

# Software Project Management

6.170 Lecture 19

Spring 2004

Today's lecture is about issues in managing software projects — both large projects, out there in the real world, and small teams like your final project group.

Looking back over the lectures so far, we've covered a number of big ideas: specifications, abstract data types, abstraction functions and rep invariants, testing, object models and module dependency diagrams, and design patterns. These topics are at the heart of software engineering; they distinguish the discipline of software engineering from mere programming.

Out of these tools and ideas we've discussed, two important themes arise. The first theme is *correctness*: producing software that is free of bugs, but without wasting a lot of time debugging. Correctness is obviously a desirable goal.

The second theme is *communication*. Software development requires communication not just with the computer, but also with other people. Many of the artifacts we've learned about aren't understood by the compiler at all — specifications and abstraction functions being obvious examples. If all we cared about was communicating with the computer, we might not bother with them. But software engineering isn't just about one person programming. It takes teams, sometimes large ones, to build significant software.

## 1 Waterfall Model

Let's start by looking at the steps of a typical software project. The *waterfall model* was one of the earliest carefully-articulated design processes for software development. It models the design process as a sequence of stages. Each stage results in a concrete product that feeds into the next stage:

- **Requirements analysis** is the process of finding out from the customer what software they need. This phase typically involves interviewing the customer, doing market research, and observing the customer's current processes that the planned software should replace or augment. The output of requirements analysis is a requirements document: a list of requirements that describe what the system should do and any constraints on how it should do it (e.g., performance, resource consumption, customizability). It's important that the requirements actually be *written down*, so that you and the customer are clear on what will be done. The requirements should be written in natural language (i.e., English), so that the customer can understand them too.
- **Design** takes the requirements and produces a system design. In design, you decompose the problem into modules with well-specified interfaces. In small-team projects, the decomposition is usually done

in a group meeting, involving the entire team in developing the overall architecture of the system. Detailed low-level design — i.e., writing the specs for each class — is then divided up among the team members. The output of the design phase is a system decomposition (expressed in module dependency diagrams and object models) and detailed specifications of each module's interface.

- **Coding** takes the specifications and implements them. This is the programming part. In *test-driven development*, coding is preceded by generation of black-box test cases from the specifications, so that you can validate as you code. Coding is always divided up among team members. Often, a different team member is responsible for writing black-box test cases. The output of coding are implemented modules.
- **Testing** validates the implementation. Testing is also called quality assurance (QA); it includes not only unit tests, but also *integration tests* that exercise a subsystem or the entire system. Testing also includes *acceptance tests*, which are tests specified by the customer as part of the requirements.
- **Release** occurs when the software is actually put into users' hands.
- **Maintenance** comprises the rest of the software project's lifetime: fixing bugs, making enhancements, and keeping the software up to date with changes in technology. For most software projects (that actually manage to make a successful release without being cancelled), the maintenance phase is by far the longest. It also suffers from more staff turnover — the programmers working on maintenance are far less likely to be the original developers of the product. The maintenance problem is one of the most compelling reasons to *write things down* — if the original designers aren't around anymore, there's no way to ask them how things should work.

The biggest improvement of the waterfall model over previous (chaotic) approaches to software development is the discipline it puts on developers to think first, and code second. Requirements and designs precede the first line of code, and result in written artifacts that codify communication between team members.

This is in stark contrast to what might be called the *hacking model*, which has only two phases: (1) write code until the system is built, and (2) ask the customer whether it's right.

The biggest danger of the hacking model is the cost of getting it wrong. Studies of actual software projects have found that the cost of fixing a defect (such as a bug, or a missing requirement) increases *exponentially* the later it is found in the software development process. (See Boehm, *Software Engineering Economics*, 1981.) Consider, for example, the difference between finding a bug during unit testing, and finding it during acceptance testing. At unit test time, the only possible place for the bug is in the module being tested. In the acceptance test, however, the bug might be *anywhere* in the code — a substantially more costly bug to find. Worse, if fixing the bug requires some redesign of the system, you may have to *throw away* code that you already implemented and tested.

Many parts of the waterfall model are designed to reduce the risk of costly rework from a latent defect. That's why we write down requirements and design, and why we do test-driven development.

## 2 Spiral Model

Unfortunately, the waterfall model doesn't overcome all the risk. In particular, a missing requirement may not be discovered until the acceptance test. So the waterfall model works well when the requirements are

stable and well-understood, or (equivalently) when the developers have built similar systems before, so they have a better sense of what requirements to expect. The waterfall model does *not* work well for risky projects, where the requirements may need to change drastically as you and the customer understand the problem better.

For these kinds of projects, a better approach is the *spiral model*. We discussed this model in the lecture about usability engineering. The spiral model has a cycle of phases:

- **Requirements analysis and design** are the same as in the waterfall model.
- **Prototyping** takes the design and produces one or more prototype implementations.
- **Evaluation** determines how well the prototypes satisfy the requirements. The results of the evaluation are fed back into another requirements and design cycle.

The model spirals outward — successive cycles create higher-fidelity, more costly implementations, until eventually producing an implementation suitable for release. The spiral model mitigates the risk of incorrect requirements or inappropriate designs by getting early feedback from a prototype implementation.

Prototyping is not the same as hacking. First, prototypes are carefully designed — obviously, since we still have requirements analysis and design phases, in which the requirements and design decisions are actually considered and written down.

Second, a prototype is intended to answer specific questions about the risky parts of the project. (Well-understood parts of the project don't need prototypes; they can be developed with the waterfall method. It's perfectly fine to mix the two methods on the same project.) Examples of these questions include: What does the client really need (requirements)? How should the user interface be designed? Is a tricky data structure specification feasible to implement? Where are the performance bottlenecks of the design likely to be? How do the building blocks on which the software will depend — programming language, libraries, hardware — actually behave in the context where we need to use them?

Third, prototypes are intended to be *thrown away* — not reused in the eventual release. The purpose of the spiral model is to allow cheap prototypes to give you experience with the risky parts of the project. But a cheap prototype usually has had some corners cut — missing assertions, undocumented assumptions, hard-coded limits. Allowing prototype code to drift through into the release is one of the biggest dangers of the spiral model. It's hard to have the discipline to throw away a working prototype, but it's far better to write it from scratch.

### 3 Throwing One Away, and the Second System Effect

Is it wasted work to throw away a prototype? Not likely. A well-known aphorism in computer system design is “Plan to throw one away; you will anyhow.” (Fred Brooks, *The Mythical Man-Month* – a classic book about software development that every software engineer should read!) It means that your first attempt at a new design is very likely to be lousy. If you plan for it, and build a cheap prototype of the risky parts instead of a complete implementation, then you'll suffer far less in the long run.

But beware of another phenomenon in system design: the *second-system effect* (also coined by Brooks). When you're building the system the first time, whether it's a prototype or a full waterfall implementation,

you inevitably think of all the cool features and improvements you could make. So the second time you build it, you pile all those features into your design, making it baroque, complicated, and slow; or, as Brooks puts it, “a big pile.”

In the waterfall model, one solution to the second-system effect is to make sure at least some of your team members are on their *third* system of this kind, so they can act as a brake on the enthusiasm. In the spiral model, you can build at least two prototypes before you do your final implementation.

A better way to avoid the second-system effect is to discipline yourself and always strive for simplicity, whether it’s your second system or your *n*th. Don’t include any features or introduce any design complexity unless it passes a high bar of necessity.

## 4 Scheduling

“More software projects have gone awry for lack of calendar time than for all other causes combined.” (Fred Brooks)

Scheduling is a crucial part of software development. Unfortunately, it usually gets far less attention than it should. Schedules are made to fit other constraints, instead of the other way around. Estimates about how long it will take to build a piece of software are almost always too optimistic, reflecting what you *wish* to be true, and rarely taking into account all the deadends and debugging it will take to get there.

The great paradox of software planning is that everything will take twice as long as you think — even if you already know that it will take twice as long as you think. Chew on that one.

Effort is not the same as progress. We can measure effort easily: how many hours you spent working. But progress in software is much harder to measure. We can’t simply count the number of widgets rolling off the assembly line, because software doesn’t have an assembly line: every piece of code is one-of-a-kind. That means it’s harder to tell when you’re slipping behind the schedule.

A schedule should consist of verifiable milestones. Examples include: design complete; BipartiteGraph 100% coded; unit testing of all modules done. For estimating the time you’ll need, here are some rules of thumb:

- Allocate 1/3 of your time to planning and design (not necessarily all up front)
- Allocate 1/6 to coding
- Allocate 1/4 to unit testing
- Allocate 1/4 to integration and system test

What do you do if you start to slip? Take aggressive action; don’t just let the slip slide, because little slips accumulate into missed milestones. To decide what to do, consider the four factors that determine a software project: time, people, quality, and scope. If your time estimate was wrong, you have to either change your estimate (extending the schedule), or else make sacrifices in one of the other factors. Adding more people rarely works, because the effort of bringing them up to speed on the project, and the time spent on communication within the team goes up quadratically with the number of people. Sacrificing quality is a bad idea. So that generally leaves scope: reducing the features you planned to include in the product.

## **5 Summary**

In this lecture, we looked at some of the problems encountered in team software development. We saw that the waterfall method, which takes a linear approach to the design, implementation, and validation of a software artifact, is effective for problem domains where the requirements are well-known and stable, but the spiral model is better for risky parts of a project.