An Empirical Analysis of eXtreme Programming Practices and its Impact on Software Quality Metrics

Roberto Pepato Mellado¹, Gabriel de Souza Pereira Moreira¹, Robson Luiz Monteiro Júnior¹, Adilson Marques da Cunha¹, Luiz Alberto Vieira Dias¹

¹Brazilian Aeronautics Institute of Technology (ITA) – São José dos Campos – SP - Brazil

{rpepato, gspmoreira}@gmail.com, robson@kimmay.com.br, {cunha, vdis}@ita.br

Abstract: This work presents an investigation of three different industrial projects of software development by a Brazilian enterprise. During projects’ execution, the company has changed its approach on software processes from RUP based process to agile like processes. To assess software product quality metrics evolution, an investigation of product metrics history was conducted in those three projects. This paper characterizes the use of eXtreme Programming practices within the analyzed projects and the observed measures of quality metrics in the developed software products.

Keywords: Agile software development, extreme programming, object-oriented software, software quality metrics

1. Introduction

On the last decade, a growing interest on agile methods for software construction from both industry and academy [Dyba et al. 2008] [Hanssen et al. 2010] has arisen. Agile methodologies rely on a set of best practices that are considered to increase quality assurance and control. These procedures are integrated across the entire life-cycle development. Agile methods state that quality should be built into the product through a combination of best practices. Studies support advantages of agile practices towards product quality improvement [Sfetsos 2010].

Extreme Programming (XP) [Beck 1999], one of established agile methods, has defined sets of values, principles, and practices. Quality is one of the principles that must be followed and tracked when applying XP. Beck [Beck 1999] states that pushing quality higher often results in faster delivery, while lowering quality standards results in later and less predictable delivery.

One approach to monitor software quality over time is the usage of metrics. Some studies have proposed sets of metrics to measure systems designed and constructed by Object Oriented (OO) principles [Chidamber et al. 1994] [Martin 1994].

Lehman [Lehman et al. 1985] defines the law of increasing entropy. He explains that, by a number of factors, a programming system undergoes never-ending maintenance. As a result of these changes, the system becomes more complex and unmanageable, requiring specific work to maintain or reduce the increase of its entropy.
Recent works have conducted studies of quality evolution metrics in agile developed systems. Capiluppi [Capiluppi 2007] was probably the first study combining quantitative and qualitative evidences, which depict that projects using XP current lower complexity.

In this investigation, an assessment of software product quality metrics was performed in three different industry projects. This assessment was performed by using OO metrics. These metrics were used to analyze over time some trends of software quality and to verify whether they were somehow affected by the application of agile engineering practices. Subjective analyses were also applied to infer the impact of the XP practices usage over software product quality metrics.

Several charts with historical data are presented. Systems analyzed in this investigation were developed using different approaches. On the first system, no specific work had been performed for improving the source code quality. On the second and third systems, low to moderate efforts were performed in source code for quality improvement by usage of XP practices. As the number of agile engineering applied practices has grown among projects, authors have expected to see some improvement on selected metrics. This investigation presents the evolution of the system’s quality metrics.

2. eXtreme Programming (XP)

The XP is a style of software development focusing on applications of programming techniques, clear communication, and teamwork [Beck 1999]. It is not a predictive process where each activity needed in software development is determined at the first stage of project planning. Otherwise, it defines some sets of values, principles, and practices that one can use during the project development. They could offer to a development team some mechanisms to achieve the agility necessary to tackle today’s dynamic business changes and deliver the most valuable outcome to customers.

Refactoring [Fowler et al. 1999], one of the practices applied in agile methods, is a disciplined way to make small changes to source code improving its design without changing its external behavior. It aims to solve the problem of source code complexity and unmanageability, by a continuous process of source code inspection, adaption, and improvement.

Table 1 presents some XP practices used for the development of each analyzed system from the subsequent investigation on section four.

Wake's Radar Chart [Wake 2000] is a visual indicator of XP practices adoption. It is composed of five axes, representing the dimensions of an XP implementation: programming, planning, customer, pairing, and team. Values on each axis represent the average of correspondent practices, obtained from a questionnaire. Results for each evaluated project are presented on Figure 1.
Table 1: XP Practices used in each project

<table>
<thead>
<tr>
<th>XP Practices</th>
<th>Project A</th>
<th>Project B</th>
<th>Project C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sit Together</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Informative Workspace</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Energized Work</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Pair Programming</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Stories</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Weekly Cycle</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Quarterly Cycle</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Slack</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Ten-Minute Build</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Continuous Integration</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Test-First Programming</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Incremental Design</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Refactoring</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Notice from the weekly cycle row the absence of data showing that this practice was not used in any analyzed projects.

Figure 1. XP Radar Chart Applied for each Evaluated Project

3. Object Oriented Software Metrics

Metrics help software engineers to gain perception in the construction of products that they develop. Even though software’s technical metrics are not absolute, they provide a systematic approach to assess quality, from a well-established set of rules. This allows the identification and correction of defects, before they became failures [Pressman 2009].

Previous works [Chidamber 1994] and [Martin 1994] had proposed metrics for tracking aspects like size, complexity, coupling, and cohesion within object oriented (OO) systems.

3.1 Size

Lines of Code (LoC) is a software metric used to measure the size of a program by counting its source code lines, sometimes excluding comments and blank lines. Methods per Class (MPC) is a software metric that indicates the number of methods of a class.
3.2 Complexity
The structural complexity evaluates the internal construction of software products, as for example, components’ numbers and relationships among them. Software with more complex structure presents analysis and development difficulties for developers. Researches have shown that the number of defects found in programs have a strong correlation with its size and internal complexity [Koscianski 2007].

According to [McCabe 1994], complexity metrics can present critical information and valuable feedback about software systems’ reliability and maintainability. Within tests and maintenance, they provide information about software modules and help to detect potentially instable areas.

A well-known complexity metric is the Cyclomatic Complexity Number (CCN) proposed by [McCabe 1976]. It provides the number of distinct execution paths of a source code block. This metric assumes a relationship with the necessary effort to understand and test the code.

3.2 Coupling and Cohesion
Coupling and cohesion influence the ability to modify the source-code of a program. As a general rule, high cohesion and low coupling among methods and components should be followed [Blunden 2003]. The coupling and cohesion metrics can be used to analyze the dependency level between program components.

Cohesion and coupling can favor or hamper programs’ source-code modifiabilty. Cohesion measures the association between two software components against the accomplishment of a given task. Coupling measures the association degree between two components. In general, a high level of cohesion and a low level of coupling between two modules are desired. As coupling decreases, the comprehension and maintainability of a system increases.

3.3 Design Metrics
Martin [Martin 1994] has proposed a set of rules, known as design quality metrics, that allows object oriented design quality measurement, in terms of dependencies between subsystems. Among these metrics are the Afferent Coupling (AC), the Efferent Coupling (EC), and the Association Between Classes (ABC). A design with highly dependent elements tends to be rigid and presents reduced reusability and maintainability.

Objects and components with many responsibilities pose a risk of modification needs. If its behavior changes, other systems objects may fail to operate properly. Objects with high dependency are fragile, in face of changes [Duvall 2007].

3.4 Metrics used on this Investigation
The following metrics were used within the analysis performed on the systems:
- Lines of Code (LoC) - Number of valid class lines of code (excluding comments and blank lines);
- Afferent Coupling (AC) - Number of classes directly dependent on the class under analysis;
• Efferent Coupling (EC) - Number of classes that the class under analysis directly depends on;
• Association Between Classes (ABC) - Number of members of others classes directly used by the class under analysis;
• Cyclomatic Complexity (CC) - Number of execution paths of a class; and
• Methods per Class (MPC) - Number of methods of a class under analysis.

4. Product Analysis
This section presents the activities performed in the investigation conducted in this analysis. It describes: (i) the context of software projects analyzed in this study; (ii) the method used for metrics extraction and consolidation into a database; and (iii) the comparison of selected metrics evolution among projects.

4.1 Context
This investigation has analyzed three software development projects from industry. These projects are identified in this investigation as Project A, Project B, and Project C. The three projects were developed by a Brazilian medium-sized enterprise specialized in Geographic Information Systems (GIS) development for its consumers, mainly in South America.

One of the major complaints of the software development team members (directors, managers, developers, testers) is the lack of quality built in its products. At the beginning of the software engineering group within the company, it was used an ad-hoc, non-formal process development for product building. Around 2006, the enterprise has changed its software development process toward a Rational Unified Process (RUP). Since 2009, the enterprise started to incorporate some agile principles and practices on technical levels, by influence of its software engineers.

4.2 The analyzed systems
The analyzed software products have used GIS technologies. They were developed by different programmers, except by three professionals, that have participated in those three projects. Table 2 presents information about the projects and the developed products.

Projects A and B were developed by a software factory within the enterprise, by using a prescriptive process based upon RUP. Products were implemented for web platform, by using C# 2.0 language.

Within Project A, it was used just one XP practice, as shown in Table 1. The requirements analysis, documentation, and system design were conducted in the beginning of the project, and were not revised during the construction phase.

The team had about 10 developers directly engaged in the project. Because of the high turnover rate, 20 different professionals have participated in the system implementation. They worked distributed in two adjacent rooms, led by a technical leader which gave directions for the whole team. Developers were expected to implement code as modeled by designers in UML case tools. Usually, design models became out of date in relation to source code.
There were not any kind of automated tests, so all regression tests were manually made by a testing team. Refactoring was totally discouraged, because of the risk of new defects due to changes.

Overwork and turnover was constant on the project. Planned architecture, considered chaotic by developers, was followed during the whole project, which led to difficulties in maintenance and testing. Finally, performance and scalability was not accepted by the client, leading to a series of workarounds to be accepted in production.

Within project B, the requirements analysis and documentation were performed at the elaboration phase, but the system design wasn’t entirely ready at this phase. Architecture evolved during the project, and coding standards were followed. Interfaces, Model-View-Controller (MVC) architectural style and design patterns were applied to keep code decoupled and reduce complexity.

Therefore, few units and integration tests were automated and, thus, refactoring practices were not very much used. The team was placed in the same room and had fewer developers than Project A, around 4 developers on average. The software product of Project B was 34% shorter than Project A, in terms of C# Lines of Code (LoC), without comments and blank lines. Project B had 78% less defects detected by the test team and by end users than Project A, as can be seen in Table 2. In this project, a framework was produced and successfully reused also in other four projects.

Within Project C, a higher number of agile engineering practices were applied in its development. The team was comprised of four developers on average. The software product consisted in a tool for spatial data extraction, transformation, and load (ETL). In this project, some concepts of Scrum [Schwaber 2004] and XP frameworks were applied. There was no Project Manager working with the team. As it was a product development, there was a Product Owner (a role of Scrum Framework that is responsible to guide the project to deliver the most valuable outcome possible [Schwaber 2004]) to establish vision and prioritize stories to be implemented.

Also in Project C, there was no single Scrum Master (a role of Scrum Framework that is responsible to ensure the process execution and to remove impediments from the team [Schwaber 2004]). The Scrum Master responsibilities were spread on the team. The team felt that lack of a dedicated Scrum Master allowed too much external interferences on sprint objectives, impacting auto-organization, productivity, and motivation of the team.

All stories were organized in kanban, a mechanism for pull systems that provides updated process visibility [Poppendieck 2003]. Product quality was always the main focus of the team, and some effort was also dedicated to mitigate technical debts. In this context, technical debt is a metaphor to show the costs involved in bad design and implementation [Fowler 2004] [Diana 2010]. The empirical process was conducted during projects and, for each sprint, a retrospective ceremony was conducted for performance improvements. The team applied extensively agile practices like Test Driven Development (TDD), refactoring, incremental design, and continuous integration. The project C used the principle of test-first programming, which generated 315 unit tests and 87% of code coverage.
Table 2: Analyzed Projects Data

<table>
<thead>
<tr>
<th>Project</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Web</td>
<td>Web</td>
<td>Console</td>
</tr>
<tr>
<td>Programming Language</td>
<td>C# 2.0</td>
<td>C# 2.0</td>
<td>C# 3.5</td>
</tr>
<tr>
<td>Development Start</td>
<td>08/2008</td>
<td>08/2009</td>
<td>07/2010</td>
</tr>
<tr>
<td>Number of Use Cases</td>
<td>103</td>
<td>93</td>
<td>N/A</td>
</tr>
<tr>
<td>Revision Count a</td>
<td>2997</td>
<td>1358</td>
<td>453</td>
</tr>
<tr>
<td>Distinct Developers</td>
<td>20</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Defects, Errors and Failures detected by testers and end users</td>
<td>1387</td>
<td>313</td>
<td>N/A</td>
</tr>
<tr>
<td>Total LoC</td>
<td>97488</td>
<td>70735</td>
<td>36202</td>
</tr>
<tr>
<td>LoC without comments and blank lines</td>
<td>51017</td>
<td>33644</td>
<td>15523</td>
</tr>
<tr>
<td>Types (classes, interfaces, structs)</td>
<td>297</td>
<td>407</td>
<td>238</td>
</tr>
<tr>
<td>Assemblies b</td>
<td>23</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>

a. Revision – Identifier of a change set in the source code, sent from the developers local copy to a Version Control System repository, through a commit or check-in operation.
b. Assembly – Software binary component in .NET platform, generally an executable file or Dynamic Linked Library (DLL).

4.3 Software Product Metrics ETL

The metrics extraction process used in this work is a simplified version of METACOM, a method proposed on [Moreira et al. 2011], based on [Moreira et al. 2010] which consists in a set of automated processes to Extract, Transform, and Load (ETL) product metrics from software development projects. This process was implemented by the integration of some open-source and commercial tools, by the development of two new tools, and also of some automation scripts. Figure 2 presents the process implemented in this experiment, as well as the tools used for process automation.

![Diagram of the Implemented Method for Software Product Metrics ETL]

The main steps followed to obtain measures were:

- **Step 1 – Revisions Checkout** – It is performed a local copy of the source code of all revisions of the analyzed projects. To automate this step, it was developed the SVNExtractor tool that performs the checkout of all revisions of specific project from the Version Control System (VCS), in this case, Subversion (SVN);
- **Step 2 – Build** – In this step, it is performed the compilation and build of the software binaries, by NAnt and MSBuild scripts usage;
• **Step 3 – Static Analysis** – In this step, source code and generated binaries are analyzed throughout static code analysis tools in order to extract metrics. In this investigation, OO measures were extracted using NDepend [NDepend 2011] and SDMetrics [SDMetrics 2011] tools, mainly related to code and design aspects like size, coupling, cohesion, and complexity; and

• **Step 4 – Data Transformation and Load** – In this step, it is performed the selection, structuring, and normalization of the measurements and its load into a Data Warehouse, designed for historical series analysis. This step was performed by the development and usage of the DataExtractor Tool, which respects both the input XML and Data Warehouse data structures.

The application of the process described above required about six months of research and development. It was generated a database of roughly 500,000 records, containing measures of system classes during time, which was analyzed in this investigation.

### 4.4 Software Product Metrics Comparison

This section presents measures evolution during projects development lifetime. In the charts, individual metrics are analyzed for the three projects.

![Figure 3. The Evolution Charts of Mean Efferent Coupling per Class](image)

Data presented on Figure 3 charts show that EC grows up rapidly in the beginning of project A and that growth rate is lower in project B and even lower in project C. This could be caused by the absence of concerns with coupling metrics in project A. From the EC growth on projects B and C, it could be inferred that the initial concerns of coupling metrics were not respected during the project development lifecycle.

![Figure 4. The Evolution Charts of Mean Afferent Coupling per Class](image)
The analysis of Figure 4 charts has shown similar measures, with values around 4, for the three projects. The authors expected that a better design, promoted by XP practices like Test-First and Refactoring, would result in classes with less responsibility, thus, more distributed references by other classes. This would decrease the mean classes AC. Therefore, it was not observed representative differences among projects.

![Mean Association Between Classes](image)

**Figure 5. The Evolution Charts of Mean Association Between Classes**

The analysis of Figure 5 charts presents an observable difference among products of ABC measures evolution. It can be seen that projects that have applied more XP practices, present lower ABC average. Some studies have found that the lower the external coupling, the more maintainable is the source code [Moreira et al. 2011].

![Mean Cyclomatic Complexity](image)

**Figure 6. The Evolution Charts of Mean Cyclomatic Complexity per Methods**

Data presented in Figure 6 charts indicate that projects applying more XP practices have presented lower average cyclomatic complexity per class values. Projects B and C, which have applied more XP practices, have presented more distributed complexity among class methods.
Within object oriented software, classes with a high number of Lines of Code could indicate excessive responsibility [Martin 1994]. Classes with more than one responsibility have many reasons to change and for this they are more error prone [Moreira et al. 2011]. Data presented in Figure 7, within the charts, indicates that projects that have applied more XP practices have presented classes with lower average lines of code per class.

![Figure 7. The Evolution Charts of Mean Lines of Code per Class](image)

Alike number of lines of code, number of methods in a class could indicate that a class has excessive responsibility. Charts presented in Figure 8 indicate that projects which have applied more XP practices have presented classes with lower average number of methods per class. This could be interpreted as classes with more distributed responsibility, a property that could positively affect the maintainability of the product code.

Within Project C, it could be observed that the average MPC has decreased during time, while total LoC (see previous charts) and class numbers have increased. This could indicate the effects of refactoring work on the code base, because team strived to keep the classes as simple and short as possible.

**5. Threats to Validity**

The analysis conducted in this paper has used historical measures from Version Control System (VCS) revisions. In this context, revision means the identifier of a change set in the source code, sent from developers local copy to a VCS repository, through a commit or check-in operation. As it was not guaranteed that each day a new revision is committed, there weren’t measures for all days during project development. This could add bias on charts time series analysis.

The software product quality metrics analyzed in this work were observed in isolation. Other variables that could cause impact on projects like: schedule pressures, resources availability, team and management turnover, marketing moment, team engagement, among others, could not be controlled in this investigation of past projects.

**6. Conclusions**

In this investigation it was conducted an investigation of three different software development projects history. The enterprise has incorporated agile practices in its projects, in special the ones proposed by XP. A subjective analysis of XP practices
usage in projects was performed by the use of XP Radar chart assessment [Wake 2000]. Some results from this subjective analysis depicted enterprise’s progress in agile practices adoption.

In this paper, OO design and code metrics were selected, described, and analyzed over time. The implemented method for extracting metrics and consolidating them into a Data Warehouse were also presented.

The major findings of this investigation pointed out that class measures average for: size (LoC and MPC); complexity (CC); and coupling (ABC) were reduced in projects which have been applying more XP practices. Also, the coupling metrics of Afferent Coupling (AC) and Efferent Coupling (EC) were low affected by XP practices usage in this investigation. Based upon previous studies, experience, and consensus of the involved development teams, authors believe that XP practices lead to improved product quality in the analyzed projects.

7. References


