Software Development Process

Giuseppe Lipari
`http://retis.sssup.it`

Scuola Superiore Sant'Anna – Pisa

April 27, 2009

Acknoledgements

These slides are an extract of Alex Liu's slides:

Alex X. Liu
Assistant Professor
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824-1266
`http://www.cse.msu.edu/~alexliu/`

CSE 335 Software Engineering Course
`http://www.cse.msu.edu/~alexliu/courses/335Fall2008/`

Programming vs. Engineering

Programming

- Small project
- You
- Build what you want
- One product
- Few sequential changes
- Short-lived
- Cheap
- Small consequences

Engineering

- Huge project
- Teams
- Build what they want
- Family of products
- Many Parallel changes
- Long-lived
- Costly
- Large Consequences

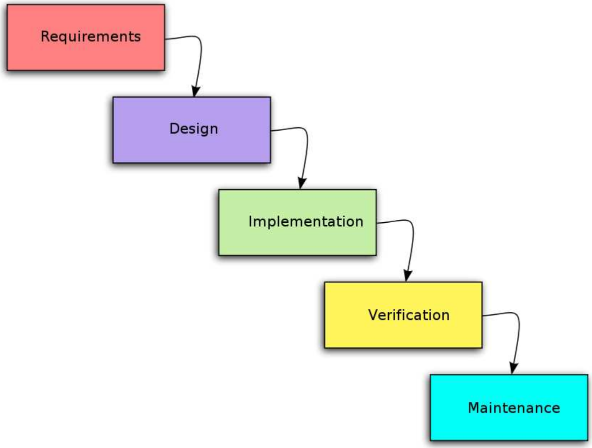
Software Engineering

Software engineering is a special type of engineering

- Software is extremely complex
- Software construction is human-intensive
- Software is intangible and invisible
- Software is constantly subject to pressures for change
- Software needs to conform to arbitrary interfaces and contexts
  - E.g., business rules and processes vary dramatically from business to business
  - E.g., existing databases of information.

Software development process

The waterfall model is the oldest and most common Software Development Process



Requirements

- Problem Definition $\rightarrow$ Requirements Specification
  - determine exactly what the customer and user want
  - develop a contract with the customer
  - specifies what the software product is suppose to do
- Difficulties
  - client asks for wrong product
  - client is computer/software illiterate
  - specifications are ambiguous, inconsistent, incomplete

Architecture/Design

- Requirements Specification & Architecture/Design
  - architecture: decompose software into modules with interfaces
  - design: develop module specifications (algorithms, data types)
  - maintain a record of design decisions and traceability
  - specifies how the software product is to do its tasks
- Difficulties
  - miscommunication between module designers
  - design may be inconsistent, incomplete, ambiguous
- Architecture vs. Design
  - Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design.
  - Design is concerned with the modularization and detailed interfaces of the design elements, their algorithms and procedures, and the data types needed to support the architecture and to satisfy the requirements.

Implementation & Integration

- Design $\longrightarrow$ Implementation
  - implement modules; verify that they meet their specifications
  - combine modules according to the design
  - specifies how the software product does its tasks
- Difficulties
  - module interaction errors
  - order of integration may influence quality and productivity

Verification & Validation

- Analysis
  - Static
  - “Science”
  - Formal verification
  - Informal reviews and walkthroughs
- Testing
  - Dynamic
  - “Engineering”
  - White box vs. black box
  - Structural vs. behavioral
  - Issues of test adequacy

Deployment & Maintenance

- Operation $\rightarrow$ Change
  - maintain software during/after user operation
  - determine whether the product still functions correctly
- Difficulties
  - lack of documentation
  - personnel turnover

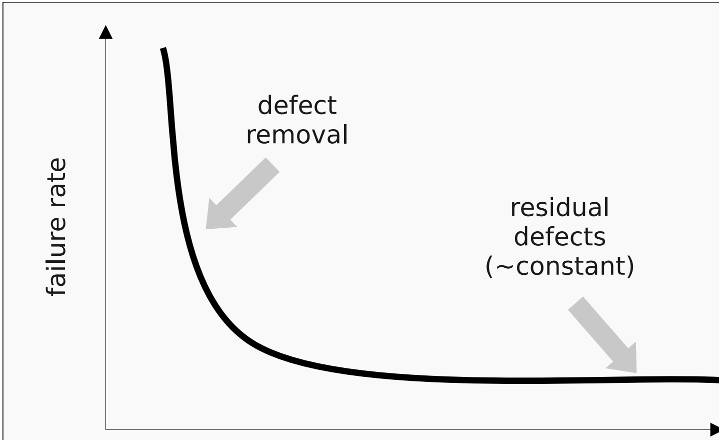
Economic and Management Aspects

- Software production = development + maintenance (evolution)
- Maintenance costs $>$ 60% of all development costs
- Quicker development is not always preferable
    - higher up-front costs may defray downstream costs
    - poorly designed/implemented software is a critical cost factor

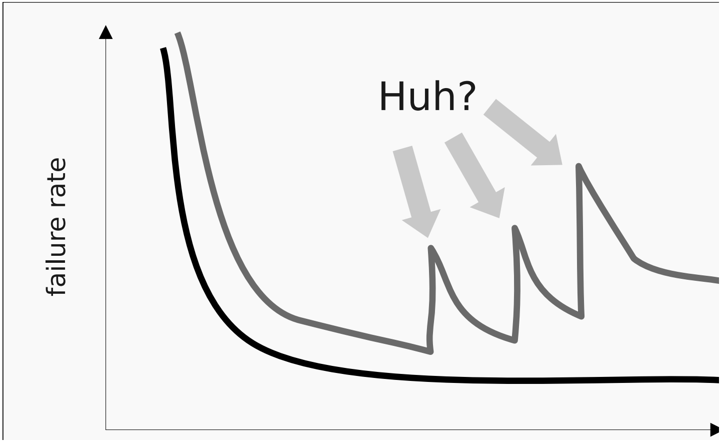
Cost of producing software

- The cost is low during the first phases
- It increases as the development progresses
  - making a change at a later phase may be very costly
  - for example, changing a design decision is critical at the testing phases
  - it is important to not be forced to change major portions of code during testing

Ideal Software Failure Curve



Actual Software Failure Curve



Failures

- The failures are due to changes in the software implementation
- Maintainance means
    - Fix bugs this means write new code that can introduce other bugs!
    - Add new features again, this may introduce bugs
    - Modify/improve existing parts again, this may introduce bugs
- The problem is when you want to introduce new features that do not fit well with the existing design!

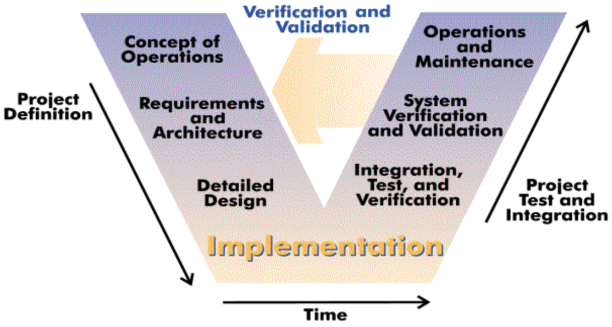
Problems of the Waterfall model

- The waterfall model is also called Big Design Up Front
- The waterfall model is argued by many to be a bad idea in practice
  - too difficult to “get it right” in one single lifecycle without ever going back
  - For example, clients may not be aware of exactly what requirements they want before they see a working prototype and can comment upon it
  - Designers may not be aware of future implementation difficulties when writing a design for an unimplemented software product.
- David Parnas, in “A Rational Design Process: How and Why to Fake It”, writes:

> *Many of the [system’s] details only become known to us as we progress in the [system’s] implementation. Some of the things that we learn invalidate our design and we must backtrack.*

V-model

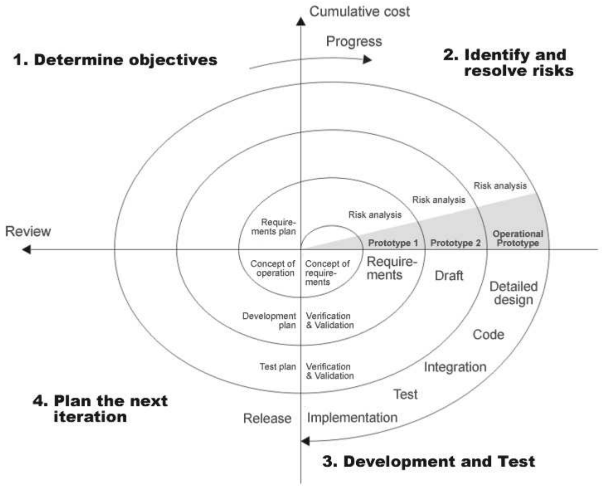
- The V-model (mostly used in industrial software development) is similar to the waterfall model



Verification and Validation
Concept of Operations
Operations and Maintenance
Project Definition
Requirements and Architecture
System Verification and Validation
Detailed Design
Integration, Test, and Verification
Project Test and Integration
Implementation
Time

Other models

- The spiral model tries to improve over the waterfall model by introducing iterations



Design

- The architecture and design activities are crucial to the success of the software
  - A wrong design choice would cause lot of problems in the implementation
  - A design too much focused on the current requirements may preclude future extensions
  - Bad design is very costly!

Software Engineering principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Rigor and formality

- Software development is a creative design process.
  - But creativity implies informality, imprecision, and inaccuracy
- Rigor and formality are necessary:
  - to improve quality and assurance of creative results
  - to ensure accuracy in defining and understanding problems
- Rigor and formality influence reliability and verifiability
- Evident in:
  - Design notations, requirements, specifications, process definitions

Separation of concerns

- We cannot deal with all aspects of a problem simultaneously
- To conquer complexity, we need to separate issues and tasks
  - Separate functionality from efficiency
  - Separate requirements specification from design
  - Separate responsibilities
- Divide conquer
- Today's applications involve interoperability of
  - Client/Server, Legacy system, COTS, databases, etc.
  - Multiple programming languages (C, C++, Java, etc.)
  - Heterogeneous hardware/OS platforms
- Separation of Concerns is Critical!

Modularity

- A complex system may be divided into simpler pieces called modules
- A system that is composed of modules is called module
- Supports separation of concerns
  - when dealing with a module we can ignore details of other modules
- Three goals with modularity
  - Decomposability: break problem into small sub-problems (divide & conquer)
  - Composability: construct solution from sub-solutions
  - Understandability: understand system by understanding sub-systems
- Two essential properties
  - Cohesion: degree to which parts of a module are related (within a module)
  - Coupling: amount of interdependence between modules (among modules)

Abstraction

- Identify the important aspects and ignore the details
- Supports separation of concerns
  - Divide & conquer vertically (Modularity: divide & conquer horizontally)
- Abstractions dominate computing
  - Design Models (ER, UML etc)
  - Programming Languages (C, C++, Java, etc.)

Anticipation of change

- Software changes and evolves throughout all stages from specification onward
  - Changes are inevitable
  - Requirement changes
  - Programmer changes
  - Technology changes
- Anticipation of change supports software evolution
  - Separation of concerns
  - Modularization: encapsulate areas for potential changes

Generality

- When given a specific problem, try to discover if it is an instance of a more general problem whose solution can be reused in other cases
  - Supermarket System vs. Inventory Control
  - Hospital Application vs. Health Care Product
  - C++ template (link list of integers, link list of floats, . . .)
- Reuse: adapt general solution to specific problem
  - Inventory Control for Supermarket, Auto Parts, Video Tape Rental, etc.
  - Health Care Product for Hospital, MD Office, Dental Office, etc.
- Additional short-term effort vs. long-term gains (maintenance, reuse, . . .)

Incrementality

- Move towards the goal in a stepwise fashion (increments)
- Software should be built incrementally
  - Identify useful subsets of an application and deliver in increments
  - Deliver subsets of a system early to get early feedback
  - Focused, less errors in smaller increments
  - Phased prototypes with increasing functions
- Separation of concerns (in terms of functionality)

Just learning the principles is not enough

- You need to learn Design Patterns of other masters
  - Example:
    - Composite pattern,
    - Visitor pattern,
    - Abstract factory pattern
    - Builder pattern
    - Adaptor pattern
    - Observer pattern
    - Mediator pattern
    - Template method
    - and more . . .
- At the end of the course, you should be able to apply these principles with proficiency in real design contexts

Example of bad design

```
class Employee {
public:
    string firstName;
    string lastName;
    Date hiring_date;
    short department;
};
```

```
class Manager {
public:
    Employee emp;
    list<Employee*> group;
    short level;
};
```

Example of bad design

```
class Employee {
public:
    string firstName;
    string lastName;
    Date hiring_date;
    short department;
};
```

```
class Manager {
public:
    Employee emp;
    list<Employee*> group;
    short level;
};
```

- Why this design is bad?
    - Inconsistent with domain knowledge: a manager is always an employee (Domain knowledge is stable over time.)
    - A manager may have another manager in their group
    - What if you want to add another role “vice president” above manager?
    - What if you want to print out the name of every employee?

Improving the previous design

```
class Employee {
public:
    string firstName;
    string lastName;
    Date hiring_date;
    short department;
};
```

```
class Manager: public Employee {
public:
    list<Employee*> group;
    short level;
};
```