Experiences Tracking Agile Projects: an Empirical Study

Danilo Sato, Dairton Bassi, Mariana Bravo, Alfredo Goldman & Fabio Kon

Department of Computer Science
University of São Paulo
Rua do Matão, 1010
Phone: +55 (11) 30916134 (FAX)
Zip 05508-090 - São Paulo - SP - BRAZIL
{dtsato | dairton | marivb | gold | kon}@ime.usp.br

Abstract

In this article, we gather results from several projects we conducted recently that use some kind of agile method. We analyze both academic and governmental software development projects, some of them using agile methods since the beginning and others in which agile methods were introduced afterwards. Our main goals are to classify the different projects, and to analyze the collected data and discover which metrics are best suited to support tracking an agile project.

We use both quantitative and qualitative methods, obtaining data from the source code, from the code repository, and from the feedback received from surveys and interviews held with the team members. We use various kinds of metrics such as lines of code, number of tests, cyclomatic complexity, number of commits, as well as combinations of these.

In this article, we describe in detail the projects, the metrics, the obtained results, and their analysis from our main goals standpoint, providing guidelines for the use of metrics to track an agile software development project.

Keywords: Agile Methods, Extreme Programming, Software Engineering, Tracking

1. Introduction

Agile Methods are becoming more popular since Extreme Programming was introduced by Kent Beck in 1999 [3]. Agile Methods propose a new way of looking into software development, focusing the attention on the interactions between people collaborating to achieve high productivity, delivering high-quality software. The approach to obtain these results is based on a set of simple practices that provide enough feedback to enable the team to know where they are and to find the best way to move towards an environment of continuous improvement.

One of the practices that enable the creation of such environment is called *tracking*. Kent Beck describes the role of a *tracker* in an XP team as someone responsible for frequently gathering metrics with data provided from the team members and for making sure that the team is aware of what was actually measured [3, 4]. These metrics should be used as guidelines for improvement and to point out, to the team, any current problems. It is not an easy task to decide which information to collect and present to the team and how to present it. Moreover, some of the development problems are not easily recognizable from quantitative data alone.

We have been experimenting with various tracking methods since early 2002 obtaining good results in different kinds of projects. In this article, we describe an empirical case study we conducted on seven of these projects, gathering and analyzing data from both academic and industrial environments.

We use both quantitative and qualitative metrics, collecting data from different sources, such as the source code, the code repository, and subjective feedback received from surveys and interviews held with team members. The qualitative metrics were gathered at the end of the first semester of 2006, while the quantitative metrics were retroactively collected from the source code repository and from the XPlanner agile management tool. Due to the empirical and observational nature of our study we classify it as *parallel/simultaneous* [39] as both types of metrics were gathered at the same time.

In this study, we are not comparing agile and non-agile projects. Our main goals are:

- to classify the projects according to the XP Evalua-

tion Framework (described in Section 3), and

- to analyze the collected data and discover which metrics are best suited to support tracking an agile project.

The remainder of this article is organized as follows. Section 2 provides an overview of Agile Methods focusing on Extreme Programming and presenting our approach. Section 3 describes seven software development projects analyzed in this article. Section 4 presents the methods that we used to collect data and the metrics chosen to be analyzed. Section 5 analyzes our empirical results and provide some new metrics to support the *tracker* of an agile project. Finally, we conclude in Section 6 providing guidelines for future work.

2. Agile Methods

In the last few years, Agile Methods for software development have gained importance in many segments of the software industry. Like conventional methods, the goal of Agile Methods is to build high quality software that meets users needs. The main difference between them lies in the means to achieve this goal, that is, in the principles used.

In software development, requirements for a project often change while implementation is still in progress. Kajko-Mattson et al. shows that the percentage of software lifecycle cost due to maintenance ranges between 40 and 90% [23]. Many companies and development teams consider changes undesirable because they break previous plans. However, requirements are likely to change as the customer sees the system after it is deployed and it is not feasible to make an initial plan that foresees such changes without having to spend too much time and money.

With Agile Methods, more value is given to practical aspects. Detailed plans are made only for the current development stage, and draft plans are created for the next phases in such a way that they can be adapted to changes when the time comes to detail and execute them. The Manifesto for Agile Software Development [5] defines the essential principles of Agile Methods and highlights the differences between agile and conventional methods by valuing:

- **Individuals and interactions** over processes and tools.
- **Working software** over comprehensive documentation.
- **Customer collaboration** over contract negotiation.
- **Responding to change** over following a plan.

These characteristics bring dynamism to development, motivation to the team, and more comprehensive information about the real situation of the project for the customer. Some conventional methods claim to be predictive. In contrast, Agile Methods are adaptive. This approach is fundamental for the success of most projects because requirements change as the customer needs change.

There are several Agile Methods based on the principles proposed by the Agile Manifesto, including: Extreme Programming (XP) [3, 4], Scrum [35], Crystal Methods [11], Lean Software Development [33], Feature Driven Development (FDD) [32], Adaptive Software Development [20], and Dynamic System Development Method (DSDM) [37]. They are used in small, medium, and large companies, universities, and governmental agencies to build different types of software and have produced excellent results extensively described in the literature [8, 15, 27, 28, 31].

2.1. Extreme Programming

The most well-known Agile Method is Extreme Programming (XP) [3, 4]. It was developed by Kent Beck after many years of experience in software development. He defined a set of values and practices that improve the productivity of a software development team and the quality of their work.

XP is based in five main values: Communication, Simplicity, Courage, Feedback, and Respect. XP values are translated to the team as a set of practices that should be followed during development. The practices proposed initially [3] are briefly described below.

Planning Game: In the beginning of the project, customers write story cards that describe the requirements. During the Planning Game, programmers and customers collaborate to select a subset of the most valuable stories: programmers provide estimates for each story while customers define the business value of each story. The decisions made at this stage are just an initial plan. Changes are welcome to be incorporated into the plan as the iteration and release progresses. The Planning Game happens at two levels: in the beginning of a release and in the beginning of each iteration. A release is composed of a series of iterations. At the end of each iteration a subset of the release stories should be implemented, producing a deployable candidate version of the system.

Small Releases: The team must implement and deliver small pieces of working software frequently. The set of story cards selected during the Planning Game should contain just enough functionality, so that the customer priorities as well as the developers estimates are respected. The iteration length should be constant throughout the project, lasting one to three weeks. This is the heart beat of the project and provides constant feedback to the customer and to the team. Programmers should not try to

anticipate requirements, i.e., they should never add unnecessary flexibility to complete their current task. XP coaches often say: "do the simplest thing that could possibly work".

Simple Design: Simplicity is a key concept that allows a system to respond to changes. To minimize the cost of design changes one should always implement the simplest – but not the most simplistic – design, with only the minimum level of complexity and flexibility needed to meet the current business needs. As development is incremental, new functionalities will be added later so improving the design through refactoring is always necessary.

Pair Programming: Developers work in pairs to perform their programming tasks. This promotes collective and collaborative work, brings the team together, and improves communication and code quality. Pairs should switch frequently, even several times a day. Generally, the selection of pairs depends on the task, the availability of programmers, and the expertise of each one. The goal is to spread the knowledge of the whole system to all team members.

Testing: The software is constantly verified by a set of automated tests written by programmers and customers. The development team writes unit tests for all system components and runs them many times a day to assure that recently-added functionalities do not break existing code. Customers write acceptance tests to assure the system does exactly what they want. These tests are executed whenever a new functionality is finished and determine when a story card is completed. They can be written by a programmer pairing with the customer if he lacks the technical knowledge to do it.

Refactoring: Refactoring is a systematic technique for restructuring existing source code, altering its internal structure without changing its external behavior [14]. Among the most common refactorings we can mention the removal of duplicated code, the simplification of a software architecture, and the renaming of elements such as classes, methods, and variables to make the code easier to understand. The goals are always to make the source code more readable, simpler, cleaner, and more prepared to accommodate changes.

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. Developers upload and download code from the repository several times a day so all team members work with an up to date, synchronized version of all of the code.

Collective Code Ownership: There is no concept of individual ownership of code. The code base is owned by the entire team and anyone may make changes anywhere. If a pair of programmers identify a good opportunity to simplify any piece of code or remove duplication, they should do it immediately without having to ask permission for the original writer. With Collective Code Ownership the team members have a broad view of the entire system, making small and large refactorings easier.

Sustainable Pace: The working rhythm should not affect the participant's health or their personal lives. During planning, the amount of hours dedicated to the project must be defined realistically. It is acceptable for the team to work overtime in rare occasions. But an extensive and recurrent overload of work will reduce code quality and lead to great losses in the long run.

On-site Customer: The team must be composed of a variety of people with a broad knowledge and experience across the necessary skills for the project to succeed. This must include a business representative – called the *customer* – who understands the business needs and has enough knowledge about the real users of the system. The customer is responsible for writing stories, setting priorities, and answering any doubts that the programmers might have.

Metaphor: All members of the project, including programmers and customers, should find a common language to talk about the system. This language should be equally understood by technical people and business people. This can be achieved by adopting a metaphor that relates system abstractions to real-world objects in a certain domain. This may be the most difficult practice to introduce in an inexperienced team because it is directly related to communication and to how comprehensive people will be when they share their ideas and wishes.

Coding Standards: Before starting to code, all programmers must agree upon a set of standards to be used when writing source code. It makes the source easier to understand, improves communication, and facilitates refactoring.

After some time, practitioners realized that applying all the practices "by the book" without considering the principles and values behind them was not always an effective approach. After all, Agile Methods should be adaptive rather than prescriptive. This observation led to the creation of a new practice, called **"Fix XP when it breaks"**, meaning that each team has to consider the agile principles and values and adapt the practices for their specific environment. Thus, XP practices should not be seen as dogmas, but rather as guidelines for organizing the behavior of a team and a basis for continuous reflection.

There are also two special roles in an XP team, that can be reassigned to different developers during the course of the project:

- **Coach**: Usually the most experienced programmer, who is responsible for verifying whether the team members are following the proposed practices and

ensuring that the methodology is being followed.

- **Tracker**: Jeffries [21] describe the role of a *tracker* as the developer responsible for providing information about the project progress through the use of appropriate metrics. The *tracker* is responsible for creating charts and posters to show points of improvement, regularly spreading this information in the walls, what Cockburn calls *information radiators* [10].

2.2. Extreme Programming - Second Edition

In 2004, Kent Beck published with his wife, Cynthia Andres, the second edition of the book that first introduced XP [4] five years before. The book was restructured in a more inclusive way, focusing on the importance of the XP values. While the first edition was focused on 'what' XP was, the second edition talks a lot more about the 'why' of XP, introducing principles that helps translate values into practices, tailoring the process to a project's specific needs. The fourteen principles proposed in the second edition are: humanity, economics, mutual benefit, self-similarity, improvement, diversity, reflection, flow, opportunity, redundancy, failure, quality, baby steps, and accepted responsibility.

As discussed in Section 2.1, teams that started implementing XP usually ended up using new or changing the original practices. In the second edition of the book, Kent Beck breaks down the 12 original practices into two categories: *primary practices* are useful independent of the context of the project, while *corollary practices* are likely to be difficult without previous experience with the *primary practices*. The full description of each practice and principle is out of the scope of this article, but the next paragraph provides a list of the new practices and Table 1 compares the original practices with their new correspondent.

The thirteen *primary practices* are: **Sit Together**, **Whole Team**, **Informative Workspace**, **Energized Work**, **Pair Programming**, **Stories**, **Weekly Cycle**, **Quarterly Cycle**, **Slack**, **Ten-Minute Build**, **Continuous Integration**, **Test-First Programming**, and **Incremental Design**. The eleven *corollary practices* are: **Real Customer Involvement**, **Incremental Deployment**, **Team Continuity**, **Shrinking Teams**, **Root-Cause Analysis**, **Shared Code**, **Code and Tests**, **Single Code Base**, **Daily Deployment**, **Negotiated Scope Contract**, and **Pay-Per-Use**.

[1] The following practices did not change and were not included in the table: **Pair Programming**, **Continuous Integration**, and **Test-First Programming**

Comparison table between XP practices [1]	
First Edition	**Second Edition**
Planning Game	Stories, Weekly Cycle, Quarterly Cycle, and Slack
Small Releases	Weekly Cycle, Incremental Deployment, Daily Deployment
Simple Design	Incremental Design
Refactoring	Incremental Design
Collective Code Ownership	Shared Code, Single Code Base
Sustainable Pace	Energized Work, Slack
On-site Customer	Whole Team, Real Customer Involvement
Metaphor	(Incremental Design)
Coding Standards	(Shared Code)

Table 1. Comparison between XP practices from the first and second edition

2.3. Our Approach

Our approach is mostly based on XP practices. In most of the projects described in Section 3, we used all XP practices. The last practice, **"Fix XP when it breaks"** was necessary in a few cases, requiring small adaptations to maintain all values balanced. To attain higher levels of communication and feedback, we used frequently two other well known agile practices: **Stand-Up Meetings** and **Retrospectives** (also known as **Reflection Workshops**). The former consists of a daily, informal and short meeting at the beginning of the work day when each developer comments on three topics: what he did yesterday, what he intends to do today, and any problems that he might be facing (that will be discussed later). In these short meetings, all participants stand up to ensure it does not take too long.

Retrospectives are meetings held at the end of each iteration where the development method is evaluated, the team highlights lessons learned from the experience, and plan for changes in the next development cycle [24]. Kerth, on his prime directive of retrospectives says [24]:

> *"Regardless of what we discover, we understand and truly believe that everyone did the best job they could, given what they knew at the time, their skills and abilities, the resources available, and the situation at hand."*

There are many formats for Reflection Workshops, but the most usual is a meeting where the entire team discusses about "what worked well?", "what we should do differently?", and "what puzzles us?". Normally, at the end of a workshop the team builds up a series of actions in

each of those categories which they will prioritize and select from to implement in the next iteration. These actions are captured in a poster, which is posted in the workspace.

The projects described in this article were often composed of inexperienced teams on Agile Methods and sometimes they even did not have enough previous knowledge about the technologies, frameworks, and tools used to build the system. In this sense, they represent almost a worst-case scenario for deploying agile methods. An extreme difference between the team members expertise could harm their productivity [40]. In our approach, in order to reduce large technical differences, we introduced an extra phase at the beginning of the project: **Training**. During this phase, we tried to equalize the team members expertise by providing training classes for specific technologies, tools, frameworks, and agile practices that would be used during the implementation. The training was structured in classes that took about 10 to 20 hours, composed of presentations followed by practical exercises. It was not intended to be complete or to cover advanced topics, but to create a minimum common technical ground for the whole team from which they could improve.

3. PROJECTS

In this article, we analyze seven software development projects. Five of them were conducted in the academia in a full-semester course on Extreme Programming [16], and two of them were conducted in a governmental institution, the São Paulo State Legislative Body (ALESP). In this section, we describe the characteristics of each project with details about the environment, the team, as well as context and technological factors.

3.1. FORMAT OF PRESENTATION

All of the projects, except for the last one, were faithful to most XP practices, so we describe them in terms of the Extreme Programming Evaluation Framework (XP-EF) and its subcategories [41, 42]. The framework records the context of the case study, the extent to which the XP practices were adopted, and the result of this adoption. The XP-EF is composed of three parts: XP Context Factors (XP-cf), XP Adherence Metrics (XP-am), and XP Outcome Measures (XP-om).

XP-cf and XP-om are described in this section. XP-am were qualitatively collected by a subjective survey, constructed based in [25], and described in detail in Appendix A.

Some of the XP-cf are common between all projects and they will be described in Section 3.3. Factors that differ will be presented for each project in the following format:

- **Project Name**
- **Description**: a brief description of the goals of the project.
- **Software Classification**: as prescribed by Jones [22].
- **Information Table**: a table describing the sociological and project-specific factors. Information about the team size, education, and experience level was gathered from the survey results (Appendix A). Domain and language expertise was subjectively evaluated by the authors. Quantitative data such as number of delivered user stories and thousands lines of executable code (KLOEC) were collected respectively from XPlanner and from the code repository (metrics are described in detail in Section 4).
- **Polar Chart**: a five-axis polar chart describing the developmental factors suggested by Boehm and Turner as a risk-based method for selecting an appropriate methodology [6]. The five axes are used to select an agile, plan-driven, or hybrid process:
  - **Dynamism**: The amount of requirements change per month.
  - **Culture**: Measures the percentage of the team that prefers chaos versus the percentage that prefers order.
  - **Size**: The number of people in the team.
  - **Criticality**: The impact due to software failure.
  - **Personnel**: The percentage of the development team at the various Cockburn levels [10] described in Section 5.3.

  When the projects points are joined, the obtained shape provides visual information. Shapes distinctly toward the graph's center suggest using an agile method. Shapes distinctly toward the periphery suggest using a plan-driven methodology. More varied shapes suggest an hybrid method of both agile and plan-driven practices.

  With the exception of the team size, the value of these risk factors for each project was subjectively evaluated by the authors of this article.
- **Outcome Measures**: a table describing the available XP-om proposed by the XP-EF framework: productivity and morale. Productivity metrics were directly calculated from the project-specific XP-cf, such as thousand lines of executable code (KLOEC), person months (PM), and number of delivered user stories. The Putnam Productivity Parameter (PPP) is a LOC-based metric that was proposed based on historical

data from several projects, and taking team size and project duration into account as follows [34].

$$PPP = \frac{TLOC}{(\text{Effort}/\beta)^{\frac{1}{3}}\ (\text{Time})^{\frac{4}{3}}}$$

$TLOC$ is the total lines of code, Effort is the number of person months of work done in the project, β is a factor chosen from a table constructed by Putnam based on production data from a dozen large software projects [34], and Time is the number of elapsed months of the project.

In addition to the productivity metrics, a qualitative morale metric was gathered subjectively through a *niko-niko calendar*, described in Section 4.2. Also, more quantitative metrics are discussed in Section 4.1.

3.2. Considerations

The academic projects had a different work schedule from the governmental projects. Each week, the students were required to be in the lab for two sessions lasting two to three hours. Besides these two mandatory sessions, it was suggested that the students came to the lab for four extra hours of work; Even though these extra hours were not verified by the instructors, many of the students did come and work the extra hours. In average, each student worked a total of 6 to 8 hours per week. The work schedule for the governmental projects was different: each team member was required to work a total of 30 hours per week on the project.

In the academia, the full semester is considered to be a release that is developed in two to four iterations. We recommended the teams to have iterations no longer than one month but, depending on the experience of the team with the technologies and the class schedule on the semester, the exact duration of an iteration varied from team to team. Special considerations about the schedule for the governmental projects are discussed in Sections 3.5.1 and 3.5.2.

Also, we were not capable of collecting historical data related to defects and bugs. At the time of the writing of this article, none of the projects were yet deployed in production, except for Project 7 (described in Section 3.5.2). Project 7 did not have a bug tracking system in place and was already deployed when we started introducing some of the XP practices.

3.3. Common Context Factors (XP-cf)

Some of the XP-cf, such as the ergonomic, technological and geographical factors, are similar for all of the projects. They are described in Table 2 to avoid duplication of information.

Ergonomic Factors	
Physical Layout	Collocated in an open space.
Distraction Level of Office Space	Moderate, as there were other teams working closely in the same environment.
Customer Communication	Mostly on-site. Occasionally by e-mail.
Technological Factors	
Development Method	Primarily XP [2].
Language	Java and related technologies and frameworks (Struts, JDBC, Swing, etc.).
Tools	Eclipse, CheckStyle, CVS/Subversion, XPlanner, xUnit, Wiki.
Geographical Factors	
Team Location	Collocated.
Customer Location	Academia: on-site customer. Industry: customer representatives that work in the same building.

Table 2. Ergonomic, Technological and Geographical Factors (XP-cf)

3.4. Academic Projects

3.4.1. Project 1 - "Archimedes":

- **Project Name**: Archimedes
- **Description**: An open source CAD (computer-aided design) software focused on the needs of professional architects. This project started a few months before the first semester of the XP class, when the students involved in it started to gather requirements with architects and to recruit the team. Here, we analyze the first 4 iterations of the project, implemented during the first semester of 2006.
- **Software Classification**: Commercial software, developed as an open source project.

3.4.2. Project 2 - "Grid Video Converter":

- **Project Name**: GVC - Grid Video Converter
- **Description**: A Web-based application that offers to its users a computational grid to convert video files among several video encodings, qualities, and formats. The file is uploaded by a registered user, converted by the server using a multi-machine computational grid and later downloaded by the user. Here,

[2]In Project 7, just some of the XP practices were introduced during the course of our intervention. See more details in Section 3.5.2.

Sociological Factors - Project 1	
Team Size	8 + 2 Customers
Team Education Level	Undergrad students: 8
Experience Level of Team	< 5 years: 8
Domain Expertise	Low
Language Expertise	High
Project-specific Factors - Project 1	
Delivered User Stories	64
Domain	Stand-alone Application, CAD
Person Months	6.4
Elapsed Months	4
New & Changed KLOEC	18.832
System KLOEC	18.832
Outcome Measures - Project 1	
Productivity	
KLOEC/PM	2.943
User stories/PM	10
Putnam Productivity Parameter	0.902
Morale	97 %

Table 3. XP-cf and Available XP-om - Project 1

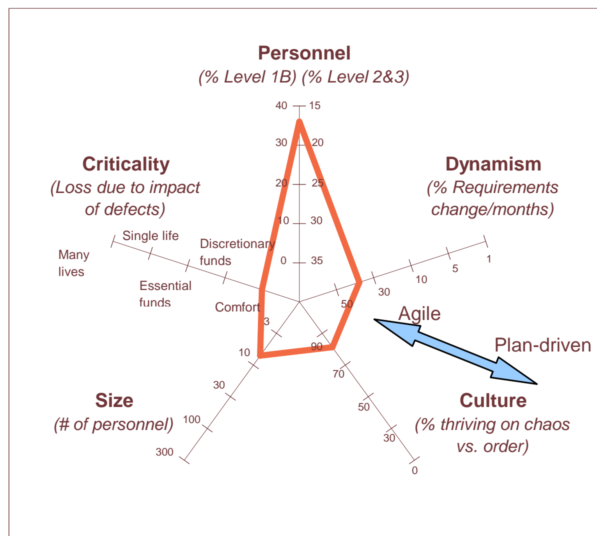

Figure 1. Developmental Factors (XP-cf) - Project 1

we analyze the first 3 iterations of the project, implemented during the first semester of 2006.

- **Software Classification**: End-user software

Sociological Factors - Project 2	
Team Size	6 (1 replaced mid-term) + 1 Customer
Team Education Level	Undergrad students: 2 Grad students: 4
Experience Level of Team	< 5 years: 4 5-10 years: 2
Domain Expertise	Low
Language Expertise	High
Project-specific Factors - Project 2	
Delivered User Stories	16
Domain	Web, Grid Computing, Video Conversion
Person Months	4.2
Elapsed Months	4
New & Changed KLOEC	2.535
System KLOEC	2.535
Outcome Measures - Project 2	
Productivity	
KLOEC/PM	0.604
User stories/PM	3.810
Putnam Productivity Parameter	0.134
Morale	64 %

Table 4. XP-cf and Available XP-om - Project 2

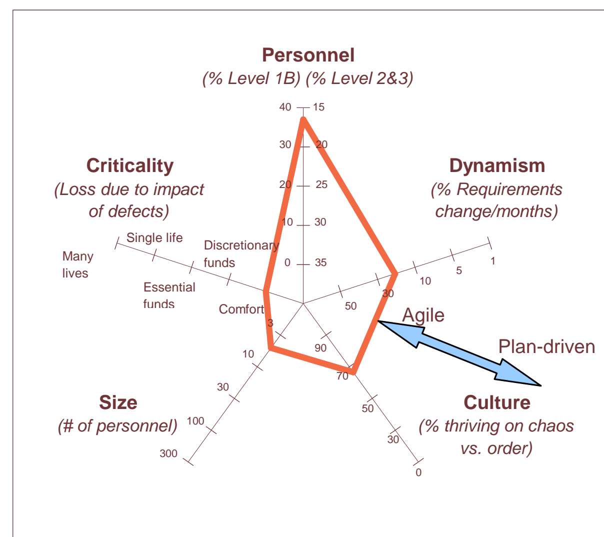

Figure 2. Developmental Factors (XP-cf) - Project 2

3.4.3. Project 3 - "Colméia":

- **Project Name**: Colméia (*Beehive*)

- **Description**: A complete library management system that has been developed at the University of São Paulo by undergraduate and graduate students during the last four offerings of the XP class. Here, we analyze the implementation of a new module that allows a user to search for any item in the library collection. Most of the database model was already developed and other modules of the system were already implemented. Hence, before coding, the team had to spend some time understanding existing source code and the database model before proceeding with the implementation of the new module.

- **Software Classification**: Information system

Sociological Factors - Project 3	
Team Size	$7-1$ (left the team) + 1 Customer
Team Education Level	Undergrad students: 7
Experience Level of Team	<5 years: 6 5-10 years: 1
Domain Expertise	Low
Language Expertise	Moderate

Project-specific Factors - Project 3	
Delivered User Stories	12
Domain	Web, Stand-alone Application, Library System
Person Months	4.2
Elapsed Months	4
New & Changed KLOEC	8.067
System KLOEC	31.252

Outcome Measures - Project 3	
Productivity	
KLOEC/PM	1.921
User stories/PM	2.857
Putnam Productivity Parameter	0.427
Morale	73 %

Table 5. XP-cf and Available XP-om - Project 3

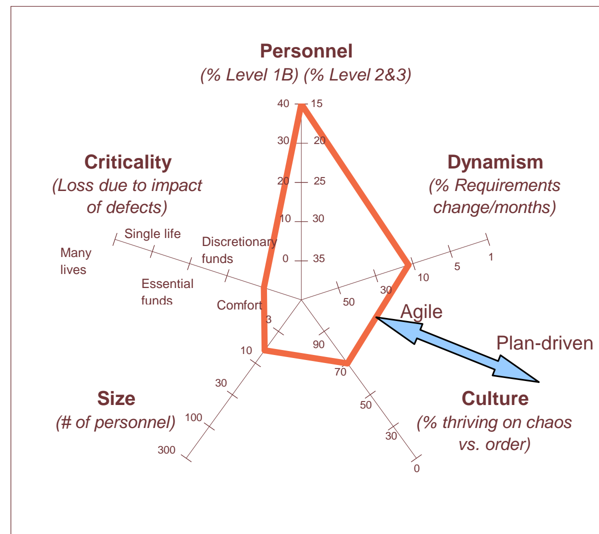

Figure 3. Developmental Factors (XP-cf) - Project 3

3.4.4. Project 4 - "GinLab":

- **Project Name**: GinLab - *Ginástica Laboral* (Laboral Gymnastics)

- **Description**: A stand-alone application to assist in the recovery and prevention of Repetitive Strain Injury (RSI). The program frequently alerts the user to take breaks and perform some pre-configured routines of exercises. Here, we analyze the first 3 iterations of the project, implemented during the first semester of 2006.

- **Software Classification**: End-user software

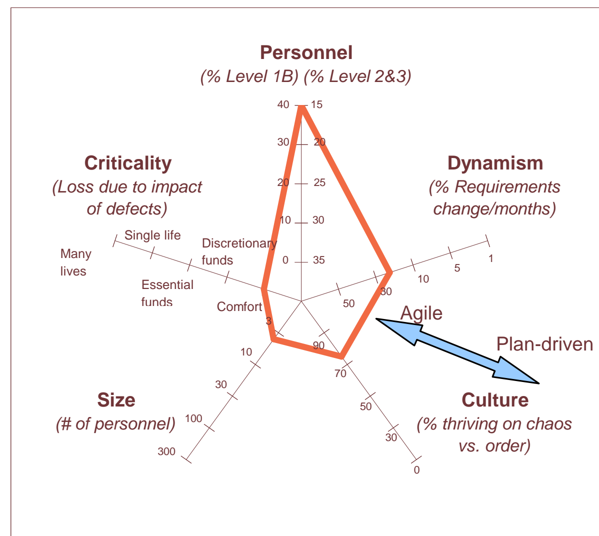

Figure 4. Developmental Factors (XP-cf) - Project 4

3.4.5. Project 5 - "Borboleta":

- **Project Name**: Borboleta (*Butterfly*)

Sociological Factors - Project 4	
Team Size	4 + 1 Customer
Team Education Level	Undergrad students: 4
Experience Level of Team	< 5 years: 4
Domain Expertise	Low
Language Expertise	Moderate
Project-specific Factors - Project 4	
Delivered User Stories	18
Domain	Stand-alone Application, Web
Person Months	2.6
Elapsed Months	4
New & Changed KLOEC	4.721
System KLOEC	4.721
Outcome Measures - Project 4	
Productivity	
KLOEC/PM	1.816
User stories/PM	6.923
Putnam Productivity Parameter	0.294
Morale	72 %

Table 6. XP-cf and Available XP-om - Project 4

Sociological Factors - Project 5	
Team Size	6 + 1 Customer
Team Education Level	Undergrad students: 1 Grad students: 5
Experience Level of Team	< 5 years: 2 5-10 years: 4
Domain Expertise	Low
Language Expertise	High
Project-specific Factors - Project 5	
Delivered User Stories	24
Domain	Mobile, Web, Medical system
Person Months	4.2
Elapsed Months	4
New & Changed KLOEC	7.753
System KLOEC	15.444
Outcome Measures - Project 5	
Productivity	
KLOEC/PM	1.846
User stories/PM	5.714
Putnam Productivity Parameter	0.411
Morale	80 %

Table 7. XP-cf and Available XP-om - Project 5

- **Description**: A telehealth software for PDAs and smart phones to assist doctors and nurses in medical appointments provided at the patients' home as part of programs under the Brazilian Public Health System (SUS). The system is composed of two parts: a Java J2ME software running on mobile devices where doctors and nurses take notes about the medical visits, and a desktop application that synchronizes all the mobile information and consolidates it in a database of the public health facility. The project started in 2005 with three undergraduate students and during the first semester of 2006 new features were implemented in the XP lab class. Here, we analyze the 3 iterations of the project, implemented during the second phase of development in the XP class.

- **Software Classification**: Information system

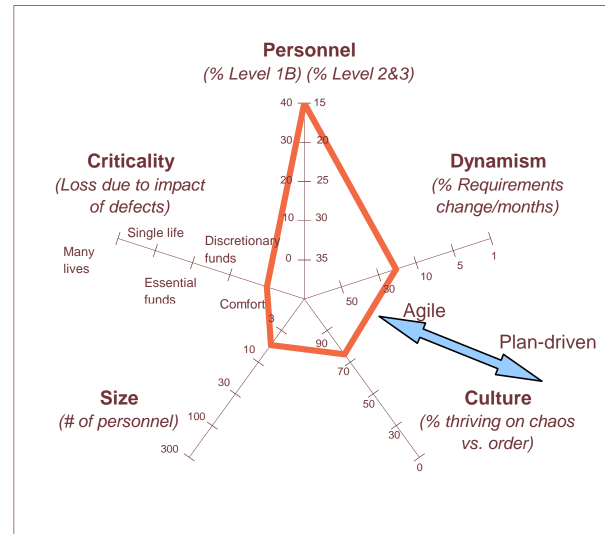

Figure 5. Developmental Factors (XP-cf) - Project 5

3.5. Governmental Projects

3.5.1. Project 6 - "Chinchilla": This project started with the initial support of our team. Training sessions were provided before the project kick-off and the coach and tracker roles were assigned initially to members of our team. After some iterations, our team gradually started to leave the project, handing over these roles to full-time employees. During the last iterations, we only had some interns supporting the development team.

- **Project Name**: Chinchilla

- **Description**: A complete human resources system to manage information of all employees in a governmental agency. It is being developed at the São Paulo State Legislative Body (ALESP) by a group of state employees guided by our team. As discussed in Section 2.3, we spent some time training the team on the technologies and on the agile method that we adopted for the project (XP). After some iterations, we decreased our level of participation in the project, allowing the team of employees to move on by themselves. Here, we analyze 8 iterations of the first release of the system.

- **Software Classification**: Information system

Sociological Factors - Project 6	
Team Size	9 ± 1 + 2 Customers
Team Education Level	Bachelors: 8 Undergrad intern: 2
Experience Level of Team	< 5 years: 2 5-10 years: 8
Domain Expertise	High
Language Expertise	Low
Project-specific Factors - Project 6	
Delivered User Stories	106
Domain	Web, Government, Human Resources system
Person Months	58.5
Elapsed Months	11
New & Changed KLOEC	48.517
System KLOEC	48.517
Outcome Measures - Project 6	
Productivity	
KLOEC/PM	0.829
User stories/PM	1.812
Putnam Productivity Parameter	0.367
Morale	67 %

Table 8. XP-cf and Available Xp-om - Project 6

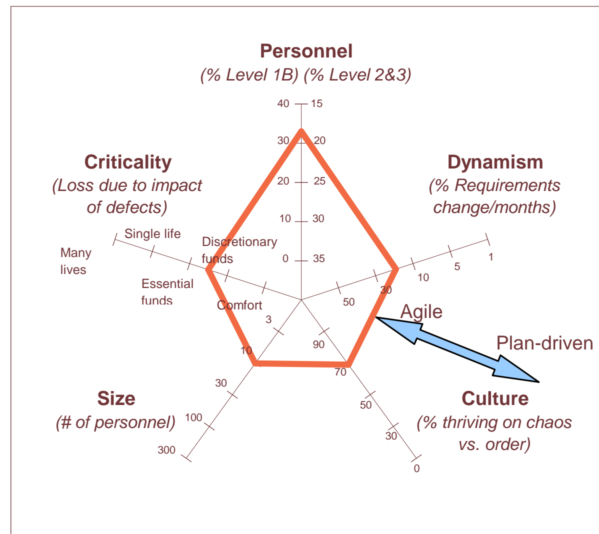

Figure 6. Developmental Factors (XP-cf) - Project 6

3.5.2. Project 7 - "SPL" - Legislative Process System: The initial development of this system was outsourced to a private contractor. After 2 years of development, the system was deployed and the team of employees at ALESP was trained and took over the maintenance of the system. Due to the lack of experience on the technologies used to build the system and to the large number of defects that arose as the system started to be used in production, the team had a tough time supporting the end-user needs, fixing defects, and implementing new functionalities. When we were called to assist them, we decided to introduce some agile practices to help them deal with the system, such as Continuous Integration [13], Testing (Automated Unit and Acceptance Tests), Refactoring [14], and Informative Workspace [4].

Also, as described in Section 2.3, we spent a couple of months training them on the technologies and topics of interest that would help them on their day-to-day job, such as object-oriented programming, Java collections, Struts, unit testing, acceptance testing, refactoring, and source control with CVS.

- **Project Name**: SPL - "Sistema do Processo Legislativo" (Legislative Process System)

- **Description**: A workflow system for the São Paulo legislators and their assistants to help them to manage the legislature documents (bills, acts, laws, amendments, etc.) through the legislative process. Here, we analyze the initial 3 months of the project after the introduction of the agile practices during the first semester of 2006.

- **Software Classification**: Information system

4. Metrics

In this section, we describe the metrics that were collected to analyze the projects described in Section 3. As proposed by Hartmann [19], there is a distinction between *Diagnostics* and *Organizational* metrics: diagnostics are supporting metrics that assist the team in understanding

Sociological Factors - Project 7	
Team Size	5 full-time employees + 3 part-time consultants + 5 Customers
Team Education Level	Undergrad students: 1 Bachelors: 7
Experience Level of Team	< 5 years: 1 5-10 years: 7
Domain Expertise	High
Language Expertise	Low

Project-specific Factors - Project 7	
Delivered User Stories	None
Domain	Web, Workflow
Person Months	15
Elapsed Months	4
New & Changed KLOEC	6.819
System KLOEC	177.016

Outcome Measures - Project 7	
Productivity	
KLOEC/PM	0.455
User stories/PM	N/A
Putnam Productivity Parameter	0.236
Morale	N/A

Table 9. XP-cf and Available XP-om - Project 7

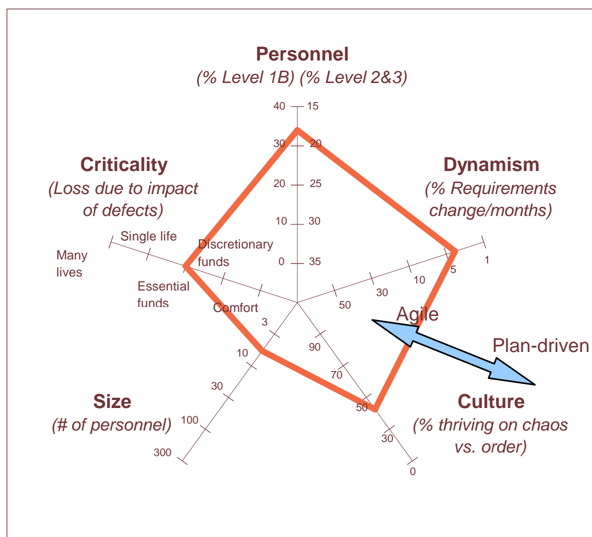

Figure 7. Developmental Factors (XP-cf) - Project 7

and improving the process that produces business value; organizational metrics are those that measure the amount of business value delivered to the customer. Goldratt warns us that teams will behave according to how they are being measured [17], so it is important that the team understand the purpose of the metrics.

In this article, we are interested in analyzing and proposing diagnostics metrics that can be used to support the *tracker* of an agile team. The purpose of such diagnostic metrics is to support the team's improvement. To avoid the problem stated by Goldratt, the team must understand that these metrics are volatile, therefore as soon as a metric is not serving its purpose, it should be discarded.

The next sections describe the quantitative and qualitative metrics gathered in this study.

4.1. Quantitative Metrics

Our quantitative metrics were directly gathered from the source code, from the source control repository and from XPlanner [30], a light-weight Web tool for planning and tracking XP projects. The metrics were collected at the end of the last iteration, but historical data from all iterations could be retrieved from the above-mentioned tools. The quantitative metrics are described below.

- **Total Lines of Code (TLOC)**: Counts the total number of non-blank, non-comment lines of production code in the system.
- **Total Lines of Test Code (TLOTC)**: Counts the total number of test points in the system, as defined by Dubinsky [12]. One test point is defined as one step in an automatic acceptance testing scenario or as one non-blank, non-comment line of unit test code.
- **McCabe's Cyclomatic Complexity** ($v(G)$): Measures the amount of decision logic in a single software module [29]. Control flow graphs describe the logic structure of software modules and consists of nodes and edges. The nodes represent computational statements or expressions, and the edges represent transfer of control between nodes. Cyclomatic Complexity is defined for each module (in our case study, a module is a method) to be $e - n + 2$, where e and n are the number of edges and nodes in the control flow graph, respectively.
- **Weighted Methods per Class (WMC)**: Measures the complexity of classes in an object-oriented system. It is defined as the weighted sum of all methods defined in a class [9]. In this study, we are using McCabe's Cyclomatic Complexity as the weighting factor for WMC, so WMC can be calculated as $\sum c_i$, where c_i is McCabe's Cyclomatic Complexity of the class' i^{th} method.
- **Class Size**: Counts the total number of non-blank, non-comment lines of a class in the system.

- **Number of Commits**: Counts the total number of individual commits to the source control repository.
- **Number of Lines Changed**: Counts the total number of lines (not only source code) added, removed, and updated in the source control repository.
- **Number of Delivered Stories**: Counts the total number of stories implemented in an iteration and approved by the customer.

4.2. Qualitative Metrics

The qualitative metrics were collected at the end of the last iteration of the first semester of 2006, so historical data for all iterations is not available. The qualitative metrics are described below.

- **Interviews**: We conducted semi-structured interviews, with a mixture of open-ended and specific questions [36] to give us a broader view and understanding of what was happening with each project. The open-ended questions in the interview derived from the answers to the following specific questions:
  - Which XP practice(s) you found to be more valuable to software development?
  - Which XP practice(s) you found to be more difficult to apply?
  - Were the *tracker* and the *coach* helpful to the team?
  - Would you apply XP in other projects?
- **Surveys**: We used a survey developed by William Krebs [25], extending it to include additional information about the team education and experience level, the extent to which each XP practice was being used (see Appendix A), and a general score (in a 0-10 scale) about *tracking* quality. For each question, each team member had to provide the "current" and "desired" level of their team in relation to that practice.
- **Team Morale**: Throughout the last iteration, we also collected information about the team morale by asking team members to update a *niko-niko calendar* [1] in a weekly basis. At the end of each work day, team members would paste a sticker in the calendar, with a color that would indicate his mood (pleasant, ordinary, or unpleasant). This was used to gather the XP-om for each project.

5. Analysis and Results

In this section, we analyze some relationships among the metrics described in Section 4 and discuss how they can assist in managing and tracking an agile project. Recalling one of our main goals in this article, this section will provide the "Diagnostics" metrics that we found suitable to support tracking an agile project and to support process and team improvement.

5.1. Analysis of Adherence Metrics Survey

By analyzing the survey results in Figure 14, we noticed that the average of the desired scores was higher than the actual scores for all practices in every project. That means that the teams wanted to improve on every XP practice. The survey was answered by 48 individuals and only one answer had a lower desired score than the actual score for one practice (Simple Design).

We also had a result similar to that reported by Krebs in his study that proposed the original survey [25]: the average desired score for each practice was higher than their desired overall score for XP. That, according to Krebs, indicates the team's enthusiasm for practical techniques and less understanding of the values and principles underlying the XP "label". Pair Programming on Project 7 was an exception to this behavior, indicating their resistance against this practice. Project 5 also had a lower desired score for some practices than their desired overall score for XP. In this case they have set a higher expectation on the process rather than the practices, having the highest desired overall score for XP among all projects (9.67). Project 5 is also the only project with an overall desired score for XP higher than the actual average of desired scores for all XP practices.

Finally, another information that can be extracted from the survey's result is the difference between the actual and desired scores for each practice. We can notice that the practices on Project 1 were going well, with a maximum difference of 1.37 on the Metaphor practice. Project 7, on the other hand, was starting to adopt some of the agile practices, showing a much larger gap on most of the practices. We can see differences of 8.00 and 7.50 for the Coding Standards and Tracking practices, respectively. These differences can inform the team about the most important points of improvement, showing which practices deserve more attention on the next iterations.

5.2. Retrospectives as a Tracking Tool

In our survey to collect the XP-am, we included a question about *tracking* (shown in Table 10) in addition to the already existing questions about lessons learned (Retrospectives) [25]. As shown in Table 10, the developer is expected to give a score between 0 and 10 to the current and the desired level of *tracking* using six sample sentences as a guideline of how well the practice is being implemented and followed by the team. The developer chooses the sentence that most resembles what the team does (and should do) and gives the score.

Tracking Current:_________ Desired:_________
There are big visible charts spread over the wall that helps us understand the project pace.
10 We have several charts that are updated daily and we remove the ones that are not being used anymore. The charts helps us understand and improve our process.
8 We have some interesting charts on the wall that are updated weekly.
6 The information in the wall is updated at the end of each release.
4 The charts are outdated and no one cares about them anymore. We have to work to finish on schedule.
2 I do not know why we have those charts on the wall. They do not seem to be related to my work. I think no one would notice if they were removed.
0 We do not have any charts on the wall. We think it is better to store important information on documents and files in our central repository.

Table 10. Question added to the survey adapted from [25]

By analyzing the answers to the question described in Table 10 (provided in Appendix A), we noticed that *tracking* was not working well for all of the teams: while projects 1, 4, and 6 had an average rate of 9.06, 8.50, and 7.56, projects 2, 3, 5, and 7 had an average rate of 6.00, 4.67, 5.33, and 1.75 respectively. That behavior was confirmed by the interviews with the team members of each project. In spite of that fact, it was an unanimous opinion that the Retrospective was a very valuable practice to support the team in understanding and improving their process. The results from the Retrospectives were posted in the workspace and worked as an important guideline to drive the team in the right direction. Even the teams that were not doing well on *tracking* were keen to follow up with the improvements proposed by the Retrospective posters. We conclude that the Retrospective is an additional practice to support *tracking*. It helps the teams to understand the project pace and improve the process and their performance on the next iterations.

5.3. Personnel Level and Agility

The developmental factors described in Section 3.1 and presented in Section 3 are proposed by Boehm and Turner as a risk-based method to classify projects between plan-driven and agile-inclined projects [6, 7]. When the polygon formed by the project data is distinctly toward the center of the graph, an agile method is suggested as the preferred approach for that project. Shapes distinctly toward the periphery suggest that a plan-driven method would be preferred.

One of the proposed risk factors is the personnel level of experience with agility and adaptability. This represents the percentage of the development team that falls at the various Cockburn levels described in Table 11. These levels consider the developer's experience in tailoring the method to fit new situations. By analyzing our project graphs (Figures 1, 2, 3, 4, 5, 6, and 7), we can observe that the experience on tailoring the process for all projects is similarly low. The top vertex of the polygon in all graphs are toward the periphery. Although Boehm and Turner suggest that you should consider the five risk factors when choosing between an agile and plan-driven approach, we noticed that the similarity between the projects' personnel level did not accurately represented the environment differences between the academic and governmental projects.

Level	Team member characteristics
3	Able to revise a method, breaking its rules to fit an unprecedented new situation.
2	Able to tailor a method to fit a precedented new situation.
1A	With training, able to perform discretionary method steps such as sizing stories to fit increments, composing patterns, compound refactoring, or complex COTS integration. With experience, can become Level 2.
1B	With training, able to perform procedural method steps such as coding a simple method, simple refactoring, following coding standards and configuration management procedures, or running tests. With experience, can master some Level 1A skills.
-1	May have technical skills but unable or unwilling to collaborate or follow shared methods.

Table 11. Personnel levels proposed by Cockburn [10]

Having people experienced in Agile Methods and in tailoring the process is an important risk factor, but they will not usually be available in a large number. As described in Section 2.3, our approach starts with training sessions led by an experienced coach. Most of our 5-year experience with XP involved inexperienced teams on Agile Methods. We did not find that a low percentage of Level 2 and 3 personnel affected the adoption of an Agile Method provided that there are a few people with a good knowledge of the method and a good coach.

From our experience, we also noticed that it is usually easier to teach agile practices to inexperienced programmers. More mature, experienced programmers sometimes tend to resist against agile practices such as Test-First Programming, Collective Code Ownership, and Pair Programming because they have to change dramatically their

work style.

Finally, we noticed that there is another personnel factor related to the successful adoption of an Agile Method: the coach's influence. After our team started to leave Project 6, the coach role was reassigned to a full-time employee. His knowledge of the practices and influence on the team was not the same as ours. Some of the practices started to be left aside, as we will further discuss in Section 5.5.

There are more personnel factors that can influence the decision between an agile and plan-driven approach. Further investigation should be conducted to understand how the cultural changes imposed by an agile approach can influence the personnel of a team, and how these changes can be evaluated when analyzing the risk factors that influence the adoption of an Agile Method.

5.4. Object-Oriented Design Metrics in Agile Projects

Chidamber and Kemerer proposed a suite of object-oriented design metrics, claiming that their measures can aid developers in understanding design complexity, in detecting design flaws, and in predicting certain quality outcomes such as software defects, testing, and maintenance effort [9]. Basili et al. were able to further validate these metrics, determining that the metrics were statistically independent and therefore did not provide redundant information [2]. It was also determined that the classes with a higher WMC were more prone to faults. An empirical study conducted by Subramanyam and Krishnan gathered data from industry projects developed in C++ and Java, and determined that larger class size and higher WMC are related to defects detected during acceptance testing and those reported by customers [38]. Li and Henry also analyzed the object-oriented metrics in two commercial systems, determining that five of six metrics (except Coupling Between Objects [9]) helped predict maintenance effort [26].

Figures 8 and 9 shows the overall average WMC and class size at the end of each iteration for all projects, respectively.

Basili et al. analyzed eight student projects using a development process derived from the Waterfall model [2], obtaining similar WMC results as our agile projects (mean: 13.4) but without controlling for the class size. Subramanyam and Krishnan analyzed two projects developed in different programming languages, obtaining similar WMC results as our agile projects in their Java project (mean: 12.2), but with significantly larger classes (mean: 136.0) [38]. Although they do not discuss the development process used for those projects, they do mention that they had access to design documents and UML diagrams, suggesting that an agile approach was not used. WMC and class size data from an open source software

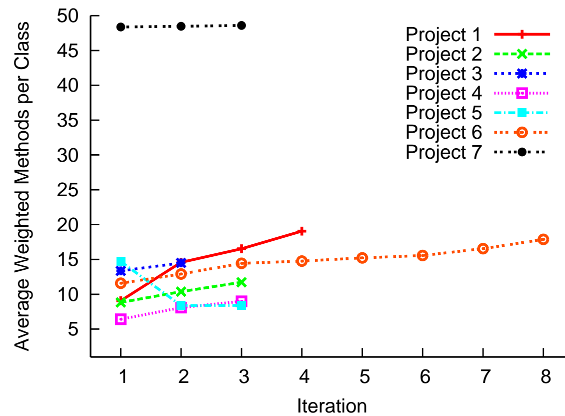

Figure 8. Average Weighted Methods per Class

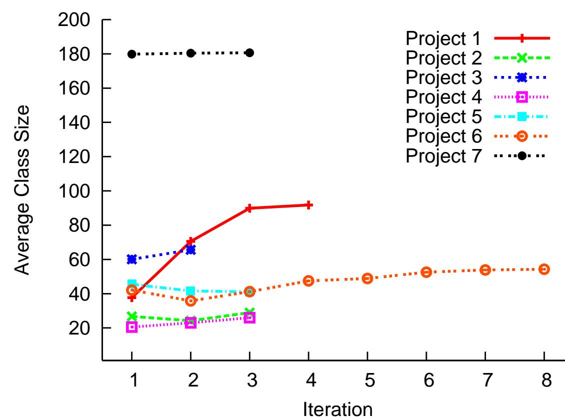

Figure 9. Average Class Size

were also gathered by Gyimóthy et al. showing the same behavior: similar WMC (mean: 17.4), but larger classes (mean: 183.3) [18].

By comparing these results with our agile projects (except Project 7), we can see that, although the average WMC metric is similar, our projects had an average class size significantly lower. Therefore the classes from our agile projects may have a similar design complexity (represented by WMC), but are much smaller, which makes it easier for developers to understand and maintain it. This might be possibly explained by the strong focus on source code quality and testing in most agile approaches. Techniques such as Test-First Programming, Continuous Integration, Refactoring, and Simple Design allow the system to evolve and accommodate change.

We can also see that Project 7 had a significantly higher average WMC and class size than the other projects. As discussed earlier, this suggests that Project 7 will be more prone to defects and will require more

testing and maintenance effort. As discussed in Section 3.5.2, this was the only project in which some agile practices were later introduced during the maintenance phase. Also, by comparing WMC and class size from Project 7 with the results from other studies, we can see that it has a similar class size average (mean $\approx$ 180), but a significantly higher design complexity (mean WMC $\approx$ 48). Many other factors prior to our intervention may have affected these metrics, but WMC and class size can be suitable metrics to be used when we later introduce more agile practices related to testing, Refactoring and Simple Design.

5.5. Measuring Continuous Integration

Continuous Integration is one of the most important XP practices to allow a team to deliver and deploy working software at the end of each release [13]. It is a technique that reduces the risk of large integrations at the end of a development cycle by providing an automated build of the entire system, frequent execution of the test suite, and the means for the entire team to understand what is happening with the system in a frequent basis.

By reflecting on data retrieved from the code repository, we propose a metric to analyze and diagnose how well Continuous Integration is going in a project. This metric can support the *tracker* to understand and improve the adoption of this practice in an agile team. We define the Integration Factor IF_i for iteration i as follows:

$$IF_i = \frac{LA_i + LR_i + LU_i}{TC_i}$$

where:

LA_i = total number of lines added in iteration i
LR_i = total number of lines removed in iteration i
LU_i = total number of lines updated in iteration i
TC_i = total number of commits in iteration i

If the team is properly doing Continuous Integration, the Integration Factor should be low, indicating that there are few line changes per commit. TC_i can be retrieved by analyzing the repository history log and LA_i, LR_i, and LU_i can be obtained from the file diffs retrieved from the repository. Figure 10 shows the value of the Integration Factor for each iteration of all projects:

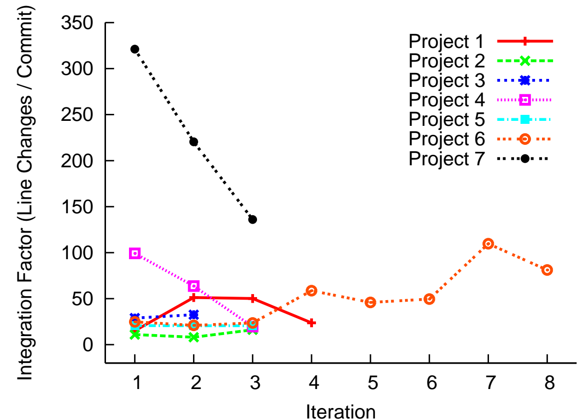

Figure 10. Integration Factor

Again, we can observe that Project 7 team members were used to wait longer before committing their changes to the repository. This behavior started to change after we introduced them to Continuous Integration, which can be seen in the graph as a rapidly decreasing line. We also observe a growing trend in the Integration Factor for Project 6. As our team gradually left the lead of Project 6 and let the state employees run the process, they kept some of the XP practices but were less rigorous with others, such as Continuous Integration. This shows that it is hard to sustain changes imposed by some of the agile practices. As opposed to what common sense would suggest, running and maintaining an agile method such as XP requires discipline.

To validate the suitability of this metric we calculated the Spearman's rank correlation between the Integration Factor and the team's average evaluation of Continuous Integration. A negative correlation of -0.57 was determined, but it was not statistically significant at a 95% confidence level (p-value $= 0.1$, $N = 7$). We believe that this is due to the small sample size, because the survey was only conducted at the end of the semester (last iteration).

5.6. Measuring the Team's Adoption of Testing Practices

Quality is one of the most important XP principles. Quality is not considered to be a control variable during XP planning [4]. Many XP practices provide the means to build and maintain the quality of a system, such as Test-First Programming, Continuous Integration, Refactoring, and Simple Design. In particular, XP suggests that both unit and acceptance tests should be automated to be frequently executed and to provide constant feedback about the system quality.

By analyzing data retrieved from the source code, we propose a metric to analyze and diagnose how well the Testing practices are going in a project. We define the Test Factor T_i for iteration i as the ratio between the number of lines of test code and the number of lines of production code:

$$T_i = \frac{TLOTC_i}{TLOC_i}$$

where:

$TLOTC_i$ = total lines of test code in iteration i
$TLOC_i$ = total lines of production code in iteration i

Figure 11 shows the value of the Test Factor for each iteration of all projects:

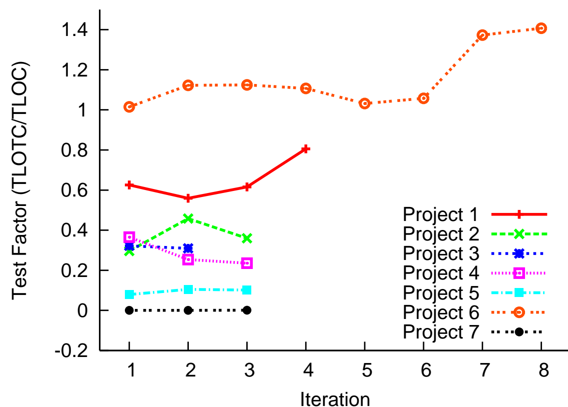

Figure 11. Test Factor

We observe that some projects may have more lines of test code than production code ($T_i > 1$). We can also observe iterations where testing was left aside, causing the Test Factor to drop and showing that more production code was developed without tests. We can also observe a low Test Factor for Projects 5 and 7 in which development started prior to the adoption of an agile method. The flat format of the graph for those projects reflects the difficulty of adopting testing practices in a legacy system where most of the production code was developed without and automated test suite.

Although the developers started to write automated tests for the legacy code, the systems had approximately $10,000$ and $170,000$ lines of code to be covered by automated tests. A great effort is still necessary to improve the test coverage for these projects. Figures 12 and 13 shows the evolution of the total lines of production and test code, iteration by iteration. An automated test suite is a highly effective means to build and maintain software with high quality, so it is important to develop a testing culture from the beginning of a project.

To validate the suitability of this metric we calculated the Spearman's rank correlation between the Test Factor and the team's average evaluation of the Testing practice at the end of the last iteration. A positive correlation of 0.72 was determined with statistical significance at a 95% confidence level (p-value $= 0.03382$, $N = 7$).

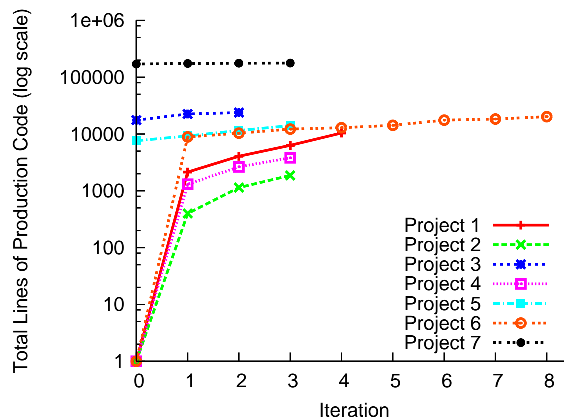

Figure 12. Total Lines of Production Code

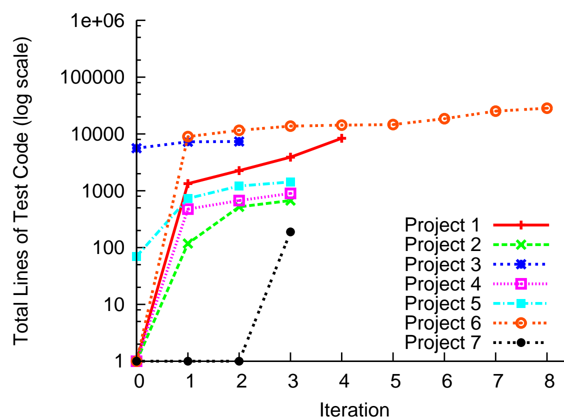

Figure 13. Total Lines of Test Code

6. Conclusions and Future Work

In our empirical case study, we analyzed seven projects from both academic and governmental environments from the point of view of tracking an agile process. The projects were conducted by experienced coaches and were mostly composed of team members without previous experience with Agile Methods. We classified the projects using the XP-EF framework, contributing to build up the weight of empirical evidence about XP and Agile Methods.

Also, we collected several metrics that were analyzed to understand their suitability to support diagnosing and tracking an agile project. Some of our findings in analyzing those metrics were:

- All projects had a higher desired score for every practice, showing their willingness to improve in all XP practices. The difference between actual and desired scores can point the team to the most important

points of improvement.

- Retrospectives and their posters are complementary tools to tracking, helping the team to understand the project pace. Even the teams that were not capable of tracking properly, could carry out good retrospectives and benefit from them.

- A low percentage of Level 2 and 3 personnel, capable of tailoring the method to fit new situations, did not affect the successful adoption of an agile approach in our projects. There are other personnel factors that can influence the adoption of an agile approach, such as the team experience and the coach's influence. Further investigation should be conducted to understand how the cultural changes imposed by an agile approach can influence the personnel of a team, and how these changes can be considered when classifying projects between an agile and plan-driven approach.

- Project 7 had a considerably larger average for WMC and class size, suggesting that it would be more prone do faults (defects) and would require more maintenance and testing effort.

- The adoption of an agile approach resulted in classes with an average WMC similar to the average WMC of classes from projects using different development processes. However, the average class size was significantly lower, which makes it less error prone and easier for developers to understand and maintain.

- We propose a new metric to diagnose how well testing is going in a project. The Spearman's rank correlation between our proposed Test Factor and the team's evaluation of Testing adoption was 0.72, which is statistically significant at a 95% confidence level (p-value $= 0.03382$, $N = 7$).

- We propose a new metric to diagnose how well Continuous Integration is going in a project. The Spearman's rank correlation between our proposed Integration Factor and the team's evaluation of Continuous Integration adoption was -0.57 but not statistically significant at a 95% confidence level (p-value $= 0.1$, $N = 7$) because of the small sample size.

In future work, we plan to continue gathering more data and more metrics to build a larger history for agile projects. We are particularly interested in measuring defects and bugs after the projects are deployed in production, trying to correlate the level of XP adherence to the quality of the system as perceived by the final users. It would also be interesting to gather more data from agile and non-agile projects to statistically confirm the suitability of the proposed metrics, and to propose new metrics to aid the *tracker* in diagnosing the adoption of different agile practices. Finally, it would be interesting to compare similar projects adopting agile and non-agile methods with respect to the speed and quality of the produced software.

References

[1] Sakata Akinori. Niko-niko calendar website. http://www.geocities.jp/nikonikocalendar/index_en.html, Jul. 2006.

[2] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1st edition, 1999.

[4] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, 2004.

[5] Kent Beck et al. Manifesto for agile software development. http://agilemanifesto.org, Jul. 2006.

[6] Barry Boehm and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley, 2003.

[7] Barry Boehm and Richard Turner. Using risk to balance agile and plan-driven methods. In *IEEE Computer*, volume 36, pages 57–66, 2003.

[8] Piergiuliano Bossi. Extreme Programming applied: a case in the private banking domain. In *Proceedings of OOP*, Munich, 2003.

[9] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

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

[11] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, 2004.

[12] Yael Dubinsky, David Talby, Orit Hazzan, and Arie Keren. Agile metrics at the israeli air force. In *Agile 2005 Conference*, pages 12–19, 2005.

[13] Martin Fowler. Continuous integration. `http://martinfowler.com/articles/continuousIntegration.html`, Jul. 2006.

[14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[15] Alexandre Freire da Silva, Fabio Kon, and Cicero Torteli. XP south of the equator: An experience implementing XP in Brazil. In *Proceedings of the 6th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP'2005)*, pages 10–18, 2005.

[16] Alfredo Goldman, Fabio Kon, Paulo J. S. Silva, and Joe Yoder. Being extreme in the classroom: Experiences teaching XP. *Journal of the Brazilian Computer Society*, 10(2):1–17, 2004.

[17] Eliyahu M. Goldratt. *The Haystack Syndrome: Sifting Information Out of the Data Ocean*. North River Press, 1991.

[18] Tibor Gyimóthy, Rudolf Ferenc, and István Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[19] Deborah Hartmann and Robin Dymond. Appropriate agile measurements: Using metrics and diagnostics to deliver business value. In *Agile 2006 Conference*, pages 126–131, 2006.

[20] Jim Highsmith. Messy, exciting, and anxiety-ridden: Adaptive software development. In *American Programmer*, volume 10, 1997.

[21] Ronald E. Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2000.

[22] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison Wesley, 2000.

[23] Mira Kajko-Mattsson, Ulf Westblom, Stefan Forssander, Gunnar Andersson, Mats Medin, Sari Ebarasi, Tord Fahlgren, Sven-Erik Johansson, Stefan Törnquist, and Margareta Holmgren. Taxonomy of problem management activities. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, pages 1–10, 2001.

[24] Norman L. Kerth. *Project Retrospectives: A Handbook for Team Reviews*. Dorset House Publishing Company, 2001.

[25] William Krebs. Turning the knobs: A coaching pattern for XP through agile metrics. *XP/Agile Universe 2002*, LNCS 2418:60–69, 2002.

[26] Wei Li and Sallie Henry. Object oriented metrics that predict maintainability. *J. Systems and Software*, 23:111–122, 1993.

[27] Kim Man Lui and Keith C.C. Chan. Test driven development and software process improvement in china. In *Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004)*, volume 3092 of *Lecture Notes on Computer Science*, pages 219–222, 2004.

[28] C. Mann and F. Maurer. A case study on the impact of scrum on overtime and customer satisfaction. In *Agile 2005 Conference*, pages 70–79, 2005.

[29] Thomas J. McCabe and Arthur H. Watson. Software complexity. *Crosstalk: Journal of Defense Software Engineering*, 7:5–9, 1994.

[30] Jacques Morel et al. Xplanner website. `http://www.xplanner.org`, Jul. 2006.

[31] Roger A. Müller. Extreme programming in a university project. In *Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004)*, volume 3092 of *Lecture Notes on Computer Science*, pages 312–315, 2004.

[32] Stephen R Palmer and John M. Felsing. *A Practical Guide to Feature Driven Development*. Prentice Hall, 2002.

[33] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, 2003.

[34] Lawrence H. Putnam and Ware Meyers. *Measures For Excellence: Reliable Software On Time, Within Budget*. Yourdon Press Computing Series, 1992.

[35] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Alan R. Apt, 2001.

[36] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25:557–572, 1999.

[37] Jennifer Stapleton. *DSDM: A framework for business centered development*. Addison-Wesley Professional, 1997.

[38] Ramanath Subramanyam and M.S. Krishnan. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003.

[39] Abbas Tashakkori and Charles Teddlie. *Mixed Methodology: Combining Qualitative and Quantitative Approaches*. Sage Publications, 1998.

[40] Laurie Williams. *Pair Programming Illuminated*. Addison-Wesley Professional, 2002.

[41] Laurie Williams, William Krebs, Lucas Layman, and Annie I. Antón. Toward a framework for evaluating Extreme Programming. In *8th International Conference on Empirical Assessment in Software Engineering (EASE '04)*, pages 11–20, 2004.

[42] Laurie Williams, Lucas Layman, and William Krebs. Extreme Programming evaluation framework for object-oriented languages – version 1.4. Technical report, North Carolina State University Department of Computer Science, 2004.

A. Adherence Metrics (XP-am) Survey

The survey that we used in our case study was based on William Krebs previous work on the subject [25], and is available online at `http://www.agilcoop.org.br/portal/Artigos/Survey.pdf`. Section 4.2 describes our adaptations to the original survey.

		Project 1		Project 2		Project 3		Project 4		Project 5		Project 6		Project 7	
		Avg	StdDev	Avg	StdDev	Avg	StdDev	Avg	StdDev	Avg	StdDev	Avg	StdDev	Avg	StdDev
Education Level		3.88	0.33	4.50	0.76	3.50	0.50	3.25	0.43	5.17	0.90	3.60	1.96	5.00	1.00
Experience Level		1.00	1.58	4.00	3.70	2.17	2.19	0.75	1.30	4.50	3.25	7.40	3.32	7.63	3.28
Pair Programming	Current	9.94	0.17	7.17	1.46	8.67	0.94	9.50	0.87	8.83	0.69	6.30	1.55	3.13	1.17
	Desired	10.00	0.00	9.33	0.75	10.00	0.00	10.00	0.00	9.83	0.37	8.60	0.92	7.63	0.70
Small Releases	Current	8.81	0.61	8.00	1.00	6.17	0.37	8.50	0.87	7.50	0.76	6.20	1.40	6.50	1.22
	Desired	9.25	0.66	9.67	0.75	10.00	0.00	9.25	0.83	9.00	1.00	8.50	0.81	8.25	1.20
Continuous Integration	Current	10.00	0.00	8.83	0.90	8.75	0.90	7.75	0.43	9.83	0.37	7.40	1.56	2.88	2.09
	Desired	10.00	0.00	10.00	0.00	9.00	1.00	9.75	0.43	10.00	0.00	9.00	1.34	7.75	2.86
Test Driven Development	Current	8.50	0.50	6.83	0.90	8.50	1.80	7.00	1.41	6.50	0.96	7.20	1.33	3.25	1.64
	Desired	9.31	0.66	9.33	0.75	9.00	1.53	9.33	0.94	9.33	0.75	8.90	1.30	8.63	1.11
Planning Game	Current	9.13	0.78	8.80	0.98	7.58	1.17	8.25	0.43	7.83	0.69	7.44	2.71	4.57	3.11
	Desired	10.00	0.00	9.50	0.76	10.00	0.00	9.00	1.00	9.50	0.76	9.33	0.94	9.14	0.99
On-site Customer	Current	8.00	0.50	8.25	1.15	8.33	0.75	6.50	0.87	5.50	0.76	6.40	1.74	5.63	2.34
	Desired	9.63	0.70	10.00	0.00	9.00	1.00	9.50	0.87	9.33	0.94	9.10	0.94	8.63	1.11
Refactoring	Current	9.75	0.43	8.75	1.82	5.67	0.94	5.75	1.30	8.33	1.70	5.60	1.50	3.25	3.38
	Desired	10.00	0.00	10.00	0.00	10.00	0.00	9.25	0.83	10.00	0.00	9.90	0.30	9.00	1.80
Simple Design	Current	9.25	0.83	8.42	1.02	7.33	0.94	7.00	1.22	7.83	1.21	5.70	1.85	2.63	1.32
	Desired	9.75	0.66	9.50	0.76	8.67	2.21	9.00	0.71	10.00	0.00	9.00	1.10	8.63	1.11
Coding Standards	Current	9.13	0.60	8.17	0.90	6.83	0.90	8.75	1.64	6.50	1.61	6.00	1.83	1.63	0.70
	Desired	9.88	0.33	10.00	0.00	10.00	0.00	9.75	0.43	10.00	0.00	9.56	0.83	9.63	0.99
Collective Code Ownership	Current	8.75	1.30	6.50	1.12	6.75	0.90	6.75	1.92	7.50	1.12	6.00	1.63	4.63	1.58
	Desired	10.00	0.00	9.33	0.94	9.67	0.75	9.00	1.00	9.50	0.76	9.11	0.99	9.13	1.36
Metaphor	Current	8.63	1.32	8.42	1.02	5.67	1.37	8.00	0.00	7.00	1.63	6.56	1.50	3.63	1.58
	Desired	10.00	0.00	10.00	0.00	9.33	0.94	8.75	0.83	10.00	0.00	9.89	0.31	9.38	0.86
Sustainable Pace	Current	9.63	0.70	6.17	2.03	7.42	1.43	8.50	1.66	9.00	1.00	8.67	1.89	6.25	2.33
	Desired	10.00	0.00	10.00	0.00	10.00	0.00	10.00	0.00	9.67	0.75	10.00	0.00	9.88	0.33
Lessons Learned	Current	8.63	0.86	8.58	0.84	8.08	0.19	7.50	0.87	7.17	0.90	7.22	1.23	5.50	2.00
	Desired	9.75	0.66	10.00	0.00	10.00	0.00	9.50	0.87	9.67	0.75	9.00	0.94	9.38	0.70
Tracking	Current	9.06	0.73	6.00	1.29	4.67	1.37	8.50	0.87	5.33	0.94	7.56	0.96	1.75	2.49
	Desired	10.00	0.00	10.00	0.00	9.67	0.75	9.75	0.43	9.33	1.49	9.89	0.31	9.25	0.83
Overall XP Score	Current	8.13	0.33	6.67	0.75	7.00	0.89	7.50	0.50	7.50	0.96	6.90	1.30	4.25	1.79
	Desired	9.00	0.87	9.00	1.00	8.67	0.94	8.75	0.43	9.67	0.75	8.40	0.80	8.13	1.76
Average (all except overall)	Current	9.08		7.78		7.17		7.73		7.48		6.73		3.94	
	Desired	9.83		9.76		9.60		9.42		9.65		9.27		8.88	

Figure 14. Results from the survey adapted from Krebs [25]