

The failure of development models

Tonnerre Lombard

February 2nd, 2006

Abstract

There is a hell of a lot of development models out there. There's Domain Specific Modelling (DSM), Model Driven Software Development (MDSD), there are Design Patterns, there's the Rational Unified Process (RUP) and its lightweight pendant, the V-Model XT. And there's Extreme Programming (XP), which is the only model that is older than a couple of years.

This document attempts to analyze why Extreme Programming appears to be the only model that is capable of sustaining the market load. It's a journey into the depths of software development, and into the principles of the UNIX Operating System.

Chapter 1

The reality of software development

The reality of software development is sad. There are literally thousands of brilliant minds employed in the software industry. These people are hacking for a living, and they read a lot of books and computer magazines. In all these magazines they find a lot of interesting concepts, algorithms and other things. Their heads are full of brilliant ideas, which are however in no way related to the problems they're trying to solve.

1.1 A typical defeat

So what usually happens in 80% of the software development cases is the following:

- A customer goes to a corporation and asks them for a solution. The customer is only roughly aware of what he wants, so he can't give a detailed description. Nor does he care, as to him it's the companies' job to find out.
- An analyzer at the company drinks a lot of coffee in order to get his mega brain to an operating temperature. Then he tinkers a strategy to create an omnipotent product in the available timeframe, and allocates resources to it. He's thereby thinking far into the future and finds out what algorithms will be used in this project that will be useful for future projects.

- A brilliant programmer sits down and starts to implement the code he has read about in a recent issue of a computer magazine. It is in no way related to the program he's supposed to write, but it will be an extra function.
- The project is running out of time, while none of the requested functionalities are already finished. Now the programmers start to panic and do quick and dirty 4am hacks in order to get the project finished 2 monthes after the deadline.
- On the first customer presentation, the customer makes a combination of mouse clicks the programmers didn't think about, and the program crashes horribly. It takes 2 more weeks to find and correct a simple bug, because the code quality sucks.
- 2.5 monthes over the time, the customer finally recieves the end product and realizes that maybe he should have been a little more clear about his wishes, because the program doesn't correspond to them at all.
- Unused, the program ends up on backup tapes in the attic in order to have its source code reused. However, due to the hackish quality, nobody ever comes even as close to it as to be able to reach out for it with his arms.
- 8.5 monthes worth of software development go down the drain.

1.2 Error analysis

The problems we're facing are mainly caused by a class of misconceptions about the way software development works. The root cause is mainly that a group of software developers is lead in the vast majority of cases by a person who studied management and human resources management, and what he does is he attempts to apply the same rules to the developers that he would apply to a bunch of workers who are trying to build a house.

What most managers neglect to take care of is that there are severe differences between these different types of development. The first and most important point here is that, contrary to houses, it is very hard for a customer to imagine the end product, and consequentially it is very hard for him to figure out his specific requirements. It is easy for a customer to realize that the product does *not* meet his requirements, but it is very hard to figure out why. Thus, it is an absolute necessity to embrace the customer with his very

specific requirements during the process of creating the software, so there's no production overhead where you're producing stuff that the customer doesn't want, and consequentially doesn't pay for.

Another thing the models use to leave out is the mind of the developer. Most developers are playful beings, and most of the developers are human beings. As such, they have problems wrapping their minds around things that are larger than a couple of screen pages. Also, the quality of their products tends to depend heavily on their mood, and most of the time, they make mistakes and implement obscure hacks that they don't see. This is why simplicity and pair programming are so important.

1.3 Conclusions

There's a set of logical conclusions to be drawn from this set of problems. The philosophical evaluation of this is following in the next chapter, so I'm only going to deliver the main points here:

- **Talk to your customer.** If the customer has the chance to see progress and watch the project grow, his possibilities to determine what exactly he likes and what not are growing too. Also, customers who just want to get as many products as possible out of one contract don't get a chance, because they need to make very specific advises.
- **Endorse simplicity.** It is very hard to make things as simple as possible but no simpler. This is however a bare necessity of software engineering. The trick is to divide every problem in so many small entities that it becomes easy to distribute. These small entities can then be put together in the construction set manner in order to make up a whole, complex program. Attempts to design these entities to fit together however are mostly the direct way to oblivion, because the complexity tends to get out of control very quickly.
- **Keep your tests running.** Don't write code before you have a test which fails. Always put your customer's requirements into a test, which is then going to fail, because the method isn't implemented yet. Then do the most specific implementation you can possibly come up with, and once you pass the test, stop writing code, because no more code is needed, and noone is going to pay you for more code. However, keep running your tests! If interactions with other code break your requirements, you must notice that at once, and fix it, because holding in the requirements is what you get paid for.

- **Promote pair programming.** While one person working alone is using his time more efficiently, two people working together tend to have a higher code quality which gives a huge advantage in terms of time invested in maintenance. Another reason is that pair programming assures that if something happens to one of the two persons working at a problem, there's always the other person, who still has the expertise to solve it. This doesn't have to be an accident or whatever, it might even be the concurrence who employs one of your workers in order to sabotage your products.
- **Keep adjusting your resource plan.** It is impossible to tell beforehand how much work programmers will be able to take during the project, so with a pre-allocation plan, you will have a lot of gaps, overloads and people idling around. The best way there is probably to hand out a stack of tasks, and every team which has finished its operation just takes the next task off the stack. This provides for low-cost automated scheduling.

Chapter 2

The history of software development

Before I actually start, I should maybe remark that I'm hereby restricting my elaborations to software programming using software, that's why I'm not going into punch cards in all that depth.

In the very early years, software development was the privilege of students and professors in universities. They were mostly studying mathematics and physics and using the machines for their calculations. At this point, code was mostly fit onto punch cards and as such very hard to decipher. However, people who were capable of deciphering the code without extensive conscious translation also mostly had a good position to maintain the code, because usually this code was pretty short.

Finally, for the first time, there were ideas to create operating systems for computers, basically universal programs that would enable you to do a variety of things without having to implement it all yourself.

However, the more services your operating system implements for you, the more people probably want it to do. The MULTICS operating system had become the perfect example of this strategy: instead of going for simplicity, it was an attempt to integrate all the components and to design them to work together. The end result was a big mess that was very hard to maintain.

This led two students, Ken Thompson and Dennis Ritchie, to the conclusion that «Software architecture is crap». They consecutively came up with an operating system which only provided a general interface to hardware access and took care of time sharing, and which was supposed to provide perfect abstraction of processes. The implementation of the system call in-

terface enabled people to see processes as entirely separate entities, and to use them in order to stack these small entities together to big products.

Their operating system was a huge success, especially because it was the only operating system at the time which was capable of running and providing the same API under many different platforms which were available at the time - from the PDP to the Honeywell DDP.

At this time, the number of available operating systems remained constantly small. Other commercial OSes came and went, and the Bell Labs production of the Plan9 Operating System got stuck very quickly because the huge success of the simplicistic UNIX attracted all available resources. Only Berkeley's UNIX derivate, the Berkeley System Distribution, was undergoing constant development, because in addition to the inherent simplicism, it had an official liberty of development, the often cited flair of «Free Software».

A lot of other operating systems had been written in the time, but only a few of them had been able to make it over the first couple of years. One of them was the Virtual Memory System (VMS) for the PDP-successor VAX (named after its Virtual Address Extension), which was also a rather simple design from DEC, a company which has achieved immortal renown for their compromiseless designs as well as the stability of their hardware. The DEC enhanced VAX processor (ev, also known as alpha) was a compromiseless RISC processor, which is again a design that endorses simplicity. The alpha processor is unbeatable up to the present in terms of reliability and performance.

The other OS which was created at the time and which survived up to the present day was SunOS, today's Solaris. It developed from the UNIX System V, and is as such yet another example of simplicism. Over time, a lot of the BSD typical software went into Solaris, making it just another commercial BSD flavor, basically.

During the late 80s and early 90s, there was a different set of operating systems coming up which is now still influencing our life. One of them was the famous MacOS, an operating system which used to be rewritten several times between the releases. In the 10th release, it was replaced by a microkernel BSD however, so even in this case, the simplicism of the UNIX like operating systems prevailed.

Another system that came up was Linux. Now Linux is a very strange design: while vast parts of the userland are simply UNIX, and as such endorse the principle of simplicity, the two most important parts of it are massive pieces of code which are very hard to maintain: the kernel and the C library. While the kernel is just trying to follow the principle of design, and is thus very fragile and has a certain tendency to break, the libc does obey the

principles of simplicity to a certain extent: however, its capability to be used on a variety of system has taken its tolls, and thus it is very hard to find the function your bug is really in.

The last system to mention in this row is a candidate in the dying out row of designed operating systems: Windows. The operating system itself is a huge collection of components from products of other companies, which are glued together in some way or other. All that Microsoft does is write this glue code and kill off bugs in the code. However, in some cases, Microsoft isn't very efficient at that, so a bug in the BSD TCP handler, which was basically fixed long ago in BSD, hit Windows with all its force.

Windows is known for its instability, and amongst the developers, it is known to be a big, broken piece of code that is very hard to maintain. For that reason, there has already been an attempt to a complete rewrite in the year 1996, and right now there is another attempt taking place in the year 2005. However, Microsoft usually stirs the entire old codebase back in for compatibility reasons, so you mostly end up with all the old bugs plus some new ones. The current code base however is known for its bugs and needs one major security patch every couple of days, so it's pretty much clear that this already got out of control.

So what we basically see here is that only UNIX and VMS managed to make their way through the decades of computing, and while all the other systems had to keep restarting their efforts, the UNIX operating systems could just evolve with the power of millions of hands all over the world. While managers came up with a variety of different concepts for the software and how to create it, the only concept which prevailed was based on the perversion of all business software development method. Yet it sounded logical in the ears of the big companies, even though it was based on taking the rule «Software architecture sucks», which was extended in all possible ways in order to make an entire religion. Apple's MacOS X is the final proof that even companies swallow this message these days.

The same is going to happen in the software development model market. While the management is going to come up with one model after another, the only constantly deployed model is going to be Extreme Programming, and again it's the message «Software development models suck», which was taken and expanded to a set of rules that people should follow if they want to write good software. This way makes it look less anti-management. Nevertheless, it cannot be denied that this model was created by technically intelligent people: it is even heavily scalable at low overhead. It has all the typical features that UNIX has on the software market.

Chapter 3

Conclusions

The conclusions are pretty much given in the analysis already. All the different software development methods that managers are going to come up with aren't helping the situation in any way. The developer community has already found a good way of organization all by itself, and since they are the people with the technical expertise, the management should look really deeply and intimately at the proposed method.

When Ken Thompson and Dennis Ritchie created UNIX, they were laughed at by a lot of professors and managers, too. Simplicist approaches are mostly taken as a sign of weakness and give products a flair of being made for children. Nevertheless, simplicity is a very important step in development, especially in the software market, where it's not a necessity to stick small functional entities together for physical reasons.

Simplicity just *is* no sign of weakness. It is more of a confession of one of the basic qualities of the human mind. We're incapable of dealing with arbitrarily complex systems, and unless we acknowledge this fact, we're not going to be able to solve arbitrarily complex tasks.

Appendix A

Overview of modern development models

This section gives you a short overview over current development model and their specific creeds.

A.1 Domain Specific Modelling

Domain specific modelling is the development favored by Microsoft, which is a very successful software development company. Nevertheless, it is a commonly known fact that even with this development model, Microsoft hasn't been able to come up with proper software yet. However, let's have a look at it before we jump to conclusions.

The basic idea behind Domain Specific Modelling is that software development has an inherent complexity that can't be avoided in the implementation process. At the same time, programmers must be very productive, so what Domain Specific Modelling is trying to do is to enable the developers to cope with the complexity by kindof «optimizing» their communication with the experts of the area they're working in. Therefore, so-called domain specific languages are developed, which can be used to freely exchange information between software engineers and domain experts that have no idea of software engineering.

This model is assuming that the biggest problem of commercial software development is a misunderstanding between technicians and experts, and that this problem can be solved in a technical way. However, as reality shows, the development of such a domain specific language is an abstract process that would have to be refined for everything that any of the two

involved parties says. Roger Penrose wrote in his book «The Emperor's New Mind» that it would be impossible to reproduce the human brain entirely with a machine because, amongst other problems, quantum physics predicts that there would always be a possible action that the programming of the machine doesn't expect and thus can't cope with.

So while the development of a full domain specific language is impossible, the development of a partial such is utterly inefficient for the exact same reason: the developers need to create an engine that interpretes every step the expert has made, which is basically what his brain already does. Thus, it would be far easier to just take notes and evaluate what the expert says.

Another problem with this approach is that customers usually don't know what they want. Thus, what they said in the beginning of the process might not have a meaning in the end, or might get an entirely different meaning by what the customer amends later in the process. It would be impossible to foresee this.

A.2 Design Patterns

Design Patterns are a very simple idea. Usually, experts keep solutions for their problems in order to reuse them when a similar problem turns up. However, the idea of having special repositories for design patterns and all is highly overrated, because there are already implementations for this right in the good old software itself: shared libraries.

In that sense, the Internet is a huge collection of design patterns. There are a lot of shared libraries spread all across the planet, and it would be likely that someone has already solved your problem and has made such a pattern, or shared library. For Perl code for example, there is CPAN, where people can download Perl modules for pretty much everything.

One could of course argue that Design Patterns are a very abstract thing and could be converted into different programming languages, whereas shared libraries are basically specific to the programming language. The argument against this is that you don't need every programming language on this planet in order to solve your problem. In fact, there are far too many programming languages that don't give you any specific benefit over all the others, except to lock you out of a huge pool of existing solutions. On the other hand, there's no shame to be taken in restricting yourself to a set of three or four mainline programming languages, which will have implementations for most of your problems. That is much more useful than abstract design patterns.

A.3 Rational Unified Process

The basic assumption of the Rational Unified Process is that software development can be divided into 4 different stages that always follow consequentially and don't have interferences: the interception phase, where aims are being drafted and resources are allocated, the elaboration phase, where the problems are being analyzed, the construction phase, where a final product is being implemented and tested, and the transition phase, where the end user is evaluating the product and judging whether it is what he wants.

The problem with this model is that it is impossible to solve. Usually, the interception phase is when your customer tells you what you want, and the elaboration phase is where your thinking machines are evaluating what actions to take in doing so. Then, you enter the construction phase, you write your code and do exactly the job allocated to you in the interception phase.

There, you will already encounter most of the time that some people have a lot more and a lot more difficult problems to solve than others, because the resource allocation was done before the amount of work per person could be estimated. This is not a show-stopper, but it will cause unrest to your workers, because there will be the group of people who feels that the other group is too slow, and the group of people who feels that the other group got nothing to do. This again causes dissatisfaction.

Then you are going to enter the transition phase, where you are going to realize that the product you have just thrown onto the market is not at all what your customer wanted. Most of the time, the differences are serious enough that you must almost restart the project and redraft most of your stuff. This means that you have just created another piece of software that noone needs.

A good comparison of this situation is when you want to go from Lausanne to Lucerne. The RUP requires you to point to a direction where you consider Lucerne to be, and jump onto a train that leaves to that direction. Then, however, you must stay on that train until it reaches the final station, and only then are you allowed to ask where you actually are. This is however probably going to be St-Gall, which is miles away from your destination. Thus, you're going to point again, jump onto the next train, and if you're out of luck, it's going to take you back to Lausanne. If you're lucky, however, you got at least closer to Lucerne.

This is an efficient method to traverse the whole of Switzerland, but it's not efficient if you wanted to get to Lucerne as soon as possible.

(Also, an additional problem in the software development world is that,

as pointed out earlier, customers tend not to be sure about their wishes, so you may be traversing the whole of Switzerland trying to find a *moving* target, just by keeping pointing at trains you think might go to the right direction.)

Additionally, the RUP model is promoting the use of UML diagrams to embody your programs, which is entirely pointless. All that does is to give you an additional layer of complexity, which you need to get beyond anyway once you compile the program. Then, you need to look at the generated code and fix the bugs, which is what you would do anyway, except that in this case you have to get behind your own code, which is therefore wasted time. Also, UML diagrams aren't really any more readable than well-written C code. And if you're not an expert, you can't read either of them.

A.4 V-Model XT

V-Model XT is basically an extended form of RUP. It tries to avoid the unavoidable problems of RUP by solving symptomatic problems of the RUP process in establishing a self-learning organization that is supposed to efficiently learn how to avoid the problems it ran into during the RUP projects. The next time, however, the organization gets a new project, and runs into the same problems.

A.5 Extreme Programming

Extreme Programming (XP) is a software development model that was invented by computer geeks in the 80s. It is based on a number of basic assumptions:

1. **Customers don't know what they want.** Therefore, it is necessary to ask the customer for his current wish, and implement it with as little effort as possible. Then, you go back to the customer to ask him about his opinion regarding the little piece of software you have given him. This has the side effect that the customer can only get one specific product from you at a time, instead of saying, «This wasn't what we wanted, now we want you to do something else.»
2. **Programmers can only handle very limited complexity.** Therefore, every problem must be divided into its simplest entities, which can then be implemented separately. The implementation of these entities

can be distributed over the involved teams, while avoiding too many topic switches for teams.

3. **Programmers tend to overcomplicate things.** Therefor, it is necessary to write a test before you write the actual problem solution code. Once the code passes the test, the programmer must stop writing code.
4. **Programmers tend to break stuff.** Therefor, every solution to a problem entity must always have their test running in order to ensure that a change in some place doesn't have implications on other places. This should include buffer overflow tests.
5. **Programmers tend to produce unmaintainable code.** Therefor, it is necessary that there are always two programmers working at one piece of code. The two is a compromise between three problems: the diminishing problem is the waste of resources, and the other problems are the code quality, which increases because there is a second person who need to be able to read the code, as well as the distribution of knowledge: if something happens to one of the developers, the other one can still complete the project without any significant delay.
6. **You can't keep resources, time, quality and scope under control at the same time.** Therefor, you must reduce the least important factor: scope.

Appendix B

Contact

Tonnerre Lombard has developed software and various commercial and non-commercial operating systems for a number of years already.

Website: *<http://users.bsdprojects.net/~tonnerre/>*

Email: *tonnerre@bsdprojects.net*

Contents

1	The reality of software development	2
1.1	A typical defeat	2
1.2	Error analysis	3
1.3	Conclusions	4
2	The history of software development	6
3	Conclusions	9
A	Overview of modern development models	10
A.1	Domain Specific Modelling	10
A.2	Design Patterns	11
A.3	Rational Unified Process	12
A.4	V-Model XT	13
A.5	Extreme Programming	13
B	Contact	15
C	Table of Contents	16