Chapter 18

Refactoring or Up-front Design?

—*Pascal Van Cauwenberghe*

Among supporters and detractors of XP, the debate rages whether up-front design or incremental design combined with refactoring is the optimal method of implementing systems. This chapter argues that neither method is clearly better in every circumstance. Instead, the experienced software engineer uses a combination of both methods. This chapter also argues that the "cost of change" curve presented in Extreme Programming Explained *[Beck1999] does not replace the classic "cost of fixing errors" curve presented by Barry Boehm in [Boehm1981]. Instead, 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.

Introduction

In *Software Engineering Economics*, Boehm presented the classic cost curve shown in Figure 18.1. As we progress from analysis to design, coding, testing and production, the cost of fixing a problem rises. Note

Copyright © 2003, Pascal Van Cauwenberghe. All rights reserved.

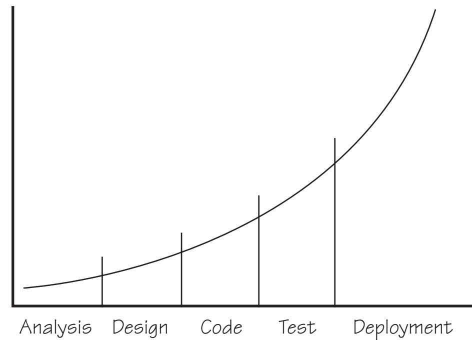


FIGURE 18.1 The "cost of fixing errors" curve

that the sharpest rise occurs when the system is released and distributed to its customers.

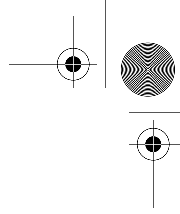
In *Extreme Programming Explained*, Kent Beck argues that this curve no longer represents the current kstate of software engineering. Instead, this curve is said to be flat. Two remarks can be made.

- 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 article "Reexamining the Cost of Change Curve," Alistair Cockburn demonstrates that the cost of *fixing errors* still rises rapidly as the project progresses [Cockburn2000].

Does This Curve Invalidate XP?

If this curve is still valid, does this mean XP is invalid? I argue that 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 that the analysis and specification of the system are precise and up to date with business requirements.



- *Pair programming* finds bugs quickly and spreads knowledge.
- *Refactoring* and *"once and only once"* ensure that 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 design, good implementation, and fast feedback). Furthermore, by using very short cycle times, the cost is never allowed to rise very high.

Not So Extreme Programming

Note that, with the exception of the very short cycle times and pair programming (which in XP replaces the inspections, design sessions, and training that are accepted in most methodologies), these practices are neither very original nor extreme.

Although 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.

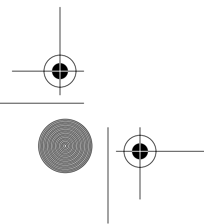
Iterative Versus Incremental

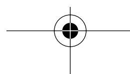
The well-known "waterfall" method is rarely used, even by those who claim (or are forced) to use it. Most development methods are *incremental* and *iterative*. What do those words mean?

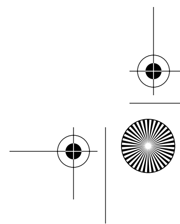
- Iterative—Repeating the same task to improve its output
- Incremental—Dividing a task into small tasks, which are completed one by one (sequentially or in parallel)

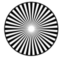
Au:
Edit okay to avoid hyphenating last word?

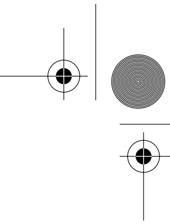
Now let's see how different types of methods use iterations and increments.











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. Refine the analysis and design as the project progresses.

For each functional group, a detailed analysis, a detailed design, coding, integration, and testing are done. Each 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.

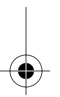
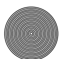
We have an iterative first step, which looks at the whole application. The application is then delivered incrementally, with each increment developed using an iterative process.

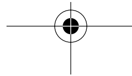
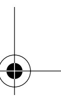
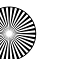
Extreme Programming

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

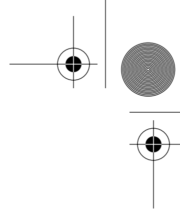
For each release, perform the planning game to allocate stories. For each story, define acceptance tests (analyze), write unit tests (design), code, refactor, integrate, test, and repeat frequently (iterate) until done.

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.

111_SUCCI.ch18.fm Page 195 Saturday, June 22, 2002 9:35 AM



From Shack to Skyscraper

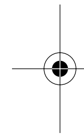
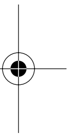
So what is extreme in XP? It is the assumption that analysis and design can be done incrementally; the assumption that *a complex system can be grown incrementally with hardly any up-front work*. XP detractors liken it to "building a skyscraper out of a shack."

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 that we can deliver regular, *useful* increments of the system to the customers.
- *Customer in team* assumes that we can refine the specification of the system gradually, when we need to.

Working incrementally delivers some benefits.

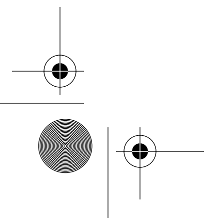
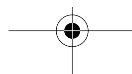
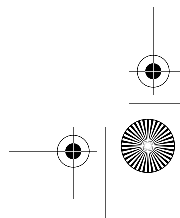
- We learn all the time from the customer, from the system being developed. If we can make decisions later, they will likely be better.
- We keep the system as simple as possible, making it easier to understand, easier to change, less likely to contain errors.
- The customer quickly gets useful output.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

Preconditions for Incremental Methods to Work

Under what conditions do incremental methods work? Let's examine how the cost of implementing one feature (which includes analysis, design, coding, integration, and testing) changes throughout the duration of a whole software system.

As shown in Figure 18.2, one feature quickly becomes more expensive to implement, while 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

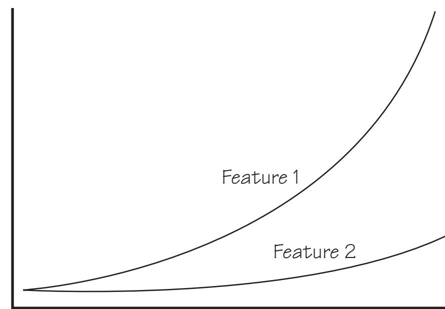


FIGURE 18.2 The cost of implementation

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.

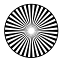
In the second case, we can safely delay addressing the feature until we really need to. It might be somewhat more costly to design and implement later. For example, the system will have more functions and thus will probably be more complex. But we can invest the unspent effort in other, more profitable features.

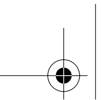
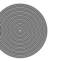
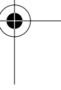
It's in this situation that the planning game brings a large benefit to customers: They can select stories to implement based on their business value, without having to be concerned about technical dependencies and future costs.

Surprising Cost Curve

One of the surprising and pleasant effects that the incremental method can have is that the cost of some functionalities decreases over time. The following factors can cause this.

- Well-designed (refactored), simple code in which no duplication is allowed often presents the developer with opportunities to reuse significant parts of the code, which makes new features easier to implement.



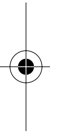
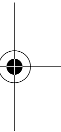




- Over time we learn to better understand the problem domain, the design, and the software. We see new abstractions, simpler ways to solve problems, and better ways to apply our designs.

So, we find another heuristic for selecting 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.

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 as you predicted, you can gain a lot. If it doesn't, you lose money. That's the risk you take.

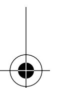
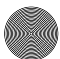



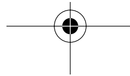
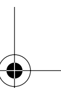
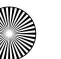
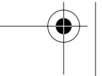
Sometimes you can buy options. These enable you to buy or sell stock in the future at a price that is agreed to now. You invest very little, but you buy the right to wait to make your decision. If the value of the stocks rises, you buy and make a profit. If the value of the stocks decreases, you don't buy and lose only the price of the option.

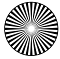
Likewise, investing in keeping your software malleable is a small investment that pays off by giving you more options. See Chapter 43, which is based on the papers presented by John Favaro (at XP2001) and Hakan Erdogmus (at XP Universe 2001), for a complete treatment of this analogy.

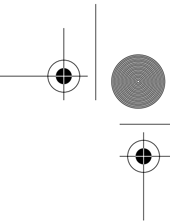
Analogy with House Building

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 that 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 they 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 more rooms to the house later. 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 they overestimate, they waste money.

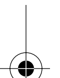
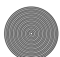
If, on the other hand, adding a room later cost not much more than building it now, the couple would be wise to postpone the decision until they really need extra rooms. In the meantime, they can invest their money elsewhere. They don't face the risk of over- or underestimating the need for rooms and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

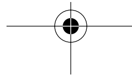
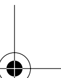
Maybe software and houses aren't the same [Jeffries2000].

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 up front. We should also try to minimize the cost so that we can gain the benefits of the incremental method. Here are some examples.

- Externally used published APIs. Once the APIs are in use, customers will demand backward compatibility or a simple upgrade path. Effort should be spent on keeping the APIs flexible, minimal, and useful.
- Development teams that aren't colocated. 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 in which 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, and so on. Having to make changes that affect all of the software makes refactoring very expensive.

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 up front. Most of the performance-related work can be done only 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 enabled my team to make fundamental changes to the database structure, without any customer noticing it.

Conclusion

The choice between up-front work and refactoring should be made case by case. There is always some up-front work and some refactoring. It is up to the software engineer to make the right trade-off, 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 up-front 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 enable us to make better-informed decisions.

Maybe software engineering should look not 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.

References

[Beck1999] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[Boehm1981] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[Cockburn2000] A. Cockburn. *Reexamining the Cost of Change Curve*. http://www.xprogramming.com/xpmag/cost_of_change.htm. 2000.

[Jeffries2000] R. Jeffries. *House Analogy*. http://www.xprogramming.com/xpmag/houseAnalogy.htm. 2000.

Acknowledgments

I would like to thank Vera Peeters and Martine Verluyten for reviewing and discussing this chapter.

About the Author

Pascal Van Cauwenberghe is the CTO of Lesire Software Engineering, where he leads the software development teams. He has been designing and building object-oriented systems for more than ten years.