Development Methodologies

Massimo Felici
Development Methodologies

• A methodology is a system of methods and principles used in a particular “school” of software design.

• There is a wide variety of published methodologies, and an even larger set of informal and/or company-specific methodologies. The most mature methodologies are often codified using specialist tools and techniques.

• All methodologies are controversial, because some people argue that any fixed methodology is an affront to a professional, creative, independent designer, while the rest argue about which methodology is best.
Example Methodologies

In this course we will focus on three main methodologies:

- The Waterfall Model
- The Unified Process (UP)
- Extreme Programming (XP)

But we may mention many others, such as Cleanroom, DSDM (Dynamic Systems Development Method), V-model, Scrum, Crystal, etc.! We will also discuss open-source design, which is more of a philosophical approach than a methodology like the others, but which has implications for methodology.
Types of Methodologies

- “Cowboy hacking” and micromanaging are at the extremes of a continuum.
- Basic distinction: agile vs. heavyweight — but it may be more useful to distinguish methods useful for small projects and teams vs those that are also useful for bigger projects and teams.
- Agile methods are more fashionable to discuss, but it is hard to tell what people are actually using.
Waterfall Model

- Inspired by older engineering disciplines, such as civil and mechanical (e.g. how cathedrals are built).

- Development of a release is broken into phases, each of which is completed and “signed-off” before moving on.

- When problems are found, must backtrack to a previous phase and start again with the sign-off procedures.

- Much time and effort is spent on getting early phases right, because all later phases depend on them.
Slide 4: References

Example: Waterfall Model of One Release

Waterfall Model

This is just one example — actual steps differ for every project!
Problems with Waterfall Model

• In practice it is rarely possible to go straight through from requirements to design to implementation, without backtracking.

• There is no feedback on how well the system works, and how well it solves users’ needs, until nearly the very end.

• Large danger of catastrophic failure:
  – Any error in key user requirements dooms entire process
  – Big chance that the design is not actually feasible
  – Big potential for unacceptable performance
Problems with Waterfall Model

- The waterfall model is simply a fundamentally flawed metaphor for software development.

- Design and debugging together account for nearly all of software development, with almost no construction step (just compilation!).

- This is a huge difference from electronic hardware design (where manufacturing and procurement typically dominate the process), or civil engineering (where construction dominates the process).
The Unified Process

Designed to be tailorable to all project sizes. Typically seen as heavyweight, but need not be.

- Component based
- Typically uses UML for all blueprints
- Use-case driven
- Architecture centric
- Iterative and incremental
Slide 8: References

We just give an overview; for details see:

Relatives of The Unified Process

The IBM Rational Unified Process (RUP) is a commercial product and toolset, superseding:

- The Objectory Process
- The Booch Method
- The Object Modeling Technique

The Unified Software Development Process (UP) is a published, non-proprietary method based on the RUP, but without specific commercial tools or proprietary methods.
Phases of UP Design

Each software release cycle proceeds through a series of phases, each of which can have multiple modelling iterations:

- **Inception**: Produces commitment to go ahead (business case feasibility and scope known)

- **Elaboration**: Produces basic architecture; plan of construction; significant risks identified; major risks addressed

- **Construction**: Produces beta-release system

- **Transition**: Introduces system to users
Each iteration *can* include all workflows, but some are more heavily weighted in different phases.
The Product: A Series of Models
How UP derisks large projects

- Architecture-centric, risk-management focused: elaboration phase chooses architecture, and identifies and mitigates risk, early and with a small team — if something proves impossible, you haven’t employed hundreds of programmers to no purpose.

- Its use of models (and other features we have not covered, like project roles) facilitates management of those huge projects.

- But what if a process did not need to work for huge projects?
Extreme Programming (XP)

- What if it were possible to make the cost of change constant across all stages, so that design and requirements can be changed even at late stages?

- XP tries to prevent backtracking by keeping the system continuously flexible, eliminating the need for determining the final correct requirements and design before implementation.

- XP started the trend toward “agile” processes (like Scrum and Crystal), focusing on closely knit, fast moving design/coding teams and practices.
Slide 14: References

XP is Controversial

An IBM Java poll on XP from www.xprogramming.com said roughly this:

- “I’ve tried it and loved it” (51%)
- “I’ve tried it and hated it” (8%)
- “It’s a good idea but it could never work” (25%)
- “It’s a bad idea – it could never work” (16%)

Of course, the UP might produce similar results...
How XP Imposes Control

• Through a set of “practices” to which designers adhere (using whatever other compatible methods and tools they prefer).

  See: http://www.extremeprogramming.org/rules.html

• Not strongly influenced by a particular design paradigm (unlike UP).

• Does require a strongly held (“extreme”) view of how to approach design. We consider some key practices in the following slides.
1. The Planning Process

- An XP project starts with a “Planning Game”.

- The “customer” defines the business value of desired “user stories”.

- The programmers provide cost estimates for implementing the user stories in appropriate combinations.

- No one is allowed to speculate about producing a total system which costs less than the sum of its parts.
User Stories vs. Use Cases

• A user story meets a similar need as a use case, but is textual, not graphical, and is something that any customer can do without training in UML.

• A user story deliberately does not include all the possible exceptions, variant pathways, etc. that go into use cases.

• Short example: “A bank customer goes up to an ATM and withdraws money from her account.”
2. On-site customer

- Someone who is knowledgeable about the business value of the system sits with the design team.

- This means there is always someone on hand to clarify the business purpose, help write realistic tests, and make small scale priority decisions.

- The customer acts as a continuously available source of corrections and additions to the requirements.
3. Small Releases

- Put a simple system into production early, implementing a few important user stories.

- Re-release it as frequently as possible while adding significant business value (a set of important user stories) in each release.
  
  E.g., aim for monthly rather than annual release cycles.

- The aim is to get feedback as soon as possible.
4. Continuous Testing

- Write the tests before writing the software.
- Customers provide acceptance tests.
- Continuously validate all code against the tests.
- Tests act as system specification.
5. Simple Design

- Do the simplest thing that could possibly work.

- Do not design for tomorrow — you might not need it.

- Extra complexity added ‘just in case’ will fossilise your design (e.g. your class hierarchies) and get into the way of the changes you will need to make tomorrow.
6. Refactoring

- When tomorrow arrives, there will be a few cases where you actually have to change the early simple design to a more complicated one.

- Change cannot occur only through small, scattered changes, because over time such changes will gradually turn the design into spaghetti.

- To keep the design modifiable at all stages, XP relies on continuous refactoring: improving the design without adding functionality.
7. Collective Ownership

- Anyone is allowed to change anyone else’s code modules, without permission, if he or she believes that this would improve the overall system.

- To avoid chaos, collective ownership requires a good revision control (configuration management) tool, but those are now widely available.
8. Coding Standard

- Since XP requires collective ownership (anyone can adapt anyone else's code) the conventions for writing code must be uniform across the project.

- This requires a single coding standard to which everyone adheres.
9. Continuous Integration

- Integration and full-test-suite validation happens no more than a day after code is written.

- This means that individual teams don’t accumulate a library of possibly relevant but obscure code.

- Moreover, it enables everyone to freely modify code at any time, because they know that they have access to the latest design.
10. Pair Programming

• All code is written by **a pair** of people at one machine.
  
  – One partner is doing the coding
  – The other is considering strategy (Is the approach going to work? What other test cases might we need? Could we simplify the problem so we do not have to do this? Etc.)

• This is unpalatable to some but appears vital to the XP method, because it helps make collective code ownership work.
11. 40-Hour week

- XP is intense so it is necessary to prevent “burnout”.

- Designers are discouraged from working more than 40 hours per week (which is low compared to the rest of the software world!).

- If it is essential to work harder in one week then the following week should drop back to normal (or less).
Limitations of XP

Not suitable for all projects (but nobody claimed it was) — e.g. needs

- Small, geographically co-located team, willing to embrace the XP practices
- Customer willing and able to work the way XP envisages
- Sufficiently flexible environment (e.g., using XP to develop software for a space mission probably loses much of the benefit)
Limitations of XP

- XP is deliberately ‘low ceremony’ — will not normally produce documents describing its architecture, for example.

- Needs special care to ‘wrap up’ properly when the team is disbanded — all crucial information must be visible in the code, tests or other durable deliverables.

- If this is not arranged, support and maintenance of an XP-developed product can be problematic.
Summary

- Methodologies: principled ways to manage large projects

- Waterfall model works in other disciplines, where most of the work is on the physical implementation, but in SE nearly all work is conceptual

- Unified Process constructs gradually more elaborate models to uncover risks and solidify architecture as early as possible

- Extreme Programming relies on continuous customer involvement, testing, and refactoring to deliver code early and continuously, minimising risk of complete failure.

- We have done overviews only – you will need to read more to actually implement any process!
Readings

Required Readings


Suggested Readings