BIF-5-SSD

Lecture 10. Software Development Process

Martin Bush, LSBU
Taken from a previous module Software Engineering Principles, modified by Peter Rosner

These lecture notes are partly drawn from chapters 21 to 27 of the set book for this module…

"Software Engineering for Students"
by Douglas Bell, Addison-Wesley, 2005.

There are several well-known approaches to software development, including…

- ad hoc software development
- the waterfall model
- prototyping
- agile methods and extreme programming
- open source software development
- formal methods.

In the context of software development, a process is an activity or a series of activities. Most processes are geared towards generating some kind of products, such as a statement of requirements, a design, or an executable program.

Think of it like this:

Resources → Process → Product(s).

Whatever the approach, the overall process of software development must incorporate all of the following:

- requirements analysis
- architectural ("high level") design
- detailed ("low level") design

- coding ("implementation")
- verification ("are we building the product right?")
- validation ("are we building the right product?!").

Ad hoc software development

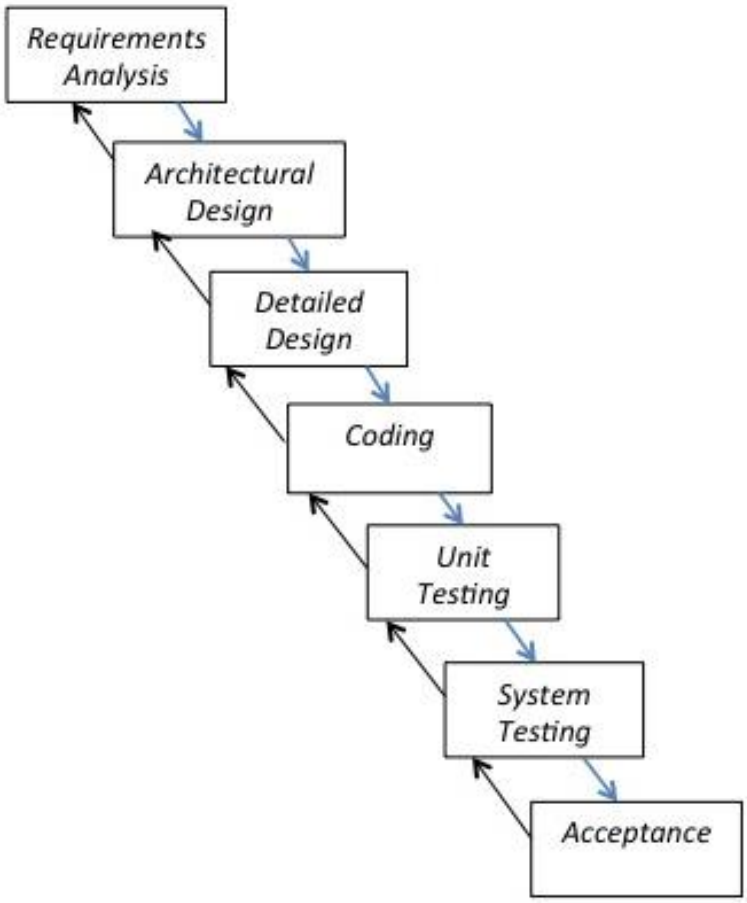
"Ad hoc" means "one off" – i.e. potentially different every time. Ad hoc processes are usually unplanned, left to personal preference, and difficult to manage. This may be OK for an individual working on a small project, but it is unsuitable for an organisation that has to develop complex products in a methodical and manageable way..

The waterfall model

The waterfall model was proposed in the late 1960's (i.e. around the time that the phrase "software engineering" was coined) in recognition of the fact that the previous ad hoc approach to software development was becoming inadequate given the growing power and (especially) the memory capacity of computers, and hence the size of typical computer programs.

The waterfall approach to software development is often spoken of as being the traditional approach.

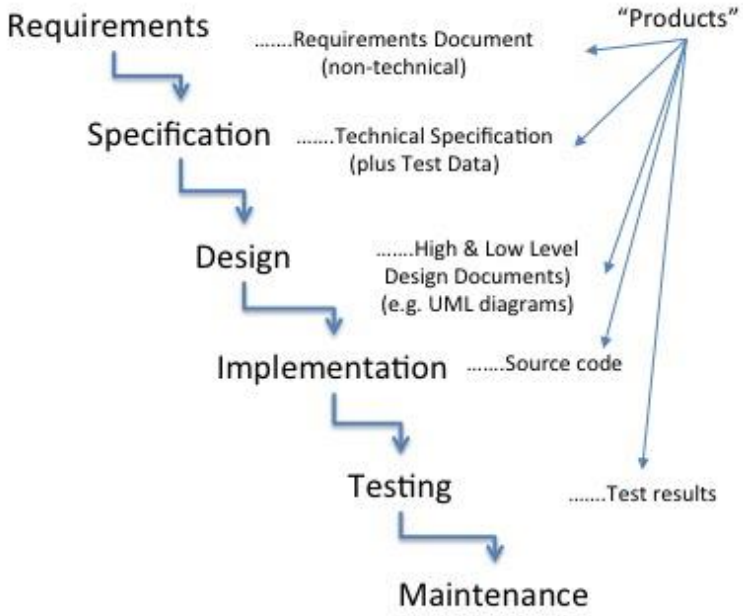
The waterfall model looks like this



Different authors often give different names to the individual stages (or processes), but the basic idea is always the same.

Although feedback is shown only between neighbouring processes, in practice it is often necessary to go back several steps, to revise a design for example, or even the requirements.

Here's another version, this time with *products* shown…



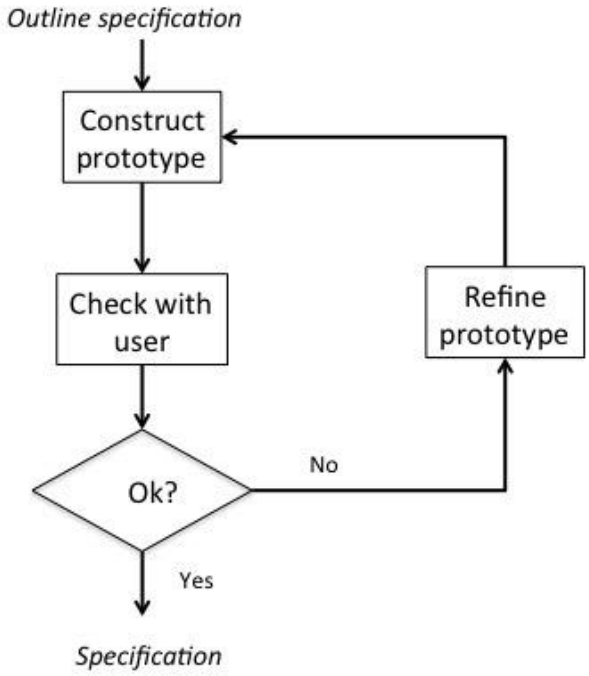
The main problem with the waterfall approach is that (in theory) the user doesn't get to see the end product until near the end of the project. The waterfall model is important for historical reasons, and it is referred to a lot in the literature on software engineering, but most organisations now adopt a much more iterative approach to developing software, based on prototyping.

Prototyping

A prototype is simply a mock-up of the finished product. It is something that a developer can show to the customer (user and/or stakeholder) to gauge his or her reaction. It could simply be a screenshot showing what the product might look like, or a half-baked implementation with partial functionality.

Prototyping is particularly appropriate during the early stages of a project, especially if the exact requirements for the system aren't entirely clear.

There are two basic kinds of prototyping; throwaway prototyping and evolutionary prototyping. Throwaway prototyping is depicted in Bell's textbook as follows…



A throwaway prototype is often constructed as an initial *proof of concept*, especially when undertaking a technically demanding project.

The concept of a throwaway prototype brings to mind this famous quote from Fred Brooks' very influential 1975 book entitled *The Mythical Man-Month*:

> "Plan to throw one away; you will, anyhow."

Bell's diagram for evolutionary prototyping looks just like the one for throwaway prototyping, except that the output this time is a working system rather than just a specification.

For example, a final year undergraduate student project may follow the evolutionary prototyping model; this is a very natural approach whenever there is uncertainty about what needs to be developed.

Prototyping – in some form or other – is extremely common, and it is used even by very experienced software developers. It has many potential benefits, including…

- misunderstandings between customer and supplier can be exposed early,
- errors in the specification may be exposed early (actually there's less emphasis on a written specification now, whereas it often formed a binding contract in traditional waterfall-style projects),
- missing functionality may be detected early,
- difficult or confusing functions can be identified and refined early,
- a working prototype can be a good way of winning support from management,
- a working prototype can be used for training purposes,
- the customer will (hopefully) be more satisfied with the final product, since they have seen it develop.

The growth in popularity of prototyping has a lot to do with changing technology. In particular, the trend towards increasing re-use of existing components – resulting partly from the emergence of object-oriented languages such as Java – has made prototyping seem very natural.

There are various potential pitfalls though, from the point of view of users, software developers and managers.

The users may…

- be led to believe that the system will be ready soon,
- keep changing their minds, if they are given the opportunity to do so,
- get carried away with some aspects of the requirements but neglect other aspects.

The developers may…

- neglect to undertake a thorough requirements analysis,

- get frustrated that the users keep changing their minds,
- neglect non-functional aspects of the design (e.g. reliability, performance, ease of maintenance).

The managers may…

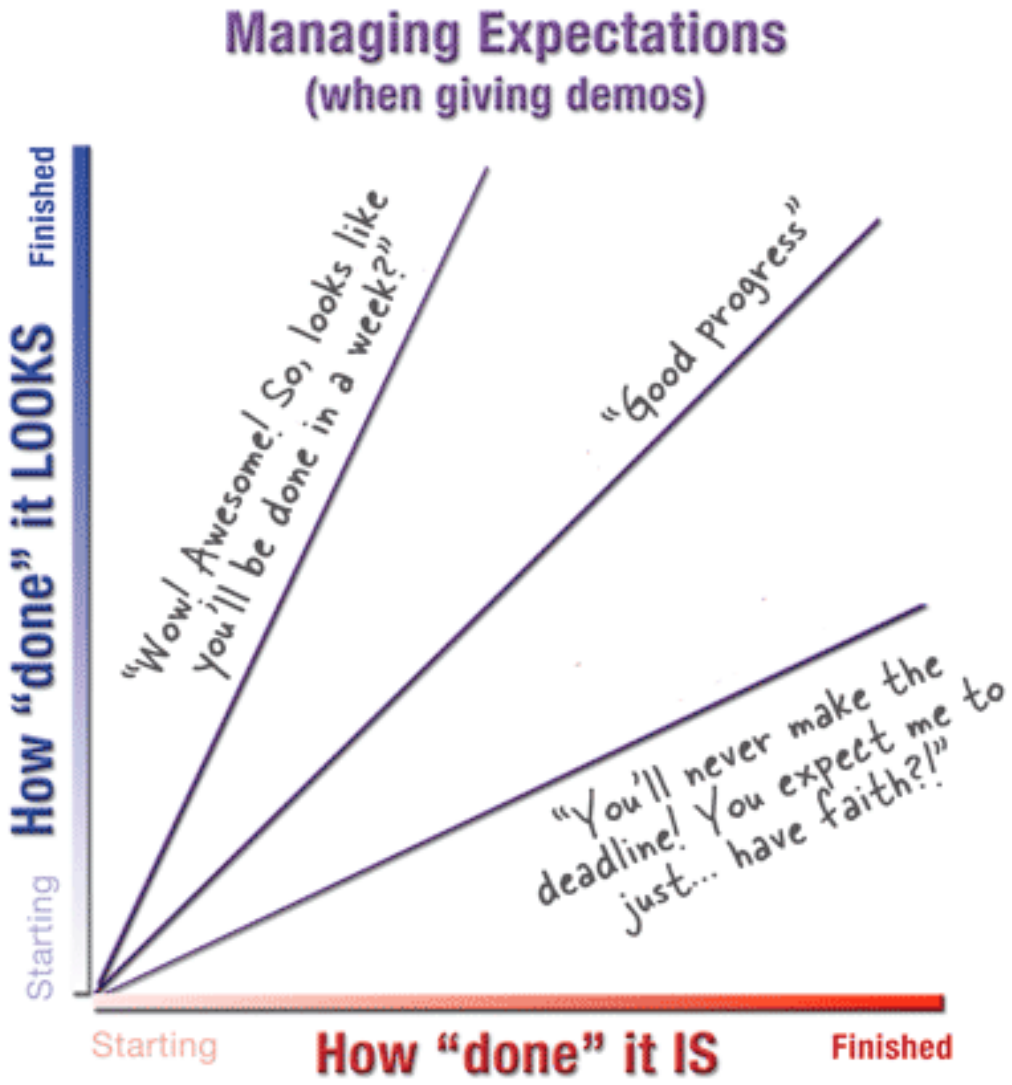
- find it hard to predict and control the number of design iterations that are necessary,
- find it hard to enforce configuration management, leading to confusion about which versions of which modules make up a given prototype.

Managing expectations

Here's a nice diagram from an article on the web…

[http://headrush.typepad.com/creating_passionate_users/2006/12/dont_make_the_d.html]

…entitled "Don't make the Demo look Done"…



The diagram is pretty self-explanatory, as is the sequence of diagrams shown below (which are taken from the same article)…

The better it looks, the more narrow the feedback

Looks Done

Mocked up in Photoshop, a multimedia program (Director, Flash, etc.), or a GUI builder (NetBeans, Visual Studio, etc.)

"Can you change the font on that "T"?
Not sure I like the bevel line weight..."

Feedback: detailed tweaks to specific features. Very focused and incremental.

Visio, Powerpoint, etc.

Illustrated using a professional drawing or presentation tool.

"I don't like the two-column layout for tools. Can we have them go across the top?"

Feedback: tweaks to the 'screen' or page as a whole. Incremental improvements.

Rough Sketch

Scanned from a hand-drawing, made with a drawing app and a tablet, or using the Napkin Look and Feel skin.

"Maybe the tools should be context-specific... Let's kill the toolbar and bring up only the tools that make sense at that moment..."

Feedback: higher-level features are questioned, bigger changes possible.

Storyboard or Use Case

The "story" of how the user might need or want to interact with the interface (app, book, product, etc.)

"We should NOT try to put a drawing feature in here... it's featuritis without a key benefit to most users."

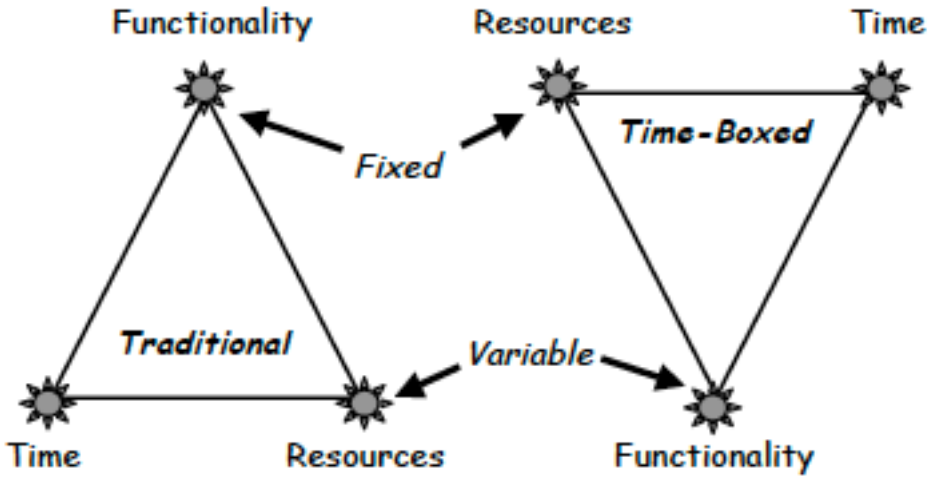
Feedback: big-picture ideas, possibility for revolutionary changes.

Agile methods

Agile methods – which are at the opposite end of the spectrum from waterfall software development – have received a LOT of attention in the computing press, and in conferences etc., over the last few years.

Agile methods are lightweight in the sense that they are intended to be less constraining than previous methods, and there's less emphasis on producing documentation. Changing requirements are seen as a normal and even a healthy feature, and it's regarded as acceptable – even desirable – to start writing some code early on, before the customer's requirements have been fully defined.

Agile methods are based on evolutionary prototyping, and the concept of Rapid Application Development (RAD). One of the key principles is time-boxing, which is the idea that the development time, rather than the specification, must be fixed…



To allow for this, individual requirements are often classified in some way – for example as…

- must have requirements,
- should have requirements,
- could have requirements.

Agile processes

- are essentially "fluid", "people-centric", "insight-led", "lower ceremony"
- are "lower documentation", but there is still a need for transparency and auditability
- suit organisations that have relatively flat management structures, as opposed to hierarchical ones, where the mode of working is predominantly team-based and networking-based – many businesses are moving in this direction
- go hand-in-hand with "shorter planning horizons" and time-boxing, and (therefore) a "priority driven approach";

Note that agile methods are not suitable for the development of safety-, mission- or business-critical systems, since these need to be engineered to a very high standard. For example, in some software systems as much as 50% of the code may be concerned with exception handling, yet exceptions (due to faulty inputs etc.) are often not considered at all during the construction of a prototype.

Extreme programming

Various different agile methods have been proposed. One of the most well known is Extreme Programming (XP); it was proposed in the late 1990's by Kent Beck.

XP is said to be applicable for small teams consisting of up to twenty developers (some authors quote a different limit). It is rather anarchic, and because of this it is probably very suitable for small, dynamic software companies where there is relatively little bureaucracy.

Adoption of XP means adhering to the principles of XP, which Bell lists as follows…

1. *re-plan frequently* – because whatever project plans are drawn up, they'll soon need adjusting!

2. *small releases* – give the customers frequent updates (e.g. one per fortnight) to build their confidence and to get their feedback

3. *adopt a metaphor* – the idea here is to create and share an easy-to-understand image/analogy/story for what the final system will be like, that is easily understood – for example: (a) a mortgage calculator might be likened to a web-based spreadsheet, (b) a payroll system might be likened to an assembly line

4. *maintain a simple design*

5. *refactor frequently* – the code should be continually refined; the aim is either to simplify the code (this is related to the previous point) or to add flexibility with a view to enabling future extension and/or re-use

6. *short code-test cycles* – i.e. test frequently, even after small amendments to the code

7. *pair programming* – code should be written by developers working in pairs, on one machine

8. *collective ownership* – all of the code should be considered to be "owned" by everyone, and therefore everyone should be entitled to edit any part of it

9. *continuous integration* – the code should be re-built (i.e. integrated and re-compiled) whenever anything has changed, so that everyone is always working with the most recent version of the code as far as possible

10. *involve the client* – if possible, have an end-user available to answer questions at any time

11. *coding standards* – everyone should follow a common coding standard

12. *avoid overwork* – XP is meant to be humane; the idea is that people work better when they're less stressed.

Clearly, XP can only be applicable for relatively small projects. Think about collective ownership for example; this would be impractical in a large software development project.

More on refactoring…

Martin Fowler, author of the book *Refactoring: Improving the Design of Existing Code*, and other books, defines refactoring as follows:

> "Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring."

More on pair programming…

Here's a nice quote about pair programming in a discussion forum on the web at
http://www.javalobby.org/java/forums/t12719.html

"One of the goals of any IT dept is to bring the proficiency of less capable members of a development team up. Pair programming does that. At first it may slow the better programmer up some but eventually the output of the pair goes up. Pair programming also helps build better teams and avoids exclusivity of knowledge of certain code. One of the biggest problems in IT depts is the "diva" programmer. The one programmer who throws fits because they know that they are sole owners of an important project. They have become unpunishable and unreplaceable because they have exclusive knowledge. Pair programming eliminates this. Nobody is irreplaceable. Finally it makes it easier to bring in new hires. All these are very important reasons to pursue a pair programming approach in your IT dept."

Process models and project management

A process model is a plan of action for developing software. It is an idealisation of the process of developing software. Given a new project, the first task facing the project manager is to draw up a work plan. If the project is similar to past projects this should be relatively straightforward, but if it's a novel project – which many software development projects are – then there's inevitably an element of guesswork involved.

The main problem is estimating the amount of effort that will be required for the various phases of the project. In some cases the overall budget will already have been determined according to someone's estimate of how much effort is likely to be required in total. Managers are notoriously over-optimistic about this, which means that many projects end up being delivered late and over budget.