CS223 Introduction to Software Engineering

Doron Peled, Roger Packwood,
Arshad Jhumka, Ananda Amatya,
Mike Joy

The Software Process

- Why is there a Software Process ?
  - why not just write the program ?
- What the customer wants
  - how it is implemented, or at least designed
  - change, for the better, sometimes ...
- Why is software "engineering" hard ?
  - what solutions does "engineering" offer ?
- The traditional software lifecycle
  - other development models

Software Specification

- "What" needs to be specified
  - many people communicate via written documents

- A specification that the customer can agree to
  - a specification for the programmer (one of many)

- Many aspects (views) of a software design

- Complete, Concise, Testable

Software Cost Estimation

- Before we even start, do we want the job ?
- Need to estimate -
  - how much effort
  - how much time
  - how much money
- Primarily based on how much last time
  - model based
- Constructive Cost Model COCOMO
- Mythical Man Month - Brooks

Safety-Critical Systems

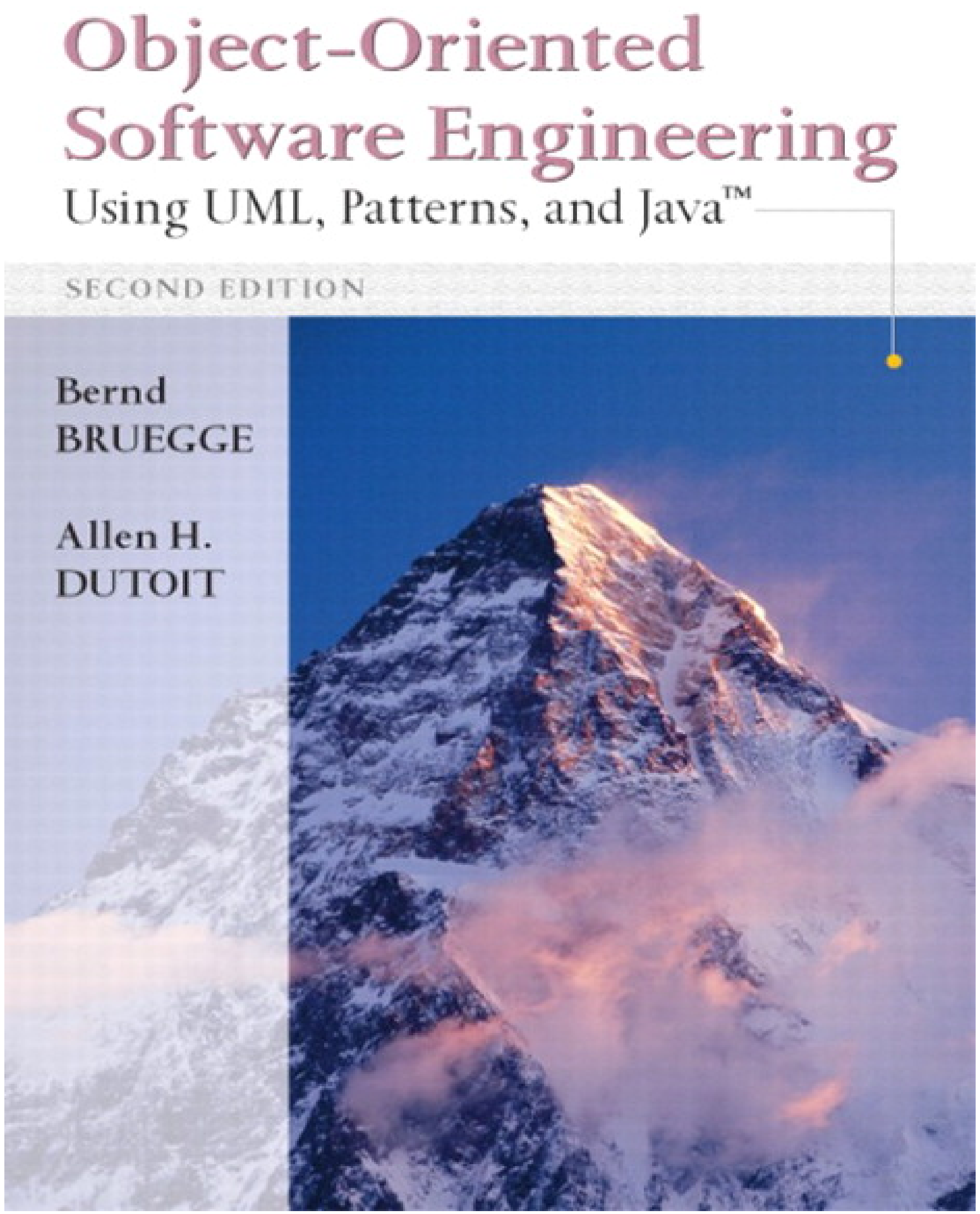
- Developing software that should never compromise the overall safety of a system
- Reliability is with respect to specification
  - safety is independent of specification
- Therac and Arianne examples
- Risk analysis
  - intolerable
  - As Low As Reasonably Practical (ALARP)
  - acceptable
- The Myths of Software Safety

Software Testing

- A successful test finds a fault
  - testing does not prove the absence of faults
- White Box, Black Box testing
- Coverage testing, Exhaustive testing
  - can you trust the test software?
- Test data values, Corner Cases, Fencepost errors
  - mistyped variable names, operators, constructs
- Unit test, integration test, System, Alpha, Beta
  - top-down, Bottom-up, Inside-out, Sandwich ??

Software Reliability

- Availability, Reliability, Safety, Integrity, etc.
- Defects, Density and Zero
- Fault Tolerance, N-Way, Recovery Blocks, Diversity
- Dangerous Programming



- Bernd Bruegge, *Adjunct, Carnegie Mellon University*
  Allen H. Dutoit, *Technical University of Munich*
- ISBN: 0-13-191179-1
- Publisher: Prentice Hall
  Copyright: 2004
  Paperback 762 pages (November 30, 2003)
- Amazon.co.uk Price: £35.99
- Warwick Bookshop: In stock

Factors affecting the quality of a software system

- **Complexity:**
  - System too complex for a single programmer to comprehend;
  - Fixing one bug introduces another bug.

- **Change:**
  - *Entropy* of a software system increases with each change:
    - Change in a system alters its structure
    - Change in structure makes the next change more difficult.
  - *Cost* of subsequent changes increase rapidly:
    - Whatever the system's application domain or technological base.

Dealing with Complexity

- Abstraction
- Decomposition
- Hierarchy

Abstraction

- Inherent human limitation to deal with *complexity*
  - The 7 +- 2 phenomena

- Chunking:
  - Group collection of *objects*

- Ignore unessential details:
  - *Models*

Models are used to provide abstractions

- **System Model:**
  - *Object Model*: system structure; object interaction
  - *Functional model*: system functions; data flow through the system
  - *Dynamic model*: system reaction to external events; event flow
- **Task Model:**
  - *PERT Chart*: dependencies between the tasks
  - *Schedule*: time limit
  - *Org Chart*: roles in the project or organization
- **Issues Model:**
  - Open and closed issues;
  - constraints posed by the client;
  - resolutions made.

Decomposition

- A technique used to **master complexity** (*divide and conquer*)
- **Functional decomposition**
  - system decomposed into *modules*
  - *Module*: a major processing step (function) in the application domain
  - Modules can be decomposed into *smaller modules*
- **Object-oriented decomposition**
  - The system is decomposed into *classes* (*objects*)
  - Each *class* is a major *abstraction* in the application domain
  - *Classes* can be decomposed into *smaller classes*

Which decomposition is the right one?

Class Identification

- **Object-oriented modelling requires Class identification:**
  - Finding *classes* for a new software system *(Greenfield Engineering)*
  - Identifying *classes* in an existing system (*Reengineering*)
  - *C*reating class-based *interface* to a system (*Interface Engineering*)

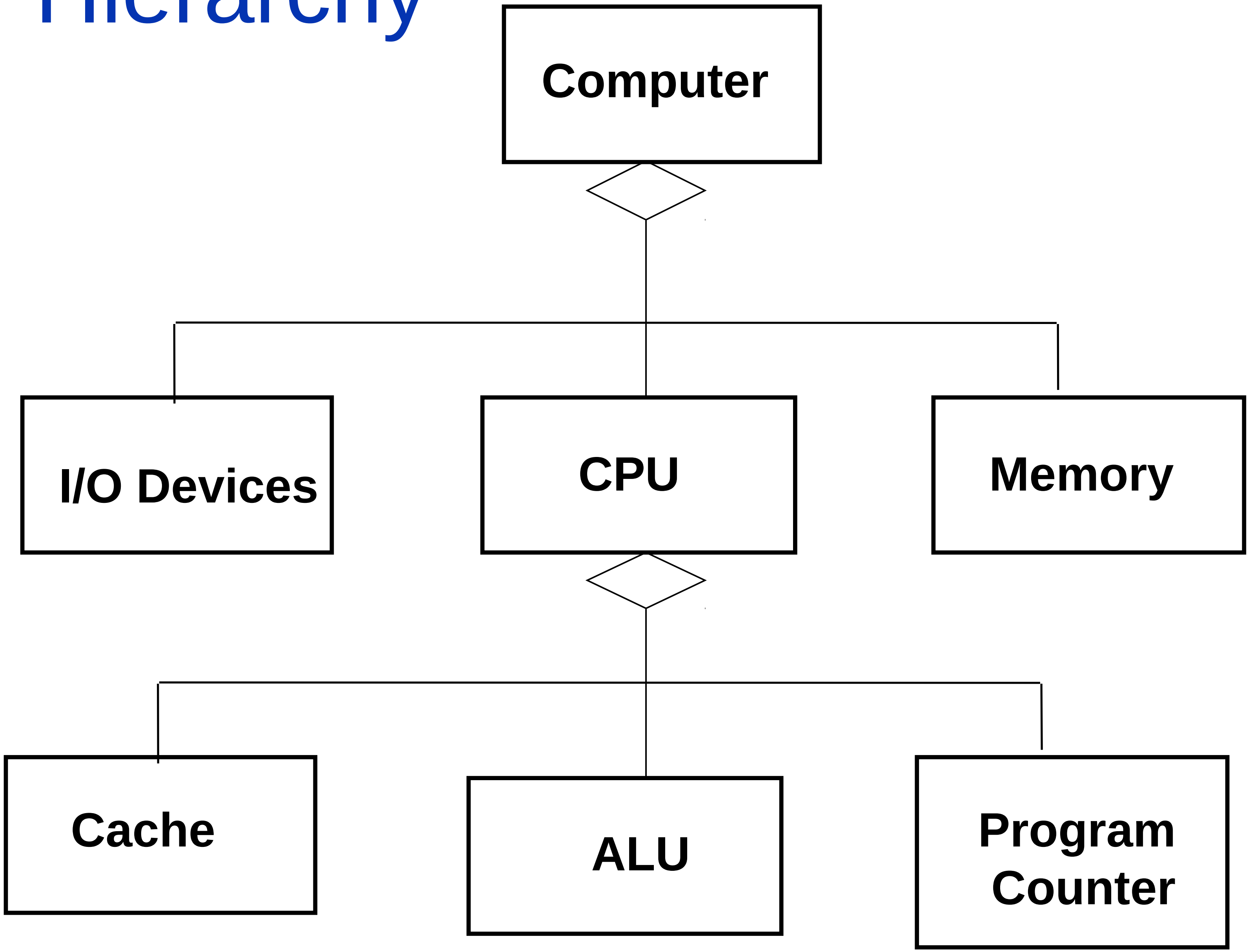
- **Class identification uses:**
  - Philosophy
  - Science
  - Experimental evidence

- **Difficulty in identifying classes:**
  - Determining the *purpose* of a system

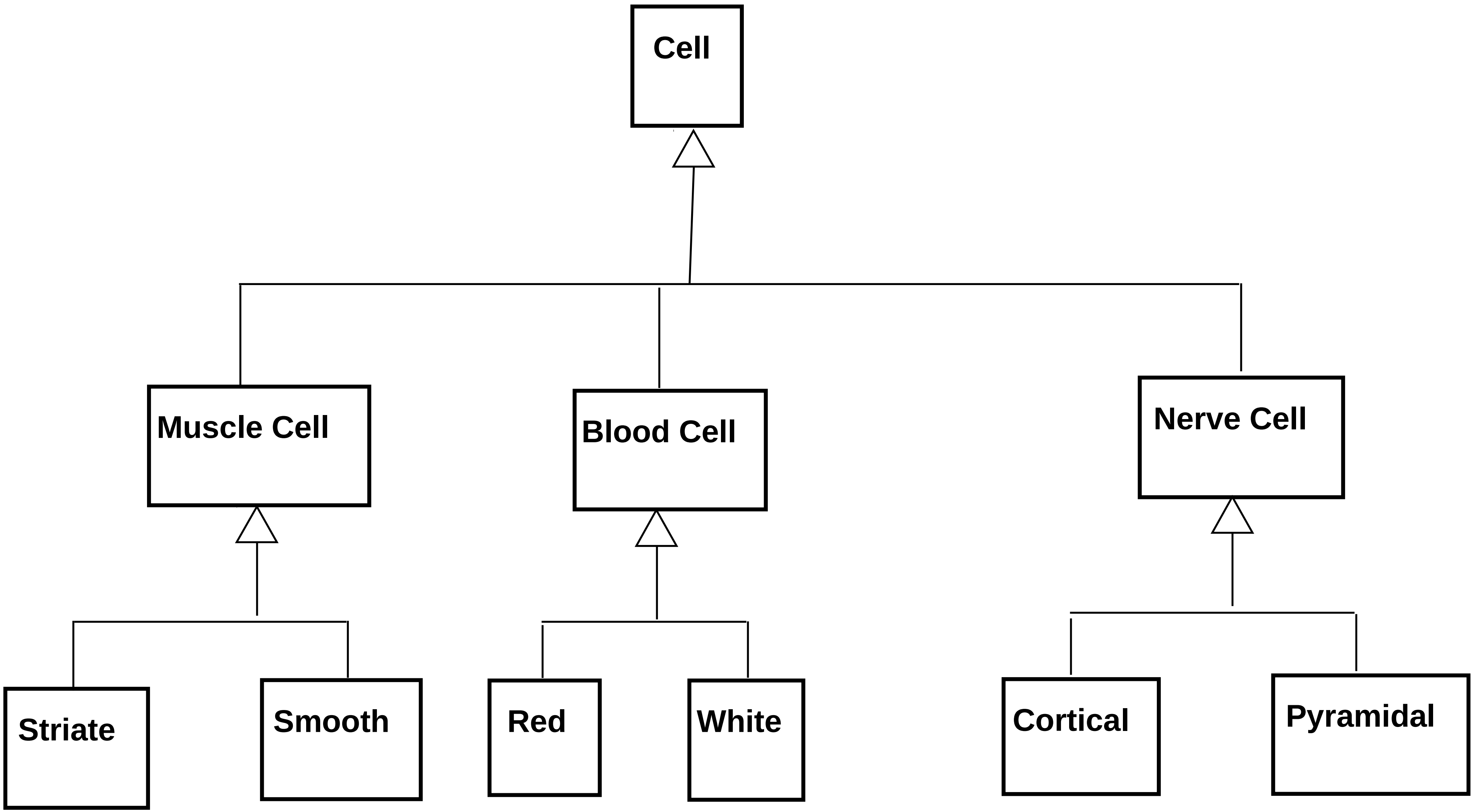
Hierarchy

- **Abstraction & Decomposition:**
  - Leads to classes & objects (object model)
- **Relationships between classes & objects**
  - Structure (static models), interactions (dynamic models)
  - Hierarchical relationships between classes
- **2 important hierarchies**
  - "Part of" hierarchy
  - "Is-kind-of" hierarchy

Part of Hierarchy



Is-Kind-of Hierarchy (Taxonomy)

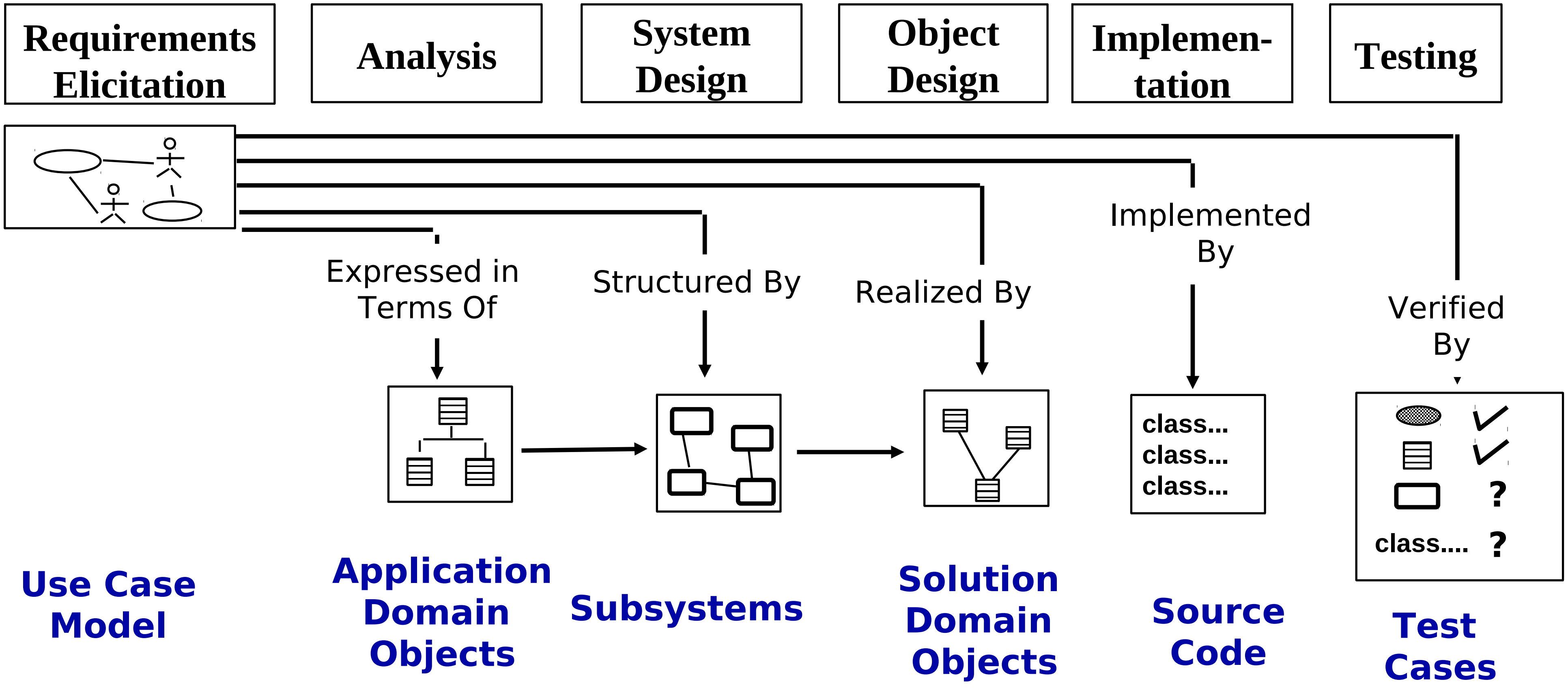


So where are we right now?

- **Three ways to deal with complexity:**
  - *Abstraction*
  - *Decomposition*
  - *Hierarchy*
- **Object-oriented decomposition is a good methodology**
  - Difficulty in determining the purpose of a system
  - Depending on the purpose, different objects are found
- **How can we do it right?**
  - Many different possibilities
  - Use Case Modelling (currently popular) approach:
    - Start with a description of the *functionality*
    - Then proceed to the *object model*
    - This leads us to the *software lifecycle*

Software Lifecycle Activities

...and their models



Reusability … living with change

- **A good software design solves a specific problem but is general enough to address future problems (for example, changing requirements)**

- **Experts do not solve every problem from first principles**
  - They reuse solutions that have worked for them in the past

- **Goal for the software engineer:**
  - Design the software to be reusable across application domains and designs
- **How?**
  - Use design patterns and frameworks whenever possible

Design Patterns and Frameworks

- **Design Pattern:**
  - A small set of classes that provide a template solution to a recurring design problem
  - Reusable design knowledge on a higher level than datastructures (link lists, binary trees, etc)
- **Framework:**
  - A moderately large set of classes that collaborate to carry out a set of responsibilities in an application domain.
    - Examples: User Interface Builder
- **Provide architectural guidance during the design phase**
- **Provide a foundation for software components industry**

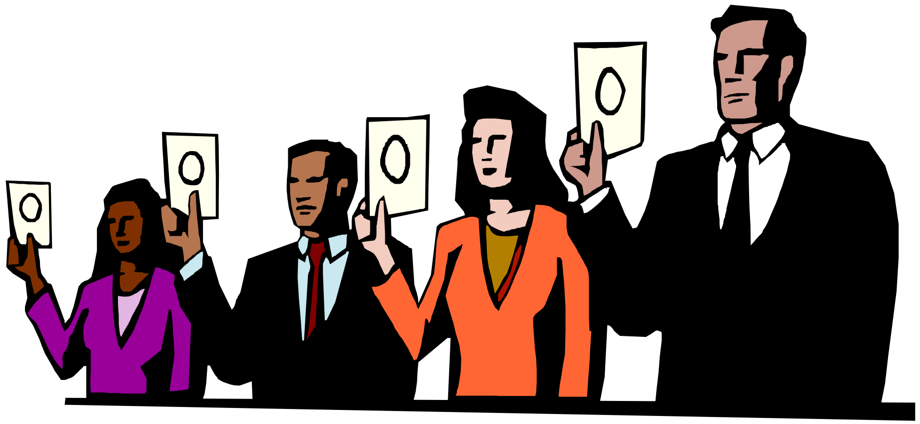
Patterns are used by many people

- Chess Master:
  - Openings
  - Middle games
  - End games
- Writer
  - Tragically Flawed Hero (Macbeth, Hamlet)
  - Romantic Novel
  - User Manual
- Architect
  - Office Building
  - Commercial Building
  - Private Home
- Software Engineer
  - Composite Pattern: A collection of objects needs to be treated like a single object
  - Adapter Pattern (Wrapper): Interface to an existing system
  - Bridge Pattern: Interface to an existing system, but allow it to be extensible
- Now Read Chapter 1 of BD book

Goal: software reliability

Use software engineering methodologies to develop the code.

Use formal methods during code development



What are formal methods?

Techniques for analyzing systems, based on some mathematics.

This does not mean that the user must be a mathematician.

Some of the work is done in an informal way, due to complexity.

Examples for FM

Deductive verification:

Using some logical formalism, prove formally that the software satisfies its specification.

Model checking:

Use some software to automatically check that the software satisfies its specification.

Testing:

Check executions of the software according to some coverage scheme.

Typical situation:

- Boss: Mark, I want that the new internet marketing software will be flawless. OK?
- Mark: Hmmm. Well, ..., Aham, Oh! Ah??? Where do I start?
- Bob: I have just the solution for you. It would solve everything.

Some concerns

- Which technique?
- Which tool?
- Which experts?
- What limitations?
- What methodology?
- At which points?
- How expensive?
- How many people?

- Needed expertise.
- Kind of training.
- Size limitations.
- Exhaustiveness.
- Reliability.
- Expressiveness.
- Support.

Common critics

- Formal methods can only be used by mathematicians.
- The verification process is itself prone to errors, so why bother?
- Using formal methods will slow down the project.

Some questions and answers...

Formal methods can only be used by mathematicians.

Wrong. They are based on some math but the user should not care.

The verification process is itself prone to errors, so why bother?

We opt to reduce the errors, not eliminate them.

Using formal methods will slow down the project.

Maybe it will speed it up, once errors are found earlier.

Some exaggerations

Automatic verification can always find errors.

Deductive verification can show that the software is completely safe.

Testing is the only industrial practical method.