Evolutionary architecture and emergent design: Environmental considerations for design, Part 1

Estimating emergent design and demonstrating technical debt

Neal Ford

October 26, 2010

As much as we'd like it to, software design doesn't exist in a vacuum. This installment of Evolutionary architecture and emergent design discusses how two environmental factors — estimation and technical debt — can affect design decisions, and how your emergent designs can survive the rough political waters in which they may find themselves.

Estimated emergent design and demonstrating technical debt

As some of my readers know, this article series is symbiotic with a talk by the same name that I present at conferences. One common theme that arises during the talk's question-and-answer portion concerns fitting agile design into an ecosystem that hasn't quite swallowed the agile red pill. The number of questions on this theme has made me realize how important it is.

Previous articles in this series have covered technical considerations for emergent design, but software designs don't occur in a vacuum. Nontechnical considerations have an impact on design too. One of the most common questions about emergent design is "How do you estimate for it?" And one of the most common obfuscators of emergent design is technical debt — compromises imposed on your design for the sake of some external force, such as schedule pressure. This installment discusses how to estimate for emergent design, and how to make a convincing case for repaying technical debt.

Estimation for emergent design

Martin Fowler presented a great analogy in a joint keynote he and I gave earlier this year about why agile practices work. Consider fruitcake and hotel showers for a moment. Fowler apparently has a killer fruitcake recipe. He has cooked it many times, and he's confident that if he measures
the ingredients properly and cooks it at the specific temperature for the indicated time, he gets the same fruitcake every time.

Contrast that experience with choosing the correct temperature for a hotel shower. It would be dangerous folly to put it at the same setting you use at home and jump in. Instead, you turn it on and sample the water temperature for a bit until it’s warmed up, then use a quick feedback loop of adjustment, measure, repeat until it feels correct.

The difference between these two types of processes illustrates the broader difference between prescribed and feedback-driven projects. If you have cooked the same dish many times before, you can follow a recipe with confidence. Similarly, estimation for a software project is easy if you can meet the following criteria:

- You are building the exact piece of software you've written before (more than once).
- You have the exact same team you've used to build it in the past.
- The environment is exactly the same.

Writing new software isn't a deterministic process; it is highly adaptable, which suggests that a feedback-based approach works better than a prescriptive one. Which brings me back around to the original question, slightly restated: "How do you estimate for the design effort in a highly adaptive system?"

**Coarse-grained estimates and wiggle room**

My employer (ThoughtWorks) mitigates the highly variable nature of software projects in several ways. First, we convince clients that the first estimate we provide will not be exact. Exact estimates in software are impossible anyway, because you are being asked to provide one of the most nuanced details of the project at the time you know the least about the project. We tell clients up front that our first estimate will be based on an understanding of the coarse-grained business requirements and technical architecture. (For more details about this process, see the Iteration 0 sidebar.)

**Iteration 0**

ThoughtWorks usually starts new projects with an inception phase called iteration 0. During this period, which varies in length depending on the estimated project size but is generally between two and eight weeks, two concurrent activities occur:

- The business analysts define coarse-grained requirements, traditionally captured on index cards (chosen precisely because you can't put too much detail on them).
- The technical members define the architecture ("the stuff that's hard to change later," as defined in the first installment) and any nonfunctional requirements (a term I despise, by the way — I much prefer quality of service) such as scalability and performance.

At the end of the inception phase, the business and technical people gather to gauge the complexity of each story, based loosely on the techniques defined in Mike Cohn's outstanding book *User Stories Applied: For Agile Software Development* (see Related topics).

Once the team completes the process, it now has an estimate of the project's total complexity, which allows the creation of burn-down charts and other statistical data about...
The coarse-grained estimate reflects an initial understanding of the scope and effort. However, we don't rely on that estimate for long, because our understanding starts growing as soon as we start work. Because agile development happens in fairly short iterations, it allows project managers to gather real statistics on all of a project's disparate details: the real velocity of this team, on this project, in this environment, for this problem domain. Because iterations consist of the same activities repeated, project managers can gather real data right away. After three or four iterations, the project manager has adjusted the load factor (the number used to convert complexity into time) to reflect the realities of this project. Thus, we usually update our estimate after a few iterations with more-accurate numbers. This is the best you can do in a highly adaptive environment: make educated guesses, then immediately start applying real data to measure (and feed back) how you've done.

Because the initial estimate is so coarse-grained, a fair amount of wiggle room exists to handle emergent design, refactoring, and other technical-hygiene activities. Unless they involve an unusually design-heavy application, ThoughtWorks projects don't specifically reserve time for these practices, just as we don't reserve time for other common practices like meetings.

An alternative to amortizing design time into the project estimates is to create design checkpoints in your schedule. This is especially attractive in organizations that are moving from a big-design-up-front methodology that reserves a specific amount of time for design. If that time is already expected, use that same amount of time, just don't do it up front. Instead, place some markers in your schedule at reasonable milestones (such as releases or a handful of iterations) to revisit existing design decisions and do the next round of microdesign. By waiting on design decisions until the last responsible moment, you have better knowledge and context for the real impact of the decision. What you'll find if you reserve the typical amount of up-front design time and spread it out over the project is that you'll have tons of time to make design decisions.

Another task that software projects often need time to address is repayment of technical debt. Next, I'll discuss some tools and techniques that can help you to expose this problem to nontechnical colleagues.

**Repaying technical debt**

I covered the basics of technical debt — a terrific metaphor first elucidated by Ward Cunningham (see Related topics) — in this series' first installment. Technical debt resembles credit card debt: you don't have enough funds at the moment, so you borrow against the future. Similarly, your project doesn't have enough time to do something right, so you hack a just-in-time solution and hope to use some future time to come back and retrofit it.

Martin Fowler has written about four quadrants of technical debt (see Related topics), as illustrated in Figure 1:
The first quadrant (reckless and deliberate) comes about consciously. This is illustrated in the classic cartoon that shows a manager telling a group of developers, "You guys get busy, I'll go upstairs and see what they want." Companies with a lot of this type of debt are in a big hurry, willing to trade ever-decreasing velocity for expediency of delivery. This is obviously not sustainable for long periods or large code bases. The second quadrant (prudent and deliberate) is the most common manifestation of technical debt. And it's the one least likely to cause big problems later, as long as you realize you've incurred debt and can pay it back. The third quadrant (reckless and inadvertent) is the most troubling, because these developers incur debt without realizing it. A common kind of statement from a developer on such a project might be, "It's really convenient to have all 5,000 lines of code embedded right in the JSP so that you can just scroll up and down to see where you've defined all your (global) variables."

The fourth quadrant (prudent and inadvertent) seems like it shouldn't exist — if you are prudent, how can it be inadvertent? In fact, this is a common outcome on projects that have expert designers. Even the best practitioners of software design can't anticipate all the ways a design will manifest and evolve over time. This last quadrant is a reflection of the fact that one of the toughest problems in software (to quote the poetry of Donald Rumsfeld) is that we don't know what we don't know. In other words, the hardest things to deal with are the things that we don't even know are problems yet.

Technical debt is a reality in the software world. It will never go away, because schedule pressure will always exist as a reflection of the fact that business decisions can be made faster than we can encode them into software. There is nothing inherently bad about technical debt, just as there is nothing inherently bad in real-world debt. And just as prudent companies take on monetary debt strategically, the same can be done with software. The problem isn't the debt, it's negotiating repayment of the debt. It has been my experience that demonstration trumps discussion. If you go to your project manager and say, "I feel sick inside, I can't sleep at night; I suspect our code is starting to suck," your manager will shoot back, "Can you prove that?" You can. Next, I'll show two illustrations of technical debt: the first generated by hand and the second using a tool called Sonar.

**Generating technical debt charts**

The first illustration comes from a real-world ThoughtWorks project for a public-facing media site, whose technical lead Erik Dörnenburg (see Related topics) is well-known in the software metrics and visualizations world. On this project, he created a graph showing increasing technical debt, shown in Figure 2:
Figure 2. Technical debt graph

Roll-your-own metrics
Dörnenburg produced the graph in Figure 2 with a clever combination of low-level tools. To generate cyclomatic-complexity numbers, he used JavaNCSS (see Related topics). He wrote a script that checks out version 1 of the code from version control, runs JavaNCSS on it to produce data, and saves it in a dated file. The script repeats this process for each of the versions in version control. When it has completed, he has a large number of XML files with raw data in them. He then wrote a little Ruby script to flatten the data from the individual files into one comma-separated value (CSV) file, then loaded that into Excel and told it to chart it. After you've generated this chart once, you can short-circuit the process by keeping the raw data around for previous builds so that you only have to update the data per build, using Continuous Integration.

The graph's horizontal axis shows the time range (April 1, 2006 to August 18, 2007), and the vertical axis shows cyclomatic complexity per line of code. (This installment of the series covers the cyclomatic-complexity metric in a discussion about testing.) The gray numbers overlaying the bottom are the project's releases.

The first public "go live" release of this code base happened at release number 3. On the graph, you can see the amount and volatility of complexity rising up until that point — a side-effect of the business people almost literally standing over the shoulders of the developers, constantly asking, "Is it done yet?" They don't make money until the first release goes live, so obviously they are concerned.

Once the project hit its ship date, Dörnenburg created the first part of this graph and showed it as evidence of mounting technical debt, then used that to negotiate a couple of short maintenance releases (versions 3.0 and 3.1) to clean up some of the debt. The inertia of that effort carried all the way to release 7, where debt starting rising again (although in a much more controlled way) for unrelated reasons.

Illustrating debt with Sonar
The chart in Figure 2 is impressive, but it requires a bit of work to bootstrap the process. An alternative to rolling your own is the open source tool Sonar (see Related topics). Sonar captures
numerous metrics for your code base and shows them in charts and other visualizations. The creators of Sonar have run it against a variety of open source projects (accessible via a running instance of Sonar at http://nemo.sonarsource.org/), including Struts. Figure 3 shows Sonar's results for Struts:

**Figure 3. Sonar showing details about Struts**

Sonar shows the output of common quality tools in the Java space, including CheckStyle (see Related topics), code coverage, and one of its own metrics, the Technical Debt Calculator (shown on the right-hand side). This formula uses a bunch of numbers derived from metrics run on the project's code.

Sonar seems to produce scarily high numbers out of the box. For example, it suggests that it would take 572 man days and $280,000 to get Struts out of debt, which I think is highly inflated. You'll certainly want to tweak these numbers for your project before you take them to your manager. If you produce a report that says that it will take $1.2 million to get your code to the point where it doesn't suck, your manager is going to jump out a window. Tweak the numbers to support your case that you need to put some full-time resources on cleaning up the debt.

Sonar has some other nice canned visualizations as well. Consider the "Time Machine" graph shown in Figure 4:
This graph shows three key metrics over time: code coverage, cyclomatic complexity, and cyclomatic complexity per method. As you can see from this chart, something horrible happened to Struts around September 1, 2009. It apparently subsumed another framework that had terrible metrics, which are now reflected in the Struts codebase. Contrast this with this same visualizations but for Spring Batch, shown in Figure 5:

### Figure 5. Time Machine chart for Spring Batch

The chart in Figure 5 shows more of what you'd like to see: relatively constant coverage, constant complexity per method, and slowly growing overall complexity as the software supports additional features.

One of the reasons the technical debt metaphor works so well lies in the similarity to money for monetary debt to time in software projects. When you have debt and you receive more money, some of that money must make interest payments on the debt. On software projects, you deal with time instead of money. When you get a new chunk of time to add new features, you have to pay some of that time back via the extra time it takes to work around all the design (and other) compromises. It is worth making the case to put full-time effort on reducing the debt, because it allows everyone to go faster once it is resolved. Fixing technical debt allows the velocity of the whole team to increase.

### Conclusion

In this installment, I started looking at nontechnical factors that have an impact on emergent design. I discussed how to estimate for the unknowable, and how to illustrate technical debt. In the next installment, I’ll continue down this path of external concerns around emergent design, including refactoring and isolating changes.
Related topics

- *The Productive Programmer* (Neal Ford, O'Reilly Media, 2008): Neal Ford's most recent book expands on a number of the topics in this series.
- **Martin Fowler**: Fowler, the chief scientist at ThoughtWorks, is a well-known author of seminal software engineering books and a highly influential blog.
- **TechnicalDebtQuadrant**: Martin Fowler writes about technical debt quadrants in this essay.
- **Erik Dörnenburg**: Dörnenburg has spoken and written extensively on the subject of visualizations for software.
- **In pursuit of code quality**: Monitoring cyclomatic complexity (Andrew Glover, developerWorks): Learn how to use simple code metrics and Java-based tools, including JavaNCSS, to monitor cyclomatic complexity.
- **Sonar**: Sonar is a open source toolkit for common metrics and advanced visualizations
- **CheckStyle**: CheckStyle is an open source static source code analysis tool

© Copyright IBM Corporation 2010
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)