Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 1: Introduction

Software Production has a Poor Track Record

Example: Space Shuttle Software

- Expensive: $10 Billion, millions of dollars more than planned
- Delayed delivery: 3 years late
- Crash: First launch of Columbia was cancelled because of a synchronization problem with the Shuttle's 5 onboard computers.
  - Error was traced back to a change made 2 years earlier when a programmer changed a delay factor in an interrupt handler from 50 to 80 milliseconds.
  - The likelihood of the error was small enough, that the error caused no harm during thousands of hours of testing.
- Errors and bugs still exist.
  - Astronauts are supplied with a book of known software problems "Program Notes and Waivers".

Software Engineering: A Problem Solving Activity

- Analysis: Understand the nature of the problem and break the problem into pieces
- Synthesis: Put the pieces together into a large structure

For problem solving we use

- Techniques (methods):
  - Formal procedures for producing results using some well-defined notation
- Methodologies:
  - Collection of techniques applied across software development and unified by a philosophical approach
- Tools:
  - Instrument or automated systems to accomplish a technique

Software Engineering: Definition

Software Engineering is a collection of techniques, methodologies and tools that help with the production of

- a high quality software system
- with a given budget
- before a given deadline

  while changes occur.

A disciplined approach to the design, production, and maintenance of computer programs

Scientist vs. Engineer

- Computer Scientist
  - Proves theorems about algorithms, designs languages, defines knowledge representation schemes
  - Do not have deadline…
- Engineer
  - Develops a solution for an application-specific problem for a client
  - Uses computers & languages, tools, techniques and methods

- Software Engineer
  - Works in multiple application domains
  - Has only 3 months...
  - …while changes occur in requirements and available technology

Learn for a lifetime ☺

Factors affecting the quality of a software system

- Complexity:
  - The system is so complex that no single programmer can understand it anymore
  - The introduction of one bug fix causes another bug
- Change:
  - The “Entropy” of a software system increases with each change: Each implemented change erodes the structure of the system which makes the next change even more expensive.
  - As time goes on, the cost to implement a change will be too high, and the system will then be unable to support its intended task. This is true of all systems, independent of their application domain or technological base.

Why are software systems so complex?

- The problem domain is difficult
- The development process is very difficult to manage
- Software offers extreme flexibility
- Software is a discrete system
  - Continuous systems have no hidden surprises
  - Discrete systems have!

Dealing with Complexity

1. Abstraction
2. Decomposition
3. Hierarchy

1. Abstraction

- Inherent human limitation to deal with complexity
- Chunking: Group collection of objects
- Ignore unessential details: => Models

Models are used to provide abstractions

- System Model:
  - Object Model:

    What is the structure of the system? What are the objects and how are they related?
  - Functional model:

    What are the functions of the system? How is data flowing through the system?
  - Dynamic model:

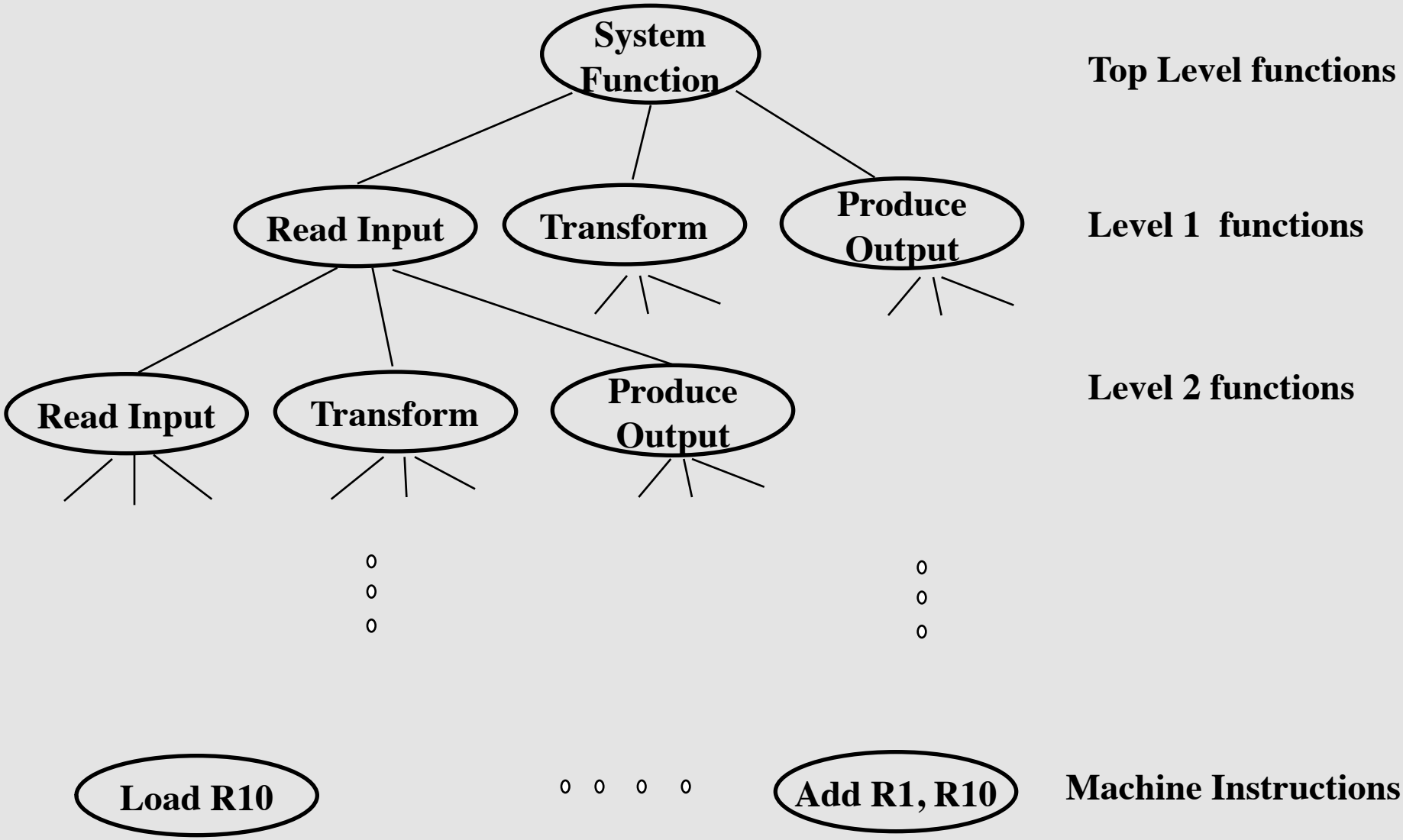
    How does the system react to external events? How is the event flow in the system ?
- Task Model:
  - What are the dependencies between the tasks?
  - How can this be done within the time limit?
  - What are the roles in the project or organization?
- Issues Model:
  - What are the open and closed issues? What constraints were posed by the client? What resolutions were made?

2. Decomposition

- A technique used to master complexity ("divide and conquer")
- Functional decomposition
  - The system is decomposed into modules
  - Each module is 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?

Functional Decomposition



Functional Decomposition

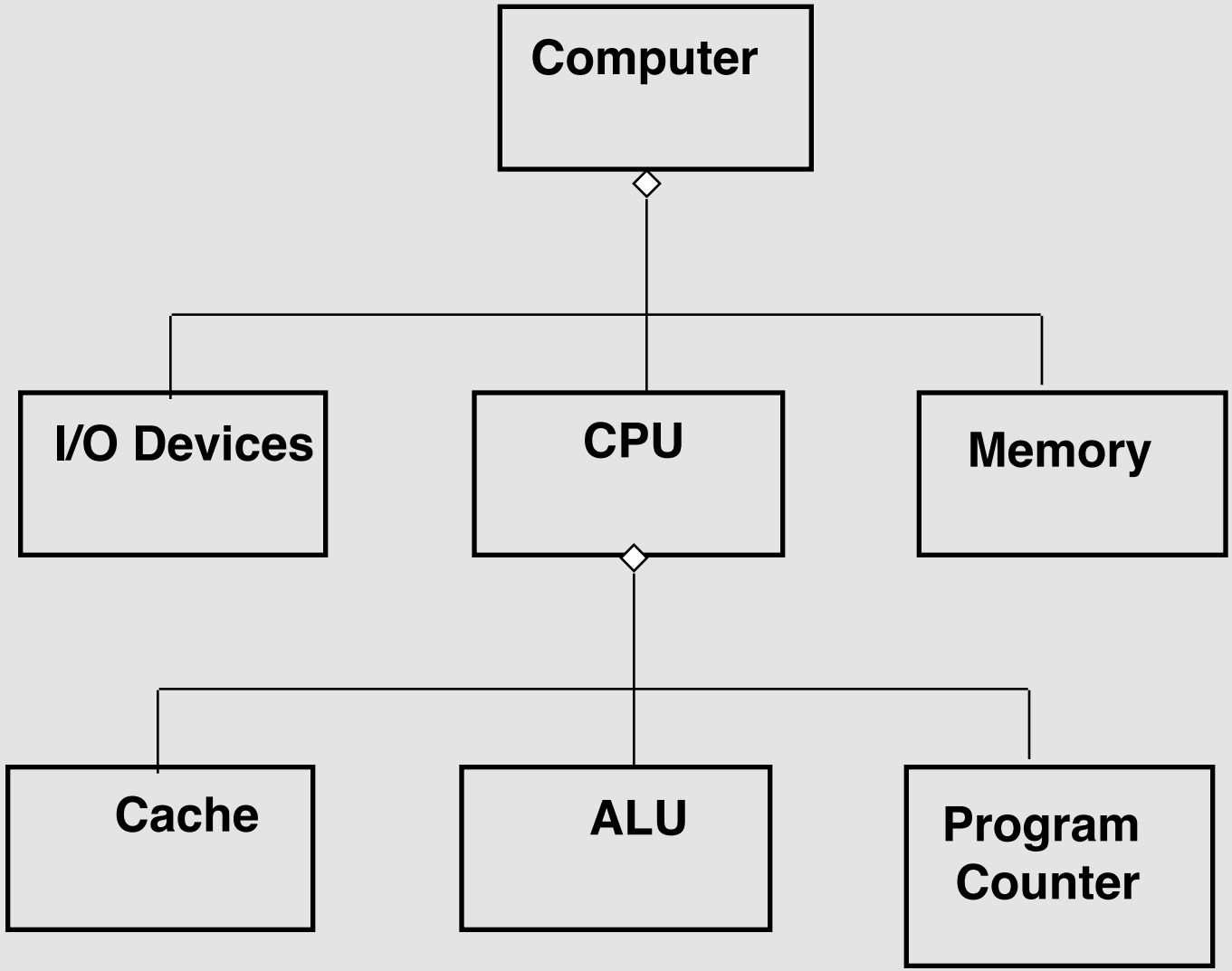
- Functionality is spread all over the system
- Maintainer must understand the whole system to make a single change to the system

- Consequence:
  - Codes are hard to understand
  - Code that is complex and impossible to maintain
  - User interface is often awkward and non-intuitive

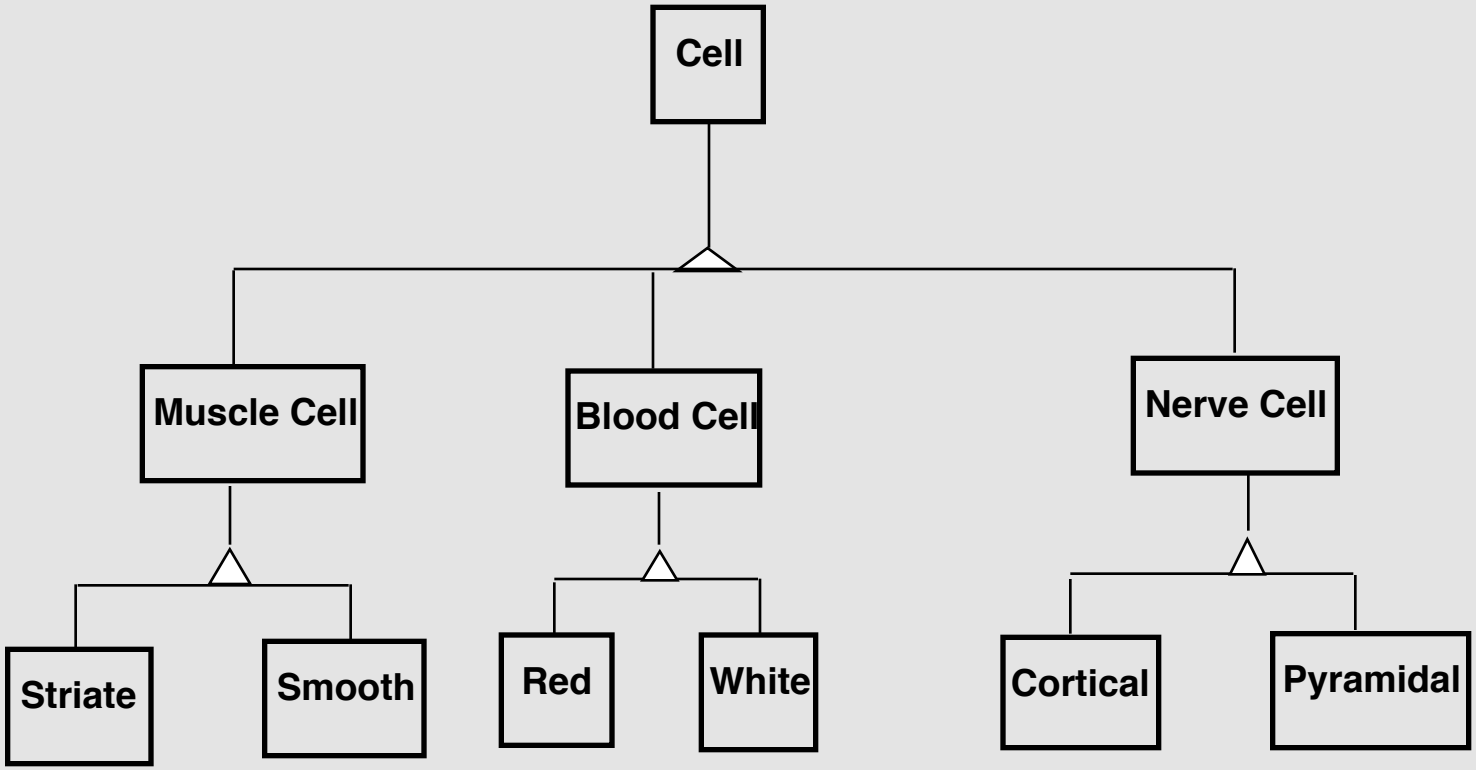
3. Hierarchy

- We got abstractions and decomposition
  - This leads us to chunks (classes, objects) which we view with object model
- Another way to deal with complexity is to provide simple relationships between the chunks
- One of the most important relationships is hierarchy
- Two important hierarchies
  - "Part of" hierarchy
  - "Is-kind-of" hierarchy

Part of Hierarchy



Is-Kind-of Hierarchy (Taxonomy)

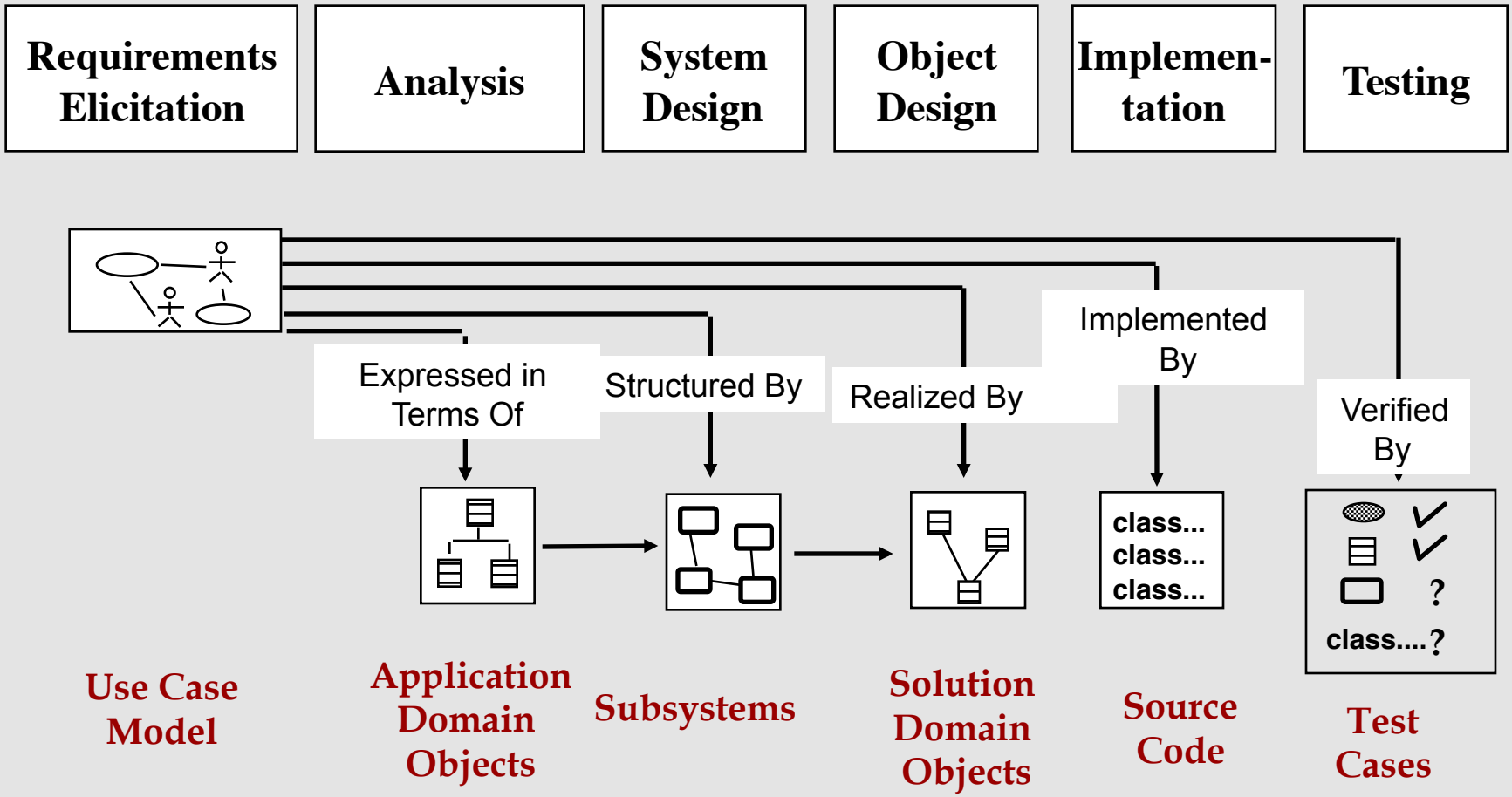


So where are we right now?

- Three ways to deal with complexity:
  - Abstraction
  - Decomposition
  - Hierarchy
- Object-oriented decomposition is a good methodology
  - Unfortunately, depending on the purpose of the system, different objects can be found
- How can we do it right?
  - Many different possibilities
  - Our current approach: Start with a description of the functionality (Use case model), then proceed to the object model
  - This leads us to the software lifecycle

Software Lifecycle Activities

...and their models



Goals of Good Software Design:

- Robust:

  handle exceptional conditions gracefully and behaves consistently

- Understandable:

  can be used by someone other than the original implementor; names of the components should be derived from the problem domain

- Modular:

  minimize the number of relationships between components

- Maintainable:

  problems are easily isolated; repair of one problem does not introduce problems in unrelated parts

- Extendible:

  accept new forms of data and new algorithms without disrupting existing software

- Reusable:

Reusability and Extensibility

less than half a typical system can be built of reusable software components (89)

- 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 scratch
  - 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, and be extensible
- How?

  e.g. use design patterns

Example:

PETSc

www.mcs.anl.gov/petsc/petsc2/documentation/

faq.html#work-efficiently

How do such a small group of people manage to write and maintain such a large and marvelous package as PETSc?

- a) We work very efficiently.

  We use Emacs for all editing; the etags feature makes navigating and changing our source code very easy.

  Our manual pages are generated automatically from formatted comments in the code, thus alleviating the need for creating and maintaining manual pages.

  We employ automatic nightly tests of PETSc on several different machine architectures. This process helps us to discover problems the day after we have introduced them rather than weeks or months later.

- b) We are very careful in our design (and are constantly revising our design) to make the package easy to use, write, and maintain.

- c) We are willing to do the grunt work of going through all the code regularly to make sure that all code conforms to our interface design. We will never keep in a bad design decision simply because changing it will require a lot of editing; we do a lot of editing.

- d) We constantly seek out and experiment with new design ideas; we retain the useful ones and discard the rest. All of these decisions are based on practicality.

- e) Function and variable names are chosen to be very consistent throughout the software. Even the rules about capitalization are designed to make it easy to figure out the name of a particular object or routine. Our memories are terrible, so careful consistent naming puts less stress on our limited human RAM.
- f) The PETSc directory tree is carefully designed to make it easy to move throughout the entire package.
- g) Our bug reporting system, based on email to petsc-maint@mcs.anl.gov, makes it very simple to keep track of what bugs have been found and fixed. In addition, the bug report system retains an archive of all reported problems and fixes, so it is easy to refind fixes to previously discovered problems.
- h) We contain the complexity of PETSc by using object-oriented programming techniques including data encapsulation (this is why your program cannot, for example, look directly at what is inside the object Mat) and polymorphism (you call MatMult() regardless of whether your matrix is dense, sparse, parallel or sequential; you don't call a different routine for each format).
- i) We try to provide the functionality requested by our users.
- j) We never sleep.