# INFORMATION HIDING

Yuqun Zhang

# A Programmer's Approach to Software Engineering

- Skip requirements engineering and design phases… just start coding

- Why?
  - Design is a waste of time.
  - We need to show something to the customer really quick.
  - I'm graded by the number of lines of code I write per unit time
  - I think the schedule is too tight

# But…

**The longer you postpone coding, the sooner you will be finished.**

# Design Overview

- Design is a trial-and-error process

- The process is not the same as the outcome of that process

- There is an interaction between requirements engineering and design
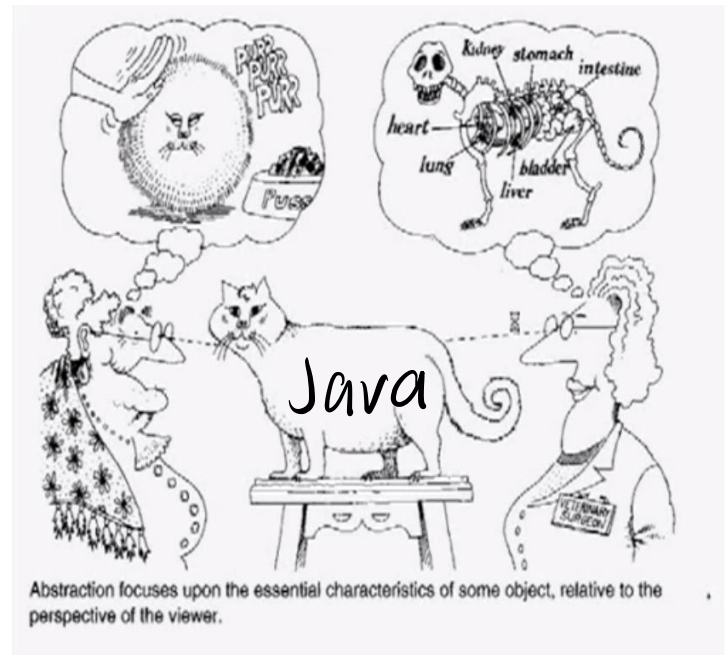
# Software Design Caveats

- There is no definite formulation

- There is no stopping rule

- Solutions are not simply true or false

# Software Design Principles

- Abstraction
- Modularity, coupling, and cohesion 模块、内凝与耦合
- Information hiding
- Limited complexity
- Hierarchical structure 层次结构

# Abstraction

- Procedural abstraction   *Algorithm*
  - A natural consequence of stepwise refinement
  - Name of procedure denotes the sequence of actions
- Data abstraction   *Data structure*
  - Goal is to find a hierarchy in the data (e.g., the range from general purpose data structures to application-oriented data structures)



*Java*

Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

# Modularity 模块化

- Modularity identifies structural criteria that tell something about individual modules and their interconnections
- Key concepts: cohesion and coupling
  - Cohesion: the glue that keeps a module together 内聚
  - Coupling: the strength of the connections between modules 耦合
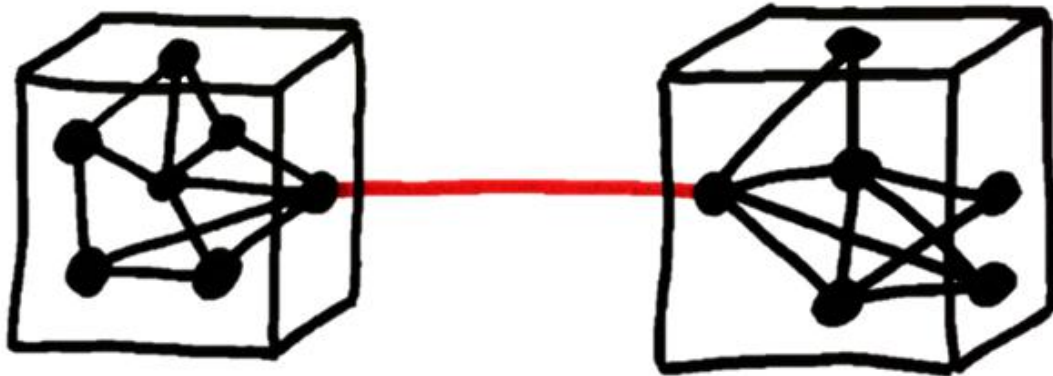
# Information Hiding

- Information hiding is a principle for breaking a program into modules

将会改变的组件放入模块内部，对其他模块不可见

*Design decisions that are likely to change independently should be* **secrets** *of separate modules*

将不会改变的功能设计成接口

*The only assumptions that should appear in the* **interfaces** *between modules are those that are considered unlikely to change*

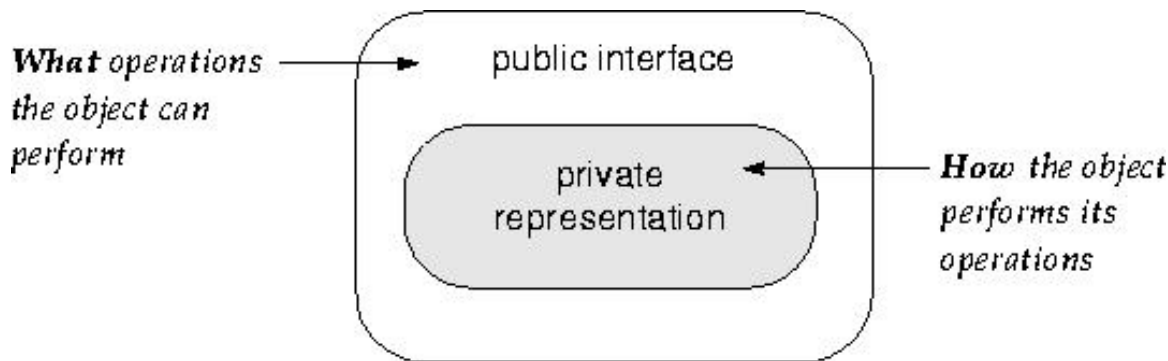⟹ 高内聚，低耦合

# Information Hiding

- Each module has a secret
- The design involves a series of decisions
  - For each decision, consider what other modules need to know and what others can be kept in the dark
- Information hiding is strongly related to
  - Abstraction: if you hide something, the user may abstract from that fact
  - Coupling: the secret decreases coupling between a module and the environment
  - Cohesion: the secret is what binds the parts of the module together

# Real-life Examples

- Your brain (your name and your experiences)


- An Email server (Account information)

# Motivation for Information Hiding

- A fundamental cost in software engineering is accommodating change 适应性改变、
  - Changes that require modifications to more modules are more costly than changes that are isolated to single modules
- The goal:
  - Anticipate likely changes 预料有可能的改变
  - Define interfaces that capture the stable aspects and implementations that capture the changeable aspects

*What* operations the object can perform → public interface

private representation ← *How* the object performs its operations

# A Simple Example

```
double sqrt (int)
```

- Can be implemented using bisection methods, Newton's method, factoring, etc.
- The client doesn't care (or need to know) how it is implemented
- The implementation ought to be able to change entirely without impacting the client code (and requiring only relinking)

# Design Case Study:
# Key Word in Context (KWIC)

- "The KWIC [Key Word in Context] index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."
  - Parnas 1972

- Consider KWIC applied to the title of this slide:

```
②        ①        ③
Design Case Study:              (KWIC) Key Word In Context
Case Study: Design              Case Study: Design
Study: Design Case              Context (KWIC) Key Word In
Key Word In Context (KWIC)      Design Case Study:
Word In Context (KWIC) Key      In Context (KWIC) Key Word
In Context (KWIC) Key Word      Key Word In Context (KWIC)
Context (KWIC) Key Word In      Study: Design Case
(KWIC) Key Word In Context      Word In Context (KWIC) Key
```
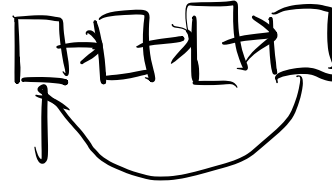
UserInput $\xrightarrow[\text{by '\n'}]{\text{divided}}$ [lines]
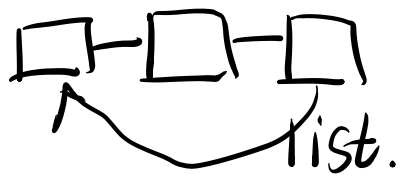
For each line

$\xrightarrow[\text{by ' '}]{\text{divided}}$ words $\longrightarrow$ add to set 生成一个 list.

Set. Sort by Alphabet.

list.

# Modularization #1

*This is a modularization in the sense meant by all proponents of modular programming. The system is divided into a number of modules with well-defined interfaces; each one is small enough and simple enough to be thoroughly understood and well programmed. Experiments on a small scale indicate that this is approximately the decomposition which would be proposed by most programmers for the task specified.*

# Modularization #1

| Changes | MasterControl | Input | CircularShift | Alphabetizer | Output |
|---|---|---|---|---|---|
| InputFormat | | X | | | |
| Storage | | X | X | X | X |
| Packing characters | | X | X | X | X |
| Index for CircularShift | | | X | X | X |
| Search | | | | X | X |

# KWIC Modularization #2 (Information Hiding)
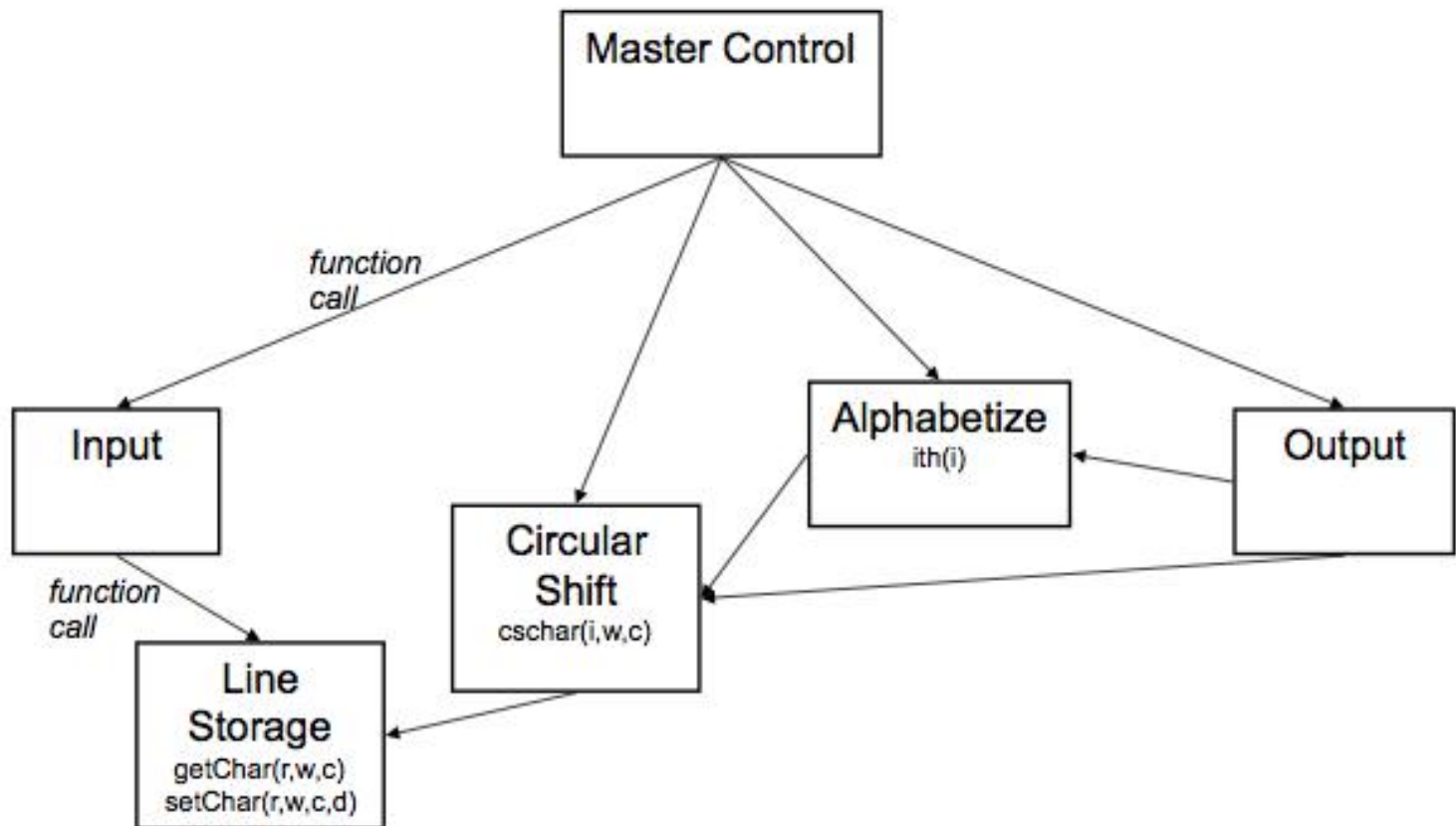
# Modularization #2

- Line storage abstracts the storage/representation of the input

- Circular shift is analogous to the circular shift in modularization #1, but…
  - The module gives the *impression* that we have a line holder creating all of the circular shifts

# Modularization #2

| Changes | Master Control | Input | Circular Shift | Alphabetizer | Output | LineStorage |
|---|---|---|---|---|---|---|
| InputFormat | | | | | | |
| Storage | | | | | | |
| Packing characters | | | | | | |
| Index for CircularShift | | | | | | |
| Search | | | | | | |
| Line Storage | | X | X | X | X | |

# KWIC Observations

- Similar at runtime
  - May have identical data representations, algorithms, even compiled code
- Different in code
  - Understanding
  - Documenting
  - Evolving
- The two versions are different in the way they divide work assignments and the interfaces between the modules

# But Software Changes…

- "… accept the fact of change as a way of life, rather than an untoward and annoying exception."
  - Brooks 1974

- "Software that does not change becomes useless over time."
  - Belady and Lehman

- For successful software projects, most of the cost is in evolving the system, not in initial development
  - Therefore, reducing the cost of change is one of the most important principles in software design

# Other Compounding Factors

- Independent development
  - Data formats (in design #1) are more complex than data access interfaces (in design #2)
  - Easier to agree on the interfaces in design #2
  - More work in design #2 is independent (because less is shared)
- Comprehensibility
  - Design of data formats in design #1 depends on the details of each module (and vice versa)
  - More difficult to understand each module in isolation in design #1

# Summary: Decomposition Criteria

- Functional decomposition
  - Break down by major processing steps
- Information hiding decomposition
  - Each module is characterized by a design decision it hides from others
  - Interfaces chosen and designed to reveal as little as possible about the hidden secrets

# Try to comment #1

```
class LegacyLine {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("line from (" + x1 + ',' + y1 + ")
to ("
       + x2 + ',' + y2 + ')');
    }
}
class LegacyRectangle {
    public void draw(int x, int y, int w, int h) {
        System.out.println("rectangle at (" + x + ',' + y + ")
with width "
       + w + " and height " + h);
    }
}
public class Demo {
    public static void main(String[] args) {
        Object[] shapes = { new LegacyLine(), new
LegacyRectangle() };
        int x1 = 10, y1 = 20, x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i) {
            if
(shapes[i].getClass().getName().equals("LegacyLine"))
            (LegacyLine)shapes[i].draw(x1, y1, x2, y2);
            else if
(shapes[i].getClass().getName().equals("LegacyRectangle"))
            (LegacyRectangle)shapes[i].draw(Math.min(x1, x2),
        Math.min(y1, y2), Math.abs(x2 - x1), Math.abs(y2 -
y1));
        }
    }
}
```

*hide this information*

# Try to comment #2

```java
interface Shape {
  void draw(int x1, int y1, int x2, int y2);
}

class Line implements Shape {
    private LegacyLine ll = new LegacyLine();
    public void draw(int x1, int y1, int x2, int y2) {
        ll.draw(x1, y1, x2, y2);
    }
}

class Rectangle implements Shape {
    private LegacyRectangle lr = new LegacyRectangle();
    public void draw(int x1, int y1, int x2, int y2) {
        lr.draw(Math.min(x1, x2), Math.min(y1, y2),
     Math.abs(x2 - x1), Math.abs(y2 - y1));
    }
}

public class Demo {
    public static void main(String[] args) {
        ArrayList<Shape> shapes = new ArrayList<Shape>();
        shapes.add(new Line());
        shapes.add(new Rectangle());

        int x1 = 10, y1 = 20, x2 = 30, y2 = 60;
        for (Shape s : shapes)
          s.draw(x1, y1, x2, y2);
    }
}
```

# Information Hiding Summary

- Decide what design decisions are likely to change and which are likely to be stable
- Put each design decision likely to change in its own module
- Assign each module an interface that hides the decision likely to change and exposes only stable design decisions
- Ensure that the clients of a module depend only on the stable interface and not the implementation

- Benefit: *if* you can correctly predict what may change and hide information properly, each change will affect one module

# Types of Secrets

- Algorithms (procedural abstraction)
- Data representations (abstract data types)
- Characteristics of a hardware device (virtual machines, hardware abstraction layers, etc.)
  - E.g., whether a thermometer measures in Fahrenheit or Celsius
- Where information is acquired
  - E.g., which search engine is used
- User interface (e.g., model-view pattern)

- What are other examples?
  - What about in the context of your projects?

# QUESTIONS?