Introduction – Object Oriented Software Design

Alex X. Liu
alexliu@cse.msu.edu
http://www.cse.msu.edu/~alexliu
2132 Engineering Building
Department of Computer Science and Engineering
Michigan State University
Phone: 517-353-5152

Reasons to take CSE 335

- Many reasons to take CSE 335
  - E.g., it is required!
- Help you to become a master programmer and designer
  - From writing small programs that simply works to design large systems
- Help you to succeed in job interviews
  - e.g, Google.
- You are very fortunate!
  - Very few computer science departments offer a course on design patterns

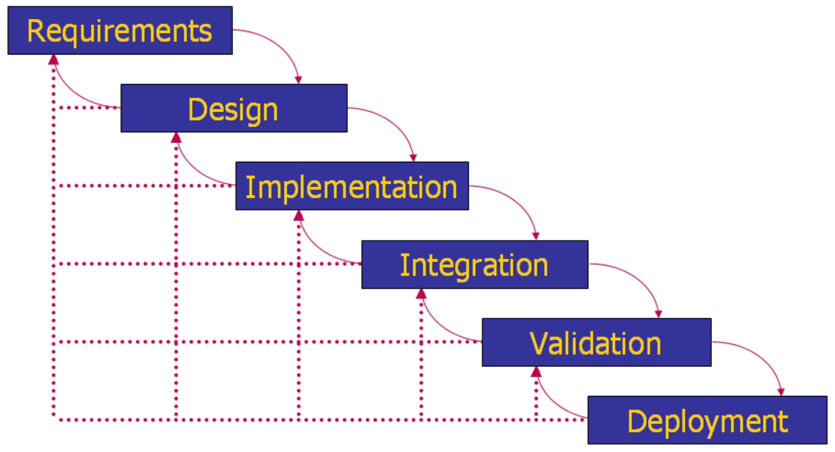
Science vs. Engineering

- Science is on discovery. Engineering is on design:
  - The application of scientific principles and methods to the construction of useful structures & machines
- 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 pressure for change
  - Software needs to conform to different interfaces and contexts
    - E.g., business rules and processes vary dramatically from business to business
    - E.g., existing databases of information.

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 Development Lifecycle: Waterfall Model



Requirements

- Problem Definition → 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 → 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

Verification and Validation

- Verification - Analysis
  - Static
  - "Science"
  - Formal verification
- Validation - Testing
  - Dynamic
  - "Engineering"
  - White box vs. black box
  - Structural vs. behavioral
  - Issues of test adequacy

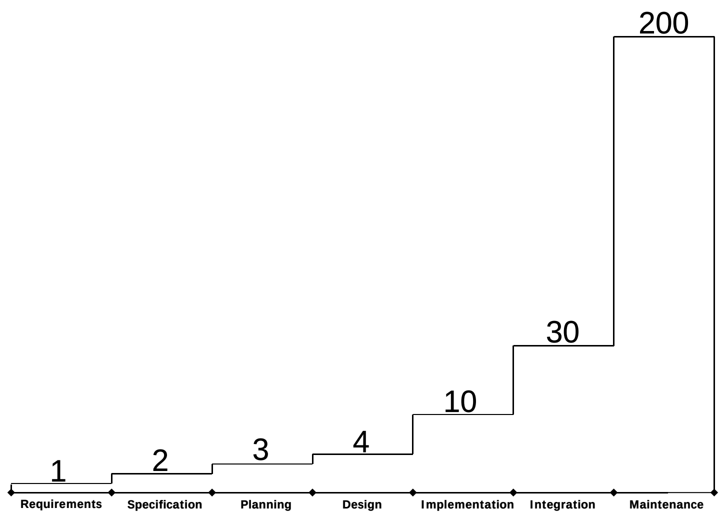
Deployment & Evolution

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

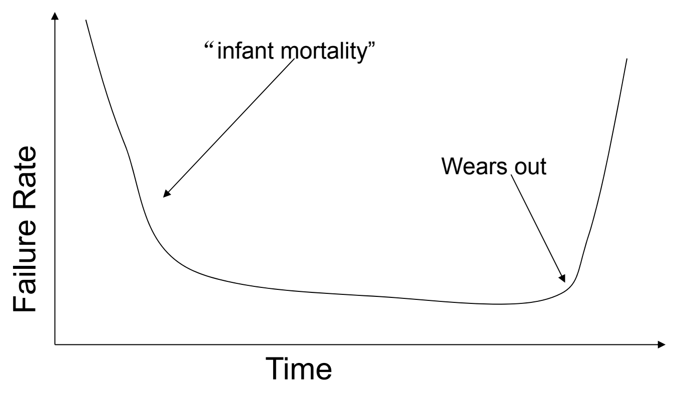
Economic and Management Aspects

- 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

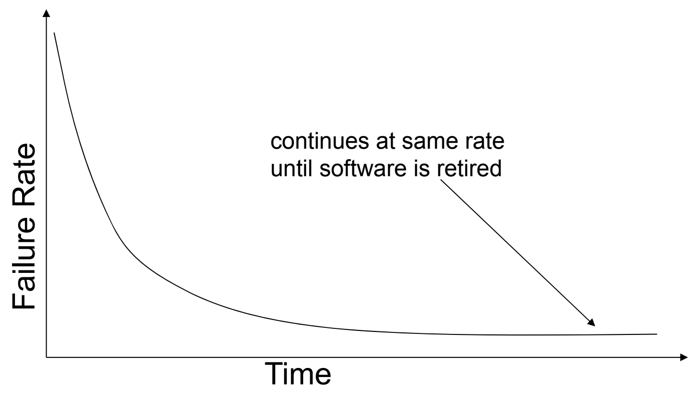
Relative Costs of Fixing Software Faults



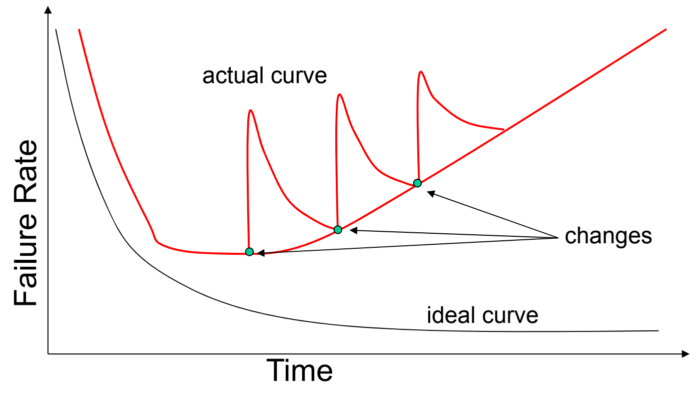
Hardware Failure Curve



Ideal Software Failure Curve



Actual Software Failure Curve



How to become a software design master?

Learning to develop good software

is similar to

learning to play good chess

To become a chess master

- First, learn the **rules**
  - e.g., names of pieces, legal movements, captures, board geometry, etc.
- Second, learn the **principles**
  - e.g., relative value of certain pieces, power of a threat, etc.
  - But principles are abstract. How to apply them in practice?
- Third, learn the **patterns** by studying games of other masters
  - These games have certain patterns that must be understood, memorized, and applied repeatedly until they become the second nature.

To become a software design master

- First, learn the **rules**
  - e.g., programming languages, data structures, etc.
- Second, learn the **principles**
  - e.g., software engineering principles such as separation of concerns, etc.
  - But principles are abstract. How to apply them in practice?
- Third, learn the **patterns** by studying designs of other masters
  - These designs have certain patterns that must be understood, memorized, and applied repeatedly until they become second nature.

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 help to improve reliability and verifiability
- Evident in:
  - Design notations, requirements, specifications, process definitions

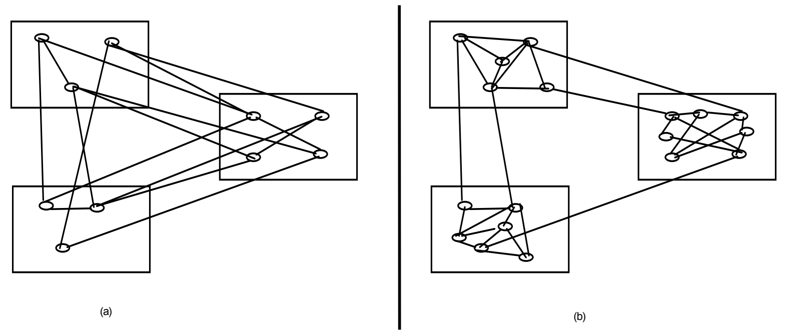
Separation of concerns

- 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 (Commercial Off-The-Shelf), 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 *modular*
- 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)

A Visual Representation

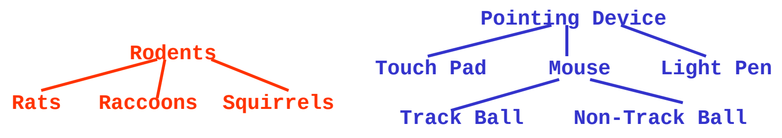


high coupling

low coupling

Abstraction

- Identify the important aspects and ignore the details
- Abstraction 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.)
- Example:



Anticipation of change

- Software changes and evolves throughout all stages from specification onward
  - Changes are inevitable
  - Requirement changes
  - Programmer changes
  - Technology changes
- This affects all aspects of software engineering
  - Make sure all artifacts are easy to change
    - Modularization and separation of concerns
  - Make sure you can maintain many versions artifacts
  - Plan for personnel turnover
  - Plan for a rapidly changing market
  - Plan for rapidly changing technology
- Anticipation of change supports software evolution
  - Modularization: encapsulate areas for potential changes
  - Separation of concerns

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, convenient stores, 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 principles is not enough

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

Exercise

- Design two classes: Employee and Manager.
  - An employee has the attributes of first name, last name, hiring date, and department ID.
  - A manager manages a group of employees.
  - A manager is also an employee, but a manager also has an attribute of managing level.
  - You only need to specify the data members of each class. You do not need to specify any member function for each class.

Criticize this Design

```
class Employee{
protected:
        string  m_FirstName;
        string  m_LastName;
        string m_DepartmentName;
        unsigned int m_HiringYear;
};
```

```
class Manager{
public:
        Employee           m_Employee;;
        vector<Employee*> m_Group;
        unsigned short    m_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?

Example of Good Design

```
class Employee{
protected:
        string  m_FirstName;
        string  m_LastName;
        string m_DepartmentName;
        unsigned int m_HiringYear;
};
```

```
class Manager: public Employee{
public:
        vector<Employee*> m_Group;
        unsigned short         m_Level;
};
```

- Recall the “abstraction” principle
- Recall the “separation of concerns” principle
- Recall the “anticipation of change” principle
- Recall the “generality” principle
- Recall the “incrementality” principle
- Recall the “modularity” principle