AGILE PROJECTS

CEFORA – Training 266 (ex SIN-81)

Abstract

Discover agile project management through a focus on eXtreme Programming

Pierre-Emmanuel Dautreppe & Norman Deschauwer

pedautreppe@pyxis-tech.com & ndeschauwer@pyxis-tech.com

1. Who are we?

1.1. Pierre-Emmanuel Dautreppe

Pierre-Emmanuel is a .NET specialist since 2002 and currently work as technical expert and agile architect in .NET projects.

He went to the agile methodologies (more specifically eXtreme Programming) in 2005. As he started to work in a team that was not used to Agile, he could see a lot of anti-patterns practices by teams coming from the Waterfall world.
Since this day, he shares his experience and recipe trying to bring agile values and culture.

He gives conferences and trainings both on .NET technology and agile methodology.
In 2009, he founded the association DotNetHub (http://www.dotnethub.be) to promote .NET and Agile and created the yearly event La Journée Agile (http://www.journeeagile.be).
In 2014, he founded Pyxis Technologies Belgium (http://www.pyxis-tech.com).

1.2. Norman Deschauwer

Norman is working as business analyst and team lead.
He went to the agile methodologies in 2007, and since then, never stopped trying to get better doing it. His current goal is to share to the agile principles, and to change the most resistant minds: Agile methodologies are not dedicated to the IT sector.

In 2009, he founded the association DotNetHub (http://www.dotnethub.be) to promote .NET and Agile and created the yearly event La Journée Agile (http://www.journeeagile.be).
In 2014, he founded Pyxis Technologies Belgium (http://www.pyxis-tech.com).

2. Introduction

Agility usually raises antagonist and paradoxical feelings. The client gets afraid, thinking he would lost any control on his project, even thinking he would sign a blank check. Or, when he is uncertain about his project content, he might see agility as the miracle recipe that will allow him to change anything at any time.
The developer – that usually likes working alone, and might be egocentric or proud – is sometimes not prone to using some practices agile brings with it. Or, on the opposite side, can become motivated as never, seeing the perfect occasion to bring dynamism in a job that he got bored of.

Agile **is absolutely not** a silver bullet or a miracle recipe, and in some cases, it won't bring any benefit. This can come from a lot of reasons as the team members (including the client), or the context for instance.
But many projects have to learn from agile practices, and the setup of such a methodology can make them reach the very private club of successful projects[1].
It can also bring for every team member a real cohesion and communication that they would not have imagined possible.

Through this document, we want to introduce agile methodologies, starting from pure theory, and to share our vision of agile through our experience.
We are pretty convinced that number of projects could gain experience with agile, and – with this presentation – we hope letting you feel which benefits you could get.

We will introduce agile under the magnifying loop of XP (eXtreme Programming) as this is one of the most complete agile methodology we can meet.

[1] The « Chaos Report » (1994) of the Standish Group states that only 16% of IT projects can be considered as successful. 31% are stopped before they are finished, and 52% come to end with number of overtime or over-budget.

3. Where does it comes from?

In early 60, Toyota gives birth to agile – also called the "Toyota way".
For the engineer Taiichi Ōno, it is necessary to reinvent the work organization, with a single goal: move Toyota to the world leader car factory. And they made it!

To achieve this, he gives numbers of advices and practices, as reducing waste, raising up quality by detecting defects along the whole process, minimalizing stock, putting everyone around the problem resolution, and much more.
He also propose tools like KANBAN, a name now common to the agile world.
The hierarchy as it existed need also to be reviewed.

In a nutshell, the "Toyota Way" is the basis of the agile methodologies we know today.
Scrum, eXtreme Programming, RAD, Crystal, PUMA, SaFE… are all methodologies that will appear much later.

Agility in the IT world will come to a decisive switch in 2001 with the signing of the "Agile Manifesto": a kind of charter, or agreement, letting away the traditional Waterfall system which is – they think – outdated and has proven its inefficiency.
From this manifesto, came out the values and practices that eXtreme Programming states as its foundation.
Thru the rest of this document, we'll explain in a few words these values and practices.

4. Presentation of eXtreme Programming

4.1. The five values

XP recognize five values that might be seen as a dogma, the way you should behave in your team so an agile project can be run through efficiently.

4.1.1. Communication

Communication is probably the most important value to be followed in an agile team[2]. Communication is omnipresent, and can be seen in different practices to be set up all the process long.
It even need to be taken into account in the team room and disposition so every teammate can easily speak to each other and to the client.
The team communication will be eased in an environment like an "Open Space".

4.1.2. Feedback

Feedback is also a key notion in agile culture, and this for all team members.
Let's take some examples:
As a developer, it is quite obvious that the sooner we meet a bug (because we have done a coding mistake), the quicker (and cheaper) it will be to correct: the implementation is still "fresh", still in the developer's mind, and had not caused any damage or a disruption in production.
For a team leader or manager, the sooner he is aware of a problem occurring in the team (for instance some delays in development planning, or a gap between implementation and the client expectations), the sooner he will be able to take some decisions and apply corrections, so the consequences will be limited.

So feedback will be met of course thru transparent communication, but also thru tools like Kanban, burndown chart, or continuous integration.

4.1.3. Courage

The courage will be needed at many times. An agile team must be reactive to changes and difficulties. Si you will need to have courage, for instance:

- To accept the feedback and thus to alert about difficulties or delays that we might cope with during our task
- To review completely an architecture because it is not resilient toward a new functionality
- To put oneself constantly in question, to put one's work in question so it can be improved and simplified

[2] When we speak about the agile team, this includes every person involved in the project: a developer, a tester, a team lead, the client…

4.1.4. Simplicity

Once again, you must be ready to face the change, and so to update your code.
Every developer will agree that it is much simpler and quicker to make evolve a simple code than a complex one, because it can be better understood.

When look at implementation, this also means that you should not anticipate probable future functionalities, just simply because there won't come up, or not in the manner we were thinking.
On the other hand, you should be very careful not to confound "simple" and "simplistic".

4.1.5. Respect

We all work in the same team. Thus it is highly important to respect our own work as our colleague work. The values of communication, feedback, courage and simplicity, will naturally imply respect toward the person that are putting them as real behavior.
That person shall of course follow some practices and rules to respect each other's work, like for instance: never "committing" any source code that breaks the compilation, or in a more general manner slow down his colleague work.

It is also important that anyone in the team is considered equally, otherwise some internal tension might occur, and will inevitably generates motivation or productivity slow down.

To keep simple things short, keep in mind: « Team First »!

4.2. The thirteen practices

4.2.1. Whole Team

Agile development breaks the general idea of a tenderer or provider doing a project for a client. We will now speak of a team – a whole team – composed of developers, analysts … including the client (or a client representative).

It is indeed highly important that someone from the client – ideally close to the final users – could be present aside the development team. Or to see it with another perspective, that the development team is on the client site.

This will allow a good communication among all project members, and will ease the broadcast of final users' requirements and needs.

4.2.2. Planning Game

The "planning game" is a key event in the eXtreme Programming.
This event occurs normally once every iteration, and will be attended by any project's team member (client, team leader, developer…).

During the planning game:

- The 'client' will explain the functionalities to add or improve
- He might also define or share the "business value" of those functionalities

- Developers will discuss or debate on those tasks, their goal being to ensure that anyone correctly understand the need, and will propose technical solutions, and their cost
- With that cost estimation and the business value, the client will be able to determine the priority of the different tasks, and so determine the “iteration backlog” (ie the “to-be-done tasks basket”) for the next iteration

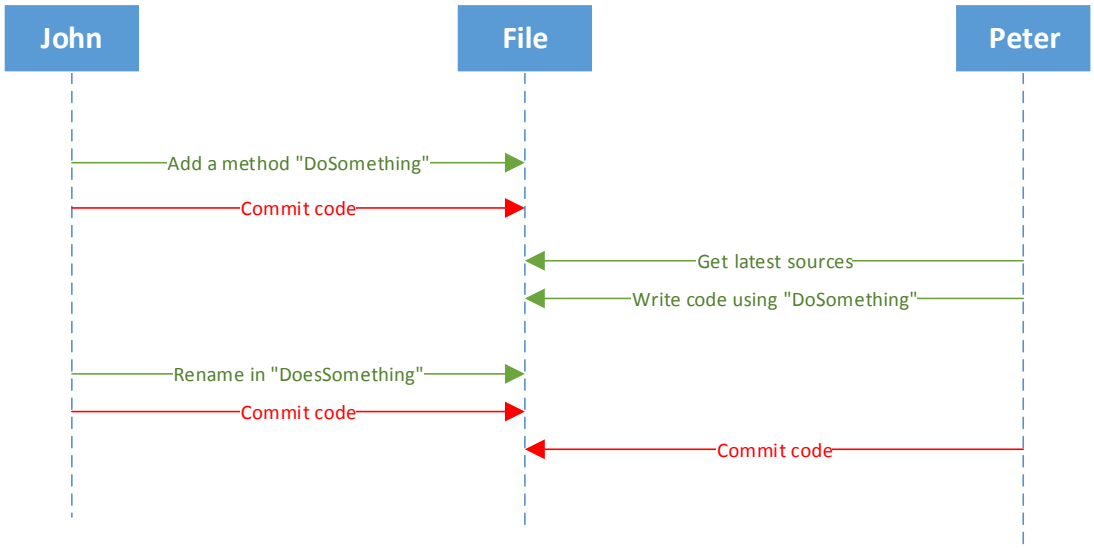
That planning game can be organized in many different ways. The most known one is called the “planning poker” where anyone will vote for a task cost using “poker card”.

Do remember that this cost is “time agnostic”. We usually speak of “story points”, which represent a relative complexity. To rephrase it, an “8 task” will be twice more “complex” than a “4 task”.
Usually we are using the numbers coming from the Fibonacci suite, meaning 1, 2, 3, 5, 8, 13, 21… A non-linear scale is used to emphasize that estimation are uncertain: the more complex a task is, the less exact the estimation will be.

But why using “story points” and not classical time estimation?
This has two main benefits. In the relation between the client and the development team, it allows reducing any tension that might arise related to “nonproductive time”, meaning administrative time (meetings) or low/no-quality time (bug resolution). If we can produce tasks estimated as “30 hours” in 40 available hours, some awkwardness can occur.
But more than that, it allows having a constant scale along the project. Indeed whatever the level or seniority of developers, the relative complexity will remain the same. But the velocity (meaning the quantity of tasks that can be done) will evolve with time.

4.2.3. Continuous Integration

When we work in a development team, it is very simple (and very quick) to create code that does not compile.



Continuous Integration will increase the level of feedback, and will put under the light the quickest possible this kind of problem.

As stated by its name, « continuous integration » allow integrating permanently (meaning quickly and in background) the source code of all developers so they can validate their integrity and compliance towards client requirements.

This validation implies to have a process that will

- Get the source code of all developers
- Compile the sources
- Validate the compliance of these sources, meaning for instance
    - Executing some code analysis (checking syntax, naming, performance… rules)
    - Executing validation tests (unit tests, acceptance tests, load tests…)
- Deploying the sources so that the product can be downloaded or tested by an external team

The main objective of continuous integration is to increase the feedback for:

- Developers that can check their code is compiling and that they haven't created any regression in any already implemented functionality
- Team Lead or client that can get a metric about the project implementation progress (for instance comparing the number of implemented tests against the ones remaining to implement)

Continuous integration is not only a source control tool, it is also a discipline! A discipline not to remain five days without pushing one's source code update, or never pushing any bugged code.

4.2.4. System Metaphor

Usually the developer place himself in a technical role, and thus produce… technical code. But he is developing a business application, and so his code shall leak out business.

This is where the metaphor has a role to play. When, inside of a group, people sometimes speak of "Developer" or "Employee" or "Person" or "Resource", they have all the chances not to understand each other.

It is important to define a common vocabulary between analyst, developer and client so the requirements can be correctly explained and transmitted to all. And of course this is the same vocabulary that we should use at source code level (naming of objects, attributes or methods, comments…).

Note also that usually the source code will include a lot of business roles (as *payment shall be done the later for the 15 of the month following the purchase*). Those rules should be documented, and ideally be included in explicit methods or properties (*for instance a property named PaymentLimitDateBeforeReminder could encapsulate this rule*).

4.2.5. Unit Tests

4.2.5.1. What is this?

There is a lot of different kind of tests, but all have a single goal: validate the coherence of the source code against the client specifications.

Unit tests are generally thought and written by developers, in order to validate the behavior of a small code unit: a class or a method for instance.
The idea of "unit" is important: the more concepts (objects) we will use inside a single test, the less the test failure can be analyzed (and so corrected) quickly (what is the object / method responsible for the failure?). Thus we will try to test the smallest possible unit of code, and to isolate them from dependencies.

There is always a debate inside the community to state whether we should only test the public API or also the private methods. Each solution has its own benefits and disadvantages, and each team need to decide its own rules.
Limiting oneself to the public API is logic as those are the objects / methods that will be used by our code clients. So we check the behavior of what our clients will be able to do.
But of course, behind a public API we can find numerous classes or private methods. Testing those ones can lead us to do more technical but also more unit testing.

4.2.5.2. Tests and isolation

« Isolation » means removing external constraints and dependencies that could lead to a slow or unpredictable test. This is typically when we are dependent from a file system, a database, a web service…
When we test such a code, we can isolate it by using different coding techniques like mocks or fakes.
Shall we test this code, dependent of an external system? Yes, but this is another kind of test, usually called "Integration Test".
The main difference is that integration tests are more demanding. They are generally more complex to write and they will also be slower to execute.

During the process of continuous integration, we will execute the different application tests. As we were saying previously, it is important that this continuous integration remains quick so that feedback to the team is the more pertinent possible.
Paul Julius[3] has introduced a metric call « Cup of Coffee ». This empirical value corresponds to the level of information (meaning feedback) a developer can receive while he leaves his desk to go for a cup a coffee. It is clearly evident that if a developer can even not know that his code compiles during this small duration, so the risk to have an unstable / incoherent source code will increase.
So it may be quite interesting to categorize the different tests to be able to choose the ones that will get executed during the continuous integration. We can so imagine:

- All unit tests will be executed at each commit (or every few minutes)
- Other tests, the slower ones, could be executed once or twice a day (during a nightly build for instance)

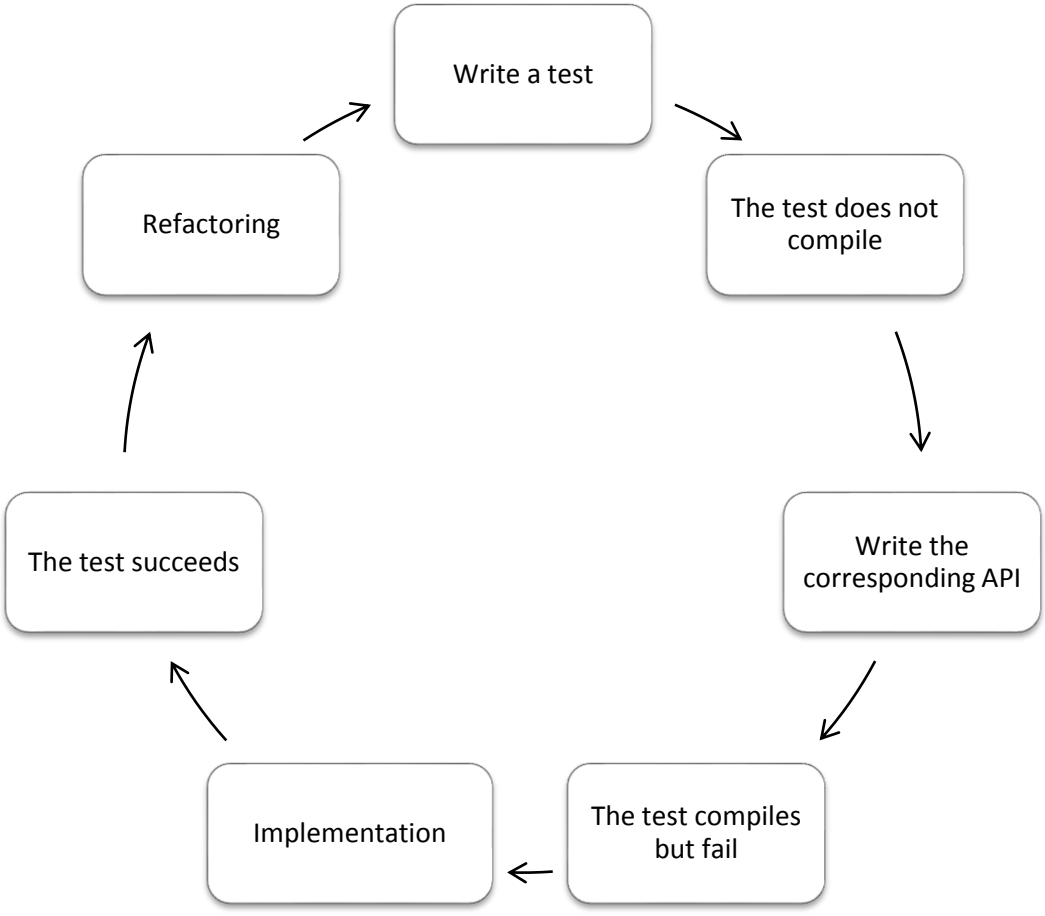
[3] Co-Founder with Jeffrey Fredrick of the OIF (Open Information Foundation)

4.2.5.3. The TDD cycle – Test Driven Development

Agility puts the focus on TDD, or Test Driven Development.
More concretely, this means that any development activity should start with a test, and once this test gets written, to implement the corresponding code.

Why should we work in that way?
This allow us to specify the functionality we want to implement in a test, and to focus only on that functionality implementation, no more, no less.
This also allow us to use testing as a real design tool, and not only as a verification tool.

It is also quite important to include in this cycle a refactoring phase (check here the corresponding practice), but never to modify both the code and the tests.
We can refactor the code, using testing as a security net, and then refactor the tests, using the code as a security net.



4.2.6. Acceptance Tests

The "acceptance tests" are another type of test, focusing on business.
They are normally defined by the client (or with the client) and are higher level tests, translating into test code the client requirements.

It can be classic user interaction (when going on that screen, then I expect to see such a result), or for instance more "non-business requirements" (I need to have an answer in less than 300ms). If the test succeeds, then this mean that the client requirement (or part of that requirement) is correct. Thus the delivery of that development could generate an official acceptance (and thus a payment).

Those tests can be used as a good progress indicator for the project itself or for a functionality: for the X requirement to be done, we still have Y acceptance test to implement.

Those tests can also be used a good business documentation as they describe, with a business or end-user point-of-view the behavior of the application. It is so important to have them readable and so usable by the business.

4.2.7. Small releases

Even if a client describe very precisely what he wants, generally speaking, it is when he will use his application or requirement that he will think to improvements or simplification to bring into the application.

On the other hand, a huge problem in IT is what we call the "tunnel effect": the more we wait before showing what we are doing, the higher is the risk that we have diverged from the client expectations.

Consequently, this practice has a clear objective: being able to deliver a functional product at an early stage so that the client can use it and can make his remarks and suggestions the soonest possible.
This practice allows a reducing in development cost, while maximizing the client satisfaction.

Another characteristic of the tunnel effect is that the client needs will change depending of external parameters. When a financial crisis starts for instance, we can easily think that the most urgent and high-priority requirements will change.
Rapid delivery allows us to increase the feedback towards and from the client, and so to be able to accept, more rapidly his new priority.

Indeed it must not be forgotten that each new delivery shall lead to a functional product. Of course, this product won't be complete, but what is delivered is functional.

4.2.8. Sustainable Pace

The idea of a sustainable pace is quite simple: take the best possible team, and ask them to work 12h a day, during 5, 10 or 20 consecutive days. Unavoidably, you will see a huge decrease either in quality or in velocity (or both), and the number of bugs will increase.

The concept of sustainable pace is quite clear: without forbidding any kind of extra hour (each project – agile or not – will face moments of tensions with time constraints), we will try to limit them. To achieve this, we can put in place some automation tools, or speak with the team (client included) to define business priorities between the desired requirements, and the available time.

The objective is quite simple: keeping a productive and focused team. And do not forget: Team First!

4.2.9. Simple Design

This practice comes directly from the value "Simplicity". As we told, we must be ready to embrace change, and quite obviously a "simple" code will evolve more easily (quickly, cheaper) than a complex one.

In agile, we will often speak of "emergent design". This practice consists of defining design on the fly, during development, and not in big design phase before the project start, with complex and complete UML design, as we would do in a more traditional project.
What we can see indeed, is that the original analysis phase will generally lead to an architecture that will search to answer to any possible case, present, or future. And so also to handle requirements that have not been asked by the client.
This over-complexity in diagrams will, of course, being translated in an over-complexity in code.
Moreover this design phase is generally quite long because the architects will try to plan and foresee anything. But in reality, planning the future change is quite difficult, and so of course, diagrams will be change during the project development.

On the opposite side, a development with an "emergent design" will allow writing the code which is strictly necessary (no more, no less), and the simplest possible.
However, this notion of "simplicity" is very important to be understood correctly and the more junior development teams can easily be misled.

We will speak of **simple code, but not simplistic**, meaning code to be simple to make evolve.
For instance, we'll prefer writing a class inheritance and using polymorphism, instead of using an enumerated type and a series of conditions.
Inheritance will be more "complex" to implement, but regarding to application evolution, this code will be "simpler".

4.2.10. Refactoring

Refactoring is a very important activity in agile development. To name it simply, it consists in "cleaning" ones code.
It can be done before starting a task, or after. Before a task, it will let us keep an iso-functional code (tests are so a safety net, guarantying that the code still respond to the functional requirements), and to simplify it in order to have a more evolvable code, to ease the integration of the new task.
After the implement, it is like our own code review, in order to improve what we have done, to bring it in an evolvable state.

Agility will emphasize this practice as a day-to-day activity (we will speak of continuous refactoring) in order not to accumulate any technical debt.
Technical Debt can be seen as the quantity of "late" work in order to have a "clean" code (meaning simple, auto-documented, evolvable…). The more important is this technical debt, the more difficult it will be to reimburse.
This will lead to planning "technical cleaning work", which unique goal will be to clean this debt. Those work start to be mandatory after a while, just to be able to implement the new requirements, but their cost is quite high and moreover, won't bring any functional benefit (meaning no added-value to the client).

Refactoring implies to be critical towards our own code, and can be simplified by pair-programming, or code review.

4.2.11. Code collective ownership

In general, in a given project, each developer will have its own competency field, either in a technical fields (he is a DB expert for instance), but also for the business field (he is the expert of the data import in the module X).
This natural specialization is highly dangerous in a project and to name it, we can use the "bus factor". This factor is equal to the number of person that need to be killed during a bus accident to seriously jeopardize your project. Of course the closest this factor is to "1", and the more your project is at risk.
To speak of less tragically circumstances, if only a single developer know a part of your code, what happen when we need to modify it (because of a bug, or client need) when this developer is ill, or in holiday, or when he quit?

To counter this factor, there is a simple practice: « code collective ownership ». To rephrase it, "code belongs to anyone", or any developer in your team must be able to modify any part of your project.
This is hugely simplified by two other practices: setting up "naming / coding conventions" and "pair-programming".

You need to create an environment where any developer can work or any part of the application, and try to reduce the "expert effect". In some teams, "technical referent", will naturally rise, either for their experience or charism, or because of the inexperience or reserved of others.
This might lead to a situation where some developers won't touch some others' code, by "fear" or "respect". This leads to a dangerous situation, by creating specialists.

4.2.12. Coding Standards

In continuity of the "code collective ownership", in order to have any developer to be able – or even invited – to modify any part of the code, each developer must feel at-ease in front of any part of the code.
And to achieve this, there is nothing more efficient than being in front of a code that could be ours, by its formatting, the way the variable are named, or the way it is organized.
Thus you should define naming rules, coding rules, or formatting rule on the project so that it is developed consistently, whatever the developer that works in it.

4.2.13. Pair-Programming

It might be the practice the more difficult to make accept by your team.
Pair-programming is simply setting a pair of developers, to work together on a single computer, on a single task. This means: one computer, one keyboard, one mouse, and two person.

These two person must be of course at-ease, either for coding, or to look at screen. We will privilege the usage of straight tables, and maybe of two replicated screens so that the comfort of both developers is maximal.

"Two person on a single PC? But that will twice the double!"

This is probably the most frequent criticism you can hear when speaking about pair programming. Well, it is not so expansive. Keep in mind the consequences of such a practice:

- Each developer has its own technical domain of expertise (one will work more on the web service design, the other more on the database part…) Working in pair will establish quickly a leveling by the top of the team knowledge
- Each developer has its own business domain of expertise. Pairing will bring a continuous sharing of each understanding.
- A good practice in a project is to put in place code reviews to check either the code quality, but also code homogeneity compared to existing code. With pairs, that code review will occur at any time! Code quality will naturally increase
- In the same way, the number of bugs will decrease as two person will permanently write and check the produced code
- It is also known that the level of concentration can vary a lot in time (phone, reading or writing mail, facebook, twitter, youtube…) Those activities are of course present in the coder daily life. Pairing will drastically reduce those activities. Of course that focus increase will come together with a tiredness increase. When setting up pair programming, do not forget the sustainable pace!

All those elements will allow an increase in developer productivity and will reduce the costs related to non-quality.
But it is also important to remember some basic rules:

- Pairing is not a wedding, and we will try to make them change regularly
- A pair is the assembly of a pilot (the one with the keyboard) and a copilot
- The role of the copilot is highly important. He is in charge, for instance, of the constant code review, of anticipating future bugs and refactorings. He is also there to give development tricks and give directions for the emergent design
- The roles of pilot and copilot switch regularly when the developers ask to

5. XPGAME

5.1. Description of the XP Game

The clients, analysts, developers and testers have often difficulties to communicate and understand each other. eXtreme Programming will help them to reconcile, but suffers of a bad image : a tool / methodology dedicated to IT people.
The “XP Game” propose, in a ludic way, to focus on what it the work in an agile team. Each one will offer his competencies to the team which have a single goal: succeeding in the project. Any profile is admitted, no technical knowledge is needed.
At the end of the game, everyone will have learned how user stories, estimations, planning, implementation and functional testing are used.
All will have understood how estimation can influence planning.
Developers and clients will have learned to know each other and to respect.

Source: this game has been developed by Vera Peter, Pascal Van Cauwenberghe and Portia Tung
More information on http://www.xp.be/xpgame/