Design Patterns

Much thanks to Professor Ian J. Davis (ret.)

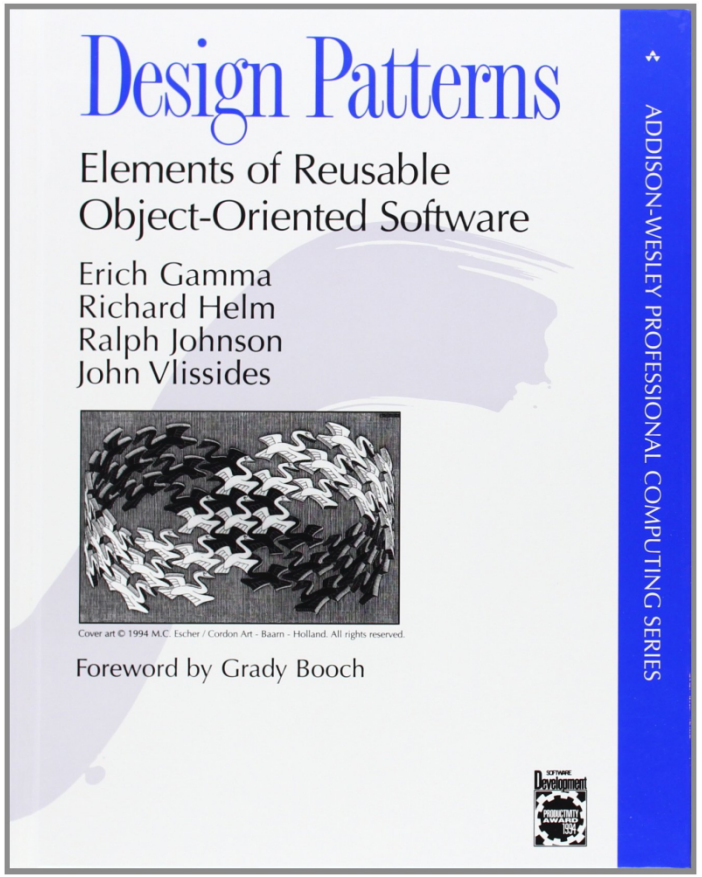
Cheriton School of Computer Science

University of Waterloo

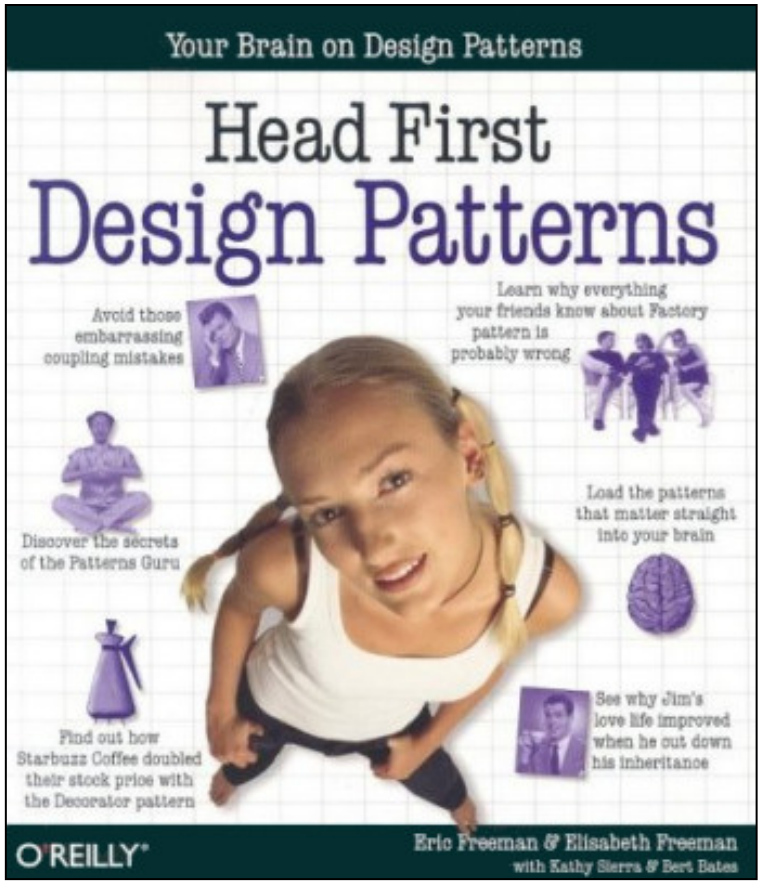
for creating many of these slides

(Any errors belong to Prof. Szajda)

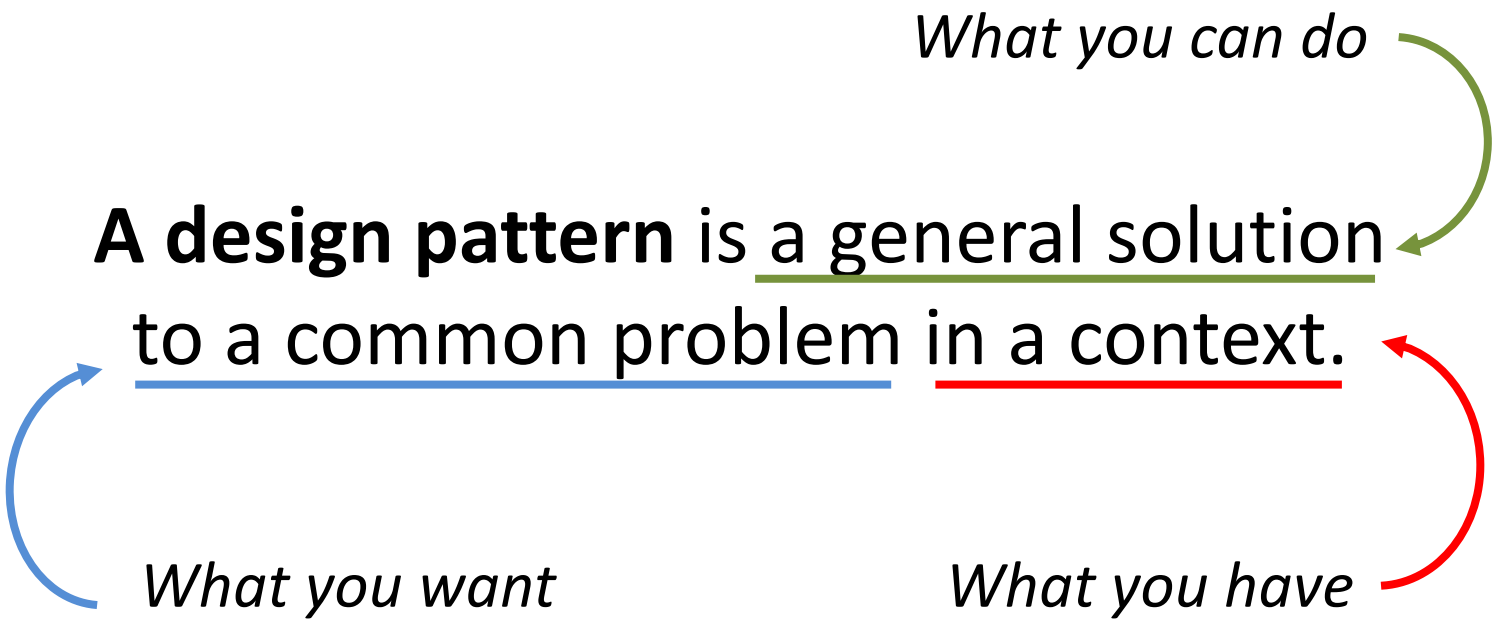
Useful books



The “Gang of Four” book
1994



Head First Design Patterns Book
2004


What you can do
A design pattern is a general solution to a common problem in a context.
What you want
What you have

Design Pattern Index

Creational	Structural	Behavioural
Factory Method	Adapter	Template
Abstract Factory	Bridge	Strategy
Builder	Composite	Command
Singleton	Decorator	State
Multiton	Facade	Visitor
Object pool	Flyweight	Chain of Responsibility
Prototype	Front controller	Interpreter
	Proxy	Observer
		Iterator
		Mediator
		Memento

Motivation

- A cook knows a lot of recipes
  - They look up the ones they forget
- Design patterns are recipes for programmers
- How can you choose a good strategy
  - If you don't know the range of available strategies
- How can you talk about the choices
  - If they don't have well understood names
- Angel cake
  - Instruction: Fold don't stir..
  - I used a food blender because it was there
    - My angel cake was afterwards remembered as a pancake

WARNING:

Overuse of design patterns can lead to code that is downright over-engineered.

Always go with the simplest solution that does the job and introduce patterns only where the need emerges.

[Head First Design Patterns]

Introduction

- Designing object-oriented software is hard, designing reusable object-oriented software is even harder
- Design should be specific to problem, but also general enough to address future problems and requirements
- Expert designers reuse solutions that have worked for them in the past
  - Recurring patterns of classes and communicating objects exist in many object-oriented systems

Thanks to Sascha Konrad for this and the next several slides

Introduction

- If details of previous problems and their solutions are known, then they could be reused
  - Recording experience in software design for others to use
- Design patterns = important and recurring design in object-oriented systems

What is a Design Pattern?

"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice"

Christopher Alexander, A Pattern Language, 1977

What is a Design Pattern

- A Pattern has 4 Essential Elements
  - Pattern name
  - Problem
  - Solution
  - Conquences

Pattern Name

- A handle used to describe a design problem, its solutions and its consequences in a word or two
- Increases design vocabulary
- Makes it possible to design at a higher level of abstraction
- Enhances communication
- But finding a good name is often hard

Problem

- Describes when to apply the pattern
- Explains the problem and its context
- Might describe specific design problems or class or object structures
- Sometimes contains a list of conditions that must be met before it makes sense to apply the pattern

Solution

- Describes the elements that make up the design, their relationships, responsibilities and collaborations
- Doesn’t describe a particular concrete design or implementation
- Abstract description of design problems and how the pattern solves it

Consequences

- Results and trade-offs of applying the pattern
- Critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern
- Includes the impacts of a pattern on a system's flexibility, extensibility and/or portability

Design Patterns Are Not

- Designs that can be encoded in classes and reused as is (i.e. linked lists, hash tables)
- Complex domain-specific designs (for an entire application or subsystem)
- They Are: “Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”

Good Uses For Design Patterns

- Finding appropriate objects
  - Hard part of object-oriented design is decomposing a system into objects: Encapsulation, granularity, dependency, flexibility, performance, …
  - Design Patterns help identifying less obvious abstractions and the objects that can capture them

Good Uses For Design Patterns

- Determining object granularity
  - Objects can vary tremendously in size and number
  - Design patterns address this also i.e., describing how to decompose an object into smaller objects

Good Uses For Design Patterns

- Specifying object interfaces
  - An object's interface characterizes the complete set of requests that can be sent to the object
  - A type can be thought of as a particular interface
  - Subtypes inherit the interfaces of its super types
  - Design patterns help defining the interfaces by identifying the key elements and the kind of data that gets sent across an interface
  - A design pattern might also tell what not to put in an interface

Good design principles

- Program to interfaces not to an implementation
- Separate what changes from what does not
- Encapsulate what varies behind an interface
- Favor composition over inheritance
  - Inheritance: derives one class from another (tightly coupled)
  - Composition: defines a class as the sum of its parts (loosely)
- Loosely couple objects that interact
- Classes should be open for extension, but closed for modification (the “open-closed principle”)
- Each class should have one responsibility
- Depend on abstractions, not concrete classes

Open-Closed Principle

- An entity should allow its behavior to be extended without modifying its source code

Good use of Design Pattern Principles

- Let design patterns emerge from your design, don’t use them just because you should
- Always choose the simplest solution that meets your need
- Always use the pattern if it simplifies the solution
- Know all the design patterns out there
- Talk a pattern when that simplifies conversation

Three Types of Design Patterns

- Creational : Used to create objects for a suitable class that serves as a solution for a problem.
  - Particularly useful when you are taking advantage of polymorphism and need to choose between different classes at runtime rather than compile time
- Behavioral : describe interactions between objects and focus on how objects communicate with each other.
  - They can reduce complex flow charts to mere interconnections between objects of various classes.
- Structural: concerned with how classes and objects are composed to form larger structures

Creational Design Patterns

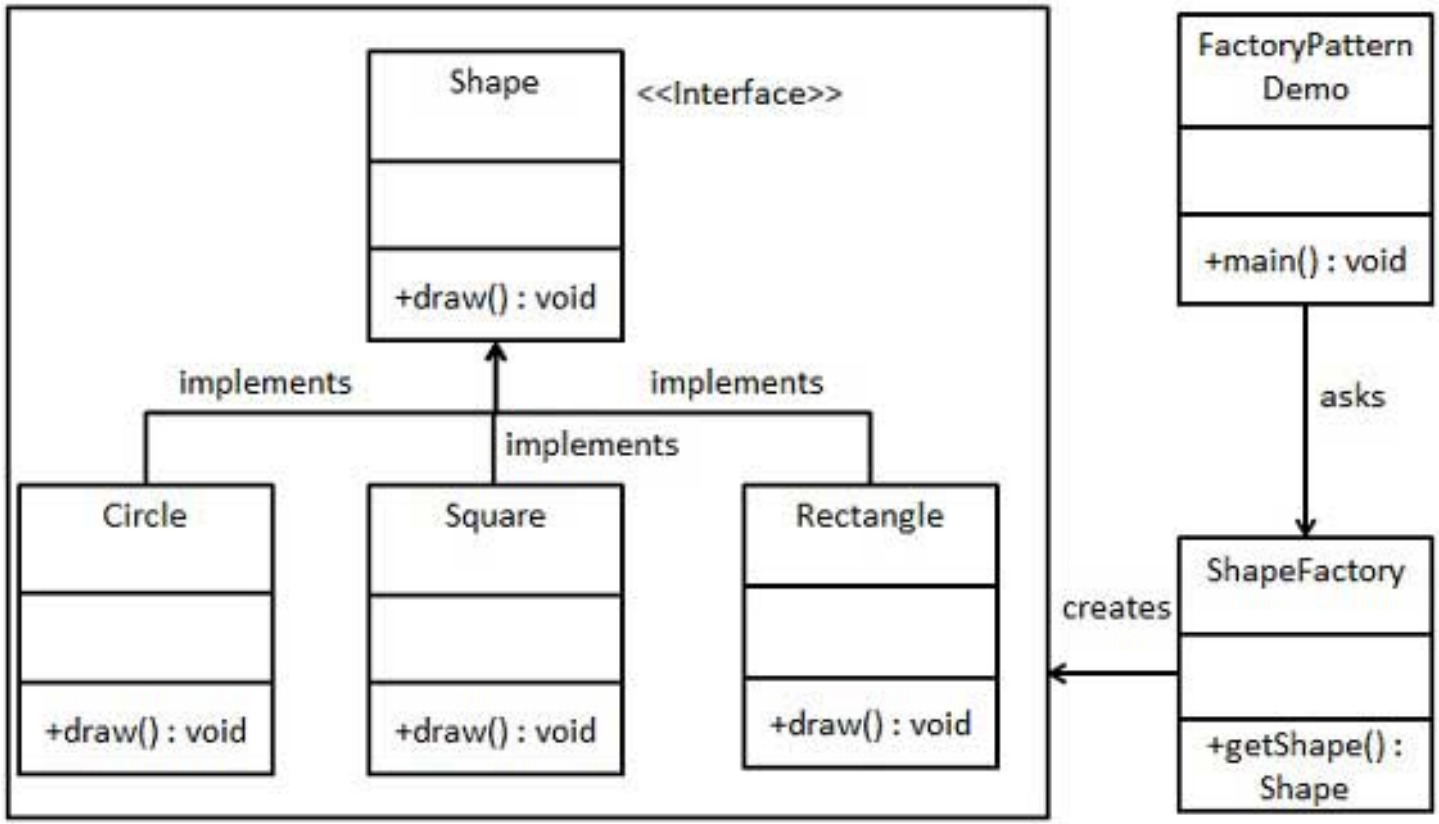
Factory Pattern

- Create an object without exposing creation logic to the client
- Refer to newly created object using a common interface
- Thanks to tutorialspoint: https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

Factory Pattern Example Implementation

- Create a `Shape` interface and concrete classes implementing the `Shape` interface
- A factory class `ShapeFactory` is defined as a next step
- `FactoryPatternDemo` will use `ShapeFactory` to get a `Shape` object
  - Passes information to `ShapeFactory` to get the type of object it requires

Factory Pattern Example Implementation



Factory Pattern Example Implementation

- Steps:
  - Create interface
  - Create concrete classes implementing interface
  - Create a factory to generate concrete classes based on provided information
  - Use factory to get object of concrete class by passing information, such as type
  - Verify output

Factory Pattern Example Implementation

- Create interface

Shape.java

```
public interface Shape {
   void draw();
}
```

Factory Pattern Example Implementation

- Create concrete classes implementing same interface (next slide)

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Factory Pattern Example Implementation

- Create a factory to generate concrete classes based on provided information (next slide)

ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```

Factory Pattern Example Implementation

- Use factory to get object of concrete class by passing information, such as type (next slide)

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of square
      shape3.draw();
   }
}
```

Factory Pattern Example Implementation

- Verify output

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

Factory Design Pattern Goals

- Separate what changes from what does not
- Encapsulate what varies behind an interface
- Depend on abstractions, not concrete classes
- Loosely couple objects that interact
- Design should be open for extension but closed to modification

Problems with using *new*

- So *myClass = new MyClass(….)* is bad..
  - It might be invoked from many places
  - It might be replaced by new MyBetterClass(…)
  - Many different subclasses might exist
    - Want to be able to decide the subclass at runtime
    - Want to create the correct subclass not old superclass
    - Without ugly code present everywhere
  - Want perhaps to pass around a tool to create whatever the tool is designed to create
  - May not know what to create until run time

Factory Solution

- Wrap code that creates objects inside a method
  - *Interface = createMyClass(parameters)*
  - We can change the class created now in one place
  - Parameter can identify what to create
- Might make createMyClass static
  - Ie. static MyClass::createMyClass(…)
  - But this reassociates the creation with the class
- If we want to pass method around, put it in a factory object (cleaner than pointer to method)

Factory Pattern

- High Level: uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

Thanks to Wikipedia: https://en.wikipedia.org/wiki/Factory_method_pattern

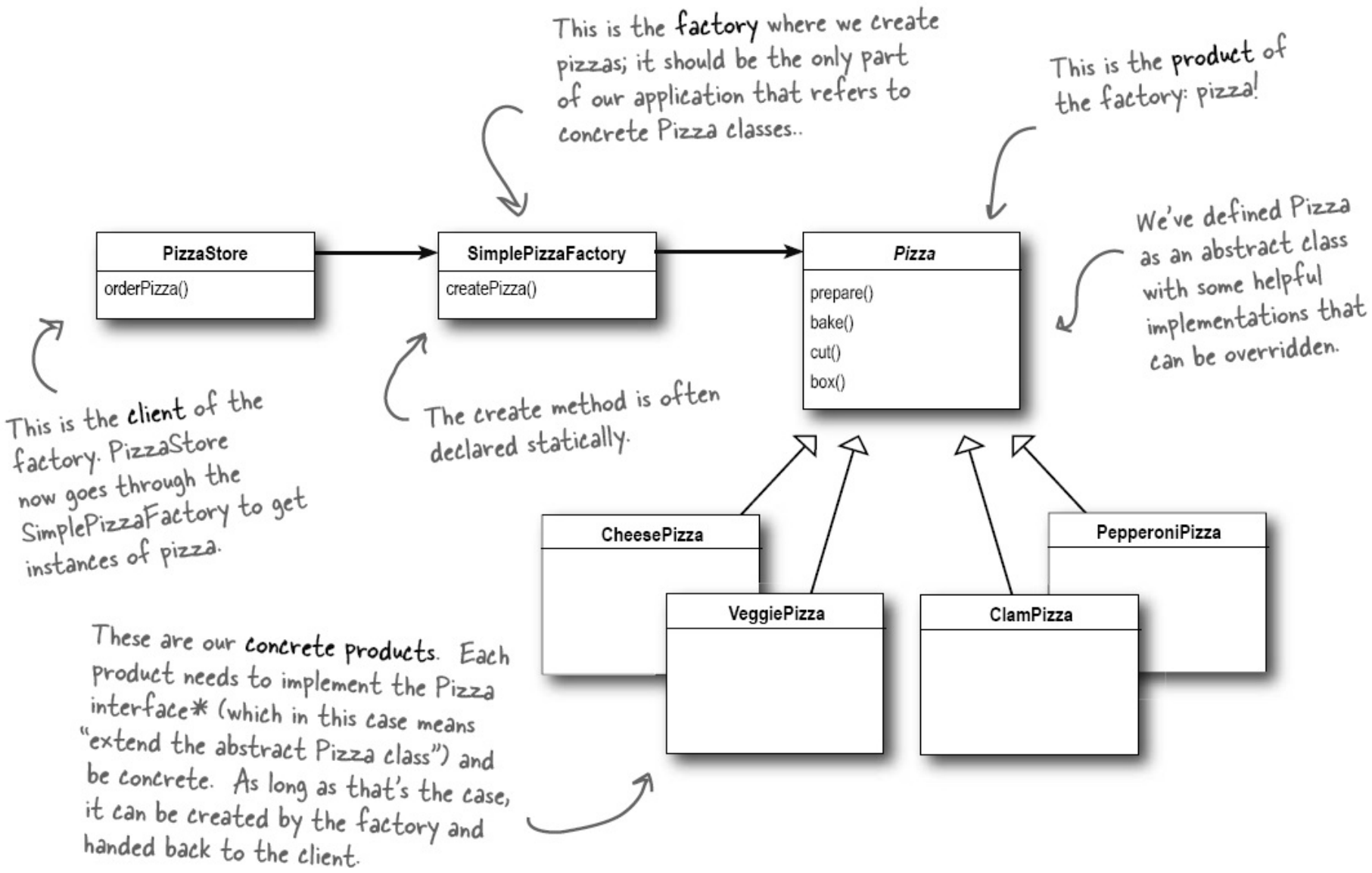
Now we give PizzaStore a reference to a SimplePizzaFactory.

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = factory.createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

    // other methods here
}
```

PizzaStore gets the factory passed to it in the constructor.

And the orderPizza() method uses the factory to create its pizzas by simply passing on the type of the order.

Notice that we've replaced the **new operator** with a create **method** on the factory object. No more concrete instantiations here!


This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes..
This is the product of the factory: pizza!
We've defined Pizza as an abstract class with some helpful implementations that can be overridden.
PizzaStore
orderPizza()
SimplePizzaFactory
createPizza()
Pizza
prepare()
bake()
cut()
box()
This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.
The create method is often declared statically.
CheesePizza
VeggiePizza
ClamPizza
PepperoniPizza
These are our concrete products. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

Advantages

- Protects higher level code from ugly lower level detail such as which class should be created
- Any change to the class created is made just once
- We can make the objects to create dynamic
  - Simply pass a pointer to a factory object to the things that need to create the objects
  - Then the factory object becomes conceptually the instruction regarding what to create
- We can override createMyClass in subclasses or use it in any other way we wish
  - (i.e. Use with Template design pattern (look it up))

Factory Pattern Definition

- The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate.

- Question: How should we delete objects created using a factory.
- Should we have a matching deleteMyClass()
- Should it make the pointer to the object null
- Or should we use reference counting??

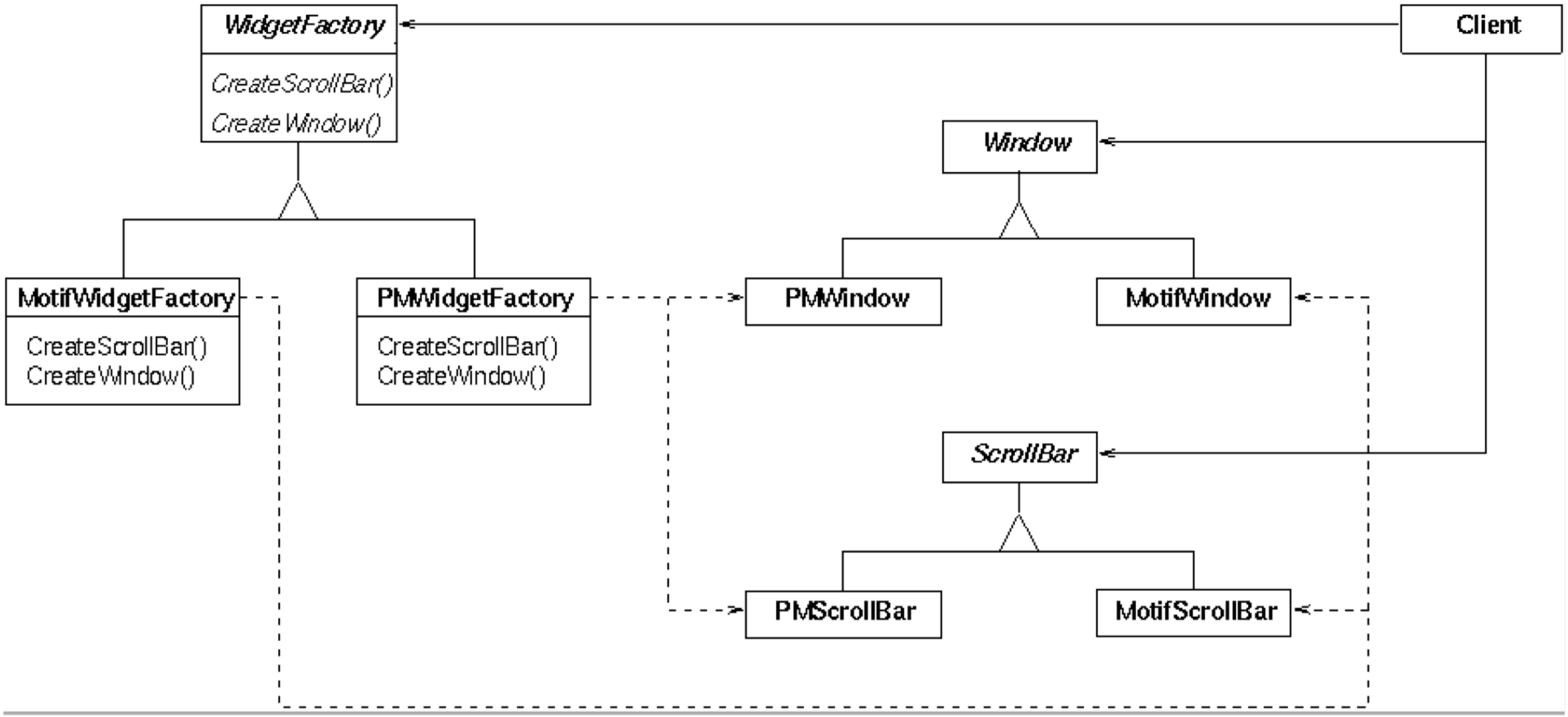
Factory Pattern Consequences

- There may be many more classes than necessary

- Factory pattern hides implementation details of concrete classes, and in fact at times details of the classes themselves, from the client. But these details may be something that the client needs to know

Abstract Factory Pattern

- Suppose we have a factory that can be told what objects to create from a choice of all within a framework
- Then we can create a new factory with the same “abstract” interfaces to create the same objects but from a very different framework
- Deciding which framework to use then becomes deciding which concrete instantiation of an abstract factory class to use


WidgetFactory
CreateScrollBar()
CreateWindow()
Client
Window
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
PMWindow
MotifWindow
ScrollBar
PMScrollBar
MotifScrollBar

Advantages

- Protects us from creating components from many frameworks that can’t talk to each other
- Makes it easy to decide at run time which framework we want to use
- The framework chosen is transparent to all higher level code
- Makes it easier to add new frameworks if we have need to do so

Example

- Domino's pizza's are flat
- Pizza Hut's pizza's are deep dish pizza's
- Both offer a variety of subclasses of pizzas
- If we create an abstract factory
  - Subclass a Domino's pizza factory
  - Subclass a Pizza Hut's pizza factory
- We won't need to create a pizza store for each
  - Just give each pizza store a different factory
- Adding support for New York Pizza
  - Piece of cake (pizza)

PizzaStore is now abstract (see why below).

This is employing the template design pattern (described later)

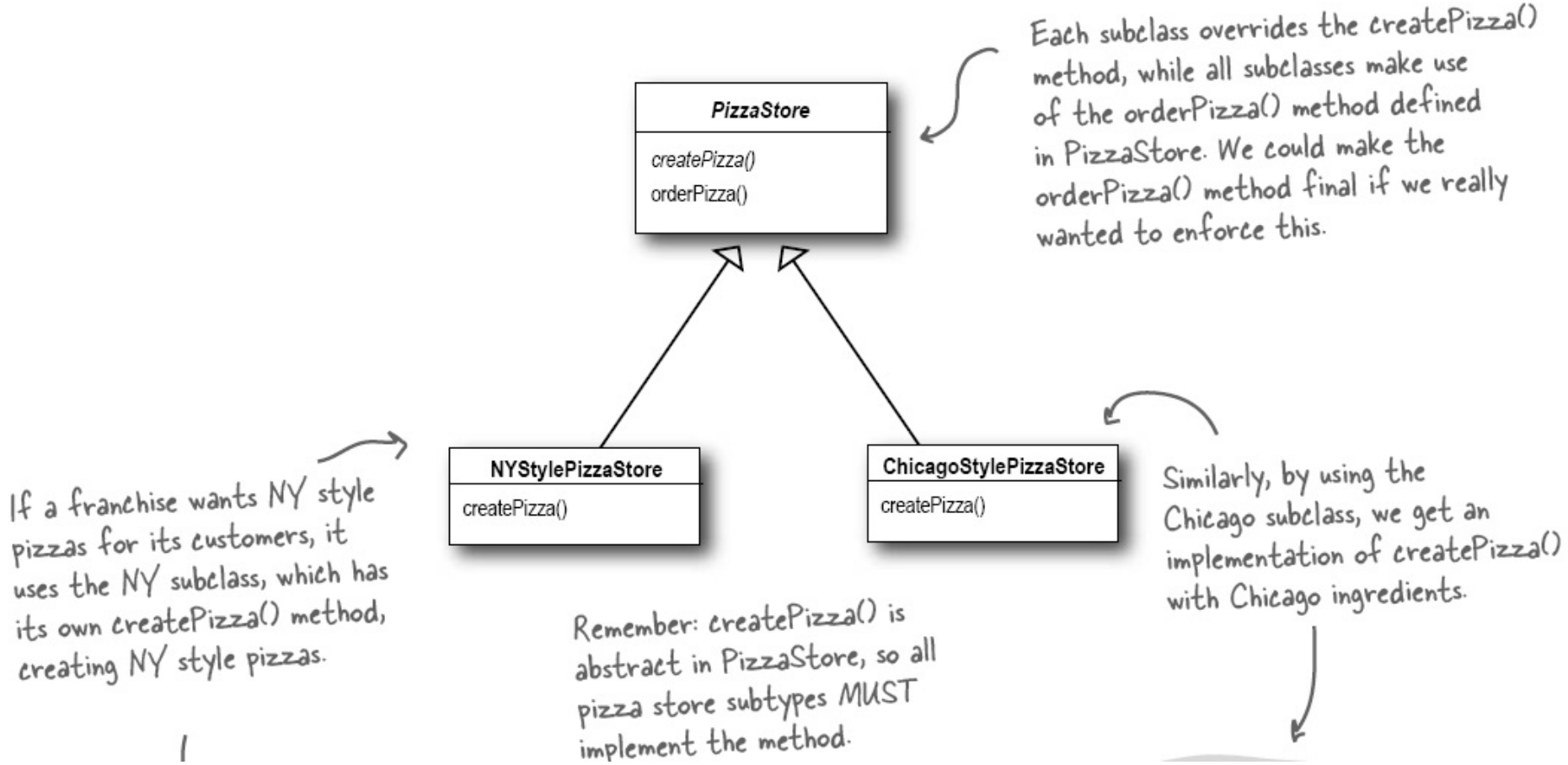
```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

    abstract Pizza createPizza(String type);
}
```

Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

Now we've moved our factory object to this method.

Our "factory method" is now abstract in PizzaStore.



- This is the template approach
- Can also pass factory to pizzastore
  - Save it internally
  - This is the strategy design pattern

Abstract Factory Definition

- The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes

Strategy Design Pattern

- A.k.a. Policy pattern
- A behavioral design pattern
- Enables objects to select their desired algorithm(s) at runtime
  - Instead of coding algorithm directly, code instead receives run-time info which determines which algorithm from a particular family should be used
  - So algorithm varies independently from clients that use it
  - Deferring to run time allows for more flexible and reusable code

Thanks Wikipedia: https://en.wikipedia.org/wiki/Strategy_pattern

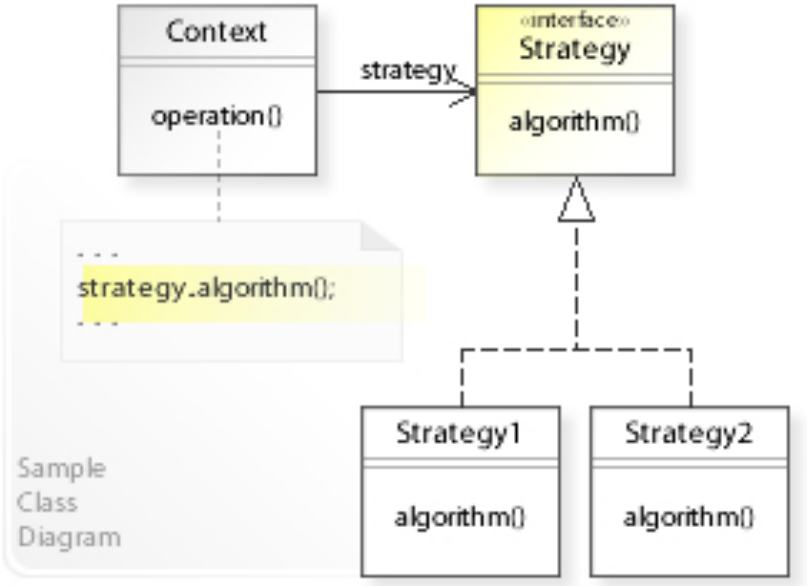
Strategy Design Pattern

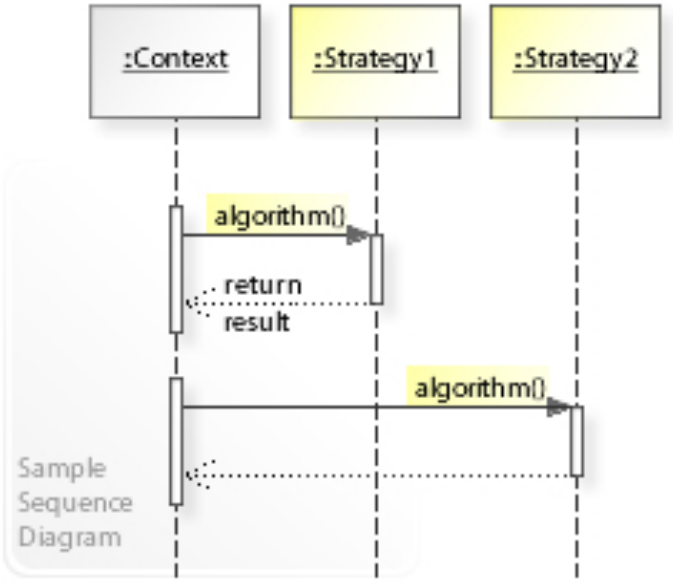
- A.k.a. Policy pattern
- A behavioral design pattern
- Enables objects to select their desired algorithm(s) at runtime
  - Instead of coding algorithm directly, code instead receives run-time info which determines which algorithm from a particular family should be used
  - So algorithm varies independently from clients that use it
  - Deferring to run time allows for more flexible and reusable code

Strategy Design Pattern

- Example: Class needs to perform validation on incoming data
  - Particular validation method might depend, for example, on data type, data source, user choice, etc.
    - Particulars not known until runtime, and may require very different validation specifics
- The validation algorithms (strategies) encapsulated separately from validating object can be used by other objects in different parts of system (or different system) without code duplication

Strategy Design Pattern





State Design Pattern

- Behavioral design pattern that allows an object to change its behavior when internal state changes
  - So effectively a variant of strategy pattern
- Can be a clean way for an object to change its behavior at run time without resorting to conditional statements
  - Improves maintainability

Thanks Wikipedia: https://en.wikipedia.org/wiki/State_pattern#cite_note-GOF-1

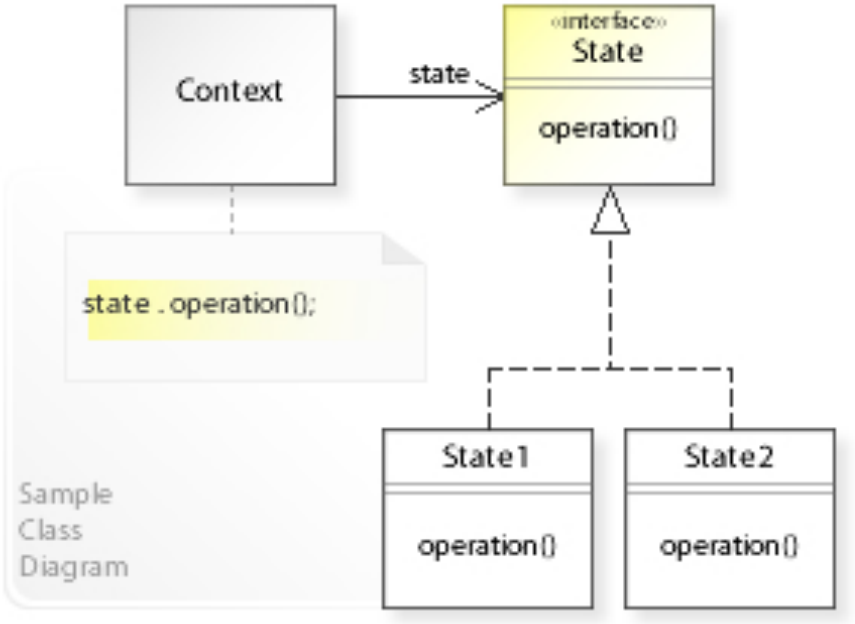
State Design Pattern

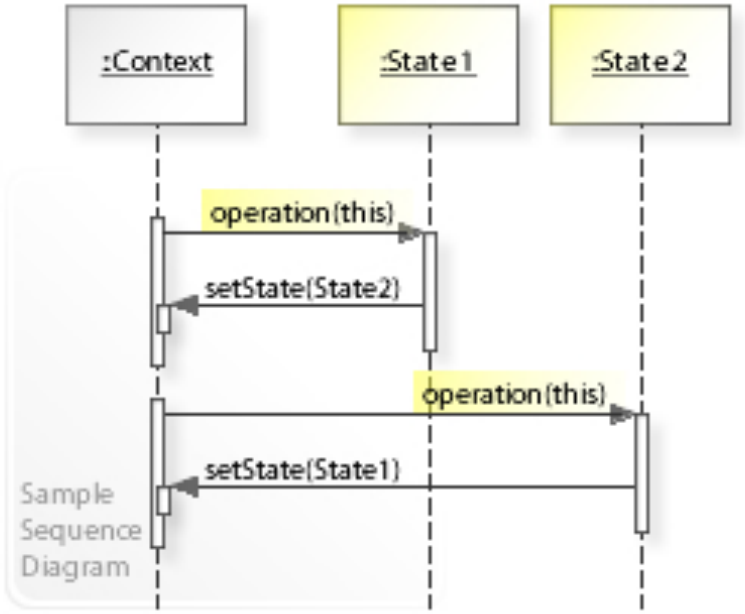
- Solves two primary problems
  - Object should change behavior when internal state changes
  - State-specific behavior should be defined independently
    - I.e., adding new states should not affect behavior of existing states
- Implementing state-specific behavior directly within a class is inflexible
  - It commits the class to a particular behavior
  - It makes it impossible to add a new state or change the behavior of an existing state later without modifying the class

State Design Pattern

- Two solutions
  - For each state, design separate object that enacapsulates specific behavior for that state
  - A class delegates state-specific behavior to its current state object
    - Rather than implementing it directly
- Class becomes independent of implementation of state-specific behavior
- New states added by defining new state classes
- Class changes state at runtime by changing current state object

State Design Pattern





Look familiar?

State Design Pattern Example

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

class LowerCaseState implements State {
    @Override
    public void writeName(StateContext context, String name) {
        System.out.println(name.toLowerCase());
        context.setState(new MultipleUpperCaseState());
    }
}

class MultipleUpperCaseState implements State {
    /* Counter local to this state */
    private int count = 0;

    @Override
    public void writeName(StateContext context, String name) {
        System.out.println(name.toUpperCase());
        /* Change state after StateMultipleUpperCase's writeName() gets invoked twice */
        if (++count > 1) {
            context.setState(new LowerCaseState());
        }
    }
}
```

State Design Pattern Example

```
class StateContext {
    private State state;

    public StateContext() {
        state = new LowerCaseState();
    }

    /**
     * Set the current state.
     * Normally only called by classes implementing the State interface.
     * @param newState the new state of this context
     */
    void setState(State newState) {
        state = newState;
    }

    public void writeName(String name) {
        state.writeName(this, name);
    }
}
```

State Design Pattern Example

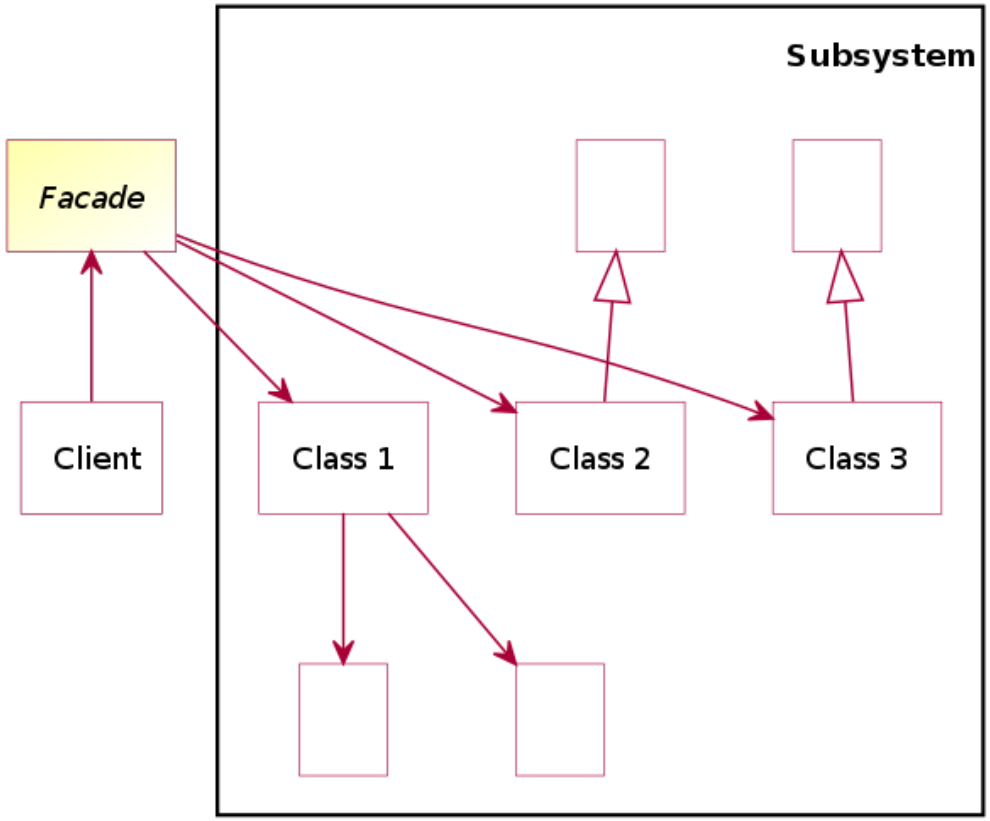
```
public class StateDemo {
    public static void main(String[] args) {
        StateContext context = new StateContext();

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

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

Façade Pattern

- A Structural design pattern
- As in architecture, a façade is a client facing interface that hides complexity of the underlying structure and code
  - Façade delegates to the underlying interfaces/system(s)
- Potential uses:
  - Improve readability and usability of software library by providing a single simplified interface
  - Provide context-specific interface to generic functionality
  - Allow for refactoring monolithic (typically tightly coupled) code into more loosely coupled code

Façade Pattern



Sample class diagram