Course Application Design

The SOLID Principles

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Part four

The SOLID design principles

SOLID

- It’s an acronym of the five principles introduced by Mr. Robert Martin (commonly known as Uncle Bob):
  - Single responsibility principle
  - Open-closed principle
  - Liskov substitution principle
  - Interface segregation principle
  - Dependency inversion principle
- When all five principles are applied together it is more likely that a system is created that is easy to maintain and extend over time.
- Look at http://hackerchick.com/tag/solid/ for a really good overview!

Single Responsibility Principle (SRP)

A class should have only one reason to change

- Later this was extended to “Every software module should have only one reason to change”
  - Software Module == Class, Function etc.
  - Reason to change == Responsibility

SRP violation

```
public class User {
    public void vote(
        String message, int rank){
        //user votes on some topic
    }

    public JTable creatVotesTable(){
        //creates a JTable for vote objects
    }
}
```

- Every time voting logic changes, this class will change
- Every time table format changes, this class will change
- When another UI technology is chosen, this class will change

Remember this slide?

A more subtle SRP violation

```
public void setZipCode(String zipCode) {
    if (isCorrectZipCode(zipCode)) {
        parseZipCode(zipCode);
    } else {
        throw new IllegalArgumentException(”Wrong zipcode " + zipCode);
    }
}
private boolean isCorrectZipCode(String zipCode) {
    if (null == zipCode || zipCode.length() == 0) {
        return false;
    }
    return zipCode.trim().matches("\\d{4} ?[A-Za-z]{2}");
}
private void parseZipCode(String zipCode) {
    zipCode = zipCode.trim();
    int offset = 0;
    if (zipCode.contains(" ")) {
        offset = 1;
    }
    this.zipCodeArea = Integer.parseInt(zipCode.substring(0, 4));
    this.zipCodeLocal = zipCode.substring(4 + offset);
}
```

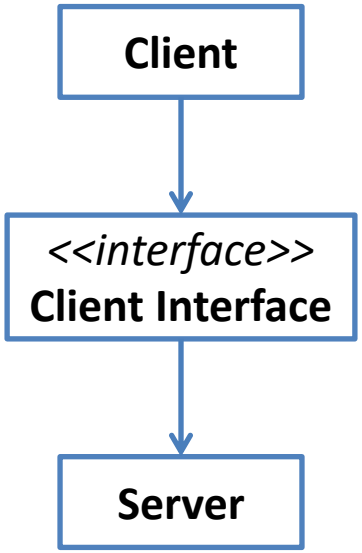
Open Close Principle (OCP)

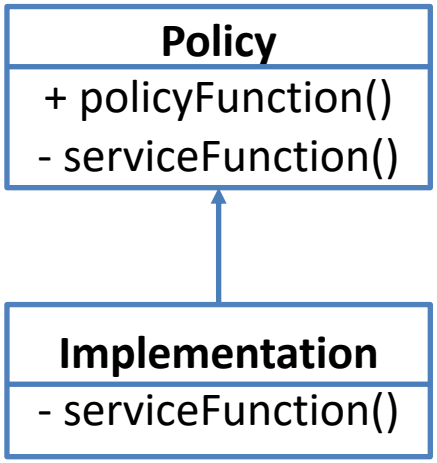
Software entities (classes, modules, functions etc.) should be open for extension, but closed for modification

- such modules allow their behavior to be modified without altering their source code
- usually through
  - Abstraction (coding against interfaces)
  - Dynamic binding (compiler figures out the object type at runtime -- polymorphism)

OCP

- Design patterns associated to the OCP:
  - Strategy Pattern
  - Template Method Pattern





OCP heuristics

- Make all object data private
  - changes to public data are always at risk to ‘open’ the module
  - all clients of a module with public data members are open to one misbehaving module
  - errors can be difficult to find and fixes may cause errors elsewhere
- No global variables
  - it is impossible to close a module against a global variable

Liskov substitution principle (LSP)

- Subclasses should be substitutable for base classes: **Derived classes must be usable through the base class interface without the need for the user to know the difference**
- Make sure that new derived classes are extending the base classes without changing their behavior

LSP example

inheritance has its limits

```
public abstract class Bird {
    public abstract void fly();
}

public class Parrot extends Bird {
    public void fly() { /* implementation */ }
    public void speak() { /* implementation */ }
}

public class Penguin extends Bird {
    public void fly() {
        throw new UnsupportedOperationException();
    }
}
```

LSP example cont.

inheritance has its limits

```
public static void playWith(Bird bird) {
    bird.fly();
}

public static void main(String[] args) {
    Parrot myPet = new Parrot();
    // myPet "is-a" bird and can fly()
    playWith(myPet);
    Penguin myOtherPet = new Penguin();
    // myOtherPet "is-a" bird and cannot fly()?!
    playWith(myOtherPet);
}
```

What went wrong?

- We did not model ‘Penguins cannot fly’
- We modelled ‘Penguins may fly, but if they try it is an error
- The design fails LSP
  - A property assumed by the client about the base type does not hold for the subtype
  - **Penguin** cannot be a subtype of **Bird**
- Subtypes must respect what the client of the base class can reasonably expect about the base class
  - But how can we anticipate what some client will expect?

LSP: Simple Heuristic

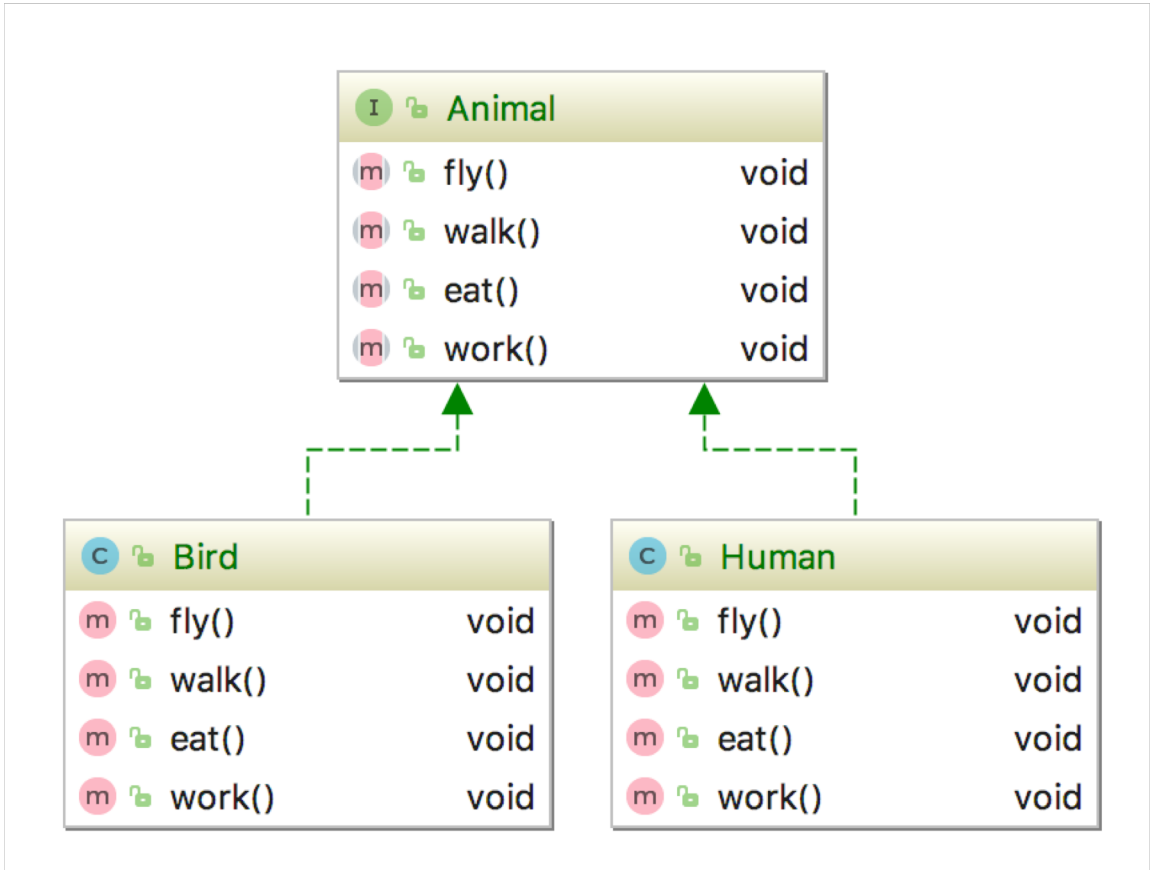
- Telltale signs of LSP violation:
  - Degenerate functions in derived classes (i.e. overriding a base-class method with a method that does nothing)
  - Throwing exceptions from derived classes
- Solution 1: inverse the inheritance relation if the base class has only additional behavior
- Solution 2: extract a common base class if both initial and derived classes have different behaviors

ISP: The Interface segregation Principle

- “Separate interfaces so callers are only dependent on what they actually use”
- Or, more simply put: “Avoid “fat” interfaces”
- Related to SRP

ISP example

Bird IS-A animal and human is-a animal, but what should humans do when asked to fly(), or Birds when asked to work()?

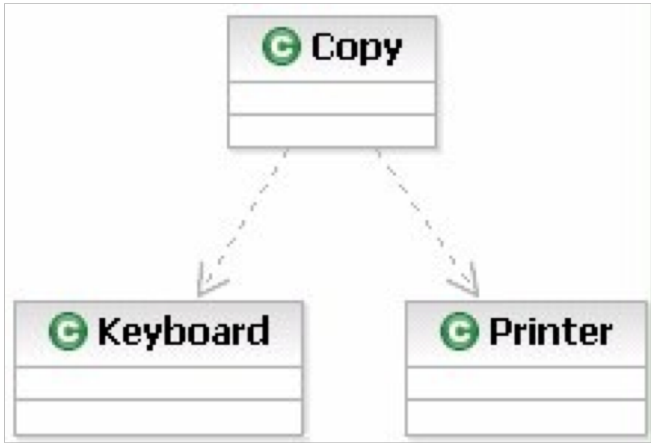


DIP: The Dependency-Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions
- Abstractions should not depend on details. Details should depend on abstractions

DIP goal: Have all of the arrowheads land on abstractions

Bad



Good