Refactoring or Upfront design?

Pascal Van Cauwenberghe
Lesire Software Engineering
Maria Theresiastraat 99
3000 Leuven
Belgium
+32 16 299727
pvc@lesire.com

ABSTRACT
Among supporters and detractors of XP the debate rages whether upfront design or incremental design combined with refactoring are the optimal methods of implementing systems.

This paper argues that neither method is clearly better in every circumstance. Rather, the experienced software engineer will use a combination of both methods.

This paper argues that the "cost of change" curve presented in "Extreme Programming Explained"[1], does not replace the classic "cost of fixing errors" curve presented by Barry Boehm in [2]. Rather, XP is a method of attacking the costs described by this curve.

XP, as an incremental method of software engineering, is only applicable in circumstances where the cost of implementing functionalities does not grow rapidly as development progresses. Some heuristics and examples for deciding when to use each technique are presented.

Keywords
Cost of change, upfront design, software engineering economics

1 INTRODUCTION

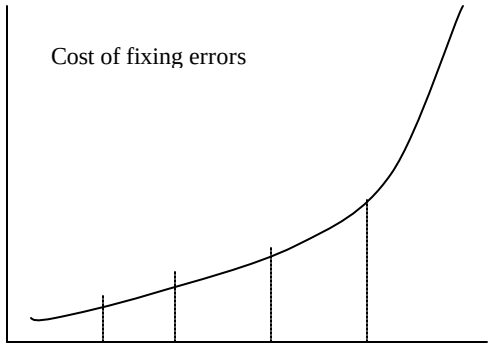
Boehm presented the classic cost curve shown below in "Software Engineering Economics". As we progress from analysis, through to design, coding, testing and production, the cost of fixing a problem rises. Note that the sharpest rise occurs when the system is released and distributed to its customers.

In "Extreme Programming Explained", Kent Beck argues this curve no longer represents the current state of software engineering. Rather, this curve is said to be flat. Two remarks can be made:

- Originally, this curve represented the cost of **fixing errors**, introduced in earlier phases of a project. Kent Beck presents the curve as the "cost of **change**" curve.
- In his online paper "Reexamining the cost of change curve"[3], Alistair Cockburn demonstrates the cost of **fixing errors** still rises rapidly as the project progresses.

2 DOES THIS CURVE INVALIDATE XP?

If this curve is still valid, does this mean XP is invalid? I



will argue it is not. Several of the XP practices specifically ensure that the costs associated with this curve are kept minimal:

- *Unit testing* and *test-first design* ensure that bugs are found quickly when they are cheap to fix
- *On-site customer* and *functional testing* ensure the analysis and specification of the system is precise and up-to-date with business requirements.
- *Pair programming* finds bugs quickly and spreads knowledge.
- *Refactoring* and *"once and only once"* ensures the system remains well-designed and easy to change.
- *Regular releases* gives regular customer feedback and forces the team to make the "release to production" and maintenance phases (where the cost of fixing errors rises dramatically) as cheap as possible.

Thus, XP attacks the roots of the high cost of fixing errors (with good specifications, good designs, good implementation and fast feedback). Furthermore, by using very short cycle times, the cost is never allowed to rise very high.

3 NOT SO EXTREME PROGRAMMING

Note that, with the exception of the very short cycle times and pair programming (which in XP replaces the inspections that are accepted in most methodologies), these practices are not very original nor extreme.

While errors are most costly to fix when found in later phases, each later phase is more likely to find errors in previous phases. This is because each phase produces a more concrete, more tangible, more testable output. Therefore, we need an iterative process that incorporates feedback to improve earlier work.

4 ITERATIVE VS INCREMENTAL

The well-known "waterfall" method is rarely used, even by those who claim (or are forced) to use it. Most development methods are *incremental and iterative*. What do those words mean?

Iterative = repeating the same task to improve its output.

Incremental = dividing a task into small tasks, which are completed one by one (sequentially or in parallel).

Let's see how different types of methods use iterations and increments.

Waterfall:

Analyze, design, code, integrate, test, done!

No iterations, no increments, no feedback, everything works the first time.

Classic RUP-like process:

Analyze until 70-80% done. Start the design phase, but keep updating the analysis with any feedback you receive. Design until 70-80% done. Start the coding phase, but keep improving with feedback.

And so on for the other phases.

This is an iterative process, feedback is used to improve the work done. The process is not incremental, except in the coding and integration phases, where some parts of the application may be delivered incrementally.

Incremental architecture-driven process:

Analyze the application until 70-80% done. Design the application so that the architecture and high risk elements are relatively complete. Define functional groups. The analysis and design are refined as the project progresses.

For each functional group, a detailed analysis, a detailed design, coding, integration, testing is done. This increment is handled like a small RUP-like project, with iterations to improve the output. When the functional group is finished it is delivered as an increment to the customer.

We have an iterative first step, which looks at the whole application. The application is then delivered incrementally, developing each increment using an iterative process.

XP process:

Gather an initial set of stories from the customers (high level analysis). Define a metaphor (high level analysis/design).

For each release: perform the planning game to allocate stories. For each story: define acceptance tests (analyze), write unit tests (design), code, refactor, integrate, test and repeat frequently (iterate) until done.

The basic process is incremental on the level of releases and stories. Within those increments, the process iterates rapidly, based on feedback from acceptance and unit tests.

5 FROM SHACK TO SKYSCRAPER

So what is extreme in XP? It is the assumption that analysis and design can be done incrementally; the assumption that *a complex system can be grown incrementally with hardly any upfront work*. XP detractors liken it to "building a skyscraper out of a shack".

This assumption is in no way trivial or obvious. Where this assumption does not hold, we will not be able to apply XP successfully.

Which XP practices depend on the incremental assumption?

- *The planning game* grows specifications story by story, expecting each story to deliver business value.
- *Simple design* solves today's problems, assuming that we will be able to solve tomorrow's problems when they arise.
- *Small releases* assumes we can deliver regular, **useful** increments of the system to the customers.
- *Customer in team* assumes we can define the specification of the system gradually, when we need to.

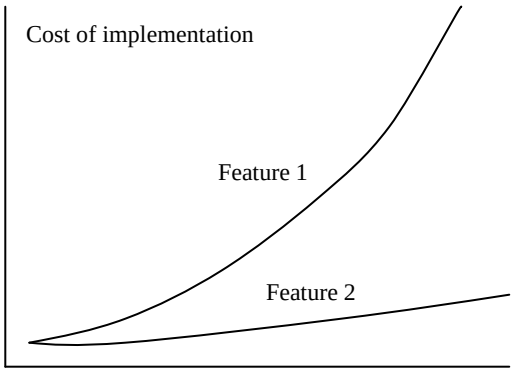
Working incrementally delivers some benefits:

- We learn all the time from the customer, from the system being developed. If we can make decisions later, they will likely be better.
- We keep the system as simple as possible, making it easier to understand, easier to change, less likely to contain errors.
- The customer quickly gets useful output. The system can be used to generate value and to guide further specification, planning and development.

6 PRECONDITIONS FOR INCREMENTAL METHODS TO WORK

Under what conditions do incremental methods work?

Let's examine how the cost of implementing (which includes analysis, design, coding, integration and testing) one feature changes over the duration of a whole software system.



One feature becomes quickly more expensive to implement, the other feature's price rises slowly.

In the first case we would be wise to spend effort as soon as possible, while the cost is low. We want to analyze and design this feature as completely as we can; we want to address not only our current needs but also our future needs. If we don't do it today, we will pay dearly for it later. A common cause for rising implementation costs is the breakdown of the design under the stress of new functions when the design is not kept up to date by refactoring.

In the second case, we can safely delay addressing the feature until we really need to. It might be somewhat more costly to design and implement later. For example, the system will have more functions and thus will probably be more complex. But we can invest the effort we have not spent on other, more profitable features.

It's in this situation the planning game brings large benefit to customers: they can select stories to implement based on their business value, without having to be concerned about technical dependencies and future costs.

7 SURPRISING COST CURVE

One of the surprising and pleasant effects that the incremental method can have is that the price of some functionalities decreases over time! The following can cause this:

- Well-designed (refactored), simple code where no duplication is allowed often presents the developer with opportunities to reuse significant parts of the code, which makes new features easier to implement.
- Over time we learn to better understand the problem domain, the design and the software. We see new abstractions, simpler ways to solve problems and better ways to apply our designs.

So, we find another heuristic to select the incremental method: use the incremental method when you expect to learn more, so that you can make better decisions later. This applies especially to situations where requirements are unclear or changing.

8 ANALOGY WITH INVESTMENT

If you want to invest in a company you can buy shares. You make the decision based on your knowledge of the market, the company, the risk you run and speculation about the future. Your money is tied up. If it turns out like you predicted you can gain a lot. If it doesn't, you lose money. That's the risk you take.

Sometimes you can buy stock options. These allow you to buy stock in the future at a price that is agreed now. You invest very little but you buy the possibility to wait to make your decision. If the value of the stocks rises, you buy and make a profit. If the value of the stocks decreases, you don't buy and lose only the price of the option. Likewise, investing in tests and refactoring is a small investment that pays off by giving you more options.

9 ANALOGY WITH HOUSE BUILDING

Often, software development is compared with more mature engineering disciplines. An analogy with building construction is sometimes used to demonstrate the value of good architectural design, detailed planning (as if construction projects always deliver on spec, on time) and a solid mathematical and scientific basis. Let's see how one would approach a house-building project under both cost-of-implementation assumptions. Imagine, an architect discusses the specifications for a house to be built for a couple. An important factor is the number of bedrooms to be built.

The couple must decide **now** how many bedrooms it will need in the foreseeable future. How many children will they have? Hard to predict. But they must decide now, because it will be very costly to add additional rooms to the house. They must invest now, their money is tied up in those rooms they may never need. If they underestimate the number of rooms needed they will be faced with costly modifications or will have to build a new house.

If, on the other hand, adding a room later costs not much more than building it now, the couple would be wise to postpone the decision until they really need extra rooms. In the mean time they can invest their money elsewhere. They don't face the risk of over- or underestimating the need for rooms and thus wasting money. They have lowered their financial risk considerably.

Maybe software and houses aren't the same [4].

10 TYPICAL EXAMPLES OF RAPIDLY RISING COST FEATURES

In some situations we are faced with features whose cost rises sharply. We should take this into account and expect to perform more work upfront. We should also try to minimize the cost, so that we can gain the benefits of the

incremental method.

- Externally used published APIs: once the APIs are in use, customers will demand backwards compatibility or a simple upgrade path. Effort should be spent on keeping the APIs flexible, minimal and useful.
- Development teams that aren't co-located. Up-front effort should be spent on partitioning the system to be developed.
- Databases used by multiple, independent applications. Common abstractions should be used to encapsulate persistence. When applications are released independently, the persistence and model layer should support some level of multiple version support.
- Software where release to customers or distribution is expensive. Frequent releases can train the development and production team to perform these tasks as efficiently as possible.
- Aspects of the application that have an overall effect on all of its parts. Examples are: internationalization, scalability, error handling… Having to make changes that affect all of the software makes refactoring very expensive.

Summary: interfaces between different teams, global properties of the system and software that is distributed to remote customers have a high cost of change.

Some areas that are commonly thought to have a high cost may have a surprisingly low cost of change:

- Except for hard time-critical software, well-factored code can be changed to meet performance criteria. A few simple and general design techniques can be used upfront. Most of the performance-related work can only be done after measurements have been made on the integrated system. A process that integrates and delivers often, combined with performance measurements is the most effective way of developing well-performing software.
- Database schema changes for software that is owned by one team. A team can get very good very quickly at dealing with schema or interface changes, if all of the software is owned by the team. Version-detection, upgrade programs and encapsulation of version-dependent modules allowed my team to make fundamental changes to the database structure, without any customer noticing it.

11 CONCLUSION

The choice between upfront work and refactoring is one to be made on a case-by-case basis. There is always some upfront work and some refactoring. It is up to the software engineer to make the right tradeoff, based on the following heuristics:

- If you can postpone decisions, you will be able to make better decisions at a later time.
- Invest in more upfront work if the implementation cost of the functionality is likely to rise rapidly in the future.
- Investing is dangerous unless you know the domain well and can make informed projections.

The choice comes down to selecting the method that implies the least risk. Good, experienced software engineers are able to make this choice. Instead of using the disparaging term "Big Design Up Front" (BDUF) we should be investigating how best to determine what is "Just Enough Design for Increments" (JEDI). This will allow us to make better-informed decisions.

Maybe software engineering should not look only to other engineering disciplines for analogies and techniques, but also to the way risk and return on investment are analyzed in the financial world.

12 INFORMATION AND QUES TIONS

For more information, contact **PVC@LESIRE.COM**

ACKNOWLEDGEMENTS

Thanks to Vera Peeters and Martine Verluyten for reviewing and discussing this paper.

REFERENCES

1. Beck, K. "**Extreme Programming Explained"** , p.21-23
2. Boehm, B, "**Software Engineering Economics**"
3. Cockburn, A, "**Reexamining the Cost of Change Curve**", on-line: http://www.xprogramming.com/xpmag/cost_of_change.htm
4. Jeffries, R. "**House Analogy**", online http://www.xprogramming.com/xpmag/houseAnalogy.htm