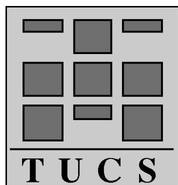
An Experiment on Extreme Programming and Stepwise Feature Introduction

Ralph-Johan Back
Luka Milovanov
Ivan Porres
Viorel Preoteasa



Turku Centre for Computer Science
TUCS Technical Report No 451
December 2002

ISBN 952-12-0979-8
ISSN 1239-1891

Abstract

In this paper we describe our first of the series of experiments with Extreme Programming during a summer project. We also discuss how XP can be used as a software process framework for performing practical experiments in software engineering. We show how the main features of XP help to minimize some problem when trying to perform such experiments in university environment.

Keywords: Extreme Programming, Software Engineering Research, Stepwise Feature Introduction, Software Production in a University Environment

1 Introduction

The Software Engineering discipline studies how to build large software systems that fulfill the users requirements, are reliable and are constructed on time and budget. This includes the study of many different concepts and techniques used in software development: software process models, modeling notations, programming languages and methods, testing and validation strategies, CASE tools, etc.

One of the problems that hinders the research and improvement of these techniques is the difficulty to perform significant controlled experiments. Many methods, such as Extreme Programming (XP) [3] or the Unified Modeling Language (UML) [18], 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 team performances.

On the other hand, universities employ highly qualified research personnel that can employ considerable time to study better ways to build software, without the pressure of having to release new software products to the market. In this sense, a university setting could be the ideal place to perform practical experiments and test new ideas in software engineering.

However, researchers also find difficulties while testing new ideas in practice. First, it is possible that an experiment does not reflect the conditions found in a development company since researchers do not need to develop actual products. Secondly, university experiments must usually be 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 the project may vary. There is also a high turnover rate as the students graduate and quit. Finally, although there is no market pressure, a researcher does not have unlimited funds, so it is necessary to optimize the costs of the experiments.

In this paper, we discuss how Extreme Programming can be applied as the base software process to perform practical experiments in software engineering in a university context. We think that many of the XP characteristic features help us to circumvent some of the problems described above. We also discuss the execution of an experiment using XP.

In this experiment we employed six undergraduate students under three summer months to develop an advanced text editor. We used XP as the base software process and we tested a new programming methodology currently under development: the Stepwise Feature Introduction (SWFI) [1].

This project was the first in the series of experiments targeted to study new software development methods and their impact in the time, cost, quality and quantity of the software produced using these methods.

The paper is divided as follows. Section 2 contains an overall description of the project: the development team, the product to be built and different software techniques to apply. The following three sections describe the three main phases of the project: the learning phase where the team got acquainted with the different development techniques, the production phase where the actual software was developed and a short clean-up phase before the project was finished. Section 8 describes what was delivered in a quantitative way while Section 9 gives out qualitative analysis of the project. We discuss our final conclusions and guidelines for future work in Section 10.

2 The Project

Our experimental project ran during three summer months and it was carried out by ten persons. One professor acted as customer, three Ph.D. students performed tasks of project leaders, coaches and customers as well, and six undergraduate students were involved as programmers. The major goal of the project was to test the SFI method in practice. Since we are considering SFI as practical methods for developing software we saw it necessary to obtain practical results from our methods by applying them to software construction.

Another subject of the experiment was extreme programming. Since we were going to produce software, we needed a well-defined software process to start with. Our choice was XP, but since we never had experience with it, extreme programming itself became a subject of experiment in our project. Therefore, we decided to collect our experience on how the XP concepts will work in university environment, and to study better ways for building software. We considered it necessary to have a well-defined product to build in the context of the project so the efforts of the developers were aimed at a non-trivial and tangible piece of code. This helped us to keep the students focused rather then on the experiment itself, on building a concrete and well-defined software product.

One more challenge was to try out new ways for teaching students and to show whether a non-trivial piece of software can be produced by unexperienced students in a short time span.

3 The Product

The product to be built was an outline editor. An outline editor is a text editor for processing *outlined texts*. Outlined text is a text which consists of indented lines. The basic rule for the indentation of the lines is that each line can be indented at most one level to the right from the previous line. An example of the outlined text

is a Python [13] script. A line followed by indented lines is called *father* and the indented lines are called *sons*.

A line with all its sons and sub-sons is called *an item*. An item which does not have sons is called *atomic*, otherwise it is called *composite*.

Here is an example of the outlined text which consists of 6 items:

```
1:    class MyClass:
2:        'A simple example class'
3:        i = 12345
4:        def f(x):
5:            return 'hello world'
6:    x = MyClass()
```

The first item (lines 1-5) is composite. It consists of 2 atomic items (2, 3) and one composite (4-5). The last item (line 6) is atomic.

3.1 Basic features

The complete requirements of the editor can be found in [2]. The outline editor was to provide facilities for browsing, editing and storing outlined text with some addition functionality for processing Python program code.

1. Multiple window interface
2. Synchronous multiple view of the same document
3. File operations: new, open, close, save, save as, recent file history
4. Edit operations: cut, copy, paste, drag and drop find, replace, undo, redo, indentation
5. Collapsing and expanding composite items
6. Python: syntax highlight, syntax checking, code execution

4 Techniques

Any piece of software is the result of the application of many different skills. In this project we tried to experiment with several techniques in order to built reliable software under a tight schedule. However, in order to get started with the experiment we needed a well-defined software process that will serve as a framework for experimenting with different techniques. On the other hand such process should be easy to learn and implement in short time and, what was the

most important, it should still lead us to the production of the software. The choice we made was Extreme Programming.

The programming language of the project was Python together with Python *Tkinter* [11] GUI package. Although Python is portable, we decided to test the software only under the Linux operating system. To keep track of the project assets we chose the CVS version control system.

The techniques we experimented with were Stepwise Feature Introduction and Design by Contracts. Also the ExtremeProgramming itself was the subject of the experiment, since that was our first experience with it.

Most of the students participated in the project were not familiar with the techniques mentioned above. In fact, only one of the students had some experience with Python and Tkinter. None of them was familiar with XP or SFI.

The software produced has been released under the GNU General Public License [10]. This license enforces the freely availability of the source code. In our case, this avoids possible conflicts in the distribution of the deliverables and the continuation of the project by a different team of students.

4.1 Extreme Programming

XP – a lightweight software methodology 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, on-site customer, promoting team communication and pair programming. XP is a recent method and there are few published independent experiences at how it works in practice.

One of the features that we appreciate most in XP is its simplicity. First of all, XP is easy to learn. That was an important for us since we really did not want to spent a lot of time teaching the students. We wanted to get project running as soon as possible.

Another reason for choosing XP approach was its short interaction cycle that facilitates the creation of running software in a short period of time. None of the students had any experiences with XP.

4.2 Stepwise Feature Introduction

Stepwise Feature Introduction is a software development methodology introduced by R.-J. Back [1] based on incremental extensions of the object-oriented software system with only one new feature at a time. The SFI methodology has much in common with the original *stepwise refinement* method. The main difference to stepwise refinement is the emphasis on object-oriented programming and the bottom-up software construction approach.

According to SFI methodology, a software system should be built in thin layers where each successive layer introduces new feature to the system and does not break the functionality implemented in previous layers. Each layer together with its ancestors represents a *running* system with the functionality that extends the previous layer. The basic layer is a system with minimal functionality.

Each successive layer is described as a collection of classes which are extensions of classes from the previous layers. The extension is implemented using inheritance (also multiple inheritance), delegation or forwarding. The inherited methods are supposed to remain unmodified or to be redefined such that the new code achieves the same effect as the inherited one, plus possibly some additional effect on newly introduced attributes. All class members, once introduced, should be present in the extensions of a class through all of the higher layers.

Each public class method once written should remain through all of class extension either unchanged or rewritten in such a way that the method modification preserves the effect of the old method on the new attributes.

The SFI methodology is still under development and it is part of a research project. Therefore, by applying the Stepwise Feature Introduction methods in a real software project, we expected a lot of practical feedback that could help us to evaluate this new methodology.

Further on, this technique seemed to be suitable to use in combination with unit testing because the previous introduced features are suppose to pass a collection of test cases, and after introducing a new feature the tests still should be passed. Obviously, none of the students was familiar to SFI.

4.3 Python

Python [13] is an object-oriented, interpreted and highly dynamic language. It features an elegant and clear syntax, powerful built-in data types like strings, tuples, lists and dictionaries and dynamically-typed variables. It is class-based, supports multiple inheritance and garbage collection. There are Python interpreters available for all major platforms and Python byte-code is portable across platforms.

There is no standard GUI for Python but there are several tool-kits that are available in several platforms, like Tkinter (Win, X11, Mac), Qt (Win,X11) or wxPython (Win,X11).

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 project and they were confirmed along the project. Only one of the students had some experience with Python.

4.4 Version Control System

To keep track of the project assets we decided to use CVS. CVS is the Concurrent Versions System, the dominant open-source network-transparent version control system [4]. Since none of the students was familiar with CVS, they really appreciated the Cervisia [9] graphical front-end.

The CVS central repository is also an important source of data for analysis of the project's progress since all revisions of all assets are stored there together with the responsible person and checking date and time. None of the students was familiar to CVS.

4.5 Design by Contract

Design by Contract [15] is a systematic method for making software reliable (correct and robust). This method proposes constructing systems as structured collections of co-operating software elements. The co-operation of the elements is communication on the basis of *contracts* which are explicit definitions of obligations and benefits.

The contracts are pre- and post-conditions of methods. These conditions are written in the programming language in the body of the method and they can be checked when a method is called. If a method call does not satisfy the contract, an error is raised. The only programming language which specifically supports design by contract is Eiffel [16].

Since Python does not support pre- and post-conditions, we decided to give them as comments inside the methods bodies and as Python *assert* statements whenever is is possible. None of the students was familiar with Design by Contract method.

5 The Schedule

Finding time to meet and work together is the most frequent problem when we consider students as developers of a software project [19]. We managed to avoid this problem by employing students full time for the whole project.

Our project ran from the end of May to the beginning of September 2001. The project had three clear phases: training, programming and cleaning up. The first meeting of the project was held the 15th of May when all the members were introduced to each other. The overall working conditions and schedule were discussed at the same meeting. The project ended on September the 8th.

The training phase happened during first weeks when the Ph.D. students held a series of short tutorials to introduce the new techniques to the undergraduate

students. This kind of tutorial week seems as a good start for the project. Some of the students started working full time on the project from the second week, while others joined on the third week. The students started working on the editor but they were still learning Python, CVS, etc.

The students worked in the laboratory five days a week from 9:00 to 17:00. They did not work overtime. Some of the students had their "private projects" and did their own small exercises in Java or C++ out of the main working hours.

The second phase started the 21th of June with the introduction of the SFI methodology. A new CVS module was created and the first release of the first layer appeared in the repository by the 26th.

The last phase started around the 22th of August. The students were told to stop introducing features in the editor and start "polishing" the project. The focus was on user validation of the editor, code reviews and written documentation.

The final meeting of the project was held the 30th of August. The students continued working irregularly during week 36. Finally, a technical report written by the programmers was delivered on Saturday 8th of September. On Monday the 10th, the administrator made a backup of the CVS repository and changed the root passwords on the machines.

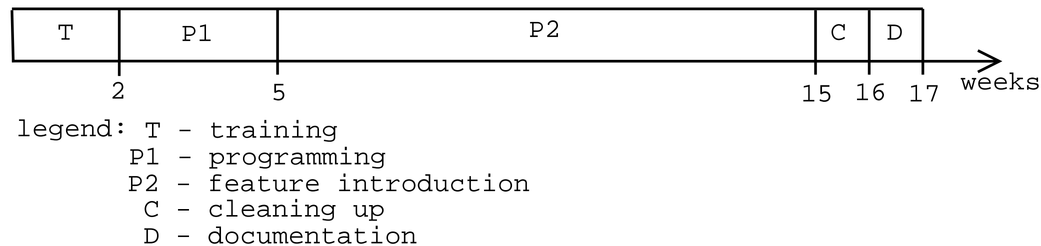


Figure 1: Project's timeline

6 The Room and the Equipment

The six programmers were siting in the same room. The room was arranged according to the advice given by Beck [3]. There was a big table in the center with 4 computers for pair programming and other tables against the walls for personal use. There were no vertical separators or cubicles. The room also had a bookshelf, a food table with a coffee maker and a white-board. The programmer were allowed to have necessary meeting in another room with more white-boards, where in fact, they held all the meetings.

The computers used were 8 brand new 1GHz PC computers using ÅA Linux. ÅA Linux is a custom version of Redhat Linux adapted by the computing center of Åbo Akademi. One of the computers acted as a file server (NFS), repository (CVS) and web server (Apache). There also were three laptops in the room (one with ÅA Linux, one running OS X and one running Windows 98) but they were used seldom, rather for presentations.

The most used software tools were Python 2.1, the XEmacs editor and the Cervisia CVS front-end. Occasionally they used the Together Control Center 4.2 tool to create UML diagrams. The written documentation was produced with LaTeX. All programmers used the KDE desktop.

All the used software is open source except for the Together Control Center. However, Together Soft has a generous academic license program and provided the tools free of cost.

All Linux applications store the user's preferences in the user's home directory that was shared via NFS. This means that the users could roam from one computer to another (and from solo to pair programming) freely and instantaneously, retaining all their files and application preferences.

The cluster worked flawless and the only down-time during the three months was produced by a power cut.

The programmers were satisfied with the room and the equipment but complained for the lack of windows and the need of more white-boards.

7 Three Phases of the Project

We managed to keep the phases of our project according to the schedule described in section 5. Furthermore, we got the software ready at the end of the project. In this section we describe in details the progress of each phase.

7.1 Fist Phase: Training

The core of the development team was composed of six undergraduate computer science students. Most of them had completed their second year in the computer science/engineering curriculum. They had basic courses on programming but none of them was familiar with any of the more advance topics that we wanted to experiment in the project such as Python, Extreme Programming, or Stepwise Feature Introduction.

Since the length of the project was only three months, it was clear that our first challenge was to teach as fast as possible the skills needed for the rest of the project. We decide to use short tutorials for that purpose. The tutorials were held by the Ph.D. students during the first two weeks. All tutorials were imminently

practical. At the end of each tutorial the students were given selected bibliography to read and small assignments to complete in pairs.

tutorial	numbers	total hours
Python	2	4
Tkinter	1	2
CVS	1	2
Extreme Programming	1	2
SFI	1	2
Design by Contracts	1	2
requirements specification	1	2
all tutorials	8	16

Table 1: Tutorials

The third week was marked by the introduction of pair programming. The students were split into pairs in such that one student in a pair was more experienced than the other. Each pair got concrete task to perform during the week. The tasks were described using UML class diagrams annotated with detailed descriptions of the methods.

The students continued practicing their programming skill, learning Python and getting acquainted with XP during three weeks. During this time, the programmers worked on the editor, but there was little emphasis on its design or the final quality of the code since the introduction of the second phase meant to start the editor again from scratch.

7.2 Second Phase: Stepwise Feature Introduction

This stage was the main and the longest stage and it took nine weeks of the total three months of the project. Once the students gained some experience with Python and the XP paradigm, they began programming. During this time the project was carried out according to the guidelines introduced in the training phase. The students started their working week normally with a short meeting and division into pairs. At the very beginning of this phase the meetings and pair divisions were organized by the Ph.D. students. Later on, the students performed those tasks independently. The members of pairs shifted each week.

The iteration planning was substituted with layering. In our case we decided to use only *sequential layering* and not to use *feature combination* [1]. First it was decided what functionality the new layer should have, then there was a design of the features that the classes in the layer should provide, and finally there came the implementation. The implementation was followed by a new release of the

product. During the usage of such iterations the software was divided into eight layers as follows.

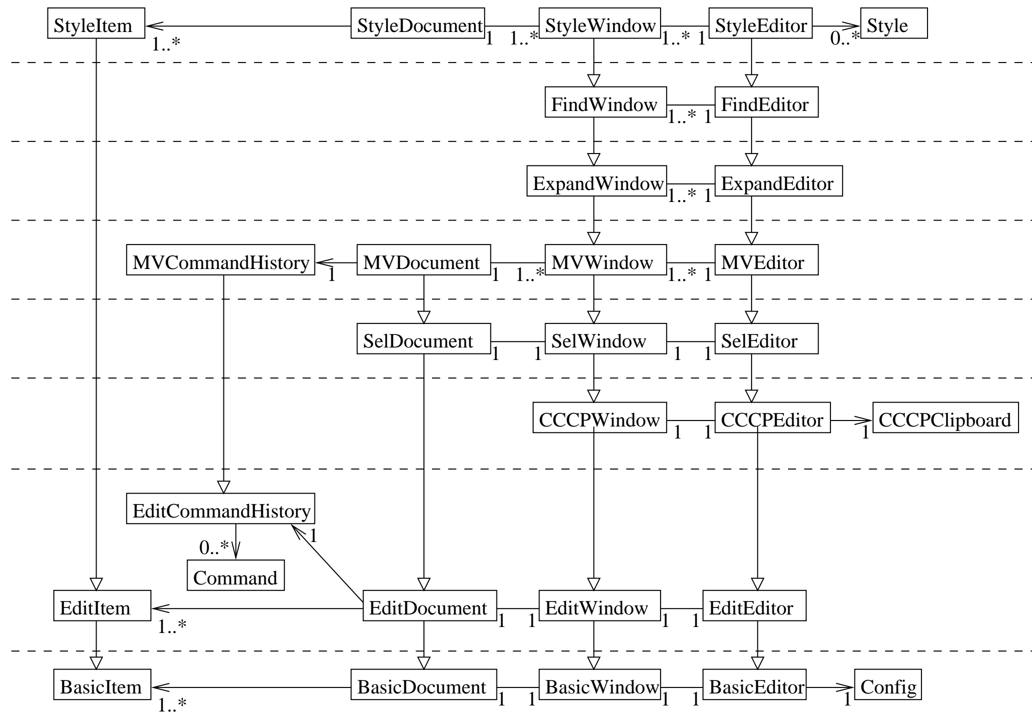


Figure 2: The layers of the editor

Layer 1. (Basic): Only one text file (document) can be opened per session. Documents are interpreted as outlined texts. Implemented functionality for *open, save, save-as* operations. The opened document, however, cannot be edited yet. The editor can also create a configuration file, write a set of configuration options in it and read them later.

Layer 2. (Edit): The basic editing operations such as type and delete characters, insert and remove items, change the indentation of items are implemented. Command history which allows *undo* and *redo* operations is also implemented. New document can now be created.

Layer 3. (Clipboard): Implemented *copy, cut* and *paste* operations. However they can be applied only to the whole document because the selection is not implemented yet. Some utility functions are bound to hot keys.

Layer 4. (Selection): Implemented the smart selection operation. The user can select characters of a string or several items. The operations from the previous layer as well as changing the indentation now work under selection.

Layer 5. (Multiple Windows): Multiple window functionality, based on the Model-View-Controller [8] pattern is implemented. The user can have many different documents opened in one session. The user can also have several views of the same document. The document can be edited in any of its views. The content of all views is synchronized.

Layer 6. (Expand and Collapse): Composite items of outlined texts can now be collapsed and expanded. The expansion operation has three models: ”expand all”, ”expand one level” and ”remember expansion”.

Layer 7. (Find and Replace): The standard operations for finding and replacing regular expressions in documents are implemented.

Layer 8. (Styles): The concept of styles is introduced. Two styles can now be applied to the items of a document: *plane* and *title*.

The project was structured ad hoc in the layers described above. Each layer consisted of about five classes. Most classes were extensions of corresponding classes at the previous layer. The extension mechanism was inheritance.

The refactoring idea of Extreme Programming was also a central part of programming using SFI. When new features for the layer were designed and the students started implementing them, they often found problems with the design of the previous layers. In order to proceed in stepwise fashion, the design of the ancestor classes needed to be improved and their code refactored. Changes in design and refactoring of these classes could in turn cause refactoring in lower layers. Such a refactoring chain can go all the way down to the first layer of software.

Because of the sequential layering, we cannot start with a new layer before the previous is ready. In some cases this is the only possibility, but in some cases this can cause unnecessary breaks in part of the development team. For example, the features of Multiple Windows layer do not interact with any other features from the neighbor layers. Furthermore, they are independent of other features. A better approach in this case would be feature combination. The Multiple Windows layer could be built in parallel with the other layers of the system. The next layer would then just combine those functionalities.

At the very beginning of this stage students did write the tests before the code. However, testing practices were gradually changing, so at the second half of this stage most tests were written after the code. The students claimed that the main reason for that was psychological: programmers usually want to see the results

of their code immediately, so they write the code first and want to see it running before they write any tests [2].

However, in the beginning of this stage it was not clear whether the SFI requires special ways for writing test cases. For each class in every layer the tests were independent of each others. New tests tested only the newest functionality introduced. In the future we will also need to check that the new features do not break the previous functionality by running old tests also on the extended classes.

7.3 Third Phase: Finishing

Two weeks before the end of the project came the final phase. At this phase we stopped introducing new functionality to the editor and focused on the next aspects: debugging, code cleaning and review, writing documentation and increasing the performance of the editor.

During debugging the programmers were running tests from the lower layers on higher layer components and writing new tests. During the feature introduction phase tests from previous layers were not run on the successive layers. Running old tests on new components did catch errors. The programmers were also fixing improper behavior of the editor. When a bug in the editor's behavior was found, the programmers got a card with the description of the bug and a possible way of fixing it. There were approximately 20 cards.

The debugging started from the initial layer and went through all successive layers. However, due to the architecture of the editor, debugging showed a so-called ”yo-yo” effect [5]: control flow passes through different layers and fixing a bug in a higher layer can require refactoring of some (and sometimes even all) of previous layers.

According to programmers, the pair programming did not work well during debugging, since the partner asking a question or giving a comment in the wrong time can force the ”driver” to lose his concentration completely. However, pair programming was really useful during introduction of new functionality in the second phase.

The code review and cleaning process was intended to make the source code more readable by adding comments, getting rid of ”spaghetti” code and duplicated code within the layers. Due to the syntax of Python, there were no special standards for how the code should look like, except that there should not be duplicated code between components of two successive layers. For catching duplicate code, the *diff* [20] utility was run on two successive layers for all layers.

During this phase the programmers also updated the documentation of the project. They added documentation strings (In Python there are comments but also a special language statement for documentation) to each class and method. They also created a technical report [2] describing the design of the editor. This

report also contains their personal evaluation of Python, XP, and SFI, and some advices for future projects.

There were also plans to increase performance of the editor. However, increasing performance would have meant that new methods needed to be written and hence, new bugs would be introduced. So, finally, we decided not to touch the performance aspect.

8 What was delivered

The programmers delivered a tarball with the editor and a technical report [2] describing the project and their own impressions. They also collected an informal lists of comments and suggestions. All these materials are freely available at the web site ”xprog0.cs.abo.fi”.

The delivered editor has most of the functionality specified in the requirements, but it lacks specific support for programming Python, like syntax highlighting and code execution.

The main problem with the editor is performance. Even in high-end machines (Atholon 1GHz) there are appreciable delays in the basic editing operations.

8.1 Basic metrics

The basic metrics are only calculated for the code produced in the second phase of the project. That is, the CVS module ExtremeEditor developed during 2 months (from the 22th of June to end of August). The metrics are divided into production code and testing code.

	Lines of Code	Modules	Classes	Methods
Code	3300	41	52	344
Test	1360	19	23	85
Total	4660	60	75	429

Lines of code do not include blank lines (lines that match the pattern ˆ$ neither lines containing comments that start at column 0 (lines that math the pattern ˆ#). This pattern matches the GNU license notices and dead code but not legal Python comments.

	Classes	LOC/Class	Methods/Class	Classes/Module
Code	52	63	6.6	1.26
Test	23	59	3.7	1.21
Total	75	62	5.7	1.25

9 Project Experiences and Impressions

During our project we gain a lot of experience that we will use in the future experiments. This will allow us to avoid some mistakes and improve the framework process for the future. The general impression of the experiment was, however very positive.

9.1 Extreme Programming

Extreme Programming turned out to be a very good software process in a university environment. It was easy for the students to learn the main concepts of XP. In addition, the students said they were enjoying working with this process. The ability to start a running project in very short time is also a great advantage for short time span university projects.

Furthermore, XP seems to be a good framework for performing practical experiments in software engineering. It allows us to test new experimental methods and still produce desirable software. Besides the standard roles in XP project such as programmer, coach and customer, we need to introduce some additional roles, such as *methodologist*. The role of the methodologist in an experimental project is to ensure that the development of the software is going according to the experimental methods of the project, SFI in our case. The role of the methodologist can be played by a coach as well. However, it is not a good idea to have the same person acting as customer and methodologist since the customer is interested in the product to be done, and does not want to take risks with experimental methods which can fail and give no results in the end.

We have to admit that having three Ph.D. students acting as customers and project leaders was not a good decision. This sometimes confused the developers since the three customers had slightly different notions of how a particular feature of the product should behave. To avoid such confusions in future we are going to have only one customer and one project leader for each project. Even if there are several persons interested in their product produced with the XP process, only one of them should interact with the development team.

9.1.1 Pair Programming and Collective Code Ownership

Pair programming worked well in our project. At the beginning we mixed the students who knew Python with the ones that were learning it. This fact lead to a quick start. The code production performance in a pair was proportional to the performance of the less experienced programmer. However, pair programming did not work so well during debugging. The students complained that it was easier

and faster to debug the code alone since “everybody has a different theory about where the bug is”.

Pair programming has many significant benefits: better detailed design (in XP the design is performed on the fly), shorter program code and better communication within team members. Also, many common programming mistakes are caught as they are being typed, etc [6]. As it has been frequently reported [6, 7, 12, 17, 22], 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.

Thanks to pair programming, we can expect that the senior students will teach the juniors while programming. The programmer’s training is more efficient since learning continues as long as one programmer in a pair knows something that his partner does not. The two are so intimately engaged in the coding task that it seems that much of the communication is non-verbal [14].

The concept of collective code ownership is also necessary since we expect a high turnover of students between different experiments, many of which may be consecutive stages in a longer range product development effort. New programmers need to read and understand the code that was written by the previous team, so the code should be very readable and understandable. As described in [21], a well-defined coding standard and the sharing culture encouraged by pair programming can help in this respect. Each pair can have it own writing style, so it takes a little time before a pair starts continuing another pair’s work. However, each programmer has a good overview of all of the code.

However, we think that pair programming should not be enforced all the time, i. e. during final debugging stage. It can be enforced during debugging for the purpose of writing new tests.

9.1.2 Continuous Integration and Refactoring

The students were performing check-in’s and integrating their code with the code in the repository daily. They did not use a special computer or token for integration.

The first major refactoring was performed after taking Stepwise Feature Introduction into the development, when the code was changed according to the SFI design. After that refactoring was performed after each new layer of the system. Refactoring is an essential part of SFI, and was therefore used quite extensively in our project (see section 7.2).

9.1.3 Unit Testing

Writing tests before coding is not easily adopted. The programmers usually wanted to see the result of their code first: to see it running before writing any tests. However, writing tests before coding could be easily adopted if one consider tests as a substitute for requirements specifications for a layer. Having a certain number of tests before the actual code will describe the functionality of the new layer to be coded and will help in partitioning the system in layers.

We did not succeed to have all tests built before the program code. At the very beginning of the project the students did write tests first but later on they started writing tests after the actual code. However, we think it is possible to force the developers to write tests first. This seems to be a good way of programming. Writing tests is not always easier than solving programming problems, but when thinking about tests one will understand the problem better, will think more carefully about the the extreme cases, and this will lead to a better approach to solve problems. On the other hand to write tests before the program seems to require rather skilled programmers, and we were working with second year students.

Due to SFI methods for building a software system, there should be certain rules for test cases. A test case in a successive layer should test the inherited functionality too. The simplest solution is to inherit and test everything from the ancestor test case.

9.1.4 Short Term Planing and Small Releases

In our project the short iteration cycles were defined by the layers of the system. Each new layer was first defined, then implemented and finally tested. When the layer was done, a new release of the system was produced. This meant that we had all together eight releases (see section 7.2) of our product.

9.2 Python

Python is elegant and easy to learn. The six students learned the basics of the language in one week and were productive after just two weeks.

Python uses indentation to mark the beginning and end of code blocks. This makes python code visually clear and easy to understand. Also, it reduces the need to set and enforce style guidelines for formatting the code. The language has a rich set of data types and an extensive library of support modules. The standard library contains a module for unit testing since release 2.1.

However, Python has also some drawbacks. Python is, in general, much slower than Java or C++. On the other hand, many important modules of the Python library are written in C, and they are usually faster than equivalent modules in

Java. Also, Python seems to use memory more conservatively than most java virtual machines.

Python is a highly dynamic language and therefore it allows many 'programming hacks' that are against any principle of structured programming, encapsulation and modularity. It is the programmer that has the responsibility to avoid these practices. Also, as many other dynamic languages, Python lacks a static syntax and type checker. We realized that most run-time errors while developing Python programs are simple typos that could be caught by a compiler. However, a good suite of unit tests should also discover these errors rather easily.

9.3 Stepwise Feature Introduction

The SFI methods force programmers to follow strict rules which later gives obvious pay-offs. A simple configuration management system supporting SFI efficiently by taking care of some of the rules would free programmers creativity.

It is not always obvious how the system should be partitioned into layers and how to order the layers However, these questions might be resolved with the help of unit testing. Writing tests for the features that should be implemented in the next layers will give an overview of the functionality required and thus help in partitioning and will also provide good requirement specification for each layer. This can also save time when free programmers can write the requirements specification for the new layer as test cases instead of waiting for their colleagues to finish.

In some cases one class had almost no code and was introduced only for some initializations. With a better design this can be changed using very little programming effort. Also the introduction of the Clipboard layer before the Selection layer does not now seem as a good decision, so this could be improved in future.

Software systems built using SFI methodology have certain advantages. It can be easier for a programmers not participating in the development of the software to understand the structure of the system and therefore to start extending it with new functionality. Since most of the changes between layers are extensions of classes, it is easy to distinguish between different pieces of system's functionality. It can therefore also be easier for a new team to take over the system without complete understanding of the whole code at once. However this is only impression which would be interesting to check whenever it is true during next experiments.

Testing of a software system built in SFI fashion is another aspect to do more research with. It is obvious that the test code of the lower layers should be reused on the code of the higher layers in order to test that the old functionality is not destroyed in the upper layers. This might require a special way of building test cases and, perhaps, a SFI testing framework.

9.4 Design by Contracts

Initially, we planned to use the design by contract method in our project. However, later on we abandoned this for a number of reasons. It seems it was too much for the tree months time we had for the project. Python does not support design by contract, so the programmers were trying to avoid writing pre- and post-conditions and class invariants as comments. In cases where it was possible to specify pre- and post-conditions using the Python *assert* statement, it was not done ether for the reason of having them already in test cases.

Further on, it seems that it is not a good idea to bring too many methods and techniques not known to the students into an experimental project.

10 Conclusions and Future Work

We succeeded to carry out the planned project tasks and we got the desirable product in schedule time. If the project would have continued longer, then we would have been able to improve the editor's performance considerably.

Using the Python language turned out to be very productive. However, it lacks some efficiency and because it has no compilation process, and type checking mechanisms it allows errors that would usual be detected by a compiler.

Stepwise feature introduction worked well in the project, but it should be improved by better guidelines for how it should be used. Because of bad design, some classes introduced classes provide almost no functionality. This contradicts somewhat the XP paradigm of no initial heavy overall design. Our main goal in this project was not to strictly follow XP, but to find the best way to build good software.

Extreme Programming can help to setup and perform practical experiments in software engineering. It is a flexible software process which is easy to learn. It keeps programmers focused on building the software product, and not on the experiment. We have shown that it is possible to perform experiments in a university setting, which will also produce needed software, as a side effect.

However, we can not always follow the all guidelines of the XP as they are described in [3]. We definitely can not apply XP if the subject of an experiment is another software process. In other cases, we can use XP, but we still need to adapt it so that some task is performed more often than it usually is. For example, if the topic of an experiment is a modeling notation such as UML, we need to be sure that modeling appears as a main task in the software process used in the experiment.

10.1 Acknowledgments

We would like to thank our students Rasmus Back, Linus Bernas, Tomas Czarnecki, Marcus Eriksson, Mats Sjöberg and Max Söderström for their enthusiastic participation in the project.

References

[1] 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.

[2] Rasmus Back, Linus Bernas, Tomas Czarnecki, Marcus Eriksson, Mats Sjöberg, and Max Söderström. Extreme editor. See http://xprog0.cs.abo.fi.

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

[4] Brian Berliner. CVS Web Site. Online at: http://www.cvshome.org/.

[5] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison-Wesley, 2000.

[6] 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.

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

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[9] Bernd Gehrmann. Cervisia manual. See http://cervisia.sourceforge.net/.

[10] GNU. General Public Licence. Version 2. Online at: http://www.gnu.org/licenses/gpl.html, June 1991.

[11] John E. Grayson. *Python and Tkinter Programming*. Manning, 2000.

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

[13] Mark Lutz. *Programming Python*. O'Reily, 1996.

[14] Robert C. Martin. RUP / XP Guidelines: Pair Programming. Rational Software White Paper. Available at: http://www.cs.unb.ca/profs/wdu/cs4025 /pairprogramming.pdf, 2000.

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

[16] Bertrand Meyer. Eiffel programming language. See http://www.eiffel.com/.

[17] 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.

[18] OMG. OMG Unified Language Specification. Version 1.4, February 2001, available at: http://www.omg.org.

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

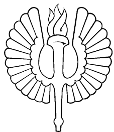
[20] Walter F. Tichy. Rcs – a system for version control. In *Software – Practice & Experience*, volume 15, 7 (July), pages 637–654, ?

[21] William C. Wake. *Extreme Programming Explored*. The XP series. Addison-Wesley, 2000.

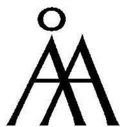
[22] Laurie A. Williams and Robert R. Kessler. Experimenting with Industry's Pair-Programming Model in the Computer Science Classroom. *Journal on Software Engineering Education*, December 2000.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi



University of Turku
- **Department of Information Technology**
- **Department of Mathematics**



Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**