TUCS

Ralph-Johan Back | Luka Milovanov | Ivan Porres

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

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 641, November 2004

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

Ralph-Johan Back
Åbo Akademi University, Department of Computer Science,
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
backrj@abo.fi

Luka Milovanov
lmilovan@abo.fi

Ivan Porres
iporres@abo.fi

TUCS Technical Report

No 641, November 2004

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 the Gaudi software process. This process combines practices from Extreme Programming with architectural design and documentation practices in order to find a balance between agility, maintainability and reliability.

Keywords: Agile Methods, Software Engineering Experiments, Gaudi Factory

TUCS Laboratory
Software Construction Laboratory

1 Introduction

One of the main problems that hinders the research and improvement of various software construction techniques is the difficulty to perform significant controlled 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. A company can rarely afford to develop the same product twice by the same team but using different methods, and then compare the resulting products and the performance of the team.

On the other hand, universities employ highly qualified research personnel that can dedicate a considerable amount of their time to study better ways to build software. Also, university researchers do not have the pressure of releasing new software products to the market or even being economically profitable to their employer. In this sense, a university setting can be an ideal place to perform practical experiments 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 some factors that cannot be controlled by the researcher. The obvious alternative is to run a software development project inside a research center in a more 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 plan 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 research in software processes.

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

As a framework process for these experiments we chose Extreme Programming [10] (XP). Extreme Programming is an agile software methodology that was introduced by Beck in 2000. It is characterized by a short iteration cycle, combining the design and implementation phases, continuous refactoring supported by extensive unit testing, onsite customer, promoting team communication and pair programming. XP is quite popular these days, but still it has been criticized for lack of concrete evidences of success [2].

1.1 Related Work

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 [19, 32] 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.

Pekka Abrahamsson follows a research approach that is similar to ours, combining software research with software development in *Energi* [33]. 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.

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 agile experience in Gaudi. Our conclusions are presented in Section 6.

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.

We emphasize for the Gaudi 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 busy on building the software, not on the experiments. This seems to work out well: in most cases the developers reported that they did not feel they were involved in an experimental project, or then they said that the experimental nature of the project did not disturb them.

Gaudi factory was started as a pilot experiment in the summer of 2001 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.

High developer turnaround 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. 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.

Although application area, the technology used and project stakeholders varied from project to project, there were common challenges in all these projects that comes from the characteristics of an academic research environment: product requirements were quite often underspecified and highly volatile and the developer turnaround was big. Also, software is 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, more changes will be required in it.

Our approach to these challenges was to base our software process on agile methods, in particular on Extreme Programming, and to split a large development project into a number of successive smaller projects. A smaller projects will typically represent a total effort of one to two person years. This is also the usual size of project that a single researcher can find financing for in a university setting per year. A project size of 1 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. These settings were applied as an subject for experiments and produced positive results. Therefore they were taken into the standards and became the basic standard settings for all of our projects. Nevertheless, we always consider possibilities to improve and extend our standard framework with new settings in future experimental projects.

In this section we describe the project settings and arrangements for Gaudi. We also present the different roles and duties involved in an experimental project, as a background for the different process practices discussed later in Section 4.

3.1 Schedule and Resources

A Gaudi experiment has a tight schedule, usually 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 come to work first of June and the final release of the software product is the last day of August. During the terms students work half time (25-30 hours a week). We tend to focus on software maintenance issues during the terms, leaving the development of new software products to the summer period. Larger products are build in a sequence of short projects, e.g., June-August, October-December, February - April, and so on. The interludes are used to evaluate the software that was produced in the last experiment and plan for the next software experiment. All the participants in an experiment are employed by the university, including the students working as developers, using standard employment contracts.

In all Gaudi projects all members of the same development team sit in the same room, arranged according to the advice given by Beck in [10]. The programmers sit by a big table in the middle of the room. Four computers are placed so that the work stations formed a clover-like square. Since the team normally consist of only two pairs, there is no special machine for integration. There are no separators which could impede communication. There is a bookshelf, a white-board and a noticeboard in the room. Outside this room is a recreation area with a coffee maker etc. that is shared with other groups of programmers.

3.2 Training

Since only a few of the developers are familiar with the tools and techniques we use in our experiments, 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 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.

Table 1 shows the complete set of tutorials for one of our projects (FiPla [5]). For the Gaudi customers we also give one tutorial which is called ”XP for Gaudi Customers”. Developers also get some selected literature to study (manuals, tech-

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

nical 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.

In our first experiment [6] the tutorials were given to the developers after the official start of the project. In subsequent experiments we have given the tutorials before the project started officially, those were given two weeks before the projects started, after agreeing with the developers of the time schedule to avoid collisions with their normal lectures and examinations.

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

3.3 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 [8]. 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 [8]. The current goals for the Gaudi factory as we see them are:

1. Focus on writing code and tests.
2. Improve productivity.
3. Improve time estimations.
4. Improve software quality.
5. Improve customer's interaction and process transparency for customer.
6. Improve developers' competence.
7. Show the impact of the experimental techniques on the Gaudi baseline average measures.

Besides stating the goals and defining the metrics to reach the goals and data collection mechanisms, we will also describe the feedback mechanisms. These

	0.1	0.2	0.3	0.4	0.5	Total
LOC	1694	3441	5517	7100	8572	
Test LOC	571	983	2174	2347	2548	
Total LOC	2265	4424	7691	9447	11120	
Classes	11	23	37	52	59	
Test classes	9	10	20	23	25	
Methods	71	122	171	256	331	
Test methods	50	68	157	167	177	
LOC/Class	154	150	149	137	145	
LOC/test class	63	98	109	102	102	
Methods/class	7	5	5	5	6	
Test methods/class	6	7	8	7	7	
Post-release defects	2	1	2	1	0	6
Post-release defects/KLOC	1.18	0.57	0.96	0.63	0	0.70
Total work effort (h)	210	112	216	320	256	1114
Productivity (LOC/h)	8	16	10	5	6	8
Test productivity	3	4	6	1	1	2
Total productivity	11	20	16	6	7	10

Table 2: Collected data for all iterations

feedback mechanisms are basically describing what one should do with the data (i.e. table 2). We have chosen an incremental approach for defining our metrics framework. The idea is to take the very basic and simple metrics, define them and their collection mechanisms and use it as a standard guideline in Gaudi. This framework is extended with more metrics as needed. 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. Responsibility for the metrics program should be assigned to specific individuals [16], furthermore the commitment of this person should also be established. The best person for this work is the coach. Some measurements such as unit test coverage and personal time tracking should be assigned to developers. But a Gaudi 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 experiment they are part of.

Another type of data we collect in Gaudi 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. The log is a plain text file divided into sections. The programmers add new sections to the log as they find necessary. The records in such logs vary from complains: ”*It is too hot in the room and no ventilation.*” to practical advices: ”*Warning, do this and you will not loose your code. . .* ” and personal experience: ”*Some of our assignments are really boring, while others are more interesting*”. Finally, 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.

3.4 Roles in Experimental Projects

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 Gaudi, we have found the need to also identify other categories that are important for carrying out the overall software development process.

Coach and Tracker: XP gives the following definition in [10] 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 skills, getting everybody else making good 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 Gaudi factory both roles of the coach and the tracker (measurements are discussed in the section 3.3) 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 unexperienced 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: The role of the customer in XP is to write and prioritize user stories (see Section 4.1.2), explain them for the development team and to define and run acceptance tests to verify the correct functionality of stories. One of the most distinctive features of the XP customer is that he or she 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.

It is very hard, if not even impossible, to obtain commitment from a person to play the role of the onsite customer during the vocation time in university. Therefore there are roles of offsite customers and customer representatives in Gaudi.

Developer: A team of a Gaudi project usually consists of 6-7 people. A professor or senior researcher acts as a top manager, a PhD student plays the role of coach, a researcher (a professor, post doctoral student or a PhD student) plays the role of a customer, and four undergraduate students perform the programmers' tasks. The role of a project manager is played by the coach, or by an experienced developer. The undergraduate students are third or fourth year students majoring in Computer Science or nearby areas. On an average about 45% of the students in a project had participated earlier in Gaudi projects. As of today, nearly 40 students have worked in Gaudi as developers.

3.5 The Gaudi Process

As we have discussed in the introduction, our intention is to develop a lightweight software process which is flexible and is easy to learn and use. This process should lead to reliable software which is also easy to maintain, and it should be applicable in academic and possibly, industrial settings.

Extreme Programming is the basic framework process for Gaudi [7]. We started with a basic set of XP practices: pair programming, unit testing, refactoring, short iteration cycles, and light documentation, to name a few. This XP tool-set has been extended with Stepwise Feature Introduction (SFI), an experimental programming methodology.

One of the features that we appreciate most in XP, and which was the main reason for choosing it for our first experiment, is its simplicity. First of all, XP is easy to learn. That is an important issue for us since there is only a short time to train new students before a project starts. Another reason for choosing XP approach was its short interaction cycle that facilitates the creation of running software in a short period of time.

4 Software Practices in Gaudi

In this section we describe the 12 main practices in our process and our observations after applying them in several projects. We started our first pilot project [6] 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. Table 3 lists all of these practices.

Table 4 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.

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	100% unit test coverage, tests written before the code, onsite customer, customer proxy, time estimations, release planning, project velocity measured	Daily stand up meetings, System metaphor, CRC cards or similar, score of acceptance tests published

Table 3: Process Practices in Gaudi Software Factory

Activity	Deve	FiPla	MED	U3D	SCS	CRL
Programming	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 4: Developers’ activities %

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.

4.1.1 Customer Model

The role of the customer in XP is to write and prioritize user stories (see Section 4.1.2), 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.

As could have been guessed directly, it is hard to implement the onsite customer model in practice [23, 24]. Our experience confirms this. Among the 18 Gaudi projects, there was a real onsite customer only in one project – FiPla [5]. Before this the customers involvement was minimal and it was in the Feature Driven Development [30] 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 [9], 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”.

Table 5 shows how the 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.

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

Table 5: Customer involvement (%)

The second row in the table 5, SCS, shows the data for 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 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. An active customer is also a great boost for the team morale, as the programmers noticed: *"It* would *be more motivating to develop a 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"*.

4.1.2 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 techno-syntax [1, 21]. Beck [10] 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 Gaudi factory we do not require customers to have complete customer or product specification for the software to be build. 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 summer 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.

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 Section 4.4.1. It is essential that each story makes sense for the developers (see Section 4.2.1) and it is estimable (we talk about the estimations in the Section 4.2.2)

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.

4.2.1 Planning Game and Small Iterations

The *planning game* is the XP planning process [10]: 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 time and people resources are fixed in a Gaudi 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.

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 4.2.2). These estimations are not very precise, the error is 20% on average, but can be smaller. E.g., in the FiPla [5] 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 JIRAtask management to keep track of task estimations. This kind 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 4.1.1).

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 Gaudi project.

The team estimates whether there is a need for reconsidering the time cost of the iteration in the presence of the customer, after which the developers proceeded to break down the iteration into tasks among themselves in order to make more precise estimations. The outcome of the iteration planning is that the set of stories is split into tasks and the release is calendarized. The customer and the team need to find the balance between the functionality to be implemented and the effort required for this. The length of an iteration is usually around two weeks, maximum is three weeks for projects with well defined requirements, and minimum is one to one and half week for projects with high requirements uncertainties. Nowadays, all planning activities in Gaudi factory are done with JIRA task management and bug tracking system (section 4.4.1). The tool is convenient to use and it supports most of the required release and iteration planning activities.

After all the stories for the iteration have been implemented, the customer gets access to the release and planning for the next iteration . After it is finished the customer starts doing acceptance testing and the found defects are reported in the form of stories, after which they are treated as regular stories: prioritized, estimated, assigned to a small release and fixed. The overall project effort can be re-estimated based on the findings of an iteration.

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.

4.2.2 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 [10]. 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 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 [28] 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, hence the story estimation for pair programming case will be:

$$E_{story_i} = 2\sum_j (\frac{5}{6} E_{task_{i,j}}) = \frac{5}{3}\sum_j E_{task_{i,j}}$$

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.

Figures 1 and 2 show estimation errors for stories and iterations in one of the Gaudi projects. Estimating tasks turns out to be rather easy even for unexperienced 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.

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.

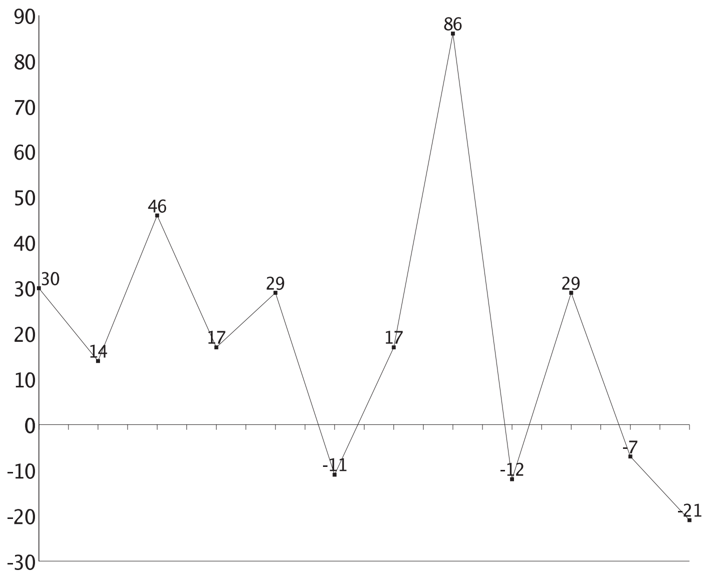

Figure 1: Story estimation error (%)

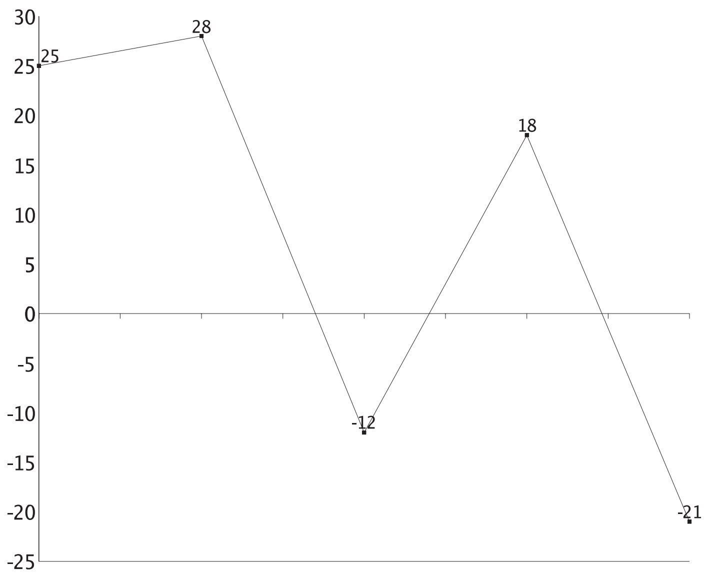

Figure 2: Iteration estimation error (%)

4.3.1 Choice of Programming Language

We use the Python programming language in most of our projects except for the four project of the summer 2003, where two projects used C++, one used Java and one used Eiffel.

Our students learn the Java language in their regular programming courses. However, Java is not always suitable for the purposes of our experiments. Nevertheless, knowledge of a programming language which is used in a particular project is not required when developers are employed for the project.

Python has a reputation of being easy to learn, use and to have a clear and elegant syntax. These aspects were the reasons for choosing Python as a programming language of our first Gaudi project and they were confirmed in this and later projects. The students who participated in Gaudi were usually familiar with Java but only a few had some previous experience with Python. Nevertheless, all students agreed that Python was extremely easy to learn and it was easy to start programming Python from the very beginning of the projects.

In the FiPla project [5] we used Eiffel [25] as the programming language of the project, because we wanted to try out Design by Contract (see Section 4.3.2) and Eiffel has very good built-in support for this technique. Eiffel is an object-oriented language that also includes a comprehensive approach to software construction: a method, and an environment (EiffelStudio) [20]. It is a simple, yet powerful language that strictly follows the principles of object-orientation. The language supports multiple inheritance, has no global variables and pointer arithmetics. Eiffel has a choice of graphical libraries, including the portable *EiffelVision* library, used in our project. Eiffel compilation technique uses C as an intermediate language. The run-time efficiency of Eiffel's executables is similar to C.

Unfortunately, ISE Eiffel has no original *unit testing* framework. Unit testing is an essential part of the Extreme Programming (see section 4.3.5) and Gaudi Process, and could not be left outside our project, in particular as we had a lot of positive experience with unit testing. Our choice was to use the *Gobo Eiffel Test* tool [11]. Gobo Eiffel Test is distributed freely under the terms and conditions of the Eiffel Forum License [34].

We got a lot of positive experience using Eiffel. First of all, the defect rate (table 6) of the software built with Eiffel was much lower then in the software built with Python. In the developers' opinion, the low defect rate of Eiffel software was due to the use of design by contract, static typing and, surprisingly, because of the poor Eiffel's documentation – this forced the team to do more spikes and testing. Another aspect we appreciated in Eiffel was hight code readability. According to the developers who were working in previous projects with Python, Eiffel code was even more readable than Python code.

4.3.2 Design by Contract

Design by Contract [26] (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 *post-conditions* 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 [14, 17] show that XP and design by contract fit well together, and unit tests and contracts compliment each other.

We tried to start using design by contract already in our first experiment [6]. However, this attempt failed due to the lack of design by contract support in Python. Our first experiment with Eiffel and design by contract showed very good results. First of all, the use of design by contract was one of the reasons for the low defect rate in the project [5]. 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. Table 6 shows the post-release defect rate of the software developed with design by contract. Most of the unit tests were written before the actual code, but the contracts

Release	0.1	0.2	0.3	0.4	0.5	Total
Post-release defects	2	1	2	1	0	6
Post-release defects/KLOC	1.18	0.57	0.96	0.63	0	0.70

Table 6: Defect rate

were specified after it because the programmers did not get any instructions from their coach on when the contracts should be written.

4.3.3 Stepwise Feature Introduction

Stepwise Feature Introduction (SFI) is a software development methodology introduced by Back [4] 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 require 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.

4.3.4 Pair Programming

Pair programming is a programming technique in which two programmers work together at one computer on the same task [35]. 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, and the productivity follows the Nosek's principle [28]: two programmers will implement two tasks in pair 60 percent slower then two programmers implementing the same task separately with solo programming.

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 [12]. As it has been frequently reported [12, 13, 22, 27, 36], pair programming also has a great educational aspect. Programmers learn from each other while working in pairs. This is specially 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. Table 7 shows the percentage of the solo-pair work in the three projects of summer 2003 and two of summer 2004, the first number indicates the percentage of pair work.

	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 7: Solo vs. pair %

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.

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 [18], as used in Energi [33] projects, where the origin (pair or solo) of the code is described by comments.

4.3.5 Unit Testing

Unit testing is defined as testing of individual hardware or software units or groups of related units [29]. 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.

4.3.6 Continuous Refactoring and Collective Code Ownership

The most popular definitions for refactoring is given by Fowler [15]: ”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 [31] 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.

We introduce refactoring right after the first short iteration (see Section 4.2.1) and promote it throughout the whole project. Refactoring is introduced early in a project for educational reasons: developers get used to change each others' code and improve on the original design. After the second iteration refactoring becomes a part of the daily routine. After the code satisfies the unit tests it is refactored. During the refactoring the programmers change the structure of the code mercilessly. We are less concerned about new bugs being introduced by refactoring or functionality changes, since the automated test suits should discover these problems. In Python projects the programming language itself promotes refactoring: Python programmers have the tendency to change their working code, constantly making it more precise and efficient. There should be enough time reserved for the refactoring in each iteration. We discussed the time issues in the Section 4.2.1.

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-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.

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.

4.4.1 Configuration Management and Continuous Integration

All code produced in the Gaudi Software Factory, as well as all tests (see Section 4.3.5), 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.

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 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 [3]. 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 4.1.2).

4.4.2 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. Table 8 shows examples from one of the Gaudi projects. 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.

5 Experiences and Observations from Gaudi

The previous practices have been used in 18 projects during a period of four years. We have discussed each issue in detail in the previous section. 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

User Story	Documentation
Story 15: Create parts: User should be able to create new parts in the diagram.	The user can add parts to the diagram by pressing the button containing a UML class pixmap or via the shortcut CTRL+P. The new part is assigned an automatic name, but can be renamed later in the property editor. The new part is added to the first free column to the right in the diagram.
Story 16: Define usage relationships	The user can add usage relationships between parts by first pressing the "Add Usage"- button or the shortcut CTRL+U, and then dragging the mouse from the source part to the target part. This will add a usage relationship such that "source part uses target part". After this, the editor will stay in "Add Usage"-mode until the button is toggled off. Usage relationships can also be added from one part to the part itself. The user can see which parts a certain part uses by selecting it. The parts it uses are colored blue.

Table 8: Documenting user stories

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 an 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.

6 Conclusions

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 environments. This approach requires a large amount of resources and effort but provides an unique opportunity to monitor and study software development in practice.

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 its 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.

The Gaudi process is based on agile methods, specially on Extreme Programming. In this, paper we have discussed the adoption and performance of 12 different agile practices. As we discussed in the introduction, agile methods have been studied by other researchers also. However, 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.

References

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

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

[3] 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.

[4] Ralph-Johan Back. Software Construction by Stepwise Feature Introduction. In *Proceedings of the ZB2001 - Second International Z and B Conference*. Springer Verlag LNCS Series, 2002.

[5] 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.

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

[7] 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.

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

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

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

[11] Eric Bezault. Gobo Eiffel Test. Online at http://www.gobosoft.com/eiffel/gobo/getest/.

[12] 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.

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

[14] Yishai A. Feldman. Extreme Design by Contract. In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2003.

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

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

[17] Hasko Heinecke and Christian Noack. *Integrating Extreme Programming and Contracts*. Addison-Wesley Professional, 2002.

[18] Hanna Hulkko. Pair programming and its impact on software quality. Master's thesis, Electrical and Information Engineering department, University of Oulu, 2004.

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

[20] Eiffel Software Inc. Eiffel in a Nutshel. Online at: http://archive.eiffel.com/eiffel/nutshell.html, 2003.

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

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

[23] 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.

[24] 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.

[25] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition edition, 1992.

[26] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition edition, 1997.

[27] 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.

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

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

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

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

[32] 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.

[33] 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.

[34] Open Source Initiative. Eiffel Forum Licence. Version 2. Online at: http://opensource.org/licenses/ver2_eiffel.php.

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

[36] 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.

ISBN 952-12-1463-5
ISSN 1239-1891