Patterns

Really? Patterns?

Lets start somewhere

Why do I care?

Software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source code. It is a description or template for how to solve a problem that can be used in many different situations. Design patterns are formalized best practices that the programmer can use to solve common problems.

- Reduce code duplication
- Apply well known recipes to common issues
- Pattern recognition – maintenance
- Next level of mastery?

Singleton

- Ensure that only one instance of a class is created and provide a global access point to the object.

Intent

- Singleton pattern should be used when we must ensure that only one instance of a class is created and when the instance must be available through all the code. A special care should be taken in multi-threading environments when multiple threads must access the same resources through the same singleton object.

Usage

- Logger Classes
- Configuration Classes
- Accessing resources in shared mode

```
class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {
        System.out.println("Singleton(): Initializing Instance");
    }

    public static Singleton getInstance() {
        return instance;
    }

    public void doSomething() {
        System.out.println("Singleton does something!");
    }
}
```

Example

Builder

- Helps create complex objects

Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations
- Can be used for objects that contain flat data (html code, SQL query, X.509 certificate...) - data that can't be easily edited. This type of data cannot be edited step by step and must be edited at once.

Usage

- Build Object/s
- Reduce params in constructor

```
public class Car {
    public int wheels;
    public String color;

    public Car() {}

    public int getWheels() {
        return wheels;
    }
    public void setWheels(int wheels) {
        this.wheels = wheels;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
}

CarBuilder builder = CarBuilder.create();
Car car = builder.withColor("blue").withWheels(4).build();
```

```
public class CarBuilder {
    private int wheels;
    private String color;
    public static CarBuilder create() {
            return new CarBuilder();
    }

    public CarBuilder withWheels(int wheels) {
        this.wheels = wheels;
        return this;
    }

    public CarBuilder withColor(String color) {
        this.color = color;
        return this;
    }

    public Car build() {
        Car car = new Car();
        car.setColor(color);
        car.setWheels(wheels);
        return car;
    }
}
```

Example

Template

- Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Let subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Intent

- Let subclasses implement (through method overriding) behavior that can vary.
- Avoid duplication in the code: the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in each of the subclasses.
- Control at what point(s) subclassing is allowed. As opposed to a simple polymorphic override, where the base method would be entirely rewritten allowing radical change to the workflow, only the specific details of the workflow are allowed to change.

Usage

- Reduce duplication among sibling classes that have same operations

```
abstract class Generalization {
  public void findSolution() {
    stepOne();
    stepTwo();
    stepThr();
    stepFor();
  }
  protected void stepOne() {
   System.out.println( "Generalization.stepOne" );
  }
  abstract protected void stepTwo();
  abstract protected void stepThr();
  protected void stepFor() {
   System.out.println( "Generalization.stepFor" );
 }
}


class TemplateMethodDemo {
  public static void main( String[] args ) {
    Generalization algorithm = new Realization();
    algorithm.findSolution();
   }
}
```

```
abstract class Specialization extends Generalization {
   protected void stepThr() {
      step3_1();
      step3_2();
      step3_3();
   }
   protected void step3_1() {
     System.out.println( "Specialization.step3_1" );
   }
   abstract protected void step3_2();
   protected void step3_3() {
     System.out.println( "Specialization.step3_3" );
   }
}

class Realization extends Specialization {
   protected void stepTwo() {
     System.out.println( "Realization   .stepTwo" );
   }
   protected void step3_2() {
     System.out.println( "Realization   .step3_2" );
   }
   protected void stepFor() {
      System.out.println( "Realization   .stepFor" );
      super.stepFor();
  }
}
```

Example

Strategy

- Enables an algorithm behavior to be selected at runtime

Intent

- A class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known for each case until run-time, and may require radically different validation to be performed.

Usage

- Defines a family of algorithms
- Encapsulates each algorithm
- Makes the algorithms interchangeable within that family

Example

```java
interface BillingStrategy {
  public double getActPrice(double rawPrice);
}

class NormalStrategy implements BillingStrategy {
  @Override
  public double getActPrice(double rawPrice) {
    return rawPrice;
  }
}

class HappyHourStrategy implements BillingStrategy {
  @Override
  public double getActPrice(double rawPrice) {
    return rawPrice*0.5;
  }
}



public static void main(String[] args) {
                Customer a = new Customer(new NormalStrategy());
                // Normal billing
                a.add(1.0, 1);
                // Start Happy Hour
                a.setStrategy(new HappyHourStrategy());
                a.add(1.0, 2);
}
```

```java
class Customer {
        private List<Double> drinks;
        private BillingStrategy strategy;

        public Customer(BillingStrategy strategy) {
                this.drinks = new ArrayList<Double>();
                this.strategy = strategy;
        }
        public void add(double price, int quantity) {
                drinks.add(strategy.getActPrice(price * quantity));
        }
        public void printBill() {
                double sum = 0;
                for (Double i : drinks) {
                        sum += i;
                }
                System.out.println("Total due: " + sum);
                drinks.clear();
        }
        // Set Strategy
        public void setStrategy(BillingStrategy strategy) {
                this.strategy = strategy;
        }
}
```

Adapter

- Helps two incompatible interfaces to work together

Intent

- It is often used to make existing classes work with others without modifying their source code.

Usage

- Convert the interface of a class into another interface clients expect
- Wrap an existing class with a new interface.
- Match an old component to a new system

```
class LegacyLine {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("line from (" + x1 + ',' + y1 + ") to (" + x2 + ','
          + y2 + ')');
    }
}


class LegacyLineAdapter {
  public void draw(Coordinate first, Coordinate second) {
    new LegacyLine().draw(first.x, first.y, second.x, second.y);
  }
}

class Coordinate {
  public int x;
  public int y;
}
```

Example

Factory

- Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses

Intent

- Large object without large constructor
- Need to create object in multiple places

Usage

- Create object without exposing the creation logic to the client and refer to newly created object using a common interface.

```
interface Dog {
  public void speak ();
}


class Poodle implements Dog {
  public void speak()
  {
    System.out.println("The poodle says \"arf\"");
  }
}

class Rottweiler implements Dog {
  public void speak()
  {
    System.out.println("The Rottweiler says WOOF");
  }
}

class SiberianHusky implements Dog {
  public void speak()
  {
    System.out.println("The husky says what's up?");
  }
}
```

```
class DogFactory {
  public static Dog getDog(String criteria)
  {
    if ( criteria.equals("small") )
      return new Poodle();
    else if ( criteria.equals("big") )
      return new Rottweiler();
    else if ( criteria.equals("working") )
      return new SiberianHusky();

    return null;
  }
}


public static void main(String[] args) {
  Dog dog = DogFactory.getDog("small");
  dog.speak();

  dog = DogFactory.getDog("big");
  dog.speak();

  dog = DogFactory.getDog("working");
  dog.speak();
}
```

Example

Facade

- A facade is an object that provides a simplified interface to a larger body of code, such as a class library

Intent

- A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Usage

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.

```
/* Facade */

class ComputerFacade {
    private CPU processor;
    private Memory ram;
    private HardDrive hd;

    public ComputerFacade() {
        this.processor = new CPU();
        this.ram = new Memory();
        this.hd = new HardDrive();
    }

    public void start() {
        processor.freeze();
        ram.load(B_ADDR, hd.read(B_SEC, SEC_SIZE));
        processor.jump(B_ADDR);
        processor.execute();
    }
}
```

```
/* Complex parts */

class CPU {
    public void freeze() { ... }
    public void jump(long position) { ... }
    public void execute() { ... }
}

class Memory {
    public void load(long position, byte[] data) { ... }
}

class HardDrive {
    public byte[] read(long lba, int size) { ... }
}

/* Client */

class You {
    public static void main(String[] args) {
        ComputerFacade computer = new ComputerFacade();
        computer.start();
    }
}
```

Example

Decorator

- Add additional responsibilities dynamically to an object

Intent

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Usage

- You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

```
public interface Window {
    public void draw();
    public String getDescription();
}

class SimpleWindow implements Window {
    public void draw() {
        // Draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
```

```
class LoggingWindow implements Window {
    Window window;

    public LoggingWindow(Window window){
      this.window = window;
    }

    public void draw(){
        Logger.warn("Warning");
        window.draw();
    }
}
```

```
Usage:
Window sw = new SimpleWindow();
Sw.draw();
Output:
  Drawings

Window lw = new LoggingWindow(new SimpleWindow());
Lw.draw();

Output:
  Warning
   Drawings
```

Example

Final thoughts

- Pattern blindness
- Don’t try to use patterns for their own sake
- Start simple, not complex
- Use TDD
- Refactor to a pattern (to remove duplication and simplifying your code)
- Don’t force yourself to get it right from a first time

Additional resources:

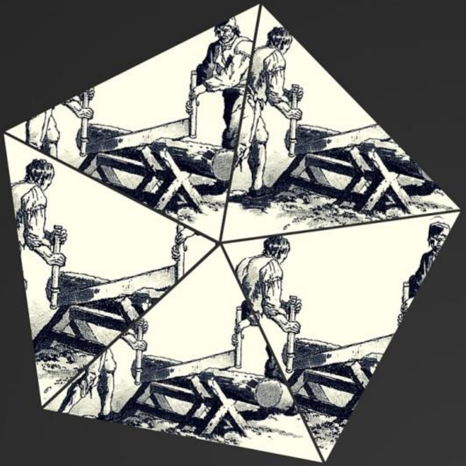
- https://sourcemaking.com/design_patterns
- http://oodesign.com
- https://youtu.be/vNHpsC5ng_E?list=PLF206E906175C7E07
- **Growing Object-Oriented Software, Guided by Tests**

  https://www.amazon.com/Growing-Object-Oriented-Software-Guided-Tests/dp/0321503627
- **Design Patterns: Elements of Reusable Object-Oriented Software**

  https://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented-ebook/dp/B000SEIBB8/ref=mt_kindle?_encoding=UTF8&me=
- **Refactoring to Patterns**

  https://www.amazon.com/Refactoring-Patterns-Addison-Wesley-Signature-Fowler-ebook/dp/B001TKD4RQ/ref=mt_kindle?_encoding=UTF8&me=

SHARPENING THE SAW



A man stops to observe the lumberjack, watching him feverishly sawing at this very large tree. He noticed that the lumberjack was working up a sweat, sawing and sawing, yet going nowhere. The bystander noticed that the saw the lumberjack was using was dull. So, he says to the lumberjack, "Excuse me Mr. Lumberjack, but I couldn't help noticing how hard you are working on that tree, but going nowhere." The lumberjack replies with sweat dripping off of his brow, "Yes... I know. This tree seems to be giving me some trouble." The bystander replies and says, "But Mr. Lumberjack, your saw is so dull that it couldn't possibly cut through anything." "I know", says the lumberjack, "but I am too busy sawing to take time to sharpen my saw."

-- blog.codinghorror.com/sharpening-the-saw

- Lunch and Learn
- Discuss best practices
- Inspire culture of learning

"if all you have is a hammer, everything looks like a nail"

-- Maslow's hammer