ADVANCED PROGRAMING PRACTICES

Design patterns

Design patterns

- *"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 the same way twice."* [Christopher Alexander – late 1970s]
  - A Pattern Language (1977)
  - A Timeless Way of Building (1979)

- Design patterns capture the best practices of experienced object-oriented software developers.

- Design patterns are solutions to general software development problems.

Pattern elements

- In general, a pattern has four essential elements.
  - The pattern name
  - The problem
  - The solution
  - The consequences

Pattern elements: name

- The pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
  - Naming a pattern immediately increases the **design vocabulary**. It lets us design at a higher level of abstraction.
  - Having a vocabulary for patterns lets us talk about them.
  - It makes it easier to think about designs and to **communicate** them and their trade-offs to others.

Pattern elements: problem

- The problem describes when to apply the pattern.
  - It explains the **problem** and its **context**.
  - It might describe specific design problems such as how to represent algorithms as objects.
  - It might describe class or object structures that are symptomatic of an inflexible design.
  - Sometimes the problem will include a list of **conditions** that must be met before it makes sense to apply the pattern.

Pattern elements: solution

- The solution describes the **elements** that make up the design, their relationships, responsibilities, and collaborations.
  - The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations.
  - Instead, the pattern provides an **abstract description** of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

Pattern elements: consequences

- The consequences are the results and **trade-offs** of applying the pattern.
  - The consequences for software often concern space and time trade-offs.
  - They may address language and implementation issues as well.
  - Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability.

Design patterns: types

- Erich Gamma, Richard Helm, Ralph Johnson and John Vlisides in their Design Patterns [*gang of four*] book define 23 design patterns divided into three types:
  - **Creational patterns** - 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 patterns** - help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
  - **Behavioral patterns** - help you define the communication between objects in your system and how the flow is controlled in a complex program.

Creational patterns

Creational patterns: concept

- The creational patterns deal with the best way to create instances of objects.
- In Java, the simplest ways to create an instance of an object is by using the `new` operator or by simply declaring the variable in the local scope:

```
Fred myFred = new Fred(); //instance of Fred class
```

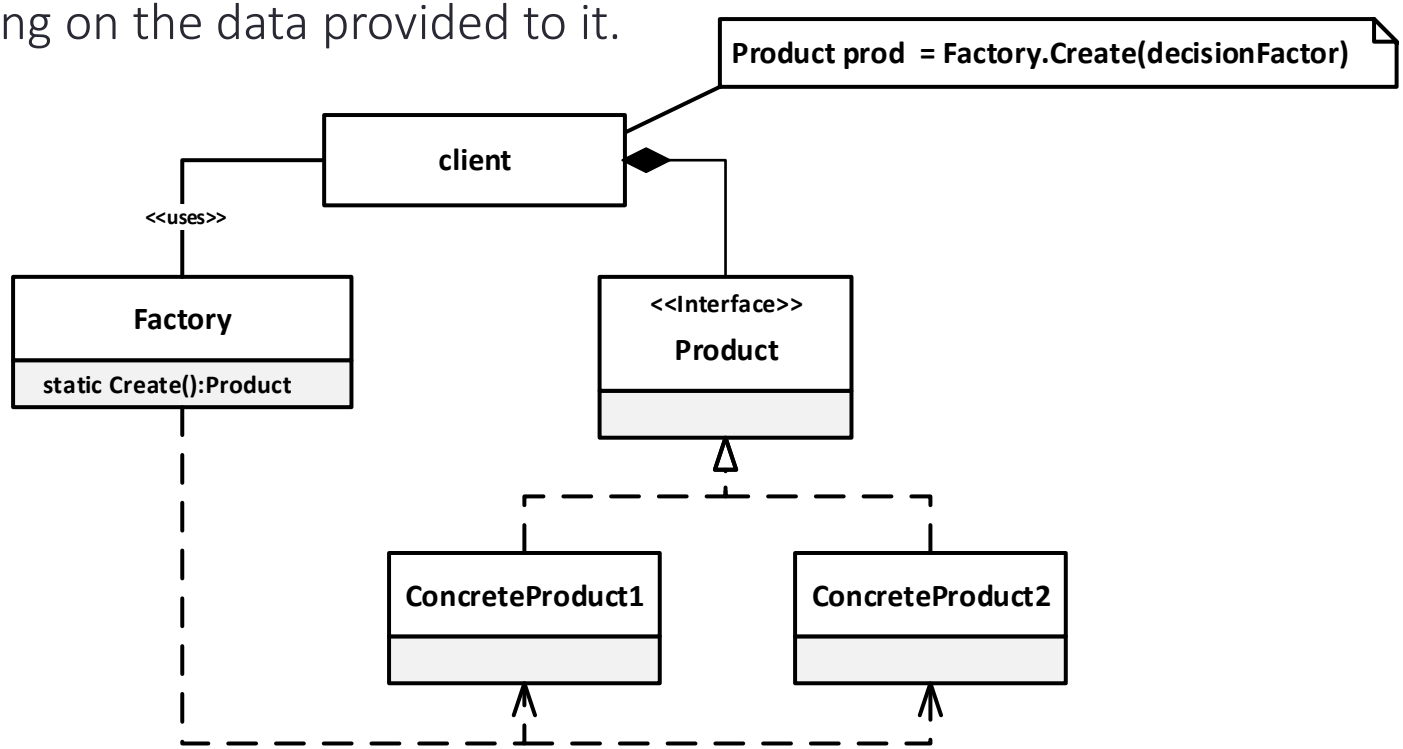
- The amount of hard coding depends on how you create the object within your program.
- In many cases, the exact nature of the object that is created could vary with the needs of the program and abstracting the creation process into a special “creator” class can make your program more flexible and general.

Creational patterns: examples

- The **Factory Pattern** provides a simple decision making class that returns one of several possible subclasses of an abstract base class depending on the data that are provided.
- The **Abstract Factory Pattern** provides an interface to create and return one of several families of related objects.
- The **Builder Pattern** separates the construction of a complex object from its representation.
- The **Prototype Pattern** starts with an initialized and instantiated class and copies or clones it to make new instances rather than creating new instances.
- The **Singleton Pattern** is a class of which there can be no more than one instance. It provides a single global point of access to that instance.

Factory pattern: structure

- The Factory pattern returns an instance of one of several possible classes depending on the data provided to it.



- **`Product`** - is the declared type of the variable that needs to be created.
- **`ConcreteProduct`** - is the type of object that will be created.
- **`Factory`** - is the class that decides which of these subclasses to return depending on the arguments you give it.
- **`Create()`** - method passes in some value (the decision factor), and returns some instance of one of the subclasses of `Product`.

Factory pattern: example

```
/** Shape Factory that creates different kinds of Shape objects
 * by calling its static factory method.
 */
  public class ShapeFactory {
    /**
     * Constructor is made private to prevent instantiation
     */
  private ShapeFactory (){};

    /** Static method to create objects
     * Change is required only in this function
     * @param shapeType Variation factor of the Factory whose value determines
     * what kind of shape will be created
     * @return a Shape object whose type depends on the value of the variation factor
     */
   static public Shape getShape(String shapeType){
      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

```
/**
 * Shape Interface (or Abstract class) that is the super-type of all
 * types of objects produced by the ShapeFactory.
 */
public interface Shape {
   void draw();
}
```

```
public class Rectangle implements Shape {

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

```
public class Square extends Rectangle implements Shape {

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

```
public class Circle implements Shape {

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

Factory pattern: example

```
public class FactoryDriver {

   public static void main(String[] args) {

      //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 circle
      shape3.draw();
   }
}
```

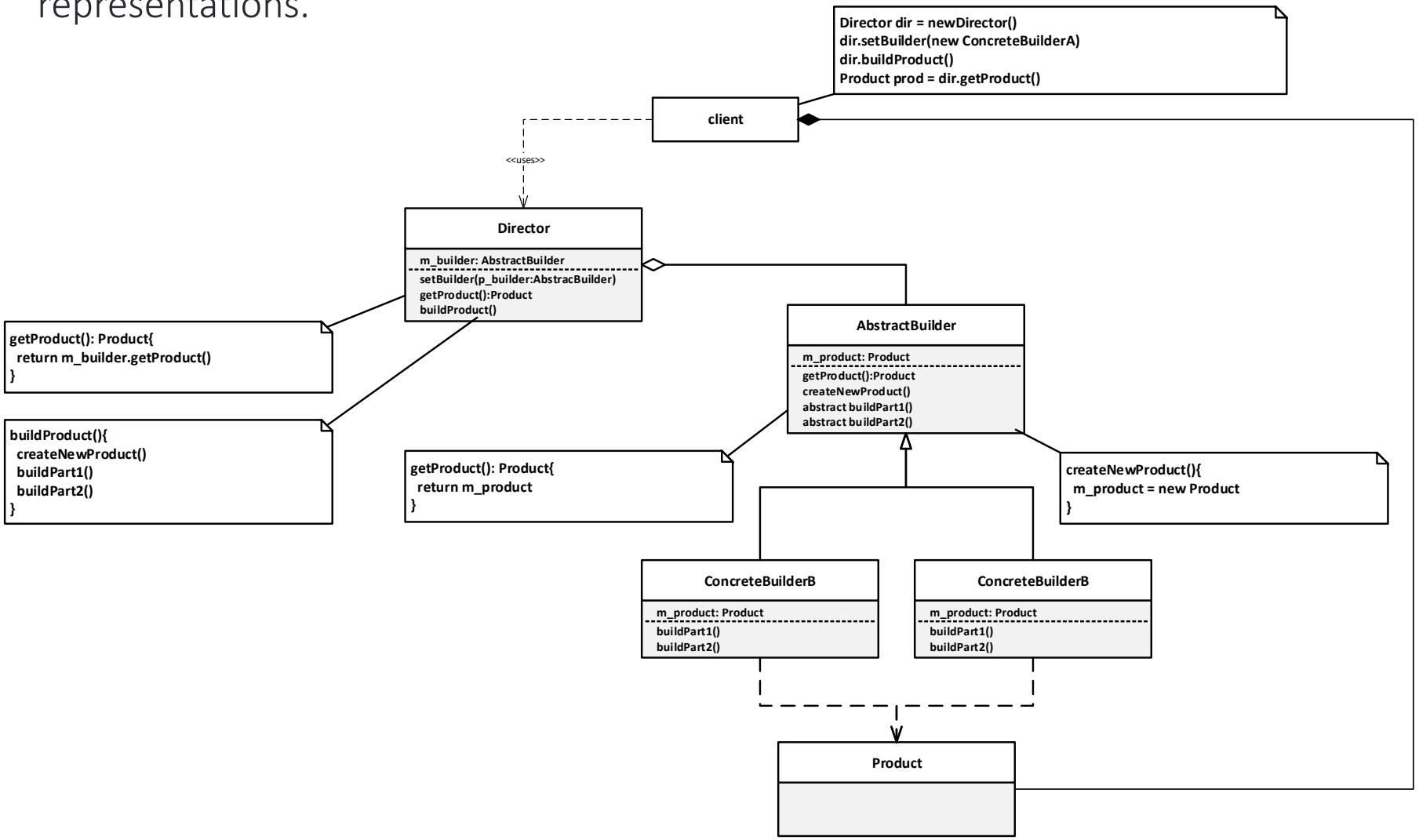
Factory pattern: context and variations

- You should consider using a Factory pattern when:
  - Various kinds of objects are to be used polymorphically.
  - A class uses a hierarchy of classes from which different objects are created, depending on the situation.
  - The code that creates the objects can't anticipate the kind of objects it must create, i.e. the decision varies at run time.
  - You want to localize the knowledge of which class the created object belongs (a.k.a. **type encapsulation**).
- How to recognize variations of the Factory pattern?
  - The base class is (most often) abstract, or an interface.
  - The base class may contain default methods and is only subclassed for cases where the default methods are inappropriate for subclasses.
  - Parameters are passed to the factory telling it which of several class types to return. The parameters may vary, depending on the situation.
  - Classes may share the same method names but may do something quite different.
  - Also often called a "factory method".

Builder pattern

Builder pattern: intent and structure

- The Builder Pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.



Builder pattern: elements

The elements of the Builder pattern are:

`Builder` - specifies an abstract interface for creating parts of a Product object.

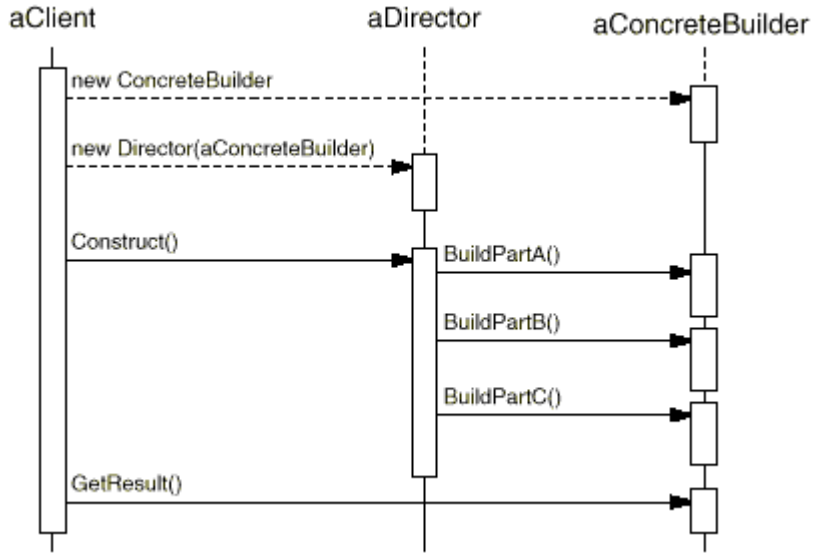
`ConcreteBuilder` - constructs and assembles parts of the product by implementing the Builder interface. Also, it defines and keeps track of the representation it creates and provides an interface for retrieving the product .

`Director` - constructs an object using the Builder interface.

`Product` - represents the complex object under construction.

Builder pattern: behavior

- The following interaction diagram illustrates how Builder and Director cooperate with a client.



- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Builder pattern: example

```
/**
 * Product class of the Builder pattern
 */
public class Shelter {
  /**
   * The components of the shelter to be constructed
   */
  private String roof, structure, floor;

  /**
   * Constructing the roof component
   */
  public void setRoof(String newRoof){
    roof = newRoof;
  }
  /**
   * Constructing the structure component
   */
  public void setStructure(String newStructure){
    structure = newStructure;
  }
  /**
   * Constructing the floor component
   */
  public void setFloor(String newFloor){
    floor = newFloor;
  }
  public String toString(){
    return new String("roof= " + roof + "\nstructure= " + structure + "\nfloor= " + floor + "\n");
  }
}
```

Builder pattern: example

```
/**
 * Abstract Builder class of the Builder pattern
 *
 */
public abstract class ShelterBuilder {
  /**
   * Product to be constructed by the builder
   */
  protected Shelter shelterProduct;
  /**
   * Get the constructed Shelter from the Builder
   */
  public Shelter getShelter(){
    return shelterProduct;
  }
  /**
   * Create a new unspecified Shelter that
   * will be eventually build by calling the
   * following abstract methods in a concrete
   * class derived from the Shelter class
   */
  public void createNewShelter(){
    shelterProduct = new Shelter();
  }
  abstract void buildRoof();
  abstract void buildStructure();
  abstract void buildFloor();
}
```

Builder pattern: example

```
public class PolarShelterBuilder extends ShelterBuilder{
  public void buildRoof(){                          // Build the different parts of the Shelter
    shelterProduct.setRoof("ice dome");
  }
  public void buildStructure(){                     // The construction of the Shelter parts
    shelterProduct.setStructure("ice blocks");      // depends on the type of Shelter being built
  }
  public void buildFloor(){                         // The construction process in a real-life
    shelterProduct.setFloor("caribou skin");        // example may be more complex.
  }
}
```

```
public class TropicalShelterBuilder extends ShelterBuilder{
  public void buildRoof(){
    shelterProduct.setRoof("palm tree leaves");     // The construction process may vary
  }                                                 // across the different Concrete Builders
  public void buildStructure(){
    shelterProduct.setStructure("bamboo");
  }
  public void buildFloor(){
    shelterProduct.setFloor("goat skin");
  }
}
```

Builder pattern: example

```
/**
 * Director of the Builder pattern
 *
 */
public class Explorer {
  /**
   * The Explorer is to use a specific "build plan": the ShelterBuilder
   */
  private ShelterBuilder builder;

  public void setBuilder(ShelterBuilder newShelterBuilder) {
    builder = newShelterBuilder;
  }
  /**
   * The Director assumes that all Shelters have the same parts
   * and each part is built by calling the same method
   * though what these specific methods do may be different.
   */
  public void constructShelter() {
    builder.createNewShelter();
    builder.buildRoof();
    builder.buildStructure();
    builder.buildFloor();
  }
  /**
   * @return gets the Shelter after it has been built
   */
  public Shelter getShelter() {
    return builder.getShelter();
  }
}
```

Builder pattern: example

```
/**
 * Driver class for the Shelter Builder Example
 *
 */
public class Expedition {
  public static void main(String[] args){
    Explorer explorer;
    Shelter hut, igloo;

    ShelterBuilder tropicalBuilder = new TropicalShelterBuilder();
    ShelterBuilder polarBuilder = new PolarShelterBuilder();

    explorer = new Explorer();

    explorer.setBuilder(tropicalBuilder);
    explorer.constructShelter();
    hut = explorer.getShelter();
    System.out.println(hut);

    explorer.setBuilder(polarBuilder);
    explorer.constructShelter();
    igloo = explorer.getShelter();
    System.out.println(igloo);
  }
}
```

Builder pattern: context

- Use the Builder pattern when:
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
  - The construction process must allow different representations for the object that is constructed.
  - Complex objects need to be created that have a common overall structure and interface, even though their internal behavior and detailed structure may be different.

Builder pattern: consequences

- A Builder lets you vary the internal representation of the product it builds. It also hides the details of how the product is assembled. It provides **construction abstraction**.
- Each specific builder is independent of the others and of the rest of the program. This improves modularity and makes the addition of other builders relatively simple.
- Because each builder constructs the final product step-by-step, you have more control over each final product that a Builder constructs.

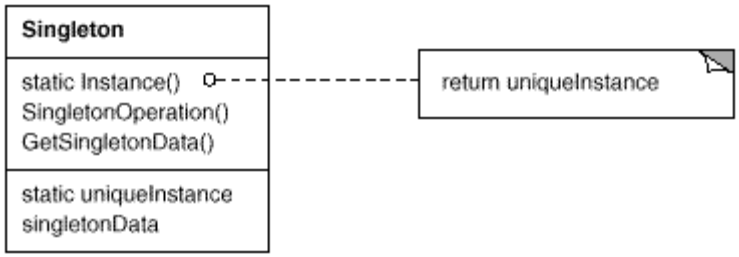
Singleton pattern

Singleton pattern: intent

- Sometimes it is appropriate to have exactly one instance of a class:
  - window managers,
  - print spoolers,
  - filesystems.
- Typically, those types of objects known as singletons, are accessed by disparate objects throughout a software system, and therefore require a global point of access.
- The Singleton pattern addresses all the concerns above. With the Singleton design pattern you can:
  - Ensure that only one instance of a class is created.
  - Provide a global point of access to the object.
  - Allow multiple instances in the future without affecting a singleton class' clients.

Singleton pattern: structure

- The Singleton pattern ensures a class has only one instance, and provides a global point of access to it.
- The class itself is responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance.
- Singletons maintain a static reference to the sole singleton instance and return a reference to that instance from a static instance() method.



Singleton pattern: example

```
public class SingleObject {
   /**
    * create an object of SingleObject embedded as a static member of the class itself
    */
   private static SingleObject instance = new SingleObject();
   /**
    * Make the constructor private so that this class cannot be instantiated
    */
   private SingleObject(){}
   /**
    * If the instance was not previously created, create it. Then return the instance
    */
   public static SingleObject getInstance(){
      if (instance == null)
      instance = new SingleObject();
      return instance;
   }
   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

```
public class SingletonDriver {
   public static void main(String[] args) {

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //use the Singleton
      object.showMessage();
   }
}
```

Singleton: programming concepts used

- The `Singleton` class employs a technique known as **lazy instantiation** to create the singleton; as a result, the singleton instance is not created until the `Instance()` method is called for the first time. This technique ensures that the singleton instance is created only when needed.
- The `Singleton` class implements a protected constructor so clients cannot instantiate `Singleton` instances.
- Protected constructors allow `Singleton` subclasses to be created.
- Even though the `Singleton` is an interesting pattern, it is essentially a global variable, and as such should be used with caution.

Structural patterns

Structural patterns

- Structural patterns describe how classes and objects can be combined to form larger structures.
- The difference between class patterns and object patterns is that class patterns describe how inheritance can be used to provide more useful program interfaces.
- Object patterns, on the other hand, describe how objects can be composed into larger structures using object composition, or the inclusion of objects within other objects.
- Some of the Structural patterns are:
  - Adapter
  - Composite
  - Façade
  - Bridge
  - Decorator
  - Flyweight

Structural patterns

- The **Adapter Pattern** is used to allow a client class to use another class that may provide a service to it, but whose API is incompatible with what the client is expecting.

- The **Flyweight** pattern is a pattern for sharing objects, where each instance does not contain its own state, but stores it externally. This allows efficient sharing of objects to save space, when there are many instances, but only a few different types.

- The **Façade** pattern is used to make a single class represent an entire subsystem.

- The **Decorator** pattern can be used to add features to objects dynamically.

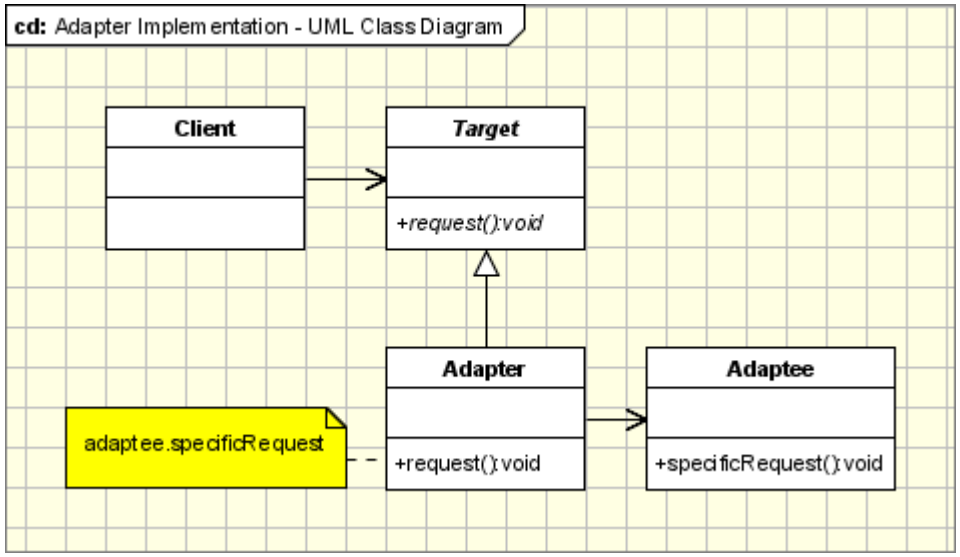
Adapter pattern

Adapter pattern: motivation and intent

- Motivation
  - Like any adapter in the real world it is used to be an interface, a bridge between two objects that have the same functionality, but that are to be used in a different manner, i.e. have a different specification to their interfaces. In the real world we have adapters for power supplies, for memory cards, for video cables, and so on.
  - The same concept applies to software components. You may have some class expecting some type of object and you have an object offering the same features, but exposing a different interface. You don't want to change existing classes, so you want to create an adapter.
- Intent
  - Convert the interface of a class into another interface that clients expect.
  - The adapter lets classes work together, that could not otherwise because of incompatible interfaces.

Adapter pattern: structure

- The classes/objects participating in adapter pattern:
  - **Target** - defines the domain-specific interface that Client uses.
  - **Adapter** - adapts the interface Adaptee to the Target interface.
  - **Adaptee** - defines an existing interface that needs adapting.
  - **Client** - collaborates with objects conforming to the Target interface.



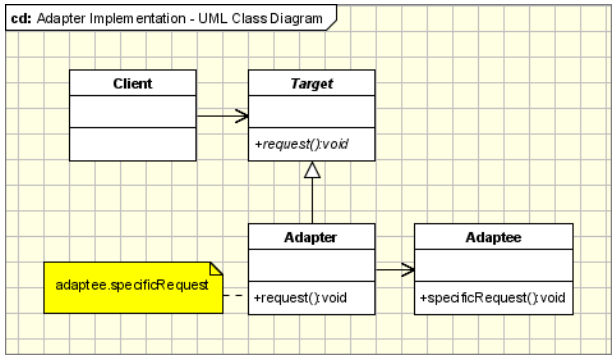
Adapter pattern: context

- The adapter pattern is used when:
  - When you have a client class that invokes methods defined in a certain class or interface (**Target**) and you have another class (**Adaptee**) that doesn't implement the same interface as the Target, but still implements a similar service.
  - You can change none of the existing code (nor **Target** or **Adaptee** can be changed).
  - You need an adapter to implement the interface that will be the bridge between the client class and the **Adaptee**.

Adapter pattern: example

- If a client only understands the `TextFileReader` interface for reading a file using the `readFile()` method, how can it read JSON files for which we have a reader, but behaves differently, using the `readJSONFileType()` method that returns a different kind of map?

```
/**
* This is the Target class.
*/
public class TextFileReader {
  public Map readFile(String str) {
    System.out.println("TextFileReader.readFile(): " + str);
    Map createdMap = new Map();
    // read the file and create the map object
    return createdMap;
  }
}
```

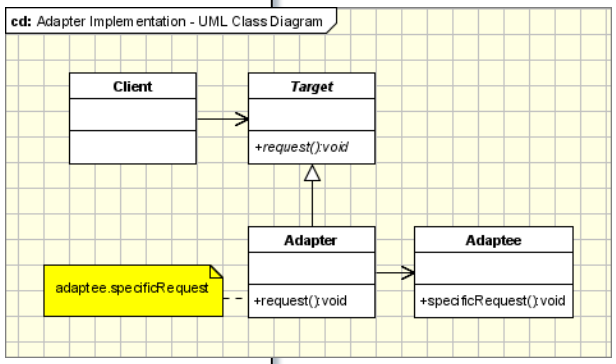



```
/**
* This is the Adaptee class.
*/
public class JSONFileReader {
  public OtherMap readJSONFileType(String msg) {
    System.out.println("JSONFileReader.readOtherFileType(): " + msg);
    OtherMap createdMap = new OtherMap();
    // read the file and create the map object
    return createdMap;
  }
}
```

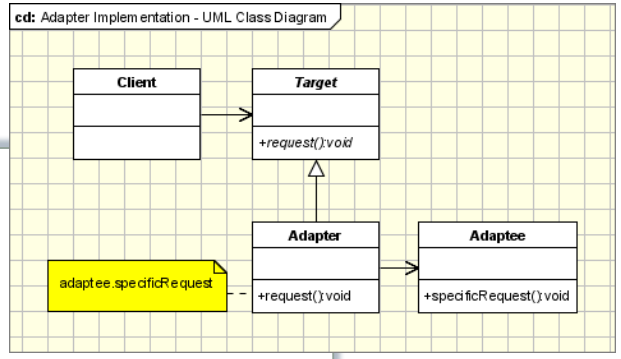
Adapter pattern: example

- Solution:
  - Design a `SquareToRoundPeg` adapter that enables to call `readJSONFileType()` on a `JSONFileReader` object connected to the adapter to be inserted as a `TextFilerReader`, using `readFile()`.

```
/**
 * The FileReaderAdapter class. This is the Adapter class.
*/
public class FileReaderAdapter extends TextFileReader {

  private JSONFileReader otherFileType;

  public FileReaderAdapter(JSONFileReader p_fr) {
    // the roundPeg is plugged into the adapter
    this.otherFileType = p_fr;
  }

  public Map readFile(String str) {
    // the roundPeg can now be inserted in the same manner as a squarePeg!
    return (translate(otherFileType.readJSONFileType(str)));
  }

  private Map translate(OtherMap p_om) {
    Map translatedMap = new Map();
    // translate the OtherMap object into a Map object
    return translatedMap;
  }
}
```



Adapter pattern: example

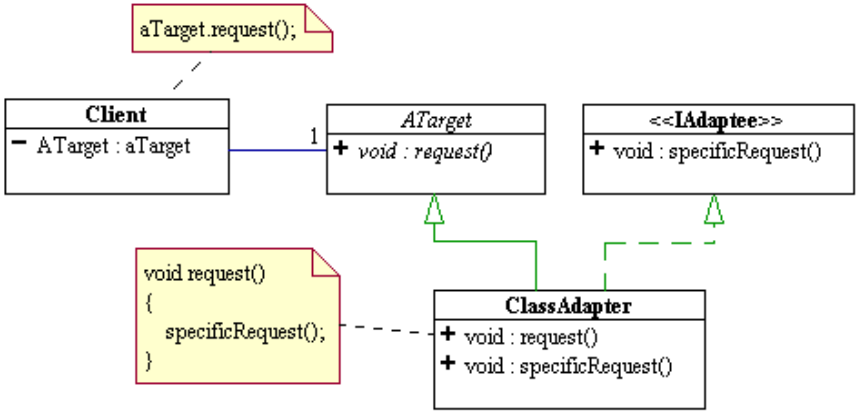



```java
/**
 * Driver program using the Peg Adapter
 */
public class FileReaderAdapterDriver{
  public static void main(String args[]) {

    TextFileReader originalFileType = new TextFileReader();
    Map map;

    // Do an insert using the original FileReader
    map = originalFileType.readFile("filename.txt");

    // Now we'd like to read a JSON file.
    // But this client only understands the readFile()
    // method of TextFileReader, not the readJSONFileType() method.
    // The solution: create an adapter that adapts
    // a JSONFileReader into a TextFileReader!

    TextFileReader wannabeAMap = new FileReaderAdapter(new JSONFileReader());
    map = wannabeAMap.readFile("filename.JSON");}
}
```

```
Execution:
TextFileReader.readFile(): filename.txt
JSONFileReader.readOtherFileType(): filename.JSON
```

Adapter variation: Class Adapter: example

- Classic adapters are “one-way”. What if we want a “two-way adapter”?
- This is called a “Class Adapter”.
- Class Adapters use multiple inheritance to achieve their goals.
- As in the classic adapter (often called an “object adapter”), the class adapter inherits the the client's Target class. However, it inherits the Adaptee as well.
- Since Java does not support true multiple inheritance, this means that one of the classes must be a Java interface type.
- Both of the Target and Adaptee interfaces could be Java interfaces, but only one needs to be an interface.
- The request to the Target is simply rerouted to the specific request that was inherited from the Adaptee interface.



Class adapter: example

```java
/**
* The IRoundPeg interface.
*/
public interface IRoundPeg {
  public void insertIntoHole(String msg);
}
```

```java
/**
 * The RoundPeg Target/Adaptee class.
 */
public class RoundPeg implements IRoundPeg {
  public void insertIntoHole(String msg) {
    System.out.println("RoundPeg insertIntoHole(): " + msg);}
}
```

```java
/**
* The ISquarePeg interface.
*/
public interface ISquarePeg {
  public void insert(String str);
}
```

```java
/**
 * The SquarePeg Target/Adaptee class.
 */
public class SquarePeg implements ISquarePeg {
  public void insert(String str) {
    System.out.println("SquarePeg insert(): " + str);}
}
```

Class adapter: example

```
/**
* The PegAdapter class. This is a two-way adapter class.
*/
public class PegAdapter implements ISquarePeg, IRoundPeg {
  private RoundPeg roundPeg;
  private SquarePeg squarePeg;

  public PegAdapter(RoundPeg peg) {
    this.roundPeg = peg;}
  public PegAdapter(SquarePeg peg) {
    this.squarePeg = peg;}

  public void insert(String str) {
    roundPeg.insertIntoHole(str);}
  public void insertIntoHole(String msg){
    squarePeg.insert(msg);}
}
```

Class adapter: example

```
/**
 * Driver class for two way Peg Adapter.
 */
public class TwoWayAdapterDriver {
  public static void main(String args[]) {

    // Create some pegs.
    RoundPeg roundPeg = new RoundPeg();
    SquarePeg squarePeg = new SquarePeg();

    // Do an insert using the square peg.
    squarePeg.insert("I am a SquarePeg into a square hole.");

    // Create a two-way adapter and do an insert with it.
    ISquarePeg wannabeRound = new PegAdapter(roundPeg);
    wannabeRound.insert("I am a SquarePeg into a round hole!");

    // Do an insert using the round peg.
    roundPeg.insertIntoHole("I am a RoundPeg into a round hole.");

    // Create a two-way adapter and do an insert with it.
    IRoundPeg wannabeSquare = new PegAdapter(squarePeg);
    wannabeSquare.insertIntoHole("I am a RoundPeg into a square hole!");}
}
```

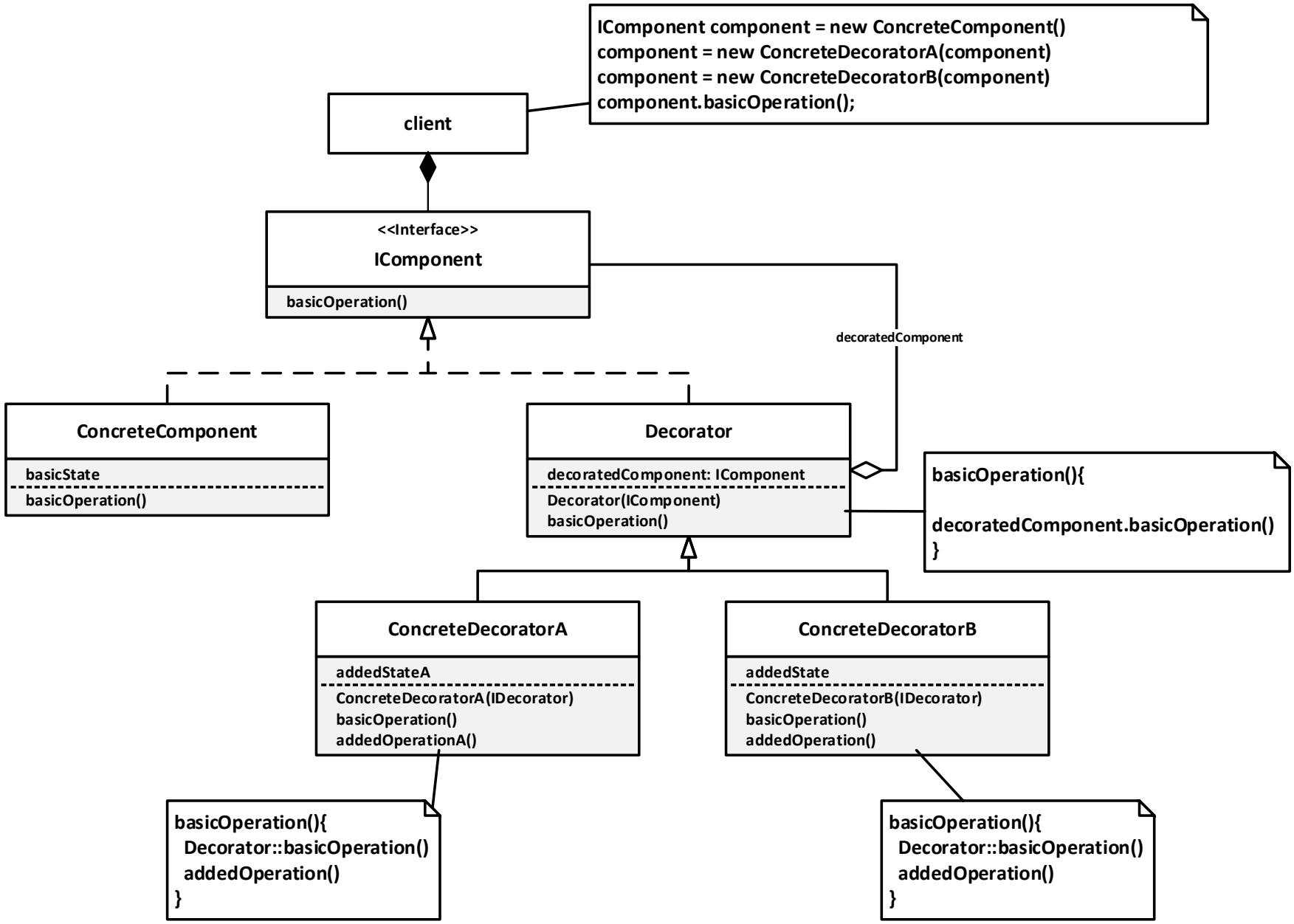
```
SquarePeg insert(): I am a SquarePeg into a square hole.
RoundPeg insertIntoHole(): I am a SquarePeg into a round hole!
RoundPeg insertIntoHole(): I am a RoundPeg into a round hole.
SquarePeg insert(): I am a RoundPeg into a square hole!
```

Decorator pattern

Decorator pattern: motivation and intent

- Motivation:
  - Sometimes we may want to dynamically add some data members or methods to an object at runtime, depending on the situation.
- Intent:
  - Allow to add new functionality to an existing object without altering its structure.
  - Create a Decorator class that wraps the original class.
  - Provides additional functionality while keeping the class' methods' signatures intact.

Decorator pattern: structure



Decorator pattern: elements

Elements of the Decorator pattern:

`Component` - Abstract class representing the objects to be decorated by the various Decorators.

`Concrete Component` - The potentially many sub-classes that can be decorated.

`Decorator` - Abstract class that wraps a Component and will have some of its subclasses to decorate it.

`Concrete Decorator` - Different decorators that add different members to the Component and use these members to provide additional behavior to the decorated object.

Decorator pattern: example

```java
/**
 * The abstract Coffee class defines the functionality of any Coffee
 * implemented by subclasses of Coffee
 */
public abstract class Coffee {
    public abstract double getCost();
    public abstract String getIngredients();
}
```

```java
/**
 * Kind of Coffee
 */
public class Espresso extends Coffee {
    public double getCost() {
        return 1.25;
    }

    public String getIngredients() {
        return "Strong Coffee";
    }
}
```

```java
/**
 * Kind of Coffee
 */
public class SimpleCoffee extends Coffee {
    public double getCost() {
        return 1;
    }

    public String getIngredients() {
        return "Coffee";
    }
}
```

Decorator pattern: example

```
/**
 * Abstract decorator class - note that it extends the Coffee abstract class
 */
public abstract class CoffeeDecorator extends Coffee {
    protected final Coffee decoratedCoffee;
    /**
     * Wraps a Coffee object inside an object of one of
     * CoffeeDecorator's subclasses
     */
    public CoffeeDecorator (Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }
    /**
     * Provides the wrapper with the Coffee interface and delegates
     * its methods to the wrapped Coffee object.
     */
    public double getCost() {
        return decoratedCoffee.getCost();
    }
    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}
```

Decorator pattern: example

```
/** Decorator that mixes Milk with coffee.
 *  It is a subclass of CoffeeDecorator, and thus a subclass of Coffee.
 */
class Milk extends CoffeeDecorator {
/**
 * When creating a decorated Coffee, pass a Coffee to be decorated
 * as a parameter. Note that this can be an already-decorated Coffee.
 */
    public Milk (Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }
    /**
     * Overriding methods defined in the abstract superclass.
     * Enables to provide different behavior for decorated Coffee methods
     */
    public double getCost() {
        return super.getCost() + 0.5;
    }
    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
    /**
     * May also add additional members for decorated-specific data
     * or behavior
     */
}
```

Decorator pattern: example

```
class Sprinkles extends CoffeeDecorator {
  public Sprinkles (Coffee decoratedCoffee) {
    super(decoratedCoffee);
  }
  public double getCost() {
    return super.getCost() + 0.2;
  }
  public String getIngredients() {
    return super.getIngredients() + ", Sprinkles";
  }
}
```

```
class Whip extends CoffeeDecorator {
  public Whip (Coffee decoratedCoffee) {
    super(decoratedCoffee);
  }
  public double getCost() {
    return super.getCost() + 0.7;
  }
  public String getIngredients() {
    return super.getIngredients() + ", Whip";
  }
}
```

Decorator pattern: example

```
public class DecoratorDriver {
  public static final void main(String[] args) {
    Coffee c = new SimpleCoffee();
    System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

    c = new Milk(c);
    System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

    c = new Sprinkles(c);
    System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

    c = new Whip(c);
    System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

    // Note that you can also stack more than one decorator of the same type
    c = new Sprinkles(c);
    System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

    c = new Espresso();
    System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

    c = new Milk(c);
    System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());
  }
}
```

```
Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
Cost: 2.4; Ingredients: Coffee, Milk, Sprinkles, Whip
Cost: 2.6; Ingredients: Coffee, Milk, Sprinkles, Whip, Sprinkles
Cost: 1.25; Ingredients: Strong Coffee
Cost: 1.75; Ingredients: Strong Coffee, Milk
```

Behavioral patterns

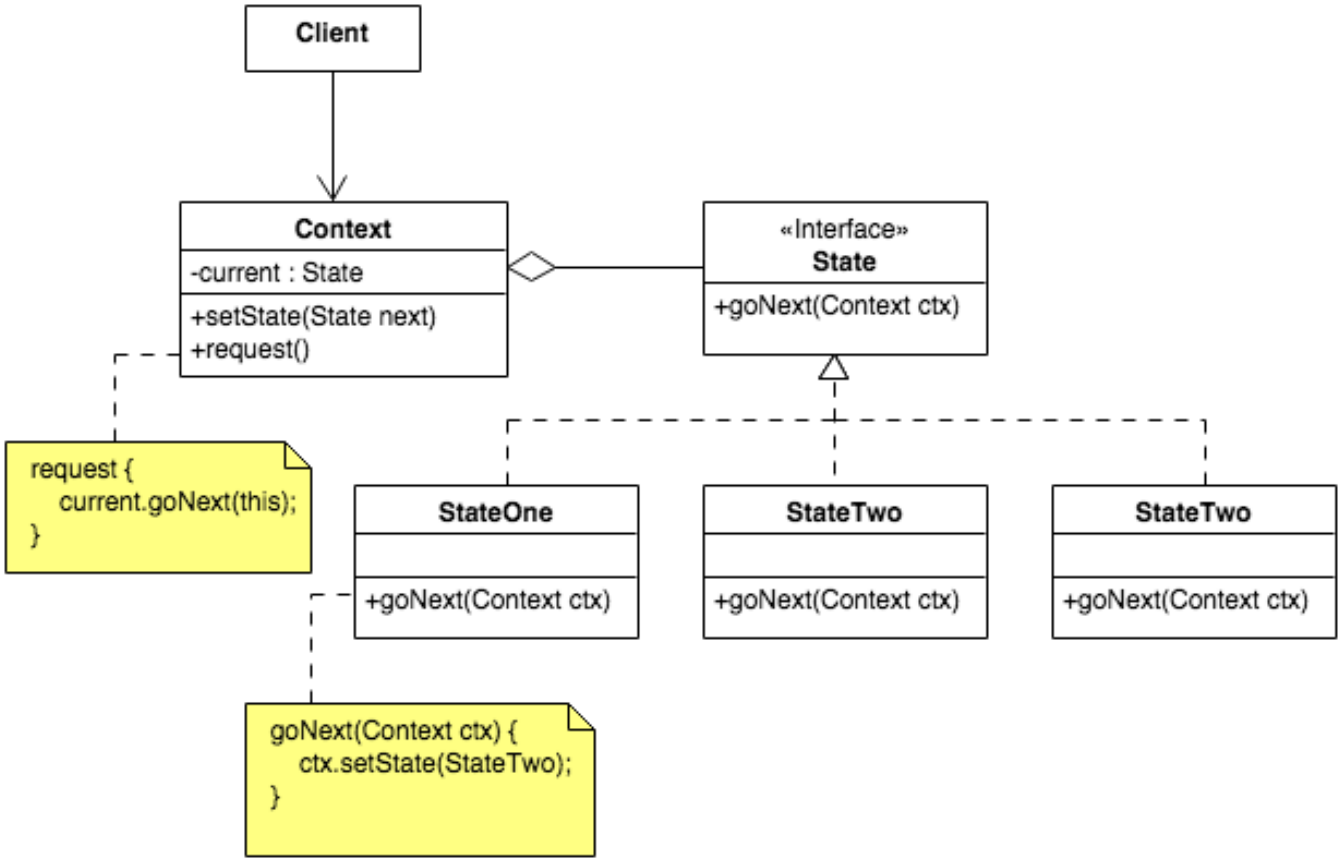
State pattern

State pattern: motivation and intent

- Motivation
  - Sometimes we want to be able to alter the behavior of an object when its internal state changes, and make it easy to add new varying behavior that comes with new states.

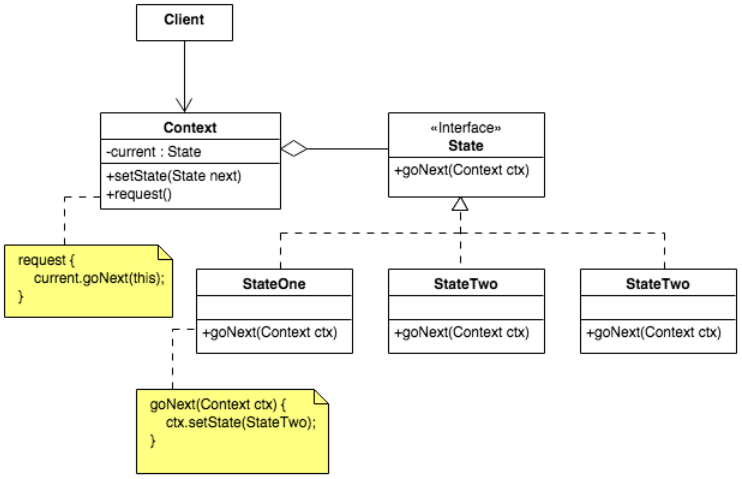
- Intent
  - Encapsulate the varying behavior in different classes associated with different states.
  - Inject the state-based behavior in the target object by having the object to contain an instance of the state-based encapsulated behavior.
  - Upon request of the object's behavior, implement its behavior so that it uses the encapsulated state-based behavior.

State pattern : structure



State pattern: elements

- The elements of the **State** pattern are:
  - **Context** – The object whose behavior needs to be state-specific. It maintains a reference to a **ConcreteState** object which is used to define the current state of the **Context** object. Some of its methods will use the state-specific behavior of the **State** object to provide state-specific behavior. When the state object is changed, the behavior of the **Context** object will change.
  - **State** – The class that defines the operations that each state must handle. Generally implemented as an abstract class or interface.
  - **ConcreteState** - The classes that implement the state-specific behavior.



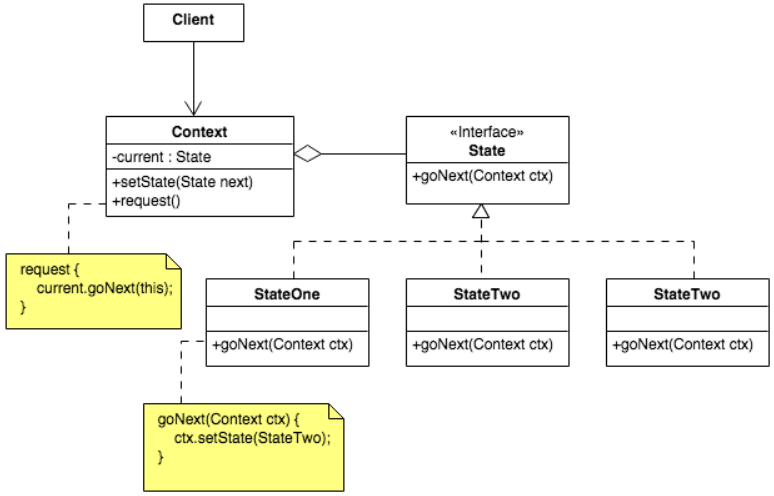
State pattern: behavior

- Behavior

1. As the **Context** object is created, it is setup with an initial **State** object.
2. When the state-dependent behavior of the **Context** object is called, it uses the **State** object's methods to provide state-dependent behavior.
3. State changes can happen:
   i. From an external call to **setState()** on the **Context** object (if this method is public.
   ii. From an internal call to **setState()** by the **Context** object, upon a certain state-transitional condition.
   iii. From a call to **setState()** from within the **State** object upon a state-transitioning condition, in which case the **State** object must contain a references to the **Context** object.

State pattern: example

```java
// Context of the State pattern.
public class GameEngine {

  private Phase gamePhase;

  public void setPhase(Phase p_phase) {
    gamePhase = p_phase;
  }

  public void start() {
    …
    // Can change the state of the Context (GameEngine) object, e.g.
    setPhase(new Preload(this));
    setPhase(new PlaySetup(this));
    …
    // Can trigger State-dependent behavior by using
    // the methods defined in the State (Phase) object, e.g.
    gamePhase.loadMap();
    gamePhase.reinforce();
    gamePhase.next();
  }
  …
}
```

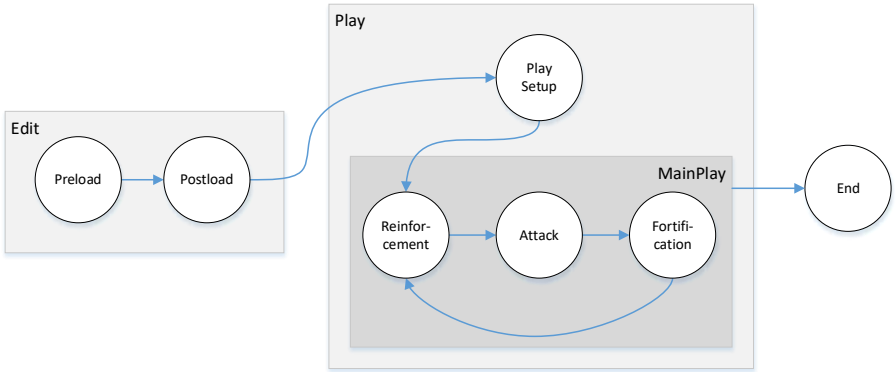


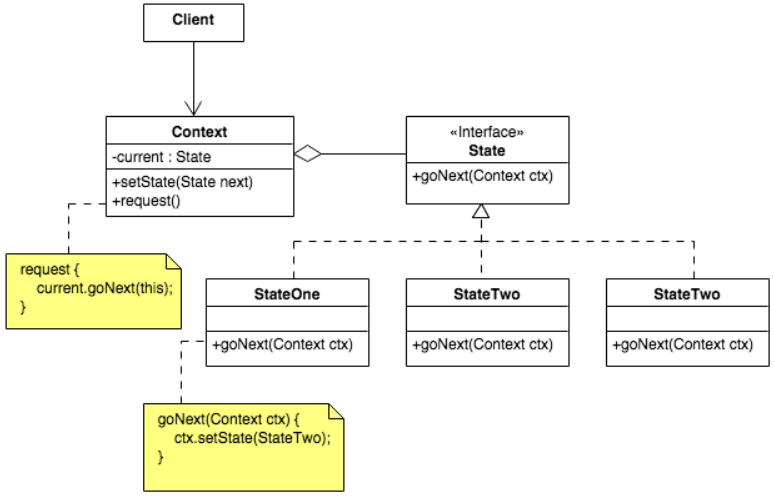

```java
// Client in the State pattern
public class GameDriver {
  public static void main(String args[]) {
    GameEngine gameEngine = new GameEngine();
    gameEngine.start();
  }
}
```

```java
// State in the State pattern
public abstract class Phase {

  GameEngine ge;

  // general behavior
  abstract public void loadMap();
  abstract public void showMap();
  // edit map state behavior
  abstract public void editCountry();
  abstract public void saveMap();
  // play state behavior
  // game setup state behavior
  abstract public void setPlayers();
  abstract public void assignCountries();
  // reinforcement state behavior
  abstract public void reinforce();
  // attack state behavior
  abstract public void attack();
  // fortify state behavior
  abstract public void fortify();
  // end state behavior
  abstract public void endGame();
  // go to next phase
  abstract public void next();

  // methods common to all states
  public void printInvalidCommandMessage() {
    System.out.println("Invalid command in state "
    + this.getClass().getSimpleName() );
  }
}
```

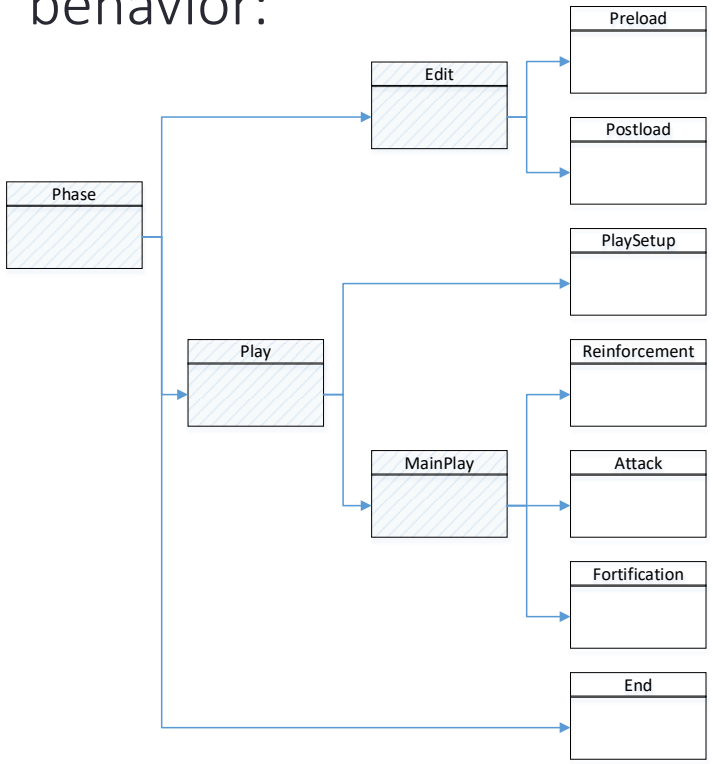
State pattern: example

- Transitions need to be carefully thought of:





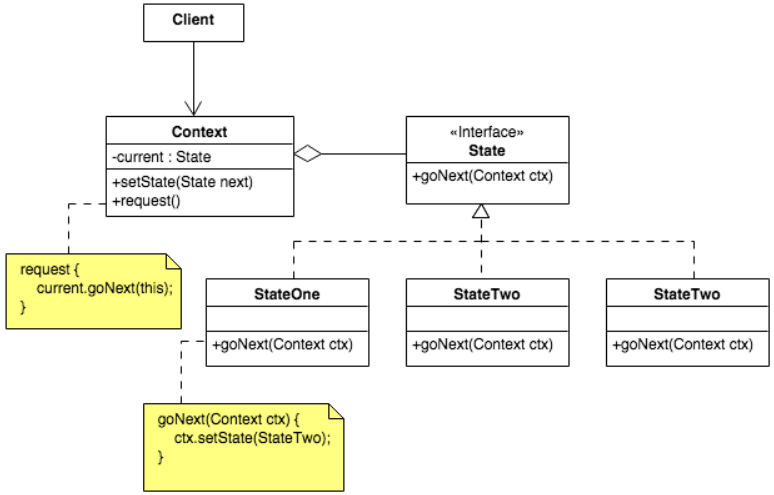
- States can be grouped to provide state group-based behavior:



State pattern: example

```
// State of the State pattern
public abstract class Play extends Phase {

  public void showMap() {
    System.out.println("map is being displayed");
  }

  public void editCountry() {
    printInvalidCommandMessage();
  }

  public void saveMap() {
    printInvalidCommandMessage();
  }

  public void endGame() {
    ge.setPhase(new End(ge));
  }
}
```

```
// State of the State pattern
public abstract class MainPlay extends Play {

  public void loadMap() {
    this.printInvalidCommandMessage();
  }

  public void setPlayers() {
    this.printInvalidCommandMessage();
  }

  public void assignCountries() {
    this.printInvalidCommandMessage();
  }
}
```




```
// ConcreteState of the State pattern
public class Attack extends MainPlay {

  public void next() {
    ge.setPhase(new Fortify(ge));
  }

  public void reinforce() {
    printInvalidCommandMessage();
  }

  public void attack() {
    System.out.println("attack done");
    ge.setPhase(new Fortify(ge));
  }

  public void fortify() {
    printInvalidCommandMessage();
  }
}
```

State pattern: example

```
public abstract class Edit extends Phase {

  public void showMap() {
    System.out.println("edited map is displayed");
  }

  public void setPlayers() {
    printInvalidCommandMessage();
  }

  public void assignCountries() {
    printInvalidCommandMessage();
  }

  public void reinforce() {
    printInvalidCommandMessage();
  }

  public void attack() {
    printInvalidCommandMessage();
  }

  public void fortify() {
    printInvalidCommandMessage();
  }

  public void endGame() {
    printInvalidCommandMessage();
  }
}
```

```
public class Preload extends Edit {

  public void loadMap() {
    System.out.println("map has been loaded");
    ge.setPhase(new PostLoad(ge));
  }

  public void editCountry() {
    printInvalidCommandMessage();
  }

  public void saveMap() {
    printInvalidCommandMessage();
  }

  public void next() {
    System.out.println("must load map");
  }
}
```

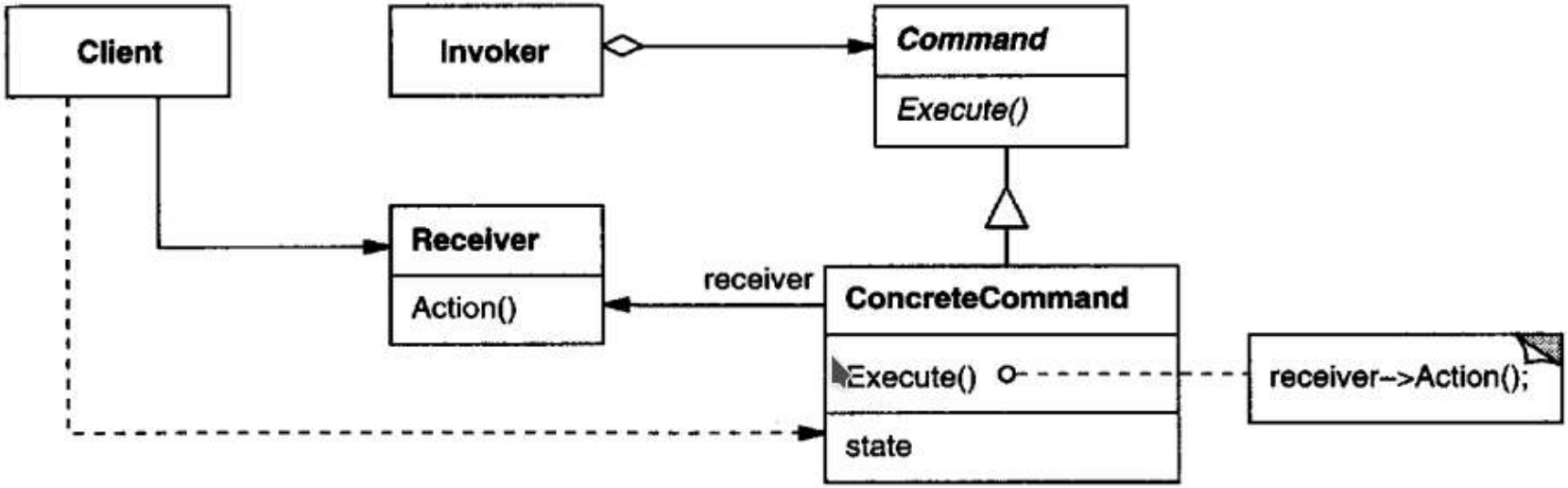
```
public class PostLoad extends Edit {

  public void loadMap() {
    System.out.println("map has been loaded");
  }

  public void editCountry() {
    System.out.println("country has been edited");
  }

  public void saveMap() {
    System.out.println("map has been saved");
    ge.setPhase(new PlaySetup(ge));
  }

  public void next() {
    System.out.println("must save map");
  }
}
```

Command pattern

Command pattern: motivation and intent

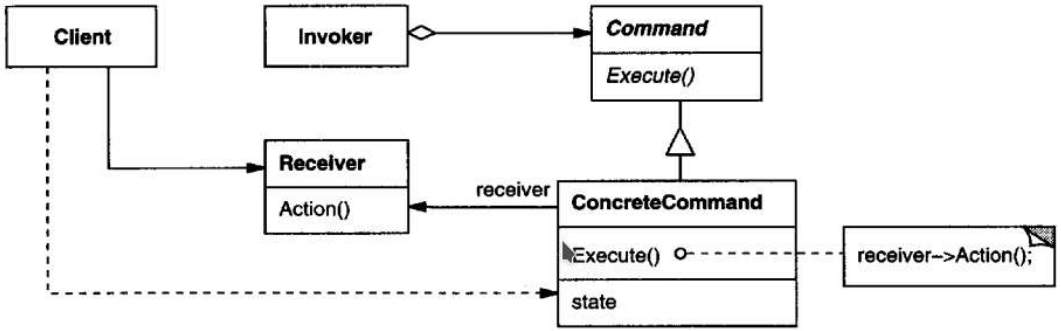
- Motivation
  - Sometimes a request for a result is created at a different time/site compared to where/when it will be executed, or we don't want to know what is the actual method being executed to process the request, or what object will eventually execute it. In these cases, we may want to create a request for a result, store it, then process it elsewhere or some time after.
- Intent
  - Separate an operation from the object that creates it and the object that executes it.
  - Encapsulate a request and all its required parameters so that they can be stored and conveyed to the executor as a self-contained unit .

Command pattern : structure



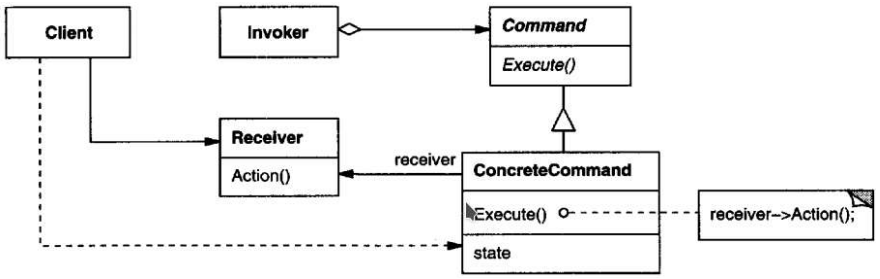
Command pattern: elements

- The elements of the **Command** pattern are:
  - **Invoker** – Object that creates the command object to carry out the operation.
  - **Receiver** – Object that will be affected/used when the command gets executed.
  - **Command** – Class that defines the operations that each state must handle. Generally implemented as an abstract class or interface.
  - **ConcreteCommand** – Object that contains the context necessary for the execution of the operation, and that implements code that carries the actual operation.



Command pattern: behavior

- Behavior
  1. The **Invoker** and the **Client** are created.
  2. The **Client** instantiates a **Command** object as an instance of one of the **ConcreteCommand** classes. All the parameters/context necessary to execute the **Command** are encapsulated inside the **Command** object. The **Receiver** is part of the context of execution of the **Command**.
  3. The **Command** object is conveyed to the **Client**.
  4. The **Client** executes the **execute()** method of the **Command** object, which calls methods and/or then affects the state of the **Receiver**.

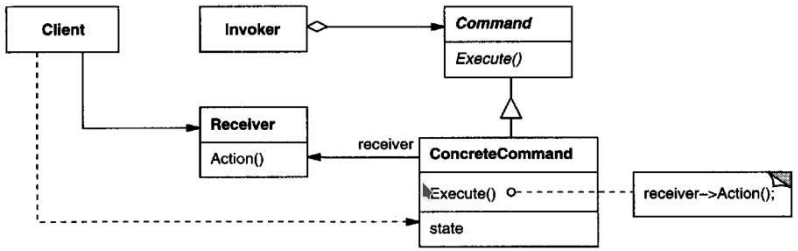


Command pattern: example

```
// Client of the Command pattern
public class GameEngine {
  List<Territory> map;
  List<Player> players;
  public void start() {

    for (int turn = 1; turn <= numTurns; turn++) {
      boolean an_order = true;
      do {
        for (Player p : players) {
          an_order = p.createOrder(map, players);
          if (!an_order)
            break;
        }
      } while (an_order);
      executeAllOrders();
      printMap();
    }
  }

  public void executeAllOrders() {
    Order order;
    boolean still_more_orders = false;
    do {
      still_more_orders = false;
      for (Player p : players) {
        order = p.getNextOrder();
        if (order != null) {
          still_more_orders = true;
          order.printOrder();
          order.execute();
        }
      }
    } while (still_more_orders);
  }
}
```

```
// Receiver in the Command pattern
public class Player {
  public ArrayList<Order> orders_list;

  public boolean createOrder(List<Territory> map, List<Player> players) {
    …
    orders_list.add(new Deploy(this, target, num));
    orders_list.add(new Advance(this, source, target, num));
    orders_list.add(new Pacify(this, player));
    …
  }

  public Order getNextOrder() {
    if (!this.orders_list.isEmpty()) {
        to_return = this.orders_list.get(0);
        this.orders_list.remove(0);
        return to_return;
    } else
      return null;
  }
}
```

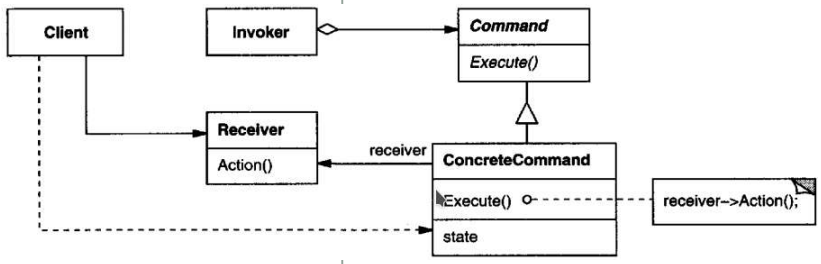
```
// Driver of the Command pattern
public class CommandsDriver {
  public static void main(String args[]) {
    GameEngine ge = new GameEngine();
    ge.start();
  }
}
```



Command pattern: example

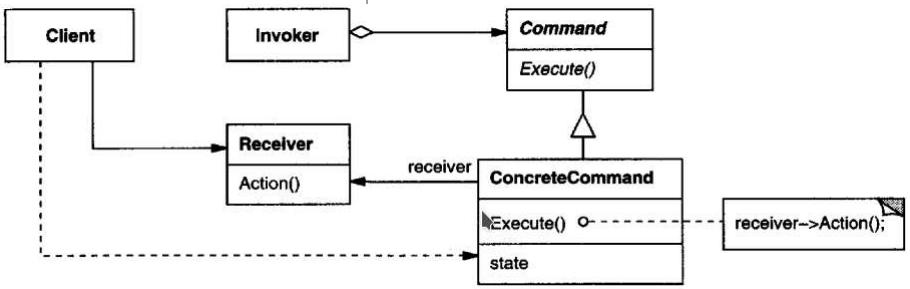
```java
// ConcreteCommand of the Command pattern
public class Deploy implements Order {

Territory target_territory;
int to_deploy;
Player initiator;

  public Deploy(Player initiator, Territory target_territory, int to_deploy) {
    // encapsulate all necessary data to execute the command
    this.target_territory = target_territory;
    this.initiator = initiator;
    this.to_deploy = to_deploy;
  }

  public void execute() {
    // Here, the target Territory object is the Receiver
    if (valid())
    // behavior of the concrete command
    this.target_territory.numArmies += to_deploy;
  }

  public boolean valid() {
    if (target_territory.owner.equals(initiator))
      // the target territory must belong to the player that created the order
      return true;
    else {
      System.out.println("invalid order");
      return false;
    }
  }

  public void printOrder() {
    System.out.println("Deploy order issued by player " + this.initiator.name);
    System.out.println("Deploy " + this.to_deploy + " to " + this.target_territory.name);
  }
}
```

```java
// Command of the Command pattern
public interface Order {
  public void execute();
  public boolean valid();
  public void printOrder();
}
```



Command pattern: example

```
// ConcreteCommand of the the Command pattern.
public class Advance implements Order {
  Territory source;
  Territory target;
  Player initiator;
  int num_to_advance;

  Advance(Player initiator, Territory source, Territory target, int num) {
    // encapsulate all necessary data to execute the command
    this.initiator = initiator;
    this.source = source;
    this.target = target;
    this.num_to_advance = num;
  }

  public void execute() {
    // Here both the source and the target Territories are Receivers
    if (valid()) {
      if (target.owner.equals(initiator)) {
        // if the source and the target belong to the same player
        // then just move the armies to the target Territory
      } else {
        // implement a battle
        if (target.numArmies < 0) {
           // move surviving attacking armies to the target country
           // transfer ownership of the conquered country
        }
      }
    }
  }

  public boolean valid() {
    if (valid conditions)
      return true;
    else {
      System.out.println("invalid order");
      return false;
    }
  }

  public void printOrder() {
    // print the order
  }
}
```

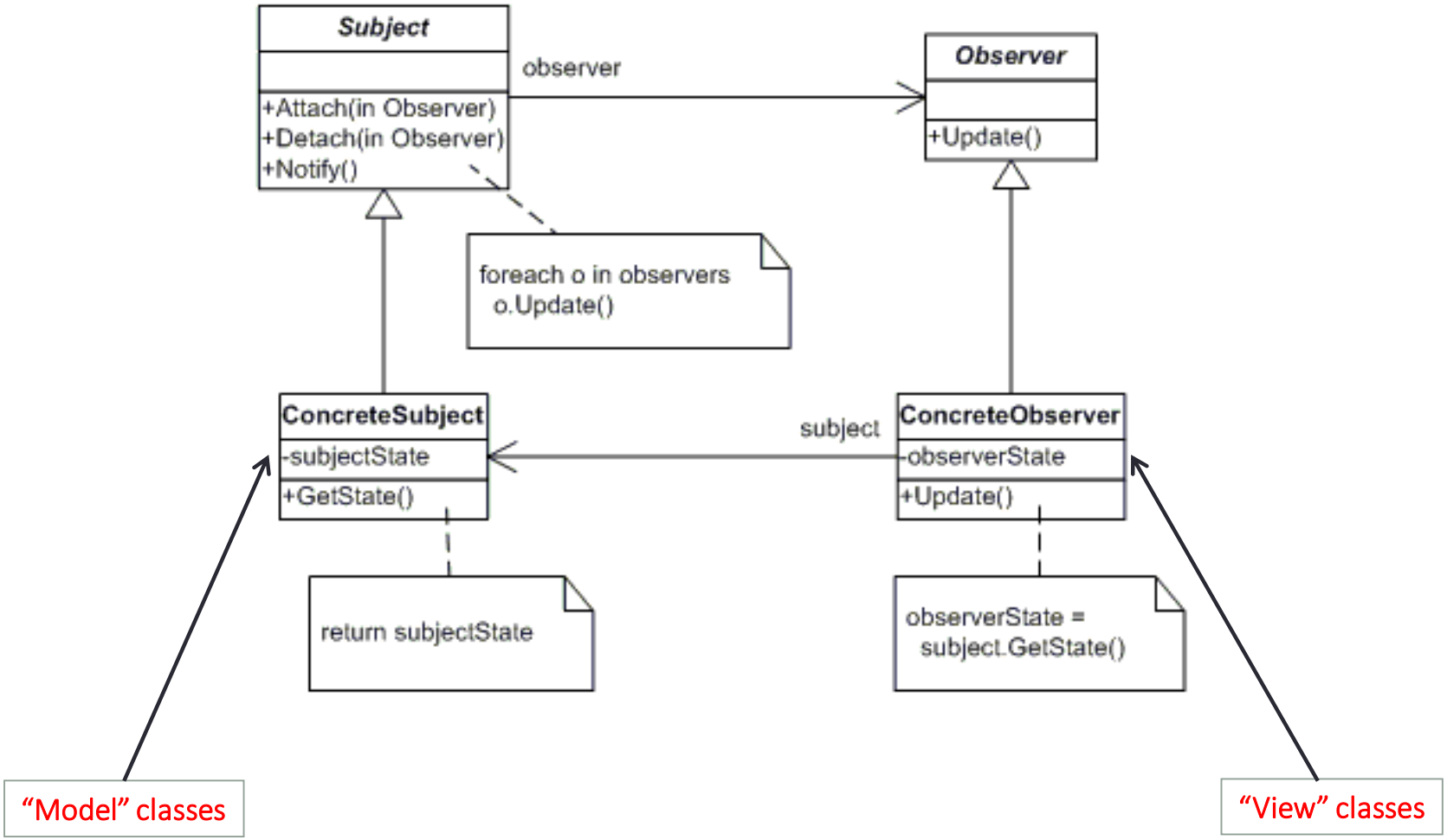


Observer pattern

Observer pattern: motivation and intent

- Motivation
  - The cases when certain objects need to be informed about the changes occurring in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- This pattern is a cornerstone of the Model-View-Controller architectural design, where the Model implements the mechanics of the program, and the Views are implemented as Observers that are as much uncoupled as possible from the Model components.

Observer pattern: structure



Observer pattern: elements

- The elements of the Observer pattern are:
  - **Subject** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. It is often referred to as “Observable”.
  - **ConcreteSubject** - concrete Subject class. It maintain the state of the observed object and when a change in its state occurs it notifies the attached Observers. If used as part of MVC, the ConcreteSubject classes are the Model classes that have Views attached to them.
  - **Observer** - interface or abstract class defining the operation to be used to update the information gathered from the Subject.
  - **ConcreteObserver** - concrete Observer subclasses that are attached to a particular Subject class. There may be different concrete observers attached to a single Subject that will provide a different view of that Subject.

Observer pattern: behavior

- Behavior
  - The client class instantiates the ConcreteObservable object.
  - Then it instantiates and attaches the concrete observers to it using the methods defined in the Observable interface.
  - Each time the (observable) state of the subject is changing, it notifies all the attached Observers using the methods defined in the Observer interface.
  - When a new Observer is added to the application, all we need to do is to instantiate it in the client class and to add attach it to the Observable object.
  - The classes already created will remain mostly unchanged.

Observer pattern: example

```
import java.util.ArrayList;
import java.util.List;
public class Observable {
  private List<Observer> observers = new ArrayList<Observer>();
  public void attach(Observer o){
    this.observers.add(o);
  }
  public void detach(Observer o){
    if (!observers.isEmpty()) {
      observers.remove(o);
    }
  }
  public void notifyObservers(Observable observable) {
    for (Observer observer : observers) {
      observer.update(observable);
    }
  }
}
```

```
public interface Observer {
   public void update(Observable o);
}
```

Observer pattern: example

- Model classes need to be made subclasses of the Observable class.
- When a part of their state that is observed is changed, call the **`notifyObservers()`** method that will notify all the attached observers by calling their **`update()`** method.

```
class ClockTimerModel extends Observable {
  public
    int GetHour(){return hour;};
    int GetMinute(){return minute;};
    int GetSecond(){return second;};
    void tick(){
      // update internal state
      second ++;
      if (second >= 60){
        minute++;
        second = 0;
        if (minute >=60){
          hour++;
          minute=0;
          if (hour >= 24){
            hour=0;
          };
        };
      };
      // notify all attached Observers of a change
      notifyObservers(this);
    };
    void start(int secs){
      for (int i = 1; i <= secs; i++)
        tick();
    };
  private
    int hour;
    int minute;
    int second;
};
```

Observer pattern: example

- View classes implement the **`Observer`** interface.
- They need to implement the **`update()`** method that is called by the **`notifyObservers()`** method of the **`Observable`** class.
- This method does the necessary updates to the view offered to the user (here a simple console output).

```
class DigitalClockView implements Observer {
  public
    void update(Observable obs) {
      //redraw my clock's reading after I was notified
      int hour   = ((ClockTimerModel) obs).GetHour();
      int minute = ((ClockTimerModel) obs).GetMinute();
      int second = ((ClockTimerModel) obs).GetSecond();
      System.out.println(hour+":"+minute+":"+second);
    };
};
```

Observer pattern: example

- The main program must:
  1. Create the Model object.
  2. Create the View object.
  3. Connect the View object to the Model object.
- Then, any change in the state of the Model object triggers the update of the Views attached to it.

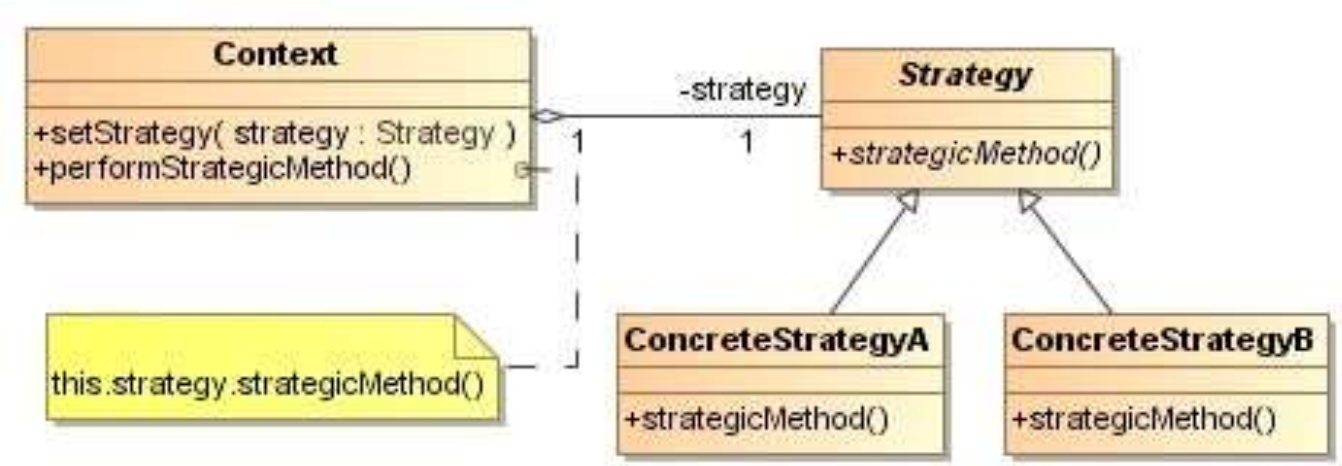
```
import java.util.Scanner;

public class ObserverDemo extends Object {
  DigitalClockView clockView;
  ClockTimerModel clockModel;

  public ObserverDemo() {
    //create the View object
    clockView = new DigitalClockView();
    //create the Model object
    clockModel = new ClockTimerModel();
    //connect the View object to the Model object
    clockModel.attach(clockView);
  };

  public static void main(String[] av) {
    ObserverDemo od = new ObserverDemo();
    od.demo();
  };

  public void demo() {
    Scanner kbd = new Scanner(System.in);
    int secs = kbd.nextInt();
    clockModel.start(secs);
    kbd.close();
  };
};
```

Strategy pattern

Strategy pattern

- Motivation:
  - Sometimes we may want to change the behavior of an object depending on some conditions that are only to be determined at runtime, or to easily add new definitions of a certain behavior without altering the class that is using it.
- Intent:
  - Define a group of algorithms that applies to a family of classes
  - Encapsulate each algorithm separately
  - Make the algorithms interchangeable within that family
- Consequence:
  - The Strategy Pattern lets the specific algorithm implemented by a method vary without affecting the clients that use it.

Strategy pattern: structure



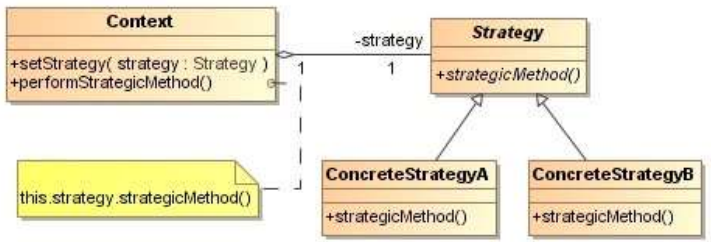
Strategy pattern: elements

Elements of the strategy pattern:

`Context` - class that uses a certain behavior that is to be changed during execution. It contains a `Strategy` object and provides a `setStrategy()` method to change its own strategy dynamically. The strategy is to be called through one or more method that will call methods defined in the Strategy class and implemented differently in the concrete strategy methods.

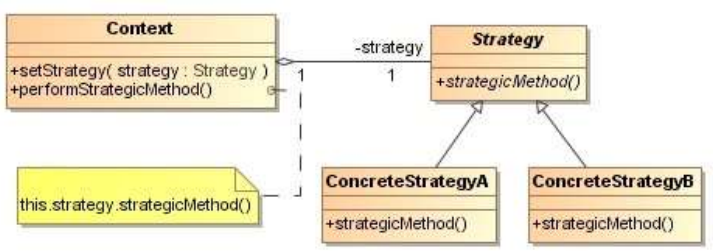
`Strategy` - Superclass of all strategies containing method(s) to be implemented by all its subclasses.

`ConcreteStrategy` - Subclasses of `Strategy` that provide a different implementation for the method(s) provided by the strategy.



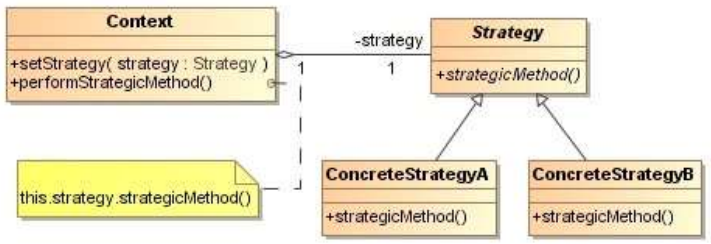
Strategy pattern: example

```
//Context in the Strategy pattern
public class Player {

  public String name;
  public ArrayList<Territory> owned_territories;
  public ArrayList<Order> orders_list;
  PlayerStrategy strategy;

  public void setStrategy(PlayerStrategy p_strat) {
    strategy = p_strat;
  };

  public boolean issueOrder() {
    Order order;
    order = strategy.createOrder();
    if (order != null) {
      orders_list.add(order);
      return true;
    }
    return false;
  }
}
```



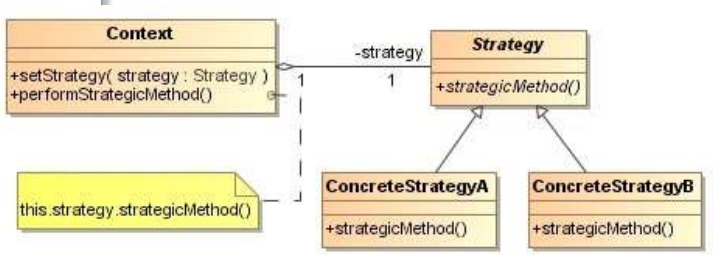
Strategy pattern: example

```
// Strategy of the Strategy pattern
public abstract class PlayerStrategy {
  // The Strategy uses some data to make decisions.
  // Embed them in the Strategy object.
  List<Territory> d_map;
  Player d_player;

  // Pass the required data to the Strategy upon creation
  PlayerStrategy(Player p_player, List<Territory> p_map){
    d_player = p_player;
    d_map = p_map;
  }

  // method that will be called by the Player to do some
  // actions that depend on the adopted ConcreteStrategy.
  public abstract Order createOrder();
  // local methods that the Strategy will use internally
  // to do some actions specifically to the ConcreteStrategy.
  private abstract Territory toAttack();
  private abstract Territory toAttackFrom();
  private abstract Territory toMoveFrom();
  private abstract Territory toDefend();
}
```



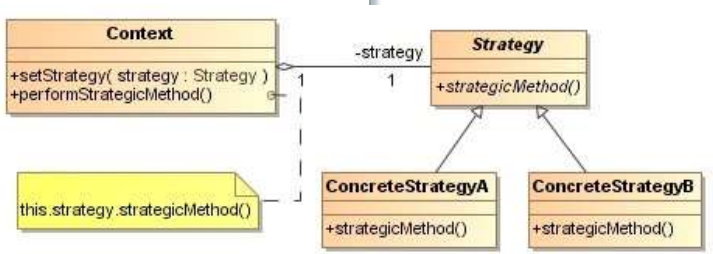
Strategy pattern: example

```
//ConcreteStrategy of the Strategy pattern
public class NeutralPlayerStrategy extends PlayerStrategy {

  public NeutralPlayerStrategy(Player p_player, List<Territory> p_map) {
    super(p_player, p_map);
  }

  protected Territory toAttack() {
    return null;
  }

  protected Territory toDefend() {
    return null;
  }

  protected Territory toAttackFrom() {
    return null;
  }

  protected Territory toMoveFrom() {
    return null;
  }

  // The Neutral player does not issue orders
  public Order createOrder() {
    return null;
  }
}
```

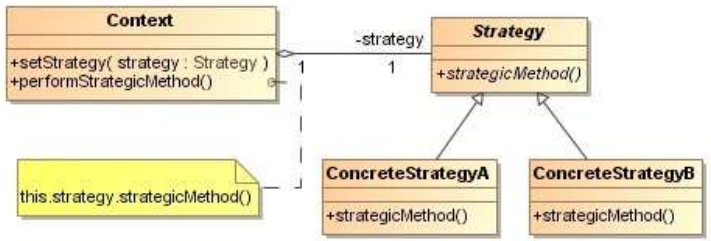


Strategy pattern: example

```
//ConcreteStrategy of the Strategy pattern.
public class DefensivePlayerStratery extends PlayerStrategy {

  //The Defensive player does not attack, so this returns null.
  protected Territory toAttack() {
    return null;
  }

  //The Defensive player decides to defend its country with the most armies.
  protected Territory toDefend() {
    Territory mymaxarmies = d_player.owned_territories.get(0);
    for(Territory terr : d_map) {
      if (mymaxarmies.numArmies < terr.numArmies & d_player.owned_territories.contains(terr))
        mymaxarmies = terr;
    }
    return mymaxarmies;
  }

  //The Defensive player does not attack, so it returns null
  protected Territory toAttackFrom() {
   return null;
  }

  // The Defensive player does not move, so it returns null
  protected Territory toMoveFrom() {
    return null;
  }

  // The Defensive player can only use Deploy orders
  public Order createOrder() {
    Random rand = new Random();
    if (rand.nextInt(5) != 0) {
      return new Deploy(d_player, toDefend(), rand.nextInt(20));
    }
    return null;
  }
}
```



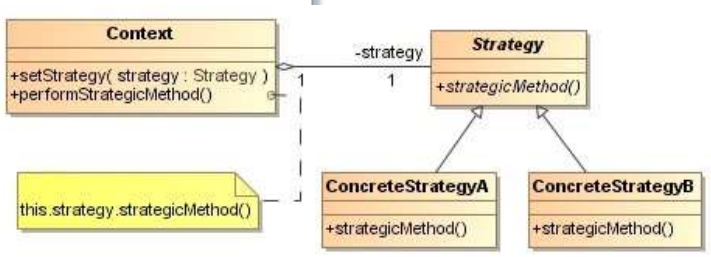
Strategy pattern: example




```
// ConcreteStrategy of the Strategy pattern
public class RandomPlayerStrategy extends PlayerStrategy {
 Random rand = new Random();

  // Random country to move to
  protected Territory toAttack() {
    return(d_map.get(rand.nextInt(d_map.size() - 1));
  }

  // Random of its own countries to defend
  protected Territory toDefend() {
    return(d_player.owned_territories.get(rand.nextInt(d_player.owned_territories.size() - 1)));
  }

  // Random of its own countries to attack from
  protected Territory toAttackFrom() {
    return toDefend();
  }

  // Random of its own countries to move from
  protected Territory toMoveFrom() {
    return toDefend();
  }

  // The Random player can either deploy or advance, determined randomly. .
  public Order createOrder() {
    int rndOrder = rand.nextInt(3);
    int rnd_num_of_armies;
    if (rand.nextInt(5) != 0) {
      switch (rndOrder) {
      case (0):
        return new Deploy(d_player, toDefend(), rand.nextInt(20));
      case (1):
        return new Advance(d_player, toMoveFrom(), toAttack(), rand.nextInt(toMoveFrom().numArmies + 5));
      }
    }
    return null;
  }
}
```

Strategy pattern: example

```
//Client in the Strategy pattern
public class GameEngine {
  List<Territory> map;
  List<Player> players;

  public void start() {
    int numTurns = 5;

    // create the players
    players.add(new Player("player1"));
    players.add(new Player("player2"));
    players.add(new Player("player2"));
    players.get(0).setStrategy(new DefensivePlayerStratery(players.get(0), map));
    players.get(1).setStrategy(new RandomPlayerStrategy(players.get(1), map));
    players.get(2).setStrategy(new NeutralPlayerStrategy(players.get(2), map));

    // run the game turns
    for (int turn = 1; turn <= numTurns; turn++) {
      boolean an_order = true;
      do {
        for (Player p : players) {
          an_order = p.issueOrder();
          if (!an_order)
            break;
        }
      } while (an_order);
      executeAllOrders();
      printMap();
    }
  }
  …
}
```



References

- Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.
- Alexander, Christopher. *The Timeless Way of Building*. Oxford University Press, 1977. ISBN 978-0-19-502402-9.
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Adisson-Wesley, 1995.
- SourceMaking.com. Design Patterns.