OO Design Principles

Dr. Isaac Griffith Idaho State University

Outcomes

Idaho State University | Computer Science

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
- Be capable of identifying violations of the SOLID principles
- Understand approaches to handle such violations

Design Principles

CS 2263

The Open-Closed Principle (OCP)

Idaho State University | Computer Science

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 inter-dependencies don't have random changes injected into them by outsiders.

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

Idaho State University | Computer Science

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

OCP Example – Fixed

Idaho State University | Computer Science

```
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

Idaho State University | Computer Science

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

    return area;
}
```

OCP Violations

Idaho State University | Computer Science

Finding

- Do you work directly with a concrete implementation rather than an abstraction?
- Does your code show issues when you extend classes to accommodate new functionality
- Other signs:
  - you have private methods that do nearly the same thing but with only slight variations
  - you use a lot of ifs/switches to control behavior
  - you use abstract classes but check for concrete implementations (`instanceof`) to control flow

Handling

- Apply the Strategy, Template Method, Decorator, or State Patterns

Liskov Substitution Principle (LSP)

Idaho State University | Computer Science

Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application.

- That requires the objects of your subclasses to behave in the same way as the objects of your superclass

- This can be achieved by following a couple of simple rules
  - An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass
    - You can have less restrictive validation rules, but not more restrictive ones
    - Otherwise, any call on this method via a superclass typed variable might lead to an exception
  - The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass
    - You may apply a stricter type (more specific subtype) than than the super class, but not a more general type than the superclass provides

LSP Example

Idaho State University | Computer Science

```
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();
    }
}
```

```
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

Idaho State University | Computer Science

```
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

LSP Violations

Idaho State University | Computer Science

Finding

- Look for subtypes that don't behave the same way as their base classes
- Check inheritance hierarchies that are deeply nested

Handling

- Reduce inheritance using composition

Interface Segregation Principle (ISP)

Idaho State University | Computer Science

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.

ISP Violations

Finding

- Look for interfaces with more than a few methods
  - Do the methods form a cohesive group and are all used by clients?
- Look for deep layers of inheritance

Handling

- Split the interfaces into multiple, smaller interfaces that define a specific need or role (also called RoleInterfaces)

Dependency Inversion Principle (DIP)

Idaho State University | Computer Science

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.

DIP Violations

Idaho State University | Computer Science

Finding

- Check dependencies between modules
  - If a high-level abstract module depends on a low-level module, that is a violation
- Look at areas where it is difficult to add or remove a low-level part of the application
- Look at areas that are difficult to test

Handling

- Replace the use of concrete types for variables with interfaces
- Add in new interfaces as abstractions to concrete classes
- Utilize the Dependency Injection approach or Factory Patterns

Other Principles

CS 2263

Basic A&D Principles

Idaho State University | Computer Science

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).

Don’t Repeat Yourself (DRY)

Idaho State University | Computer Science

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.

Loose Coupling

Idaho State University | Computer Science

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.

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

Idaho State University | Computer Science

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

Idaho State University | Computer Science

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

Idaho State University | Computer Science

```
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

Idaho State University | Computer Science

```
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***!

Separation of Concerns

Idaho State University | Computer Science

- 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, SRP is "one concern per class"

The Principle of Least Knowledge

Idaho State University | Computer Science

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 intermediary 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.

For Next Time

Idaho State University | Computer Science

- Review this Lecture
- Read Chapter 5 from the book
- **Continue working on Project Part 1**
- Come to Class

Are there any questions?