

CS 2110

Software Design Principles I

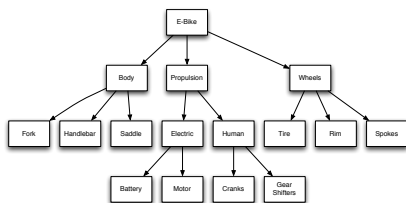
Juan Altmayer Pizzorno
port25.com

Overview

- **Today:** Design Concepts & Principles
 - Top-Down, Bottom-Up Design
 - Software Process (briefly)
 - Modularity
 - Information Hiding, Encapsulation
 - Principles of Least Astonishment and “DRY”
 - Refactoring (if there's time)
- Next week: Test-Driven Development

Top-Down Design

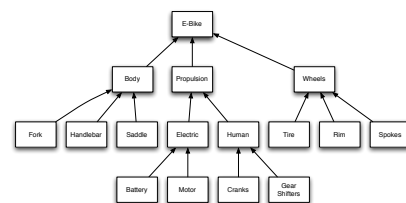
- Let's build an electric bicycle!



- Refine the design at each step
- **Decomposition** / “Divide and Conquer”

Bottom-Up Design

- Just the opposite: start with parts



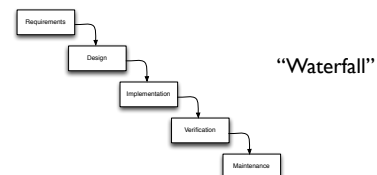
- **Composition**
- Build-It-Yourself (e.g. IKEA furniture)

Top-Down vs. Bottom-Up

- Two **ways** to **think** about a problem
 - It's sometimes good to alternate
 - **Not** the **only ways** to go about it
- With **Top-Down** it's **harder** to **test early** because parts needed may not have been designed yet
- With **Bottom-Up**, you may end up **needing** things **different** from how you built them

Software Process

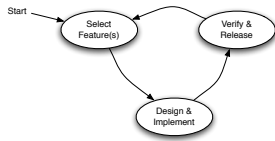
- How do you go about developing software?



- Are the **requirements fixed** and **well understood**?
Most software problems are not like that
- Not enough customer **feedback**

Incremental & Iterative

- Deliver **parts** in several **small cycles**



- “Don’t build for the **future**, for it is **unknown**”
- Software development is more like gardening than architecture

Modularity

- **Module**: **component** of a system with a **well-defined interface**. Examples:
 - Tires in a car
 - Tuner in a TV
 - Screen Savers
 - ...
 - And in Java??
- Modules “**hide information**” behind their interfaces

Information Hiding

- What “information” do modules hide? “**Internal**” **design decisions**.

```
class Set {  
    ...  
    public void add(Object o) ...  
    public boolean contains(Object o) ...  
    public int size() ...  
}
```

- A class’s **interface** is everything in it that is **externally accessible**

Encapsulation

- By hiding **code** and **data** behind its interface, a class **encapsulates** its “inner workings”
- Why is that good?
 - **Revising** those **design decisions** takes **changing** the inner workings of the **module only**: helps “keep software soft”

```
class LineSegment {  
    private Point2D _p1, _p2;  
    ...  
    public double length() {  
        return _p1.distance(_p2);  
    }  
}  
  
class LineSegment {  
    private Point2D _p;  
    private double _length;  
    private double _phi;  
    ...  
    public double length() {  
        return _length;  
    }  
}
```

Encapsulation

- Why is that good? (continued)
 - There can be **multiple implementations** of the same thing; they all **behave** more or less **the same way**
 - That is the point of Java’s **interface**

```
Iterator it = collection.iterator();  
  
while (it.hasNext()) {  
    Object next = it.next();  
    doSomething(next);  
}
```

Degenerate Interfaces

- **Public fields** are usually a **Bad Thing**:

```
class Set {  
    public int _count = 0;  
  
    public void add(Object o) ...  
  
    public boolean contains(Object o) ...  
  
    public int size() ...  
}
```
- Anybody can change them; the class has **no control**

Interfaces vs. Implementations

- This says “I need this specific **implementation**”:

```
public void doSomething(LinkedList items) ...
```

- This says “I need something with this **interface**” and can be used with any class that implements it:

```
public void doSomething(Iterable items) ...
```

- **Interfaces** represent higher levels of **abstraction** (focus on “**what**” and leave out the “**how**”)

Principle of Least Astonishment

- Have your designs work how a **user** would **expect**

Bad:

```
public int product(int a, int b) {  
    return a*b > 0 ? a*b : -a*b;  
}
```

Better:

```
public int absProduct(int a, int b) {  
    return a*b > 0 ? a*b : -a*b;  
}
```

- **Names** and other **clues** play a big role!

Principle of Least Astonishment

- Unexpected **side effects** are a **Bad Thing**

```
class Integer {  
    private int _value;  
  
    ...  
  
    public Integer times(int factor) {  
        _value *= factor;  
        return new Integer(_value);  
    }  
}  
  
...  
  
Integer i = new Integer(100);  
Integer j = i.times(10);
```

Duplication

- **Duplication** makes software **less soft**: when it changes, either
 - you change **all** instances
 - some become **inconsistent**
- Duplication can arise in many ways:
 - **constants** (repeated “magic numbers”)
 - code vs. **comment**
 - within an **object’s state**
 - ...

Duplication in Comments

```
public double totalArea() {  
    ...  
    // now add the circle  
    area += PI * pow(radius,2);  
    ...  
}  
  
public double totalArea() {  
    ...  
    area += circleArea(radius);  
    ...  
}  
  
private double circleArea(double radius) {  
    return PI * pow(radius, 2);  
}
```

- **What** is being done should be **readable** from **code**
- **Comments** should answer **why** it is being done; try to code so that you don’t need them

Duplication of State

- Duplication of state can lead to **inconsistency** **within an object**:

```
class LineSegment {  
    private Point2D _p1, _p2;  
    private double _length;  
  
    ...  
    public double length() {  
        return _length;  
    }  
}  
  
class LineSegment {  
    private Point2D _p1, _p2;  
  
    ...  
    public double length() {  
        return _p1.distance(_p2);  
    }  
}
```

- Can you see how it can become inconsistent?
- Duplication may be **desirable** for **performance**, but don’t optimize **too soon**

“DRY” Principle

- Don't Repeat Yourself
- Strive to have each piece of knowledge in one place
- But don't go crazy over it
 - DRYing up at any cost can increase dependencies between code
 - “3 strikes and you refactor” (i.e., clean up)

Refactoring

- Refactor: to improve code's internal structure without changing its external behavior
- Most of the time we're modifying existing software
- “Improving the design after it has been written”
- Refactoring steps can be very simple:

```
public double weight(double mass) {    static final double GRAVITY = 9.80665;
    return mass * 9.80665;            }
public double weight(double mass) {
    return mass * GRAVITY;
}
```

- Other examples: renaming variables, methods, classes

Extract Method

- A comment explaining what is being done usually indicates the need to extract a method

```
public double totalArea() {
    ...
    // now add the circle
    area += PI * pow(radius,2);
    ...
}

public double totalArea() {
    ...
    area += circleArea(radius);
    ...
}

private double circleArea(double radius) {
    return PI * pow(radius, 2);
}
```

- One of the most common refactorings

Extract Method

- Simplifying conditionals with Extract Method

before

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
    charge = quantity * _winterRate + _winterServiceCharge;
}
else {
    charge = quantity * _summerRate;
}
```

after

```
if (isSummer(date)) {
    charge = summerCharge(quantity);
}
else {
    charge = winterCharge(quantity);
}
```

Refactoring & Tests

- Eclipse supports various refactorings
- You can refactor manually
 - Automated tests are essential to ensure external behavior doesn't change
 - Don't refactor manually without tests!
- More about tests and how to drive development with tests next week

