Design Principles

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

Outcomes

After today's lecture you will be able to:

- Understand the basic principles of Object Oriented Design
- Be capable of applying these principles in you daily development

Idaho State University
Computer Science

Inspiration

"Most of you are familiar with the virtues of a programmer. There are three, of course: laziness, impatience, and hubris." – Larry Wall

ROAR

Design Principles

- This lecture is a collection of design principles for making better software.
- Every great programmer has a toolbox of design principles they use to help them produce great code
- Yes, these principles are admittedly fuzzy and not mutually exclusive
- They must be learned by specific coding examples/experiences

Basic A&D Principles

Here are some basic design principles you probably have already heard about:

- Well-designed software is easier to debug, change and extend.
- Code to interfaces, not implementations
- Share common behavior via inheritance
- If design is proving to be inflexible, refactor it to restore it to be a good design (Refactoring is a lecture topic on its own later)
- Make classes cohesive: class should have a single, clearly stated purpose which fits its name and all of its fields and methods.
  - Similarly at the lower level of methods: the name should be (all that) it does.
  - We will cover a similar principle below, the Single Responsibility Principle (SRP).

Idaho State University Computer Science

Separation of Concerns

- Separation of Concerns (SoC)
  - don’t have many different concerns in one class; instead, different tasks/aspects should be in different classes/functions.
  - also related to SRP below, SRP is “one concern per class”

ROAR

Encapsulate What Varies

aka encapsulate code that changes a lot. This basic O-O principle you may not know as well, here is a brief overview.

- One way that ugly code arises is new code has to be patched in as new features are added
- If you push code deeper into classes and behind encapsulation boundaries the change is isolated, code is more maintainable.

Encapsulate What Varies

Original smelly code, any changes to policy on when customer can check out a book or when a book is available requires change to **checkoutBook** method:

```
var library = {
  checkoutBook: function (customer, book) {
    if (customer && customer.fine <= 0.0 && customer.card &&
        customer.card.expiration === null && book && !book.isCheckedOut &&
      (!book.reserveDate || book.reserveDate.getTime() > (new Date()).getTime())) {
      customer.books.push(book);
      book.isCheckedOut = true;
    }
    return customer;
  }
};
```

Encapsulate What Varies

Improved code: pull out the concepts of a customer that **canCheckoutBook** into its own method, and similarly for a book that **isAvailable** (plus, in turn pull out even more methods **hasFine** and **hasActiveLibraryCard** etc from those actions):

```
var library = {
  checkoutBook: function (customer, book) {
    if (customer.canCheckoutBook() && book.isAvailable()) {
      customer.checkout(book);
    }
    return customer;
  }
};
```

Encapsulate What Varies

```javascript
var customer = {
  canCheckoutBook: function () {
    return !this.hasFine() && this.hasActiveLibraryCard();
  },
  hasFine: function () {
    return this.fine > 0.0;
  },
  hasActiveLibraryCard: function () {
    return this.card !== null && this.card.expiration === null;
  },
  checkout: function (book) {
    //implementation
  }
};
```

Encapsulate What Varies

```
var book = {
  isAvailable: function () {
    return !this.isCheckedOut && !this.isReserved();
  },
  isReserved: function () {
    return this.reserveDate !== null &&!this.isFutureReserve();
  },
  isFutureReserve: function () {
    return this.reserveDate.getTime() > (new Date()).getTime();
  }
};
```

The **library** code is now not needing to change at all if there is a change in the policy on when customers can check out books or what defines a book being available – we Encapsulated what Varied - !

Patterns and Anti-Patterns

- The term "design pattern" means a particular structure and relationship between objects that is a common good pattern in object-oriented programming – the term originates from the Design Patterns book
- A bad pattern that takes you in the opposite direction of where you should be going is an anti-pattern aka bad smell

God Class Anti-Patterns

One easy trap to fall into is data-centric design

- A data-centric design has classes with no meaningful methods – they are just passive data holders
- Data-centric designs usually have one really fat class doing all the operations (the “God Class”) and a bunch of other classes with no real methods (the “Data Classes”)
- C programmers often produce data-centric OO designs; the central class hold all the functions, and the little data classes are like C structs.
- Data-centric designs should be refactored to push methods from the big class in the center out to the data classes
- There is a principle hiding here, lets make up a name for it: Push Operations to the Data Classes! (PODC).

God Class Example

Here **library** is doing all operations on books/customers and not pushing out to book/customer classes

```
var library = {
  checkoutBook: function (customer, book) {
    if (customer && customer.fine <= 0.0 && customer.card &&
        customer.card.expiration === null && book && !book.isCheckedOut
      (!book.reserveDate || book.reserveDate.getTime() > (new Date()).ge
      customer.books.push(book);
      book.isCheckedOut = true;
    }
    return customer;
  }
};
```

Database vs objects

If you have persistent data in a database that can lead you to a data-centric design.

- Databases are all data and no code, so it is tempting to build corresponding objects which are just “data holders” – data-centric, **BAD**

Bad Example: in a chess game, having a **Board** class with only board position data and putting move legality checking code in e.g., **MoveController**. Don’t do that, put move logic with the **Board** (which also should be called something else since its more than board data)

The Open-Closed Principle (OCP)

Make code which is open for extension, but closed for modification

- i.e., keep your codebase easily extensible by isolating & limiting the spots that need to change
- (Note its related to Encapsulate what Varies)
- Inheritance with significant overriding can violate OCP since overriding is modifying – either
  - "Favor Composition over Inheritance" (another principle, btw) – use composition to "plug in" the part that incorporates the extension
  - Allow subclassing but declare nearly all methods **final** to greatly limit or eliminate overriding
- OCP makes code more reliable since complex interdependencies don't have random changes injected into them by outsiders.

Idaho State University
Computer Science

OCP Example

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}

public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

OCP Example

```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

OCP Example – Fixed

```
public abstract class Shape
{
    public abstract double Area();
}
```

OCP Example – Fixed

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

OCP Example – Fixed

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

Don’t Repeat Yourself (DRY)

Avoid duplicate code – abstract out things in common to a single location

- Finding this smell is easy, nearly-identical code blocks will Repeat
- The problem is if you don’t abstract it out, you have two parallel codebases to try to keep consistent and you often fail
- And maybe two copies turns into three turns into four before you realize what is happening.
- You can solve it by moving code to a common method, making a common superclass, etc.

DRY Example

Single Responsibility Principle (SRP)

Classes should not have more than one focus of responsibility

- Classes can reasonably be involved in different interactions, it is the focus that is the issue.
- This principle is similar to the cohesion principle and Separation of Concerns (SoC) principle
- Data-centric designs always violate this principle: the fat class in the middle with all the methods has many different foci.

SRP Analysis

- A way to figure out if certain methods belong in a given class:
- If not, move them to another existing, or new, class

```
class Automobile
    def start()
    def stop()
    def changeTires()
    def drive()
    def wash()
    def checkOil()
    def getOil()
end
```

– for each method **X**, ask, "does the **Automobile** have primary responsibility for **X**-ing?" If the answer is no, the method doesn't belong.

Liskov Substitution Principle (LSP)

```
public class Rectangle
{
    int height, width;
    Rectangle(int w, int h) { height = h; width = w; }
    public int getHeight() { return height; }
    public int getWidth() { return height; }
    public int setHeight(int h) { height = h; }
    public int setWidth(int w) { width = w; }
    public int findArea()
    { return getHeight() * getWidth(); }
}
```

LSP Example

```java
public class Square extends Rectangle
{
    Square(int s) { new Rectangle(s,s); }
    // set both to preserve square-ness
    public int setSide(int h) { width = h; height = h; }
    public int setHeight(int h) { setSide(h); }
    public int setWidth(int w) { setSide(w); }
}
```

LSP Example

```
s = new Square(10);
 Rectangle r = s; // imagine this was a function call
                  // passing a Square to function asking
                  // for a Rectangle
//  equivalence r.getWidth() == (r.setHeight(4).getWidth())
//  should hold but would fail for the above
//      - bad attempt at Square is-a Rectangle.
```

LSP Example

- Does `Square` is-a `Rectangle` hold??
- First, we had to override width/height to try to keep square from becoming a non-square with `aSquare.setHeight(4);`
- But, the above code attempt violates rectangle invariant that setting a rectangle's width should not alter height: really bad code!
- If there is no mutation (the width/height fields are `final`) the is-a relationship is reasonable - that interface supports is-a fully

Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods (inherit from or implement) they don't use

- To be more precise, the bad methods are ones that not only don't they use now, they will never conceivably want to use them because they intuitively "don't belong": the interface is **too fat**
- These extra methods are "junk" and clutter the design space; more fundamentally, they are a sign that the class/method structuring is not correct.
- If you have this pattern it means you need to refactor, often by turning one interface into many.

ISP Example

- The Java Swing GUI library has different `Listener` interfaces for different events
- Mouse events have many types of events: clicking but also just cursor movement or wheel motion
- Most of the time programmers only care about clicks, not how the mouse is moving.
- If there was one single Mouse Listener interface users would need to write empty methods for the wheel/motion/etc events they don't care about
- Example Solution: Java Swing uses three separate interfaces for mouse events, `MouseListener`, `MouseWheelListener`, `MouseMotionListener` to "segregate" the types of mouse events; only `implement` the interfaces you need.

Dependency Inversion Principle (DIP)

Don’t depend on concrete classes, depend on abstractions

Don’t have high-level (user) code directly call/inherit from low-level (library) code; instead,

1. Library or component publishes an interface (or, if that is not possible, an abstract class)
2. Users write a class conforming to that interface (or extending abstract class), which then interacts with the other library classes.

This is an inversion: in traditional software the higher-level components directly invoke the lower-level ones: this principle inverts that since the user code now depends on a high-level interface: dependency inversion has taken place.

Why does DIP help?

- First, it allows different low-level implementations to be swapped out; as long as they implement the common interface all is well.
- More generally, it increases encapsulation
- This principle is widespread in well-written libraries; for example when you run the debugger on your Swing app you will see all these strange implementation class names you have never heard of which subclass or implement the class/interface you were interacting with.

Loose Coupling

Strive for loosely coupled designs of autonomous, interacting objects

Examples

- Swing **Listener**'s: the Swing event system and the user's action code need to know almost nothing about each other besides the methods on the listener.
- MVC in general illustrates the advantages of loose coupling.

LC Deeper Philosophy

There is a deeper principle here:

- The more complex the system the more loosely coupled, autonomous, and multi-layered it needs to be.
- Think of the human body for example: there are components, sub-components, sub-sub-components, etc.
- It wasn’t consciously designed that way, it emerged that way.

The Principle of Least Knowledge

Talk only to your immediate friends

- Don't dig deep inside your friends for friends of friends of friends and get in deep conversations with them – don't do `aWindow.getPane().getRasterizer().setUpdateFrequency(60)`
- Code is more convoluted if too many objects are directly interacting with one another, and bugs are more likely to be introduced as the code evolves over time.
- Solution: let the shared friend be an intermeidary instead of introducing lots of long-range dependencies. `aWindow.useHighUpdateFrequency()`
- This is related to the principle of loose coupling, things close are coupled tightly and things far are coupled loosely.

Are there any questions?