

The Skopeo Pro project, an Extreme Programming experience report

Willem van den Ende and Erik Groeneveld, © Seek You Too BV 2002, 2003
www.cq2.nl

Enumerating the practices of eXtreme Programming could not possibly give full appreciation for what XP can achieve. In this article we try to make things more lively through a real-life example of a project we coached in 2001.

The members of project *SkopeoPro* had developed ten systems for image retrieval in the past few years. Most of these programs had many features in common with each other. It was hard for the programmers to add features to each of these programs, because if they added a feature to one program, it would not be in the other. When users saw a new feature in another program, they also wanted it in their program. The programs were different, because the clients had slightly different wishes and different ways of describing their images.

Clients differed in the subject area they were interested in. A town archive for instance, wants to describe different aspects of an image than an art museum.

Because of the poor maintainability caused by the many versions, management was not convinced that they should support development of this product in the long run. Management gave the development team half a year to prove they could build one maintainable application instead of ten. At the beginning, the team did not know how to get there. They could have followed a product line approach, and sum up all requirements for the new system, then design a framework and then design ten new applications with the framework. Instead they decided to *do things different*.

The team hired us to help them transition to Extreme Programming. XP helped them to achieve the necessary focus, set priorities and develop working, maintainable software in a short period of time. Because management wanted them to prove their ability to deliver maintainable software, delivering only a design at the end of the project was not an option. To ensure both focus and minimal generality, two persons were enlisted as customer: an expert on image description standards, and an archivist from a dutch city archive who used one of their existing programs.

Learning to be either a Customer or a Programmer

Growing the team meant educating the developers as well as the customers. The customers learned to ask for functionality in small bits. The developers learned to ask for help from the customer when some of the functionality was unclear. After three iterations responsibilities were clearly separated. Developers did not try to force functionality on the customers, and the customers did not force a particular implementation on the developers. This created a friendly atmosphere, with a safe environment for hefty debates, which never escalated to interpersonal conflicts. Truly shared vision can only be achieved through an open dialogue, with room for everyone's thoughts and feelings.

Achieving clear responsibilities was facilitated by the *Planning Game*, which was held every two weeks. In this meeting, the developers demonstrated the user stories (comparable to one-paragraph use-cases) which they had realized in the past iteration. The planning game was conducted with a computer showing the web-based

system. This was very important, because it is easy for customers to use the system and show where they want things different.

The customers signed off on the demonstrated stories, or asked for rework if they are not completely satisfied. The customers then got to brainstorm about new functionality. The developers estimated how much time it would take them to develop this functionality. In the first planning meeting estimation was very difficult, because developers had no idea how long something would take. This, however, is no reason not to estimate. The developers just guessed how long it would take, and a guess is just an estimation with very low confidence. Customers then prioritized the stories and decided which ones they found most important for the coming iteration.

Having the planning game every two weeks forced the developers to come clean, and to show what really had been done. Holding the game even when there was no progress required some courage of the developers.

Estimation improved rapidly, every two weeks the team could learn a bit. As iterations passed, guesswork was slowly replaced with confidence. Both the amount of work estimated per story and the number of stories estimated per iteration became more accurate. After a few months estimation was accurate enough to create a release planning. The release planning had a three month horizon. Features in the software released after three months were quite close to the estimates. The customers still used their right to change their minds every iteration, therefore not everything planned originally was contained in the release. The customers were happy with this, because the things they felt were most important at the time of release had been built. The release was fairly small (*Small Releases*), but was interesting enough to show to prospects. After seven months, the team had delivered useful, reliably working software that could easily be extended for new customers, which they subsequently attracted.

Communication and Feedback between customers and developers

Because customer could not be *On-Site* full time, extra effort was needed to ensure sufficient communication. Luckily, the new program was web-based (unlike the previous applications), so a server was set-up where the customers could see new functionality as it was built. Whenever the developers had a question during an iteration, they would phone or e-mail one of the customers and ask them to look at the server. Getting the developers to frequently contact the customers took some time.

Learning Test Driven Development

The developers had built their previous applications using Visual Basic. They found that it was one of the things that prevented them from delivering flexible, maintainable software. The team wanted to switch to Java and do full Object Oriented development. Before the start of the project we introduced them to Java and OO. We warned the team that changing both their development process and their technology might be too big a step, but they were willing to take a chance.

Changing from structural to Object Oriented development is a big step. Like transitioning to Extreme Programming developers have to forget their old behaviors and adopt new ones. Taking Java as a starting point for learning Object Oriented programming does not make it easier, since the language comes with very large libraries which make it unclear where to start.

In this respect, *Test Driven Development* was a big help. A unit test forces the

programmer to focus on a very small piece of the program, and it also acts as a safety net. Focus and a safety net are a great help when learning a new Application Programmer Interface such as the Java SDK. The programmer only needs to learn that small part of the API necessary for the operation being developed, and the safety net ensures that his understanding of how to use the API is correct.

Writing a unit test before writing an operation proved to be very difficult at first. A blank screen with a blinking cursor is daunting. When the test-suite contained about twenty unit tests though, the team got addicted writing unit-tests. Developing a new test similar to an existing a test in the suite is much easier than starting from scratch.

Writing tests first is a behavior pattern that takes a lot of exercise and discipline before it is completely integrated. It is very tempting to say “*well, this new behaviour is so complicated, I don't know how to write tests for it (So I will leave the tests for now, and write them later...)*”. These are the moments that courage is most needed. When programming, you have to stop and think of a way to make the new code testable, or better, make the new behaviour simpler, so that it becomes easy to test. One way to ensure that the program as a whole remains simple and thus easily testable is to continuously *Refactor* it. We will explain how we did Refactoring after we explain *Continuous Integration*.

Learning to Integrate

For a coach the combination of *Test Driven Development* and *Continuous Integration* was very useful. Continuous integration forces programmers to check-in their code to the version management system several times a day. We combined this practice with an automated nightly build, which ran all the unit tests and deployed a new trial-system to the web server. An e-mail containing the results of the unit-tests was sent to developers and coaches every night.

The nightly build was a very useful development practice. It always ensured running, reliable software, and was a great communications tool that enabled the team to focus on real progress. For instance, there were two iterations in which there was no growth in the number of unit tests. After some investigation it appeared the team got stuck on some new design and re-design work. When the team got unstuck and solved their design problem, the number of unit tests started to grow again.

Setting up the nightly build and version management tools was quite a bit of work though. In the start of the project we spend quite a lot of time to get CVS and the nightly build up and running, and only after two iterations it started to work really well. *Continuous Integration* was something that needed constant attention throughout the project, because it needs courage of the programmers to be done, and the payoff of frequent integrations is not immediate. The payoff was considerable, however:

- Before the release it was not necessary to collect the code and spend some painful weeks in trying to fit the pieces of the puzzle together.
- There was always a working system for the customer to play with, which greatly facilitated communication with the customer. The visible program forced the programmers to be open and honest to the customer: if there was no progress, the customer did not see changes in the system.
- *Pair Programming*, the practice forcing all production code to be written by two programmers sitting together at one workstation, was greatly facilitated. Because it was so easy to get fresh code to a developer workstation, changing pairs was not a problem. *If you really want version control to work, make the version control system the easiest way to transfer files between developer workstations.*

- *Collective Code Ownership*, which means no programmer owns a particular part of the code and everyone can make changes as they see fit, can only work if all the code is available to all programmers all the time.
- The combination of *Continuous Integration* with *Pair Programming*, *Collective Code Ownership* and *Test Driven Development* created a safe environment for the programmers to try out *Refactoring*.

Learning to Refactor mercilessly

Refactoring is continuous restructuring of the program in tiny steps. *Test Driven Development* was essential, because the unit tests create a safety net for the programmer. *Pair Programming* acts as a safety net as well as a learning and design tool. While one programmer was typing, the other could think about the design and conduct a kind of instant review which prevented errors from creeping into the program. *Collective Code Ownership* enabled the programmers to clean up code made by others, which was important because a fresh look often brings the insight necessary to simplify the design. *Continuous Integration* made it possible to restructure in small steps and make sure that other developers did not have to do too much rework.

Refactoring was helpful in learning the Java API, because in the beginning the developers were unlikely to choose those classes and operations from the API that create the simplest program the simplest. After some time the developers were more aware of what was available and had grown their bag of Object Oriented tricks (e.g. Design Patterns), so they could simplify their program. Refactoring gave them the freedom to get the program to 'just do its' job' and then make it maintainable in small steps.

Learning to do merciless Refactoring however is something which takes a great deal of effort, and constant attention of a coach. Refactoring works best when done in very small steps. Taking smaller steps can not be taught: it can only be learned by doing. Having little experience with Object Oriented Design (as embodied in for instance Design Patterns) made it extra difficult for the programmers to select appropriate steps.

For instance, the programmers got stuck two full iterations when they tried to refactor eleven classes at once, which proved too much for their current capabilities. The programmers learnt from this experience the importance of trying harder to find small steps, and thinking harder when a small step does not obviously present itself at first.

Changing a whole program in one fell swoop is something which often brings out the hero in a programmer. After an experience of getting stuck in a large Refactoring and having experienced successful small refactorings before, searching for small steps in the future becomes much more attractive: once one gets addicted to having running unit-tests every few minutes, programmers feel uncomfortable when they break the tests for a long time in order to do a large restructuring.

Pair programming did not always require much effort from the coaches. Normally, because the programmers enjoyed working together, pair programming was the obvious way to work. The programmers reported that they found pair programming the most productive way to work. If two programmers are closely matched in capacities, and have opposite ideas of what to do next, a heated and lengthy debate can be sparked. Working like that can be very tiresome. The programmers report that they can not sustain this way of working for more than four hours. In such a case it is advisable to switch pairs (Which should be done several times a day anyway).

Another way of resolving conflict is to let both programmers perform a small experiment (called a *Spike*) on their own. They subsequently discuss their ideas based on the code in the experiment, which is much more concrete. The spike usually leads to one of the two solutions being clearly better than the other.

Spikes are never integrated into the baseline code. A pair has to re-implement the winning solution. This way, only code understood by at least two people enters the version control system. Writing the code again also prevents messy 'trial-and-error-code' in the baseline.

-There are some practices for which we did not yet describe what effort they took: *Coding Standard*, *Collective Code Ownership*, *Sustainable Pace* and *Metaphor*. These practices took much less effort than the ones above, but they were very essential to keep the whole system of patterns in place.

Sustainable pace was easy, because the programmers spent only three days a week on this project. On other days they worked on other projects, or had a weekly day off. We believe that *Pair Programming* was essential in learning Test Driven Development and Object Oriented Design. Because pair programming was done regularly, and there were only four programmers, defining an explicit *Coding Standard* was not really necessary. We did, however, regularly spend time with the whole team discussing design issues at the whiteboard. The *Metaphor* was not made explicit. The goal of *Metaphor* is to give programmers and customers a common language. Because programmers and customers alike had experience in the application domain, they could already communicate easily. If we would have defined a *Metaphor*, it would be something like "a standards based image retrieval system". Finally, *Collective Code Ownership* in itself was not so hard, because none of the programmers wanted to shield 'their' code from the others.

Summary

The *SkopeoPro* project was a success, because eXtreme Programming enabled all participants, customers and programmers alike, to learn continuously and profit from the insight gained as the product matured. Working closely together forced everyone to put issues out in the open, no matter if the issues were personal, technical or progress-related. Having such an open atmosphere requires a lot of effort by everyone. The end-result was very rewarding, and in our opinion more than compensated the trouble we went through.

Acknowledgments

We would like to thank the programmers, Henk Laloli, Laurents Sesink and Joris van Zundert and of course the customers Rene van den Horik and Constant Kroeze for their active participation in the project and their review of this report. We would also like to thank Marc Evers, Ruud Bronmans and Peter Schrier for reviewing.