Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie
Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki
Katedra Informatyki

Rozprawa doktorska

Wspieranie „zwinnych" metodologii wytwarzania oprogramowania w środowisku rozproszonym

mgr inż. Jacek Dajda

Promotor: prof. nadz. dr hab. inż. Grzegorz Dobrowolski

Kraków, 2008

AGH University of Science and Technology
Faculty of Electrical Engineering, Automatics, Computer Science and Electronics
Department of Computer Science

Ph.D. Thesis

Supporting agile methodologies in distributed setting

Jacek Dajda M.Sc.

Supervisor: Grzegorz Dobrowolski Ph.D., Hab., Prof. of AGH

Krakow, 2008

To Gosia

Acknowledgements

My utmost gratitude to my supervisor Prof. Grzegorz Dobrowolski for inspiring discussions, splendid remarks and constructive critique of my work.

Warmest thanks to my friend Gosia for her unparalleled patience and never-ending motivation.

The highest appreciation to my family for the encouragement and daily support.

Special thanks to all students who participated in my experiments.

Contents

Chapter 1

Introduction

Software development is a complex process. A collection of various elements ranging from specific technical issues to human interaction and other psychological aspects. It is also driven by several variables such as cost, available time and resources. To make it worse, the list of both elements and variables is not static and depends on the project, its specific needs and current circumstances. Having assumed that these needs and circumstances can easily change at any point of development, it becomes clear how dynamic and complicated system the software production process can be.

Undoubtedly, it requires a special approach: a software development methodology. Although software engineering is a considerably young discipline, a number of approaches were proposed and practiced. The first ones emerged from the motivation to bring a systematic and disciplined approach to chaotic ad-hoc coding, sometimes referred to as *Cowboy coding* [34]. In late 70s and 80s the growing need for large and complex systems resulted in a specification of formal (hereinafter referred to as traditional and classic) models of software development. Based on ISO [50], TickIt [51] standards and CMM [27] model they allowed for setting order in the process of software production. On the other hand, the costs of standardization, among others, included: the increase of bureaucracy in software process, deterioration of process immunity to frequent changes of project requirements, lower pace of progress resulting in a prolongation of final results delivery. These factors combined with short-sighted project management resulting in unrealistic plans and marketing-driven decisions, led to failure of numerous software projects. This phenomenon, known as *LOOP syndrome* [45], became a subject of thorough studies and publications. One of them is a well known Yourdon's work [75], a comprehensive guide on how to identify and avoid *death march* projects.

All of these strongly motivated the need for changes in the software development model, which took place in the late 90s. They were affected by industrial transformations, that were caused by the reception of the Japanese approach to company management [48] and the economic recession leading to reduction of production costs [31]. On the ground of software engineering, the inception of agile movement was one of the effects of these trends. Agile movement proposed a new software development model, by some regarded as revolu-

tionary in opposition to the standard one. Formulated in February 2001 by 17 experienced methodologists, Manifesto for Agile Software Development (also known as Agile Manifesto) [94] articulated a collection of principles [34] that redefined the priorities of the development process. Among them, the manifesto emphasized fast delivery of a high quality product to the end user as well as close collaboration among development team including customers and business experts. These and other principles became a core of agile methodologies, which were emerging from the new movement. As an example eXtreme Programming (hereinafter referred to as XP) [3, 2, 67, 64], Scrum [62] and Crystal Family [11] might be given.

From the current perspective, it can be stated that agile methodologies, while proposing a resolution to several existing management problems, came up with new challenges and research issues. One of them is overcoming the limitation of close physical proximity of project participants, imposed by the close collaboration principle. Nowadays, with an increasing number of programmers working in geographically distributed settings (as an effect of globalization) and growing importance of offshore development, this limitation is reckoned as a severe drawback of the agile approach. The attempts to overwhelm it form a new research area called Distributed Agile Development (hereinafter referred as DAD). While it corresponds to whole agile movement, first attempts are dedicated to Distributed eXtreme Programming (hereinafter referred as DXP, a subdiscipline of DAD), the XP equivalence in distributed setting.

One of DAD research domains is building communication channels capable of substituting face to face communication in distributed setting. As experiments show, this can be achieved by a proper software tool support. Verification of provided support is the subject of another DAD research domain. Being a fairly new discipline, DAD is still regarded more in terms of academic studies rather than industrial realities. While Agile Development gains a wide industry recognition and gradual appreciation, DAD is still a rare practice. It seems that this situation is mainly caused by considerable communication problems among distributed team members and the lack of proper support. However, the first studies indicate its promising feature in industrial applications.

Having recognized the current status of DAD and its perspectives, the thesis of this dissertation is as follows:

> **By providing the proper software support, it is possible to overcome limitations of agile methodologies, which emerge from geographical distribution of development team, and maintain the overall costs of agile software production in a distributed setting.**

Two following elements of the thesis require detailed explanation:

1. maintaining the overall costs
2. the proper software support

The first element, *maintain*, points at comparative analysis of agile software production in co-located (AD) and distributed (DAD) settings . For the needs of the analysis, the following classification of the *overall costs* is considered:

(a) indirect costs - expenditures necessary to "have the team together" (co-located team), e.g. organization of the workspace, meetings, business travels, delegations.

(b) direct costs - the direct spending on the development work, aimed at obtaining a software product of the assumed quality. The main part of these costs are the salaries of team members.

(c) other costs - the costs which are assumed negligible for the comparative analysis as they remain at the approximate level regardless on the project setting. The example are hardware investment costs.

Having considered this classification, two relations can be observed:

- DAD offers the reduction of costs of type (a) by the substitution of traditional communication methods (based on face-to-face communication) with new ones based on telecommunication.
- Relation between costs of type (b) in both cases can be judged using several metrics proposed in [18, 29, 16, 21, 47, 46] and also in this dissertation.

When these two relations are considered together, *maintain* means here that the transposition of AD into a DAD setting is acceptable when costs of type (b) do not spoil the positive balance realized by reduction of costs of type (a), or, what would be better, maintain at the comparable level.

The second element of the thesis, *the proper software support*, means that thanks to the dedicated tools, which meet specific requirements, the significant practices of AD are effective also after their transposition into the DAD setting. As result, they can be used in a development, regardless on the available setting.

The above discussed two elements form the three following goals of this dissertation that must be fulfilled:

1. Comprehensive specification of the significant agile practices with a strong consideration of their features, especially distribution sensitivity (the degree of distributed setting influence) and posed requirements towards supporting software.
2. Development of the *proper software support* for selected practices with accordance to the results obtained during completion of the previous goal (that specific requirements for significant and sensitive practices) .
3. Comparative effectiveness analysis, in the sense of the costs of type (a), of the selected practices, with regard to their transposition from AD into DAD setting by application

of the results of the two previous goals (that is developed software support fulfilling the specific requirements).

This dissertation is organized in the following manner.

Second chapter is dedicated to the introduction of Agile Development and its distributed approach. The chapter starts by explaining the motivation and indicating the possible sources of inspiration. Next, it outlines the Manifesto of Agile Software Development and gives an overview over the core agile practices. In the third part, the chapter focuses on the Distributed Agile Development (DAD) with an emphasis on its motivation, needs, major problems, current work and existing solutions.

Chapter three proposes a conceptualization of DAD. The conceptualization is divided into three parts. In the first one, the major DAD elements and their dependencies, which are then mapped onto presented agile practices. The second part discusses the effect of team distribution on the presented practices. It indicates the required communication channels and suggests how they can be supported. Based on the conclusions coming from these two parts, three agile practices are selected for further investigation. The third part of the chapter three describes proposed set of metrics for the selected practices. The metrics are required to measure the practices' efficiency and compare co-located and distributed settings, which is needed to confirm the posed thesis.

Fourth and fifth chapters are the most important chapters of the dissertation. Chapter four overviews the developed supporting solution for the selected agile practices, called Agile Studio. It was necessary due to the lack of a proper support, which would allow for examination of the posed thesis. The chapter starts by outlining the proposed concept and architecture of Agile Studio. Next, each from its four components is discussed, with an emphasis put on the developed requirements and demonstration of the implemented features.

Chapter five presents how these requirements and features were developed using the proposed concept of Experiment-Driven Development. The concept assumes a development based on series of evaluating experiments, which are meant to simulate the real working environment. The chapter overviews the development plan, organization of experiments and the proposed method for Agile Studio evaluation. Next, the chapter describes how this concept was applied. The presented development story illustrates the evolution of Agile Studio requirements in the scope of conducted experiments. Furthermore, the chapter demonstrates the comparison of results obtained during conducted experiments for co-located and distributed teams supported by Agile Studio. The results are based on the proposed set of metrics. Eventually, the summary of chapter five provides rationale for verification of the posed thesis.

In chapter six, the main conclusions of this dissertation are formulated. In addition, the chapter outlines interesting observations, encountered problems and possible assignments for future work. It also proposes an informal summation of the carried work in the form of revision of the Manifesto of Agile Development in terms of distributed setting.

The regular chapters are accompanied by the appendices presenting selected details from the conducted research.

Appendix A provides selected details from the organized experiments, including available

resources and the plan of a single experiment.

Appendix B shows an example task, which was assigned to participants during experiments aimed at Planning Game evaluation.

Appendix C contains schemas, which illustrate the evolution of the Agile Studio requirements with regard to the organized experiments.

Appendix D collects obtained results for the proposed metrics with regard to all three discussed agile practices and all conducted experiments.

Chapter 2

Agile development: current state and perspectives

Agile movement reflects a new direction in the philosophy and techniques of modern software development. This movement, also known as *light-weight*, emerged from a disappointment with traditional disciplined and plan-driven engineering methodologies. In a sense, it may be seen as a collection, or, more precisely, a compilation of well proven and reliable techniques of project management and development. However, this gives only a partial view of the agile approach. What actually counts is a novel perception of development process that brings to front values considered subsidiary in a traditional models. That is individuals, innovation and communication. While traditional approach aims at process *modeling*, agile movement proposes process *adaptation*. Before these issues are explained, it is worth to overview in greater detail factors that affected the inception of agile concept.

It is also a proper place to clear out the terminology issue considering the usage of *methodology* term in the agile context. According to [81], *methodology* defines *a set or system of methods, principles, and rules for regulating a given discipline, as in the arts or sciences.* In other words, it relates to *a rationale and philosophical assumptions that underlie a particular study* [82], a collection of methods in particular. In this context, term *agile methodologies* may seem inaccurate. However, in opposition to traditional methods, which offer a **predefined set of procedures and frameworks**, agile methodologies are primarily an **adjustable combination of paradigms and concepts**. The usage of *methodology* term in the agile context emphasizes these core differences.

2.1 The origin and insights of the agile movement

There is a number of factors that contributed to the formulation of agile movement. The direct ones emerge from the disappointment with traditional methods and include gained experiences and observations made during *death-march* projects. One of the sources of inspiration was

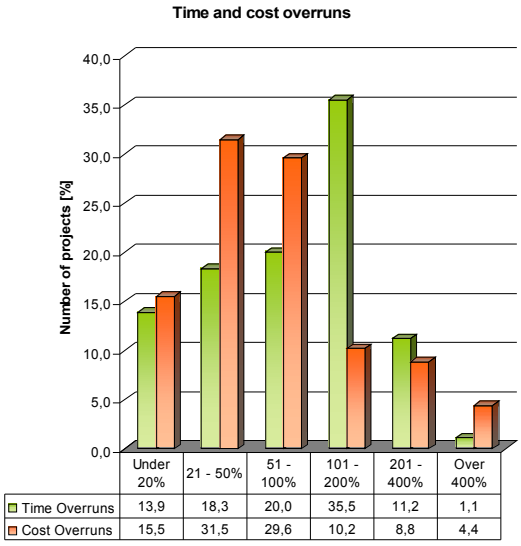
also a novel approach to management which evolved in Japanese industry and strongly added to its world-wide success. Similarly, various philosophical and psychological concepts affected the new movement formulation.

2.1.1 Lessons learned in death-march projects

The introduction of traditional [59] and formal models [27] as well as application of standardization procedures [50, 51] were driven by the need for a repeatable, predictable and easy-to-control process. The side-effects were, among others, the increase bureaucracy and vulnerability to requirements changes, not to mention long product delivery and exceeding project budgets.

A LOOP syndrome describes these problems. It is an acronym for Late, Over budget, Overtime, Poor quality. The last one emerges from the previous three: when project is late, the money run out and people are tired, the first thing to resign from is, in most cases, quality. Unfortunately, lack of quality involves a greater risk of critical errors to track, which means more time spent on debugging. In result, it boosts the LOOP effect and closes the vicious circle (to which LOOP term also refers to).

A numerous studies confirmed this tendency, also described as death-march projects [75]. One of them was the annual report of Standish group [19, 20] that covers IT companies in USA. According to it, 1995 year came up with 29.5%, 37.1% and 21.6% canceled projects respectively in large, medium and small companies, whereas the completely successful project rated at 9%, 16.2% and 28%. Another results concerning the time and cost overruns are depicted in Figure 2.1. In its view, the average time overrun across all companies was 222% of the original time estimate, while the cost was 189% of the original cost estimate. Based on carried questionnaire, the report indicated crucial factors of failed and succeed projects. The following ones belong to the first group: lack of user input, incomplete and changing requirements, unrealistic plans and expectations. The second group includes: user involvement, executive management support, clear statement of requirements and proper planning. Surprisingly, hard-working focused staff was seen as a minor factor of successful projects.



	Under 20%	21 - 50%	51 - 100%	101 - 200%	201 - 400%	Over 400%
Time Overruns	13,9	18,3	20,0	35,5	11,2	1,1
Cost Overruns	15,5	31,5	29,6	10,2	8,8	4,4

Figure 2.1: Time and cost overruns

To conclude, the gained experiences emphasized the major role of quality, user involvement, realistic planning and communication, especially when requirements and project goals are being defined. Interestingly, small teams working on limited-budget projects were found more successful than large ones developing systems of considerable size and budget.

2.1.2 Lessons learned in a Far East

In [48] authors shared their insights on the sources of Japanese companies success. Based on studies performed in such industrial competitors like Honda, Canon, NEC and Sharp, they introduced a notion of *knowledge-creating company*, i.e a company whose sole business is *continuous innovation*. By continuous knowledge acquisition and creation, it is possible to constantly embody new technologies and respond to market and customer current needs. This is an opposition to the tradition of Western approach aimed at processing of knowledge which is quantified and systematic.

To illustrate these differences, authors distinguished 2 types of knowledge: explicit and tacit. The first one refers to formal, systematic and objective information, easy to process and share. Tacit knowledge, on the contrary, is related to personal experiences, skills, intuitions and insights. Therefore, it is usually difficult to articulate and share. According to authors, Western companies based on explicit knowledge and formal procedures, miss the benefits of tacit knowledge, which is a source of invention and progress. Obviously, this brings individuals and communication to front as important factors of success.

Based on these observations, authors identified four patterns in the process of knowledge creation: from explicit to explicit, from tacit to tacit, from tacit to explicit, from explicit to tacit. The first two are limited to combinations of already possessed knowledge and skills. The innovative potential hold the other two as they allow for systemization of "hidden" individual insights, embedding them in developed products and enriching the tacit knowledge of other people. This process, a core of continuous innovation, is sometimes called a *spiral of knowledge*.

Finally, continuous innovation requires suitable management. It was noticed that most successful were these managers who joined top management visions with realities of the working team. This kind of management, called *middle-up-down*, allows for merging two different areas of knowledge, usually explicit and tacit ones, in this way supporting the spiral of knowledge.

2.1.3 Lessons learned in self-help studies

In 90s, a growing popularity of self-help studies resulted in a number of books and courses dedicated to a variety of psychological and personal aspects, such as increasing self-esteem, improving relationships, personal effectiveness and time management. One of them was a best-selling Stephen Covey's self-help book [14], which proposed a set of paradigms aimed at increasing the personal efficiency and improving interpersonal communication.

In the book, the paradigms and principles are collected into 7 *habits*. The second one, called *Begin with the End In Mind*, articulates the importance of setting proper goals and identifying own roles in interpersonal relations. To support these, visualization techniques are proposed. In addition, author discusses the way the organizational mission statements are created. In his opinion, a suitable selection of mission goals done by all members of organization is usually more successful than a predefined and imposed one. The way these goals are achieved is a subject of next habit which is *Put First Things First*. It suggests

a framework for prioritizing tasks in the perspective of long term goals. This allows for identification and cancellation of task which are urgent but of minor significance for achieving the goals. According to Covey, the framework should be based on current run-time results rather than predefined detailed plans. This makes it more flexible and adaptable to future unexpected changes.

All activities and techniques can be improved by practicing the last two habits: *Synergize* and *Sharpen the saw*. The first one relates to the power of collaboration, especially in problem solving and decision making. By stating that "The whole is greater than the sum of its parts", the author claims that open-minded and intensive collaboration, built on a variety of people's strengths and insights, can lead to innovative results, unreachable when people work individually. The second habit discusses the way the level of productivity and efficiency of the previous habits can be maintained in a long time perspective. Among others, author points out the importance of balance in the process of renewal, including suitable selection of relaxation activities.

The Covey's studies correspond to well-known *parapeto principle* (also known as 80/20 rule) [33]. It states, based on performed observations and experiences, that in many areas 80% of results are consequences of only 20% of causes. To put it in more practical way: the 80% of success usually depend on 20% of efforts. This brings to conclusion that a key to success and efficiency is suitable prioritization of performed work and tasks to be completed. To support prioritization, numerous techniques and tools, often referred as *mind-tools* [43, 6]. While these and other concepts were targeted mainly at improving personal lives of individuals, they also gained appreciation of business and industry.

To sum up, the inception of agile movement was a response for the disappointment of traditional methodologies and their inability to solve common engineering problems known as LOOP syndrome. The proposed agile approach is a compilation of insights and techniques of various sources, such as industrial transformations and self-help studies. While their scope is varied, the common theme is the important role of communication and human aspects in the process of software development.

2.2 Agile movement principles

When in 1996 Kent Beck for the fist time employed XP methodology in a project for Daimler Chrysler company, probably no one thought it would so strongly contribute to the recent transformations in the philosophy of software development. These were launched by Agile Manifesto (full name is Manifesto for Agile Software Development) [94] signed in February 2001 by 17 agile practitioners, followed by the creation of Agile Alliance [77], non-profit organization to support those using agile approaches.

2.2.1 Terminology

Before the Agile Manifesto and principles are presented, a terminological remarks need to be added. Being an opposite to the formality of traditional approach, agile movement places

a considerably small importance to the regulation of the used terminology. In addition, the movement strongly relates to psychological aspects of software development, which cannot be precisely measured and expressed. Nevertheless, a coarse regulation of the terminology is possible and advisable, as it clears out several issues before further reading. What is more, it also depicts a structure of agile methodology.

Agile movement is based on the postulates and principles of Agile Manifesto. The *postulates* are general keywords of agile development, the paradigms of agile movement. The *principles* indicate more concrete actions. They are meant to introduce, emphasize and maintain *values* being significant elements of the development process. All agile methodologies are driven, explicitly or implicitly, by a set of selected values, most of them emerging from the postulates of the manifesto.

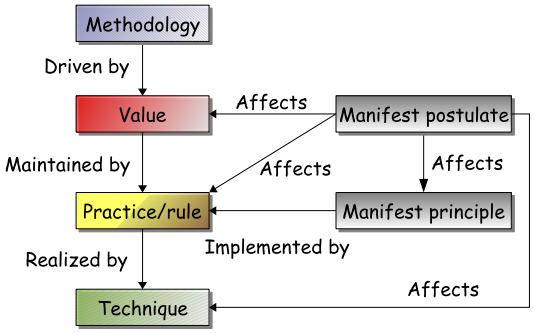


Figure 2.2: Terminology map

As the manifesto principles are mainly concepts and ideas, their require a practical implementation, that is, *practices* and *rules*. These, in general, are used as synonyms. Agile methodologies propose a variety of practices and rules, ranging from general suggestions to concrete activities. While, in some cases, the first ones remind of manifesto principles, the concrete activities are build on *techniques*, being detailed recipes, an algorithmic description of actions to realize. The presented relations are illustrated in the terminology map in Figure 2.2.

It must be emphasized that this is only a coarse systemization. One can find several exceptions. Additionally, some methodologies employ terms which do not comply with presented systemization. However, the concept of methodology and its structure, in general, remains the same.

2.2.2 Agile Manifesto

The aim of the manifesto was to establish a common set of principles and paradigms for the emerging agile methodologies. This set was formulated based on gained experiences with *death-march projects* (see Section 2.1). The main idea behind new methodologies was to provide high-quality values that meet customer's expectations, especially with regard to required functionality, project budget and schedule. According to Agile Manifesto this can be achieved by appreciating:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

Individuals and interactions The first postulate refers to formal procedures, rules and environments of traditional methods that overshadow the needs of people working in the development teams. The strict frameworks imposed by top management, usually unfamiliar with the day-to-day reality of development process, frequently turned into obstacles rather than expected assistance. By following the predefined rules, not adapted to individual requirements, some of the key project points were missed or neglected. The lack of required communication effected in developers isolation which led to misconceptions and conflicts. The results such as unclear requirements and unrealistic expectations were depicted in Section 2.1.

Working software The second postulate of manifesto emphasizes the real value of the project, that is working software meeting customer's expectations. In case of traditional methods, the implementation phase was preceded by long-term analysis and design studies, producing suitable documentation. Although these studies could be valuable to the developers team, to the customer they were of no use as he/she required and paid for working software. Furthermore, it was not seldom that the costs needed to prepare and maintain the project documentation exceeded the value of final product.

Customer collaboration Next postulate proposes a change of customer's project role. Kent Beck and others noticed that teams with a customer actively involved in the project and having a real influence on the developed software, are more efficient and achieve greater results, comparing to these working in isolation and meeting customer only during presentations. When customer is a member of a team, the presentations are no longer needed. The time needed for their preparation can be used in more productive way. Furthermore, active customer collaboration allows for a rapid reaction to unexpected changes. When change of requirements is considered, collaboration immediately clears out any misconceptions, saving time spent on implementing useless functionalities. Needless to say, this reduces project risk and solves several problems of LOOP syndrome. Finally, customer collaboration positively influences interpersonal relationships and the spirit of the project, motivates the developers and reduces tensions related to deadlines and contract negotiations.

Responding to change The last point of manifesto strongly depends on the previous ones. As mentioned, close customer collaboration allows for fast responding to changes. However, these requires suitable approach to project planning and managing. Traditional methods, with a strict plan followed from analysis till implementation and final release, did not leave much space for any alterations. Any unexpected change could cause a considerable delay, which was obviously an indication of LOOP syndrome. What is more, as experiences indicated, initials plans hardly ever remained the same through the project duration. Therefore, it seems advisable to put more attention on preparation for inevitable changes, with a planning reduced to limited scope, rather than creating and closely following detailed long-term schedules.

2.3 Agile methodologies and practices

The group of agile methodologies is numerous and constantly growing. The well-known ones include eXtreme Programming (XP) [67, 3, 2, 71, 64], Scrum [62, 61], Crystal [9, 11],

Lean Development [55], Feature-Driven Development [54], Adaptive Software Development [22, 23], Dynamic Systems Development Method (DSDM) [63, 13] and Agile Unified Process [108]. Although they are driven by the same manifesto postulates, the way agile methodologies realize them is varied. In general, methodologies differ in the level of abstraction: from studies on adjusting process to project size and criticality (Crystal), through generic management frameworks (Scrum, AUP) to sets of concrete techniques (XP).

Agile methodologies are built upon a set of practices and techniques. Each one provides certain values of higher or smaller importance for the whole project. However, these values can be empowered when applied together. This is called a synergy. It is often stated that this synergy, not practices individually, determines the methodology efficiency.

2.3.1 Agile practices and techniques

The practices and techniques are a core of every agile methodology. Some of them are common for all agile methodologies. The only difference lies in their naming or description. On the other hand, the are unique practices, typical only for the specific methodology. A practice can be a general advice and suggestion as well as concrete procedure. Some practices gained appreciation even in traditional processes. In some cases, their usefulness led to formulation of a new methodology. An example is Test-Driven Development [4, 1].

There is also another way they can be partitioned and approached. That is: by the aspects of software development process to which they refer to. Among others, these can be: team management, project planning, coding standards and even work space arrangement. Here, the second option is used, as it provides better organization and is compliant with further concepts and findings.

Practices of planning and customer collaboration

One of the basic practice that belongs to this group is iteration-driven development based on small and frequent releases. Described as *Small Releases* (XP), *Sprint* (Scrum) or *Frequent Delivery* (Crystal), this practice is a core of every agile methodology. It assumes delivering working high-quality software that meets customer expectations, with every iteration. The length of iteration depends on the methodology and varies from one to five weeks. For instance, Scrum methodology strictly defines Sprint as 30 days work period. The delivered software can be used by customer in several ways. Firstly, it is a base to inspect and review the current concepts and requirements. Secondly, it might be tried out by end-users which provides a valuable feedback. Thirdly, it is a measure of progress; simple, clear and understandable to customer. This strengthens trust and openness in the relationships between developers and customer.

Small releases require frequent planning meetings. The first one, covering the whole project or release scope, is organized before the project is launched. This phase is known as *Release Planning* (XP), *Pre-Game* (Scrum), *Pre-Project Phase* (DSDM), *Inception* (AUP), etc. The others are held before and after every iteration. Therefore they are known as *Iteration Planning* (XP, LSD), *Sprint Planning Meeting* and *Sprint Review Meeting* (Scrum),

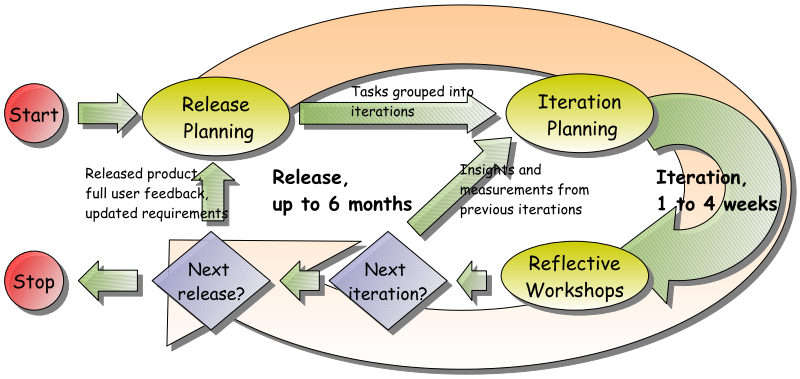


Figure 2.3: Agile project life-cycle

Feasibility Study and *Business Studies* (DSDM), *Reflective Workshops* (Crystal) etc. The goal of release planning is creating a coarse project schedule. This means partitioning of work, like desired features of the developed software, into the frames of individual iterations. Iteration planning meetings provide an opportunity to verify the initial schedule and modify it, so that it is more accurate and meets customer expectations. This is possible as every meeting requires the presence of customer or his/her representative. While customer gains additional insights on the project status, developers receive useful feedback. In addition, this increases the chances of resolving any misunderstandings or implicit changes in the requirements. In this way, this one of the major project risks (see Section 2.1 and Standish Report summary [19]) is considerably reduced. Finally, reflective meetings held at the end of every iteration are dedicated to results presentation and a summary of the performed work. The summary should include, preferable in a form of free discussion (this follows the first postulate from manifesto), occurred problems and possible enhancements. In some way, this reminds of *Sharpening the Saw* habit mentioned in section 2.1. Figure 2.3 depicts a typical life-cycle of an agile project.

There are several techniques supporting planning meetings. XP promotes *User Story cards*, which are used to represent important system functionalities. Every card holds the user scenario description related to the current functionality, its priority and time estimate. While priorities are provided by customer, the estimates are set by developers team. This stands in opposition to traditional methods, where the estimates were set by top management and therefore occurred unrealistic.

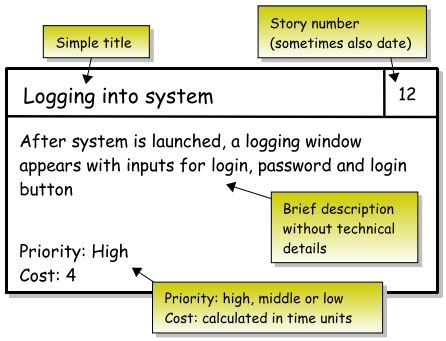


Figure 2.4: User Story card example

What is more, User Story card (small piece of paper) is a comfortable and easily understandable tool: it can be moved or trashed, one can quickly create a new one or add something to existing one, everybody can see it and take to hand, etc. These advantages seem important as customer, usually not a computer science specialist, actively participates in the planning

process. An example of a card is shown in Figure 2.4.

Alike User Stories, Backlog (Scrum) is another technique that supports planning. Backlog is a list of tasks to complete such as desired features, bugs to remove etc. They are prioritized, estimated and grouped into increments. However, the grouping can change whenever it is needed. Any modifications, agreed by developers and customer, are performed by *Product Owner* only, who plays a role similar to *middle-up-down manager*, described in 2.1.

Another techniques are *Time Boxing* and *MoSCoW* proposed by DSDM. The idea behind the first one is splitting the project into prioritized portions with given budget and delivery date. Whenever time or budget is running out, the features of lowest priority are omitted. This is compliant with well-known *parapeto principle*, already discussed in Section 2.1. This technique is supported by *MoSCoW* technique, which relates to setting the priorities. Being an acronym for *Must have, Should have, Could have, Would have* it establishes four groups of requirements according to their business values and purposes. This and other similar techniques for prioritizing tasks are present in various kinds of self-study and self-books. One of them was mentioned in Section 2.1.3, along with the so-called *Put First Things First* habit.

Frequent planning meetings involve customer in the development process. The role of customer includes: specifying user scenarios (User Story Cards), prioritizing them and assigning for iterations, explaining misconceptions and resolving business problems. The last two ones, according to agile methodologies, should not be limited to meeting sessions only. Misconceptions are often identified during implementation phase. Similarly, implementation may involve numerous business decisions, such as these concerning GUI design and operation details. Thus, all agile methodologies emphasize the importance of frequent and efficient communication with a customer, at every moment of project life-cycle. To attain this, *Onsite customer* (also *Whole Team*) (XP) practices or *Ambassador User* and *Advisor User* (DSDM) roles are introduced. They assume that a customer or his/her representative shares his/her knowledge and viewpoint in a daily work of development team. Not only reduces it the project risk but also increases the quality (in a sense of usefulness) of the final product.

The quality is also referred by another practice which is *Customer Tests*. Although the concept was known in traditional approach, agile methodologies modify it in two following aspects: automation and customer involvement. The first one assumes that the acceptance tests should be automated and run frequently whenever it possible. The second one, places the customer (or user) in a role of test suite designer and sometimes, even, in a test writer. This is possible thanks to *Onsite Customer* practice.

In order to facilitate communication and better explain technical concepts to customer, *Metaphor* practice (XP) was introduced. A metaphor is a meaningful description of common concept regarding technical issues, usually the way the software work. An example metaphor might be a spider web for illustration of Internet and its functioning. It is believed, accurate metaphors are a valuable part of product design.

Practices of work organization

This group of practices relates to developers' work organization. The most fundamental one is *Continuous Integration* (XP) also known as *Regular Builds* (FDD) and *Daily integration,*

Rapid Builds (LSD). This practice suggests frequent (at least one per day, XP suggests several times a day) automated integration of the developed code. This ensures that the code is up-to-day and the software can be demonstrated to customer at any time. In addition, frequent integration provides early feedback on any integration errors, which are especially troublesome in the late stage of development, when the software structure is complex, bugs difficult to track and their resolutions limited by various dependences. This situation, sometimes referred to as *integration hell*, may led to project delay, which is one of the symptoms of the LOOP syndrome. Continuous integration considerably reduces this risk. Furthermore, by early indication of existing errors, it increases the quality of developed code. Finally, it prevents from *code freezes*, when all efforts are concentrated on repairing broken code, not on adding new functionalities, being the actual project goal.

Another significant practice is *Collective Ownership* (XP). It assumes that the developed code can be accessible and modified by anyone from the developers team. In fact, the code is a property of the whole team. This stands in opposition to certain traditional habits, when the code creator was the only owner. Collective ownership benefits in several ways. Firstly, it encourages developers to perform code reviews and enhancements, which increases its quality. Secondly, it solves the common problem of code fixes or new functionalities waiting till code owner finishes current tasks. This advantage of Collective Ownership is especially valuable during integration builds, as integration errors should be fixed immediately. In addition, it contributes to elimination of code freezes as new functionalities are not blocked by one busy developer. Finally, Collective Ownership causes a diffusion of project knowledge among the whole team. This phenomenon increases the project resistance to the loss of single developer, caused by his/her sickness or leave. In traditional approach, a loss of developer, especially responsible for a core project part, could lead to project fall.

Collective Ownership requires setting work standards. Practice *Coding standard* (XP, LSD) answers this need. It ensures that everybody in the project speaks the same language. In other words, the code looks familiar and is understandable to everybody, giving no clue, who wrote it.

Affected by several factors (see Section 2.1), Agile Manifesto emphasized the importance of individuals and interactions between them. Agile methodologies developed this postulate into a few practices and techniques, whose role is to intensify people communication and maintain individuals productivity. Some of them, like Metaphor, relate to planning and customer collaboration. The others like, *Osmotic Communication*, *Information Radiators*, *Stand-up meetings* and *Sustainable Pace* are meant for developers team support.

Osmotic Communication (Crystal), also known as *Common Room* (Scrum, AUP), *War-Room*, *Open Workspace* (XP), *Radical collocation* techniques and included in *Feature Teams* (FDD) practice, opposes the traditional cubicles-based workspace organization. Agile methodologies prefer to place people in the open space with no communication barriers. Open space involves suitable furniture such as round conference tables and large whiteboards. The furniture deployment plays also an important role. For instance, the developers' desks are located in close proximity to each other, to facilitate team communication. This is a base for *Osmotic Communication*, which is actually a phenomenon emerging from the workspace organization.

This phenomenon relies on the diffusion of the project knowledge by overhearing background conversations and communicating with a several people at one time (one could call it *multicast communication*). As result, one can, among others, discover conceptual conflicts, pick up hints (e.g. configuration instructions) and get informed on the work status. On the other hand, Osmotic Communication has its disadvantages. Firstly, a background conversation may be disturbing, especially when one needs to concentrate on difficult problem. Secondly, one can pick up false information or be overloaded by a number of irrelevant data, missing the key issues.

Information Radiators (Crystal, LSD) is a technique closely related to the open workspace. *Information Radiator* is a display of valuable project information, usually placed on whiteboard in a well-visible place. The most common radiators includes project backlogs, to-do lists, diagrams and progress charts. One of them is *burndown chart*, which shows the status of completed tasks in the time perspective. Figure 2.5 shows an example burndown chart, with a trend line indicating the ideal work progress. The actual burndown line marks the real progress. When the burndown line is below the ideal trend, the team works faster than it was estimated. Otherwise, it works slower and might be late for a deadline (as shown in the example).

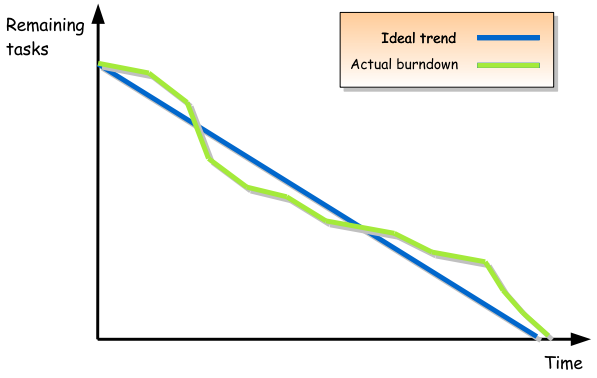


Figure 2.5: Burndown chart example

Another practice, which emphasizes the role of communication, is *Stand-up meetings* (XP), similar to *Daily Scrum Meetings* (Scrum). The purpose of this meetings, usually held every morning, is communicating entire team. More precisely, it is an opportunity to express any problems, concepts and other issues that require team attention. This keeps everybody informed on the current project status and allows for saving time required for individual knowledge acquisition. In addition, it allows for a coarse daily work organization and tasks assignment. To make the meetings efficient and short (their duration is approximately 15 minutes) team members stand in a close circle. This is where the name of stand-up meetings comes from.

The importance of individuals and their needs is the core of *Sustainable Pace* (XP) practice. One of the LOOP syndrome's elements is overtime work emerging from unrealistic plans, unstable requirements or other reasons such as implementation problems like integration hell. Based on experiences and observations, Agile methodologists concluded that overtime work brings more harm than benefit. Although overtime might be efficient and productive, in a long term it burns-out developers, which drastically reduces productivity and quality. The proposed *Sustainable Pace*, former *40-hours week* practice (XP), tries to control this issue. It states that the long term productivity can be maintained only by a reasonably managed team, with work times adjusted to its needs and abilities. Overtime should be limited to

emergency cases (project release) and other unusual purposes. This reminds of *Sharpen the saw* habit presented in Section 2.1.3.

Practices of programming

Practices of programming support developing a high quality code. It is achieved on several levels from code reviews to introducing design patterns. Here, the following practices are presented: *Pair Programming, Side-by-Side programing, Test-Driven Development, Refactoring, Simple Design*.

Pair Programming is one of the most recognizable XP practice. The idea is to assign a couple of programmers to one work station. One of them becomes a *driver*, the second one is assigned a *navigator* role. These roles can be changed anytime. While the driver owns the keyboard and types the code, the navigator tracks and controls it. This allows for detecting spelling mistakes, implementation errors, algorithms bugs and conceptual weaknesses, at the same time, increasing the developed code quality. On the other hand, some say these benefits do not balance the double costs of code development, caused by two programmers working on a task that one could successfully complete. However, these statements do not take under consideration two facts. Firstly, a pair of programmers tend to work faster than a single one. Secondly, a low quality code can consequence in considerable costs. For instance, a bug or misconception detected just before product release, expose the project to the risk of high contract penalties.

In order to verify theoretical speculations, several experiments were organized. One of the first ones was Nosek's experiment [49]. It lasted for 45 minutes and engaged 15 full-time professionals. According to gathered data, pairs were able to finish task 40% faster, producing code of greater quality and readiness. In order to strengthen these results, another experiment was organized [72]. It stated that pairs of programmers were approximately 44% faster than single programmers in completing tasks. With regard to code quality, pairs reached an average of 89% passed test cases, whereas individuals scored 75%. These promising results were questioned by Nawrocki's experiment [47] based on four tasks. Only one of them reported results comparable to these from Nosek's experiment. The other three showed a minor superiority of pair programmers. This confirms Nawrocki's conclusion that more experiments, preferably based on real industrial conditions, are required.

Another group of experiments and studies relate to the educational application of Pair Programming. During Pair Programming session partners exchange their experiences, learn partner's techniques and habits. The gained knowledge can be both explicit and tacit (see Section 2.1.2). While the first one, systematic and objective, can be passed on also through lectures or books, the second one, related to individual skills and insights, is difficult to express and require, often long, practice. Pair Programming is believed to shorten this time. One of the experiments was described in [39]. It aimed at confirming Pair Programming usefulness in introductory course. The gained results stated that students from *pairing* class scored more points during final exam (median was 83% of all possible points) than students from non-pairing class (median was 75%).

While Pair Programming is a proposal of XP methodology, Crystal methodology came up

with *Side-by-Side Programming* practice (hereinafter referred as SbS Programming). That is, two programmers working on their own computers but sitting close to each other so they can see partner's screen. This allows them to benefit both from Pair Programming and individual work on separate problems. Furthermore, it supports peer code reviews. In Pair Programming, the review is done with the presence of the code's author. However, this is not always necessary. When SbS Programming is employed, the review can be done individually. Whenever the code's author support or presence is required, a brief pair programming session is immediately started. Therefore, this might be seen as an optimized approach to code reviews. Finally, placing programmer is close physical proximity enforces Osmotic Communication practice. Alike Pair Programming, SbS Programming efficiency become a research subject. In [46] authors present results from experiment aimed at comparing these two programming practices. According to the report, side-by-side programmers completed their task 40% faster than individuals, whereas pair programmers scored only 26%. Another criterion was the code familiarity. In this case, pair programmers outperformed their side-by-side colleagues by scoring 20% better. Unfortunately, nothing can be said about the quality of performed work. The experiment did not included such criterion.

Another agile practice closely related to programming activity is *Test-Driven Development* (XP initially) [4, 1], sometimes known as *Test-First Development*. As mentioned, due to its high efficiency and developers appreciation, it was developed into an independent methodology. The idea behind it, clearly manifests the second name: automated tests are written before the code. This benefits in several ways. Firstly, it enforces developers to deeply consider the code's functionality before it is actually written. In some way, this reminds of *Begin with the End In Mind* habit discussed in Section 2.1.3. Secondly, tests naturally restrict complex and low-quality designs. The simpler design, the greater number of interfaces and independent modules it has, the easier and faster, it becomes, to write and pass all tests. Thirdly, automated tests provide immediate feedback on the developed code (e.g. bugs indication) and can be used as a measure of progress (e.g. percent of passed tests). Furthermore, tests are often treated as a form of documentation showing the possible usage on the example of implemented test cases (alike library tutorials do). Lastly, practicing *Test-First Development* develops valuable habits such as continuous maintenance and creation of new test cases, on the contrary to traditional testing, done at the end of implementation phase and often abandoned or coarse due to upcoming deadlines.

Test-Driven Development is a base for another appreciated practice which is *Refactoring* [99] (XP, LSD initially) [3, 17]. Martin Fowler [17] gives the following definition for it: *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*. In other words, refactoring is a disciplined transformation of the code body, not altering its functionality. Although a single refactoring is a little restructuring, a sequence of refactorings results in considerable improvement in the code structure and clarity. To verify that the observable behavior remains unchanged, the existing test cases are launched, preferably after every single refactoring.

In most cases, refactorings are related to various software patters such as well-known design patterns. From this perspective, a single refactoring is a recipe on how to transform the

given code into a proven pattern. To make this easier, refactorings are usually described with their application, mechanism and expected benefits. The catalogue of refactorings is numerous. Among many others, it includes the following: *Rename Method (Field, Class, Parameter, Variable), Extract Method (also Interface, Class), Move Method (Field), Replace Parameter with Method, Collapse Hierarchy, Pull Up (Push Down) Method, Replace Conditional with Polymorphism.* While the first ones describe simple changes which improve readiness of the refactored code, the last ones define recipes for more advanced restructuring and redesigning of the given code.

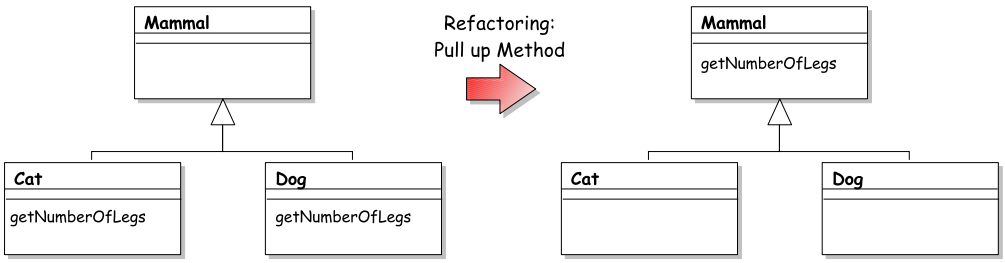


Figure 2.6: The example of Pull up Method refactoring

To illustrate the idea of refactoring, Figure 2.6 presents the example of *Pull up Method*. A common method `getNumberOfLegs` for all subclasses (here `Cat` and `Dog`) can be moved (pulled up) to the `Mammal` superclass (every mammal has some legs), in this way, simplifying the design. A detailed description of this and other refactorings may be found in [17].

Simplicity is one of the most appreciated values in agile movement. As emphasized, it is maintained by several practices such as Test-Driven Development and Refactoring. In fact, it influences a number of agile project aspects from people iterations, through planning and project managing to coding and testing. Therefore, some methodologies proposed separate practices dedicated to the value of simplicity. These are: *Simple Design* and *Simplicity* (DSDM, AUP). All of them suggest most simple and effective solutions that meet customer expectations. On contrary to complex ones, they are more open to any, even radical, changes. In addition, they are less expansive and can be delivered faster. In some way, agile simplicity follows the habit of *Put First Things First* (see Section 2.1.3) by concentrating on the most important issues at the given time, which usually simplifies the problem resolution.

2.3.2 See the whole. Systems thinking

In many aspects, the agile approach seem to benefit from systems theory, particularly the field of systems thinking [70, 7]. This relation was also noticed by creators of LSD methodology and influenced their proposals for project estimation and contracting. As result, agile methodologies indicated resolutions to several existing problems, such as the *problem of sub-optimization* and the *problem of 4 variables of software project.* On the other hand, systems thinking affects also other aspects of agile methodologies such as their structure and relations between the proposed practices.

The problem of suboptimization

Although systems thinking is not a novel discipline, a typical approach to complex problems is to break them into smaller subproblems, which can be solved individually. According to [55], this may lead to widely-known problem of suboptimization, also referred as *tragedy of commons* [97] in various scientific disciplines. The essence of this problem is a conflict over common resources between subsystems. As a result of this conflict, the optimization of the individual subsystem outcome may deteriorate the outcome of other subsystems. Thus, in general, the system remains unoptimized.

To illustrate this problem, a *Prisoners' Dilemma* [57, 56] game is often given as an example. The game assumes two prisoners trying to minimize the total punishment for, let's say, failed escape attempt. To achieve this, they can cooperate with (e.g. stay silent) or betray the other prisoner. However, each one aims at maximizing his/her own benefit, without taking under consideration the benefit of the other prisoner (a suboptimization metaphor). As in most cases, an egoistic behavior guarantees the greatest benefit, both prisoners decide to betray. Thereby, they both get severely punished. On contrary, a decision of cooperation concerning their all benefits (a global optimization metaphor) results in a mild punishment only.

What stays behind the problem of suboptimization is a basic systemic principle stating that "the whole is more than the sum of its parts". This statement indicates a significant role of the interactions between the system parts. In suboptimization approach, these interactions are not considered. A global optimization includes both parts and interactions, in this way, providing a greater outcome. While the traditional software engineering methods approach the development process by suboptimization, agile methodologies emphasize the importance of system parts' interactions and employ global optimization.

The problem of four variables of software project

The problem of four variables (sometimes also known as dimensions) of software project [28] affects every project and development team. The four variables are: *Scope, Quality, Resources* and *Time*.

Scope denotes the amount of planned efforts in the project, in particular the scope of functionalities to implement, test and release.

Quality describes both internal quality (refactored readable code which is easily to extend) and external one (no bugs, product meets all customer expectations).

Resources represent people, methods, tools and finances available for the project. Sometimes, Resources are replaced by Cost variable, which covers people, methods and tools, as these greatly depend on available finances.

Time is the amount of work hours available until the established deadline.

These four variables are driven by a tenet similar to the rule of connected vessels: when one variable changes its value (water level in one of the vessels), others change their values consequently (water levels in other vessels). In other words, there is no way to control all four variables at the same time, only three at most. The ignorance of this rule is a common cause of projects failures and main reason behind the LOOP syndrome (see Section 2.1.1) (e.g. setting scope without concern of other variables may result in serious delay or overtime hours).

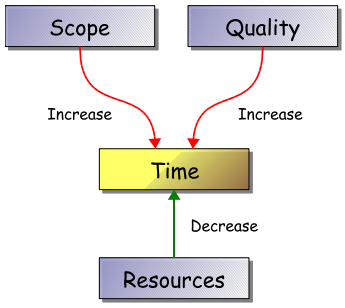


Figure 2.7: Four variables of software project

Figure 2.7 shows the most common dependencies between the variables. In most cases, *Time* is that fourth variable which cannot be controlled. Therefore, to some extend, the figure presents the interactions from a *Time* variable perspective. The common scenario assumes increasing the *Scope* of project which automatically extends the required *Time*. Similarly, to deliver a higher *Quality*, an additional *Time* is needed. To cope with an increasing amount of required work hours, extra *Resources* can be added. This usually reduces the required *Time*. Other option is to postpone a deadline, so that the extended *Time* is acceptable to customer. To conclude: the changes of *Scope* and *Quality* cause the change of at least one of the other two variables, that is *Time* and *Resources*. It is not possible to maintain both of them on the same level. Moreover, it is clear that while *Time*, *Quality* and *Scope* are proportional to each other, *Resources* variable is inversely proportional to all of them.

Obviously, the presented problem is unsolvable. However, it can be properly controlled when rules behind the four variables are known. What is more, this knowledge must be learned both by developers and customer. However, this significant requirement often remained unknown in traditional methods. The roles of developer and customer used to be limited to completion of specific process tasks (developer) or contract negotiation (customer).

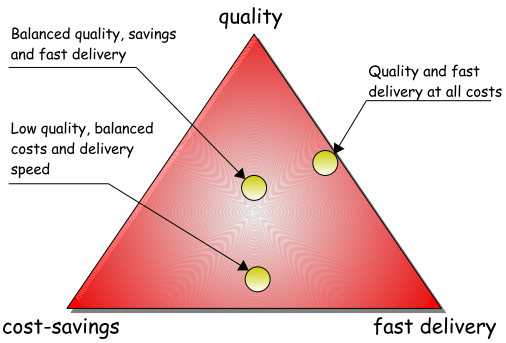


Figure 2.8: Priority triangle with examples

On contrary, agile methodologies impose an active involvement of project participants in the planning process, which is driven with respect to the problem of four variables. To control it, a number of already presented in Section 2.1.2 practices and techniques are employed such as *Small Releases*, frequent *Release* and *Iteration Planning*, *Time Boxing*.

Another interesting technique, directly concerning the problem of four variables, is *Priority Triangle* (Figure 2.8). It is a triangle whose vertices correspond to three crucial project values: quality, cost-savings and fast delivery. Based on it, a customer can define the priorities for the project values. This is done by placing a point inside a triangle. The closer the point is to a vertex, the higher priority the corresponding value posses. Obviously, the priorities selection is balanced: it is not possible

to select high priorities for all of the values. This is a direct mapping from the rule of four project variables. Figure 2.8 shows the triangle with three example priorities definitions. Be that as it may, *Priority Triangle* plays also an educational role. By clearly illustrating the idea behind project variables and values, this technique encourages customer to revise his/her unrealistic expectations and adjust them to both actual business needs and available resources.

Synergy of practices

Systems thinking affected also the structure of agile methodologies. As discussed in Section 2.1.1, they are built upon a set of practices and techniques. However, there is also one more significant element: their synergy. A popular belief states even that the synergy determines the real effectiveness of a methodology, not individual practices. While the notion of synergy emerges directly from systems theory, it appears also in other areas that contributed to agile methodologies. Section 2.1.3 mentions a *Synergize* behavior emphasizing the power of human collaboration, its effectiveness and creativeness factors.

According to XP, the synergy emerges from the *extreme* use of suggested practices. However, this synergy is possible due to a specific graph of relations and dependencies between the practices. Figure 2.9 illustrates them on XP example. The practices are organized in three groups which correspond to project areas they support (see Section 2.1.2): green indicates practices of planning and customer collaboration, blue is for practices of work organization, green marks the practices of programming. A single practice can interact with practices from all groups. In addition, some of the shown dependencies can vary depending on the way the XP is adopted by developers team. For instance, the Figure 2.9 shows the *Metaphor* practice can interact with *Customer Tests*. However, the tests can be defined with no reference to elaborated *Metaphors*. On contrary, *Refactoring* strongly relies on *Test-Driven Development* (TDD) practice. TDD acts as a code guard that keeps it running and bug-free during the process of code restructuring (refactoring). On the other hand, a refactored code is usually easier to test. This shows the reinforcement of both practices emerging from their co-existence. The cancellation of TDD practice would have serious consequences turning *Refactoring* into an uncontrolled process likely to break or spoil the release code with severe bugs.

There are two important issues concerning the complexity of the interactions graph. Firstly, the graph in some way maps the complexity of the real world and software process particularly. While this increases the efficiency and flexibility (for adaptation purposes) of methodology, a process based on the synergistic dependencies is difficult to express and measure. This directly leads to second issue. Agile methodologies are usually meant to be adopted to individual needs of every project and its participants. However, as measurement and analysis are aggravated, the consequences of the adoption is difficult to anticipate.

2.4 Distributed Agile Development (DAD)

One of the fundamental values behind agile movement is communication and people interaction. According to Agile Manifesto [94]: *The most efficient and effective method of conveying*

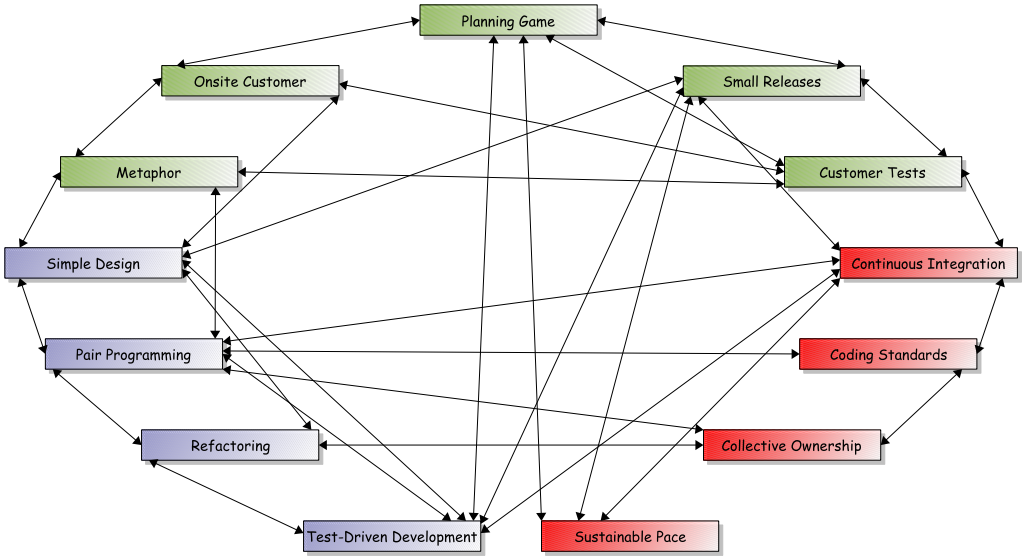


Figure 2.9: XP practices dependencies

information to and within a development team is face-to-face conversation. This concepts affected a variety of agile practices from *Release and Iteration Planning*, *Onsite Customer*, trough *Osmotic Communication*, *Common Room*, *Information Radiators* and *Stand-up meetings*, to *Pair Programming* and *SbS Programming*. As direct communication requires close physical proximity of project participants (preferably reduced to one room and even one table), agile methodologies imposed certain limitations concerning project logistics and organization.

There is also another trend in software development that gained popularity and affected software production in latest years. Offshore development [32, 60] was the response for growing costs of production in West World. While it concerned a variety of branches, software development was also included inevitably. It resulted in locating development centers in more cost-effective countries (such as India) and a growing number of projects realized by geographically distributed teams. Other reasons behind distributing software development include:

- Local availability of qualified personnel - this is, beside lower development costs, the most significant factor for founding development centers in distant locations.
- Development of projects for distant customers - the "global village" encourages companies to broaden their area of business resulting in projects conducted for distant customers. This involves distant collaboration with respect to project analysis, planning and feedback. To improve customer collaboration, some companies locate a part of development team in close proximity to customer, which causes team distribution.
- Specialization of development centers - this results in development team cooperating with distant experts and specialists as they are not available on project site. As most projects require specialists of various technologies and disciplines, specialization strongly contributes to team distribution.

- Individual limitations of development team members - this includes sickness, home responsibilities (children care), delegations, conferences and many others preventing a member to locally participate in a development process.

The distribution of developers team apparently stands in opposition to the agile requirement of close physical proximity of project participants. Thereby, the requirement started to be reckoned as a severe drawback of the agile approach. The attempts to overcome it refer mainly to the most popularized agile methodology, which is XP. The efforts begin to form a new scientific discipline known as **Distributed eXtreme Programming** (hereinafter referred as **DXP)**. One of DXP research domains is building communication channels capable of substituting face to face communication in distributed setting. This is achieved by either employing existing tools of general applications or developing dedicated solutions. While more agile methodologies gained global appreciation, the DXP moved into **Distributed Agile Development** (hereinafter referred as **DAD**) discipline. This section presents the reported work and achieved results.

While DAD concentrates on supporting agile development in distributed setting, the achieved results can be beneficial in wider perspective as well. Among the expected benefits, one can point out:

- maintaining development pace in case of individual limitations of developers, such as sickness
- increasing professional activity of disabled people
- supporting e-learning and e-education concepts [58, 76]
- reducing costs and increasing time efficiency of commuters (a solution for a major problem of developing cities which is an intensive traffic)
- increasing mobility of project members (e.g. participating in a distant conferences and meetings)
- reducing initial costs and risk while starting new companies (no office required)
- supporting the concept of e-work (not only e-development) based on people working from their home sites

2.4.1 Supporting tools of general application

According to [66] supporting eXtreme Programming in dispersed environment is "possible with a few simple non-custom, widely-available tools (screen sharing, Internet-based audio communications)". It seems that a majority of people driven agile practices may be supported by video and audio conferencing tools accompanied by simple text chat communicators. The number of such applications is intensively growing (e.g. Microsoft NetMeeting[95], Skype[100]). To enrich all kinds of workshops and concept debates as well as for demonstration purposes application (or desktop) sharing approach might be utilized. In addition, whiteboard

solutions that allow for a real time exchange of graphical information (e.g. sketches, models) might be found useful. All these functionalities have been implemented in Microsoft NetMeeting framework which makes them easy available. For practicing Pair Programming or SbS Programming distant developers may employ desktop sharing tools, accompanied by at least voice or text communication channels.

In general, process driven practices aim at automaton and simplifying the project management and therefore they depend on the available software. At the moment, there is a number of systems [79] satisfying the needs of continuous integration principle. They include CruiseControl [80], BeetleJuice [78] or Luntbuild [93]. They are usually integrated with version control systems (CVS, Subversion [102]), which implement Collective Code Ownership property and perform the synchronization of project products.

Among other agile practices the Crystal's technique of Information Radiators appear appealing and promising in the non-collated setting. As a replacement of whiteboard drawings, notes or sticky-papers, developers can successfully arrange web forums, mailing lists, issue trackers, wiki or even blog pages. The last ones seem to gain much appreciation in the development environments.

The list of usable solutions extends Sun's Java Web Start technology [91] that allows for seamless lunch of remote server-stored application. It can be adopted both for demonstration purposes and updating remote customer's software.

Worth mentioning concepts of e-Presence and e-Awareness have been presented in [10]. They are the adoption of Osmotic Communication Crystal's core property to distributed ground. It is achieved with a use of audio-visual support accompanied by standard internet communicators and chat boxes. Experiments show their ability to successfully replace Osmotic Communication with its distant equivalent.

2.4.2 Dedicated work and solutions

As mentioned, first attempts to overcome close proximity requirement are mainly related with DXP discipline and Pair Programming practice. One of them is TUKAN [67] environment build upon a shared code repository, version management and distributed integration system. In addition, it supports Pair Programming with a devoted code editor as well as video and voice connections. Another project is MILOS ASE [38] equipped with a Story Management System meant for team coordination and information routing. For distributed pair programmers it employs video conferencing and application sharing approaches provided by Microsoft NetMeeting. An interesting concept has been presented in "Office of Real Soon Now" project [5]. It extends Internet conferences into tele-immersion approach by building an illusion of a real office for distributed teams based on large wall displays.

These solutions are sometimes regarded as "heavy" due to costs and high network requirements. Thus, more simple and "light" tools gained interest. One of them is Sangam plug-in [25] to Eclipse IDE including editor, resource and basic refactoring synchronisation. Another one is Pair Programming plugin [15] for Jext Editor [109] supported by web based integration architecture including build-and-test server and co-ordination task log. A novel concept of transparent video interface is introduced by Facetop project [65]. It integrates and lines up

side-by-side the video streams of distant programmers, overlying them over the transparent desktop at the same time. Not only can they see each other pointing to some spot at the shared screen but also manually control the mouse pointer. In order to join the advantages of Sangam and Facetop projects, [44] proposes a prototype combination of these tools.

The organization of tasks and User Stories was also the aim of the Storymanager tool [30], developed as a plugin to Eclipse framework. Similar solutions but with regard to system features were presented in [86] as the first FDD implementations. These server-side applications provide simple HTML interfaces for managing FDD driven projects such as plan view, implementation progress or summary and trend reports. Although they were developed as an automaton for standard FDD process, they perfectly suit the needs of the distributed approach.

The number of dedicated solutions that may support distributed project management is intensively increasing. While they aim at assisting in standard co-located development, their architecture make them suitable also for scattered teams. The range of issues they cover is considerably large from simple appointments managers, work schedulers and report generators to more specialized systems taking care of method specific values such as XP's User Stories. The most advanced ones are XPlanner [106], XPWeb [107]. In case of other methodologies, the creation of similar tools is just a matter of time.

An interesting approach to *Customer Tests* practice evinces a FitNesse framework [40] [87]. Based on wiki, FitNesse enables customers to specify in form of HTML tables their expectations (test cases) towards functionality of developed software. Afterwards developers implement dedicated proxy classes which apply customer test cases to developed classes. In result, customer learns on the project status. On the other hand, developers are provided with additional testing scenarios which meet customer expectations. While this software is intended for collated teams, due to client-server architecture, it also suits perfectly for distributed ones.

2.5 Summary

In this chapter, the movement of agile methodologies was presented, which emerged from the disappointment with traditional methods of software development, often leading to a LOOP syndrome. Formulation of agile movement was affected by a variety of elements and concepts. The most important include gathered experiences and performed reports, systems thinking, Japanese concept of knowledge-creating company and even self-help literature with their philosophy and mind tools.

The main assumptions of agile movement were formulated in 2001 in Agile Manifesto. They emphasized the following software project values: communication, people interactions, customer collaboration, working software and quality. These postulates affected the structure of agile methodologies and set of proposed practices. Additional attention was put to the synergy of practices emerging from their extended net of dependencies and relations. Although it is claimed to empower the efficiency of agile methodology, it considerably aggravates the measurement and systematized studies.

Agile methodologies came with resolutions to some of the existing problems. An example is the problem of four software project variables. On other hand, they imposed certain limitations on the developer team. One of them is the close physical proximity of project participants. This limitation seems serious when one takes under account the modern trend of distributing development teams. While this problem existed before, it was not a considerable one when teams utilized traditional methods of development. These methods were based on a detailed documentation and a formal process which were not greatly affected by the team distribution. Contrarywise, agile methodologies are based on intensive communication, which is directly affected by the distribution of team.

To enable Agile Development in distributed setting, the requirement of close physical proximity of project participants must be cancel out or, at least, relaxed. To achieve this, communication support is needed. Several supporting software tools are reported to exist. However, the provided support refers only to selected aspects of Agile Development. What is more, the support is unsatisfactory due to several reasons such as small number of specific agile features, small number of communication channels or poor connection quality. A significant problem is also the latency and connection speed, which is a key factor when remote collaboration is considered. Another interesting issue is the analysis of the requirements for the supporting tools. People interactions and communication involve a human factor, which makes the requirements space a fuzzy one. Therefore, it is difficult to perform the analysis and requirements specification a priori. As result, Agile Development is rarely practiced by distributed teams, despite the increasing popularity of the agile movement.

Chapter 3

Conceptualization of Distributed Agile Development

Software development process consists of numerous aspects of various types: from technical ones such as hardware and software, through organizational and methodological issues to people behavior and psychological nuances. Obviously DAD added several new variables. Among them, team spatial distribution and team communication patterns belong to the most significant ones. As they directly affect other variables, they complicate the picture of the whole process.

As DAD is a new research area, these issues are not exhaustively explored yet. Therefore, it seems purposeful to organize them into a general conceptualization. While it attempts to classify the DAD domain, it also assures that no crucial issue is omitted in the conducted study. Moreover, by emphasizing several issues, it allows for introduction of the specific DAD model.

It must be clear that it is acceptable here to argue with certain observations and simplifications. Including too many details will made the conceptualization unreadable. In addition, future extensions and modifications of the conceptualization are expected as a result of constant development of all: agile movement, software engineering and technology. Still, beside possible disadvantages, it positively carries out the role of a roadmap for further, more detailed, studies (such as this dissertation).

3.1 Conceptualization

The proposed conceptualization is done in three following aspects:

1. A map of the basic DAD elements joined with generalization-specialization and aggregation relations (a kind of extended terminology)

2. Mutual relations between elements at the given level of specification, being the essence of an agile process

3. Grounding of agile practices in the whole

3.1.1 Elements map

Figure 3.1 presents the map[1] of basic DAD elements.

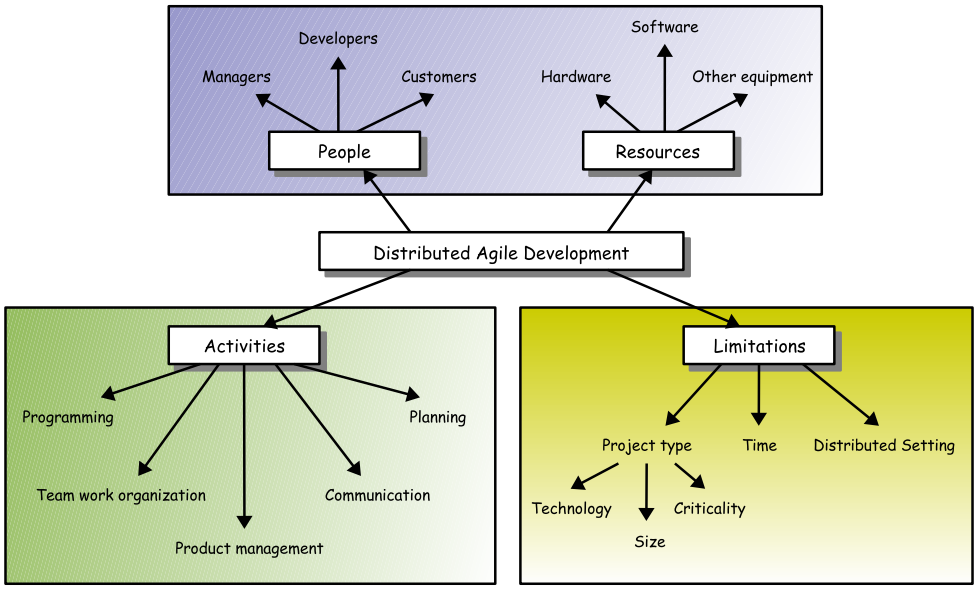


Figure 3.1: The map of conceptualization elements

In the DAD structure, conceptualization distinguishes four groups: People, Resources, Activities, Limitations.

People and Resources

People and Resources denote real world objects that are a part of software development process. Within People group, three general types are specified: Managers, Customers and Developers. These types cover the most important people roles in DAD. As for Resources, they can be divided into Software, (computer) Hardware and Other Equipment that may affect software development. An example of Other Equipment is a whiteboard, which may be used for project tracking.

[1]The form of the concepts map is influenced by two different techniques: UML and Mind-Mapping. It reminds an UML class diagram in a sense of the aggregation and generalization-specialization relations. The presented form is an informal variation of class diagram which is more suitable for the work purposes. As for Mind Mapping technique, it suggests building the conceptualization structure in a form of a tree, with a central (crucial) element as the root and related aspects in the tree branches and leafs. It is worth to add that Mind Mapping technique was recently brought onto agile ground [6, 35] [24].

Activities

To make a use of the available Objects, a set of Activities is required. In fact, this set builds a software methodology. The form of these activities directly emerge from the accepted development methodology.

For example, Programming includes all programming activities such as coding, testing, debugging and designing. One can argue on this selection as designing is often regarded as a separate activity. This is partially true for the traditional software development methods such as waterfall model. In agile approach, the design evolves incrementally through frequent code refactorings and intensive communication between project collaborators (i.e. customers and developers). Another specific activity typical for Agile Development is Communication. While it is present in traditional methodologies, Agile Development puts it into the front, which results in a considerable number of communication principles. As for Team Work Organization, it denotes such aspects as work synchronization, workspace preparation and work pace maintenance. Product Management involves marketing and contract issues such as product release, installation, support and others. Finally, Planning activity stands for the agile planning cycle with considerable customer involvement and time estimation techniques.

A question arises on how these activities correspond to the presented agile practices. An activity groups a set of practices. The grouping is done based on mutual dependencies of practices, their goals and common development aspects. For instance, within Programming activity one can point out Pair Programming and Test-Driven Development practices. They both directly refer to code development. This classification corresponds also to the organization of Section 2.3, in which the agile practices are presented.

Limitations

Limitations emerge from the specificity of environment in which the activities (methodology) operate. Obviously, DAD adds a new limitation of work environment which is Distributed Setting. Among other limitations, Criticality requires a brief explanation. Criticality states the project significance, its risk and estimated costs of improper behavior (e.g bugs). Higher criticality usually requires more reliable solutions, additional support and proper strategy of the integration and project control.

3.1.2 Elements relations

The identified elements do not exist in separation. They are involved in the relations with other elements. These relations constitute their roles and meaning for the DAD process. The second aspect[2] of the proposed conceptualization, which is presented in Figure 3.2, attempts to identify these relations. An interesting observation is that the conceptualization corresponds to Systems Thinking approach (see Section 2.3.2). As already explained, this approach introduces the notion of synergy [83], in which the effect of two elements acting

[2]This aspect of the conceptualization is influenced by UML collaboration diagrams and ERD model. They both depict mutual interactions between diagram elements, such as objects and entities. Both of them attribute the relations. In the proposed conceptualization the attributes of relations are replaced by relations types.

together is greater than the sum of effects of these elements acting individually. This concept can be applied also to the identified elements. The possible relations, which are depicted in Figure 3.2, indicate that the elements do not exist and act in separation, but as a part of the whole synergistic system.

In this aspect, the conceptualization consists of Elements and Relations. Elements are already identified People, Resources, Activities and Limitations. Relations are classified into three following named types: Participate In, Require and Affect. The use of verbs here instead of nouns improves the readiness of the conceptualization. Table 3.1 contains a brief description of these relations.

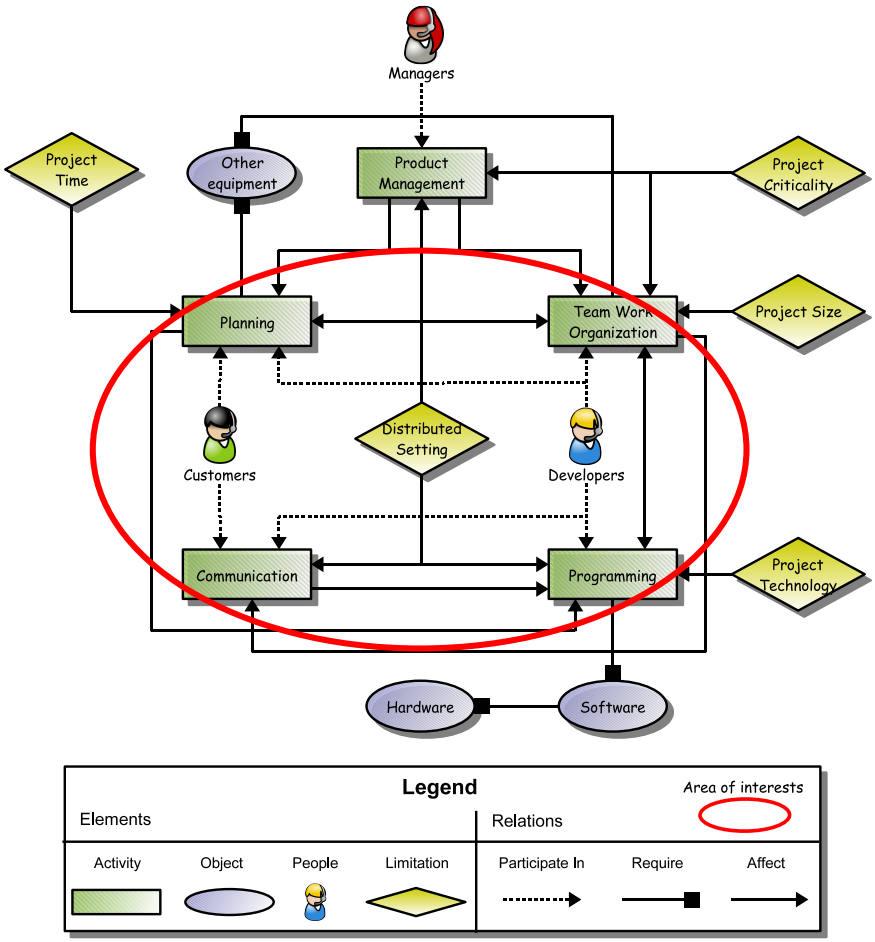


Figure 3.2: Relations between conceptualization elements

Table 3.1: Conceptualization relations specification

Relation	**Specification**
Participate In	Every activity must be participated by a human. By participating in an activity people actually collaborate with each other. As activities group practices, participating in an activity means practicing at least one practice within that activity.
Require	An activity may require a resource to be consumed. In addition, a resource may require other resource. An example is Programming which requires Software, which requires Hardware.
Affect	Activities are not performed in isolation. They interact with other activities. The interaction usually means affecting other activity by providing significant data or work frames including environment and project goals. As the Affect relation is single-directional, it actually constitutes a work flow. For example, Product Management affects Planning, which affects Programming. This is a typical project work flow. Another case for the relation emerges from the existence of limitations. Undoubtedly limitations have a strong influence on project activities. For example, Team Work Organization activity strongly relies on the Project Size limitation. In case of large projects, the organization of team work (e.g. synchronization) is much more complex than for a small project. In fact, Project Size affects Team Work Organization.

The central element in the conceptualization is Distributed Setting limitation. Undoubtedly, this illustrates the unique specifics of DAD. Distributed Setting affects all DAD activities. As activities consists of agile practices, the setting affects, in fact, some of the grouped practices.

Another unique aspect of DAD is the role of Customers in the project. According to the conceptualization, Customers participate in Planning and Communication. This is on contrary to traditional methodologies where the role of customer is limited to requirements specification. Similarly, agile methodologies extend the role of developers. In DAD, they actively participate in Planning and Communication. What is more, they directly collaborate with customers through these activities. While traditional methodologies tend to separate developers from customers, this close collaboration is, again, a specific agile aspect. It is often regarded as the source of agile success. Be that as it may, the collaboration forms a working

flow in the agile methodologies. An example is given in specification of Affect relation (Table 3.1).

To summarize, the conceptualization emphasizes two issues: distributed setting and collaboration. These issues, in fact, constitute DAD discipline. When both are considered, two following questions arise:

- What are the effects of applying distributed setting limitation?
- In what ways distributed setting affects collaboration?

In order to answer these questions, the third aspect of the conceptualization is presented in next section. As collaboration refers mainly to Planning, Communication (collaboration of Developers and Customers), Team Work Organization and Programming (collaboration among Developers), these activities are selected as the area of interest (see Figure 3.2). One can notice that these activities actually determine the development course. While traditional methods emphasize the role of Product Management activity, agile methodologies considerably limit Product Management in favor of direct people interactions, which are covered by the selected activities.

3.1.3 Agile practices

The specified area of interest (see Figure 3.2) consists of four collaboration activities which are affected by distributed setting limitation. As already mentioned, these activities represent certain groups of agile practices. The grouping criteria include the place and role of each practice in the whole process. Thus, when the Affect relation is applied to an activity, in fact, it is applied to the grouped practices within this activity. However, not all of the grouped practices need to get affected. Therefore, when the distributed setting is considered, a study on affected practices is required.

In order to point out the affected practices, they must be identified first. This is one of the roles of the third aspect of the conceptualization. It is presented in Figure 3.3. It depicts the relations between selected activities from a perspective of agile practices. There are two interesting issues here:

- How the relations between activities map onto relations between practices? This seem significant as it shows the role and importance of individual practice in the process. A crucial practice must be supported in distributed setting to maintain the process.
- Which of the identified practices are involved in the collaboration? Involved practices are more likely to get affected by the distributed setting limitation, which means they can require support to be maintained.

Relations between practices

Sixteen practices can be identified within the four selected activities. All of the identified practices are described and explained in Section 2.3.1. The convention used for denoting Re-

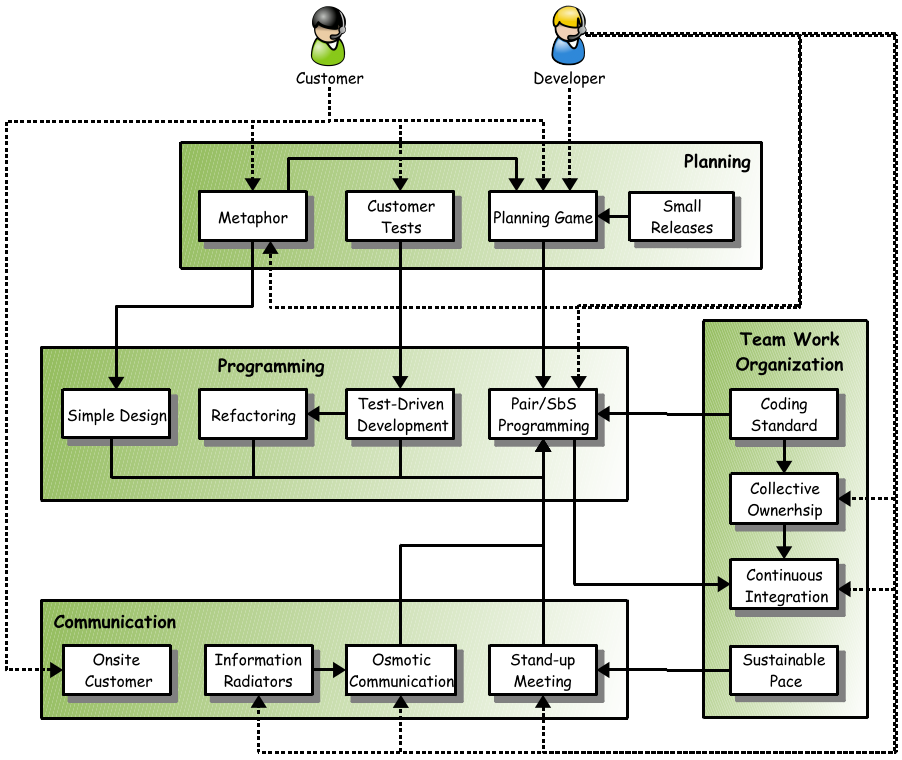


Figure 3.3: Agile practices and their relations

lations remains the same as for relational conceptualization. One can notice that *Participate In* and *Affect* relations are preserved.

There are five cases for the relations between selected activities: Planning affect Programming, Communication affects Programming, Team Work Organization affects Communication, Team Work Organization affects Programming, Programming affects Team Work Organization. Practices conceptualization maps them onto individual practices. For example, the last case is implemented by Pair/SbS Programming practice which influences Continuous Integration by producing and committing (releasing for integration) concurrent versions of developed code accompanied with tests. Judging by the number of relations between practices, it seems that the two first of the five listed activity relations are stronger, and therefore more important, than the other three. This observation is used in further discussion on collaboration.

The relations between practices form a visible work flow: from planning practices (Planning Game, Customer Tests) supported by communication practices (Osmotic Communication, Stand-up Meetings), through Programming (TDD, Simple Design, Pair/SbS Programming) to work organization practices (Continuous Integration). One can notice a central role of Pair/SbS Programming in this work flow. This should not surprise as programming is a core of software development.

An interesting observation is that the discussed conceptualization reminds of *XP practices dependencies* scheme (Figure 2.9) aimed at illustrating the source of XP practices synergy.

While the conceptualization focus on the most important relations between selected agile practices, it can be also seen as a illustration of their synergy.

Practices in collaboration

The collaboration emerges from practices in which both developers and customers directly participate. In addition, the collaboration can indirectly include other practices based on identified relations. For instance, Planning Game, which is a collaborative practice, is affected by Small Releases. While Small Releases practice is not participated by anyone, this relation might, in certain cases, involve Small Releases in the collaboration as well.

Judging by the number of collaborative practices, one can notice that Planning and Communication are the most collaborative activities. This corresponds to the previous observation of the most numerous relations between following activities: Planning affecting Programming and Communication affecting Programming. When both relations and collaboration are combined, it appears that the triangle which vertices are Planning, Communication and Programming instantiates a core of Agile Development. It is due to intensive relations and collaboration within this triangle.

Furthermore, it can be noticed that Pair/SbS Programming practice plays, again, a crucial role here. Not only is it a significant element of a work flow (most affected by other practices), but it is also a collaborative practice. Therefore, when a support for DAD collaboration is considered, Pair/SbS Programming must be approached with a considerable attention. However, once supported, it requires the related practices to function properly. Obviously, the problem appears when a related practice is a collaborative one as it also needs to be supported in distributed setting. Among the related practices, the following two involve direct and intensive collaboration: Planning Game, Osmotic Communication.

3.2 Communication model of DAD

The proposed conceptualization shows the role of agile practices in the DAD process. As mentioned, team distribution affects these practices in a different way. Some of them are vulnerable to team distribution, while some are not. To indicate which practices are vulnerable, and therefore require support, a classification of communication channels is proposed.

Based on the classification and the status of the available support, it is possible to select channels for support development. In order to evaluate the developed support, the presented model proposes also metrics of the selected practices and channels.

Communication problems are the essence of DAD discipline. Here the following aspects are investigated:

- Classification of practices in a sense of their sensitivity to a team distribution
- Communication channels involved by each practice
- Status of the available support for the identified channels

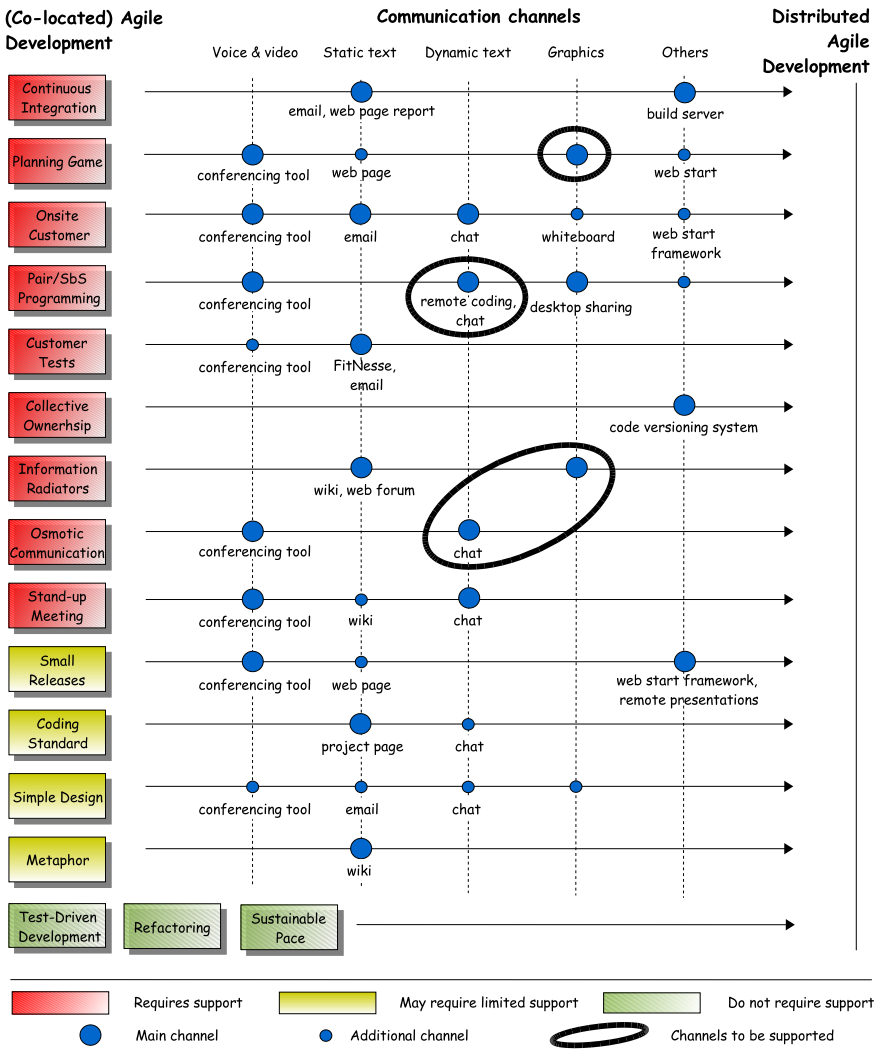


Figure 3.4: Classification, channels and available support of agile practices

3.2.1 Classification of practices in distributed setting

The presented conceptualization identifies collaborative practices which are supposed to get affected by the team distribution. Based on this assumptions, the model proposes a classification for three types of practices: practices that do not require support, practices which may require a limited support, practices that require support.

Obviously distribution affects the communication and data exchange. It appears that some of the agile practices do not rely on communication. An example is Sustainable Pace, which is a disciplined approach to the working hours. This approach, once established, do not require further attention. The model identifies two other practices which can be adopted onto distributed ground with no additional support.

The model distinguishes four practices, that may require a limited support. For example Metaphor assumes establishing a common dictionary for the project participants. As this

dictionary must be shared and collectively developed, a limited support may be required. Coding standard is a similar case (just replace dictionary with common coding convention), however usually it is established at the project beginning. Small Releases practice requires releasing a product to a remote customer which may involve additional developers support (presentation, installation, training). Finally, while Simple Design is only a suggestion, it refers to designing, which requires considerable collaboration.

With regard to other practices, they vary in the type and effectiveness of required support. Such practices like Pair Programming and Osmotic Communication usually involve more intensive communication than Customer Tests, especially that these are practices of a day-to-day team work. In addition, these practices involve a variety of data types. For Pair Programming, these are, apart from direct conversation, developed code, design sketches and concept drawings.

3.2.2 Communication channels

Each practice, which requires support, involves some amount of communication. Every communication is instantiated through a communication channel. The model presents five types of channels for the discussed agile practices. These types are described in Table 3.2.

Table 3.2: Communications channels types

Channel type	**Description**
Voice & video	Voice and video channel assumes a real time (life) transmission of voice and video image of communicating people. There are two fundamental properties that describe this type of channel: latency and quality. As for latency, it is often assumed that it should not exceed a limit of two seconds in both directions, otherwise the bidirectional communication is seriously aggravated. In case of quality, which is a subjective property, it is difficult to make similar assumptions. Therefore, a basic requirement is the understandability of transmitted information. Additionally, quality can be subjectively described by the comfort and easiness that the information is acknowledged with.
Static text	This channel assumes an infrequent transmission of text information. A typical implementation are emails and web pages. At present, with the growing popularity of wiki pages, replacing emails in a development environment, static text becomes more of text sharing. Obviously the main property of the channel is its reliability and accessibility.

Continued on next page ↓

Table 3.2 – continued from previous page

Channel type	Description
Dynamic text	Real time text channel assumes considerably frequent transmission of text information. A single text transmission is usually treated as a message. The main property of this channel is a latency. It can be assumed that a maximum latency is no more than several seconds. However, in more sophisticated cases, the maximum is the same as for voice and video channel.
Graphics	This channel assumes a transfer of graphical information. It has three main properties: latency, quality and accessibility. A maximum latency depends on the case. A dynamic collaboration requires a latency of two seconds at maximum. Sharing graphical information does not pose any latency requirements. Quality refers to the quality of the incoming graphical data. Accessibility refers to two aspects: channel accessibility and the format in which the graphical information is available.
Others	Other channel types may emerge from specific purposes. An example can be a binary channel used for broadcasting a new release of the developed product. While this channel can be implemented based on the already specified channel types, due to its special purpose it seems reasonable to separate it in the other type.

Using the presented types, the model attributes each practice with a set of required channels. In addition, main channels (which are either required or considerably facilitate the communication) are distinguished from the additional ones (sometimes may occur useful). In this way, a kind of map of practices is formed. For example, On-site Customer practice may involve all channels types, while Metaphor requires only one. As for Simple Design, which is, as already mentioned, a kind of suggestion or motto, it is attributed with additional channels only.

3.2.3 Available support for crucial practices

The model gives some overview over the available software solutions that may be used to support channels of the identified practices. The listed solutions can be both general (e.g. conferencing tool) and domain specific (e.g. FitNesse [87]).

There are also cases where no solution is available. The examples are Graphics channel for Information Radiators and Planning Game. These practices were identified as significant

in previous sections. Likewise, Pair Programming and Osmotic Communication were pointed out as crucial for the whole process. With regard to these practices, the presented model reports inefficient support. Pair programming lacks of the reliable support for *Dynamic Text* channel, which means remote collaboration and sharing of the same developed code. The main problems with available solutions are, as mentioned (see Section 2.4.2), communication latency, quality and features. In case of Osmotic Communication, the Dynamic Test channel is currently maintained by text chat applications, however, these solutions are not satisfactory as they provide only limited set of features.

As result, the crucial practices are not maintained in distributed setting. Therefore, the decision was made to develop a solution which would support the selected channels of the crucial practices: Pair Programming, Planning Game and Osmotic Communication supported by Information Radiators technique. This technique is often used in Osmotic Communication practice. Therefore, to simplify further discussion, hereinafter it is referred to as a part of this practice.

3.3 Metrics for crucial practices

When the support for the crucial practices is considered, a question of its evaluation arises. The evaluation requires a metric, or better, a set of metrics, which should provide objective and reliable results.

However, a problem arises as agile practices involve numerous factors (e.g. psychological aspects of people interaction and collaboration), which are difficult to quantify and objectively measure. This makes it difficult to propose a reliable metric which can provide valuable and accurate data

This problem is approached mainly with regard to Pair Programming so far [49, 72, 12, 47, 46, 36, 73, 52, 53, 41, 69, 74, 37]. As for the other two practices, that is Planning Game and Osmotic Communication, no research, and therefore no metrics, in this field are reported yet.

The common theme of Pair Programming related studies is to examine the efficiency (in terms of direct costs and productivity) of pair programmers in comparison to the traditional approach (solo programming).

One of the most popular productivity metric is the Lines Of Code (LOC) metric [49, 72, 12]. Its enhanced variant *LOC per hour per person* is proposed in [47]. While this metric appears to be a direct measure of programmers productivity, it is not reliable enough, as it depends on several factors. The factors include code formatting, readiness and quality in terms of design and redundancies. Another issue is the individual style of coding that every programmer has. As result, the metric would give different results for programmers which are equally productive.

An interesting concept of Cognitive Programming is developed in [36]. The model assumes that the programming process consists of consecutive stages from *Definition* (of a problem), through *Model* to *Code*. The idea behind the concept is to measure development time in each of these stages separately. However, it seems this concept disregards the fact of stages

overlapping (or even concurrency), which aggravates (or even prevents) the estimation of individual steps duration. The overlapping of stages is even more likely in case of Pair Programming (e.g. the model can evolve as a result of partners' discussion during code development).

In order to measure the quality of the developed code, the number of failures (defects) is used [18, 29, 16, 21, 41, 69]. A similar metric is the number of passed tests (usually acceptance) proposed in [47, 74, 37]. To include the assessment of defect severity, the paper [53] suggests to measure also the time of defect removal. In the same paper (which is, unfortunately, conceptual only) authors propose to take advantage of the relative (to LOC) rate of defects, called defects density, rather than total number of defects. Another variant of the quality metric, that is utilized in [47], is the number of re-submissions of the corrected (due to defects) code. Furthermore, an interesting concept of measuring the code quality through the rate of code comments can be found in [26], however, it cannot be applied to agile programming due to the refactoring practice (which assumes a self-explanatory code instead of code comments).

It is often assumed that psychological and educational aspects of Pair Programming considerably increase the efficiency of the programming pair. Thus, some of the studies investigate these aspects as well. In [76] authors report on the use of asynchronous PP (that is peer reviews) in the e-learning student platform during a course of object-oriented programming. As for metrics, the examination grades are used. In [42] approaches the *feelgood* factor. This factor is meant to estimate programmers' attitude towards their work, which has a considerably impact on maintaining programmers efficiency during long-term projects. To measure this factor, a questionnaire is used.

From the presented standard metrics, a few ones draw attention in the context of this work. In addition, some of them, such as (development) time, can be also applied to the other two practices (Planning Game and Osmotic Communication). On the other hand, it seems that the standard metrics do not fully cover the needs of AD and DAD. A proposed solution is to modify the existing metrics and to formulate new ones. Therefore, the metrics utilized in this work can be organized into three following groups:

- standard metrics
- standard metrics enhanced and adopted
- domain specific proposed metrics

The proposed metrics assume specific conditions. To provide them, a suitable task is proposed, which also depends on the observed practice.

3.3.1 Standard metrics

This group consists of three metrics: time, number of passed tests and number of found errors.

Time

Time refers to the duration of the observed action or phenomenon. In the given context, this metric measures the duration of work (in minutes) on an assigned task. This can be applied to all three practices.

Depending on the practice, the proposed tasks are:

- Pair Programming: implementation of complex algorithm
- Planning Game: extraction of User Stories from the customer
- Osmotic Communication: searching for a specific information in the available data sources

Obviously, this metric can be applied to the other test scenarios to shorten the required time of the simulation. The exception is Osmotic Communication, where the task must be dedicated.

Passed tests and found errors

These are standard quality software-oriented metrics, which are especially popular in a programming context. A good practice is to normalize the measurements by the total number of tests and errors, as it is presented in Equation 3.1:

$$Observable\ quality = \frac{p}{q} \tag{3.1}$$

where: p and q mean number of passed tests (found errors) and number of all tests (errors) respectively. As for errors, a suggested approach is to introduce them into well designed problem, for example a well tested piece of code. Still, the result value can be greater than 1 due to the nature of (unexpected) errors. The time of work (needed to find the bugs) is fixed in this metric.

This metric applies only to Pair Programming. The task should be a code with a set of unit tests, in which several bugs were introduced. The task is to remove the bugs, in this way making all the unit tests passed. The time of work is fixed.

3.3.2 Standard metrics enhanced and adopted

This group consists of two metrics: created features and feelgood factor.

Created items

This is a straightforward productivity metric which measures the output of the observed process. A well-known example is the LOC metric for programming, which is, as pointed out, questionable and unreliable though. A better choice is to measure the number of implemented items.

Therefore, the value is obtained using Equation 3.2:

$$Items\ productivity = \frac{p}{t * n} \tag{3.2}$$

where: p is the number of created items, t is the time of work (in hours) and n is the number of people who created the items. The time and number of people are meant to normalize the result. While in Pair Programming the number of participating people is, by definition, two, in Planning Game and Osmotic Communication it varies, and therefore, have an influence on the result.

Depending on the practice, the proposed tasks are:

- Pair Programming: a set of unit tests to be implemented for a given code with well isolated methods. Unit tests are treated as items.
- Planning Game: User Stories for a problem with strictly isolated requirements. User Stories are treated as items.

The metric is not applied to Osmotic Communication as it seems difficult to experimentally simulate the real natural communication as well as to quantify the communication results. An example task for Planning Game is presented in Appendix B.

Feelgood factor

As mentioned, this metric estimates the people attitude to their work. However, it seems that the factor can evince in more than just one aspect of people work. Therefore, in the given context, the metric is modified so that it includes the three following aspects:

- subjective assessment of the efficiency and usability of the utilized process or tool
- work satisfaction
- subjective assessment of task easiness

The data is gathered via questionnaire. All three assessments are scaled from 1 to 5 where 5 means the highest efficiency, satisfaction and easiness respectively. The final value can be obtained using the Equation 3.3:

$$Feelgood\ factor = \frac{a_1 + a_2 + a_3}{15} \tag{3.3}$$

where: a_1, a_2 and a_3 stand for the efficiency, satisfaction and easiness assessments respectively, and 15 is meant to normalize the result.

The task should be of moderate difficulty so that the participants answers may vary. No more special assumptions need to be done. Alike Time, this metric is to be used during completing tasks proposed for other metrics.

3.3.3 Domain specific metrics

This group of metrics emerge from the specific aspects of the three discussed practices. In [71] authors identified seven synergistic behaviors of pair programmers, which are believed to strongly impact the efficiency of the work and the quality of the produced code. Among them, the following three are selected to be the base for the proposed metrics: *Pressure*, *Negotiation* and *Learning*. While these behaviors are identified in respect to the practice of Pair Programming, they also apply to the other two practices.

As result, the following three metrics are proposed respectively to the selected behaviors: E-Factor (*Pressure* behavior), *Negotiation*, *Learning*.

It is likely that the presented metrics may be further improved and tune to some specific conditions. However, this requires a considerable amount of experiments for metrics verification and examination in different conditions, which is a subject for a dedicated study.

E-Factor

Pressure refers to the fact that pair programmers tend to be more focused on their mutual goal. Observations show that they are less keen to answer phone calls or check e-mails, which is a quite common phenomenon when working individually. To explain the role of this phenomenon, a parameter known as "Environmental Factor" or E-Factor was defined by Equation 3.4:

$$E\text{-}Factor = \frac{p}{p+q} \tag{3.4}$$

where p is the total amount of time of the uninterrupted work and q is the total amount of the time of the interrupted intervals (e.g. due to email writing or social conversations).

Obviously, individuals and teams with high E-Factor are more productive than ones with the lower one.

Pair programmers admit to be more productive to make a positive impression on their partners. The same issue refers to people practicing Planning Game and Osmotic Communication. The presence of the second person is also perceived as motivating and helps in fighting discontented moods or indolence. Finally, since co-operating people value their time, they take maximum advantage of it by completely filling it with work.

This metric relies on the environmental setting rather than task. It is suggested to organize simulation environment with several distracting elements, such as toys, pictures and computers with launched games. Additionally, during task completion, participants should be interrupted two or three times. This metric cannot be applied to Osmotic Communication as the communication is not a continuous process but rather periodical.

Negotiation

The *Negotiation* behavior relies on the observation that several minds work better than one. They inspect greater number of alternatives and consider different approaches to the same problem. In result, co-operating people are more likely to elaborate better solutions by using

the power of joined skills, experiences, knowledge and intuitions. The brainstorming effect allows collaborators for resolving even most complex issues that seemed nearly impossible at the first glance.

Hence, the idea of the metric is to examine the quality of the collaboration which is the base of the three selected practices. The metric assumes that a specific knowledge is distributed among collaborating participants. The knowledge must be new to the participants, but not too complex. Participants are assigned a set of tasks related to the distributed knowledge. To complete them, the knowledge be merged, as the result of the *Negotiation* behavior.

The final result can be computed using Equation 3.5:

$$Negotiation\ efficiency = \frac{p}{q} \tag{3.5}$$

where: p is the number of completed tasks, q is the total number of tasks. The time of work is fixed. It should be short to differentiate the results.

The task should require a specific knowledge which won't be familiar to the participants. The knowledge is divided among collaborating participants. The proposed tasks are:

- Pair Programming: an implementation of method based on specific interfaces or rare library specification (knowledge to be distributed)
- Osmotic Communication: typical programming assignment (similar to that for Pair Programming) with separated developers and communication achieved through Osmotic Communication

The metric is not applied to Planning Game practice because Planning Game assumes rather one-direction information flow, that is from customer to developers.

Learning

The *Learning* behavior emerges from the fact that team members (e.g. pair programmers) posses a variety of different skills, abilities, experiences, knowledge and even programming habits. During work they automatically share this information and, in result, continuously learn from each other. What is more, in case of Pair Programming, learning process extends beyond the single pair due to the frequent pair swaps. As result, as the time passes, learning becomes more intense and valuable owing to the greater contribution of the individual team members. In case of Planning Game, the Learning allows developers for the acquisition of business knowledge, which is necessary to proper understanding of the customer expectations and needs.

Thus, the idea behind the learning metric is to measure the quality of the knowledge transfer. A selected participant is familiarized with a task specific knowledge, unknown to the rest of participants. During the work, the knowledge is expected to be transfered to the other participants. To measure the degree of obtained knowledge, a simple questionnaire is used. The value of the metric is computed by the Equation 3.6:

$$Learning\ quality = \frac{\sum_{k=0}^{m} q_k}{mp} \tag{3.6}$$

where: m is the number of participants to whom the knowledge is transfered, q_k is the number of valid answers of participant k and p is the number of valid answers of the selected participant familiarized with the specific knowledge. It is assumed that $p \neq 0$ to make this metric applicable. The value of $p = 0$ can be caused by the questionnaire being too detailed or inadequate to the posed task. Another reason can be participants not properly focused on the task or not motivated to the proper experiment activity. These issues should be avoided during experiment preparation (this includes selection of participants).

This metric requires a similar task as the Negotiation metric. The only difference is that the knowledge is provided to only one person from the collaborating participants. The task must encourage the selected person to broadcast the possessed knowledge. As for Planning Game, the problem for analysis and planning should contain a specific knowledge, which must be passed during User Stories extraction. An example task for Planning Game is presented in Appendix B.

3.3.4 Metrics assignment to selected practices

The above presentation of metrics indicates (and explains) that not all proposed metrics are assignable to all three discussed practices. To emphasize this issue, Table 3.3 shows the assignment of each metric to the individual practices.

Table 3.3: Metrics assignment to the selected practices

Practices and metrics	**Time**	**Tests/Errors**	**Items**	**Feelgood**	**E-Factor**	**Negotiation**	**Learning**
Pair Programming	×	×	×	×	×	×	×
Planning Game	×		×	×	×		×
Osmotic Communication	×			×		×	×

3.4 Summary

This chapter presented the proposed conceptualization of Distributed Agile Development (DAD). The conceptualization includes the model of communication channels for major agile practices as well as the proposed set of metrics for evaluation of the selected practices.

The conceptualization is divided into two aspects. The first one, identified basic elements in DAD and organized them with a use of generalization-specialization and aggregation relations. Among the basic elements, the major agile practices are identified. The second aspect presented the mutual relations between the elements in a sense of the agile process. This

explains the roles of the practices.

The presented model classifies agile practices in a sense of their sensitivity to the team distribution. The model identifies also the communication channels and specifies which are required by the practices sensitive to the distribution. It also describes the available support status for these channels and discuss the ways they can be supported.

The proposed conceptualization and model allow to explain the selection of three agile practices for which the supported is developed: Pair Programming, Planning Game, Osmotic Communication. The selection is done based on three criteria: practices sensitiveness to team distribution, their role and significance for whole agile process, the lack of efficient support for them. For the selected practices, this chapter presents the proposed set of evaluating metrics. The set consists of three types of metrics: standard metrics, modified and adopted standard metrics, new domain specific metrics proposed for the purpose of the presented research.

The proposed conceptualization reflects the author's observations and point of view on the matter of DAD. It may be further developed to replace the proposed simplifications and supplement the missing aspects.

Chapter 4

Developed supporting solution: Agile Studio

This chapter presents the proposed supporting solution to DAD called Agile Studio. It was developed with a use of concept of Experiment-Driven Development, which is presented in next chapter. Generally speaking, the concept proposes an iterative process based on evaluation experiments to revise the continuously developed requirements and verify consecutive implementations of the product. As result, the requirements specification considerably evolve during the development.

The chapter is organized in the following manner. Firstly, the concept of Agile Studio is presented including very general conceptualization requirements. These requirements outline the base assumptions for the whole developed solution and influence the presented architecture of Agile Studio. The second part of this chapter describes all Agile Studio components. As they are meant to support selected agile practices, a strong emphasis is put on the requirements, which must be realized to effectively support given practice. As mentioned above, these requirements were continuously developed during the research. To illustrate how they are realized, the current release versions of components are shown and their key functionalities outlined.

It should be emphasized that the developed requirements for Agile Studio components reflect the current state of development, and it is likely that they can evolve as a result of future work.

4.1 Conceptualization requirements

At present, due to the novelty of the DAD discipline, there is no common concept on how agile methodologies and their practices should be maintained in a distributed setting, nor a common vision of a supporting software and the proper design. It seems that, at present, the remote collaboration is limited mainly to teleconferences with a use of standard com-

munication applications such as voice communicators and desktop sharing tools. It appears, however, that these tools do not fulfill the specific needs of selected practices. Therefore, the development of Agile Studio was started with the following four requirements posed:

1. The solution must support the three selected practices: Pair Programming, Planning Game and Osmotic Communication.
2. The solution must support (preserve, stimulate, not suppress) the phenomenon of synergy of various factors (e.g. human interactions), which is not only the most valuable but also crucial factor, especially under the circumstances of a team and its pairs distribution.
3. The solution ought to cover all functions that are recognized as necessary or useful in the *geographically co-located* mode which stay in accordance with the primary requirements including also functions which are decisive only for *friendliness* of it.
4. The solution must fulfill all requirements for a modern computer system of its type as soon as a conflict with the second and third requirements (necessary ones) does not arise.

The very interesting observation was made that as long as requirement 4 can be fulfilled based on a usual practice, the others needed a special approach. It is due to the fact that the synergy cannot be explicitly specified, which makes it difficult to define the preliminary requirements.

To overcome the problem, the concept of Experiment-Driven Development was applied (see Chapter 5). Through the series of evaluating experiments, it was possible to formulate the detailed functions of the developed solution and tune its parameters to the needs. The obtained feedback provided necessary information to elaborate proper design and detailed requirements specification, leading to continuous improvement of the developed Agile Studio. Having considered this approach, it is likely that the elaborated detailed requirements, which are presented in this chapter for every component of Agile Studio, are not fixed. They may alter as a result of future experiments.

With regard to the fourth requirement of meeting expectations towards modern computer system it seems obvious and indisputable. These included such features as multiuser and real-time work, portability, extensibility, quick installation, user friendliness and others. Among these, real-time work required a special attention. On the other hand, poor networks bandwidths and high latencies are serious limitations for full synchronization of the remote collaborators. Therefore, a proposed solution was expected to balance the costs and benefits of the real-time support. To establish the balance, the proposed approach was used. This required a lightweight and modular solution with an open design preferably.

4.2 Design of Agile Studio

This section presents the design of Agile Studio. Firstly, it concentrates on depicting the general concept of the developed solution. Afterwards, the architecture details are overviewed.

4.2.1 General concept

The developed Agile Studio consists of four components:

- three functional components supporting selected agile practices:
  - Distributed Pair Programming Editor (DPPE) supporting Pair Programming
  - Planning Desktop (PD) supporting Planning Game
  - Osmotic Communicator (OC) supporting Osmotic Communication
- one central component, called Agile Server (AS), which provides communication layer for the functional components

Agile Studio is based on a server-client architecture. It provides a common communication layer for all three functional components, despite their different purposes. It was possible thanks to the observation that most of the collaboration activities rely on similar pattern, which is sharing of specific objects, for instance text documents.

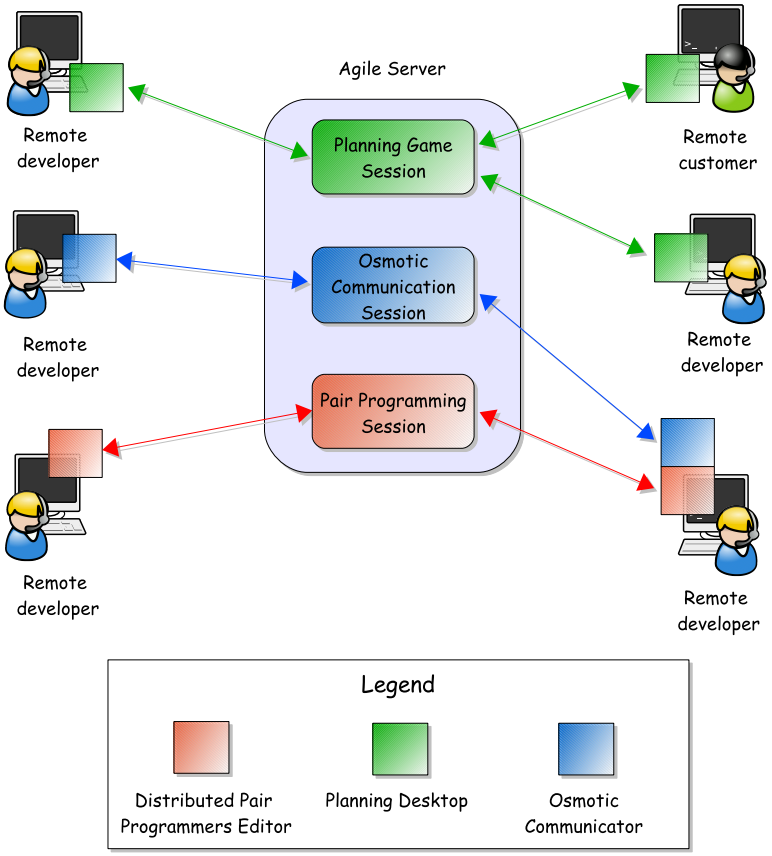


Figure 4.1: Agile Studio concept

To better explain the concept of Agile Studio, it is illustrated in Figure 4.1. The figure shows a distributed team, which consists of five developers and one customer. Two developers

practice pair programming, one of them practices also Osmotic Communication with other developer at the same time. The other two developers collaborate with remote customer in the Planning Game session. To collaborate with remote colleagues, the team members utilize the functional components of Agile Studio: DPPE, PD, OC. Functional components connect to the same Agile Server, in which separated sessions are dedicated to their specific tasks. By connecting to the common server, collaborators can easily switch between sessions. Additionally, they can communicate with other members, regardless on the their current session and activity. The use of a server-based approach also frees collaborators from having the public IP address in order to establish connection with a remote member.

The proposed concept is driven by the fourth of the conceptualization requirements, which assumed extensibility and portability. The goal behind Agile Server is to provide a common communication layer for all applications that require distant communication. This allows for extending Agile Studio with new components supporting other agile practices. What is more, Agile Server can be also utilized separately from Agile Studio as a communication layer for external applications. The chosen architecture model can be implemented using well-known and reliable technologies, which follows the base assumption of portability.

4.2.2 Architecture

The architecture of Agile Studio is presented in Figure 4.2. The emphasis is put on the communication layer, which is provided by Agile Server to the three functional components: Distributed Pair Programming Editor, Planning Desktop and Osmotic Communicator.

As already mentioned, it has been observed that every collaboration is likely to take advantage of certain shared objects. This is also true for the selected agile practices. In Pair Programming developers operate on the source code files, in Planning Game User Stories are shared. As for Osmotic Communication, team members share specific Information Radiators available in the open workspace in which they work. Thus, the main role of Agile Server is to manage and provide real-time access to the repository of shared objects.

To achieve this, three following aspects need to be considered:

1. Access to the common messaging interface of Agile Server
2. Repository that handles shared objects of separate sessions
3. Synchronization of shared objects among session participants

As shown in the figure, the first aspect is handled by Agile Client, an important element of Agile Studio. Agile Client implements suitable mechanisms for connecting, sending and receiving messages from the server's Messaging Interface. The interface is based on Java object streams, which means that every message type is represented by a class. Obviously, this simplified POJO-based solution[1] facilitates extending the server with new features and messages, which allows to use it for different purposes. For the functional components, Agile

[1]POJO term is an acronym for Plain Old Java Object, and represents the popular idea of superiority of simple designs over complex frameworks and environments. It strongly corresponds to the assumption of simplicity proposed by Agile Methodologies and the practice of Simple Design described in Section 2.3.

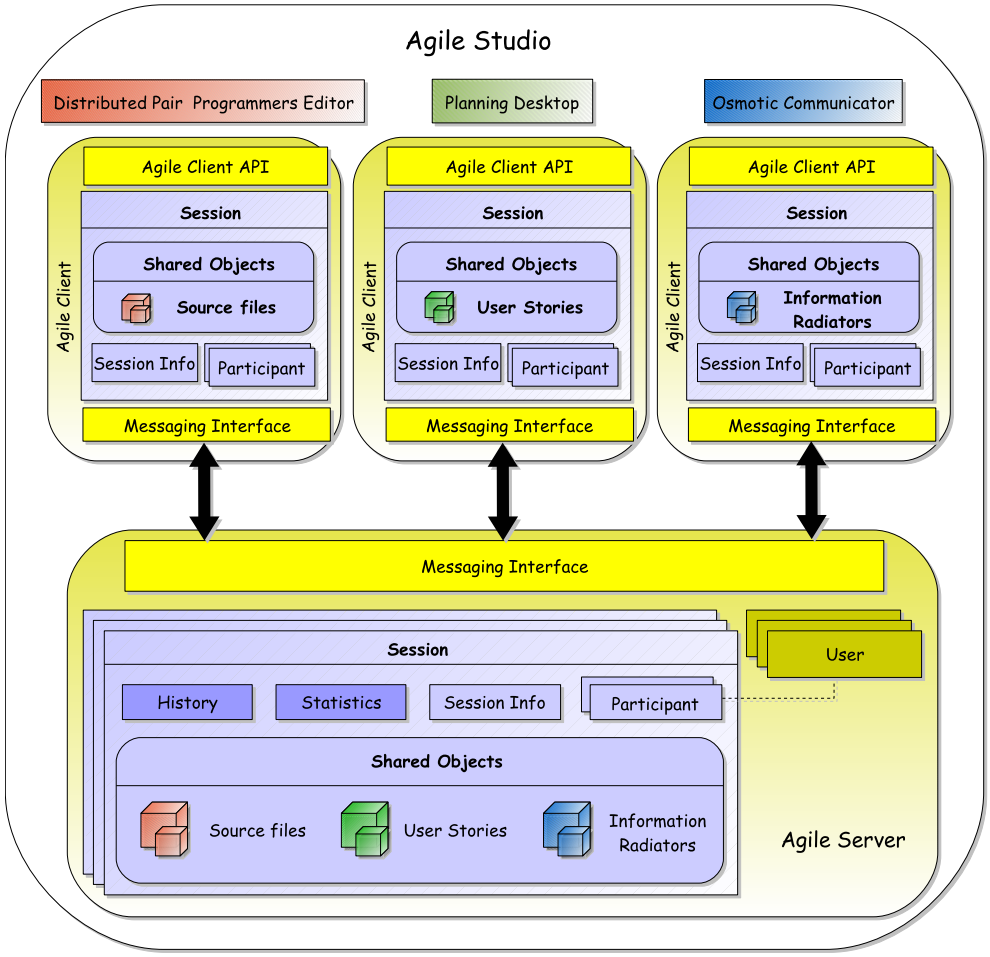


Figure 4.2: The architecture of Agile Studio

Client exposes API, which includes all required server functions, such as connecting to the server, creating and joining a session, storing shared object and others[2].

The second aspect is the repository of shared objects. It is realized by sessions which store the shared objects. Server can handle multiple sessions at once. The access to the session repository is limited to the session participants only. To obtain access, a connected user can create or join already existing session. Every session contains the list of session participants. It tells the server to which connected Agile Clients it should propagate the session data and changes.

This leads to the third aspect, which is the synchronization of the shared objects. It is realized in two steps. Whenever an event occurs to the shared object (e.g. addition, modification), the first step is to update the server repository so that it is always up-to-date. Then, the second step is to propagate the change to all session participants by sending a proper messages. At the client-side, the message is handled by Agile Client which preserves a

[2]More details on the server functions can be found in the requirements specification for Agile Server in Table 4.1.

local copy of the session repository. After updating the local repository, the event is exposed to client application (e.g. DPPE) by a suitable event from the Agile Studio API. The local copy of the server repository makes the communication time-efficient as only the changes need to be propagated. All server data is available locally without the need for querying the server every time the information is needed.

There are few issues to be added here. The Messaging Interface of the server allows for sending common text messages of three types: unicast (to one user), multicast (all users in session) and broadcast (all users connnected to server). The interface is also capable of sending custom messages. However, this requires extending the Agile Client as well. As for the server sessions, beside shared objects, they contain also other information, such as session description (Session Info) and history of session messages and events. This data is accessible from the Agile Client API and can be used in the client components.

Another interesting issue is the role of session master. By default, every session participant has a write access to the session repository, which means he/she can freely modify a shared object. However, the server provides a role of session master, which can be changed via the Agile Client API. In order to remain open, the implementation of blocking mechanism is left to client applications. This was done for DPPE, where the navigator has only read-only access to the shared source code.

4.3 Agile Server (AS)

This section presents the developed requirements for Agile Server. The requirements are divided into functional and realizational. Realizational requirements refer to technologies and working efficiency. Each requirement is assigned a unique ID symbol, which is used as a reference in Chapter 5.3 where the development story is presented.

Table 4.1 presents functional requirements. They are organized into sections that refer to different aspects of developed server.

Table 4.1: Functional requirements for Agile Server

ID	Specification
	GENERAL
AS-FR1	The current state can be stored and loaded at the beginning of the work.
AS-FR2	The port of the server is free to choose at a run time.
AS-FR3	All events and messages must be logged.

Continued on next page ↓

Table 4.1 – continued from previous page

ID	Specification
	USERS AND SESSIONS
AS-FR4	The server must handle multiple users and sessions.
AS-FR5	Every user that connects to the server needs to be identified by a unique nickname.
AS-FR6	Every user can create any number of sessions.
AS-FR7	To create a session, the user needs to provide unique session name and a password.
AS-FR8	Every user can join a session if he/she provides the correct password.
AS-FR9	Every user can join only one session at the time. To join another, he/she needs to leave the current session first.
AS-FR10	Every user can delete an empty session, if he/she provides the correct session password.
AS-FR11	Every user can leave a session any time. When all users leave the session, it remains empty on the server.
AS-FR12	Every user can get a list of all users and sessions (with a list of participating users).
AS-FR13	Every user is notified about any change concerning connected users and available sessions.
AS-FR14	Every user can send a text message: unicast (to specific user), multicast (to specific session), broadcast (to all connected users).
	SHARED OBJECTS
AS-FR15	A user in the session can an add, modify or delete shared objects within the session.
AS-FR16	Shared object is assigned to exactly one session.
AS-FR17	All users within a session get immediately notified when an object in this session is changed.

Table 4.2 presents the realizational requirements for Agile Server.

Table 4.2: Realizational requirements for Agile Server

ID	Specification
AS-RR1	The server must be run on every machine with a Java Runtime Environment (1.4 or upper) installed .
AS-RR2	The messaging protocol should be based on Java object streams.
AS-RR3	The transmission delay of the server messages should allow for a real-time work.
AS-RR4	The quality of the collaboration data must be maintained during transmission (the transmission must be lossless).
AS-RR5	The server must provide a plugin interface for its future extensions (extensions include new types of messages and shared objects).

Most of the presented specification is dedicated to sessions, users and shared objects. Requirements related to users and session can be treated as close to standard. However, requirements which describe shared objects (from AS-FR15 to AS-FR17) and related issues are specific for the discussed solution, being at the same time quite important as they realize the idea behind the proposed Agile Studio.

Realizational requirements also require a few words of comments. While the quality of communication (AS-RR4) and extensibility (AS-RR5) can be treated as standard requirements, an interesting idea stands behind the requirement of POJO-based messaging protocol (AS-RR2). It is intended to facilitate implementation and extensibility. A requirement which is likely to be out-of-date soon is the assumption of Java 1.4 compatibility (AS-RR1). The idea was to target the developed server also onto older machines, which still run the Java Virtual Machine in version 1.4. This would increase the portability of the solution.

4.4 Distributed Pair Programmers Editor (DPPE)

This section is dedicated to the overview of the developed support for Pair Programming. Firstly, the developed requirements are presented. Secondly, the implemented application is shown and its features are described.

4.4.1 Requirements for Pair Programming support

As for Agile Server, the requirements are divided into functional and realizational. The ID symbols are used as a reference in Chapter 5.3 where the development story is presented.

Table 4.3 presents functional requirements that are divided into sections that refer to different aspects of discussed support. The largest section groups requirements for Shared Documents, which denote source code files.

Some of the requirements were abandoned during the development process, mainly in the *Communication* section . They are listed in the presented specification to illustrate how the specification evolved. They are also being referred in the development history which is described in Chapter 5.3. To differentiate them from other requirements, they are written in italics and * symbol is put by the ID symbol.

Table 4.3: Functional requirements for Distributed Pair Programmers Editor

ID	Specification
	GENERAL
DPPE-FR1	DPPE must provide a set of features typical for advanced IDE framework (such as advanced editor with code completion, debugger, code repository client, plugin mechanism etc.).
DPPE-FR2	The provided editor must handle, at least, the most popular programming languages, such as Java, C#, C++, PHP, Ruby etc. which includes syntax highlighting at minimum.
	CONNECTION ISSUES
DPPE-FR3	User can connect and disconnect from Agile Server, only one at a given time.
DPPE-FR4	To connect to a server, user needs to pass server URL and his/her nickname.
DPPE-FR5	Connected user can obtain information on server sessions and users at anytime.
DPPE-FR6	User can perform operations on a server which are compliant with requirements from AS-FR6 to AS-FR14.
DPPE-FR7	Any change of the server status (including users, sessions) must be indicated by appropriate system text messages.

Continued on next page ↓

Table 4.3 – continued from previous page

ID	Specification
	SESSION
DPPE-FR8	A user in session can be in one of the two following roles: driver and navigator. Within a session, there can be only one driver at maximum and unlimited number of navigators. Default role after joining a session is navigator. To change the role, the user must obtain a keyboard by clicking a dedicated button.
DPPE-FR9	The current state of the session must be preserved for all session participants at all time (particularly during the change of the session roles).
	SHARED DOCUMENTS
DPPE-FR10	A list of all shared documents must be available in a form of separate view of window.
DPPE-FR11	Only a driver can change the view of the current document, including the change of the current document.
DPPE-FR12	Only the driver can add, remove and modify a shared document.
DPPE-FR13	On connecting to a session, the shared documents get opened and listed in the dedicated view or window.
DPPE-FR14	In a driver role, all editors containing shared documents are editable, which is indicated by the green editors background. In navigator role, the editors are read-only with a blue background. White editor background indicates a non-shared document.
DPPE-FR15	Whenever a shared document is changed, this change is immediately reflected on navigators' side.
DPPE-FR16	Whenever a view of shared document is changed, it is immediately reflected on navigators' side. View synchronization includes: code folding, scrolling, caret position and text selection.
DPPE-FR17	Whenever driver changes the active shared document, this change is immediately reflected on navigators' side.
DPPE-FR18	In order to add a shared document, driver needs to select an already existing non-shared document.

Continued on next page ↓

Table 4.3 – continued from previous page

ID	Specification
DPPE-FR19	When a new shared document is added, it is opened on navigators' side and reflected in their shared document views. In case when a similar document exists (same name but different content), navigator can choose to overwrite existing document or share the new document under different name.
DPPE-FR20	When a driver removes document from the session, it becomes a non-shared document and can be stored, stored or added to the session again.
	COMMUNICATION
DPPE-FR21	A text chat must be provided with a coloring mechanism to differentiate messages from different participants.
DPPE-FR22	The mouse pointer of every session participant is reflected on other participants' side with the attached name of this participant.
DPPE-FR23	Should be supported by a voice communicating software.
DPPE-FR24*	*The driver can see the navigator's selection.*
DPPE-FR25*	*DPPE should integrate voice communication.*
DPPE-FR26*	*DPPE should integrate video connection so that developers can see each other.*

Note that * sign by the ID symbol means that the requirement was dropped during development process.

It is important to emphasize here that based on the collected requirements, the DPPE is meant to be supported by voice communicating software.

Table 4.4 presents two realizational requirements for DPPE.

Table 4.4: Realizational Requirements for Distributed Pair Programmers Editor

ID	Specification
DPPE-RR1	DPPE should be developed in Java to make it portable.
DPPE-RR2	DPPE should be developed as a plugin to one of the advanced IDE.

The presented specification contains both standard and non-standard problem-specific requirements. The first group consists mainly of requirements related to connection and session issues. Among the second group, the most important requirements are these related to Shared Documents and their handling (from DPPE-FR10 to DPPE-FR20). Another crucial requirement is the assumption of IDE integration (DPPE-FR1, DPPE-RR2). An interesting requirement is also the additional communication channel, which is provided by the remote mouse pointers (DPPE-FR22).

4.4.2 Current release of DPPE

Figure 4.3 presents the current release of DPPE. For better understanding, important points were marked and labeled with numbers, which are utilized as references (e.g. ❹) for this very description.

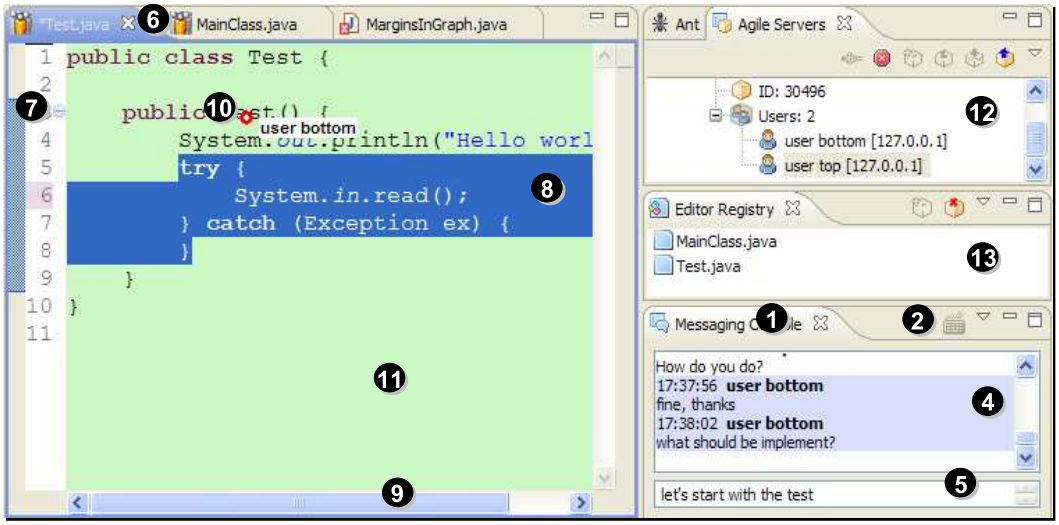
Following the presented requirements, the editor is developed as a plugin to the popular development environment Eclipse [85]. Among other advantages, Eclipse provides support for various programming languages. This feature is mandatory for the discussed support and it is listed in the above requirements (DPPE-2). To realize this requirement, DPPE offers an editor wrapper❻. The goal of the wrapper is to enable all available Eclipse editors to be shared within the distributed pair programming session. This means that distributed programmers can work on all types of files that can be opened inside Eclipse.

Beside the wrapper, DPPE consists of three views. The views are: connection panel with a list of users and sessions in the connected server⓬, view of shared editors (i.e. source files) in the current session⓭, view❶ of server❸ and user messages❹.

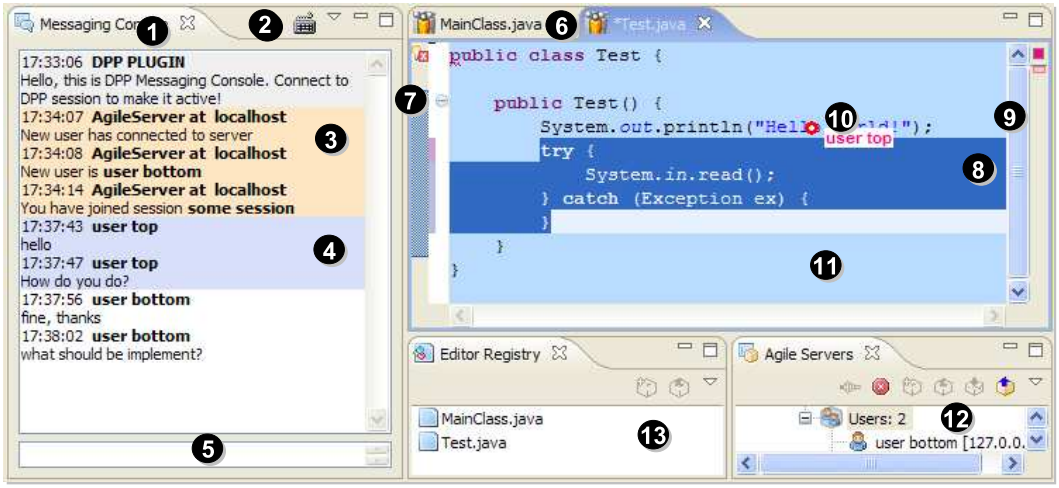
Figure 4.3 presents all these views in action during the collaborative session, which involves Navigator (*user bottom*) and Driver (*user top*) coding a Java class example. One can notice differences between Navigator's and Driver's views layout and editors size. This feature of Eclipse, which allows for adapting the work environment to user needs, requires special attention during synchronization of the distant editor instances. For example, when the remote editors differ in size, the Navigator may not see the code fragment that the Driver works on. In order to prevent this, the editor implements a mechanism of caret tracking, which enables the Navigator to follow the Driver's area of view. Additionally, DPPE synchronizes the scrollbars❾.

An interesting issue is that Driver and Navigator roles of Pair Programming [71] simplify the design of the supporting tool as only the driver can edit the source file (which is a shared object stored in the Agile Server). In order to switch the roles, one need to obtain the keyboard in the pair programming session (i.e. write access to session files). This keyboard switch❷ is synchronized by the server in order to maintain the same state of shared files in the Driver's and Navigator's editors. To enable a real-time collaboration and decrease the network requirements, the editor transmits only the session changes, not a whole session state. As Eclipse IDE allows for code folding[3], it was also synchronized❼, improving the

[3]Code folding is a common functionality of IDEs and source code editors. It allows the user to hide and display selected fragments of code. This enables user to handle large regions of complicated code by hiding the fragments which are not needed at the moment.



Pair Programmer **user top** in the role of **Driver**



Pair Programmer **user bottom** in the role of **Navigator**

LEGEND: **(1)** View of server and user messages **(2)** Change keyboard ownership button **(3)** Server messages **(4)** Remote partner's messages **(5)** Message input field **(6)** Icon indicating that the editor is in shared mode **(7)** Buttons to perform synchronized code folding **(8)** Remote partner's code selection **(9)** Synchronized scrollbars **(10)** Remote partner's mouse pointer **(11)** Shared editor with a blue/green background indicating the session role (Navigator or Driver) **(12)** Connection panel with a list of users and sessions in the connected server **(13)** View of shared editors in the current session

Figure 4.3: Distributed Pair Programmers Editors during a collaborative session

code-based communication quality. What is more, the user an see the selection made by remote partner❽. Another, very interesting feature, is real-time remote partner's cursor❿, which does not disturb driver during coding.

Among others, Figure 4.3 emphasizes the feature of colored editor background⓫, which indicates the current role of the programmer. When green, it informs the programmer about his/her Driver role and the ability to modify the shared code. Blue color informs about Navigator role and read-only access.

An interesting feature of the presented editor is that there is no limit for the number of users who can collaborate in the distributed pair programming session. Typically, Pair Pro-

gramming practice assumes there are two users in the Driver and Navigator roles. However, there are cases when a pair of programmers need to collaborate with other project participants such as expert, team leader or other project member. The developed editor handles these cases. It allows for infinite number of Navigators in the same session. The Driver can be only one. This role is not obligatory, which can be handy during code-review sessions.

In comparison to the concept of desktop sharing implemented by the existing supporting tools such as VNC[98, 104], DPPE has several advantages. The most important ones are the efficiency and quality. Desktop sharing assumes transmission of compressed screenshots, while DPPE transmits only the changes. Among other advantages, the following ones can be listed: introduced session roles preventing Navigator to edit the code (in VNC there are no roles), integration with advanced IDE, ability to selectively share documents, remote mouse pointers, integrated chat.

4.5 Planning Desktop (PD)

This section describes the developed support for Planning Game which is Planning Desktop. Likewise previous section, it gives an overview over the requirements and current release of PD.

4.5.1 Requirements for Planning Game support

Again, the requirements are divided into functional and realizational. As previous, each requirement is assigned a unique ID symbol, which is used as a reference in Chapter 5.3.

Table 4.5 presents functional requirements organized into sections. The largest sections group requirements related to User Stories, which is rather expected outcome. Two of the requirements were abandoned during the development process.

Table 4.5: Functional requirements for Planning Desktop

ID	Specification
	GENERAL
PD-FR1	User can perform operations on a server which are compliant with requirements from AS-FR6 to AS-FR14.
	WORKING AREA
PD-FR2	User stories are added to the planning area of unlimited size.

Continued on next page ↓

Table 4.5 – continued from previous page

ID	Specification
PD-FR3	Every user can scroll the planning area. Scrolling is not synchronized within the session.
PD-FR4	The working area can be zoomed in and out. Zooming is not synchronized within the session.
PD-FR5*	*The visible part of the working area should be synchronized among session's participants.*
	USER STORIES
PD-FR6	Every user story has its ID, title, description, priority and cost. ID is unique among whole server and automatically set by server. The others can be modified by user.
PD-FR7	Every user story is represented by a rectangle on the working area. Within the rectangle, the story title is shown. To edit the story data, a double-click on the rectangle is needed.
PD-FR8	Once in session, every user can add, modify, move, resize and remove a User Story.
PD-FR9	When in modification mode, a new window appear at the top of the given User Story with inputs for title, description, priority and cost.
PD-FR10	There are no limitations considering the length of the story title, description and the range of cost, except it must be of natural value.
PD-FR11	The priority is set according to the MoSCoW rule (see section 2.3.1), that is: Must (the highest priority), Should, Could and Would (the lowest priority).
PD-FR12	The priority of the given User Story is indicated by the background color of the representative rectangle card: Red for Must, Yellow for Should, Green for Could and Blue for Would .
PD-FR13	Every change to the user story (both in content and appearance) is broadcasted to all users in the given session.
	STORIES GROUPS AND DEPENDENCIES
PD-FR14	Every user can select multiple user stories and moved together in the working area. The selection can be done in two ways: by drag&drop-based selection of the area with interesting stories, or by clicking each story with a SHIFT button pressed.

Continued on next page ↓

Table 4.5 – continued from previous page

ID	Specification
PD-FR15	The selected stories can be grouped/ungrouped. The grouped stories are moved together.
PD-FR16	Group of stories can be assigned with a name, after double-clicking on it.
PD-FR17	Every group of stories can be folded, so that only the rectangle representing the group and the its name are visible. The folded group can be unfolded.
PD-FR18	Every user can add and remove a connection between the rectangles representing two User Stories. Connection can be described by a text of unlimited length, which can edited after performing double-click on the selected connection.
	STORIES TRACKING AND COMMUNICATION
PD-FR19	All stories are listed in a table in a separate summary view. The table contains the following information: title, priority and cost. The table rows can be sorted using the three column types. The view can be turned on/off from the main menu.
PD-FR20	The summary view contains a bar chart indicating the number of stories per each priority.
PD-FR21	All session data is automatically stored on a server. Every fixed amount of time, a backup of this data is created.
PD-FR22	A separate view contains the chat panel, with input box and chat window.
PD-FR23	Should be supported by a voice communicating software.
PD-FR24*	*Should be supported by a video communicating software.*

Among others, it should be noted that alike for DPPE, PD is meant to be supported by voice communicating software, such as [100].

Table 4.6 presents four realizational requirements for PD.

Table 4.6: Realizational Requirements for Planning Desktop

ID	Specification
PD-RR1	Must be developed in Java to offer portability and compliance with the server.
PD-RR2	Must allow for a real time collaboration including story moving and resizing.
PD-RR3	Must be intuitive and easy to handle for a person with only basic computer-operation background.
PD-RR4*	*It should be possible to launch the application as a Java applet.*

To remind, sign * and italics font indicate requirements that were dropped during development process. More details can be found in Chapter 5.3.

The major part of the presented specification are non-standard problem specific requirements. Among them, the most important are requirements describing working area (from PD-FR2 to PD-FR5) and user stories, especially referring to their representation and modification (from PD-FR6 to PD-FR9). A unique and interesting requirements are also these related to stories coloring (PD-FR12), grouping (PD-FR17) and dependencies (PD-FR18). A valuable and surprising observation is that the video channel was considered not mandatory for this kind of support.

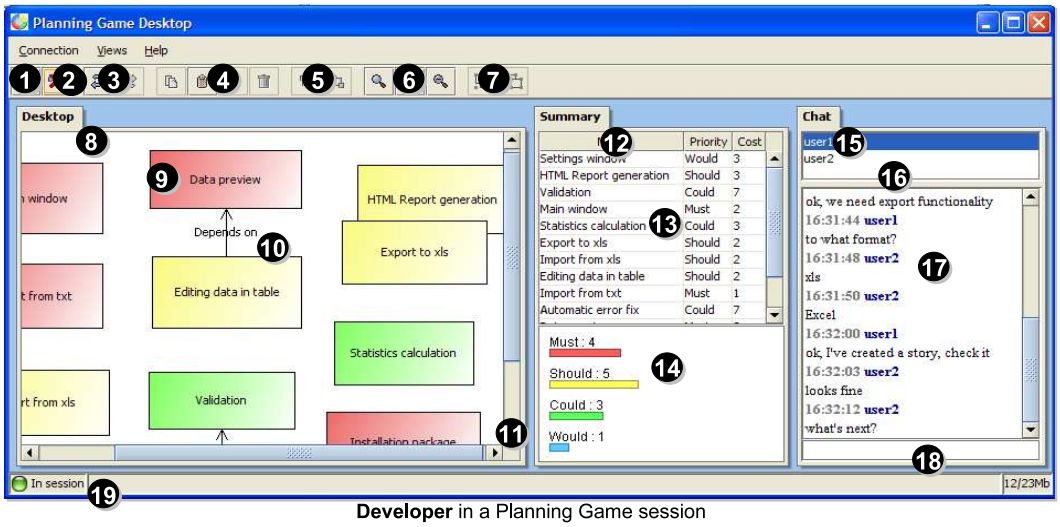
4.5.2 Current release of PD

Figure 4.4 demonstrates two PD applications in a collaborative session. For better understanding, important points were marked and labeled with numbers, which are utilized as references (e.g. ❹) for this very description.

It can be noticed that the two application workspaces have different layouts. While this was not specified in the obtained requirements, it seemed advisable to add docking feature so that the user can adjust the workspace to his/her preferences. It also allows to combine the available views as shown in the figure ⓴.

There are three views available: Desktop❽, SummaryDesktop⓬ and ChatDesktop⓯. Desktop is the most important view as this is the working area, which was specified in the presented specification PD-FR2. As it is shown in the figure, Desktop view contains user stories❾ of the session. User Stories can be added by any participant of the session using a button from the toolbar❶. They can be also resized (one can notice different stories' sizes) and moved within the Desktop. Evidently, each of these events is immediately reflected in the remote participants' workspaces.

The colorful rectangle representing a single user story show only its title. To edit the contents of the story, a special popup edit view⓫ is used. It allows for modifying story



Developer in a Planning Game session

Customer in a Planning Game session editing User Story

LEGEND: **(1)** Adding new story button **(2)** Turning on/off adding connections by dragging between stories **(3)** Undo/redo buttons **(4)** Copy/Paste/Cut/Delete buttons **(5)** Z-order story arrangement buttons **(6)** Zooming buttons **(7)** Grouping/ungrouping buttons **(8)** Desktop view **(9)** User Story **(10)** Directional connection with description **(11)** Scrolling bars for Desktop view **(12)** Summary view **(13)** Created stories summary **(14)** Priority bar chart **(15)** Chat view **(16)** Users in session **(17)** Chat history **(18)** Chat input **(19)** Connection status **(20)** Chat and Summary views docked together **(21)** User Story edit view **(22)** Story title **(23)** Story ID **(24)** Story description text area **(25)** Priority combo box **(26)** Story estimation spin edit

Figure 4.4: Two Planning Desktop applications during collaborative session

title㉒, description㉔, priority㉕ and estimation㉖. As for the priority, the MoSCoW rule (see Section 2.3) was applied. An interesting solution is the introduced color code, which is used to paint the background of the story rectangle. Each priority is represented by a color, for instance story with the highest priority *Must* is painted in red.

Another interesting feature are the directional connections between user stories. They are meant to provide additional information on relations between the stories. An example is visible in the shown figure⑩, when a connection indicates that story "Editing data in table" depends on "Data preview". A connection is added via mechanism of drag&drop from one story to the other. If inconvenient, the mechanism can be turned off②. Apparently,

the connections are the extension to the User Stories technique, and the advantage of the "virtual" Planning Game over its classic version.

Among other important features, one can find copy-paste❹ and undo/redo❸, which are available from the toolbar. Another ones are grouping❼ and z-order arrangement❺ functionalities. The last ones facilitate manipulation of the stories, which can be useful when the number of stories is considerable. While these features get synchronized among session participants, zooming❻ and scrolling⓫ of the working area are not. This would handicap the user personal navigation, making it dependent on other users' actions.

The second view is the Summary ⓬. It contains the table with information on created stories. This gives a brief overview⓭ over the current session status. Below the table, a priority bar chart⓮ is placed. For each priority, the number of user stories is counted. The idea behind this chart is to control the distribution of the priorities. It often happens that the customer promotes most of the stories with the highest priority (Must). The bar chart allows developers to quickly identify such situations and ask the customer to revisit the priorities. Planning process is much more simpler when the priorities are equally distributed among the user stories.

The last view is the Chat view ⓯. It provides a list of users⓰ in a session as well as standard chat functionality⓱.

An interesting fact is that while the Applet realizational requirement (PD-FR4) was abandoned during development process as not needed, it was already realized. The presented version of the application can be launched as a Java applet. As for possible enhancements, the synchronization of the Story edit view was considered. Another possible improvement would be to extend User Story description with graphical data, e.g. attached image.

Finally, an interesting observation was done that the developed solution enhanced the Planning Game process itself by adding such features as connections and color code (for denoting stories priorities).

In comparison to the concept of desktop sharing implemented by the existing supporting tools such as NetMeeting[95], PD has several advantages. Alike for DPPE, PD transmits only the changes, not compressed screenshots. The second important advantage are dedicated story cards, with priorities and estimates. Among other advantages, one can point out: zooming capabilities of working area, cards grouping, labeled connections between cards, statistics, automatic colors for indicating the current priority of the story, integrated chat and list of participants.

4.6 Osmotic Communicator (OC)

This section describes the developed support for Osmotic Communication, which is Osmotic Communicator. First the requirements are presented. Afterwards, the current release version of OC is overviewed.

4.6.1 Requirements for Osmotic Communication support

As previously, the requirements are divided into functional and realizational. The ID symbols are used as a references in the description of the OC development in Chapter 5.3.

Table 4.7 presents functional requirements. As expected, the largest section is related to Information Radiators. Four requirements were abandoned during the development process.

Table 4.7: Functional requirements for Osmotic Communicator

ID	Specification
	GENERAL
OC-FR1	User can perform operations on a server which are compliant with requirements from AS-FR6 to AS-FR14.
OC-FR2	After the application is launched, it is accessible from the icon in the system tray. When the icon is clicked, a popup menu with options is shown.
OC-FR3	The current state of the default session is stored and loaded at the application start. The default session is set to the last session, if it exists.
	WORKING AREA
OC-FR4	The working area is the system desktop. Therefore, it has a fixed size and no scrolling capabilities.
OC-FR5	The size of the working area depends on the local settings and can be different for the remote users
OC-FR6	By default, the contents of the working area appear on the top of the other system windows.
OC-FR7	It is possible to hide working area or put it to the back, so that is accessible when the desktop is shown. It is possible to put the working area on the top of other windows again.
OC-FR8	All operations are accessible from the popup menu from the system tray.
OC-FR9*	*The working area is a resizable and scrollable window.*

Continued on next page ↓

Table 4.7 – continued from previous page

ID	Specification
	INFORMATION RADIATORS
OC-FR10	Information Radiators are represented by a colorful seizable rectangles on the working area.
OC-FR11	The color of the Information Radiator can be set from a predefined set of colors in the Radiator's properties. The properties are opened from a mouse right-click on the information radiator.
OC-FR12	Every Information Radiator has a title and text, both of unlimited size. In case of not enough place for displaying the text, scrollbars appear.
OC-FR13	Information radiators can contain any file, which are accessible via link. The file is by default opened by a suitable application based on system settings.
OC-FR14	It is possible to save all files from the Information Radiator to a specified directory.
OC-FR15	It is possible to add multiple files to a Information Radiator either by selecting them manually or specifying a folder.
OC-FR16	Every user can add, modify or delete any Information Radiator in the given session.
OC-FR17	Modification of Information Radiator includes changing its title, text, files as well as resizing and moving to a new location in the working area.
OC-FR18	The system clipboard must be supported for pasting its content into the title or text of a Information Radiator.
OC-FR19	Every modification made to a Information Radiator is broadcasted to all users in the session.
OC-FR20	It is possible to turn on/off the synchronization of the Information Radiators locations.
OC-FR21	It is possible to hide/show selected Radiator in the working area.
OC-FR22	A list with all Information Radiators in a given session must be available.
OC-FR23	It is possible to search for a specific keyword in the content of all radiators. Radiators with the specified keywords are emphasized by black background and white font. The keyword is painted with red font color. Searching in not synchronized among the session.

Continued on next page ↓

Table 4.7 – continued from previous page

ID	Specification
OC-FR24*	*Information Radiator can contain an image. In case of large image exceeding the size of Radiator, scrollbars should be available. The image should be loaded from the file. Jpeg and Bitmap format should be supported at least.*
OC-FR25*	*Synchronization of Information Radiators should be performed in a real time, continuously during modification operations, e.g. resizing, dragging.*
	COMMUNICATION
OC-FR26	It is possible for a user to communicate with other session participants using a chat window.
OC-FR27	Every user can send text messages of three types: unicast, multicast, broadcast. It is possible to filter out selected types of messages.
OC-FR28	The unicast messages appear as a dedicated message window. The multicast and broadcast messages are displayed for a fixed amount of time in a bottom-right corner of the working area.
OC-FR29	All messages are stored in a local history, which is accessible from the popup menu in the system tray.
OC-FR30*	*Communication should be supported by a voice communicating software.*

To remind, the italics font and * sign by the requirement ID indicate that the requirement was dropped during development process.

Table 4.8 presents two identified realizational requirements for OC.

Table 4.8: Realizational requirements for Osmotic Communicator

ID	Specification
OC-RR1	Must be developed in Java to offer portability and compliance with the server.
OC-RR2	Must allow for a real time collaboration including communication and modification of the Information Radiators.

As expected, the most important part of the specification is taken up by problem-specific issues. Among important requirements, one should indicate these related to Information Radiators, such as OC-FR10 describing their representation and OC-FR17, OC-FR21 which specify how they can be handled by the user. Another important and non-standard part of specification refers to the working area, which is, quite uniquely, based on system desktop (OC-FR4). As for the more standard and less significant requirements, the example is the embedded chat feature OC-FR26.

Among others, the specification indicates also two interesting aspects by including the requirements that were abandoned during development process. The first one is OC-FR26 assuming the voice channel support. While it may be surprising at the first glance, it occurred that the osmotic communication does not require the high sensitivity provided by this communication channel. Moreover, the voice channel may conflict with other voice channels, which are utilized in other practices, e.g. Pair Programming. The second interesting abandoned requirement is the OC-FR25. It was discovered that despite the initial assumptions, full synchronization of the workspace including Radiators movement is unnecessary. Osmotic Communication can be efficient enough using more static exchange of information. Another issue were the performance problems.

4.6.2 Current release of OC

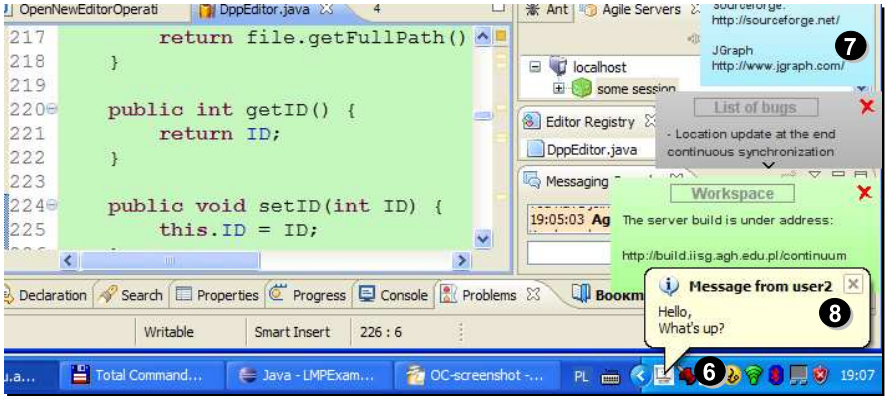
The idea of reminding notes on the system desktop is not new. There is a range of existing applications such as KDE KNotes[92], Sticky Notes[101] and others. Therefore, the idea was to reuse existing Open Source solution and extend it with the necessary functionality, instead of creating new application from the scratch. After conducted research, it was decided to take advantage of the Open Source project[96] called *pin 'em up*. The extension includes, among others, connecting the application to the Agile Server, handling server messages and synchronizing the notes with the shared objects repository on the server.

Figure 4.5 demonstrates two Osmotic Communicators launched on remote machines represented by the screenshots in the top and bottom of the figure. In the top one, the user works with OC on the common windows desktop❶ along with standard icons and shortcuts. In the bottom one, the user utilizes OC during distributed Pair Programming session. As shown, the session is conducted via Distributed Pair Programmers Editor.

As shown in the figure, every Radiator is attributed with a title❷ and text content❸. One can notice that the positions and font sizes of the Radiators differ for the remote users❼. These properties can be customized individually so they suit the needs of the user and current layout of opened windows.

Beside these properties, every Radiator note can be given a different background color. This feature can be used for example to indicate the category of the information or the priority of the task. There is a difference between color and previous properties though. The colors are synchronized among session participants and cannot be changed locally only. It is because the color may contain an important information while the position and font size are mainly visual properties. It should be noted that the color synchronization was included in the obtained requirements. It was implemented as an extra feature, which emerged during

Osmotic Communicator with the windows desktop in the background



Osmotic Communicator with the DPPE in a distributed Pair Programming session

LEGEND: (1) Windows desktop with casual elements **(2)** Information radiator with a descriptive title **(3)** Contents of Information Radiator **(4)** Button that hides radiator **(5)** Extended pin 'em up popup menu **(6)** Tray icon with the popup menu which shows after a right-mouse click **(7)** Information Radiator on a remote machine with a different location and font size **(8)** "Balloon" box with a session message arrived

Figure 4.5: Osmotic Communicator applications during collaborative session

the development process.

According to the presented specification, it must be possible to hide selected Radiator. This is realized by the button in the top-right corner of the note❹. This allows the user to hide unwanted or disturbing Radiators. An example is shown in the figure 4.5, where the bottom user hid some of the shared Radiators. Another issue is the default position of new local Radiator, which can be configured individually. This can be also applied to the new Radiators created by remote users. By default, the remote Radiators are positioned at the same coordinates as in the remote working area. In case the screen resolution is too small, they position is appropriately adjusted.

An important part of the Osmotic Communicator is also the chat feature. While another communicator can be used, the idea was to integrate a common text channel into all Agile Studio components. In this way, the team can utilize a common messaging channel, boosting

up its efficiency. An example is depicted in the figure 4.5. During pair programming using DPPE, the user not only receives messages from his/her partner, but also from other team members through Osmotic Communication practice. In OC the messages are displayed in a "balloon" box❽ above the system tray, which is less distractive for the working user.

The OC is controlled via a tray icon❻. When a right-mouse button is clicked on the OC icon, a popup menu is displayed❺. This is a standard menu of the pin 'em up application, however extended with OC specific settings (e.g. connection properties for Agile Server).

In comparison to the popular wiki tools[8, 68], which can be utilized for supporting the communication, OC has the following advantages: dedicated control for Radiator including coloring feature, immediate notification of made changes, no need to switch windows in order to see Radiators on the wiki page, integrated chat, ability to graphically organize the radiators into custom layout

4.7 Summary

In this chapter the developed solution, called Agile Studio, was presented. It is meant to support three selected agile practices in distributed setting, that is: Pair Programming, Planning Game, Osmotic Communication.

The chapter overviewed the general concept and main requirements for the discussed support. It also described the architecture of the Agile Studio, which consists of four components: Agile Server, Distributed Pair Programmers Editor, Planning Desktop and Osmotic Communicator. The presented architecture is based on the concept of object sharing, which was also explained in this chapter.

The chapter described all Agile Studio components with emphasis put on the aspect of developed requirements for the discussed support. To present how these requirements were realized, the chapter overviewed the current release versions of the components. They key functionalities were described and illustrated on the included screenshots.

Chapter 5

Development and evaluation of DAD support

This chapter presents the concept of the Experiment-Driven Development (hereinafter referred to as EDD), the proposed approach to the development and evaluation of software applications. While the basic idea behind the concept has been around for a while, the available literature reports no definition nor specification for this kind of development.

The motivation behind the concept was the need for a development method, which would be suitable for developing a specific software such as DAD support. This kind of software can be described, among others, by the two following attributes:

- the software usability and efficiency highly depends on the human factor,
- the set of its requirements is difficult to be specified a priori.

Dynamic and partly unknown specification of the product requirements are the common challenges of every software development process. Among others, this is due to the intangible factors that influence the proper and expected product functioning. By *intangible factor* it is understood a factor which cannot be clearly observed or objectively measured. The most common is the human factor, for instance, a user of the developed product. Thus, the question arises on how to include these factors into requirements specification if they are, by definition, non-specifiable.

A typical practice is to take advantage of the empirical approach to the evaluation of a product during its development. The first release of a developed product is used to obtain user feedback, which is a base for further enhancement of the product. An example can be the strategy of product pre-releases (named as Alfa and Beta), which are usually free for the testing users. Another example is Microsoft company which releases not fully functional Windows OS supported by upcoming Service Packs, which are based on users feedback[1].

[1]Though, it seems this is a side-effect of the Microsoft's marketing and financial strategy to make the profit as early as possible, even with a product of low quality.

Obviously, this approach has two main disadvantages:

- The process of evaluation is difficult to control which means that the obtained data (if any) is unreliable
- The obtained data is strongly subjective which precludes any objective measure of the performed progress

Therefore, the idea behind the proposed concept of EDD is to take advantage of a controlled evaluation in a form of experiments. It appears that, while experimenting is a known research practice, its utilization as an integral part of the continuous development process is not reported so far. It should be noted that the concept agrees with the philosophy of agile development. Among others, it corresponds to close customer collaboration (Customer Collaboration practice) and early releases of the product (Short Releases practice). Another observation is that experimenting is a specific implementation of the systems approach.

The final remark is that the concept can be applied during every development process, however, it can be particularly useful in development of human-oriented software and in any other case in which the final shape of the developed product remains uncertain or unknown.

The chapter is organized as follows. Firstly, it presents the EDD concept. Afterwards, the application of the concept is presented with regard to the components of Agile Studio. It should be noted that the order, in which the components are presented, corresponds to the actual course of their development. This explains why Agile Server (AS) component is presented together with Distributed Pair Programmers Editor (DPPE), in spite of the fact that AS is a central element of Agile Studio, also used by the other two components. During development of DPPE, the server-side represented by AS evolved as well.

5.1 The aspect of cost in the evaluation of DAD support

Agile Studio seems an ideal subject for EDD examination because it is strongly human-oriented software (it supports humans in a collaboration process) with only partial preliminary specification of the requirements. Moreover, it occurs that a comparative analysis of AD and DAD can be realized in the framework of EDD. For the purpose of the analysis, it is proposed to structuralize the overall cost of software development into three separate elements: direct, indirect and other costs. It is assumed that the last element, other cost, do not depend on the development setting and can be neglected in further discussion. The example of other cost are the investments for computer hardware and software, which remain on the similar level in both settings[2]

[2]There are some exceptions from this rule. It seems that when a team practices pair programming, the number of required development machines strongly rely on the setting. In co-located setting, only one machine is needed per a pair of developers, while the distributed setting doubles this rate (two machines per pair of developers). However, it should be remembered that pair programming assumes breaks and individual work of developers as well. In addition, hardware investments include also development servers and other spending. Therefore, the actual differences are considerably reduced and this exception can be neglected in presented analysis.

With regard to direct and indirect costs, they depend on the development setting. Indirect cost cover the need for communication and "having the team together". This includes organizational matters from workspace organization, meetings arrangements to business travels and long-term delegations. On contrary, direct cost relate to the development aimed at obtaining a software product of assumed quality. Undoubtedly, the major element of this cost are salaries. Therefore, it can be stated that, when the quality is assumed and constant, the direct cost is the function of time. This observation strongly corresponds to the four variables of software project, which are overviewed in Section 2.3.2.

This classification allows to observe two following relations:

- DAD approach allows to considerably reduce indirect cost by providing new telecommunication methods such as efficient software support. For instance, these cost-effective methods can replace expensive business trips and long-term delegations. Beside these obvious organizational and logistical savings, DAD is also expected to reduce office maintenance cost by enabling people to work in their homes.

- In case of the direct cost, which are a measure of team productivity, it seems that AD is preferable as it offers convenient and reliable face-to-face communication. However, it is supposed that properly supported DAD can maintain the direct cost at sufficient, or, what would be better, a comparable level in comparison to AD. What is more, an interesting assumption is that DAD can have positive influence on the reduction of direct cost. This may come from the fact that properly designed and implemented software support will focus, or even force, team members' attention on achieving their goal. As the result, the waste of time (note that time is the main variable in the function of direct cost), which is usually the effect of "social" activities, will be considerably limited.

Obviously, in order to *maintain the overall cost*, the possible increase of direct cost must be balanced by savings in indirect cost. Interestingly, DAD setting can prove more cost-efficient than AD in case when the direct cost in DAD setting remain at the approximate or inconsiderable higher level in comparison to AD.

To examine these assumptions, both direct and indirect costs should be estimated for both settings. A question arise on a technique of this estimation though. Unfortunately, the estimation of indirect cost is a difficult task. It requires financial calculations and estimations based on a number of real life cases. Apparently, due to the strict companies' policies and the fact that DAD is still a rare practice, this can be considered as an unreachable goal. A solution might be a financial model of DAD, which provides estimations of distributed development costs. However, these are the issues of economics, which exceeds the scope of this work.

Therefore, the attention is put on the direct cost. The direct cost is assumed to be the function of time and quality and can be referred to in terms of productivity (e.g. pair programmers productivity).

5.2 The plan of experiment-driven development

The presented concept of experiment-driven development proposes a framework, which should be easily adapted to the specific needs of the development software and the available resources.

As mentioned, the framework assumes a series of experiments, in which the evaluation is done based the consecutive versions of the developed software. Therefore, two aspects can be identified:

- The development sequence consisting of the consecutive experiments
- Individual experiments preparation and method of evaluation

5.2.1 Development sequence

EDD assumes three following phases in the development sequence:

1. **Guidelines mining**. The goals of this phase are exploration and examination of available support in particular domain as well as elaboration of experimental development plan with accordance to available resources and discussed problem. The output of this phase should be: an evaluating overview of available solutions, comprehensive report on existing limitations and problems, general guidelines for the future experiments organization. This phase should consist of one or two experiments considering the available solutions, organizational resources and time.

2. **Search for suitable functionality and design**. The goal of this phase is to work out the appropriate design and features set for the developed software. The proposed approach suggests early experimental verification of the new design concepts and very first prototypes in order to save time and provide valuable feedback in early stages of development. This phase should consist up to three experiments with regard to the developed product complexity and available time.

3. **Product releases and support**. This phase is believed to examine consecutive releases of developed software product and provide additional feedback on possible enhancement. This allows for tuning and tailoring the working product to new or evolving user needs and expectations. This phase should consist of at least one experiment with no limitation to the maximum number, except the available time and resources.

To sum up, the plan assumes several experiments divided into three stages. It is assumed that the most important from the design point of view is the second stage. It is expected to produce working solution, which can be later improved during third stage. The plan does not precise the method of verification of the developed software. However, it is suggested to provide two types of verification. The first ones are users feedback and general observations of the experiments' organizers. The feedback and observations are also the source of new requirements for the developed solution. The second type of verification is a metric, or better, a set of metrics, which would allow for measuring specific aspects of the developed software.

5.2.2 Evaluation method

The evaluation of the developed support must take into account the aspect of direct development cost of DAD, as was mentioned above. This allows for examining the posed thesis. Having proved that the developed solution Agile Studio efficiently supports the significant agile practices, it can be stated that DAD can be successfully used.

The idea behind the proposed method of evaluation is to compare the results obtained from examining co-located and distributed approach to the three selected practices: Pair Programming, Planning Game and Osmotic Communication. The comparison is driven by an experimental simulation of a real work (both co-located and distributed) environment, in which these practices are used. For each practice, Chapter 3 proposed a set of metrics. Each metrics is applied both to co-located and distributed settings.

To simulate both settings participants of experiments must be divided into two groups: for co-located and distributed settings. In each group participants are organized into teams. The metrics are applied to all teams. The final test result for a group is the average[3] result of all its teams. In order to minimize the influence of a human factor, the groups and teams are different for each metric.

The proposed size of a single team during experiments depends on the evaluated practice:

- 2 for Pair Programming (pair of developers)
- 2 for Planning Game (developer and customer)
- 3 for Osmotic Communication (3 developers)

As for Pair Programming, the size of 2 is the only possible option. As for other practices, the team size can be larger. The maximum size for both practices is undefined, but suggested to stay under 8 (within large team maintaining efficient direct communication is troublesome). However, having considered the limited number of available participants, the proposed sizes being close to minimum (which is 2 for both practices) seem the best compromise between simulation and statistical (number of teams for calculating the average result) needs.

The above proposed sizes are constant for the whole sequence of experiments. This makes the obtained results comparable.

5.2.3 Experiments organization

The common problem with this kind of research is providing unified conditions for the consecutive experiments, especially if they take place in distant moments of time. Apparently, this involves differences in two main aspects:

- participants
- available infrastructure

[3]It may happen that for a given team, the *Learning quality* metric produces undefined result. In this case, the result is not counted during average calculation.

As for the first aspect, the ideal case would be to take advantage of the same set of participants for all experiments. Unfortunately, this is hardly possible due to the length of the development process. What is more, the same set of participants does not guarantee the stability of this factor. It is because of the fact that people skills and performance are volatile. Thus, to minimize these issues, the following decisions were made:

- participants would be students of the 4th and 5th year of Computer Science faculty with a grounded knowledge and skills in programming and Java language
- participants would possess good theoretical and practical knowledge of the agile development with emphasis on the three investigated practices

As for the number of participants, it was assumed that it won't exceed 20, due to the limitations of available resources (including participants, time, laboratories, PCs, experiments' supervisors). This constitutes the main weakness of the conducted evaluation. The small amount of obtained data allows for formulation of only preliminary conclusions and prerequisites, rather than objective and general statements.

The second aspect, which is available infrastructure, involves the capabilities of the utilized network, hardware and software. The importance of these factors can be considerably limited by organizing experiments in the same spots with a reuse of the same machines and networks. Fortunately, during the conducted experiments, it was possible to reuse the same machine configuration in the three students' laboratories. As for the network connection, a local network in laboratories was utilized. This guaranteed that the testing conditions, even in the one-year or two-years perspective, were approximately the same.

Another interesting issue is the reliability of results due to so-called Hawthorne effect [73]. The effect refers to a temporary change of people behavior or performance due to the awareness of being a subject of an observation or study. According to [73], the effect usually results in an increased performance of the experiment participants. Thus, the results are not representative for the real working conditions. It seems that the proposed evaluation method minimizes the potential influence of this effect. The evaluation is based on the comparison of results obtained from the distributed and co-located settings. The effect influences both settings in the same way. Therefore, the comparison is expected to neutralize it.

Finally, the experiment space should be monitored during the whole experiment due to limited human perception and events generated by the supervisor, which also need to be registered. The minimal assumed registration is the cctv camera (common security camera) output.

5.3 Development story of Agile Studio

This section overviews the development of Agile Studio with respect to the proposed development framework of EDD. As described in Chapter 4, Agile Studio consists of four components, from which three are functional and meant to support the three discussed agile practices. To remind, these are:

- Distributed Pair Programmers Editor (DPPE) for Pair Programming practice
- Planning Desktop (PD) for Planning Game practice
- Osmotic Communicator (OC) for Osmotic Communication practice

The development story is presented from the perspective of requirements evolution throughout the sequence of experiments. Thus, it closely corresponds to the contents of Chapter 4 and the appropriate symbols preserve their meanings. The development story for each of the components is overviewed separately, except Agile Server and Distributed Pair Programmers Editor components which are overviewed together. This reflects the actual course of development. It should be also kept in mind that the components were not developed in parallel but in the following sequence: AS and DPPE, PD, OC. Hence, development of AS and DPPE influenced requirements for PD and OC.

To be able to refer to individual experiments, the following convention is introduced: $Ex.y$, where x and y represent the number of phase and the number of experiment within the phase respectively.

5.3.1 Distributed Pair Programmers Editor and Agile Server

Before the first phase of development was started, the preliminary requirements were specified. It was assumed that the developed solution must provide an advanced editor with a syntax highlighting and completion (DPPE-FR1) as well as support for the most popular programming languages (DPPE-FR2). As for editor synchronization (AS-FR17, DPPE-FR15) and collaboration, it should be possibly "real-time", which means transmission delays up to 2 seconds in one direction (AS-RR3). While this is a realizational requirement, it can be also treated as functional because it actually represents the idea behind DPPE. It was also assumed that the developed solution should provide voice and video communication channels (DPPE-FR25, DPPE-FR26).

Guidelines mining

This stage consisted of two experiments described by symbols **DPPE-E1.1**, **DPPE-E1.2**.

DPPE-E1.1 aimed at examining distributed pair programming supported by the common Internet communicators such as Skype [100], GoogleTalk [90], Gadu-Gadu [88] for voice communication (and also text communication) and VNC family of tools (TightVNC [104], UltraVNC [105], RealVNC [98]) for collaboration over the same code (achieved thanks to the shared Eclipse IDE on the remote desktop).

The experiment signaled significant vulnerability of supporting tools to the network configuration and capabilities. This included connection problems (as result of the strict firewall policy of the utilized network), low quality of voice and VNC transmissions (the voice was deformed which aggravated collaboration) and long transmission delays (even 2 seconds and more in one direction which practically disabled real time collaboration). Having considered these problems, the following three core aspects for the developed

support were emphasized: availability, quality (AS-RR4) and speed (as 2 seconds delays were to large, the requirement AS-RR3 was modified to assume 1 second delays) In order to provide greater availability and avoid blocked ports, it was decided that DPPE should provide a selection of the connection port (AS-FR2).

DPPE-E1.2 aimed at testing other available solutions that can be employed for distributed pair programming: TeamSpeak [103] for voice communication and NetMeeting [95] for both: voice&video communication and code collaboration. The selection was done based on the performed study on the most popular conferencing tools.

Apart the already known aspects (discovered in previous experiment), the experiment manifested other problems, which referred to the code collaboration. Firstly, the code could be modified by both driver and navigator, which sometimes led to conflicts, especially when the voice communication did not function properly (e.g. long delays). The second problem was, again, the latency, which considerably aggravated coding for the remote partner. As result, the remote partner was limited to the role of navigator. In order to become a driver, a new session on his/her computer needed to be instantiated. These problems were a motivation for applying the roles of driver and navigator to the DPPE requirements (DPPE-FR8, DPPE-FR9, DPPE-FR12, DPPE-FR13, DPPE-FR15, DPPE-FR19). Interestingly, according to participants, the video connection (established in NetMeeting) had no influence on their collaboration (DPPE-FR26).

Based on the experiences and gathered feedback, it was decided to take advantage of the client-server architecture with a server handling multiple users and collaborating sessions (AS-FR4, AS-FR6, AS-FR9, AS-FR11, DPPE-FR3, DPPE-FR6).

With regard to the collaboration on the shared documents (source files), it was specified that only the driver can add, modify and remove a shared document from the session (DPPE-FR12). Furthermore, the driver was indicated as the only responsible for the change of the current active document (DPPE-FR17). The partipants nicknames were also suggested (DPPE-FR4).

In order to make DPPE portable, it was decided to use Java as the implementation language (AS-RR1, DPPE-RR1) and base the messaging handling on polymorphism (AS-RR2).

Search for suitable functionality and design

This stage consisted of 3 experiments described by symbols **DPPE-E2.1**, **DPPE-E2.2**, **DPPE-E2.3**.

DPPE-E2.1 was meant to examine the first prototype of supporting tool. Its architecture was based on the remote method invocation (RMI). The client application was developed as a plugin to a lightweight Jext editor [109].

The experiment indicated several issues. Firstly, according to gathered feedback, the programming support offered by Jext editor occurred to be not satisfactory (lack of

debugger and integrated tools such as CMS client). It was suggested to implement DPPE as a plugin to rapidly developing feature-rich Eclipse IDE based on a mature plugin-based RCP framework (modification of DPPE-FR1, 2). Another point was to provide more synchronization considering the view of the code, such as the caret position (DPPE-FR16), remote partner's selection (DPPE-FR16, DPPE-FR24) and the active document (DPPE-FR17)

As for communication, the developed voice transmission module occurred unsatisfactory (quality) and unreliable (connection problems). Therefore, it was decided to add a text chat functionality (DPPE-FR21) and drop the integrated voice connection (DPPE-FR25) in favor of the existing mature solutions (DPPE-FR23), such as tested Skype and TeamSpeak.

DPPE-E2.2 evaluated a new version of DPPE implemented as a plugin to Eclipse IDE. This automatically provided developers with a large set of programming features. As for the server side, it remained the same (RMI).

Several enhancements were proposed. The first one was the list of sessions and users updated every time a new session or user is added (AS-FR12, AS-FR13, DPPE-FR5). Provided a list of sessions, it should be possible to delete a selected empty session (AS-FR10). Another enhancement proposal concerned the management over the shared documents, including listing and removing them from the session (DPPE-FR10, DPPE-FR20, DPPE-FR13). As RMI-based server-side occurred unsatisfactory (lack of notification mechanism, portability problems, mediocre connection speed), it was decided to use Java sockets as a well-known, reliable and fast technology. This resulted in a new design of Agile Server, which was based on the concept of sharing objects, such as pair programming documents. Sharing the objects (documents) among session participant's assumed that every document belongs to exactly one session (AS-FR16) and the session state should be transmitted to every new session participant (DPPE-FR13). In addition, it was assumed that while in DPPE only the driver modifies the data and view of the shared document (DPPE-FR14, DPPE-FR11), Agile Server should be left opened to other concepts and, therefore, allow any session participant for managing the (shared) session objects (AS-FR15).

DPPE-E2.3 evaluated next version of DPPE with a new approach to server-side (Java RMI was abandoned in favor of dedicated server). The experiment reported several enhancements and requirements changes. Firstly, it was suggested to remove the restriction of only two participants in a session. While the driver can be only one, the number of navigators should be unlimited (DPPE-FR8), which can be useful in a case when the pair collaborates with an expert programmer.

Having considered more than two programmers in a session, it was decided to abandon the remote selections (DPPE-FR24) in favor of mouse pointers of the remote programmers (DPPE-FR22). Additionally, to clearly indicate the current role of the programmer, it was suggested to take advantage of the editor background color: *green* for driver and *blue* for navigator (DPPE-FR14). What is more, it was pointed out that both

users nicknames (AS-FR5) and session names (AS-FR7) should be unique within a single instances of DPPE and Agile Server.

The experiment resulted also in the extension of the DPPE-FR2 requirement which assumed the support for only the most significant programming languages. Having considered the large number of programming languages handled by the employed Eclipse IDE, it was decided to extend the DPPE support onto all of them.

Product releases and support

This stage consisted of one experiment **E3.1**. The following enhancements were proposed. The first one concerned the security issues by suggesting a password-based protection to the session access (AS-FR7, AS-FR8). The second assumed more synchronization concerning the view of the shared documents. Some of the Eclipse editors support code folding which allows for hiding large fragments of code which facilitates its analysis. It was decided that code folding should be added to the synchronized driver actions (DPPE-FR16). Thirdly, as a result of the new synchronized action, it was decided that Agile Server must provide a plugin-based interface for future extensions concerning new messages types and shared objects (AS-RR5).

Other enhancement proposal (DPPE-FR6, AS-FR14) included various messages types for the integrated text chat: unicast, multicast (to session participants) and broadcast (all connected users). Similarly, it was suggested to extend the chat with a ability to display important system messages such as notification on new user connected (DPPE-FR7).

5.3.2 Planning Desktop

Before the first phase of development was started, the preliminary requirements were specified. It should be noted that, at this time, the Agile Server was already developed. This considerably influenced the preliminary analysis.

As result, the first requirement (PD-FR1) referred to the features provided by Agile Server, that is users, sessions and connection issues. Similarly, based on previous experiences it was stated that the the solution must be supported by a voice communicating software (PD-FR23). What is more, it was assumed that the supporting application should provide a fixed size working area (PD-FR2) within the user stories (PD-FR6) can be created, modified and deleted (PD-FR8). Based on the development of Agile Studio and Distributed Pair Programmers Editor, it was assumed that full synchronization of workspaces should be implemented (PD-FR13, PD-FR5). Moreover, as for non-functional requirements, it was assumed that the support should enable a real-time collaboration (PD-FR2). Another requirements considered its intuitiveness and user friendliness (PD-FR3) so that the supporting software can be operated by a person with only basic computer skills (e.g. customer). Due to the previous design decisions, Planning Desktop was to be implemented in Java, which was also meant to make it portable (PD-FR1).

Guidelines mining

This stage consisted of one experiment (described by symbol **PD-E1.1**). It examined the distributed Planning Game with a use of common Internet communicator such as Skype [100] and Google Docs and Spreadsheets [89] as well as NetMeeting [95] for sharing the same workspace.

The experiment indicated several issues. Firstly, according to its participants, NetMeeting and sharing the same screen in a real-time better suited the needs of Planning Game then Google Docs. While using NetMeeting, it was suggested that the screen size should be unlimited with scrolling functionality (PD-FR2, PD-FR3), which was modification of previous assumptions. As NetMeeting offers a video communication tool, a similar support was suggested for the developed solution (PD-FR24). Having experimented with different ways of representing User Stories information (common text, spreadsheet form, text files on a desktop), it was decided that the story should be represented by a rectangle (story card) with story title and a double-click for accessing the edit window (PD-FR7). Finally, a simple text chat was suggested as well (PD-FR22).

Search for suitable functionality and design

This stage consisted of 2 experiments described by symbols **PD-E2.1** and **PD-E2.2**.

PD-E2.1 evaluated the fist prototype of the supporting tool. As result, several new requirements were added, mainly regarding the User Story card. Firstly, it was suggested to make the User Story card resizable for every user (PD-FR7, PD-FR8) to handle long story titles which could not fit in a fixed size of the card. At the same time, it was assumed that the User Story data (e.g. the length of the story description) should not be limited (PD-FR10). Thirdly, participants suggested to take advantage of the MoSCoW rule (see section 2.3.1) for prioritizing the stories. (PD-FR11). Another changes considered the synchronization of the session data. The full synchronization occurred troublesome, mainly because of the bandwidth requirements and, in result, poor performance. Additionally, according to participants, it was not needed. Therefore, it was decided to synchronize only the significant session events, such as User Story modification, relocation or size change (PD-FR5, PD-FR13). An interesting suggestion was also to provide zooming functionality, which would allow for adjusting the current view to the individual needs of the Planning Game participants (PD-FR4). Finally, storing session data (including both data and UI properties of User Story) on the server-side was proposed (PD-FR21).

As for realizational requirements, it was suggested to provide a java applet interface for the developed tool (PD-FR4). It was meant to allow unexperienced users for a quick and easy access to the tool. Needless to say, this was compliant with the previous requirement of tool intuitiveness and user friendliness (PD-FR3).

PD-E2.2 examined the enhanced prototype, which seemed to suit the needs of participants. As result, only three new requirements were added. Firstly, it was reported that it

may be useful to relocate a whole group of story cards, instead of relocating each story individually. By a group it was defined a selection of Story cards. This required defining the selection mechanism (PD-FR14). An interesting proposal was made to provide a feature of linking User Story cards. Such a link (connection) can be used to indicate dependencies between User Stories (PD-FR18). Another valuable idea was to provide an overview of the current work status in a form of a table with all stories listed (PD-FR19).

Additionally, the experiment examined the requirement of applet interface (PD-RR4). It occurred that applet technology imposes some constraints. What is more, it can be replaced by a Java Web Start technology [91]. As result, the requirement was dropped.

An interesting observation was made that the provided video connection support was not actually utilized. When asked, participants confirmed this observation by stating that this channel provides irrelevant data and takes up the communication bandwidth required by other, more important channels (PD-FR24).

Product releases and support

This stage consisted of 2 experiments described by symbols **PD-E3.1** and **PD-E3.2**.

PD-E3.1 examined the first release of Planning Desktop. It resulted in adding three new requirements of rather minor importance. First requirement considered better visualization of the User Story priority. It was suggested to take advantage of the card color, e.g. red for highest priority (PD-FR12). Another suggestion was to improve the multiple selections feature (PD-FR14) and provide a way to group and ungroup the selected User Story cards (PD-FR15). As these operations require synchronization, the requirement (PD-FR13) was updated as well. The third suggested enhancement referred to the list of User Stories (PD-FR19). According to participants, a simple bar chart, which shows the number of created Stories with respect to their priorities, would be useful (PD-FR20).

PD-E3.2 resulted in adding two new requirements. They referred to the grouping feature (PD-FR15), which was added based on the feedback gathered during previous experiment. It was advised to enhance it by allowing user to fold the group so that multiple stories can be represented by one group card (PD-FR17). This can be useful when Stories need to be organized into iterations or a set of specific Stories. To make the group cards more descriptive, a group name should be provided (PD-FR16). Obviously, all these operations require synchronization. As result, the synchronization requirement (PD-FR13) was modified.

5.3.3 Osmotic Communicator

Specification of the preliminary requirements was considerably influenced by the previous experiments dedicated to other two components of Agile Studio. As result, the first defined

requirement was to handle the basic Agile Server operations, such as user and session issues (OC-FR1).

Having considered the needs of the Osmotic Communication practice, it was assumed that the developed solution must provide a shared working area in a form of window (OC-FR9). The Information Radiators (see sections 2.3.1 and 3.1.3) should be represented as sizeable and movable rectangles with text information (OC-FR12, (OC-FR10). Every user in session should be allowed to perform basic operations on the Radiators, such as creation, modification, change of location and size etc. (OC-FR16, OC-FR17). Obviously, add changes to a Radiator should be immediately broadcasted to all users in session (OC-FR19).

Having in mind results of previous experiments, it was decided to support developed Osmotic Communicator with a voice communication software (OC-FR30).

Guidelines mining

This stage consisted of one experiment (described by symbol **OC-E1.1**). It was aimed at examining the usefulness of the available solutions. The selected combination was a popular implementation of wiki concept[8, 68] called Doku Wiki[84], which was accompanied by a common text communicator Gadu-Gadu [88]. The idea was to handle Information Radiators with Doku Wiki, while Gadu-Gadu was meant to provide the communication channel.

According to participants, the proposed environment lacked of several important features. Firstly, handling two communication channels in two separate applications occurred troublesome. Therefore, it was strongly suggested to integrate text communication (chat) in the developed solution (OC-FR26). The lack of an instant notification after radiators were modified was the reason to drop the requirement of a dedicated application window (OC-FR9) and integrate the solution with a system desktop (OC-FR4). The idea was that any modification to the radiators would be immediately spotted. This new concept required an assumption that the local working areas (desktops) can differ in size, depending on the local settings (OC-FR5).

Having assumed a desktop integration, a very interesting issue was how to operate the application without any dedicated window. A suggestion was made to take advantage of the system tray (OC-FR2). After clicking the application icon in the tray, a menu with all required options appears (OC-FR8).

As for the Information Radiators, it was assumed that for better reception of the actions of the remote users, a full synchronization, including dragging and resizing the Radiators, should be provided (OC-FR25). Additionally, a suggestion to specify titles for the Radiators was made as well (OC-FR12).

Finally, participants defined that the session state should be persistent on the Agile Server (OC-FR3).

Search for suitable functionality and design

This stage consisted of 2 experiments described by symbols **OC-E2.1** and **OC-E2.2**.

OC-E2.1 evaluated the first prototype of the Osmotic Communicator application, having

a considerable impact on the defined requirements. Firstly, while the idea behind the full real-time synchronization seemed reasonable, its real application occurred troublesome. The large amount of the broadcasted data drastically decreased the application performance and resulted in a long delays. What is more, it occurred that a real-time synchronization of the Radiators changes was not really needed. According to participants, it can also detract remote users from their tasks. As result, this requirement (OC-FR25) was rejected.

A surprising remark was that the provided voice channel was not as important as it was expected. In most cases, the information could be efficiently exchanged using provided text channel and information radiators. Another advantage of the text channel was that it did not interfere with the ongoing work as the voice channel did. Another argument to quit the voice support requirement (OC-FR30) was that it may lead to conflicts with the voice channel of other practices (e.g. Pair Programming), in which the channel is a must.

Another issue was the synchronization of the Radiators locations. Having considered the differences between the remote environments (desktop size, opened windows, desktop content) it was suggested to provide an option to turn off this kind of synchronization (OC-FR19, OC-FR20)

An interesting suggestion was made to enable Radiators to store images loaded from the file (OC-FR24). Obviously, this modified the requirements regarding Radiator presentation (OC-FR10) and modification (OC-FR17).

To make the work more convenient, it was requested to implement feature of hiding, showing and searching Radiators (OC-FR21, OC-FR23). In order to hide the Radiator, a proper option from its popup menu needs to be selected. To show the already hidden radiator, a list of all radiators was suggested (OC-FR22). Another proposed feature referred to the communication chat and its lacking ability send unicast (to one user), multicast (to users in the same session) and broadcast (all users on the server) messages (OC-FR27). Finally, it was suggested that the default session loaded on the application start, should be the last utilized session, if it exists (OC-FR3).

OC-E2.2 resulted mainly in changes to the Radiators contents. While adding images to Radiators (OC-FR24) seemed a useful solution in the beginning, during longer work it turned out to be inconvenient (scrolling and refreshing large images). Therefore, the idea was replaced with a concept of files addition (OC-FR13, OC-FR10, OC-FR17). The file should be represented as a link and opened by application suggested by a system. In this way, images of any types should be handled as well.

Another requested improvement was to allow for changing the Z-order of the working area (OC-FR7). In this way, the whole working area and its Radiators can be moved back to desktop, leaving other application windows at the top. Next proposed feature referred to the communication chat and its lacking ability to filter out some of the message types (OC-FR27). An example can be filtering global messages. Finally, a message history was requested as well (OC-FR29).

Product releases and support

This stage consisted of 2 experiments described by symbols **OC-E3.1** and **OC-E3.2**.

OC-E3.1 positively evaluated the first release of Osmotic Communicator. As result, several, rather minor, enhancements were proposed. One of them was ability to set the background color of Radiator from the popup menu (OC-FR11). This was meant for differentiating between different types of Radiators. Another suggestions considered adding multiple files to a Radiator at once (OC-FR15), using clipboard content for pasting text into a Radiator (OC-FR18) and, by default, showing the workspace area on the top of other windows (OC-FR6).

OC-E3.2 was assumed to verify previous enhancements. Not many new enhancements were expected. The following two were proposed. First one assumed saving files stored in the Radiators to the disk (OC-FR14). The second one specified a different way of showing the incoming messages (OC-FR28). For private messages, a separate window was suggested, while for other messages, a fixed-time preview in the bottom-right corner of the workspace area was suggested.

5.3.4 Observations and findings

Several observations and findings were made during the presented development story. Firstly, in order to verify the expected outcome of the realized concept, the history of all changes to the requirements specification was collected. Than, for each component, the number of changes was calculated with regard to the organized experiments. Table 5.1 and Figure 5.1 show the obtained result. The **Before** represents the preliminary analysis of the requirements, before the development and evaluation of a given component was started.

Table 5.1: Requirements changes during Agile Studio development

	Before	**E1.1**	**E1.2**	**E2.1**	**E2.2**	**E2.3**	**E3.1**	**E3.2**
AS	2	3	6	0	6	3	3	-
DPPE	5	0	12	10	7	5	3	-
PG	10	5	-	8	5	-	4	2
OC	10	9	-	12	7	-	4	2
Avg	**6.75**	**4.25**	**9**	**7.5**	**6.25**	**4**	**3.5**	**2**

It can be noticed that the number of changes considerably decreases in the advance stage of development. In the last experiments it reaches the average value of value of 3.5 and 2. Having also in mind that these last changes were rather minor, it can be stated that the preliminary

expectations towards EDD were correct. The most productive in a sense of requirements extraction are the first experiments from the first and second phase of development.

The obtained results indicate two cases in which experiments produces no requirement changes, that is in E1.1 for DPPE and E2.1 for AS. The first case was the result of the unexpected connection problems which limited the scope of the experiment. In the second case, the experiment focused on evaluating functional component DPPE. The proposed changes did not affect requirements for the server-side AS.

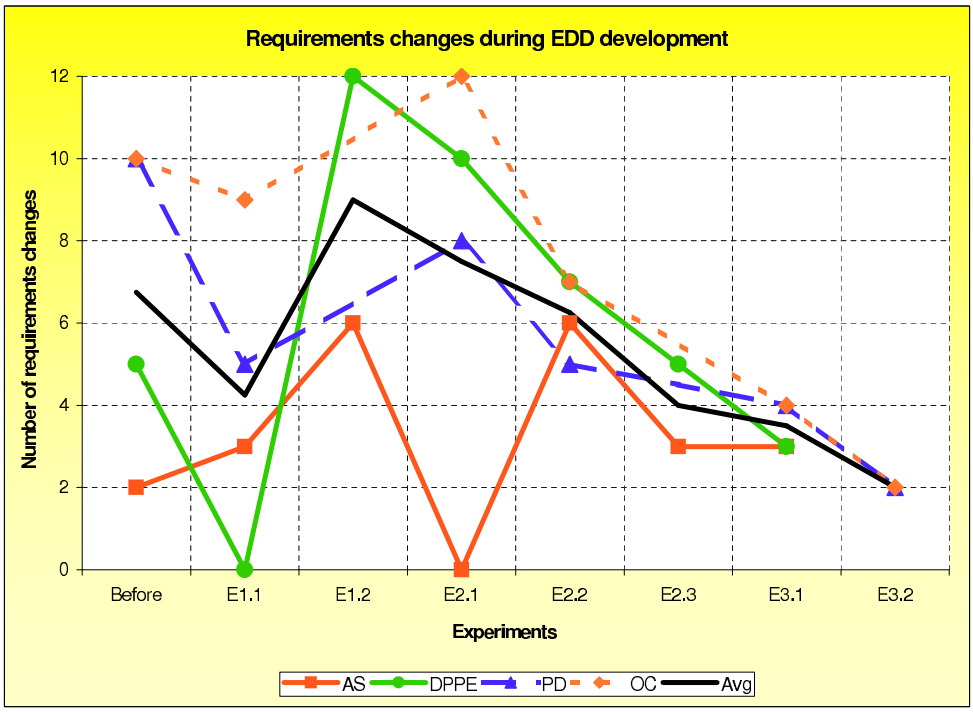


Figure 5.1: Requirements changes during Agile Studio development

While the first phase is dedicated to examining the subject of study, the results differ considerably. From the beginning of the second phase, the number of changes gets stabilized. The most productive is the first experiment, which examines the first prototype. The third phase, as expected, produces a few, mainly minor, changes.

A surprising outcome was dropping of the video channel requirement in Distributed Pair Programmers Editor and Planning Game. According to participants, they data it provided was not valuable for the task completion. In addition, it disturbed participants during their work as well as took up a considerably part of the available bandwidth. Alike, voice channel was considered as not mandatory in Osmotic Communicator, mainly due to two facts: low intensity of the exchanged data and no search/history capabilities of the voice channel. Another issue was a likeable conflict with other voice channels, for example with these utilized in Pair Programming.

5.4 Evaluation results

This section presents the results obtained from the experiments with a use of proposed metrics. The evaluation of Agile Studio components was started at the beginning of second phase (first prototype) of development and lasted until the current release. As mentioned, during the first phase, the experiments evaluated the existing and popular tools that can be utilized to provide support for selected practices. Measurements from these experiments allow to make a preliminary comparison against Agile Studio performance.

An issue that must be strongly emphasized here is the number of participants. Due to the existing limitations, the metrics were applied to a small number of participants, especially when consider the need for their division among co-located and distributed modes. Due to this limitation, the obtained results are studied only with the use of descriptive statistics (average, standard deviation). The small number of tested population is a considerable weakness of this research and should be taken under account when the conclusions are drawn.

Before the results are presented, the utilized symbols must be explained:

C stands for co-located and standard work with the practice

D stands for distributed work with a use of the developed Agile Studio

Detailed results can be found in Appendix D.

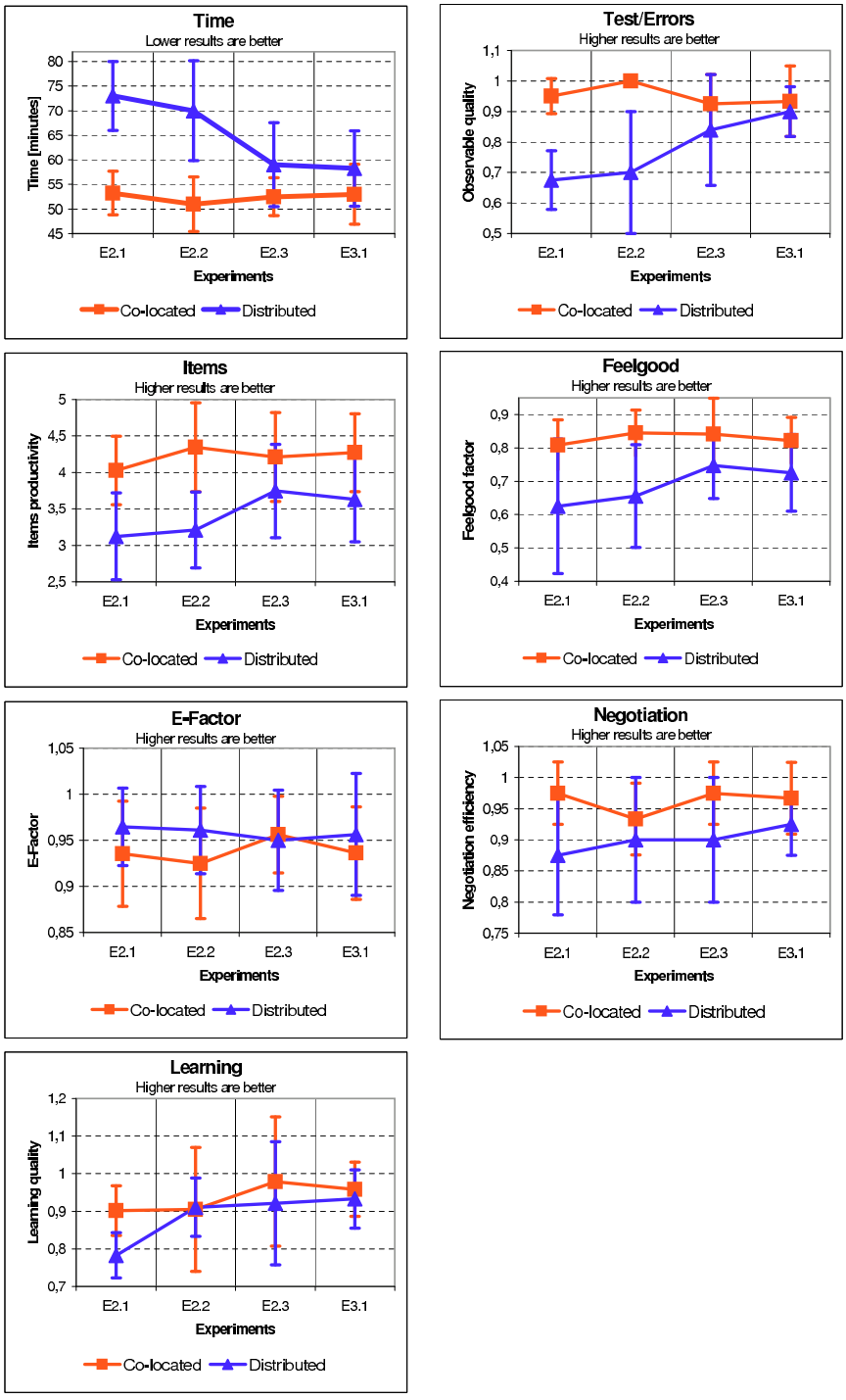
5.4.1 Distributed Pair Programmers Editor

Table 5.2 presents the collected results for DPPE. These results are visualized by the charts in Figure 5.2. Higher results are better, except Time metrics were lower results indicate better performance.

Table 5.2: Evaluation results for Pair Programming support

Test scenario	Setting	E2.1	E2.2	E2.3	E3.1
Time	C	53.25	51	52.5	53
	D	73	70	59	58.25
Tests/Errors	C	0.95	1.0	0.93	0.93
	D	0.68	0.7	0.84	0.9
Items	C	4.03	4.34	4.21	4.27
	D	3.12	3.21	3.74	3.63
Feelgood	C	0.81	0.84	0.84	0.82
	D	0.63	0.66	0.75	0.73
E-Factor	C	0.94	0.93	0.96	0.94
	D	0.96	0.96	0.95	0.96
Negotiation	C	0.98	0.93	0.98	0.97
	D	0.88	0.9	0.9	0.93
Learning	C	0.9	0.9	0.98	0.96
	D	0.78	0.91	0.92	0.93

Figure 5.2: Evaluation results for Pair Programming support



The results obtained during consecutive experiments for co-located Pair Programming are relatively similar. The reported variations are acceptable. As for DPP, most of the

results indicate a gradual improvement of the evaluated solution. This is especially visible for experiments E2.2 and E2.3 in the following metrics: Time, Tests/Errors, Items. This can be interpreted as a sign of considerable changes made to the developed solution, which is actually true. The substantial change was the replacement of the server-side. The removal of few bugs also made the effect.

One can notice that Time and Test/Errors project similar trend. It is because the measurements for Time was partly taken during Test/Errors scenario. Similar case was done for Items and Feelgood, whose results are corresponding as well. As for Items, this is productivity measurement indicate considerable improvement of the DPPE efficiency. Taken the maximum result, the DPP was worse than PP for approximately 0.5 tasks per hour per person. Having considered the duration of a single task (unit test implementation) which was approximately 7 minutes in average, it can be calculated that DPP with a use of DPPE is approximately 12% less productive than co-located PP. This gives about 1 hour in a 8-hours working day. This is an acceptable result.

As for the Feelgood metric, the improvement is mainly the result of the higher assessment of support efficiency. The work satisfaction reported mediocre improvement. According to participants, lack of the direct contact in Pair Programming is a serious drawback. As for the task easiness, no relation to the setting or support efficiency was reported.

As for Negotiation and Learning metrics, the reported performance of DPPE is very close or even same as for co-located setting. There might be two reasons for this. Firstly, it was observed that a majority of information was passed through the voice channel (in both settings). This explains only slight improvement of the results, in spite of the problems with the first prototypes of DPPE (in case of Learning in E2.1, there was a connection problem). The second reason is the participants' learning factor of the utilized DPPE. These scenarios were run as the last ones. Having considered this, a question arises on the efficiency of the prepared task. The problem here is to properly simulate and stimulate they knowledge transfer, so that it reflects the real work conditions.

E-Factor shows interesting results. It seems that distributed setting positively influences the work concentration. Perhaps the reason for this are, paradoxically, limitations of distributed communication. Because of them, developers tend to take a maximum advantage of the available channel. As result, they minimize off-topic conversations and other activities.

5.4.2 Planning Desktop

Table 5.3 presents the collected results for PD. These results are visualized by the charts in Figure 5.3. Higher results are better, except Time metrics were lower results indicate better performance.

Table 5.3: Evaluation results for Planning Game support

Test scenario	Setting	E2.1	E2.2	E3.1	E3.2
Time	C	51.33	51	52.5	51.67
	D	79.5	65.33	59	60
Items	C	7.24	7.59	7.37	7.26
	D	4.32	5.82	6.07	6.17
Feelgood	C	0.79	0.81	0.82	0.82
	D	0.56	0.72	0.78	0.78
E-Factor	C	0.94	0.96	0.96	0.93
	D	0.98	0.98	0.95	0.98
Learning	C	0.97	0.92	0.94	0.96
	D	0.88	0.88	0.9	0.9

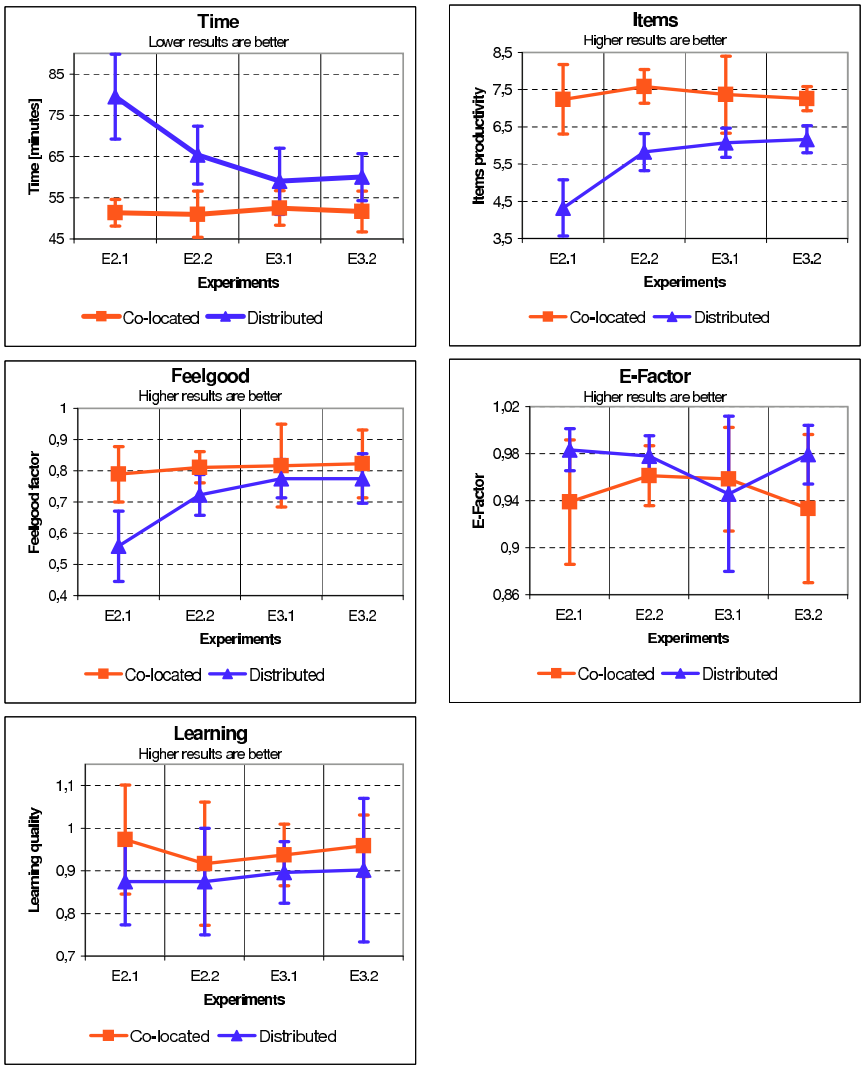
Firstly, based on the results, it can be stated that the performance of co-located mode remained stable. As for distributed mode, a considerable improvement was reported, especially for metrics Time, Items and Feelgood. In all these metrics, the initial results from the first prototype (E2.1) of PD are considerably worse than for next experiments. The reason for this was the cumbersome full synchronization of the session data, mainly position and size of the User Story cards. It often cluttered the available bandwidth, which effected in a poor performance of the team. After the requirement of full synchronization was dropped, the performance improved. As for the last two experiments, the requirements changes (see Table 5.1) were minor and they considered minor features. This is reflected by the obtained results as they are quite similar.

One can notice the same trends of Time and Items metrics. This is due to the fact that most measurements for Time was taken during Items scenario. The Time metric reflects the effectiveness of the Planning Game process. While the distributed teams worked longer, the last results are considerably close to these ones achieved by co-located teams. Taken the average of 7.5 minutes per hour, it gives approximately 1 hour longer planning during 8-hours planning session. This is, again, acceptable.

The question arises, however, how to verify whether the planning was actually finished. The metric measured only the length of the task. To complete the task, all aspects of the assigned problem needed to be covered by the created stories. Therefore, the Time metric should be analyzed together with Items. The results indicate that distributed teams produced less amount of stories. This means they were not as effective as it was reported in Time. Still, the differences are minor and acceptable, especially when one will consider the fact that too atomic and detailed Stories are often merged into larger ones.

As for the Feelgood factor, the results of distributed setting considerably improved during development, which was caused by the increase in the Satisfaction and Efficiency assessments. The third assessment (Easiness) was not influenced and remained similar for both settings during whole development process. The Satisfaction assessment is particularly interesting.

Figure 5.3: Evaluation results for Planning Game support



While in case of Pair Programming participants complained on the lack of direct contact, in case of Planning Game this was not the case. A possible explanation is that while Pair Programming is more creative and intensive activity, Planning Game is focused on listening and collecting identified requirements. Therefore, it requires less intensive communication and the indirect contact is acceptable. Another explanation is that the task, which was prepared for Planning Game, failed to force collaborators to more intensive communication.

The results for E-Factor remind these of Pair Programming. The distributed teams were slightly more focused on the task than co-located ones. A very interesting observation is that the higher result was obtained during first experiments, when the troublesome prototype was evaluated. The conducted observations indicated that the teams, which were experiencing prototype problems, were strongly motivated to finish the task and did not have time for off-topic activities. This confirms the conclusion made in previous section. In addition, the obtained results indicate higher deviations than in case of Pair Programming. This is due to

different set of participants, with some more talkative students taking role in.

Finally, Learning results report similar performance in both modes. Moreover, the performance in distributed setting was not considerably changing. As mentioned previously, this may be due to the fact that the majority of the knowledge is transferred through the voice channel. This signals possible improvements to the prepared task. On the other hand, it should be remembered that the task should simulate real work assignments and conditions.

5.4.3 Osmotic Communicator

Table 5.4 presents the collected results for OC. These results are visualized by the charts in Figure 5.4. Higher results are better, except Time metrics were lower results indicate better performance.

Table 5.4: Evaluation results for Osmotic Communication support

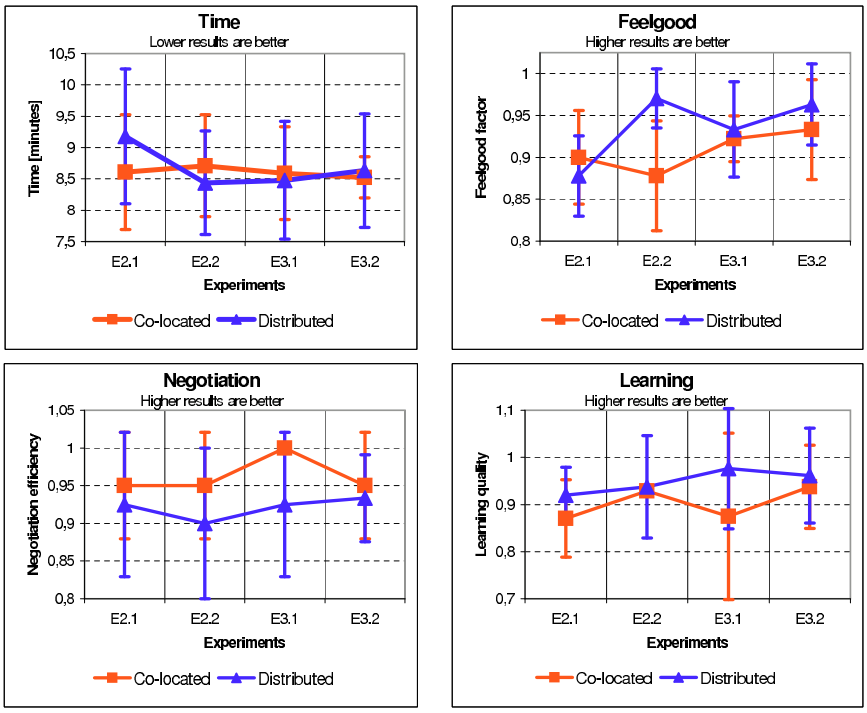
Test scenario	Setting	E2.1	E2.2	E3.1	E3.2
Time	C	8.61	8.71	8.59	8.53
	D	9.18	8.44	8.48	8.63
Feelgood	C	0.9	0.88	0.92	0.93
	D	0.88	0.97	0.93	0.96
Negotiation	C	0.95	0.95	1	0.95
	D	0.93	0.9	0.93	0.93
Learning	C	0.87	0.93	0.88	0.94
	D	0.92	0.94	0.98	0.96

Judging by the charts presented in Figure 5.4, there are two main differences when compared to previous practices. The first one is a slight instability of the results, mainly of the Feelgood factor and the co-located teams. The second difference is that the results for distributed setting are comparable or even better than for co-located teams.

There are two reasons for the aforementioned instability. Firstly, in case of the co-located teams, their number was reduced to only two, which was due to the resources limitations (to remind, the team consisted of 3 people in the OC evaluation). The purpose of this decision was to formulate a larger number of distributed teams, which would produce more reliable results. The second reason might be the practice itself, which is only dimly specified, and, thus, difficult to measure.

As for Time metric, it measured the effectiveness of the information search. The results show a large improvement between first and consecutive experiments. This is because of the full synchronization feature causing troubles in the first prototype. Interestingly, next experiments reported comparable results, slightly outperforming the co-located teams. This is due to the fact that searching process is much easier on a computer. While co-located

Figure 5.4: Evaluation results for Osmotic Communication support



teams utilized mail clients for searching in the provided inboxes (for mails with the specific information), some part of the information was displayed on physical Information Radiators on a whiteboard. Be that as it may, the slight differences are within the bounds of a statistical error.

The Feelgood factor reports stable easiness assessments regardless setting and evaluated version of OC. As for the other two, distributed teams produced better assessments than co-located. According to participants, the co-located Osmotic Communication may be disturbing. This is not the case for distributed setting, when the incoming messages can be ignored and read later. In addition, in case of OC, the fast access to the information located on system desktop positively affected the Satisfaction assessments.

As for Negotiation and Learning results, they revealed the advantages and disadvantages of the developed solution. The disadvantage is that it is not designed for an intensive communication which was required to complete task for Negotiation. Still, it occurred possible for obtain satisfactory results in the last experiment. As for the advantage, the OC is more efficient when it comes to more static information. The written information is easier to receive and remember. Moreover, being easily accessible, it is unconsciously reviewed several times during work, which rather does not happen with the overheard or emailed information. This explains better Learning results for distributed teams.

5.5 Results discussion

Having analyzed the obtained results, several aspects need to be discussed. The most important one is the comparison between results for co-located and distributed teams. According to the results for metrics Time, Test/Errors and Items co-located teams occur to be more efficient. However, their advantage over distributed teams is not significant. In case of Pair Programming and Planning Game, the difference is estimated as 10% to 12%, which is an acceptable result, especially when one considers high cost of delegations and related expenditures. In the case of Osmotic Communication and Time metric, the results for distributed teams are relatively similar or even better than for co-located ones. This is because software-based communication is superior over the face-to-face communication in such aspects like information storing and searching.

A very interesting conclusion can be drawn from the results obtained for metrics Feelgood and E-Factor. It seems that software-supported collaboration is estimated higher in terms of enjoyment and subjective comfort (that is only based on ones feelings and impressions). While these are physiological aspects, they can be important in a longer perspective. As for the assessment of a task easiness, it seems independent from the working conditions. Therefore, it should not be taken under consideration in future work. Another conclusion is that distributed setting positively influences the E-Factor (slightly higher results for distributed teams). Observation was made that people tend to be more focused on their tasks when the communication is limited.

In case of the last two metrics Negotiation and Learning, it seems that these behaviors depend mainly on the utilized voice channel. This conclusion is based on the observation that the improvement of the results for distributed teams was minor throughout the series of experiments. On the other hand, supporting software can considerably improve the process of understanding and remembering new information. This is observed in the case of Osmotic Communication.

Another interesting aspect is how the developed Agile Studio compares to the existing tools of general application that can be utilized to support selected agile practices. As mentioned in the development story, first experiments were dedicated to examination of these tools. Based on the conducted investigation (see sections 2.4 and 3.2) the following ones were selected: VNC family of tools[98, 104] for Pair Programming, NetMeeting[95] for Planning Game, DokuWiki[84] and Gadu-Gadu[88] for Osmotic Communication. Alike Agile Studio, they were supported by Skype, which provided the voice channel. The comparison is possible thanks to the fact that the proposed metrics were also applied during first experiments for each practice. Table 5.5 shows the collected results. The utilized symbols D_{worst} D_{best} D_{exist} mean respectively: the worst result of Agile Studio, the best result of Agile Studio, the result obtained for the existing tools. To remind, higher results are better, except Time metric where lower result means better performance.

Table 5.5: Results comparison between AS and existing solutions

Practices		Time	Tests/Errors	Items	Feelgood	E-Factor	Negotiation	Learning
Pair Programming	$\mathbf{D_{worst}}$	73	0.68	3.12	0.63	0.95	0.88	0.78
	$\mathbf{D_{best}}$	58.25	0.9	3.74	0.75	0.96	0.93	0.93
	$\mathbf{D_{exist}}$	67.6	0.73	3.31	0.61	0.98	0.9	0.92
Planning Game	$\mathbf{D_{worst}}$	79.5	-	4.32	0.56	0.95	-	0.88
	$\mathbf{D_{best}}$	59	-	6.17	0.78	0.98	-	0.9
	$\mathbf{D_{exist}}$	64.6	-	5.3	0.7	0.98	-	0.91
Osmotic Communication	$\mathbf{D_{worst}}$	9.18	-	-	0.88	-	0.9	0.92
	$\mathbf{D_{best}}$	8.44	-	-	0.97	-	0.93	0.98
	$\mathbf{D_{exist}}$	8.49	-	-	0.94	-	0.95	0.95

According to the comparison, Agile Studio significantly outperforms existing solutions in terms of four metrics: Time, Tests/Errors, Items and Feelgood metrics. With regard to the other three metrics, the differences are minor. This can be explained by the fact that these metrics depend on the utilized voice channel, which was the same in both cases. With regard to Osmotic Communication, the results for both approaches are relatively similar. An exception is the Feelgood factor, which indicates better developers' attitude towards Agile Studio. Be that as it may, the wiki concept seems to be successful in replacing local Osmotic Communication. However, at the time of starting this research, the wiki solutions were not common (as they are now).

To sum up the presented conclusions and observations, two following statements can be formulated:

- The results verify the accuracy of the proposed EDD concept. It allows for adjusting the developed software to the specific needs (even those which are difficult to specify during preliminary analysis).
- The results obtained for distributed teams supported by Agile Studio are relatively close to those obtained for co-located teams. In some aspects, distributed teams seem to outperform the co-located ones. This confirms the posed thesis stating that with a proper software support, it is possible to overcome the limitations of distributed setting, emerging from the large distance and lack of direct contact.

While these statements confirm the posed thesis, two facts need to be remembered. The results are obtained based on a simulation environment with a considerably small testing

population. The second, less important, fact is not including the aspect of time zones in terms of distributed collaboration. This facts should be considered in future work.

5.6 Summary

The chapter presented the proposed concept of Experiment-Driven Development and they way it was applied to develop Agile Studio, a DAD supporting solution described in Chapter 4. The main idea behind the proposed concept is to take advantage of experimental simulation of a real working conditions in order to evaluate the developed software and tune it to the users' needs.

The chapter was started from the discussion on the aspect of DAD evaluation. Next, the development plan of EDD was outlined including the proposed evaluation method and organizational issues. Furthermore, the chapter described the development story of each Agile Studio component with emphasis put on the developed requirements, which were defined in Chapter 4. Finally, the chapter presented and discussed the obtained results from the evaluating experiments. The discussion is concluded with a confirmation of the posed thesis.

Chapter 6

Conclusion and future work

The goal of this dissertation was to examine two following statements:

- Supported by specific software tools, it is possible to apply Agile Development and its practices to distributed teams
- The overall cost of software production in DAD case can be maintained or even decreased when compared to AD case applied to a distributed team

This involved a classification of the overall production cost and elaborating the proper approach for the needs of this work. As result, the dissertation attempted to perform a conceptualization of DAD discipline, which was necessary to select proper agile practices for the purpose of this work. An important issue was also to propose a set of metrics, which should allow for evaluating the selected practices both in distributed and co-located settings. Another crucial goal of the work was to develop a dedicated supporting software for the selected practices. This required establishing a suitable development approach which should allow for frequent evaluation of the development results with a use of the proposed metrics. The evaluation was to be done with an assumption that the overall cost can be classified into three categories. Being able to approximate only direct cost, the research was meant to examine the productivity and quality of the output of selected practices in both settings. The expected outcome was the statement that, while DAD obtains worse results in comparison to standard co-located setting, the differences are minor, and, for sure, they do not spoil large savings coming from the lower indirect cost of production process.

The author, besides confirming the posed thesis, would like to emphasize the following specific and detailed achievements of this work:

- Developed supporting solution called Agile Studio consisting of four components: common Agile Server, Distributed Pair Programmers Editor, Planning Desktop and Osmotic Communicator. The solution is based on the proposed concept of object sharing, which

allows for connecting three different applications to the same server. The obtained results positively verified this concept.

- Organization of fifteen experiments dedicated to evaluation of Agile Studio and obtaining data for AD and DAD comparison. The experiments required preparation of proper experimental environment in order to simulate real working conditions.
- The proposed set of seven metrics, from which three metrics are new and specific for the selected practices and two are improved standard metrics
- The concept of Experiment-Driven Development and its verification during development of Agile Studio components. Application of the concept resulted also in a formulation of requirements regarding the support for Pair Programming, Planning Game and Osmotic Communication practices.
- The proposed conceptualization and model of DAD, which collect all major agile practices and indicate their roles in the whole process. In addition, they overview communication channels for each practice, report on their sensitiveness to team distribution and suggest the possible supporting solutions.

The work resulted also in some interesting findings and conclusions. Among them, the following ones should be emphasized:

- Based on the conducted experiments, it occurred, surprisingly for the author of this dissertation, that the video channel between distant collaborators is of minor importance. According to participants and the author's observation of their work, this channel was rarely used. As it was reported, not only did it failed to provide any valuable information but also distracted from the actual task. Participants also claimed that the opened video window conflicted with others and occupied precious space on the screen. Finally, the transmission of the video data decreased the real-time performance of the evaluated solutions. These findings should be considered in case of the tele-immersion concepts. An example implementation of such concept is, already mentioned, "Office of Real Soon Now" project [5], which is based on projecting large live videos from the remote locations.
- An interesting finding was also abandoning the voice channel requirement in the Osmotic Communicator component. Three reasons for this were emphasized: low intensity of the exchanged data, no search/history capabilities of voice channel, a likeable conflict with other voice channels such as those utilized in Pair Programming.
- The employed concept of Experiment-Driven Development may seem expensive and troublesome as it requires additional resources, such as properly prepared participants and well-suited, divided workspaces. While these caused some problems for the author of this work, for a large company this should not be a case. In addition, the proposed

testing scenarios are short which should not considerably conflict with development schedules.

- It seems that a stronger emphasis should be put on the evaluation of the usefulness of the developed software. Agile Development achieves this indirectly by the practice of Small Releases and early contact of the customer with the working software. However, it is very likely that more formalized and objective evaluation of the developed product can improve the customer understanding of his/her needs, leading to meaningful product enhancements.

Having considered gained experiences and above observations, the future work should include the following issues:

- Revisiting and enhancement of the proposed metrics. It seems that this issue can be a subject of a dedicated study, including experimental verification of metrics accuracy and improvement. This would allow for a thorough comparative study involving proposed solution and all other available applications.
- Verification of the developed Agile Studio by organization of large-scaled experiments, involving considerable amount of participants.
- Extension of Agile Studio with a support for another agile practice, preferably with a use of the Experiment-Driven Development concept. The new Agile Studio component based on the proposed Agile Server would be an additional confirmation of the Agile Server's usability and accuracy of the selected approach.
- Enhancement of the developed Agile Studio, in particular focused on new features, better user-friendliness and, what is most important, the maturity and stability of the whole solution in various environment and settings.
- Extension of the research scope with a real-life verification, possibly including the aspect of different time-zones.

The results of this work encourage the author to suggest a revision of the Manifesto of Agile Software Development, which was presented and discussed in Chapter 2. Stimulated occurs the question whether the four postulates of Manifesto apply also to DAD. The answer seems to be positive having assumed some of the proposed alterations, which are presented below:

MANIFESTO FOR
DISTRIBUTED AGILE SOFTWARE DEVELOPMENT

Agile Manifesto revised.

Individuals and interactions over processes and tools This postulate emphasizes the importance of communication in the agile development. However, DAD strongly depends on provided support. Without proper software tools that maintain the interactions of individuals, this principle is not valid. This changes the role of tools. While co-located agile development places tools in opposite to communication, DAD embeds selected tools in the communication and collaboration process. It is proposed here to modify the postulate in the following manner: *individuals and software-based interactions over processes and distance isolation.*

Working software over comprehensive documentation The documentation, even the comprehensive one, can be used as a supplement of the communication based on the provided support. This seems especially preferable, and actually the only reasonable solution, in the case when team members collaborate in different time zones. The postulate can be replaced by: *Working software and development documentation over document-driven development.*

Responding to change over following a plan The idea behind this postulate is to reduce the risk of the agile process (e.g. inadequate requirements, requirements change). The distributed setting can be considered as another risky factor in the development process (e.g. network or hardware failure, unexpected bandwidths problems etc.). This increases the value of this postulate. To emphasis this change, the postulate is moved up in the postulates list (in agile manifesto it is the last one). In addition, it is proposed to modify it as follows: *Responding to change and unexpected communication problems over following a plan.*

Customer collaboration over contract negotiation It seems, this postulate remains the same in DAD setting. The fact that customer collaboration may be considerably aggravated by the physical distance and imperfect supporting tools does not change the adoptive (not plan-driven) character of the agile process.

Bibliography

[1] Dave Astels. Test Driven development: A Practical Guide. Prentice Hall Professional Technical Reference, 2003.

[2] David Astels, Granville Miller, and Miroslav Novak. The Practical Guide to Extreme Programming. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[3] Kent Beck. Extreme programming explained: embrace change. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] Kent Beck. Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[5] Gary Bishop and Greg Welch. Working in the office of "real soon now". IEEE Comput. Graph. Appl., 20(4):76–78, 2000.

[6] Tony Buzan and Barry Buzan. The Mind Map Book: How to Use Radiant Thinking to Maximize Your Brain's Untapped Potential. Plume, 1996.

[7] Peter Checkland. Systems Thinking, Systems Practice. John Wiley & Sons Ltd, May 1981.

[8] Mark Choate. Professional Wikis. Wrox, 2007.

[9] Alistair Cockburn. Selecting a project's methodology. IEEE Softw., 17(4):64–71, 2000.

[10] Alistair Cockburn. Agile Software Development. Addison Wesley Professional, 2001.

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

[12] Alistair Cockburn and Laurie Williams. Extreme programming examined, chapter The costs and benefits of pair programming, pages 223–243. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[13] DSDM Consortium and Jennifer Stapleton. DSDM: Business Focused Development. Pearson Education, 2003.

[14] Stephen Covey. The Seven Habits of Highly Effective People. Free Press, 1989.

[15] Jacek Dajda and Stanislaw Ciszewski. Supporting extreme programming in distributed setting. AGH Computer Science, Vol.7, pages 49–62, 2005.

[16] Yael Dubinsky, David Talby, Orit Hazzan, and Arie Keren. Agile metrics at the israeli air force. In Proceedings of the Agile 2005 Conference, pages 12–19. IEEE Computer Society, 2005.

[17] Martin Fowler. Refactoring: Improving the design of existing code. In Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002, page 256, London, UK, 2002. Springer-Verlag.

[18] Hillary Goranson. The agile virtual enterprise: cases, metrics, tools. Quorum Books, Westport, CT, USA, 1999.

[19] The Standish Group. Chaos. Technical report, The Standish group international Inc, 1995.

[20] The Standish Group. A recipe for success. Technical report, The Standish group international Inc, 1999.

[21] Deborah Hartmann and Robin Dymond. Appropriate agile measurement: Using metrics and diagnostics to deliver business value. In AGILE '06: Proceedings of the conference on AGILE 2006, pages 126–134, Washington, DC, USA, 2006. IEEE Computer Society.

[22] Jim Highsmith. Adaptive software development: a collaborative approach to managing complex systems. Dorset House Publishing Co., Inc., New York, NY, USA, 2000.

[23] Jim Highsmith. Agile Project Management: Creating Innovative Products. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[24] Kenji Hiranabe. Agile modeling with mind map and uml. `StickyMinds.com`, 2007.

[25] Chih-Wei Ho, Somik Raha, Edward Gehringer, and Laurie Williams. Sangam: a distributed pair programming plug-in for eclipse. In eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, pages 73–77, New York, NY, USA, 2004. ACM Press.

[26] Hanna Hulkko and Pekka Abrahamsson. A multiple case study on the impact of pair programming on product quality. In ICSE '05: Proceedings of the 27th international conference on Software engineering, pages 495–504, New York, NY, USA, 2005. ACM Press.

[27] Watts Humphrey. Managing the Software Process. Addison-Wesley Professional, Boston, MA, USA, 1989.

[28] Ron Jeffries. The king's dinner. XP Magazine, 1999.

[29] Ron Jeffries. Big visible charts. XP Magazine, 2004.

[30] Jukka Kaariainen, Juha Koskela, Pekka Abrahamsson, and Juha Takalo. Improving requirements management in extreme programming with tool support - an improvement attempt that failed. In EUROMICRO '04: Proceedings of the 30th EUROMICRO Conference (EUROMICRO'04), pages 342–351, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Even-Andre Karlsson and Lars-Goran Andersson. Extreme programming examined, chapter XP and large distributed software projects, pages 119–134. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[32] Mark Kobayashi-Hillary. Outsourcing to India: The Offshore Advantage. Springer-Verlag Berlin and Heidelberg GmbH & Co. K, 2004.

[33] Richard Koch. The 80/20 Principle: The Secret of Achieving More With Less. Doubleday, 1999.

[34] Stefan Koch. Agile principles and open source software development: A theoretical and empirical discussion. In Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings, volume 3092 of Lecture Notes in Computer Science. Springer, 2004.

[35] Craig Larman. Agile and Iterative Development: A Manager's Guide. Addison-Wesley Professional, 2003.

[36] Kim Man Lui and Keith Chan. A cognitive model for solo programming and pair programming. In ICCI '04: Proceedings of the Third IEEE International Conference on Cognitive Informatics (ICCI'04), pages 94–102, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Lech Madeyski. Software Engineering: Evolution and Emerging Technologies, Edited by: K. Zielinski and T. Szmuc, volume 130 of Frontiers in Artificial Intelligence and Applications, chapter Preliminary Analysis of the Effects of Pair Programming and Test-Driven Development on the External Code Quality, pages 113–123. IOS Press, 2005.

[38] Frank Maurer and Sebastien Martel. Process support for distributed extreme programming teams. In ICSE 2002 Workshop on Global Software Development, 2002.

[39] Charlie McDowell, Linda Werner, Heather Bullock, and Julian Fernald. The effects of pair-programming on performance in an introductory programming course. SIGCSE Bull., 34(1):38–42, 2002.

[40] Rick Mugridge and Ward Cunningham. Fit for Developing Software: Framework for Integrated Tests. Prentice Hall, 2005.

[41] Matthias Müller. A preliminary study on the impact of a pair design phase on pair programming and solo programming. Information & Software Technology, 48(5):335–344, 2006.

[42] Matthias Müller and Frank Padberg. An empirical study about the feelgood factor in pair programming. In METRICS '04: Proceedings of the Software Metrics, 10th International Symposium on (METRICS'04), pages 151–158, Washington, DC, USA, 2004. IEEE Computer Society.

[43] Jamie Nast. Idea Mapping: How to Access Your Hidden Brain Power, Learn Faster, Remember More, and Achieve Success in Business. John Wiley & Sons Canada, Ltd., 2006.

[44] Kanyamas Navoraphan, Edward Gehringer, James Culp, Karl Gyllstrom, and David Stotts. Next-generation dpp with sangam and facetop. In eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange, pages 6–10, New York, NY, USA, 2006. ACM Press.

[45] Jerzy Nawrocki and Michal Jasinski. Programowanie ekstremalne i prince 2. In Problemy i metody inynierii oprogramowania, pages 525–535, Warszawa-Wrocaw, Poland, 2003. WNT.

[46] Jerzy Nawrocki, Michal Jasinski, Lukasz Olek, and Barbara Lange. Pair programming vs. side-by-side programming. In Proceedings of EuroSPI, pages 28–38, Budapest, November 2005.

[47] Jerzy Nawrocki and Adam Wojciechowski. Experimental evaluation of pair programming. In Proceedings of European Software Control and Metrics (ESCOM 2001), London, UK, apr 2001.

[48] Ikujiro Nonaka and Hirotaka Takeuchi. The Knowledge Creating Company: How Japanese Companies Create The Dynamics of Innovation. Oxford University Press, USA, 1995.

[49] John Nosek. The case for collaborative programming. Commun. ACM, 41(3):105–108, 1998.

[50] International Organization of Standardization. Iso 9000:2000, 05.06.2007.

[51] TickIT organization. The TickIT guide. A Guide to Software Quality Management System Construction and Certification to ISO 9001. DISC TickIT Office, London, UK, 1998.

[52] Frank Padberg and Matthias Müller. On the impact of warmup phases on the economics of pair programming. In Proceedings of the International Workshop on Economics-Driven Software Engineering Research (EDSER), 2003.

[53] Frank Padberg and Matthias M. Müller. Analyzing the cost and benefit of pair programming. In METRICS '03: Proceedings of the 9th International Symposium on Software Metrics, page 166, Washington, DC, USA, 2003. IEEE Computer Society.

[54] Steve Palmer and Mac Felsing. A Practical Guide to Feature-Driven Development. Pearson Education, 2001.

[55] Mary Poppendieck and Tom Poppendieck. Lean Software Development: An Agile Toolkit. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[56] William Poundstone. Prisoner's Dilemma. Doubleday, New York, NY, USA, 1993. Based On Work By-John Von Neumann.

[57] Anatol Rapoport and Albert M. Chammah. Prisoner's Dilemma. University of Michigan Press, December 1965.

[58] Marc Rosenberg. E-Learning: Strategies for Delivering Knowledge in the Digital Age. McGraw-Hill, 2000.

[59] William Royce. Managing the development of large software systems: concepts and techniques. In ICSE '87: Proceedings of the 9th international conference on Software Engineering, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[60] Mac Sathyanarayan. Offshore Development. Globaldev Publishing, 2003.

[61] Ken Schwaber. Enterprise Scrum. Microsoft Press, Redmond, WA, USA, 2007.

[62] Ken Schwaber and Mike Beedle. Agile Software Development with Scrum. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[63] Jennifer Stapleton and David MacDonald. DSDM: The Method in Practice. Addison Wesley, 1997.

[64] Daniel Steinberg and Daniel Palmer. Extreme Software Engineering A Hands-On Approach. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[65] David Stotts, Jason Smith, and Karl Gyllstrom. Support for distributed pair programming in the transparent video facetop. In XP/Agile Universe, pages 92–104, 2004.

[66] David Stotts, Laurie Williams, Nachiappan Nagappan, Prashant Baheti, Dennis Jen, and Anne Jackson. Virtual teaming: Experiments and experiences with distributed pair programming. In Proceedings of the Extreme Programming/Agile Universe Conference 2003, 2003.

[67] Giancarlo Succi and Michele Marchesi, editors. Extreme programming examined. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[68] Don Tapscott and Anthony Williams. Wikinomics: How Mass Collaboration Changes Everything. Portfolio Hardcover, 2008.

[69] Jari Vanhanen and Casper Lassenius. Effects of pair programming at the development team level: an experiment. In Proceedings of the International Symposium on Empirical Software Engineering, 2005.

[70] Gerald Weinberg. An introduction to general systems thinking (silver anniversary ed.). Dorset House Publishing Co., Inc., New York, NY, USA, 2001.

[71] Laurie Williams and Robert Kessler. Pair Programming Illuminated. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[72] Laurie Williams, Robert Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. IEEE Softw., 17(4):19–25, 2000.

[73] Shaochun Xu and Xuhui Chen. Pair programming in software evolution. In Proceedings of the Canadian Conference on Electrical and Computer Engineering CCECE 2005, pages 1846–1849, Washington, DC, USA, 2005. IEEE Computer Society.

[74] Shaochun Xu and Vaclav Rajlich. Empirical validation of test-driven pair programming in game development. In Proceedings of 5th IEEE/ACIS International Conference on Computer and Information Science and the 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse. ICIS-COMSAR 2006, pages 500–505, Washington, DC, USA, 2006. IEEE Computer Society.

[75] Edward Yourdon. Death March: The Complete Software Developer's Guide to Surviving Mission Impossible Projects. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[76] Abdullah Mohd Zin, Sufian Idris, and Nantha Kumar Subramaniam. Improving learning of programming through e-learning by using asynchronous virtual pair programming. Turkish Online Journal of Distance Education-TOJDE, 7(13):162–173, 2006.

Online references

[77] Agile alliance home page. http://www.agilealliance.org.

[78] Beetlejuice home page. http://www.pols.co.uk/beetlejuice.

[79] Continuous integration tools, wikipedia. http://en.wikipedia.org/wiki/Continuous_integration.

[80] Cruise control home page. http://cruisecontrol.sourceforge.net.

[81] Definition of word *methodology* in an online english dictionary Dictionary.com. http://dictionary.reference.com/browse/methodology.

[82] Definition of word *methodology* in an online free encyclopedia wikipedia.org. http://en.wikipedia.org/wiki/Methodology.

[83] Definition of word *synergy* in an online free encyclopedia wikipedia.org. http://en.wikipedia.org/wiki/Synergy.

[84] Dokuwiki home page. http://wiki.splitbrain.org/wiki:dokuwiki.

[85] Eclipse ide home page. http://www.eclipse.org.

[86] Fdd implementations - nebulon home page. http://www.nebulon.com/articles/fdd/fddimplementations.html.

[87] Fitnesse framework home page. http://fitnesse.org.

[88] Gadu-gadu communicator home page. http://www.gadu-gadu.pl.

[89] Google docs and spreadsheets. http://docs.google.com.

[90] Googletalk communicator home page. http://www.google.com/talk.

[91] Java web start technology. http://java.sun.com/products/javawebstart.

[92] Knotes in kde home page. http://pim.kde.org/components/knotes.php.

[93] Luntbuild home page. http://luntbuild.javaforge.com.

[94] Manifesto for agile software development. `http://www.agilemanifesto.org`.

[95] Microsoft netmeeting home page. `http://www.microsoft.com/windows/NetMeeting`.

[96] Pin 'em up - your personal reminder home page. `http://pinemup.sourceforge.net`.

[97] The problem of suboptimization. `http://pespmc1.vub.ac.be/SUBOPTIM.html`.

[98] Realvnc remote desktop control home page. `http://www.realvnc.com`.

[99] Refactoring home page. `http://refactoring.com`.

[100] Skype communicator home page. `http://www.skype.com`.

[101] Sticky notes home page. `http://www.sticky-notes.net`.

[102] Subversion home page. `http://subversion.tigris.org`.

[103] Teamspeak voip solution home page. `http://www.goteamspeak.com/`.

[104] Tightvnc remote desktop control home page. `http://www.tightvnc.com`.

[105] Ultravnc remote desktop control home page. `http://ultravnc.sourceforge.net`.

[106] Xplanner home page. `http://www.xplanner.org`.

[107] Xpweb home page. `http://xpweb.sourceforge.net`.

[108] Scott Ambler. The agile unified process (aup) home page. `http://www.ambysoft.com/unifiedprocess/agileUP.html`.

[109] Jext Team. Jext editor home page. `http://www.jext.org`.

Appendices

Appendix A

Overview of experiments organization

This appendix provides selected details on the utilized resources and organization issues of the conducted experiments. Figures A.1 and A.2 present the photos taken during one of the experiments dedicated to DPPE.

A.1 Resources

The utilized resources can be divided into: participants, laboratories, hardware and software.

The participants were the volunteered students of the fourth and fifth year of Computer Science faculty. Thus, they possessed a knowledge of the object oriented analysis, software methodologies and skills in Java programming language. The students of fifth year were already familiar with agile development, eXtreme Programming methodology and its core practices, as a result of dedicated course. For the students of fourth year, an introduction to the required practices was made before the experiment start.

Having considered all conducted experiments, the total of five students laboratories were used, however not in once. Two of the laboratories are located next to each other with a door in the wall between them. Another two are also located next to each other, however to switch between them, one need to pass the corridor. The last one is located in a larger distance from the mentioned four. The capacity of laboratories is approximately from 10 to 20 work stations.

The hardware employed during experiments can be divided into computers, headphones and web cams. The majority of computer are equipped with Pentium IV 1.7GHz-1.8GHz processors and 512MB of RAM. Several ones are: AMD Duron 1GHz 256 RAM and Pentium III 700MHz 256 RAM. Most of experiments were based on the Pentium IV machines. The utilized headphones were standard headphones with a microphone: I-Box HP-9600MV and Logitech Dialog 812. As for webcams, 320x240 cameras were used: Creative WebCam Instant and I-Cam 350. Additionally, four of the utilized laboratories are monitored using standard cctv cameras. Finally, experiments required a server to host Agile Server and code repository.

For this purpose Pentium II 800Mhz with Gentoo linux installed was used.

Beside developed Agile Studio and other evaluated tools (such as Skype[100, 95]), the following software was used: Subversion[102] for hosting the experiment tasks, JRE 1.5, Eclipse.

Figure A.1: Co-located pair programmers during experiment

A.2 Plan of a single experiment

Before experiment was started, the following preparations needed to be done. Participating students were asked to:

- verify that the accounts to the computers in students laboratories are still active,
- create or verify the active account on the Skype[100] or other utilized communicator
- create or verify the account on other communicator, which would be used as a backup solution, in the case of unexpected problems with the primary ones,
- provide a list of active email addresses, which were used to send important experiment information.

Figure A.2: Distributed pair programmer during experiment

Most experiments were organized in two adjacent laboratories, which considerably facilitated their control and monitoring. The participants were monitored using the cctv cameras and direct observation of the experiment's supervisor. Before experiment was started, a short revision of the required knowledge (mainly regarding the evaluated practice) was performed. Afterwards, the evaluated tools were presented and tested. The utilized hardware (mainly headphones) needed to be tested as well. Having successfully passed these tests (in some cases it took longer than expected to prepare working environment), participants were divided into co-located and distributed teams, and introduced with the task to obtain the currently tested metric. Afterwards, the distributed team members were delegated to different laboratories, so that they were forced to communicate via the provided software. Whereas it was possible, participants were located in a distance of several meters to each other, so that they would not interfere the work of other team.

Before starting the current task, distributed teams were provided with approximately 10 minutes for familiarizing with the utilized tools. During work, they were allowed to take breaks, for instance to answer the phone call. This allowed for better simulation of the real working environment. Depending on the current metric, the work was stopped after a time limit or after completing the task. Participants were asked to send the produced artifacts

(e.g. User Stories) via email. The example artifact with produced User Stories is presented in Figure A.3 (the stories are written in Polish).

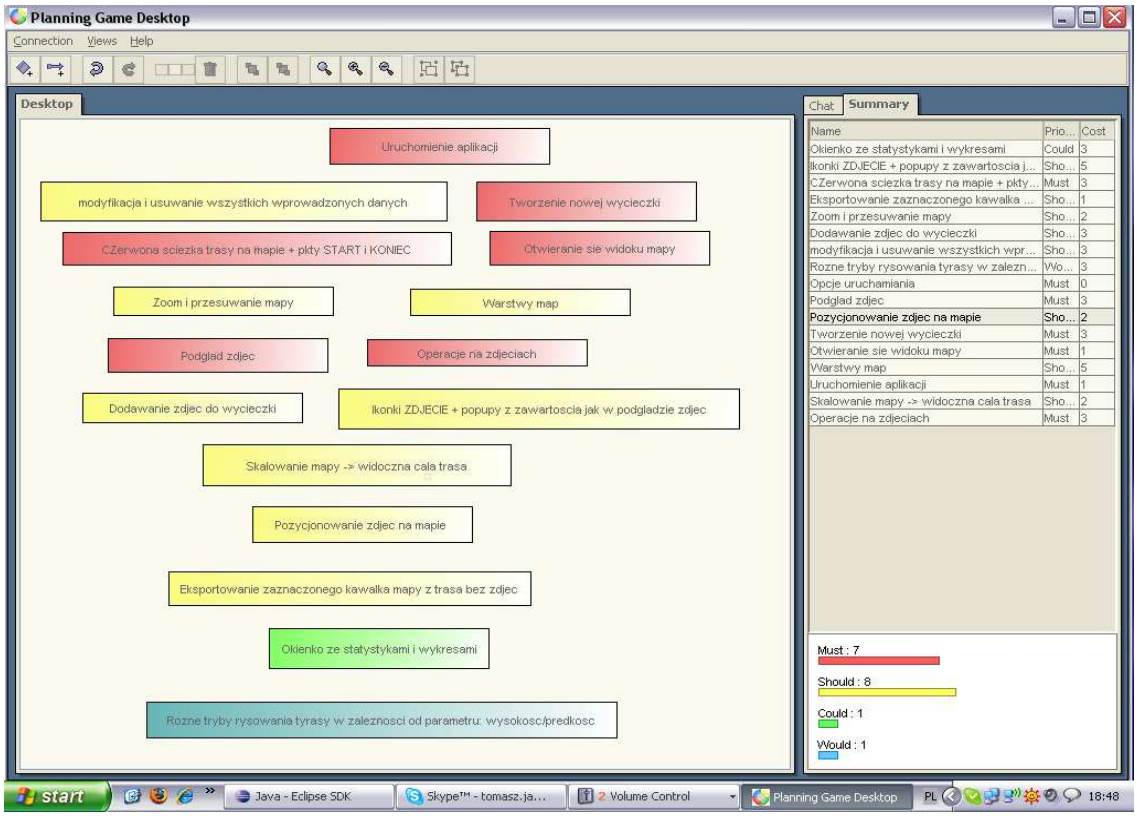


Figure A.3: Produced stories during Planning Desktop evaluation

Afterwards, next metric was randomly selected. After a 10-minute break, the work started again. To provide more reliable results, the teams were mixed before every new task. As result, participants had a chance to work with different colleagues and experience both settings.

The experiments manifested several possible problems such as strict firewall policies in university net, preventing participants from connecting to each other, and the quality of the transmitted voice. The last one can be solved using local servers offering voice communicator feature.

Appendix B

Example task for Planning Game evaluation

This appendix presents the example task for the evaluation of Planning Game practice. This task was utilized along with *Time*, *Items* and *Learning* metrics. As the last one requires a questionnaire related to the task, it is included here as well. The task is constructed in a way that it reminds a typical customer's problem description, which is usually informal and a little bit unorganized.

The task and questionnaire were originally written in Polish.

B.1 Task

GPS TREK

The goal of the GPS Trek application is to manage the history of the bicycle and mountain trips.

Two kinds of data can be associated with a single trip: photos and GPS log. There is a large number of GPS logging applications for mobile devices. An example is OziExplorer, which generates the GPS path in PLT (Track Plot Log File) format. Another example is the AutoMapa application, which can be utilized for GPS logging as well. In this case, the logs are written in files with extension *.gps.

Here is the possible scenario for the GPS Trek application. After a trip, the user downloads the photos from the camera and the gps path from the gps receiver. Next, he/she launches the GPS Trek application. Inside it, he/she creates new trip and points to the folder with photos as well as to the gps log file (in OziExplorer format for a start, later also for AutoMapa format). It is possible to decide whether the photos should be copied to application folder or linked from the current location. The user gives the name of the trip, which is mandatory, Optionally, the trip date and description can be given as well. The created trip can be empty, without any photos and log files. If a user selects a folder without photos, GPS TREK is able

to refresh the folder and add the new photos, whenever they appear in the folder. There is also a possibility to add multiple folders with photos (in this way several cameras, like friends cameras, can be handled)

After the creation of the new trip and data import, the user can choose from two possible views:

- photo gallery
- track view

The photo gallery is a scrollable panel with photos miniatures. When a miniature is mouse-clicked, the full-sized photo is displayed. If the user click the right-mouse button on the photo, a context-menu is shown. In this menu, the user can delete the photo, change its description and specify its quality (three suggested degrees: good, average, poor). The user can also select the photo to be shown on the map. A full-sized photo is rendered with its description and the exact time of the shot (which is obtained from the EXIF header = Exchangeable Image File Format). The description is also shown under every miniature in the photo gallery view.

The second view shows the map of the track. The map should be built from, at least, following layers: roads (squares included), rivers and lakes, woods, parks, cities and significant points (e.g. mountain peak, famous monument). The track, which is stored in the gps log file, should be visualized on the map as a red line. When the map view is opened, the map should be automatically rescaled so that whole track is visible. The start and finish of the track are properly indicated. Beside them, on the red line indicating the track, the user can see photo icons, which represent the selected photos to be shown on map (as mentioned above, this is done through the context menu). When the user clicks on the icon, a popup window appears with corresponding photo and its information (alike in the photo gallery). The map can be zoomed and moved. The zoom does not need to be smooth, but must be provide at least 5 levels of details. Additionally, there has to be option for exporting the visible map fragment with the displayed track to standard image file.

An interesting feature would be also a window showing the treks statistics and the plot of absolute hight. The statistics can refer to the trek's distance, total duration, movement duration (including percentage of the total duration), the number of taken photos as well as average, maximum and minimum speed.

Extra feature would be also option of rendering the track line based on a selected parameter, such as speed or absolute hight. When speed is selected, the track line would be painted with gradients of various tones of green and red depending on the current speed in the given point. The maximum speed for the coloring purpose can be set in the settings window. Alike, selecting the absolute height as the rendering parameter would result in gradient line from red tones (high hight) to green tones (low hight).

All data can be modified (including the name of the trek) and deleted (except the name of the trek).

B.2 Questionnaire

Below, the questionnaire for presented task is shown. It is divided into three parts: basic participant info, feedback part asking for participant opinion and observation (the length of Planning Game was measured objectively, it was interesting to obtain the participant time impression through), questions for Learning metric purpose.

BASIC INFO

- Your name:
- Partner's name:
- Your role (customer or developer):

FEEDBACK PART

1. How many User Stories have you created?
2. Do you completely understand the posed problem? Is there something unclear?
3. Was something causing you problems during Planning Game
4. Do you think the created User Stories are meaningful in the sense of the posed problem?
5. How long has your Planning Game taken?
6. How long has the customer been reading the task?

QUESTIONS

1. What does the PLT abbreviation mean and when it is used?
2. What does the EXIF abbreviation mean and when it is used?
3. What is the purpose of the photo miniatures on the map?
4. What types of statistics are meant to be provided according to the presented concept?
5. What is the extension **.gps*
6. Is the zooming meant to be smooth? If not, how many levels of details must be provided?
7. What are the paramaters that can be used to render the track line?
8. How to synchronize the gps log with photos?

Appendix C

Evolution of Agile Studio requirements

This appendix illustrates how the requirements for Agile Studio components evolved during conducted development. Requirements and experiments are represented by the symbols introduced in chapters 4 and 5. The presented figures C.1 and C.2 show when the requirements were introduced in terms of the conducted experiments. For some requirements, their modification or dismissal is indicated as well. To differentiate these states, the following coloring scheme is utilized:

Green color - requirement was introduced

Yellow color - requirement was modified

Red color - requirement was dropped

White color - requirement was not changed

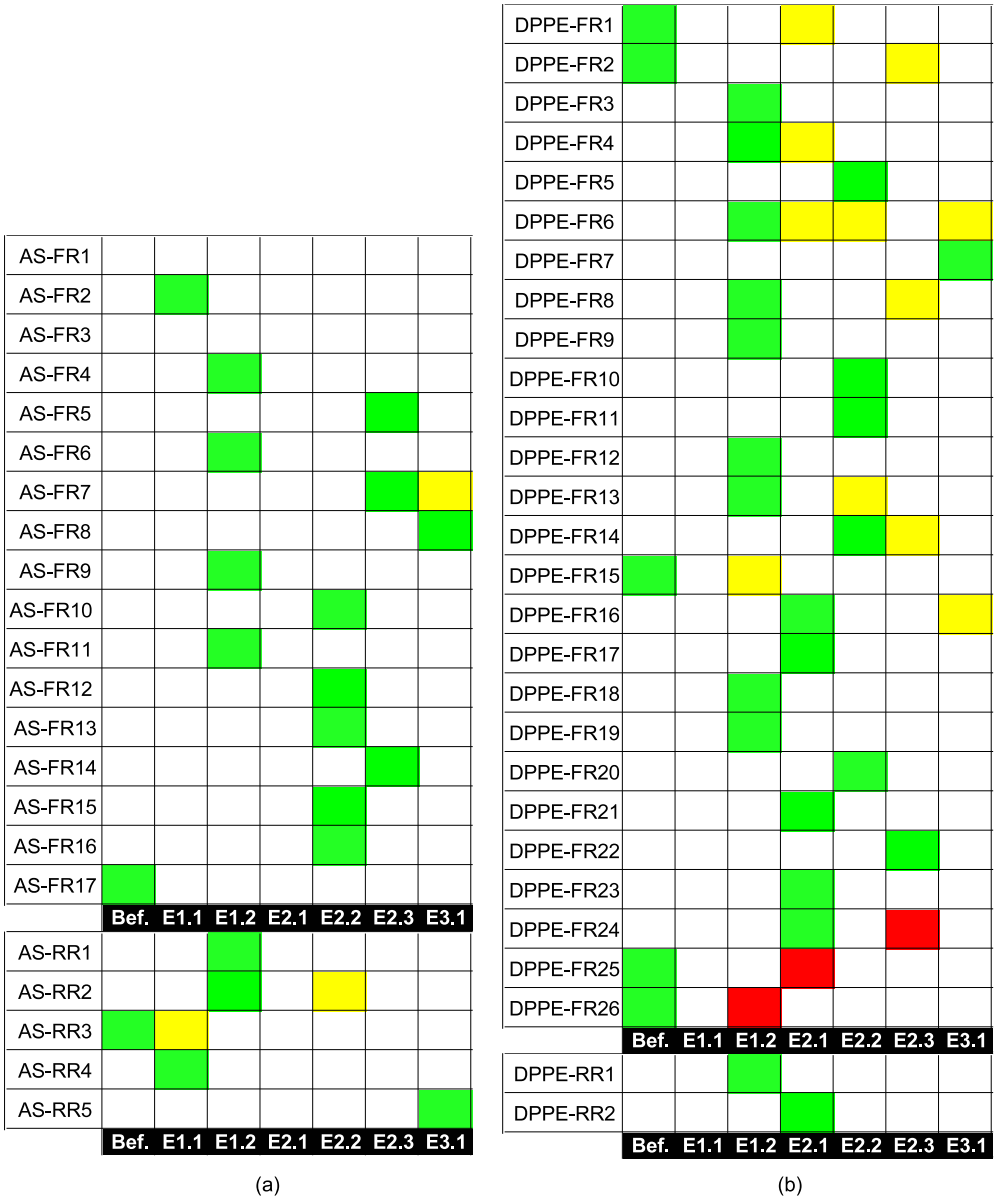


Figure C.1: Evolution of requirements for Agile Server (a) and Distributed Pair Programmers Editor (b)

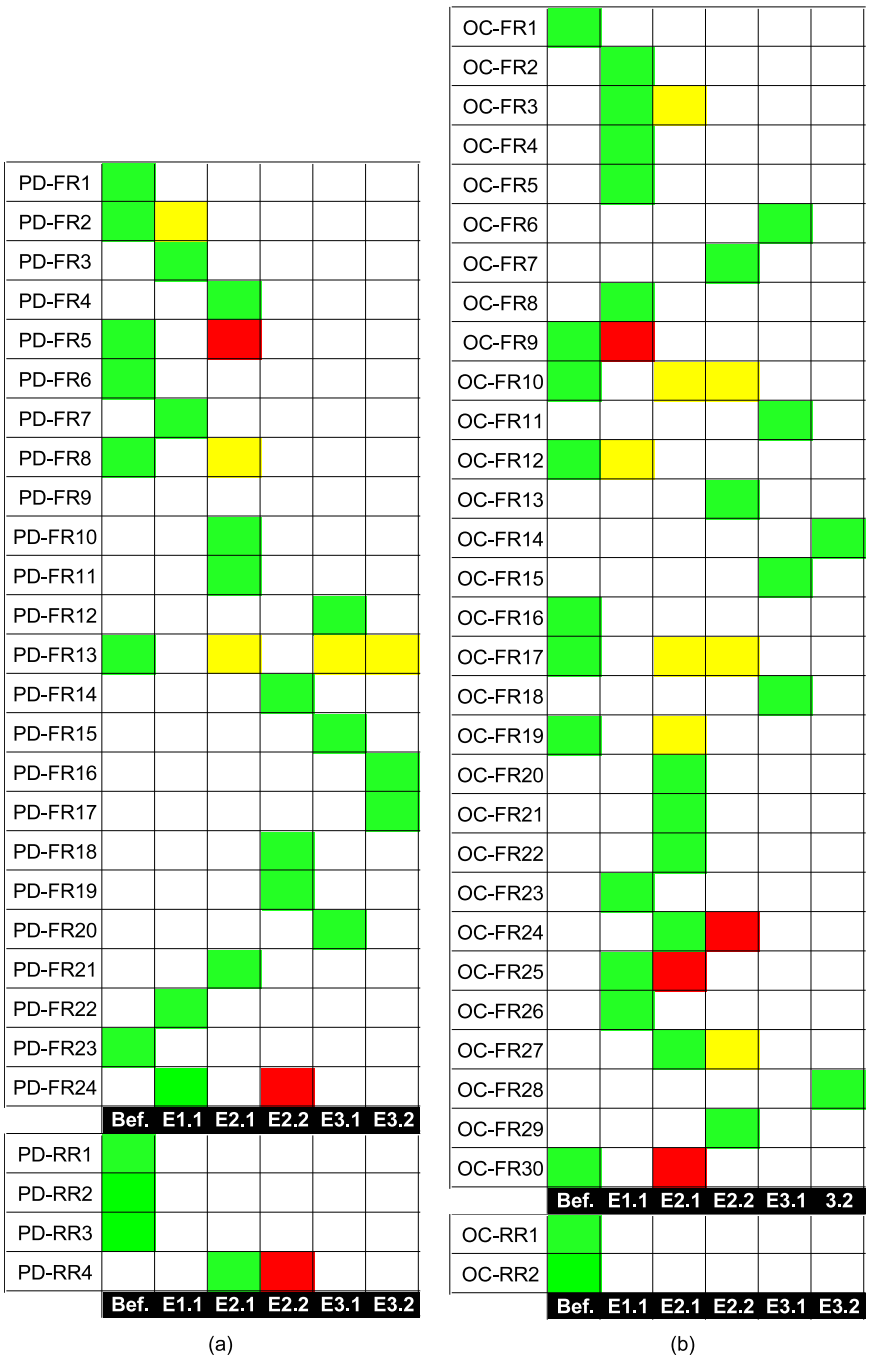


Figure C.2: Evolution of requirements for Planning Desktop (a) and Osmotic Communicator (b)

Appendix D

Detailed experiments results

This appendix presents the details experiments results for the proposed metrics. The results are organized into tables. Every table shows results obtained for each team or person, depending on the current metric. The utilized symbols were introduced in Chapter 5 and preserve their meaning.

D.1 Pair Programming evaluation

Table D.1 presents the Pair Programming results obtained for the Time metric.

Table D.1: Pair Programming experiments results for metric Time

Teams	Development time [min]							
	E2.1		**E2.2**		**E2.3**		**E3.1**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	58	65	50	68	51	51	56	50
2	50	76	57	81	58	70	46	60
3	56	81	46	61	52	60	57	68
4	49	70	-	-	49	50	-	55
5	-	-	-	-	-	64	-	-

Table D.2 presents the Pair Programming results obtained for the Test/Errors metric. The number of failing tests, which needed to be fixed was 10.

Table D.2: Pair Programming experiments results for metric TestsErrors

Teams	Passed tests							
	E2.1		**E2.2**		**E2.3**		**E3.1**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	10	8	10	9	10	10	10	9
2	9	6	10	5	8	7	10	9
3	10	7	10	7	9	10	8	8
4	9	6	-	-	10	9	-	10
5	-	-	-	-	-	6	-	-

Table D.3 presents the Pair Programming results obtained for the Items metric.

Table D.3: Pair Programming experiments results for metric Items

Teams	Created items / creation time[min]							
	E2.1		**E2.2**		**E2.3**		**E3.1**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	10/72	10/76	10/61	10/86	10/72	8/81	10/81	10/67
2	10/79	9/100	10/68	8/92	10/63	10/71	10/63	10/94
3	10/85	9/85	10/81	10/85	9/80	9/85	10/69	10/90
4	10/65	8/90	-	-	10/66	10/76	-	9/77
5	-	-	-	-	-	10/68	-	-

Table D.4 presents the Pair Programming results obtained for the Feelgood metric. To remind, the maximum point per each aspect is 5.

Table D.4: Pair Programming experiments results for metric Feelgood

Persons	Efficiency / Satisfaction / Easiness							
	E2.1		**E2.2**		**E2.3**		**E3.1**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	4/4/3	4/5/3	5/5/4	2/3/3	4/5/2	3/3/3	4/5/3	4/4/4
2	3/4/4	2/3/1	4/5/4	3/4/2	5/5/4	5/5/2	4/4/3	3/3/2
3	4/5/2	3/4/4	5/5/3	4/5/3	4/4/3	4/4/4	5/5/4	4/5/3

4	5/5/4	2/4/2	4/4/3	2/2/3	5/5/4	4/5/2	5/5/2	4/4/2
5	4/4/4	4/5/4	4/4/4	4/4/2	4/5/3	4/4/3	5/4/4	4/4/4
6	4/5/4	2/2/1	5/5/3	4/5/4	5/5/5	5/4/3	5/5/2	3/4/2
7	5/5/3	4/5/3	-	-	4/4/3	4/4/3	-	4/4/3
8	4/5/3	2/4/2	-	-	4/5/4	4/3/2	-	5/5/3
9	-	-	-	-	-	4/4/3	-	-
10	-	-	-	-	-	4/5/5	-	-

Table D.5 presents the Pair Programming results obtained for the E-Factor metric. The total working time was 2 hours.

Table D.5: Pair Programming experiments results for metric E-Factor

Teams	Uninterrupted time [min]							
	E2.1		**E2.2**		**E2.3**		**E3.1**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	115	120	113	117	120	115	113	103
2	120	110	117	120	111	105	118	120
3	110	120	103	109	110	110	106	117
4	104	113	-	-	118	120	-	119
5	-	-	-	-	-	120	-	-

Table D.6 presents the Pair Programming results obtained for the Negotiation metric. The quantity of all tasks was 10. The working time was unlimited.

Table D.6: Pair Programming experiments results for metric Negotiation

Teams	Completed tasks							
	E2.1		**E2.2**		**E2.3**		**E3.1**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	9	10	10	8	10	9	9	10
2	10	8	9	9	10	8	10	9
3	10	8	9	10	9	10	10	9
4	10	9	-	-	10	8	-	9
5	-	-	-	-	-	10	-	-

Table D.7 presents the Pair Programming results obtained for the Learning metric. The questionnaire consisted of 8 questions.

Table D.7: Pair Programming experiments results for metric Learning

Teams	Developer good answers / Customer good answers							
	E2.1		**E2.2**		**E2.3**		**E3.1**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	7/8	6/8	8/8	6/7	6/8	5/6	7/8	8/8
2	6/7	5/7	6/6	7/7	7/7	7/7	6/6	6/7
3	6/6	5/6	5/7	7/8	8/8	6/8	8/8	7/8
4	7/8	5/6	-	-	7/6	7/6	-	6/6
5	-	-	-	-	-	6/7	-	-

D.2 Planning Game evaluation

Table D.8 presents the Planning Game results obtained for the Time metric.

Table D.8: Planning Game experiments results for metric Time

Teams	Development time [min]							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	49	79	45	66	50	60	46	50
2	55	93	56	72	54	59	56	60
3	50	68	52	58	48	70	57	68
4	-	78	-	-	58	54	-	55

Table D.9 presents the Planning Game results obtained for the Items metric.

Table D.9: Planning Game experiments results for metric Items

Teams	Created items / creation time[min]							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	15/73	14/90	16/66	16/79	18/67	14/76	18/72	17/79
2	18/69	13/85	20/74	14/80	19/69	16/79	17/69	16/78
3	17/66	14/87	15/83	17/83	15/63	17/81	17/74	16/75
4	-	11/103	-	-	16/80	16/75	-	13/69

Table D.10 presents the Planning Game results obtained for the Feelgood metric. The maximum points that can be given for each of the subjective assessments is 5.

Table D.10: Planning Game experiments results for metric Feelgood

Persons	Efficiency / Satisfaction / Easiness							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	4/5/2	1/2/4	4/4/4	3/5/2	4/2/3	4/5/3	5/4/5	4/4/3
2	4/4/3	3/3/3	4/5/3	4/2/4	5/4/4	4/3/4	5/3/3	5/3/3
3	5/4/4	2/4/1	5/4/3	4/3/3	5/5/4	5/3/3	5/5/4	3/4/4
4	5/5/3	3/3/5	4/3/4	4/5/2	4/5/4	4/5/4	4/5/4	4/4/5
5	4/5/4	2/3/4	4/5/4	3/4/5	4/3/3	4/4/3	5/4/1	5/5/3
6	5/1/4	4/2/2	5/5/3	4/4/4	5/4/3	3/4/4	4/5/3	5/4/1
7	-	2/1/3	-	-	5/5/2	5/4/4	-	3/4/4
8	-	3/2/5	-	-	5/5/5	4/4/3	-	4/5/4

Table D.11 presents the Planning Game results obtained for the E-Factor metric. The total (maximum) working time was 120 minutes.

Table D.11: Planning Game experiments results for metric E-Factor

Teams	Uninterrupted time [min]							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	109	115	118	119	114	120	119	116
2	109	119	112	115	118	117	104	114
3	120	120	116	118	120	115	113	120
4	-	118	-	-	108	102	-	120

Table D.12 presents the Planning Game results obtained for the Learning metric. The number of questions in the evaluating questionnaire was 8.

Table D.12: Planning Game experiments results for metric Learning

Teams	Developer good answers / Customer good answers							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	8/7	7/8	8/8	7/7	7/8	7/8	7/8	6/7
2	7/8	6/8	7/7	7/8	7/8	5/6	8/8	6/7
3	8/8	7/8	6/8	6/8	6/6	7/7	6/6	8/7
4	7/8	7/7	-	-	8/8	7/8	-	6/8

D.3 Osmotic Communication evaluation

Table D.13 presents the Osmotic Communication results obtained for the Time metric.

Table D.13: Osmotic Communication experiments results for metric Time

Persons	Searching time [s]							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	498	572	550	513	491	584	537	550
2	540	630	512	538	564	542	496	602
3	470	589	493	487	522	447	526	468

4	553	485	570	499	440	561	485	465
5	592	544	445	511	528	473	520	543
6	446	658	566	560	548	512	505	472
7	-	463	-	459	-	546	-	452
8	-	515	-	575	-	432	-	569
9	-	475	-	415	-	578	-	541
10	-	513	-	-	-	418	-	-
11	-	625	-	-	-	511	-	-
12	-	539	-	-	-	500	-	-

Table D.14 presents the Osmotic Communication results obtained for the Feelgood metric. The maximum number of points for each of the three assessments is 5.

Table D.14: Osmotic Communication experiments results for metric Feelgood

Persons	Efficiency / Satisfaction / Easiness							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	4/4/5	4/4/5	5/4/5	5/5/5	4/4/5	4/4/4	4/5/4	5/5/4
2	5/4/4	4/4/5	4/5/5	5/5/5	5/4/5	5/5/5	4/5/5	5/5/5
3	5/4/5	5/5/4	3/4/5	4/5/5	4/5/5	5/4/5	5/5/5	5/4/5
4	5/5/5	4/4/5	4/5/5	5/5/5	4/5/5	5/5/5	5/4/5	5/5/5
5	4/4/5	4/4/5	5/4/4	5/5/5	5/4/5	4/5/5	5/5/5	5/5/4
6	4/5/4	4/4/5	4/4/4	4/5/5	5/4/5	5/4/5	4/4/5	4/4/5
7	-	5/5/5	-	5/4/5	-	5/5/4	-	5/5/5
8	-	4/4/4	-	5/5/5	-	5/4/5	-	5/5/5
9	-	3/5/5	-	5/5/4	-	5/5/5	-	5/5/5
10	-	4/4/5	-	-	-	5/4/5	-	-
11	-	4/4/5	-	-	-	5/4/5	-	-
12	-	4/4/5	-	-	-	5/4/4	-	-

Table D.15 presents the Osmotic Communication results obtained for the Negotiation metric. The number of tasks was 10 and the working not limited.

Table D.15: Osmotic Communication experiments results for metric Negotiation

Teams	Completed tasks							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	10	10	9	10	10	10	9	9
2	9	10	10	8	10	10	10	10
3	-	9	-	9	-	8	-	9
4	-	8	-	-	-	9	-	-

Table D.16 presents the Osmotic Communication results obtained for the Learning metric. The number of questions in the questionnaire was 8.

Table D.16: Osmotic Communication experiments results for metric Learning

Persons	Developer good answers / Customer good answers							
	E2.1		**E2.2**		**E3.1**		**E3.2**	
	C	**D**	**C**	**D**	**C**	**D**	**C**	**D**
1	7/7	6/8	7/7	7/8	8/8	7/6	6/6	8/7
2	7/8	8/8	6/7	8/7	7/8	6/8	7/8	7/8
3	6/7	6/7	6/7	7/7	8/8	7/7	6/6	7/8
4	6/8	7/7	7/7	6/8	5/8	8/8	7/8	7/7
5	-	8/8	-	6/7	-	6/6	-	8/8
6	-	6/8	-	7/7	-	7/8	-	7/8
7	-	7/7	-	-	-	8/7	-	-
8	-	7/7	-	-	-	7/8	-	-

List of Symbols and Abbreviations

Abbreviation	Description
AD	Agile Development
AS	Agile Server, server side of Agile Studio
CMM	Capability Maturity Model
DAD	Distributed Agile Development
DPP	Distributed Pair Programming
DPPE	Distributed Pair Programmers Editor, component of Agile Studio
DXP	Distributed eXtreme Programming
EDD	Experiment-Driven Development
OC	Osmotic Communicator, component of Agile Studio
PD	Planning Desktop, component of Agile Studio
PP	Pair Programming
LOOP	Late, Over budget, Overtime, Poor quality
XP	eXtreme Programming

List of Figures

List of Tables