ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 21 — March 7, 2013

Patrick Lam *version 1*

Software Development Lifecycle

If you're asked to develop a software project, you're likely to follow a process that looks like this:

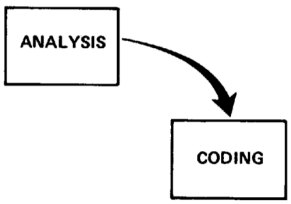


Figure 1. Implementation steps to deliver a small computer program for internal operations.

(Figures are from [Roy70].)

This is fine, but it doesn't scale. You will fail if you try this on a sufficiently-large project, or you will evolve some sort of more refined process: managing large software projects is notoriously difficult. Software development lifecycles attempt to provide more structure to keep the activities surrounding software development on-track.

Key idea: Iterations. Lifecycles always involve iterations of design stages. More complex projects will typically have both more design stages and more iterations.

Lifecycle models help organize the stages of the design and implementation process.

Process. Good process can help with avoiding fiascoes and deathmarches. Project management is a huge part of the software development lifecycle. Without effective project management (of some sort), a software project is likely to be delayed and poorly-designed.

The software design process resembles the engineering design process, in that both attempt to build the best possible design given sets of project requirements, project constraints, and criteria for evaluating design success.

A key difference between general engineering design and software design is that you can deploy software immediately after implementing it; typically, the result of engineering design gets dispatched to manufacturing (or construction companies).

High-Level Overview

A general list of steps in the software design process is:

- Problem Definition
- Requirements Development
- Project Planning
- High-Level Design
- Detailed Design
- Coding and Debugging
- Integration Testing
- System Testing
- Corrective Maintenance

The different lifecycle models link these steps in various ways. We're going to talk about four software lifecycle models in this course:

- Waterfall
- Spiral
- Concurrent Engineering
- Extreme Programming

There are other models, but they're similar to the ones we'll talk about.

If you follow a model, then good things may happen. Following a model poorly is a potential recipe for disaster, in the form of poorly designed and implemented software, and many bug-fixing design iterations.

Waterfall Model

The old classic idealized "model of a software engineering process" is the waterfall model.

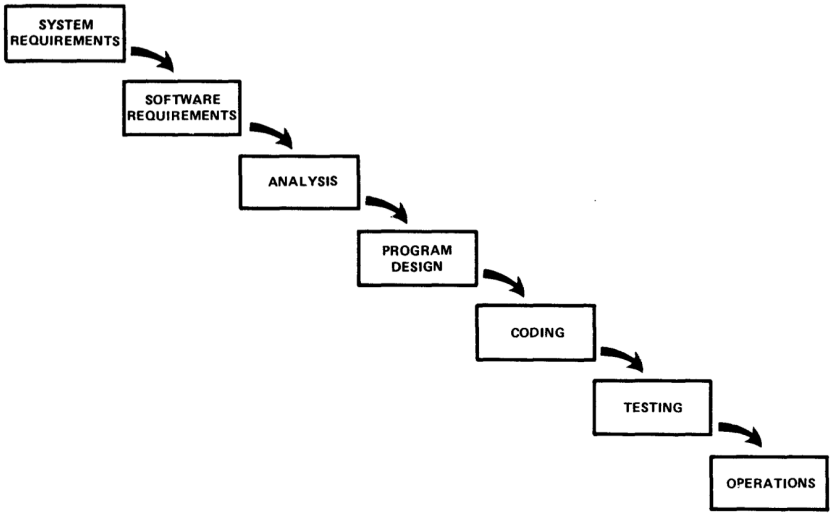


Figure 2. Implementation steps to develop a large computer program for delivery to a customer.

- The waterfall model is highly sequential: stages may not overlap. The project moves onto the next stage following a review.
- Advantages: 1) it fixes customer requirements early in the design process (hopefully the right requirements); 2) in principle, models like this would identify problems early in the design process, when changes are less expensive.
- Disadvantages: 1) you're working blind, so that you don't see any software until the end of the implementation stage (causing critical failures in practice); and 2) changes late in project development imply lots of wasted work.

Actually, no one seriously advocates this model—that is, it's a straw man. Even in the original paper showing the waterfall model, the author pointed out that you'd be more likely to get this:

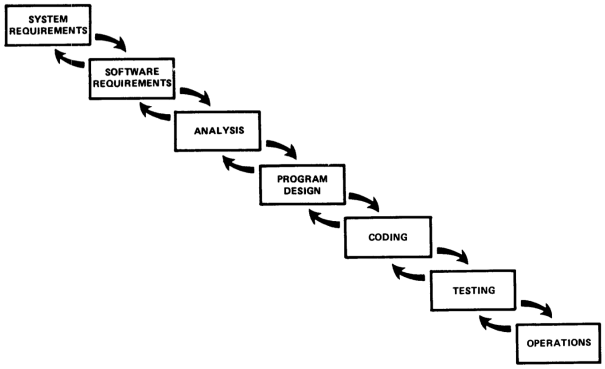


Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.

and if you were less lucky, you'd get something more like this:

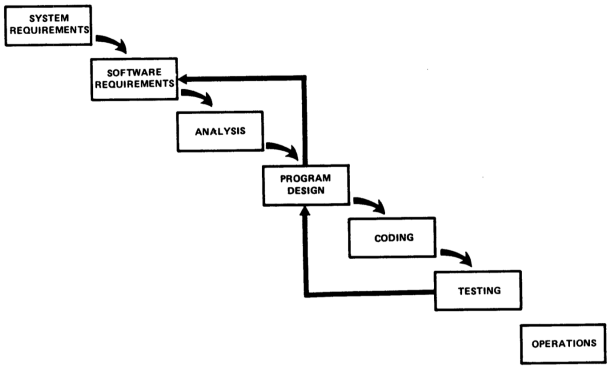
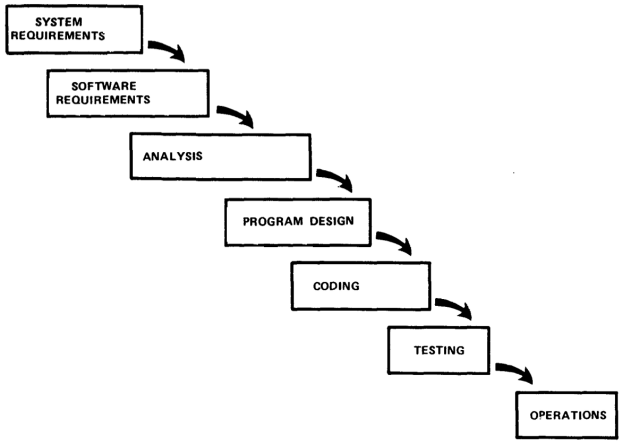


Figure 4. Unfortunately, for the process illustrated, the design iterations are never confined to the successive steps.

Concurrent Engineering

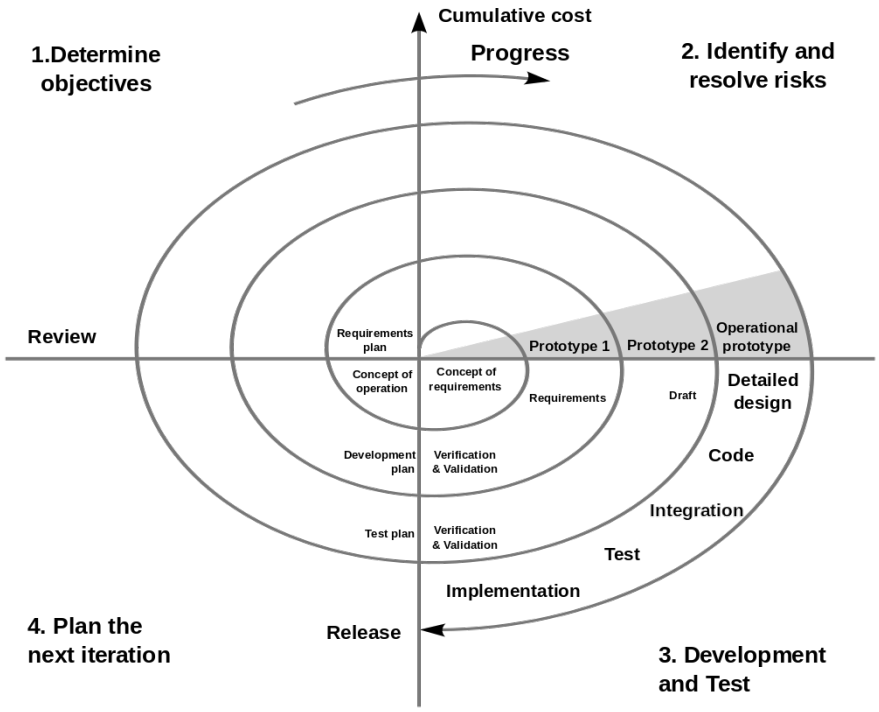
A variant on the waterfall model is the concurrent engineering model. Instead of waiting on the previous stage to finish, you can start the next stage once you have something to work with. Because the stages overlap, this model is also known as the "sashimi" model.



- The concurrent engineering model works well for many projects (why wait?) Because you're trying to use the result of a stage, you are more likely to discover problems with it and correct it, in collaboration with the team that produced the result.
- Advantages: 1) because you don't need to write down every last (irrelevant) detail, you might need less documentation; 2) projects need not be subdivided into smaller projects; 3) testing may reveal problems earlier in the development process.
- Disadvantages: 1) milestones may be more ambiguous; 2) progress is more difficult to track: how done is stage x, anyway?; 3) poor communication can lead to disaster in the presence of parallel design stages.

Spiral Model

The spiral model is a much more iterative variant on the waterfall model. You continue going through stages, in order, until you get to a satisfactory solution. Projects are split into a number of smaller sub-projects, with each iteration corresponding to a smaller project. You'll iterate many times. Not all stages of the design process require equal effort: testing may, and often does, require more effort than coding.



http://en.wikipedia.org/wiki/File:Spiral_model_%28Boehm,_1988%29.svg

- The spiral model is risk-oriented, and each sub-project addresses one or more risks, in order of magnitude, until you get to a satisfactory solution where all of the major risks have been addressed. (We'll talk more about risks later).
- Advantages: 1) this model addresses the biggest risks first, when changes are least expensive; 2) progress is visible to the customer and to management.
- Disadvantage: some projects don't have clearly identifiable sub-projects with verifiable milestones; this model is generally identified with more heavyweight process than extreme programming.

Extreme Programming

We can call extreme programming [BA04], or XP, another software lifecycle model, although it differs from the other models quite a bit. It's closest to the spiral model, scaled down and made

more agile. The initial idea was to take the "good" parts of good programming practice, like reviews and testing, and "crank up all the knobs to 10"[1] on those, leaving everything else behind.

Agile Methodologies. XP is one of several agile methodologies, which all attempt to be less bureaucratic than the traditional "heavyweight" methodologies.

Values. XP supports five values in software development:

- Communication: Work together. Includes pair programming. Face-to-face communication. Workspaces to support collaboration. Don't generate paperwork. Share knowledge.
- Simplicity: Don't do more than you need to. Take small, simple steps to goal. "You Ain't Gonna Need It". Requires refactoring later.
- Feedback: Get feedback from system (unit tests), from client (functional/acceptance tests), from the team (time estimates). Demo working software.
- Courage: Code for today, not tomorrow. Refactor when necessary, don't be scared. Throw code away when necessary. Work together to avoid failure and don't fear it.
- Respect: Respect contributions of other team members (devs and customers). Management must respect judgment of programmers. Don't break the build or otherwise waste others' time.

References

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[Roy70] Winston W. Royce. Managing the development of large software systems. In *Proceedings IEEE WESCON*, 1970.

[1] `http://www.informit.com/articles/article.aspx?p=20972`