Kari korpinen Ohjelmistotuotanto k 2012

Tehtävä 5 referenssi

Fowler Is design dead?

It seems that XP calls for the death of software design.In XP design techniques as the UML, flexible frameworks, and even patterns are de-emphasized or downright ignored. In fact XP involves a lot of design, but does it in a different way than established software processes. It also provides new challenges and skills as designers need to learn how to do a simple design, how to use refactoring to keep a design clean, and how to use patterns in an evolutionary style.

Extreme Programming (XP) challenges many of the common assumptions about software development.To its detractors this is a return to "code and fix" development - usually derided as hacking. To its fans it is often seen as a rejection of design techniques (such as the UML), principles and patterns. Don't worry about design, if you listen to your code a good design will appear.

Planned and Evolutionary Design

For this paper I'm going to describe two styles how design is done in software development. The most common is evolutionary design. Essentially evolutionary design means that the design of the system grows as the system is implemented. Design is part of the programming processes and as the program evolves the design changes.

In its common usage, evolutionary design is a disaster. The design ends up being the aggregation of a bunch of ad-hoc tactical decisions, each of which makes the code harder to alter.As Kent puts it, design is there to enable you to keep changing the software easily in the long term.You have the state of software entropy, over time the design gets worse and worse. Not only does this make the software harder to change, it also makes bugs both easier to breed and harder to find and safely kill. This is the "code and fix" nightmare, where the bugs become exponentially more expensive to fix as the project goes on.

Planned Design is a counter to this.So you begin with engineering drawings,partly by mathematical analysis, but mostly by using building codes. Building codes are rules about how you design structures based on experience of what works

Designers think out the big issues in advance. They don't need to write code because they aren't building the software, they are designing it. So they can use a design technique like the UML that gets away from some of the details of programming and allows the designers to work at a more abstract level.

Now the planned design approach has been around since the 70s, and lots of people have used it. It is better in many ways than code and fix evolutionary design. But it has some faults. The first fault is that it's impossible to think through all the issues that you need to deal with when you are programming.The programmers start coding around the design and entropy sets in.It takes time to sort out the design issues, change the drawings, and then alter the code. There's usually a quicker fix and time pressure.

Furthermore there's often a cultural problem. Designers are made designers due to skill and experience, but they are so busy working on designs they don't get much time to code any more. However the tools and

materials of software development change at a rapid rate. When you no longer code not just can you miss out on changes that occur with this technological flux, you also lose the respect of those who do code.

This tension between builders and designers happens more intense in software.Any programmer working in high design environments needs to be very skilled. Skilled enough to question the designer's designs, especially when the designer is less knowledgeable about the day to day realities of the development platform..

There's another problem: changing requirements. Changing requirements are the number one big issue that causes headaches in software projects that I run into.

One way to deal with changing requirements is to build flexibility into the design so that you can easily change it as the requirements change.

Now some of these requirements problems are due to not understanding requirements clearly enough.

So all this makes planned design sound impossible. Certainly they are big challenges. But I'm not inclined to claim that planned design is worse than evolutionary design as it is most commonly practiced in a "code and fix" manner. Indeed I prefer planned design to "code and fix". However I'm aware of the problems of planned design and am seeking a new direction.

The Enabling Practices of XP

The change curve says that as the project runs, it becomes exponentially more expensive to make changes. The change curve is usually expressed in terms of phases "a change made in analysis for $1 would cost thousands to fix in production".

The fundamental assumption underlying XP is that it is possible to flatten the change curve enough to make evolutionary design work.

There are many parts to the enabling practices. At the core are the practices of Testing, and Continuous Integration. Without the safety provided by testing the rest of XP would be impossible. Continuous Integration is necessary to keep the team in sync.Together these practices can have a big effect on the change curve.

Refactoring has a similar effect. People who refactor their code in the disciplined manner suggested by XP find a significant difference in their effectiveness compared to doing looser, more ad-hoc restructuring.

Jim Highsmith, in his excellent summary of XP, uses the analogy of a set of scales. In one tray is planned design, the other is refactoring. In more traditional approaches planned design dominates because the assumption is that you can't change your mind later. As the cost of change lowers then you can do more of your design later as refactoring. Planned design does not go away completely, but there is now a balance of two design approaches to work with. For me it feels like that before refactoring I was doing all my design one-handed.

These enabling practices of continuous integration, testing, and refactoring, provide a new environment that makes evolutionary design plausible.I'm sure that, despite the outside impression, XP isn't just test, code, and refactor. There is room for designing before coding. Some of this is before there is any coding, most of it occurs

in the iterations before coding for a particular task. But there is a new balance between up-front design and refactoring.

The Value of Simplicity

Two of the greatest rallying cries in XP are the slogans "Do the Simplest Thing that Could Possibly Work" and "You Aren't Going to Need It" (known as YAGNI).

The way YAGNI is usually described, it says that you shouldn't add any code today which will only be used by feature that is needed tomorrow.

XP's advice is that you not build flexible components and frameworks for the first case that needs that functionality. Let these structures grow as they are needed. If I want a Money class today that handles addition but not multiplication then I build only addition into the Money class. Even if I'm sure I'll need multiplication in the next iteration, and understand how to do it easily, and think it'll be really quick to do, I'll still leave it till that next iteration.

One reason for this is economic.This economic disincentive is compounded by the chance that we may not get it right.Working on the wrong solution early is even more wasteful than working on the right solution early. And the XPerts generally believe that we are much more likely to be wrong than right (and I agree with that sentiment.)

The second reason for simple design is that a complex design is more difficult to understand than a simple design.However when the balance between planned and evolutionary design alters, then YAGNI becomes good practice (and only then).

So to summarize. You don't want to spend effort adding new capability that won't be needed until a future iteration. And even if the cost is zero, you still don't want to add it because it increases the cost of modification even if it costs nothing to put in.

What on Earth is Simplicity Anyway

So we want our code to be as simple as possible. "What is simple?"

In XPE Kent gives four criteria for a simple system. In order (most important first):

- Runs all the Tests
- Reveals all the intention
-No duplication
- Fewest number of classes or methods

XP places a high value on code that is easily read. In XP "clever code" is a term of abuse. But some people's intention revealing code is another's cleverness.

JUnit uses decorators to add optional functionality to test cases, such things as concurrency synchronization and batch set up code. By separating out this code into decorators it allows the general code to be clearer than it otherwise would be.

I think that the focus on eliminating duplication, both with XP's "Once and Only Once" and the Pragmatic Programmer's DRY (Don't Repeat Yourself) is one of those obvious and wonderfully powerful pieces of good advice.

"it's easier to refactor over-design than it is to refactor no design."

The best advice I heard on all this came from Uncle Bob (Robert Martin). His advice was not to get too hung up about what the simplest design is. After all you can, should, and will refactor it later. In the end the willingness to refactor is much more important than knowing what the simplest thing is right away.

Does Refactoring Violate YAGNI?

Basically the question starts with the point that refactoring takes time but does not add function. Since the point of YAGNI is that you are supposed to design for the present not for the future, is this a violation?

The point of YAGNI is that you don't add complexity that isn't needed for the current stories. This is part of the practice of simple design. Refactoring is needed to keep the design as simple as you can, so you should refactor whenever you realize you can make things simpler.

Simple design both exploits XP practices and is also an enabling practice. Only if you have testing, continuous integration, and refactoring can you practice simple design effectively. But at the same time keeping the design simple is essential to keeping the change curve flat.

Patterns and XP

The relationship between patterns and XP is interesting, and it's a common question.But it's worth bearing in mind that for many people patterns seem in conflict to XP.

The essence of this argument is that patterns are often over-used. The world is full of the legendary programmer, fresh off his first reading of GOF who includes sixteen patterns in 32 lines of code.The question is how you use them.

One theory of this is that the forces of simple design will lead you into the patterns. Many refactorings do this explicitly, but even without them by following the rules of simple design you will come up with the patterns even if you don't know them already.For me effective design argues that we need to know the price of a pattern is worth paying.

But reading some of the mailing lists I get the distinct sense that many people see XP as discouraging patterns, despite the irony that most of the proponents of XP were leaders of the patterns movement too. Is this because they have seen beyond patterns, or because patterns are so embedded in their thinking that they no longer realize it?

My advice to XPers using patterns would be

- Invest time in learning about patterns
-Concentrate on when to apply the pattern (not too early)
- Concentrate on how to implement the pattern in its simplest form first, then add complexity later.

- If you put a pattern in, and later realize that it isn't pulling its weight - don't be afraid to take it out again.

Growing an Architecture

What role does an architecture play when you are using evolutionary design? Again XPs critics state that XP ignores architecture, that XP's route is to go to code fast and trust that refactoring that will solve all design issues. Don't put in a database until you really know you'll need it. Work with files first and refactor the database in during a later iteration.

early architectural decisions aren't expected to be set in stone, or rather the team knows that they may err in their early decisions, and should have the courage to fix them. Others have told the story of one project that, close to deployment, decided it didn't need EJB anymore and removed it from their system. It was a sizeable refactoring, it was done late, but the enabling practices made it not just possible, but worthwhile.

How would this have worked the other way round. If you decided not to use EJB, would it be harder to add it later?Certainly working without a complex component increases simplicity and makes things go faster.

So my advice is to begin by assessing what the likely architecture is. If you see a large amount of data with multiple users, go ahead and use a database from day 1. If you see complex business logic, put in a domain model.Also be ready to simplify your architecture as soon as you see that part of the architecture isn't adding anything.

UML and XP

some people find software diagrams helpful and some people don't. The danger is that those who do think that those who don't should do and vice-versa. Instead we should just accept that some people will use diagrams and some won't.

The other issue is that software diagrams tend to get associated with a heavyweight process. Such processes spend a lot of time drawing diagrams that don't help and can actually cause harm.

So here's my advice for using diagrams well.

First keep in mind what you're drawing the diagrams for. The primary value is communication.Don't draw every class - only the important ones. For each class, don't show every attribute and operation - only the important ones. Don't draw sequence diagrams for all use cases and scenarios - only... you get the picture.

A common use of diagrams is to explore a design before you start coding it.Many people say that when you have a sticky task it's worth getting together to have a quick design session first. However when you do such sessions:

- keep them short
-don't try to address all the details (just the important ones)
- treat the resulting design as a sketch, not as a final design

Changing the design doesn't necessarily mean changing the diagrams. It's perfectly reasonable to draw diagrams that help you understand the design and then throw the diagrams away. Drawing them helped, and that is enough to make them worthwhile. They don't have to become permanent artifacts. The best UML diagrams are not artifacts.

A lot of XPers use CRC cards. That's not in conflict with UML. I use a mix of CRC and UML all the time, using whichever technique is most useful for the job at hand.

Another use of UML diagrams is on-going documentation.The idea is that keeping this documentation helps people work on the system. In practice it often doesn't help at all.

- it takes too long to keep the diagrams up to date, so they fall out of sync with the code
- they are hidden in a CASE tool or a thick binder, so nobody looks at them

So the advice for on-going documentation runs from these observed problems:

- Only use diagrams that you can keep up to date without noticeable pain
- Put the diagrams where everyone can easily see them. I like to post them on a wall. Encourage people to edit the wall copy with a pen for simple changes.
- Pay attention to whether people are using them, if not throw them away.

On Metaphor

The XP practice of Metaphor is built on Ward Cunninghams's approach of a system of names. The point is that you come up with a well known set of names that acts as a vocabulary to talk about the domain. This system of names plays into the way you name the classes and methods in the system

Often people criticize XP on the basis that you do need at least some outline design of a system. XPers often respond with the answer "that's the metaphor".

Do you wanna be an Architect when you grow up?

In software, the term architect means many things. (In software any term means many things.) In general, however it conveys a certain gravitas, as in "I'm not just a mere programmer - I'm an architect". This may translate into "I'm an architect now - I'm too important to do any programming".

This question generates an enormous amount of emotion. I've seen people get very angry at the thought that they don't have a role any more as architects. "There is no place in XP for experienced architects" is often the cry I hear.

Much as in the role of design itself, I don't think it's the case that XP does not value experience or good design skills.However it does mean that their role changes from what a lot of people see as a role of technical leadership.

As an example, I'll cite one of our technical leaders at ThoughtWorks: Dave Rice. Dave has been through a few life-cycles and has assumed the unofficial mantle of technical lead on a fifty person project. His role as leader means spending a lot of time with all the programmers. He'll work with a programmer when they need help, he

looks around to see who needs help. A significant sign is where he sits. As a long term ThoughtWorker, he could pretty well have any office he liked. He shared one for a while with Cara, the release manager. However in the last few months he moved out into the open bays where the programmers work (using the open "war room" style that XP favors.) This is important to him because this way he sees what's going on, and is available to lend a hand wherever it's needed.

is that it calls the leading technical figure the "Coach". The meaning is clear: in XP technical leadership is shown by teaching the programmers and helping them make decisions. It's one that requires good people skills as well as good technical skills. Jack Bolles at XP 2000 commented that there is little room now for the lone master. Collaboration and teaching are keys to success.

At a conference dinner, Dave and I talked with a vocal opponent of XP.We all liked adaptive, iterative development. Testing was important. So we were puzzled at the vehemence of his opposition. Then came his statement, along the lines of "the last thing I want is my programmers refactoring and monkeying around with the design". Now all was clear. The conceptual gulf was further explicated by Dave saying to me afterwards "if he doesn't trust his programmers why does he hire them?". In XP the most important thing the experienced developer can do is pass on as many skills as he can to the more junior developers. Instead of an architect who makes all the important decisions, you have a coach that teaches developers to make important decisions.

Reversibility

At XP 2002 Enrico Zaninotto gave a fascinating talk that discussed the tie-ins between agile methods and lean manufacturing.

In this view one of the main source of complexity is the irreversibility of decisions. If you can easily change your decisions, this means it's less important to get them right - which makes your life much simpler.Rather than trying to get the right decision now, look for a way to either put off the decision until later (when you'll have more information) or make the decision in such a way that you'll be able to reverse it later on without too much difficulty.

This determination to support reversibility is one of the reasons that agile methods put a lot of emphases on source code control systems.

Designing for reversibility also implies a process that makes errors show up quickly. One of the values of iterative development is that the rapid iterations allow customers to see a system as it grows, and if a mistake is made in requirements it can be spotted and fixed before the cost of fixing becomes prohibitive. This same rapid spotting is also important for design.

The Will to Design

In order to work, evolutionary design needs a force that drives it to converge. This force can only come from people - somebody on the team has to have the determination to ensure that the design quality stays high.

This will does not have to come from everyone (although it's nice if it does), usually just one or two people on the team take on the responsibility of keeping the design whole.

This responsibility means keeping a constant eye on the code base, looking to see if any areas of it are getting messy, and then taking rapid action to correct the problem before it gets out of control.

A lack of will to design seems to be a major reason why evolutionary design can fail.

Things that are difficult to refactor in

Can we use refactoring to deal with all design decisions, or are there some issues that are so pervasive that they are difficult to add in later? At the moment, the XP orthodoxy is that all things are easy to add when you need them, so YAGNI always applies. I wonder if there are exceptions. A good example of something that is controversial to add later is internationalization.

Part of the justification of YAGNI is that many of these potential needs end up not being needed, or at least not in the way you'd expect.

Another issue to bear in mind in this is whether you really know how to do it. If you've done internationalization several times, then you'll know the patterns you need to employ. As such you're more likely to get it right. Adding anticipatory structures is probably better if you're in that position, than if you're new to the problem. So my advice would be that if you do know how to do it, you're in a position to judge the costs of doing it now to doing it later. However if you've not done it before, not just are you not able to assess the costs well enough, you're also less likely to do it well. In which case you should add it later. If you do add it then, and find it painful, you'll probably be better off than you would have been had you added it early. Your team is more experienced, you know the domain better, and you understand the requirements better. Often in this position you look back at how easy it would have been with 20/20 hindsight. It may have been much harder to add it earlier than you think.

This also ties into the question about the ordering of stories.Kent is in favor of letting business value be the only factor in driving the ordering of the stories.I believe it is a balance between business value and technical risk. This would drive me to provide at least some internationalization early to mitigate this risk. However this is only true if internationalization was needed for the first release. Getting to a release as fast as possible is vitally important. Any additional complexity is worth doing after that first release if it isn't needed for the first release. The power of shipped, running code is enormous. It focuses customer attention, grows credibility, and is a massive source of learning. Do everything you can to bring that date closer. Even if it is more effort to add something after the first release, it is better to release sooner.

Is Design Happening?

The danger of intermingling design with programming is that programming can happen without design - this is the situation where Evolutionary Design diverges and fails.

If you're in the development team, then you sense whether design is happening by the quality of the code base. If the code base is getting more complex and difficult to work with, there isn't enough design getting done.

If this lack of visibility is hard for technical people, it's far more alarming for non-technical members of a team. If you're a manager or customer how can you tell if the software is well designed? It matters to you because poorly designed software will be more expensive to modify in the future.

Listen to the technical people. If they are complaining about the difficulty of making changes, then take such complaints seriously and give them time to fix things.

Keep an eye on how much code is being thrown away. A project that does healthy refactoring will be steadily deleting bad code. If nothing's getting deleted then it's almost certainly a sign that there isn't enough refactoring going on - which will lead to design degradation.

So is Design Dead?

XP design looks for the following skills

- A constant desire to keep code as clear and simple as possible
- Refactoring skills so you can confidently make improvements whenever you see the need.
- A good knowledge of patterns: not just the solutions but also appreciating when to use them and how to evolve into them.
- Designing with an eye to future changes, knowing that decisions taken now will have to be changed in the future.
- Knowing how to communicate the design to the people who need to understand it, using code, diagrams and above all: conversation.