Software Design Principles

Carl Erickson
Atomic Object
SRP: Single Responsibility

A class should have only one reason to change

• Change ripples through the system when you violate SRP
  – rectangle example
Rectangle: Bad Design
Exercise

What’s a better design?
Rectangle: Better Design
Observations on SRP

- Not always easy to see
  - modeling fidelity may lead to it
  - perspective of abstraction influences it
  - modem interface example

- Modem example

```java
interface Modem {
    public void dial(String num);
    public void hangup();
    public void send(char c);
    public char recv();
}
```
OCP: Open-Closed

*Software should be open for extension, but closed for modification*

- The system can be extended to handle change without having to be modified
- Refactoring towards OCP means future changes are handled without modification
- Abstraction makes OCP possible
Exercise: Shape

• Create a procedural C design to
  – represent generic shapes
  – represent circles and squares
• Sketch a function DrawAllShapes() which takes a list of shapes and draws them
• Exercise
  – Work in pairs
  – Use “reflective articulation”
Shape: Bad Design

- C implementation
  - Rigid: add triangle, recompile/redeploy every class and module that uses them
  - Fragile: hard to find all the places that must be changed
  - Immobile: using DrawAllShapes in a new program requires bring Circle and Square along
Exercise: OO Shapes

Redesign your shape problem using some OOPL
  – take advantage of polymorphism
  – make it extensible for new shapes
Shape: Better Design

• OO implementation
  – Robust: no need to hunt down code that will need to be changed (DrawAllShapes)
  – Flexible: no existing source files need to be modified
  – Reusable: DrawAllShapes can be used without Square or Circle
Observations on OCP

• What change are you closed against?
  – Can’t be closed to every change
    • adding shape vs ordered drawing
  – Depends on the model you use
  – Depends on what you need

• Abstraction has a cost
  – so we take the first bullet
  – do things to get shot at early and often
LSP: Substitutability

Subtypes must be substitutable for their base types.

• Everything true of the base type should be true of the subtype.
  – passing subtype objects to functions written with base type references should work fine
public class Rectangle {
    public void setWidth(double w);
    public void setHeight(double h);
    double width;
    double height;
    ...
}

public class Square extends Rectangle {
    ...
}
Smells

• Don’t need both height and width
  – maybe the waste of memory doesn’t matter

• What about setting height and width?
  – They must be the same for a square
    void setHeight(float h) {
      height = h;
      setWidth(h);
    }

• Could clients of Rectangle have problems with Squares?
Substitutability Problem

- What about this usage?
  ```java
  void testArea(Rectangle r) {
    r.setWidth(5);
    r.setHeight(10);
    assertTrue(50, r.area());
  }
  ```

- When Squares can’t be substituted for Rectangle you are violating LSP
Observations on LSP

• Validity is not intrinsic
  – every class or design assumes some model
  – you can’t judge a model in isolation
  – the strength of a design depends on the assumptions made by the user

• Needless Complexity vs Fragility
  – there is a tension between these

• Violation of LSP may cause violation of OCP
DIP: Dependency Inversion

*High level things should not depend on low level things. Both should depend on abstractions.*

- Layering example the wrong way
  - Policy -> Mechanism -> Utility

- The right way
  - Policy -> Policy Service Interface <- Mechanism
Heuristic

• Program to an interface, not an implementation
• Variables should not be of concrete class type
• No class should derive from a concrete class
• No subclass should override a method implemented in its base class
• Clients own the interface, not servers
Two Common Approaches

Two common ways to invert dependencies

• Template Method Pattern
  – inheritance

• Strategy Pattern
  – composition
1. Analyze the classic implementation
   a. What is the high level aspect of this code?
   b. What are the low level details of this code?
   c. Are high and low level separated?
   d. Can we sort doubles with this code?
Design of Classic

1. The classic implementation
   a. Idea of sorting is high level, independent of any particular data type.
   b. The specific data type being sorted (integer) is a low-level detail.
   c. Not at all. Intimately coupled.
   d. Not without copy/paste.
Better Design: Template

1. Apply Template Method
   a. Inheritance of high level algorithm by low level details.
   b. Specific subclasses for each data type.
   c. Abstract base class + concrete subclasses
BubbleSorter by Template

Add a new data type
no change to existing (closed)
extend with new subclass (open)
Very tightly coupled.

Reuse of outOfOrder() and swap() in other sort algorithms impossible.
Better Design: Strategy

1. Apply Strategy
   a. Interface (SortHandler) for doing sorting.
   b. Data-specific subclasses which implement the interface.
   c. BubbleSorter composes a SortHandler, delegates work to it.
   d. Client chooses subclass, instantiates BubbleSorter, calls sort()
BubbleSorter by Strategy

Add a new data type
no change to existing (closed)
extend with new subclass (open

Handler subclasses know nothing of BubbleSorter
(so can be used with other sort algorithms)
Detailed (low level) implementations can be manipulated by high level algorithm
ISP: Interface-Segregation

Clients should not be forced to depend on methods they do not use.

- Classes have a tendency to grow “fat”
  - unrelated groups of methods
  - different aspects for different clients

- Clients should not know this
  - separate, cohesive, related interfaces
ATM Example

- Interface will have multiple technologies
- Several types of transactions
Integrated ATM

- ATM UI
  - Screen UI
  - Braille UI
  - Speech UI

- Transaction
  - Deposit Transaction
  - Withdrawal Transaction
  - Transfer Transaction

- UI
  - {interface}
  - requestDepositAmount()
  - requestWithdrawalAmount()
  - requestTransferAmount()
  - informInsufficientFunds()
Segregated ATM
Client Changes

• Change an interface
  – of course we know this is going to be costly
• But change comes from clients
  – pressure for new or different functionality
  – the interface may need to change
• Client coupling
  – overly fat interfaces cause change coupling between unrelated clients