Extreme programming virtues and benefits

In part two of the Seeing Machines case study, Timothy Edwards discusses human–machine interfaces, the company's revolutionary technology that is underpinning its global success

Timothy Edwards is Principal Engineer of Seeing Machines. He has experience in systems and software engineering, from developing high-speed digital logic to software engineering and team management. Edwards has bachelor degrees in Systems Engineering and Computer Science and eight years' experience in the areas of RADAR design, graphics and animation, robotics and computer vision.
E-mail: Tim.Edwards@seeingmachines.com
Visit: www.seeingmachines.com

Seeing Machines is a company that has strong beliefs about how to develop software. Fundamentally, software engineering is crucial to success. The process of software engineering must continuously evolve as developers, development technologies, networks, platforms, customers, markets and business goals change over time. To deliver high-quality products now and in the future, it is essential that a culture able to embrace change is generated.

The startup struggle

In the first six months of development, enough source code had been generated to reach that awful mental limit where it was no longer possible for a single person to comprehend the system in its entirety.
The attempts to cope were through:

- Using the skills as experienced developers to write high-quality, well-documented and reusable object oriented code
- Regularly meeting and discussing designs
- Using strict source-code control
- Segmenting code into many libraries and ensuring strict hierarchical dependencies were maintained in these libraries
- Reading textbooks ad hoc to try to learn new skills

When the company launched, there was extraordinary pressure to deliver the first product. It was a make-or-break situation in an environment where:

- Only the needs of a single customer were known, therefore, how most of the features would be used had to be guessed
- No time was available to properly test the system, given the scope of what had to be done
- There were only one or two developers with expert system knowledge, and who were heavily relied upon to complete 'their sections' of the code
- There was a culture of 80-hour weeks, all-night coding and general martyr-like behaviour, which existed, not only in the development team, but across the entire company

The release date was missed by several months, the software was very buggy and the mood around the company was one of pinning blame ('blame-storming').

Fortunately, there was a great marketing warrior and enough systems were sold to start working on the next version. However, it wasn't long before the pressure was on again for bug-fixes and feature improvements, due to the low-quality of the first release.

Thankfully, instead of rushing back to the grindstone, a little breathing space was taken as a post-mortem was performed on what had occurred.

Becoming agile

Most software engineers will have heard of the 'agile' way of doing things. To the experienced ear it probably sounds like yet another silver bullet. It is simply 'applied common sense'.

From past employment experiences, it was recognised that changing established methods of software engineering can be extremely difficult, even if the existing process is unrewarding for developers and delivers poor results. Therefore, the top priority was to attempt to integrate a philosophy of continuous improvement and learning into the environment, so that if things started going wrong, they could be changed.

Despite formal educations in traditional development techniques, what was known was eagerly erased on the course in extreme programming – a popular new technique applicable to small teams, and delivered by the inspirational Dwayne Read of Strategic Systems Pty Ltd.

What was learned, besides the simple and powerful 'XP' methodology (some techniques of which are discussed below), was that this basic idea of embracing and incorporating change on a regular basis is the key to staying in control of this non-linear beast called software development.

Termites and software

Generally speaking, complex dynamic systems (of which the software development process is a good example) can exhibit emergent behaviours. Such behaviours can be remarkably sophisticated, relative to the system's fundamental processes.

Frequently used is the example of termites. Termites have tiny brains and a small set of behavioural patterns, yet somehow, despite no obvious central intelligence, they can build incredible colonies with towering structures containing fungus gardens, nurseries and even air-conditioning systems. These things are not designed; they are the emergent properties of the termites' behavioural rules.

Central to the success of the termites is that their behavioural rules have evolved.

Evolutionary processes require feedback from the environment. With termites, success is measured in terms of reproduction, and the unavoidable feedback comes in the form of echidnas, droughts, floods and bacteria.

With software engineering, success is a function of delivery dates, features and bugs, and the feedback that should really occur at every level of the process – from the way code is entered to the way we listen to our customers – is, unfortunately, quite easy to avoid. This is because incorporating change is often too hard and time-consuming.

As complex-systems experts, we know that as our environment changes, things can go wrong in two ways:

1. The system becomes chaotic. Rules are bent and ignored because they no longer seem to fit, or get in the way of rapid progress. Bug levels rise, features mutate and creep, schedules slip and programmers lose sleep.
2. The system stagnates and remains impervious to attempts to push it along. The process feels 'heavy' and programmers become demotivated and sluggish. As bugs levels rise, schedules slip and things get 'foobared' ('foobar', or more correctly, 'fubar', is an old hacker term accurately describing this situation).

Extreme programming

What extreme programming gave us was:

- A formalisation of what seemed to be plain old common sense
- A starting point for the company to evolve its own way of doing things

After the experience with the first release, the following was implemented:

- Handing over the responsibility of feature prioritisation to marketing (this removed a lot of pressure).
- Discouraging ego-coding and code ownership through 'pair programming'.
- Setting up a unit testing framework, unit testing all new classes and writing quick tests for the most important older classes.

▶

BENEFITS

Emergent QA

Some of the hype seems to be true. With extreme programming, desirable things like quality assurance just seem to come out 'in the wash' (it is an emergent property). This is because a lot of time is saved by not making mistakes in the first place and catching those mistakes early.

This is not to say that the QA team should be sacked. It just means faster testing.

Emergent happiness

At over 700 classes with 14 threads and some scary real-time stuff, it can be admitted – nobody knows how faceLAB works in its entirety. The development team is just like the termites in that regard.

However, they are also confident that they will get the next version out on time, that it will have the features most wanted by the customer and that the system will be reliable. Generally, this all adds up to more sleep and happier and more creative engineers.

A cultural shift

Ultimately, the approach was recognised as a major success inside the company, to the point where some of the techniques have been taken on board by senior management. 'Prioritising goals' and 'refactoring the plan' are expressions heard quite often around the company.

Environmental change

Recently, the environment at Seeing Machines has shifted drastically. Having gotten to their feet with the flagship product faceLAB, now out the door with several solid releases and raising some fairly significant revenues, they are now in a position to take the next steps.

On the technical side, several new research projects have recently begun to discover even better vision-processing techniques, as well as embedded system and hardware engineering projects aimed at reducing costs and meeting automotive specifications.

Consequently, there are wonderful new ways for things to go wrong. The code base is diverging due to opposing development forces. Some developers are working on multiple teams. Bugs are getting fixed in one branch and not propagating to the other branches. Resource pressure is breaking down pairing and code ownership and specialisation are all creeping back in.

Feedback. Time to evolve the rules ...

- Encouraging 'courage' (diving into unfamiliar code) and refactoring principles using unit-testing as a foundation for confidence.
- Identifying and stopping many instances of overdesign.
- Introducing morning stand-up meetings – five minutes to plan the day over the morning coffee table.
- Developing in three-weekly iteration cycles where task breakdown is performed, prioritisation and pair assignment. At the end of each iteration, the software is planned to be fully operational and is internally released for smoke testing. Every three iterations, the software is fully system tested and is suitable for customer release. Iteration deadlines are strict to within one or two days at most.
- Changing the attitude of heroic late-night coding efforts to working up the courage to go home at normal times.

- Committing significant time to achieving basic levels of automation and continuing to spend a day or two every few weeks improving the system, firstly by ensuring automatic builds, unit tests and html source documentation tools ran nightly. Breaking the build became a crime! A bug-tracking database was started. Lint was automated. The net was explored for new developer tools, which were evaluated and purchased where necessary. The Technical Support and the faceLAB Calibration processes were automated.
- Formal code inspections were started to both communicate and improve company standard practices – placing an emphasis on existing reusable library code and all new classes.
- Collecting an e-mail knowledge base of development ideas for use as feedback into the process.
- Starting a weekly reading group to work through several of the classic software engineering textbooks including Design Patterns, Refactoring and Modern C++ Design, as well as discussing existing design structures in the codebase.
- Establishing the expectation that after every major release, the development team would leave the premises for a day or two, slap each other on the back for doing a good job and then think about how the environment has changed – subsequently finding ways to revise and improve the processes. ■

FURTHER INFORMATION
Refer to *Software* journal (April 2003) for part one of Seeing Machines' case study: 'Technology that sees'.