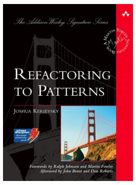
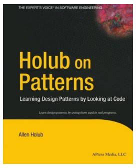
Software Design::The OOP Trifecta

Smells->Refactoring->Patterns

The Trifecta

- Code Smells
  - In computer programming, a code smell is any characteristic in the source code of a program that possibly indicates a deeper problem. Determining what is and is not a code smell is subjective, and varies by language, developer, and development methodology.
- Refactoring
  - Code refactoring is the process of restructuring existing computer code—changing the factoring—without changing its external behavior.
- Design Patterns
  - A design pattern is the re-usable form of a solution to a design problem. The idea was introduced by the architect Christopher Alexander and has been adapted for various other disciplines, notably software engineering.



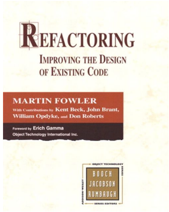


Common Smells

- Hard Coding
  - `Aliens[] a = new Aliens[3];`
- Magic Numbers
  - `double circ = 6.28*r;`
- Programming by Permutation
  - making small changes and testing
- Cargo Cult Programming
  - e.g. setters and getters without good reason *(Pacific Islands, WWII)*
- Premature Optimization
  - Typically 3% needs optimization
- Not Invented Here Syndrome
  - Don't reinvent the wheel
  - *The opposite is also a smell*
- Error Hiding
  - Should we catch and handle exceptions?
- Coding by Exception
  - Adding new handling for every recognized special case
- Tester-Driven-Development
  - Allowing bug reports to drive development of new features (putting out fires!)
- Busy Waiting
  - Continually checking for a condition
- Boat Anchor
  - Obsolete or useless code that continues to encumber the system.
- Action at a Distance
  - Code in one part affects completely different part
    - e.g. caused by globals
- Inappropriate Intimacy
  - Direct access of object internals
    - e.g. Much of the existing SubHunter code

Martin Fowler's Smells

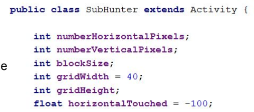
- **Duplicated Code**
- **Long Method**
- **Large Class**
- **Long Parameter List**
- **Divergent Change**
- **Shotgun Surgery**
- **Feature Envy**
- **Data Clumps**
- **Primitive Obsession**
- Switch Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- **Lazy Class**
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- **Inappropriate Intimacy**
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- **Data Class**
- Refused Bequest
- **Comments**



Martin Fowler's Smells

- Duplicated Code
  - Probably the most pungent of code smells. Duplicate code is a sequence of source code that occurs more than once, either within a program or across different programs owned or maintained by the same entity.
  - **Q: How do we fix duplicated code?**
- Long Method
  - "The object programs that live best and longest are those with short methods." Fowler
    - Long methods are more difficult to understand.
    - Long methods are more difficult to reuse.
    - Long methods are often easier to write because we like to think procedurally.
  - **Q: How long is too long?**
  - **Q: What is the right length?**
  - **Q: How do we fix long method?**
    - Comments are a good signal!
- Large Class
  - Close cousin of long method
    - Often signaled by too many instance variables
    - With too many instance variables, duplicated code cannot be far behind.
  - Focus on **SRP**

```
distanceFromSub1 = (int)Math.sqrt(
((horizontalGap1 * horizontalGap1) +
(verticalGap1 * verticalGap1)));

distanceFromSub2 = (int)Math.sqrt(
((horizontalGap2 * horizontalGap2) +
(verticalGap2 * verticalGap2)));

distanceFromSub3 = (int)Math.sqrt(
((horizontalGap3 * horizontalGap3) +
(verticalGap3 * verticalGap3)));
```



```
public class SubHunter extends Activity {

    int numberHorizontalPixels;
    int numberVerticalPixels;
    int blockSize;
    int gridWidth = 40;
    int gridHeight;
    float horizontalTouched = -100;
```

Martin Fowler's Smells

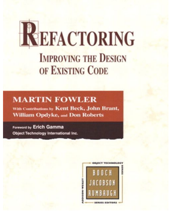
- Long Parameter List
  - Hard to understand: We are forced to *chunk* the parameters in our mind rather than rely on OOP practices to do it for us.
  - The historical opposite of evil global data, pass everything in as a parameter.
    - Objects change this:
      - We can ask other objects for what we need
      - We can encapsulate related parameters into objects
- Divergent Change
  - When one class is commonly changed in different ways for different reasons.
  - We should be localizing where change happens with respect to some concept, again **SRP.**
- Shotgun Surgery
  - When many classes have to be changed in little ways to accomplish a goal.
  - Similar to Divergent Change, but the opposite, we've gone class crazy!
  - **Q: What is the solution here?**
- Feature Envy
  - When a method seems more interested in a class other than the one that it is in.
  - If we're calling a bunch of methods on another object ot calculate a value, maybe the method is in the wrong class.

Martin Fowler's Smells

- Data Clumps
  - Data likes to hang out in groups.
  - When you see little bits of data hanging out with each other, then they probably should be grouped together in a class.
- Primitive Obsession
  - Don't be afraid to write little classes that capture behavior of the use of primitive data types, e.g., Currency, Telephone Numbers, Points.
- Lazy Class
  - Classes cost energy to create and maintain, make sure that each class is pulling its weight.
- Inappropriate Intimacy
  - When classes become to intimate and spend too much time delving in each other's private parts.
- Data Class
  - If a class is just holding data, why?
  - Note: It may be ok, but ask why.
- Comments
  - Comments smell great!
  - However, ask if they are there because they are making bad smells easier to understand

Martin Fowler's Refactorings

- Simplifying Method Calls
- Composition
- Generalization
- Extraction
- Organizing Data


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
OBJECT TECHNOLOGY SERIES
BOOCH JACOBSON RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

Simplifying Method Calls

Preserve Whole Object

You are getting several values from an object and passing these values as parameters in a method call. Send the whole object instead.

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

⇓

```
withinPlan = plan.withinRange(daysTempRange());
```

Replace Parameter With Method

An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.

Remove the parameter and let the receiver invoke the method.

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice =
      discountedPrice (basePrice,discountLevel);
```

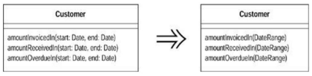
⇓

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Simplifying Method Calls

Introduce Parameter Object

You have a group of parameters that naturally go together. Replace them with an object


Customer
amountInvoicedIn(start: Date, end: Date)
amountReceivedIn(start: Date, end: Date)
amountOverdueIn(start: Date, end: Date)
⇛
Customer
amountInvoicedIn(DateRange)
amountReceivedIn(DateRange)
amountOverdueIn(DateRange)

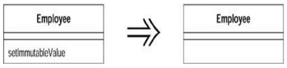
```
void takeShot(float touchX, float touchY)
```

⇓

```
void takeShot(GridSquare gs)
```

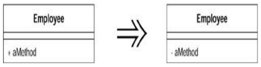
Remove Setting Method

A field should be set at creation time and never altered. Remove any setting method for that field.



Hide Method

A method is not used by any other class. Make the method private.



Composition::Extract Method

You have a code fragment that can be grouped together

Procedure:

1. Turn the fragment into a method whose name explains the purpose of the method.
2. Pass any local variables as parameters into the new method

Useful for:

- Duplicated Code
  - to separate similar bits of code
- Long Method
  - 99% of the time this is all that you need to fix Long Method
- Data Class
  - Useful when you can't move an entire method into a data class to manage its field variables

```
void printOwing(double amount) {
      printBanner();
      //print details
      System.out.println ("name:" + _name);
      System.out.println ("amount" + amount);
}
```

⇓

```
void printOwing(double amount) {
      printBanner();
      printDetails(amount);
}

void printDetails (double amount) {
      System.out.println ("name:" + _name);
      System.out.println ("amount" + amount);
}
```

Composition::Replace Temp with Query

You are using a temporary variable to hold the result of an expression

Procedure:

1. Extract the expression into a method.
2. Replace all references to the temp with the expression. The new method can then be used in other methods

Useful for:

- Long Method
  - eliminate temps

Helpful in composing more complex refactorings wherever temporary variables need to be eliminated

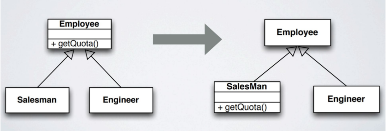
```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
      return basePrice * 0.95;
else
      return basePrice * 0.98;
```

⇓

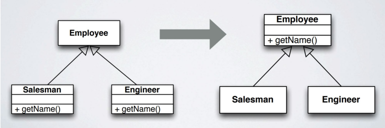
```
if (basePrice() > 1000)
      return basePrice() * 0.95;
else
      return basePrice() * 0.98;
...

double basePrice() {
      return _quantity * _itemPrice;
}
```

Moving Methods Up and Down

Behavior on a superclass is relevant only for some subclasses



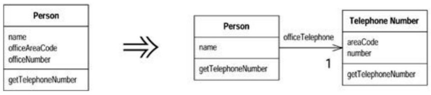
You have methods with identical results in subclasses



Extracting Classes and Interfaces

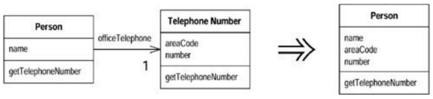
Extract Class

You have one class doing work that should be done by two. Create a new class and move the relevant fields and methods from the old class into the new class



Inline Class

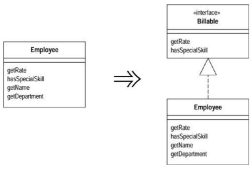
A class isn't doing very much.Move all its features into another class and delete it.



Extracting Classes and Interfaces

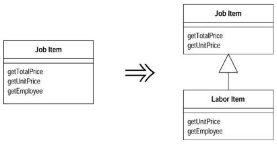
Extract Interface

Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.



Extract Subclass

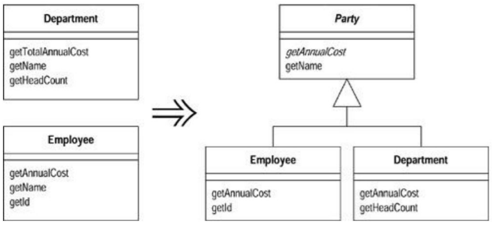
You have methods with identical results in subclasses



Extracting Classes and Interfaces

Extract SuperClass

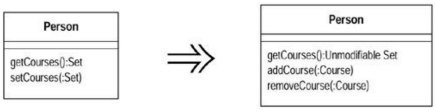
You have two classes with similar features. Create a superclass and move the common features to the superclass.



Organizing Data

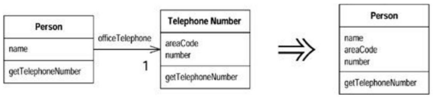
Encapsulate Collection

A method returns a collection. Make it return a read-only view and provide add/remove methods.



Replace Data Value With Object

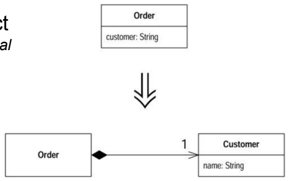
A class isn't doing very much.Move all its features into another class and delete it.



Organizing Data

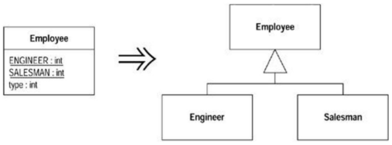
Replace Data Value with Object

You have a data item that needs additional data or behavior. Turn the data item into an object.



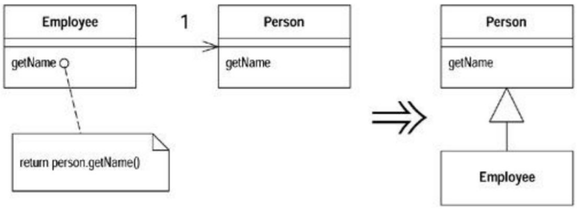
Replace Type Code with Class

You have an immutable type code that affects the behavior of a class. Replace the type code with subclasses



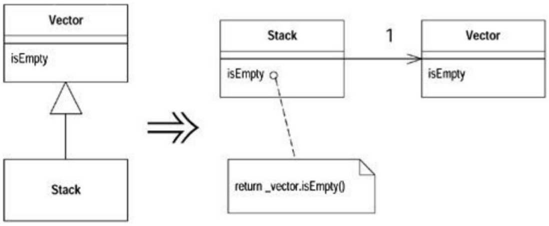
Replace Delegation with Inheritance

You're using delegation and are often writing many simple delegations for the entire interface. Make the delegating class a subclass of the delegate.



Replace Inheritance with Delegation

A subclass uses only part of a superclasses interface or does not want to inherit data. Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.



Design Patterns

- Creational
  - Creational patterns are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- Structural
  - These concern class and object composition. They use inheritance to compose interfaces and define ways to compose objects to obtain new functionality.
- Behavioral
  - Most of these design patterns are specifically concerned with communication between objects.

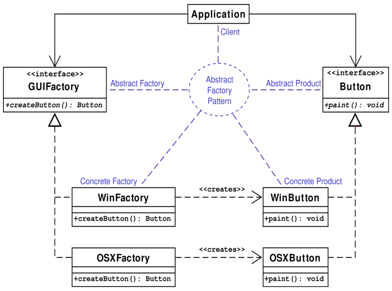
1. **Abstract factory**
2. Builder
3. Factory method
4. Prototype
5. Singleton

1. **Adapter**
2. Bridge
3. **Composite**
4. **Decorator**
5. Facade
6. Flyweight
7. Proxy

1. Chain of responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. **Observer**
8. **State**
9. **Strategy**
10. Template Method
11. Visitor

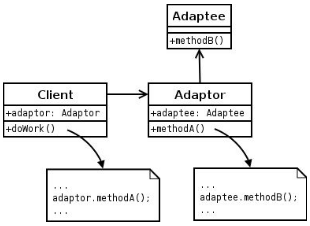
Abstract Factory Pattern

- Intent
  - Create families of objects without specifying concrete classes
- Motivation
  - Want multiple look-and-feel widget sets
  - Support multiple database implementations need to make connections throughout
- Benefits
  - Isolates concrete classes
  - Makes exchanging product families easy
  - Promotes product consistency
- Limitations
  - Supporting new product types is difficult



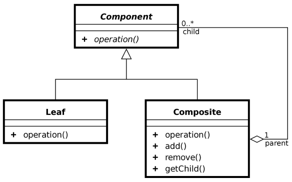
Adapter Pattern

- Intent
  - Convert the interface of one class to another
  - Lets classes work together that aren't compatible
- Motivation
  - Be able to change databases, etc. without rewriting entire application
  - Be able to integrate new library into existing application
- Benefits
  - Available implementation does not need to match required interface
  - Creates a reusable "decoupling" of interface and implementation
- Limitations
  - Runtime overhead
  - Makes overriding Adaptee behavior more difficult



Composite Pattern

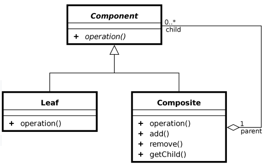
- Intent
  - Compose object into tree structures to represent part-whole hierarchies
  - Allow uniform treatment of individual and composite objects
- Motivation
  - Want to have drawing elements and group of drawing elements treated the same
- Benefits
  - Defines class hierarchies of primitive and composite objects
  - Makes client simple
  - Makes adding new components easier
- Limitations
  - May make design overly general



```
interface Graphic {public void print();} // Component

class CompositeGraphic implements Graphic {
    private final ArrayList<Graphic>childG = new ArrayList<>();
    public void add(Graphic graphic) { childG.add(graphic);}

 public void print() {
        for (Graphic graphic : childG) {graphic.print();}}}

class Ellipse implements Graphic {
    public void print() {System.out.println("Ellipse");}}
```

Composite Pattern

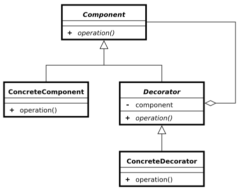
```
public class CompositeDemo {
    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Creates two composites containing the ellipses
        CompositeGraphic graphic2 = new CompositeGraphic();
        graphic2.add(ellipse1);
        graphic2.add(ellipse2);
        graphic2.add(ellipse3);

        CompositeGraphic graphic3 = new CompositeGraphic();
        graphic3.add(ellipse4);

        //Create another graphics that contains two graphics
        CompositeGraphic graphic1 = new CompositeGraphic();
        graphic1.add(graphic2);
        graphic1.add(graphic3);

        //Prints the complete graphic (Four times the string "Ellipse").
        graphic1.print();
    }
}
```

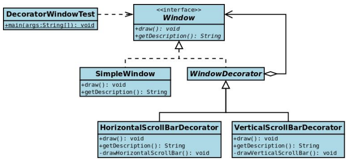


```
interface Graphic {public void print();} // Component

class CompositeGraphic implements Graphic {
    private final ArrayList<Graphic>childG = new ArrayList<>();
    public void add(Graphic graphic) { childG.add(graphic);}

 public void print() {
       for (Graphic graphic : childG) {graphic.print();}}}

class Ellipse implements Graphic {
    public void print() {System.out.println("Ellipse");}}
```

Decorator Pattern

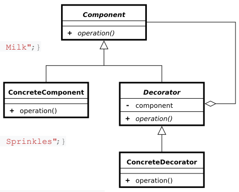
- Intent
  - Attach additional responsibilities to an object dynamically
  - Flexible alternative to subclassing for extension
- Motivation
  - Dynamically add different types of borders to a window (possibly scroll bars)
- Benefits
  - More flexible than inheritance
  - Avoids feature-laden classes high in the hierarchy
  - Decorator and components aren't identical
  - Lots of little objects (easy to maintain)
- Limitations
  - Lo's of little objects (hard to learn)




DecoratorWindowTest
+main(args:String[]): void
<<interface>>
Window
+draw(): void
+getDescription(): String
SimpleWindow
+draw(): void
+getDescription(): String
WindowDecorator
HorizontalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawHorizontalScrollBar(): void
VerticalScrollBarDecorator
+draw(): void
+getDescription(): String
-drawVerticalScrollBar(): void

Decorator Pattern

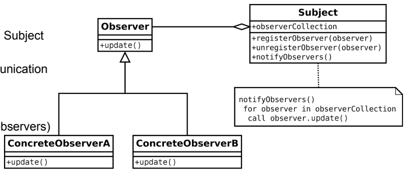
```
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {super(c);}
    public double getCost() {return super.getCost() + 0.5;}
    public String getIngredients() {return super.getIngredients() + ", Milk";}
}


class WithSprinkles extends CoffeeDecorator {
    public WithSprinkles(Coffee c) {super(c);}
    public double getCost() {return super.getCost() + 0.2;}
    public String getIngredients() {return super.getIngredients() + ", Sprinkles";}
}
public class Main {
    public static void printInfo(Coffee c) {
       System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());}

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
  }
```



```
public interface Coffee {
    public double getCost();
    public String getIngredients(); }

public class SimpleCoffee implements Coffee {
    public double getCost() {return 1;}
    public String getIngredients() {return "Coffee";}}

public abstract class CoffeeDecorator implements Coffee
{
     private final Coffee decoratedCoffee;
     public CoffeeDecorator(Coffee c) {
            this.decoratedCoffee = c;}
     public double getCost()       {
            return decoratedCoffee.getCost();}
     public String getIngredients()    {
            return decoratedCoffee.getIngredients();}}
```

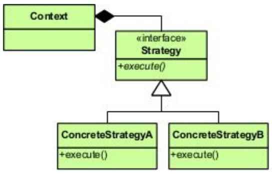
Observer Pattern

- Intent
  - Define one-to-many dependency between objects
  - When object changes state, all dependents are notified (or updated automatically)
- Motivation
  - Want consistency of partitioned system without needing to tightly couple classes
  - Multiple windows representing data in different ways, update of data should reflect in all windows
- Benefits
  - Abstract coupling between Subject and Observer
  - Supports broadcast communication
- Limitations
  - Unexpected updates (cascading of updates to observers)



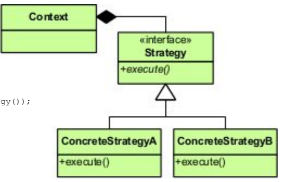
Strategy Pattern

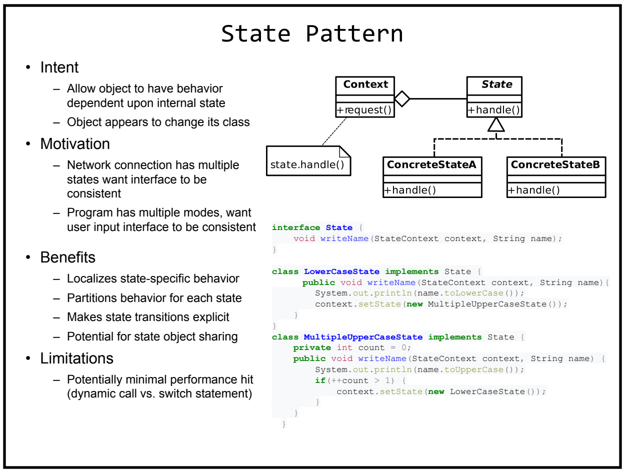
- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
  - Capture the abstraction in an interface, bury implementation details in derived classes.
- Motivation
  - One of the dominant strategies of object-oriented design is the "open-closed principle".
  - Different algorithms will be appropriate at different times
- Benefits
  - Eliminate conditional statements
  - Can provide choices of implementations for the same behavior
- Limitations
  - Clients must be aware of different Strategies
  - Communication overhead
  - Increased number of objects



Strategy Pattern

```
public interface CompressionStrategy { //Strategy Interface
 public void compressFiles(ArrayList<File> files);
}

public class ZipCompressionStrategy implements CompressionStrategy {
 public void compressFiles(ArrayList<File> files) {/using ZIP approach}
}

public class RarCompressionStrategy implements CompressionStrategy {
 public void compressFiles(ArrayList<File> files) {/using RAR approach}
}

public class CompressionContext {
 private CompressionStrategy strategy;

 public void setCompressionStrategy(CompressionStrategy strategy) {
   this.strategy = strategy;
 }

 public void createArchive(ArrayList<File> files) {
   strategy.compressFiles(files);
 }
}

public class Client {
  public static void main(String[] args) {
    CompressionContext ctx =new CompressionContext();
        ctx.setCompressionStrategy(new ZipCompressionStrategy());
        ctx.createArchive(fileList);
     }
}
```

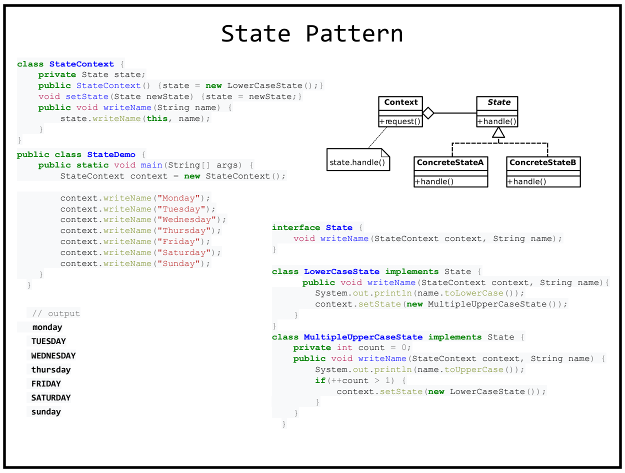




State Pattern

- Intent
  - Allow object to have behavior dependent upon internal state
  - Object appears to change its class
- Motivation
  - Network connection has multiple states want interface to be consistent
  - Program has multiple modes, want user input interface to be consistent
- Benefits
  - Localizes state-specific behavior
  - Partitions behavior for each state
  - Makes transitions explicit
  - Potential for state object sharing
- Limitations
  - Potentially minimal performance hit (dynamic call vs. switch statement)

```
interface State {
    void writeName(StateContext context, String name);
}

class LowerCaseState implements State {
      public void writeName(StateContext context, String name){
        System.out.println(name.toLowerCase());
        context.setState(new MultipleUpperCaseState());
    }
}
class MultipleUpperCaseState implements State {
    private int count = 0;
    public void writeName(StateContext context, String name) {
        System.out.println(name.toUpperCase());
        if(++count > 1) {
            context.setState(new LowerCaseState());
        }
    }
  }
```



State Pattern

```
class StateContext {
    private State state;
    public StateContext() {state = new LowerCaseState();}
    void setState(State newState) {state = newState;}
    public void writeName(String name) {
        state.writeName(this, name);
    }
}
public class StateDemo {
    public static void main(String[] args) {
        StateContext context = new StateContext();

        context.writeName("Monday");
        context.writeName("Tuesday");
        context.writeName("Wednesday");
        context.writeName("Thursday");
        context.writeName("Friday");
        context.writeName("Saturday");
        context.writeName("Sunday");
    }
  }
```

```
  // output
  monday
  TUESDAY
  WEDNESDAY
  thursday
  FRIDAY
  SATURDAY
  sunday
```

```
interface State {
    void writeName(StateContext context, String name);
}

class LowerCaseState implements State {
      public void writeName(StateContext context, String name){
        System.out.println(name.toLowerCase());
        context.setState(new MultipleUpperCaseState());
    }
}
class MultipleUpperCaseState implements State {
    private int count = 0;
    public void writeName(StateContext context, String name) {
        System.out.println(name.toUpperCase());
        if(++count > 1) {
            context.setState(new LowerCaseState());
        }
    }
  }
```

```
System.exit();
```