Studying the Evolution of Quality Metrics in an Agile/Distributed Project*

Walter Ambu[2], Giulio Concas[1], Michele Marchesi[1], and Sandro Pinna[1]

[1] Dipartimento di Ingegneria Elettrica ed Elettronica, Universitá di Cagliari,
Piazza d'Armi, 09123 Cagliari, Italy
{concas, michele, pinnasandro}@diee.unica.it
http://agile.diee.unica.it
[2] AgileTec, Via G. Murat, 26
09134 Cagliari, Italy
w.ambu@agiletec.it
http://www.agiletec.it

Abstract. This paper analyzes the development of a project initiated by a co-located agile team that subsequently evolved into a distributed context. The project, named JAPS (Java Agile Portal System)[1], has been monitored on a regular basis since it started in January 2005, collecting both process and product metrics. Product metrics have been calculated by checking out the source code history from the CVS repository. By analyzing the evolution of these metrics, it has been possible to evaluate how the distribution of the team has impacted the source code quality.

1 Introduction

In recent years many projects have been developed in a distributed context using agile practices [2][3][4][5]. Obviously opportunities for a co-located team differ from those for a dispersed team. Some XP/agile practices can be adopted at the same level in both contexts, while others cannot [6][5]. Several case studies have been published reporting experiences in applying agile practices in distributed projects, but as far as we are aware nothing has been published to date concerning the analysis of the evolution of source code quality metrics in this kind of project.

1.1 CK Metrics

The quality of a project is usually measured in terms of lack of defects or maintainability. It has been found that these quality attributes are correlated with specific metrics. For Object Oriented systems the Chidamber and Kemerer metrics suite [7] [8], usually known as the CK suite, is the most validated. The CK suite is composed of six metrics:

* This work was supported by MAPS (Agile Methodologies for Software Production) research project, contract/grant sponsor: FIRB research fund of MIUR, contract/grant number: RBNE01JRK8.

P. Abrahamsson, M. Marchesi, and G. Succi (Eds.): XP 2006, LNCS 4044, pp. 85–93, 2006.

- **Weighted Methods per Class (WMC):** a weighted sum of all the methods defined in a class. Chidamber and Kemerer suggest assigning weights to the methods based on the degree of difficulty involved in implementing them [7]. Since the choice of weighting factor can significantly influence the metric value, this is a matter of continuing debate among researchers. Some researchers resort to cyclomatic complexity of methods while others use a weighting factor of unity for validation of OO Metrics. In this paper we also use a weighting factor of unity, thus WMC is calculated as the total number of methods defined in a class.
- **Coupling Between Object Classes (CBO):** a count of the number of other classes with which a given class is coupled, hence it denotes the dependency of one class on other classes in the system. To be more precise, class A is coupled with class B when at least one method of A invokes a method of B or accesses a field (instance or class variable) of B.
- **Depth of Inheritance Tree (DIT):** the length of the longest path from a given class to the root class in the inheritance hierarchy.
- **Number of Children (NOC):** a count of the number of immediate child classes inherited by a given class.
- **Response for a Class (RFC):** a count of the methods that are potentially invoked in response to a message received by an object of a particular class. It is computed as the sum of the number of methods of a class and the number of external methods called by them.
- **Lack of Cohesion of Methods (LCOM):** a count of the number of method-pairs with zero similarity minus the count of method pairs with non-zero similarity. Two methods are similar if they use at least one shared field (for example they use the same instance variable).

1.2 Literature on CK Metrics

CK metrics have been widely validated in the literature. In a study of two commercial systems, Li and Henry [9] explored the link between CK metrics and the maintenance effort. Similarly, based on an investigation of several coupling measures (including CBO) and the NOC metric of the CK suite in two university software applications, Binkley and Schach [10] found that the coupling measure was associated with maintenance changes made in classes. Studying eight medium-sized systems Basili et al. [11] observed that several of the CK metrics were associated with class fault proneness. In a commercial setting, Chidamber et al. [12] noticed that higher values of the coupling and cohesion metrics in the CK suite were associated with reduced productivity and increased rework/design effort. Cartwright and Shepperd [13] studied a medium-sized telecommunications system and found that the inheritance measures of the CK suite (DIT, NOC) were associated with class defect density.

2 JAPS Process Evolution

JAPS is an open source j2EE solution for building web portals, integrating services and handling contents through a content management system (CMS). The

project was started in January 2005 by the agile team of AgileTec [14], an IT company based in Italy. JAPS was conceived as a result of some team members' experience in developing web portals and CMS with open source and legacy software. The goal was to create an adaptive, non predictive system that was simple, flexible and easily adaptable to customer needs.

The JAPS kernel was first built by a co-located team of two experienced software engineers applying agile practices. These practices include pair programming, testing, refactoring, planning game, short iterations [15][16]. After two months the team released a prototype of the system.

Subsequently, a partnership agreement was drawn up with an IT company and a commitment made to build two portals. As a result the number of team members was increased from two to seven. As the new members came from different IT companies, it was decided to adopt an open source-like development model. In particular the team applied dispersed agile development [4] where developers were physically alone most of the time and connected through communication channels. Thus, in this phase the team started working in a distributed context. In defining an agile methodology for this context and integrating agile practices with open source principles [17], they allowed for the fact that all team members lived in the same city. For instance, in order to share knowledge and experience, it was decided to meet once or twice a week. Being located in the same city also made it possible to schedule pair programming sessions as needed. The lack of face to face communication in the distribution, made it necessary to define effective communication strategies. Voip systems, e-mail and mobile phones allowed the team to communicate [18] effectively during development sessions even if this involved several iterations.

Frequent releases with working functionalities allowed continuous customer feedback. Requirements were gathered by using a prioritized backlog list shared among team members [19]. After a first tuning phase, requirement management using the backlog list became effective.

The other agile practices had to be adapted to the new distributed context. This required several iterations before the team developed maturity in adopting agile distributed practices.

The distributed phase initiated with an already defined test infrastructure. This included testing frameworks for web-applications, xml and mock objects. Several iterations were needed for the new team members to effectively implement the testing practices in a JAPS context. Once the team had become more comfortable with test harnesses, refactoring practices were applied more effectively.

The JAPS development process is thus characterized by two distinct phases. In the first phase, the team experimented and optimized some key agile practices in a distributed context. In the second phase, the team developed maturity in implementing these practices. The main phases of the evolution of the JAPS process are summed up below:

– phase 0 (January 2005-February 2005). The kernel was built by a co-located team of two experienced programmers using agile practices.

– phase 1 (March 2005-July 2005). The 7-strong team, (2 kernel developers + 5 new members), experimented key agile practices in a distributed context.
– phase 2 (August 2005- January 2006): the team developed maturity in the application of key practices.

In the next section, we will analyze how the source code quality metrics evolved during phases 1 and 2.

3 JAPS Metrics Evolution

In this section we analyze the evolution of source code metrics at regular two-week intervals. Each source code snapshot has been checked out from the CVS repository and analyzed by a parser that creates an xml file containing the information needed for calculating the metrics. This xml file is parsed by an analyzer that calculates all the metrics. Both the parser and the analyzer have been developed by our research group as a plug-in for the Eclipse IDE. The analyzed metrics are: Number of Classes, Class Size, Number of Test Cases, Number of Assertions, WMC, RFC, LCOM, CBO, DIT, NOC.

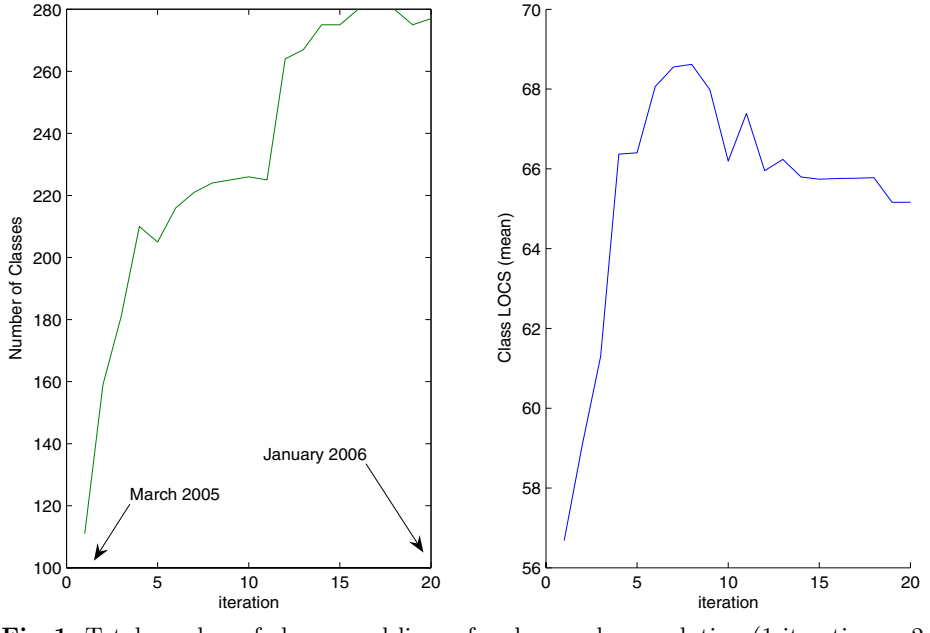

Fig. 1. Total number of classes and lines of code per class evolution (1 iteration = 2 weeks)

Number of Classes. This metric measures the total number of classes (abstract classes and interfaces are included) and is a good indicator of system size. When the distributed phase started, the system comprised 111 classes, then evolved

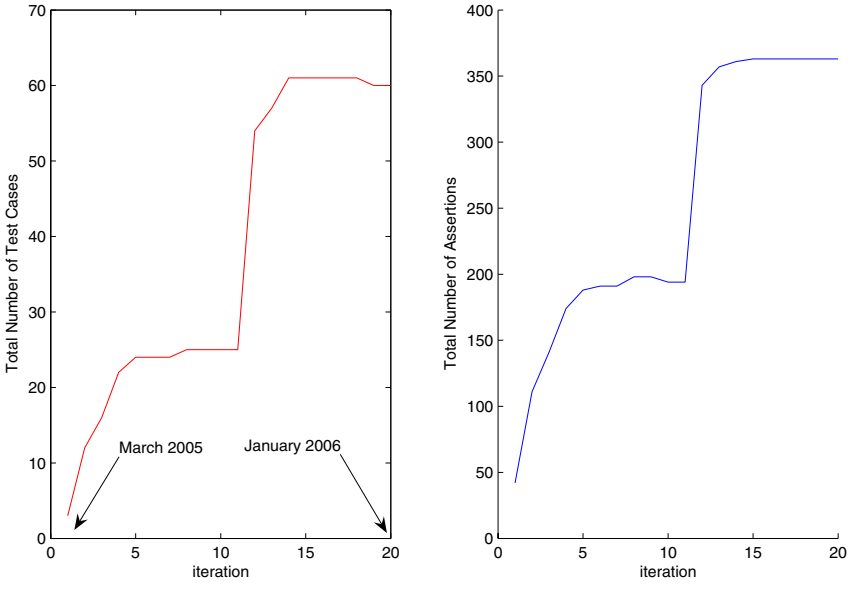

Fig. 2. Number of test cases and number of assertions for each iteration (1 iteration = 2 weeks)

rapidly as shown in fig. 1. The last CVS snapshot consists of 277 classes, indicating that the system doubled in size during the distributed phases (phases 1 and 2).

Class size. The size of a class has been measured by counting the lines of code (LOC), excluding blanks and comment lines. The mean value of class LOC has been plotted in Fig 1 for each iteration. It is known that a "fat" class is more difficult to read than an agile one. High values of this metric indicate a bad code smell that should be corrected using refactoring technics. Fig 1 shows a first phase in which the metric grows rapidly followed by a second phase in which it decreases.

Number of test cases. The number of test cases may be considered as an indicator of testing activity. As shown in fig. 2, the metric increases more rapidly in the second phase than in the first one. This might be explained by the faster growth of the total number of classes in the second phase but examination of the plot in fig 1 shows that this hypothesis can be reasonably ruled out. The main reason is certainly the maturity developed by the team in the second phase, that enabled them to write more tests during development.

Number of Assertions. Simply using the number of test cases, however, could be considered a poor indicator of testing activity. In fact, new test methods could be added to existing test cases without increasing their total number. The number of test methods might be a better indicator of testing activity than the simple test case count. On the other hand, a test method may have

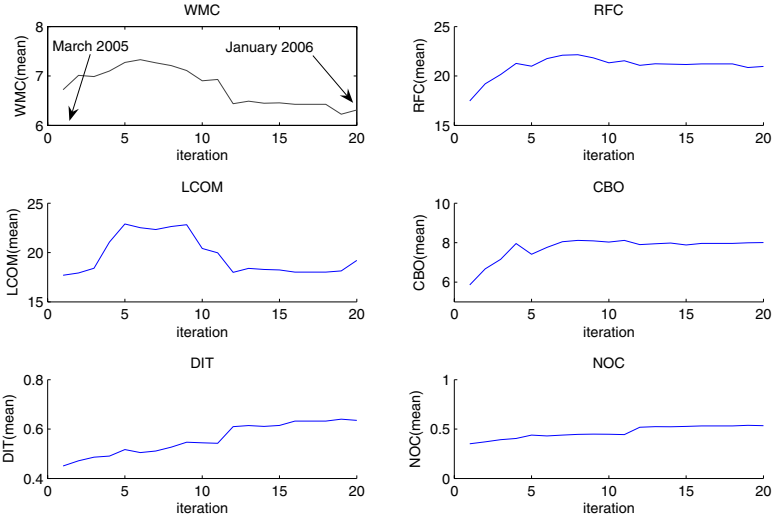

Fig. 3. CK Metrics Evolution (1 iteration = 2 weeks)

one or more assertions that compare expected and actual values. An assertion is a call to those methods of TestCase that have a name beginning with the string "assert"(assertEquals, assertSame, assertNotNull.....). The total number of assertions may be regarded as a more comprehensive indicator of testing activity. This metric, reported in fig. 2 shows the same trend observed for the number of test cases.

LCOM and WMC. The evolution of LCOM reported in fig. 3 shows a first phase where classes are characterized by low cohesion and a second phase where this metric has been progressively improved through refactoring. The same considerations discussed above also apply to WMC: a first phase characterized by a growing number of methods per class and a second phase where fat classes were split into cohesive classes with a small number of methods.

CBO. The evolution of this metric reported in fig. 3 shows a first phase where class complexity increases followed by a second phase where this metric remains approximately constant. The mean value increases from 6 to 8 during phase 1 and stabilizes at 8 during phase 2.

RFC. As previously mentioned, the response for a class is calculated by summing the number of methods and the number of calls to external methods. The RFC evolution (fig. 3) shows an initial increasing phase followed by a second phase in which the metric decreases slightly. This decrease could be explained by the strong reduction of WMC and an approximately constant trend of coupling between objects.

DIT and NOC. These metrics, that measure class inheritance characteristics, exhibit an increasing trend during the distributed phase.

4 Discussion

In this section we attempt to match the observed metrics evolution with the development process phases. To do this we can group metrics exhibiting similar behavior.

LCOM, WMC. The initial increasing phase can be explained by the lack of rigorous application of certain key practices like testing and refactoring. In the second phase, the team was able to reduce these metrics by applying simple refactoring practices. The bad smell was due essentially to the large number of methods and their low cohesion. These smells were eliminated by splitting the fat classes into classes with a small number of more cohesive methods, and by eliminating duplicated code. This also resulted in a reduction in the number of lines of code, as shown in fig. 1.

CBO, RFC. The interesting consideration that emerged from observation of these metrics lies in the second part of the plots. In fact, the effective adoption of key practices by the distributed team did not lead to the expected reduction in coupling and response for a class. This might be explained by the very nature of these metrics, that measure class interrelationship. To reduce this metric it is necessary to modify not only the single class but also the complex relationships with other system classes. Distribution of the team resulted in the programmer developing specialized knowledge on specific modules. Each time a programmer performed refactoring he did so on components of his competence. Programmers were apprehensive about changing something they knew little about. Their uneasiness grew as system complexity increased. It should also be noted that the kernel was built by two senior programmers and several meetings were planned at the beginning of the distributed phase to disseminate knowledge to new team members. Weekly meetings and a number of pair programming sessions did not enable effective knowledge sharing across team members in the distributed environment. This specialization resulted in the impossibility of reducing those metrics that depend on class interrelationships.

DIT and NOC. The same considerations made above hold here too. In fact, refactoring a class hierarchy requires a broad vision of the system and this is exactly what the distributed team did not have.

5 Conclusions

In this paper we have analyzed a project initiated by a co-located team and subsequently developed in a distributed manner. We have also presented the strategies employed by the team to effectively implement agile practices in the distributed context. The project has been divided into three main phases:

- phase 0: A co-located team developed the kernel.
- phase 1: The team experimented and optimized agile practices in a distributed environment.

– phase 2: The team applied agile practices effectively despite not being co-located.

The project was monitored by calculating product metrics during its development. These metrics include the CK suite of quality metrics. Analyzing the evolution of these metrics we found that in phase 1 the team increased system complexity. In phase 2 we observed that the effective implementation of agile practices resulted in system simplification. However, we also observed that the team was unable to improve all metrics to the same extent. In particular it proved impossible to reduce the value of those metrics that measure class interrelationships (CBO, DIT, NOC). This is likely due to the specialization of team members in specific components of the system. Therefore, in our experience, the adoption of agile practices in a distributed context may be effective only in reducing a subset of complexity metrics. Moreover, in the initial experimental phase of agile distributed practices system complexity was found to increase significantly. This study has given the team an opportunity to reflect on how to improve knowledge dissemination in a dispersed development environment. The JAPS project has now been released as open source [1] and we will continue monitoring both the process and metrics evolution in this new "phase 3".

References

1. JAPS: Java agile portal system. Url: http://www.japsportal.org (2005)
2. Poole, C.J.: Distributed product development using extreme programming. In Eckstein, J., Baumeister, H., eds.: Extreme Programming and Agile Processes in Software Engineering. (2004) 60–67
3. Fowler, M.: Using an agile software process with offshore development. http://www.martinfowler.com/articles/agileOffshore.html (2004)
4. Braithwaite, K., Joyce, T.: Xp expanded: Distributed extreme programming. In Baumeister, H., Marchesi, M., Holcombe, M., eds.: Extreme Programming and Agile Processes in Software Engineering. (2005) 180–188
5. Baheti P., Williams L., G.E., D., S.: Exploring pair programming in distributed object-oriented team projects. In: OOPSLA Educator's Symposium. (2002)
6. Maurer, F.: Supporting distributed extreme programming. In: Proceedings of the XP/Agile Universe 2002: Second XP Universe and First Agile Universe Conference. (2002)
7. Chidamber, S., Kemerer, C.: Towards a metrics suite for object oriented design. Proc. Conf. Object Oriented Programming Systems, Languages, and Applications (OOPSLA'91) **26**(11) (1991) 197–211
8. Chidamber, S., Kemerer, C.: A metrics suite for object-oriented design. IEEE Trans. Software Eng. **20** (1994) 476–493
9. Li, W., Henry, S.: Object oriented metrics that predict maintainability. J. Systems and Software **23** (1993) 111–122
10. Binkley, A., Schach, S.: Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. Proc. 20th Int'l Conf. Software Eng. (1998) 452–455
11. V. Basili, L.B., Melo, W.: A validation of object oriented design metrics as quality indicators. IEEE Trans. Software Eng. **22** (1996) 751–761

12. S.R. Chidamber, D.D., Kemerer, C.: Managerial use of metrics for object oriented software: An exploratory analysis. IEEE Trans. Software Eng. **24** (1998) 629–639
13. Cartwright, M., Shepperd, M.: An empirical investigation of an object-oriented software system. IEEE Trans. Software Eng. **26**(7) (2000) 786–796
14. AgileTec: Agiletec it company. Url: http://www.agiletec.it (2005)
15. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
16. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change- Second Edition. Addison-Wesley (2004)
17. Koch, S.: Agile principles and open source software development: A theoretical and empirical discussion. In Eckstein, J., Baumeister, H., eds.: Extreme Programming and Agile Processes in Software Engineering. (2004) 85–93
18. Steven Fraser, Angela Martin, M.A.C.C.D.H.M.P.M.S.: Off-shore agile software development. In H. Baumeister, M. Marchesi, M.H., ed.: Extreme Programming and Agile Processes in Software Engineering. (2005) 267–272
19. Bent Jensen, A.Z.: Cross continent development using scrum and xp. In Marchesi, M., Succi, G., eds.: Extreme Programming and Agile Processes in Software Engineering. (2003) 146–153