DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-2004-01

BEING EXTREME IN THE CLASSROOM: EXPERIENCES TEACHING XP

ALFREDO GOLDMAN, FÁBIO KON, PAULO J. S. SILVA AND JOE YODER

Janeiro de 2004

Being Extreme in the Classroom: Experiences Teaching XP

Alfredo Goldman   Fabio Kon   Paulo J. S. Silva
Department of Computer Science
University of São Paulo, Brazil
{gold,kon,rsilva}@ime.usp.br
http://www.ime.usp.br/~xp

Joe Yoder
The Refactory, Inc. and
Department of Computer Science
University of Illinois at Urbana-Champaign
joeyoder@joeyoder.com
http://www.refactory.com

Abstract

Agile Methodologies propose a new way of looking at software development that questions many of the beliefs of conventional Software Engineering. Agile methods such as Extreme Programming (XP) have proved to be very effective in producing high-quality software in real-world projects with strict time constraints.

Nevertheless, most university courses and industrial training programs are still based on old-style heavyweight methodologies. This article, based on our experiences teaching XP in academic and industrial environments, presents effective ways of teaching students and professionals on how to develop high-quality software following the principles of agile software development. We also discuss related work in the area, describe real-world cases, and discuss open problems not yet resolved.

1 Introduction

In the last few years, agile software development methodologies have become widely known and have been successfully adopted by hundreds of organizations worldwide. Agile methodologies such as XP [Bec99], Scrum [SB01], and Crystal [Coc02] are now used in small, medium, and large companies, universities, research institutes, and governmental agencies. However, the large majority of organizations have a long history of using old-style, heavyweight methodologies and most of their programmers and managers are educated to develop software in a bureaucratic way in which software quality is usually not the top priority.

The Manifesto for Agile Software Development [B+01] indicated the four most important aspects of agile methodologies that differentiate them from conventional software development. Agile methods value:

- **Individuals and interactions** over processes and tools;
- **Working software** over comprehensive documentation;
- **Customer collaboration** over contract negotiation;

- **Responding to change** over following a plan.

While the items on the left (in bold face) are the core principles of a successful agile software development project, most programmers and managers are educated in a culture that values more the items on the right.

After participating in agile software development projects, a large number of academic and industrial software developers have come to the conclusion that agile methodologies is the most effective way for developing high-quality software in time-constrained projects. There are plenty of examples of successful agile projects. However, there is still a lot of work to do in the field of teaching agility. Most undergraduate Computer Science courses and training courses for industry are based on conventional methods that focus on tools, documentation, contracts, and following plans.

What CS education needs is a reality shock! We need to modernize our courses to show students that personal communication, working software, customer collaboration, and dynamic adaptation are, at least, as important as the traditional values that we are used to teach.

In this article, we present our findings in agile methodology education from our experiences in teaching Extreme Programming (XP) in academic and industrial environments. In Section 2 we describe a few related works in this area. In Section 3 we give a brief overview of XP and discuss the points in favor and against XP, addressing when one should and should not use this methodology. In Section 4, we describe the adaptations that must be made in order to carry out an XP project in an educational environment. In Section 5 we describe some of our experiences in teaching XP both in the university and in the industry and in Section 6 we discuss problems that are still unresolved. Finally, in Section, 7 we present our conclusions.

2 Related Work

Many new articles have recently emerged in the literature on the subject of teaching XP or some of its practices. Even though both approaches may seem closely related, there are good reasons to treat them separated. XP is clearly based on the synergistic result of 12 practices working together – this is one of the aspects of the "extreme" word in the methodology name. Therefore, the net result of teaching isolated practices deserves careful reflection. Among the individual practices, two emerge as being beneficial even when detached from the others: pair programming and test-driven development.

Pair programming has been extensively studied by Williams et. al. In [WK00, CW00, WK02] the authors show that pair programming is highly efficient. Working in pairs, programmers work twice as fast and think of more than twice as many solutions to a problem as two programmers working alone, while attaining higher defect prevention. The result can be faster code with higher quality. The experiments were performed with senior-level students in a Software Engineering course at the University of Utah and confirmed previous anecdotal evidence.

In [NWW+03] the authors focus on the pedagogical benefits of pair programming. It stimulates cooperation, which is highly regarded in the working environment but somehow neglected in the academic setting. Pair programming can lead to improved success rates, that is the rate of students that complete the course with grade of C or better, and the future performance of the paired students is about the same as the solo ones. This last observation addresses the concerns of some instructors that some paired students might pass the workload to his/her partner and not learn the course material. Another interesting benefit of pair lab sessions is that the students solve, with their partners, most of the simple questions, alleviating the lab instructors workload. Finally, the authors give some suggestions on how to use pair programming effectively in the teaching lab. For example, they suggest that the instructors must constantly emphasize the different roles in the pair (driver and navigator) and encourage role and pair changes. Another interesting advice is to use some kind of peer evaluation to avoid the "free ride" on the partner's work.

Test-driven development is another practice that fits easily in many different development methodologies and, hence, can be taught detached from the other practices. A good book on this subject is [Bec02b]. Actually, test-driven development should be taught before debugging techniques, as a better way to avoid and catch errors. The instructor must be prepared to face resistance from many students that feel that tests are a waste of time. Here some good anecdotal stories may be beneficial, such as McBreen's testimony that he moved from daily debugging sections to only three or four sessions a year after adopting XP style unit tests [McB03].

Finally, another practice that may be beneficial to teach to advanced students is refactoring [MS99]. However, caution has to be taken since this practice is closely related to other XP ideas such as incremental development, not planning for the future, tests, and continuous integration. Another report on teaching and using just a few practices is given in [ADW01].

Other works describe experiences on teaching full XP. In [Tom02], Tomek presents his experience on teaching XP in two Computer Science courses and proposes several recommendations. According to his experience, it is very important to provide a very agile environment, so he used VisualWorks Smalltalk and its IDE. In his first course, Tomek used two projects, a first one to get the feeling of XP followed by a more realistic one. During the second course, he focused on a single project with a real customer.

Wilson's experience [Wil01] is similar, although he used Java instead of Smalltalk. The course project was to improve a prototype Java IDE. Finally, Lappo taught an eXtreme Programming course to a group of Masters students that spent 12 weeks working full-time to produce a Web-based resource management application with Java technologies. In all these cases, the instructors accumulated the role of mentor, coach, and usually even custumer.

All three experiments presented small problems, such as the lack of real custumers, the overload of being mentor, coach, and client, the short period that can be dedicated to the project in ordinary CS courses, and the lack of adequate space for the XP team.

At this point, we can point out a few suggestions: students like meaningful projects that have real use, students should be relatively advanced to be able to get the most out of the course, it would be ideal to provide a dedicated room for the XP class, and it might be interesting to provide the students with an implemented core of the application. Wilson suggests that this prototype can work to give unity to the end result substituting the metaphor, which is one of the most difficult practices to teach.

The courses also revealed some problems in trying to follow all XP practices. In some cases pair programming was partially neglected, even by the instructor that refactored the code alone. Other problems were the lack of a real client, the absence of a metaphor, and some slips in the release schedule. Interestingly, it seems like test-first development was quickly assimilated by the students, who learned to appreciate its advantages.

In Sections 4 and 5 we show how to avoid many of the problems described above. In particular, we show effective ways to teach XP, putting into practice most, or even all, of the practices in both industrial and academic environments, giving a complete XP experience to students.

3 Overview of Extreme Programming

The Extreme Programming methodology was formulated by Kent Beck based on his long experience in object-oriented software development in Smalltalk together with Ward Cunningham. XP is composed of a collection of practices that, in isolation, have been well known and used widely for many years. The main contribution of XP is the conjunction of these practices in a cohesive methodology, which fosters the synergistic effects of this mixture.

3.1 The 12 Practices

When the methodology was first introduced in 1999 [Bec99], it consisted of 12 practices: Planning Game, Small Releases, Metaphor, Simple Design, Testing, Refactoring, Pair Programming, Collective Code Ownership, Continuous Integration, 40-hour Week, On-site Customer, and Coding Standards. A few years later, the 40-hour Week practice was renamed to Sustainable Pace and a new rule was added: *Fix XP When It Breaks.* We will now describe briefly each of these practices, for more detailed descriptions see [Bec99] and `www.extremeprogramming.org`.

Planning Game. A project starts with a short exploratory phase in which the customer expresses the requirements (through user stories written in *story cards*) and the development team, together with the customer, creates a release plan specifying which story cards should be implemented for each system release. The team negotiates, with the customer, dates for each release based on business priorities and technical estimates. However, the most important point here is that the plan is just a plan, i.e., the team and the customer know that it is not the reality that they will face. As reality overtakes the plan, the plan must be updated. So, rather than being completed upfront, in XP, planning is an everyday activity. A good XP team must know how to adapt dynamically to changes at any moment in the development process.

Small Releases. Rather than developing big pieces of software at a time, the team should implement a very small piece of working software first and then enhance it incrementally. Ideally, the team should deliver new releases of working software every few weeks or, in some cases, days. The time between releases cannot be a few months or more.

In each release the team implements a set of story cards. Each story card is assigned to a specific programmer who becomes responsible for its completion (although it does receive help from its colleagues to achieve that). Stories that are more important to the customer receive a higher priority and are implemented in the first releases. Developers and customer may negotiate during development to move cards from one release to the other or to create, remove or modify them as the team learns new things and business requirements evolve.

A key rule of incremental development in XP is: do not code for the future, do not anticipate requirements. This spirit is usually expressed in the sentence *do the simplest thing that could possibly work.* This implies that one should not add flexibility that is not needed to complete the current task. If you think that a little more flexibility will be valuable in a couple of weeks, don't do it now; wait until it is really needed and then refactor the code to add the required flexibility.

Metaphor. A simple story of how the system works should be shared by all the stake holders in the project. This helps all the participants to understand the basic elements and their relationships and may improve communication.

Simple Design. The system should always have the simplest possible design at any moment. If extra complexity is found, it must be removed as soon as possible. And again: *do the simplest thing that could possibly work.*

Testing. Programmers write unit tests for all system components so their confidence in the correct behavior of the system becomes part of the system itself. In a more recent book [Bec02b], Beck describes *Test-Driven Development* in which the unit tests are written even before the code to be tested, which is also called *Test-First Programming.*

Customers write functional (acceptance) tests demonstrating that the required features are implemented correctly. If the customer is not a programmer, one of the developers pairs with the customer to write the tests.

Refactoring. Using techniques such as the ones described in [Fow99], programmers restructure the system continuously to improve it without changing its behavior. Possible improvements include simplifications, optimizations, enhancing clarity, adding flexibility, etc.

Pair Programming. Each line of production code is written with two programmers simultaneously at a single machine. As explained in Section 2, pair programming improves code quality greatly without impacting the speed of development. Communication will flow better across team members if the pairs change frequently (e.g., every day). The pairs are selected based not only on availability but also in expertise. For example, if it is necessary to build a Web interface to a database, one could select a pair in which one of the programmers is an expert in databases while the other is an expert in frameworks for building Web interfaces.

Collective Code Ownership. Any developer can change any piece of code in the system at any time without requesting permission. This introduces a high level of agility in the team. Since there are unit tests for each component, programmers are less likely to break each other's code.

Continuous Integration. The source code must be kept in a shared repository and every time a task is completed, the new code must be built, tested, and, if correct, integrated into the repository.

Sustainable Pace. The team should work in a pace that it can sustain without harming its participants, for example, 40 hours per week. A team that is physically or intellectually tired is very likely to produce low-quality software. Working overtime in a certain, special week is acceptable; however, if the team is asked to work overtime two or more weeks in a row, this is a sign that there is something very wrong with the project.

On-site Customer. A real user of the system should be included in the team and be available full-time for answering questions. No matter what happens to the project (good or bad), it will never be a big surprise to the customer since he/she is following the development daily.

Coding Standards. In the initial phase, all the developers must agree on a common set of rules enforcing how the system must be coded. This facilitates communication and enable groups of many programmers to produce consistent code. Recent tools such as the Eclipse *Checkstyle* plug-in (see `eclipse-cs.sourceforge.net`) can help automating part of the process.

It is important to emphasize that the value of XP is in applying all the practices in conjunction. Applying a subset of the practices, without careful consideration, can even be harmful. For example, applying aggressive refactoring without a good collection of unit tests may lead to disastrous results as the programmers cannot verify if their changes are breaking the code or not. Adopting the planning game, changing the plan dynamically, without a close contact with the customer may lead the team to build a system that is not the one the customer wants.

3.2 Adapting XP

Teams that are new to XP should try to follow all 12 practices as rigorously as possible. More experienced XP developers, however, will notice that this may not be possible, or even desirable, in all situations. When this happens you may need to adapt XP by applying the *Fix XP When It Breaks* rule.

We present an example to illustrate this rule. An experienced XP developer, Klaus Wuestefeld, working in a project for a cable-TV scheduling system realized that he would not be able to have an on-site customer since the company contracting their services was located in another state. The solution was to adapt XP introducing the concept of *Customer Proxy*. Klaus acted as a customer by answering programmer questions immediately. He would then call the real customers on the phone or email them later with the questions verifying his response. Most of the times the proxy's guesses were correct and the development evolved quickly. The few times that he made the wrong guess, he simply came back to the programmers and said: "I changed my mind", which is completely acceptable within the rules of XP.

Another possibility is to select analysts that have worked closely with the customers to become the customer proxy. This adaptation has been reported by Martin Fowler in projects carried out by Thoughtworks.

In another project, Klaus noticed that developers were worried too much about the story cards assigned to them and were not always willing to help their colleagues by pair programming with them[1]. The solution he adopted in that case was to create a new role: the *Libero*. One of the programmers, called the Libero, was not assigned any story card; his task was simply to pair program with the others helping them finish their cards.

A limitation of XP is that, since it requires direct communication among all team members, it does not scale well for groups with much more than 10 developers. To overcome this limitation, practitioners have extended the methodology to work with larger projects of up to 100 developers. This was achieved by dividing the team in sub-groups of at most 10 people and integrating periodically the software produced by each of the groups. Ron Crocker has worked many years with large-scale agile projects for Motorola. His extension of XP is called the Grizzly method and a new book on the subject is coming out in 2004 [Cro04].

As a last example, sometimes teams allow for more individual spike solutions to be developed. Then, these solutions are released into the main code base only after test cases are developed and a pair of eyes looks over the solution. This can be a solution when pair programming is not always possible.

A few more examples of interesting adaptations of XP can be found in [TF00].

3.3 When not to use XP

There are some situations when using XP should be avoided. The possible pitfalls for XP adoption fall in three categories: resistance from the development team to embrace XP, resistance from the organization that houses the XP team or from the client to accept XP corollaries, and inadequacies inherent to the software that have to be developed. McBreen has recently written an interesting book on this subject called *Questioning Extreme Programming* [McB03].

The resistance from the development team may be associated with habits acquired during the team members' life as programmers. At a first view, even pair programming may be odd, test-driven development a burden, and simple design an excuse from lazy minds. After an adaptation phase, however, many developers learn to appreciate the practices and the agile development environment. On the other hand, the resistance from the development team may be associated with one of XP most profound facets: XP is a subversive methodology, in the sense that it requires a completely new organization of the team. In traditional water-fall methodologies there are distinct and well defined roles for the team members such as requirement analysts, system analysts, programmers, testers, and so on. In XP, all team members play all these roles, they are all *developers*. This creates a completely new balance of power within the team that may face great resistance. In order to adopt XP, the team must feel comfortable with the idea of working together as a group with the single goal of delivering high-quality software in time.

To overcome this resistance, the XP instructor or mentor should pick as members for the first XP experiment a group of people that is naturally inclined to experiment with new ideas and that are self-confident enough not to feel threatened by the new balance of power. To achieve this in the industry, it is essential to have the support of the management level of the company, which will help the mentor to identify good candidates. In the university setting, this is better achieved by using

[1]This is actually not very common in XP projects; usually, programmers negotiate among themselves to help each other implementing their cards.

elective courses. After the first successful XP experience, the word of mouth of the participants will spread the news and it will be much easier to introduce XP into the entire organization or to make the course mandatory for all students.

Examples of resistance from the organization and/or the client are the lack of commitment of the client in participating actively in the development process, a requirement for long and formal descriptions of the product to be developed before it is developed, the need to have a single person to blame should anything go wrong[2], the need for a long, detailed documentation for the maintenance phase. All these XP consequences must be clearly stated and understood both by managers and developers before starting the first XP project in an organization.

Finally XP is not meant for all software development projects. Certain aspects of the software to be developed may conflict with basic XP assumptions. Does the project require a very large team (e.g., more than 20 people)? Does the edit-compile-run cycle take too long to complete? Do the tests demand several minutes to run? Is it possible to find an on-site customer that will faithfully represent the future users of the system? If the answer to any of these questions is yes, than XP is probably not a good choice for this project or the methodology will need to be adapted significantly.

4 Adapting XP for the Classroom

In an educational environment, not all of the aspects of a real production environment are present. Thus, when teaching XP in the university or in corporate training, some adaptations are required.

Differently from most academic courses, an XP course must focus on practice rather than on theory. Students must spend most of the time programming in the lab, not attending lectures. We identified two types of courses that can produce good results: short courses and long courses.

In an academic environment, a long course would typically be a full-semester course in which the students attend, initially, a few lectures describing the methodology and then spend 3 to 4 months working in the lab, 2 to 4 sessions per week. A short course can range from a full-day, 6 hour workshop in which the students are exposed to both theoretical and practical aspects of XP up to a one-month Summer course in which more details can be covered.

In industrial environments, the long "course" takes the form of *mentoring*. In this case, an experienced XP consultant spends several hours per week working in a real project of interest to the company, acting as the team *coach*. The role of the coach is not to guide the development but to make sure that all XP practices are being followed and to use its experience to resolve conflicts and show the group how XP can help overcoming the difficulties that arise. After a few months, the role of coach can be handed over to one of the developers and the consultant becomes a *meta-coach*, gradually decreasing his/her responsibilities. It is often said that the job of an XP consultant is to put himself out of business in the long run by empowering the team to work by itself using XP.

Short courses in industrial environments typically take the form of immersion workshops in which developers spend 2 to 4 days working full-time in a simple project going through all the steps of an XP project, producing a few releases of working software.

Except from the mentoring case, which can mimic a production environment perfectly, the other cases may require some adaptations. The time span of the courses are very different from a real software development project. There might not be a real customer available. The same person (e.g., instructor or professor) may need to play the role of both coach and customer, which is probably not a good idea.

All these issues must be analyzed carefully by the instructor to enable the course participants to have an XP experience as real as possible so that they will be capable of applying the methodology in real life afterwards.

Our experience shows that, with proper planning, it is possible to overcome all these difficulties and provide a real XP experience to students. In the next section we describe some of the long and short curses we carried out in both academic and industrial environments.

[2]That does not go well with collective code ownership.

5 Experiences Teaching XP

Over the last few years we had experiences in teaching the XP methodology at the University of Illinois at Urbana-Champaign, at the University of São Paulo, and in work as consultants both in the United States and in Brazil. This wide variety of previous experiences let us have a broad view of what is teaching the methodology to different people inserted in different cultures.

In this section, we describe our experiences in a full-semester course at the University of São Paulo, in consulting for the Illinois Department of Public Health, and in a shor-term course for a private company.

5.1 University of São Paulo

We started to disseminate the use of XP in Brazil in early 2001 with a series of lectures about different aspects of XP including an overview of the methodology, refactoring, debugging, testing, and coding style[3]. Besides these individual lectures, we hold an annual 4-month course called *Extreme Programming Laboratory*. This course is for undergraduate students in the 3rd and 4th year of the Bachelors program in Computer Science. Course attendance is limited to 20 students and they are divided in groups with 6 to 10 students. This is *per se* something new for the students since they usually never have an opportunity to work in such a large group in which all the participants actually work. In fact, most courses discourage students working together. The rest of this section will describe the most important aspects that must be addressed when implementing such a course.

Workload. The students were required to be in the lab during two weekly sessions lasting 2 to 3 hours. We found that 3-hour sessions are much more productive in general. However, due to schedule restrictions, we were forced to have 2-hour sessions in some cases. A good way of keeping the students for a longer period in the lab was to provide a modest lunch in the lab. Thus the students could stay focused on programming for longer periods holding their sandwiches while pair programming. Having food around a software development lab is considered important by many researchers who say that people become more relaxed and communicate better while eating.

Besides these two mandatory sessions, it was suggested to the students that they should come to the lab 2 to 4 additional hours per week for pair programming or to learn about the technologies used in the project. These additional hours were not verified by the instructors.

Development site. The laboratory in which the project is developed followed some guidelines which enabled a large level of osmotic communication [Coc02] among the members of the same team and a few, smaller, communication channels across teams. Alistair Cockburn has studied and experimented with many different room layouts and identified their advantages and drawbacks (see [Coc02], Chapter 3, *Communicating, Cooperating Teams*).

The University of São Paulo lab where the XP courses are carried out was reorganized to follow these guidelines. As shown in Figure 1, the workstations are set so there is space for two people siting in front of each one and all the members of the team sit facing each other. This contrasts with many laboratories where the developers face a wall or in which workstations are separated by divides or enclosed in cubicles. The two groups working in the same lab are partially separated from each other by two whiteboards, one for each group, which they use to draw UML diagrams, notes, etc. as shown in Figure 2. Whiteboards act as what Cockburn calls *information radiators* [Coc02] that can be seen and accessed easily by anyone entering the room. A large wall space is reserved for another kind of information radiator: posters taped to the wall showing information posted by the trackers (see below) about project progress (see Figure 3). The type of information posted was chosen by the students themselves and it includes a list of story cards and related information, graphs showing

[3] Slides, in Portuguese, available at `http://www.ime.usp.br/~xp`.

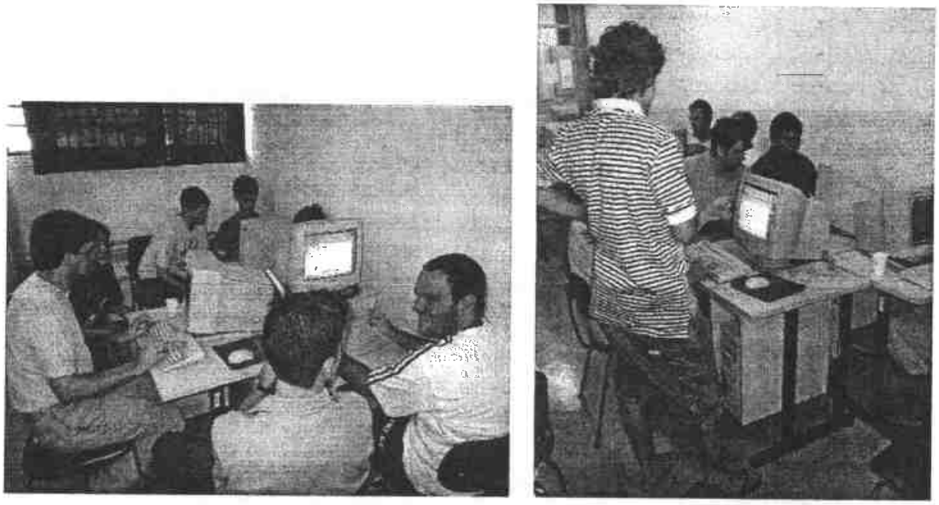

Figure 1: XP students in the lab

number of unit tests written and number of user stories implemented, and subjective evaluations of source-code quality and team productivity.

Coaching. We learned that choosing a good coach is very important for the success of an XP course. Over the years, we tried three different options for coach: a professor knowledgeable in XP, a graduate student that had attended the same course years before, and one of the students taking the course and being a novice in XP. We found that the best experience happened when the coach had both an authority over the students and were knowledgeable in XP. The conjunction of these two factors happened only when the professor was the coach. Nevertheless, we do believe that the other cases are also viable and, with proper care, can lead to good results; one must make sure that the two requirements are met (authority and knowledge of the methodology).

Customer. In the two initial years, the role of the customer was also played by CS professors. They were available during the two mandatory sessions and would be real users of the system to be built. In 2003, we developed a library management system so we invited a professor from another area and some staff members of our university library to act as customers. The experience was effective and very enlightening since the students realized that they had to use a completely different language to communicate with people that were not educated in CS.

Choosing the system. The choice of which system to build is very important: it must motivate the students, it must be interesting from a technological point of view, and it must be so that we can find real future users that can act as customers. To meet all these requirements, we chose systems that the university needed to manage its resources and people.

We started with a Web-based system for managing course selection; the students could to use it to express which elective courses they would like to take and the professors could express which courses they were able to teach and which one they would like to teach. The system then collected the results and were supposed to use optimization techniques to create a course schedule for the following year. The system is now online at `mico.arca.ime.usp.br` and is used every year.

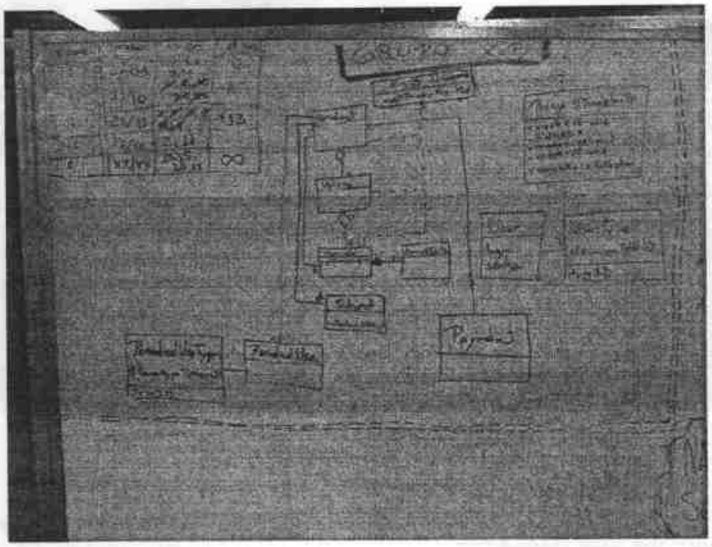

Figure 2: Whiteboard dividing the space

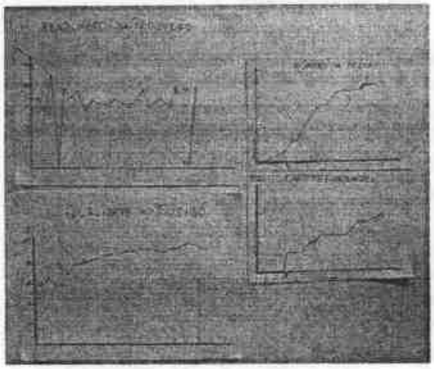

Figure 3: Information radiators maintained by the tracker

Tracker. In XP, the tracker is one of the developers who is responsible for collecting statistics about the performance of the team and act as its conscience, evaluating the progress of the project. Asking one student from each team to volunteer to be the tracker showed to be a very successful approach. By doing this we usually got people that were really motivated for this task.

After the trackers are selected they are asked to read a few articles and book chapters on tracking and try to come up with creative ways of capturing team progress. Most of the times, the tracker chose to keep a copy of the story cards in a Web site[4] so that team members could easily access them from any location at any time. The trackers are also responsible for maintaining the information radiators posted to the walls as we mentioned before.

Technologies. For developing the projects, students utilize the latest real-world technologies, which is very valuable for their future professional life. Most students consider this challenge motivating and work hard to learn the new tools.

The professors teaching the course do not specify any specific tool, language or environment for the system to be developed. All the decisions are made by the team itself during the initial exploratory phase.

The systems were developed using mostly modern free software tools such as Java, Eclipse, CVS, ant, Apache, Tomcat, JSP/Struts, PostgreSQL, and Checkstyle. For unit testing, JUnit has been used in all our projects. User acceptance tests verifying the correct behavior of Web interfaces were carried out using HHTPUnit. Server-side Java code, such as Servlets, EJBs, and Tag Libs, has been tested using Cactus.

Student grading. Grades in Brazil are a numeric value between 0 and 10. We chose not to apply any exam during the XP course. So, the grades are calculated, at the end of the semester, based on four weighted criteria: attendance (30%), commitment to the XP methodology (35%), quality of the software produced (25%), and self-evaluation (10%).

The weights show that what is more important for us is that the students do come to all programming sessions and that the XP methodology be applied. Simply developing a good software system without using XP is not the objective of the course and this is made clear since the beginning of the course.

5.2 Illinois Department of Public Health

In 1998 and 1999, The Refactory, Inc. provided a team to the Illinois Department of Public Health (IDPH) in order to assist with the development of medical software. Many applications at the Illinois Department of Public Health manage information about patients and people close to the patient, such as parents, children, and doctors. The programs vary in the kind of information (and the representation) they manage. However, there are core pieces of information that are common among the applications and can be shared among applications.

IDPH recruited The Refactory to assist with the development of an Enterprise framework for creating these medical applications with the primary goals of 1) achieving reuse, 2) creating easier and quicker ways to deploy applications, and 3) to share common data across applications.

The primary development environment was Smalltalk, which was used for creating Windows-based client-server applications that interacted with a relational database running on a UNIX box. Joseph Yoder was the main software architect and led a 10-person team using XP practices (though not pure XP). This section will describe the experiences at attempting to integrate XP into IDPH and will point out some success and problems associated with incorporating XP into industry.

Open Space. IDPH used cubicles for each developer. One of the first thing that we did was to remove the cubicles and create a shared common space. This common space was for us to pair

[4]See, for example, `http://www.ime.usp.br/~xp/2003/xops/storycards`.

program and to communicate more openly. We setup our workspaces in the open area to allow two people to share a single computer, primarily to facilitate pair programming. We put tables in the middle of our open area where we could gather around and share ideas. We also added a couple of whiteboards in which we could openly get together for brainstorm or shared design.

By setting up a shared area, we created an environment for good communication among developers. When new ideas were presented or new code released, everyone in our area immediately knew about it. This helped on the integration of new code. Refactoring was also easier since everyone had immediate access to all of the developers.

However, in general, most of the IDPH staff was uncomfortable with the open space idea. Our open space was almost too open. We were so open that all people within the IDPH staff could see what we were doing and hear us. Thus, we were too visible. We had some advocates to support what we were doing, but staff members not directly involved with our project were not as open to what we were doing. This sometimes stirred some inner controversy. For example, people would hear us talking amongst each other and at times our conversations would be misinterpreted.

For example, comments would be made about how we might be wasting time talking about items that were not directly related to our project. It is a common social phenomenon for people to discuss many items while dialoguing and quite often, people not involved with our project would complain to management that we were wasting time. They did not see the additional benefit that was created from the open social environment.

Because of this, the mentor quite often had to protect or defend what the group was doing. In a sense, the mentor had to isolate or protect the team from the rest of the IDPH staff. It might have been better if we had isolated our shared space from the rest of IDPH. For example, we could have been in a separate room rather than in the middle of a large space. This would provide for an open environment that was still private from the rest of the organization. We could then prove our concept by our deeds rather than someone judging our process while in action.

However, everyone that was part of our shared spaced really liked what we were doing. We all felt that we were more productive in this environment based upon our previous experiences. We felt that we achieved more and had fun while we were doing it. We do not have empirical data backing up our feeling but each of us have had enough development experience to believe strongly that we produced more higher quality production code in this environment than the old "cubicle" style of developing software.

Pair Programming. Pair programming was used to develop most of the production code. Pair programming worked extremely well for us as knowledge about our frameworks was shared. Also, we all had an understanding of all of the code and we were never dependent on a single individual. We had people both leave the project and join our group. Because of pair programming we had a group understanding of the code and was able to adapt to changes in our development team. Pair programming also provided new developers with good support for learning how to use our frameworks, thus becoming more productive in a less amount of time.

We would let individuals develop some spike solutions and what they developed would often be good enough to be incorporated into the code-base without pair programming. However, this code was only released with test cases and once the code was released into the the shared repository, anyone could change the code. Therefore, there was no explicit code-ownership. Everyone "owned" the code and we worked together to make sure that we never left anything broken.

One thing we did not do was the rotation of the pairs. Certain people tended to gravitate together and worked better together. We also had certain individuals that were very good at working with the spike solutions and integrating them into the environment. They worked well with the team but did not want to work in pairs.

Testing. Test cases and suites was an area where we went very "extreme". We generated many test cases and suites. Our test cases were not always created first but we were very diligent about creating tests to validate our code and also to show how to use our frameworks. We always made sure to run the test cases at the end of each day.

These tests were invaluable during refactoring and integration. We all became strong believers when the test cases pointed out problems while we were integrating new functionality. Problems that would not have normally been found until late in the game were immediately found and fixed. It also made us comfortable about refactoring the code. We could apply a design pattern such as applying the Template Method design pattern and know immediately if we broke someone's code.

One of the problems with building an application with our reusable frameworks was that our frameworks could be hard to understand and use. The tests provided a way to document how to use them, thus making it easier for developers to see how to use and build applications correctly with them.

Also, since we were using Smalltalk, we were able to evolve SUnit easily to make it so that we could create GUI tests. This allowed us to extend our test cases and suites to provide more extensive functional tests. We could then create complete user acceptance tests, thus ensuring the application worked according to the prescribed requirements.

Releases. We had regular internal releases and did what it took to keep a working version. This allowed us to demo the application often and get immediate feedback on what worked and what did not work. However, IDPH's process for releasing applications did not let us release our applications to state employees on a regular basis. We could use our working version to meet with users and show them the application working. However, we did not receive the additional benefits that arise from regular feedback provided by a real customer using a current released version of the application.

Since our releases were never released to the customer until we were near completion, we did not receive the benefits of the regular feedback that XP promises. This is one of the XP principles that can be difficult in industry. Many users may feel that it is a waste of their time dealing with applications that are not completely functional.

User Stories. We did not create formal user stories. This was due to the unfortunate fact that we did not have direct access to the users. Instead Joseph Yoder worked with the State Analysts to get the requirements and helped coordinate the team in an XP fashion. This is one of the biggest problems we had with our process.

For an application called The Refugee System, we had a customer that worked very closely with us. This helped to ensure that the system we developed was very close to what they needed. Therefore, when we were ready to go into production, the application pretty much met the needs of the end users.

However, we worked on another application called Newborn Screening (NBS), which had lots of problems. We did not have a relationship with the end user until the end of the development process. Because of the lack of a close relationship to a "real" customer, the system we developed was quite disparate from what the users needed. This led to many problems and complaints by the customers and management.

Upon reflection, we can see that a closer relationship with the customer was vital for success. Only relying on an analyst for the requirements was not good enough and by creating user stories for all of our applications, we might have been able to minimize some of the problems associated with NBS. Of course this is not unique to XP.

Assessment. In summary, we always kept things working, we were strongly test-driven, we did benefit from pair programming and the like. Our open space was invaluable to us though we would probably have benefited more by creating our open space in a semi-private area.

We wish we had pushed XP even further. However, it was hard to even push the principles as far as we did, given the political structure of a state organization. What we did worked well for us but our experience tells us that we know it could have worked even better.

For example, generating user stories and having regular releases could have helped ensure that our applications stayed on target. Rotating pairs would have helped shared knowledge more.

There are a couple variations on XP that might have helped more such as possibly creating a proxy customer and using this proxy to generate the user stories. For example, we know of one organization that has successfully used analysts that worked very closely with the customer as a user proxy. Then user stories were created as part of the XP process [TF00].

5.3 Recife Short Course

In August 2003, a company called Qualiti located in Recife presented a short XP course for industry professionals taught by Joseph Yoder. This section will outline how the course was presented along with some learning experiences.

Course Description The course was taught on site at Qualiti in Recife. We had 12 attendees which were from various areas of industry. The course duration was three four-hour days and its description was as follows:

> Evolving and adapting to changing requirements has become a crucial part of the design and programming process. Agile methods such as eXtreme Programming (XP) empowers all those that have an investment in the software being created. This ranges from the manager to the developer and end-user.
>
> This short course will teach attendees the basic premise of Agile methods and will explore the details of the XP process. The course will consist of a mixture of lectures, reading groups, dialogs, and labs. The attendees will read some online materials, discuss the details of the techniques, and apply them in a group setting.

The three-day course was broken down by presenting, on the first day, an overview of the XP process followed by two days of hands-on experiences actually working with the XP process.

Overview of XP. The first day really focused on ensuring that the students understood the main principles of XP and how the process worked. The first day overview presented: What is XP; Why XP; Principles of XP; The XP Process. This four-hour session emphasized issues such as the Customer Bill of Rights, the Programmer Bill of Rights, Rules and Practices of XP, and the overall process which included a detailed description of the iteration cycles and releases. We concluded this section with an introduction to the hands-on example that was worked on for the rest of the course.

Hands-on Example. The only way to really learn the principles of XP is by actually working with them. This is why any short course should have at least part of the course force students to actually try and work through some of the principles. The students were broken down into two six-person teams working through the XP process. We would meet at regular intervals to compare notes and to learn from each other.

The primary goal of this task was to put into practice some of the principles of XP. Some of the main principles of XP, that were described on the first day, included items such as:

- Get user stories from the customer
- Create acceptance tests
- Create spike solutions to understand the problem
- Create a system metaphor
- Work with the customer to create a release plan

- Do small iterations
  - Iterations include doing an iteration plan
  - Break the stories up into 1-3 day tasks
  - Do informal design such as CRC cards
  - Do test-driven development.

The example problem for practicing with XP dealt with the early design of a Conference Paper Submission System. The instructor knew this problem well and could thus work as the coach and customer; he ultimately really wants to build such a system to use for the Patterns Languages of Programming (PLoP) conferences.

The task included creating user stories, generating acceptance tests, outlining an architectural spike to get a system metaphor, creating a release plan, and working through the start of an iteration where they broke the story up into small 1-3 day tasks. They, then, did some initial design and outlined the unit tests for validating that the system would work properly. Rough requirements for the system were presented. This should be no surprise as this quite often happens in the real world. So, part of the task was to get better user stories from the user to make better estimates.

Analysis. Forcing the students to work through the process really emphasized how XP worked. The instructor could easily present a detailed overview of XP but many items were not understood until the students worked through the process. There was also a huge benefit from the students interacting with one another, specifically when we came together and compared the results of the two groups.

One thing that was noted from the students was that using CRC cards for the design was difficult. Many of the students already knew UML well and they could draw class diagrams more easily than trying to learn a new way to describe their objects. The course did not dictate CRC but most students wanted to try it so that they could understand it and compare it to methods they were familiar with. XP does not dictate CRC and encourage developers to use whatever works well for them as long as they do not over design.

The main problem that the students had was trying to limit their designs. The students that attended the course were all very sharp developers from industry that had quite a lot of experience developing production systems. Thus, when they would work on an iteration, the temptation would be to go ahead and add some extra complexity or over design knowing what some of the next iterations would need. This was when refactoring and keeping it simple was emphasized and it is a difficult point to make to experienced developers. It goes against what they have learned in the past and they will probably not be convinced until they see the results by working many months on a successful XP project. From this, we can conclude that short courses are useful for introducing the concepts in industrial settings. However, this should be followed up by a long-term mentoring process where a coach works a few days a month on-site with the XP team.

6 Open Problems

Perhaps the most difficult XP practice to teach is Metaphor. Although it usually does not receive the deserved attention, a good metaphor can be very important to improve the communication. We were not yet able to use the Metaphor practice consistently at the University of São Paulo courses, for example. A possible way of introducing the use of metaphors in an organization it to give a talk presenting some examples of good and bad metaphors and emphasizing its benefits. The keynote speech given by Kent Beck at OOPSLA 2002 [Bec02a] could be used as a starting point.

In an academic environment, another problem is related to the students motivation. Even if there is a selection of the more interested students in the beginning, this may change over the semester. For example, in our experience, problems related to other courses (midterm and final exams, exercises, etc.), personal problems, caused important interference in the development of some XP projects. Maybe this can be solved with shorter courses. However, since similar problems will also occur in real-life projects, it may be a good thing that they appear in the academic setting so that the participants learn to deal with them.

Finally, a problem that is often mentioned is the difficulty of performing unit tests in stand-alone applications based on GUIs. However, this problem will probably not last too long since the tools for testing graphical interfaces have been improving significantly in the last years.

7 Conclusions

Agile software development methodologies, such as XP, are gradually being adopted by hundreds of organizations in the five continents. Nevertheless, the spirit of agile development is still not present in most of the organizations developing software. A new culture of agility and adaptation to change must be developed and educators have a major role to play in this regard.

In this article, we have described our experiences in teaching XP in both academic and industrial environments and have discussed how one can be effective in teaching and implementing XP in an organization.

We have observed that, although there may be *a priori* fears of the consequences and effectiveness of XP, once developers and managers have real contact with a well-run XP project, the fears quickly dissipate. XP has proved to be a very attractive methodology both in academic and corporate environments due to the lack of surprises for customers and developers (thanks to the on-site customer practice) and to the high-quality of the software produced. Besides, the environment created is optimal for the developers who feel free to put all their energy in producing high-quality working code without the distractions required by bureaucratic processes that focus on tools and documents.

Within the next few years we expect that XP and agile methodologies will become part of the curriculum in many more universities around the world (at least as elective courses) and that industrial training and mentoring in XP will become more frequent. It is the role of educators and researchers to enable this leap forward.

Acknowledgments The authors gratefully acknowledge the energy, excitement and great ideas provided by all the students and professionals that participated in our courses and mentoring projects. We thank Carlos Eduardo Ferreira for being a good customer and coach, Klaus Wuestefeld for all the ideas shared with us and Eduardo Seiti Teruiya for driving out fear and embracing change. We also thank Paulo Borba, Ralph Johnson, Steven Wingo. Alfredo Goldman, Fabio Kon and Paulo J. S. Silva were supported by CNPq under the grants 303246/2002-2, 302455/2003-5, and 304691/2002-0.

References

[ADW01] O. Astrachan, R. Duvall, and E. Wallingford. Bringing extreme programming to the classroom. In *Proceedings of XP Universe 2001*, Raleigh, NC, USA, 2001.

[B+01] Kent Beck et al. Manifesto for Agile Software Development. Home page: `http://agilemanifesto.org`, 2001.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[Bec02a] Kent Beck. The metaphor metaphor. Keynote speech - ACM OOPSLA'02, November 2002.

[Bec02b] Kent Beck. *Test-Driven Development: By Example.* Addison-Wesley, 2002.

[Coc02] Alistair Cockburn. *Agile Software Development.* Addison-Wesley Longman, 2002.

[Cro04] Ron Crocker. *Large-Scale Agile Software Development.* Addison-Wesley, 2004.

[CW00] A. Cockburn and L. Williams. The costs and benefits of pair programming. In *Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2000)*, Cagliari, Sardinia, Italy, June 2000.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[McB03] P. McBreen. *Questioning Extreme Programming.* Addison Wesley, 2003.

[MS99] K. Maruyama and K. Shima. Automatic method refactoring using weighted dependence graphs. In *Proceedings of the 21st international conference on Software engineering*, pages 236–245. IEEE Computer Society Press, 1999.

[NWW+03] N. Nagappan, L. Williams, E. Wiebe, C. Miller, S. Balik, M. Ferzli, and M. Petlick. Pair learning: With an eye toward future success. In *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, volume 2753 / 2003 of *Lecture Notes in Computer Science*, pages 185 – 198. Springer-Verlag Heidelberg, September 2003.

[SB01] Ken Schwaber and Mike Beedle. *Agile Software Development with SCRUM.* Prentice Hall, 2001.

[TF00] C. Taber and M. Fowler. An iteration in the life on an XP project. *Cutter IT journal*, 13(11), November 2000. Updated eletronic version: `http://www.martinfowler.com/articles/planningXpIteration.html`.

[Tom02] I. Tomek. What i learned teaching XP. In *Proceedings of the ACM OOPSLA Educators Symposium*, pages 39–46, Seattle, Washington, USA, November 2002.

[Wil01] D. Wilson. Teaching XP: a case study. In *Proceedings of XP Universe 2001*, Raleigh, NC, USA, 2001.

[WK00] L. A. Williams and R. R. Kessler. All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM*, 43(5):108–114, May 2000.

[WK02] L. Williams and R. Kessler. *Pair Programming Illuminated.* Addison-Wesley, 2002.

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 2000 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail (mac@ime.usp.br).

Marcelo Finger and WanbertoVasconcelos
SHARING RESOURCE-SENSITIVE KNOWLEDGE USING COMBINATOR LOGICS
RT- MAC-2000-01, março 2000, 13pp.

Marcos Alves e Markus Endler
PARTICIONAMENTO TRANSPARENTE DE AMBIENTES VIRTUAIS DISTRIBUÍDOS
RT- MAC-2000-02, abril 2000, 21pp.

Paulo Silva, Marcelo Queiroz and Carlos Humes Junior
A NOTE ON "STABILITY OF CLEARING OPEN LOOP POLICIES IN MANUFACTURING SYSTEMS"
RT- MAC-2000-03, abril 2000, 12 pp.

Carlos Alberto de Bragança Pereira and Julio Michael Stern
FULL BAYESIAN SIGNIFICANCE TEST: THE BEHRENS-FISHER AND COEFFICIENTS OF VARIATION PROBLEMS
RT-MAC-2000-04, agosto 2000, 20 pp.

Telba Zalkind Irony, Marcelo Lauretto, Carlos Alberto de Bragança Pereira and Julio Michael Stern
A WEIBULL WEAROUT TEST: FULL BAYESIAN APPROACH
RT-MAC-2000-05, agosto 2000, 18 pp.

Carlos Alberto de Bragança Pereira and Julio Michael Stern
INTRINSIC REGULARIZATION IN MODEL SELECTION USING THE FULL BAYESIAN SIGNIFICANCE TEST
RT-MAC-2000-06, outubro 2000, 18 pp.

Douglas Moreto and Markus Endler
EVALUATING COMPOSITE EVENTS USING SHARED TREES
RT-MAC-2001-01, janeiro 2001, 26 pp.

Vera Nagamura and Markus Endler
COORDINATING MOBILE AGENTS THROUGH THE BROADCAST CHANNEL
RT-MAC-2001-02, janeiro 2001, 21 pp.

Júlio Michael Stern
THE FULLY BAYESIAN SIGNIFICANCE TEST FOR THE COVARIANCE PROBLEM
RT-MAC-2001-03, fevereiro 2001, 15 pp.

Marcelo Finger and Renata Wassermann
TABLEAUX FOR APPROXIMATE REASONING
RT- MAC-2001-04, março 2001, 22 pp.

Julio Michael Stern
FULL BAYESIAN SIGNIFICANCE TESTS FOR MULTIVARIATE NORMAL STRUCTURE MODELS
RT-MAC-2001-05, junho 2001, 20 pp.

Paulo Sérgio Naddeo Dias Lopes and Hernán Astudillo
VIEWPOINTS IN REQUIREMENTS ENGINEERING
RT-MAC-2001-06, julho 2001, 19 pp.

Fabio Kon
O SOFTWARE ABERTO E A QUESTÃO SOCIAL
RT- MAC-2001-07, setembro 2001, 15 pp.

Isabel Cristina Italiano, João Eduardo Ferreira and Osvaldo Kotaro Takai
ASPECTOS CONCEITUAIS EM DATA WAREHOUSE
RT – MAC-2001-08, setembro 2001, 65 pp.

Marcelo Queiroz , Carlos Humes Junior and Joaquim Júdice
ON FINDING GLOBAL OPTIMA FOR THE HINGE FITTING PROBLEM
RT- MAC –2001-09, novembro 2001, 39 pp.

Marcelo Queiroz , Joaquim Júdice and Carlos Humes Junior
THE SYMMETRIC EIGENVALUE COMPLEMENTARITY PROBLEM
RT- MAC-2001-10, novembro 2001, 33 pp.

Marcelo Finger, and Fernando Antonio Mac Cracken Cezar
BANCO DE DADOS OBSOLESCENTES E UMA PROPOSTA DE IMPLEMENTAÇÃO.
RT- MAC - 2001-11- novembro 2001, 90 pp.

Flávio Soares Correa da Silva
TOWARDS A LOGIC OF PERISHABLE PROPOSITIONS
RT- MAC- 2001-12 - novembro 2001, 15 pp.

Alan M. Durham
O DESENVOLVIMENTO DE UM INTERPRETADOR ORIENTADO A OBJETOS PARA ENSINO DE LINGUAGENS
RT-MAC-2001-13 – dezembro 2001, 21 pp.

Alan M. Durham
A CONNECTIONLESS PROTOCOL FOR MOBILE AGENTS
RT-MAC-2001-14 – dezembro 2001, 12 pp.

Eugênio Akihiro Nassu e Marcelo Finger
O SIGNIFICADO DE "AQUI" EM SISTEMAS TRANSACIONAIS MÓVEIS
RT-MAC-2001-15 – dezembro 2001, 22 pp.

Carlos Humes Junior, Paulo J. S. Silva e Benar F. Svaiter
SOME INEXACT HYBRID PROXIMAL AUGMENTED LAGRANGIAN ALGORITHMS
RT-MAC-2002-01 – Janeiro 2002, 17 pp.

Roberto Speicys Cardoso e Fabio Kon
APLICAÇÃO DE AGENTES MÓVEIS EM AMBIENTES DE COMPUTAÇÃO UBÍQUA.
RT-MAC-2002-02 – Fevereiro 2002, 26 pp.

Julio Stern and Zacks
TESTING THE INDEPENDENCE OF POISSON VARIATES UNDER THE HOLGATE BIVARIATE DISTRIBUTION: THE POWER OF A NEW EVIDENCE TEST.
RT- MAC – 2002-03 – Abril 2002, 18 pp.

E. N. Cáceres, S. W. Song and J. L. Szwarcfiter
A PARALLEL ALGORITHM FOR TRANSITIVE CLOSURE
RT-MAC – 2002-04 – Abril 2002, 11 pp.

Regina S. Burachik, Suzana Scheimberg, and Paulo J. S. Silva
A NOTE ON THE EXISTENCE OF ZEROES OF CONVEXLY REGULARIZED SUMS OF MAXIMAL MONOTONE OPERATORS
RT- MAC 2002-05 – Maio 2002, 14 pp.

C.E.R. Alves, E.N. Cáceres, F. Dehne and S. W. Song
A PARAMETERIZED PARALLEL ALGORITHM FOR EFFICIENT BIOLOGICAL SEQUENCE COMPARISON
RT-MAC-2002-06 – Agosto 2002, 11pp.

Julio Michael Stern
SIGNIFICANCE TESTS, BELIEF CALCULI, AND BURDEN OF PROOF IN LEGAL AND SCIENTIFIC DISCOURSE
RT- MAC – 2002-07 – Setembro 2002, 20pp.

Andrei Goldchleger, Fabio Kon, Alfredo Goldman vel Lejbman, Marcelo Finger and Siang Wun Song.
INTEGRADE: RUMO A UM SISTEMA DE COMPUTAÇÃO EM GRADE PARA APROVEITAMENTO DE RECURSOS OCIOSOS EM MÁQUINAS COMPARTILHADAS.
RT-MAC – 2002-08 – Outubro 2002, 27pp.

Flávio Protasio Ribeiro
OTTERLIB – A C LIBRARY FOR THEOREM PROVING
RT- MAC – 2002-09 – Dezembro 2002 , 28pp.

Cristina G. Fernandes, Edward L. Green and Arnaldo Mandel
FROM MONOMIALS TO WORDS TO GRAPHS
RT-MAC – 2003-01 – fevereiro 2003, 33pp.

Andrei Goldchleger, Márcio Rodrigo de Freitas Carneiro e Fabio Kon
GRADE: UM PADRÃO ARQUITETURAL
RT- MAC – 2003-02 – março 2003, 19pp.

C. E. R. Alves, E. N. Cáceres and S. W. Song
SEQUENTIAL AND PARALLEL ALGORITHMS FOR THE ALL-SUBSTRINGS LONGEST COMMON SUBSEQUENCE PROBLEM
RT- MAC – 2003-03 – abril 2003, 53 pp.

Said Sadique Adi and Carlos Eduardo Ferreira
A GENE PREDICTION ALGORITHM USING THE SPLICED ALIGNMENT PROBLEM
RT- MAC – 2003-04 – maio 2003, 17pp.

Eduardo Laber, Renato Carmo, and Yoshiharu Kohayakawa
QUERYING PRICED INFORMATION IN DATABASES: THE CONJUNTIVE CASE
RT-MAC – 2003-05 – julho 2003, 19pp.

E. N. Cáceres, F. Dehne, H. Mongelli, S. W. Song and J.L. Szwarcfiter
A COARSE-GRAINED PARALLEL ALGORITHM FOR SPANNING TREE AND CONNECTED COMPONENTS
RT-MAC – 2003-06 – agosto 2003, 15pp.

E. N. Cáceres, S. W. Song and J.L. Szwarcfiter
PARALLEL ALGORITMS FOR MAXIMAL CLIQUES IN CIRCLE GRAPHS AND UNRESTRICTED DEPTH SEARCH
RT-MAC – 2003-07 – agosto 2003, 24pp.

Julio Michael Stern
PARACONSISTENT SENSITIVITY ANALYSIS FOR BAYESIAN SIGNIFICANCE TESTS
RT-MAC – 2003-08 – dezembro 2003, 15pp.