Software Development and Experimentation in an Academic Environment: The Gaudí Factory

Ralph-Johan Back and Luka Milovanov and Ivan Porres *

Åbo Akademi University Department of Information Technologies
Turku Centre for Computer Science (TUCS)
Joukahaisenkatu 3-5 A, FIN-20520 Turku, Finland

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 five years to define and evaluate an agile software process. This process combines practices from Extreme Programming with architectural design and improved documentation in order to find a balance between agility, maintainability and reliability.

Key words: Agile Methods, Empirical Software Engineering, Case Studies

1 Introduction

One of the main challenges in software engineering research is the experimental and empirical validation of new software development languages, methods and tools. Like in any other discipline, it is important to test and evaluate if new and existing advances in software engineering provide the expected benefits when applied in practice. We consider this validation process is very weak and often neglected by both researchers and industrial practitioners. This means that good languages, tools and methods are not adopted fast enough and that bad approaches linger on longer than they should.

Therefore, we consider that there is needed is a stronger emphasis on empirical and experimental software engineering, where the new tools and methods are tried out

* Corresponding author.
Email addresses: lmilovan@abo.fi, http://www.it.abo.fi/ (Ralph-Johan Back and Luka Milovanov and Ivan Porres).

in case studies and controlled experiments, and the results gained from these are carefully analyzed. The problem is how to perform this research in a way that is cost effective and efficient.

Doing experiments in the industry is difficult. Software is nowadays built under severe time and resource constraints and is often critical to the success of the company as a whole. Experimenting with new methods is risky and therefore it is often avoided. In many cases, it is almost impossible to perform significant studies in an industrial setting. A company can rarely afford to develop the same product twice by the same team but using two different methods, and then compare the resulting products and the performance of the team. Software researchers can have access to industrial software construction processes, observing and measuring them, but they usually cannot influence the way the projects are carried out to any larger degree. In other words, researchers do not have control over the software processes and methods that used in the industry.

Further on, most software engineering research has been following a research-then-transfer methodology [56]. In this way, the research and the application of the research in the industry are considered as separate, sequential activities. The problem with this approach is that the researchers working in the software engineering research units even in close collaboration with the industry do not have the control over industrial projects. Software experiments can be carried out in an academic environment, but the setting there is usually not very realistic. Software is built by inexperienced students using more or less ad hoc processes, and the penalties for schedule slips or unsatisfactory quality are usually low. Maintaining larger software systems is very difficult in an academic environment because the students turnover is high.

First, a synthetic development project arranged by a researcher does not reflect the conditions and constraints found in an actual software development project. This happens especially if there is no actual need for the software to be developed. Researcher have limited funds, and therefore it is necessary to optimize the costs of the experiments, therefore university experiments must be performed by students. Although students are not necessarily less capable of these tasks than employed software developers, they must, however, be trained. In addition to this, their programming experience and motivation in the project may vary considerably. Another problem may be that during their curricula, students are taught more theoretical aspects of Computer Science rather than practical issues of software engineering [36]. Added to these difficulties there is also the high turnover rate as students graduate and leave the project. Finally, although there is no market pressure, a researcher often has very limited resources and therefore it is not always possible to conduct large experiments or case studies.

On the other hand, university researchers do not have any pressure to release new software products to the market or even being economically profitable to their em-

ployer. In this sense, a university setting can be an ideal place to perform experiments and practical case studies and test new ideas in software engineering. 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 run a software development project inside a research center or university in a more controlled environment. Still, this approach has some important shortcomings, including the lack of clear vision and requirements for the system to be build and difficulties to fund the developers.

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

In this paper we describe our experiences using this approach: how we created our own software production unit – the *Gaudí Software Factory* in the university settings. The purpose of the factory is to provide us with a realistic environment where we can empirically study software development in practice while building software for other research projects. Our experience is based on different case studies conducted since the years 2001. The objective of these case studies was to find and document software best practices in a software process that focus on product quality and project agility.

This paper is structured as follows: in Section 2 we present the Gaudí Software Factory and the Laboratory for Software Construction – our environment for empirical software engineering. The settings of the software factory – a sandbox for empirical investigation are described in Section 3. We present the typical setting for software project and illustrate the application areas of the software built in Gaudí in Section 5. The software process used in Gaudí is discussed in Section 4, while Section 5 summarizes our observations from agile experience in Gaudí. Our conclusions are presented in Section 6.

1.1 Related Work

There has been active of research on different aspects of agile methods. First of all, agile methods and especially XP have been criticized for the lack of concrete evidence of success [4], or in general [63], but they also have been gaining popularity

both in Industry and Academia. There are many papers published on different aspects of agile methods, i.e. in industrial [23,35,37,52,58] case studies. 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.

In various academic [4,26,34,51] case studies the researchers can monitor the projects in full details and determine the performance of XP quantitatively. But in many cases the environment of the projects which is created in academic settings does not reflect the reality, e.g. there is no real customer who needs the software, as opposed to Gaudí, where the software is going to be used by its customers. Other papers are concerned with evaluation of different agile practices: onsite customer [46,47], pair programming [53], refactoring [57] or what is the significance of the practices combination [67]. There is research on teaching of agile practices where the educational value of agile methods is studied [20,30,42,44,49], research on human aspects of agile processes [28,45], and finally papers on combining software research with software development in e.g. NASA SEL [16] and *Energi* at VTT Technical Research Centre of Finland.

The Industry-driven Experimental Software Engineering Initiative [59] (ENERGI) is an environment in VTT Oulu, which was originally targeted to evaluate and improve agile methods proposed by other researchers in the field. In contrast, our intention was not only to evaluate existing agile practices but also to propose new practices that we think will improve the overall software process. The Software Factory at the Arizona State University *gathers student programmers, in a common facility where a professional software engineer mentors and manages them. As a result, researchers of non-computing disciplines receive well-designed, documented and tested software, and students receive experience working in a professional software development organization* [1]. However, it looks like with such great potential for the research in software engineering, the ASU Factory seems to lack the research part, as we did not manage to find any scientific paper telling otherwise.

Another example is University of Sheffield in UK. This university has an initiative called the Software Hut [33] where students undertake projects for external clients in this imitative several student teams produce the same product for a client and the client selects the best implementation. There is also a student run software house Genesys, that provides software products for external clients. Similarly, the Software Hut and Genesys have been used by the Sheffield staff to evaluate agile methods.

2 Gaudí and its Working Principles

Gaudí 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 case studies. We focus on lightweight or agile software processes. Similarly to the EF model that *offers an organizational structure that separates the product development focus from the learning and reuse focus, collects data and packages the experience for further reuse*, in our approach, we made a clear separation between the software construction and the research units, which work together in a very close relationship. These units are the Gaudí Software Factory and the Laboratory for Software Construction. Together they present a research environment that results in realistic and empirically valid evidence of different aspects of software engineering methods.

Gaudí is a research project which is related to the *Experience Factory* [3] (EF) approach. The *Experience Factory* [3] (EF) approach promotes organizational learning in such a way that the organization manages and learns from its own experience. In this approach the organization observes and collects data about itself, creates conclusions based on this data and packages the experience for further reuse. Finally, and most importantly, the organization feeds these experience packages back to itself to share them inside and outside the organization. An example of such an approach is the NASA GSFC Software Engineering Laboratory [16].

2.1 Software Factory

Gaudí Software Factory is a part of CREST at Åbo Akademi University, which is a research center focusing at the construction of reliable software. CREST consist of four laboratories: Embedded Systems (ES), Distributed Systems (DS), Software Construction (SC) and Mechanized Reasoning (MR). The Gaudí Software Factory was built as a central resource for constructing software for these laboratories. Later Gaudí was also used to build software for research outside CREST.

The goal of the Gaudí 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 provide the resources required to carry out the project. A characteristic of the factory is that the developers are students. However, programming in Gaudí is not a part of their studies, and the students get no credits for participating in Gaudí – they are employed and paid a normal salary according to the university regulations. We emphasize to the Gaudí software developers that the purpose of their work is to produce working software using the specified software process, methods and tools. Our intention is to keep the programmers occupied with the constructing of the software, and to be

mindful that our research does not disturb them in any way. This seems to work out well: in most cases the developers reported that they did not feel they were involved in a research project, or they said that the experimental nature of the project did not disturb their day-to-day routine.

2.2 Software Laboratory

The *Laboratory for Software Construction* 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 case studies run in the Gaudí factory. The goal of the Software Construction Laboratory in collaboration with Gaudí factory 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 laboratory uses the Gaudí factory as a sandbox for software process improvement and development. Software projects in the factory are run as a series of monitored and controlled case studies. The settings of those projects including tools, methods, techniques and a software process are defined a priori by the laboratory. Based on these case studies we have defined a standard collection of practices which has proved to be successful in these case studies. Nevertheless, we always consider possibilities to improve and extend our standard framework with new settings in future research projects. We call these settings *software best practices* as they focus on product quality and project agility.

One of the main challenges in the factory is high developer turnaround. This is a consequence of the environments where the software projects are carried out. Programmer 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. Some students can be employed for more then one project, 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.

Other common challenges in Gaudí have come from the characteristics of an academic research environment: product requirements were quite often underspecified and highly volatile and the developer turnaround was high. Software is also often built in the context of a research project to validate and demonstrate promising

but immature research 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, the more changes will be required in it.

Our approach to these challenges has been to base our software process on agile methods, in particular on Extreme Programming, and to divide a large development project into a number of successive smaller projects. The main characteristics behind agile methods and particularly XP fit into our framework [12] for empirical software engineering. A central feature of XP is its simplicity. First of all, XP is easy to learn. As we have seen in our first project [11], students – the programmers in Gaudí, learn XP quickly while doing what they like: programming. The ability to start a running project in a very short time is also a great advantage for short time span university projects. That has been important for us since we really do not want to spend too much time teaching the students. We want to get project running as soon as possible. We simply have not seen other possibilities then agile process methods. Agile methods can help us by providing a flexible software process that is easy to learn, keep the programmers focused on the product and not on the experiment and allow us to observe the results of the programmers as early as possible.

2.3 Projects

Each project carried out in Gaudí should involve the development of (part of) a software system according to the specifications of a customer that commissions the project. This is an important aspect in Gaudí: all development project should have a customer that defines the product to be built and pays for its development. In some cases, the system is developed from scratch, but often the goal of the project is to develop additional features into an existing system.

The Gaudí factory was started as a pilot study [11] in the summer of 2001 with a group of six programmers working on a single product (an outlining editor). The study was carried out as a survey, while most of the subsequent studies were case studies. The following summer we introduced two other products and six more programmers. The work continued with part-time employments during the following fall and spring. In the fourth cycle, in the summer of 2003, there were five parallel projects with five different products, each with a different focus but with approximately the same settings. Altogether, we have carried out over 20 software construction projects in Gaudí to this day. The application areas of the software built in Gaudí are illustrated in Section 5.

3 Settings of the Gaudí Factory

In this section we describe the project settings and arrangements for Gaudí, as well as the different roles and duties involved in a project. Here present our approach on how to set up the environment for empirical research in software engineering. This approach has required a large amount of resources and effort but it has provided us with unique opportunity to monitor and study software development in practice.

3.1 Project Roles

Traditionally, the division of labor in software development has been performed based on the different phases of a water fall or sequential process: developers are specialized into analysts, architects, designers and testers. In many agile methods, personnel is split into only two main groups: technical developers and customers. In Gaudí, we have found the need to also identify other categories that are important for carrying out the overall software development process.

A team in a Gaudí project usually consists of 6-7 people. Four students perform the *programmers'* tasks. A professor or senior researcher acts as a *top manager*, a PhD student plays the role of *coach*, and a researcher (a professor, post doctoral student or a PhD student) plays the role of a *customer* or *customer proxy*.

Developer Four undergraduate students are employed to perform the developers' tasks for each project. These students are usually third or fourth year students majoring in Computer Science or Computer Engineering. On an average about 45% of the students in a project had participated earlier in Gaudí projects. Having one experienced student is important for a new team to take over the old code in continuation and maintenance (see Section 5) projects. As of today, nearly 50 students have worked in Gaudí as developers.

Coach and Tracker XP gives the following definition in [19] for the role of the coach: "A role of the team for someone who watches the process as a whole and calls the team's attention to impending problems or opportunities for improvement". In XP the traditional project management is divided into two roles: the coach and the tracker. Coaching is concerned with technical execution of the process, while tracking is about measurements and their validation against project's estimates. Main responsibilities of the coach are to be available as a development partner for new programmers, encourage refactoring, help programmers with technical tasks, helping everybody else making decisions and explain the process to

the upper-level management. The job of the tracker is to collect the defined metrics, ensure that the team is aware of the measures and remind the earlier made predictions.

In the Gaudí factory both roles of the coach and the tracker (measurements are discussed in the section 3.4) are played by the same person, a PhD student. The coach is mostly needed by the team during the first weeks of a project. It is often necessary for the coach to spend a few hours with the developers weekly, performing the tasks of the developers, especially when a completely new team takes over an old project or in case of very inexperienced developers. But after the first small release the programming team becomes more autonomous and needs their coach less and less. At this point the coach becomes less concerned with various types of technical solutions and his or her main concern becomes the overall process monitoring and execution, and the customer's involvement.

Customer and Lead Developer Customer model is a key issue in an agile process because all phases of a project require communication with the customer. The role of the customer in XP is to write and prioritize user stories, explain them for the development team and to 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 [18]. XP customer should remain focused on understanding the needs while the developers concentrate on programming. An ongoing dialog between these roles is crucial for the success of the project [48].

3.2 Schedule and Resources

A characteristic of the Gaudí factory is that the developers are students. Finding time to meet and work together is the most frequent problem when we consider students as developers of a software project [60]. We have avoided this problem by employing students full time for the whole project paying them a normal salary according to the university regulations. The schedule for the projects is defined by the fact that the students are usually employed for the summer only. That is, we set ourselves a strict three-month deadline: the product has to be released by August 31 at latest. Developers work 40 hours a week, no overtime. There have been a number of projects during fall and spring, when the students worked part-time (25-30 hours a week), but such schedule better fits for maintenance projects.

A typical software project in Gaudí represents a total effort of one to two person-years. This is also the usual size of a 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 case study. 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. In some cases we were dividing the projects in three clear phases: training, programming and cleaning up [11]. But in the majority of the projects and later on as a rule, we rather divide our summer projects into five equal iterations. The first project meeting where all the members are introduced to each other usually takes place in the middle of May.

The developers working with the same project have flexible and normal daily working hours: 8:00 to 16:00 or 10:00 to 18:00. All of the developers in a project sit in the same room arranged according to the advice given by Beck [19]. There is a big table in the center with four computers for pair and solo programming and other tables against the walls for personal use. There is no vertical separators or cubicles. The room also has a bookshelf, a food table with a coffee maker and a white-board. Another room with more white-boards used by the programmers for meetings with customers and for presentations. The programmers have been satisfied with the room.

3.3 Training

Usually only a few of the developers are familiar with the tools and techniques we use in our projects. Therefore, we have to provide proper training for them. However, the projects are short so we can not spend much time on the training. We choose to give the developers short (1-4 hour) 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 Gaudí 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.

Table 1 shows the complete set of tutorials for one of our projects (FiPla [9]). For the Gaudí customers we also give one tutorial which is called ”XP for Gaudí Customers”.

Developers also get some selected literature to study (manuals, technical documentation, books) after the tutorials. During the project, they have the possibility to ask the project coach for help with the practical application of the techniques and tools used in the project. Those developers who did not participate in our previous projects find these tutorials very helpful and their the number and length is sufficient.

The first week at the beginning of the project is also reserved for training. During

Tutorial	Numbers	Total hours
Eiffel and DBC	2	4
CVS	1	2
Extreme Programming	1	2
SFI	1	2
Unit testing	1	2
All tutorials together	6	12

Table 1
Tutorials

this time, the programmers do not get the actual development tasks, but they spend time getting acquainted with the tools to be used during the project, writing their own small programs or completing simple assignments given by their coach. During this phase the developers also need supervision and help from the people in charge of training and tutorials.

3.4 Metrics Collection and Evaluation

When properly collected, software metrics [25] help to improve the software and the software process in organizations. On the other hand, one should know exactly what is the use of a particular metric and why it is collected. Clear measurement objectives driven by organizational business goals also improve the motivation to measure, while the ad-hoc approach in most cases leads to a large amount of data which is hard if possible to make use of. In mature organizations the metrics collection depends on the business goals of the organization [64]. After the goals have been identified, the focus is set on the metrics which are relevant to the business goals. On the other hand, some organizations collect a lot of different metrics, but have no strategy for its usage.

The first Gaudí project [11] explored the use of XP practices: our objectives were more or less to see whether we could produce software in the university settings with inexperienced students. In the subsequent projects, e.g. [9,32], we concentrated on the establishment and maintenance of a light-weight software process. Under these settings, we needed concrete quantitative evaluation of the methods under study. We 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. The starting point for our measurements was the Goal Question Metric (GQM) approach [14] and Quality Improvement Paradigm [15,17] (QIP).

QIP combines the evolutionary and revolutionary experimental aspects of the scientific method, tailored to the study of software, i.e., the development of complex systems that need to have models built and evolved to aid our understanding of the artifact. It involves the understanding as well as the evolutionary and revolutionary improvement of software. The steps of the QIP are:

- *Characterize* the current project and its environment.
- *Set* the quantifiable goals for successful project performance and improvement.
- *Choose* the appropriate process model and supporting methods and tools for this project.
- *Execute* the processes, construct the products, collect and validate the prescribed data, and analyze it to provide real-time feedback for corrective action.
- *Analyze* the data to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements.
- *Package* the experience in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects and save it in an experience base for future projects.

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 [14]. Exemplar goals for the Gaudí factory are: *"Focus on writing code and tests"*, *"Improve customer's interaction and process transparency for customer"*, *"Improve customer's satisfaction"*, etc. More of the goals can be found in [9,32], for example. We have chosen an incremental approach to define our metrics framework. The idea behind it is to take the very basic and simple metrics, define them and their collection mechanisms and use it as a standard guidelines in Gaudí. This framework is extended with more metrics as needed.

Besides stating the goals and defining the metrics to reach the goals and data collection mechanisms, we will also describe the feedback mechanisms. These feedback mechanisms are basically describing what one should do with the data.

Another type of data we collect in Gaudí is qualitative. During the project developers are asked to keep a shared log of their personal feelings, experience and anything else which in their opinion concerns the project. In addition, at the last day of work, each programmer gets a list with many questions concerning the projects. Customers are also asked to keep a free-form diary where they should write down all activities they performed in their project and time spent for it. Finally, if we need more qualitative data, we interview developers and customers after the project has finished.

It is important to identify the person in charge of collecting the defined metrics. One of the requirements for the success of a metric program is commitment. Re-

sponsibility for the metrics program should be assigned to specific individuals [29]. Furthermore, the commitment of this person should also be established. In our case, we found that the best person for this work is a project coach. Some measurements such as unit test coverage and personal time tracking should be assigned to developers. But a Gaudí developer should not be responsible for the measurements because this data has to deal with the process improvement and experimenting, while we want to keep our developers focused on the software they build and not on the research they make a part of.

4 Development Practices and Impact in Experimentation

Agile methods provide good results when used in small projects with undefined and volatile requirements. We started building the software process of Gaudí factory with just a few basic practices from XP, evaluating them and gradually including more and more practices into the Gaudí process. After trying out a new practice in Gaudí we evaluated it and then, depending on the results of the evaluation, it either became a standard part of the Gaudí process, was abandoned, or was left for later re-implementation, adaptation and re-evaluation. Table 2 lists the agile practices we have had experience in Gaudí.

Adopted	Under Evaluation	Abandoned
No overtime, pair programming, code standards, unit testing, refactoring, collective code ownership, continuous integration, automated tests and daily builds, coach as project manager, user stories, short iteration, iteration planning, spike solutions, lightweight documentation, customer proxy, time estimations	100% unit test coverage, tests written before the code, onsite customer, release planning, project velocity measured, system metaphor	Daily stand up meetings, CRC cards or similar, score of acceptance tests published

Table 2
Process Practices in Gaudí Software Factory

The set of adopted practices evolved from the set of practices under evaluation. We started the first Gaudí project [11] with the simplified XP process including the following practices: pair programming, collective code ownership, continuous integration, refactoring, unit testing, short term planing and small releases. We chose particularly these practices because in our opinion it was the only possibility to start with. We saw them as a minimum which we require in order to release software in three months, yet we saw them as all we needed and could handle at that point. After the project was over, the basic process for Gaudí was established, therefore

we could start improving our process with the introduction of more agile practices which seemed necessary.

Software process improvement was one of the drivers for the introduction of new practices. For example, we saw that in the early projects we needed more customer-team interaction – so we brought in the onsite customer [9]. However, our goal was not to follow the existing practices blindly. The fact that agile methods worked for us did not mean that it was not possible to improve existing agile practices. First, we were trying to implement a practice as it was originally defined in literature and test it in the Gaudí environment. After the data on the impact of the practice was obtained and analyzed we were searching the ways to adopt this practice in our environment in order to get the maximum benefit with minimal cost out of it. An example of such an adaptation is the evolution of the onsite customer into the customer proxy [32].

Table 3 shows percentage of activities performed by developers out of total project effort. The first four rows show data for the projects of Summer 2003, the remaining two for Summer 2004. All activities were performed in the listed projects, but the amount of time for some projects and activities was insignificant, therefore some values in the table are zeros.

Activity	Deve	FiPla	MED	U3D	SCS	CRL
Programming and Unit Testing	19	39	48	39	34	56
Refactoring	0	9	7	13	4	6
Debugging	7	14	15	19	19	14
Integration	0	0	1	1	1	0
Design	1	6	4	8	3	7
Meetings	5	1	1	1	4	2
Research	33	6	3	4	11	3
Planning game	0	2	3	0	1	0
With Customer	0	2	0	0	7	0
Miscellaneous	31	21	18	15	9	9

Table 3
Developers' activities %

While improving the Gaudí process by introducing new or modifying the existing agile practices we saw that in order to find a balance between agility, maintainability and reliability we needed other methods and techniques – something outside the agile world. Particularly architectural design and documentation practices are neglected by agile processes, yet we could see that these practices would benefit the Gaudí factory. Therefore, we established project and product documentation as

an important practice of the Gaudí process. Reliance on oral communication in agile world could not be used in our environment for the products to be developed over a number of summer projects 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. Stepwise Feature Introduction [8] provides a simple architecture that goes well with the agile approach of constructing software in short iteration cycles.

In the rest of this section we will briefly introduce the core practices of the Gaudí process. For the readers' convenience we group the practices in four categories:

- *Requirements management*. Requirements management in XP is performed by the person carrying out the customer role. The requirements are presented in the form of user stories. The practices considered here are Customer Model and User Stories.
- *Planning*. The most fundamental issues to be decided in XP project are 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. The practices considered here are the Planning Game and Time Estimations.
- *Engineering*. Engineering practices include the day-to-day practices employed by the programmers in order to implement the user stories into the final working system. The practices considered here are Pair Programming, Refactoring, Collective Code Ownership, Unit Testing, Design by Contract and Stepwise Feature Introduction.
- *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. The practices considered here are Configuration Management, Continuous Integration and Documentation.

The goal of this paper is to introduce the Gaudí environment, not the evaluation of the practices which is an issue for a separate paper. Those interested can read about particular practice evaluation in [8,9,10,32].

4.1 Requirements management

Customer Model The role of the customer in XP is to write and prioritize user stories, explain them for 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. The XP customer practice appears to be achieving excellent results, but it also appears to be unsustainable [46].

An active customer is also a great boost for the team morale, as the Gaudí programmers noticed: *"It* would *be more motivating to develop software that somebody is actually going to use. The customer could have been more active, and at least pretend to be interested in the product"*. The XP customer practice appears to be achieving excellent results, but it also appears to be unsustainable [46]. Among the 20 Gaudí projects, there was a real onsite customer only in one project [9]. Before this the customers involvement was minimal and it was in the Feature Driven Development [55] style: the offsite customer wrote requirements for the application, then the coach transformed these requirements into product requirements. After that the coach compiled the list of features based on the product requirements, and the features were given to the developers as programming tasks.

Being an onsite customer does not increase the customer's work load very much [9] and has a number of benefits such as: improved communication between customers and the programming team, decreased number of false features and feature misses, etc. But despite the well-known benefits of an onsite customer, this model is hard to implement in practice [24,41,40,65] due to the lack of commitment or the high value of the customer. Yet, without sufficient developers-customer interaction it becomes very hard to cope with changing requirements even though that is one of the main goals of agile software development [31].

In one of our case studies [32] we showed how the mentioned problems were tackled with the effective use of a *customer representative* or a *customer proxy*. In our experience, customer feedback with the customer representative model works at least as well as with the onsite customer [32]. Bringing the customer closer to the development process is important for keeping the project on track even with changing requirements. Iterations, the heart beat of agility, provide only a short time to influence things. Project time competes with the customer's restricted and valuable time. We believe that the involvement of a customer representative can be the required piece in making needed software functionality, uncertainty and time resources meet. Truly active customer interaction enables agility.

Table 4 shows how the Gaudí customer's time was spent on project issues. Apparently, being an onsite customer does not increase the customer's work load very much. 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. The second row in the table 4, SCS, shows the data for

	Available	Writing stories	With team	Testing	Idle
FiPla	100	2.5	3	2.5	92
SCS	71	5	9	20	37

Table 4
Customer involvement (%)

the project of summer 2004 where we did not have an onsite customer, but used a customer representative or so-called *customer proxy*. The difference between these two customer models were that in the SCS project the customer representative did 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, no e-mails, no phone discussions. It is essential to have an active customer or customer's representative [32] in a project when the customer model itself is not a subject for the case study. This allows us to keep the developers focused on the product, not the research 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. 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 customers terminology without using technical jargon [2,38]. Beck [19] provides additional recommendation for stories: they should also include such information as the title, date, status and a short description of what the user should be able to do after the story was finished. The time needed to implement the stories should be estimable and they must make sense to the programmers.

In the Gaudí factory we do not require customers to have complete customer or product specification for the software to be build. However, we do expect our customer to write stories, either themselves or via their representatives. The most comprehensive written instructions are formulated as customer stories which followed the guidelines given by the XP practice. The division of the product's features into the stories is made by the customer based on an intuitive idea about what meaningful chunks the system could be divided into. A typical one person year project normally has 15-25 user stories. The stories can also be the result of joint work between the customer and the coach. While most of the stories are written before the project or in the beginning of it, customers still bring new stories throughout the project's time and delete or change existing stories.

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. It is essential that each story makes sense for the developers and it is estimable.

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.

4.2 Planning

Planning Game and Small Iterations The *planning game* is the XP planning process [19]: 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 used 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 time and people resources are fixed in a Gaudí project: the schedule is usually three months and there are only four programmers available. Therefore we do release planning by time. Because the developers (and often also the customer) lack experience, the coach usually selects the stories for the first short iteration. The selection is based on two factors: selected stories should be implemented in two weeks maximum and those stories should have the highest priority. The process also teaches the customer how to create good stories – after estimating the stories the coach often asks the customer to rewrite them in order to produce smaller and better estimable stories. The coach also asks the customer to write tests or testing scenarios based on the stories. After the coach and the customer decide on the functionality for the first two weeks, the team and the coach will together split the stories into technical tasks and then the developers will implement the tasks. No time estimations are done at this point. By the time the first iteration functionality is implemented, the team is better acquainted with the programming language and the product, so they are in a better position to provide time estimations.

Each new iteration starts with the customer selecting the stories from the project plan that should be implemented in the next release. The development team and the customer meet in the beginning of each iteration to discuss the features to be implemented. Since the customer stories usually do not provide very detailed guidelines for the desired features, the development team and the customer need to discuss in

order to clarify open issues and provide more precise requirements. These meetings usually take about an hour. During these meetings, some of the time is used to make sure the team understands the application logic correctly, the rest of the discussions often concern aspects of the user interface. There are typically five iterations in a usual summer Gaudí project.

Time Estimations The XP release planning meeting is based on the idea that the development team estimates each user story in terms of ideal programming weeks [19]. An ideal week is how long a programmer imagines it would take to implement a story and its tests if he or she had absolutely nothing else to do.

We have two estimation phases in the Gaudí 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 E_{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. Estimated time E_{story_i} for a story $story_i$ split into number of tasks $task_{i,j}$ is twice the sum of all its task estimations:

$$E_{story_i} = 2 \sum_j E_{task_{i,j}}$$

The sum is doubled to reserved the 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 [53] principle into consideration: *two programmers will implement two tasks in pair 60 percent slower than two programmers implementing the same tasks separately with solo programming.* The story estimation for pair programming case in Gaudí is:

$$E_{story_i} = \frac{5}{3} \sum_j E_{task_{i,j}}$$

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. 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.

Task Management As we already mentioned, we have abandoned user stories written on paper. We are using a web-based issue tracking system for user stories, tasks and bug report. The issues on a web-based system are as easy to handle as the paper stories or tasks. This becomes especially obvious, when it comes to the modification of the stories, or attaching e.g. design documents or test reports to them. Electronic issues also have a number of advantages when compared to the paper cards.

With an issue tracking system projects become more transparent to the customer and the coach. It allows the customer to work offsite. Each electronic issue can be linked to the source code in the repository. They can also contain the user documentation (see Section 4.4). Also such a system makes it easier to collect some of the metrics such as defect rate, deviation from the schedule, customer's and developers' productivity, to mention a few. Currently, we are using JIRA [43] as a standard issue tracking system in Gaudí.

4.3 Engineering

Pair Programming Pair programming is a programming technique in which two programmers work together at one computer on the same task [66]. The programmer who types is called a driver, the other programmer is called a navigator. While the driver works tactically, the navigator works strategically: looking for misspells and errors and thinking about the overall structure of the code. All code in XP is written in pairs. Productivity is assumed to follow the Nosek's principle. Pair programming has many significant benefits: better detailed design (in XP the design is performed on the fly), shorter program code and better communication between team members. Also, many common programming mistakes are caught as they are being typed, etc [21]. As it has been frequently reported [21,22,39,50,68], pair programming also has a great educational aspect. 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 projects 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. Table 5 shows the percentage of the pair-solo work in the three projects of summer 2003 and two of summer 2004, the first number indicates the percentage of pair work. 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.

	FiPla	MED	U3D	SCS	CRL
Programming	79/21	88/12	84/16	62/38	60/40
Refactoring	77/23	85/15	76/24	49/51	62/38
Debugging	27/73	84/16	86/14	74/26	51/49

Table 5
Pair vs. solo %

Unit Testing Unit testing is defined as testing of individual hardware or software units or groups of related units [54]. 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.

Refactoring The most popular definitions for refactoring is given by Fowler [27]: ”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”. XP promotes refactoring throughout the entire project life cycle to save time and increase quality [57] by removing redundancy, eliminating unused functionality, rejuvenating obsolete designs. 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.

4.4 Asset Management

Continuous Integration According to XP, developers should integrate code into the code repository every few hours, whenever possible, and in any case changes should never be kept for more than a day. In this way XP projects detect early compatibility problems, or even avoid them altogether, and ensure that everyone

works with the latest version. Only one pair should integrate at a time. Due to the small size (four to six programmers) of the development teams in Gaudí, 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.

All code produced in the Gaudí Software Factory, as well as all tests, are developed under a version control system. 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. It is important to be able to trace every check-in to concrete tasks and user stories [6]. For this purpose programmers add the identification of the relevant task or story to commit 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.

Documentation There are two types of documentation produced in Gaudí: software documentation or source code documentation and user documentation or manuals. Since our intention has been the continuation of some projects with new development teams, the practice of documenting the source code has been in Gaudí from the very first project. The programmers were adding documentation strings to each class and method. In Python there are comments but also a special language statement for documentation which then are processed by the *PyDoc* (similar to *JavaDoc*) tool which produces e.g. HTML documentation for the code. The approach of the self-documenting code goes well with the continuous refactoring discussed before because the code and its documentation are available for the developer’s reference and updates at the same place, and the whole documentation of the system, e.g. in HTML, is always kept up to date by simply typing ”`make doc`” command. In the case of Eiffel, sufficient up-to-date software documentation is provided in the form of pre- and post-conditions and class invariants [9].

User documentation received special attention in Gaudí as the expertise areas of project’s customer and coach started differ, e.g. a project which developed an application in information systems, a tool for financial benchmarking using self-organizing maps [62]. The fact that the built software would be used by people outside the Gaudí factory and the laboratory influenced our growing interest in the user documentation. The idea behind the documentation process is the following: 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 di-

rectly on the story (issue on JIRA system) 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-based task management system, the documentation is written directly in the stories – this simplifies the bidirectional traceability for stories and their documentation, and helps keeping the documentation up-to-date. 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. In order to ensure that each of the stories is documented, the customer closes a story (accepts its implementation) only after it has been documented.

These approaches allow us to embed the software and the user documentation into the development process. The documentation for the new developers is provided by the self-documenting code, while the 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 Gaudí projects can be found in [10].

5 Experiences from Gaudí

In the previous sections we have presented the environment for empirical software engineering research – Gaudí Software Factory and the Laboratory for Software Construction. We have described the settings for the Gaudí Software Factory and presented the software process which has been used in over 20 projects during a period of five years in the factory. In this section we would like to discuss some of the overall experiences obtained from the Gaudí environment.

Past Projects The application areas of the software built in Gaudí are quite varied. Examples of produced software are: 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.

Table 6 lists 20 projects in the Gaudí factory. The effort of each project is given in person-months (PM) and the size of produced software size is given in thousands of lines of code (KLOC). We distinguish between four types of projects:

- **NEW:** New software project where a software product is built from scratch. Usually most of the tools, methods and experimental techniques selected for such a project have already given positive feedback in previous Gaudí projects. Only few new methods can be investigated.

- **CNT:** Continuation of an existing software project, where a new development team takes over the code of an existing software product. The new team usually consists of 50-75% of programmers without previous experience with the software or even without any previous experience in Gaudí. More of the new experimental methods can be used for such projects.
- **MNT:** Maintenance project where the old team takes care of the maintenance of the existing software. Some minor functionality can be added to the product, but the main software goal is maintenance. Only a few new methods can be investigated.
- **EXP:** Technology exploration project where the software goal is not to build a product for a customer, but to explore specific software technology, such as a new programming language or technique, or CASE tool. These projects often involve the use of COTS or open source libraries, etc.

Since there is no software goal as such for the technology exploration projects, we do not list the lines of code produced in them.

All software in Gaudí was built in time and on budget. We have managed to deliver the desired software for its customers in time among all the Gaudí project which had clear goals for the software to be build. The software products reach their functional level that is set at the end of the overall planning period for the projects. This does not only concern the projects with well-defined requirements, but also the projects with very high requirement uncertainties [32]. Examples of this software are: Coral Modeling Framework [5], Software Construction Workbench [7], MathEdit – an Editor for Mathematical Derivations [13], Domino – a tool for financial benchmarking with self organizing maps [62] and SOCOS – an Editor for Refinement Diagrams [8].

We introduce refactoring right after the first short iteration and promote it throughout the whole project. Pair programming, continuous refactoring, collective code ownership, and the layered architecture make the code produced in the Gaudí factory simpler and easier to read, and hence more maintainable. As mentioned before, larger products are developed in a series of three-months 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.

Agile Methods Support our Experimental Setup 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 limita-

Year	Project	Effort, PM	Type	Size, KLOC
Summer 01	EXE	18	NEW	4.7
Summer 02	SCW	12	NEW	18.6
	MTR	3	EXP	–
	MED	12	NEW	14.8
	SMW	12	CNT	36.1
Spring 03	SCW	6	MNT	20.5
	SMW	6	MNT	45.6
	MED	6	MNT	20.3
Summer 03	FPL	12	NEW	11.1
	U3D	12	NEW	–
	MED	12	CNT	31.6
	WLN	12	EXP	–
	DVE	12	EXP	–
Summer 04	NS2	18	EXP	–
	CRL	12	NEW	11.8
	SCS	12	NEW	10.0
Fall 05	SCS	6	CNT	13.2
	DOM	8	NEW	7.4
Summer 05	SCS	12	CNT	10.5
	CRL	12	CNT	28.9
Total		215		160.3

Table 6
Software projects in Gaudí, 2001-2005

tions 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.

Tension between Development and Research Finally, we want to note that during these five years we have observed a certain tension between the development of software and the research of software development. We have had projects that produced good products to customer satisfaction but were considered bad case studies since it was not possible to collect all the desired data in a reliable way. Also, there have been successful case studies 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.

6 Conclusions

The software engineering discipline studies how to build large software systems that fulfill the user's requirements, are reliable and are constructed on time and within budget. This includes the study of many different concepts and techniques used in software development: software process models, modeling notations, programming languages and methods, testing and validation strategies, CASE tools. Various methods such as surveys, case studies and formal experiments have been proven to be useful for empirical software engineering. However, most software engineering research has been following a research-then-transfer methodology due to the lack of the proper environment which reflects the real settings of the industry and yet allows researches to have control over the software process.

In this paper we have presented Gaudí, our approach to empirical research in software engineering based on the development of small software products in a controlled environments. This approach requires a large amount of resources and effort but provides a unique opportunity to monitor and study software development in practice. And at the same time it allows us to produce maintainable and quality software for its customers – researchers. The Gaudí factory started in 2001 and has carried out 20 projects during a period of five 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 Gaudí process. Once this process is completely defined it can be used to assess new software engineering techniques, methods and tools. We believe that other universities and research units can benefit

from us in their research in empirical software engineering by creating a factory-laboratory environment similar to our Gaudí settings.

The software which has been built in Gaudí environment provides the validity of Gaudí as a factory for software construction. All software in Gaudí was built in time and on budget. We have managed to deliver the desired software for its customers in time among all the Gaudí project which had clear goals for the software to be build. The software products reach their functional level that is set at the end of the overall planning period for the projects. This does not only concern the projects with well-defined requirements, but also the projects with very high requirement uncertainties.

Another evidence for the validity of our research is its benefit to our environment. We are not just improving or inventing software engineering methods, but we are ”using our own medicine” – implementing our finding into the way the software is built in Gaudí factory. Here the evidence is satisfied customer and software delivery in time when there are large requirement uncertainties – as a result of the customer proxy application, those successful products which were built using the Stepwise Feature Introduction methodology, etc.

6.1 Limitations and Future Work

A good working software factory would need to have a permanent staff of software developers who would built up the competence of the factory. We see the biggest limitation of our work in the competence build-up. People are the greatest assets in software organizations because they represent intellectual capital [61]. However, we are not able to keep these valuable assets because Gaudí developers are students who will eventually graduate and leave, and the funding infrastructure for employing permanent staff in order to build the competence is missing. We could use this limitation as an advantage and investigate new software methods within the high turnaround factor, but this would seriously limit the spectrum of our research. And even a greater threat for us could be a high turnaround of the researches involved in Gaudí as coaches and project managers. The Gaudí factory uses agile process with much reliance on oral communication which perhaps could be mitigated by the means of documentation and standardization. Yet it is unclear whether documents and standards would actually help and will a heavily documented process be still flexible and agile.

We see our future work in Gaudí in moving towards formal experiments. We do not think we should abandon other investigation techniques: surveys and case studies. We have collected enough of valuable data for retrospective studies and we can foresee that case studies will be the most reasonable approach for some of the future software projects in Gaudí. A case study is also the most likely technique for

future collaboration of Gaudí with the industry. Nevertheless, formal experiments will become especially useful for us as the Gaudí process became defined and the number of tools, methods and practices used became set. In this situation formal experimentation will be the best way to consider alternative tools, methods and practices. Formal experiments would also provide us with with the data on generalizability of our experience which would be critical for the experience transfer outside Gaudí.

References

[1] ASU Software Factory. http://softwarefactory.asu.edu/.

[2] Extreme Programming: A gentle introduction website. Online at: http://www.extremeprogramming.org/.

[3] *Encyclopedia of Software Engineering*, volume 1, chapter The Experience Factory, pages 469–476. John Wiley & Sons, Inc., 1994.

[4] Pekka Abrahamsson. Extreme Programming: First Results from a Controlled Study. In *Proceedings of the 29th EUROMICRO Conference "New Waves in System Architecture"*. IEEE, 2003.

[5] Marcus Alanen and Ivan Porres. The Coral Modelling Framework. In Johan Lilius Kai Koskimies, Ludwik Kuzniarz and Ivan Porres, editors, *Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004*, number 35 in TUCS General Publications. TUCS Turku Centre for Computer Science, Jul 2004.

[6] Ulf Asklund, Lars Bendix, and Torbjörn Ekman. 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*, August 2004.

[7] Ralph Back, Dag Björklund, Johan Lilius, Luka Milovanov, and Ivan Porres. A workbench to experiment on new model engineering applications. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, San Francisco , CA, USA, Oct 2003.

[8] Ralph-Johan Back, Johannes Eriksson, and Luka Milovanov. Using stepwise feature introduction in practice: An experience report. In *RISE*, pages 2–17, 2005.

[9] Ralph-Johan Back, Piia Hirkman, and Luka Milovanov. Evaluating the XP Customer Model and Design by Contract. In *Proceedings of the 30th EUROMICRO Conference*. IEEE Computer Society, 2004.

[10] Ralph-Johan Back, Luka Milovanov, and Ivan Porres. Software Development and Experimentation in an Academic Environment: The Gaudi Experience. Technical Report 641, TUCS, 2004.

[11] Ralph-Johan Back, Luka Milovanov, Ivan Porres, and Viorel Preoteasa. An Experiment on Extreme Programming and Stepwise Feature Introduction. Technical Report 451, TUCS, 2002.

[12] Ralph-Johan Back, Luka Milovanov, Ivan Porres, and Viorel Preoteasa. 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*, May 2002.

[13] Ralph-Johan Back and Magnus Myreen. Tool support for invariant based programming. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 711–718, Washington, DC, USA, 2005. IEEE Computer Society.

[14] Victor Basili, Gianluigi Caldiera, and Dieter Rombach. *The Goal Question Metric Approach. Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[15] Victor R. Basili. Quantitative Evaluation of Software Engineering Methodology. In *Proceedings of the First Pan Pacific Computer Conference, Melbourne, Australia*, September 1985.

[16] Victor R. Basili, Frank E. McGarry, Rose Pajerski, and Marvin V. Zelkowitz. Lessons learned from 25 years of process improvement: The Rise and Fall of the NASA Software Engineering Laboratory. In *IEEE Computer Society and ACM International Conference on Software Engineering (ICSE 2002)*, May 2002.

[17] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Trans. on Software Engineering*, 14(6):758–773, June 1988.

[18] Kent Beck. Embracing Change with Extreme Programming. *Computer*, 32(10):70–73, October 1999.

[19] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[20] Joseph Chao. Balancing hands-on and research activities: A graduate level agile software development course. In *Proceedings of Agile 2005 Conference, July, Marriott Denver City Center*, 2005.

[21] Alistair Cockburn and Laurie Williams. The Costs and Benefits of Pair Programming. In *Proceedings of eXtreme Programming and Flexible Processes in Software Engineering XP2000*, 2000.

[22] L. L. Constantine. *Constantine on Peopleware*. Englewood Cliffs: Prentice Hall, 1995.

[23] Ko Dooms and Roope Kylmäkoski. Comprehensive documentation made agile – experiments with rapid7 in philips. In *Proceedings of the 6th International Conference on Product Focused Software Process Improvement - PROFES 2005, Oulu, Finland*, 2005.

[24] C. Farell, R. Narang, S. Kapitan, and H. Webber. Towards an effective onsite customer practice. In *Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, May 2002.

[25] Norman E. Fenton. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, Boston, MA, USA, 1996.

[26] Thomas Flohr and Thorsten Schneider. An xp experiment with students - setup and problems. In *Proceedings of the 6th International Conference on Product Focused Software Process Improvement - PROFES 2005, Oulu, Finland*, 2005.

[27] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 1999.

[28] Robert Gittins and Sian Hope. A study of Human Solutions in eXtreme Programming. In *Proceedings of 13th Workshop of the Psychology of Programming Interest Group – PPIG, Bournemouth UK*, April 2001.

[29] Tracy Hall and Norman Fenton. Implementing effective software metrics programs. *IEEE Softw.*, 14(2):55–65, 1997.

[30] Görel Hedin, Lars Bendix, and Boris Magnusson. Introducing software engineering by means of Extreme Programming. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, Washington, DC, USA, 2003. IEEE Computer Society.

[31] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–122, 2001.

[32] Piia Hirkman and Luka Milovanov. Introducing a Customer Representative to High Requirement Uncertainties. A Case Study. In *Proceedings of the International Conference on Agile Manufacturing*, 2005.

[33] M. Holcombe, M. Gheorghe, and F. Macias. Teaching xp for real: Some initial observations and plans; proceedings xp2001; sardinia, 2001.

[34] Hanna Hulkko and Pekka Abrahamsson. A multiple case study on the impact of pair programming on product quality. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, New York, NY, USA, 2005. ACM Press.

[35] Sylvia Ilieva, Penko Ivanov, and Eliza Stefanova. Analyses of an Agile Methodology Implementation. In *Proceedings of the 30th EUROMICRO Conference*. IEEE Computer Society, 2004.

[36] Sadahiro Isoda and Motoshi Saeki. Software engineering in asia. *IEEE Software*, 11(6):63–68, 1994.

[37] Andreas Jedlitschka, Dirk Hamann, Thomas Göhlert, and Astrid Schröder. Adapting PROFES for Use in an Agile Process: An Industry Experience Report. In *PROFES*, 2005.

[38] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2001.

[39] David H. Johnson and James Caristi. Extreme Programming and the Software Design Course. In *Proceedings of XP Universe*, 2001.

[40] Mikko Korkala. Extreme Programming: Introducing a Requirements Management Process for an Offsite Customer. Department of Information Processing Science research papers series A, University of Oulu, 2004.

[41] Mikko Korkala and Pekka Abrahamsson. 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.

[42] Noel F LeJeune. Teaching software engineering practices with extreme programming. *J. Comput. Small Coll.*, 21(3):107–117, 2006.

[43] Atlassian Software Systems Pty Ltd. JIRA – bug tracking, issue tracking and project management software. http://www.atlassian.com/software/jira/, 2007.

[44] Daniel Lübke and Kurt Schneider. Agile hour: Teaching xp skills to students and it professionals. In *Proceedings of the 6th International Conference on Product Focused Software Process Improvement - PROFES 2005, Oulu, Finland*, 2005.

[45] Katiuscia Mannaro, Marco Melis, and Michele Marchesi. Empirical Analysis on the Satisfaction of IT Employees Comparing XP Practices with Other Software Development Methodologies. In *5th International Conference on Extreme Programming and Agile Processes in Software Engineering – XP 2004, Garmisch-Partenkirchen, Germany*, June 2004.

[46] Angela Martin, Robert Biddle, and James Noble. The XP Customer Role in Practice: Three Studies. In *Agile Development Conference*, 2004.

[47] Angela Martin, James Noble, and Robert Biddle. Being Jane Malkovich: A Look Into the World of an XP Customer. In *4th International Conference on Extreme Programming and Agile Processes in Software Engineering – XP 2003, Genova, Italy*, May 2003.

[48] Robert C. Martin. eXtreme Programming Development through Dialog. *IEEE Softw.*, 17(4):12–13, 2000.

[49] Grigori Melnik and Frank Maurer. Introducing Agile Methods in Learning Environments: Lessons Learned. In *Third XP and Second Agile Universe Conference on Extreme Programming and Agile Methods – XP/Agile Universe*, August 2003.

[50] Mathias M. Müller and Walter F. Tichy. Case study: Extreme programming in a university environment. In *Proceedings of the 23rd Conference on Software Engineering*. IEEE Computer Society, 2001.

[51] Roger A. Müller. Extreme programming in a university project. In *5th International Conference on Extreme Programming and Agile Processes in Software Engineering – XP 2004, Garmisch-Partenkirchen, Germany*, 2004.

[52] Orlando Murru, Roberto Deias, and Giampiero Mugheddu. Assessing XP at a European Internet Company. *IEEE Softw.*, 20(3):37–43, 2003.

[53] J.T. Nosek. The Case for Collaborative Programming. *Communications of the ACM*, 41(3):105–108, 1998.

[54] Institute of Electrical and Electronics Engineers. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York, 1990.

[55] Stephen R. Palmer and John M. Felsing. *A Practicel Guide to Feature-Driven Development*. The Coad Series. Prentice Hall PTR, 2002.

[56] Colin Potts. Software-engineering research revisited. *IEEE Software*, 10(5):19–28.

[57] D. B. Roberts. *Practical Analysis of Refactorings*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[58] Bernhard Rumpe and Astrid Schröder. Quantitative survey on extreme programming projects. In *Third International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP2002, May 26-30*, pages 95–100, Alghero, Italy, 2002.

[59] Outi Salo and Pekka Abrahamsson. 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.

[60] Dean Sanders. Student Perceptions of the Suitability of Extreme and Pair Programming. In *Proceedings of XP Universe*, 2001.

[61] Ian Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[62] Patrik Ståhl, Tomas Eklund, Franck Tétard, and Barbro Back. A cooperative usability evaluation of the domino software. Technical Report 775, TUCS, Jun 2006.

[63] Matt Stephens and Doug Rosenberg. *Extreme programming refactored: the case against XP*. Apress L.P., 2003.

[64] CMMI Team. Capability Maturity Model Integration, Version 1.1, CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1), Continuous Representation, CMU/SEI-2002-TR-011, ESC-TR-2002-011, March 2002.

[65] Nathan Wallace, Peter Bailey, and Neil Ashworth. Managing xp with multiple or remote customers. In *Proceedings of the Third International Conference on eXtreme Programming and Agile Processes in Software Engineering - XP2002*, May 2002.

[66] Laurie Williams and Robert Kessler. *Pair Programming Illuminated*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[67] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Softw.*, 17(4):19–25, 2000.

[68] Laurie A. Williams and Robert R. Kessler. Experimenting with Industry's Pair-Programming Model in the Computer Science Classroom. *Journal on Software Engineering Education*, December 2000.