1. Introduction

Agile Methods (AMs) are a set of development techniques designed to address some problems of modern software development (i.e. projects over budget and over schedule). Such methods do not pretend to be useful in any kind of software project or to be the solution to reduce costs and increase quality of any product. However, in specific contexts and for specific problems, AMs simply helps developers to focus on the objectives of their customers and deliver the right product for them without wasting time and effort in activities that are not able to generate value for the customer.

Traditional software development approaches (i.e., waterfall, spiral, iterative, etc.) require a deep knowledge of the application domain and of the actual needs of the customer (including the final user). However, this precise knowledge is rarely available and even in such cases the customer usually asks for changes during the development. Unfortunately, software development is characterized by uncertainty and irreversibility (Berry, 2002; Highsmith and Cockburn, 2001), therefore planning everything upfront is not useful in many application domains.

Uncertainty means that requirements are not stable and the customer is not able to specify them in a complete and coherent way. Often customers are not even able to tell which are the main functionalities required and they change their mind very frequently.

Irreversibility means that even if software is intangible some decisions cannot be changed without deeply affecting the schedule and the budget of a product. This is well understood in other disciplines such as civil engineering. The customer is aware that he cannot change the shape of a building just before building the roof. However, in the software area, customers are often unaware that certain modifications have a comparable impact.

The main consequences of uncertainty and irreversibility are the following:
1. the complete knowledge required to build a system is not always available and/or it is subject to change during development
2. make assumptions trying some solutions and throw them away if they do not fit is not always applicable (waste of time and money)

Traditional software development techniques try to limit uncertainty and irreversibility through very detailed plans. They try to specify everything at the beginning in order to avoid expensive changes later in the project. However, in certain application domains and for certain problems, plans simply do not work or they are inefficient. This is true regardless the quality of the people involved in the project.

Even if plans are supposed to help organizations, many project managers acknowledge that in many kinds of projects they cannot follow them due to market needs, and often they succeed.

AMs acknowledge the difficulty of defining detailed plans at the beginning of a project and include the support to changes into the development process. In this way, requirements are collected and discussed with the customer for the entire duration of the project according to the feedback
provided by the customer. The development is performed iteratively and the product is shipped to
the customer at every iteration (lasting from 2 weeks to 2 months in XP) for the evaluation.

AMs focus only on the final outcome of the development process (the code) and on the value
provided to the customer. Additional artefacts (i.e., design documents, huge documentations, etc.)
are considered waste, since they require time for their development and become useless and
misleading if not properly maintained. A poor maintenance of the documentation is common in case
of delays in the development. In such cases, all the effort is spent in coding (and testing) the code
without updating the documentation properly. Since this scenario is quite common, the main idea is
to reduce the time spent for the unnecessary documentation and use automated tools for reverse
engineering to produce it when needed.

Often, AMs are misinterpreted. For instance, some people say that they are applying AMs because they:

1. jump directly into the code without writing analysis and design documents
2. do not write documentation
3. reduce the time spent in meetings

However, this is not AMs, this is cowboy coding. AMs provide a way to organize software
development beyond plans but still with a rigorous process to follow. Often, using AMs is more
challenging than applying traditional software development methods since AMs require a higher
level of commitment, higher kills, etc. They are very difficult to implement and to manage but they
are designed to cope with the challenges of the modern software market (i.e. short time-to-market,
high quality, etc.).

Moreover, AMs require more agility to the organizational structure of a company (i.e. new kinds of
contracts, more decisions left to the team, etc.) and more flexible people with multiple skills and the
ability to play several roles inside the development team. Therefore, AMs are not for everyone and
they do not fit every software project.

This chapter analyzes the key concepts of the agile development and the difficulties for their
effective implementation. Even if there are several AMs available (XP, Scrum, DSDM, Crystal,
Agile Modelling, etc.), we focus on the first version of Extreme Programming (XP) (Beck, 1999)
since it is the most popular one.

2. Agile Methods

AMs are a family of development techniques designed to deliver products on time, on budget, and
with high quality and customer satisfaction (Abrahamsson et al., 2002; Cohen et al., 2003). This
family includes several and very different methods. The most popular include:

- eXtreme Programming (XP) (Beck, 1999; Beck, 2004)
- Scrum (Schwaber and Beedle, 2001)
- Dynamic Systems Development Method (DSDM) (Stapleton, 1995)
- Adaptive Software Development (ASD) (Highsmith, 1996)
- The Crystal family (Cockburn, 2002)

The aim of these methods is to deliver products faster, with high quality, and satisfy customer needs
through the application of the principles of the lean production to software development
(Poppendieck and Poppendieck, 2003).
The concepts of the lean production (Womack and Jones, 2003) have been developed during the '50s at Toyota (Ohno, 1988). Such concepts involve several practices that are now part of most manufacturing processes, such as just-in-time, total quality management, and continuous process improvement. The basic principle of lean production is the constant identification and removal of waste (muda in Japanese), that is, anything that does not add value for the customer to the final product.

Since AMs are in some sense the implementation of the lean production in the software area, they focus on:

1. delivering value for the customer
2. satisfying the customer

Delivering value to the customer implies that the development team has to produce only what provides value and reduce to the minimum everything else. AMs emphasize producing and delivering to the customer only useful features (from the point of view of the customer). Producing anything that is not required is considered a mistake. In particular, adding a feature that is not needed not only requires effort but also creates additional code, which may contain errors and make the code larger and more complex to maintain, to correct, and to improve.

To achieve such reduction of waste, AMs claim to be (Abrahamsson et al., 2002):

- adaptive rather than predictive
- people-oriented rather than process-oriented

To ensure customer satisfaction, a close collaboration between the development team and the customer is required. In this way:

- requirements are fully identified and correctly understood
- final products reflects what the customer needs, no more and no less.

3. The Agile Manifesto

The Agile Manifesto (http://www.agilemanifesto.org/) summarizes the basic and common background of all the AMs. Such document defines the aim of the AMs and it is the reference point for the entire community.

AMs emphasize the human factor in the development process and the importance of direct, face-to-face communications among stakeholders, the value of simplicity, perceived as an elimination of waste, and continuous process improvement, as the transposition to the Software Industry of Total Quality Management (Poppendieck and Poppendieck, 2003).

The Agile Manifesto defines AMs as a set of development methods sharing the following four values:

1. **Individuals and interactions over process and tools**: AMs emphasize the collaboration among developers and the human role in the process and in the organization, as opposed to institutionalized processes and development tools (Abrahamsson et al., 2002).

2. **Customer collaboration over contracts**: AMs give more importance to the cooperation between developers and customers rather than to a careful and detailed definition of contracts (Abrahamsson et al., 2002). Informal communication between the team and the customer can replace most of the written documentation, even very detailed contracts.

3. **Working software over documentation**: one main source of waste in software development is extra processes or documentation. Paperwork consumes resources, slows
down response time, hides quality problems, gets lost, degrades, and becomes obsolete. When paperwork is required, it is necessary to keep it short, high level, and done off line. AMs focus on the understanding of the product through collaboration with the customer and delivery of working software, thus reducing the amount of documents required (Highsmith, 2002).

4. **Responding to change over planning**: “Agility is the ability to both create and respond to change in order to profit in a turbulent business environment” (Highsmith, 2002). Changes are opportunities to help customers to address the turbulence of the marketplace rather than problems for the development.

The former two values refer to management of human resources, while the latter ones refer to process management.

From these values, there are some derived principles that are shared among all the AMs. The main principles listed in the Agile Manifesto are the following:

- **The highest priority is to satisfy the customer through early and continuous delivery of valuable software**: developers have to ship the product incrementally starting from the most important requirements for the customer.
- **Welcome changing requirements, even late in development. Agile processes harness change for the customer’s advantage**: changes are not a devil but a natural state of software projects. It is not possible to prevent changes that are always unpredictable, therefore the development process has to accommodate and not fighting them.
- **Business people and developers must work together daily throughout the project**: a close collaboration between the development team and the customer is a way to reduce the risk of a project since the correct interpretation of the customer needs is verified at every step. In this way, the rework caused by misunderstandings is reduced to the minimum and developers always know to proceed in the right way.
- **Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale**: working software is the only valuable thing for the customer even if it is not complete. Shipping a few working functionalities very frequently and not just prototypes give the customer the possibility of using the product early. Moreover, it increases the visibility of the status of the project.
- **Build projects around motivated individuals**: AMs are not for everyone, skilled and motivated developers that are able to work in team are an essential factor for the success.
- **Give them the environment and support they need, and trust them to get the job done**: managers should not interfere with the development team. Trust and support from the management is a good way to support a skilled and motivated team.
- **The most efficient and effective method of conveying information to and within a development team is face-to-face conversation**: even if there are several high-tech ways of communicating, face-to-face communication is still the most effective preventing misunderstandings and shortening the time spent for exchanging information.
- **Working software is the primary measure of progress**: the only outcome important for the customer is working software. It does not matter if the team has produced wonderful design documents but the product is not working or it does not satisfy the needs of the customer. The customer measure the progress of a project measuring the number of working functionalities delivered.
- **Agile processes promote sustainable development**: The sponsors, developers, and users should be able to maintain a constant pace indefinitely: software development should be carried out through a constant and continuous collaboration among the stakeholders. Moreover, the effort of the development team should be constant in time trying to avoid
stressful periods that are not sustainable in the long term affecting the motivation and the productivity of the team.

- **Continuous attention to technical excellence and good design enhances agility:** top developers produce top software. Top developers are able to create systems that can be enhanced and maintained minimizing the time and the effort required.

- **Simplicity – the art of maximizing the amount of work not done – is essential:** identifying the code that has not to be written is a way to improve the efficiency of the development team. Simpler code is easier to write, understand, and maintain. Moreover, less code means less potential errors and tests.

- **The best architectures, requirements, and designs emerge from self-organizing teams:** the team is the only responsible for the entire product. There is no competition between teams that focus on specific activities (e.g., design and implementation). All the team members works to achieve a common goal: satisfying the customer.

- **At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly:** the development process is not fixed. Teams have to continuously identify areas of improvement experimenting and evaluating new techniques.

The listed principles are implemented and stressed in different ways in the several AMs available. However, the focus on the customer needs and the continuous improvement are the main ones for all of them.

The ideas of the customer that drives the production and the continuous improvement of the process are not a novelty. These are the main concepts of the Lean Production (Ohno, 1988; Womack and Jones, 2003) and of the just-in-time production used at Toyota in the '60 for building cars. AMs are the implementation of these concepts in the software industry (Poppendieck and Poppendieck, 2003).

### 4. Extreme Programming (XP)

Probably, the most well-heard-of AM is eXtreme Programming, a.k.a. XP. At present, there are two version of XP defined in the two editions of the Beck’s book (Beck, 1999; Beck, 2004). However, in this chapter, we are going to consider only the first one since it is the most popular and accepted by the community. The second one is the most recent proposal by Kent Beck in which he proposes interesting modifications and enhancements. However, the community is still discussing it and it is not widely accepted yet. Moreover, the first version is simpler and it should be the starting point for people approaching AMs for the first time.

XP practices are defined as the description of a successful process followed by the C3 team (Beck, 1999) that developed the huge payroll system of Chrysler. The C3 team lead by Kent Beck was successful and delivered a working system after two years, while the previous team was not able to deliver anything in four years.

XP defines in details a sequence of development principles and practices. However, these are not defined directly but they are derived from values and drivers that are the foundation of the methodology (Figure 1).

The **values** of XP are the following:

- **Simplicity:** the system should be as simple as possible to satisfy the needs of the customer but not simpler. Implement the needed features, but do not include features to support future requirements that may not become real ones.
• **Communication**: everything must improve communications between customers and developers, among developers and between source code and its reader. Effective communication reduces misunderstandings and the need of spending time and effort for formal and written documentation.

• **Feedback**: at all levels people should get very fast feedback on what they do. Customers, managers, and developers have to achieve a common understanding of the goal of the project, and also about the current status of the project, what customers really need first and what are their priorities, and what developers can do and in what time. This is clearly strongly connected with communications. There should be immediate feedback also from the work people are doing, that is, from the code being produced. A substantial percentage of programming effort (about 50%) must be spent in automated test development. In this way, the quality of the developed system is reasonably high.

• **Courage**: every stakeholder involved in the project should have the courage (and the right) to present his position on the project. Everyone should have the courage to be open and to let everyone inspect and also modify his work. Changes should not be viewed with terror and developers should have the courage to find better solutions and to modify the code whenever needed and feasible. Using the right tools and a comprehensive test suite it is possible to program and try new solutions without fear. This means that it is easy to test whether a modification introduces errors and it is easy to return to a previous working version, if needed.

The drivers of XP are the following:

- **Focus on value**: developers have to focus on what provides the most value to the customer. Priorities of the development are based on the priorities of the customer, not on technical issues that do not provide any value to the customer.

- **Constant flow of activities**: the development team has to work constantly, avoiding too much or too few workload.

- **No defects**: small and simple defects today may become large and difficult to manage bugs in the future. The team has to fix all the known defects and should verify that they are not coming back in future versions through automated testing.

The connections between the values and the drivers listed are shown in Figure 2.

The **focus on value** is evident in the simplicity. The team develops only the features that are important for the customers. Such focus is also present in the **communication** with the customer for eliciting requirements and the related priorities. Moreover, **feedback** from the customer is the driver of the development, since he defines the priorities of what to add and/or improve in the product.
The **constant flow** is clear in **feedback**, where the developers ask the customers their priorities and in the **courage**, where developers negotiate with the customers the amount of functionalities to deliver, without any fear from the customer side that developers “do not work enough.”

The aim at the **absence of defect** requires **simplicity** of design, to avoid inserting defects, **feedback** from tools and customers, to eliminate existing errors and detect non conformance with the wishes of the customers. Moreover, **communication** among developers and **courage** to test the code under the most severe circumstances help very effectively to eliminate defects.

From the values and the drivers, XP defines a set of **practices**:

- **Planning game**: planning should be done together by developer, managers, and the customer. Together, these three stakeholders write user stories of the system; then the customer sets the priorities, the manager allocates resources, and the developers communicate what is likely to be feasible to do. The communication on plan should be kept honest and open.
- **Short releases**: the development of the system should proceed in small increment with frequent releases that the customer should use to provide useful feedback.
- **Metaphor**: the overall project team should use a jargon shared among developers, customers and managers, so that developers could better understand the problem domain and customers could better appreciate developers’ solutions. Such jargon could be built around an overall analogy of the system being built.
- **Simple design**: the design should be kept simple and developed incrementally as the system evolves.
- **Test-driven development**: test should be written together with the customer and before the actual code; they should also cover all the most relevant aspects of the system. In this way, they serve as a way to ensure that requirements are met and as a form of formal specification of the behavior of the system.
- **Refactoring**: the code should be constantly revisited and made simpler and more understandable, with a clear constraint that (a) such simplifications should be done test first, (b) the simplifications should be checked in only when passing all the existing and new tests, and (c) the simplifications should be done in pair.
- **Pair programming**: programmers should always work in pair, where one stays at the keyboard and write the code, while the other proposes ideas and verifies the code being written.
- **Collective code ownership**: everyone in the team should have complete accessibility to any piece of code developed by anyone else in the team, and should be able to modify in and
check in a new version of it, provided that (a) proceeds test first, (b) checks in only a new
version passing all the existing and new tests, and (c) works in pair.

- **Continuous integration:** the code should be integrated frequently, to ensure that all the
pieces fit seamlessness together.

- **40 hours working week:** the project should proceed at a sustainable pace, along the lines of
the constant flow advocated by lean management. Therefore, major efforts might be
tolerable for a week or two per term, but overall the effort distribution should be flat and not
exceed what is normally bearable: 40 hours per week.

- **On-site customer:** the customer and the developers should be easily accessible and, if
possible, co-located. In this way, the customer would be ensured that the developers are
working along their plan and the developers could receive fast feedback from the customer.

- **Coding standards:** the code should be written in a style agreed upon by all the people in the
team, to promote sharing and easy understanding. What standard to use is not so important,
providing it is reasonable and it is accepted by everyone, but it is important to have one.

Drivers, values, and practices are deeply connected in XP. These connections are showed in Table
1.

<table>
<thead>
<tr>
<th>Drivers</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focus on value</td>
<td>Constant flow</td>
</tr>
<tr>
<td>Planning game</td>
<td>No defects</td>
</tr>
<tr>
<td>Short release</td>
<td>Communication</td>
</tr>
<tr>
<td>Metaphor</td>
<td>Simplicity</td>
</tr>
<tr>
<td>Simple design</td>
<td>Feedback</td>
</tr>
<tr>
<td>Testing</td>
<td>Courage</td>
</tr>
<tr>
<td>Refactoring</td>
<td></td>
</tr>
<tr>
<td>Pair programming</td>
<td></td>
</tr>
</tbody>
</table>
| Collective code
  ownership           |                      |
| Continuous integration |                      |
| 40 hours per week       |                      |
| On-site customer        |                      |
| Coding standards        |                      |

### 4.1. Structure of XP teams

The size and the structure of the team are very important for a successful implementation of XP. XP
is designed to work with a small team (2-12 developers) located in the same room. Such small size
and the co-location is of paramount importance for a correct implementation of some practices such
as planning game, collective code ownership, customer on-site, etc. If such requirements are not
satisfied, there are other AMs that may help (i.e., SCRUM, Crystal, etc.).

The level of agility is often related to the size of the development team. Direct communication and
limited documentation is possible only in small teams. On the contrary, when the team grows, the
level of overhead grows as well. The overhead includes:
• Documentation
• Mediated communication (through a media such as paper)

To share knowledge and trace the status of the project is required more documentation because direct, many-to-many interaction is not possible anymore (Cockburn, 2002). Moreover, the importance of the documentation increases and it becomes a way to improve knowledge sharing. In this case, the code itself is not enough and the direct communication between the development team and the customer is not possible due to the size of the team.

In XP teams there are three main components:
1. The customer
2. The developer
3. The manager

The customer is highly involved in the development process and often he is a member of the development team. His presence is important in XP, since most of the communication is performed face-to-face and requirements collection phase is spread across the entire project, not just at the beginning. Therefore, development team often asks the customer to answer questions regarding requirements and check the correctness of the implementation.

The presence of the customer reduces the amount of documentation required to describe requirements in details and his direct contribution is a key factor for the success of the project. Moreover, the customer provides feedback to the developers to identify potential problems early in the development and avoid a major impact on project schedule.

The main activities of the customer are the following:
• Definition of the functionalities of the product
• Definition of the priorities for such functionalities
• Definition of acceptance tests (with the help of developers)

Developers in XP are not just the people in charge of the implementation of the product. They have to interact closely with the customer providing working software and collecting valuable feedback. Therefore, they not only need to be good developers able to work in team but also they need to be able to communicate with the customer with his own language.

Developers have to provide working and high quality software to the customer at every iteration and collect valuable feedback. This approach is valuable for both developers and customers. Developers can collect useful information to avoid the implementation of useless or wrong features that reduces the time spent in useful features; customers can test the product after a few weeks from the beginning of the project.

The main activities of the developer are:
• Analysis, design, test, coding, and integration of the system
• Estimation of the difficulty of the implementation of the requirements expressed by the customer
• Works in pairs. Two programmers with one keyboard: one types, the other tells what to do
• Share the code with all the other developers in the team

In AMs, managers have to create and sustain an environment where it is possible a productive interaction between the development team and the customer. They can achieve this goal identifying
the best people to be included in the team, promoting collaboration, and negotiating contracts with the customer.

Usually, agile teams work with *variable scope-variable price* contracts rather than *fixed price-fixed scope* ones. This approach relies on the ability of the manager in the contracts definition to satisfy the customer and allow flexibility in the development process.

The main activities of the manager are:

- Outlines the contractual terms and negotiate with the customer the application of the agile practices
- Helps the customers and the developers to become one, cohesive team
- Facilitate customers and developers in doing their tasks

A key factor for the success of an XP project is the commitment of the team members. Not only the commitment of developers but also of the managers and of the customers. Managers have to support the development team and create the environment for a productive application of XP; customers have to be available to answer the questions coming from the development team and evaluate the proposed solutions.

### 4.2. Requirements management in XP

Requirements are the base of all software products and their elicitation, management, and understanding are very common problems for all development methodologies (Sommerville and Sawyer, 2000).

According to a study of the Standish Group (Standish, 1994), five of the eight main factors for project failure deal with requirements (Table 2): incomplete requirements, low customer involvement, unrealistic expectations, changes in the requirements, and useless requirements.

<table>
<thead>
<tr>
<th>Problem</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incomplete requirements</td>
<td>13.1</td>
</tr>
<tr>
<td>Low customer involvement</td>
<td>12.4</td>
</tr>
<tr>
<td>Lack of resources</td>
<td>10.6</td>
</tr>
<tr>
<td>Unrealistic expectations</td>
<td>9.9</td>
</tr>
<tr>
<td>Lack of management support</td>
<td>9.3</td>
</tr>
<tr>
<td>Changes in the requirements</td>
<td>8.7</td>
</tr>
<tr>
<td>Lack of planning</td>
<td>8.1</td>
</tr>
<tr>
<td>Useless requirements</td>
<td>7.5</td>
</tr>
</tbody>
</table>

In XP the requirements collection is not performed only at the beginning of the project but this activity lasts for the entire development. In particular, the customer can change both the requirements and their priorities at every iteration of the project after evaluating the system shipped by the development team in the previous iteration. This continuous feedback is important to keep the project on track and deliver software that is really able to satisfy the needs of the customer.

XP acknowledges that requirements variability is a constant problem in nearly all software projects; therefore, the support to such changes is included in the process as a key strength (Tomayko, 2002). Moreover, XP does not try to forecast changes or future needs, it focuses only on the features for
which the customer is paying. This approach avoids the development of a too general architecture that requires additional effort (Beck, 1999).

To improve the mutual understanding and the effectiveness of the requirements collection, XP teams use user stories and metaphors.

User stories are a description of the functional behaviour of the system. They are quite similar to use cases but they focus only on micro-functionalities, therefore a use case can be divided into several user stories. User stories are written in natural language by the customer with the help of the developers. Customers assign priorities to user stories while developers estimate their difficulty and the effort needed for their implementation (story points).

Effective user stories should:
- Be understandable by both customer and developers
- Be written in natural language
- Be testable (and the test should be written by the customer with the help of the developers)
- Be independent from all the other stories to the largest possible degree (at least from the stories developed within the same iteration)
- Not exceed two weeks of work (typical length of XP iterations)

A typical user story is described in Figure 3.

Developers estimate the effort required by a user story on the base of their experience and on the base of the ongoing project. Large user stories may require a deeper investigation to access their
feasibility and the effort required. In these cases, developers implement spike solutions which are a more accurate investigation and an experimental implementation of the skeleton of the solution to generate a more accurate estimate. The effort is specified in story points that have to be related to a specific amount of effort such as one day of a pair (developers always work in pairs!). If the story seems to require more than two weeks, the customer will be asked to split it.

Attached to every user story there is the acceptance test that specifies when the story is implemented correctly and the implementation is accepted by the customer.

Effective acceptance tests should:
- Verify that the stories are correctly implemented
- Be written by the customers after writing each individual user story
- Be at least one for each story
- Verify when it is possible to consider a story finished and start developing a new story

To write effective user stories customer and developers should use metaphors. Metaphors are a kind of lingua franca that can be easily understood by both the customer and the developers and should help them to communicate without using too much technical language that may generate misunderstandings if the application domain is not well known by the entire team.

4.3. Introduction to the XP development process

Traditional development processes such as the waterfall model (Figure 4) identify phases of the development process (analysis, design, coding, testing) through which a product has to go through. This rigid structure has been modified in different ways with other development processes such as the spiral model and several others.

![Figure 4: Waterfall process](image)

XP organizes the development process in a radical different way (Figure 5). The formal definition of phases is not present anymore and the development process is organized in iterations in which developers have to produce something valuable for the customer. For every requirement identified through a user story, developers perform a little bit of analysis, a little bit of design, a little bit of testing, and a little bit of coding.

This process has some interesting points. Testing is performed before coding because XP uses the test-first approach (Beck, 2002): tests are written before that the code that satisfies them is written. It is not true that in XP there are no analysis and design. XP spreads the analysis and design across the entire development process and does not concentrate them only at the beginning of the project. This approach allows developers to react quickly to the changes and reduce the waste of resources caused by the implementation of wrong requirements (Beck, 1999).
4.4. Comparing XP with other approaches

Table 3 compares some common approaches to software development of traditional software engineers, cowboy coders, and XP-ers.

Table 3: Comparing XP with other approaches

<table>
<thead>
<tr>
<th>Traditional Software Engineer</th>
<th>Cowboy Coder</th>
<th>XP-er</th>
</tr>
</thead>
<tbody>
<tr>
<td>“I need to have completed analysis and design before proceeding to code”</td>
<td>“I do not need any analysis and design”</td>
<td>“I do not need to have completed analysis and design before proceeding to code”</td>
</tr>
<tr>
<td>“I need to write all the documentation in a complete and pervasive way so that people in the future will be able to understand what is in here”</td>
<td>“I do not need any documentation”</td>
<td>“I need to write the code so that people in the future will be able to understand what is in here. I need to write only the documentation that is needed by people”</td>
</tr>
<tr>
<td>“Especially close to the deadline, I need to work like crazy to get the project done. Programming is hard”</td>
<td>“Only close to the deadline, I need to work like crazy to get the project done. Programming is fun”</td>
<td>“Especially close to the deadline, I need to work no more than 40 hrs a week to get the project done, keeping a constant pace and a fresh mind. Programming is fun”</td>
</tr>
<tr>
<td>“The code is mine and none is allowed to touch it!”</td>
<td>“The code is mine and none is allowed to touch it!”</td>
<td>“The code is of the team and everyone is allowed to modify it also extensively, provided the tests keep running!”</td>
</tr>
<tr>
<td>“At the end we will do integration. It is going to be hard, so we need to define precise interaction protocols, and to document them with maximal details”</td>
<td>“At the end we will do integration! No problem, it’s easy: it will take 5”</td>
<td>“We need to integrate our system at least daily, so that at the end we will not have any problem”</td>
</tr>
<tr>
<td>Traditional Software Engineer</td>
<td>Cowboy Coder</td>
<td>XP-er</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------</td>
<td>-------</td>
</tr>
<tr>
<td>“The customer should only see working and cleaned-up versions of the product. It is important to balance the contact with the customer so time is not wasted”</td>
<td>“If possible, the customer should only see the final versions of the product. It is important to minimize the contact with the customer so time is not wasted”</td>
<td>“The customer should (a) be constantly exposed to the product being build and to the development team, and, whenever possible, (b) have a representative on site”</td>
</tr>
<tr>
<td>“If it is not broken, do not touch it!”</td>
<td>“Even if it is broken, do not touch it! Try to hide it!”</td>
<td>“Even if it is not broken, constantly refactor it! Use the test cases to ensure you do not introduce an undesired bug”</td>
</tr>
<tr>
<td>“Plan everything well in advance so that there will be no need to change! A change is a clear symptom of bad and/or not sufficient planning”</td>
<td>“Do not plan anything and try not to change! A change is a clear symptom of an annoying customer or manager”</td>
<td>“Plan everything that you can reasonably foresee and get ready to change! Changes occur naturally in software projects”</td>
</tr>
<tr>
<td>“Change is evil! Fight it!”</td>
<td>“Change is evil! Fight it!”</td>
<td>“Change is human! Get ready to cope with it!”</td>
</tr>
</tbody>
</table>

4.5. Control mechanisms in XP

Any kind of production process is regulated through some control mechanisms that define how to synchronize different activities to achieve a common goal.

There are two main ways to control a production process: exogenous and endogenous control. Exogenous control defines rules added to the process. This means that the process itself does not include such rules but they are added later to implement a control mechanism. On the contrary, endogenous control defines the control rules as part of the process. This means that the process has been designed so that control mechanisms are embedded in the process and it is not possible to separate them.

Traditional software engineering methods use mostly exogenous control. On the contrary, AMs take advantage of endogenous control. This means that several AM practices are designed to force developers to coordinate without asking them to do it explicitly, limiting the not directly productive activities needed only for coordination. Therefore, all the stakeholders can concentrate on their core business while problems are solved when they arise: the endogenous control prevents going ahead if a problem is not solved. Needless to say, this is a clear driver for quality: anything that does not match the specified quality control cannot go ahead.

The Agile Manifesto includes endogenous control in all its principles. Endogenous control is way more effective than exogenous control, however it may be hard to achieve. AMs try to obtain an adaptive development process whose control is built-in, rather than added later on the process. This approach is one of the tenets of Lean Management (Womack and Jones, 2003).

According to Malone and Crowston (1994) the only way two tasks can be dependent is via some kind of common resources. Consequently, there are three major kinds of dependencies among tasks:

- **Sequential control:** it arises when a task creates a resource (output) that another task requires as an input (Thomson, 1967). In this case there is a precedence dependency
between the two tasks, requiring the correct execution order (Malone and Crowston, 1990) (Figure 6).

**Figure 6: Sequential control**

- **Shared resource:** it arises whenever multiple tasks share some limited resources (Malone and Crowston, 1994) (Figure 7).

**Figure 7: Shared resource**

- **Common output:** it occurs when two tasks contribute to create the same output. This dependency can have either positive or negative effects. In fact, if both tasks do exactly and unintentionally the same thing, it may result in a problem of duplication or waste of resources. Nevertheless, two tasks can affect different aspects of a common resource. This is the case of more actors that collaborate in order to achieve a common goal (Figure 8).

**Figure 8: Common output**

The easiest coordination mechanism is the sequential one. In this case the activities are completely independent and the only relation is through the input and output documents. The shared resource coordination mechanism is more complex since activities have to define priorities for assigning the shared resources to the most important activity. The most difficult one is the coordination through common output. In this case activities have to work together to produce the same output.
Traditional software engineering techniques are mostly based on exogenous control and sequential coordination. On the contrary, AMs and XP in particular are based on endogenous control and common output coordination. Table 4 lists the control mechanisms used in the single XP practices.

<table>
<thead>
<tr>
<th>Practice</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning game</td>
<td>Endogenous</td>
</tr>
<tr>
<td>Short releases</td>
<td>Endogenous</td>
</tr>
<tr>
<td>Metaphor</td>
<td>Exogenous</td>
</tr>
<tr>
<td>Simple design</td>
<td>Exogenous</td>
</tr>
<tr>
<td>Test-driven development</td>
<td>Endogenous</td>
</tr>
<tr>
<td>Refactoring</td>
<td>Endogenous</td>
</tr>
<tr>
<td>Pair programming</td>
<td>Endogenous</td>
</tr>
<tr>
<td>Collective code ownership</td>
<td>Endogenous</td>
</tr>
<tr>
<td>Continuous integration</td>
<td>Endogenous</td>
</tr>
<tr>
<td>40 hours week</td>
<td>Exogenous</td>
</tr>
<tr>
<td>On-site customer</td>
<td>Exogenous</td>
</tr>
<tr>
<td>Coding standards</td>
<td>Exogenous</td>
</tr>
</tbody>
</table>

In parallel to the control of processes, there is the organizational problem. Ouchi (1980) has identified three major mechanisms, which depend on the ability to measure the output and the knowledge available on the processes (Figure 9):

- **Behavioral**: used when it is clear and transparent the operations needed to produce the specified output. This is typical of clerical work.
- **Outcome**: used when it is possible to measure the amount and/or the quality of the output. This is typical in simple professional or technological tasks, which might also be outsourced.
- **Clan**: used when neither the process is clear nor the output is easy to measure. This is the typical situation in the most complex knowledge-based works, where the final evaluation of outcome of the work can be done only after a while.

![Figure 9: Organization and control types](image)

The understanding of the development process is higher in traditional, plan-based development techniques where there are formal definitions of tasks, procedures, and roles. For instance, in the waterfall model every phase of development has a very clear output and procedure to go from its input to its output (it is a different issue in which circumstances such approach is effective).

AMs acknowledge the difficulty in understanding the software development process and in measuring its output. Therefore, they prefer to oversight resources with a clan approach. This is clear in the Agile Manifesto, where “collaboration” is considered more important than “negotiation” and “interaction” more than “processes”.
5. Tools support for XP

There are several software tools to support XP development. However, many XP teams prefer low-tech and traditional tools such as paper, pencil, pin board, and face-to-face meetings.

For instance, in XP, user stories are written on small pieces of paper with the size of a postcard and hang on a pin board. The pin board is divided in three sections: user stories to be implemented, user stories under implementation, and user stories completed. This layout provides a visual representation of the project status.

Even if many XP teams do not use software tools, some of them are useful. Among these, there are standard applications not designed to support XP and ad-hoc applications developed specifically to support it.

Among the general purpose tools there are:

- **UML modeling tools**: such tools are used in two ways:
  1. to write a high level description of the application;
  2. to reverse engineer the code to create documentation.
- **Requirements negotiation tools**: this kind of tools helps developers and customer to identify, prioritize, and manage requirements in different environments.
- **Instant messaging tools**: these tools are useful to keep in touch with the customer to discuss requirements when he is not on-site or to exchange information among developers when they cannot sit at the same desk.
- **Version control systems**: these tools help developers to trace changes in the code and retrieve a working version of the code in case a modification is not working properly.
- **Automated testing tools**: continuous testing of the code is one of the key aspects of XP. Running automated unit and integration tests very frequently is of paramount importance to identify defects in the code as soon as they are introduced.

Among ad-hoc applications there are:

- **Project management tools**: such tools focus on specific practices used in XP and helps to store and retrieve user stories in an electronic format, organize iterations, etc.

6. Conclusions

AMs are the implementation of the basic concepts of the Lean Management in the software industry. The focus on the continuous improvement of the development process and on the customer satisfaction are two key components for their success. Such aspects are present in all the practices that developers apply every day.

The goal of AMs and XP in particular is to deliver high quality software on time and on budget focusing on what is valuable for the customer, create a continuous feedback channel with him, and supporting changes that he may require. However, these methodologies do not pretend to be useful for any kind of software project and for any kind of organization. Like any other development techniques, AMs have specific areas in which they perform well and areas in which other techniques are better. Nevertheless, since these techniques are quite recent, the identification of such areas is still an ongoing activity.

References


