SOLID Design Principles

Agile Design

Fundamental principle of agile design:
The code is the design. – Jack Reeves, 1992

Design Smells

- Rigidity – system is too hard to change becuase change in one place forces changes in many other places
- Fragility – changes break things that are conceptually unrelated
- Immobility – too hard to resuse components in other systems
- Viscosity – hard to do it right, easy to do it wrong
- Needless Complexity – infrastructure with no direct benefit
- Needless Repetition – repeated structures that could be unified under a single abstraction
- Opacity – hard to read and understand

Design smells avoided or fixed by applying design principles like SRP, OCP . . .

SOLID Design Principles

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

These all boil down to (high) cohesion, (loose) coupling, and reuse.

SRP Counterexample – 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 change the behavior of the a button, we have to modify the `actionPerformed` method.

SRP Refactoring

```java
private class GreetButtonListener implements ActionListener {

  private JLabel greetingLabel;

  public GreetButtonListener(JLabel greetingLabel) {
    this.greetingLabel = greetingLabel;
  }
  public void actionPerformed(ActionEvent e) {
... }
}
```

```java
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

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.

OCP Counterexample – Extension Requires 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 with new SQL query types requires modifying this class.

OCP Refactoring

Abstract base class that doesn't change:

```
public abstract class Sql {
  public Sql(String table, Column[] columns)
  public abstract String generate();
}
```

Extended by adding new subclasses without touching other classes:

```
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()
}
```

This is high cohesion, low coupling, and reuse of the interface declared in the base class.

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their supertypes.

Most important principle in object-oriented design

LSP Counterexample

A suprising counter-example:

```java
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 (author of `g`) assumes width and height are independent in `r` because `r` is a `Rectangle`.
- If the `r` passed to `g` is actually an instance of `Square`, what will be the value of `r.area()`?

The Object-oriented `is-a` relationship is about behavior. `Square`'s `setWidth` and `setHeight` methods don't behave the way a `Rectangle`'s `setWidth` and `setHeight` methods are expected to behave, so a `Square` doesn't fit the object-oriented *is-a* `Rectangle` definition. Let's make this more formal ...

Conforming to LSP: Design by Contract

Require no more, promise no less.

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).
- Postconditions of overriden methods must be equal to or greater than those of the superclass (enforces all of the constraints of the superclass method and possibly more).

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

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

cannot be satisfied by `Square`, which tells us that a `Square` doesn't satisfy the object-oriented *is-a* relationship to `Rectangle`.

Georgia Tech

LSP Conforming 2D Shapes

```
public interface 2dShape {
    double area();
}
public class Rectangle implements 2dShape {
    public void setWidth(double w) { ... }
    public void setHeight(double h) { ... }
    public double area() {
        return width * height;
    }
}
public class Square implements 2dShape {
    public void setSide(double w) { ... }
    public double area() {
        return side * side;
    }
}
```

Dependency Inversion Principle

a. High-level modules should not depend on low-level modules. Both should depend on abstractions.

This basically means program to an interface, not a particular implementation of the interface.

Dependency Inversion Principle

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.

This basically means program to an interface, not a particular implementation of the interface.

DIP Counterexample[^1]

```java
public class RealBillingService {
  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    PaypalCreditCardProcessor processor = new
         PaypalCreditCardProcessor();
    // Card charging code ...
  }
}
```

- Dependence on particular implementation of credit card processor

`new` is a code smell.

[^1] https://github.com/google/guice/wiki/Motivation

DIP Refactoring

```java
public interface CreditCardProcessor { ... }

public class RealBillingService {
  private final CreditCardProcessor processor;

  public RealBillingService(CreditCardProcessor processor) {
    this.processor = processor;
  }

  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) {
    // Credit card charging code ...
  }
}
```

- Now `RealBillingService` depends on the `CreditCardProcessor` interface, not any particular implementation

Dependency Injection

```java
public interface CreditCardProcessor { ... }

public class RealBillingService {
  private final CreditCardProcessor processor;

  public RealBillingService(CreditCardProcessor processor) {
    this.processor = processor;
  }

  public Receipt chargeOrder(PizzaOrder order, CreditCard creditCard) { ... }
}
```

Note that we've eliminated `new` by passing an instance of `CreditCardPricessor` in the constructor

- This now satisfies the OCP because we can extend `RealBillingService` to work with additional `CreditCardProcessor`s without modifying `RealBillingService`
- Wiring a class to its concrete dependencies external to the class is known as *dependency injection* and it gets much fancier than the manual approach shown here

Interface Segregation Principle

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.

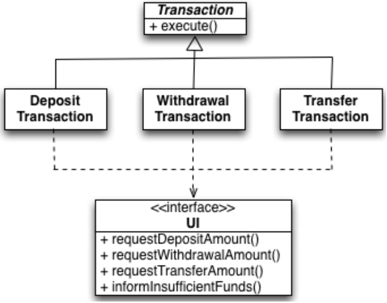

Figure 1: Fat UI Interface

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

ISP Refactoring

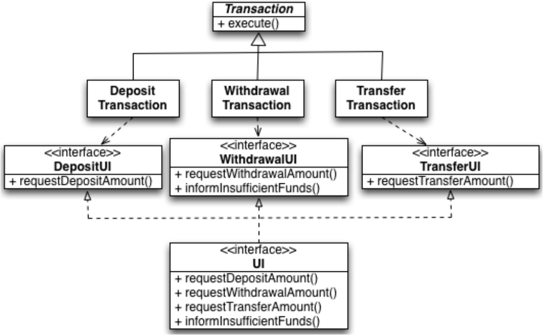

Figure 2: Segregated UI Interfaces

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

Conclusion

Design is art and science. If something smells, fix it.