Defining a Distributed Agile Methodology for an Open Source Scenario

Manuela Angioni and Raffaella Sanna and Alessandro Soro
CRS4 - Center for Advanced Studies, Research and Development in Sardinia
Pula (Cagliari), Italy
{angioni, sanna, asoro}@crs4.it

Abstract – **In this paper we propose and describe an agile methodology for distributed development (MADD - Methodology for Agile Distributed Development). In particular, it's illustrated a set of best practices to apply in a distributed and agile context, chosen on the base of their impact software quality and team interoperation. Beyond the proposed methodology, we show the results of a survey that we submitted to various contributors of Open Source projects. The survey has been of support to the definition of the MADD, helping to more understand and estimate if, how and how much agile practices and values are already present in the OS world, that today represents one of the most emblematic examples of distributed development. The MADD methodology will be adopted on a software development project at the University of Cagliari (Italy), by a group of students that will work like an Open Source community.**

Keywords: distributed development, agile development, open source

I. INTRODUCTION

In this paper we describe an agile methodology for a distributed scenario that includes some practices and guidelines of general validity and that could be applied in different distributed contexts. In particular it is suited for a team which works like an open source community, and it will be applied in an experimental OS project, which involves about 30 students, that will provide a benchmark to test and refine it. In section §II we describe different distributed scenarios, like outsourcing, e-lancing and Open Source. Section §III contains a description of different approaches and proposals that, by the adoption of agile methodologies, try to improve the quality and the efficiency of a distributed development. In section §IV we compare agile methodologies and open source values, evidencing their points in common and their differences, while in section §V, we propose some results of a survey in which we asked team leaders and developers of OS projects to answer some questions about agile rules and principles they adopt in their projects. Finally, section §VI describes our proposal of a Methodology for Agile Distributed Development (MADD).

II. DISTRIBUTED SOFTWARE DEVELOPMENT

In the world of software development there are today different examples and scenarios of distributed development, that is a software development process in which the various actors of the process (teams of developers, managers, customers) are not co-located.

Indeed, in various scenarios the partners that cooperate at the software development project are forced or prefer (for practical or economical reasons) to work in a distributed (or "dispersed") context. Outsourcing is a first example of such context: a company delegates the development of a module or part of a software to a different company. In "offshore outsourcing" part of the software development is entrusted to a foreign company, with increasing difficulties of coordination and communication (different languages, time zone, standards). The reasons behind outsourcing and offshore outsourcing are basically the availability of highly qualified professionals at a low costs, with respect to western and central Europe and north America.

A different scenario comes from international partnerships in research and precompetitive development projects, in which dispersion is encouraged to enforce knowledge transfer and researchers mobility. Again, several teams based in universities or research centers all over the world need to cooperate, communicate and coordinate their work, regardless of geographical distance.

Also, e-lancing is a growing reality: in order to reduce projects costs, more and more sub-projects or activity lines are assigned to single free-lance developers, or groups of them, that provide with professional services, being not physically reachable by the customer and available only through the Internet.

Finally, let think to the rise of Open Source projects: the Open Source scenario is one of the most emblematic and widespread example of distributed development. In this case, typically, the programmers are constantly dislocated, they never met and communicate mostly with mailing list, leaving to a core team of developers the task of integrating the produced code.

III. DEFINING PROPOSALS OF METHODOLOGIES FOR DISTRIBUTED AGILE DEVELOPMENT

In the last years, researchers are trying to more and more design processes that improve the efficiency and quality of distributed development, adapting agile methodologies to this context. Surely, agile methodologies can supply many practices applicable to this scenario, but how many, which and how? The distribution of members that concur to the process of software development involves, in fact, a set of new problems and new requirements. Communication is one of the critical elements, even in the analysis and planning phases: it is necessary to understand what the customer wants, to have a common vision of the project and the requirements, it is in effect necessary to work like a team [10]. Moreover, in some scenarios, a problem is the continuous changing among team members, that constantly alternate on-site and off-site activity. This leads to a strong cultural shock, that, in turn, affects the ability of communication and cooperation: cultural and geographical distance tend to reduce and complicate communication and feedback, though sometimes partners are chosen among those culturally nearer. This is especially true in extreme dispersion contexts like offshore outsourcing. In sure, before attempting to extend agile practices to a distributed working environment, it is largely preferable to gain experience in these practices within a traditional non-dispersed context, and best to move the first steps with a dispersed team already experienced in agile methodologies. In any case, the idea is that every context of distributed development has such its own characteristics and peculiarities that requires an ad-hoc methodology: it's necessary to analyze the peculiarities of the process of

distributed development and then understand which practices of agile development, that have proved to be good for co-located teams, can be applied in a context of geographical distribution, and how to replace those invalidated or evidently inapplicable.

The MADD methodology, as we will see in §VI, although adopting some practices and guidelines that are of general validity and can be applied in different contexts of dispersion, is mainly suited for the organization of a team that cooperates as an open source community.

Different researchers have already studied the distributed agile development. In the case of the DXP (Distributed eXtreme Programming) [2, 3], as an example, following a practical approach, it's been proposed a set of practices that allow to adopt the XP in a distributed manner, inheriting its merits and adapting it to the case of development processes geographically distributed. Four XP practices, in fact, need the co-location of the team members: the Planning Game, the Pair Programming, the Continuous Integration and the Customer On-site. To bypass this necessary physical proximity, it has been proposed the adoption of many tools as Internet, videoconferences, screen and application sharing, or remote access to systems for continuous integration. Also instruments like mailing lists, daily or weekly reports with feedback from customer to developer, or Pair Programming adopted in the core teams, can increase the communication [3, 4], as well as the familiarity, that is the spirit of collaboration and trust between the members of the team. However, although the DXP can work keeping high the communication, the coordination and the availability of the team members, obviously it can't bring the same benefits and catch up the levels of communication reached with the physical proximity between members of the same team or team and customer. We recall, in fact, that the Pair Programming, for example, consists for the main part in a dialogue between developers that simultaneously try to plan, program, analyze, test and understand together how to better program. In the same way, according to the XP, it is necessary to have a representative of the customer that is all the time with the developers (on-site) and the daily stand-up meeting, typical of XP, are impracticable, at least daily, in distributed teams. The DPP (Distributed Pair Programming) [5, 6, 7] is another example of practical approach that studies how the Pair Programming can be adopted in a distributed context. It is about experiments led on groups of students geographically distributed. In this type of experiments the phase of definition and analysis of requirements, as the design one, has been carried on in a co-located situation. On the other hand, coding, testing and releases have been made in a distributed situation, trying to adapt the agile methodologies only in these parts of the development process. Also in this case, a set of instruments is indicated as necessary to promote communication and collaborative job between participants, like tools to share monitor and development environment, version management systems or Instant Messengers.

Starting from the need to define an agile methodology for the distributed development, many different tools to support developers, that concur also to apply the DPP, were proposed [8, 9].

IV. DEFINING AN AGILE METHODOLOGY FOR AN OPEN SOURCE PROJECT: A COMPARISON OF VALUES

Agile methodologies and the Open Source (OS) world have a lot in common. The same nature and the bazaar [11] organization of an OS team increase the value of the adaptation to changes, such as agile methodologies, preferring frequent releases and an immediate and continuous feedback from the contributors. Like agile methodologies, the OS emphasizes individuals with high skills and puts them in the center of a self-organized team of contributors. It's known [2, 3], moreover, that also in the OS projects there are some characteristics like the use of coding standards, the fast feedback (active and always updated mailing lists), the collective code ownership and the habit of embracing change. The Open Source world shares the value of communication, and in projects in which there are tens or hundreds of contributors, this value must be above all a requirement of the small group of developers around which works all the team. In the same way, the values of mutual trust and respect are basic conditions for a deep collaboration. Moreover, most of the OS teams, getting used to frequent releases, stimulate and encourage the developer. We recall that the agile values are fundamental: the agile practices cannot be applied, and are not agile, without the values that are their foundations. Finally, both worlds of agile methodologies and Open Source criticize the high costs of a debugging made too much late and promote a frequent debugging.

We observe that, however, some characteristics of OS projects differ from the agile world. Usually, in fact, a real customer does not exist, therefore falls the part of the development cycle that deals with the customer participation to the definition of specific functionalities: in place of it, an objective requirement is fixed, for example a tool or a specific library, and a call for participants starts. In OS projects, finally, there are many developers with many roles or roles not such defined.

So, there are agile principles in the OS development, but also essential characteristics, like the distribution of contributors, that are not agile principles. However the OS word is not mentioned as a term in contrast with the agile development, such as terms like the "cowboy coding" [12], for example, and on the other hand the code sharing, the cooperation between developers with a rigorous peer-review and a parallel debugging [11] are the main characteristics, certainly agile, of an OS project.

As we said before, because of peculiarity and variety of the distributed development scenarios, we think that the proposed methodology fits well with the organization of an OS community, although contains some practices and principles from which various contexts can benefit from. Moreover, we thought that the point of view of OS developers and their vision of agile methodologies is useful for better knowing OS world and to help us in defining the MADD. We have tried to have an idea of this submitting to various developers of OS projects a survey about the adoption of agile practices and methodologies in their projects.

V. AGILE PRACTICES AND OPEN SOURCE: A SURVEY

Although Open Source and Agile world share many principles and practices, as we have emphasized in §IV, do developers/users of the OS projects agree with this theory? Which practices or principles do they apply as indispensable? And, more in general, which rules and principles a distributed agile methodology should include, from which the Open Source development process could benefit from? In order to answer to all these questions, useful in the definition of our proposal, we have asked team leaders and developers of many OS projects to answer some questions. We have selected projects from repositories of OS projects as SourceForge [13] or incubators as Apache [14] or the more recent Codehaus [15]. The submitted questions and the results are available at http://www.crs4.it/nda/maps/index.html. Data carried out through the survey and collected in literature, have helped us in defining the guidelines of the MADD. The survey includes five main sessions about Communication,

Analysis and Planning, Coding, Refactoring and Testing. We sent an email to 75 projects and 27 of them answered (41 respondents in total). Table 1 shows that the analyzed OS projects involve mainly less than 20 committers (contributors that can commit code).

Projects' size	N. of answers
0 – 10 committers	17
10 – 20 committers	14
> 20 committers	10

Table 1. Projects' characteristics

Below there's a summary of results emerged from the survey.

To compensate the reduction of communication, developers mainly use mailing lists, private email, bug tracking systems, chat, and, when it is possible, face-to-face meetings and forums. In particular 50% of developers save the chat session, while only 30% of them use conventions for the subject and/or for the content of the email.

For the analysis and planning phase, 68% of respondents have answered that project requirements are analyzed, as shown in Table 2, but formally only in 36% of cases and frequently, at least once every two months, only for about 25% of them.

		N. of answers
Frequency	Frequently (< 2 months)Medium (2 – 6 months)Infrequently (> 6 months) Never	10 6 13 12
Method	Formally Informally No project requirements analysis	15 13 13

Table 2. How often and how project requirements are analyzed?

Concerning the coding phase, the survey evidences that:

- about 51% of developers release a new version at least once a month;
- about 85% of respondents adhere to coding standards, even if for about 29% of them standards are often enforced by the key developer who messages contributions in his own style;
- about 61% of them adopt the "collective code ownership" practice, allowing everyone to make and commit changes directly;
- 58,5% of interviewed practice continuous integration, using in particular tools like CVS, Eclipse, Gump or CruiseControl.

Moreover, most of the developers practice refactoring and testing, thinking that their systematic use improves code reliability and quality. In particular, 41,46% of them practice refactoring regularly but only when is being too difficult add new functionalities, while 36,58% during all the coding development phase.

Concerning testing, 55% of respondent answered that they must test the code they develop before submitting it, and in 65% of cases they do this since the project start. Only 30% of them write tests before the code itself and adopt the "Test Driven Development" (TDD) practice as XP, while another 30% of them write tests after the code itself and about the 14% only when is needed.

All these answers confirm us that OS developers really apply some practices or agree with some agile principles, and we've considered them in the MADD definition.

VI. MADD PROPOSAL

Our goal is to define and propose an agile methodology for distributed software development. This methodology will be applied in experimental projects that will provide a benchmark to test and refine it. At the moment, our experimental set consists of an open source project, bootstrapped by the University of Cagliari (Italy), for the development of a Web portal and a CMS, integrated with an e-learning platform and the University database, which will engage about 30 participants, mainly students, that will apply the proposed MADD. In particular they will work as an OS community: they are both developers and future users of the final product, useful for several needs like registration to exams, management of courses and management of teachers activities, and they will develop the different modules of the project, starting from a kernel developed by a core of about 10 committers. Student/developers will work in a distributed manner, sometimes at home, sometimes at the university, someone at our research center, partner in this project. They are now in a learning phase, being introduced to OS and agile worlds and technologies involved. After that, the development phase will start. We have divided the methodology into four different areas of scope, tracing from the OS survey (see §V): Communication, Planning and Design, Coding and Testing, Feedback. For each topic are indicated general guidelines and some specific aspects and details, with the related advantages.

Communication. Communication is one of the critical points in defining the MADD, being moreover one of the main principles of agile methodologies. In order to solve distance problems and to maximize the effectiveness of conversation, we think that it's necessary to:

- establish interpersonal relationships, like complicity and confidence, in order to gain trust and know the different work approaches of team members;
- allow a common vision of the project and the required functionalities.

The following tools and practices are useful to enable and improve distributed agile communication. In particular the MADD includes:

- the use of a dictionary, which defines userID and roles for the team members, by means of which it is possible to apply some rules to emails sent by the community, classifying them by the object;
- the use of a mailing list to send messages to all the team members, using the following rules:
    - the "Reply To" should be extended to all the mailing list, so that all the team could have a complete and homogenous knowledge of the project, its progress, its tasks and the assigned roles;
    - apply the dictionary rules to the mail subject, in order to immediately establish the topic and the level of interest;
- the use of an Instant Messaging tool, if there is not a policy against it, choosing the same userID used for the mail, in order to guarantee immediacy in the communication. Its systematic use helps people in establishing relationships and confidence, in particular for those who have never met themselves before, and decreases in part the idea of the distance;
    - save always the chat content, in order to share with all the team knowledge and decisions useful for the project;
- the use of a repository, to guarantee the sharing of the knowledge and a common view of the project. It could be:
    - CVS, or a similar version control system,

which provides visibility of activity and artifacts developed by the team. Moreover, using the commit message, and specifying in it userID and role, developers always know who added or created something;
- a Wiki, in order to share via web documents or information about the project, only by editing text and immediately viewed by everyone.

Planning and Design. Planning of activities is another important point in the distributed scenarios, where is often difficult to have face-to-face meetings. Required features should be described in a sufficiently detailed way, throughuser stories or use cases, in order to allow developers to easily translate them into simple programming tasks to be implemented. During the planning phase, the team should communicate mainly online, by mail or chat. Once features are divided into tasks, developers decide which ones they prefer to develop, choosing first the most important ones, according to a priority previously defined by the development needs. The following practices are only some general criteria of the planning phase, which we have identified as indispensable for the MADD:

- periodic releases, flexible in the contents but rigid in the date. Different versions of the software are released at fixed dates, about every 2 or 3 weeks, and include a set of implemented tasks. If, at the established date, the defined features will not be released, the team will try to understand the reason of the delay and try to better plan the following release. There will be different levels of detail for the planning:
    - quarterly (strategic planning): each month a work plan for the three following months is planned and reviewed, depending on the features developed by the team; if the project will need new features, they will be inserted in the plan. Developers discuss the plan online, according to the decided priority (1 or 2 releases each month);
    - monthly (operating planning): it's a more detailed plan than the quarterly; it describes in details each task and defines which developers implement them, according to their experience and their capabilities. The monthly plan happens online, through one of the tools described above. The progress of the plan and developers assigned to each task have to be known in every instant. A planning tool like XPSwiki [16] could be useful in this phase;
    - weekly (informal report): team members write, at least weekly, a short report about what they are doing, problems and progress state of their tasks. This is not a formal document, but only a report that allows the team to have a global vision of the project and stay in touch with all the developers.

The planning phase should not represent an excessive engagement for the team. Every document should be as essential as possible, simply a development track. The main point in this phase is that everyone should know what the other developers are doing in every moment, in which task they are involved, if they have problems or if there will be delay in releases, in order to avoid problems or correct them.

Coding and Testing. The coding phase is still more critic if developers work in a distributed team. Adhering to coding standards, defining standard rules like conventions on classes names, methods or variables and formatting rules, it's therefore very important, because it improves the legibility and, consequently, the maintainability of the code. But these rules should simply emerge from a common requirement to improve the productivity.

Even if a well written code should be self explanatory, code documentation is important too: every class or method should be commented in order to immediately understand its functionality. But if the code is not robust, all the rules described are not so useful: in fact, a such system could not be maintainable and it could be too expensive in terms of resources and time to be extended. To avoid the previous problem, it seems important to apply refactoring, a practice which can guarantee a code evolution with a behavior preservation, in order to improve the design of the code, making it more reusable and flexible to changes. But a tested and well written code is not enough to guarantee the correct behavior: the testing practice is useful to verify and to certify the correctness of a class. In fact, unit and functionality tests help in measuring the correctness and the robustness of the developed software. Functional testing (also known as black-box testing) is the process of verifying the behavior of a system, having no knowledge of the internal functionality/structure of the system. Unit tests, or white box test, allows instead to test every class or parts of the system, because they focuses specifically on using internal knowledge of the system.

Another practice inserted in the proposed MADD is the continuous integration, which allows the incremental development of the software and makes easier for developers to integrate changes of the project. Continuous integration should be performed automatically, to build the project continuously: to update sources, to compile them, to run tests. When the build has finished, we obtain a precise indication on the result: failed or performed correctly. The automated continuous integration provide an easy-to-use build system that increases productivity.

In order to automate such activity it is possible to use some scripts (for example ANT), that allow to specify a series of operations in a systematic way. After all, continuous integration founds itself on the following four principles:

- a Version Control System (CVS, for example);
- an automatic compilation process;
- the systematic run of tests;
- a scheduling process for tasks.

Feedback. A goal of agile methodologies is to regulate and reduce the cost of frequent changes in the software development process. So, not only frequent iterations but also regular feedback are necessary: a real time feedback, given by the development team, and a feedback given by the customer. Both of them should have a frequency and a code coverage that allow time for changes and avoid too high cost of changes. In fact, one of the risks is that, after the requirement analysis and the planning phases, customer and developers don't have a common vision of the project, which is the fundamental requirement in order to share the same goal during all the project. In a scenario like the OS one, the distribution of the team limits the communication and sometimes people know only a part of the module they are developing and it is not always simple to have a common vision of the whole project. In our context, the role of the customer is played by the developers themselves. But, even if they receive feedback, it is not as immediate as in a co-located team, then the following practices could help the dispersed team in receiving feedback:

- collaborative management of the code quality;
- developers could apply a sort of distributed pair programming, choosing another developer as a reviewer of their code;
- unit testing for all the code:
    - tests should cover all the code;
    - tests should be documented in order to emphasize their function and the parts of the code they cover;
    - unit tests should be written in order to emphasize bugs;
- use of automatic tools for useful activities, like metrics, test coverage and naming;
- feedback given by the core developers:
    - every time developers release a new version core developers (committers) should release a feedback document.

Research experiences in this field are generally limited to specific phases of the development process, and/or conducted on groups of students. As such, although surely a good start point, they cannot be easily generalized or extended to all the distributed scenarios.

VII. CONCLUSION AND FUTURE WORK

The paper has shown a part of a work in progress. We've described a proposal of agile methodology (MADD) that can be applied in the case of a distributed team that operates like an open source community. For the definition of this agile methodology the results of a questionnaire were relevant. The survey was supplied to the developers of different OS projects, and its results helped to better understand how agile values and practices are present in these communities. Part of the future work will be the results' analysis of the MADD experimentation, its eventual refining and its adoption in other scenarios and projects.

VIII. ACKNOWLEDGEMENTS

This work was supported by MAPS (Agile Methodologies for Software Production) research project [1], contract/grant sponsor: FIRB research fund of MIUR, contract/ grant number: RBNE01JRK8.

IX. REFERENCES

[1] MAPS Project, http://www.agilexp.org/
[2] Kircher, M., Jain, P., Corsaro, A., Levine, D.: Distributed eXtreme Programming, Proceedings of XP 2001, Villasimius, Italy, 20 - 23 May 2001, 66–71
[3] Kircher, M.: eXtreme Programming in Open-Source and Distributed Environments, JAOO (Java And Object-Orientation) conference, Aarhus, Denmark, 10 - 14 September 2001
[4] Kircher, M., Levine, D.: The XP of TAO – eXtreme Programming of Large, Opensource Frameworks, Proceedings of XP 2000, Cagliari, Italy, 21 - 23 June 2000
[5] Stotts, D., Williams, L., Nagappan, N., Baheti, P., Jen, D., Jackson, A.: Virtual Teaming: Experiments and Experiences with Distributed Pair Programming, XP/Agile Universe 2003, 129–141
[6] Baheti, P., Gehringer, E., Stotts, D.: Exploring the Efficacy of Distributed Pair Programming, XP/Agile Universe 2002, 208–220
[7] Brian F. Hanks: Distributed Pair Programming: An Empirical Study, XP/Agile Universe 2004, 81–91
[8] Soro, A., Sanna, R., Angioni, M., Carboni, D., Paddeu, G.: DJ-lab: Laboratorio di Informatica Distribuito, DIDAMATICA 2004, Ferrara, Italia
[9] Maurer, F., Martel, S.: Process Support for Distributed Extreme Programming Teams, Proceedings of the International Workshop on Global Software Development - ICSE 2002, Orlando, Florida, 21 May 2002
[10] Poole, C.J.: Distributed Product Development, Proceedings of XP 2004, LNCS 3092/2004, 60–67, Springer
[11] Raymond, E.S.: The Cathedral and the Bazaar, O'Reilly, Cambridge, Mass., 1999 12. Koch, S.: Agile Principles and Open Source Software Development: A Theoretical and Empirical Discussion, Proceedings of XP 2004, LNCS 3092/2004, 85–93, Springer
[13] SourceForge Web Site, http://www.sourceforge.net/
[14] Apache Web Site, http://www.apache.org/
[15] Codehaus Web Site, http://www.codehaus.org/
[16] XPSwiki Web Site, http://www.agilexp.org/xpswiki/