Software Development and Experimentation in an Academic Environment: The Gaudi Experience

Ralph-Johan Back, Luka Milovanov, and Ivan Porres

Turku Centre for Computer Science,
Åbo Akademi University, Department of Computer Science,
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
{backrj, lmilovan, iporres}@abo.fi

Abstract. In this article, we describe an approach to empirical software engineering based on a combined software factory and software laboratory. The software factory develops software required by an external customer while the software laboratory monitors and improves the processes and methods used in the factory. We have used this approach during a period of four years to define and evaluate a software process that combines practices from Extreme Programming with architectural design and documentation practices in order to find a balance between agility, maintainability and reliability.

1 Introduction

One of the main problems that hinders the research and improvement of various software construction techniques is the difficulty to perform significant experiments. Many processes and methods in software development have been conceived in the context of large industrial projects. However, in most cases, it is almost impossible to perform controlled experiments in an industrial setting due to resource constraints.

However, university researchers also meet with difficulties when experimenting with new software development ideas in practice. Performing an experiment in collaboration with the industry using newly untested software development methods can be risky for the industrial partner but also for the researcher, since the project can fail due to factors that cannot be controlled by the researcher. The obvious alternative is to perform software engineering experiments inside a research center in a controlled environment. Still, this approach has at least three important shortcomings.

First, it is possible that a synthetic development project arranged by a researcher does not reflect the conditions and constraints found in an actual software development project. This happens specially if there is no actual need for the software to be developed. Also, university experiments are quite often performed by students. Students are not necessarily less capable than employed software developers, but they must be trained and their programming experience and motivation in a project may vary. Finally, although there is no market pressure, a researcher often has very limited resources and therefore it is not always possible to execute large experiments.

These shortcomings disappear if the software built in an experiment is an actual software product that is needed by one or more customers that will define the product requirements and will carry the cost of the development of the product. In our case, we found such customer in our own environment: other researchers that need software to be built to demonstrate and validate their research work. This scientific software does not necessarily need to be related to our work in software engineering.

In this paper we describe our experiences following this approach: how we created Gaudi, our own laboratory for experimental software engineering, and how we studied software development in practice while building software in Gaudi for other researchers. This experience is based on experiments conducted during the last four years. The objective of these experiments was to find and document software best practices in a software process that focus on product quality and project agility.

As we proposed in [1], we chose Extreme Programming [2] (XP) as the framework process for these experiments. Extreme Programming is an agile software methodology that was introduced by Beck in 2000. It is characterized by a short iteration cycle, integration of the design and implementation phases, continuous refactoring supported by extensive unit testing, onsite customer, promoting team communication and pair programming. XP has become quite popular these days, but it has also been criticized for lack of concrete evidences of success [3].

This paper is structured as follows: in Section 2 we describe the Gaudi Software Factory as a university unit for building software in the form of controlled experiments. Section 3 present the typical settings of such experiments and portrays their technical aspects. Section 4 discusses the practices of the software process, while Section 5 summarizes our observations from our experience in Gaudi. Due to space limitations, this article focuses on presenting the qualitative evaluation of these experiments. The reader can find more detailed information about the actual experiments in [5].

2 Gaudi and Its Working Principles

Gaudi is a research project that aims at developing and testing new software development methods in a realistic setting. We are interested in the time, cost, quality, and quantitative aspects of developing software, and study these issues in a series of controlled experiments. We focus on lightweight or agile software processes. Gaudi is divided into a software factory and a software laboratory.

2.1 Software Factory

The goal of the *Gaudi software factory* is to produce software for the needs of various research projects in our university. Software is built in the factory according to the requirements given by the project stakeholders. These stakeholders also provided the required resources to carry out the project. A characteristic of the factory is that the developers are students. However, programming in Gaudi is not a part of their studies, and the students get no credits for participating in Gaudi – they are employed and paid a normal salary according to the university regulations.

Gaudi factory was started as a pilot experiment in the summer of 2001 [4] with a group of six programmers working on a single product (an outlining editor). The following summer we introduced two other products and six more programmers. The work continued with half-time employments during the following fall and spring. In the fourth cycle, in the summer of 2003, there were five parallel experiments with five different products, each with a different focus but with approximately the same settings. Altogether, we have carried out 18 software construction experiments in Gaudi to this day. The application areas of the software built in Gaudi are quite varied: an editor for mathematical derivations, software construction and modeling tools, 3D model animation, a personal financial planner, financial benchmarking of organizations, a mobile ad-hoc network router, digital TV middleware, and so on.

2.2 Software Laboratory

The goal of the *Gaudi software laboratory* is to investigate, evaluate and improve the software development process used in the factory. The factory is in charge of the software product, while the laboratory is in charge of the software process. The laboratory supplies the factory with tasks, resources and new methods, while the factory provides the laboratory with the feedback in the form of software and experience results. The laboratory staff is composed of researchers and doctoral students working in the area of software engineering.

The kinds of projects carried on by the factory are quite varied and the application area, technology, and project stakeholders changed from project to project. However, there were also common challenges in all the projects to be addressed in the laboratory: product requirements were quite often underspecified and highly volatile and the developer turnaround was big.

Research software is often built to validate and demonstrate promising but immature ideas. Once it is functional, the software creates a feedback loop for the researchers. If the researchers make good use of this feedback, they will improve and refine their research work and therefore, they will need to update the software to include their improved ideas. In this context, the better a piece of research software fulfills its goal, more changes will be required in it. High developer turnaround is a risk that needs to be minimized in any software development company and the impacts of this have to be mitigated. In a university environment, this is part of normal life. We employ students as programmers during their studies. But, eventually, they will graduate and leave the programming team. A few students may continue as Ph.D. students or as part of a more permanent programming staff, but this is more the exception than the norm.

Our approach to these challenges was to use agile methods, in particular on Extreme Programming, and to split a large development project into a number of successive smaller projects. A smaller project will typically represent a total effort of one to two person years. This also is the usual size of project that a single researcher can find financing for in a university setting per year. A project size of one person year is also a good base for a controlled experiment. It is large enough to yield significant results while it can be carried out in the relatively short period of three calendar months using a group of four students.

3 Experiments in the Gaudi Factory

The Gaudi laboratory uses the Gaudi factory as a sandbox for software process improvement and development. Software projects in the factory are run as a series of monitored and controlled experiments. The settings of those experiments are defined a priori by the laboratory. After an experiment is completed, the settings are reviewed and our project standards are updated. In this section we describe the project settings and arrangements for Gaudi.

3.1 Schedule and Resources

A Gaudi experiment has a tight schedule, usually comprising three months. Most of the experiments are performed during summer, when students can work full-time (40 hours a week). In practice, this means that the developers start their work the first day of June and the final release of the software product is the last day of August. In projects carried out during the terms, students work half time (25-30 hours a week).

All the participants in an experiment are employed by the university, including the students working as developers, using standard employment contracts. All members of the same development team sit in the same room, arranged according to the advice given by Beck in [2].

3.2 Training

We should provide proper training to the developers in a project before it starts. However, the projects are short so we can not spend much time on the training. We chose to give the developers short (1-4 hours) tutorials on the essentials of the technologies that they are going to use. The purpose of these tutorials is not to teach a full programming language or a method, but to give a general overview of the topic and provide references to the necessary literature. We consider these tutorials as an introduction to standard *software best practices*, which are then employed throughout the Gaudi factory. Besides general tutorials that all developers take, we also provide tutorials on specific topics that may be needed in only one project, and which are taken only by the developers concerned. An example of the complete set of tutorials can be found in [5].

3.3 Experiment Supervision, Metrics Collection and Evaluation

We have established an experimental supervision and metric collection framework in order to measure the impact of different development practices in a project.

The complete description of our measurement framework is an issue for a separate paper, but in this section we outline its main principles. Our choice is the Goal Question Metric (GQM) approach [6]. GQM is based upon the assumption that for an organization to measure in a meaningful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals [6]. The current list of goals for the Gaudi factory as we see them can be found in [5] and its metric framework in [7].

4 Software Practices in Gaudi

In this section we describe the main practices in our process and our observations after applying them in several projects. We started our first pilot project [4] with just a few basic XP practices, evaluating them and gradually including more and more XP practices into the Gaudi process. After trying out a new practice in Gaudi we evaluate it and then, depending on the results of the evaluation, it either becomes a standard part of the Gaudi process, is abandoned, or is left for later re-implementation and re-evaluation. In this section, we discuss our experience with the agile practices which have been tried out in our projects. Some of the practices are adopted into our process and became a standard part of it, while some are still under evaluation.

The list of all these practices can be found in [5]. The technical report also includes percentages of activities performed by developers.

We now proceed as following: first we give a general overview of a practice, then we present our experience and results achieved with this practice. Finally we discuss possible ways to improve these practices in Gaudi environment. For the reader's convenience we split the practices into four categories: requirement management, planning, engineering and asset management.

4.1 Requirement Management Practices

Requirement management in XP is performed by the person carrying out the customer role. The requirements are presented in the form of user stories.

Customer Model
The role of the customer in XP is to write and prioritize user stories as defined in the next practice. The customer should also explain and clarify the stories to the development team and define and run acceptance tests to verify the correct functionality of the implemented stories. One of the most distinctive features of XP is that the customer should work onsite, as a member of the team, in the same room with the team and be 100% available for the team's questions.

As could have been guessed directly, it is hard to implement the onsite customer model in practice [8, 9]. Our experience confirms this. Among the 18 Gaudi projects, there was a real onsite customer only in one project – FiPla [7]. Before this the customer's involvement was minimal and it was in the Feature Driven Development [10] style: the offsite customer wrote requirements for the application, then the coach transformed these requirements into product requirements. Then the coach compiled the list of features based on the product requirements, and the features were given to the developers as programming tasks.

Studying the advantages of an onsite customer was one of the main objectives of the FiPla project. In this project the customer was available for questions or discussions whenever the development team felt this was necessary. However, the customer did not work in the same room with the development team. This was originally recommended by XP practices [11], but it was considered to be unnecessary because the customer's office situated in the same building with the development team's premises – this was considered to be "sharing enough".

Apparently, being an onsite customer does not increase the customer's work load very much [7]. One might even wonder whether an onsite presence is really necessary based on these figures. However, the feedback from the development team shows that an onsite customer is very helpful even though the customer's input was rather seldom needed. The developers' suggestion about involving the customer more in the team's work could also be implemented by seating the customer in the same room with the programmers. The feeling was that there could have been more spontaneous questions and comments between the developers and the customer if she had been in the same room.

Since summer 2004 we started using a customer representative or so called *customer proxy* model in Gaudi. The difference between these two customer models were that the customer representative does not commit himself to be always available to the team and in order to make decisions he had to consult the actual customer who was basically offsite. In both cases all customer-team communications were face-to-face, without e-mail neither phone discussions.

It is essential to have an active customer or customer's representative in an experimental project when the customer model itself is not a subject for the experiment. This allows us to keep the developers focused on the product, not the experiment and not be disturbed by the experimental nature of project.

User Stories

Customer requirements in XP projects are presented in the form of user stories [2]. User stories are written by the customer and they describe the required functionality from a user's point of view, in about three sentences of text in the customer's terminology.

We have used both paper stories and stories written into a web-based task management system. An advantage of paper stories is their simplicity. On the other hand, the task management system allows its users to modify the contents of stories, add comments, track the effort, attach files (i.e. tests or design documents) etc. It is also more suitable when we have a remote or offsite customer. Currently we are only using the task management system and do not have any paper stories at all.

In many projects, product or component requirements are represented in the form of tasks written by programmers. Tasks contain a lot of technical details, and often also describe what classes and methods are required to implement a concrete story. A story normally produces 3-4 tasks. When a story is split into tasks, the tasks are linked as *dependencies* of the story, and the story becomes *dependent* on tasks. When we used paper stories, we just attached the tasks to their stories. This is done in order to ensure the bidirectional traceability of requirements. Moreover, it is possible to trace each story or task to the source code implementing it. This is discussed in the Configuration Management practice. It is essential that each story makes sense for the developers (see Section 4.2) and it is estimable (we talk about the estimations in the Section 4.2).

4.2 Planning Practices

The most fundamental issues in XP project planning are to decide what functionality should be implemented and when it should be implemented. In order to deal with these issues we need the planning game and a good mechanism for time estimations.

Planning Game and Small Iterations

The *planning game* is the XP planning process [2]: business gets to specify what the system needs to do, while development specifies how much each feature costs and what budget is available per day, week or month. XP talks about two types of planning: by scope and by time. Planning by time is to choose the stories to be implemented, rather than taking all of them and negotiating about a release date and resources to be used (planning by scope).

The team estimates all the stories for the project and writes their estimations directly for the stories (we will discuss the estimation process in more details in the Section 4.2). These estimations are not very precise, the error is 20% on average [5], but can be smaller. E.g., in the FiPla [7] project the estimation error for the whole effort was 10% (approximately 30 hours). The estimations create an overall project plan and immediately tell us whenever some stories should be postponed to the next project or whether there is time to add more stories.

The task managements and bug tracking system allows us to submit tasks and bugs, and to keep track of them. Currently, we use the JIRA task management to keep track of task estimations. These kinds of systems are easy to use and provide an overall view of which tasks and bugs are currently under correction, which are fixed and which are open. This is especially important when the customer cannot act as an onsite customer (see Section 4.1).

In our experience, the planning game, the small releases and time estimations are very hard to implement without well-defined customer stories and technical tasks, and hence, without an active customer or customer representative.

Time Estimations

The essence of the XP release planning meeting is for the development team to estimate each user story in terms of ideal programming weeks [2]. An ideal week is how long a programmer imagines it would take to implement a story if he or she had absolutely nothing else to do. No dependencies, no extra work, but the time does include tests.

We have two estimation phases in the Gaudi process. The first phase is when the team estimated all of the stories in ideal programming days and weeks. These estimations are not very precise and they are improved in the second estimation phase when the team splits stories into tasks. When programmers split stories into technical tasks they make use of their previous programming experience and try to think of the stories in terms of the programs they have already written. This makes sense for the programmers and makes the estimating process easier for them.

The estimated time for a task is the number of hours it will take one programmer to write the code and the unit tests for it. These estimations are done by the same programmers that are signed up for the tasks, i.e., the person who estimates the task will later implement it. This improves the precision of the estimations. The sum is doubled to allocate time for refactoring and debugging. This is the estimation of a story for solo programming. In case of pair programming we need to take the Nosek's [12] principle into consideration: *two programmers will implement two tasks in pair 60 percent slower then two programmers implementing the same task separately with solo programming*. This means that a pair will implement a single task 20% faster then a single programmer. Similarly, to get the estimation for an iteration we have to sum the estimations of all stories the iteration consists of. Project estimation will be the sum of all its iteration estimations.

Estimating tasks turns out to be rather easy even for inexperienced programmers. The accuracy of the estimations depends, of course, on the experience of the developer. Experience in the particular programming language turns out to be more important than experience in estimation. Examples of the estimation accuracy in Gaudi can be found in [5].

XP-style project estimation is useful to plan the next one or two iterations in the project, but they can seldom be used to estimate the calendar length or resources needed in a project.

4.3 Engineering Practices

Engineering practices include the day-to-day practices employed by the programmers in order to implement the user stories into the final working system.

Design by Contract

Design by Contract [14] (DBC) is a systematic method for making software reliable (correct and robust). A system is structured as a collection of cooperating software elements. The cooperation of the elements is restricted by *contracts*, explicit definitions of obligations and guarantees. The contracts are *pre-* and *postconditions* of methods and *class invariants*. These conditions are written in the programming language itself and can be checked at runtime, when the method is called. If a method call does not satisfy the contract, an error is raised. Some reports [15, 16] show that XP and design by contract fit well together, and unit tests and contracts compliment each other.

Our first experiment with Eiffel and DBC showed very good results. First of all, the use of this method was one of the reasons for the low defect rate in the project [7]. As the development team commented out: *"All the tests written (to a complete code) always pass and the tests that don't pass have a bug in the test itself"*. Most of the bugs were caught with the help of preconditions, when a routine with a bug was called during unit testing. Most of the unit tests were written before the actual code, but the contracts were specified after it because the programmers did not get any instructions from their coach on when the contracts should be written. Example data for the post-release defect rate of the software developed with DBC is reported in [7].

Stepwise Feature Introduction

Stepwise Feature Introduction (SFI) is a software development methodology introduced by Back [17] based on the incremental extension of the object-oriented software system one feature at a time. This methodology has much in common with the original *stepwise refinement* method. The main difference to stepwise refinement is the bottom-up software construction approach and object orientation. Stepwise Feature Introduction is an experimental methodology and is currently under development.

We are using this approach in our projects in order to get practical experience with the method and suggestions for further improvements. Extreme Programming does not say anything about the software architecture of the system. Stepwise Feature Introduction provides a simple architecture that goes well with the XP approach of constructing software in short iteration cycles. So far we have had positive feedback from using SFI with a dynamically typed object-oriented language like Python. An experiment with SFI and Eiffel, a statically typed object-oriented language showed us some aspects of the methodology which need improvement. The explanation of these findings requires a more thorough explanation of SFI than what is motivated in this paper, so we decided to discuss this in a separate paper. Developers found SFI methods relatively easy to learn and use. The main complain was the lack of tool support. When building a software system using SFI, programmers need to take care of a number of routines which are time consuming but which could be automated. The most positive feedback about SFI concerned the layered structure: it clarifies the system architecture and it also helps in debugging, since it is relatively easy to determine the layer in which the bug is introduced.

Pair Programming

Pair programming is a programming technique in which two programmers work together at one computer on the same task [18]. Pair programming has many significant benefits for the design and code quality, communication, education etc [19, 20, 21, 22, 23]. Programmers learn from each other while working in pairs. This is especially interesting in our context since in the same project we can have students with very different programming experience.

In our first experiments we were enforcing developers to always work in pairs, later on when we had some experienced developers in the projects, we gave the developers the right to choose when to work in pair and when to work solo. In the 2003 projects pair programming was not enforced, but recommended, while in summer 2004 two months were pair programming and one month solo. We leave it up to the programmer whether to work in pairs while debugging or refactoring. The percentage of the solo-pair work for the five projects of 2003-2004 is reported in [5].

All of the developers agree that the code written in pairs is easier to read and contains less bugs. They also commented that refactoring is much easier to do in pairs. However there are different opinions and experiences on debugging. In some projects developers said that it was almost impossible to debug in pair because "*everyone has his own theory about where the bug is*" and "*while you want to scroll up, your pair want to scroll down, this disturbs concentration during debugging*". In other projects programmers preferred pair debugging because they found it easier to catch bugs together. We think that working in pairs should be enforced for writing all productive code, including tests, while it should be up to the developers, whenever debug or refactor in pairs or solo. It would be interesting to know which part of the code is actually pair programmed and which solo. A possible solution to distinguish between pair and solo code is to use specific annotations in the code [24], as used in Energi [25] projects, where the origin (pair or solo) of the code is described by comments.

Unit Testing

Unit testing is defined as testing of individual hardware or software units or groups of related units [26]. In XP, unit testing refers to tests written by the same developer as the production code. According to XP, all code must have unit tests and the tests should be written before the actual code. The tests must use a unit test framework to be able to create automated unit test suites.

Learning to write tests was relatively easy for most developers. The most difficult practice to adopt was the "write test first" approach. Our experience shows that if the coach spends time together with the programmers, writing tests himself and writing the tests before the code, the programming team continues this testing practice also without the coach. Some supervision is, however, required, especially during the first weeks of work. The tutorial about unit testing focused at the test driven development before the project is also essential. The implementation of the testing practice also depends on the nature of the programming task. Our experience showed that the "write test first" approach worked only in the situation where the first programming tasks had no GUI involved because GUI code is hard to test automatically.

In many projects the goal is to achieve 100% unit test coverage for non-GUI components. A program that calculates test coverage automatically provides an invaluable help to achieve this goal to both programmers and coaches.

Continuous Refactoring and Collective Code Ownership

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [27]. XP promotes refactoring throughout the entire project life cycle to save time and increase quality [28]. This practice together with pair programming also promotes collective code ownership, where no one person owns the code and may become a bottleneck for changes. Instead, every team member is encouraged to contribute to all parts of the project.

Pair programming, continuous refactoring, collective code ownership, and the layered architecture make the code produced in the Gaudi factory simpler and easier to read, and hence more maintainable. As mentioned before, larger products are developed in a series of three-month projects and not necessarily by the same developers. To ensure that a new team that takes over the project gets to understand the code quickly, we usually compose the team with one or two developers who have experience with the product from a previous project, the rest of the team being new to the product. In this way new developers can take over the old code and start contributing to the different parts of the product faster. When the team is completely new, the coach will help the developers to take over the old code.

4.4 Asset Management

Any nontrivial software project will create many artifacts which will evolve during the project. In XP those artifacts are added in the central repository and updated as soon as possible. Each team member is not only allowed, but encouraged to change any artifact in the repository.

Configuration Management and Continuous Integration

All code produced in the Gaudi Software Factory, as well as all tests (see Section 4.3), are developed under a version control system. We started in 2001 using CVS but now most projects have migrated to Subversion, which is now the standard version control system in Gaudi. The source code repository is also an important source of data for analyzing the progress of the project, since all revisions are stored there together with a record of the responsible person and date and time for check-in. The metrics issues were discussed in the Section 3.3.

Due to the small size (four to six programmers) of the development teams in Gaudi, we do not use a special computer for integration, neither do we make use of integration tokens. When a pair needs to integrate its code, the programmers from this pair simply inform their colleagues and ask them to wait with their integration until the first pair checks in the integrated code. The number of daily check-ins varies, but there is at least one check-in every day. In many cases integration is just a matter of few seconds.

It is important to be able to trace every check-in to concrete tasks and user stories [29]. For this purpose programmers add the identification of the relevant task or story to the CVS or Subversion log. The identification is the unique ID of the story or task in the task management system (SourceForge or JIRA). The exception is when the programmers refactor or debug existing code, it is then very hard (or impossible) to trace this activity to a concrete task or story. Therefore check-ins after refactoring or debugging are linked to the "General Refactoring and Debugging" task (see Section 4.1).

Agile Documentation

When a story is implemented, the pair or single programmer who implemented it should also write the user documentation for the story. The documentation is written directly on the story or in a text file located in the project's repository. This file is divided into sections, where each section corresponds to an implemented story. If the stories are on a web-base task management system, the documentation is written directly in the stories – this simplifies the bidirectional traceability for stories and their documentation. Later on the complete user documentation will be compiled from the stories' documentation. Documenting a user story is basically rephrasing it, and it takes an average of 30 minutes to do it.

This approach allows us to embed the user documentation into the development process. Bidirectional traceability of the stories and user documentation makes it easy to update the corresponding documentation whenever the functionality changes. The documentation examples from one of the Gaudi projects can be found in [5].

5 Conclusions and Related Work

In this paper we have presented Gaudi, our approach to empirical research in software engineering based on the development of small software products in a controlled environment. This approach requires a large amount of resources and effort but provides a unique opportunity to monitor and study software development in practice.

The software process used in Gaudi is based on agile methods, especially on Extreme Programming. In this, paper we have discussed the adoption and performance of 12 different agile practices. We believe that our collected data represents a significant sample of actual software development due to its size and diversity, and lends support for many of the claims made by the advocates of Extreme Programming.

There have been several efforts to study and validate how agile methods are used in the industry, such as the survey performed i.e. in [30, 31]. An industrial survey can help us to determine the performance of a completely defined process such as XP, but it cannot be used to study the effects of different development practices quantitatively, since the researchers cannot monitor the project in full details. Instead, the survey has to be based on the qualitative and subjective assessments of project managers of the success of the different development practices used in their projects. Abrahamsson follows a research approach that is similar to ours, combining software research with software development in *Energi* [25]. The main focus of his research is to evaluate agile methods proposed by other researchers in the field. In contrast, our intention is to perform empirical experiments not only to evaluate existing practices but also to propose new practices that we think will improve the overall software process.

The Gaudi framework project started in 2001 and have completed 18 projects during a period for 4 years representing an effort of 30 person years in total. This work has been measured and the results of these measurements are being used to create the so called Gaudi process. Once this process is completely defined it will be tested again in empirical experiments. It is possible to argue that this approach will result in a software process that is optimized for building software only in a university setting. Although this criticism is valid, it is also true that most of the challenges found in our environment such as scarce resources, undefined and volatile requirements and high programmer turn around are also present in many industrial projects.

In the previous section, we have evaluated each practice in detail. However, we would like to discuss some of the overall experiences obtained from the framework project.

Agile Methods Work in Practice: As overall conclusions of our experiences is that agile methods provide good results when used in small projects with undefined and volatile requirements. Agile methods have many known limitations such as difficulties to scale up to large teams, reliance on oral communication and a focus on functional requirements that dismisses the importance of reliability and safety aspects. However, when projects are of relatively small size and are not safety critical, agile methods will enable us to reliably obtain results in a short time.

The fact that agile methods worked for us does not mean that is not possible to improve existing agile practices. Our first recommendation is that architectural design should be an established practice. We have never observed a good architecture to "emerge" from a project. The architecture has been either designed a priory at the beginning of a project or a posteriori, when the design was so difficult to understand that a complete rethinking was needed.

Also, we established project and product documentation as an important task. XP reliance on oral communication should not be used in environments with high developer turnaround. Artifacts describing the software architecture, design and product manual are as important as the source code and should be created and maintained during the whole life of the project.

Project Management and Flying Hats: Another observation is that in many cases the actual roles and tasks performed by the different people involved in a project did not correspond to the roles and tasks assigned to them before the project started. This was due to the fact that the motivation and interest in a given project varied greatly from person to person. In some cases, the official customer for a project lost interest in the project before it was completed, e.g. in less than three months. In these cases, another person took the role of a customer just because that person was still interested in the product or because a strong commitment to the project made this person to take different roles simultaneously even if that was not his or her duty.

Our conclusion is that standard management tasks such project staffing, project supervision and ensuring a high motivation and commitment from the project staff and different stakeholders are as relevant in agile process in a university setting as in any other kind of project.

Tension Between Product and Experiment: Finally, we want to note that during these four years we have observed a certain tension between the development of software and the experimentation with methods. We have had projects that produced good products to customer satisfaction but were considered bad experiments since it was not possible to collect all the desired data in a reliable way. Also, there have been successful experiments that produced software that has never been used by its customer.

To detect and avoid these situations a well-defined measurement framework should be in place during the development phase of a project but also after the project has been completed to monitor how the products are being used by their customers.

References

1. Back, R.J., Milovanov, L., Porres, I., Preoteasa, V.: XP as a Framework for Practical Software Engineering Experiments. In: Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering – XP2002 (2002)
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
3. Abrahamsson, P.: Extreme Programming: First Results from a Controlled Study. In: Proceedings of the 29th EUROMICRO Conference "NewWaves in System Architecture", IEEE (2003)
4. Back, R.J., Milovanov, L., Porres, I., Preoteasa, V.: An Experiment on Extreme Programming and Stepwise Feature Introduction. Technical Report 451, TUCS (2002)
5. Back, R.J., Milovanov, L., Porres, I.: Software Development and Experimentation in an Academic Environment: The Gaudi Experience. Technical Report 641, TUCS (2004)
6. Basili, V., Caldiera, G., Rombach, D.: The Goal Question Metric Approach. Encyclopedia of Software Engineering. John Wiley and Sons (1994)

7. Back, R.J., Hirkman, P., Milovanov, L.: Evaluating the XP Customer Model and Design by Contract. In: Proceedings of the 30th EUROMICRO Conference, IEEE Computer Society (2004)
8. Korkala, M.: Extreme Programming: Introducing a Requirements Management Process for an Offsite Customer. Department of Information Processing Science research papers series A, University of Oulu (2004)
9. Korkala, M., Abrahamsson, P.: Extreme Programming: Reassessing the Requirements Management Process for an Offsite Customer. In: Proceedings of the European Software Process Improvement Conference EUROSPI 2004, Springer Verlag LNCS Series (2004)
10. Palmer, S.R., Felsing, J.M.: A Practicel Guide to Feature-Driven Development. The Coad Series. Prentice Hall PTR (2002)
11. Beck, K.: Embracing Change with Extreme Programming. Computer 32 (1999) 70–73
12. Nosek, J.: The Case for Collaborative Programming. Communications of the ACM 41 (1998) 105–108
13. Meyer, B.: Eiffel: The Language. second edition edn. Prentice Hall (1992)
14. Meyer, B.: Object-Oriented Software Construction. second edition edn. Prentice Hall (1997)
15. Feldman, Y.A.: Extreme Design by Contract. In: Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering, Springer (2003)
16. Heinecke, H., Noack, C. In: Integrating Extreme Programming and Contracts. Addison-Wesley Professional (2002)
17. Back, R.J.: Software Construction by Stepwise Feature Introduction. In: Proceedings of the ZB2001 – Second International Z and B Conference, Springer Verlag LNCS Series (2002)
18. Williams, L., Kessler, R.: Pair Programming Illuminated. Addison-Wesley Longman Publishing Co., Inc. (2002)
19. Cockburn, A., Williams, L.: The Costs and Benefits of Pair Programming. In: Proceedings of eXtreme Programming and Flexible Processes in Software Engineering XP2000. (2000)
20. Constantine, L.L.: Constantine on Peopleware. Englewood Cliffs: Prentice Hall (1995)
21. Johnson, D.H., Caristi, J.: Extreme Programming and the Software Design Course. In: Proceedings of XP Universe. (2001)
22. Müller, M.M., Tichy, W.F.: Case study: Extreme programming in a university environment. In: Proceedings of the 23rd Conference on Software Engineering, IEEE Computer Society (2001)
23. Williams, L.A., Kessler, R.R.: Experimenting with Industry's Pair-Programming Model in the Computer Science Classroom. Journal on Software Engineering Education (2000)
24. Hulkko, H.: Pair programming and its impact on software quality. Master's thesis, Electrical and Information Engineering department, University of Oulu (2004)
25. Salo, O., Abrahamsson, P.: Evaluation of Agile Software Development: The Controlled Case Study approach. In: Proceedings of the 5th International Conference on Product Focused Software Process Improvement PROFES 2004, Springer Verlag LNCS Series (2004)
26. Institute of Electrical and Electronics Engineers: IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York (1990)
27. Fowler, M.: Refactoring: Improving the Design of Existing Code. Object Technology Series. Addison-Wesley (1999)

28. Roberts, D.B.: Practical Analysis of Refactorings. PhD thesis, University of Illinois at Urbana-Champaign (1999)
29. Asklund, U., Bendix, L., Ekman, T.: Software Configuration Management Practices for eXtreme Programming Teams. In: Proceedings of the 11th Nordic Workshop on Programming and Software Development Tools and Techniques NWPER'2004. (2004)
30. Ilieva, S., Ivanov, P., Stefanova, E.: Analyses of an Agile Methodology Implementation. In: Proceedings of the 30th EUROMICRO Conference, IEEE Computer Society (2004)
31. Rumpe, B., Schröder, A.: Quantitative survey on extreme programming projects. In: Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002, May 26-30, Alghero, Italy (2002) 95–100