Extreme Component Based Development

Rob van der Veer

Abstract

This paper describes improvements in the software development process at Sentient Machine Research, making use of Extreme Programming ideas. The relevance of Extreme Programming for creating and reusing components is discussed, by showing how its principles have been applied in solving the problems Sentient had with adopting Component Based Development.

Keywords: industry, extreme programming, component-based development, reusability, middleware, refactoring

1 Introduction

Sentient Machine Research is a Dutch company, currently counting 25 employees, specialized in bringing artificial intelligence technology to the market by creating tools for data analysis and pattern recognition. Because of the trendy market, cutting-edge applications, continuously evolving AI technology, and uncertain results from research and prototyping, requirements vary constantly. This continuous change and our substantial investment in technology create high standards for the software development process, especially regarding reuse.

A few years ago, we have started to take a serious look at our software design process, focusing on the setup of a component library and applying Component Based Development (CBD) principles. Since then, many improvements have been made with respect to infrastructure, organizational structure and methodologies. The European Software and Systems Initiative (ESSI) has funded part of this work.

This paper describes the problems we have encountered in applying CBD and how we solved them using principles from Extreme Programming (XP) [1]. XP is a discipline of software development, designed to work with projects that can be built by teams of two to ten programmers. None of the XP ideas are new.

The innovation is that all practices support each other and are executed thoroughly.

The next sections each discuss a main problem and the way we solved it, illustrating how well XP and CBD can be combined, despite of some aspects that do not seem to match.

2 Library components bottleneck

At first, the creators of a component for our library became so-called 'owners': all component changes had to be done by them. This rule was introduced because we decided that the risk of breaking our legacy components was too high if other developers would perform changes. When a change was required, this situation had one or more of the following undesired effects:

- Interference with the projects the owners were working on
- Project delay as a result from waiting for the owners to create time
- Rushed change by the owners - often because of lacking motivation (it was not their project) or because of the time stress on their own projects.

Other developers decided to perform the changes and did not integrate them in the library component because of the risk they would break it - the new features of the component were lost for the library.

If the original owners were not working for the company anymore, the necessary knowledge for applying the change had been lost. Leaving programmers are a problem when restricted code ownership is applied.

It has been shown by others that collective code ownership, which would solve these problems, is a feasible goal [1], providing the software complies with certain rules:

- High adaptability. The result of frequent refactoring, pair programming and complying with coding standards (see below).
- Software is bundled with testcode (see below). Automatic testcode is essential to reduce the risk of breaking a component

when applying a change. In addition it serves as documentation of the use cases.

The illustrated problems made us decide to adopt collective code ownership based on the ideas from Extreme Programming; component changes can be done by any programmer, unless the changed component is tested and the code is integrated right away, to reduce the chance of multiple changes happening at the same time. Hence, we first added the requirement for a component to be bundled with testcode for full functionality testing, adopted pair programming (see *quality* section), created code standards, and started training developers to write more readable code. Next we had to further address component adaptability, which already was a problem in our company, even for the owners themselves.

3 Insufficient adaptability

Components are hard to adapt, because preserving component quality is essential – developers need to rely on library components in their project. So whenever a component was modified in our library, changes to the existing code were kept to a minimum, reducing the risk of introducing bugs. Typically, this resulted in bad structure (*spaghetti code*), where the new feature did not really fit in and redundant code was created. If the component is seen as a machine, this way of writing new code can be seen as attaching another machine on the outside of the existing one. The changes through time could often be identified in our code by looking at subclasses that were created to carefully introduce changes – resulting in unreadable code.

Components adaptibilty is important because reusable components are subject to change in every project in which they are reused. Furthermore, when creating components to be used in a software system, the requirements not only change when system behavior has to change, but also when system design changes. Therefore component requirements are often altered during the development process - when the customer and when the developers want the system to behave differently (for example during refactoring).

In his book [1], Beck talks about the schoolbook example of software change costs rising exponentially over time. Because of modern tools and methodologies, software engineering has somewhat reduced this exponential increase. He states that the only way of making the costs rise very slowly is to put an emphasis on continuous refinement of the program design (i.e. refactoring). A component probably has to be changed long after the project in which it originated was finished, and maybe even by other developers, so keeping the cost of change low over time is essential.

3.1 Refactoring

Repeatedly applying changes to our components has resulted in unreadable structure and because of that, bad adaptability - even for the original developers. Extreme Programming states that after or during a change, *refactoring* [2] has to be applied when clear structure or adaptability is at risk. If so, you are doing more work than actually needed that moment, but it ensures you can add the next features in a reasonable amount of time.

Refactoring is the process of taking an object design and rearranging it in various ways to make the design more flexible and reusable. There are several reasons for doing this; efficiency and maintainability being probably the most important. Refactoring eliminates redundancy and *ugliness*, and increases simplicity and clarity.

XP states that this frequent refactoring is only possible when other practices are applied: pair programming (for reducing risks in the restructuring, see the *quality* section), and unit testing (for testing if the refactoring did not alter the behavior, see the *quality* section). As discussed earlier, we already raised our standards for readability and required testcode bundling. Hence, the necessary practices for refactoring were already there. To promote refactoring, we motivated the developers by explaining the advantages of refactoring and by teaching them appropriate techniques.

Initially, there was some concern in the company about interface changes through refactoring. Changing the interface puts component users in the difficult situation where they have to decide if they need to start using the improved component and change their code accordingly. However, the goal of refactoring is not to change the behavior, hence the chance that an interface changes during refactoring is small. If it does occur, for example to make the interface more elegant, there are ways around this problem, by adding a new interface when changing it, while still supporting the previous one. How easy this is, depends on the used middleware (if any). When COM or C++ is used, our standard is to create a new interface and implement the old interface with it.

4 Insufficient quality

Component quality is of course important because it not only pays off in the current project, but also in future projects. According to XP, component quality is affected by several combined practices, where pair programming, unit testing and planning are the most important. Furthermore, software quality is highly related to software adaptability (see previous section), because the risk of introducing a fault is by definition smaller in case of high adaptability; changes to software that is not fully understood are likely to cause problems.

4.1 Pair programming

'Pair programming' means that two programmers, in the same role, sit behind one workstation. This approach increases quality because one person is programming while the other is observing and providing immediate feedback.

At first, pair programming seemed not very useful to us, especially to our management, until we realized we were already doing it successfully in several cases - mostly complex jobs or jobs under high time pressure. Currently, we are applying pair programming more and more. Prototyping jobs or other projects with reduced complexity and risk are still done by a single programmer. Whenever there is doubt we perform peer review of code changes. By doing so we are learning when to decide to program in pairs and when not.

4.2 Unit testing

XP's states that a piece of software should always have automatic test tools, created by the developer of the software, as a first task – before any component code is written. Apart from serving as sample code / documentation this testcode increases quality because:

- Every time a component is changed, the testcode can be used to test if quality is maintained.
- The software module is tested in its full functionality instead of testing it inside the an application, which tests all features of the application but not necessarily of its modules. For this reason, unit tests improve component quality in generic situations, outside of the current projects scope.
- The programmers themselves are responsible for testing their software. This is contrary to the common belief that it's better to let others do the testing because other people think different and might think of situations where the software fails. Letting developers create the testcode themselves has several advantages: the developer has better understanding of functionality boundaries and critical sections that need to be tested. That way the developer feels more responsibility: instead of just releasing the software and let the testers test it in all kinds of circumstances, the developer is forced to anticipate these circumstances in advance (by writing the testcode) and hence prevent bugs instead of fighting them.
- Because programmers write testcode for their own components first, they are forced to take on the role of component user, resulting in interfaces that are more straightforward.

At Sentient, we adopted the unit testing principles. However, testing is still done manually in some user interface situations because exact user interaction is hard to simulate. In order to reduce the risk of oversight we often apply pair programming when creating unit tests - mostly by letting another developer take a second look at the created testcode.

4.3 Planning

XP planning is a kind of timeboxing: time limits are set for each element of the system, by the developers themselves. Developers have the best knowledge of what needs to be done, so it is important that they learn to make time estimates. Furthermore, the motivation to make it in time is higher than when others impose a deadline - they cannot blame somebody else making a bad estimate.

In order to improve the estimation capabilities, it's important to inform the programmers with the actual time taken for each element. It is also important to make the estimate elements small, possibly by spitting up a process into milestones.

Whenever a deadline is not going to be made, XP stresses that secondary features should be dropped instead of quality. At Sentient we have been doing timebox planning by developers for a long time, but unfortunately quality has often been the victim of bad planning. Recently, we have started making feature priority lists with the customers, to be able to start with the most important ones. Project leaders are now aware that low-priority features have to be dropped as

soon as a schedule turns out to be too tight, instead of rushing the completion.

5 Middleware complications

During the 7 years in which we have used our middleware standard COM we learned about its merits and its shortcomings. We realized these shortcomings were going to be a real problem if we would create a typical CBD component library: a database of components that can be used to build systems using a standard middleware solution, and therefore are required to have a middleware-compliant interface (e.g. DCOM, CORBA). This technology allows using any type of components from anywhere, which is great - but there's a price to pay:

- Implementing the middleware interface takes time. Several programming languages are used within Sentient and therefore components typically start out as native classes or sets of routines. Converting a class to a COM component is easy for just a few languages (e.g. Visual Basic).
- The middleware interface is typically less powerful than a native interface. For example the possibilities with C++ interfaces are much greater than with a COM interface [3]. A further restriction is the way the component is going to be used - for example: memory pointers are out of the question if the component is going to be used by a Visual Basic tool or if it has to run out of process [3].
- Middleware interfacing is often slower, because of two reasons: 1) virtual functions are used [3], which means a small performance penalty, and 2) data often has to be communicated in less efficient ways because of the less powerful interface. For example: as mentioned above, memory pointers cannot be used in some cases, so a datastructure has to be communicated using a COM object for each element in the datastructure, which introduces memory and processor overhead.
- Dynamic linking creates versioning problems

5.1 Versioning problems with dynamic linking

Dynamic linking means that components are used in an opportunistic way: at the moment it is needed, the software actively requests a link to a component from the operating system or middleware. In static linking, the component is embedded into the software.

The advantage of dynamic linking is that component does not have to be loaded more than once on a machine, which is especially useful for system components. For other components, dynamic linking has shown to have many disadvantages, in our practice.

The problem with dynamic linking is that you don't know if you get the same component with which you developed the software, causing the following problems:

- Upward version incompatibility of components: Often, only one version of a component can be present on a machine - preferably a recent version. The problem is that not all components have been tested with that version, which may cause failures. This happened many times at Sentient, and not just with our own software.
- Clutter of dependent components: Sometimes component versions have different identities (filenames), mainly because of the compatibility problem mentioned. That way, every component is able to load the exact component version it needs. This results in more than one version of the same system software loaded in memory, causing performance problems.

The solution to this problem, called *side-by-side installation*, is to bundle software with the components of the exact same version it was created with, and make sure they stay on the system. In COM this would mean forcing the component to have a unique component identity (classID, progID and filename), and storing it in the same location on the file system as the main software, instead of a system location. Microsoft has decided to support this kind of installation in their new operating systems (Windows 2000, Windows 98SE).

By doing so, the advantages of dynamic linking are taken away and static linking becomes a more obvious option. Furthermore, the middleware component administration has to store much more component references, which can reduce performance.

These are the biggest problems Sentient has experienced with third party components, especially because the side-by-side installation is very hard to apply on binary components. That is why we decided to focus more on open source third party components, which are unfortunately still uncommon.

5.2 Our solution

Because of the discussed middleware complications, we have decided to follow the XP principle 'Build what you need and nothing more', when it comes to component interfaces. This means that components keep their native form and can be published in the library as such. Therefore, some of our components are C-libraries, or Prolog procedure code, which are not accessible from other languages. The component can always get a middleware interface later, when it actually turns out to be reusable in a project written in a different language or on another platform. This also decreases the pressure on creating components - extra effort is postponed to the project that is going to reuse it.

Because most component library systems assume a standard interface form, we had to create our own library system, based on a file system database with the following component types:

Source code procedures/ classes

- C libraries
- Windows DLL's
- COM DLL's
- COM controls (OCX)
- COM EXE's

6 Small component production

Our component library did not grow as much as we had hoped for because of the following reasons:

Publishing a component required the developer to take responsibility for maintaining it, creating stress and consuming time

A library component required extra effort in documentation

A library component had to be extended with features by anticipating the generic purpose. Such features are not needed in the current project and therefore put an unanticipated pressure on it.

Reusable elements were hard to extract from our software structures, in which clean design had been eliminated by applying change after change without restructuring.

6.1 Our solution

By introducing collective code ownership (see above) we took away the pressure of being responsible for all component changes as a component creator.

By increasing our adaptability standards (see above), the documentation requirements of a library component became the same as for any other piece of software; hence the second obstacle for publishing a component was removed.

Extreme Programming assumes 'you aren't going to need it' when it comes to anticipating future requirements. In contrast, component based development dictates that a full-featured component should be produced, by generalizing the current component requirements. We recently adopted the XP principle to build components exactly how they are needed and nothing more. By doing so, the developer can focus on what is important and does not have to create extra features, with the risk of it being unneeded after all or not meeting future requirements. We already had decided to apply this principle to interface standards (see above). By not forcing the developer to do anything outside the scope of the current project, we took away the third obstacle of publishing a component.

At first, there was some resistance within the company against this new approach because it was expected to result in rigid components. This opposition however, turned out to be unjust given the fact that software always needs to have some flexibility in order to cope with changes, even for the current project. Furthermore, if the project’s goal is to deliver a flexible system, flexibility is part of the requirements and consequently will be implemented. The idea is to not make it more flexible than specified

By taking away the extra effort needed for publishing a component, there is no more interference in a project when it has a component spin-off, apart from some small administrative work. Modifications that might be necessary for reuse are made later, in the budget of the appropriate project, relying on the adaptability and code clarity, established through XP principles.

These measures have resulted in a higher component production already, although it is too early to tell if it is significant. Still, running projects are already less bothered whenever a components are created from them.

6.2 Creating components through refactoring

By frequent refactoring we also expect to increase component production because many refactoring practices focus on isolating reusable elements. The following refactoring methods generate reusable components from software:

Inheritance refactoring: 2 classes that implement similar behavior are made to use one shared superclass. Chances are higher that this superclass can be reused in other software, simply because its functionality is more abstract than its subclasses.

Composition refactoring: one class implementing two responsibilities that are not related very much, is refactored into two different classes. Creating smaller classes does not make the current system lighter, but it improves reusability because smaller, more lightweight parts can be reused.

7 Insufficient communication

Often, as was the case at Sentient, a software development organization is separated into groups (e.g. departments, rooms, offices, product lines), each with its own subculture (standards, values, technology). One of the goals of Component Based Development is to increase the reuse of software within an organization. Bringing subcultures together is essential for that reuse to succeed, in order to understand each other's components and documentation.

Within Sentient, many of the process improvement efforts are focused on improving communication, mostly through the use of tools such as electronic discussion groups, requirement and change management, automatic change notification, a searchable knowledgebase and messaging tools. Other improvements have been the development of unified terminology, documentation rules and coding standards.

Because of the importance of code as a communication means, we developed a standard for every language used, as well as a standard for commenting code. We configured a commercial reporting tool to use these comment templates to create hypertext manuals of the code. These manuals are directly available from the development environment, in a context sensitive way. There is no other code-documentation than the code and its comments, just as XP prescribes.

For components, there are two kinds of documentation: for the component developer and for the component user. The latter carries a special tag in Sentient's comment standard, so two documentation reports can be generated: one for the developer, and one for the user, containing just an explanation of the public interface.

8 Conclusions

This paper has shown how we dealt with our problems with Component Based Development by applying Extreme Programming principles. These principles are very well suited for components because they address the relevant key issues: high quality payoff, and long time adaptability. The results are as follows:

- By introducing pair programming, unit testing, frequent refactoring and timebox planning, component adaptability and quality increased.
- Collective code ownership was introduced in order to be able to change a library component quickly and reduce pressure on component creators.
- By building nothing more than exactly needed (keep it simple), unnecessary middleware complications and extra stress on the developer and the project were eliminated. Traditional software design and especially CBD tell us to plan for the future, to design for reuse. In contrast, XP says to solve today's problem and trust your ability to add or change features in the future, when necessary.
- Because of collective code ownership, high adaptability and keeping it simple, most of the reasons for not publishing a component have been removed
- Through constant refactoring, more new components are being created.

There is still a great deal of legacy code that is not very adaptable within Sentient. We treat this code differently than we treat the new sourcecode. Ownership for example is restricted. However, we are constantly refactoring our legacy systems, increasing adaptability, towards collective ownership.

9 References

[1] K. Beck, Extreme programming explained, Addison-Wesley, 1999

[2] M.Fowler et al., Refactoring: Improving the design of existing code, Addison-Wesley, 1999

[3] D. Kruglinski et al., "Programming Microsoft Visual C++, fifth edition", Microsoft press, 1998.

10 About the author

Rob van der Veer is head of software development at Sentient Machine Research in Amsterdam, The Netherlands.

He studied computer science at the University of Twente and graduated in 1993 with a research project on hybrid artificial intelligence – combining AI techniques from different disciplines. This research was performed at Sentient Machine Research and resulted in technology that has become the heart of Sentient's data mining toolset – DataDetective.

e-mail: rob@smr.nl

Company website: http://www.smr.nl