# Evaluating the XP Customer Model and Design by Contract

Ralph-Johan Back, Piia Hirkman, Luka Milovanov
Turku Centre for Computer Science
Åbo Akademi University, Department of Computer Science and IAMSR
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
{backrj, phirkman, lmilovan}@abo.fi

## Abstract

*In this paper we describe one of the series of experiments with Extreme Programming, carried out as a summer project. The focus in this experiment was to try out the XP customer model and Design by Contract. The experiment indicates that the Extreme Programming emphasis on having an on-site customer available during the project improves the communication between customers and the programming team, and markedly decreases the number of false features and feature misses. It also indicates that the systematic use of Design by Contract leads to a low post-release defect rate for the software system built.*

## 1. Introduction

Software development projects tend to be very cumbersome; the process itself is frustrating with constantly changing requirements, the resulting software does not conform to expectations, and deadlines are often overrun. This motivates repeated attempts at finding "the perfect" method for software development. The larger the software to be written is, the more serious is the problem. New methods are also proposed, but it is not so easy to study their impact on software construction, and it can be difficult to find the time and resources needed to improve these methods. The *Gaudi Software Factory* is an effort to provide an environment where software methods can be tried out, and at the same time to establish an environment where academic and industrial needs and interests can meet.

Gaudi is an experimental software factory which aims at developing and testing new software development methods in practice. The influence of new methods on the time, cost, quality, and quantitative aspects of developing software is studied in a series of controlled experiments, one of which is presented in this report. A characteristic of Gaudi is that the programmers 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 just employed in Gaudi throughout the experiment. A typical experiment is the development of a new software product (or a new release of an existing product).

The setting of this series of experiments follows a pattern. An experiment has restricted resources (4-6 programmers), goals, and time (3-6 months). Product development is the main activity, programmers are assisted by a coach, and well-defined software practices are followed in the construction process. For the moment, the basic software process that is followed in Gaudi (Gaudi process) is based on Extreme Programming (XP). We choose this method because it is flexible and light-weight and it can be taught rather quickly to the students [4].

We started with a basic set of XP practices: pair programming, unit testing, refactoring, short iteration cycles, and light documentation, to name a few [3]. This XP toolset has been extended with *Stepwise Feature Introduction*[2] (SFI), an experimental programming methodology, and the use of various programming languages and GUIs.

Altogether, we have carried out 15 software construction experiments in Gaudi to this day. This paper portrays one of these experiments, the development of a personal financial planner.

## 2. Goals for the Experiment

We had essentially three main goals for the experiment: to test the *XP customer model*, to test how *Design by Contract* worked in practice, and to test *Stepwise Feature Introduction* with a statically typed object-oriented language. The result for the last goal will be reported in a separate paper.

### 2.1. XP Customer Model and the Planning Game

One of the reasons for the growing popularity of XP in the industry is its stress on customer's satisfaction. However, in our previous experiments we failed to have an on-



IEEE COMPUTER SOCIETY

site customer for the product development. Trying out the XP customer model was therefore one of the main objectives of this experiment. A secondary objective was to see if we could make time estimations of the programmers' work. XP has a notion of *planning game*, a part of which are time estimations and release planing. However, this planning game requires an active customer, so we have not been able to try out the planning game before.

### 2.2. Design by Contract and Eiffel

Design by Contract [13] is a systematic method for making software reliable (correct and robust). A system is structured as a collection of cooperating software elements. The cooperation of the elements is restricted by *contracts,* explicit definitions of obligations and guarantees. The contracts are *pre-* and *postconditions* of methods and *class invariants*. These conditions are written in the programming language itself and can be checked at runtime, when the method is called. If a method call does not satisfy the contract, an error is raised.

Design by Contract was the second objective of the experiment, so we choose Eiffel [12] as the programming language of the project, because it 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) [8]. 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.

Eiffel's documentation[12] claimed portability of the language. We also wanted to test how portable Eiffel really is, so we planned to release versions of the software for both the Windows and the Linux platform. The development team in our project was using ISE Eiffel Enterprise version 5.3 for Linux. We also had one machine with the same version of Eiffel Enterprise for Windows for building Windows executable.

Unfortunately, ISE Eiffel had no original *unit testing* framework. Unit testing is an essential part of the XP 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 [7]. Gobo Eiffel Test is distributed freely under the terms and conditions of the Eiffel Forum License [15].

### 2.3. Stepwise Feature Introduction

*Stepwise Feature Introduction* is a software development methodology introduced by Back [2] 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. SFI 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. XP does not say anything about the software architecture of the system. SFI 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 and we wanted to test SFI with a statically typed object-oriented language like Eiffel.

### 2.4. The Product

As in the previous experiments, we wanted to keep the developers concentrated on their work and not be disturbed by the experimental nature of the project and product. To achieve this we decided to develop a piece of software which would not seem too experimental to the programmers – in this case a personal financial planner. The original idea was that the user would first create a desirable but realistic version of her/his financial future. As time passed, financial transactions were carried out. They would be recorded in the system, and the future financial scenario would be updated based on the new information.

The features required of this product type include tracking of actual events (manually or automatically), planning (such as budgeting and creating scenarios), and showing future scenarios. The system should also provide advice and warnings, and support multiple user profiles. These features should exist for different categories of financial data, including basic transactions (income and expenditure), investments, and loans. The system should provide ways for data input, calculation, presentation (numerical and graphical), customization, accessibility, and persistency.

Based on priorities provided by the customer, the most important (and basic) features were selected as the ones to be implemented during the first three-month-period. This limited the functionality of the product into manually recording basic transactions, providing graphical presentations, and creating a budget/scenario. Naturally, saving data and printing were also included in the requirements.



IEEE COMPUTER SOCIETY

## 3. Setup of the Experiment

The experiment was set up as a typical Gaudi development project. We describe below the central parts of this: the product development team, the resources available, and environment for the experiment.

### 3.1. The Team

The composition of the team was somewhat different from earlier experiments in Gaudi [4]. This time, the team included two professors, one in Computer Science and the other in Accounting Information Systems, that acted as managers, one Ph.D. student acting as both coach and project manager, and four undergraduate students, employed as programmers. Additionally, another Ph.D. student with a background in Accounting Information Systems played the role of the customer. As stated earlier, the separation of the ”business representative” from the other roles was the biggest change in the standard Gaudi team composition. This allowed us to implement the XP recommendation for having an on-site customer [6].

The programmers were third-fourth year computer science students employed for the project. Two of them had experience in previous Gaudi experiments, but the other two were unfamiliar with this software process. On other issues their experience level varied. There was only one who had not developed software in a team before, one admitted having previous experience in both unit testing and Design by Contract, and one who had a basic knowledge of Eiffel.

### 3.2. The Schedule

The schedule for the project was defined by two factors. First and foremost, the students were employed for the summer only. Additionally, the experiment was to be comparable with the other Gaudi projects running at the same time. That is, we set ourselves a strict three-month deadline: the product was to be released by August 31 at latest.

We decided not to split the project into three stages, training, programming and debugging, as we did before. Instead, we held tutorials on the Gaudi process for the programmers for two weeks before they started working, and they then programmed from the beginning to the last day of the project, 40 hours a week, according to the XP principle.

### 3.3. Environment

As during the previous Gaudi experiments, all programmers were sitting in the same room, arranged according to the advice given by Beck [6]. The four programmers sat by a big table in the middle of the room. Four computers were placed so that the work stations formed a clover-like square. Since the team consisted of only two pairs, there was no special machine for integration. There were no separators which could impede communication. There was a bookshelf, a white-board and a noticeboard in the room. Outside this room was a recreation area with a coffee maker etc. that was shared with four other groups of programmers. We also decided to use the same platform as in the previous projects – Linux. The most used software tools were EiffelStudio [10] (see section 2.2), XEmacs editor, CVS and a CVS front-end – Cervisia. All the software was open source except for the EiffelStudio and Windows 2000.

| Writing stories | With team | Testing | Idle |
|---|---|---|---|
| 2.5 | 3 | 2.5 | 92 |

**Table 1. Customer involvement (%)**

## 4. Running the project

As mentioned earlier, the project was preceded by a short training period for the programmers. The actual development work followed quite closely an XP style project flow: the iterations were short, planning games were played, and so forth. The following sections provide some details on the actual project phases and elaborate more on the newer XP elements present in the experiment.

### 4.1. Development

At project kick-off, the managers, the coach, and the customer had a meeting where the software to be developed was discussed in general. Based on this discussion, the customer created a more detailed idea of the product. Since the development team had already been trained,the planning games could begin immediately. The main interactions took place in the development team itself, with the coach, and with the customer, all following XP practices. The development work itself proceeded in XP style short iterations.

### 4.2. Customer Involvement

The customer worked basically as an on-site customer. She 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 [5], 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”.

There was no complete requirement specification of the product to be built. Communication between the customer


IEEE COMPUTER SOCIETY

and the development team took place both through instructions given on paper and during face-to-face meetings. The most comprehensive written instructions were formulated as customer stories which followed the guidelines given by the XP practice: they stories were about three sentences of text in the customers terminology without technosyntax [1, 11]. The customer wrote 15 stories, which were divided into six different releases as a joint effort between the customer and the coach.

The customer stories were written in the form given by Beck [6] providing information about the title, date, status and a short description of what the user should be able to do after the story was finished. The division of the product's features into the stories was made by the customer based on an intuitive idea about what meaningful chunks the system could be divided into.

Since the customer stories did not provide very detailed guidelines for the desired features, the development team and the customer had a meeting at the beginning of every iteration, during which the requirements were further specified. These meetings usually took about an hour. During the meetings, some of the time was used to make sure the team understood the application logic correctly, but mostly the discussions concerned aspects of the user interface. The scantiness of time spent on logic issues could stem from the fact that business logic in the system was not very complicated; no experts are required to know how day-to-day personal finances should work.

Table 1 shows how the customer's time was spent on project issues. Apparently, being an on-site customer does not increase the customer's work load very much. One might even wonder whether an on-site presence is really necessary based on these figures. However, the feedback from the development team shows that an on-site customer is very helpful even though the customer's input was needed rather seldom.

### 4.3. Planning Game

This Gaudi project was the first where an actual planning game and release estimation took place. The goal of the planning game was to choose the stories to be implemented, rather than taking all of them and negotiating about a release date and resources to be used (that is, planning by time [6]).

The initial release planning was made without the programmers, because the coach had more experience in making estimates at that point of time. The coach and the customer discussed the schedule and priorities of the 15 stories written by the customer. The coach doubted if the project's schedule (10 weeks after training) would allow programmers to implement all of these stories. Consequently, it was decided that the story with the lowest priority (loans, N12, table 3) was left to the last release and would be implemented only if there was enough time. The other stories were divided into five different releases of equal lengths based on the priorities given by the customer and the time estimates provided by the coach. Each release was defined to take two weeks and included different amounts of stories depending on how complex they were.

| release | date | release | date |
|---|---|---|---|
| 0.1pre | June 26 | 0.1 | June 30 |
| 0.2pre | July 8 | 0.2 | July 10 |
| 0.3pre | July 22 | 0.3 | July 28 |
| 0.4pre | August 6 | 0.4 | August 14 |
| 0.5pre | August 21 | 0.5 | August 22 |

**Table 2. Release frequency**

Since the time frames of the releases were very short, the stories chosen for a release were actually the same that were chosen for an iteration. As we mentioned above, the development team and the customer met in the beginning of each iteration and discussed the features to be implemented. The team estimated whether there was a need for reconsidering the temporal cost of the iteration in the presence of the customer, after which the developers proceeded to break down the iteration into tasks among themselves.

The planning game then continued with task planning. The programmers split the customer stories into task cards. One story normally produced 3-4 tasks. When the tasks were written, they were estimated. The developers were writing on each card how many hours it will take for one programmer to implement the task. Based on these task estimations, the customer stories and the whole release were estimated as well. Then we added 50% to the time estimation for refactoring and debugging. This gave us estimation in human-hours. The actual time for estimating the date of the release was calculated following the Nosek's [14] principle: two programmers will implement two tasks in pair 60 percent slower then two programmers implementing the same task separately with solo programming.

### 4.4. Iteration Cycles

Our project consisted of five two-week long iteration cycles. Each cycle was finished with a small pre-release of the software. The pre-release was published on the project's web page and given to the customer for testing. It took an average of three working days (see table 2) to fix the pre-release according to the customer comments and release a "customer approved" package. Tables 3 and 4 show the difference ($\Delta_h$ and $\Delta\%$) between the developers' estimations $E_h$ and the actual time $A_h$ it took them to implement stories and releases. Figures 1 and 2 show the story and release es-


IEEE COMPUTER SOCIETY

| Story | $E_h$ | $A_h$ | $\Delta_h$ | $\Delta$ % |
|---|---|---|---|---|
| 1 | 17 | 12 | 5 | 30% |
| 2 | 14 | 12 | 2 | 14% |
| 3 | 12 | 6.5 | 5.5 | 46% |
| 4 | 20.5 | 17 | 3.5 | 17% |
| 5 | 7 | 5 | 2 | 29% |
| 7 | 19 | 21 | -2 | 11% |
| 8 | 6 | 5 | 1 | 17% |
| 9 | 14 | 2 | 12 | 86% |
| 6 | 48 | 54 | -6 | 12% |
| 10 | 69 | 49 | 20 | 29% |
| 11 | 29 | 31 | -2 | 7% |
| 12 | 80 | – | – | – |
| 13 | 53 | 64 | -11 | 21% |

**Table 3. Customer stories estimations**

| release | $E_h$ | $A_h$ | $\Delta_h$ | $\Delta$ % |
|---|---|---|---|---|
| 0.1 | 70.5 | 52.5 | 18 | 25% |
| 0.2 | 39 | 28 | 11 | 28% |
| 0.3 | 48 | 54 | -6 | 12% |
| 0.4 | 98 | 80 | 18 | 18% |
| 0.5 | 53 | 64 | -11 | 21% |
| 1.0 | 308.5 | 278.5 | 30 | 10% |

**Table 4. Release estimations**

timation errors respectively. Obviously, it was easier to estimate the smaller stories. In some cases the big estimation errors were caused by issues in the Eiffel programming language and by the developers inexperience. For example, the story asking to print some data was underestimated: printing in Eiffel turned out to be rather complicated. On the other hand, sorting tasks were overestimated because the developers did not know about sort routines in the Eiffel libraries.

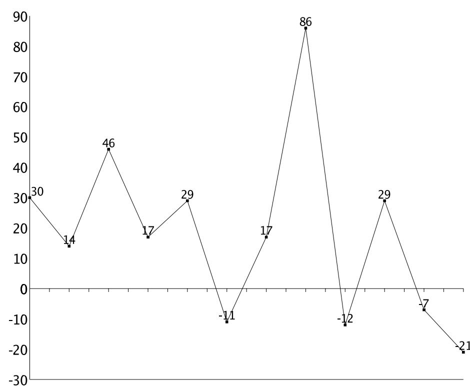


**Figure 1. Story estimation error (%)**

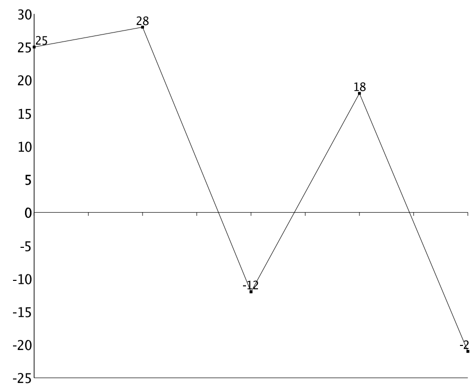


**Figure 2. Release estimation error (%)**

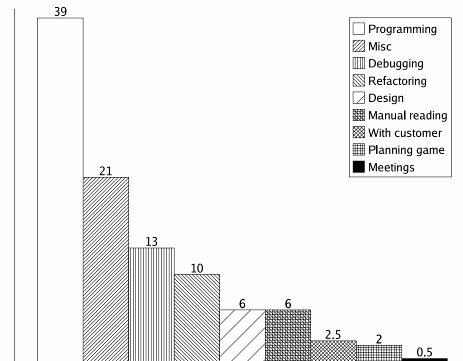


**Figure 3. Activities distribution (%)**

Figure 3 shows the percentage of time distribution for the different activities of the development team. Table 5 shows the percentage of pair-solo work distribution. The data in these figure and table is total from the project. There seldom was a need for great changes after testing. This had mostly to do with the fact that the logic of the product was rather simple.

## 5. Deliverables

In this section we compare the final release of the software to the initial specification and show some basic metrics of the project.

| | Programming | Refactoring | Debugging |
|---|---|---|---|
| Solo | 21 | 23 | 73 |
| Pair | 79 | 77 | 27 |

**Table 5. Pair vs Solo (%)**



IEEE COMPUTER SOCIETY

### 5.1. The Software Product

The delivered product includes the very basic features of a tool for planning personal finances. It enables creating a financial history (noting income and expenditure as one-time or repetitive transactions), drawing pre-defined graphs, and budgeting.

Thus, the system includes most of the functionality specified in the restricted requirements which excluded automated data input, loans, and investments. In the realized functionality, there are issues that require ”re-engineering” and further development. For example, the GUI is considered to be somewhat old-fashioned and needs streamlining. Also, the product has only reached the stage where real value-adding functionality can be created. The basis is there, but further iterations are needed to reach a fully functional – and useful – version.

FiPla is an experimental tool and as such is not intended for general use. However, it is freely available under the GNU General Public License to all interested in it.

### 5.2. Project Metrics

Collecting all possible kinds of metrics is an essential part of a software engineering experiment. Table 6 shows some basic metrics collected after the final release of the product. Some values in the tables are rounded up to the integers for the better readability. By total work effort in the table we mean the number of actual hours needed to implement all of the stories of a release from table 2 multiplied by four – the number of developers. The developers stopped writing tests for GUI after the first release. This explains the decrease in the number of tests for the second release (table 6). Structuring the software system into the layers was done according to the release plan. Each new release was introducing a new layer into the software system. The five releases of the product formed the following five SFI-layers:

**Layer 1. (Basic):** Manually recording one-time transactions (incoming and outgoing), display, save for multiple users.

**Layer 2. (Calc):** Calculating sums both according to the type of transaction and time periods.

**Layer 3. (Repetitive):** Manually recording repetitive transactions.

**Layer 4. (Graph):** Displaying three types of graphs.

**Layer 5. (Prognosis):** Creating a financial plan.

## 6. Project Experiences and Impressions

As in the earlier projects, we got valuable feedback on the subjects for the experiment. There was both positive and negative feedback. When we get positive feedback on a method, especially in a series of experiments, that normally means that we will have a definite benefit from its further use and that the method is a candidate for incorporation into the standard Gaudi process. On the other hand, negative feedback usually provides good guidelines for improving the method. Below we present the most important findings and impressions of the project.

### 6.1. Customer Model

One problem with the customer model we used was revealed by the fact that among the activities of the programmers (Figure 3), the second largest category was ”miscellaneous”. According to the team, the reason for that was the following work order.

After a pre-release was sent to the customer, the team was trying to find bugs in it, while waiting for the customer's response. Since the acceptance testing was not characterized as the developers' task in this project, this time was marked as ”miscellaneous” hours. While the team was correcting bugs found by the customer in a pre-release, the amount of work was not enough to keep all the developers busy, that again caused the ”miscellaneous” hours. The developers did not start working on the next release before fixing the previous one. This was due to an unfortunate misunderstanding. This was required in one iteration since the customer could not make a decision straight away, but the rule was not supposed to be generalized to all iterations.

The developers gave some suggestions for decreasing those ”miscellaneous” hours and for decreasing the risk of false features implementation by increasing the communication with the customer and having the customer more involved in the team work. The first suggestion was to plan the next release and to perform the planning game together with the customer. When all the stories are implemented, the customer should briefly go through the functionality of the system together with the team. After that, if no feature misses of false feature were found (otherwise the team fixes it), the team together with the customer plans the next release. Finally, the customer performs the acceptance tests, while the team starts implementing the new set of stories.

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. Even though the customer was available throughout this experiment, she was physically only ”rather close by” (that is, in the same building). Consequently, the overall impression 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.



IEEE COMPUTER SOCIETY

| | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | Total |
|---|---|---|---|---|---|---|
| LOC | 1694 | 3441 | 5517 | 7100 | 8572 | 8572 |
| Test LOC | 571 | 983 | 2174 | 2347 | 2548 | 2548 |
| Total LOC | 2265 | 4424 | 7691 | 9447 | 11120 | 11120 |
| Classes | 11 | 23 | 37 | 52 | 59 | 59 |
| Test classes | 9 | 10 | 20 | 23 | 25 | 25 |
| Methods | 71 | 122 | 171 | 256 | 331 | 331 |
| Test methods | 50 | 68 | 157 | 167 | 177 | 177 |
| LOC/Class | 154 | 150 | 149 | 137 | 145 | 145 |
| LOC/test class | 63 | 98 | 109 | 102 | 102 | 102 |
| Methods/class | 7 | 5 | 5 | 5 | 6 | 6 |
| Test methods/class | 6 | 7 | 8 | 7 | 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 6. Collected data for all releases**

## 6.2. Eiffel and Design by Contract

We got a lot of positive feedback about using Eiffel. First of all, according to all developers, Eiffel was easy to learn. However, not all of them agreed with the suggestion to teach Eiffel in an introductory programming course. They also said that writing good pre- and postconditions and class invariants is not an easy task. Another feature of Eiffel they really appreciated was its readability. Eiffel code written according to the Eiffel standard [9] was very easy to read. *"Eiffel code seems easier to read than Python code."* – commented two of the developers that had participated in our earlier Python project.

All of the Eiffel code in our project was written using Eiffel Studio, which is a very convenient IDE. According to the developers, most of all they appreciated the debugger in Eiffel Studio. However, there was a sense of incompleteness in Eiffel, as one of the developers wrote in the diary: *"The developers of Eiffel seem to spend more time bragging about how good their incomplete language is than actually trying to make it any better"*.

Design by Contract worked well in our project and its use was one of the reasons for the low defect rate. *"All the tests written (to a complete code) always pass and the tests that don't pass have a bug in the test itself"* – commented the developers in the middle of the project. Most of the bugs were caught with the help of preconditions, when a routine with a bug was called during unit testing.

Most of the unit tests were written before the actual code, but the contracts were specified after it. It would be interesting to study the relationship between Design by Contract and "test first" approach in the next experiments. In the beginning of the project we left out automatic GUI testing and left it to the customer. It was also difficult to come up with non-trivial pre- and postconditions during the graphical user interface development.

## 6.3. Stepwise Feature Introduction

This project was very fruitful for the evaluation of SFI. As we mentioned above, this was for the first controlled time when SFI was applied with a statically typed programming language. Using SFI with Eiffel showed us aspects of the methodology that we could not see when applying SFI with Python. These discoveries allow us to improve this experimental methodology and make its practical application more systematic. The explanation of these findings will be discussed in a separate paper.

## 6.4. Planning Game and Time Estimation

Having all customer stories written allowed us to make time estimations. Time estimation turned out to be rather easy in this case. The accuracy of the estimations depends, of course, on the experience of the developer. Experience of the particular programming language seems also to be more important than experience in estimation.

In our project the developers did not sign up for the tasks during the planning game. A task was estimated by all of them and a pair was signing up for the concrete task during the short iterations. Having individuals signing up for the tasks during the planning game and being responsible



IEEE COMPUTER SOCIETY

for them during implementation would probably have decreased the errors in the time estimates.

### 6.5. Software Quality

Comparing this project to our previous experimental projects, which were carried out in a very similar setting but using Python instead of Eiffel and without the on-site customer, the software produced in this project had a much lower defect rate (0.70 for this project, compared to an average of 5.2 for previous projects).

Design by Contract and especially preconditions helped to catch many bugs during the development. While such features of the Eiffel programming language as lack of public variables and static typing prevented introducing many bugs into the software system. Programmers found the Eiffel's documentation very poor. This also held for the SFI method, which is experimental and has almost no practical documentation for programmers. This forced the developers to do a lot of spikes, testing and trying before writing the actual code. According to the developers, this caused the resulting code to be more bug-free.

Another reason for the low defect rate was the nature of the application. The tasks given to the programmers were relatively easy in their opinion. The system they were developing was easy to understand due to the nature of the application. Finally, the implementation of a new pre-release never started before releasing the software in the "customer approved" package. While the customer was testing the pre-release alone, four programmers were also testing the same system. This meant that some minor bugs were fixed without documenting them as bugs and without the customer mentioning them.

## 7. Conclusions and Future Work

The objectives of the project were met both on the product and the research level. The product reached the functional level that was set at the end of the overall planning period for the project. As stated, it still requires further development, but this was known from the beginning.

Experiences on a clearer customer model and customer stories were altogether very positive, but they also gave directions for considering a few aspects, particularly the physical location of the customer and the programmers.

The use of a formal method, Design by Contract in our case, dramatically decreased the post-release defect rate of the software system. The project also showed ways to improve SFI. There were improvements to be made when using the method with a statically typed language.

Experimenting with these issues should be continued, giving room for the experiences gained during this project. Software development may never become an art of perfection, but it may well be worth to strive for it.

## References


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

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

[3] R.-J. Back, L. Milovanov, I. Porres, and V. Preoteasa. An Experiment on Extreme Programming and Stepwise Feature Introduction. Technical Report 451, TUCS, 2002.

[4] R.-J. Back, L. Milovanov, I. Porres, and V. 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.

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

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

[7] E. Bezault. Gobo Eiffel Test. http://www.gobosoft.com/eiffel/gobo/getest/.

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

[9] Interactive Software Engineering. An Eiffel Tutorial. Online at: http://docs.eiffel.com/general/guided_tour/language/tutorial-00.html, 2001.

[10] Interactive Software Engineering. EiffelStudio: A Guided Tour. Online at: http://docs.eiffel.com/general/guided_tour/environment/, 2001.

[11] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2001.

[12] B. Meyer. *Eiffel: The Language*. Prentice Hall, second edition edition, 1992.

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

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

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





IEEE COMPUTER SOCIETY