Strategy / State patterns

Advanced Java Programming 2012-13

OO Design Patterns and Principles

Lecture Outline

- Strategy pattern
- State pattern

The Strategy Pattern

- The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable
  - What does that mean?
- To illustrate – pretend that you are working on the next big MMORPG
- You want to design an abstract class to use as a base for the various types of characters in the game (*Wizard*, *Knight* and *Thief)*
  - It starts off pretty easy....

The Strategy Pattern

```
class Character {

    private String name;
    private int hitPoints
    private int strength;


    public Character (String name, int hitPoints, int strength) {

        this.name = name;
        this.hitPoints = hitPoints;
        this.strength = strength;
    }


    // Accessor and mutator methods for fields

}
```

Basic fields and constructors

The Strategy Pattern

- But then it gets difficult
  - You want to implement an *attack*() method that is inherited by all sub-classes
  - Inside the *attack*() method, you want to select different forms of attack according to in game conditions
  - Different attacks involve different calculations
    - Attack with bare fists, do damage equal to strength
    - Attack with a dagger, do damage equal to strength+2
    - Attack with sword, do damage equal to strength * 2
    - Attack with magic, do damage equal to HP

So, you start with a SWITCH statement

The Strategy Pattern

```
class Character {

    public int formOfAttack;
    …
    public void attack() {
        switch(formOfAttack) {
        case 0:
            System.out.println(“You do “ + strength + " damage.");
            break;
        case 1:
            System.out.println(“You do “ + (strength + 2) + " damage.");
            break;
        case 2:
            System.out.println(“You do “ + (strength*2) + " damage.");
            break;
        case 3:
            System.out.println(“You do “ + hitPoints + " damage.");
            break;
    }
```

The actual attack used is controlled using this variable

Imagine each print statement is a massive calculation / operation

The Strategy Pattern

```
// In some other class
Character c = new Knight(“Baric”, 78, 12);
c1.setFormOfAttack(2); // sword
c1.attack();

Character c2 = new Thief(“Herin”, 57, 9);
c2.setFormOfAttack(1); // dagger
c2.attack();

Character c3 = new Wizard(“Dumbledore”, 34, 4);
c3.setFormOfAttack(3); // spell
c3.attack();
```

But, there is a problem with this solution

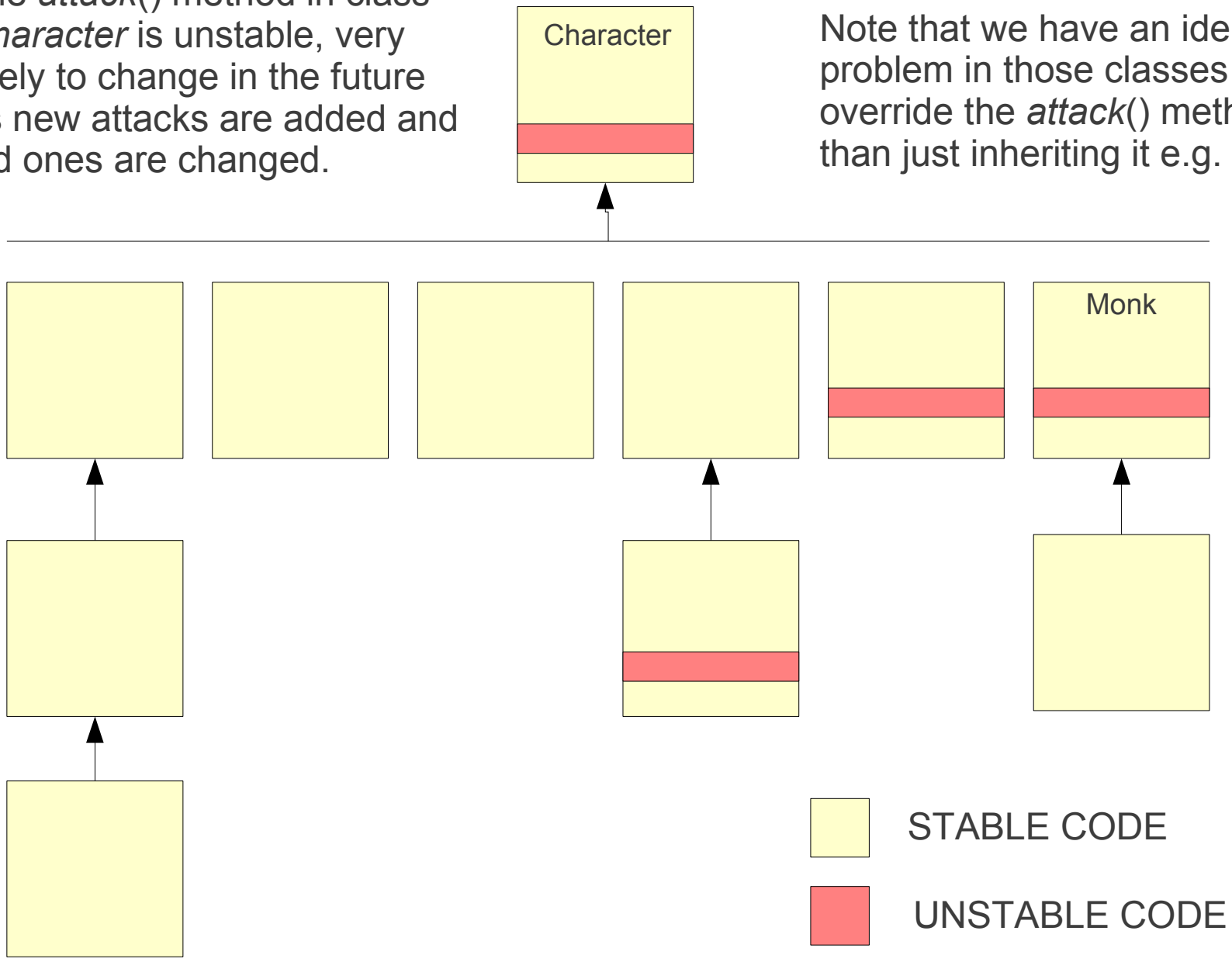
The Strategy Pattern

- As the game is developed, it seems likely that new forms of attack will be introduced e.g. crossbow
- Every time this happens, the *Character* class will have to be edited and the *attack*() method gets bigger / harder to maintain
- A lot of the code in *Character* is stable and unlikely to change
- It would be good if we could separate the volatile parts of the class (i.e. the different ways to attack) from the stable parts
- How do we solve this problem?

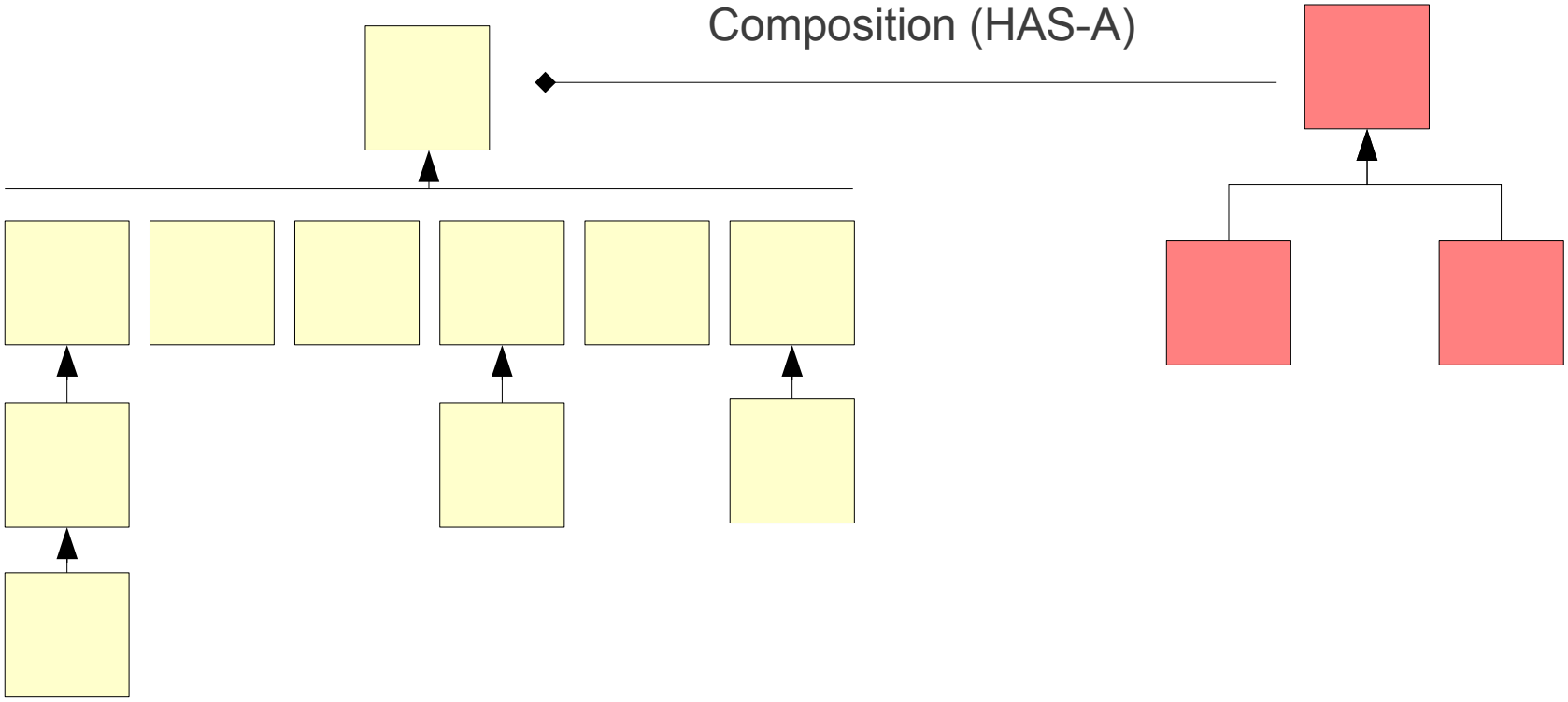
The Strategy Pattern

- The solution to this particular problem is known as the *strategy pattern*
- We have a *family* of algorithms (i.e. the different ways a character can attack)
- We want to *separate* these algorithms (which will change a lot) from the rest of the code (which will not)
- We do this by *encapsulating* each algorithm (fist attack, dagger attack) as a separate class
- Then we use the attacks interchangeably...

The *attack*() method in class *Character* is unstable, very likely to change in the future as new attacks are added and old ones are changed.



Note that we have an identical problem in those classes that override the *attack*() method, rather than just inheriting it e.g. *Monk*



Separate the parts of your code that will change the most from the rest of your application. Put volatile code in the objects your applications contains, rather than inheriting that code

STABLE CODE

UNSTABLE CODE

The Strategy Pattern

- The strategy pattern begins with an interface
  - All of the different algorithms (i.e. all of the different ways to attack) must implement this interface
  - Note that the attack method defined in this interface has two parameters

```
interface AttackAlgorithm
{
   public void attack(int hitPoints, int strength);
}
```

The Strategy Pattern

- Next we create a concrete strategy object, one for each type of attack

```
public class FistAttack implements AttackAlgorithm
{
   @Override
   public void attack(int hitPoints, int strength)
   {
      System.out.println("You do " + strength + "
      damage.");
   }
}
```

Here you can see why the *attack*() method has parameters. Because the *attack*() algorithm has been encapsulated (i.e. moved out of the *Character* class) we need to pass the character attributes to make it work properly

Again, you have to imagine that the *attack*() method is some lengthy algorithm

The Strategy Pattern

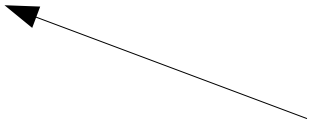
- In the *Character* class, we add a new instance variable, plus a setter method
  - Now we compose a character with a particular attack

```
public abstract class Character
{
   protected AttackAlgorithm a;

   public void setAttack(AttackAlgorithm a)
   {
      this.a = a;
   }
}
```

The Strategy Pattern

- Finally, in the *Character* class, we delegate any calls to the attack() method to the encapsulated algorithm

```
public abstract class Character
{
   protected AttackAlgorithm a;

   public void attack()
   {
      a.attack(hitPoints, attack);
   }
}
```

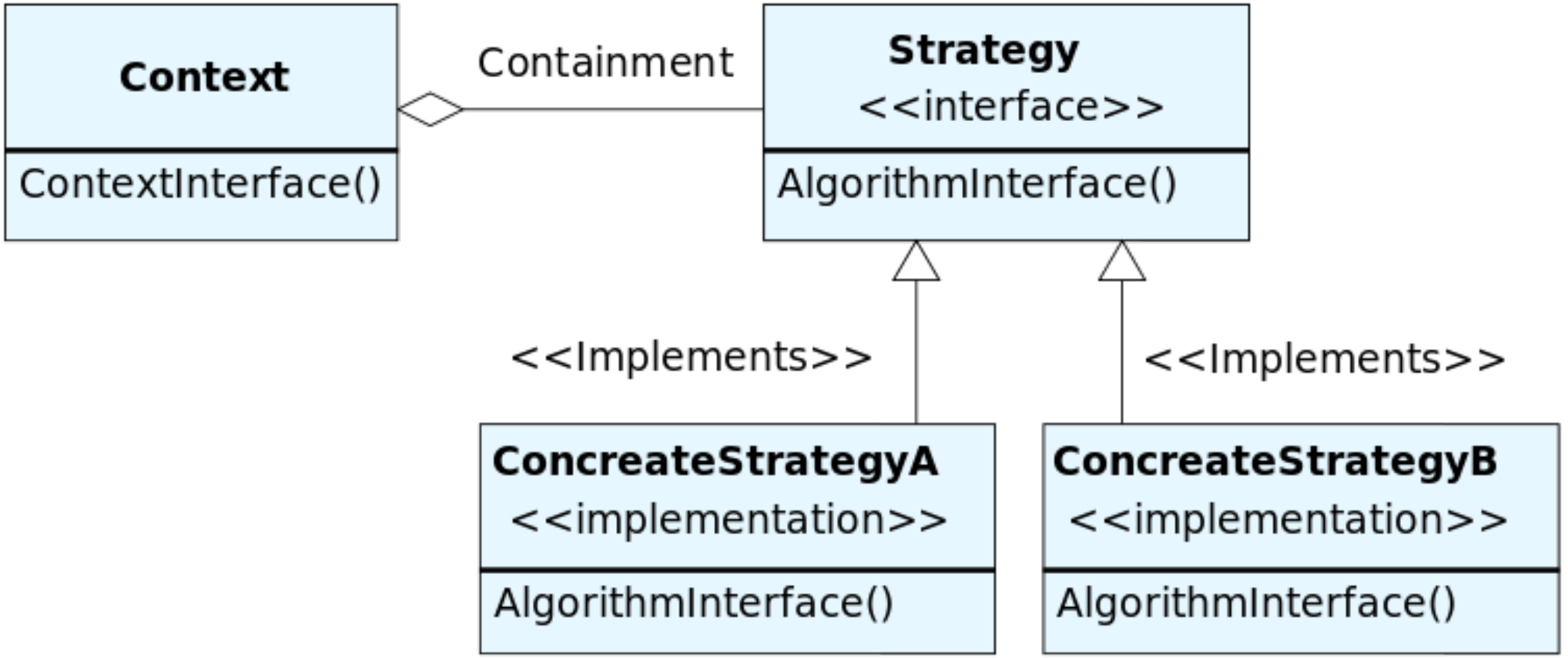


All method calls are delegated to the encapsulated algorithm, which can vary independently from the client (the *Character* class)

The Strategy Pattern

```
// In some other class
Character c = new Knight("Baric", 78, 12);
c1.setFormOfAttack(new SwordAttack());
c1.attack();

Character c2 = new Thief("Herin", 57, 9);
c2.setFormOfAttack(new DaggerAttack());
c2.attack();

Character c3 = new Wizard("Dumbledore", 34, 4);
c3.setFormOfAttack(new MagicAttack());
c3.attack();
```

The Strategy Pattern



Context = *Character*, Strategy = *AttackAlgorithm*

The Strategy Pattern

- Advantages of this approach
  - Reduction in selection statements (if/then, switch) which can be hard to maintain
  - The attack algorithms can be *altered* centrally without varying the client (i.e. the *Character* classes and sub-classes are not touched)
  - New attack algorithms can be *added* without modifying any of the character classes
- Disadvantage
  - Extra object instantiation


The State Pattern

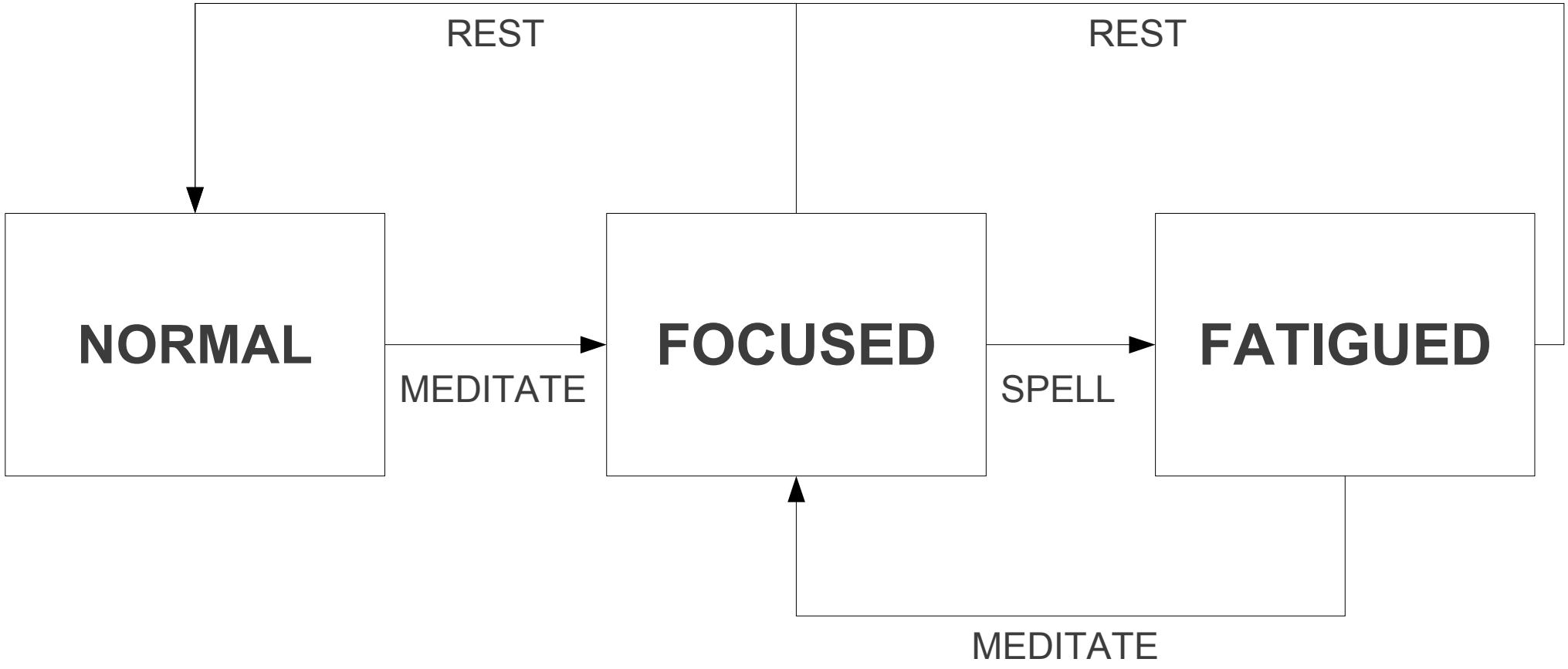
The State pattern

- The *State* pattern allows an object to change its behaviour based on its internal state
  - What does that mean?
- Returning to our MMORPG, let us pretend that a wizard can be in one of 3 different emotional / physical states
  - Rested
  - Focused
  - Fatigued
- Each state has implications for the character's behaviour

The State pattern

- When a wizard is *normal*
  - He/she is able to fight
  - He/she is unable to cast a spell
- When a wizard is *focused*
  - He/she is unable to fight
  - He/she is able to cast a spell
    - If he/she does cast a spell, he/she will become *fatigued*
- When a wizard is *fatigued*
  - He/she is able to fight, badly
  - He/she is unable to cast a spell
- At any time
  - He/she can meditate to become focused
  - He/she can rest to become normal

The State pattern



STATE TRANSITIONS FOR A WIZARD

The State pattern

- We want to convert this into code
- Let's try a simple implementation using an *enum* to define the 3 possible states a wizard can be in
  - Then, each state dependent method of the *Wizard* class (e.g. cast()) will switch on the value of the *enum* to produce the correct behaviour

```
public enum WState {
   NORMAL, FOCUSED, FATIGUED
}
```

The State pattern

```
class Wizard {
    private String name;
    private int hitPoints;
    private int strength;
    private WState ws;

    public Wizard (String name, int hitPoints, int strength ) {
        this.name = name;
        this.hitPoints = hitPoints;
        this.strength = strength;
        this.ws = WState.NORMAL;
    }

    public void setState (WState ws) {
        this.ws = ws;
    }
    ...
}
```

The State pattern

```
…
public void sleep() {

    System.out.println(“You sleep 6 hours and awake refreshed”);
    setState(WState.NORMAL);
}


public void meditate() {

    System.out.println(“You chant some stuff and make omm noises”);
    setState(WState.FOCUSED);
}
```

Non-volatile methods of the *Wizard* class (the same regardless of state)

The State pattern

```
…
public void fight() {

    switch(ws) {
    case NORMAL :
        System.out.println(“You swing with the force of 10 bears”);
        break;

    case FOCUSSED :
        System.out.println(“You are still in a trance!”);
        break;

    case FATIGUED:
        System.out.println(“You make a feeble lunge”):
        break;

    default:
        System.err.println(“Check enum”);
}
```

attack(), a state dependent method of the *Wizard* class

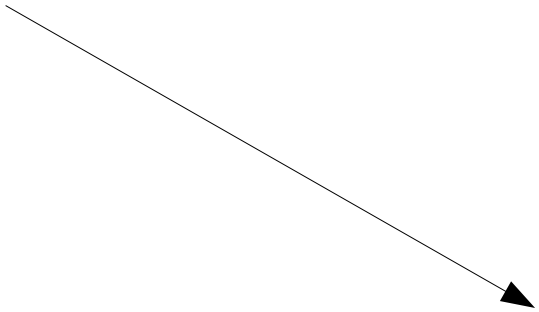
The State pattern

```
…
public void cast() {

    switch(ws) {
    case NORMAL :
        System.out.println(“You realise you have to meditate”);
        break;

    case FOCUSSED :
        System.out.println(“You unleash magical terror and imps”);
        setState(WState.FATIGUED);
        break;

    case FATIGUED:
        System.out.println(“You realise you are exhausted.”);
        break;

    default:  System.err.println(“Check enum”);
}
```

cast(), a state dependent method of the *Wizard* class

The State pattern

```
Wizard w = new Wizard(“Enwor”, 45, 6);
w.fight();
w.fight();
w.cast();
w.meditate();
w.cast();
```



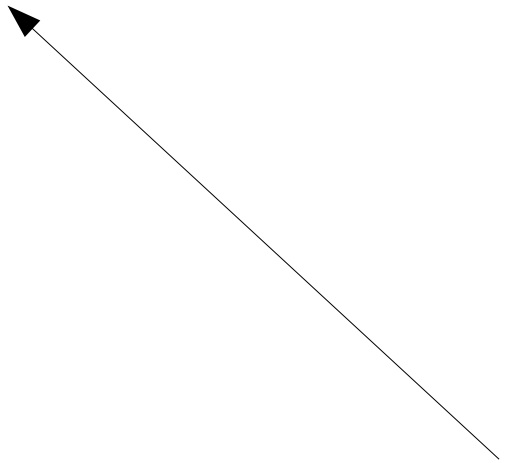
```
You swing with the force of 10 bears
You swing with the force of 10 bears
You realise you have to meditate
You chant some stuff and make omm noises
You unleash magical terror and imps
```

The State pattern

- Problems with this solution
  - Too much selection (switch, if/then), which is hard to maintain
  - Stable code is mixed in with volatile code
  - The stable code (meditate(), sleep()) applies equally to all object states, and is not change much during lifespan of class
  - The methods *fight*() and *cast*() are volatile because we are likely to extend the number of states as the game develops
- The solution – the state pattern!

The State pattern

```
interface WizardState {

    void fight();

    void cast();

}
```



We start with an interface. All concrete *Wizard* states must implement this interface

The State pattern

```
class NormalState implements WizardState {


    void fight() {

        System.out.println(“You swing with the force of 10 bears”);

    }

    void cast() {

        System.out.println(“You realise you have to meditate”);
    }


}
```

A concrete implementation of *WizardState*

The State pattern

```
class FocusedState implements WizardState {


    void fight() {

        System.out.println(“You are still in a trance!”);

    }

    void cast() {

        System.out.println(“You unleash magical terror and imps”);
    }


}
```

A concrete implementation of *WizardState*

The State pattern

```
class FatiguedState implements WizardState {


    void fight() {

         System.out.println(“You make a feeble lunge”):

    }

    void cast() {

        System.out.println(“You realise you are exhausted.”);

    }


}
```

A concrete implementation of *WizardState*

The State pattern

```
class Wizard {
    private String name;
    private int hitPoints;
    private int strength;
    private WizardState ws;

    public Wizard (String name, int hitPoints, int strength ) {
        this.name = name;
        this.hitPoints = hitPoints;
        this.strength = strength;
        this.ws = new NormalState();
    }

    public void setState (WizardState ws) {
        this.ws = ws;
    }
    ...
}
```

Now we compose the *Wizard* class with a reference to the *WizardState* interface

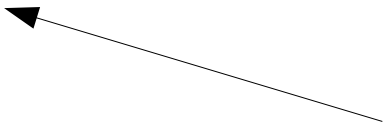
The State pattern

```
… // inside Wizard....

public void fight() {

   ws.fight();
}

public void cast() {

   ws.cast();
}
```

And we delegate all calls to the state-dependent methods like so....

The State pattern

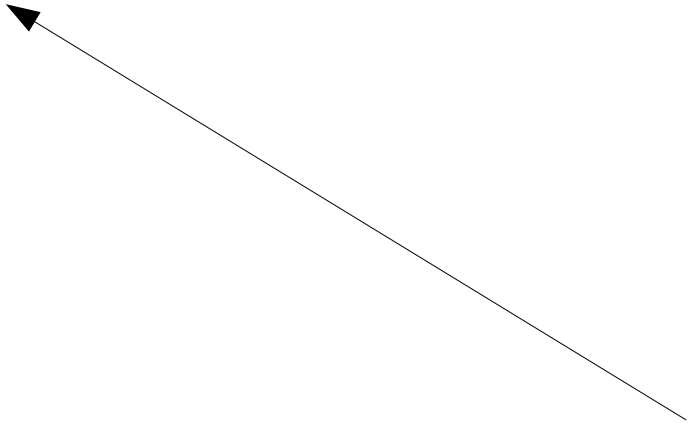
```
…
public void sleep() {

System.out.println(“You sleep 6 hours and awake refreshed”);
setState(new NormalState());

}


public void meditate() {

System.out.println(“You chant some stuff and make omm noises”);
setState(new FocusedState());

}
```



Changing state from within the context class (*Wizard*) is simple..

The State pattern

```
interface GameState {

   void fight(Wizard w);

   void cast(Wizard w);

}
```

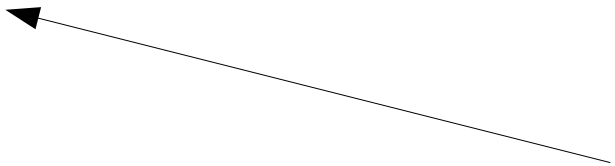


First, we change the interface

Changing state from *within another state* some more work....

The State pattern

```
class Wizard {

    ....

    public void fight() {

        ws.fight(this);
    }

    public void cast() {

        ws.cast(this);
    }

}
```

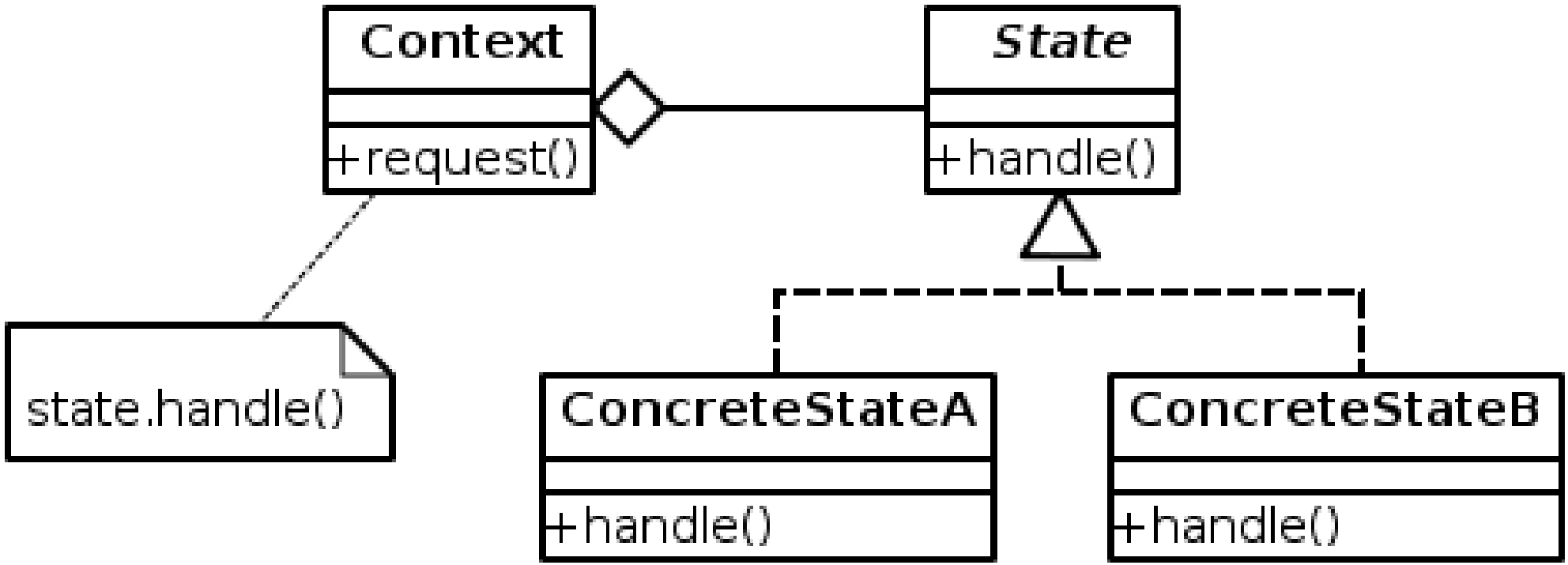


Then we change the *Wizard* class. The keyword *this* refers to the current object. This line of code invokes the *attack()* method of the encapsulated state object, and passes it a reference to the current *Wizard* object

The State pattern

```
class FocusedState implements WizardState {


    void fight(Wizard w) {

        System.out.println("You are still in a trance!");

    }

    void cast(Wizard w) {

        System.out.println("You unleash magical terror and imps");
        ws.setState(new FatiguedState());
    }


}
```

The result – we can change the state of the linked *Wizard* object from within another state!



SEEM FAMILIAR??

Context is *Wizard*, State is *WizardState*, request is *fight*() or *cast*()

The State pattern

- Advantages of this pattern
  - Reduced selection structures (if/then, switch)
  - States can vary independently of the context class
  - You can add states without disturbing the context class
- Disadvantages
  - Object explosion

Summary

- **The Strategy pattern**
  - Sometimes we have a class uses a family of algorithms i.e. different ways of doing the same thing e.g. *attack*()
  - Rather than using selection statements (switch, if/then) to pick the correct algorithm inside the class, we should encapsulate the algorithms and allow them to vary independently of the class
- **The State pattern**
  - Sometimes we have a class whose ***overall behaviour*** changes as it passes through several well known states
  - Rather than using selection statements (switch, if/then) to implement these states within the class, we should encapsulate the states and allow them to vary independently of the class
- **Pre-reading**
  - *Decorator* and *Adapter* pattern in HFDP and DPFD