

Date of acceptance

Grade

Instructor

Agility in software development

Maarit Kivioja

Helsinki 20.March.2008

UNIVERSITY OF HELSINKI

Department of Computer Science

HELSINGIN YLIOPISTO - HELSINGFORS UNIVERSITET - UNIVERSITY OF HELSINKI

Tiedekunta - Fakultet - Faculty Matemaattis-luonnontieteellinen tiedekunta		Laitos - Institution – Department Tietojenkäsittelytieteen laitos	
Tekijä - Författare – Author Maarit Kivioja			
Työn nimi - Arbetets titel - Title Agility in software development			
Oppiaine - Läroämne – Subject Computer Science			
Työn laji - Arbetets art – Level Seminaari	Aika - Datum - Month and year 20.03.2008	Sivumäärä - Sidoantal - Number of pages 12	
<p>Tiivistelmä - Referat – Abstract</p> <p>Some project leaders believe software development is a statistically controllable process and that with the right process, they can become immune to the turnover of key people. Others do not believe this. The heart of good software development will always reside in the individual people on the project. However, this document goes thru Cockburn's principles for selecting agile process model. It is trying to answer to question like "what would be the optimal formality in the software process?" and "when would documentation replace communication?"</p> <p>Chapter 4 describes an ideal setting for software development.</p> <p>Finally, the chapter 5 describes how a small data-processing department adapted Extreme Programming principles in 2001.</p>			
Avainsanat – Nyckelord - Keywords Agile			
Säilytyspaikka - Förvaringsställe - Where deposited Department of computer science			
Muita tietoja – Övriga uppgifter – Additional information			

1. Introduction	1
2. Money talks	2
3. Cockburn's principles for selecting a process model.....	3
3.1 Formality	3
3.2 Documentation and communication	4
3.3 Discipline.....	5
3.4 Concurrency	5
3.5 Efficiency	6
4. Ideal software development.....	6
4.1 Dedicated Developers.....	6
4.2 Experienced Developers	6
4.3 Small Collocated Team	7
4.4 Automated Regression Tests	7
4.5 Easy Access to Users.....	7
4.6 Short Increments and Frequent Delivery to Real Users	7
5. Examples of using agile ideas in software development	8
5.1 Characterizes of Extreme Programming.....	8
5.2 Using XP in Elisa Internet Ltd	9
6. Summary	12
7. Sources.....	13

1. Introduction

Craig Larman and Victor R. Basili describe history of iterative software development in their article "Iterative and Incremental Development: A Brief History". They claim that iterative, evolutionary, and incremental software development can be viewed as the “modern” replacement of the waterfall model, but its practiced and published roots go back decades.

Larman and Basili start the history from early ages of computer software development. They describe in the article that at least in NASA and IBM tried iterative and incremental development techniques (IID’s) even before year 1970.

Also some scientific articles about iterative techniques were published as early as in mid-seventies but the Barry Boehm’s “A Spiral Model of Software Development and Enhancement” was one of the landmarks of the iterative development in the mid eighties. Although the idea of iterative cycles was not new, the spiral model proposed by Boehm, formalized and made prominent the risk-driven-iterations concept. Boehm also emphasized the need for discrete steps for risk assessment during iterations [LaB03].

Then, in 1986, David Parnas and Paul Clements published “A Rational Design Process: How and Why to Fake It.”³⁰ In it, they listed many reasons to choose iterative development style. Some of the reasons to choose iterative development cycle are still popular nowadays. For example, Parnas and Clements wrote that even if we could define all requirements, there are many details that we can only discover once we are well into implementation. Parnas and Clements stated also, that even if the requirement designer could master all the details and complexity that is needed, external forces lead to changes in requirements, some of which may invalidate earlier decisions [LaB03].

The information society was born in the nineties. It was time to tens of methods known today, including SCRUM, rapid application development (RAD), Dynamic Systems Development Method (DSDM), Extreme Programming (XP) and Feature-Driven Development (FDD). Also the lean product development strategy, originally developed at Toyota, became more common for software developers by that time. However, articles about lean software development are dated mostly after year 2000.

In February 2001, a group of 17 process experts —representing DSDM, XP, Scrum, FDD, and others— interested in promoting modern, iterative and incremental development methods and principles, met in Utah to discuss common ground. From this meeting came the Agile Alliance and the now popular catch phrase “agile methods,” all of which apply iterative and incremental development [Bec01].

Finally in 2002, Alistair Cockburn, one of the participants, published the first book under the new appellation, "Agile Software Development" [LaB03].

2. Money talks

Alistair Cockburn describes the core of software management in terms of where one should invest money [Coc02]. (Money can be seen as usage of all the resources in addition of funding, but as Cockburn describes the real world, we can see e.g. time as money.)

Money is used either for investigating and planning right now (Money-for-information, MFI) or for getting ready for changes in plans later, enabling those changes with flexible software (Money-for-flexibility, MFF). On Money-for-information (MFI) issues the team can spend money now to obtain information that puts them in a better situation for later in the project. Work-plan breakdown structures, system performance under load, and user reaction to system design are typical MFI issues.

Money-for-flexibility (MFF) issues are those on which the team cannot possibly obtain information now to put them in a better situation later. The better strategy is to spend money on making the change easier later. According to Cockburn, movements in the stock market, emerging standards, and staff continuity are MFF issues.

Cockburn describes, that many of the differences between agile methodologies and cost- and plan-driven approaches are in deciding which issues of software development are MFF and which MFI issues and what are the best allocation of resources for them. Old, waterfall approach trusting plan-driven methodologies would spend money for planning in the beginning of the development. Instead, agile development strategies would not spend money now but prepare the software for change. Both might agree that the question of system performance under load is an important MFI issue, and both might agree to spend money early to build a simple system simulator and load generator to stress test the design.

The selection of suitable process model for software development is also issue of certainty and possibility to answer the questions. Cockburn divides the issues in three categories, here listed in rising uncertainty.

Predictable issues can be investigated using breakdown techniques. Such an issue might be creating a schedule for work similar to that successfully performed in the past.

Unpredictable but resolvable issues can be investigated through study techniques such as prototypes and simulators. Such issues include system performance limits. These are also MFI (money-for-information) propositions. Agile and plan-driven teams are likely to use similar strategies for these issues as part of basic project risk management.

Irresolvable issues tend to be sociological, such as which upcoming standard will gain market acceptance, or how long key employees will stay around. These issues cannot be resolved in advance, and so are not MFI propositions, but are MFF propositions.

Agile and plan-driven teams are intrinsically likely to use different strategies for these issues. Agile teams will set up to absorb these changes, while plan-driven project teams must, by definition, create plans for them.

3. Cockburn's principles for selecting a process model

Alistair Cockburn describes ten principles that one should take into account when selecting a process model in his two-part article “Learning From Agile Software Development” [Coc02]. I merged some of them here to go briefly thru the differences and similarities between traditional and agile software development process models.

3.1 Formality

Cockburn has following principles about methodology and validation:

Principle 1: Different projects need different methodology trade-offs.

Principle 2: A little methodology does a lot of good; after that, weight is costly.

Principle 4: Projects dealing with greater potential damage need more validation elements.

The sensible amount of methodology depends on risks. The more there is quality risk, like money or human lives involved, the better is the traditional, plan-driven approach with all its requirement documents and validation practices. As Cockburn puts it, when there is risk associated with skipping planning or making mistakes with the plan, then a plan-driven approach is more appropriate. For projects dealing with greater potential damage it is a MFI (money-for-information) situation: It is worth spending a lot of money now to discover information about where those next defects are located.

On the other hand, when there is risk associated with taking a slow, deliberate approach to planning, then agile techniques are more appropriate. For example, too much planning and time used to development could cause that the product will be too late on the market. I think that this principle is essential especially when entering to new markets.

The validation practices are also more important as the number of people involved increases. Of course, this is an issue of communication, but Cockburn shows that the bigger is the team, the better it endures when there is more and more methodology. I would see that so, that in a big team, there is always somebody who has time for documentation. Now and then it is best to read the documentation of others instead of asking questions. Perhaps the methodology also helps to maintain discipline in bigger teams, some kind of hierarchy is needed to make decisions effectively.

3.2 Documentation and communication

On documentation and other communication Cockburn has the following principles:

Principle 3: Larger teams need more communication elements.

Principle 6: Interactive, face-to-face communication is the cheapest and fastest channel for exchanging information.

Principle 7: Increased communication and feedback reduces the need for intermediate work products.

It should be trivial, that is it impossible to have a single set of coordination elements to fit all the software projects. The bigger the team or amount of people involved, the more documentation and additional presentations are needed to share the information. Agile software development strategies emphasize face-to-face communication. Often all the people involved in projects don't even live in the same continent, thus the principle six about cheapest communication is not necessary true in all circumstances.

According to Cockburn, the intermediate work products tend to have two forms: a) promises as to what will eventually be constructed, and b) intermediate snapshots of the developers' knowledge (design descriptions). If users, managers and proxies get regularly see the developing system, they stop needing elaborate promises of what they will be given. Thus, communication reduces the need for intermediate work products. However, since direct face-to-face communication is not always possible, there are the markets for documentation and design descriptions.

The conclusion of Cockburn seems to be, that there is sense in documentation when other communication is impossible. Cockburn emphasizes the meaning of mock-ups, prototypes, and simulations. Those deal with the MFI (money-for-information) aspects of the situation. They are an expenditure of resources to discover information sooner. These can be used as well with plan driven approach as with agile methods. In addition of prototyping, the agile methods emphasize allocating resources for the inevitable surprises resulting from real delivery.

3.3 Discipline

Cockburn's principle 5 says: "Formality, process, and documentation are not substitutes of skill, discipline and understanding". "There is clear difference between discipline and formality, skill and process and understanding and documentation", he continues.

In my opinion, the waterfall model formally demanding all requirements and design be finished first and allowing the implementation to start only after all that is done, can only guarantee that formality is followed. In reality life things are done more or less in chaos, by starting the implementation even though the requirements are still not done, without preparing the result for change as the agile methods require. In disciplined agile project some, limited set of requirements are made ready and the implementation for those starts when all agree about the end-result. By keeping the set of requirements small enough, the risk of doing unsuitable design is not so crucial for the whole project or product, as in worst case we are throwing away all the work done in three weeks. For software projects continuing over a year, this is not so much, particularly as this is quite unlikely.

Skills and ability to follow process can also be seen as two different things. However, the agile philosophy seems to assume, that it is easy to get skilled people to software projects. Cockburn states, that an agile project manager relies on discipline, skill, and understanding, while requiring less formality, process, and documentation. I admit, that it is possible to write unnecessary, hundred paged document due to process requirements. Or when reading formal documents done during software projects, the understanding may not increase at all. But when the face-to face communication is not possible, some kind of written format might be the only way to share understanding.

3.4 Concurrency

Cockburn's principle 8 says: "Concurrent and serial development exchange development cost for speed and flexibility". Serial and concurrent development have opposing characteristics. Cost-sensitive projects should use serial development where they can, while projects sensitive to shifting requirements benefit more from concurrent development. As I mentioned above, it is better to split the design tasks in parts and start to work with them concurrently, than trying to put all in serial order by force. Inside one part, cycle of iteration, a serial approach makes sense. According to Cockburn, agile project teams almost always use concurrent development assuming a significant number of surprises will arise during development. The close communication needed for effective concurrent development also lets them respond to late breaking changes effectively. Anyway, for example the sprints in Scrum use serial development inside of one sprint [Scr08].

3.5 Efficiency

Cockburn has principle 9 about efficiency: "Efficiency is expendable in non-bottleneck activities".

The principle says that rework is an expendable commodity everywhere except at the bottleneck station. Rework can be expended to improve a design, to investigate multiple designs, or to get a head start on a downstream activity. Applying this principle to different circumstances produces different optimal project strategies.

4. Ideal software development

In his last principle, Cockburn says that "Sweet spots speed development". In other words, the fastest development happens in ideal conditions. Cockburn calls aspects of ideal conditions as sweet spots.

"The ideal project uses dedicated, experienced people who sit within earshot of each other; use automated regression tests; have easy access to the users; and deliver running, tested systems to those users every month or two", writes Cockburn. Such a project is clearly in a better position to complete successfully than one missing those characteristics. According to Cockburn the surprise is that sponsoring executives do not pay more attention to these important success factors. So, what to do to strive for the ideal? Cockburn has following six "sweet spots" to be considered [Coc02].

4.1 Dedicated Developers

Dedicated software developers love their work and want to concentrate into it. Cockburn describes his experiences: "In my project reviews, I find that once people get interrupted at the rate of about three times per day, they stop even trying to focus on their main assignment and simply wait for the next interruption to happen. One senior project manager reported that he simply does not count as productive staff anyone assigned less than half-time to the project".

Thus, in ideal development conditions, the developers can concentrate in their work. I see that, the challenge here might be, that a dedicated developer might be the best professional to tell the others what a certain software is really doing, what are the development possibilities and so on. Lot of software developer's time is spent in telling the managers what is possible with the system they manage.

4.2 Experienced Developers

According to Cockburn, the experienced developers know the domain, they know the technologies or how to adopt them, and they know their computer science material. They move at multiple times the speed of their

slower colleagues. I would add excellent social abilities and positive, humble attitude towards work. The developer can't be an artist.

4.3 Small Collocated Team

Cockburn describes the ideal team with two to eight people sitting in the same room. They can ask each other questions without raising their voices. They are aware of when others are available to answer questions. They overhear relevant conversations without pausing in their work. They keep the design ideas and project plans on the board in ready sight and share information faster. Cockburn claims, that the developers he has interviewed uniformly say that while the environment can get noisy, they have never been on a more effective project than when a small team sat in the same room.

According to Cockburn, cameras and chat applications can replace the working room. I have same kind of experiences when we used IRC in Elisa Internet as communication tool. One reason might be, that the character of typical software developer might be less social and talkative - but using internet based communication was very easy to adopt.

4.4 Automated Regression Tests

With automated regression unit and acceptance tests, the developers can revise the code base and retest the entire system at the push of a button. So, they execute previously run tests and check whether previously fixed faults have re-emerged. This ideality requires test-driven approach from the beginning of the development.

4.5 Easy Access to Users

Ideally, the development team has a customer or usage expert available at all time. That means that the feedback cycle from nominated solution to evaluated idea is much shorter. I would add that in real life, the proxy of customer is often a sales expert, not necessary familiar with challenges of software development, thus the ideal customer is also a software development professional.

4.6 Short Increments and Frequent Delivery to Real Users

Cockburn emphasizes the importance of short development cycles. "With short increments, the process itself gets tested and can be repaired quickly, and the requirements for the product can be tested and varied quickly. Projects that cannot deliver to an end user every few months should integrate a full build every few

months and pretend as though it were delivered. This way, they exercise every part of the development process."

5. Examples of using agile ideas in software development

My first experiences of agile software development are from the beginning of year 2001, when XP was the hit word. Extreme programming, XP, garnered significant public attention because of its emphasis on communication, simplicity, and testing, and its sustainable developer-oriented practices. Maybe one reason was also its interesting name, extreme programming [Lab03].

XP is a light-weighted, incremental and iterative process model. The core in XP philosophy is a compact project team that work together in intense interaction, even programming in pairs. The aim is that everybody knows what the others are doing and what the team should do next.

I will first shortly go thru some XP principles for planning, design, coding and testing and then describe one real world scenario, how some of these principles were used in a small data-processing department of a Finnish company.

5.1 Characterizes of Extreme Programming

The website www.extremeprogramming.org describes all the principles of extreme programming. I picked up the following

Planning as a game

The planning is sometimes seen as a planning game in XP. That game consist of customers, who first describe their needs in form of short, 3-5 sentence stories. The aim is to divide the work using these stories, to make some of the related stories possible in each iteration. The developers then estimate how much effort is required to build each story. Then the customer chooses which stories s/he wants to be built in the next cycle, based on the time available and the estimates from the developers.

The responsibility is devolved on customer

XP emphasizes intense collaboration with customer. It is a must, that the customer is always available. A successful planning game requires the customer to be in intense relationship with developers during all the cycles. On the other hand, XP states, that it is not even possible that the customer knows all the requirements for the software first and then just leaves the producing to software team. Instead, the customer should be available all the time. Asking questions and negotiating scope require more than just the developers be involved in producing the software. It is the customer (or a proxy of the customer) who must make the

decisions that affect their business goals. The customer is a real professional capable to give feedback for project team.

Iteration

The project is divided into iterations. The aim is that the time between releases should be from one to three weeks. If this is not possible, then some requirements are moved to next iteration.

Short meetings

Daily stand up meetings are also introduced in XP, but the format of meetings is not that precise as in scrum. No person like scrum master is introduced in XP.

Code is not inviolable

Code must be written to agreed standards, no code artists are tolerated. In orthodox XP, all production code is pair programmed. In that, two programmers are working on one desk, one is writing code and the another checks the result all the time. By this, there is no code that is only in one persons mind. Code and the whole project is in collective ownership of the project team. Thus, also the responsibility is shared. Anyone is allowed to change written code if needed and nothing should be taken personally. Best practice for a XP process is one working room where all the coding pairs fit sit in. This kind of working environment may also fortify the feeling of collective responsibility.

Trust in unit testing

Planning and implementing tests is essential part of planning and coding phases in XP. This is anyway not so much emphasized as in test-driven development [Tes08]. Anyway, in XP it is emphasized that all code must have unit tests and all code must pass all the unit tests before it can be released.

No overtime working

The work estimates are done based on user stories written by customer. If the workload was underestimated, the customer must decide what is postponed to next version. Overwork is not XP philosophy, in hurry the work quality decreases and people can't give their best when tired.

5.2 Using XP in Elisa Internet Ltd

My own experiences about agile software development start from year 2001, when I started as summer trainee in Elisa Internet Ltd. By that time Elisa Internet Ltd. was a subsidiary company of Finnish telecom group Elisa Communications Corp. The company was also called by a name Kolumbus Ltd, but it had recently changed the name. Elisa Internet offered Internet services in Finland, for both type of customers,

companies and individuals. By that time, all the customer relationship management systems were built in house and Elisa had relatively large and complex IT infrastructure.

In 2001, I started working in a tiny data-processing department that both developed and maintained several softwares for order, agreement and customer relationship management. We also had so called "electric customer services", mainly web sites where customers could make orders, change their passwords and so on. By the time I arrived to the company, our software architect had heard about XP, extreme programming. He was very excited, and wanted to apply the principles in daily development.

The manager of software department was excited as well and encouraged us to try new process models.

By that time, we were supposed to build a new system for management of orders and agreements.

The requirements for the system were so demanding that it should fit for orders of any product or service type the company (Elisa Internet) could ever want to sell to its customers. The system was also to have a self-service interface for self-motivated customers, where they could e.g. make additional orders and change passwords for on-line services. As the selected software development style was the trendy XP, also the new software was named in Finnish to "AittaXP", which means something like storehouse-XP. (I have to emphasize here, that Microsoft published its Windows XP after us!)

The software development started with architecture plans. The architecture style selected was something like product line architecture consisting of layers. The idea was, that we will first code a product platform and then it will be easy to integrate all products on it. So, the first "iteration" was not XP style three week phase but it was planned to be half a year, spent in building the platform.

Planning

We forget all the traditional requirement documents. By general shared opinion, it was so difficult to guess what there are in the minds of product managers and sales people, that it is better to plan a system where everything is possible. So, we borrowed from XP the planning seen as a game, but did not follow XP's formal way in iteration planning. Either someone of the developers or a project manager (we had no project manager's all the time) discussed with sales and product management people and then the whole development team sit around a coffee table informally discussing about the requirements. Sometimes we draw work-flows on a whiteboard. All the people involved in the process sit in same building.

When we had knowledge enough what should be done, we draw the user interface workflows and architecture description of java classes.

As the proxy of customers was not even interested about the software but only the timelines (when they could tell sales that they can give the internet self-service prochures to customers) devolving the responsibility to customer was not successful.

Iterations

We were supposed to develop product -platform first and there were no iterations and no timelines in the beginning. That was a short-term delight, because after an half of year, the management started to be anxious and set a timeline for the first product to be released. The product was supposed to be out so quickly, that no-one had time to plan schedules or XP style iterations. We just coded.

Short meetings

As the whole development team (including the product managers and sales people) was in the same building, it was relatively easy to organize meetings. We had no formal way for meetings but had two "fixed by law" coffee breaks in a day. Sometimes those coffee breaks take more than hour and we did not necessary talk about work related issues. Instead, we had good time and joked a lot. When there were something work related problems, the we discussed about them. We also used IRC to discuss online when sitting at home (remote work) or in work rooms. The IRC helped shy people to participate in discussions and strengthened team spirit.

No own code

From the beginning it was clear, that anyone could touch the code of the others if needed. We had a shared plan how one should code. We used a lot comments in code. We released the code to common environment as it seems to work in own desktop. After that, anyone could use it and rewrite it when needed. In the beginning, we also used pair programming. There were two pairs, where one was a practised designer and the other was a beginner. I took my first programming courses in autumn 2000, so I was a complete freshman by that time. I sat two-three days in the week in summer 2001 on same desk with a practised designer and tried to catch errors. On the same time, my pair was telling what he was doing and I learned a lot about the structure of the system. The rest of the work week I sat on my own desk trying to figure out how to do my tasks. In autumn 2001, the management figured out that speeding up is needed, and that was stop for the pair programming.

6. Summary

According to Cockburn, agile teams put more emphasis on the ideas presented here in the chapters two, three and four, than do plan-driven teams [Coc02]. Recall the Agile Alliance introduced in chapter one. Their Agile manifesto includes following principles [Bec01]:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

These principles could summary as well the ideas described in this document. After all, nowadays, it is not enough that the software development team is agile and flexible in all circumstances. Also the whole company from up to sales and upstream management to bottom to testing and server maintenance should participate in the process. There have been discussion about that spreading the agile thinking to management level is nothing more than a new name to lean practices already widely spread in economy and management schools [Jam05]. However, agile software development process increases the ability to respond to change, uncertainty and unpredictability in the software business environment, whatever its source - customers, competitors, new technologies, suppliers or government regulation [Jam05].

7. Sources

- Bec01 Beck, K., et al., *Manifesto for Agile software Development*, 2001.
<http://agilemanifesto.org/> [20.3.2008]
- Coc02 Alistair Cockburn, *Learning From Agile Software Development - Part One*.
Crosstalk October 2002 and
Alistair Cockburn, *Learning From Agile Software Development - Part Two*.
Crosstalk November 2002
- Jam05 T. James, *Stepping back from lean [lean vs agile manufacturing]*.
IEEE software, Volume 84, Issue 1, Feb.-March 2005 (Pages 16 – 21)
- Lab03 Craig Larman and Victor R.Basili, *Iterative and Incremental Development: A Brief History*.
Computer , June 2003 (Vol. 36, No. 6) (Pages 47-56)
- Scr08 Scrum
[http://en.wikipedia.org/wiki/Scrum_\(development\)](http://en.wikipedia.org/wiki/Scrum_(development))
[20.03.2008]
- Tes08 Test-driven development
http://en.wikipedia.org/wiki/Test-driven_development
[20.03.2008]