Università degli Studi di Cagliari

Dipartimento di Ingegneria Elettrica ed Elettronica
Dottorato di Ricerca in Ingegneria Elettronica ed Informatica

Settore Scientifico Disciplinare: ING–INF/05

Adopting Agile Methodologies in Distributed Software Development

Tutor:
Prof. Michele Marchesi

Tesi di Dottorato di:
Katiuscia Mannaro

Febbraio 2008

A mia madre

Contents

List of Figures

List of Tables

Introduction

From the second half of the '90s, some software engineering practitioners introduced a new group of software development methodologies called ***Agile Methodologies (AMs)***. Interest in Agile Methodologies has increased dramatically over these past years, not only in academic research, but also in an industrial perspective. These new methodologies have been developed to overcome the limits of the traditional approaches in the software development. In the '90s it was evident that only a limited number of software projects has success and this led to the elaboration of methodologies adaptable to new internet applications or mobile devices.

Moreover the ***FLOSS*** (Free Libre Open Source Software) has been proposed as possible different solution to the software crisis that is afflicting the ICT worldwide business.
If the AMs improve the quality code and allow to respond quickly to requirement changes, on the other hand, FLOSS approach decreases the development costs and increases the spreading of competences about the software products.

A debate is shaping about the compatibility of these two approaches.

Now, more often than not, software development teams have been spreading around the world, with users in Europe, management in the USA and programmers in the USA and India. The scattering of team members and functions around the world introduces barriers to productivity, cultural and languages differences can lead to misunderstanding of requirements, time zone differences can delay project schedules.

Agile methods can provide a competitive advantage by delivering early, simplifying communication and allowing the business to respond more quickly to the market by changing the software. Trying to distribute a development project in an agile way isn't easy and will involve compromises. However, there is a number of benefits that can be gained both by the enlightened businesses and developers.

The study and the application of the Agile Methodologies for the software development, as validation, modeling, impact on firm organization and their use for the distributed development and for the development of open source software, represents the main part of the research activity of my PhD course. The goal of this

thesis, also on the basis of this debate, is to determine the application of the AMs in several contexts so to define which of these can be used effectively in non traditional software projects as distributed technologies or open source development. In particular the peculiarities of this kind of applications will be emphasized and moreover it will be put in evidence the agile practices for the software development that can be efficiently used in this context.

The thesis is organized as follows:

Chapter 1 gives an overview of the software engineering methodologies. On the specific the last part of the chapter outlines the differences between the *Agile Metholodogies* and the traditional methodologies such as *Plan-Driven*.

Chapter 2 introduces an analysis of the distributed development process and the problems related to the definition of the distributed development model based on agile methodologies.

Chapter 3 describes an emblematic example of distributed development: the Open Source Scenario.

In Chapter 4 three projects I've been involved in over the years of my PhD are described: the project MAD, the DART project and the Eurace project. These projects have allowed to find the best practices to adopt and to adapt in the distributed development.

Chapter 5 takes a close look at how to adopt a Distributed Agile Methodology. It provides a landscape overview of best practices and major tools for managing and executing agile software development projects across geographically and temporally distributed teams.

Finally Chapter 6 provides the conclusions of my work.

Chapter 1

Brief Overview of Software Development Methodologies

There are several definitions of the term *software development methodology* (SDM): in simple terms it is a collection of procedures, techniques, principles and tools used for software production. That is if you string an appropriate set of software practices together and this set accomplishes all the fundamental activities listed earlier, you create a software development methodology. Another name for software development methodology is *software development process* (SDP). In this thesis these terms are used interchangeably along with *software development lifecycle* (SDLC), *software development approach* and *software development method*.

A *software process model* (SPM) is a simplified description of a software development process. Because of the simplification, several software development methodologies may share one process model: the differentiation is in the details of the process itself. The model generally specifies the common techniques and philosophies, while the methodology "overrides" these general specifications with specifics. In object-oriented terms, *"process model"* is a class and "software development methodologies" are instances of the class.

How to manage changing requirements is at the core of the one of the major debates in software project management: the debate over "traditional" (*plan-driven*) and *agile* methodologies. The main distinction between the two methodologies is in their approach to change. The *plan-driven methodologies* have an implicit assumption that a good deal of information can be obtained up front, and they can be summarized as "Do it right the first time." These methodologies are very appropriate for projects for which numerous requirements and/or technology changes are not expected during the development cycle. Alternately, *agile methodologies* (AM) are considered best suited for projects in which a great deal of change is anticipated. They focus on spending a limited amount of time on planning and requirements gath-

ering early in the process and much more time planning and gathering requirements for small iterations throughout the entire lifecycle of the project. Agile methodologies can be summarized as "Do it right the last time."

This chapter provides a brief overview of the two different methodologies. In particular we will take no much time to describe some of the common development approaches and many of their specific practices, then we will compare the two methodologies.

In Fig. 1.1 we show a classification developed by Boehm [10]. He puts in order, in a scale from more disciplined to more free, the planning approaches adopted in several development methodologies. You can note that the Agile Methodologies (AM) are second only to cowboy coding [1].

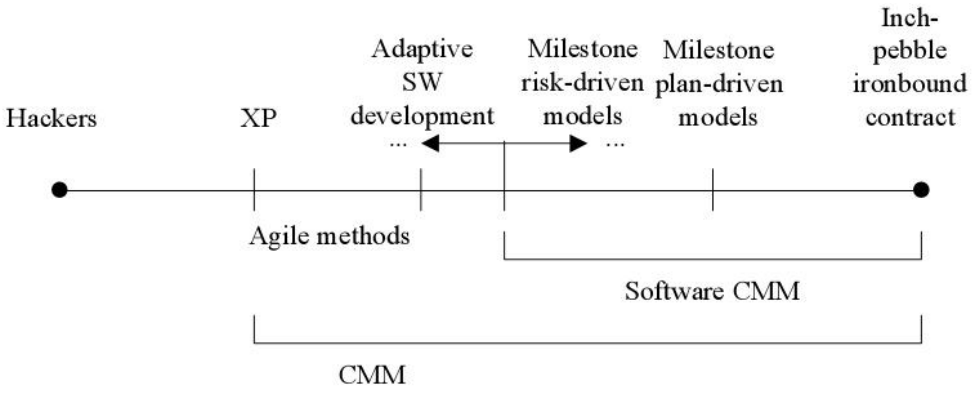

Figure 1.1: *The spectrum of several planning levels.*

1.1 Plan Driven Methodologies

The plan-driven methodologies are considered as the "traditional" way for software development and they have an implicit assumption that a good deal of information can be obtained up front. The product cycle is divided into several types of phases: from requirements gathering to analysis, from design to coding, that is the different stages of development are clearly separated to clarify what is to be done.

The plan-driven methodology that has dominated software development projects for decades is called *waterfall*. The Waterfall approach was developed by the U.S.

[1] "Cowboy Coding is a software development methodology without an actual defined method: team members do whatever they feel is right. Typical cowboy coding will involve no initial definition of the purpose or scope of the project, no formal description of the project, and will often involve one programmer. It is characterized by a single programmer jumping into the writing of the software often working from his own idea of what the software should do. It is often characterized by a lack of any documentation for either the requirements of the project or the design of the software overall." (Source: wikipedia.)

Navy in the '60s to enable development of complex military software. Winston Royce coined the term in 1970 to describe a serial method for managing software projects through the six stages shown in Figure 1.2. In its simplest form waterfall project consist of sequential phases of analysis, design, coding and test, with all functionalities being delivered at the end. Each phase is done by a different role or speciality, that is, analysts do analysis, designers do design, programmers write programs and testers test the programs. Here the different phases of software development are defined first, and then they are ordered in a strictly sequential way: the process never returns to its originating state. The Waterfall is quite formal, it is document-driven (it has a large number of deliverables) and the planning is its core value. The Waterfall approach can work well for tightly controlled and rigorous environments but has serious shortcomings in a business scenario.

The *incremental software development model* (see Fig. 1.3) is based firstly on developing the overall system architecture and then building in increments subsystems fitting that architecture. This provides a stricter integration among the different phases of development: the capture of requirements, the analysis of the system, the development of a product architecture, and the division of the product architecture into a set of independent parts. After the creation of a common architecture, the different subsystems are developed in "increments of development" or, simply, "increments". The different increments can be developed independently (thus simplifying the task of developers) and can be tracked (thus providing clear evidence of the progression of work).

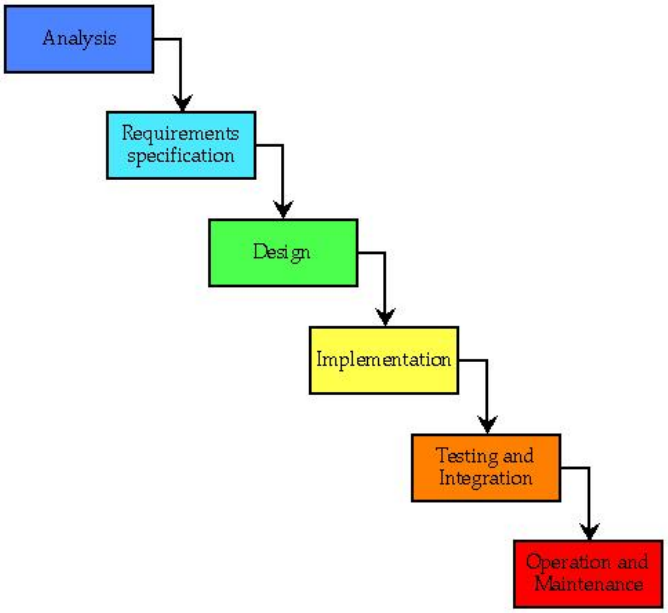

Figure 1.2: *The waterfall process for software development.*

In this context the requirements change is not well accepted because it is too expensive. Then the irreversibility would not be an issue, because almost all decisions would be considered irreversible. This approach is also considered as more suitable

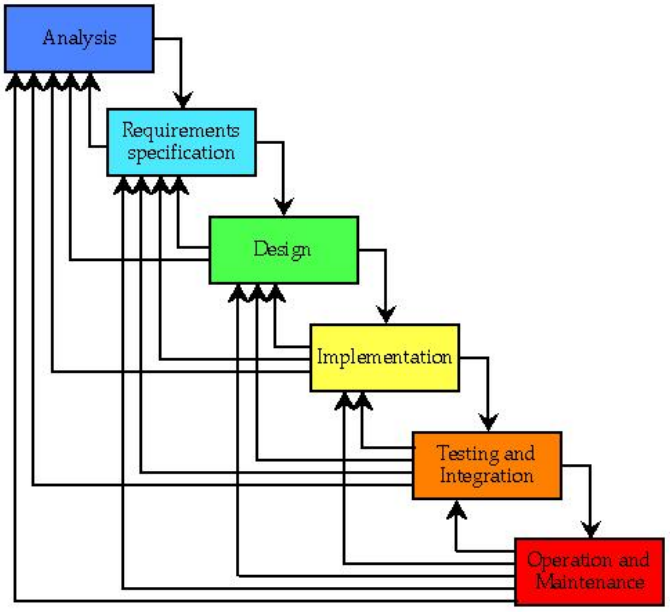

Figure 1.3: *Incremental Approach.*

for safety and mission critical because of their emphasis on defect prevention and elimination (e.g. operative system or control software of industrial plants) where the requirements are well defined and an initial good knowledge of them. But it is not adaptable in environments where it is not possible to have perfect understanding of the requirements from the start. As matter of fact stakeholders often do not know what they want and cannot articulate their requirements.

All methodologies have one underlying characteristic: they are designed to enable developers to solve customer problems. Successful software development is challenged by ability to manage complexity, technology innovation and requirements change. Not all of the factors are predictable. Software development is rarely a sequential task like in Figure 1.2, and even more rarely its tasks are predictable. In some circumstances, plans are inefficient or do not motivate the people who should enact them. In other circumstances, plans simply do not work. Plans simply have an inherent inadequacy to cope with certain problems. Successful software needs to cope with three factors to deliver software "on time and within budget":

Uncertainty: We do not often have a complete and accurate picture of the overall scenario of the software development;

Irreversibility: Some of the actions we take developing software cannot be (easily) undone;

Complexity: Software development may go out of control; we need to coordinate the actions of all the key stakeholders toward the goal of building a system that satisfies the customer.

Management of this complexity requires a software methodology able to consider that change in software development is not an avoidable evil, but is rather part of

the essence of software development itself.

The alternatives to Plan-Driven Methodologies are called Agile Methodologies (AM). They usually require that organizations have more agility, absence of a big architectural phase upfront, requirements to keep the system flexible, and possibility to stop the development at any time, while still providing the customer with something valuable.

1.2 Agile Methodologies

In response to questions as how to develop systems quickly while staying open to change, maintaining quality, and controlling costs, new light-weight development methodologies began to emerge during the '90s. Collectively these approaches are known as *Agile methodologies* (AMs). AM was proposed in 2001, with an initial 17 signatories, as possible solution to the software crisis that is afflicting the ICT worldwide business. Agile software development isn't a set of tools or a single methodology, but a philosophy with a significant departure from the heavyweight document-driven software development methodologies, such as waterfall, in general in use at the time.

Agile processes allow to respond quickly to requirement changes and stress collaboration between software developers and customers (customer-driven) and early product delivery. From a work flow point of view, they deliver software in small, frequent (e.g. weekly) iterations. At the end of each iteration the software is fully built and tested and could be released to the customer. The customer may reprioritise the requirements to benefit from any learning.

The publication of the "Manifesto for Agile Software Development" [2] establishes a common framework for these processes: *"Value individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan."* [37]

Principles of the "Agile Manifesto" [3]*:*

- Highest priority is to satisfy the customer through early and continuous delivery of valuable software;
- Welcome changing requirements, even late in development. (This is an advantage for the customer and provides a surplus for agile processes.);
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale;

[2] Visit http://www.agilemanifesto.org/ to read about the manifesto. The site includes an overview of agile concepts and viewpoints of the original signatories.

[3] From the Agile Alliance: www.agilealliance.com

- Business people and developers must work together daily throughout the project;
- Developers are motivated, work in the right environment and they have all resources necessary for them. Moreover they must be willing and capable of working with others;
- The most efficient and effective method of conveying information to and within a development is face-to-face conversation;
- Working software, also with limited functionalities, is the primary measure of progress;
- Agile processes promote sustainable development;
- Continuous attention to technical excellence and good design;
- Simplify the way you do things and eliminate the waste in what you do.
- The best architectures, requirements, and design emerge from self-organizing teams;
- At regular intervals, the team reflect on how to become more effective, then tune and adjust its behavior accordingly. Team members focus first on what is most important and most urgent.

Adapting to changes is one of the best practices for software development processes. We define some practices used to define agile software development methodologies. These practices are as follows:

- Iterative approach: small software releases, with rapid cycles. Developing in iterations allows the development team to adapt quickly to changing requirements;
- Incremental approach: the project is divided in small bite-sized pieces and as a result, we can get feedback on the small pieces, which allows to iterate or refine the deliverable. New functionalities are added with each new release;
- Time Bounded: each increment (called also iteration) has a fixed duration;
- People oriented: AMs work best with small teams, it promotes a strong team spirit, helps to empower developers to be part of evolving the process and to improve their own productivity. As the team grows, communication becomes more of a challenge.
- Adaptive: an agile process allows new activities to be added as well as allowing existing activities to be modified as required;
- Parsimonious: the developers should focus first on those activities that are most important and most urgent for the customer. In this way we address what has the highest return first, creating the highest possible value.

- Collaborative: Developers are collaborative, work constantly together with close communication.

All these practices can change with the AMs used but in general they are classified as belonging to three general areas [9]:

1. ***Communication*** (e.g., metaphor, pair programming);
2. ***Management*** (e.g., planning-game, fast cycle/frequent delivery);
3. ***Technical*** (e.g. simple design, refactoring, test-driven design).

All Agile Methodologies have common values (that is the "philosophy" agile), but each approach differs in execution. The focus is for all AMs on delivering working software. The foremost of these Agile methodologies are Extreme Programming and Scrum.

Extreme Programming (XP) [6] is undoubtedly the hottest Agile approach to emerge in recent years. XP is a collection of values, principles and practices to maximize the software quality. It defines some practices, in particular related to requirements elicitation and coding phases in order to make the process lighter and changes adaptable, as well as informal and customer oriented. These practices are organized in "feedback cycles", the most relevant are the phase of requirements gathering (*"User Stories"*), coding (*Pair Programming and Refactoring*) and testing, during the development (*TDD - Test Driven Development*) and validation by the customer (*Acceptance Test*). XP has become the most used and famous approach in agile methods so that many practices like *unit testing* and *refactoring* have been used also over the AMs.

SCRUM [49] emphasizes a set of project management values and practices, rather than those in requirements, implementation, and so on. As such, it is easily combined with or complementary to other methods.

In table 1.1 we present a brief description of some Agile methodologies, such as also ***Feature Driven Development*** (FDD) [48], ***Adaptive Software Development*** (ASD) [28], ***Agile Modeling*** (AM) [1], ***Crystal*** [18] e ***Dynamic Systems Development*** (DSDM) [29].

Among the MAs in table, only XP and Crystal propose practices for the whole software lifecycle; the others cover only a part (FDD e SCRUM), or they are frameworks (DSDM), or cultural approaches (ASD e AM) that need other agile methodologies. In particular, ASD is the most abstract, and it has a goal: to role the individual relationships; diametrically opposed, we can find Crystal: the one that proposes concrete practices adapting to several projects in size and so on.

1.2.1 Agile practices

The Agile methodologies are not either alternative, but rather they are complementary. A few of these give only core principles and just a few little of them

Table 1.1: *AM's Key Features.*

Agile Approach	Key Features	Special Features	Weakness
XP	Customer-oriented, small team size, small and frequent releases.	Refactoring	No management practices
FDD	Process based on 5 steps, object development model, short iterations (max 2 weeks).	Simple model, system implementation in features.	Manage only design and implementation; it needs other approaches (e.g. for the requirements, etc.)
SCRUM	Process management, self-directed and self-organizing team, 30-calendar day iterations (sprint)	Backlog (wish-list of features to implement)	No code and testing practices (it needs other approaches).
ASDS	Adaptive Approach, mission-driven development, feature-based and iterativ	The caos is governed with self-organizing team.	Such as SCRUM, it doesn't define development practices; it is a cultural approach.
DSDM	More a framework than a methodology; formalization of the RAD (Rapid Application Development) practices	It tries to make the waterfall approach more flexible (changes are reversible)	This is a framework and it doesn't define team size or iteration time, etc.
Crystal	Methodologies for several type of projects, incremental development, max 4 months per iteration.	Weaknesses and project size imply the methodology to use.	Exist only 2 methods on 4 expected; little attention to the communication in the team, so not very adapted for distributed teams.
AM (Agile Modeling)	Agile principles applied to the models and documentation; a lot of values are common to XP's these (communication, simplicity, feedback, courage)	Cultural approach fostering communication and organization; strictly practices oriented (11 best practices in 4 categories).	Philosophic approach complementary and parallel to methodologies such as XP or SCRUM.

gives effective practices for all phases of the process. This paragraph overviews AM core practices, all of which you must adopt to claim that you are truly following an agile process. Most of these come from the XP because in this approach the agile practices have been well formalized, and we can find them in the *Agile Modeling* that supports and applies them in the modeling. In the explicative diagram of Fig. 1.4 [4] we show the XP development.

Small team size (the team must be composed by max 10 - 15 developers).

[4]From www.extremeprogramming.org

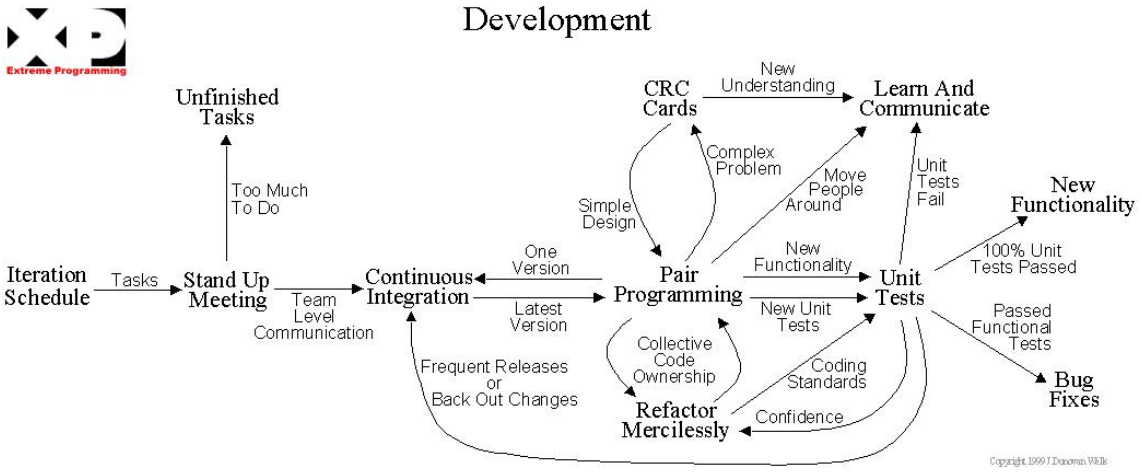

Figure 1.4: *XP practices.*

This favours the communication and the lightweight of the process.

Pair Programming. Two developers sit at the same workstation and share development tasks. They take turns writing code, reviewing and thinking. Laury Williams [30], [35], from North Carolina University, has conducted research into the costs and benefits of Pair Programming, and her study found some verifiable economic benefits (e.g.: pair programming yielded at least 15 % less defects than individual programming.) but also less tangible aspects such as better quality work, higher morale and increased creativity. In [4] it is interesting to note that this practice gives good benefits also if the team is not localized in the same workstation;

Continuous Integration. Frequently releases of new versions of the system and short iterations. Developers should be integrating changes every two hours in order to ease integration headaches.

Use cases driven, with time and costs estimated. This focuses on concrete needs and on the way approaching to the problem.

Continuous refactoring. Refactoring is the technique of improving code without changing functionality, it is an ongoing process of simplification that applies to code, design, test, all process. Whenever you saw the chance to make the system simpler in order to improve the quality, reduce duplication [5].

TDD (Test Driven Development). Rather than test after development, developers must write the test before coding. Tests must be automated, that is you have a complete suite of tests for your system and you works on the class continues until it passes all the test in the suite. Testing is the basis for refactoring.

User Acceptance Testing. Customers write tests demonstrating that features are finished.

[5] http://www.refactoring.com

No external documents. Documents are the same code.

Simple design. The system should be designed as simply as possible.

Collective ownership. Anyone can change any code anywhere in the system.

On-site customer. The user is available full-time to answer questions.

User Stories. The functionalities of the system are described using stories, short descriptions of functionalities visible to the customer (see Fig. 1.5). Stories also drive system development.

Planning Game It is the primary tool used for planning, it involves customers and programmers working through user stories, estimating and prioritizing. This practice is informal, lightweight and pragmatic. The output of this phase is a release schedule: all must have a common understanding of what the release will contain.

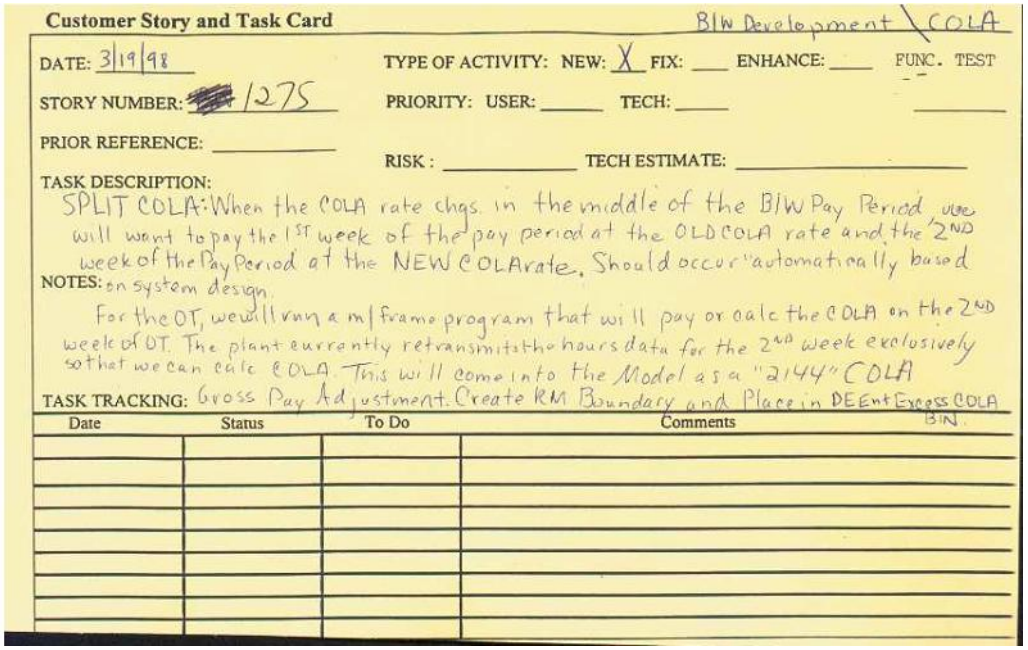

Figure 1.5: *Example of User Story on the card index.*

1.3 Two Agile Software Development Methodologies: XP and Scrum

The most widely used methodologies based on the agile philosophy are *Extreme Programming* (XP) and *Scrum*. These differ in particulars but share the iterative approach described above. XP, on the contrary compared to Scrum, concentrates on the development rather than managerial aspects of software projects. XP is designed so that organizations would be free to adopt all or part of the methodology.

1.3.1 Extreme Programming

Extreme Programming (XP) [6] is the agile methodology that has received most attention in these last years. In XP there is a strong emphasis on informal and immediate communication, automated tests and pair programming.

XP was created by Kent Beck and Ward Cunningham during a Chrysler Corporation's project in which an integrated system for paying the salaries (Chrysler Comprehensive Compensation's Project or C3) was developed. This experience was then formalized and published in 1999 by Kent Beck in the first edition of "Extreme Programming Explained–Embrace Change" [6]. He defined a set of 12 practices that embody four fundamental values: *communication*, *feedback*, *courage*, *simplicity*.

Five years later a new edition of the same book appeared, with a renewed vision of the same methodology. This new version is based on the lessons learned by the XP community since the publication of the first book. The new XP [8] includes five values (communication, feedback, courage, simplicity and respect), fourteen principles, thirteen primary practices and eleven corollary practices.

XP is driven by a set of shared *values*. The *practices* are what XP teams use each day to develop systems and they are complementary. *Principles* guide software development and they are the link between values and practices, in fact throughout the principles the values become practices.

XP values are the following:

Communication This is the most important aspect in the development software. XP practices are designed to encourage interaction, developer-to-developer and developer-to-customer. The focus is on oral communication and not on documents, reports and plans.

Simplicity. The software system must be the simplest thing that could possibly work. Naturally customer requirements must be satisfied. The XP originators contend that it is better to do a simple thing today and pay a little more tomorrow for change than to do a more complicated thing today that may never be used.

Feedback. The feedback is vital to the success of any software project. In XP the feedback must be constant and concrete. XP teams obtain feedback by testing their software early and often. They deliver the system to the customers in order to demonstrate it and implement changes and re-prioritization as suggested by feedback from the customer.

Courage. The courage is the confidence to make changes also radical in order to improve the project. It is the ability to work rapidly and redevelop if required.

Respect. The previous four values imply a fifth: respect. If members of a team don't care about each other and their work, no methodology can work.

[6] Visit www.extremeprogramming.org for a full description of XP.

XP Development

XP projects start with a release planning phase, followed by several iterations, each of which concludes with user acceptance testing. When the product has enough features to satisfy users, the team terminates iteration and releases the software.

Users write *"user stories"* to describe the need the software should fulfill. User stories help the team to estimate the time and resources necessary to build the release and to define user acceptance tests. A user or a representative is part of the XP team, so s/he can add detail to requirements as the software is being built. This allows requirements to evolve as both users and developers define what the product will look like.

To create a release plan, the team breaks up the development tasks into iterations. The release plan defines each iteration plan, which drives the development for that iteration. At the end of an iteration, users perform acceptance tests against the user stories. If they find bugs, fixing the bugs becomes a step in the next iteration.

XP rules and concepts

Here the most important concepts follow: [7]

Integrate often. Development teams must integrate changes into the development baseline at least once a day. This concept is also called *"continuous integration"*.

Project velocity. Velocity is a measure of how much work is getting done on the project. This important metric drives release planning and schedule updates.

Pair programming. All code for a production release is created by two people working together at a single computer. XP proposes that two coders working together will satisfy user stories at the same rate as two coders working alone, but with much higher quality.

User story. A user story describes problems to be solved by the system being built. These stories must be written by the user. Since user stories are short and somewhat vague, XP will only work if the customer representative is on hand to review and approve user story implementations. This is one of the main objections to the XP methodology, but also one of its greatest strengths.

1.3.2 Scrum

The term Scrum comes from a 1986 study by Takeuchi and Nonaka [54] that was published in the Harvard Business Review. In that study, Takeuchi and Nonaka

[7] The full list of XP rules and concepts is available at www.extremeprogramming.org/rules.html.

note that projects using small, cross-functional teams historically produce the best results. They write that these high-performing teams were like the Scrum formation in Rugby. When Jeff Sutherland developed the Scrum process [8] at Easel Corporation in 1993, he used their study as the basis for team formation and adopted their analogy as the name of the process as a whole. Ken Schwaber [49] formalized the process for the worldwide software industry in the first published paper on Scrum at OOPSLA 1995.

In contrast with XP, Scrum is based on a managerial process.

Scrum is a simple and adaptable framework that has three roles, three ceremonies, and three artifacts:

- Roles:
    1. Product Owner;
    2. Scrum Master;
    3. Team.
- Ceremonies:
    1. Sprint Planning;
    2. Sprint Review;
    3. Daily Scrum Meeting.
- Artifacts:
    1. Product Backlog;
    2. Sprint Backlog;
    3. Burndown Chart.

In Fig. 1.6 we show the Scrum Process.

Key Roles and Process

First of all we will describe the key roles and then the process.

Product Owner: S/he is usually the client who commissions the project or the product. The Product Owner can be a customer representative, for example for product companies the customer is a market, and the Product Owner serves as a proxy for the market. A Product Owner needs a vision for the product that frames its ultimate purpose, a business plan that shows what revenue streams can be anticipated from the product in which time frames, and a road map that plans out several releases, with features ordered by contribution to return on investment (ROI). S/he prepares a list of customer requirements prioritized by business value.

[8] Visit www.scrumalliance.org. for a full description of Scrum.

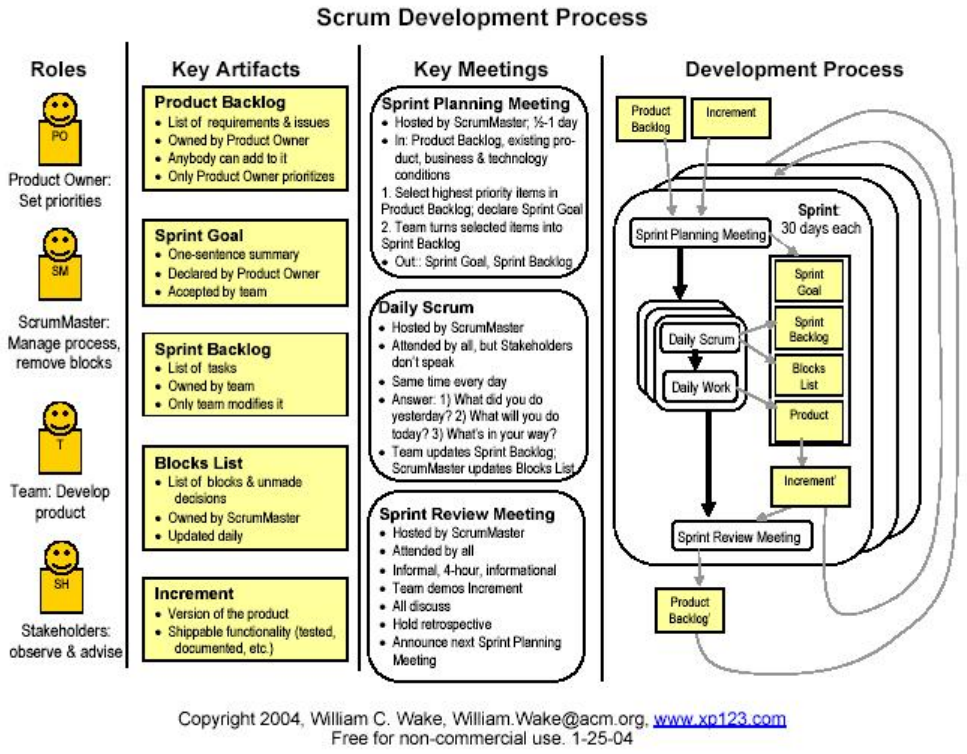

Figure 1.6: *The Scrum Development Process*

This list is the ***Product Backlog***, a single list of features prioritized by value delivered to the customer. S/he has the following responsibilities:

- Define the features of the product;
- Decide on release date and content;
- Be responsible for the profitability of the product (ROI);
- Prioritize features according to market value;
- Adjust features and priority every 30 days, as needed;
- Accept or reject work results.

Scrum Master: S/he is a facilitative team leader working close to the Product Owner. S/He must ensure that the team is fully functional and productive and the process is used as intended. The Scrum Master needs to know what tasks have been completed, what tasks have started, any new tasks that have been discovered, and any estimates that may have been changed. This makes it possible to update the Burndown Chart which shows the cumulative work remaining day by day. The Scrum Master must also look carefully at the number of open tasks in progress. Work in progress needs to be minimized to achieve lean productivity gains; S/He helps to remove impediments and shields the team from external interferences. Last but not least, the Scrum Master may notice personal problems or conflicts within

the Scrum that need resolution. These need to be clarified by the ScrumMaster and be resolved by dialogue within the team, or the Scrum Master may need help from management or the Human Resources. The Scrum Master must pay attention to ensure the team is fully functional and productive.

Team Members: They are a cross-functional group of people with all the different skills that are needed to turn requirements into features, and features into working software. The group organizes themselves and their work and demos work results to the Product Owner. Moreover the team has the right to do everything within the boundaries of the project guidelines to reach the goal.

The Scrum Process

Scrum begins with the establishment of a *"product backlog"*, that is to say a list of customer requirements. Each element of the backlog is prioritized as to what is most likely to deliver value, and the highest is worked on first. Backlog should start relatively high, get higher during planning, and then be whittled away as the project proceeds, either by being solved or removed, until the product is completed. Under Scrum, each iteration is a *"Sprint"* of around a month's duration. Each Sprint (iteration) starts with a planning meeting, a *"Sprint Planning Meeting"*, it decides what features to implement in the Sprint. It begins with the Product Owner reviewing the vision, the roadmap, the release plan, and the Product Backlog with the Scrum team. The team reviews the estimates for features on the Product Backlog and confirms that they are as accurate as possible. The team decides how much work it can successfully take into the sprint based on team size, available hours, and level of team productivity. It is important that the team "pull" items from the top of the Product Backlog that they can commit to deliver in a thirty-day sprint. Pull systems have been shown to deliver significant productivity gains in lean product development. When the Scrum team has selected and committed to deliver a set of top priority features from the Product Backlog, the Scrum Master leads the team in a planning session to break down Product Backlogs features into sprint tasks. These are the specific development activities required to implement a feature, for each Task is assigned an estimated effort and who is responsible for doing the work and form the Sprint Backlog. This phase of the Sprint Planning Meeting is time-boxed to a maximum of four hours. These tasks are broken down into pieces that will require less than two days (or sixteen developer-hours) of work. When the Sprint Backlog is complete, the total work estimated is compared with original estimates from the Product Backlog. If there is a significant difference, the team negotiates with the Product Owner to get the right amount of work to take into the Sprint with a high probability of success. Each day the Scrum Master leads the team in the *Daily Scrum Meeting*, time-boxed to 15 minutes. During the meeting the team members synchronize their work and progress and identify impediments to productivity. Each team member should answer three questions: what did I do yesterday, what did I do today, and what impediments got in my way? While anyone can attend this meeting, only team members who have committed to deliver work to the Scrum are allowed to speak. At the end of each Sprint there is a Sprint

Review Meeting that provides an inspection of project progress. This meeting is time-boxed to a maximum of four hours. The first half of the meeting is set aside to demonstrate to the Product Owner the potentially shippable code that has been developed during the sprint. The Product Owner leads this part of the meeting and invites all interested stakeholders to attend. The state of the business, the market, and the technology are reviewed. The Product Owner determines which items on the Product Backlog have been completed in the Sprint, and discusses with the Scrum team and stakeholders how best to reprioritize the Product Backlog for the next sprint. The goal for the next sprint is defined. Moreover a Sprint Retrospective meeting is also held after each Sprint to discuss the just concluded Sprint and to improve the process itself. This second half of the Sprint Review Meeting is led by the ScrumMaster. The team assesses the way they worked together in the sprint and identifies positive ways of working together that can be encouraged as future practice and develops strategies for improvement. After the Scrum Review Meeting, the process begins again. Iterations proceed until enough features have been done to complete or release a product. In order to guide the development team to successful completion of a Sprint on time, Scrum uses a tool defined *Burndown Chart*. The Burndown Chart shows the cumulative work remaining in a Sprint, day-by-day. At the Sprint Planning Meeting the Scrum Team identifies and estimates specific tasks that must be completed for the Sprint to be successful. The total of all Sprint Backlog estimates of work remaining to be completed is the cumulative backlog. When tasks are completed as the Sprint proceeds, the Scrum Master recalculates the remaining work to be done and the Sprint Backlog decreases, or burns down over time. It the cumulative Sprint Backlog is zero at the end of the Sprint, the Sprint is successful.

1.4 Comparing Agile Software Development and Plan-Driven Methodologies

In this section we will compare Agile Methodologies with Plan-Driven Methodologies, also if this is not very simple because the requirements for each software development process are different. However, there are several important software project characteristics for which there are clear differences that help to choose between agile and plan-driven methods.

Software development methodologies grew out of the desire to deliver high-quality software products to customers. In a plan-driven approach each project is divided into tasks, tasks are linked together via deliverables, and so on. As we said earlier, the cornerstone of this approach is the plan. But not all the projects are identical. Nor can all of them be organized and managed using the same planning technique A plan requires a clear definition of actions to accomplish and a good knowledge of the target environment.

The world of software development is characterized by uncertainty and irre-

versibility, which means that we do not often know all the details of what to do. In some circumstances, plans are inefficient or do not motivate the people who should enact them. In other circumstances, plans simply do work. Plans simply have an inherent inadequacy to cope with certain problems and projects.

Plan-Driven methodologies attempt to control the outcome of software development by adding restrictions and tightly monitoring change, so they work well where change is either slow and manageable. Agile methodologies allow for change in the project lifecycle and improve quality by resolving defects early and provide constant feedback on the product.

1.4.1 Fundamental Assumption

In the Plan-Driven methodologies the systems are fully specifiable, predictable, and are built through meticulous and extensive planning. If there are not requirements change the up-front planning can work and it is possible to achieve the objectives. But rarely it is possible to create up-front unchanging and detailed specifications. The users are not sure what they want and many details of what they want will only be revealed during development. External forces (such as a competitor's product) lead to changes.

In Agile methodologies a high-quality adaptive software is developed by small teams using the principles of continuous design improvement and testing based on rapid feedback and change. The focus of the Agile methodologies is on delivering valuable software for the customer and staying open to change maintaining quality. So they use rapid and short feedback loops to quickly learn if to-date product is meeting customer's needs, and they design for only the current iteration (simplicity).

1.4.2 Size

AMs seem to work best with small to medium teams developing software on small applications. But there are many case studies on big projects that had success. For example it is possible to split a big project in projects more little assigning them to small teams. When this solution in not practicable then Plan-Driven methodologies are welcome.

Plan-Driven methodologies are adapted to large-sized projects. The plans, documentation, improve the communication and the coordination across large teams. Moreover a bureaucratic plan-driven organization is not very efficient on small projects.

1.4.3 Environments

AMs are adapted for turbulent, high-chance environments where the requirements change frequently. The changes for this approach must be a resource and the

practices are a support for managing the unpredictable projects.

Plan-Driven methodologies work well when the requirements are determinable in advance. The change is difficult to manage.

1.4.4 Customer Relationships

Agile methodologies are customer focused. The customer defines what the project is, the business value and the direction of the project and so s/he takes part in the project working through business requirements, prioritizing and defining. In AMs the high level of communication and feedback that team thrives on is a feature. The customer representative must be on the project full-time for the duration: s/he can be on-site (*customer on-site*) and forms part of the team, or s/he is available on site only in fixed day.

Plan-Driven methodologies formalize the relationships between developers and customer throughout detailed contracts. The process is performed, planned, and checked, based on well-defined and documented procedures. But this approach is advantageous only when the requirements are stable, that is the customers know what they want and the contract protects them. But a detailed contract causes delays in the projects and it is very difficult to adapt it when new requirements are revealed during development.

1.4.5 Planning and Control

Plan-Driven methodologies are based on a rigid planning of the process and an amount of documentation is required. This approach is very appropriate for projects for which numerous requirements and/or technology changes are not expected during the development cycle.

Alternately, Agile methodologies are considered best suited for projects in which a great deal of change is anticipated. Spending significant amounts of time creating and inspecting an architecture and detailed design for the whole project is not advisable; it will only change too. The Agile methodologies spend a limited amount of time on planning and requirements gathering early in the process and much more time planning and gathering requirements for small iterations throughout the entire lifecycle of the project.

1.4.6 Communication

AMs focus on the face-to-face communication . This activity is executed throughout practices such as "*stand-up meetings*", *"pair programming"* and *"planning-game"*. This approach is effective for small-sized time but it in not very scalable in

distributed teams.

Plan-Driven methodologies use explicit, documented knowledge. They look for sign offs, written confirmation and documentation. The communication tends to be one-way, that is from one entity to another rather than between two entities.

1.4.7 Requirements

Understanding the customer's requirements is vital to the success of any project.

In the AMs the requirements are informal. User stories are a lightweight way of capturing customer's requirements. Customers are opening dialogue with the developers about what they want, and this approach allows requirements evolving.

In Plan-Driven approach the customer defines his requirements through functional specifications (use cases or prototyping). These are detailed for both estimation and design purposes, they are very structured. Development will not start until the functional specification is signed off. In this approach the phase of requirements gathering is very rigorous and it is the basis for the formal contract between developers and customers. There are situations where formal requirements specifications are required, for example in order to achieve certification. ISO 9001:2000 certification requires the organization to define specific procedures to establish how each significant activity in their development process is conducted. This means that the organization must always be able to show objective evidence of what is planned, what has been done, how it was done, the current status of the project and product, and it must be able to demonstrate the effectiveness of its quality system. This doesn't seem possible using an Agile approach. But we demonstrated that it is possible to certificate also XP process, and the use of an appropriate tool, that support both XP practices and ISO 9000 standard, can simplify this integration process [40].

1.4.8 Development

One of the main features of the AMs is the (*"simple design"*), that is the design must be as simple as possible in order to reduce the effort related to the changes in progress. Adding new functionalities to the system implies an effort according to the system complexity. Additional functionalities that are not useful at the moment should be avoided because they make the system more complex and so they are not necessary.

Plan-Driven methodologies are based on a different concept from the *"simple design"* which could be defined as "architectural design". It is an approach to design based on the architecture and planning in order to anticipate changes. In this case the component reuse is the better choice. The aim is to develop the system to be able to support new functionalities needed in the future. Therefore the simpler solution

is discarded for a more complex solution which can be reused. This approach shows again its predictive nature rather than adaptive.

1.4.9 Testing

Testing is one way of validating customers' specifications. Testing could be very expensive if done only at the end of the project and not systematically during development. AMs and Plan-Driven methodologies face this problem in different ways.

In the agile approach there are some XP practice which support the testing, such as the *"test first programming"* that performs the automated testing before writing the code. Furthermore the *"pair programming"* reduces the probability of errors in the code since that two developers always check the code. Finally the *code integration*, according to which the code must be integrated several times a day, allows to identify and correct promptly the errors derived from the integration of different modules developed by several developers.

In Plan-Driven methodologies the testing implies too much bureaucracy making developer work very difficult and boring. Namely the testing is carried out with automated test suites to support the considerable planning and preparation before running tests. Plan-driven methods address the expensive late-fix problem by developing and checking requirements and architectural specifications early in the development process. This creates a consistent amount of documentation that may undergo breakage due to changing requirements. Under certain conditions the "traditional" testing is better than the agile, for instance when the documentation is relevant.

1.4.10 People

Besides the technical characteristics described above, you need analyze a few peculiar characteristics of developers and customers such as cultural aspects useful to identify the most suitable methodology.

Customers. Agile methodologies require a continuous availability of the customer towards the team. Sometimes the companies send key users who don't well play the role of interface between the real customer and the developers. This risk is reduced when the target customer represents the real needs of his/her company.

The plan-driven methodologies also could take advantages of the continuous availability of the customer. These methods don't necessarily need this practice because they have requirement specifications and further documents which let the developers work without any doubt.

Developers. Generally MAs are regarded as particularly depending on the ability, talent and skill of the developers.

Naturally plan-driven methodologies would rather use skilled developers. However, they are planned to allow all developers, even if they are less skilled, to contribute to the project. In addition these methodologies well define the roles and the activities within the team.

Agile and plan-driven culture. Agile developers embrace the philosophy according to which programming is an art rather than a technic. The role of the developer is similar to that of the artist who models and creates objects rather than a simple worker. In the agile culture, people feel comfortable because they have many degrees of freedom available for them to define work.

In the Plan-Driven culture, people are comfortable when there are clear policier and procedures that define their role in the enterprise. Each of them has a limited knowledge of what others are actually doing. The programming in the Plan-Driven approach is more formal because there is a rigorous assignment of the roles, tasks and activities within the team.

1.5 Comparing XPers and NonXPers

Many XP projects have been completed but to our knowledge no quantitative study, to point out the efficiency of this light approach objectively compared to others Non-XP practices, has been accomplished yet. Here we will report the comparative results of our research study on job satisfaction of IT employees that use XP practices in their software development process and IT employees that do not use them [38].

The development of an effective, validated and reliable survey for evaluating job satisfaction is fundamental to this purpose.

We will describe the rationale and procedures for developing a survey to assess the job satisfaction of IT employees.

Understanding the factors affecting software development is one of the main goals of empirical software engineering, and explorative surveys are very useful to this purpose. Since our research interest is the opinions and perceptions of adopted software development methodologies, we have carried out an empirical study using a questionnaire, in order to gather quantitative data about the population of IT employees. We have chosen to perform an on-line Survey, using a commercially available web survey tool: Create Survey [9]. This tool helped us reduce time and development costs in carrying out the survey. Data can be directly entered by the respondents, avoiding the tedious error-prone tasks of manual data entry by an operator. A Web-based survey presents a number of advantages compared to

[9] www.createsurvey.com

telephone, e-mail or other traditional techniques. For instance, in traditional paper-based surveys, questions can be passed over for lack of care, or other reasons. On the contrary, on-line surveys allow to answer a subsequent question only if the previous one has been answered. Moreover, we chose a web-based survey also because all the members of our sample population are daily web-browsers users.

Structure of the Survey

The research process has been carried out according to the following phases:

- Formulation of the problem by establishing study objectives and a research plan: GQM approach;
- Sampling by definition of population characteristics;
- Data gathering;
- Data processing and data analysis;
- Research report.

GQM approach

To structure our research, we followed the Goal-Questions-Metrics (GQM) approach, the well known paradigm proposed by Victor Basili [5]. Our purpose in this survey is to understand the feelings and satisfaction of XP users (XPers) and non-XP users (Non-XPers) about their software development process. A secondary goal is to evaluate how Non-Xpers consider XP practices and their possible adoption. The GQM framework of the research is shown in Table 1.2.

Once the goals have been defined, a set of questions is used to achieve a specific goal, while a set of metrics may be associated with every question. We developed two questionnaires, one for persons using XP practices and one for those not using XP practices.

Data Gathering

In order to avoid biases and ensure greater homogeneity in the answers, the period of data collection was limited to the Autumn-Winter 2003-2004.

The quantitative survey uses a non-systematic sampling approach. Though a probability sample was not drawn, the significance of the sample is guaranteed by the fact that the respondents have been recruited in many and very different ways, including mailing lists, newsgroups, interpersonal relations, and by self-recruiting (many developers spontaneously found the questionnaire on our web-site and decided to answer).

Table 1.2: GQM Approach

Goal	*Purpose*	Evaluate
	Issue	the job satisfaction related to
	Object	the software development process adopted
	Viewpoint	from the viewpoint of XP and Non-XP software developers.
Question	*Q1*	Is there a difference in background and application sector between XP and Non-XP developers?
Metrics	*M1*	Background Questions: age, gender, level of education completed, kind of product/service developed,...
Question	*Q2*	What is the job satisfaction and the team's work productivity as perceived by XP Users and Non-XP Users?
Metrics	*M2*	Assessment of quality of life and quality on the job of the developers, on the basis of the adopted software development method.
	M3	Productivity rating by the developers on the basis of the adopted software development method.
Question	*Q3*	How are XP practices evaluated by XP developers and Non-XP developers?
Metrics	*M4*	Subjective evaluation by the IT manager and IT developers.
Question	*Q4*	How willing are non-XP developers to adopt XP practices?
Metrics	*M5*	Subjective evaluation by the Non-XP developers.

It is impossible to eliminate survey errors completely, but we have taken special care to avoid obvious errors. A sample survey is subject to four major sources of error [22]. In this survey we have taken the following steps to obviate them:

Coverage Error We believe coverage error is low, because the analyzed population is a population of Internet users. This makes sample bias a non-concern in this population.

Sampling Error The web-based survey has been carried out using a sample of available and volunteer respondents.

Measurement Error We cannot check the results of inaccurate responses, because we do not know whether the respondents understood the questions correctly. However, some questions have very similar goals, and are written in several ways. So a check was made on the answers to these questions in order to eliminate incoherent responses.

Non Response Error Respondents can only go on to the next question after having answered the previous one.

Design of Questionnaire

We have developed 69 questions for XPers and 57 questions for Non-XPers. For the sake of brevity we cannot report every question. The full questionnaires are available on our web site.[10]

Background Questions

We organized the questionnaires in various sections. This first and last part of the survey include questions about personal data, providing several data capable of classifying XPers and Non-XPers.

Satisfaction Rating Questions

The second section proposes the questions about satisfaction which are quite similar in the two questionnaires. Job satisfaction is analyzed by comparing the effects of variables on satisfaction with overt behaviors. We related some economic variables to subjective variables and we adopted a scale from 1 to 6, where 1: *"I Strongly Disagree"* and 6: *"I Strongly Agree"*.

A major determinant of our survey is the empirical analysis on job satisfaction. By combining the rating of generic questions with the rating of specific questions on satisfaction, it is also possible to evaluate the job productivity of the sample. Some of these questions are: *"The adoption of XP practices/the development process adopted by my team has favoured a better management of my time"*; *"The co-ordination among the team members taking part in the project work is satisfactory"*; *"The team developers are highly motivated towards the present software development method"*.

[10] http://www.agilexp.org

Satisfaction on XP practices Rating Questions

A third section, included only in the XPers questionnaire, was needed to estimate their level of satisfaction in the use of XP practices in the project. We adopted a scale from 1 to 5, where 1: *"Very Dissatisfied"* and 5: *"Very Satisfied"*. Examples of questions are: *"How satisfied are you with Pair Programming?"*, *"How satisfied are you with On-site Customer?"*.

Potential XP User Rating Questions

Finally, we included a fourth section only in the Non-XPers questionnaire. These questions help to estimate the propensity of Non-XPers to use XP practices. We adopted a scale of 1 to 5, where 1: *"Potentially Not at All Desirable"* and 5: *"Potentially Very Desirable"*. Some of these statements are: *"The project is directed by the customer,who is available to answer questions, set priorities, and determine project requirements any time"*; *"Every developer may be free to modify any code"*.)

1.5.1 Results

Q1: Structure of the Population Sample

The population sample is made up of 55 XPers and 67 Non-XPers. In this section, we report some significant results about the answers received. We characterized our sample by studying the answers to the two questionnaires and the cross-correlations between them.

We found no significant statistical demographic difference between the two groups in terms of gender (91% male), age (a significant 75% of the respondents is aged between 26 to 40 years), and level of education (the majority has a bachelor's degree). Respondents by country are structured as follows:

- **XPers:** 64% Europe, 24% America, 2% Oceania, 9% Asia, 2% Africa
- **Non-Xpers:** 69% Europe, 20% America, 7% Oceania, 4% Asia.

The subdivision of Non-XPers respondents in relation to the particular software methodology adopted and their professional role is the following:
among the Non-XPers respondents, 35% use an Iterative Incremental process, 9% the more traditional Waterfall process, and 7% RUP. Eighteen percent use agile methodologies such as *Scrum*, *Feature Driven Development* or other agile customised processes, while 18% of the Non-XPers declare: *"We do not use any specific development process"*.

Q2: Job/Life Satisfaction and Productivity

The variables representing personal/familiar sphere, which cause an improvement in life quality, have shown a significant difference between XPers and Non-XPers (Table 1.3).

The adoption of XP practices seems to have a significant effect on job quality. In Table 1.4 the number of the variables representing job quality are been reduced to 6 macro areas:

1. the development process adopted favours the relationships and communication with colleagues (TC);
2. the job environment is pleasant and comfortable (JE);
3. the development process adopted has reduced the amount of perceived stress (RS);
4. the development process adopted has significantly increased the team's work productivity (P);
5. the developers are very motivated and have a positive attitude towards the project (M);
6. how respondents are willing to adopt the current development process again (W).

Table 1.3: *"The adoption of the adopted methodology has Reduced the perceived Stress during my job"* (ReducedStress); *"My job does not interfere with my Family Relationship and/or with the management of my spare time"* (FamilyRelations); *"The development process adopted by my team favours a better management of my time"* (TimeMngmt). 1=Strongly Disagree, 6=Strongly Agree

	ReducedStress		**FamilyRelations**		**TimeMngmt**	
	Mean	StdDev	Mean	StdDev	Mean	StdDev
Waterfall	2,8	1,10	2,6	1,14	2,2	1,30
Incremental Iterative	3,47	1,43	2,79	1,47	3,32	1,34
RUP	3,00	1,41	3,25	2,06	2,25	1,89
Agile practices	3,64	1,03	2,64	0,92	4,27	1,42
other	4,00	1,41	3,67	1,37	4,17	1,17
none	3,60	1,43	2,90	1,29	2,80	1,14
XP	4,51	1,35	3,71	1,36	4,27	1,22

Table 1.4: *Results (Mean and Standard deviation) regarding satisfaction on: Team Communication (**TC**), Job Environment (**JE**), Reduced Stress (**RS**), Productivity (**P**), Motivation (**M**), Willingness (**W**).* 1 = Strongly Disagree, 6 = Strongly Agree

	TC	**JE**	**RS**	**P**	**M**	**W**
XPers	4,12 (1,25)	4,33 (1,39)	4,51 (1,35)	4,75 (1,12)	4,54 (1,15)	5,35 (0,96)
NonXPers	3,46 (1,38)	3,96 (1,26)	3,49 (1,30)	3,78 (1,39)	3,79 (1,25)	3,47 (1,51)

We have defined a *Team Productivity Index*. *Agree/Disagree* variables with statements related to team productivity have been weighted in the following way:

- Concentration (10%)
- Work Attitude (10%)

- Productivity (40%)
- Team developers' high motivation towards current development process(10%)
- Release Planning (15%)
- Time spent to supply the first version of product since the beginning of the project (15%).

Using this index, we found that 78% of XPers versus 57% of Non-XPers believe that the adoption of the adopted process methodology has increased their team productivity. In this connection we can say that the former percentage is very important and significant in this survey, while the latter is not very significant because the sample of Non-XPers is very heterogeneous on account of the adopted methodology.

Q3, Q4: about XP practices

Of the XPers respondents, 85.5% claim that they have a medium to high knowledge of XP practices. We analyzed their satisfaction on the 12 XP practices, which were rated on a scale from 6 (*Very satisfied*) to 1 (*Very dissatisfied*).

Of XPers respondents, only 22% has had some difficulties adopting XP practices and 53% did not adopt the Metaphor practice while 65% of those adopting the Metaphor practice are satisfied with it. We also found that the same percentage of XPers respondents (27,3%) do not adopt the *Planning Game* and *On Site Customer* practices. All the remaining XP practices are adopted with satisfaction by the majority of XPers.

Pair Programming is felt to positively affect job satisfaction and quality: 72,7% claimed that Pair Programming speeds up the overall software development process.

We have measured the assessment of some XP elements, from the Non-XPers viewpoint, which was rated on a scale from 5 (*Very desirable*) to 1 (*Not at all desirable*). It is highlighted a positive attitude toward Pair Programming practice.

1.5.2 Considerations

We have presented some results from an experimental analysis on IT Employees Satisfaction by comparing XP practices with other software development methodologies.

It should be noted that the question whether *"I would like my Company to carry on adopting the present software development method"*, was answered with "*Agree*" by 92% of XPers and 40% of Non-XPers. Moreover the question whether *"I think I will adopt our current development process again in the future"*, was answered with "*Agree*" by 96.4% of XPers and 54.6% of Non-XPers.

Clearly, there is a very favourable feeling about XP practices, indicating that they ensure more satisfaction, the job environment is more comfortable and productivity increases. The tendency of Non-XPers towards XP core practices is positive and usually they are not very satisfied with their software development process.

Chapter 2

Distributed Development

Distributed development isn't a new thing. Many large software systems are developed across multiple sites. For example several teams working at different locations may be working on different parts of the same software product, perhaps with an integration team somewhere with the responsibility of overseeing the assembly of each part into a cohesive product.

The overwhelming majority of software produced in the open source world is created by individual hobbyist developers working from home offices or student terminal rooms. This type of development is characterised by the fact that almost every developer is physically separated and that a small number of trusted developers are responsible for integrating submissions from a wide pool of contributors. These trusted developers effectively work as editors deciding which submitted "patches" to accept, and they decide how each accepted patch will be worked into the released product. The developers rarely meet, and integration of each patch or commit happens relatively infrequently, certainly by extreme programming standards.

The multi-site corporate development effort and the typical open-source project are only two examples of distributed development. There is a number of different possibilities, each one driven by different motivations. These can be roughly categorised as *"distributed"* or *"dispersed"* development.

Distributed development. This usually means co-operation between several teams located at different sites. This includes large software companies developing a single product out of many parts where each part could be built at a separate location. This classification also includes efforts where the bulk of the work is outsourced to developing countries with a small team of consultants working on site with the business stakeholders.

Dispersed development. Dispersed development refers to individual developers located separately, working together over a network. One company consisting of a small group of experienced developers decided to work from home using high-bandwidth connections. This is essentially a lifestyle choice. Some companies prefer to hire contractors that work from home, so that they don't have

to provide office facilities for them during the project. Another example of dispersed development includes forming project teams of specialists that would be too expensive or impractical to relocate to a single site project room.

Here we will use the term *distributed development* and *dispersed development* with the same meaning.

Today there are different 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 and, for practical or economical reasons, work in a dispersed (or distributed) context.

Wide spreading of distributed development is increasing: a first scenario is the *outsourcing*, that is a company delegates the development of a module or part of a software to a different company. More precisely *outsourcing* is the delegation of non-core operations or jobs from internal production to an external entity (such as a subcontractor) that specializes in that operation. "*Outsourcing* is a business decision that is often made to focus on core competencies (Source: Wikipedia)." Outsourcing is the term originally applied to work sent overseas. Many people still use this term. However, work that is "outsourced" isn't necessarily sent offshore. Work may be sent to a company located in the US that specializes in some particular aspect of a business process that, as indicated above, is not part of the organization's core competencies.

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). *Offshoring* is relocating business processes to a lower cost location overseas. (Source: Wikipedia)

Onshoring or *homeshoring* is relocating business processes to a lower cost location in the US, for example, usually in smaller cities where workers can work at lower cost than in larger urban areas.

Nearshoring is a form of outsourcing in which business processes are relocated to locations that are geographically close.

Co-sourcing relies on a long-term, strategic, and symbiotic relationship between a client and a vendor.

In *International partnerships in research* dispersion is encouraged to enforce knowledge transfer and researchers mobility. Several teams based in universities or research centers all over the world need to cooperate, communicate and coordinate their work, regardless of geographical distance.

Last but not least *Open Source projects* are typically projects where the programmers are constantly dislocated, they never meet and communicate mostly by

mailing list. A core team of developers have the task of integrating the produced code.

The reasons behind outsourcing in general are basically two:

1. low costs, with respect to western and central Europe and north America;
2. the availability of highly qualified professionals at low costs,.

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.

The countries that adopt outsourcing and offshore outsourcing to reduce costs (a reduction of two-third compared with same work performed in UK and USA) are mainly North America (Canada, USA, Mexico) and Western Europe. The main countries where the outsourcing is applied are Eire, Asia (India, China, North Corea, Malesia, Indonesia, Indochina, Filippine) and South Africa.

Cultural and geographical distances can reduce communication very much.

Are the advantages related to distributed development real or not? This scenario seems to be linked to factors which could change over time, for instance the economic situation of some foreign countries which could reduce the gap between countries that locate processes around the world and the other countries. Companies adopting offshore outsourcing have to protect themselves from probable geopolitical crises, calamities or other risks. You cannot neglect the fact that cost saving is due to phenomena such as exploiting children or violation of human rights.

The *scenario of geographical dispersion* consists of:

- the same venue but different office/floor/building;
- the same city but different venues;
- the same time zone, but different cities and countries;
- the same continent, but different time zone (with the difference of 3-4 hours);
- different continents (12-24 hour trip to meet face-to-face).

2.1 Application of the Agile Methodologies on the Distributed Development

In general we have already defined Agile methodologies as a software development approach which is able to adapt to frequent requirement changes and with short and repeated cycles of planning-design-coding-testing (iterations). Agile methodologies

have become very popular in the last years arousing the interest of a larger number of projects and companies that choose to follow its principles.

A recent survey conducted by Dr. Dobb's Journal shows 41 % of development projects have now adopted agile methodology, and agile techniques are being used on 65 % of such projects [1].

In the last years, researchers are trying more and more to 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 concurs to the process of software development involves 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 [46]. 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 communicating and cooperating: 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.

You should take in consideration some issues: just to give an example, since the developers teams can be numerous and de-localized, how can we overcome the problem that the manager could be just one from the start to the end of the process? The current opinion is that the importance of interpersonal communication, pair programming and on-site customer, is so relevant that Agile methodologies are not easily adaptable to software distributed development.

Here is a list, not exhaustive, of the major issues to deal with in an agile distributed development process:

- have a common vision of the project;
- share a common development model;
- share the same enthusiasm;
- have an updated status of the project;
- promote the communication with the customer;
- foster the communication among team members;
- use an informal communication among developers;

[1] Results from Scott Ambler's March 2006 "Agile Adoption Rate Survey" posted at www.agilemodeling.com/surveys

- keep in touch with the other team members (meeting);
- know when the others are available or not;
- developers having different time zone;
- developers and customers having different time zone;

Distributed development causes difficulties not only in the relationships between developers, but also between customers and developers.

Communication in the Distributed Development

Most agile methods emphasise concrete, low-tech face-to face collaboration methods like status reporting at daily stand-up meetings, a project status notice board, and "user stories" written on index cards. These low tech, face-to-face techniques are simple, fast and have very little management overhead. In a distributed environment, these face-to-face mechanisms must be replaced with some kind of server set aside for common file storage, web-based collaboration and a private instant messenging service. Typically, the web-based collaboration will involve discussion forums and some form of project planning and tracking tool.

Setting up a distributed project of this nature clearly requires more technical facilities. Rather than buying a notice board, pins and index cards, you now have to take time out regularly to administer a discussion forum and a messaging server. The communication is one of the most risky values in a distributed development process. Replacing significant amount of informal, face-to-face communication with web-based discussion software has the advantage that the customer can be given a much better picture of the state of the project. However, some extreme programmers have criticised this point because most of the communication between developers is about development issues rather than business ones. Extreme Programming sets out a "developers bill of rights" and a "customers bill of rights". Exposing too much development information to customers may tempt them to start micromanaging developers.

We are going to offer some solutions focused especially on communication problems:

Teleconference or daily chat: by means of use of these tools the *daily stand-up meeting* is adapted to dispersed environments; to use the webcam is for many people superfluous;

Auricular and microphone: developer must have free hands in order to use the computer while s/he takes part in the teleconference;

Tools for the synchronous cooperative work in the long run: tools such as Instant Messaging allow to communicate in synchronous mode. Microsoft NetMeeting [2], for example, provides the possibility to share one or more open

[2]URL: http://www.microsoft.com/windows/netmeeting/default.asp.

applications through the remote control of input and output. NetMeeting allows a direct audio/video communication if you have a webcam or a microphone. Using the video can be ineffective if there is network latency and when high-speed network connection is not available.

Daily Report via mails: they are reports on the work done. These reports are made by the team just for the team and/or for the customer, in order to have feedback on the work done;

Tools for the asynchronous cooperative work in the long run: in these cases a tool for managing the access to documents and the versioning (e.g. CVS, Subversion [3], and so on) is necessary, and that can more easily integrate with common development environments. Secondly, for reasons of communications and sharing, it is essential requirement to have one or more mailing lists and a web site editor (e.g. a Wiki). During code development the tools for the continuous integration are very useful, and they allow to control the incompatibilities among modules developed one at a time and then integrated;

Two/Three-Monthly Meeting: Dispersion allowing, (for example in Open Source projects it is not possible) it is advisable to meet every two-three months in order to take stock of the situation;

Agile Documentation: in order to obviate problems related to the dislocation, it is likely that a strong exchange of documents among the team members happens. In this case it is a good rule that documentation must answer to the agility fundamental requirement applied to the process;

Number of Developers: it is recommendable a team composed by 10-15 people;

Project Schedules: in order to best adopt the agile methodologies, the project must last no less than three months.

Security

Many customers of agile projects are rightly concerned about confidential data and intellectual property. Using broadband routers that have a VPN feature, it allows to create a secure, private network shared between the dispersed development team and their customers.

Code Repositories

A source code control system like CVS, Subversion [4] or Perforce, is a requirement for all professional software development projects. Installing CVS on a dedicated server may suffice for a small team, even for a small, dispersed team. Take into consideration the volume of network traffic and scalability of your source control

[3] http://www.subversion.tigris.org
[4] http://www.subversion.tigris.org

system in an environment with large numbers of users, or systems that require synchronization across continents.

In one project it took four hours to synchronize a developer workspace in Italy with a CVS server in England. Given that a typical extreme programmer would expect to release code every fifteen to thirty minutes, requiring synchronization, catch-up, build and a commit, a four-hour catch-up is clearly unacceptable. This problem is thought to lie in either in network connections or in the VPN encryption infrastructure. Some source control systems like Perforce provide proxy servers that can reduce the volume of traffic required between remote sites and the server location. This issue clearly can't be ignored for big distributed development efforts.

2.2 Related Works

In literature there are a few researches that have already studied the way to introduce the agile methodologies in projects of distributed software development.

In the case of the *Distributed eXtreme Programming* (DXP) ([33] and [32]) 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 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, 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 *Distributed Pair Programming* (DPP) ([4], [27] and [51]) 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. It'is important to note that the tools helping to work in synchrony way in no co-located situation are not born for the distributed development.

Chapter 3

Open Source: an emblematic example of Distributed Development

In the world of software development there are today different examples and scenarios of distributed development.

Open Source scenario is one of the most emblematic and widespread example of distributed development. Typically the programmers are constantly dislocated, they never meet and communicate mostly by mailing list, leaving to a core team of developers the task of integrating the produced code.

In this thesis, focus is directed at *FLOSS* (free/libre and open source software development). FLOSS [1] can be briefly defined as software whose licenses give users the freedom to run the program for any purpose, to study and modify the program, and to redistribute copies of either the original or modified program (without having to pay royalties to previous developers).

The Free Software Definition is maintained by FreeSoftware Foundation (FSF) and mandates four fundamental freedoms, including freedom to run, study, redistribute, and improve software. The Open Source Definition, maintained by the Open Source Initiative (OSI), puts forth a similar set of requirements, with which a piece of software must conform to be considered Open Source. Legally, Free Software and Open-Source take quite different attitudes to sharing source code and what obligations those who share legally require.

FLOSS development is based on a relatively simple idea: the core of the system is developed locally by a single programmer or a team of programmers. A prototype system is released on the Internet, which other programmers can freely read, modify

[1] The Free Software Definition is reported in http://www.gnu.org/philosophy/free-sw.html

and redistribute the systems source code. The evolution of the system happens in an extremely rapid way; much faster than the typical rate of a closed project.

Software conforming to the Open Source Definition [2] must comply with 10 criteria (Free redistribution, Source code, Derived works, Integrity of the author's source code, No discrimination against persons or groups, No discrimination against fields of endeavor, Distribution of license, License must not be specific to a product, License must not restrict other software, License must be technology-neutral).

In literature, Free (as in freedom) Software (FS) and Open Source Software (OSS) are often treated as the same thing. However, there are differences between them with regards to the licenses assigned to the respective software. Free software generally appears licensed with the GNU General Public License (GPL), while open source software (OSS) may use either the GPL or some other license that allows for the integration of software that may not be free software.

Basically, there are several types of licensing models available, the most notable being the GNU GPL (General Public License). There is, however, a great number of other FLOSS licenses, such as the Lesser General Public License (LGPL), IBM Public License, BSD license, Apache license and many others.

The hallmark of free software and most OSS is that the source code is available for remote access, open to study and modification, and available for redistribution to other with few constraints, except the right to insure these freedoms. OSS sometimes adds or removes similar freedoms or copyright privileges depending on which OSS copyright and end-user license agreement is associated with a particular OSS code base. More simply, free software is always available as OSS, but OSS is not always free software.

3.1 Scaling Agile Methodologies in Open Source projects

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 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 necessary to work like a team [46].

The Open Source world has much more in common with the Agile Methodologies. The same nature and the bazaar [47] 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 and fast feedback from the contributors

[2] The text of *The Open Source Definition* can be found in http://www.opensource.org/docs/definition.php.

(active and always updated mailing lists). Like the 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 [33] [32] moreover, that also in the OS projects there are some characteristics like the use of coding standards, 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. 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 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, which 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" ([50]), for example, and on the other hand the code sharing, the cooperation between developers with a rigorous peer-review and a parallel debugging [47] are the main characteristics, certainly agile, of an OS project.

3.1.1 Jakarta case study: an analysis

We analyzed the OS *Jakarta* community in order to analyze differences and similiraties with the agile process. OS Jakarta is a good case study either for the quality of the results or the development process, the code maintenance and the documentation are rigorous.

Jakarta project is developed by the Apache Software Foundation, then the Apache Software License licence is adopted.

Roles and responsibilities, procedures, communication and project management are well disciplinated.

Users use the products of the Project in order to report bugs or make future requests. They don't write or change source code or documentations and when they do it then they become *Contributors*.

A *Contributor* writes and changes source code or documentation patches in order to contribute positively to the project quality. The Contributor who gives frequent

and valuable contributions to a subproject of the Project can be promoted to a *Committer*. Changes in the repository have effect only behind a careful examination and then acceptation by a *Committer*.

The *Committer* can change the source code in the repository and give his vote about the decisions related to the project. A Contributor becomes a Committer only after that another Committer nominates him. Once a Contributor is nominated, all of the the Committers will vote: with at least three positive votes and no negative votes, the Contributor becomes Committer and s/he has write access to the source code repository for that subproject.

Communication

Communication is a fundamental value in OS projects. Since people are distributed, the main tool for members to easily communicate with each other, is the mailing list. Jakarta provides many mailing lists divided into categories:

Announcement Lists: mailing list where the traffic is very low because they are designed to communicate important information, such as final releases of a subproject's code;

User Lists: messages, related to the installation, the configuration and the use of the software, exchanged among users of a product;

Developer Lists: they are for developers of the project. There are suggestions and comments for code changes. Moreover action items are raised and voted on;

Commit Lists: here is where are all cvs code commit messages are sent. Committers are required to subscribe to this list so that they can be aware of changes to the repository.

Coding Convention

Source code has to compliant to the "Coding Convention' defined by the Sun for Java6 languages.

Alternatively, a subproject can define specific coding convention.

Planning

All Contributors are encouraged to participate in decisions, but only the Committers take final decisions.

In practice, what happens is that this voting is made throughout a message in the mailing list, and the votes of the Contributor are used to orient the decision but the only binding votes are those cast by a Committer.

Votes can be made in one of three flavors:

- +1: "yes", "agree" o "the action should be performed";

- +/-0: “abstain”, “no opinion”;
- -1: “no”.

An action requiring consensus approval must receive al least 3 binding plus 1 votes and no binding vetos. One negative vote implies the veto: in this case the veto has to be correlated by a justification. A veto cannot be ignored: the voting proponent has to convince who gives the veto to change his/her position.

All decisions revolve around *“Action Items”*. Action items consist of the following:

long term plans: they are announcements that group members are working on particular issues related to the Project;

short term plans: they are announcements that a volunteer is working on a particular set of documentation or code files with the implication that other volunteers should avoid them or try to coordinate their changes;

release plan: it is announced a release that has to be approved by an absolute majority. If it is approved, the responsible seals the repository and releases;

release testing: after a new release is built, it must be tested before released to the public. For the publication is required majority approval;

showstoppers: A issue in the releases is a Showstopper when it must be resolved before the next public release;

product changes: they are changes to the products of the Project, including code or documentation in the repository.

Source Repository

Jakarta project provides a repository for the shared information. In other words the Project’s codebase is maintened in a CVS server or Subversion [3] and only the Committer has write access to these repositories. All source code committed to the Project’s repositories must be covered by the Apache Licence or contain a copyright and licence that allows redistribution under the same conditions as the Apache Licence. There are many jar files that don’have this licence and so they are not admitted in the repository. Each of the Project’s active source code repositories contain a file names Status which is used to keep track of the agenda and plans for work within that repository. It is recommended that the active status files are automatically posted to the developer mailing lists three times per week.

[3] http://www.subversion.tigris.org

Contribution Management

As Contributor cannot have write access in the repository, their contribution must follow a specific procedure based on the patches. In practice, a patch is a file generated by an automatic tool that describes the differences between two different source code versions. The patch should be created by using the diff -u command from the original software file to the modified software file. The following procedure have to be followed by the Contributor:

1. check out of the source code by the CVS repository;
2. changes of the source code for incorporate necessary changes and introduce related documentation;
3. check of the code throughout compilation;
4. check of the functionalities throughout the execution of the unit and regression tests normally expected.

This patch is created by the following commands:

```
cd cvs # location where you store cvs modules

cd site # module your patch applies to

cvs diff -u > site.patch
```

The CVS client examines all changes and generates the "site.patch" file. This file has to be sent to the project mailing list, and after that this has been analyzed, a Committer applies the patch at the repository or s/he communicates to the Contributor in order to make all suitable changes before the application.

Project Management

The Project management Committe (PMC) was formed in September 1999, and in the jakarta site there is the list of current members. The PMC takes the strategic decisions for the project, resolves possible conflicts and removes obstacles in the procedures or any bottlenecks.

3.2 Open source and XP: a comparison between them

We can compare the XP practices with a OS process like as Jakarta process described above.

3.2.1 Values

Extreme Programming is based on core values: *communication*, *semplicity*, *feedback*, *courage* and *respect*. Can these values be share on also Open Source projects? Here we present a short analysis.

All methodologies have built-in communication processes but in XP it is a core value: more important. In XP the constant *communication* among all team members is fundamental and are used practices that require communication to work as pair programming, customer-on-site, or work spaces sharing. In Open Source projects this practices are not adaptable but the communication is the strength of all projects. In fact the communication is included in the document that defines the guidelines of the Jakarta Project [4] and it is supported by a variety of mailing lists. Moreover in the Jakarta Project is recommended, in order to reduce the bandwidth demands on everyone, mail should not contain attachments and place only interesting material such as patches. In XP project the *simplicity* is solved with the design. Being extreme, XP keeps the design as simple as possible, that is *"Do the simplest thing that could possibly work"*, and *"design for today"*. In the OS world an approach that considers the simplicity like an intrinsic value does not exist. However, since the community exists beyond the participation of individual volunteers, the OS project design must be as simple as possible. The OS community could learning it by the XP process.

The value of *feedback* is vital to the success of any software project.

The *courage* is a boldness to act. This value gives the confidence to work quickly and redevelop if required. Not all of the OS projects can be based on courage. It can be necessary more time, than not in XP, before the community needs make changes.

The previous four values imply a fifth: *respect*. If members of a team don't care about each other and their work, no methodology can work. In the Jakarta Projects respect is assured because roles and responsibilities are defined and based on merit. Everybody can help no matter what their role.

3.2.2 Practices

Here we analyze an OS process applied to Jakarta related to the XP practices.

The Planning Game

XP is successful because it stresses customer satisfaction. The Planning Game involves customers and programmers working through user stories, estimating and prioritizing. OS don't has a "customer". As noted in Jakarta description, in general OS projects have users who can become developers. In this sense we can think that customers and developers work together. In XP the functionalities are defined by the customers through user stories, in Jakarta the functionalities are demanded by the users through mailing list or Feature Tracking systems. In XP the developers

[4] Available to http://jakarta.apache.org

estimate how long each story will take to build and how much is it, warn the customer of technical risks, in Jakarta there is not this practice and could be used. In XP customers decide about the scope for the next releases, in Jakarta every release adds value to the product but it is difficult to make deadlines as in XP. In XP customers decide which user story will be provided in the next release. In Jakarta the mechanism is opposed to it: voters pass or reject proposals for new functionalities. If these have been accepted then they are developed. Committers or PMC can propose a release related to what has been developed. Whatever a proposal for a new release requires the voto. Every project has a status file which is used to keep track of the agenda and plans for work within that repository.

Short Release

In general a lot of OS projects share this practice.

Simple Design

A lot of OS projects are successful just because they adopt an effective design, very simple, that aimed to obtain results. In this comparison it must be pointed out that there are differences between the XP products and those of OS process. Frequently OS projects were born for creating libraries, utilities, programming support tools and less for creating applications for final users. Maybe the typology of the open source users (they are developers) imply it. On the contrary XP projects seem adapted to business applications that must meet the customer needs.

Testing

In XP before updating and adding code, it is necessary to write tests in order to verify the code (*"first test then code"*). Although a lot of OS projects require the tests, in particular before a release, this practice could be expressed in formal way and executed to extremes.

Refactoring

In XP process Refactoring is a daily practice and unit tests, for the classes subjected to the refactoring, and code integration in the repository, are indispensabile to it. In the Jakarta project this is not simple because the access to the source repository is regulated by the roles, and in some cases the code changes must be approved by the Committer.

Pair Programming

Related to the Pair Programming, we need to know what are the goals in order to apply this practice. Pair programming is where two developers sit at the same workstation and share development tasks and the work of programming includes

coding, thinking, designing, testing, listening and talking. In particular the benefits derived from pair programming are that all code is reviewed all the time (*"code review"*, at least two developers are familiar with that part of the system (knowledge sharing), and there is less chance that tests will be missed. In a OS project developers are distributed and this practice, in its strict form, is not applicable, but those same goals are achieved in other way. In Jakarta project the code is frequently reviewed because the code is public and all of the contributors can suggest improvements and detect potential bugs or report bad smells in code. Knowledge transfer is less immediate, because only the mailing lists are used to communicate.

Collective Code Ownership

In XP, Collective Code Ownership enables anyone to improve, or change, any part of the code at anytime. This practice is not applicable in Jakarta because there is a Committer team that analyzes the proposed changes and perhaps approves them. In this manner the Committers have the control on the project. This difference is related to same nature of a XP team and an OS team. XP team is made up of 6-10 developers all skilled in the project, whereas OS team is made up of tens or hundreds of potential Contributors that haven't the same knowledge of the project. Moreover the Contributors haven't a mutual trust and a sense of team identity, unlike an XP team. So, if any OS project can be modifiable, anyone could cause damages to the project.

Continuous Integration

Continuous Integration works in practice like this: developers work on a module, class, or component until complete. Then they add to the integration machine, run tests, fix problems and move to the next tasks. In less rigorous engineering environments, developers work for weeks or months and only integrate at delivery time: results are disastrous. But in Jakarta there is a daily continuous integration, that is they have an automatic procedure for providing daily (or nightly) builds. Doubtful changes, new features, and large scale overhauls need to be discussed before committing them into the repository. Any change that affects the semantics of an existing API function, the size of the program, configuration data formats, or other major areas must receive consensus approval before being committed. Related changes should be committed as a group, or very closely together. Half-complete projects should never be committed to the main branch of a development repository. All code changes must be successfully compiled on the developer's platform before being committed. Also, any unit tests should also pass. The current source code tree for a subproject should be capable of complete compilation at all times

No overtime

The No overtime practice has a well defined goal in XP: to avoid overtime work, bad moods, tiredness, and little concentration. In the OS project this in not a

problem because the developers are volunteers and they like to contribute it.

On-Site Customer

XP extremes customer involvement: the customer has to be on the project full-time for the duration and be located on-site with the team. In OS world, users and developers, take part to the same project, and are team members with the same goals. In Jakarta project are defined recognized roles and there is a system of meritocracy, which is the core of the Jakarta Project. The PMC is responsible for the strategic direction and success of the Jakarta Project. This governing body is expected to ensure the project's welfare and guide its overall direction, therefore it can be considered like as on-site customer.

Coding Standard

The overall purpose of coding standards is to produce software that has a consistent style, independent of the author. This results in software that is easier to understand and maintain. Coding standards have long been recognized as a best practice when developing software. In OS project the coding standard is well followed, in order to have homogeneity in the code also because the source code is changed by too many developers. These standards are documented and formalized as part of the guidelines of The Jakarta Project. It includes definitions of the various categories of membership, who is able to vote, how conflicts are resolved by voting, and the procedures to follow to propose and make changes to the codebase of the Project.

3.3 Introducing Test Driven Development on a FLOSS Project: a Simulation Experiment

The section presents our study on the effects of the adoption of agile practices on open source development. XP has been devised for small, co-located teams and it seems that XP is not suited to FLOSS development, which involves tens or even hundreds of distributed developers. There are several FLOSS projects in which some XP practices have been applied with both large and distributed teams [25], [46].

In particular, we evaluated the application of TDD (Test Driven Development, also known as Test-First programming) agile practice to FLOSS development because it is easer to apply in a distributed environment than most other agile practices. Including automated tests in a FLOSS project allows volunteers to trust each other, because tests demonstrate that their code works, and lower the fears to change code written by other developers. We investigated whether or not the defect density injected, the total number of lines of code (LOCs) and the number of developers are

affected by adopting TDD practice.

In order to reach this goal we used the *simulation modeling approach*. We developed a simulation model of open source software development process.

Software Process Simulation (SPS) is becoming increasingly popular in the software engineering community, both among academics and practitioners [31]. New and innovative software engineering techniques are being developed constantly, so a better understanding of them is useful to evaluate their effectiveness and to predict possible problems. Simulation can provide information about these issues avoiding real world experimentation, which is often too costly in terms of time and money. This field of SPS has attracted growing interest over the last twenty years, but only in recent years it is beginning to be used to address several issues concerning the strategic management of software development and process improvement support. It can also help project managers and process engineers to plan changes in the development process. The development of a simulation model is an inexpensive way to collect information when costs, risks and complexity of the real system are very high.

In order to create a connection between real world and simulation results, it is mandatory to combine empirical findings and knowledge from real processes. In general, empirical data are used to calibrate the model, and the results of the simulation process are used for planning, designing and analyzing real experiments.

A model is always an approximation and a simplification of the real system, and the model developer has to perform an accurate analysis to identify and model the aspects of the software process that are relevant to address the issues and questions s/he is investigating, neglecting other aspects.

So, we analyzed the effects of the adoption of agile practices in FLOSS development using a SPS model. In addition, since FLOSS dynamics are not yet fully understood, simulation modeling may be a valid instrument for understanding the dynamics of this development process.

3.3.1 Test-First Programming

Test-First Programming is also known as Test Driven Development [7]. From here on, we referred to this practice as TDD.

Using TDD, the tests are written before the code itself. Through a rapid cycle of adding new tests, making them pass, and then refactoring to clean the code, the software design evolves through the tests. In XP, developers should accustom themselves to the rhythm “test - code - refactor”. In fact, with TDD robust code is delivered along with a comprehensive suite of tests, allowing developers to make future changes with confidence.

All tests have to be automated [21]. In fact, if tests had to be run manually, their execution and the examination of the results would be error-prone and time consuming. Automated testing means that the tests themselves are coded. The tests can then be run over and over again with very little effort, at any time, and by anyone.

One of the biggest benefits of TDD is the decrease of defect density throughout the whole software life cycle, because the automated test suite enables the developers to discover and fix bugs earlier. Since TDD drives the entire development process, it allows to improve the design quality. It helps to understand better requirements and to develop systems more capable of accepting changes. Finally, it provides a documentation of the desired behavior.

Many studies have been conducted in order to investigate the benefits of TDD in terms of code quality and team productivity. George and Williams [26] carried out an experiment with 24 professional pair programmers. The developers were divided into two groups each one made up of six pairs: the first group developed code using TDD, the second adopting a conventional design-develop-test-debug approach typical of a waterfall process.

This case study found that the TDD developers produced higher quality code. This code passed 18% more functional black box test cases than the code developed by the other group. On the other hand, the TDD developers took 16% more time to write the application than the other group. However, the variance of team performance was large, so these results give only a rough idea of TDD validity. The extra time taken by TDD could be due to the time needed to write and update the test cases.

Muller and Hagner [43] conducted an experiment as part of an XP course held with computer science graduate students. The case study involved 19 participants, divided into two groups: one developed using TDD, the other followed a traditional development process. The two groups solved the same task. The research measured the effectiveness of TDD in terms of development time, code quality and requirements understanding. The case study found that the development using TDD increased the understandability of the program, measured as proper reuse of existing methods. On the other hand, it provided neither an increase in quality nor in development time.

Lui and Chan performed another important study [36]. They assisted two teams from two different software companies in China to implement TDD in their software development process. The performance of the TDD teams were compared with third project. In particular they compared how long developers took to fix defects reported by users during acceptance testing and production operations. The research showed that TDD teams produced far fewer defects and fixed them much faster than non-TDD teams.

Erdogmus et al. in [24] conducted a controlled experiment with undergraduate students divided into two groups. The experiment group (Test-First) developed using TDD, while the control group (Test-Last) wrote tests after coding. They reported that Test-First programmers wrote more tests per unit of programming effort than Test-Last group. They observed that writing more tests led to higher level of productivity and improved the minimum quality achievable.

3.3.2 Model Description

Our FLOSS development model was inspired by a similar work carried out by Antoniades et al. [3], but it differs in several ways. To calibrate the model parameters we used real FLOSS project data obtained from a case study on the Apache HTTP server project [41], [42].

Based on what normally happens, we made the assumption that the initial kernel of the system is developed by a small team of programmers (*Core contributors*). The system kernel is then released on the Internet where other programmers (*Normal contributors*) can freely read, redistribute, and modify the system's source code. Our model deals with the evolution of the FLOSS once its initial core has been released.

A project is composed of a set of modules, each of which is made up of source code files. Each file is characterized by a number of lines of code (LOC), and by a number of defects. These files are developed by programmers who can contribute to the project in four kinds of activities:

Program writing task: at the end of this task, a new source file is committed to a specific module of the project (**Task S**).

Functional improving task: this task consists of adding new functionalities to a specific file, whose lines of code (LOCs) correspondingly increase (**Task F**).

Testing task: the activity of testing a specific source file and report its defects (**Task T**).

Debugging task: correcting a defect that was previously reported for a specific file (**Task B**).

Model dynamic

Each day t of the project, a number of tasks $\Delta T(t)$ is started by the community of developers. We made the assumption that each developer handles one task per time. For each started task the model calculates: the exact time period (in days) that it will take for the results of the task to be submitted to the development release, and the deliverables of each task (added LOCs, injected/detected/corrected defects, etc).

The number of tasks starting at a certain day t is obtained from equation 3.1:

$$\Delta T(t) = N'_{core}(t) + N^*_{norm}(t) \tag{3.1}$$

where

$\mathbf{N'_{core}(t)}$ is the number of core contributors available at day t. We made the assumption that, after having completed a task, each core developer is immediately available to start another one.

$\mathbf{N^*_{norm}(t)}$ is a subset of the available normal contributors $N'_{norm}(t)$ at day t. In particular, each available developer can decide to subscribe to a task with a certain probability (authors' assumption: $p = 0,7$).

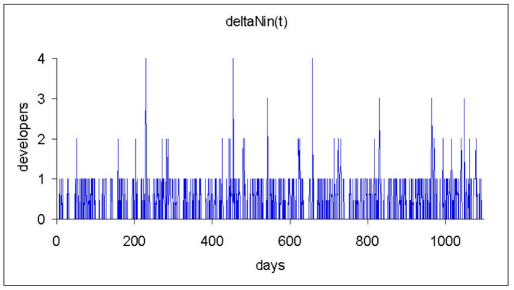

Figure 3.1: New developers entering the project.

In FLOSS projects, the number of contributors is not fixed in time. As any qualified programmer can freely contribute to the project at any time as often as s/he desires, the number of contributors varies depending on the interest that the specific FLOSS project arouses. Our model contains an explicit mechanism for determining the flow of new contributors as a function of time and relates this mechanism to specific project-dependent factors that affect the overall 'interest' in the project. On each day t, a number $\Delta N_{in}(t)$ of new programmers decide to participate to the project. At the same time, a number $\Delta N_{out}(t)$ of programmers leave the project. In particular, each developer can decide to leave the project with a probability that increases with the time s/he spent working on it: $p = k_{out} \cdot t_{work}$.

$\Delta N_{in}(t)$ is taken from a lognormal distribution with mean $\overline{\Delta N_{in}(t)}$ proportional to the overall attractivity of the project $Q(t)$, which is related to the frequency of commits $c(t)$ and to the percentage of increase in LOCs $\Delta L(t)$ within a time window of 60 days.

$$\overline{\Delta N_{in}(t)} \propto Q(t) \tag{3.2}$$

$$Q(t) = 1 - e^{-q_1 \cdot w_q} \tag{3.3}$$

$$q_1 = w_L \cdot \Delta L(t) + (w_c * c(t)). \tag{3.4}$$

where w_q, w_L and w_c are specific model coefficients which were used to calibrate the model.

Figure 3.1 shows a typical trend of $\Delta N_{in}(t)$ obtained from a simulation run of the Apache project. Analyzing other FLOSS projects, we found similar trends.

The number of normal developers contributing to the project at a specific day t is given by the equation 3.5.

$$N_{norm}(t) = \sum_{i=0}^{t} (\Delta N_{in}(i) - \Delta N_{out}(i)) \tag{3.5}$$

Once the number of tasks $\Delta T(t)$ initiated on each day t has been determined, the probability P_{ij}^{K} that a developer will choose a specific task K ($K = S, F, B, T$) in module i and file j is defined in equation 3.6. This probability depends on the

interest of developers for a specific module, file and type of task.

$$P_{ij}^{K} = \frac{I_{ij}^{K}}{\sum_{ij}(I_{i}^{S} + I_{ij}^{F} + I_{ij}^{B} + I_{ij}^{T})} \tag{3.6}$$

Equations 3.7, 3.8, 3.9, and 3.10 model this interest.

$$I_{i}^{S} = \frac{A^{S}}{M} \cdot e^{-k \cdot NrFile_{i}} \tag{3.7}$$

$$I_{ij}^{F} = \frac{A^{F}}{M} \cdot \frac{e^{k \cdot (maxLocs - loc_{ij})}}{NrFile_{i}} \tag{3.8}$$

$$I_{ij}^{B} = \frac{A^{B}}{M} \cdot \frac{(NrBug_{ij})}{\sum_{w=1}^{NrFile_{i}}(NrBug_{iw})} \tag{3.9}$$

$$I_{ij}^{T} = \frac{A^{T}}{M} \cdot \frac{1}{(NrTest_{ij} + 1)\sum_{w=1}^{NrFile_{i}} \frac{1}{NrTest_{iw}+1}} \tag{3.10}$$

Where M is the number of modules and $NrFile_i$ denotes the number of files of the module i. A^S, A^F, A^B, A^T identify the raw developers' interests to work on a specific kind of task (S, F, T or B). The $maxLocs$ parameter is an upper limit for a file size in terms of LOCs. loc_{ij} is the number of LOCs of the file j in module i. $NrBug_{ij}$ is the number of defects reported during the test of the file j in module i. Finally, $NrTest_{ij}$ is the number of times the file j in module i has been tested.

We assumed that the interest to initiate an S task I_i^S is reduced exponentially with the number of files contained in module i. The interest to start an F task on a specific file j contained in the module i (I_{ij}^F) decreases exponentially as the LOCs of the file i increase. I_{ij}^B is proportional to the ratio of the reported bugs for a file j to the reported bugs for the whole module i. Finally, I_{ij}^T decreases with the number of times the file j has been tested, and it increases with the number of times the module i has been tested.

Finally, the deliverable quantities (LOC, defects, etc) and the delivery time of each task are drawn from lognormal probability distributions. In table 3.1 we show these project-specific quantities that must be provided in input to the simulation model. The time to write one LOC for a normal contributor is so high because in a

open source project s/he doesn't work full time. Approximately 80% or more lines of code are written by core developers.

Table 3.1: Mean and Standard Deviation of lognormal distribution for the deliverable quantities and the delivery time of each task. The first three values pertain normal developers.

Constant	Description	Value
$\overline{T}_{LOC}, \sigma_{T_{LOC}}$	Mean and standard deviation for the time (in days) to write one loc for a file in a module	(1 , 0.1)
$\overline{T}_{deb}, \sigma_{T_{deb}}$	Mean and standard deviation for the time (in days) to correct a single defect in 1000 LOC	(40 , 20)
$\overline{T}_{test}, \sigma_{T_{test}}$	Mean and standard deviation for the time (in days) to test and report detected defects for a file in a module	(2 , 3)
$\overline{N}^S_{loc}, \sigma_{N^S_{loc}}$	Mean and standard deviation for the initial number of LOC in a new file	(400 , 200)
$\overline{N}^F_{loc}, \sigma_{N^F_{loc}}$	Mean and standard deviation for the number of LOC added in an existing file with a functional improvement task	(33 , 40)
$\overline{D}_{inj}, \sigma_{D_{inj}}$	Mean and standard deviation of the number of defects injected per loc for a file in a module	(1.5 , 3)
$\overline{D}_{rep}, \sigma_{D_{rep}}$	Mean and standard deviation of the fraction of actual defects that are reported for a file in a module with a task T	(0.5 , 0.25)

Calibration and Validation

Each real FLOSS project could be characterized by specific parameters values (depending on the interest of the particular project, the programming language used, the relative complexity of the system to build, etc), so it is not sensible to calibrate the parameters on a few projects, hoping that this "*one size fits all*" approach will be valid for all, or most, other FLOSS projects. Therefore, we decided to start our research concentrating mainly on a well known and thoroughly studied FLOSS project: the Apache project. We used the data available for this case study to calibrate the parameters of our model. When no data were available from the Apache project, we had to make our own sensible assumptions for the parameter values. Eventually, we produced simulation results that compare very well with the respective results of the Apache case study, as shown in table 3.2.

Data coming from different FLOSS projects would be used to validate the model of the process in order to improve future applications of the model. In order to do this, one has to simulate other FLOSS projects, or different time intervals of the same FLOSS project, comparing simulated data with actual data. One of the

Table 3.2: Comparison between simulation results averaged over 100 runs and the Apache case study. Standard deviations are shown in brackets.

Output variable	Simulation	Apache
Total number of LOCs	230889 (6482)	220000
Number of contributors	352 (30)	388
Defect density per KLOC	2.51 (0.26)	2.64
Total activity in task type B	689 (62)	695
Total Activity in task type T	3912 (74)	3975
Total Activity in task type S+F	6217 (200)	6092
Total Activity in all task types	10819 (278)	10762

major problems in the validation of a simulation model is to find project data with a sufficient level of detail. These data are difficult to obtain for several reasons. In fact, to validate our model we need to have data related to:

- the source code committed after each task activity;
- the number of problems reported for each source file
- the number of developers contributing to the project
- the number of tasks activity differentiated for task type

Many of these data can be obtained from an analysis of the source code repository and of its version control system (CVS, Subversion [5], ...), and from bug-tracking systems (like Bugzilla). However, from these data sources we can extract only partial information. For example, the developers who have a write access to the Apache repository are only a subset of the whole community of developers.

Adding TDD to the Model

In order to introduce the Test-First Development practice into the FLOSS simulation model, we make the following assumptions:

1. The average time needed to write a line of production code increases.
2. The number of defects injected during coding activities (S, F) decreases.

[5] http://www.subversion.tigris.org

3. The debugging time to fix a single defect decreases.

The first assumption is due to the fact that TDD prescribes that the testing code is written before the production code itself. Therefore, since our model only considers the production code (no test code), the time needed to write a line of code increases consequently. The second and third assumptions are based on the fact that doing tests before coding increases the probability of finding defects before committing the code to the configuration management system, and quite likely decreases the time needed to correct them. As stated by Kent Beck in [7]:

> *The gain in productivity comes from reduction in the time spent debugging – you no longer spend an hour looking for a bug, you find it in minutes.*

Therefore, it is plausible that the time spent on debugging tasks decreases. This is in agreement with a case study conducted by Lui et al. [36] showing that non-TDD Teams were able to fix only 73% of their defects against 97% fixed at the same time by TDD-Teams.

The exact amount of these changes is difficult to quantify because of the lack of experiments and case studies on these topics. Only recently, some studies have been performed to verify whether or not TDD increases productivity, but the most of them where carried out on students. Among others, we can cite a study conducted by George et al. [26] on three software companies. The experimental results showed that TDD developers took more time (16%) than non-TDD developers. So we decided to increase the time needed to write production code (T_{LOC}) with TDD by a similar quantity. Concerning the defects injected during coding activities, it has been qualitatively found that TDD decreases their number [36], but we have no quantitative data to this regard. The only data regarding product quality has been reported by George et al. [26]. They found that with TDD the number of functional black box test cases passed increases by 18%. So, we set the injected defects parameter D_{inj} to 18% less than the parameter value without TDD. Also we

supposed that the time needed to fix a defect T_{deb} decreases of 25% in accordance with Lui et al. [36].

The introduction of the TDD inside the FLOSS model has been done adjusting some model parameters to account for the specificity of TDD, according to the discussion outlined above (see table 3.3).

Table 3.3: Parameters used to differentiate the two models. Means and standard deviations are reported. T_{LOC} is the time (in days) to write a LOC, D_{inj} are the defects injected per KLOC, T_{deb} is the time (in days) to fix a defect.

Parameter	Team	nonTDD	TDD
T_{LOC}	core	0.02 (0.02)	0.023 (0.02)
	normal	1 (0.1)	1.16 (0.2)
D_{inj}	core	1.5 (3)	1.23 (2)
	normal	1.5 (3)	1.23 (2)
T_{deb}	core	40 (15)	30 (15)
	normal	40 (20)	30 (20)

3.3.3 Experimental Results

In table 3.4 we compared the simulation results obtained with the model described (nonTDD-simulator) and those obtained with the TDD-simulator. Note that the number of LOCs produced with the TDD-simulator is lower (-9%) than that obtained from the nonTDD-simulator. The total number of contributors in the TDD simulation is the same, while the quality of the code produced looks much better (-40% of defects/KLOC) in the case of TDD-simulator.

Table 3.4: Comparison of simulation results averaged over 100 runs obtained with the base model described (noTDD simulator) and those obtained with the TDD-simulator. Standard deviations are shown in brackets.

Output variable	nonTDD	TDD
Total LOCs	230889 (6482)	228832 (6093)
Contributors	352 (30)	351(30)
Defect density [defects/KLOC]	2.51 (0.26)	1.51 (0.30)

We can note that the simulations predict that the source code produced using TDD has an higher quality than that produced with a more traditional FLOSS development process.

It is interesting to observe the time evolution of the defects (reported but not yet fixed) of the project (figure 3.2). In particular, in the first part of the project, we noticed a similar increasing trend in both approaches. In the second part of the project, the non-TDD number of defects is basically constant, while the TDD number of defects decreases.

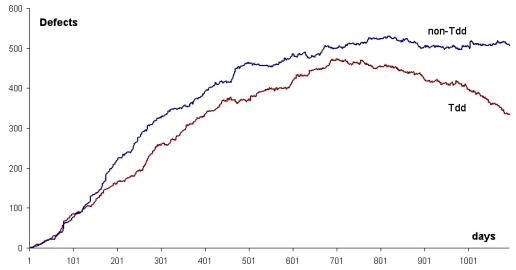

Figure 3.2: Cumulated reported defects versus time in the first 3 years of the simulated Apache project.

An alternative view of this phenomenon can be observed relating the number of defects with the growth in size of the project in terms of LOCs (figure 3.3). Again, in the first phase of the project both approaches have similar trends. In both cases, after a time interval where the trend is quite flat, the defect density decreases as the time advances. Anyway, in the TDD project the number of defects is always lower than the other one, and the decreasing slope is more pronounced.

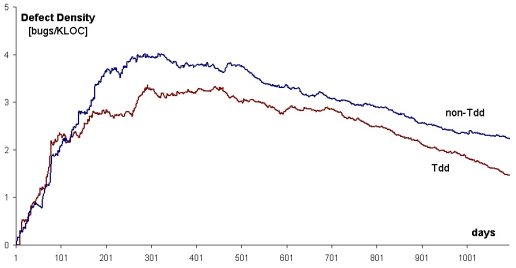

Figure 3.3: Density of the residual reported defects versus time in the first 3 years of the simulated Apache project.

3.3.4 Sensitivity Analysis

In order to better understand the mechanism influencing the decrease of defect density on the TDD-simulator, we analyzed in depth its variations against some key TDD input model parameters. In particular we conjunctly modified the "time required to write a LOC" (T_{LOC}) and the "number of defects injected per KLOC" (D_{inj}). Since there are no empirical data on the effects on the percentage of injected bugs, produced by the addition of TDD, we adopted the conservative assumption that an increase of factor k in time spent in coding – due to TDD – corresponds to a decrease of the same factor k in the average number of injected bugs. So, we multiplied T_{LOC} and divided D_{inj} by the same coefficient k. Then we performed a series of simulations varying k from 1 to 2 with step 0.04 (see table 3.5).

Table 3.5: TDD parameter values used in the sensitivity analysis.

Parameter	k=1		k=2	
	core	*norm*	*core*	*norm*
T_{LOC} [days/LOC]	0.02	1	0.04	2
D_{inj} [defects/KLOC]	1.5	1.5	0.75	0.75
T_{deb} [days/defect]	30	30	30	30

In figure 3.4, we report the results of the residual defect density varying T_{LOC} and D_{inj} with k, which can be approximated with a quadratic curve (dashed line). As you can see, the average number of defects in the final product decreases steeply as k increases, yielding a decrease in total defect density much lower than k. For instance, setting k to 1.4 - which amounts to a 40% increase in coding effort - yields a decrease in the number of defects of more than one half. For higher values of k this phenomenon fades, and the average number of defects in the final product stabilizes at a value of about 0.4 [defects/KLOC].

Note that, for all cases, the overall developing effort is approximately the same, the final number of LOCs and overall number of developers involved in the project is almost unchanged. This is because the greater effort spent in coding is offset by the lesser effort spent in testing and debugging. This result indicates that using TDD in a project greatly helps to increase the project's overall quality, but adding too

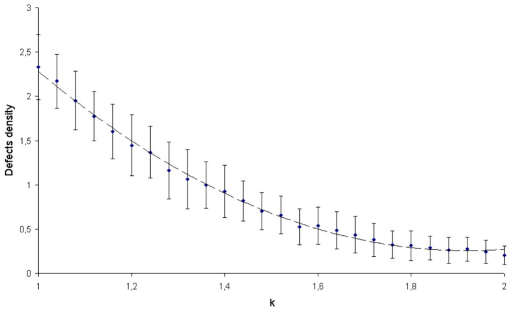

Figure 3.4: Variation of the defects density with K coefficient.

many tests does not yield further improvements, above a given threshold. Of course these are speculative, although sensible results of a model simulation, and should be validated by empirical findings.

3.3.5 Validation of Hypotheses

In order to validate our hypotheses we performed a two-sample t-test[6] on the two groups of data (nonTDD and TDD). The test assesses if the means of the two distributions are statistically different or not. Since empirical evidence in this research area is not sufficiently strengthened we have used two-tails test.

Hypothesis A: The defect density injected using the TDD practice is different from that obtained without TDD.

Hypothesis B: The total number of LOCs written using TDD is different from the LOCs written without TDD.

Hypothesis C: The number of developers does not change using the TDD.

Table 3.6 formalizes the research hypotheses.

The results of the t-tests are shown in table 3.7. It can be seen that the z_{score} of H_A and H_B are greater than z_{crit}, so it can be concluded that the null hypotheses of H_A and H_B are rejected at the 95% confidence level.

[6]We have previously verified that the standard deviations of the two groups were similar and the size of the samples was large enough (degrees of freedom = 198). So we have used the proper formula for large independent samples and equal variance.

Table 3.6: Hypotheses to be tested, where $\overline{b}$ is the average value of the defect density, $\overline{Locs}$ is the average value of the produced LOCs and $\overline{d}$ is the average number of contributors to the project.

	H_0 (null hypothesis)	H_1 (alternate hypothesis)
H_A	$\overline{b}_{noTdd} = \overline{b}_{Tdd}$	$\overline{b}_{noTdd} \neq \overline{b}_{Tdd}$
H_B	$\overline{Locs}_{noTdd} = \overline{Locs}_{Tdd}$	$\overline{Locs}_{noTdd} \neq \overline{Locs}_{Tdd}$
H_C	$\overline{d}_{noTdd} = \overline{d}_{Tdd}$	$\overline{d}_{noTdd} \neq \overline{d}_{Tdd}$

Table 3.7: Results of the two-sample t-test ($\alpha = 0.05$).

	z_{score}	z_{crit}	H_0
H_A	25.84	1.96	rejected
H_B	2.30	1.96	rejected
H_C	0.23	1.96	accepted

On the other hand, the z_{score} of H_C is less than z_{crit}, so we accept the null hypothesis, and it can be concluded that the two groups do not differ significantly in this respect. This result is aligned with the H_C hypothesis, which states that the number of developers is unrelated to the usage of the TDD practice.

Finally, we should emphasize that the z_{score} obtained for the H_B is quite close to its z_{critic} value. For this reason, even though the sample size is large ($n \geq 30$) and the null hypothesis has been rejected, further studies and experiments are needed to validate this statement.

3.3.6 Considerations

We presented a simulation model aimed to understanding whether or not the adoption of the agile practice of Test Driven Development can improve an open source development process. The starting point of our research can be summarized in the following hypothesis:

> *TDD yields code with superior quality, in terms of defect density per KLOC.*

This hypothesis was tested and statistically validated using a simulator of a FLOSS project. We found that the defect density decreased by 40% in the case of TDD. Also we investigated two other hypotheses. We found that the number

of developers contributing to a FLOSS project is unrelated to the usage of TDD practice and that code productivity decreases by 9% when TDD is fully adopted.

Chapter 4

Implementing the Agile Methodologies: three case studies

So far we've presented a mix of theories and related works to understand the Agile Methodologies and the issues related to its application in a distributed environment. In this chapter we will describe the results of our research activities related to investigating how the agile methodologies can be applied in a distributed environment. We will show real cases of distributed projects in order to understand if agile practices can be completely applied.

We will describe and analyze three exemplar cases in which the distributed agile development processes have been applied with success, they represent any real projects in order to evaluate the distributed agile methodology which is the main goal of this thesis.

The first of the cases we have studied concerns the application of an agile and distributed development process on a project which has involved a developers team distributed in the academic workspace. This study will best described in section 4.1: ***MAD*** Project. Rather than define how project should use Agile methodologies, we will attempt to explain what has worked for us.

Section 4.2 introduces the second case study: the **DART** Project. This is a research project on Web applications and one of the main goal to achieve was analyze how Agile methodologies fit the needs of Web development. We have introduced also the process metrics and automated testing technics which are best adaptable to the Web processes.

Finally the thirst case study shows the implementation of a Agile methodology, in the specific instance Scrum, to a research project. Can a research project be agile? In section 4.3 we describe our proposal of applying Scrum for the management of an European research project aimed at developing an agent-based software platform for European economic policy design. The use of an agile, adaptive methodology is justified because successful research projects are complex, unstable processes, which should be continuously adapted along their way. We describe in detail the roles, artifacts and practices of the proposed process, and the first steps of its adoption.

These case studies has been a wrap-up of lessons learned from which we arise some considerations in order to propose a distributed agile methodology.

4.1 MAD Project

Here we describe an experience, performed at the University of Cagliari, published in [13], [12], in which we try to use the XP process within a distributed context. We will seek to demonstrate through empirical evidence that XP values can be supported by multi-site team and we will present how to redesign some XP practices in order to fit the needs of a software distributed team.

This case study, called ***MAD*** (acronyms of ***"Metodologie Agili Distribuite)"*** was supported by MAPS [1] and closed in July 2007. The project has involved our research group (Agile Group of DIEE - Department of Electrical and Electronic

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

Engineering), CRS4 (Center for Advanced Studies, Research and Development in Sardinia), the Libera Università of Bolzano, the University of Genova and the University of Milano.

This research proposal was about a network of centers of competence on software development methodologies. Such centers, able to supply both computer science and firm organization competence, was specifically targeted to the most modern software development technologies, that is agile methodologies.

The main scientific objectives of the proposal was:

- To study, to be center of competence, and to spread agile methodologies for software development, and in particular Extreme Programming.;
- To study the agile methodologies from the point of view of the organization of the business processes, and the relationships between agile methodologies and quality certification;
- To validate empirically the software development processes, and in particular agile processes;
- To study the application of agile methodologies to software development made by programmers distributed along the Internet, with particular care to open source software development.

MAD Project is a case study included in the research project mentioned above. An outline of the case study's goals follows:

- To develop specific competence in software development among the students of the academic course in Electronic Engineering throughout the targeted education and learning in the field by applying software development with the support of the coach;

- To validate a "*Distributed Agile software development Model*" defined in iterative manner during the project and implemented in the development practices. The development monitoring and the software measuring allowed to validate it;

The real project consisted of the development of a web portal, integrating service and handling contents through a contents management system (CMS), for the University of Cagliari.

The software development process was made up of two phases.

The first one was the development of the kernel of the system and has been carried out by a co-located core team. In this phase the system has been developed following the classic XP practices [6].

In the second phase the system has been released as an open source project, and XP methodology has been redesigned to adapt it to a distributed environment.

Moreover the project saw also a phase called "zero". The aim of this phase was to bring the developers to common information level on the required elements in order to partecipate to the project productively. We carried out several activities (workshops, lesson courses, and so on) in order to provide a specific knowledge:

- Methodological Activity: with reference to the software development methodologies to adopt;
- Technological Activity: with reference to specific tools and environments, such as Java, Eclipse and Sourceforge, to learn.
- Specific Activity: with reference to the installation and advanced use of Open Source projects on which the software development phase must base it.

The main goal of this experience was to understand how the pure XP approach evolves while passing from the first to the second phase. We have investigated how

the values and practices of XP must be modified, removed or tool-supported as the first co-located team becomes dislocated.

4.1.1 Project Description

In this section we analyze and discuss the results obtained from our experience with this academic case study: the MAD project. Our goal was to investigate how the agile methodologies can be applied in a distributed environment and understand whether XP practices can be completely applied.

Briefly in order to reach this goal we have performed ideally two identical projects, the first one with a co-located team and the second one with a distributed team. Then, we have compared data coming from these two projects. Because this approach was obviously too expensive in terms of time, money and human resources, so we decided to establish only the project which follows the distributed methodology while the co-located one has been simulated.

The kernel was first built by a co-located team of 8 experienced software engineers applying agile practices. These practices include pair programming, testing, refactoring, planning game, short iterations. Among the core members there was two PhDs, one of them playing the role of proxy customer for the teacher and administrative sides, six PhD students and six undergraduate students, one of them playing the role of proxy customer for the student side. Moreover, the core team was supported by the work of a psychologist participating observant, who has played a role not contemplated by the XP methodology. We call her “The Spy”. The Spy has helped us to understand the dynamics of human interaction, in order to improve the quality of communication and the sense of team partnership. Each core team member has played a specific XP role (See Table 4.1).

After two months the team released a prototype of the system. Subsequently, undergraduate students have been join the project, and as a result the number of team

Table 4.1: *The core team.*

Background	XP Role
Ph.D.	Proxy Customer (Teacher and Administrative Sides)
Ph.D.	Coach
Ph.D. Student	Tracker
Ph.D. Student	Tester
Ph.D. Student	Developer
Ph.D. Student	Developer
Ph.D. Student	Developer
Ph.D. Student	Developer
Undergraduate Student	Proxy Customer (Student Sides)
Undergraduate Student	Developer
Undergraduate Student	Developer
Undergraduate Student	Developer
Undergraduate Student	Developer
Undergraduate Student	Developer
Psychologist	The Spy

members increased. So we have adopted an open source-like development model: 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. All team members lived in the same city and then 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 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. The other agile practices have been adapted to the new distributed context. This required several iterations before the team developed maturity in adopting agile distributed practices.

Five main steps have been followed:

1. Collection of metrics during the co-located phase;

2. Calibration and validation of the XP simulator using data gathered in the first phase. This goal has been achieved using an simulation modeling approach [56];

3. Simulation of the project as if it had been performed following the XP co-located methodology;

4. Collection of metrics during the dis-located phase.

5. Evaluation of the differences between the real and simulated project in terms of code productivity, rate of released functionalities (User Stories), project quality (defect density) and so on. Several metrics have been used either for the calibration and validation purpose or the quantitative comparison between the co-located and distributed approach. In order to compare it the XP-Evaluation Framework (XP-EF), described by Laury Williams of the University of Carolina [59], has been used. This is an ontology-based benchmark which defines the metrics that must be collected for each case study and to assess the efficacy of XP. In particular, *context metrics* are used to characterize the distributed and co-located teams and *effectiveness metrics* to quantify differences and analogies between the two methodologies. Among context metrics we cite Team Size (Number of Team Members), Team Education Level (Number of PhDs, Undergraduates) and Team Skill (Domain Expertize and Language Expertize). The effectiveness metrics include both process and product metrics. These Process Metrics have been extracted from XPSwiki [2] [44], [45], a web based XP project management tool. The tool can also be used to extract metrics such as Number of stories, Number of tasks per story, estimated and actual effort for each story and task, team velocity, release and iteration length and so on.

[2]http://www.agilexp.org/xpswiki

4.1.2 Co-located Experience: the First Phase

The first phase consisted of the development of a kernel set of functionalities. The development of the application kernel has been performed by the core team, within an almost pure XP co-located environment.

The XP methodology is based on a set of five values and a set of practices to be applied in order to make those values explicit [6], [8]. This section describes the kernel development, explaining which XP practices have been adopted by the core team in order to create a community that embraces XP values [13].

- **Communication** - Communication among team members must be maximized. Among core members there is a strong emphasis on direct communication creating a sense of team and effective cooperation. The core members apply some XP practices - such as *Sit Together*, *Informative Workspace*, *Pair Programming*, *Stories*, *Weekly Cycle* - in order to improve communication.

- **Simplicity** - Do the simplest thing that could possibly work. In the kernel development many XP practices such as *Weekly Cycle*, *Stories*, *Testing* and *Emerging Design from Coding* and *Refactoring* are applied in order to help development team to do the simplest thing.

- **Feedback** - Continuous feedback at different time-scale. *Weekly Cycle, Pair Programming, Stand Up Meeting, Real Customer Involvement, Testing* allow programmers to obtain feedback on their work.

- **Courage** - No Fear. All methodologies and processes are tools to handle and reduce the teams' fears. In order to reach this goal the core team is supported by the following practices: *Pair Programming, Testing, Simple Design and Refactoring.*

- **Respect** - "No King and No Queen." The core team applies *Sit Together* and *Shared Code* that help team-members to be respectful to their colleagues.

Now we explain in more details how the most important XP practices have been adopted during kernel development. This phase went on two months, and it is made up of seven iterations. During this phase we improved and tuned the adopted methodology.

- **Stories** - All system functionalities were written by two proxy customers and then estimated by developers to quantify the development effort. Until 3th Iteration each story was considered done when its tasks were completed, since 4th Iteration it was done only if its tasks were done and its acceptance tests were passed. This is an example of the methodology evolution.

- **Informative Workspace** - In order to track the project evolution, we used both an automated tool (XPSwiki) and a blackboard (see Fig. 4.1). The developers usually hang the stories on the blackboard. It is made up of three different parts: "to do", "in progress" and "done" where are collected the user stories depending on the status. This tool allowed the developers to know at any time the project evolution in detail.

Figure 4.1: Stories on blackboard.

- **Pair Programming** - During the first phase of the project, the software was developed in pair programming. The Extreme Programming [8] suggests that pairs should rotate frequently. We noted that the pairs did not rotate during the whole development task. At the beginning this behavior was (partially) tolerated, but since the second half of the project, the team leader imposed

the pair members exchange every development session. After a difficult, initial adjustment period, it has been observed an improvement in the communication and knowledge sharing. A proof of this is represented by a lower estimation error in terms of effort to implement the stories. In fact, as shown in Fig. 4.2, after the 4th iteration the estimation error decreased.

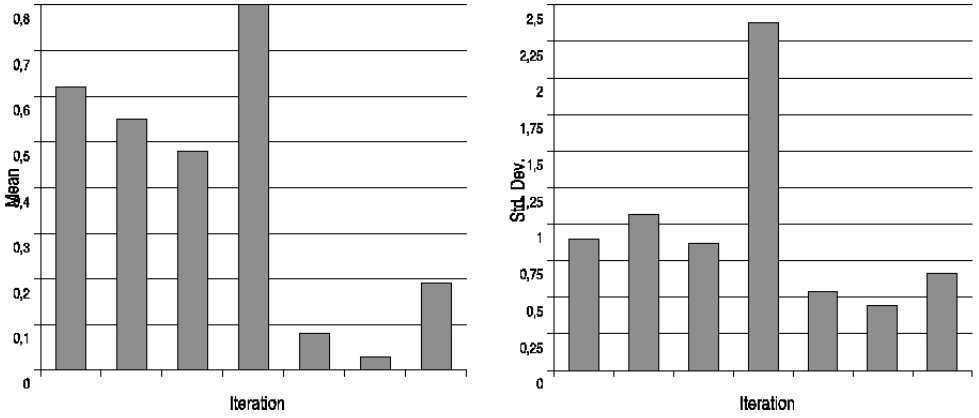

Figure 4.2: Estimation Error.

- **Weekly Cycle** - The team work was planned in a weekly cycle. At the beginning of every week the core team had a meeting. During these meetings, the team leader with proxy customers picked up the stories to implement for the week. Then developers broke the stories into task and each one was able to choose a story to develop. As said above, the system kernel was developed in pair. So, a developer who had not assigned neither a story or a pair was able to choose a pair.

- **Sit Together** - The kernel was developed in an open space big enough for the whole core team. The layout of the room (see Fig. 4.3) allowed the communication among different pairs.

- **Incremental Design** - This practice is quite difficult to apply and requires developers to have a great amount of experience. Also it is simple to misunderstand the meaning of this practice falling into a no-design behavior. In our project, we are hardly trying to make design naturally rising from coding,

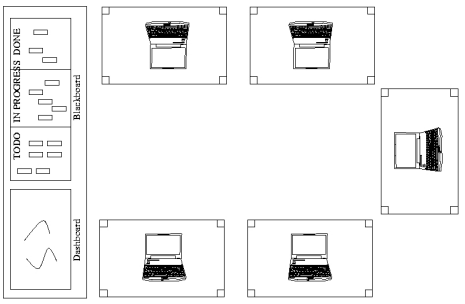

Figure 4.3: Open space map.

always doing the simplest thing that could possibly work.

4.1.3 Simulation Approach

Our research group developed an XP simulator that allows to forecast the evolution of an XP project [14], [57], [56]. This simulator implements some of the most significant XP practices (Pair Programming, TDD, Planning Game, etc.), and it is able to vary the adoption level of some of them. This simulator has been used to study how the MAD project would be evolved if it had been performed following the XP traditional methodology.

The product metrics we have gathered include project, class and method metrics. The project metrics are: *Number of Classes*, *Number of Methods* and *Total Lines of Code*. The class metrics include: *Number of methods*, *Lines of code*. Finally method metrics are the *Lines of Code of each method*. These metrics have been extracted from Subversion [3], the version control system that has been adopted in our case study. Subversion easily let us to examine the evolution of the source code. In fact it allows to capture snapshots of the project source code at any time instant. In particular we have analyzed the source code snapshots at the end of each iteration. The analysis has been performed using a system that is able to parse the source code and extract the desired metrics.

[3] http://www.subversion.tigris.org

The input parameters and the output variables that can be obtained from the simulation model are shown in table 4.2. Data used to calibrate and validate the simulator coming from the first release of the project. During this release the system kernel was developed.

Table 4.2: *Input parameters and output variables of the model.*

Input parameters	**Output variables**
Number of initial User Stories	Number of final USs
Number of developers	Defect density
Mean and standard deviation of initial USs estimation	Number of Classes, methods, DSIs
Initial Team velocity (pts/iteration)	
Number of Iterations per Release	
Typical iteration duration	

The calibration of the model parameters has been performed using data from the fifth iteration, such as the number of developers, the number of user stories and so on (see table 4.3).

Table 4.3: *Input parameters to calibrate the model.*

Input parameters	**Values**
Number of initial USs	30
Number of developers	14
Mean and standard deviation of initial USs estimation	265 (210)
Initial Team velocity (pts/iteration)	810
Typical iteration duration (days)	5

With these input parameters a number of simulation runs have been performed. In particular, we have iteratively calibrated the model parameters in order to fit better the real data of the project. In table 4.4 the simulation output are compared with the ones taken from our case study. Then, a model validation has been done using data gathered from the sixth iteration of the real project. In detail, we have changed only the initial number of stories developed (35), while we have maintained unchanged the model and project parameters obtained during the calibration of the

model (see table 4.5). We decided to choose data from these two iterations because we noted empirically that they were more significant than the previous ones because the methodology was not completely tuned. Note that the number of user stories developed at the end of the fifth/sixth iteration (see table 4.4, 4.5) is greater than the number of initial user stories set as input parameter. The initial user story could be split or new user story could be added during the development.

Table 4.4: *Calibration of the model parameters on the fifth iteration. Comparison between simulation results averaged on 100 runs and our case study. Standard deviations are reported in parenthesis. A story point corresponds to 1 minute of work.*

Output variable	**Simulation**	**Real Project**
Total days of Development	36.9(8.1)	36
Number of User Stories	38.2 (4.3)	39
Estimated Effort [Story points]	13697.6 (5191.1)	13252
Actual Effort [Story points]	17902.4 (4206.3)	18443
Developed Classes	77.2 (19.2)	80
Developed Methods	407.4 (100.7)	400
LOCs	3259.7 (805.2)	3260

Table 4.5: *Validation of the model parameters on the sixth iteration. Comparison between simulation results averaged on 100 runs and our case study. Standard deviations are reported in parenthesis.*

Output variable	**Simulation**	**Real Project**
Total days of Development	42.5 (9.5)	41
Number of User Stories	45.3 (5.2)	44
Estimated Effort [Story points]	15680.7 (5182.5)	15442
Actual Effort [Story points]	21369.8 (5337.2)	21570
Developed Classes	92.1 (25.4)	91
Developed Methods	488.0 (134.3)	451
LOCs	3904.0 (1075.2)	3951

After calibration and validation of the simulation model we have simulated the co-located project. We have performed 100 runs using the user stories of the whole project. In table 4.6 mean and standard deviation of the simulator output variables are reported. These data have been compared with those coming from the distributed project. The XPSwiki has been used as source of data to monitor and

control the distributed development. The data gathered at the end of the project have been used to evaluate the differences between the XP distributed methodology and the XP like co-located process.

Table 4.6: *Simulation results of the whole project averaged on 100 runs on our case study. Standard deviations are reported in parenthesis.*

Output variable	Simulation
Total days of Development	163.9 (20.9)
Number of User Stories	69291.5 (19009.5)
Actual Effort [Story points]	93109.1 (11198.2)
Developed Classes	2492.9 (405.1)
Developed Methods	5
LOCs	19940.8 (3239.3)

4.1.4 Distributed Experience: the Second Phase

The second phase consisted of a set of add-on functionalities to be plugged on the kernel.

The main objective of the second phase of the project was gradually to move towards a distributed environment, while keeping XP values adapting XP practices and introducing supporting tools.

We explain how the adopted XP practices are affected by the transition from a co-located to a distributed environment.

- **Pair Programming** - While this practice represents a powerful tool for supporting communication and knowledge sharing among team members, it strongly needs the co-location of the team. As there are few examples of techniques to support distributed pair programming [51],[27], we initially decided to renounce in applying this practice. Then we tried to use some tools such as sobalipse [4] and skype [5], that guaranteed a good communication level between the two developers of a distributed pair. Sobalipse allows developers to sit at

[4]http://sourceforge.net/projects/sobalipse/
[5]http://www.skype.com/

different machines to edit and review the same file, and to communicate using a chat or better using the VOIP technology. One of the most important features of these tools is that they are completely integrated in the development environment [6].

- **Real Customer Involvement** - This practice has been applied also in the second phase of the project because there are two developers playing the role of proxy customer.

- **Sit Together** - In a sit together team all developers work in an open space in order to maximize communication among team-mates. It absolutely needs the co-location of the team so it is not applicable to a distributes team. Anyway, we used some tools such as a wiki, an instant messenger and a mailing list in order to favor a "sense of team" among distributed developers.

- **Informative Workspace** - This practice allows the team to improve the communication. In the second phase of the project this practice is represented only by XPSwiki [7] [45]. The Fig. 4.4 shows the XPSwiki user interface.

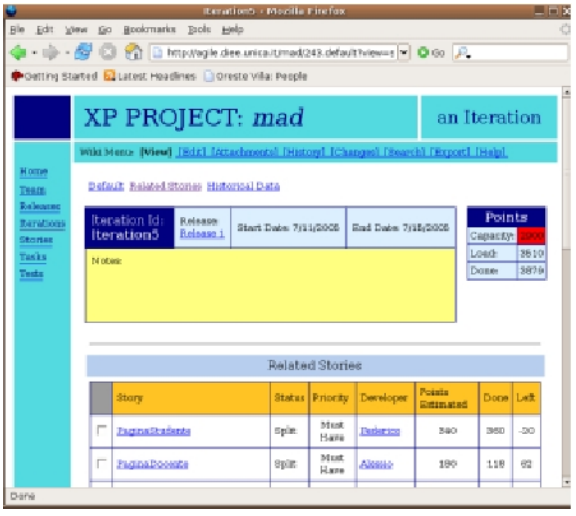

Figure 4.4: XPSwiki User Interface.

[6] http://www.eclipse.org/
[7] http://www.agilexp.org/xpswiki

- **Shared Code** - Each team member is enabled to access to a SVN code repository in order to improve and change any part of the system at any time.
- **Continuous Integration** - In the second phase the continuous integration is automated and performed after no more than a couple of hours.
- **User Stories** - The functionalities of the system are described using stories that are published in the XPSwiki.
- **Weekly Cycle** - By using the XPSwiki tool, each developer can understand which user stories are done, in progress or to do in the weekly cycle.

4.1.5 Metrics

The project was monitored by calculating product metrics during its development. 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 [15] [17], usually known as the *CK suite*, is the most validated.

In this case study we have adopted this CK suite.

CK Metrics

The CK suite is composed of six metrics:

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 [15]. 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 aClass (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).

Metrics Evolution

We have analyzed the 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. In this section we show an other analysis. The project has been divided into three main steps:

- step 0: A co-located team developed the kernel;
- step 1: The team experimented and optimized agile practices in a distributed environment;
- step 2: The team developed maturity in the application of key practices.

We have analyzed 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.

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 rapidly as shown in fig. 4.5. The last CVS snapshot consists of 277 classes, indicating that the system doubled in size during the distributed phases.

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 4.5 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 4.5 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. 4.6, the metric increases more rapidly

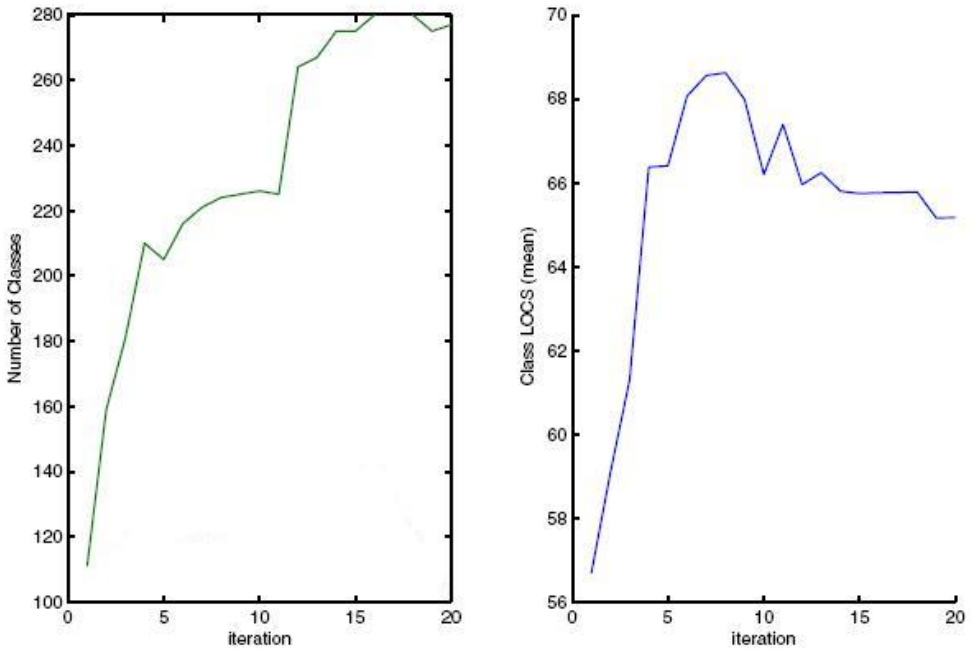

Figure 4.5: Total number of classes and lines of code per class evolution (1 iteration = 2 weeks.

in the step 2 than in the step 1. This might be explained by the faster growth of the total number of classes in the step 2 but examination of the plot in fig 4.5 shows that this hypothesis can be reasonably ruled out. The main reason is certainly the maturity developed by the team in the step 2, 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. 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 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, etc.). The total number of assertions may be regarded as a more comprehensive indicator of testing activity. This metric, reported in fig. 4.6 shows the same trend observed for the number of test cases.

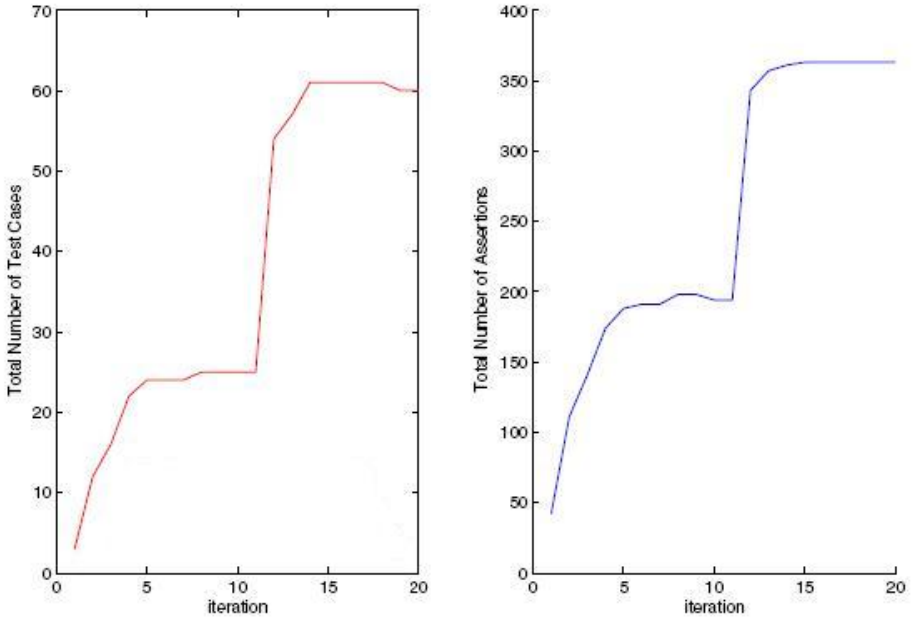

Figure 4.6: Number of test cases and number of assertions for each iteration (1 iteration = 2 weeks).

LCOM and WMC. The evolution of LCOM reported in fig. 4.7 shows a step 1 where classes are characterized by low cohesion and a step 2 where this metric has been progressively improved through refactoring. The same considerations discussed above also apply to WMC: step 1 is characterized by a growing number of methods per class and a step 2 where fat classes were split into cohesive classes with a small number of methods.

CBO. The evolution of this metric reported in fig. 4.7 shows a step 1 where class complexity increases followed by a step 2 where this metric remains approximately constant. The mean value increases from 6 to 8 during step 1 and stabilizes at 8 during step 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. 4.7) shows an initial increasing phase followed by a 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.

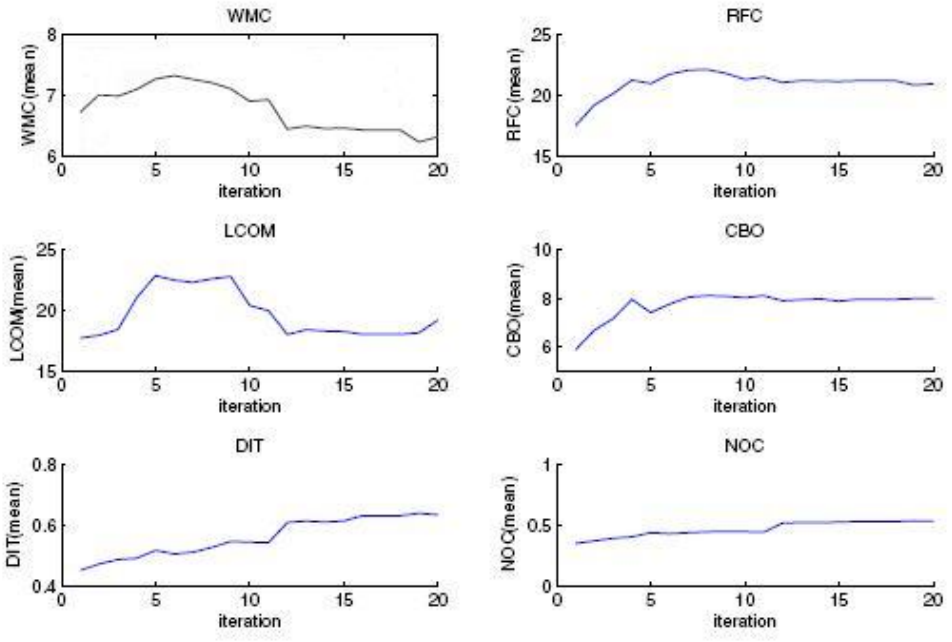

Figure 4.7: CK Metrics Evolution (1 iteration = 2 weeks).

4.1.6 Discussion and Considerations

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 step 2, 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. 4.5.

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 (Ph.D.) 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.

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 was monitored by calculating product and process metrics during its development. Product metrics included the CK suite of quality metrics. Analyzing the evolution of these metrics we found that in step 1 the team increased system complexity. In step 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.

4.2 DART Project

The *DART* Project [8] has two main aims: to develop models and system architecture for building a semantic and distributed research motor and to study and realize a framework application in order to maximize effectively the offer of contents that is measured like as user satisfaction.

The project takes place in the labs of three research partners: CRS4, Tiscali, University of Cagliari (in the specific instance the DIEE), all located in Cagliari. The project structure is spread in 36 months and subdivided in 7 Objectives to realize (OR).

Our research group has took part in the development of the OR7's activity. We have defined the methods and the metrics in order to validate and develop the software. In particular, we have defined the Agile Methodologies for the Web applications development.

4.2.1 Agile Metodologies for Web Applications

The current project sets itself a goal that is make industrial research activity on Agile Methodologies applied to Internet distributed systems.

[8] DART is a project of *Distributed Architecture for the Semantic Research and the Personalized Fruition of Contents*, Project n. 11582/02

Most software development is performed either maintaining existing systems, or developing Web applications. We define Web application development as building software that interacts with the user through a Web browser, and where many, or most of its functionalities are obtained by assembling together software components already available. The Web development industry comprises many disciplines. It formed out less than years ago and it includes teams of programmers, graphic designers, usability specialists, project managers, and so on, all look at projects in very different ways. A Web application is commonly structured as a three-tiered application. The outer tier is a standard Web browser; the middle tier is an engine using some dynamic Web content technology, while the inner tier is a data repository. The Web browser sends requests to the middle tier, which services them by making queries and updates against the repository and generates the proper HTML code in response.

Agile methodologies are becoming mainstream in software engineering. More and more projects worldwide are being managed in this way. The availability of qualitative and quantitative data about Web application projects using agile methodologies is still quite scarce.

Can Agile methodologies map the needs of Web projects? Agile Methodologies could potentially deal with a number of Web development problems, but there are a number of key differences between the two.

Teams Web projects involve multidisciplinary teams made up of graphic designers, copywriters, Flash programmers, server side programmers, interface programmers, testers and project managers. All have several skills and abilities and the roles and responsibilities are not very clear (e.g.: responsible person for how the page displays is the designer but also the interface programmer).

XP is very good at getting programmers to communicate among themselves and with customers.

Support for Multiple Environments. Traditional software projects are developed to handle different user environments (e.g.: an application can have a version for Windows, a version for Unix and for Mac, depending on its audience. Web applications have only one version of the Web site and it must be able to support multiple browsers on multiple operating system simultaneously.

How does XP allow for the varied support needed in one product?

Quality. Many original Web sites have a low quality code. Among the Web developers object-oriented approach to development are not very used. But we know that object-oriented approach simplifies the refactoring and code maintenance.

How can XP help to improve quality?

Testing. In Web projects a lot of testing practices are necessary because testers must account multiple customers and test interfaces in totally new ways. (e.g.: page layout, design, screen color, etc.)

Rapid Deployment. Normally software release are large in the time and patches and updating can be used to extend the life of a program. Web projects can be deployed as often as the customer want. So Web projects need to harmonize continuous integration with new releases.

How does XP accommodate the need for frequent deployment?

Above we analyzed only some aspects to consider in these projects, but we can start from it in order to identify the best practices more adapted in the Web development.

4.2.2 FlossAr Project: a Related Work

Here we present the development of a Web application using agile practices. This software project consists in the implementation of FlossAr [20], a Register of

Research software for universities and research institutes, developed with a complete object-oriented (OO) approach, released with an Open Source license. FlossAr manages a repository of data about research groups and research results (papers, reports, patents, prototypes) aimed at helping research evaluation and matching between firms looking for technologies and knowledge, and researchers supplying them. It is a Web application, because both researchers who input their profiles and products, and people looking for information access the system through a standard Web browser. FlossAr has been implemented through a specialization of an open source software project. We intend for specialization the process of creating a software application customized for a specific business, starting from an existing, more general software application or framework. The general framework we customized is jAPS (Java Agile Portal System) [9], a Java framework for Web portal creation released with GNU GPL 2 open source license. jAPS comes equipped with basic infrastructural services and a simple and customizable content management system (CMS). It is able to integrate different applications, offering a common access point. FlossAr has been developed by a co-located team of four junior programmers, coordinated by a team leader, with no previous experience of agile development. The project consists in Java development of a Web application. Throughout the project we collected metrics about the software being developed. Since Java is an OO language, we used the Chidamber and Kemerer (CK) OO metrics suite [15] [17] [16] described in 4.1.5 .

The FlossAr project evolved through five main phases, that are summarized below:

1. Phase 1: an exploratory phase where the team studied both the functionalities of, and the way to extend the underlying system (jAPS). It started on February 15th, 2007 and ended on March 7th.

2. Phase 2: a phase characterized by the full adoption of all practices, including

[9] http://www.japsportal.org

testing, refactoring and pair programming. It started on March 7th and ended on May 12th, comprising the systematic implementation of a set of the needed features.

3. Phase 3: this is a critical phase, characterized by a minimal adoption of pair programming, testing and refactoring, because a public presentation was near, and the system still lacked many of the features of competitor products. So, the team rushed to implement them, compromising the quality. This phase started on May 12th, included the first release of the system on May 28th, and ended on July 3rd.

4. Phase 4: an important refactoring phase, characterized by the full adoption of testing and refactoring practices and by the adoption of a rigorous pair programming rotation strategy. This phase was needed to fix the bugs and the bad design that resulted from the previous phase. It started on July 3rd and ended on August 3rd with the second release of the system.

5. Phase 5: Like phase 2, this is a development phase characterized by the full adoption of the entire set of practices, until the final release on October 1st, 2007.

Agile Practices and Software Quality

Now we present and discuss the agile practices that have been used in the software process, and the trend of various CK metrics, in the context of what actually happened to the project, and to the actual extent of use of agile practices.

Very often, software teams intending to pursue an agile approach do not follow a specific AM, but discuss and decide a set of agile practices to be used, and from time to time review the project and make adjustments to these practices. Web application development is relatively new, and it lacks the many consolidated programming practices applied in traditional software development. One of the main peculiarities

of this kind of development is a heterogeneous team, composed by graphic designers, programmers, Web developers, testers. Moreover, the application has typically to be run on different platforms, and to interact with legacy systems. Consequently, the choice of the development practices to use is of paramount importance for the success of the project.

The team first defined some principles to be followed:

- Code reuse: not only the underlying framework, but also the specialized software produced must be reusable. There is no such thing as a general purpose register of research, but each university or research institution wish to customize such a system to cope with its specific features and requirements.
- Evolvability: it is very important that the software can be easily modified and upgraded, in a context where requirements were unclear since the beginning, and new requirements might be added continuously.
- Maintainability: errors and failures must be fixed quickly, in every software module composing the system, minimizing the probability to introduce new when old bugs are fixed.
- Modularity: the functional blocks of the underlying framework must be kept, and new modules must be added on each of them, with no intervention on the framework itself.
- Portability: this was a specific requirement of FlossAr, to be able to spread it in the hardware and software contexts of different research institutions. For other specialization projects it might not be important.

Following the principles above, the team chose and defined a set of agile practices, most of them derived from XP methodology. They were:

- Pair Programming: this practice might be considered difficult to apply in the context of a heterogeneous team. In our case, however, it was one of the keys

to the success of the project. All the development tasks were assigned to pairs and not to single programmers. Given a task, each pair decided which part of it to develop together, and which part to develop separately. The integration was in any case made working together. Sometimes, the developers paired with external programmers belonging to jAPS development community, and this helped to grasp quickly the needed knowledge of the framework.

- Whole Team (On Site Customer): a customer's representative was always available to the team. This enabled a customer-driven software specialization which, in the first months of development, led to a deep redefinition of the structure and features of the system.

- Continuous integration: the written code was integrated several times a day.

- Small Releases: taking advantage of the customer on site, the development was divided in a sequence of small features, each separately testable by the customer, guaranteeing a high feedback level. There were three major releases, at a distance of two months each other.

- Test-Driven Development (TDD): all code must have automated unit tests and acceptance tests, and must pass all tests before it can be released. In its most extreme acception, tests are even written before the code. In the presented project, the choice whether to write tests before or after the code was left to programmers. However, they had a strong requirement that all code must be provided of tests.

- Design Improvement: a continuous refactoring was practiced throughout the project, to eliminate code duplications and improve hierarchies and abstractions.

- Coding Standards: the same coding standards of the original jAPS project were kept to increase code readability.

- Collective Code Ownership: the code repository was freely accessible to all programmers, and each pair had the ability to make changes wherever needed. This was eased by the uniformity of technical skills of all team members.

- Suitable Pace: this practice was enforced throughout the project, with the exception of the week before the main releases, when the team had to work more than forty hours to complete all the needed features on time.

- Stand-up Meeting: every day, before starting the work, an informal short meeting was held by the team, to highlight issues and to organize the daily activities.

- Feature List and Build by Feature: these practices were inspired by FDD agile methodology. The Feature List is also called *Backlog* in the terminology of Scrum AM. A list of the features to implement, ordered by their relevance, was kept in a Wiki, and the system development was driven implementing them. These features are user-oriented, meaning that most of them describe how the system reacts to user inputs, and have a priority, agreed with the on site customer. Each feature must be completed at most in one week of pair programming; longer features are divided in shorter sub-features satisfying this constraint. The Feature Design activity of FDD was seldom performed before programming, except in the first iterations of the development, when architectural choices were made, and UML diagrams were created to document them.

The resulting software process is a standard agile one, proceeding by short iterations and taking advantage of many *classical* agile practices, mainly taken from XP. Its main peculiarity is the control process, which is less structured than Scrum Sprint and XP Planning Game, with less meetings and standard artifacts.

The goal of some of the agile practices quoted above is to enable the team to successfully answer to changes in the requirements, and to maximize feedback with

the customer and among the team. Whole Team, Small Releases, Stand-up Meeting, Feature List and Build by Feature are all practices of this kind. The practices have no direct impact to the quality of the code, expressed for instance as number or defects. The other practices concern how the code is written and upgraded, and have a direct impact on the quality of the code. Among them, we deem that the most effective to improve code quality are Pair Programming, TDD and Design Improvement, or Refactoring.

Pair Programming, by forcing two developers to work simultaneously at the same piece of code on the same computer, allows the team to share the knowledge of the system. The effectiveness of the Pair Programming also depends on the pair rotation strategy, that allows the maximization of communication among developers. A software system can be described as a network of interconnected components and the collective knowledge of such a system allows the team to easily add new functionalities and fix the bugs. On the other hand, the reduction of the amount of communication leads to specialization. Specialization means that a single developer has knowledge of, and is comfortable with, only a piece of the entire system. Given that in Object Oriented programming a class will cooperate with other classes, an issue can arise when a programmer needs to use a module developed by another programmer. This gap of knowledge can lead both to an increased fault proneness of the module, and to difficulties in performing refactoring activities on it. For example, a refactoring activity which aims to reduce the value of coupling metrics will certainly take advantage of a global knowledge of the system.

TDD means that the production code should be adequately covered with automatic tests. Tests are an excellent documentation tool because they describe the behavior of a piece of code in term of assertions that compare actual and expected values. Tests may also be used as a design tool by writing them before the production code. Following a rigorous testing strategy, programmers are forced to write

testable code and this encourages the production of quality code. In the case of classes with many collaborations with other classes, testing is difficult because these dependencies lead to other objects that must be properly initialized, or simulated with mock objects, during the test set up. It is reasonable to assume that a reduction of the testing level has an impact on coupling metrics and on the method length.

Refactoring is a practice which aims to simplify the system, without changing its functionalities. A software system can be represented as a network of interconnected entities and its readability can be improved by applying some refactoring practices. Some of these practices need a global knowledge of the system, and perform better if they are combined with pair programming and testing. For example, reducing the coupling among classes requires not only a local knowledge of them, but also the knowledge of the complex network of relations among these classes. Other refactoring practices are finalized to the reduction of local complexity, and can be performed by programmers with no global picture of the system.

Metrics Evolution

In this section we show the evolution of FlossAr source code metrics. At regular intervals of two weeks, the source code has been checked out from the CVS repository and analyzed by a parser that calculates the metrics. The parser and the analyzer have been developed by our research group as a plug-in for the Eclipse IDE. Among CK metrics, we do not show NOC and DIT metrics, that are not particularly interesting.

Number of classes. This metric measures the total number of classes (including abstract classes and interfaces), and is a good indicator of system size. The number of classes generally increases over time, though not linearly. As reported in Fig. 4.8, the project started on January 29th, 2007 with 362 classes (those of jAPS release 1.6). At the end of the project the system had grown up to 514 classes, due to the development of new features that constituted

the specialized system. Besides the class number, we analyze in detail the evolution of CK and LOC metrics, that are very useful for understanding the complexity of the system developed, and to assess the quality of the product built and how the agile practices used during the project affected the metrics in the different phases.

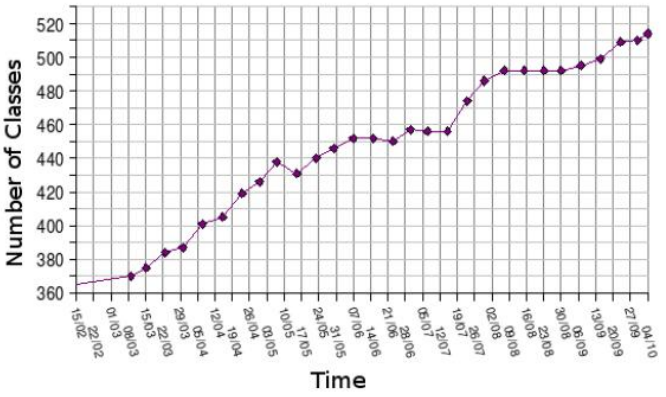

Figure 4.8: The evolution of the number of classes.

WMC (Weighted Methods of a Class). WMC is calculated as the total number of methods defined in a class (NOM). The evolution of the mean value of WMC, reported in Fig. 4.9, shows a peak of about 7.2 methods per class at the end of the third phase, which decreased in the subsequent phase.

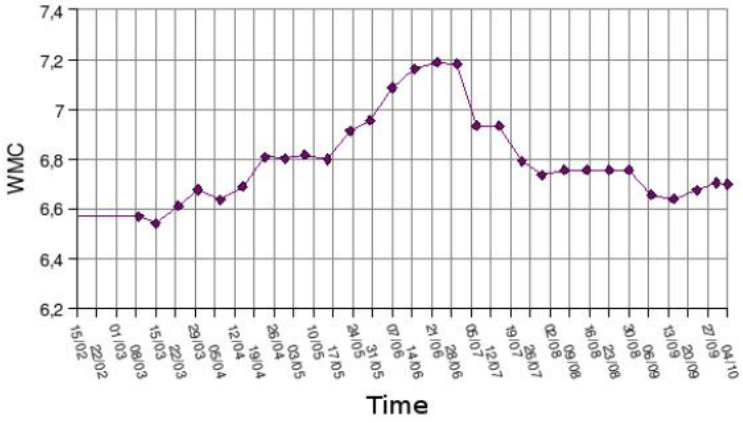

Figure 4.9: The evolution of the mean value of WMC metric.

RFC (Response for a class). The RFC is a measure of the complexity level of system classes. The evolution of the mean value of RFC, as reported in Fig.

4.10, is very similar to that of WMC. These two metrics always show a strong correlation in literature, and the peak of RFC, like the peak of WMC, happened at the end of the third phase, with a mean value of 16.38 methods callable in response to a generic message sent to an instance of the class.

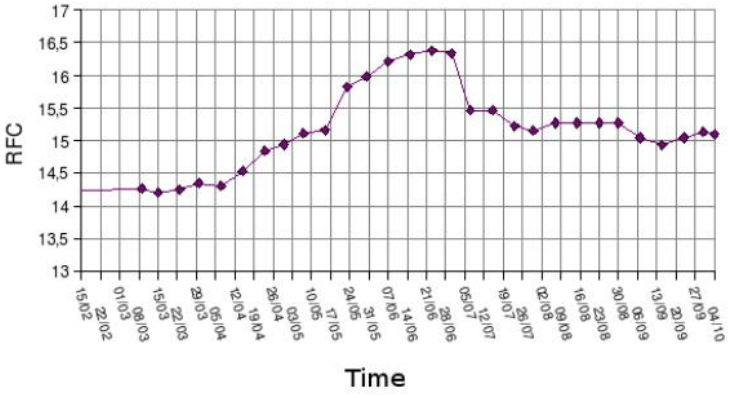

Figure 4.10: The evolution of the mean value of RFC metric.

CBO (Coupling Between Objects). When this metrics increases, the refactoring, development and testing operations become more problematic. In literature, CBO always show a high correlation with RFC. As mentioned for the previous metrics, there is an increasing trend that reaches the average value of 4.7 classes at the end of the third phase. CBO, compared with RFC, shows a considerable growth at the beginning of the third phase, and a slower growth until the end of the last phase, just before the third release.

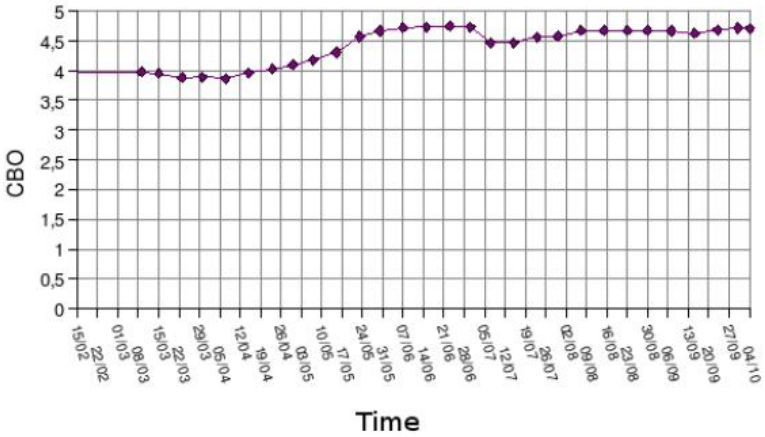

Figure 4.11: The evolution of the mean value of CBO metric.

LCOM (Lack of Cohesion in Methods). In literature LCOM is considered highly correlated with WMC, and as reported below its behavior shows many similarities with WMC. The evaluation of LCOM reported in Fig. 4.12, shows an increase and reach a peak between 7th and 15th July, at the beginning of the fourth phase (WMC peak happens two weeks before), where it reaches a mean value slightly less than 36 methods. Like other CK metrics, LCOM shows an ascending trend between May 30th and June 15th, just after the first release, and a subsequent decline until it stabilizes at an average value just below 33 methods.

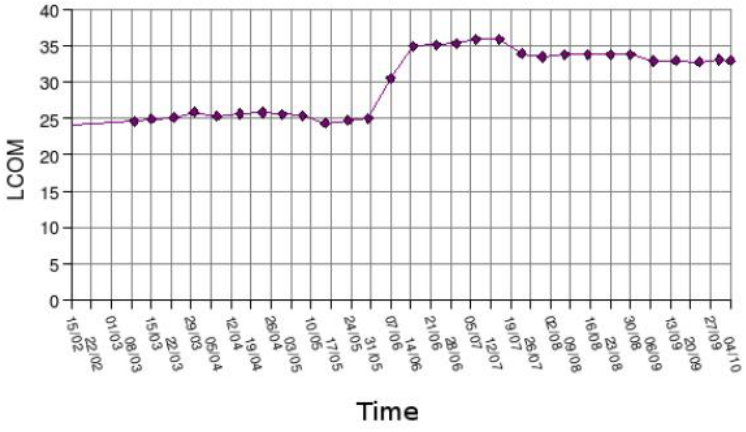

Figure 4.12: The evolution of the mean value of LCOM metric.

Class-LOCs. This metric roughly represents the average size, and consequently the average complexity of the classes of the system. The evolution of average Class LOCs, reported in Fig. 4.13, shows a high correlation with RFC metrics. The Class LOCs, reached an average value around 83 lines of code on June 23rd , before declining gradually to 76 lines of code per class in the last release.

Method LOCs. This metric is the average number of lines of code of a method. The evolution of Method LOCs, reported in Fig. 4.14, shows an increasing trend until June 7th, and then a stabilization. This metric shows also a slight decrease, also visible in other metrics, just at the beginning of the second phase. It also shows two oscillations in July and September, when the product

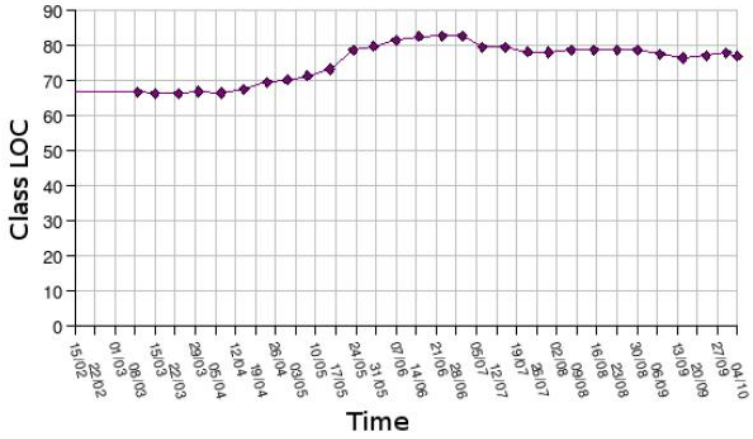

Figure 4.13: The evolution of the mean value of Class LOCs.

was already mature.

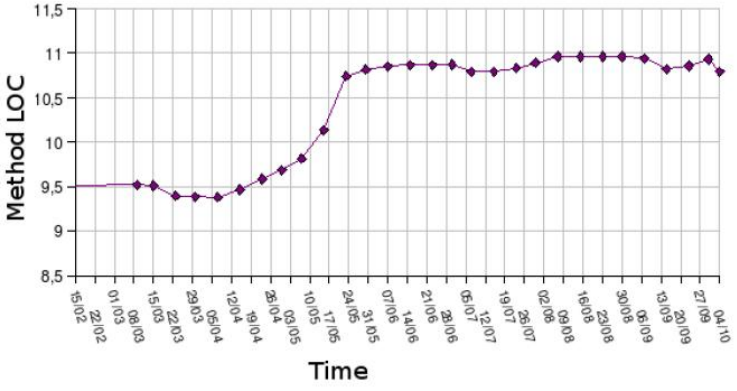

Figure 4.14: The evolution of the mean value of Method LOCs.

Dicussion

As described the level of adoption of some agile practices, like pair programming, testing and refactoring was highly variable in the different phases of the process. We observed that this variability in the adoption level of some practices has influenced the evolution of the metrics. Fig. 4.15 shows the evolution of three metrics (RFC, method LOCs and LCOM) in conjunction with the process phases. All of these three metrics should be kept low for having a system of good quality.

Phase 2 is characterized by a growing trend both of RFC and LOCs, and by a

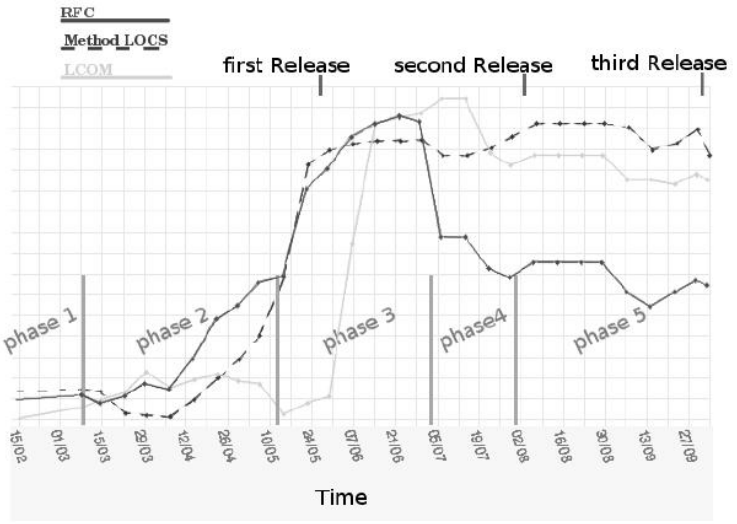

Figure 4.15: Development phases and evolution of some key metrics related to software quality.

stationary trend of LCOM. Note that phase 2 is characterized by a rigorous adoption of agile practices, but we should consider are two aspects: some agile practices require a time to be mastered and an OO system in the initial growing phase has a sub-optimal structure, that evolves towards an optimal configuration.

Phase 3 is characterized by strong pressure for releasing new features and by a minimal adoption of pair programming, testing and refactoring practices. In this phase we observe a growth both in the coupling metrics and in the local metrics, indicating that in this phase the quality has been sacrificed for adding more new features. Phase 3 is followed by a strong refactoring phase where the team, adopting a rigorous pair programming rotation strategy in conjunction with testing and refactoring practices, were able to reduce the values of the most important quality metrics.

It is worth noting that the values of the metrics at the end of phase 4 seem to have reached an equilibrium that depends on the size of the system and on its topology.

The next development phase (phase 5) is characterized by the adoption of pair programming, testing and refactoring practices, and by the addition of new classes associated to the new features. In this phase the metrics don't change significantly, though in the end their values are slightly lower than at the beginning of the phase, because the team has become more effective in the adoption of the agile practices compared to the initial phase 2.

In general, the coupling and complexity metrics evolution seems to agree with what we would expect considering this evolution under the perspective of practice application.

4.2.3 Best Practices

In this section, on the basis of our empirical studies, we propose the best agile practices can be easily applied in the Web development and in tab. 4.2.3 we show their applicability degree. Web projects are different from software projects and there is a need to adapt XP practices to suit the needs of a multidisciplinary team.

Pair Programming. In pair programming practices two developers work together on the same problem. This practice allows knowledge sharing. In Web development there are different skills and knowledge and this practices could be useful. For example, if the graphic designers work with a developer, for sure s/he could understand how pages are put together in order to improve its creations. This collaboration provides best results. It is advisable also interfacing customers with testers since the customers generally don't have experience in testing and it is hard to define acceptance tests. Moreover this practice is useful also for testers: customer feedback improves test development. Finally the pair programming seems supporting only co-located team members. Actually in literature there are some examples in which the pair programming is adopted in distributed environments (see [51] and [27]). These problems can be

overcome adopting tools such as sobalipse [10] or skype [11]. These tools assure a good level of communication in the case of distributed pair programming. We used these tools also in MAD project (see sec. 4.1), and we obtained the same results. Moreover they are completely integrated in distributed environments such as Eclipse. [12] Finally as shown in section 4.2.2, when the development phase (phase 3) is characterized by strong pressure for releasing new features and by a minimal adoption of pair programming, we observed a growth both in the coupling metrics and in the local metrics, indicating that in this phase the quality has been sacrificed.

Sit Together. In "sit together team", all team members would work together in open spaces in order to maximize the communication. This practice is easily manageable with the co-located team. A wiki or mailing list can be used in distributed team. Also customer always present is an effective practice because having someone to approve a design, explaining a business process, or suggesting a way around problems drives the project faster.

Short Iterations. Our advice is to keep iterations small. In [58] it is recommended that Web project follow a two-week iteration schedule. This is a good practice because it allows you to work on a number of stories across the Web suite without having to include one that depends on another. Generally the graphic design process calls for a number of stories that take two weeks. Two weeks

[10] http://sourceforge.net/projects/sobalipse/

[11] http://www.skype.com

[12] Eclipse (http://www.eclipse.org) is an open source community whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the entire lifecycle. Eclipse is well known for its Java IDE. However, there are Eclipse base language IDEs from most of the popular languages. There are a number of Eclipse project that provide framework that can be used functional building blocks to accelerate the software development process. Eclipse projects provide tools and frameworks that span the entire software development lifecycle, including modeling, development, deployment tools, reporting, data manipulation, testing and profiling. The tools and frameworks are primarily focused on building JEE, web services and web applications. Unlike developer tools, application framework are deployed with the actual applications. these framework can be used either as standalone addition to Java applications, or can be leveraged as components on top of the Eclipse RCP. This enables applications to use an integrated stack of open source framework on RCP for quickly building and deploying their application. This provides a powerful combination for building software.

is along enough iteration to make good progress without adding unnecessary risk.

User stories. In the XP approach user stories are written from the customer. In Web development the customer is not expert in Web projects. A priority is to write simple stories (to use a natural language, that is the customer language). In this environment using tools supporting practices during the process is an important requirement. We propose XPSwiki [45], a tool supporting the process in the phase of requirement gathering or planning game

Design. In Web applications the code can be optimized through:

- writing code as more simple as possible;
- using CRC cards (Class Responsibilities and Collaboration) to define the classes to implement;
- naming conventions: at the beginning of each project, the team should discuss and agree to a directory structure and file naming conventions in order to save time;
- using prototypes in order to reduce risks: every Web project is different from the one before. There is always a feature that you haven't done, new navigations, and so on. These risks can be reduced with the prototypes;
- refactoring. Over time Web site components become obsolete and the design needs to be updated. At the end of each iteration refactoring is useful in order to verify that multiple objects doing the same thing are not present or that the site communicate the original vision. Refactoring the stories is also very important to improve customer relationships.

Testing. Unit testing for Web projects is almost identical to that for software development because most programs run on the server side. The difference is that in Web projects many programs require input from and output to a remote customer, typically the communication between the server object and the

customer is via HTTP over TCP/IP. An other aspect to consider are multiple browsers: the most important nonfunctional requirement of any Web project is the browser you will support. Testing must be independents from the browser because each browser has a different version.

In Web application is not very simple write automated acceptance tests to verify all functionalities. There are some frameworks such as jWebUnit that help this practice. Nevertheless the human validation, in order to evaluate the interface of the system and so on, is necessary.

Table 4.7: *Agile practices in Web applications.*

Practice	**Very Adapted**	**Applicable**	**Not Recommended or not Tested**
Pair Programming	X		
Work Space		X	
Sit Together		X	
User Stories		X	
Test Driven Development		X	
On Site Customer		X	
Coding conventions		X	
Simplicity		X	
Code collective ownership			X
Refactoring	X		
Short Releases	X		
Incremental Project	X		
Short Iterations	X		
Continuous Integration		X	

4.3 Eurace Project

This chapter describes our experience implementing an agile process on a management of a research project called Eurace [39] [13] aimed at developing an agent-

[13]Eurace has been approved by CE on the April 2006 with contract nr. 035086. Full title: *"An agent-based software platform for European economic policy design with heterogeneous interacting agents: new insights from a bottom up approach to economic modeling and simulation"*

based software platform for European economic policy design with heterogeneous interacting agents. In particular, the Eurace project proposes an innovative approach to macroeconomic modeling and economic policy design according to the new field of agent-based computational economics (ACE).

The Eurace project is for his nature multidisciplinary. The partners are Research Units composed by engineers, computer scientists, economists, physicists and mathematician. The management of the project involves coordination of audit certificates and the maintenance of the consortium agreement, managing the central finances of the project, the overall legal, contractual, ethical, financial and administrative management, coordination of the knowledge management, implementing competitive calls for the potential participation of new contractors and coordination of the technical activities of the project. The structure of the Eurace project and the objective need an adequate management and an instrument of exchange and coordination throughout the duration of the all project.

In our opinion Eurace project such as each research project is a complex process, not stable. This means that every action, practice, and technique is not simple or repeatable, its predictability is limited and that is difficult to control. The activities may require constant change in directions, it may add new tasks or it may require unforeseen interactions with many others participants. We can affirm that activities such as scientific research, innovation, invention, and software development, typically exhibit such behavior common to an empirical process.

In this context we will describe common pitfalls and effective approaches to affect not only the development team members, but also other teams, departments, management, research scientists or software developers. We want transfer the software engineering approaches to manage a research project, in particular we will using agile processes as an attempt to micromanage the Eurace project.

The assumption is that this scientific project is an empirical process, in other words the project cannot be well defined and the uncertainty is inevitable. In order to manage this complexity we propose the use of Agile processes, and in particular

Scrum: a lightweight process that can manage and control software and product development, in other words a project management process.

4.3.1 Why Scrum?

Why Scrum in a research project? Our group is also responsible for introducing new technologies and processes into the project and moreover we have long used models and frameworks successfully, not just for software design but also for organizations and process management. The structure of the Eurace project and its objectives need an adequate management and a tool to exchange and coordination throughout the duration of the whole project. Our goal is to provide an effective management system for the project as a whole as well as the coordination of the individual members of the Consortium. The assumption is that this scientific project is an empirical process, in other words the project cannot be well defined, and uncertainty is inevitable. In order to manage this complexity, we propose the use of Scrum: a lightweight process that can manage and control software and product development, in other words a project management process. So we transfer the software engineering approaches to manage a research project. We propose Scrum because it is scalable from single process to the entire project. Scrum controlled and organized development and implementation for multiple interrelated products and projects, with over a thousand developers and implementers. Moreover Scrum can be implemented at the beginning of the project or in the middle of the project or product development effort that is in trouble. To our knowledge, this is the first time an Agile Methodology is applied to the management of a distributed research project such as EURACE. This is a big challenge. However, the effectiveness of Agile Methodologies to control software, and not only software, projects and to manage changes in the requirements even late in the development, is promising. The majority of project management methods and techniques are very prescriptive; they tie us down to a fixed sequence of events and offer little in the way of flexibility. Scrum is mature; it has a track record going back as far as 1995 and earlier. Similarly, it

is scalable: Scrum can be used on projects of any size and to manage enterprise-level projects. The rules and practices from Scrum, a simple process for managing complex projects, are few, straightforward, and easy to learn. To adapt Scrum to the management of a research project, we must keep in mind the analogies and differences between a development project and a research one.

4.3.2 Implementation of EURACE Scrum

Although Scrum was intended originally to manage of software development projects, we believe it can theoretically be applied to any context where a group of people need to work together to achieve a common goal such as a scientific research project. The Scrum methodology is designed to be quite flexible and the controlled involvement of all team is facilitated. The following teams are formed for the Eurace Scrum project:

Project Owner (formerly Product Owner): This person controls that the artifacts delivered by the process are in line with the research project aims and the required deliverables. A natural candidate for this role is the Project Coordinator.

Scrum Master: In a distributed research project the Scrum Master is a person who is responsible for enforcing the rules of the process, helping to remove impediments and ensuring that the process is used as intended. A natural candidate for this role is a person expert in agile methodologies, belonging to the unit which is in charge of the software engineering process (University of Cagliari).

Unit Coordinator: This is a new role, specific of a distributed research project. It is a person who coordinates the Team Members belonging to a research unit and controls that the artifacts delivered by the unit are in line with the required deliverables. The Unit Coordinators, together with the Project Owner, decide the feature prioritisation.

Unit Members: The people working on the project. Each member belong to a Research Unit.

Research Unit: A group of people working in the same location, including the Unit Coordinator. Each unit has specific duties in the project, being responsible for features and deliverables. One or more teams of between three and six people each are used. Each is assigned a set of packets (deliverables), including all backlog items related to each packet. The team defines changes required to implement the backlog item in the packets, and manages all problems regarding the changes.

The main artifacts of EURACE Scrum are *Project Backlog*, *Sprint Backlog*, *Impediment List*, *Project Increment* and *Burndown Graphs*.

The *Project Backlog* is a prioritized list of project requirements with estimated times to turn them into parts of project deliverables and/or completed system functionality and the Sprint Backlog a list of tasks that defines a team's work for a Sprint.

Moreover we have a *Impediment List* of anything around a Scrum project that impedes its productivity and quality.

At the end of every Sprint the team should have delivered a production quality increment of the deliverables or of the system, demonstrated to the Project Owner and, at each project review, to external reviewers (*Project Increment*).

Finally the *Burndown Graphs* show the trend of work remaining across time in a Sprint, a release or the whole project. Other reports on the project status, the bug status, etc. can be necessary.

How does Eurace Scrum work?

Eurace Scrum was designed to allow average participants to self-organize into high performance teams. The Eurace Scrum process is based on specification of what has to be done through a list of features; each feature is assigned an estimated

value for the project, an estimated effort, a responsible. Project advancement occurs using timeboxed iterations called Sprints. Sprint duration is from 4 to 6 weeks. Each Sprint (iteration) implements a the subset of the Project Backlog, in other words a list of features prioritized according to feature's value, effort and risk. This is done by the Project Owner together with Coordinators. Finally Sprints are grouped in Releases. A Release covers one year of works, and delivers project official deliverables. At the beginning of each Sprint, a Sprint Planning Meeting decides what features to implement in the Sprint, and decomposes them into the Tasks needed to implement the feature. Tasks for a sprint must be well quantified and assigned to one individual, and if the task is shared it is strategic to give one person the primary responsibility. The Sprint Planning Meeting is held through the Internet or, when possible through a physical meeting. During the meeting the team members synchronize their work and progress and report any impediments. Moreover the Daily scrum is done via instant messaging. The Sprint Review meeting provides an inspection of project progress at the end of every Sprint and the Sprint Retrospective meeting is held after each Sprint to discuss the just concluded Sprint and to improve the process itself. The Sprint Review and Sprint Retrospective Meetings are held through the Internet or, when possible through a physical meeting. They typically precede the Sprint Planning Meeting deciding about the next Sprint.

Distributed Eurace Scrum

In a research project as Eurace all participants are fully distributed and each team has members at multiple locations. We consider three distributed Scrum models commonly observed in practice: *Isolated Scrums*, *Distributed Scrum of Scrums*, *Totally Integrated Scrums* [53].

In *Isolated Scrums* the Teams are isolated across geographies. In most cases offshore teams are not cross-functional and may not be using the Scrum process.

In *Distributed Scrum of Scrums the Scrum* teams are isolated across geographies

and integrated by a Scrum of Scrums that meets regularly across geographies.

Finally in *Totally Integrated Scrums* the Scrum teams are cross-functional with members distributed across geographies.

In Eurace Scrum project the best practice is a Distributed Scrum of Scrums model. Scrum Masters are all in Germany (Universitaet Bielefeld), France (Université de la Méditerranée), Turkey (National Research Institute of Electronics and Cryptology– TUBITAK UEKAE), Italy (Università di Genova, Università di Cagliari, UPM in Ancona), UK (University of Sheffield and the Council for the Central Laboratory of the Research Councils) and one affiliate member of the Columbia University in New York City (US). The project is coordinated across teams, in fact Scrum teams are linked by a Scrum-of-Scrums where all Unit Coordinators gather up to talk via instant messaging and teleconference. Before the Scrum meeting, each individual reports, by means of email exchange, on what they did since the last meeting, what they intend to do next, and what impediments are blocking their progress, in order to proceed the meeting more smoothly and efficiently [53]. The top issues in distributed development researched by SSCI [23], Strategic, Project and Process Management, Communication, Cultural, Technical, Security, are resolved in Eurace Scrum project. There is a mechanism of effective communication that allows to overcome also the cultural conflicts, the work is synchronized between distributed sites, the data formats are standard and compatible. Moreover some people are not very good at planning their workload, so Sprint goals are an effective tools for keeping people on track.

4.3.3 Considerations

We've found that Scrum is appropriate for research projects in which all teams consist of participants from multiple sites. This requires an adaptation of Scrum along with best practices described above (e.g.: daily Scrum team meeting of all participants from multiple sites, etc.). We think Scrum provides a better project

management than the conventional management theory [9]. Everything is visible to everyone, team communication improves, a culture is created where everyone expects the project to succeed. A disadvantages maybe is that Scrum requires constant monitoring both quantitatively and qualitatively, moreover the introduction of a new and different methodology sometimes is view with reticence by the people resisting to change.

Chapter 5

Adopting and Adapting a Distributed Agile Methodology

In the world of software development today there are different examples and scenarios of distributed development, and in the last years, researchers have been trying to design processes that improve the efficiency and quality of distributed development, adapting agile methodologies to this context.

The scattering of team members and functions around the world can introduce barriers to productivity: time zone differences can wreak havoc on project schedules, cultural and language differences can lead to misunderstanding of requirements. The failure of distributed project teams to work together effectively results in extra hours, schedule delays, cost overruns, and sometimes even outright failure of the project.

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 necessary to work like a team. Moreover, in some scenarios, a problem is the continuous changing among team members, that constantly alternate on-site and off-site activity. Cultural and geographical distance tend to reduce and complicate communication and feedback.

In this chapter we propose and describe an agile methodology for distributed development. Is there a best way to adopt Agile methodologies in distributed processes? Many of the practices espoused by Agile methodologies can be readily applied by any development team and then also by any distributed team. A team can be agile without necessarily being Agile. As before we mentioned the organizational factors that determine the success or failure of any software development process are complex. There are relationship and cultural aspects, or also physical distance, to consider.

In particular, we illustrate a set of best practices to apply in distributed and agile contexts, chosen on the basis of their impact software quality and team cooperation.

Before attempting to extend agile practices to a distributed working environment, it is certainly 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 needs 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 geographic distribution, and how to replace those invalidated or evidently inapplicable.

The proposed Distributed Agile Methodology (DAM) adopts some practices and guidelines that are of general validity and can be applied in different contexts of dispersion. DAM has been unwrapped on the basis of the MAPS project, from this we started [2].

Finally a landscape overview of major tools and techniques for managing and executing software development projects across geographically and temporally distributed teams is provided.

5.1 Method

There are two different approaches to investigate the applicability and effectiveness of a software methodology: *empirical studies* and *simulation process modeling*.

The *Simulation model* generalizes empirical studies and provides a framework for the evaluation of empirical models. On the other hand, *empirical studies* provide the necessary fundament for simulation models because through empirical studies it is possible to collect real data to validate the simulation model.

In the present research these two approaches are combined.

In chapter 4 we showed three case study carried out by our research group. These experiences has been valuated on the basis of empirical studies and a simulation model [56] to formalize the gathered data and define a *Distributed Agile Methodology*.

DAM has been defined applying the *PPT (People - Process- Technology)* model: one of the more convenient ways for conceptualizing the impact of organizational factors that determine the success or failure of any software development.

In general an improvement of productivity in the software development is driven by three key factors: involved *People*, organizational development *Process*, and used *Technology*. People are very important factor in any software organization and the results of successful project are better guarantied from a structured process in which

the cooperation among all involved people is possible. Finally, also the more advanced technology cannot be fully effectively or to make matters worse the process, if an organizational framework doesn't support it.

People. The "people" dimension is perhaps the most difficult to manage in a distributed project. Physical distances and cultural differences among members of a project team are the cause of schedule delays, cost overruns, and sometimes even outright failure of the project. Software engineering is done "of the people, by the people and for the people." That is people organize themselves into teams to develop mutually satisfactory software systems, they identify what software capabilities they need, and other people develop it for them, finally people pay the bills for software development and use the resulting products.

Process. Standardized processes are important for any software development project, and most were developed without distributed teams in mind. Most traditional software development methodologies have no specific provisions for distributed software development teams and Agile methodologies specifically require co-located teams. When teams need to reengineer or modify processes, all problems related to difficulties coordinating distributed teams emerge.

Technology. At the moment a distributed team must use tools designed to support co-located processes and so it is necessary to adapt them.

Within this context we have defined three macro-practices, in order to propose the *best-practices* for implementing the agile methods in a distributed process:

1. ***Identify and Manage Key resources*** (either people or technology) in order to make friendly the distributed process;

2. ***Standardize processes for distributed development*** in order to provide many *"best practices"* that can help the team to understand all development process.

3. **Use Tools designed for distributed development** in order to improve the communication and team relationships.

DAM has been designed on the basis of a common structure of the Agile methodologies. Methods for agile software development constitute a set of software development practices that have common characteristics.

The DAM has been defined on four main activities:

1. Study of existent theory;

2. Study of several case studies;

3. Data Gathering;

4. Data Analysis;

5. Generalization of the knowledge.

Study of existent theory

In chapter 1, 2 and 3 I reported a brief overview on an "holistic" study carried out in all these years of my PhD course in order to achieving a deep understanding of the problem and of its meaning in analyzed context.

Study of several case studies

In chapter 4 I described some case studies and related works which have been conducted studying real projects. The studies of comparison described in this thesis are largely qualitative, primarily because the lightweight nature of agile methods excludes much gathering of project data. But there are relevant data and also useful

in determining under which project conditions one should use a practice rather than one other and what kind of results one should expect.

Data Gathering

Measures are important data in all engineering disciplines. These data allow engineers to understand how things work and how to make changes to produce desired results.

Measure related to gathered data can be traditional or agile. The data, for each single practice, have been gathered using GQM approach, discussed in Section 1.5. Time values and the use of a practice are an example of quantitative data. In XP the code is its documentation, so quantitative data are gathered from the code and user stories. Moreover code releases are easily tracked by means of code collective ownership. A user stories is the smallest amount of information necessary to allow the customer to define a path through the system.

Tools allow data acquisition but they must also provide any advanced and integrated analysis. A completely automated acquisition tool should be able to both improve data quality and reduce to zero the acquisition effort spent by programmers. Agile philosophy advices using automated tools which allow developers to collect process data and analyze them automatically. The tools allow a multi-channel interaction with process data both via Web pages and by means of rich user interfaces directly embedded in the IDE. Free tools, lightweight and completely based on open-source components and standard protocols are used in the software development with agile methodologies.

We use *XPSwiki* [1]. It was developed to meet the concrete need of software companies to collect requirements and schedules in an electronic format. Managers, customers and developers can access XPSwiki by means of a Web browser. The immediate advantages of a Web based tool for the management of process data can

[1] http://www.agilexp.org/xpswiki

be summarized as follows:

- the tracker and the developers can insert effort data directly from the Web without handling paper sheets;
- dispersed teams can cooperate and share a common view of the whole project status;
- user stories, task assignments, and any relevant effort data is immediately available on the Web, directly from the workstation of a programmer/couple.

Qualitative analysis is more difficult to tackle. Data are subjective and they are related to the motivation and participation of the interviewee. In general qualitative data are gathered by means of questionnaires and direct interviews (the questionnaire is useful, for example, to gain the developer perspectives on its satisfaction level to use a practice [38]). Moreover because of their subjective nature, qualitative data cannot be gathered in automatical ways or continuously, but they are gathered on the basis of deadlines. For example, since an XP iteration is not more long than four weeks, you can have a good number of qualitative data if you foresee a gather at the beginning and at the end of each iteration. A continuous presence of the customer helps in the analysis of the customer satisfaction, main example of qualitative data.

Data Analysis

We have considered two kind of analysis: *analysis of the practices* and *analysis of the common features*. The goals are different. The former should identify analogies and differences between agile methodologies and distributed processes, the latter should allow the study of common data between them.

Generalization of results

Usual procedure for advancing from distinct case studies to generally valid knowledge is based on the fact that most case study researchers in reality have quite a lot of knowledge of other comparable cases already when they start their study. This generally valid knowledge then permeates their case study in the form of concepts, models and variables that relate with the other cases. All this means that it becomes easier to generalize the findings of case study observed. This has been done. Moreover the comparison is a fundamental technique allows to validate the results. We have compared iterations that belong to the same project, or projects with common features and same agile practices, or among projects use same agile methodology.

These steps have allow to pull out DAM, agile practices and guidelines that can be applied in a distributed environment.

5.2 Identify and Manage key resources

5.2.1 Role

In distributed teams, it is important that all team members know their roles in relation to each other, and how the goals, priorities, and tasks are set in the project. This *basic knowledge* equips team members to understand directives they receive, prioritize work, and resolve inevitable conflicts that arise from competing demands for their attention. The idea is not that distributed teams should follow a particular organization or governance structure. The important point is that all team members understand how the project is organized, and how their role is expected to interact with the others. Maintaining consistency in team structures and management structures avoids confusion, which is easily sowed in a distributed team, and very, very difficult to stamp out.

Whole Team

The team must be composed of members with all the skills and the perspectives needed for the project to succeed. They must have a strong sense of belonging, and must help each others.

Travelers: On-Site Visit

We propose the *On-site Visit* practice. In [25], Martin Fowler describes a very effective approach to promoting face-to-face communication with a team distributed between the United States and India: they have decided to ensure that at all times there was someone from the US team present in India to facilitate the communication. We call this "someone": traveler.

In [1], a "traveler" is someone who moves back and forth between locations, perhaps spending a certain amount of their time at "home" and portions at other locations, to facilitate communication between the groups. This is a role often taken on by people who have a high-level view of the project. M. Fowler [25] has found useful to send a US developer and a US analyst to India to communicate on both technical and requirements levels, and to send someone from India to the US team. The plane fares soon repay themselves in improved communication. Moreover Fowler recognizes that important trust relationships are built very quickly with face-to-face interaction, which results in improved productivity and effectiveness across the distributed team. Having a subject matter expert on-site and accessible to the entire offshore team provides an immediate payoff in the form of rapid knowledge transfer. Team members are able to ask and receive answers to questions in a much more interactive and rich context, allowing them to more easily explore a topic in minute detail. The entire India-based team was able to build up a good understanding of requirements in a relatively short amount of time.

Customer on call

Customer on call is a reinterpretation of the On-Site Customer practice. When the customer is not present for several reason (distributed development is one) then a representative customer must be available via mobile phone, or on line.

5.2.2 Communication

Each phase of the software development process comprises different communication flows and challenges. In distributed software development normal communication lags or delays are compounded by time zone differences and the simple fact that people are easier to ignore when they're not physically standing next to you. Meetings are harder to conduct because it's difficult to tell if the people on the other end of the conference line are nodding their heads or looking befuddled. Scheduling meetings become more difficult due to limited overlapping work hours. Server outages may waste entire workdays, as one team waits for a support person in another time zone to wake up and drive to work to resolve the problem. Therefore Communication is one of the critical points to face up to each main phase of the software development process (Requirements, Design, Development, Testing, Deployment) with success. The different methodologies provide different frameworks for communicating through each of these phases.

Many traditional software development methodologies create formal documentation as a matter of course: use cases, functional specifications, lists of requirements, business rule descriptions, data dictionaries and validation rules, nonfunctional or technical requirements, interface requirements, user interface mockups, architecture diagrams, data models, etc. Distributed teams rely more heavily on these artifacts for communication. It thus becomes imperative that the documents are clear, easily accessible, and current.

Development relies heavily on the artifacts created in previous activities, but is also much more reliant on conversational communications, as developers create and

test code while reacting to inevitable changes in requirements and design. The principal activity in this phase centers on the developers and the creation of source code that implements the features described in the requirements and design documents. In a distributed team, the challenge will be to integrate code from different team members, and keep all the developers abreast of changes as new modules are added, existing modules are refactored, and outdated code is eliminated. At the same time, the requirements and design documents must be kept updated, and relevant changes broadcast to concerned team members. This is the phase where changes happen very quickly and affect the most people, so transparency, dialogue, and ease of communications are absolutely crucial.

The communication model in the testing phase is perhaps the most mature, with a wide variety of defect tracking tools available to manage the test-fix-retest cycle. Dialog is still important, since the team needs to understand what's wrong and how it should be fixed, but changes to existing documentation should be minimal. Upon deployment, ongoing support and maintenance of an application requires even more attention to communication and coordination. There will be additional enhancements and bug fixes to coordinate, and perhaps even major new development efforts for the next application release. Coordination across the different teams, projects, perhaps even organizations is required to ensure changes are not lost, and that all team members understand the implications of their efforts on others.

Communication is one of the critical points in defining a Distributed Agile Methodology, being moreover one of the main principles of agile methodologies. So, practices such as *pair programming* (for the knowledge sharing) and to eliminate heavily documents in favor to a *face-to-face communication* seem only complicate the distributed development because of physical barriers (oceans, languages, cultures, etc.). The greatest challenge here is to superare these barriers.

In order to solve distance problems and to maximize the effectiveness of conversation, tools and practices are useful to establish interpersonal relationships (like

complicity and confidence, in order to gain trust and know the different work approaches of team members) and to allow a common vision of the project and of the required functionalities. Teams have multiple channels, from e-mail lists to bulletin boards and so on, available to them, so they can choose the most appropriate medium for communicating ideas. Communication becomes less and less effective as the tools become less expressive and physical and temporal proximity of the participants become greater and greater.

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

- *Telephone.* The telephone is still one of the most effective tools for communicating across distances. *Conference calls* or, if available, *videoconferencing* should be often used. Regular weekly conference calls can help distant teams keep connected and feel engaged, as so long as the participants in the call have a turn to speak and be heard. Phone calls between individual team members should be also used liberally. The main challenge to this medium is that person-to-person phone calls can be difficult or awkward to initiate when people are unfamiliar with each other. It is a natural human reaction to be reluctant to call an unfamiliar person on the phone. Managers and project leads should be proactive in making introductions between team members and build familiarity through group activities such as conference calls and Web meetings. Posting each team member's picture on a project Web site is also helpful for breaking the ice and helping team members build familiarity with each other. Conference calls need to be well-organized, they need to adhere to a specific agenda, and they should limit themselves to a set time frame. Conference call facilitators should also encourage participants to engage in the conversation by asking questions and soliciting opinions, while preventing individuals from hijacking conversations with long-winded speeches.

- *Web Meeting* are a good tool for interactive work sessions. Most Web conferencing tools provide a virtual whiteboard for brainstorming, which helps participants visualize problems and communicate complex concepts more clearly;

- *Weekly meetings* conducted by Web meeting or conference call, help distributed teams work better in many ways. First, they keep all team members apprised of project news and status, allowing them to better understand how the project is faring and how their work affects others. Second, they build community among the team by helping participants feel like they are part of an organized effort working toward a common goal. Third, they provide an opportunity to reinforce key concepts and important points by discussing them as a group and forming communal knowledge, rather than relying on knowledge to percolate to the group via individual communications.

- The use of a *dictionary*, which defines for the team members *userID* and *roles*, by means of which it is possible to apply some rules to emails sent by the community members, classifying them by the object;

- *Instant Messaging* (IM) is one of the most useful tools for collaborative teams. IM is immediate and perfectly suited for short, ad hoc conversations. It is less disruptive than phone calls or meetings, since participants can stay at their computer without breaking their focus on ongoing tasks. IM clients also broadcast each team members' presence to the rest of the team, indicating if they are at their computer or if they are away, allowing others to quickly determine if they are available for consultation. In order to guarantee immediacy in the communication it is a good practice to choose the same userID used for the mail. 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 is an other rule, in order to share with all the team knowledge and decisions useful for the project. You can find a list of IMs in appendix A.

- The use of a *mailing list* to send messages to all the team members, both developers and customer, using the following rules:
  - "ReplyTo" 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;
- *E-mail* is still one of the main channels of coordination for distributed projects. E-mail's advantages are that it is easily targeted to specific individuals; it can be easily replicated for off-line access; and it allows easy tracking of the history of a conversation. The disadvantages of e-mail are that it is difficult to archive and search for general teams if communication rules have not been imposed; messages are easily lost if not well managed.
- 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.
  - Using systematically the *commit message*, and specifying in it userID and role, developers always know who added or created something;
  - With distributed teams separated by large time zone differences, interactive dialog can be difficult to coordinate. *Wikis* can help make these drawn-out conversations more manageable. A wiki (see also appendix A is a tool for creating project information pages, in order to share via web documents or information about the project, only by editing text that can be immediately viewed by everyone. Wikis work well because they are simple to use, can be worked with any browser, and are simple to set

up, moreover they are by nature unstructured, and this lack of structure is part of the benefit.

– *Web sites* are also key communication channels, especially for storing long-lived information such as reference documents, status reports, tutorials, architecture and design documents, and other artifacts. These channels allow team members to search for project information in a structured medium and so they allow a community management.

The main challenge to using *Web sites* and *wikis* effectively is in keeping information well-organized, up-to-date, and "alive." A site that becomes stale or difficult to navigate loses most of its utility. Project leaders should plan a strategy for organization project documents and artifacts so they are easy to locate. The site should also contain regularly updated content to encourage team members to consult it regularly. A good strategy is to post project updates, metrics, and links to important project documents to the site's main page regularly, thus providing team members with a specific incentive to visit the site regularly.

5.3 Standardize Processes

Standardized processes are important for coordinating distributed teams. Only the use of same procedures allow all team members to understand the plan, their role in the plan, and how they affect others in the plan. When people located at different sites can be confident that their colleagues at other sites follow the same procedures, not only are they able to collaborate more effectively, but also they are able to discuss process issues with the same baseline of knowledge. There are many best practices that can be put into distributed process, such as standards for code comments and source code commits, test-driven development practices, and continuous integration, all of which promote communication and foster early detection of defects. All these

practices work in synergy and some of their may support different phases of the software lifecycle.

5.3.1 Planning-Design

Planning of activities is an 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, through user 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.

The ***Planning Game*** practice is a vital point of interaction between the customer and the developer.

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.

Periodic releases, flexible in the contents but rigid in the date, are necessary. Different versions of the software must be 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.

In general agile methods use shorter iterations than a lot of other iterative approaches. The experience at ThoughtWorks project reported by Fowler [25], in which almost all projects use iterations of one or two weeks in length, shows which short iterations are much better.

The functionalities of the system should be described using **User Stories**. In the case of a distributed team the User Stories are not paper index card but electronic index card (for example they can be managed by an automatic tool, e.g. XPSwiki) on which must be written brief feature requests.

One of the best ways to communicate frequently is to *communicate regularly.*

Establishing a formal schedule is the best to inject discipline into the process of communication. Regular team meetings and conference calls are a must. Meetings between sub-teams whether horizontally focused, such as developers and testers, or vertically focused, such as teams building a particular subsystem, should be also conducted. Individuals may also schedule daily calls to key counterparts in other offices to keep coordinated. A planning meeting at the beginning of each iteration that involves the whole team really helps to get everyone coordinated on the next chunk of work. Moreover 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.

Agile methods promote regular short status meetings for the entire team (Scrums in Scrum, stand up meetings in XP). Carrying this over to remote groups is important, so they can coordinate with other teams. Time zones are often the biggest problem. Fowler [25] found that twice a week stand-ups seemed to work well and provide enough coordination.

The following practices are only some general criteria of the planning phase, which we have identified as indispensable:

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 from all team members. A planning tool like XPSwiki 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. As shown in the MAD project (see section 4.1) by using an automatic tool (e.g. XPSwiki) each developer can understand which user stories are done, in progress or to do in the weekly cycle.

Informative Workspace in an another useful practice that should be adopted in the distributed software development. This practice allows the team to improve the communication and on the basis of our experience in the MAD project the use of an automatic tool such XPSwiki is always advisable.

Finally **Feature List and Build by Feature** is an another best practice that we propose. This practice derives also from our experience in DART project (see section 4.2): a list of the features to implement, ordered by their relevance and kept in a Wiki.

Refactoring is the continual effort to simplify the code while still ensuring all tests pass, that is, cleaning the code and design without changing functionalities. This goal is achieved by small change steps, verifying tests after each and using refactoring tools (available in some IDEs) that make large-scale changes easier and faster.

5.3.2 Coding

The coding phase is still more critical if developers work in a distributed team. Adhering to *coding standards* in order to increase code readability, 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.

Simple design is an essential practice to avoid duplicate code, have a relatively minimal set of classes and methods, and be easily comprehensible. In the experiences here reported we have demonstrated that the software quality increases adopting a simple design. Also in Jakarta project (see Section 3.1.1) we saw that developers adopt an effective design, very simple.

Code reviews are an important technique for producing quality software, and is a standard practice for many software development teams. There are many problems with conducting effective code reviews in a distributed team. One of some barriers for conducting code reviews for distributed teams, however, is the frequent logistical impracticality of convening both authors and reviewers to conduct a code review in the first place.

Code authors must assemble all the files for review, highlight all the code that they have added, removed, or changed from previous versions, and package them for reviewers. For wide-ranging or complex code changes, some guide or rationale for the changes must also be provided to help reviewers understand the changes in context. All of this requires no small amount of time and effort for preparation, documentation, assembly and distribution. Code reviews are typically conducted in a meeting room where all reviewers and authors can gather to discuss the changes. All parties are expected to have reviewed and familiarized themselves with the changes beforehand. Differences of opinion and discussion of the merits of different approaches can often lead reviews astray and spawn additional meetings to resolve differences. For

highly distributed teams, code reviews may mean that some team members make radical adjustments to their work hours so that they can make an early morning or late night conference call. Frequently, Web meetings must be scheduled in addition to conference calls so that all parties can walk through the code simultaneously.

Some examples of coding standard you can find in the Sun's site[2], also if it concerns only Java, but some common standards can be adaptable according to need.

An other good practice improving the code is that each *class and each method should be commented* in order to provide easily a knowledge of functionality.

But if the code is not robust, all the rules described are not so useful: a such system could be not 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.

The **continuous integration** allows the incremental development of the software and makes easier for developers to integrate changes of the project. The basic idea is that the code created by all the developers on the team is regularly compiled and tested to make sure that all the code changes are compatible.
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. This practice has been adopted with success both in the MAD project (Sec. 4.1) and in the DART project (Sec. 4.2.2). Continuous integration in XP is supported by the practices of Pair Programming, refactoring, collective

[2]URL: http://java.sun.com/docs/codeconv/

ownership, testing and coding standards. In order to automate such activity it is possible to use some scripts (for example ANT [3]), that allow to specify a series of operations in a systematic way.

Using a tool like CVS [4] makes continuous integration easy. CVS merges your local changes into the repository version, updating your local copy, but leaving the repository unchanged. You can work confidently knowing that you are always working against the most recent version of the system.

The build should be working at all times. If someone breaks the build, they are responsible for fixing the problem immediately. Run the build often, preferably after every check-in to the code repository.

By definition, **Pair Programming** requires programmers to work on the same computer. But the trend towards distributed development needs programmers on the same team work at different sites, even on different continents. Recent researches find that pair programmers produce code with higher quality without sacrifice of productivity and have more job satisfaction [19]. Moreover as shown in FlossAr case study (see Section 4.2.2), when the development phase (phase 3) is characterized by strong pressure for releasing new features and by a minimal adoption of pair programming, we observed a growth both in the coupling metrics and in the local metrics, indicating that in this phase the quality has been sacrificed.
To enjoy the benefit of Pair Programming, distributed team members need to work as if they were sitting next to each other. During the second phase of the MAD project (Section 4.1) we used some techniques to support distributed pair programming and these guaranteed a good communication level. In appendix A we propose some tools to support the distributed pair programming (see also Section 5.4.3). These tools provide functionalities such as application sharing and/or synchronous

[3]URL: http://ant.apache.org
[4]URL: http://www.cvshome.org/

communication (textual and vocal chat). These tools have been tested during MAPS project and other our experiences. Finally in a OS project this practice is not applicable in its strict form but those same goals are achieved in other way. In Jakarta project (see Section 3.1.1) the code is frequently reviewed because the code is public and all of the contributors can suggest improvements and detect potential bugs or report bad smells in code. Knowledge transfer is less immediate, because only the mailing lists are used to communicate. Then with a virtual collaboration environment distributed developers can work as if they were setting together, using the same input devices and looking at the same monitor.

5.3.3 Testing

The **Testing** practice is useful to verify and to certify the correctness of a class. 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.

In XP approach you write a test before you write the code you are testing. This is the *test-driven development* practice that follows this philosophy: "test first, test often." The focus is on building automated unit tests which can be executed repeatedly against the code. By creating tests before code, developers establish a baseline of expected functionality. As long as the code passes the tests, they can be reasonably assured that the system meets the expected requirements. Many studies have been conducted in order to investigate the benefits of TDD in terms of code quality and team productivity. These studies have been reported in section 3.3. In our simulation experiment (we reported the results in section 3.3.3) we found that defect density decreased by 40 % in the case of TDD application. Also in

FlossAr project (see section 4.2.2) we found that when the development phase is characterized by a minimal adoption of testing and refactoring the code quality is compromised.

Automated tests help make sure code works. This is the best way to make sure your team creates quality code, moreover they help support refactoring of code, providing a means of ensuring that bugs are not inadvertently introduced. There are cppUnit, perlUnit, JUnit, and so on, collectively know as xUnit. The port getting most of the attention is jUnit, which is the port for the Java programming language.

5.3.4 Concrete Feedback

Agile methodologies were born with the need to regulate and reduce the cost of frequent changes throughout the software development process, therefore not only frequent iterations but also regular feedback is necessary in order to provide it: 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 coverage of the developed code in order to have time for changes or to avoid too high cost of changes. The feedback must give a real sense of software itself. With XP the feedback is taken early and it is based on metrics that describe the state of system

The distribution of the team could limit the communication and all the other agile practices in which feedback is required, like Pair Programming, Customer On-Site or the Collective Code Ownership. In a distributed scenario, like the OS one, sometimes people know only a part of the module they are developing and is not always simple to have a common vision of the whole project.

The following practices could help a distributed team in immediate receiving feedback:

Short Build cycles. Build cycles must be very short, so the customer can see the exact status of the software itself (rather than documents, reports and other

deliverables that are simply reflections, the customer can see unit tests passed or user stories completed).

Collaborative Management. Developers could apply a sort of distributed pair programming, choosing another developer as a reviewer of its developed code.

Feedback given by the customer. Every time developers release a new version, the customer should release a feedback document. Without the feedback of the customer the ability to deliver the system becomes trial and error. Technology that facilitates high level of communication must be available in order to allow customers to be part of the development process.

Unit Testing for all the developed code. Tests should cover all the code and be documented in order to emphasize their function and the parts of the code covered by them. Moreover Unit Test should be written in order to emphasize bugs.

Use of automatic tools for all useful but boring activities. Activities like collection of metrics, test covering and naming.

5.4 Technology: Agile Tools

Many tools exist that can be used to support distributed development. These are collaborative modeling tools, collaborative writing tools, conferencing tools, virtual meeting tools, and so on. Most of these tools used in XP software development are either free or inexpensive.

Automated builds provide teams with comprehensive up-to-date status of their project, what tasks have been completed, what tasks remain, whether the current work meets test requirements, and whether all the moving parts are working together properly. Any member of the team can get this information at any time, greatly

simplifying communications and ensuring that everyone understands the state of the project.

In addition to adopting and adapting existing tools and techniques, the open source movement has spawned an entirely new segment of software tools specifically designed to facilitate communication and coordination among distributed software teams. Originally written to host open source communities on the Internet, these collaborative development tools are now being packaged and marketed to enterprise software development teams, offering a wide range of features ranging from issue tracking to knowledge sharing, to source code management. These tools are now offered by a host of major software companies, from CollabNet and SourceForge, started by one of the founders of the open source Apache Web server project, to Microsoft, frequently reviled as the great bastion of proprietary software.

We will examine these collaborative development platforms, and compare and contrast different offerings to understand how they help distributed teams communicate and cooperate.

Distributed teams need access to systems at all hours, and increased distance and time between team members means more infrastructure and coordination is required to establish connections for communication.

5.4.1 Integrated Development Tools

Integrated Development Systems is the main work environment for a developer. An *Integrated Development Environment* (abbreviated as IDE) is a programming environment that has been packaged as an application program, typically consisting of a code editor, a compiler, a debugger, and a graphical user interface (GUI) builder. The IDE may be a standalone application or may be included as part of one or more existing and compatible applications. IDEs provide a user-friendly framework for many modern programming languages, such as Visula Basic, Java and PowerBuilder. IDEs for developing HTML applications are among the most commonly used.

5.4.2 Tools for Synchronous and Asynchronous Communication

Synchronous communication (SC) can be defined as real-time communication that occurs between two or more people through a virtual or electronically mediated system. SC systems have become more prevalent as high-speed network connections has increased and become both more readily available and affordable. Synchronous communication tools are interactive programs that allow two or more users to communicate by means of textual, audio, video messages or graphics messages, such as virtual whiteboard for brainstorming. Instant Messaging tools (IMs) show team member who is online and allow a real time communication.

Asynchronous communication tools allow more users to exchange information in not interactive ways, that is they don't keep busy who communicates in order to wait a response. An example of asynchronous communication is the use of a blackboard where more users can "hang" messages. These tools suffer from real communication but they don't require all users take part in the project simultaneously. Moreover, typically all messages are stored and available to the next consultation.

We report in appendix A the tools that support synchronous and asynchronous communication.

5.4.3 Tools for Distributed Pair Programming

Pair Programming is one of the practices promoted by Extreme Programming. Pair programming is where two developers share the same machine and work side-by-side to solve and produce code. One developer (driver) types at the keyboard while their partner (observer) reviews, analyzes, talks, helps, prompts, and thinks. The knowledge is shared equally across the team. A valuable benefit to pair programming is that the observer can take care of any logging or record keeping while the driver continues to work.

For its nature a practice such as pair programming doesn't seem proposable for globally dispersed teams. As discussed, many of the key principles and practices of the Agile methodologies can be adapted by distributed development teams in the Free/Open Source world.

In appendix A we report the tools, which we have tested in our case studies, useful to adapt this practice in distributed development.

Pair programming requires synchronous communication, therefore, it can't rely on the use of disconnected tools such as email. Real-time collaboration tools, such as instant messaging or chat applications are much more helpful. A dispersed development team must have high-speed network connection to offset their lack of verbal communication.

In our case studies tools allow real-time communication between developers, such as Instant Messaging (MSN Messenger, Yahoo!Messenger, ICQ, AOL Chat, and so on), have shown more advantageous: they are usually, free, easy to learn and use, moreover they show team members who is online.

Audio supporting should be used without to involve hands using in order to allow, at the same time, code writing or messages via chat. For this reason, tools of audio communication with walkie-talkie mode have shown uncomfortable.

Moreover when two developers know each other the use of a webcam is not necessary but it becomes useful when developers don't know one other in order to give them a face.

In order to ease distributed pair programming (DPP) it's need to exclude dial-up connections, since too slow to allow the exchange of audio and video data. Then high-speed network connections are essential.

Among tools tested in our case studies, *NetMeeting* [5] and *VNC* [6] seem unsuitable for the Pair Programming. NetMeeting has been created for videoconferencing, and VNC for remote desktop sharing.

NetMeeting is very more comfortable than VNC because it includes in a one tool the audio supporting, video supporting, chat supporting, desktop sharing and application sharing and moreover it manages the request for desktop control: a functionality not present in VNC.

Among the IMs, *AIM* (for all platforms) and *MSN* (only for Microsoft Windows) seem enough efficient: they are clear in the graphic and to use, they haven't inconvenient banners in the chat windows (unlike ICQ) and they are good tools in audio communication, where the reception and broadcasting are contemporaneous (and not in walkie-talkie mode, such as PalTalk).

On the basis of the tests we have carried out, an ideal and suitable configuration can include the use of AIM or MSN for audio and text chat, simultaneously to NetMeeting or VNC for desktop and applications sharing.

An other useful tool for DPP is Sangam [7] for Eclipse: a plug-in for synchronous, distributed collaborative programming, that is, several developers see the same piece of source code (driver is writing, navigators are watching). Usually it used together with VoIP chat program.

5.4.4 Build Tools

A build is the heartbeat of development and is the best measure of the state of the system, moreover it assures the system quality by passing tests. It is the built of a system, that is a sequence of procedures to make the code working and usable.

Martin Fowler outlines, among the practices used at Thoughtworks, the following:

[5] http://www.microsoft.com/windows/netmeeting/default.asp.
[6] http://realvnc.com/vnc/index.html
[7] http://sangam.sourceforge.net/plugin

"Always have a successful build available". A successful build is the result of the following:

- all sources are checked out of version control (ideally, the latest version of every file);
- everything is compiled from scratch;
- the system is linked and deployed (that is, put into jar files or ear files for Java development);
- the system is started and the full test suite is run without intervention or failure.

There are many approaches to build automation, such as Ant [8] or Nant [9]. For flexibility and power, XP developers tend to use Ant as their build tool of choice. Ant is a Java-specific replacement for make extensible in the Java programming language. It is a product of the Apache groups's Jakarta project. Moreover it offers integration with JUnit and can be used to run post build smoke tests. Nant is modeled closely on Ant, the difference is that Nant is written for the Miscrosoft.NET framework and as such does not require Java.

5.4.5 Continuous Integration Tools

Continuous Integration is one of the practices of XP. This practice is used to integrate the functionalities developed in independent way by several developers in the same project. "Continuous" means that you should integrating often, in particular after each task you complete. In this way, the chance of your changes conflicting with someone else's is minimized. Using a tool like CVS makes this easy. At any time you can freshen your working files from the repository. CVS merges local changes into the repository version, updating your local copy.

[8] http://ant.apache.org
[9] http://nant.sourceforge.net/

CruiseControl (CC) is an automated monitoring tool downloadable from SourceForge.net [10](it is open source and free). It works in conjunction with ANT and your source control system to automate the update-build-test cycle. CC runs in the background, performing a cycle at a predefined frequency (e.g. every 10 minutes). To run Cruise Control you will need Java, Ant, a Web server, and some kind of servlet engine such as Tomcat. As soon as the build or a test fails, you are alerted. CC ensures that you are always working against the most recently released sources. It alerts you as soon as there is a problem. You can stop immediately and correct the problem.

5.4.6 Source Control Tools

Storing all application source code into some kind of repository is absolutely vital. The repository should be able to store any kind of file type used and should support versioning of changes, moreover these Source Control Tools allow the source code configuration management, including versioning and change control. All developers working on the code will interact with these tools on a regular basis, whether committing changes or simply refreshing their working copy of the code with the latest version. All development teams benefit from disciplined use of these tools.

The most popular open source control system is CVS (Concurrent Versions System) [11], but there are other tools commonly used like Visual Source Safe, Clear Case and PVCS.

Because a software development project lives and dies by the source code it generates, choosing a SCM (Source Code Management or Software Configuration Management) tool that meets the unique needs of the team is absolutely critical.

CVS is recommended for small development teams, well organized and always in contact with them, or for projects that don't imply frequently releases. Moreover,

[10]http://cruisecontrol.sourceforge.net/
[11]http://www.cvshome.org/

because CVS allows to update only changed files, it can be used also with slow connections. Check-in or commit changes as often as possible to ensure the team has the latest working source, but info can be lose. It is convenient provide for this case an administrator figure to control the releases.

Visual Source Safe, like CVS, is adapted for small development teams, but it is limited only to Windows platforms and it doesn't allow versioning concurrent management. Microsoft's Visual Source Safe (VSS) tool is difficult to adopt for distributed teams. The VSS architecture locates all SCM functions and logic within a desktop client, which interacts with a centralized repository that operates as a relatively simple file server. This architecture has difficulties with scalability for large numbers of users, or for remote users accessing the repository over slow networks or the Internet. It is not intended for day-to-day usage by a geographically distributed team.

ClearCase [12], is a commercial tool adapted for large development teams and structured in complex ways. ClearCase is normally accessed over a LAN using a Windows-based client executable. The client executable relies heavily on local domain controllers and local RPC calls to interact with the central repository. Rational ClearCase provides a Web-based client for remote access to its central repository to circumvent such problems. The problem with this approach is that remote developers do integration renders many common tasks involving changes to multiple files particularly cumbersome and error prone, since developers must navigate a clumsy Web-based interface to check out individual files, one-by-one, make their changes, then manually check the files back into the repository, one-by-one. Furthermore, because ClearCase Web still relies on RPC calls on the back-end to perform most operations, which must then be translated back to HTTP and posted over the Web, performance suffers.

[12]http://www.ibm.com/developerworks/rational/products/clearcase

Several CSC projects involving distributed project teams and Rational ClearCase have taken to deploying remote instances of CVS and Subversion to serve as an SCM to remote teams, and then manually synchronizing changes to the main ClearCase repository. The latest release of ClearCase, purports to resolve some of the issues with remote access.

5.4.7 Tracking Tools

Any software has some defects during development, always around the first releases. Defects can be separated into failures (observable problems with the software) and faults (underlying problems with the code). Any defect-tracking should be able to support these two types. Defect can result in changes or new functionalities being requested. and the best place to keep these is in some kind of tools. Some of the commonly available defect-tracking tools used by XPers are Bugzilla [13], ScarabTigris [14] and so on.

[13]http://www.bugzilla.org
[14]http://scarab.tigris.org

Chapter 6

Conclusions

In recent years, Agile Methodologies and distributed development are two popular trends of software engineering. They have been proposed as two possible different solutions to the software crisis that is afflicting the ICT worldwide business. The former allows to respond to requirement changes while improving the overall quality of application released. The benefits of the latter are both the reduction of costs and of development time.

A debate is shaping about the compatibility of these two approaches.

The IT services industry in general is embracing distributed software development, as major firms scramble to build and recruit highly skilled, low-cost talent in the technology centers across India, China, and a multitude of other nations. Executing a software project with a distributed team is not a simple proposition. There are many, many challenges injected into an already complex process, many of which are impossible to quantify and difficult to control. As a result, many companies are looking at using agile methodologies in distributed environments.

Many agile practitioners claim that agile methodologies are not compatible with such a distributed development, since they are based on practices like pair programming, face-to-face communication, the customer on place and so on. The goal of

this research was to understand how the agile development process can affect the distribution of the team members and how development activities are enhanced, influenced or sometimes misdirected when developers use the Internet as main co-ordination and cooperation tool.

This work started with the goal to evaluate the adoption of some agile practices in a distributed software project.

The idea is that every context of distributed development has such its own characteristics and peculiarities that requires an ad-hoc methodology: it needs 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 geographic distribution, and how to replace those invalidated or evidently inapplicable.

I analyzed Agile Methodologies with regard to their philosophy and to their main practices.

Recent researches have been carried out in order to understand whether there are similarities or not between these two approaches [34], [11], [46], [4], [51], [25]. The findings showed that there are many similarities between AM and Distributed Software Development, and they tried to validate the effectiveness of adopting some agile practices in open source projects. Some open source software projects have already adopted some agile practices, such as Testing and Continuous Integration (see Thoughtworks [55] and SUnit [52]).

We have carried out some academic case studies with a distributed team adopting some XP practices. These research projects we carried out, have shown with empirical evidence which of those practices should be supported in a distributed environment.

My research is grounded on the following steps:

1. Understand if agile values are suitable for multi-site development as they are

for co-located teams.

2. Evaluate the applicability and efficiency of the Agile Methodologies for large and multi-site projects using:

   - Discrete Event Simulation Model. My research group developed a simulation model to evaluate the applicability and effectiveness of XP process, and the effects of some of its individual practices.
   - Case studies. See section 4.1, 4.2 and 4.3.

3. Analyze how to apply agile practices for distributed and large team and seek which among principles will become more important.

4. Define a guideline for a Distributed Agile Methodology. I described some good practices and a general approach towards distributed development.

I demonstrated that limits related to communication problems and also to difficulties of managing distributed teams are significant. But by means of a well-defined process, good tools, and a valid human resources management can do much to mitigate the problems related to the distributed development, but the fact remains that distributed teams are harder to run than co-located teams.

Bibliography

[1] S. W. Ambler and R. Jeffries. *Agile Modeling.* John Wiley and Sons, NewYork, NY, 2002.

[2] M. Angioni, R. Sanna, and A. Soro. Defining a distributed agile methodology for an open source scenario. In *Proceedings of The First International Conference on Open Source Systems*, 2005.

[3] I. P. Antoniades, Ioannis Stamelos, Lefteris Angelis, and Georgios L. Bleris. A novel simulation model for the development process of open source software projects. *International Journal of Software Projects: Improvement and Practice (SPIP), special issue on Software Process Simulation and Modelling*, 2003.

[4] P. Baheti, E. Gehringer, and D. Stotts. Exploring the efficacy of distributed pair programming. In *XP/Agile Universe*, pages 208–220, 2002.

[5] V. Basili. *Software Quality Assurance and Measurement: A Worldwide perspective*, chapter Applying the Goal/question/metric Paradigm in the Experience Factory, pages 21–44. International Thomson - Computer Press, 1995.

[6] Kent Beck. *Extreme Programming Explained.* Addison Wesley, second edition, 2000.

[7] Kent Beck. *Test Driven Development: By Example.* Addison-Wesley, 2003.

[8] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, second edition, 2005.

[9] Barry Boehm and Richard Turner. *Balancing Agility and Discipline*. Addison-Wesley, 2004.

[10] Barry W. Boehm. Get ready for the agile methods, with care. *Computer*, 35(1):64–69, 2002.

[11] K. Braithwaite and Tim Joyce. Xp expended distributed extreme programming. *Baumeister H., Marchesi M., Holcombe M. - Extreme Programming and Agile Processes in Software Engineering*, pages 180–188, 2005.

[12] R. Camplani, A. Cau, G. Concas, K. Mannaro, M. Marchesi, M. Melis, S. Pinna, N. Serra, A. Setzu, I. Turnu, and S. Uras. A comparison between co-located and distributed agile methodologies. In *Atti workshop SESAMOSS'05, 11 Luglio 2005, Genova*. Ed. Polimetrica, Milano, 2005.

[13] R. Camplani, A. Cau, G. Concas, K. Mannaro, M. Marchesi, M. Melis, S. Pinna, N. Serra, A. Setzu, I. Turnu, and S. Uras. Using extreme programming in a distributed team. *Polimetrica Publisher*, pages 39 – 46, 2006.

[14] A. Cau, G. Concas, M. Melis, and I. Turnu. Evaluate xp effectiveness using simulation modeling. In *Extreme Programming and Agile Processes in Software Engineering, 6th International Conference, XP 2005*, pages 48–56. Springer, June 18-23 2005.

[15] S. Chidamber and C. Kemerer. Towards a metrics suite for object oriented design. In *Proc. Conf. Object Oriented Programming Systems, Languages, and Applications - (OOPSLA'1)*, volume 26, pages 197–211, 1991.

[16] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Trans. Software Eng.*, 24(8):629–639, 1998.

[17] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Software Eng.*, 20(6):476–493, 1994.

[18] A. Cockburn. *Crystal Clear*. Addison-Wesley, Reading, MA, 2004.

[19] A. Cockburn and L. Williams. *The Costs and Benefits of Pair Programming. In Extreme Programming Examineted*, pages 223–247. Addison Wesley, 2001.

[20] Giulio Concas, Marco Di Francesco, Michele Marchesi, Roberta Quaresima, and Sandro Pinna. Study of the evolution of an agile project featuring a web application using software metrics. In *submitted to XP 2008: 9th International Conference on agile Processes and eXtreme Programming in Software Engineering*, Lecture Notes in Computer Science. Springer.

[21] L. Crispen and L. House. *Testing Extreme Programming*. Addison Wesley, MA: Addison Wesley Pearson Education edition, 2003.

[22] Don A. Dillman. *Mail and Internet Survey: The Tailored Design Methode.* 2000.

[23] K. Nidiffer E. and D. Dolan. Evolving distributed project management. *IEEE Software*, 22:63–72, Sep/Oct 2005.

[24] H. Erdogmus, M. Morisio, and M. Torchiano. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(3):226–237, 2005.

[25] Martin Fowler. Using an agile software process with offshore development. http://www.martinfowler.com/articles/agileOffshore.html, 2004.

[26] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139. ACM Press, 2003.

[27] B. F. Hanks. Distributed pair programming: An empirical study. In *XP/Agile Universe*, pages 81–91, 2004.

[28] J. A. Highsmith III. *Adaptive Software Development.* Prentice Hall, Upper Saddle River, NJ, 2002.

[29] DSDM Consortium J. Stapleton. *DSDM: Business Focused Development (2nd Ed.).* Addison-Wesley, Reading, MA, 2003.

[30] R. Jensen. A pair programming experience. *The journal of defensive software engineering*, 17(4):19–25, 2003.

[31] Marc I. Kellner, Raymond J. Madachy, and David M. Raffo. Software process simulation modeling: Why? What? How? *The Journal of Systems and Software*, 46(2–3):91–105, April 1999.

[32] M. Kircher. eXtreme programming in open-source and distributed environments. In *JAOO (Java And Object-Orientation) Conference*, 10 - 14 September 2001.

[33] M. Kircher, P. Jain, A. Corsaro, and D. Levine. Distributed extreme programming. In *Proceedings of XP 2001.*

[34] Stefan Koch. Agile principles and open source software development: A theoretical and empirical discussion. In Jutta Eckstein and Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings*, volume 3092 of *LNCS*, pages 85–93. Springer, 2004.

[35] R. Kessler L.A. Williams. *Pair Programming Illuminated.* Addison Wesley, Boston, MA, 2002.

[36] Kim Man Lui and Keith C.C. Chan. Test driven development and software process improvement in china. In Jutta Eckstein and Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings*, volume 3092 of *LNCS*, pages 219–222. Springer, 2004.

[37] Agile Manifesto. Manifesto for agile software development. http://www.agilemanifesto.org/, 2006.

[38] Katiuscia Mannaro, Marco Melis, and Michele Marchesi. Empirical analysis on the satisfaction of it employees comparing xp practices with other software development methodologies. In Jutta Eckstein and Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings*, volume 3092 of *LNCS*, pages 166–174. Springer, 2004.

[39] Michele Marchesi, Katiuscia Mannaro, Selene Uras, and Mario Locci. Distributed scrum in research project management. In Giulio Concas, Ernesto Damiani, Marco Scotto, and Giancarlo Succi, editors, *Agile Processes in Software Engineering and Extreme Programming, 8th International Conference, XP 2007, Como, Italy, June 18-22, 2007, Proceedings*, volume 4536 of *Lecture Notes in Computer Science*, pages 240–244. Springer, 2007.

[40] Marco Melis, Walter Ambu, Sandro Pinna, and Katiuscia Mannaro. Requirements of an iso compliant xp tool. In Jutta Eckstein and Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings*, volume 3092 of *LNCS*, pages 266–269. Springer, 2004.

[41] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. A case study of open source software development: The apache server. In *Proceedings of the International Conference on Software Engineering*, pages 263–272, 2000.

[42] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[43] M.M. Muller and O. Hagner. Experiment about test-first programming. In *IEEE Proceedings - Software*, volume 149, pages 131–136, 2002.

[44] S. Pinna, P. Lorrai, M. Marchesi, and N. Serra. Developing a Tool Supporting XP Process. In F. Maurer and D. Wells, editors, *XP/Agile Universe*, pages 151–160, 2003.

[45] S. Pinna, S. Mauri, P. Lorrai, M. Marchesi, and N. Serra. XPSwiki: an Agile Tool Supporting the Planning Game. In M. Marchesi and G. Succi, editors, *Extreme Programming and Agile Processes in Software Engineering*, pages 104–113. Springer, 2003.

[46] C. J. Poole. Distributed product development using extreme programming. In J. Eckstein and H. Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering*, pages 60–67, 2004.

[47] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[48] J. M. Felsing S. R. Palmer. *A Practical Guide to Feature-Driven Development*. Dorset House,New York, NY, 1999.

[49] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[50] Sourceforge web site. http://sourceforge.net/.

[51] D. Stotts. Virtual teaming: Experiments and experiences with distributed pair programming. In *XP/Agile Universe*, pages 129–141, 2003.

[52] Sunit. Url: http://sunit.sourceforge.net.

[53] Jeff Sutherland, Anton Viktorov, Jack Blount, and Nikolai Puntikov. Distributed scrum: Agile project management with outsourced development teams. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 274a, Washington, DC, USA, 2007. IEEE Computer Society.

[54] H. Takeuchi and I. Nonaka. The new new product development game. *Harvard Business Review*, 64:137–146, 1986.

[55] Thoughtworks open source. http://opensource.thoughtworks.com.

[56] I. Turnu, M. Melis, A. Cau, A. Setzu, G. Concas, and K. Mannaro. Modeling and simulation of open source development using an agile practice. *Journal of System Architeture*, 52(11):610–618, 2006.

[57] Ivana Turnu, Marco Melis, Alessandra Cau, Michele Marchesi, and Alessio Setzu. Introducing tdd on a free libre open source software project: a simulation experiment. In *QUTE-SWAP '04: Proceedings of the 2004 workshop on Quantitative techniques for software agile process*, pages 59–65, New York, NY, USA, 2004. ACM.

[58] Doug Wallace, Isobel Raggett, and Joel Aufgang. *Extreme Programming for Web Projects.* Addison-Wesley, 2003.

[59] Laurie Williams, Lucas Layman, and William Kreb. *Extreme Programming Evaluation Framework for Object-Oriented Languages–Version 1.4*. North Carolina State University, Department of Computer Science, TR-2004-18, 2004.

Appendix A

Tools for Synchronous and Asynchronous communication

Here we list some synchronous and asynchronous communication tools.

A.1 Asynchronous Communication

MailMan

Mailman is free software for managing electronic mail discussion and e-newsletter lists. Mailman is integrated with the web, making it easy for users to manage their accounts and for list owners to administer their list. Mailman supports built-in archiving, automatic bounce processing, content filtering, digest delivery, spam filters and more.

- Lycence: GNU GPL
- Platforms: Unix/Linux
- URL: http://www.gnu.org/software/mailman/mailman.html

Wiki

- Licence: Artistic License

- Operating System: Windows, Linux, Mac OS X
- URL: http://www.wiki.org/

Wiki is a piece of server software that allows users to freely create and edit Web page content using any Web browser. Wiki supports hyperlinks and has a simple text syntax for creating new pages and crosslinks between internal pages on the fly.

Ward Cunningham, and co-author Bo Leuf, in their book "The Wiki Way: quick collaboration on the web" described the essence of the Wiki concept as follows:

- A wiki invites all users to edit any page or to create new pages within the wiki Web site, using only a plain-vanilla Web browser without any extra add-ons.
- Wiki promotes meaningful topic associations between different pages by making page link creation almost intuitively easy and showing whether an intended target page exists or not.
- A wiki is not a carefully crafted site for casual visitors. Instead it seeks to involve the visitor in an ongoing process of creation and collaboration that constantly changes the Web site landscape.

A wiki enables documents to be written collaboratively, in a simple markup language using a web browser. A single page in a wiki website is referred to as a "wiki page", while the entire collection of pages, which are usually well interconnected by hyperlinks, is "the wiki". A wiki is essentially a database for creating, browsing, and searching through information.

A defining characteristic of wiki technology is the ease with which pages can be created and updated. Generally, there is no review before modifications are accepted. Many wikis are open to alteration by the general public without requiring them to register user accounts. Sometimes logging in for a session is recommended, to create a "wiki-signature" cookie for signing edits automatically. Many edits, however, can be made in real-time and appear almost instantly online. This can facilitate abuse

of the system. Private wiki servers require user authentication to edit pages, and sometimes even to read them.

A.2 Syncronuos Communication

Sangam 1.6.4

Sangam is a plug-in for synchronous, distributed collaborative programming. Sangam provides a user interface specifically for distributed Pair Programming and synchronizes the development environments for both programmers. The plug-in itself is developed by a distributed team.

- Category: Chat, Conferencing, Firewalls;
- Operating System: OS Indipendent (written in an interpreted language;
- licence: GNU General Public Licence (GPL);
- URL: http://sangam.sourceforge.net/plugin

The development of Sangam is an ongoing effort. In the current version, Sangam provides the following features:

- Editor synchronization: Including typing, selection, opening, closing, and view port scrolling synchronization.
- Launching synchronization: The programmers can launch or debug the same Java application or JUnit test at the same time.
- Resource synchronization: When the driver adds, deletes, or modifies the files in his or her local disk using Eclipse, the same change will also be reflected in the navigator's local disk.
- Refactoring synchronization: From the aspect of Eclipse API, refactoring is a complicated form of resource and editor change. A single refactoring may affect

many files and the content of different editor windows. Sangam is designed to deal with some special cases raised with refactoring within Eclipse.

AIM

- Category: Instant Messaging;
- Licence: Freeware
- Operating System: Windows, Windows CE, Mac OS, Linux, Palm OS
- URL: http://www.aim.com

AIM (America OnLine Instant Messaging) is a software for instant messaging. AIM allows to communicate in synchronous way to friends, colleagues an so on, moreover it allows to interact with the client e-mail in order to send/receive messages, to establish a vocal communication, to send and to receive files.

ICQ

- Category: Instant Messaging;
- Licence: ICQ Terms Of Service - ICQ License Agreement:
- http://www.icq.com/legal/end-user-license.html
- Operating System: Windows, MacOS, Palm OS, PocketPC
- URL: http://web.icq.com/

ICQ ("I Seek You") allows to contact friends/colleagues who are on line in instant way. ICQ allows:

- to send and receive messages;
- to send and receive emails;
- to send and receive URL;

- to send and receive SMS;
- to send and receive files o directories;
- audio/video comumnication (Webcam).

Jabber

- Category: instant messaging;
- Operating System: Jabber is a protocol, then it is independent from the platform.
- URL: http://www.jabber.org/

Jabber is best known as "the Linux of instant messaging", an open, secure, ad free alternative to consumer IM services like AIM, ICQ, MSN, and Yahoo. Under the hood, Jabber is a set of streaming XML protocols and technologies that enable any two entities on the Internet to exchange messages, presence, and other structured information in close to real time. Jabber technologies offer several key advantages:

- Open: the Jabber protocols are free, open, public, and easily understandable; in addition, multiple implementations exist for clients, servers, components, and code libraries.
- Standard: the Internet Engineering Task Force (IETF) has formalized the core XML streaming protocols as an approved instant messaging and presence technology under the name of XMPP, and the XMPP specifications have been published as RFC 3920 and RFC 3921.
- Proven: the first Jabber technologies were developed by Jeremie Miller in 1998 and are now quite stable; hundreds of developers are working on Jabber technologies, there are tens of thousands of Jabber servers running on the Internet today, and millions of people use Jabber for IM.

- Decentralized: the architecture of the Jabber network is similar to email; as a result, anyone can run their own Jabber server, enabling individuals and organizations to take control of their IM experience.
- Secure: any Jabber server may be isolated from the public Jabber network (e.g., on a company intranet), and robust security using SASL and TLS has been built into the core XMPP specifications.
- Extensible: using the power of XML namespaces, anyone can build custom functionality on top of the core protocols; to maintain interoperability, common extensions are managed by the Jabber Software Foundation.
- Flexible: Jabber applications beyond IM include network management, content syndication, collaboration tools, file sharing, gaming, and remote systems monitoring.
- Diverse: a wide range of companies and open-source projects use the Jabber protocols to build and deploy real-time applications and services; you will never get "locked in" when you use Jabber technologies.

Kizna SyncShare

- Category: Chat, Conferencing, File Sharing, Cryptography, Networking;
- License: GNU General Public License (GPL), GNU Library or Lesser General Public License (LGPL), Other/Proprietary License;
- Operating System: All 32-bit MS Windows (95/98/NT/2000/XP), All POSIX (Linux/BSD/Unix-like OSes), Linux, Solaris;
- URL: http://www.kizna.org/.

SyncShare Server is a secure messaging and real-time collaboration platform. Build solutions to connect people, data, and applications through firewalls and mo-

bile devices. It handles all networking, security issues and lets one focus on the business.

Microsoft NetMeeting

- Category: Chat, Conferencing, File Sharing, and so on;
- Licence: Microsoft commercial software ; NetMeeting for Windows 95, 98, Me e NT is freeware downloadable to URL below reported. From Windows 2000 and XP, NetMeeting is integrated in the platform Windows and it shares the licence terms;
- Operating System: All 32-bit MS Windows (95/98/NT/2000/XP);
- URL: http://www.microsoft.com/windows/netmeeting/default.asp.

Microsoft NetMeeting is a VoIP and multi-point videoconferencing client included in many versions of Microsoft Windows (from Windows 95 to Windows XP). Since the release of Windows XP, Microsoft has deprecated it in favour of Windows Messenger and Microsoft Office Live Meeting, although it is still installed by default. Note that Windows Messenger, MSN Messenger and Windows Live Messenger hooks directly into NetMeeting for the application sharing, desktop sharing, and Whiteboard features exposed by each application. NetMeeting is no longer included with Windows Vista, and has been replaced by Windows Meeting Space and Microsoft Office Live Meeting. Windows Meeting Space only has collaboration features and lacks NetMeeting's conferencing features. Live Meeting includes the conferencing features that Windows Meeting Space doesn't offer.

Netmeeting is a conferencing client developed by Microsoft that allows users to interact in real time over the internet. Most people call it a "video conferencing client", but it is actually capable of much more than that. It includes:

- A Chat client, similar to AOL Instant Messenger;
- An Audio and Video conferencing client

- A Whiteboard, which is a shared drawing space where people can collaborate in real time.
- Application Sharing, where you can choose to share an application you are running with others in your conference. Note here that while other users will need to have Netmeeting installed, they do not need to have a copy of the application you are sharing installed.
- A file transfer application for sending files

MSN Messenger

- Category: Chat, Conferencing, File Sharing, and so on;
- Licence: Freeware;
- Operating System: Windows XP, Windows NT4 (SP6), Windows 2000, Windows 98/ME, Mac OS, Pocket PC;
- URL: http://messenger.msn.it/default.asp? client=1

Some features are:

- Text/Voice/Video chat;
- File sharing;
- Whiteboard sharing.

Yahoo! Messenger

- Category: text, video, voice and data communication;
- Licence: Application software is freeware, but the service is supplied by contract terms in http://docs.yahoo.com/info/terms/messenger.html.
- Operating System: Windows, Mac OS
- URL: http://messenger.yahoo.com/