"...And that, in simple terms, is what's wrong with your software design."

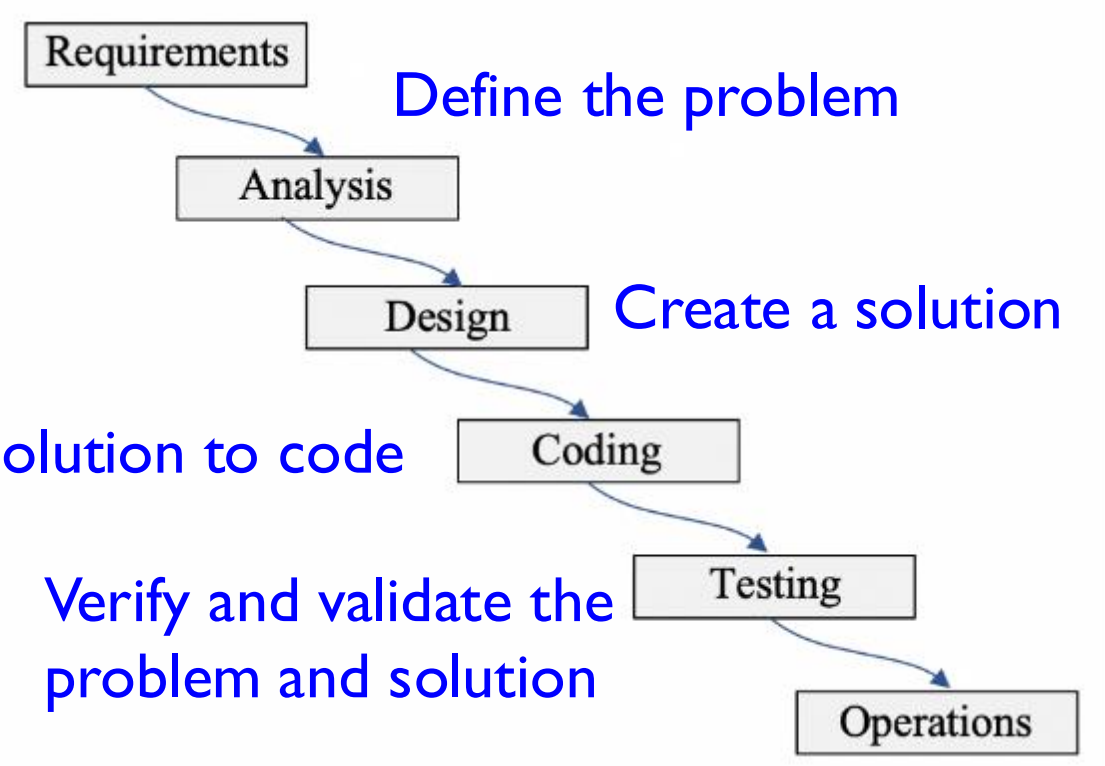
Chapter 4.
Fundamentals of Software Design

Basic Design Principles

Outline

- What Is Software Design?
- Modularity
- Cohesion
- Coupling
- Data Encapsulation
- Information Hiding
- Separation of Concerns

Waterfall vs. Problem Solving



What Is Design?

- **Design** is the *creative* process of transforming the problem into a solution
- The description of a solution is also known as design
  - The requirements specification defines the problem
  - The design document specifies a particular solution to the problem

Customer requirement

1. Have one trunk
2. Have four legs
3. Should carry load both passenger & cargo
4. Black in color
5. Should be herbivorous

Our Solution

1. Have one trunk ☑
2. Have four legs ☑
3. Should carry load both passenger & cargo ☑
4. Black in color ☑
5. Should be herbivorous ☑

Our Value add:

Also gives milk ☺

Design Is a Sloppy Process

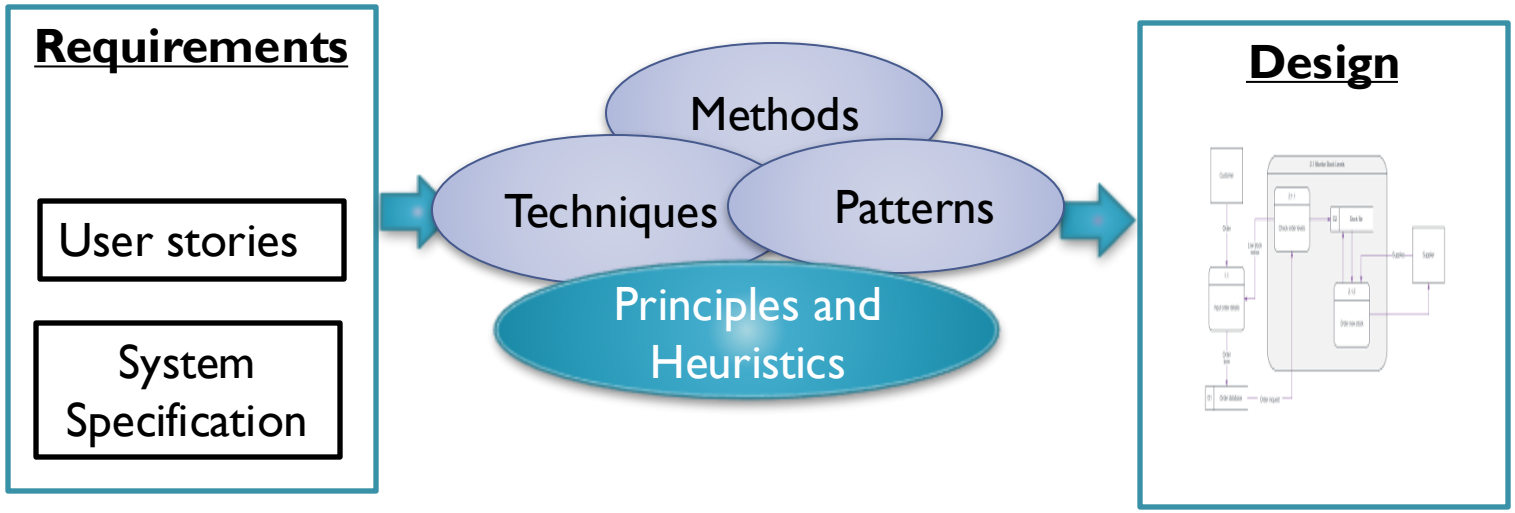
- When design is "good enough"?
- How much detail is enough?
  - How much should be done with a formal design notation
  - How much should be left to be done at the keyboard?
- When are you done?
  - Most common answer: when you are out of time!
  - Design is open-ended
- Making mistakes is the point of design

Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. 2nd Edition. Microsoft Press, 2004.

Design Process

- This is probably the most that can be hoped for
- Relying on miracles isn't a reliable way

Design Process – cont'd



- Can be made more systematic and predictable through the application of methods, techniques and patterns, all applied according to principles and heuristics

Conflicting Approaches in History

- The history of software design has been marked by fanatic advocates of *wildly conflicting* design approaches
  - In 1990s, design zealots were advocating *'design everything'*
  - In 2000s, some software swamis are arguing for "*design nothing*" - *"Big Design Up Front* is *BDUF* - *BDUF* is bad"
  - Alternative to BDUF isn't no design up front, but Little Design Up Front (*LDUF*) or Enough Design Up Front (*ENUF*).

Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. 2nd Edition. Microsoft Press, 2004.

Modularity

- Modularity: the degree to which the components of a system are separated
- A modular design divides complex software into uniquely named components (*modules*)
  - After the modules are developed, they are integrated to meet the software requirements
- Divide and conquer
  - Solve a complex problem by breaking it into manageable pieces
- Typical modules
  - Method (function), class, interface, package
  - "module" in Java 9

Modularity – cont'd

```
public class TicTacToe {
        static Scanner in;
        static String[] board;
        static String turn;

        public static void main(String[] args) {
                in = new Scanner(System.in);
                board = new String[9];
                turn = "X";
                String result = null;
                for (int a = 0; a < 9; a++) {
                        board[a] = String.valueOf(a + 1);
                }
                System.out.println("Welcome to 2 Player Tic Tac Toe.");
                System.out.println("----------------------------");
                printBoard();
                System.out.println("X's will play first. Enter a slot number to place X in:");
                while (result == null) {
                        int numInput;
                        try {
                                numInput = in.nextInt();
                                if (!(numInput > 0 && numInput <= 9)) {
                                        System.out.println("Invalid input; re-enter slot number:");
                                        continue;
                                }
                        } catch (InputMismatchException e) {
                                System.out.println("Invalid input; re-enter slot number:");
                                continue;
                        }
                        if (board[numInput - 1].equals(String.valueOf(numInput))) {
                                board[numInput - 1] = turn;
                                if (turn.equals("X")) {
                                        turn = "O";
                                } else {
                                        turn = "X";
                                }
                                printBoard();
                                for (int a = 0; a < 8; a++) {
                                        String line = null;
                                        switch (a) {
                                        case 0:
                                                line = board[0] + board[1] + board[2]; break;
                                        case 1:
                                                line = board[3] + board[4] + board[5]; break;
                                        case 2:
                                                line = board[6] + board[7] + board[8]; break;
                                        case 3:
                                                line = board[0] + board[3] + board[6]; break;
                                        case 4:
                                                line = board[1] + board[4] + board[7]; break;
                                        case 5:
                                                line = board[2] + board[5] + board[8]; break;
                                        case 6:
                                                line = board[0] + board[4] + board[8]; break;
                                        case 7:
                                                line = board[2] + board[4] + board[6]; break;
                                        }
                                        if (line.equals("XXX")) {
                                                result = "X";
                                        } else if (line.equals("OOO")) {
                                                result = "O";
                                        }
                                }

                                if (result == null) {
                                        for (int a = 0; a < 9; a++) {
                                                if (Arrays.asList(board).contains(String.valueOf(a + 1))) {
                                                        break;
                                                } else if (a == 8)
                                                        result = "draw";
                                        }

                                }

                                if (result == null)
                                        System.out.println(turn + "'s turn; enter a slot number to place " + turn + " in:");

                        } else {
                                System.out.println("Slot already taken; re-enter slot number:");
                                continue;
                        }
                }
                if (result.equalsIgnoreCase("draw")) {
                        System.out.println("It's a draw! Thanks for playing.");
                } else {
                        System.out.println("Congratulations! " + result + "'s have won! Thanks for playing.");
                }
        }

        static void printBoard() {
                System.out.println("/---|---|---\\");
                System.out.println("| " + board[0] + " | " + board[1] + " | " + board[2] + " |");
                System.out.println("|---|---|---|");
                System.out.println("| " + board[3] + " | " + board[4] + " | " + board[5] + " |");
                System.out.println("|---|---|---|");
                System.out.println("| " + board[6] + " | " + board[7] + " | " + board[8] + " |");
                System.out.println("/---|---|---\\");
        }
}
```

Initialize the game

Get input (make a move)

Display updated board

Check if there is a winner

Check if it is a draw

Change turn to continue

Display the game result

Modularity – cont'd

- Problems
  - Inflexible to deal with requirements changes
    - The entire main method
    - hardly reusable for new requirements
  - Not unit-testable
  - In essence: inadequate requirements analysis

Modularity – cont'd

- Decomposability
  - The extent to which the problem can be broken into sub-problems with simple relations
- Composability
  - The extent to which the modular solutions to the sub-problems can be assembled as a solution to the whole problem.
- Modules should be reusable: plug and play
- Modules should be testable and confine runtime exceptions and errors to very few modules

Java "Modules"

- A new feature in Java 9 via the Java Platform Module System (JPMS)
  - aka Java Jigsaw or Project Jigsaw
  - Jigsaw was the internally used project name during development
  - Later Jigsaw changed name to JPMS
- Java Module: a mechanism to packages into modules, specifying
  - which of the packages a module contains that should be visible to other Java modules using this module
  - which other Java modules is requires to do its job

Cohesion

- Cohesion: the degree to which the elements inside a module belong together
  - High (low) cohesion: the elements inside the module have a high (low) degree of connectedness

- Good design aims for high cohesion!
  - Method: statements belong together (to one function)
  - Class: public constructors & methods belong together
  - Rule of thumb: a descriptive name

Cohesion: Method

```
public void doStuff(int flag){
        calculateHolidays()
        if (flag==0){
                updateZipCode();
        } else
        if (flag==1){
                predictRetirementBenefit()
        } else
        if (flag==2){
                String message = composeMessage();
                sendEmail(message);
        }
        updateW4Form();
}
```

- Difficult to summarize their functionality

Cohesion: Class

- Public interface as an abstract data type
  - Public constructors and methods belong together and represent the essential properties of an object

```
class Stack<E> {
        public Stack();
        public E push(E element);
        public E pop();
        public E peek();
        public boolean empty();
        public int search(E element);
}
```

- Looks Good!

Cohesion: Class - cont'd

```
class Stack<E> extends Vector<E>{
        public Stack();
        public E push(E element);
        public E pop();
        public E peek();
        public boolean empty();
        public int search(E element);
}
class Vector<E> extends AbstractList<E> {
        …
        public void add(int index, E element);
        public void remove(int index);
}
```

- Bad!

Cohesion: Class – cont'd

```
public class TTT extends JFrame {

   public static final int CANVAS_WIDTH = CELL_SIZE * COLS;
   public static final int CANVAS_HEIGHT = CELL_SIZE * ROWS;
…
   public static final int SYMBOL_SIZE = CELL_SIZE - CELL_PADDING *
   public static final int SYMBOL_STROKE_WIDTH = 8;
   public TTT ();
   public void initGame();
   public void updateGame(Seed theSeed, int row, int col);
   public boolean isDraw();
   public boolean hasWon(Seed theSeed, int row, int col);
}
```

- Bad: mixed abstractions

What's Wrong with Low Cohesion?

- Difficult to understand, test, maintain, and reuse
  - The doStuff method
- Perfect cohesion is not the goal
  - A module is perfectly cohesive if it only consists of a single, atomic element.
  - Such modules are either hardly useful for complex tasks or tightly coupled to other modules.
  - Cohesion should be balanced with module complexity and coupling.

Coupling

- Coupling: degree of interdependence between modules
  - or the strength of the relationships between modules
- Consequence of tight coupling
  - If module Q depends on module P, a change to P can require a corresponding change to Q
  - If the latter is not made, it leads to faults
- Good design aims for loose coupling!

Coupling: Example

```
public class Client extends Superclass{
   private A a = new A ();

   public void foo(C c, D d) {
       a.am();
       a.var.bm();
           E e = d.getE();
           e.doIt();
    }
}

public class A {
            public B var;
            public A();
}
```

Class	Constructor /Method	Variable
A	A(), am()	var
B	bm	
C		
D	getE()	
E	doIt()	

Different dependencies (consider changes)

- var: public variable
- A vs C

Coupling Criteria

- Size: number of connections between modules
  - Method m1(x) is more loosely coupled to its calling modules than method m2(y1, y2, y3, y4, y5, y6)
  - A class with 4 well-defined public methods is more loosely coupled to modules that use it than a class that exposes 40 methods.
- Visibility: prominence of the connection between two modules
  - Passing data in a parameter list is making an obvious connection - good
  - Modifying global data so that another module can use that data is a sneaky connection – bad
- Flexibility: how easily the connections between modules can be changed

Which Is Less Coupled?

```
public class Foo {
    public void example(Bar b) {
        C c = b.getC();
        c.doIt();
    }
}
```

```
public class Foo {
    public void example(Bar b) {
        b.doItOnC();
    }
}
```

- 2nd: Foo is not directly dependent on C
- Law Of Demeter ("only talk to friends") reduces coupling

Coupling: RemoteControl

```
public class RemoteControl {
   private SamsungTV tv;
   public void turnOn() {
         tv.on();
   }
  …
}
public class SamsungTV {
   public void on();
   public void off();
   public void tuneChannel(int channel) ;
}
```

- RemoteControl
  - tightly coupled to SamsungTV
  - affected by changes to SamsungTV.
  - does not work for other TV brands

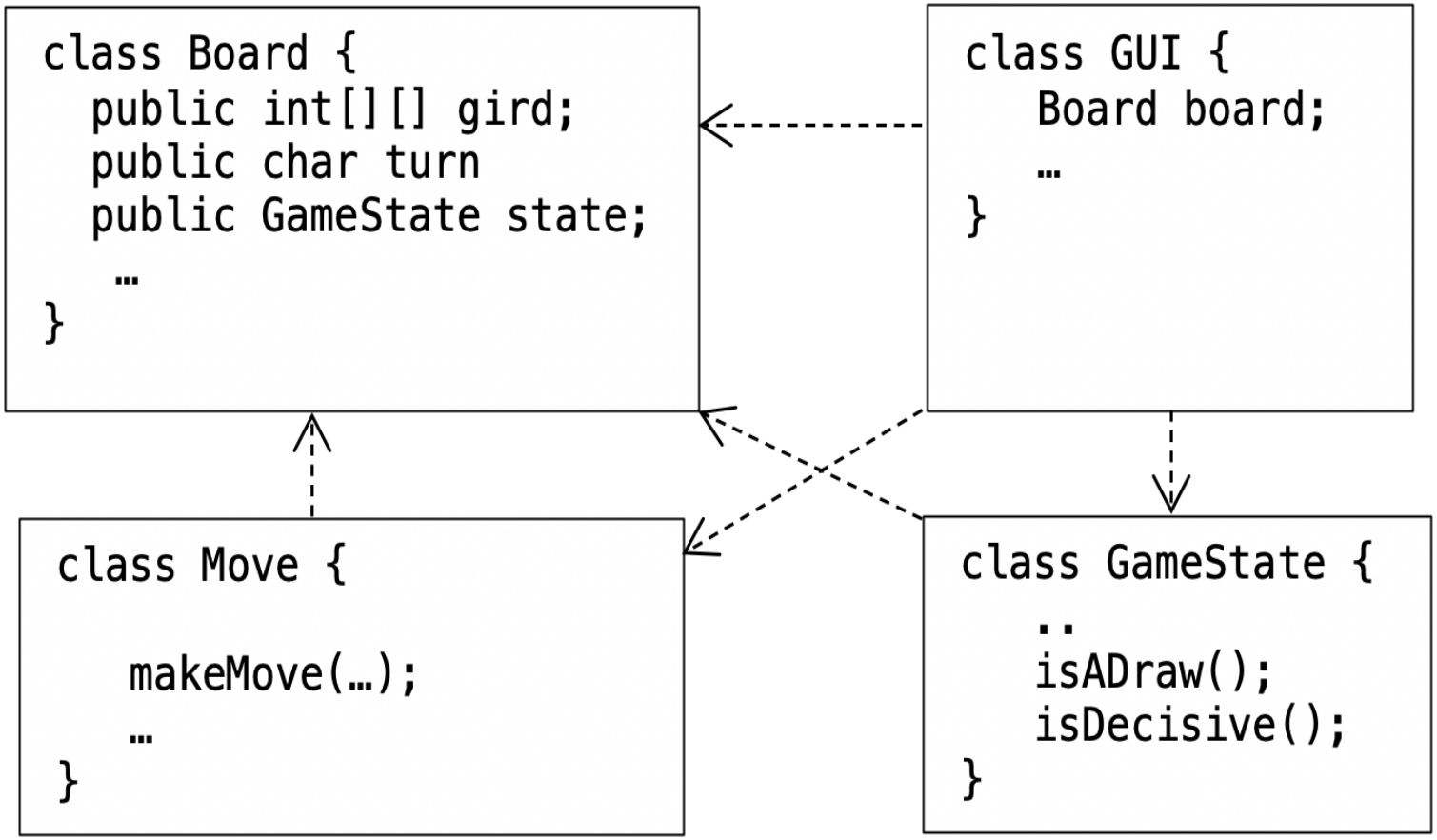
Coupling: RemoteControl - revised

```
public class RemoteControl {
    private TV tv;
    public void turnOn() {
            tv.on();
    }
}
public interface TV {
    public abstract void on();
    public abstract void off();
    public abstract void tuneChannel(int channel);
}

public class SamsungTV implements TV {
    public void on();
    public void off();
    public void tuneChannel(int channel);
}
```

Encapsulation

- Oxford Languages:
  - (a) the action of enclosing something in or as if in a capsule
  - (b) the succinct expression or depiction of the essential features of something
- Encapsulation: bundling of data with the methods that operate on that data in one module
  - Representing the essential features of something
- Benefits:
  - Localizes requirements change and bug fixes.
  - The change of a data structure usually requires the change of its operations.



- Not encapsulated because the data structures and related operations are scattered in several modules.

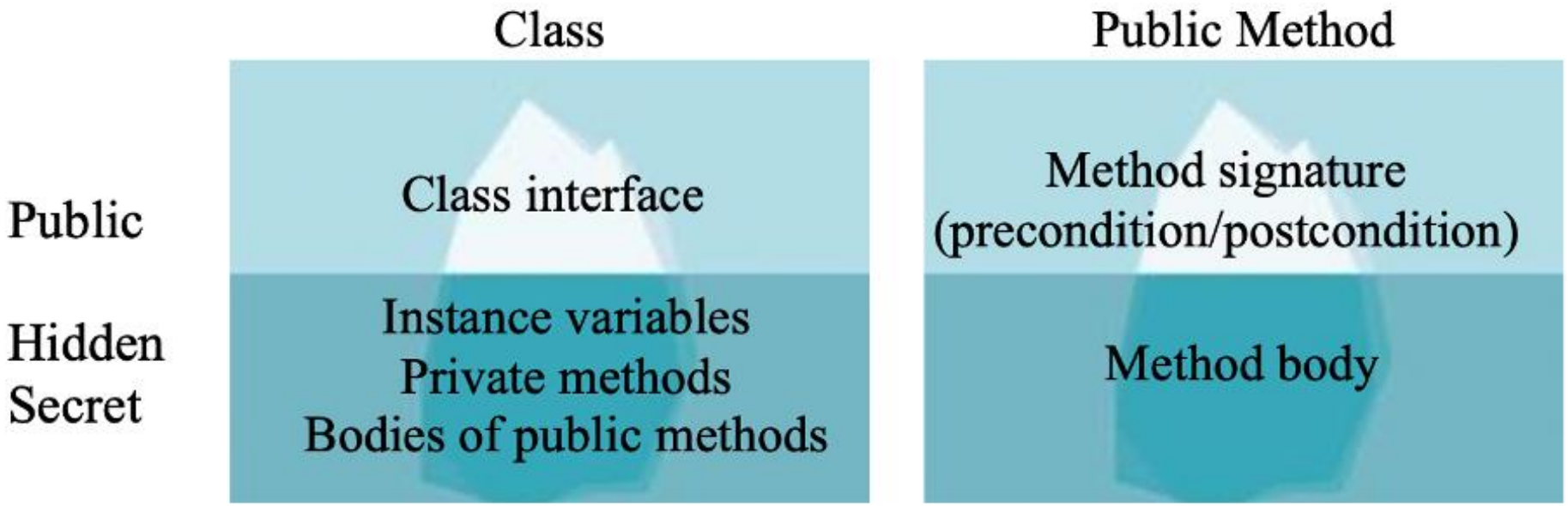
Encapsulation and Changes

- Encapsulation provides a way to cope with change and maintenance
- Identify the aspects that are likely to change
  - Business rules: Congress changes the tax structure
  - Hardware dependencies
  - Nonstandard language features
  - Data-size constraints
  - …
- Design to minimize the effects of change
  - Data structures are unlikely to change
  - Implementation details may change

Information Hiding

- Make the information inside a module invisible to the module's clients.
  - Data structures and implementation details
- Public method
  - Visible: signature, pre/postcondition
  - Hidden: method body
- Class
  - Visible: public constructors and methods
  - Hidden: private variables and methods

Information Hiding



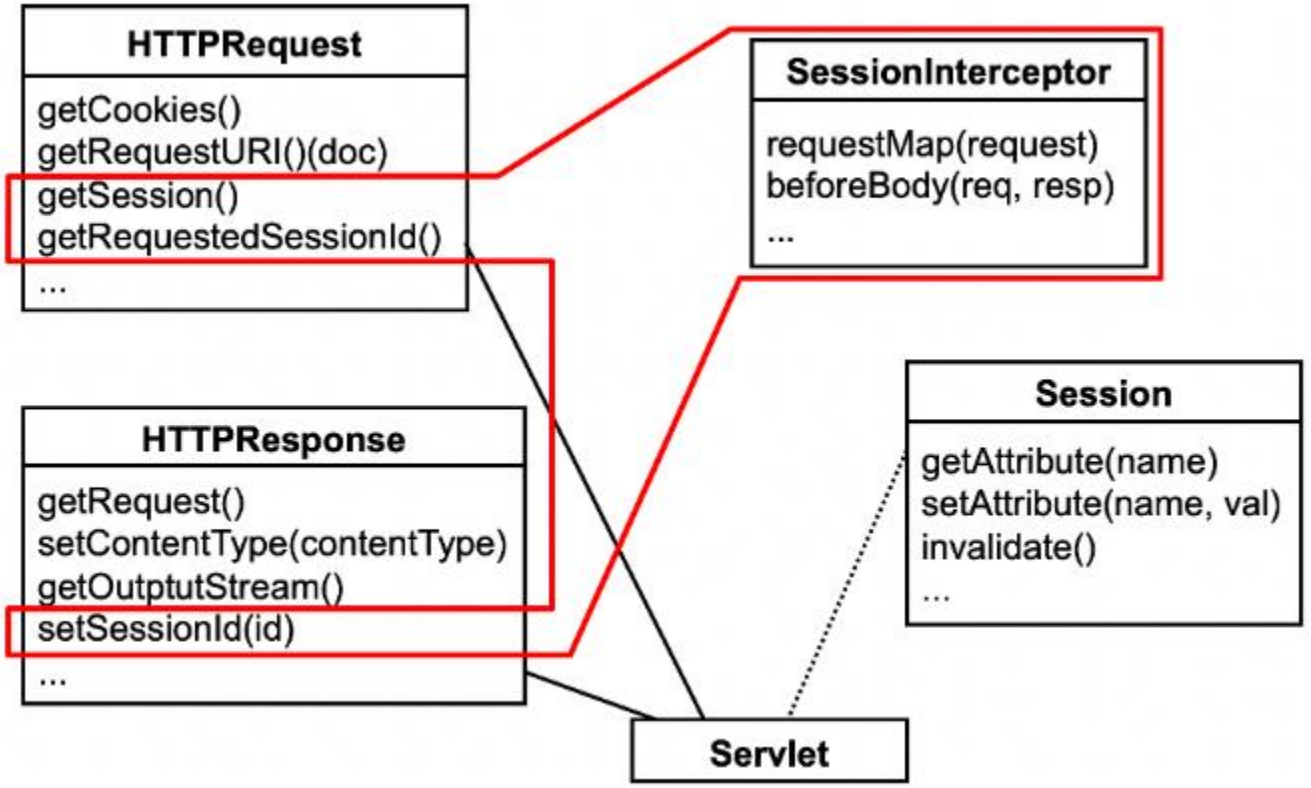
- Make instance variables private
- Provide public getters/setters if necessary
- Avoid excessive unnecessary getters/setters

Separation of Concerns

- Separate a program into distinct modules such that each module addresses a separate concern.
  - Facilitate module upgrade, reuse, and independent development.
- General concerns (example: Websites)
  - Organization of webpage content: HTML:
  - Content presentation style: (CSS)
  - how the content interacts and behaves with the user: Javascript
- Specific concerns
  - Audit log of login, print a sales report

Crosscutting Concerns

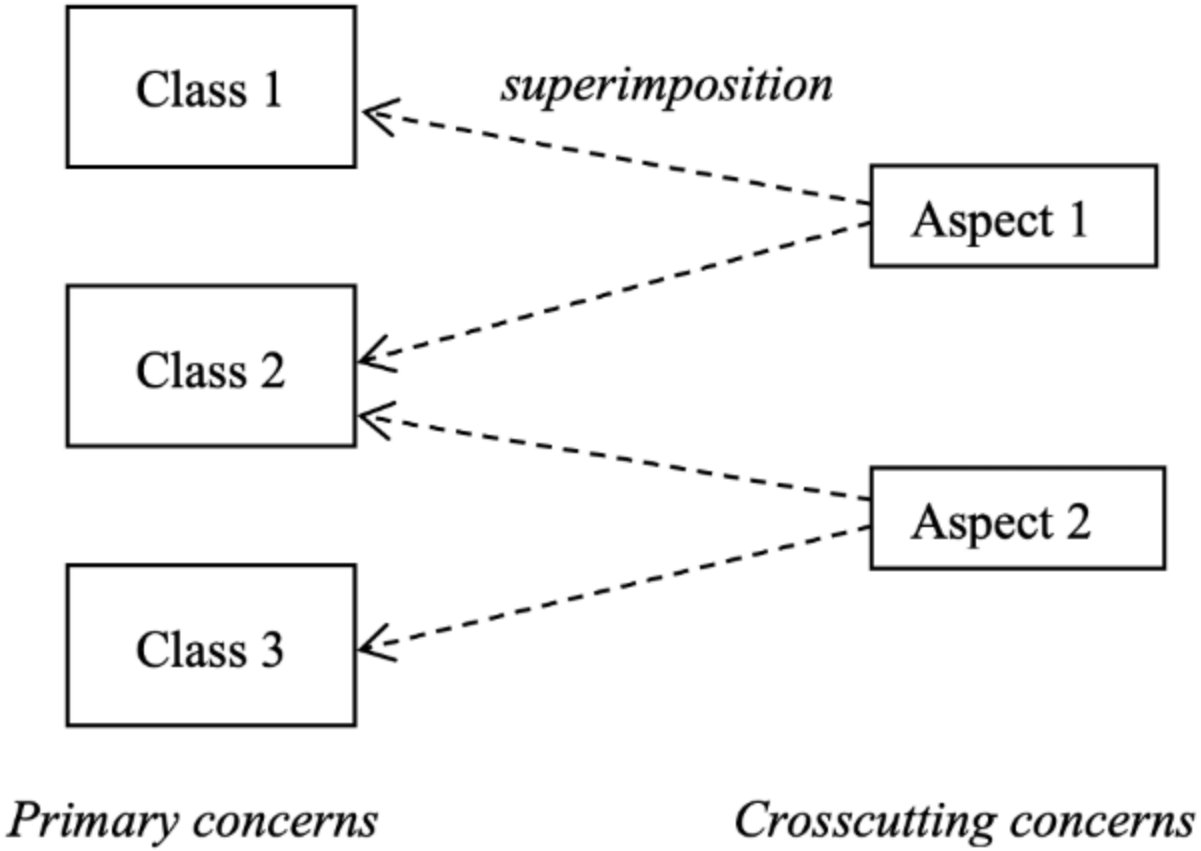
- Crosscut many modules
  - Logging, error handling, data persistence, security check, …



Crosscutting Concerns: cont'd

- Usually viewed as secondary concerns
  - The classes and methods address the primary concerns of business goals.
- Cannot be modularized using the modularization mechanisms in object-oriented or procedural languages
  - They inherently follow different rules for functional decomposition.
  - Scattered in multiple modules (called "tangled")
- Solution: Aspect-oriented programming

Crosscutting Concerns: AOP



Representative AOP language: AspectJ for Java

"You never actually find a perfect answer to a problem. You just find the answer that has the fewest problems."

— James Gosling