Home Blog Articles Books About Me Contact Me ThoughtWorks

Is Design Dead?

Martin Fowler
Chief Scientist, ThoughtWorks

Last Significant Update: May 2004

For many that come briefly into contact with Extreme Programming, it seems that XP calls for the death of software design. Not just is much design activity ridiculed as "Big Up Front Design", but such 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. XP has rejuvenated the notion of evolutionary design with practices that allow evolution to become a viable design strategy. 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.

| Japanese | Russian | Chinese | Spanish | Korean |

(This paper was written for my keynote at XP 2000 conference and it's original form was published as part of the proceedings.)

Extreme Programming (XP) challenges many of the common assumptions about software development. Of these one of the most controversial is its rejection of significant effort in up-front design, in favor of a more evolutionary approach. 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.

I find myself at the center of this argument. Much of my career has involved graphical design languages - the Unified Modeling Language (UML) and its forerunners - and in patterns. Indeed I've written books on both the UML and patterns. Does my embrace of XP mean I recant all of what I've written on these subjects, cleansing my mind of all such counter-revolutionary notions?

Well I'm not going to expect that I can leave you dangling on the hook of dramatic tension. The short answer is no. The long answer is the rest of this paper.

Planned and Evolutionary Design

For this paper I'm going to describe two styles how design is done in software development. Perhaps 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. In many ways you might argue this is no design, certainly it usually leads to a poor design. As Kent puts it, design is there to enable you to keep changing the software easily in the long term. As design deteriorates, so does your ability to make changes effectively. 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, and contains a notion born from other branches of engineering. If you want to build a doghouse, you can just get some wood together and get a rough shape. However if you want to build a skyscraper, you can't work that way - it'll just collapse before you even get half way up. So you begin with engineering drawings, done in an engineering office like the one my wife works at in downtown Boston. As she does the design she figures out all the issues, 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 (and some underlying math). Once the design is done, then her engineering company can hand the design off to another company that builds it.

Planned design in software should work the same way. 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. Once the design is done they can hand it off to a separate group (or even a separate company) to build. Since the designers are thinking on a larger scale, they can avoid the series of tactical decisions that lead to software entropy. The programmers can follow the direction of the design and, providing they follow the design, have a well built system

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. So it's inevitable that when programming you will find things that question the design. However if the designers are done, moved onto another project, what happens? The programmers start coding around the design and entropy sets in. Even if the designer isn't gone, 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. Hence entropy (again).

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 in building too, but it's more intense in software. It's intense because there is a key difference. In building there is a clearer division in skills between those who design and those who build, but in software that's less the case. 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.

Now these issues could be fixed. Maybe we can deal with the human tension. Maybe we can get designers skillful enough to deal with most issues and have a process disciplined enough to change the drawings. There's still 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. However this requires insight into what kind of changes you expect. A design can be planned to deal with areas of volatility, but while that will help for foreseen requirements changes, it won't help (and can hurt) for unforeseen changes. So you have to understand the requirements well enough to separate the volatile areas, and my observation is that this is very hard.

Now some of these requirements problems are due to not understanding requirements clearly enough. So a lot of people focus on requirements engineering processes to get better requirements in the hope that this will prevent the need to change the design later on. But even this direction is one that may not lead to a cure. Many unforeseen requirements changes occur due to changes in the business. Those can't be prevented, however careful your requirements engineering process.

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

XP is controversial for many reasons, but one of the key red flags in XP is that it advocates evolutionary design rather than planned design. As we know, evolutionary design can't possibly work due to ad hoc design decisions and software entropy.

At the core of understanding this argument is the software change curve. 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". This is ironic as most projects still work in an ad-hoc process that doesn't have an analysis phase, but the exponentiation is still there. The exponential change curve means that evolutionary design cannot possibly work. It also conveys why planned design must be done carefully because any mistakes in planned design face the same exponentiation.

The fundamental assumption underlying XP is that it is possible to flatten the change curve enough to make evolutionary design work. This flattening is both enabled by XP and exploited by XP. This is part of the coupling of the XP practices: specifically you can't do those parts of XP that exploit the flattened curve without doing those things that enable the flattening. This is a common source of the controversy over XP. Many people criticize the exploitation without understanding the enabling. Often the criticisms stem from critics' own experience where they didn't do the enabling practices that allow the exploiting practices to work. As a result they got burned and when they see XP they remember the fire.

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, so that you can make a change and not be worried about integrating it with other people. Together these practices can have a big effect on the change curve. I was reminded of this again here at ThoughtWorks. Introducing testing and continuous integration had a marked improvement on the development effort. Certainly enough to seriously question the XP assertion that you need all the practices to get a big improvement.

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. That was certainly my experience once Kent had taught me to refactor properly. After all, only such a strong change would have motivated me to write a whole book about it.

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. However one thing we haven't yet figured out is where the balance point is. 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). Both are manifestations of the XP practice of Simple Design.

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. On the face of it this sounds simple. The issue comes with such things as frameworks, reusable components, and flexible design. Such things are complicated to build. You pay an extra up-front cost to build them, in the expectation that you will gain back that cost later. This idea of building flexibility up-front is seen as a key part of effective software design.

However 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. If I have to do any work that's only used for a feature that's needed tomorrow, that means I lose effort from features that need to be done for this iteration. The release plan says what needs to be worked on now, working on other things in the future is contrary to the developers agreement with the customer. There is a risk that this iteration's stories might not get done. Even if this iteration's stories are not at risk it's up to the customer to decide what extra work should be done - and that might still not involve multiplication.

This economic disincentive is compounded by the chance that we may not get it right. However certain we may be about how this function works, we can still get it wrong - especially since we don't have detailed requirements yet. 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. Therefore any modification of the system is made harder by added complexity. This adds a cost during the period between when the more complicated design was added and when it was needed.

Now this advice strikes a lot of people as nonsense, and they are right to think that. Right providing that you imagine the usual development world where the enabling practices of XP aren't in place. 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 it because it increases the cost of modification even if it costs nothing to put in. However you can only sensibly behave this way when you are using XP, or a similar technique that lowers the cost of change.

What on Earth is Simplicity Anyway

So we want our code to be as simple as possible. That doesn't sound like that's too hard to argue for, after all who wants to be complicated? But of course this begs the question "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

Running all the tests is a pretty simple criterion. No duplication is also pretty straightforward, although a lot of developers need guidance on how to achieve it. The tricky one has to do with revealing the intention. What exactly does that mean?

The basic value here is clarity of code. 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.

In his XP 2000 paper, Josh Kerievsky points out a good example of this. He looks at possibly the most public XP code of all - JUnit. 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.

But you have to ask yourself if the resulting code is really simple. For me it is, but then I'm familiar with the Decorator pattern. But for many that aren't it's quite complicated. Similarly JUnit uses pluggable methods which I've noticed most people initially find anything but clear. So might we conclude that JUnit's design is simpler for experienced designers but more complicated for less experienced people?

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. Just following that alone can take you a long way. But it isn't everything, and simplicity is still a complicated thing to find.

Recently I was involved in doing something that may well be over-designed. It got refactored and some of the flexibility was removed. But as one of the developers said "it's easier to refactor over-design than it is to refactor no design." It's best to be a little simpler than you need to be, but it isn't a disaster to be a little more complex.

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?

This topic came up on the XP mailing list recently, and it's worth bringing out as we look at the role of design in XP.

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. Any unneeded complexity makes a system harder to change in all directions except the one you anticipate with the complex flexibility you put in. However people aren't good at anticipating, so it's best to strive for simplicity. However people won't get the simplest thing first time, so you need to refactor in order get closer to the goal.

Patterns and XP

The JUnit example leads me inevitably into bringing up patterns. The relationship between patterns and XP is interesting, and it's a common question. Joshua Kerievsky argues that patterns are under-emphasized in XP and he makes the argument eloquently, so I don't want to repeat that. 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. I remember one evening, fueled by a very nice single malt, running through with Kent a paper to be called "Not Design Patterns: 23 cheap tricks" We were thinking of such things as use an if statement rather than a strategy. The joke had a point, patterns are often overused, but that doesn't make them a bad idea. 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. This may be true, but is it really the best way of doing it? Surely it's better if you know roughly where you're going and have a book that can help you through the issues instead of having to invent it all yourself. I certainly still reach for GOF whenever I feel a pattern coming on. For me effective design argues that we need to know the price of a pattern is worth paying - that's its own skill. Similarly, as Joshua suggests, we need to be more familiar about how to ease into a pattern gradually. In this regard XP treats the way we use patterns differently to the way some people use them, but certainly doesn't remove their value.

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? I don't know the answers for others, but for me patterns are still vitally important. XP may be a process for development, but patterns are a backbone of design knowledge, knowledge that is valuable whatever your process may be. Different processes may use patterns in different ways. XP emphasizes both not using a pattern until it's needed and evolving your way into a pattern via a simple implementation. But patterns are still a key piece of knowledge to acquire.

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.

I think XP should emphasize learning about patterns more. I'm not sure how I would fit that into

XP's practices, but I'm sure Kent can come up with a way.

Growing an Architecture

What do we mean by a software architecture? To me the term architecture conveys a notion of the core elements of the system, the pieces that are difficult to change. A foundation on which the rest must be built.

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. Interestingly they are right, and that may well be weakness. Certainly the most aggressive XPers - Kent Beck, Ron Jeffries, and Bob Martin - are putting more and more energy into avoiding any up front architectural design. 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.

I'm known for being a cowardly XPer, and as such I have to disagree. I think there is a role for a broad starting point architecture. Such things as stating early on how to layer the application, how you'll interact with the database (if you need one), what approach to use to handle the web server.

Essentially I think many of these areas are patterns that we've learned over the years. As your knowledge of patterns grows, you should have a reasonable first take at how to use them. However the key difference is that these 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? Should you thus never start with EJB until you have tried things without and found it lacking? That's a question that involves many factors. Certainly working without a complex component increases simplicity and makes things go faster. However sometimes it's easier to rip out something like that than it is to put it in.

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. However in deference to the gods of YAGNI, when in doubt err on the side of simplicity. Also be ready to simplify your architecture as soon as you see that part of the architecture isn't adding anything.

UML and XP

Of all the questions I get about my involvement with XP one of the biggest revolves around my association with the UML. Aren't the two incompatible?

There are a number of points of incompatibility. Certainly XP de-emphasizes diagrams to a great extent. Although the official position is along the lines of "use them if they are useful", there is a strong subtext of "real XPers don't do diagrams". This is reinforced by the fact that people like Kent aren't at all comfortable with diagrams, indeed I've never seen Kent voluntarily draw a software diagram in any fixed notation

I think the issue comes from two separate causes. One is the fact that 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 I think that people should be advised how to use diagrams well and avoid the traps, rather than the "only if you must (wimp)" message that usually comes out of the XPerts.

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. Effective communication means selecting important things and neglecting the less important. This selectivity is the key to using the UML well. 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 problem with the common use of diagrams is that people try to make them comprehensive. The code is the best source of comprehensive information, as the code is the easiest thing to keep in sync with the code. For diagrams comprehensiveness is the enemy of comprehensibility.

A common use of diagrams is to explore a design before you start coding it. Often you get the impression that such activity is illegal in XP, but that's not true. 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

The last point is worth expanding. When you do some up-front design, you'll inevitably find that some aspects of the design are wrong, and you only discover this when coding. That's not a problem providing that you then change the design. The trouble comes when people think the design is done, and then don't take the knowledge they gained through the coding and run it back into the 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. In its usual form this is a model residing on a case tool. 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.

The last aspect of using UML is for documentation in a handover situation, such as when one group hands over to another. Here the XP point is that producing documentation is a story like any other, and thus its business value is determined by the customer. Again the UML is useful here, providing the diagrams are selective to help communication. Remember that the code is the repository of detailed information, the diagrams act to summarize and highlight important issues.

On Metaphor

Okay I might as well say it publicly - I still haven't got the hang of this metaphor thing. I saw it work, and work well on the C3 project, but it doesn't mean I have any idea how to do it, let alone how to explain how to do it.

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

I've built a system of names by building a conceptual model of the domain. I've done this with the domain experts using UML or its predecessors. I've found you have to be careful doing this. You need to keep to a minimal simple set of notation, and you have to guard against letting any technical issues creeping into the model. But if you do this I've found that you can use this to build a vocabulary of the domain that the domain experts can understand and use to communicate with developers. The model doesn't match the class designs perfectly, but it's enough to give a common vocabulary to the whole domain.

Now I don't see any reason why this vocabulary can't be a metaphorical one, such as the C3 metaphor that turned payroll into a factory assembly line. But I also don't see why basing your system of names on the vocabulary of the domain is such a bad idea either. Nor am I inclined to abandon a technique that works well for me in getting the system of names.

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". But I still don't think I've seen metaphor explained in a convincing manner. This is a real gap in XP, and one that the XPers need to sort out.

Do you wanna be an Architect when you grow up?

For much of the last decade, the term "software architect" has become popular. It's a term that is difficult personally for me to use. My wife is a structural engineer. The relationship between engineers and architects is ... interesting. My favorite was "architects are good for the three B's: bulbs, bushes, birds". The notion is that architects come up with all these pretty drawings, but it's the engineers who have to ensure that they actually can stand up. As a result I've avoided the term software architect, after all if my own wife can't treat me with professional respect what chance do I stand with anyone else?

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". The question then becomes one of whether separating yourself from the mundane programming effort is something you should do when you want to exercise technical leadership.

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. Indeed many of the proponents of XP - Kent Beck, Bob Martin, and of course Ward Cunningham - are those from whom I have learned much about what design is about. 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.

Those who know XP will realize that I'm describing the explicit XP role of Coach. Indeed one of the several games with words that XP makes 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. As we discussed what we did, the similarities in our approach were quite marked. 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. As Ward Cunningham pointed out, by that he amplifies his skills, and adds more to a project than any lone hero can.

Reversibility

At XP 2002 Enrico Zaninotto gave a fascinating talk that discussed the tie-ins between agile methods and lean manufacturing. His view was that one of the key aspects of both approaches was that they tackled complexity by reducing the irreversibility in the process.

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. The consequence for evolutionary design is that designers need to think about how they can avoid irreversibility in their decisions. 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, and of putting everything into such a system. While this does not guarantee reversibility, particularly for longed-lived decisions, it does provide a foundation that gives confidence to a team, even if it's rarely used.

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. This means that you have to set things up so that potential problem areas are rapidly tested to see what issues arrive. It also means it's worth doing experiments to see how hard future changes can be, even if you don't actually make the real change now - effectively doing a throw-away prototype on a branch of the system. Several teams have reporting trying out a future change early in prototype mode to see how hard it would be.

The Will to Design

While I've concentrated a lot of technical practices in this article, one thing that's too easy to leave out is the human aspect.

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 is one of the tasks that usually falls under the term 'architect'.

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. The keeper of the design doesn't have to be the one who fixes it - but they do have to ensure that it gets fixed by somebody.

A lack of will to design seems to be a major reason why evolutionary design can fail. Even if people are familiar with the things I've talked about in this article, without that will design won't take place.

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. Is this something which is such a pain to add later that you should start with it right away?

I could readily imagine that there are some things that would fall into this category. However the reality is that we still have very little data. If you have to add something, like internationalization, in later you're very conscious of the effort it takes to do so. You're less conscious of the effort it would actually have taken, week after week, to put it in and maintain it before it was actually needed. Also you 're less conscious of the fact that you may well have got it wrong, and thus needed to do some refactoring anyway.

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. By not doing them, you'll save a good deal of effort. Although there will be effort required to refactor the simple solution into what you actually need, this refactoring is likely to be less work than building all the questionable features.

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. In Planning XP, Kent and I openly indicated our disagreement. Kent is in favor of letting business value be the only factor in driving the ordering of the stories. After initial disagreement Ron Jeffries now agrees with this. I'm still unsure. 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.

With any new technique it's natural that its advocates are unsure of its boundary conditions. Most XPers have been told that evolutionary design is impossible for a certain problem, only to discover that it is indeed possible. That conquering of 'impossible' situations leads to a confidence that all such situations can be overcome. Of course you can't make such a generalization, but until the XP community hits the boundaries and fails, we can never be sure where these boundaries lie, and it's right to try and push beyond the potential boundaries that others may see.

(A recent article by Jim Shore discusses some situations, including internationalization, where potential boundaries turned out not to be barriers after all.)

Is Design Happening?

One of the difficulties of evolutionary design is that it's very hard to tell if design is actually 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. But sadly this is a subjective viewpoint. We don't have reliable metrics that can give us an objective view on design quality.

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. There's no easy answer to this, but here are a few hints.

- 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. However like any metric this can be abused, the opinion of good technical people trumps any metric, despite its subjectivity.

So is Design Dead?

Not by any means, but the nature of design has changed. 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.

That's a fearsome selection of skills, but then being a good designer has always been tough. XP doesn't really make it any easier, at least not for me. But I think XP does give us a new way to think about effective design because it has made evolutionary design a plausible strategy again. And I'm a big fan of evolution - otherwise who knows what I might be?

Acknowledgments

Over the last couple of years I've picked up and stolen many good ideas from many good people. Most of these are lost in the dimness of my memory. But I do remember pinching good ideas from Joshua Kerievski. I also remember many helpful comments from Fred George and Ron Jeffries. I also cannot forget how many good ideas keep coming from Ward and Kent.

I'm always grateful for those that ask questions and spot typos. I've been lax about keeping a list of these to acknowledge, but they do include Craig Jones, Nigel Thorne, Sven Gorts, Hilary Nelson, Terry Camerlengo.

Revision History

Revision History

Here's a list of the major updates to this paper

- *May 2004:* Added sections on 'The Will to Design', 'Reversibility' and 'Is Design Happening'
- *February 2001:* Article updated with sections on growing an architecture, the role of an architect, and where things that are difficult to add with refactoring.
- *July 2000:* Original article submitted to XP 2000 and posted to martinfowler.com

---
