Design Principles

Week 8 Lecture 2
September 15, 2006

Design smells…

- **Rigidity** – The design is hard to change
- **Fragility** – The design is easy to break
- **Immobility** – The design is hard to reuse
- **Needless complexity** – The design contains infrastructure that adds no direct benefit
- **Needless repetition** – The design contains repeating structure that could be unified under a single abstraction
- **Opacity** – It is hard to read and understand.

**Rigidity**

- The tendency for software to be difficult to change, even in simple ways.
- A design is rigid if a single change causes a cascade of subsequent changes in dependent modules.

**Fragility**

- The tendency of a program to break in many places when a single change is made.
- Often, the new problems are in areas that have no conceptual relationship with the area that was changed
- There are modules that are constantly in need of repair, the more you fix them, the worse they get.

**Immobility**

- A design is immobile when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great

**Needless complexity**

- A design contains needless complexity when it contains elements that aren’t currently useful.
- Developers anticipate changes to the requirements, and put facilities in the software to deal with those imaginary changes
- Too many contingencies make design becomes littered with constructs that are never used
Needless Repetition

- Cut and paste are disastrous code-editing operations.
- Developers working independently might try to find usable code from other's code which again might be from somebody else code.
- Finding all repetition and eliminating it with an appropriate abstraction may never be high on developers' priority list.
- Redundant code make the job of changing arduous.

Opacity

- The tendency of a module to be difficult to understand.
- Code that evolves over time tends to become more and more opaque with age
- Require constant effort to keep the code clear and expressive

Design to handle changing requirements

- SRP
  - The Single Responsibility Principle
- OCP
  - The Open-Closed Principle
- DIP
  - The Dependency Inversion Principle

SRP: the Single-Responsibility Principle

- Rectangle class has two sets of responsibilities
- Problems
  - We must include the GUI in the Computational Geometry application
  - If we change the way of rendering the rectangle class, the change will force us to rebuild, retest and redeploy the Computational Geometry Application.

Separate the two sets of responsibilities

- A class should have only one reason to change
- Responsibility set is an axis of change

OCP: the Open-Closed Principle

- Software entities (classes, modules, functions, etc) should be OPEN for extension, but CLOSED by modification
  - Behavior of the module can be extended
  - Extending the behavior of a module does not result in changes to the source or binary of the module
  - When a single change to a program results in a cascade of changes to dependent modules, the design smells of Rigidity
Abstraction is the key

- Client and Server are all concrete classes
  - Does not conform to OCP
  - If we add a new server, Client has to be changed
- Client uses ClientInterface
  - Conform to OCP
  - New server can be added by implementing the interface

The Shape application

- We need an application that must be able to draw circles and rectangles on a standard GUI. The circles and rectangles must be drawn in a particular order. A list of the circles and rectangles will be created in the appropriate order, and the program must walk the list in that order and draw each circle or rectangle.

The Shape application design – does not conform to OCP

The Shape application design -- OCP

Change is coming

- Figure in previous slide is closed against changes like adding new shapes
- We decide that all Circles should be drawn before any Rectangles.
- The display function is not closed against a change like this!

Anticipation and “Natural” structure

- The abstraction we used are more of a hindrance to this kind of change than a help.
- The model is not natural in a system where ordering is more significant than shape type.
- There is no model that is natural to all contexts
- Closure can not be complete, must be strategic. It is designer’s responsibility to choose which change to close against
Anticipation and “natural” structure (cont)
- Confirming to OCP is expensive
- Takes time and effort to create appropriate abstractions
- Too many abstractions increase the complexity of the software design

How do we protect ourselves from changes
- Avoid needless complexity
- Don’t load the design with lots of unnecessary abstraction
- “Fool me once, shame on you. Fool me twice, shame on me”
  - Permit ourselves to be fooled once
  - We wait until changes happen!
- Stimulating change
  - Write test first
  - Use iterative methods
  - Release software early and often

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

Layering
- All well-structured object-oriented architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interface

An inversion of ownership
- Utility libraries do not own their own interfaces
- Don’t call us, we’ll call you

Abstract Class vs. Interface
- At a superficial (language) level
  - Abstract classes allow for common state and behaviour
  - Java, C#: you can derive (inherit) from only one class
- At design level
  - Abstract class: how concrete items should be designed to be used in the same way
  - Interface: what is the interface these things must have in common if they are going to be used in the same way
A simple example

Dependency inversion can be applied wherever one class sends a message to another.

The “Copy” program: example using OCP and DIP

Problem
Your boss comes to you early Monday morning and asks you to write a program that copies characters from the keyboard to the printer.

Your response
Hmm... around 10 lines of code
Give me three weeks

The "Copy" program: initial design

The “Copy” program – change request!

Problem
3 months later
Your boss comes to you and says that sometimes they’d like the Copy program to be able to read from the paper tape reader.

Your response
Dame it!
You should do your analysis job!
Plan the modification

Second version of "Copy"

The “Copy” program – change again!

Problem
3 weeks later
Your boss tells you that the customers would sometimes like the Copy program to output to the paper tape punch.

Your response
Writing software would be a lot easier if it weren’t for customers!
I’ll do it using another?: operation but my design are ruined
Any more change will collapse the loop permanently.
Perhaps it is time to dust off my resume...
A better design

```java
interface Reader{
    public String read();
}
class Copier{
    public void copy(Reader reader){
        String s;
        Printer printer = new Printer();
        while (s = reader.read())!=null)
            printer.writePrt(s);
    }
}
class Keyboard implements Reader{
    public String read(){
        return "hello";
    }
}
```

Improved design of copy