Methodologies

Software Architecture, Process, and Management

In this lecture I will detail the main axis upon which we categorise software development methodologies and the merits of each kind.

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:
  1. The Waterfall Model (discussed in many courses)
  2. The Unified Process (UP) (partly covered in CS2)
  3. Extreme Programming (XP)
- But there are of course lots of others:
  - Cleanroom, DSDM, V-model, Scrum, Crystal

  to name a few
- The relationships between them are complex as well
- 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

Searching for a Solution

- We can think of "building" as more of a search problem in which we search for the correct solution
- We may search by heading off in some chosen direction
- Backtracking occurs when we realise we have gone of in the wrong direction
- Heavyweight methodologies focus on making as correct a choice of direction as possible to avoid backtracking
- Agile methodologies focus on assuring that changing direction is as painless as possible thus avoiding actual backtracking
- There are other alternatives:
  - Going off in several directions at once
  - Sending a scout who has to come back and report, also known as prototyping

Waterfall

Plan Driven Model: Waterfall

- Inspired by older engineering disciplines, such as civil and mechanical, for example, 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

Waterfall

Problems with the 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

The Unified Process

Relatives of the Unified Process

- The IBM Rational Unified Process is a commercial product and toolset, superseding:
  - The Objectory Process
  - The Booch Method
  - The Object Modeling Technique
- The relationship is that these methods inspired the design of UML which in turn drove the creation of the Unified Process
- The Unified Software Development Process is a published, non-proprietary method based on the Rational Unified Process, but without specific commercial tools or proprietary methods

The Unified Process

- Iterative modification of waterfall model using modeling to forestall backtracking:
  - Iterative and incremental
  - Uses UML for all blueprints
  - Use-case driven
  - Architecture centric
  - Component based
- Still a somewhat heavyweight approach
- We just give an overview; for more details see:
  - Jacobson, I., Booch, G., \ Rumbaugh, J. (1998). The Unified Software Development Process. Reading, MA: Addison-Wesley.

The Unified Process

Phases

- Each software release cycle proceeds through a series of phases, each of which can have multiple modeling iterations:
  1. Inception: Produces commitment to go ahead, business case feasibility and scope known
  2. Elaboration: Produces basic architecture; plan of construction; significant risks identified; major risks addressed
  3. Construction: Produces beta-release system
  4. Transition: Introduces system to users

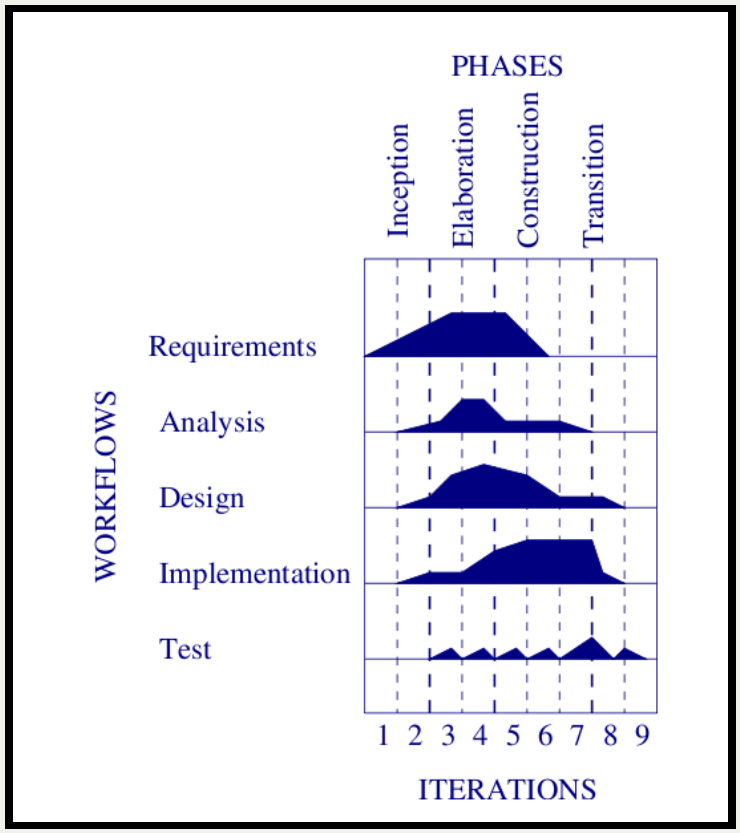
The Unified Process

Workflows

- The workflows of The Unified Process resemble those of the entire process under the Waterfall methodology:
  1. Requirements
  2. Analysis
  3. Design
  4. Implementation
  5. Test

SAPM: The Unified Process

Waterfall Iterations within Unified Process Phases



- Each phase can have multiple iterations - project proceeds top-to-bottom and then left-to-right
- Each iteration *may* include all workflows but some are more heavily weighted in different phases
- Still relatively hard to **change** requirements once implementation is under way

The Unified Process

Use Cases and the Unified Process

- The unified process is driven by *use cases*
- *"A use case specifies a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor."*
- These drive:
  - Requirements capture
  - Analysis and design of how system realises use cases
  - Planning of development tasks
  - Acceptance/system testing
  - Traceability of design decisions back to use cases

The Unified Process

Assumption of Unified Process

- Unified Process, and other heavyweight methodologies, concentrate on carefully controlled, up-front, well-documented thinking
- Based on the assumption that the cost of making changes rises significantly through the development stages
- To minimise backtracking, The Unified Process establishes rigorous control over each stage
- At each stage of The Unified Process, a model acts as a proxy for the whole system, helping to bring out problems as early as possible before they are set in code

Unified Process

UML Problems

- The size, it contains many diagrams and constructs which are redundant or infrequently used, but at least such standards are all in one place
  - Can be difficult to learn
- Complaints of ambiguity **and** inconsistency
- Intersection of UML and implementation language may influence the implementation, this is true of any standard notation language
- Model interchange format, since UML is a graphical notation you still require some standard to promote exchange of models between two software products

The Unified Process

Problems with Unified Process

- Heavy training, documentation, and tools requirements - learning and using UML, modeling, process, tools, techniques
- UML is not a native language for customers, and so it can be hard for them to provide good feedback until system is implemented
- Requirements are very difficult to change at late stages, if needed to match changes in business world, address new competition, or fix mistakes in requirements capture

Observation

- Although The Unified Process has, and continues to, enjoy some success, it was observed that a common reason for project failure was changes in the requirements
- Heavyweight processes try to minimise the risk of this by working as hard as possible to correctly capture the requirements in the first place
- This may well work for some kinds of projects, but some have requirements change due to external factors outwith the control of the development team, or even the customer
- Examples:
  - Apple computers become greatly more popular and suddenly having both an Apple and Windows version is important
  - The law is changed

Agile Methodologies

- 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?
- Agile methodologies try to prevent backtracking by keeping the system continuously flexible, eliminating the need for determining the final correct requirements and design before implementation
- In other words, agile methodologies challenge the assumption that making late changes is necessarily expensive

Extreme Programming (XP)

- We will look at Extreme Programming (XP) as an example of an *"agile"* methodology
- XP started the trend toward *"agile"* processes, such as *Scrum* and *Crystal*, focusing on closely knit, fast moving design/coding teams and practices
- see Beck, K. (1999). Extreme Programming Explained. Addison-Wesley.

Extreme Programming

How Extreme Programming Imposes Control

- Through a set of "practices" to which designers adhere
  - using whatever other compatible methods and tools they prefer
  - See: **the rules here**
- In contrast to The Unified Process, XP is deliberately not strongly influenced by a particular design paradigm
- Does require a strongly held ("extreme") view of how to approach design
- We consider some key practices in the following slides

Extreme Programming

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
  - Even though this is of course *plausible*

Extreme Programming

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.
- Short example: *"A bank customer goes up to an ATM and withdraws money from her account."*

Extreme Programming

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
- Best case scenario: developers are their own customers
  - Development tool creators: JetBrains, github, Visual Studio
  - Tools for general population, gmail, facebook, iOS etc
  - Known as *"eating your own dog food"*

Extreme Programming

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 in each release
  - Significant business value will usually equate to a set of important user stories
- For example: aim for monthly rather than annual release cycles
- The aim is to get feedback as soon as possible
- The difficulty in fixing a particular **issue** is proportional to the time between the developer developing the **issue** containing code and the time when it is reported and worked upon

Extreme Programming

4. Continuous Testing

- Write the tests before writing the software
- Continuously validate all code against the tests
- Tests act as system specification

Extreme Programming

4. Continuous Testing

- Advantages
  - Aids in refactoring
  - Version control can be used to indicate the changes to code which caused a test to go from green to red
  - Tests become an executable specification of the application
  - Can be useful for debugging legacy code
- Disdvantages
  - Tests are yet more code to maintain
  - Changing requirements may require changes to tests
  - Tests written by the same developer as the code may share the same blind spots

Extreme Programming

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
  - For example your class hierarchies get in the way of the changes you will need to make tomorrow

Extreme Programming

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
  - Refactoring *may* fix bugs, but the intention is that the behaviour is exactly the same, and bugs fixed are only done so as an unintended consequence

Extreme Programming

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
- This also helps to keep each developer's *"bus factor"* low

Extreme Programming

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

Extreme Programming

9. Continuous Integration

- Integration and full-test-suite validation happens no more than a day after code is written
- This means that individual teams do not 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
- *Even more* extreme is the idea of *Continuous Deployment*, this has advocates but even those generally agree it is not suitable for **all** projects

Extreme Programming

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?
- Perhaps the most polarising of the specified XP practices
- Pair programming unpalatable to some but appears vital to the XP method, because it helps make collective code ownership work
- Many also say it is a very enjoyable way to code

Extreme Programming

11. 40 Hour Work 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)

Extreme Programming

Problems with XP

- Published interfaces (e.g. APIs): some code is not practical to refactor, because not all uses can be known, so that code must anticipate all reasonable tomorrows
- Many programmers resist pair programming or other XP guidelines; teams are often spread geographically
  - You may have/covet a corner desk but they are awful for even showing someone your work much less pair programming
- The customer is not always available or willing, and may not be able to agree to an open-ended process
- Over time XP has become more heavyweight, trying to incorporate new realisations, just as Unified Process did

Extreme Programming

Problems with XP

- XP is deliberately lsquo;low ceremony’ and will therefore not produce as much documentation as a matter of course:
  - For example the architecture may not be documented rigorously
- Needs special care to lsquo;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, long-term support and maintenance of an XP-developed product can be problematic
- Additionally you may fail to produce data for future projects

Extreme Programming

The rules

- These are rules or guidelines standard for a project using what it calls an XP methodology
- Of course project managers are free to pick and choose which of those rules to use and which to leave out
- It is important that the project manager makes clear what the rules and guidelines are

Extreme Programming Offshoots

- Extreme programming has fostered many development practices some of which have matured into consideration as a methodology in their own right
- Many choose aspects of XP and focus on those and others add their own related rules:
  - For example test-driven development focuses on the test-first principle

Agile Methodologies and the Web

- With the growth in popularity of the web, agile methodologies have become more popular/relevant
- Some factors I believe have contributed to this:
  - Web application deployment is easier, no CDs etc
  - You can ensure no-one is using an old version
    - A new version is downloaded for each visit and/or
    - The running code is on the server
  - Feedback from users is easier to gather
    - Sometimes even automatically, for example *"Does anyone ever click here/use this feature?"*
  - Easier deployment has meant greater competition and more frequent changes to requirements
  - Security fixes must often be released with extreme haste