An Experience Report on Implementing a Custom Agile Methodology on a C++/Python Project

Giovanni Asproni, aspro@acm.org, http://www.GiovanniAsproni.com
Alexander Fedotov, fedotov@ebi.ac.uk
Rodrigo Fernandez, rodrigo.fernandez@bnpparibas.com

Introduction

In this article we'll describe our experience in implementing an agile methodology in a C++/Python project.

The promise of agile methodologies is to make software development faster and, at the same time, have higher code quality, higher customer satisfaction, and happier programmers. This is very appealing! However, their implementation is not easy, especially when it is the first one in the organisation, and the main language used is C++ ;-)

In fact, introducing a new methodology—and more generally, any kind of change—in an organisation may create several technical, human, and political problems. We'll describe the choices we made in order to minimise their occurrence, and how we managed to solve the ones we faced.

We'll start by briefly introducing the project. Then, we'll explain what agile software development means and why we decided to use an agile approach.

We'll describe the methodology with the rationale behind it, and we'll show the reasons why we decided to not pick an out of the box one like Extreme Programming [3].

Finally, we'll give an assessment of the results of our approach, including what worked well and what we would do differently the next time in similar circumstances.

Project Overview

The project consisted of the development of a family of C++ and Python programs used to produce, store, manage and distribute biological data to and from an Oracle database. These were developed for internal use.

The purpose was to substitute a family of old C programs that used text files as a storage medium. During the years the amount of data had increased so much that the usage of text files became a serious bottleneck in the data production process—the usage of files, and the way the programs were written, forced a serial process that was becoming too time consuming.

The usage of a relational database seemed the perfect solution to this: it could allow the execution of several production tasks at the same time, having the certainty that the data was kept consistent thanks to the isolation and synchronisation given by the transactional model. Implementing transactions in the C programs in order to keep the data in text files, would have been a very difficult task, and an obvious waste of time.

Six years before the three of us started to work on the project, some people already understood the limitations of those C programs, and started to work on their substitutes: a family of C++ programs that used an Oracle database (Python was something the three of us decided to introduce). During those years, the team size varied from a minimum of one to a maximum of five programmers. As far as we know, not all of them worked always full-time on the project.

When we were assigned to the project, we inherited a code base of about fifty thousand lines of C++ code that had not being put into production yet. At that point, the project had become of critical importance to the business, and our task was to make it production ready in about one year—this

meant fixing the bugs as well as implementing several new requirements.

When we started our work, the programming team was composed entirely of the three of us. None of us had been involved in the project before—all members of the original team either left the company, or were assigned to other tasks. From time to time, when some specialists joined the team, its size increased to four or five, to go back to three after a while.

Here is where our story begins.

The Beginnings

Unfortunately, the code we inherited was not in a very good shape—it was very difficult to modify and extend, there was not a single test and it was bug-ridden. Furthermore, there were a big number of (very volatile) new requirements to implement.

However, at first, we tried to improve it—part of the code-base was shared with another project that was maintained by a team with members in our office and in a room nearby, so we had the opportunity to ask questions and understand the code better. We added some tests, refactored some parts, and rewrote some others. But, after two months spent doing this, we realised that we were not making any real progress: the code was too tightly coupled in totally unpredictable ways. Fixing a bug caused others to show up from nowhere—and the number and the volatility of new requirements caused even more trouble. Furthermore, the code shared with the other project was in common for the wrong reasons—trying to exploit commonality where there was none—and that caused maintenance problems for the other team as well.

We analysed the situation, and decided that rewriting everything from scratch was a much better option: the basic logic of the programs was simple, and one year, if well used, was more than enough.

We had a few problems to solve though

- Convince our boss
- Convince our customer. This was the person that had the ultimate responsibility on the requirements, and also one of the users of the programs
- Organise our activities properly

To solve these problems, we decided to use an agile methodology. Before explaining why, we'll summarise what Agile Development is about.

Agile Software Development

> "Agility is more attitude than process, more environment than methodology." Jim Highsmith [9]

The term Agile Software Development is a container that includes all the methodologies that share the values and principles stated on the *Agile Manifesto* (see sidebar). The main characteristic of all these methodologies is that they are *people-centric*—i.e., they emphasise teamwork, face-to-face communication, and short feedback-loops. Furthermore, the processes and tools used have to be suited to the needs of the people involved: they may differ for specific methods, but the general recommendation in the agile community is to use the simplest ones that help in solving the problem at hand, and dismiss or substitute them as soon as they are not useful anymore.

That said, the following practices are becoming de-facto standards in all agile methods (and most of them are standard also on several non-agile methods as well)

- Extensive unit-testing

- Test-driven development (i.e., write the tests before the code)
- Short iterations (never longer than two months, usually one or two weeks long)
- Merciless refactoring
- Continuous integration
- Configuration management
- Automatic Acceptance testing

Agile methodologies are especially suited for projects with highly volatile requirements, tight schedules and up to 10-12 developers. In fact, with more people involved, communication may become difficult. However some techniques to apply them to large projects are starting to appear [8].

The most important goal that this kind of methodologies tries to achieve is to deliver value to *all* the stakeholders of the project, including the *developers*.

The value delivered to the customer may be obvious: software that is fit for its intended purpose on time and within budget.

The value delivered to the developers is much less obvious: job satisfaction and self-motivation. In fact, the usage of an agile methodology can be a big help in increasing and keeping developers' motivation [2], and, more interestingly, self-motivation is the most important factor influencing developers' productivity [5]. An important consequence of these facts is that, delivering value to developers has a direct positive impact on the value delivered to the customer.

To learn more about agile development, a good starting point is the AgileAlliance web site [10].

Agile Manifesto

The Agile Manifesto has been taken from [4]

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

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

**That is, while there is value in the items on
the right, we value the items on the left more.**

Principles behind the Agile Manifesto

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective methodology of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity—the art of maximising the amount of work not done—is essential.

The best architectures, requirements, and designs emerge from self-organising teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Why an Agile Methodology

At this point, many of the reasons why we decided to use an agile methodology should be clear.

First of all, we wanted to organise our development activities properly making the most out of the time and resources available. Our company didn't mandate the use of any particular methodology so we had total freedom of choice.

We wanted a methodology that was able to cope with unstable requirements. In our case, they were changing very often—sometimes from one day to the next—and we wanted to be able to respond to

these changes as smoothly as possible.

We wanted to have early results and short feedback loops, so we could demonstrate our progress early and often.

We wanted to write high quality software with a clear architecture, no code duplication, and fully tested.

Each of us wanted to be involved in every development activity, from discussing the requirements to coding the solution.

We wanted to have fun, to learn new things, and to write code we could be proud of.

We didn't want to have too much overhead: we were on a tight schedule, so we wanted to work only on what was essential to achieve the goal.

The methodology had to be simple to implement but effective.

Finally, the methodology shouldn't get in to our way. It had to be natural to follow its practices.

Given these requirements, and the fact that one of us (Giovanni) had already experience with Extreme Programming and agile development, it seemed natural to use an agile methodology. However, we didn't choose one out of the box (e.g., Extreme Programming), because, doing that, there could have been the risk of focusing more on the methodology itself than on the goals of the project. What we did instead, was to follow a “methodology-per-project” approach [6]: we created our own one based on our needs and experience, in other terms we used our common sense.

The Methodology

Our methodology was based on some values: communication, courage, feedback, simplicity, humility, and trust. They were never stated explicitly as such—i.e., we never sat around a table to discuss them—but they were always present on the background.

We always strove to have open and honest communication among us and with the customer. We tried to make every assumption explicit—sometimes asking seemingly obvious questions—in order to avoid misunderstandings and wrong expectations.

We had the courage to take decisions and to accept accountability for them. We had the courage to never give up to pressure to deliver on unrealistic schedules—of course we always explained why, and tried to offer alternative solutions. We had also the courage to be honest, not hiding problems from the customer, or from our boss.

We had always the humility to recognise our own limitations, so we always listened to other people's opinions and advice, or asked for help when necessary.

We always looked for simple and clean solutions to the current problems. Our requirements were so unstable, that was pointless to try to predict the evolution of the system for a time period longer than one month.

We trusted each other; we knew that each of us had the skills and the motivation to make the project succeed.

The methodology itself was quite simple. Actually was easier to implement than to explain

- As every modern method (agile or not), it was iterative and incremental, with, usually, one-week long iterations. This was to give the customer—and us—a better feeling of progress and better control of the project. In fact the short feedback loop allowed her to change idea on some functionality without loosing too much development time, and allowed us to know how well we were doing
- Team meeting every day before starting our activities. The meetings were usually ten minutes

long. They were organised in a Scrum like fashion [11] [12]: everybody said what he did the day before, what problems he encountered, and what was going to do that day. Their purpose was to put everybody in synch with the others, and to decide how to solve the problems encountered along the way

- Metaphor. This was the very high level architecture of the system. Its purpose was to communicate with the customer, and to drive our development activities. It helped us in making all important design decisions. Every time we had a doubt on how to do something, the metaphor had a fundamental role in helping us finding a solution
- Coding standards. They were agreed at the beginning of the project. Their purpose was to allow us to guess the precise meaning of a name, to read code faster, and to give guidance on where to look for classes, functions, etc.
- Collective code ownership. Every member of the team had the power to change anything in the code to accommodate new requirements, or refactoring odd-looking code. Its main purpose was to avoid the typical problems created by “personal” code ownership, such as being dependent on a particular developer for the modifications in his or her parts of the system. It helped also in enforcing the coding standards.
- Collective accountability. If someone in the team made a mistake, the entire team would have been accountable for it in front of the customer and the boss. Its main purpose was to avoid scapegoating and finger pointing. Of course, in order to make this work, trust among us was essential
- Direct face-to-face communication was preferred to other forms of more indirect communication (i.e., written documentation, phone calls, e-mails, etc.). Its purpose was to make communication faster, and more precise
- Everybody was involved in all phases of development. Its purpose was to allow us to have fun, learn new things, and keep self-motivation high. It was also instrumental in improving the design of the system, since we always had brainstorming sessions before taking any important decision that could affect the architecture of the system
- Requirements prioritisation. Its purpose was twofold: to allow the customer to have the certainty of having the most important requirements implemented, and to give us a clear goal for each iteration
- Configuration management. Its purpose was to keep track of the modifications made to the system, rollback changes, and to be able to rebuild a specific version if we needed to. We managed the configuration of every artefact: source code, documentation, build scripts and also the external tools and libraries we used
- Extensive testing. Code was not finished until we had tests for it. Is purpose was to verify that the system worked as expected, and to allow us to make changes to the code with the confidence that we hadn't broken any existing functionality. As a side effect, it helped in improving our development speed as we proceeded because we had to spend less time hunting for bugs
- Merciless refactoring. Its main purpose was to keep the code clean and supple. One of its side effects (in concert with extensive testing) was to allow us to develop a set of reusable libraries. These, in turn, allowed us to increase our speed substantially, since, as we went along, the development of new functionality became more assembling of reusable assets and less development of new code
- Automation of all repetitive tasks—and of code generation when possible. Repetitive tasks are boring and slow if done manually, and bored people make silly mistakes. Enough said

- Never, ever, compromise design or code quality for any reason. The expected lifetime of the code we produced was in the order of several years, so it had to be maintainable and extensible. In conjunction with refactoring and testing, it allowed us to go faster as we proceeded
- Modify the methodology if necessary. Its purpose was to not get stubbornly stuck to practices or tools that were not useful anymore. For example, some times we modified the length of an iteration to fit our needs

In order to make our methodology work properly, we had to do also some complementary things right from the beginning of the project

- Define very clearly the responsibilities of the different stakeholders. This was fundamental to reduce problems and finger pointing in case of problems
- Ask our boss to not put any of us on other teams working on other projects at the same time. This would have been a teamicide [7], would have lead to time fragmentation, and would have made estimating impossible
- Strong emphasis on teamwork. We asked the customer to address her requests to the team, not to individuals; for example, if she wanted to ask something by e-mail she had to address it to each of us
- During an iteration, the customer was not allowed to change the functionality being implemented. Its purpose was to avoid chaos, and had also the nice side effect of forcing the customer to think in more depth about her needs and priorities
- When possible, we used available code and libraries instead of writing our own—e.g., boost, CppUnit, Log4cplus, etc. This reduced development time, reduced the code base to maintain, and improved overall code quality
- Colocation in the same room, this had the purpose of increasing the communication bandwidth, and avoid any of us to be out of synch with the others—luckily we hadn't had to ask for this because we were already in the same office

From time to time we had to ask some “specialists” (i.e., Oracle database experts) for help. As far as we were concerned, they were akin to external consultants, and, since they worked on the project only for limited amounts of time, the fact that they were not located in our room was not a problem.

Every time we dealt with one of these specialists, we managed to learn enough from them so that, after a while, we could take over their tasks and carry on without their help. We acted as what Scott Ambler calls “generalizing specialists” [1]: each of us had some strong skills—e.g., C++ programming, or Object Oriented Design, etc.—but had also the ability and the will to learn and apply new ones.

We deliberately decided not to enforce the following practices used by Extreme Programming and other agile methods

- Pair programming
- Test driven development (TDD)
- Planning game and story cards
- Automated acceptance tests written in collaboration with the customer

Pair programming all the time could be tough. It is a very intense experience, and it requires a bit of time to get used to. Apart from that, we thought that, in our specific case, we could work faster without pairing. However, from time to time, we used this technique, especially for working on some particularly difficult tasks.

Test driven development does not come natural to everybody, and, since we wanted to implement the methodology in a way that was natural for each of us, we didn't mandate—or forbid—the usage of this practice. However, as time passed we started to use it more and more.

The planning game, story cards, and automated acceptance tests written with the user, were never used because we already introduced many changes, and we thought that introducing more could have shifted too much the attention from the product to the process, or overwhelm our customer. In fact she was very keen on manual acceptance testing, and also more traditional requirements gathering meetings.

To (partially) overcome the absence of automated acceptance tests, we wrote tests for high level functionality using CppUnit. This was less than ideal, but better than nothing.

Implementing the Methodology

> "People hate change...
> and that's because people hate change...
> I want to be sure that you get my point.
> People *really* hate change.
> They *really, really* do." Steve McMenamin as quoted in [7]

Introducing any kind of change in an organisation can be very difficult. Humans are creatures of habit and tend to despise change. Furthermore, some people may perceive a change as a threat to their position of power. This fact can cause several political and technical problems.

In our case, in order to minimise the impact of the potential problems we implemented our methodology with great discretion. We never mentioned it explicitly—actually, we didn't even give it a name—we only explained our practices to our boss and our customer presenting them as simple common sense. The reason for doing that was to minimise the chance of having endless (and useless) methodological discussions.

In order to make the project succeed, we had to perform four concurrent, but strongly connected, activities

- To manage the customer
- To manage the boss
- To carry out the technical tasks
- To have fun

These activities were all equally important. In fact, the failure of performing any of them could have made the project fail.

In the following paragraphs we'll show how our methodology helped us in carrying out these activities, leading to the successful completion of the project.

Managing the Customer

Managing the customer and her expectations was the most challenging and time-consuming activity. In many respects, she was a typical customer: she had the tendency to give us solutions, instead of explaining the problems; she had the very common "everything has top priority" mentality and, finally, we had to buy her trust.

All practices of our methodology had an important role in helping us to change her attitude, but face-to-face communication, incremental delivery, and short iterations were the ones that had a more immediate and direct effect.

Face to face communication is the fastest and more effective channel for exchanging information. Part of this information—that cannot be easily conveyed by other means of communication—is composed by the feelings, and emotions that are transmitted through gestures, and facial and body expressions. This last point is particularly important, for human beings are not entirely rational, and often make choices based on feelings or fears instead of logic or facts.

Being aware of that, we always presented facts to substantiate our choices, decisions, and proposals, but also put great care in being reassuring and in looking confident. This had been helpful in several occasions.

We preferred face to face communication to written documentation to the point that, usually, we answered our customer's e-mails by going to her office and talking to her directly. This kind of answer was much more powerful than answering by e-mail for a couple of reasons: ambiguity and misunderstandings could be dealt with much more easily, and showing up in person sent a clear message of commitment to the project.

Finally, it was of great help in keeping her focused on the business side of the project, was instrumental in eliminating ambiguities from the requirements, solve problems more easily, helping her in prioritising the requirements, and in avoiding implicit assumptions and expectations by making every aspect of the project explicit and visible.

Incremental delivery and short iterations helped greatly in buying her trust. In fact, she could verify early and often our real progress by running the programs delivered at the end of each iteration. Furthermore, she could make estimates on the completion dates based on real data.

After a while, she started to realise the advantages of all the other practices as well, and to see how they reinforced each other.

Unfortunately, not everything went as well as we wanted. A couple of times we had to discuss some technical requirements with her (and our boss), and accept to implement an inferior solution for political reasons. Of course, when we did that we always made them aware of the drawbacks in order to avoid finger pointing in case of problems.

Managing the Boss

Managing the boss was easier than managing the customer. As long as the customer was happy, he was happy as well.

Initially, he was dubious about our decision of starting from scratch, and also about our approach. To convince him to give us a try, Giovanni—the officially appointed technical leader—accepted to link his career advancements and salary increase to the successful outcome of the project. This apparently careless gamble was actually a calculated risk. We had enough experience and domain knowledge to know that the risk of failure was quite low.

In fact, after a few months of work, the results were so good that our boss started to promote our way of working around the company. He also convinced us to give some seminars about configuration management and testing to other development teams.

Managing the Technical Activities

During our purely technical tasks, e.g., design, coding, and testing, we had been able to apply the methodology without effort. In fact it felt very natural to work in that way.

Since the language used were C++ and Python, we hadn't the possibility of having any refactoring tool, but we never had any special problems for this reason. We managed to do merciless refactoring by hand quite well.

One of the main purposes of using Python was to reduce our coding effort. All C++ code used to

access the database was generated by a Python program we wrote for that purpose. We accessed all the tables with the same pattern, so we could generate the C++ code directly from the SQL used to create the database. This saved us a huge amount of time and reduced the occurrence of bugs.

As the project went on, we were able to increase our development speed, due to the number of tests we had, to the absence of duplication, and to the refactoring that allowed us to improve our architecture and create a set of reusable libraries.

This last point deserves some explanation. A very common approach to create reusable libraries is to work bottom-up: trying to write them as reusable artefacts right from the start. Unfortunately, this approach doesn't always give the results intended, because trying to predict all possible uses is very difficult. Our approach, instead, was “use before reuse”. When we had a specific problem to solve, the first thing we did was to look if we already solved a similar one. If we found it we would refactor the solution to an abstraction that could solve both of them; otherwise we wrote just the code for the specific case. Working in this way, we created several reusable libraries that helped in increasing our productivity quite a lot: we reached a point in which most programming tasks were mainly assembling components in our libraries.

The methodology was also instrumental in helping us solve some technical problems we had along the way.

Most of them were caused by our target operating system: Tru64 Digital Unix. Unfortunately, this wasn't a very well supported platform either by the software houses, or by the open source community. We had to write our own compilation scripts for boost, CppUnit, and other libraries. And the particular configuration of our hardware was less than optimal for developing software: as the code base grew, the compilation process became a painfully slow experience—from forty minutes to four hours, depending on the load of the machines.

To solve this we decided to port our code base under Linux and Windows—for which we had much faster machines that were also under our control—and compile under Digital Unix (usually once per day) only to run the tests when everything was already working on Windows and Linux.

Thanks to our extensive test suite, the full automation of the build process, and our obsession in keeping the code clean, the porting proved to be a painless experience. It had also a nice side effect: we ended up with truly portable software, with fewer defects. Some bugs that were missed from a compiler were almost certainly found by one of the others.

Having Fun

Fun is a fundamental part of work. It comprises all the things that make work something to look forward to. It comprises all the factors that motivate us in doing a job in the best possible way.

We decided, right from the start, that we wanted to really enjoy the project, and our methodology was strongly influenced by this decision.

Our strongest motivating factors (in no particular order) were

- Doing a quality job
- Delivering value
- Involvement
- Learn new things
- Having a challenge

The first three were directly addressed by our methodology.

The fourth factor was only partially addressed by it—i.e., learning a new approach to software

development. So, we made some very deliberate choices to learn even more: we explicitly decided to learn more about Oracle, Python, and some modern (but not arcane) C++ techniques, and also made some design experiments.

The fifth was actually created by our boss, and our customer, with their initial belief that we were going to fail.

It was OK to experiment and make mistakes—we had configuration management and lots of tests to protect us from their consequences.

We certainly managed to enjoy the project, and this showed up in the final product, and in the satisfaction of the customer.

Conclusion

The first production quality version of the programs was released in about nine months. It had a very low number of bugs and no memory leaks.

Before releasing to production, our customer decided that she wanted even more functionality implemented, so the first production version was released in sixteen months after we started to work on it. So far there has been only one bug—that was easily fixed—in more than five months of operation.

Compared with the system we had to maintain when we were assigned to the project, the new one had much more functionality implemented, the performances were five to ten times better than the previous one, and was much easier to maintain,

The last point deserves some more explanation. Maintainability is something that is usually difficult to achieve, and even more difficult to quantify. So how can we claim to have written maintainable code?

We certainly put great care in keeping the architecture simple and tidy, in localising responsibilities as much as possible, and in writing clean self-documenting and well-tested code. But, more importantly, soon after the production release, there have been several changes and additions in the implemented functionality that really put maintainability under test: so far, the implementation of each of them has been quite straightforward, causing, at most, very localised changes in the code-base.

Even if the project has been quite successful, there are some things that, with hindsight, we would do differently. In particular, we would use TDD right from the start; we would push to have automated acceptance tests written in conjunction with the customer; and we would try to fight harder to avoid political compromises for deciding on purely technical issues.

Manual acceptance testing had been the cause of many problems, being a very time consuming and error prone activity. Sometimes the customer simply hadn't the time to run her tests, and, because of that, we couldn't proceed to work on the next iteration. For this reason, about four months out of sixteen have been wasted. Furthermore, it was easy to make mistakes during the tests' execution: several times, the customer came to us claiming, wrongly, to have found a bug in the code, when what had really happened was that she had made an error in the testing procedure.

The few technical compromises we had to make for political reasons always caused the problems we forecasted, that regularly gave rise to long and painful discussions about the reasons of the problems with our boss and with the customer.

Finally, we would keep the possibility of dynamically adapting the methodology to the needs of the project. We are aware of the fact that there is no one-size-fits-all methodology, but, from our experience, we are convinced that, sometimes, a methodology cannot fit even a single project if it is not flexible enough to change along with the needs of the project.

We don't claim that our methodology can work for every team, or project—in particular, we don’t recommend anybody to put his or her career at stake as Giovanni did. However, we think that our approach can be used by other teams to develop their own methodology. In fact the most important assumption behind our choices (and behind the agile manifesto) was that the process and tools should fit the needs of the people involved in the project, not the other way round. In fact, happy programmers produce better software.

References

[1] Ambler, S., *Generalizing Specialists: Improving Your IT Career Skills*, http://www.agilemodeling.com/essays/generalizingSpecialists.htm
[2] Asproni, G., *Motivation, Teamwork, and Agile Development*, Agile Times Vol. 4, 2004 http://www.giovanniasproni.com/articles
[3] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison Wesley, 1999
[4] Beck, K., et al., *The Agile Manifesto*, http://www.agilemanifesto.org
[5] Boehm B. W., *Software Engineering Economics*, Prentice Hall, 1981
[6] Cockburn, A., *Agile Software Development*, Addison Wesley, 2002
[7] DeMarco, T., Lister, T., *Peopleware: Productive Projects and Teams*, Dorset House Publishing, 1999
[8] Eckstein, J., *Agile Software Development in the Large: Diving into the Deep*, Dorset House Publishing, 2004
[9] Highsmith, J., *Agile Project Management*, Addison Wesley, 2004
[10] N.A., *AgileAlliance*, http://www.agilealliance.org
[11] Schwaber, K., *Scrum*, http://www.controlchaos.com
[12] Schwaber, K., Beedle, M., *Agile Software Development with Scrum*, Prentice Hall, 2002