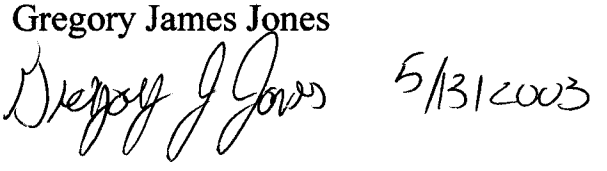
The Software Engineering Process:
Insight into Building Machines of the Mind

An Honors Thesis (HONRS 499)

By

Christopher J Helms

Thesis Advisor
Gregory James Jones



Ball State University
Muncie, Indiana

May 2003

3 May 2003

Abstract

My intention is for the reader to get an insight into the software development process. It is written for the non-technical reader. The thesis includes an explanation of the program, an explanation of software engineering and some of the practices as well as how they relate to the engineering of the program. A description of the design process, the programming language used and why we chose that language will be given. Next there will be a short description of the team members and what their roles were. Finally a post-mortem of the project will be given including certain items that went right as well as what went wrong. I wish to instill in others a greater sense of what happens on a small-scale software project. Perhaps this will give them a greater appreciation each time they start up their computers and use their favorite software program. Lots of work goes into creating even the smallest program, and I take less for granted being involved first-hand in the creation of such a program, and I hope I can pass that along.

Acknowledgements

- I wish to thank Jim Jones for advising me through this thesis paper. If it were not for his suggestions and comments, this paper would not have realized the potential that it has found. I also would like to thank him for keeping me going in the honors program; since there were many times I would have left it for an easier way out. You have been a great friend, boss, mentor, and advisor throughout most of my college career. Thank you for being there for me.

- I wish to thank my parents for always being there for me. Although it is a long way from home, they were always a phone call away.

- I wish to thank my Software Engineering group for making my life during the past year a little more interesting.

The Software Engineering Process:
Insight into Building Machines of the Mind

The Theory of Software Engineering

Somewhere along the line, it was discovered that if you apply a process to programming, the software that is generated becomes more consistent. This process is known as Software Engineering. "Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines." (Pressman 20) This means that when you apply engineering methods to software as one would to building a physical product, the end product will have the potential to work better than one that did not utilize these methods, and it will be able to do so with fewer revisions.

Just as engineers build machines through engineering, software engineers do likewise. Since software "machines" are not tangible, they can be thought of as machines of the mind.

Explanation of Software Engineering Course

The software engineering sequence at Ball State University spans two semesters at the end of each computer science undergraduate's career. During the sequence, students are grouped together toward the beginning of the first semester with two to three other students. These are the people that will be with them through the course of the year. The goal of that year is to learn about the practice of software engineering and apply those techniques to a software-based end-project for a "real world" client or organization.

As the class learns about software engineering methods, practices, et cetera, we also get to think of or choose a project, and then eventually propose it to a client who will accept a set of requirements. Throughout the first semester and several weeks of the second

semester, we plan the requirements, scheduling, and create many other documents and diagrams that will help us later in the engineering process when we start coding (slang from the actual writing of the programs computer source code). In the second semester, we take all of these documents and start to build the project, completing it by the end of the year.

The purpose of the course is not just to learn how to make software. After all, that's what we have been doing our entire college careers, and sometimes starting before that. The software engineering sequence is designed to teach us a few more things. One is teamwork; working in a team is completely different than working alone. You are removed from all but what your tasks are, and it is amazing seeing everything start to fit together toward the end of the project. We also learn about project management and design through reading, outside research, and in-class lectures. Finally, we are supposed to apply all of this acquired knowledge towards the creation of our final project. It's amazing seeing how all of these details fit together and match what you do with your team on the project.

SiRS

Our program has been given the title of Simplified Reporting Software, or SiRS for short, created by the Coalition of Programmers, or Team COP. It is designed for the Ball State University Police Department to aid them in filing their Case Reports electronically in order to cut down on paper usage, and for ease of searching through their cases.

Previously, the Police Department had been doing most, if not all of their work with paper forms. They would store these forms in file cabinets, arranged by case number. Because of this system, there is no quick and easy way to search for specific cases unless you know the number. Also, filing cabinets have a limit of space and are very bulky. Several years ago, one of the officers with programming knowledge created a computer program to aid in the Department's work. However this program was not easy to use, it

was quite buggy, meaning it was not reliable, and the creator had left the University some time ago and was not around to support the outdated software.

Our team knew that we could create better software than they had, and make their current way of operating more efficient. With a copy of the forms they use in hand, as well as the specifications and requirements that we discussed with our client, and using the software engineering techniques we learned in class, we set out to create Simplified Reporting Software for the Ball State University Police Department.

Overview of Modern Common and Uncommon Practices

Here I have chosen a few methods of engineering to outline. Three of the methods were used by my team in one form or another, conventional methods, object oriented development, and rapid application development; and the other, extreme programming, is extremely interesting, and could be the wave of the future for software engineering.

Conventional Methods

Conventional methods of software engineering rely on a more linear and sequential approach. This means that first there is analysis, then design, then coding, and then testing. Sometimes there are variations on that model, such as creating a cycle out of the linear model, but it mostly stays linear.

Object Oriented (OO) Development

The object oriented approach is similar to the conventional methods approach. Object oriented software engineering in the past described using an object oriented programming language such as Java or C++, which emphasized code reuse. Code, or component reuse, describes the process of taking previously written code and making it work with your program, either by plugging it in or modifying it and then inserting it into the program. Now it has its own model that is evolutionary in nature, resembling a spiral, and emphasizes the reuse of code.

The model looks like a spiral because you make many trips around the same set of stages. These stages include customer communication, planning, risk analysis, engineering, evaluation, and then returns back to customer communication. This allows the customer to be involved in the beginning and end stages of development. Once you discuss the specifications with the customer, you can build the software, and then bring it back to the customer who will tell you what is right and what is wrong. You then continue the cycle until everything works as the customer expects.

Rapid Application Development (RAD)

The rapid application development approach means just that: it is used to create software rapidly. Like object oriented programming, it relies heavily on component reuse to save time and energy. Since it emphasizes reuse, many of the software components have already been tested, so that cuts down on the testing time as well. This can also be used with the prototyping model so that the customer is able to see mock-ups or not fully-functional prototypes of the program so that the developers can quickly get feedback from their clients.

Extreme Programming (XP)

Extreme programming is a new paradigm, only about six years old, which is designed for large projects with changing requirements. It improves upon processes in four distinct ways: communication, simplicity, feedback, and courage.

> "XP programmers communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. With this foundation XP programmers are able to courageously respond to changing requirements and technology." (Wells)

While relatively new compared to the methods mentioned above, many companies have already started to embrace its advantages. Such companies are Ford Motor Company, DaimlerChrysler, and quite a few others. We did not get to implement this new approach to software development, but I think it is very interesting and could be very popular in the future.

Using Those Practices

We combined aspects of the linear, object oriented, and rapid application development models for our project. We started with a linear approach because that is what was outlined in the book and lectures at first, and our project seemed to fit with it really well. We decided that our language would be C#, which is an object-oriented language. We figured that even though we were using an OO language and taking an OO approach, we could still fit our project to a linear model. And besides, we weren't looking forward to the extra work that would be required to rewrite our analysis to fit the object oriented paradigm. Since our time was limited, we also implemented a rapid application development approach, as well as creating prototypes. After getting the main interface finished, we filled in the blanks with the separate forms and other functionality that was required.

It seems like we came to the conclusion of what language we were going to use and what practices we were going to use at the same time. Each decision grew off of one another in a natural progression, so there was no second guessing any of the decisions. The object-oriented approach led to an OO language. Our language led to the ability to create prototypes in a rapid application development style. The progression just flowed naturally.

Designing the Program

Design is broken up into a series of documents in conventional methods of development. From the list of requirements, you should be able to create a data flow diagram (DFD). This is a diagram that breaks the program down into many levels, from general to very specific. As it gets more specific, you can see the individual functions of the program, how the data goes from one point to another, and also what transformations the data undertakes in the process. After creating the data flow diagram, you take that information and make a structure chart. This takes the functions in the DFD and breaks

them down into input, output, and transformations of data. You also will create a data dictionary, which defines the functions and the data for easy reference.

After the architectural design, you are able to design the user interface. This means that you give the ability to the user to access the functions that you will create. It is important to know your users when designing an interface for them. You will want to make the interface familiar to them. For example, when designing an interface for a bill-paying program, you could design the interface to look like a checkbook since that is familiar to everyone. This helps the user easily understand how to use the program and generally cuts down on learning time and training costs. We designed the on-screen forms to look as close to their written forms as possible, because the forms were familiar to our clients and potential end-users.

From the design phase, you are able to create test-cases. These outline how the finished program will be tested. One way to test is to make sure that all the requirements were met, and that they work as good as possible. Another way is to test the code itself. To be thorough, the tester should utilize many methods. Our testing phase was short simply because we were pressed for time, but it was important because it found errors that we were able to fix before delivering our project to our client.

Description of C#

Naming conventions for programming languages are very interesting, and usually include in-jokes that only programmers would (or would want to) understand. Long ago, there was a language called 'B'. Then, someone made several improvements on that language and called it 'C,' as C is the next letter after B in the alphabet. Several years later, more improvements were made on the C language, and therefore C++ was brought into existence. The "++" operator in C++ added one to a number, so in essence C++ was one language better than C. In the year 2000, Microsoft unveiled C# (pronounced C-sharp), which in musical terms is the note a half-step higher than C.

"The goal of C# is to provide a simple, safe, modern, object-oriented, Internet-centric, high-performance language for .NET development." (Liberty 3) Now with that sentence of technical jargon out of the way, we can concentrate on what each of those means. Starting with the last item, .NET, created by Microsoft, is a platform, which is a base for programs to run on similar to an operating system but more limited, with four major components. The first component is a set of languages including C#, a set of tools for building applications and services, as well as an environment to run those (known as the Common Language Runtime). Next we have a set of servers that provide specialized functionality for running the computer programs (or applications) created by the languages in the first set. The third component is a set of services that software developers can use to build certain types of applications. Lastly is a set of new devices that are ".NET-enabled," meaning they can run applications created to run on the .NET platform that are not necessarily traditional computers, such as cell phones, game consoles, et cetera.

The term high-performance is nearly self-explanatory. It simply means that an application will run to the best of its ability, and as fast and as efficient as possible. Internet-centric means that all the necessary tools needed to easily create applications that make use of the Internet are built in to the language. Object-oriented describes a programming language that allows programmers to focus on data and the manipulation of that data, rather than the old ways of focusing on the procedure that your program used. Modern describes its age as well as the fact that it is built upon the techniques for languages of the past, both good and bad. A modern language is essentially the next step in building software. Safe means that since the program runs in the .NET environment, which is a controlled area that limits the access of the program, it is not able to be potentially malicious to your system. Finally it is simple for a programmer to learn, since it is built on the fundamentals of several languages such as C++, Java, and Visual Basic.

Since C# is a new language that builds from three previous very popular languages, the designers got to pick and choose the very best features from each of them. C++ is a very robust language that is used to create powerful applications that can run very quickly and

efficiently, but the object-oriented aspect of it was added as an afterthought. Java is built from the ground up to be object-oriented, but it is inherently slow. Visual Basic was built for prototyping and rapid application development to pump out simple applications very quickly, but it is not able to create very robust or applications relying on speed.

Why We Chose C#

The semester previous to this sequence, I took a database design course. A database is a collection of information that is used to sort through or find relations between the data very quickly. In this course we had to design a program that would interface with a database file of our own making. It was also around this time that I had first heard about C# and was greatly intrigued. Actually using a computer programming language is the best way to learn about its features, so I set out to use it to create my database project. And, I was successful. I liked it so much that I persuaded my software engineering group that C# would be the perfect language for our project. Getting them to go along with my suggestion was made slightly difficult by the fact that I was the only one that had worked with it before.

There are a few reasons that they went along with my suggestion to use C# as our project's language anyway, and it wasn't just to get me off their backs. First of all, since C# takes its cues from three very common languages (languages that we had all been exposed to during the course of our studies), it would be instantly familiar to them after spending only a little time with it. The program that we would use to create our software, Visual Studio, is one of my favorite programs to use, especially when designing graphical interfaces. We knew that our project would probably be graphical in nature and would be able to make use of this programming environment. Visual Studio's designer is extremely simple to use, yet very powerful at the same time. It is because of this ease of use, as well as the fact that it is one of the only applications that is used to program in C#, *and* the fact that Visual Studio was free to us through the Computer Science department that we chose it as our primary programming environment.

We also knew that the target platform would be for a Windows desktop computer. Rather than going with Java, which is platform [computer] independent, yet slow, we chose to go with C#, which is designed to create Windows applications. Our clients are currently running on older model computers, so we would need for the application to run as quickly and efficiently as possible with the hardware constraints.

With the rapid application development aspects of the language, we can see the fruits of our labor as soon as we complete a module. A module is a small piece of a program that is able to be added in to the main program; think of it as a video game cartridge. Then, we can plug that module in to the rest of the program and see how it interacts within the environment. This can help us spot problems and resolve them very quickly with how it works, or to fix aesthetic problems with the interface.

The Final Product

On the Monday of finals week, we demonstrated our software to Gene Burton and other members of the University Police Department. It did not go perfectly, but it did give them a good idea of what we had created for them. It also allowed us to see their reactions and to have them test the software out to see how they took to it. They were impressed and pointed out a few errors that we needed to fix before we could call it final. On that Thursday, a completed version of the Simplified Reporting Software was given to Gene. He approved and allowed us to pass our Software Engineering series of classes.

Team COP – the Coalition of Programmers

Tony – Group Leader

I've known Tony since his freshman year living in Swinford Hall, and we had become good friends. The quality I look for in a leader is the knowledge of when to stop being your friend and to start being your boss. That is why we elected Tony the team leader. When he is our boss, he will be very direct with you. He will always tell you what is on his mind and will give you praise or criticism when you deserve it.

John – Lead Programmer

I had not met John until the beginning of the software engineering classes. We took him in because he knew Tim and we needed a fourth man for the project. The first semester, John's skills lay in designing the PowerPoint Slides and intro graphics (such as flying doughnuts) to our presentations. This semester, however, he really came through with the task of coding the main part of the program, and also designing and implementing the main interface.

Chris – Forms Designer

I started out being the in-guy by getting our group the project through one of my supervisors in University Computing Services. I also was the only person that had heard of the language we ended up using. I was later known as "The Brain" for having a knack for creating the diagrams and charts that we needed. The second semester, I used my knowledge of designing with Visual Studio to complete the online forms based on the original paper-design.

Tim – Testing

I've been in several classes with Tim in the past, though I did not get to know him until our software engineering class. Tim's talents were utilized toward the end of the process with the knack and the passion to break things. We needed those skills during the testing process to make sure that John and I had done everything correctly so that they match up to the requirements set up in the first semester. Also, he made sure that the code did not break under normal and abnormal conditions.

How Well We Worked Together

It seems like from the beginning of our team's creation, there has always been a division between us, or maybe it was just my perspective. I felt that since I knew Tony the longest and would always have venting sessions on the way home from class, there was us, and then John and Tim were on their own. Since I felt that the project itself centered

on the fact that it was separate people doing their own thing, and then finally everything would fit together at the end, any dissonance between people was not too much of a concern. However, even with our group being cliquey, there were never any ill feelings between any of the members of our group. We worked well together when we were together and got things done very efficiently.

Our Problems

Our main problem, and perhaps our only problem, was the fact that we did not communicate very much, and it only got worse between the first and second semesters. In the beginning we were in the same class and had regular team meetings, and that is because we had regular assignments that dealt with the whole team to get together and brainstorm. After all of those assignments were done, we no longer had any need for team meetings. Then in the second semester, three of us moved to another section of the class because of scheduling conflicts, and John had to remain in the original section because he had a conflict with the section that we moved to. The biggest problem with the separation of the team is that our lead programmer who was going to build the entire interface and backbone of the program was not in constant contact with the rest of the team. This caused some changes in the program that were not part of our original vision such as extra unneeded features.

How We Resolved Them

E-mail was a huge part of overcoming the communication gap, since one of our team members lives out of our local calling area. We would post updates through e-mail, and later when the program started getting too large for e-mail, post them on web pages so that they could be downloaded. Then whoever would be evaluating the program at the time would send their thoughts through e-mail, and be able to keep everyone up to date on what problems we were having, and the direction we needed to take the program.

Postmortem – What Went Right

Knowing Our Limitations

In the beginning, we were very ambitious while designing our project. We wanted to add network capabilities, expandability, the ability to import forms, as well as other extra features. We soon realized that many of those features would not be possible with the time we had to complete the project. It is a good thing that we caught ourselves early, or we may have been stuck scrambling to get some of those features implemented at the end.

Locking Our Requirements

One of the interesting things about our client is that they were not exactly sure what they needed, so the task of creating a requirements list fell on us. This was beneficial because we had a good idea what we could do in the amount of time that we had. We also knew how far we could go with the resources that we had available to use. That meant we could plan to be working until the very end and pack as many features as we could, or we could take it easy and do the bare minimum that we knew they would need and pace ourselves throughout the semester. We kind of took a middle road, because we are not slackers by any means, and we knew that anything can happen to throw a monkey wrench into our plans and put us behind schedule, which meant that we would be working on the project over the summer. And we did not want to do that at all.

Solid Engineering Tools

I went on and on before on how great the program is that we used, Microsoft Visual Studio .NET. Its tools are solid, and our team worked well with them. That's not to say that the program wasn't without its faults, but it is the best program that is out there for what we needed. Another program that we utilized is called COCOMO II; this program estimates the cost and effort that will go into creating your program through some insanely complicated calculations.

Postmortem – What Went Wrong

Client Level of Technological Knowledge

This presents a couple problems. First, we had to design the software by the lowest common denominator. We did not know the skill levels of all of the officers that would be using the software, but we imagined that there are many levels ranging from very proficient, to not being very computer literate at all. It was our original intent to keep the interface as familiar to them as possible by emulating the forms that they currently use. It was important for us to not cloud the interface with features but to just keep it as simple as possible. We did enable the software to perform many tasks in different ways so that each officer could use the method that he or she was most comfortable with. The second issue came when we came to install the software and demo it for them. We needed to help our audience with simple tasks as much as we were explaining the details of our software. We created thorough manuals for the department to alleviate some issues, since we tried to make the manuals as easy to follow as possible.

Communication

As I mentioned before, communication was an issue with the group being split up between different sections of the class, and the long distances between some of the members. This allowed the common vision of the project to go in different directions on some occasions. Our leader Tony helped keep the project on its tracks most of the time, but sometimes you can see slight variations in the design. Tony also handled most of the interactions with our client, Gene Burton. These manly occurred through e-mail, since Gene is a very busy man and could respond to them at his convenience.

Tool Limitations

Before I praised C# and Visual Studio .NET for being the best at what we needed for the project. It was, but not without some limitations. There were some points when I was designing forms, that it seemed that the developers of Visual Studio forgot to add a certain functionality or consistency within itself. For example, we created a certain look with the forms by using a specific style for the text boxes. The style was very

streamlined and compact, because we needed to fit a lot of boxes on a single page. In addition to normal text boxes, we needed lists to limit the input to a set of entries, which are called combo boxes. Combo boxes are similar to text boxes, but have an arrow on the right-hand side so that when you click on it, it drops down a list of choices. However, the style we used for the text boxes was not available for the combo boxes. The different style led to certain inconsistencies that were annoying at best.

We also used larger text boxes for narrative fields that were able to be many lines long, but if they typed past the end of the text box it did not show up on the printed form. Visual Studio allows you to assign a maximum character length to text boxes, but there is no maximum line function for large text boxes. Even though we set a maximum length, if they kept hitting return, which counts as one character, they could easily scroll past the end of the text box. The remedy for that problem was just to let them know not to type too much information, or all of their text would not show up on the printed document.

Time

In projects of this nature, time is never on our side. Some projects' deadlines can be pushed back in practice, but when dealing with a class, the deadline is the end of the second semester or you are not able to graduate on time. The last week of the project, we were feeling the mounting pressure to get it finished to propose it to our client. Even then, we showed them an unfinished product that required minor tweaking before we gave them a final disc.

I can not even imagine what would have happened in terms of deadlines if we had decided to keep in some of the functionality that we were toying around with at the beginning of the process. We possibly could have started programming earlier in the semester, even though that is frowned upon by our professor. Spring Break was also a week-long gap where I had no communication with the rest of my team, nor could I have worked on anything since I was in Florida during that week. It helped me regain some lost sanity, but it did not help me get any work done.

Conclusion

Software engineering is a long, meticulous journey through a system life-cycle. It is tedious in some places, and at the end it is very rewarding to be holding a product that you helped create from nothingness. I hope that this look at my team's project has helped you appreciate what goes into creating a software package. I have only scratched the surface as this project was simple compared to what you would use on a daily basis. The concepts, however, remain the same. Next time you start your favorite word processor or play your favorite computer game, think about all the detail and hard work that has gone into the creation of the program, from start to finish.

A Note on the Enclosed CD

The enclosed CD is a duplicate of the final CD that we gave to Gene Burton and the Ball State University Police Department. It includes the installation for the latest version of Internet Explorer (version 6), the latest version of the Microsoft .NET Framework (version 1.1), and SiRS release version 1.0. It also includes the four documentation files: the Operations Manual, which covers installation, backup, and troubleshooting; the User Manual, which outlines how to use the software; the Programming Manual, which includes all of the source code and its documentation; and the System Manual, which includes the design diagrams as well as the cases we used to test the software.

Works Cited

Liberty, Jesse. Programming C#, Second Edition. Sebastopol, CA: O'Reilly & Associates, Inc., 2002.

Pressman, Roger. Software Engineering – A Practitioner's Approach. New York, NY: McGraw-Hill Companies, Inc., 2001.

Wells, Don. Extreme Programming: A Gentle Introduction. 29 April 2003. <http://www.extremeprogramming.org>