].

We have analyzed the grade score of the students, since the score normally indicates the best students and this could influence the experimental results. We have found statistically significant differences in the scores of the participants. The participants of the Group 1 (control group) had higher scores than the participants of Group 2. This difference could be a disadvantage for SELF, but this fact did not impact SELF results, as we can see below.

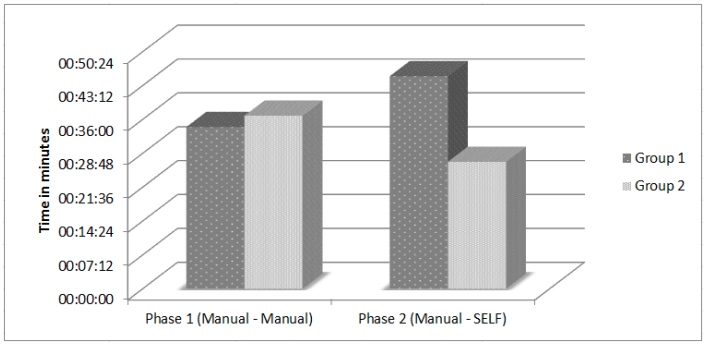


Figure 8. Bar graph with the time spent comparison.

The comparison among the data collected in Phase 1 has shown that Group 1 and Group 2 have not had statistically significant differences with respect to time spent. Both groups have used the usual AAT. In other words, the groups have had the similar results. This is an important finding, since it indicates that the groups are similar.

The second analysis was the same comparison but using the data collected in Phase 2. In this phase, Group 2 has used SELF to help AAT generation. There are statistically significant differences among the time spent. The use of SELF decreased the time spent to create the specified tests.

Another evaluation was performed with the help of a post-questionnaire associated to SELF use. The answers we have gotten indicated a SELF approval. The strongest point in SELF is related to its automation support for AAT development. The weakest point is related to the absence of the setup data for test environment preparation. All the participants have thought Fitnesse more suitable for AAT and have suggested that they could use it in a software project again.

4.4. Threats to validity

The experimental study was planned to reduce threats related to its Internal Validity and External Validity, which are the most important ones related to the studies in the area of Software Engineering [Wohlin et al. 2000].

Internal validity. The internal validity defines if the relationship observed between the treatment and the result is causational and not an influence of other factors which are not controlled. The experimental design used reduces the risk of having a bias, since it was planned with a control group. We have performed several analyses among the groups and we have not found anything that could influence the results besides the treatments. We have planned to minimize the effect of instrumentation by performing measurements on a single person along the experiment. We also planned the use of use cases with different features in order to minimize *Maturation* effects. The use cases have few aspects in common; therefore they minimize a positive influence on the treatments used. We also believe that this helped us to avoid negative influence, caused by the repetition of a similar activity.

External validity. The external validity defines the conditions which limit the ability to generalize the results of an experiment for the industrial practice. The subjects, students from the 5th and 6th semesters of the course, are not representative in terms of professionals. But they are in the final phase of undergraduation. The performed trainings, along with the fixation exercise, contributed to a good formation of the subjects in the used tools. The SUT was an industrial software with size greater than 100kloc. Thus, we do not believe that the conclusions obtained here are applied for developers, but we believe that could be extented for computer science students. It is believed that the conclusions obtained in the study can be extended to other systems, with the usual size and used by students, without losses in the observed results.

5. RELATED WORK

Executable Acceptance Test Driven Development (EATDD) provides promising possibilities on software development, since it drives the development from the requirements. It directly links requirements and quality assurance. The tight integration be-

tween the executable acceptance tests and the code allows better progress tracking, better communication between stakeholders and better software quality. These statements were discussed by Park and Maurer in a work about the benefits of acceptance testing [Park and Maurer 2008].

Hanssen and Haugset performed a study about the use of acceptance testing by developers [Hanssen and Haugset 2009]. The first conclusion in the study was the fact that this kind of testing is somewhat inappropriate for customers to actively express requirements. But the use of AAT can help developers reflect over requirements. This conclusion is exactly one motivation for SELF development, since it addresses the cost of creating AAT by introducing in a method and providing a tool to facilitate its use. As shown in the preliminary evaluation, the use of SELF could decrease the testing costs.

There are some tools created with similar SELF and SELF tool goals. EasyAccept tool, for instance, allows the creation and execution of acceptance testing based on text files containing the requirements [Sauvé et al. 2006]. At this point, EasyAccept is more similar to an AAT tool, but its main goal is to make the AAT easier, similarly to SELF. However, the implementation is totally different. EasyAccept requires a lot of additional work and, in practice, it is very difficult to be used by developers and customers. SELF method and SELF tool was created to facilitate the acceptance testing development by using testing scripts automatically generated by some recording/playback tools.

Another similar work was developed focusing on Rational Robot. Robot is a record/playback tool for test cases implementation, however, it also has as a drawback related to the low readability of its tests. Because of this the maintenance effort increases [Maciej GABOR and NAWROCKI 2004] and the end users are not able to deal with the scripts. In order to facilitate the writing and the reading of acceptance test cases with the use of Rational Robot it was proposed the use of the language and its compiler, the EasyRobot. The authors performed an initial evaluation of EasyRobot trying to quantify the benefits of its use. However, the study has presented a lot of threats that were not discussed in the work. Besides, they created only a new script, dissociated of a method and having the same problems of most of all testing script languages: i) they do not allow reuse and ii) they are not configurable.

Ricca et al [Ricca et al. 2009] presented the results of a series of controlled experiments aimed at assessing whether the adoption of table-based acceptance tests affects the understanding of requirements. They showed that the adoption of Fit tables to argument textual requirement descriptions led to a significantly better understanding of the requirements, when compared to the case where only textual descriptions were available. The results indicated an improvement in the comprehension four times (400%) bigger, without additional effort. SELF addresses just the main weakness of using ATDD: the difficulty to create an initial environment for ATDD development.

6. CONCLUSION

This paper has presented SELF, a method and a tool that combines the utility of regression testing with the clarity of acceptance testing. The main goal of SELF is to create an environment for AAT that reduces the effort in this activity, making easier the monitoring of development by customers and developers.

One the goals of the SELF development is just to increase the adoption of AAT

in the organizations. One obvious cost of adopting any testing practice, including AAT, is the time used to define and maintain tests in synchronization with the code being tested. Particularly maintenance of tests is reported to require a lot of effort. This gets even more prominent when tests have hardcoded data for check of results [Haugset and Hanssen 2008]. SELF deals exactly with this point, providing a simple environment to create tests, automatically splitting test actions (test procedure) and test dada (test case).

We conducted a study in order to evaluate our proposal. The results indicated that SELF can reduce the time spent in the activity. In summary, the original contributions of this work are described below:

- A definition of the SELF method to deal with Automated Acceptance Testing and to allow the use of acceptance testing both by developers and customers;
- A tool that implements the method prescriptions;
- An experimental study that shows that SELF can reduce the time spent in the activity .

The main contribution of this work is neither the method or the tool, but the idea of grouping several testing tools in a common environment in order to facilitate an important approach (automated acceptance testing) in a new way, where end users (customers) could be easily involved in the development process.

It is important to emphasize that SELF is been used in a industrial scenario. The company owner of the target system used in the experiments described here has manifested interest in using SELF in its activity. However, we have noticed some problems that make this task not so simple. We have found several problems related to the integration of SELF with systems intensely based on pages with several commands associated to rows in a table. It is necessary to improve the tool in order to extend its behavior to overcome this weak. This problem was expected, since the SELF development was made in an academic environment.

Another future work is the inclusion of testing setup environment in the SELF behavior. This will require a big work to integrate SELF with some previous work already developed by our group and directly related to this theme [Santos et al. 2011].

References

Binder, R. V. (1999). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley.

Børge Haugset, G. K. H. Automated acceptance testing: A literature review and an industrial case study.

Hanssen, G. K. and Haugset, B. (2009). Automated acceptance testing using fit. In *HICSS*, pages 1–8. IEEE Computer Society.

Haugset, B. and Hanssen, G. K. (2008). Automated acceptance testing: A literature review and an industrial case study. In *Proceedings of the Agile 2008*, AGILE '08, pages 27–38, Washington, DC, USA. IEEE Computer Society.

Maciej GABOR, G. J. and NAWROCKI, J. R. (2004). Making a capture-and-play tool suitable for agile software development. Foundations Of Computing And Decision Sciences.

Mugridge, R. and Cunningham, W. (2005). *Fit for Developing Software: Framework for Integrated Tests (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Park, S. S. and Maurer, F. (2008). The benefits and challenges of executable acceptance testing. In *Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral*, APOS '08, pages 19–22, New York, NY, USA. ACM.

Ricca, F., Torchiano, M., Penta, M. D., Ceccato, M., and Tonella, P. (2009). Using acceptance tests as a support for clarifying requirements: A series of experiments. *Information & Software Technology*, 51(2):270–283.

Santos, I., Santos, A., and Santos-Neto, P. (2011). Reusing functional testing in order to decrease performance and stress testing costs. *Proceedings of the 22nd International Conference on Software Engineering and Knowledge Engineering (SEKE'11)*.

Sauvé, J. P., Neto, O. L. A., and Cirne, W. (2006). Easyaccept: A tool to easily create, run and drive development with automated acceptance tests. In Zhu, H., Horgan, J. R., Cheung, S.-C., and Li, J. J., editors, *AST*, pages 111–117. ACM.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA.