Evaluation of a Hybrid Software Development Strategy for Reengineering Large Legacy Applications

Daniel Heddelin
Evaluation of a Hybrid Software Development Strategy for Reengineering Large Legacy Applications

Daniel Heddelin
Dept. of Computer Science, University of Trollhättan/Uddevalla, P.O. Box 957, SE-46129 Trollhättan, Sweden
daniel@heddelin.se

Abstract

Software is an enormous and important field, and it helps us access and visualize information in a vast number of ways. However, the advanced software systems the society demands expand in size and complexity which pushes the limits of what can be developed. This technological advancement weakens the business value of legacy systems and developers struggle with modernizing these systems and still keeping the functionality intact.

This thesis will use the reengineering of StålSpec 2000 as a test case, and evaluate the hybrid software development strategy being used in this legacy application reengineering. The thesis investigates whether a combination of the Rational Unified Process and Extreme Programming software development methodologies is more effective than using only one methodology when reengineering legacy software. The aim of the test case is to find the best combination for a small development team with limited resources. The average software developer in a similar situation will find this study highly interesting since it shows how to customize an agile and flexible hybrid software development strategy for reengineering legacy code.

In order to fully understand the depth of this thesis, intermediate knowledge within the Software Engineering field is required.

1. Introduction

Software is a huge area on which modern businesses are run and governments rule. Software helps us access, create and visualize information in a vast number of ways and forms. The progress in software has definitely helped to drive the growth of the world’s economy, and it is an indispensable part of our modern world. World wide economies depend more and more on software, which is good news for software engineers. [1]

But the kinds of advanced software systems that the society demands expand in complexity, importance, distribution and size. This pushes the limits of what the software industry knows how to develop, and constant technological change weakens the business value of legacy systems that have been developed over several years through huge investments [2]. But reengineering legacy systems to modern technology poses a new set of technical problems.

Many complex legacy systems are still very important, and a lot of research is underway to discover effective techniques for reengineering and modernizing the vast amount of legacy code that is in use today [3, 4]. Developers struggle with the problem of modernizing these systems and still keeping their functionality intact. This problem will not disappear. Even systems being built today, according to the best practices known, will also need to be reengineered eventually as new software engineering processes evolve and technology advances.

Even though more cost-effective technology is available, about 80% of all IT systems are still running on legacy platforms [5].

This thesis will use a program called StålSpec 2000 as a test case, a large legacy application that needed to be rewritten entirely. More information about the background to this can be found in section 2.

Taking on such a big project does not only require courage, but also good structure. Thankfully, we are not the first persons to develop software, so a few different development strategies already exist. These strategies have been developed to work as best practices for software developers, and are meant to help with creating well structured and stable software. Even though all of these methodologies strive for the same goal; to be a silver bullet; to create “perfect software”, they look very different.

We decided to investigate two of the better known methodologies, the Rational Unified Process (RUP) and Extreme Programming (XP), and came up with an
interesting matter. Both RUP and XP leads to high quality software, but not all parts of these strategies suit us. Should not a hybrid of these two strategies provide a result that is at least as satisfactory or maybe even better, since we actually select parts of both strategies that suits us well?

To achieve a scientifically sustainable answer to this, all three approaches would have to be tried; one team developing according to the Rational Unified Process, another team using Extreme Programming and a third team developing with a hybrid of RUP and XP, but since there were not enough resources for this, the goal of this thesis is to assess the work completed by us and draw a conclusion of how successful our development strategy hybrid was.

The background to our specific application can be found in Section 2, and information on the Rational Unified Process and Extreme programming development strategies can be found in sections 4 and 6. In section 3, critical mistakes in StålSpec 2000 are being investigated whereas section 5 and 7 shows our choice of the hybrid software development strategy applied when creating the new software for MVR. Results, conclusion and recommendations for further work can be found in sections 8, 9 and 10.

This thesis will refer to the creator of StålSpec 2000 as ‘Carl’.

2. Background

Mekaniska Verkstädernas Riksförbund (MVR) today has an application written in Visual Basic 6 called StålSpec 2000. StålSpec 2000 deals with many different issues when it comes to dealing with steel and steel management, all the way from making tenders and calculating prices, weights, amount of working hours for manufacturing the requested amount of steel, to optimizing steel cutting to minimize steel waste.

Unfortunately, StålSpec 2000 contains a lot of bugs of different kinds and of course neither MVR nor the companies that bought licenses to use StålSpec 2000 enjoy this. Our task was hence to do something about this issue.

StålSpec 2000 is a very advanced program that contains an enormous amount of functions, and there is no software like it, it is truly unique. Many of the actions StålSpec 2000 performs are highly advanced, such as the cutting optimization. This level of advanced software requires a very structured development. It is obvious that the person who made StålSpec 2000 had little or no knowledge about this, so it is not strange that the program is so full of bugs.

At first, MVR wanted the bugs to be fixed so that the program would run smoothly without crashing. They realized that the bugs were so numerous that it would be more desirable to have the program rewritten entirely in a modern programming language. In this way, all the bugs would (hopefully) disappear, and the application would be both easier and cheaper to maintain since Visual Basic 6 is getting old. The new code will also without doubt look much better; the code in StålSpec 2000 is hardly readable.

Since our task was to refactor something and not make something new, we did not have a specified set or requirements, thus making a complete implementation of one software methodology difficult, if not impossible. Instead we decided to pick parts from the Rational Unified Process and Extreme Programming; parts that suited our needs.

3. Critical Failures of StålSpec 2000

There is a vast amount of problems that could occur when developing software but many of the symptoms are common. In StålSpec 2000 it is not hard to see which the root causes to the evil code are. The developer has hardly been following a development strategy.

3.1. Code Smells

The term Code Smell [6] is a hint that something is wrong somewhere in the code. This smell can be used to track down code problems. Note that a code smell is a hint that something is wrong and not a certainty, but it still is a sign that a closer look is warranted. StålSpec 2000 is full of code smells. Here are some of them:

3.1.1. Duplicated code. The StålSpec 2000 code is full of redundancy and duplicated code. So full it is hard to read the code at all, more so understand it.

There are two kinds of code duplication. One is functional duplication and the other is exact duplication. Assessing the functional duplication that exists in StålSpec 2000 would be very difficult and time taking. It would require manual review of the code, enough to gain an understanding of what the different methods do, in order to decide whether a certain method is duplicated or not.

The exact duplication however, is easier to measure. There are in fact tools for automated duplication detection that can be used for this operation, one of which is called Simian [7]. What Simian does is that it compares lines of code with each other and then gives a summary of any possible duplication. In this case, for Simian to give correct figures of the duplication in StålSpec 2000, all the code that had been generated by Visual Studio 6 had to be removed, so that Simian only examined the written code. But even after this, the result was shocking. Simian had found no less than
1024 duplicated lines of code in 66 blocks in 10 files, and the size of the blocks ranged from chunks of 9 lines up to 68 lines of exactly duplicated code. The average size of the duplicated blocks was 32 lines.

3.1.2. **Classes with too many instance variables.** This is another one of the code smells. StålSpec 2000 however, does not suffer from this odor, but rather from another variable issue, namely global variables. About 75-80% of all the variables in StålSpec 2000 are global variables. The code intro consists of page after page with declaration of globals, several per row.

On the run-time of a program, the main procedure call overhead includes global variable access [8]. But globals do not only decrease the performance, they also cause instability. Some of the arguments raised against the use of global variables are [9]:

1. Functions modifying variables other than their own locals can cause a number of surprises. If pass-by-reference [10] is used then aliasing can occur when not expected. Aliasing is when several different identifiers refer to the same object.
2. Indiscriminate access. The programmer can not prevent a sub-procedure from modifying the values of a local variable’s procedures.
3. Vulnerability. New declarations could be interposed between when a variable is declared in an outer scope and when it is used in an inner scope.
4. No overlapping definitions. Shared access to variables is difficult to control.

StålSpec 2000 suffered from very poor performance, sometimes it even crashed, seemingly for no reason. There is no doubt the enormous amount of global variables was involved in causing program termination.

3.1.3. **Classes with too much code** [11, 12, 6, 13]. In fact, StålSpec 2000 consists of more or less ONE class with all the code. The other files are GUI form files which contain very little code apart from graphics. 40.000 lines of code in one file with hundreds of global variables will never make a good and stable program.

The big module file should have been split up into smaller classes in accordance with the “hi cohesion” [11] pattern. A class should be kept as uncomplicated as possible. Functions that are not necessary for the respective class should not be there.

3.1.4. **Instance variables that are not always initialized**, because the class is so full of things the initiation of some variables is forgotten. This may cause heisenbugs [14], which can lead to a highly unwanted software performance. A heisenbug is a bug that appears randomly since it is depending on what already exists in the memory of the computer. Because the bug is random it is very hard to debug it since it is difficult to find.

3.2. **Other Software Development Failures**

Apart from the failures described as code smells, there are a lot of other software development malpractices that you can notice in StålSpec 2000.

3.2.1. **Ad hoc requirements management.** Gathering requirements is an important step when developing software, which tells the developers what the end-users really want. Without a well-defined development process, the development team will proceed in an ad hoc manner, with success relying on major efforts of a few individual contributors. This is not a sustainable condition [1].

Looking at the cluttered architecture of StålSpec 2000 it is easy to see that Carl had not followed a well-defined development process. Mr. Svedje, the CEO of MVR, also told us that Carl had visited CEO’s of certain companies to gather requirements. A new build of StålSpec 2000 was not released to all clients after all the requirements had been gathered from the users, and from this set decide which ones to implement. Instead, end-users requirements were implemented there and then, on the spot at the customer. This lead to a large amount of versions of StålSpec 2000 though no major changes had been made to the entire software’s functionality; only small tweaks that individual users had requested.

3.2.2. **Inability to deal with changing requirements.**

The requirements for software systems are always changing. One reason for this is that different users have different requirements and priorities. These may be contradictory or conflicting, so the final system requirements are inevitably a compromise. Now let us reason; StålSpec 2000 is an application that today has circa 150 licenses spread all over the country. If a user discovers a bug, that bug probably exists not only in that certain users’ version but also in all the others. Current best software practices [11, 12] advocate collecting the user input into a document then taking care of the collection of requirements/bugs that are found crucial, and then released a new version [13] of the software. In retrospect, these best practices should have been implemented with StålSpec 2000. As for the personal requests some users had, best software practices would have left to the project leaders to decide what should be implemented for a new release.
of StålSpec 2000, and not let one developer take it in his own hands.

3.2.3. Late discovery of serious project flaws. StålSpec 2000 is a big program with endless functions. It did not evolve overnight, it took months to complete, if not even years. When looking through the code it is obvious that very few lines seem to have a lot of systematic thought behind them. Had someone noticed this in the beginning of the creation of StålSpec 2000, it would most likely be a more robust program today.

Instead, the flaws are so many and vast that tools that exist to convert Visual Basic 6 code will not even work on the source code of StålSpec 2000.

3.2.4. Indefensible naming of variables. When analyzing the code of StålSpec 2000 you encounter many hundreds of global variables. They are all very badly named. Many of them are only one character wide while others are given random names that makes absolutely no sense.

3.2.5. Unacceptable software performance. This is the reason Mekaniska Verkstärdernas Riksförbund wanted us to recreate StålSpec 2000 in the first place. The flaws and faults in the application are many; it even contains bugs that are severe enough to cause the program to terminate. The users find this unacceptable.

"StålSpec 2000 crashes all the time. My employee’s and I have stopped using parts of it because it never works as it should" [15].

4. A Brief Introduction to the Rational Unified Process

The Rational Unified Process is a software engineering process which provides a disciplined approach to assigning responsibilities and tasks within a development organization. The goal of RUP is to ensure high-quality software is being produced that meets the needs of its end-users, within a predictable budget and schedule.

RUP uses the Unified Modeling Language (UML) [16] as the main notation for the several models that are being built during the development. A model is a simplification of reality that describes a system, or a part of a system, from a particular perspective. Modeling is an important part of software development, and using a standard modeling language like UML helps the development team to visualize, construct and document the structure of a system’s architecture. In short, visual modeling improves a team’s ability to grasp and manage software complexity [1].

At the core of the Rational Unified Process are fundamental principles that represent the spirit of RUP. Among the principles behind the spirit of RUP are [17]:

- Attack major risks early and continuously; or they will attack you.
- Solve the most critical problems first.
- Working iteratively, like in figure 1, will help discovering risks early on.

Figure 1. The basic flow of the Rational Unified Process.

- Accommodate change early in the project.
- Software is too complex in order for the requirements, design and implementation to be totally correct from the beginning; they will change. Working iteratively will make it easier for changes to be accommodated in the inception, elaboration, construction and transition phases.
- Stay focused on executable software.
- Requirements and designs are important, but fine tuning could go on forever. The development team needs to progress and stay focused on producing software that works, not software that is completely flawless and perfectly designed.
- Baseline an executable architecture early on.
- The architecture provides a skeletal structure for your system, and helps getting a good overview.
- Build your system with components.
- Components encapsulate data and functions that belong together. This means changes are isolated in the appropriate unit
and hence will help changes accommodate better.

Figure 2. The Spirit of RUP.

- Ensure that you deliver value to your customer.
  The developers need to stay focused on the key requirements, and not forget about other users and the stakeholders in the requirements work. Figure 2 shows the steps of use-case-driven development which is a key driver in RUP’s iterative approach. This helps not only to capture the requirements, but also to implement and share them with the customers.

- Make quality a way of life, not an afterthought.
  The later problems are discovered, the more expensive. This makes a quality driven iterative process important to follow.

- Work closely together as one team.
  Software development is in most cases a team effort. It is important that the team is empowered and self-managed, and can communicate effectively for high-quality software to be produced. [17]

5. Parts of the Rational Unified Process That Suited Our Development Strategy

The Rational Unified Process captures many of the best practices in modern software development. In particular it covers six practices [1]:

5.1. Develop Software Iteratively

We developed in an iterative fashion and tried to follow figure 1 above, much because we were very aware this project was filled with risks; we had seen what can happen when not developing iteratively. We made sure to break down bigger methods and problems into smaller ones, and applied iterative development on each part.

5.2. Manage Requirements

The requirements management proved to be something rather complex in our case. In the bottom, “StålSpec 2000” was the only requirement, but the reference group had a lot of additional requirements than the original functionality.

Identifying a system’s true requirements - those that weigh most heavily on the system’s goals - is a continuous process. We had to deal with changing requirements since we got new information on each meeting we attended. As an evolving system changes a user’s understanding of the system’s requirements also changes.

5.3. Use Component-based Architectures

Component-based development (CBD) is an important approach to software architecture because it enables the customization and reuse of many commercially available sources. In our case, third-party components were not of interest, but we made sure our new version of StålSpec 2000 was built with a component-based architecture. This would make it both easier to customize our new application, but also to exchange or remove whole components, as well as easily adding new ones.

5.4 Visually Model Software

A model is a simplification of reality that completely describes a system from a particular perspective. Modeling is important because it helps the developers to visualize the behavior of a system’s architecture.

On each and every occasion we were coding, we were in a room with a whiteboard which we used frequently. Since we had StålSpec 2000 as a ground, we drew class diagrams and other UML diagrams off of the information we would extract from running certain functions in the program, and then alter them as we found necessary.

Since none of us had any extensive knowledge within the steel industry, many complicated functions
in StålSpec 2000 were difficult to understand, so the kind of agile modeling that we used proved to be very efficient for us.

5.5 Continuously Verify Software Quality

Software problems are much more expensive to find and repair after deployment than beforehand. For this reason, it is important to continuously assess the quality of a system with respect to its functionality.

Since StålSpec 2000 was full of functional bugs we practiced frequent testing throughout our development to make sure each and every part worked as it should and gave the correct values.

5.6. Control Changes to Software

One of the challenges when developing software-intensive systems is the cooperation with multiple developers. Working together on multiple iterations and releases might cause problems without disciplined control. Coordinating the activities and the artifacts of developers is an important task which allows a better allocation of resources based on the project’s priorities and risks. This, together with developing software iteratively helps with monitoring changes continuously.

We made sure to thoroughly allocate our resources as good as possible since we did not have much time for such a big project. Using a Concurrent Versions System (CVS) [19] made it easier for us all to track the changes to our software.

6. Parts of the Rational Unified Process We Did Not Implement

When developing the new StålSpec we made sure to work iteratively, carefully manage requirements, create components, verify our software, control any changes that occurred, and also modeled our software visually.

As mentioned in section 5.4, the modeling we did do was drawn on a whiteboard. This kind of flexible modeling was very useful for us since it was fast. RUP however, recommends using UML modeling tools to create many different types of diagrams, such as object diagrams, use case diagrams, collaboration diagrams component diagrams and so on. This approach provides extensive documentation but takes a lot of time to do when working with a project as large as StålSpec 2000.

With much more time, we would most likely have made sure to structurally draw many of those diagrams using a tool to create solid files, instead of making instant sketches on a whiteboard.

7. A Brief Introduction to Extreme Programming

Extreme Programming is a relatively new discipline that encourages a number of practices, one of which is team communication. XP favors simplicity instead of complex solutions. You could say that Extreme Programming is a rearranged and simplified version of the traditional approach of software development.

“Extreme Programming is a discipline of software development based on values of simplicity, communication, feedback, and courage. It works by bringing the whole team together in the presence of simple practices, with enough feedback to enable the team to see where they are and to tune the practices to their unique situation.” [20].

XP is called “extreme” because it takes common, accepted programming practices and implements them with extreme devotion. XP also challenges many conventional ideas behind software development, such as the theory that the cost of changing a piece of software will always increase drastically over time. Extreme Programming disagrees, and uses practices to reduce this cost. For example, XP takes advantage of the benefits of being able to easily and comfortably change software late in its lifecycle, even when the software is already a product that has been shipped to the user community. The waterfall approach that the traditional software developers follow tends to mask the real risks to a project until it is too late to do anything meaningful about them [1].

But instead of planning, analyzing and designing for the distant future, XP programmers do all of these activities, a little at a time, throughout development.

8. Parts of eXtreme Programming That Suited Our Development Strategy

Extreme Programming consists of 12 core practices, of which a number suits our development well. The practices in XP suitable to us will be briefly described below, one and one, followed by a description of how we applied the specific practice to our project.

8.1. The Planning Game

The customer and development team cooperate to produce the maximum business value in as little time as possible. The planning game takes place at different scales, but the basic rules are the same.

a) The customer comes up with a list of desired features for the system. Each feature is written out as a “User Story”, which gives the feature
a name, and describes what is required in broad strokes.
b) The development team estimates how much effort each story will take, and how much effort the team can produce in a given time interval; the iteration.
c) The customer then rates the priority; decides which stories to be implemented and in what order. [21]

Since StålSpec 2000 is full of bugs, both MVR and the users of StålSpec 2000 are anxious to receive an application that really works reliably. Hence it was not hard for Lars Svedje, the CEO of MVR, to gather a reference group to give us ideas and tell us about desired features. This reference group, which whom we have had a couple of meetings in Stockholm consists of, except for Lars himself, three CEO’s from companies that use StålSpec 2000.

On the first meeting we had with the reference group, we sat down and went through the different parts of StålSpec 2000, step by step, to find out what needed to be done. The reference group informed us about what worked as it should and where different bugs were. They also explained to us what the desired features for the system were. This is how we gathered more extensive information about the system requirements. Instead of only having “StålSpec 2000” as our “requirement”, we also wrote down the input from the reference group as User Stories and used these as requirements [22].

With this information in front of us, we talked openly about what we thought we could produce given the little amount of time we have. We were also told which parts were crucial and needed to be implemented first. After further discussion the original User Stories had changed, and looked more like something we actually can accomplish within the frame of the project duration.

8.2. Simple Design

XP practices say you should always use the simplest possible design that meets the current requirements. The requirements may change as soon as tomorrow, so do only what is needed to meet today’s requirements. Do not anticipate future stories or features. Write only the code that is needed to complete the current task. Of course it is necessary to ensure that a good design is used, and in XP this is brought about through refactoring.

Refactoring is the process of changing a software system to improve its internal structure and reusability without altering the external behavior of the software itself.

Simple design is something we certainly took into consideration. StålSpec 2000 consists of spaghetti coded Visual Basic 6, and MVR wanted the new application to be easy to maintain. Hence, we decided to build the program as a set of components; each component representing one User Story – one feature.

8.3. Refactoring

Programmers practice frequent refactoring in order to keep the code as clean and simple as possible. If duplication is found, it should be removed or altered as necessary.

This activity took place through all of our development, especially in places where we encountered problems, since those places got extra attention and more thorough review.

We noticed that the days we spent on pair programming lead to more effective refactoring, we felt a better flow. An empirical study on how much more effective it was in our case was not done, but studies have found that pair programming increases the productivity by 15% [23]. We estimated the initial amount of logical errors to two per 100 lines of code, and that this was reduced by roughly 40% as a result of working in pairs. We could also work longer without losing headway.

8.4. Pair Programming

All production code is written in pairs, two programmers working together at one machine. This means all code is reviewed as it is written. Pair programming has been shown to produce better software at similar or lower cost than programmers working alone. It also strengthens the team spirit of the development team.

Pair programming is an interesting and controversial part of extreme programming. Many programmers of the old school, used to traditional development methods, might not consider this to be a good and efficient way of working [24, 25], but it really does improve the quality of the code that is being written [26].

Both since there were three persons working on the application, and since the application consisted of many smaller parts, we did not spend all the development time programming in pairs. We picked different parts of the program to work on, one User Story each. Much of this consisted of very simple things, such as making the Graphical User Interface (GUI) for each component, and writing basic functions. But when one of us encountered a problem, or knew that something massive was ahead, we applied pair programming.
8.5. Collective Code Ownership

All the code belongs to all the programmers. Any developer pair can check out and modify any part of the codebase at any time. This means that all code gets the benefit of many people’s attention, which increases code quality and reduces defects.

We stored all our code in a CVS repository. All the code was available to all of us at all times, and on more than once occasion we worked with the same piece of code, not only to help each other out. This helped all of us get a wider understanding of the new application as a whole since we focused on more than our own code.

8.6. Continuous Integration

The team integrates changes into the codebase at least once per day. This enables very rapid process and keeps all the programmers on the same page.

In the end of every day we all made sure to integrate the code we had been working on during the day. This was not done only on a daily basis, but also when one of us had completed a certain bigger part, or User Story. Doing this made it easy for us to actually see our progress, as well as assuring us that the newest code available was accessible.

8.7. 40-Hour Work Week

Programmers go home on time in order to be able to maintain a sustainable rate of effort. When it is required, up to one week of overtime is allowed. Two or more weeks of overtime, is a sign that something is very wrong with the process.

This is probably one of the hardest things to obey in most software development, and the situation does not get any simpler when having a project that demanded more time and resources than those available to us during the course of this work.

We did however try to limit the time we spent on working together each day to fit inside the 40 hour frame, but many of the days the work continued in the afternoons and evenings after getting home.

Since there were three individuals in our group working on the project, but we each had to write individual theses, the time spent together mostly went to programming, and the writing of our theses had to be transferred to our “spare time”.

8.8. On-site Customer

An on-site customer is a customer that the development team has continuous access to; someone who will actually be using the system. “Lack of user involvement traditionally has been the No. 1 reason for project failure. Conversely, it has been the leading contributor to project success. Even when delivered on time and on budget, a project can fail if it doesn’t meet user needs or expectations.” [27]

Mekaniska Verkstärdernas Riksfördun’s main office is situated in Stockholm, approximately 420km from Trollhättan. This makes the on-site issue difficult. But both Lars Svedje and the reference group were very interested in our success, so we definitely had continuous access to these people, even though it was mostly based on e-mails and phone calls.

8.9. Coding Standards

For a team to work effectively and to share ownership of the code, all the programmers need to write the code in the same way. At its best, you should not be able to tell by looking at the code who on the team wrote a specific piece of code. The specifics of the coding standard are not too important; just that all team members follow the same standard.

When adopting a coding standard there are many things to think about, and not only where to place curly brackets. We decided in the beginning of the project that it would be a good idea to standardize the way we wrote our code, especially since we knew at an early stage that we would work a lot with “each other’s” code. [28]

- Use meaningful, descriptive words to name variables. Do not use abbreviations, do not use single character variables, except for well known counter variables such as i.
- Names on certain objects should have a specific prefix. E.g. a name on a textbox begins with txt, a label with lbl. This should make the GUI construction and maintenance easier.
- Indentation. In order for the code to be easy to read it is important that the persons coding indent in the same way, we decided a certain indentation rule and stuck to it. Tab should be used for indentation, not space. Curly braces should be in the same level as the code outside the braces. One blank line should be used to separate logical groups of code. Comments should be below one blank line and in the same level as the code.
- Camel Notation (first letter in lowercase) should be used for variables and Pascal Notation (first letter in uppercase) for methods. Each word after the first word in the name of both variables and methods should start with a
capital letter. For example: steel, showHeading, retrieveRowValues.

- Method names should tell what it does. Do not use misleading names.
- Display descriptive error messages. An error message should help the user to solve the problem; “An error occurred” does not do this.
- Do not comment excessively. Comment only wherever required; good readable code does not require many comments.
- Catch exceptions thoroughly. Give the user a simple message but log the error with all details, especially where it occurred.

Microsoft Visual Studio .NET 2003 is good at making the indentation look good, and placing curly brackets and such in the right places, thus making a common indentation standard somewhat easier to follow.

9. Parts of Extreme Programming We Did Not Implement

Not all the twelve core practices of Extreme Programming suited us. Here are a few more with an explanation to why we did not implement these.

9.1. Small Releases

The developers start with the smallest useful feature set. Release takes place early and often, adding a few features each time which gives the Customer a better opportunity to steer the project, or put it back on track if it is deviating.

This is a practice we did not implement. The project was simply too large for any releases to be made. On the meetings we had with the reference group we made sure to demonstrate our work at that point though, but we never reached far enough to make a release.

9.2. System Metaphor

The system metaphor is a practice that prevents the system from going sour; it is another practice that is focused on design. The system metaphor for a project defines the architecture of the application and enables developers to talk about the aesthetics of the overall design.

As for the aesthetic design of the new StålSpec, MVR had already told us to follow the already existing design so the users would recognize themselves when using the program.

9.3. Testing

The team focuses on validation of the software at all times. Programmers develop software by writing tests first. Customers provide acceptance tests that enable them to be assured that the features they need are provided. Unit tests are automated tests written by the developers to test functionality as they write it.

We did not implement test-first programming. Instead we focused on practically testing what we had coded did what we told it to do. StålSpec 2000 would perform a certain action as expected, but after Carl had added something new on a completely different place in the source code, this certain action would invoke an error. This fact made us extra cautious during our implementation, so a lot of practical testing was done.

10. Result

Developing software is a complicated process, and with the increasing demands of society it does not get any easier. We knew this long before we started working with Mekaniska Verkstärdernas Riksförbund. It did not take long though before we noticed StålSpec 2000 includes a vast amount of functions that we were never going to be able to recreate in the short period of time our exam work offered. After having decided which parts were the most important, we were faced with the challenge that there was still a very limited time frame to complete even these.

Due to limited time, we realized it would be suitable to adopt one of the agile software development methodologies such as Extreme Programming. There was a lot of pressure, so agility was the keyword to us. We had StålSpec 2000 to analyze and recreate, so we felt that using the Rational Unified Process, which would mean restarting from scratch with hundreds of possible use cases leading to hundreds of other diagrams would simply be too time consuming. But at the same time we knew that RUP offers good structure and its diagrams are still very useful; therefore we used the parts of RUP you read about in section 5. You could say we made RUP more lightweight, more agile, by not using strict rules for diagrams and such.

Even though we worked as diligently as possible and tried our best to reengineer the most important parts of StålSpec 2000, we did not complete it on time. However I do not think it was realistic for any of us to believe we could have completed the entire task at hand in the time that was allotted, had we worked in any other way. Our achievement was not how much we accomplished, but how well we succeeded in working according to our choice of development strategy. Let us not forget that MVR asked for an upgrade of StålSpec 2000, a program without bugs
written in a modern programming language. This is what we have done. We may not be finished, but so far MVR is pleased with our progress.

“It is a shame StålSpec 2000 was so poorly designed that you have not been able to reach further with your work. What you have done looks great and we are hoping you are willing to continue the development even after your exam work, but I do not believe this new version of StålSpec is ready yet to meet the public eye. It would have been exciting to be able to bring a demonstration of the new, functional StålSpec to our congress. Oh well, you will impress the audience on our next congress instead.” [29]

The obtained results were so encouraging that the customer decided to entrust us with reengineering the entire system.

11. Conclusion

Despite all the challenges, legacy modernization is crucial for organizations spending much to maintain the value of their obsolete information systems. Adopting newer computing systems can decrease operating costs. Completely rewriting obsolete information systems will eliminate legacy applications in which the code quality is too poor to migrate. StålSpec 2000 fits in this category.

This thesis has illustrated how a small development team with limited resources can tailor parts of software development strategies into a development strategy hybrid, in order to be able to create high quality software in a flexible way.

The methods used proved to be very useful since they achieved what was originally intended, and we conclude that the average software developer that stands before the challenge of reengineering a large legacy application will find this study interesting, since it shows one could customize an agile and flexible hybrid software development strategy for reengineering legacy code.

Had there been more time, a lot more time, we would advocate using a tool for UML modeling such as Rational [30] or Microsoft Visio [31] and actually creating the UML diagrams. They provide very good documentation and are useful in many ways. In our case it would have been a clear disadvantage to have tried to do this since it would have taken all our development time away.

We found our means of practical testing to be satisfactory when developing the new StålSpec. In a more security intense project, written tests may have been preferred to provide an even more fail-safe environment.

12. Recommendations for Further Work

We are uncertain about the future usage of StålSpec, mostly because neither our customer, MVR, nor the users of StålSpec 2000 have much computer knowledge. It is very possible that the next version of StålSpec becomes a client-server system with the most computer demanding functions on a powerful server, and only thin clients to access the system.

As for the future development strategy of the continued work with the new version of StålSpec, there is not much to add. What we did worked very well for MVR and for us, and we intend to continue using this hybrid method while continually assessing its effectiveness.

13. References


