Ensuring Quality in the Development of an Automated Testbed via Concepts of Extreme Programming[1]

Clauirton A. Siebra, Angela F. Freitas, Kleber R. Carneiro, Paulo R. Costa, Raquel X. Freitas, Fabio Q. B. da Silva, Andre L. M. Santos

Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Caixa Postal 7851 – 50.740-540 – Recife – PE – Brazil

{cas,aff,krc2,phrc,rxf,fabio,alms}@cin.ufpe.br

Abstract. *This paper describes our experience on using concepts of the Agile methodology during the development cycle of an automated testbed solution. The initial idea was to consider one of the Agile methods, the Extreme Programming, to lead our process. However, the particular features of our system have required adaptations in our initial process so that we could ensure the final product quality. This paper details all the steps of this adaptation, stressing the rationale for each of our decisions.*

Resumo. *Este artigo descreve nossa experiência no uso da metodologia Agile durante o ciclo de desenvolvimento de um ambiente de automação. A idéia inicial foi considerar um dos métodos Agile, Extreme Programming, como base do nosso processo. Porém, características específicas do nosso sistema forçaram uma adaptação dessa idéia inicial, de modo que pudéssemos garantir a qualidade final do produto. O artigo detalha todos os passos dessa adaptação, destacando as razões para cada uma das nossas decisões.*

1. Introduction

The development of mobile phones has evolved into a complex engineering process, mainly because of the capabilities that such devices currently support and are going to support in their new generations. This has increased the pressure on the test stage of mobile phones, which is required to apply more extensive and efficient procedures of evaluation so that the final product meets the fast time-to-market goals and can compete in the global marketplace. The test automation [Graham and Fewster 1999] is one alternative to this emerging scenario, because it enables tests to be launched and executed without the need for user intervention. In this way, common delays and errors associated with the manipulation of test parameters by humans can be avoided. Furthermore, tests can be continuously run during hours or days, ensuring maximum test coverage.

This paper relates our experience on the implementation of an automated testbed for mobile phones in close cooperation with the client and customers, in this case Samsung/SIDI and its engineering team. Our initial problems were that customers had only an abstract idea about the testbed and their functions, at the start of the

[1] The results presented in this paper have been developed as part of a collaborative project between Samsung Institute for Development of Informatics (Samsung/SIDI) and the Centre of Informatics at the Federal University of Pernambuco (CIn/UFPE), financed by Samsung Eletrônica da Amazônia Ltda., under the auspices of the Brazilian Federal Law of Informatics no. 8248/91.

development, and there were many doubts about the possibility of integrating this new testbed to third-party components, currently used in the test process of mobile phones. Thus, the development of this testbed was classified as a high risk project, which should be evolved in phases based on new requirements that could be raised along the project. For this main reason, we decided for applying several ideas of *Extreme Programming* (XP) [Beck 2000], which were compatible to the features of our project, as commented along the paper.

The remainder of this paper is organized as follows: Section 2 introduces the principal concepts of Extreme Programming and how we initially have used them to ensure the testbed development quality. Section 3 details the modifications on the original development method, stressing the reasons for such modifications. Finally, Section 4 concludes this work, presenting some final remarks.

2. Appling the Extreme Programming Concepts

Figure 1 shows an abstract architecture of the automated testbed, after an initial meeting with customers. In brief, we should develop an *automation tool* module and to integrate it to both a scenarios simulator and the mobile phone to be tested. The communication channels (CH) of this testbed gave us a better impression about the modules interactions, sequence of events and general function of the system as a whole. Then, based on Figure 1, we could imagine a system able to: sense and recognize the events generated by the scenarios simulator (CH_2); manipulate such events, generating specific commands to be executed into the equipment in test (CH_3); emulate the test execution outputs as events in the simulated scenario (CH_4); capture (failure or completion) feedbacks about the performance of the mobile phone in test (CH_5) and; generate logs that explain all the test process, such as detailing failures and their reasons (CH_1).

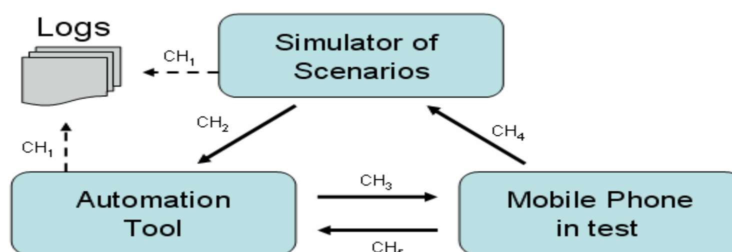

Figure 1. An abstract view of the testbed environment

After a first planning meeting and with this abstract understanding about the system and their features, we decided to apply several concepts of the Extreme Programming method. The main reasons for that were:

- Abstract initial idea about the system so that it was impossible to define a complete specification. Then, the system should evolve in iterations, with a continue aggregation of new functionalities and concepts;
- As the customers themselves were not sure about the real possibility of implementing some technical details, such as the integration of the testbed components, it was important to release small versions of the system, so that they could see its evolution and raise, from each release, new requirements to next iterations;

- We intended to have the customer constantly aware of what was being done, so that they could collaborate with the team to construct a product that would satisfy their expectations;
- We had a short time, about four months, to create something functional and useful. Furthermore, this development has involved activities of research and prototyping to validate some concepts. Thus, we needed to eliminate or summarize parts of the development that were not actually essential.

Extreme Programming (XP) is a software engineering methodology, the most prominent of several agile software development methodologies [Abrahamsson *et al.* 2003]. Like other agile methodologies, Extreme Programming differs from traditional methodologies primarily in placing a higher value on adaptability than on predictability. Proponents of XP regard ongoing changes to requirements as a natural, inescapable and desirable aspect of software development projects. They believe that being able to adapt to changing requirements at any point during the project life is a more realistic and better approach than attempting to define all requirements at the beginning of a project and then expending effort to control changes to the requirements. XP prescribes a set of 12 practices, for managers and developers, which are meant to embody and encourage particular values such as communication, simplicity and feedback. Table 1 lists such practices, indicates if we have initially employed them, and traces some comments about our initial use of these practices.

Table 1. XP practices and their initial use in our project

XP Practice	Adoption	Comments about the initial use of the practices
Planning game	Partially adopted	A plan was defined for each iteraction and their tasks (analysis, design or tests), including release dates (Figure 2). But we did not define requirements for each interaction in any level of abstraction.
Small releases	Adopted	Iterations were 15 days long each, with releases at the end of each one.
Metaphor	Adopted	A common vision about how the program works was defined (Figure 1), also with a system of names. This metaphor has mainly evolved during the first two iterations.
Simple design	Adopted	We did not care about a complex design, at the beginning of the project, which could anticipate any new functionality or concept.
Test-first design	Not adopted	The main integration test was left to the end of each iteration, while some functional tests were carried out during the implementation.
Refactoring	Adopted	We decided for regular refactoring and a flat design evolution.
Pair programming	Not adopted	We have used an adaptation of this idea (Section 4.4), mainly as a way to communicate knowledge throughout the team.
Collective ownership	Adopted	The team collectively owned the code. In several occasions the code required special knowledge to deal with particular problems. However this knowledge was always shared so that all the team could change the code.
Continuous integration	Adopted	We tried to keep the system integrated at all times. For that, it was common to perform more than one code consolidation during each interaction.
40-hour week	Partially adopted	We tried to keep an average time of 40-hour per week. A few times we put in overtime to meet the release dates.
On-site customer	Not adopted	The customer was geographically separated from the team. However several resources (e.g., Skype™ and teleconference) were used to surpass this limitation.
Coding standards	Adopted	A manual of coding standards was specified and used for the team. Thus, the code was kept uniform and familiar to all members, supporting the collective ownership.

During the planning game, it was defined that we should deliver the system in five iterations of 15 business days each (Figure 2). Each of these iterations was defined as follows: three days to requirements specification, five days to analysis and design, seven days to implementation and one day to final tests. After that, the development team could build a release and send it to the project manager and customer so that the interaction could be approved. Apart from the code implementation itself, other artifacts should also be delivered, such as: an interface prototype document, a document of requirements and a document of user cases.

	ℹ	Nome da tarefa	Duration	Work	Start	Finish	Predec
265	✓	⊟ **Tool - Iteration 1**	**29,06 days**	**202,6 hrs**	**Tue 7/11/06**	**Sat 16/12/06**	
266	✓	Definition of the requirements - Iteration 1	3 days	26,48 hrs	Tue 7/11/06	Fri 10/11/06	260
267	✓	Analysis and Design iteration 1	5 days	51 hrs	Mon 13/11/06	Fri 17/11/06	266
268	✓	Building iteration 1	7 days	86 hrs	Mon 20/11/06	Mon 27/11/06	267
269	✓	Test iteration 1	1 day	13 hrs	Tue 28/11/06	Tue 28/11/06	268
270	✓	Submit automation tool - iteration 1 to Tartan Approval	0,5 hrs	0,13 hrs	Wed 29/11/06	Wed 29/11/06	269
271	✓	Tartan Approval - automation tool - iteration 1	13 days	26 hrs	Wed 29/11/06	Sat 16/12/06	270
272		⊞ **Tool - Iteration 2**	**17,19 days**	**213,13 hrs**	**Wed 29/11/06**	**Thu 21/12/06**	**269**
279		⊞ **Tool - Iteration 3**	**14 days**	**155,13 hrs**	**Tue 2/1/07**	**Fri 19/1/07**	
286		⊞ **Tool - Iteration 4**	**17 days**	**313,13 hrs**	**Fri 19/1/07**	**Mon 12/2/07**	
293		⊞ **Tool - Iteration 5**	**17,06 days**	**374,13 hrs**	**Mon 12/2/07**	**Mon 12/3/07**	

Figure 2. Schedule to 5 iterations defined during the plan game phase.

The phases of requirements definition were the moments when the team had more opportunities to discuss about the system evolution. The team itself was composed by one customer (representing the company), one part time project manager, one technical coordinator and three developers. From this team, only the project manager had experience with XP. However all the team had opportunity, before the project start up, to review the method. During the evolution of the iterations, more functions and external components (e.g., database) were adding complexity to the development so that we were forced to tailor the XP method according to our needs.

3. Adaptations and their Rationale

Several projects have made adaptations to XP [Grenning 2001] [Silva *et al.* 2005] during their software development process. These adaptations are very connected to the final product features. An interesting case, for example, is the adaptation of XP for developing lightweight ontologies for Artificial Intelligence applications [Hristozova and Sterling 2002]. The important point of this discussion is that all these adaptations are in fact supported by the Agile Methodology itself, which argues that there is no process that fits every project as such, but rather practices should be tailored to suit the needs of individual projects. Based on this context, the next sections discuss the adaptations carried out during our project and the reasons for them.

3.1. Documentation

The documentation on the development process was the activity that required more adaptations during the project. According to XP, documentation is a very time-consuming process so that it is very important to discuss about which kind of documentation is in fact important for the project. Based on this affirmation, we have

defined: (1) a *Coding Standard* (CS) document to clear the understating of source code; (2) a *Requirement Specification* (RS) document to specify the system requirements; (3) an *Use Cases* (UC) document to support technical reviews and system maintainers; and (4) an *Interface Prototype* (IP) document to validate the requirements with the customer. Note that, while the CS was prepared during a pre-iterations phase, the other documents should be updated at each interaction (RS during the definition of requirements phase and UC/IP during the analysis and design phase).

Our first setback was about the UC documentation. This kind of document, in particular, can and should be written in a way that avoids frequent hard modifications. For example, we could avoid the definition of alternative and exceptional fluxes, keeping the focus on the main UC goal. However we did not follow this recommendation so that the updates of this documentation, during the second and third iterations, were very time-consuming. Furthermore, as we were using the principle of *collective ownership*, all the team was always up to the code (we did not have a division between developers and programmers). Such facts have reduced the interest for UC in our project. On the other hand, the specification of UC has helped us to think about the system in a more detailed way. In addition, UC documentation is very important to maintain the system, mainly if the maintenance process will be carried out by other team in the future. To deal with this treading-off, we decided by relaxing the UC specification during the iterations and create a complete version at the end of the project, as commented later.

The second setback was related to the IP documentation. We have noticed that it was very useful during joint discussions because the team could discuss better the design ideas in such an easier way to all participants. For example, the IP documentation helped to reveal misunderstandings regarding what the customer expected from the first iteration. This was a decisive factor to consider the IP as an instrument to validate the understanding of customers during the requirements elicitation phase, rather than only creating it during the analysis and design phase.

Our third setback was about the difference between documenting new requirements and solicitations of changes (e.g., interfaces adjustments). As a way of avoiding mix of concepts, we decided to introduce a new document (SLA – *Service Level Agreement*) to register such solicitations. SLA keeps tracking of the tasks that are developed by the team. It contains information about the tasks to be developed, the iteration to which each task was assigned and estimated time for tasks completion.

Other motivation to use the SLA documentation was to offer a better support to the *release planning*, which is part of the XP *Planning Game* practice. However XP implements this document in a different way. XP suggests that solicitations to be implemented (called user stories) should be written on cards, and then assigned priorities and estimations. These cards are then moved around a table, in a meeting involving all the team, to create a set of stories to be implemented during the iterations. In our case, as the team was not located at the same site, we could not work in this way. The tasks needed to be documented using a media that could be easily sent to everyone involved, and thus we decided to document them in an Excel™ document.

A fourth setback was the sporadic need of *Technical Research Reports* (TRR). TRR are documents that describe the results of investigations about one or more

technologies or concepts. The main aim of this document is to provide knowledge to the team, supporting its process of decision making. Thus, TRR is much like an explanatory and comparative document. During our project, the team has produced four TRR. Generally one of the members was allocated to this task, working part of an interaction in this subject. The TRR was included as part of the package released at the end of some interaction.

A last issue was concerned with the documents related to the *Death Phase* (after iterations phase or the end of the project). The main aim of this last documentation package was to support the future generations of engineers that will maintain the product we are building. Actually the death phase documentation can be seen as an evolution of the documents produced along the project. The documents that we have considered to the death phase are:

- Final and complete version of User Cases;
- Generation of *Javadoc* to provide a high-level document to navigate the system;
- Document about the system architecture. This was an appropriate moment to do that because the last refactoring was already carried out;
- User manual, which was evolved from the interface prototype documentation.

Considering all this discussion about documentation, we can conclude we were in accordance with XP when we decided to create all the documentation that in fact can help the development process. Ever the SLA and TRR, which are not common documents of a development process, have their roles in our process. The SLA has improved our ability to monitor the development progress, mainly from the point of view of customer and project manager. Furthermore, it was employed as a basis to establish metrics of development, so that we could predict average values to codification tasks. TRRs have avoided the losing of specialized knowledge, as well as supported the sharing process of such knowledge and its maintenance for future iterations.

3.2. Refactoring

XP advocates that we should only codify what it is required at the moment, keeping the implementation as simply as possible. During the evolution of the code, this may result in a system that is stuck. For example, with multiple copies of the same/similar code, or with codes very related so that the change of one affects another. To deal with this problem, XP doctrine says that when this occurs, the system is telling you to refactor your code by changing the architecture, making it simpler and more generic.

We have initially agreed to such practice. However the very constant adjustment of requirements, mainly during the first iterations, has changed our minds so that we decided to perform only two main refactorings. A first refactoring was done when we had a better idea about the system (end of the second interaction). The second refactoring was scheduled to the end of the project. This decision was possible because our project is complex in terms of routines and functions (e.g.; routines for external integration or synchronization algorithms). However it is not so extensive in terms of code lines (the system has about 120 classes), so that bad-design related problems could temporarily coexist with the ongoing codification. An interesting idea is to use the concepts of refactoring and patterns together [Kerievsky 2004]. Fundaments of this

matching suggest that using patterns to improve an existing design is better than using patterns early in a new design. Then we could improve the design with patterns by applying sequences of low-level design transformations in the form of refactorings.

3.3. Tests

XP encourages the practice of creating unit tests before coding the functionalities. According to XP, the creation of unit tests helps developers to really consider what needs to be done, and thus is a good guiding to the development of code. We have used continuous tests (in parallel to implementation) to validate several parts of our system. However the test definition was done only after the implementation of each functionality. The main reason was we already had enough documentation to know our needs and the additional creation of unit tests documents, before the coding, could delay the process.

From our experience we could say that continuous tests are in fact a very good practice that adds quality to the code as it ensures each individual component is working before integration. However we do not have the experience to say if it is better/easier to think about such tests before or after the implementation. Our big mistake related to tests was that we did not document them. Actually we did not feel the need of this documentation as a way to help our development, but to revalidation of functions after significant code changes and refactoring processes. Thus we certainly corroborate to the idea of having a very well documented suite of tests, independently if they are produced before or after the implementation of each functionality.

3.4. Additional issues

We have applied other three minor but important adaptations on the XP practices. First, we have not used the *On-side customer* practice. However this by no means should be interpreted that geographically distributed teams are not entitled for XP methodology. Actually there are already reported experiences on successful projects with distributed teams [Baheti 2002]. In our case we have used multimedia resources (e.g., Skype™ and teleconference) to "connect" the team. Note that it is important that, even geographically dispersed, the team must be available for eventual discussions.

The second adaptation is also related to the *On-side Customer* practice. Apart from eventual discussions, we have defined fixed meetings, at the beginning of each interaction (Definition of Requirements phase), with all the team to define the next directions. Such meetings were important to attenuate the dispersed feature of our team. To improve the effectiveness of such meetings, we have noticed the importance of short pre-meeting, without the participation of the customer, so that we could align our ideas and avoid as much as possible conflicts between members of the development team.

The third adaptation is related to the *Pair Programming* practice. We have observed that the process of codification swaps between deliberate and mechanical moments. Deliberative moments are the part of the codification where developers make decision about how to implement the code (algorithms). Mechanical moments are the part of the codification more related to edition and organisation of the code (e.g., choice of names, creation of classes, imports, implementation of algorithms, etc.). We have only used pair of developers during deliberative moments, when critical decisions

should be done. In such cases, it was very important two developers to exchange ideas to reach better solutions. The code integration is a different moment where developers are working side-by-side. Here we have used a merging tool the shows the difference between the original and new code. Thus, before integrations, pair of developers must review all the changed code so that at least two persons are aware of its modifications.

4. Final Remarks

We had a good productivity applying the techniques discussed here. At the end of five iterations we had a functional testbed, whose features were compatible to the pre-defined requirements so that the customer accepted the final product. Some numbers about our final products are: 122 work days (1.560 hours), 5 developers, 25 specified packages with 120 classes, 10720 code lines, 843 methods and 415 attributes. As future directions, an extension plan was already specified to this testbed and our intention is still improving/adapting our method during the development of such extension.

Acknowledgements

The authors would like to thank all the test engineers of the CIn/SIDI-Samsung Test Center (Amanda Araujo, Karine Santos, Luiz Filgueiras and Talita Spiller). The team is also very grateful for the support received from Samsung/SIDI team, in particular from Ariston Carvalho, Miguel Lizarraga, Ildeu Fantini and Vera Bier. The National Council for Scientific and Technological Development (CNPq) has provided valuable support to the project through the Brazilian Federal Law no. 8010/90.

References

Abrahamsson, P., Warsta, J., Siponen,.T., and Ronkainen, J. (2003). “New Directions on Agile Methods: A Comparative Analysis”. In *Proceedings of 2003 International Conference on Software Engineering*, Portland, Oregon, USA, pp.244-254.

Baheti, P., Gehringer, E. and Stotts, D. (2002) “Exploring the Efficacy of Distributed Pair Programming”, *Extreme Programming and Agile Methods*, Lecture Notes in Computer Science, 2418:208-220, Springer Berlin.

Beck, K. (2000) “Extreme Programming Explained”, Addison-Wesley, Reading, Mass.

Graham, D. and Fewster, M. (1999) “Software Test Automation: Effective Use of Test Execution Tools”, Addison Wesley.

Grenning, J. (2001) “Launching Extreme Programming at a Process-Intensive Company”, *IEEE Software*, 18(6):27-33.

Hristozova, M. and Sterling, L. (2002) “An eXtreme method for developing lightweight ontologies”. In *Workshop on Ontologies in Agent Systems*, First International Joint Conference on Autonomous Agents and Multi-Agent Systems, Bologna, Italy.

Kerievsky , J. (2004) “Refactoring to Patterns”, The Addison-Wesley Signature Series.

Salo, O. (2004) "Improving Software Process in Agile Software Development Projects: Results from Two XP Case Studies," In *Proc. of 30th EUROMICRO Conference*, pp. 310-317.

Silva, A., Kon, F. and Torteli, C. (2005) “XP South of the Equator: An eXPerience Implementing XP in Brazil”, *Extreme Programming and Agile Processes in Software Engineering*, Lecture Notes in Computer Science, 3556:10-18, Springer Berlin.