Object-Oriented Design

Christopher Simpkins
chris.simpkins@gatech.edu

Object-Oriented Design

Engineering design

> *A systematic, intelligent process in which designers generate, evaluate and specify designs for devices, systems, or processes whose form(s) and function(s) acheive clients' objectives and users' needs while satisfying a specified set of constraints. – Dym and Little, quoted in Carlos Otero, Software Engineering Design*

Object-oriented design:

- identifying problem domain objects and representing them with classes (classes),
- factoring classes so they have the right granularity (responsibilities),
  - typically means defining class and interface hierarchies
- defining relationships between the classes (collaborations).

Object-Oriented Design Principles

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle (SRP)

A class should have only one reason to change.

- A responsiblity is an axis of change.
- Classes with multiple responsiblities will couple the responsibilities, making changes to one responsilbity more difficult due to the risk of breaking the other responsibilities handled by the class.

SRP Example - Too Many Responsibilities

```java
public class GreetingFrame extends JFrame implements ActionListener {
    private JLabel greetingLabel;

    public GreetingFrame() {
        ...
        JButton button = new JButton("Greet!");
        button.addActionListener(this);
        ...
    }
    public void actionPerformed(ActionEvent e) {
        Greeter greeter = new Greeter("bob");
        String greeting = greeter.greet();
        greetingLabel.setText(greeting);
    }
}
```

- If we add other buttons or menu items to the GUI, we have to modify the `actionPerformed` method to handle an additional event source.
- If we chnage the behavior of the a button, we have to modify the `actoinPerformed` method.

SRP Example - Single-Responsibility Classes

```
private class GreetButtonListener implements ActionListener {
  private JLabel greetingLabel;

  public GreetButtonListener(JLabel greetingLabel) {
    this.greetingLabel = greetingLabel;
  }
  public void actionPerformed(ActionEvent e) {
    ...
  }
}
public class GreetingFrame extends JFrame {
  ...
  public GreetingFrame() {
    ...
    button.addActionListener(new GreetButtonListener(greetingLabel));
    ...
  }
}
```

- Additions to the UI require changes only to `GreetingFrame`.
- Changes to greet button behavior require changes only to `GreetButtonListener`.

Open–Closed Principle (OCP)

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

- Open for extension means the module should be extendable with new behavior.
- Closed for modification means the module shouldn't need to be touched in order to add the extension.

Object-oriented polymorphism makes this possible, namely, to write new code that works with old code without having to touch the old code.

Open for Modification

```
public class Sql {
  public Sql(String table, Column[] columns)
  public String create()
  public String insert(Object[] fields)
  public String selectAll()
  public String findByKey(String keyColumn, String keyValue)
  public String select(Column column, String pattern)
  public String select(Criteria criteria)
  public String preparedInsert()
  private String columnList(Column[] columns)
  private String valuesList(Object[] fields, final Column[] columns)
    private String selectWithCriteria(String criteria)
  private String placeholderList(Column[] columns)
}
```

This class violates SRP becuase there are multiple axes of change, e.g., updating an exising statement type (like create) or adding new kinds of statements. Extension requires opening the class for modification.

Open for Extension, Closed for Modification

```
public abstract class Sql {
  public Sql(String table, Column[] columns)
  public abstract String generate();
}
public class CreateSql extends Sql {
  public CreateSql(String table, Column[] columns)
  @Override public String generate()
}
public class SelectSql extends Sql {
  public SelectSql(String table, Column[] columns)
  @Override public String generate()
}
...
```

These classes are open for extension and closed for modification.

- Extend with new functionality by adding a new subclass of `Sql`.
- Existing classes need not be touched.

Note that with good OO design you often end up with many small classes instead of fewer large classses.

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their supertypes.

A suprising counter-example:

```
public class Rectangle {
  public void setWidth(double w) { ... }
  public void setHeight(double h) { ... }
}
public class Square extends Rectangle {
  public void setWidth(double w) {
    super.setWidth(w);
    super.setHeight(w);
  }
  public void setHeight(double h) {
    super.setWidth(h);
    super.setHeight(h);
  }
}
```

- We know from math class that a square “is a” rectangle.
- The overridden `setWidth` and `setHeight` methods in `Square` enforce the class invariant of `Square`, namely, that `width == height`

LSP Violation

Consider this client of `Rectangle`:

```
public void g(Rectangle r) {
  r.setWidth(5);
  r.setHeight(4);
  assert(r.area() == 20);
}
```

- Client assumed width and height were independent.

The Object-oriented `is-a` relationship is about behavior.

Conforming to LSP: Design by Contract

Author of a class specifies the behavior of each method in terms of preconditions and postconditions. Subclasses must follow two rules:

- Preconditions of overriden methods must be equal to or weaker than those of the superclass (enforces or assumes no more than the constraints of the superclass method).
- Postconditins of overriden methods must be equal to or greater than those of the superclass (enforces all of the constraints of the superclas method and possibly more).

  Require no more, promise no less.

In the Rectangle-Square case the postcondition of `Rectangle`'s `setWidth` method:

```
assert((rectangle.w == w) && (rectangle.height == old.height))
```

tells us that a `Square` doesn't satisfy the object-oriented *is-a* relationship to `Rectangle`.

Dependency Inversion Principle (DIP)[1]

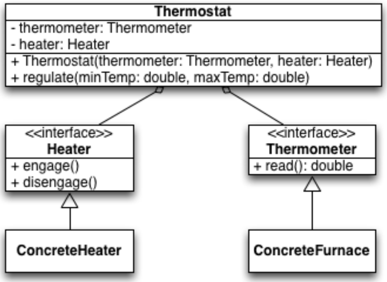
a. High-level modules should not depend on low-level modules. Both should depend on abstractions.
b. Abstractions should not depend on details. Details should depend on abstractions.

A counter-example:

```
public void regulate(double minTemp, double maxTemp) {
  for (;;) {
    while (in(THERMOMETER) > minTemp)
      Thread.sleep(1000);
    out(FURNACE, ENGAGE);

    while (inTHERMOMETER) < maxTemp)
      Thread.sleep(1000);
   out(FURNACE, DISENGAGE);
  }
}
```

- The `regulate` method depends on a particular thermometer and furnace implementation.
- The regulate algorithm is cluttered with low-level details of the

DIP Example



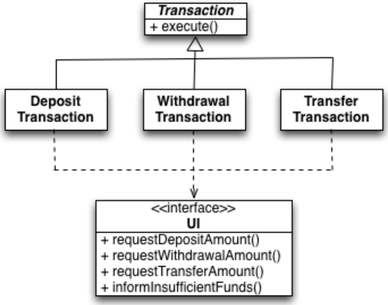
```java
public void regulate(double minTemp,
                     double maxTemp){
  for (;;) {
    while (thermometer.read() > minTemp)
      Thread.sleep(1000);
    heater.engage();

    while (thermometer.read() < maxTemp)
      Thread.sleep(1000);
    heater.disengage();
  }
}
```

- `Thermostat` depends on interfaces `Thermometer` and `Heater`
- Concrete heaters and thermometers also depend on `Thermometer` and `Heater`
- Low-level detail is removed form `regulate` method
- Can be extended with different concrete implementations of `Thermometer` and `Heater` while maintaining OCP

Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they don't use.

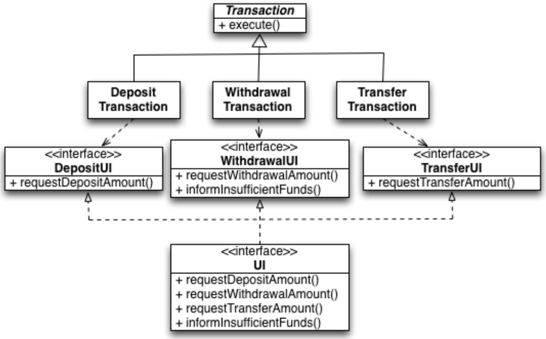
Break up fat interfaces into a set of smaller interfaces. Each client depends on the small interface it needs, and none of the others.



- Additional UI methods in `UI` require recompilation of all the transaction classes, even the ones that don't use the new methods.

ISP Example

Segregated UI interfaces:



- Each transaction gets its own UI interface.
- Adding transactions doesn't require touching or recompiling other transactions or UIs.