

Lecture 4: Design Patterns

EC3375: Software Design Pattern and
Technology

Overview

- Fundamentals of OO
- Design Patterns and Design Principles
- Strategy Pattern
- Factory Method
- Abstract Factory
- Decorator

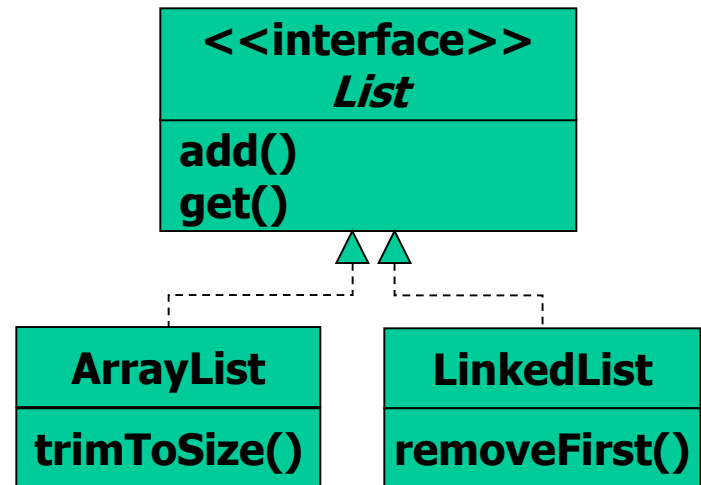
An Example

- Leporello wants to keep lists of his master's girlfriends in various countries.
- He has decided to implement these using an ArrayList (for the Italians) and a LinkedList (for the Spaniards). Should he declare types like this...

ArrayList italians;
LinkedList spaniards;

Or like this...

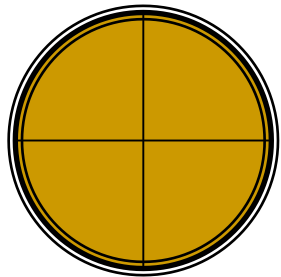
List italians;
List spaniards;



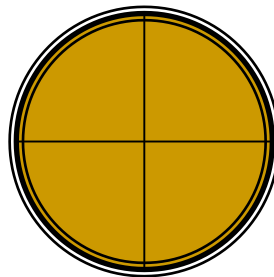
Treating the variables as Lists rather than exposing their concrete types makes the code simpler, and easier to CHANGE!

Fundamentals of OO

Abstraction
Polymorphism
Inheritance
Encapsulation



bject



rientation

Fundamentals of OO

- Abstraction. Ability to access objects in ways that obscure details that we do not want to be dependent on, or which would unnecessarily complicate things.
- Polymorphism. Different classes can implement the same methods in different ways.
- Inheritance. Classes inherit methods and properties from their superclasses.
- Encapsulation. Classes bundle together methods and properties and control access to those properties (e.g. through private, public, protected modifiers in Java).

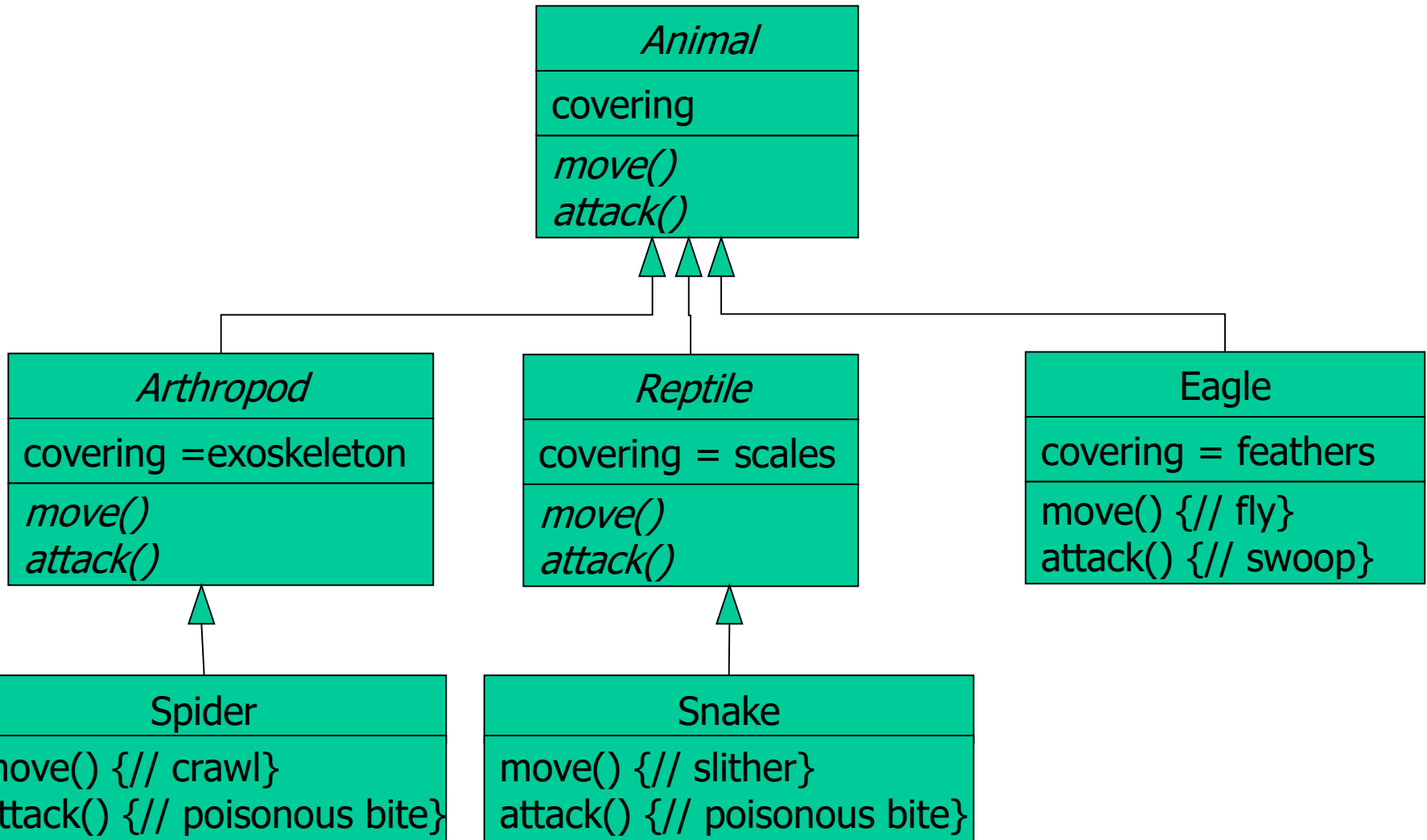
Is that all there is?

- If we understand abstraction, polymorphism, inheritance, and encapsulation, do we know all there is to know about Object Orientation?
- Let's look at an example.

The Game of Survival

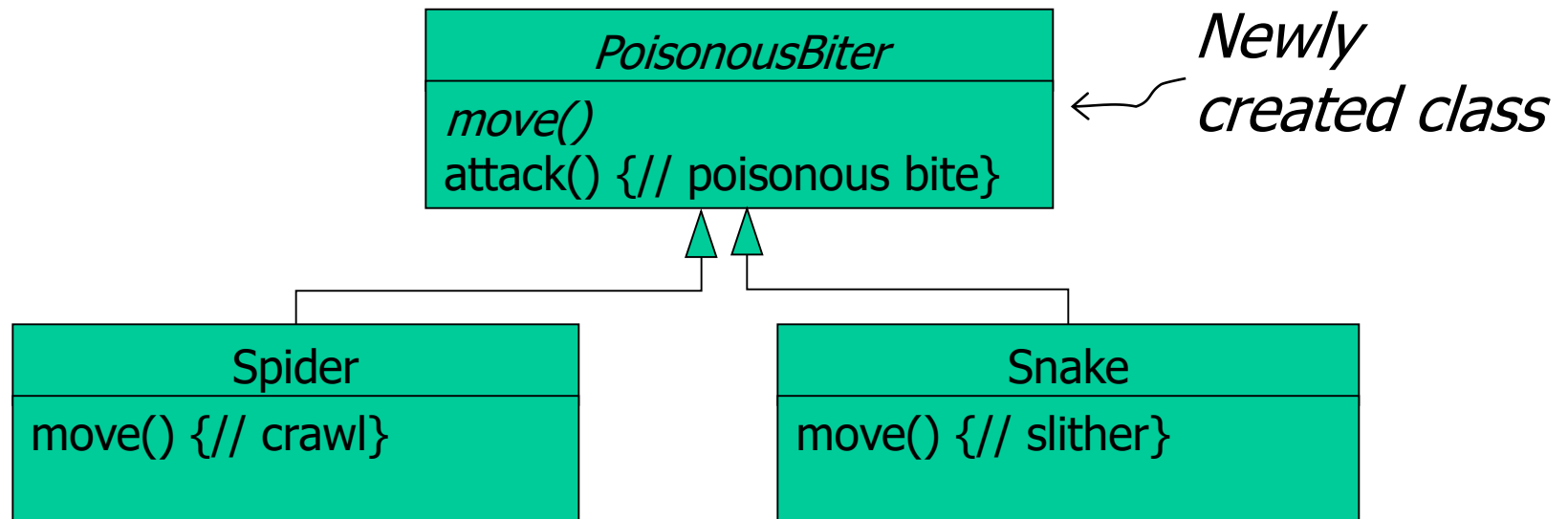
- Survival is a multi-player game in which the playable characters are animals inhabiting some virtual environment. Each animal implements methods allowing it to
 - Move
 - Attack
 - Give birth (if female)
 - *etc.*

Animal Classes



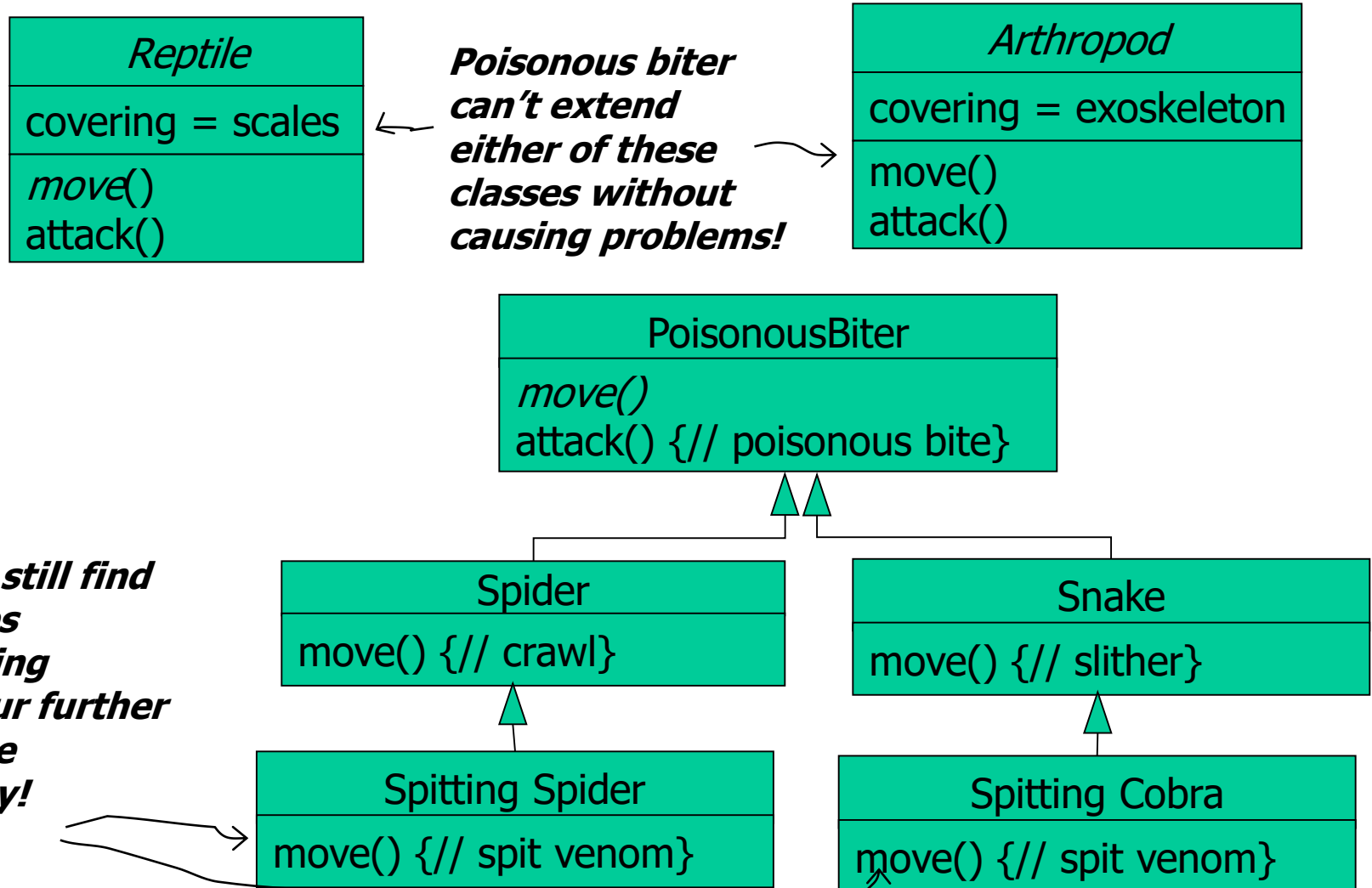
Implementing Poisonous Bites

- How do we represent the fact that both snakes and spiders attack using a poisonous bite. Here's one possible solution...



Implementing Poisonous Bites:

First attempt



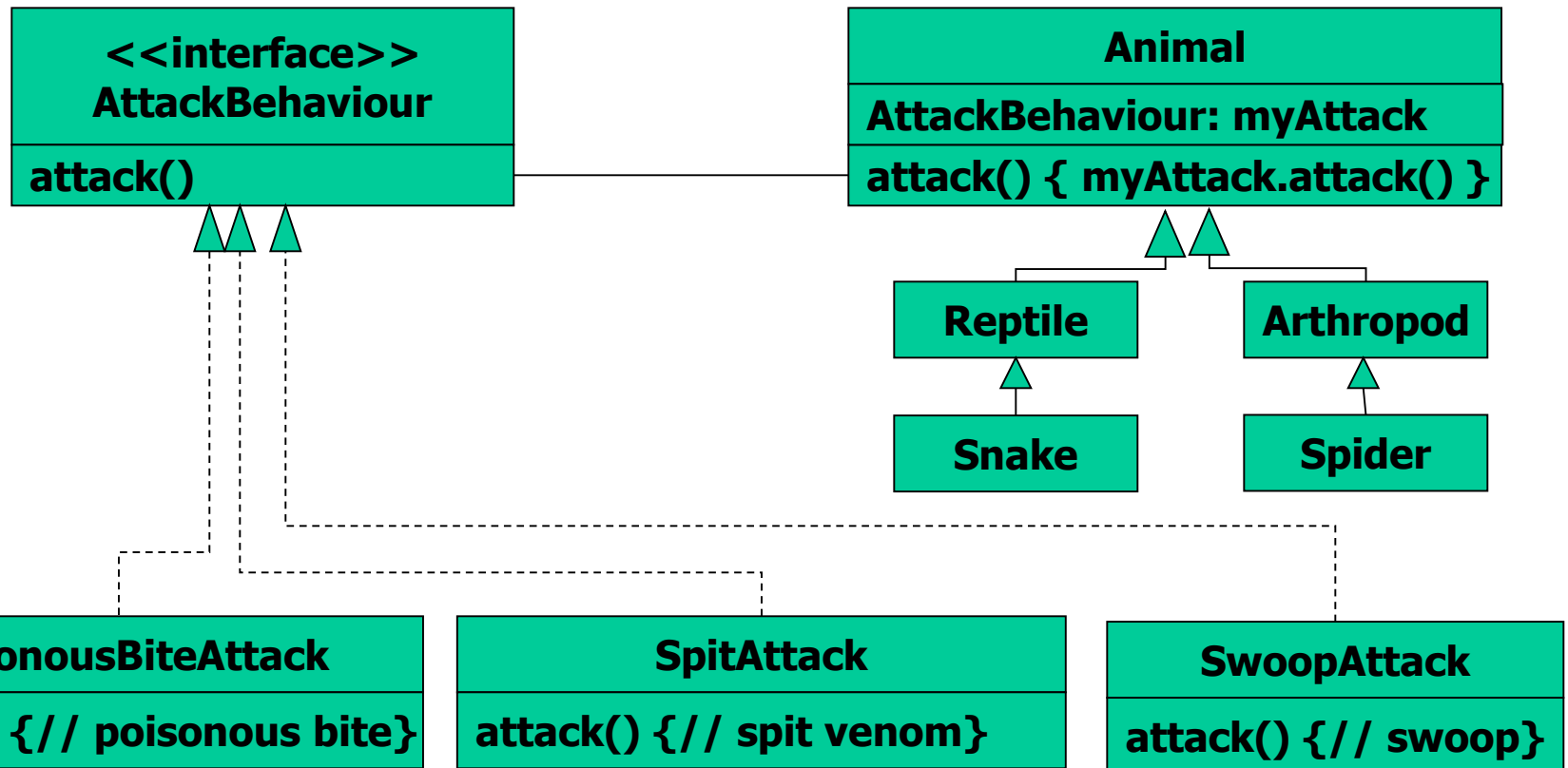
How can we improve on this?

- The example illustrates a problem that often crops up in OO design.
 - We want classes to behave in different ways, but...
 - We can't easily implement this using inheritance

A Better Solution

Encapsulate the behaviour.

- If behaviour varies between classes then it may make sense to encapsulate that behaviour into a separate class.



Animal Classes

```
public abstract class Animal {  
    String name;  
    String description;  
    String covering;  
    AttackBehaviour myAttack;  
  
    public Animal (String name) {  
        this.name = name;  
    }  
    abstract void move();  
    void performAttack() {myAttack.attack();}  
}
```

*Reference to separate
object that performs
the attack*

```
public class Eagle extends Animal {  
    public Eagle(String name) {  
        super(name);  
        covering = "feathers";  
        description = "a raptor";  
        myAttack= new SwoopAttack();  
    }  
    void move() {  
        System.out.println("fly");  
    }  
}
```

Snakes and Other Reptiles

```
public abstract class Reptile extends Animal {  
  
    public Reptile(String name) {  
        super(name);  
        covering = "scales";  
    }  
}
```

```
public class Snake extends Reptile{  
  
    public Snake(String name) {  
        super(name);  
        myAttack = new PoisonBiteAttack();  
        description = "a slithering reptile";  
    }  
  
    public void move() {  
        System.out.println("slither");  
    }  
}
```

AttackBehaviour

```
public interface AttackBehaviour {  
    void attack();  
}
```

```
public class PoisonBiteAttack implements AttackBehaviour {  
  
    public void attack() {  
        //in reality we would put something more complicated here!  
        System.out.println("I just bit you - you're poisoned!");  
    }  
}
```

```
public class SwoopAttack implements AttackBehaviour {  
  
    public void attack() {  
        System.out.println("I swooped down and grabbed you!");  
    }  
}
```


Testing the Classes

```
public class AttackDemo {
```

```
    public static void main(String[] args) {  
        Animal sidney = new Snake("Sidney");  
        Animal eddie = new Eagle("Eddie");  
        doAttack(sidney);  
        doAttack(eddie);  
    }
```

*In a real application we
would use getters and
setters here!*

```
    public static void doAttack(Animal animal) {  
        System.out.print("Hello I'm " + animal.name);  
        System.out.print(" I'm " + animal.description);  
        System.out.println(" I'm covered in " + animal.covering + " and...");  
        animal.performAttack();  
    }  
}
```

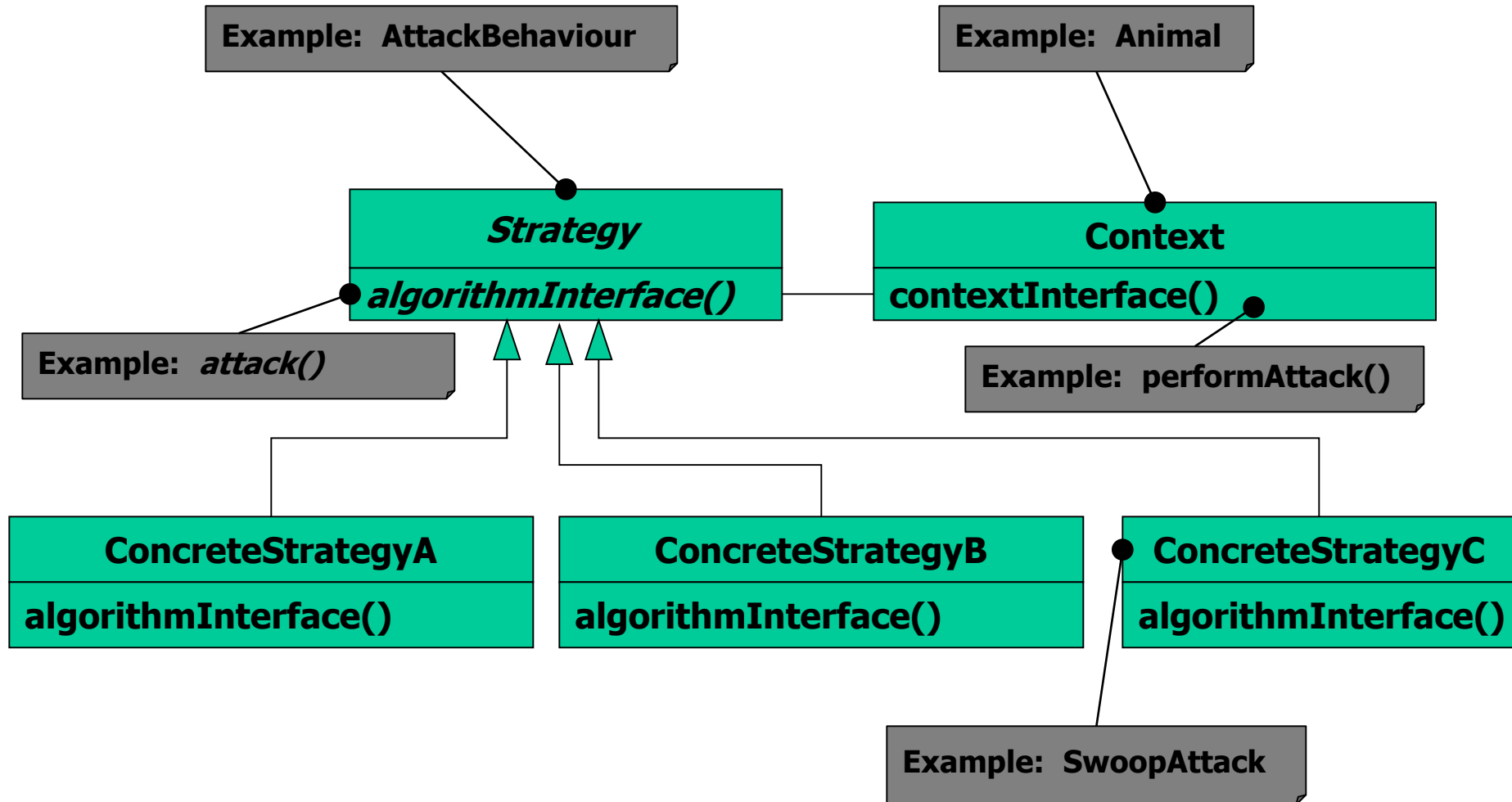


```
Hello I'm Sidney I'm a slithering reptile I'm covered in scales and...  
I just bit you - you're poisoned!  
Hello I'm Eddie I'm a raptor I'm covered in feathers and...  
I swooped down and grabbed you!
```


A pattern emerges...

- The example illustrates a *design pattern*.
- This pattern has a name (names are very important!). It's called the *strategy pattern*.
- Note that
 - The pattern addresses a problem that crops up reasonably frequently, and
 - The pattern has been used more than once in the past, and
 - It sometimes solves the problem (if it didn't we'd call it an *anti-pattern*).

Structure of the Strategy Pattern.



Design Principles

- The examples we have seen also illustrate a number of design *principles*:
 - Favour composition over inheritance.
 - Encapsulate what varies.
 - Program to an interface not an implementation.

Small print: these are the way these particular principles are normally expressed, however some of the terms are used in slightly unfamiliar senses. “Composition” is used to mean what UML describes as “aggregation”, and the term “interface” is not limited to java interfaces – for instance the methods of a superclass can be treated as an “interface” of its subclasses.

Also the “principles” are not hard and fast rules – they are best treated as guidelines or suggestions.

How do we describe a pattern?

- To describe a pattern you must (at least):
 - Give it a name.
 - Describe the problem it solves.
 - Explain how it solves that problem.
 - Describe the advantages and disadvantages of the solution.
- Various pattern catalogues exist, the most well known is that created by the “gang of four” (Gamma, Helm, Johnson, Vlissides, “Design Patterns”, Addison Wesley 1995).

Gang of Four Patterns

- **GoF pattern descriptions have the following template:**
 - **Name:** *allows pattern to become part of design vocabulary.*
 - **Classification:** *what kind of pattern it is. See next slide.*
 - **Intent:** *short description outlining what the pattern does and what problem it addresses.*
 - **Motivation:** *concrete example of the use of the pattern.*
 - **Applicability:** *when should you use the pattern.*
 - **Structure:** *diagram introducing the classes that participate in the pattern and illustrating the relationships between them.*
 - **Participants:** *describes the classes that participate in above structure.*
 - **Collaborations:** *describes how the patterns work together.*
 - **Consequences:** *good and bad effects of the pattern.*
 - **Implementation:** *techniques and issues you should be aware of when implementing the pattern.*
 - **Sample code:** *code fragments to illustrate how you might implement that pattern.*
 - **Known uses:** *a pattern should have at least two or three of these.*
 - **Related patterns.**

Classifying Patterns

- Patterns can be classified as
 - **Behavioural:** *deal with the way in which objects interact and distribute responsibilities.*
 - **Creational:** *concern the process of creating objects.*
 - **Structural:** *deal with composition of objects into larger systems.*

Gang of Four Patterns

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

Patterns in **red** are covered in this lecture.

A Creational Pattern

Factory method

- Imagine that we have created a first version of the “survival” game in which animals inhabit an environment consisting of patches of jungle divided by rivers.
- Now we want to create a “polar” version where the animals inhabit patches of snow divided by crevasses.

Jungle Survival



Jungle

River Jungle

Polar Survival

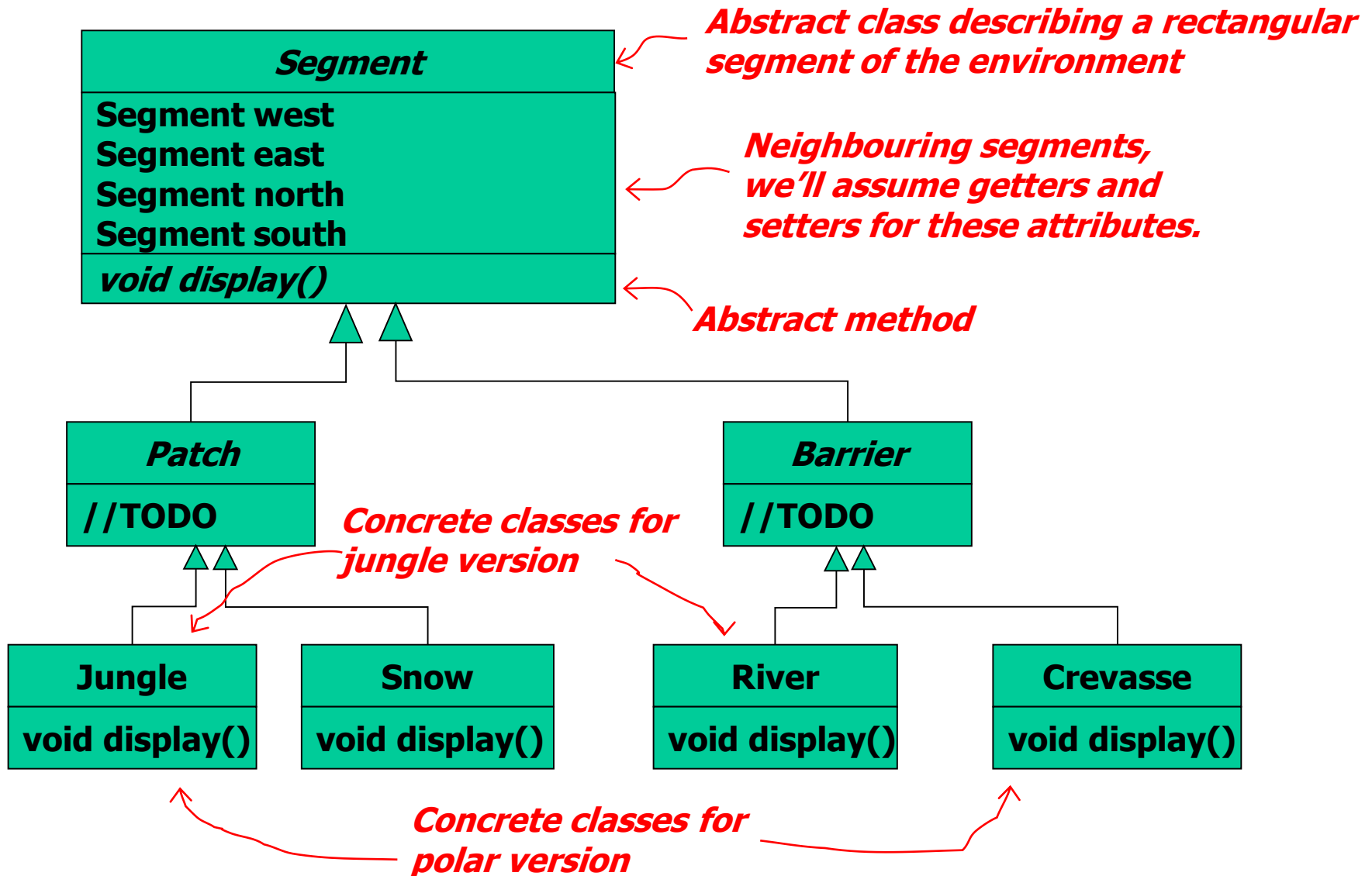


Snow

Crevasse

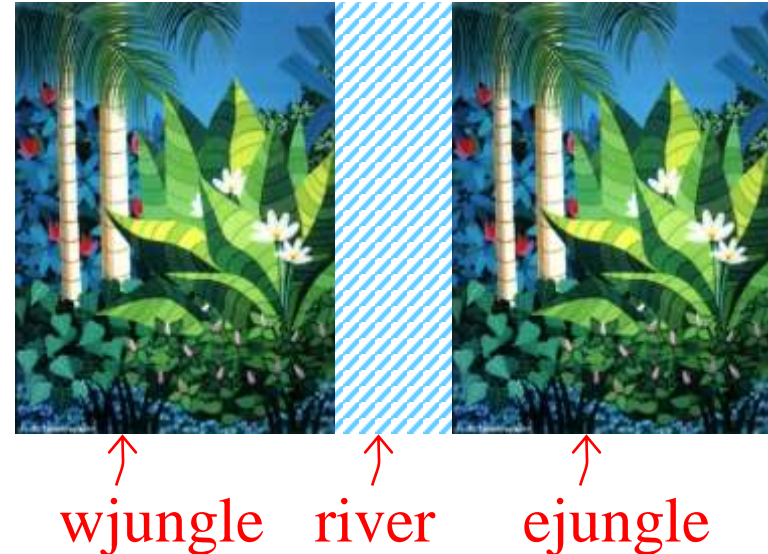
Snow

Classes for Survival Environments



Setting up the Jungle Game

```
class JungleSurvival {  
    Segment currentPosition;  
  
    void setUpEnvironment() {  
        Patch wjungle = new Jungle();  
        Barrier river = new River();  
        Patch ejungle = new Jungle();  
  
        wjungle.setEast(river);  
        river.setWest(wjungle);  
        river.setEast(ejungle);  
        ejungle.setWest(river);  
        currentPosition = wjungle;  
    }  
    JungleSurvival() {  
        setUpEnvironment();  
    }  
}
```



Setting up the Polar Version

Bad idea number 1

- Copy the jungle version and change code as appropriate
- Gives us two separate games to maintain!



```
//class JungleSurvival {  
class PolarSurvival {  
  
    setUpEnvironment() {  
        //Patch wjungle = new Jungle();  
        Patch wsnow = new Snow();  
        //Barrier river = new River();  
        Barrier crevasse = new Crevass();  
        //Patch ejungle = new Jungle();  
        Patch esnow = new Snow()  
  
        //wjungle.setEast(river);  
        wsnow.setEast(crevasse);  
        etc.  
    }  
}
```

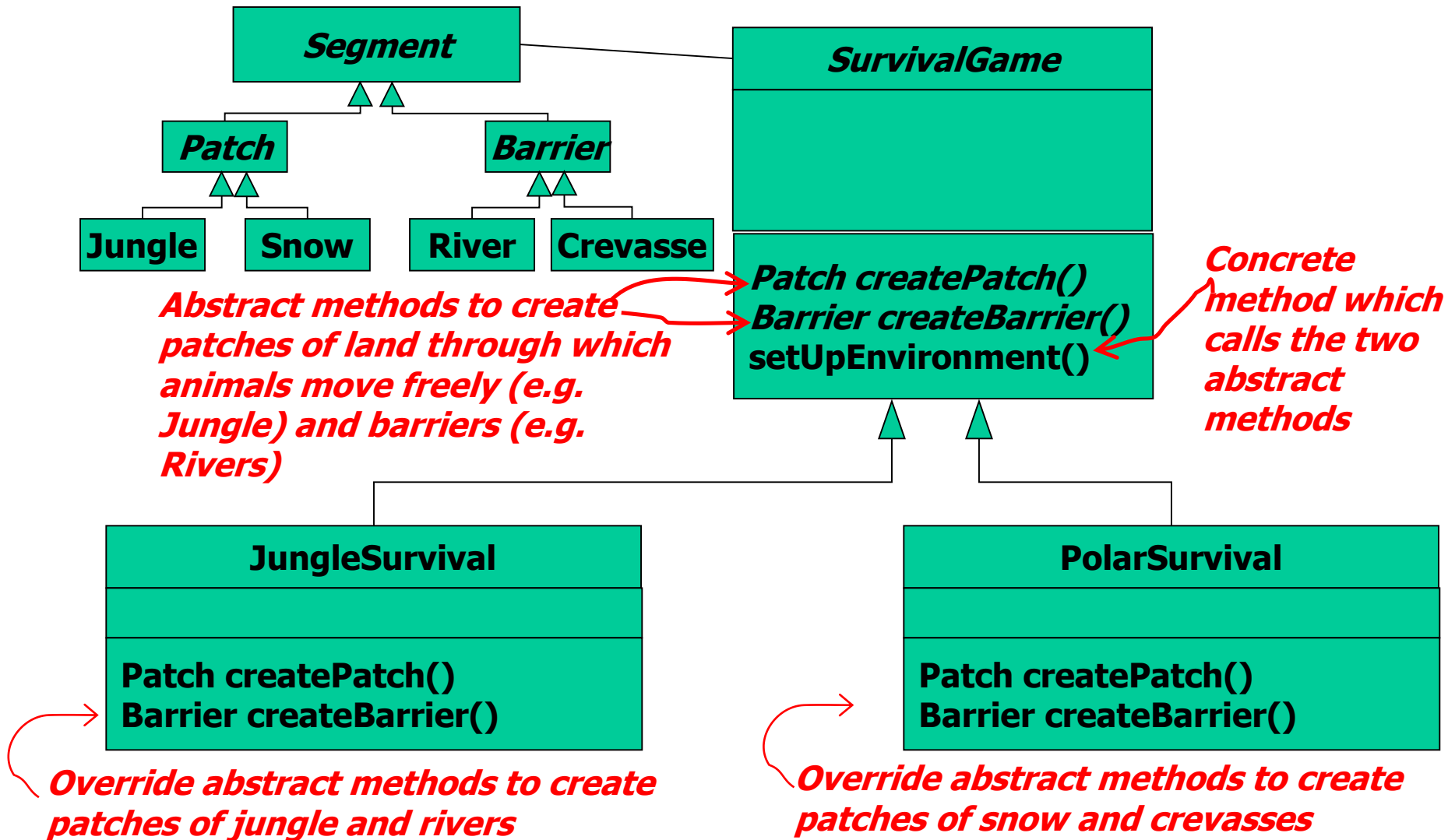
Setting up the Polar Version

Bad idea number 2

- Introduce if statements that allow us to switch between the two versions.
- This class is complicated and we will have to *modify* it if we want to add another type of game (e.g. “savannah survival”)

```
class JungleOrPolarSurvival {  
  
    String type;  
  
    setUpEnvironment {  
  
        Patch westPatch;  
        if (type == "jungle")  
            westPatch = new Jungle();  
        else if (type == "polar")  
            westPatch = new Snow();  
  
        Barrier barrier;  
        if (type == "jungle")  
            barrier = new River();  
        else if (type == "polar")  
            barrier = new Crevasse();  
  
        etc.  
  
    }
```

A better idea



The SurvivalGame Class

```
abstract class SurvivalGame {  
  
    Segment currentPosition;  
  
    setUpEnvironment() {  
        Patch westPatch = createPatch();  
        Barrier barrier = createBarrier();  
        Patch eastPatch = createPatch();  
  
        westPatch.setEast(barrier);  
        barrier.setWest(westPatch);  
        barrier.setEast(eastPatch);  
        eastPatch.setWest(barrier);  
  
        currentPosition = barrier;  
    }  
  
    abstract Patch createPatch();  
    abstract Barrier createBarrier();  
}
```

JungleSurvival and PolarSurvival

```
class JungleSurvival extends SurvivalGame{  
    Patch createPatch() {return new Jungle();}  
    Barrier createBarrier() {return new River();}  
}
```

```
class PolarSurvival extends SurvivalGame{  
    Patch createPatch() {return new Snow();}  
    Barrier createBarrier() {return new Crevasse();}  
}
```

The Segment Class

```
abstract class Segment {  
    Segment west, east, north, south; //neighbouring segments  
  
    Segment() {west = null; east = null; north=null; south=null;}  
  
    Segment getWest() {return west;}  
    void setWest(Segment w) {west=w;}  
  
    Segment getEast() {return east;}  
    void setEast(Segment e) {east=e;}  
  
    Segment getNorth() {return north;}  
    void setNorth(Segment n) {north=n;}  
  
    Segment getSouth() {return south;}  
    void setSouth(Segment s) {south = s;}  
  
    abstract void display();  
}
```


Patches and Barriers

```
abstract class Patch extends Segment{
    //we'll put something in here later
}
abstract class Barrier extends Segment {
    //we'll put something in here later
}

//for the time being overrides of the display method just print something out
class Jungle extends Patch {
    void display() {System.out.print("a patch of jungle");}
}

class River extends Barrier {
    void display() {System.out.print("a river");}
}

class Snow extends Patch {
    void display() {System.out.print("a patch of snow");}
}

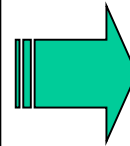
class Crevasse extends Barrier {
    void display() {System.out.print("a crevasse");}
}
```

Testing the Classes

```
class TestGames {  
  
    public static void main(String [] args) {  
  
        SurvivalGame jungleGame = new JungleSurvival();  
        System.out.println("JUNGLE");  
        describe(jungleGame.currentPosition);  
  
        SurvivalGame polarGame = new PolarSurvival();  
        System.out.println("\nPOLAR");  
        describe(polarGame.currentPosition);  
    }  
}
```

TestGame (continued)

```
static void describe(Segment seg) {  
  
    System.out.print("You are in ");  
    seg.display();  
    System.out.print(". To the west is ");  
    describeNeighbour(seg.getWest());  
    System.out.print(". To the east is ");  
    describeNeighbour(seg.getEast());  
    System.out.print(". To the north is ");  
    describeNeighbour(seg.getNorth());  
    System.out.print(". To the south is ");  
    describeNeighbour(seg.getSouth());  
}  
  
static void describeNeighbour(Segment seg)  
{  
    if (seg==null) System.out.print("nothing");  
    else seg.display();  
}  
}
```



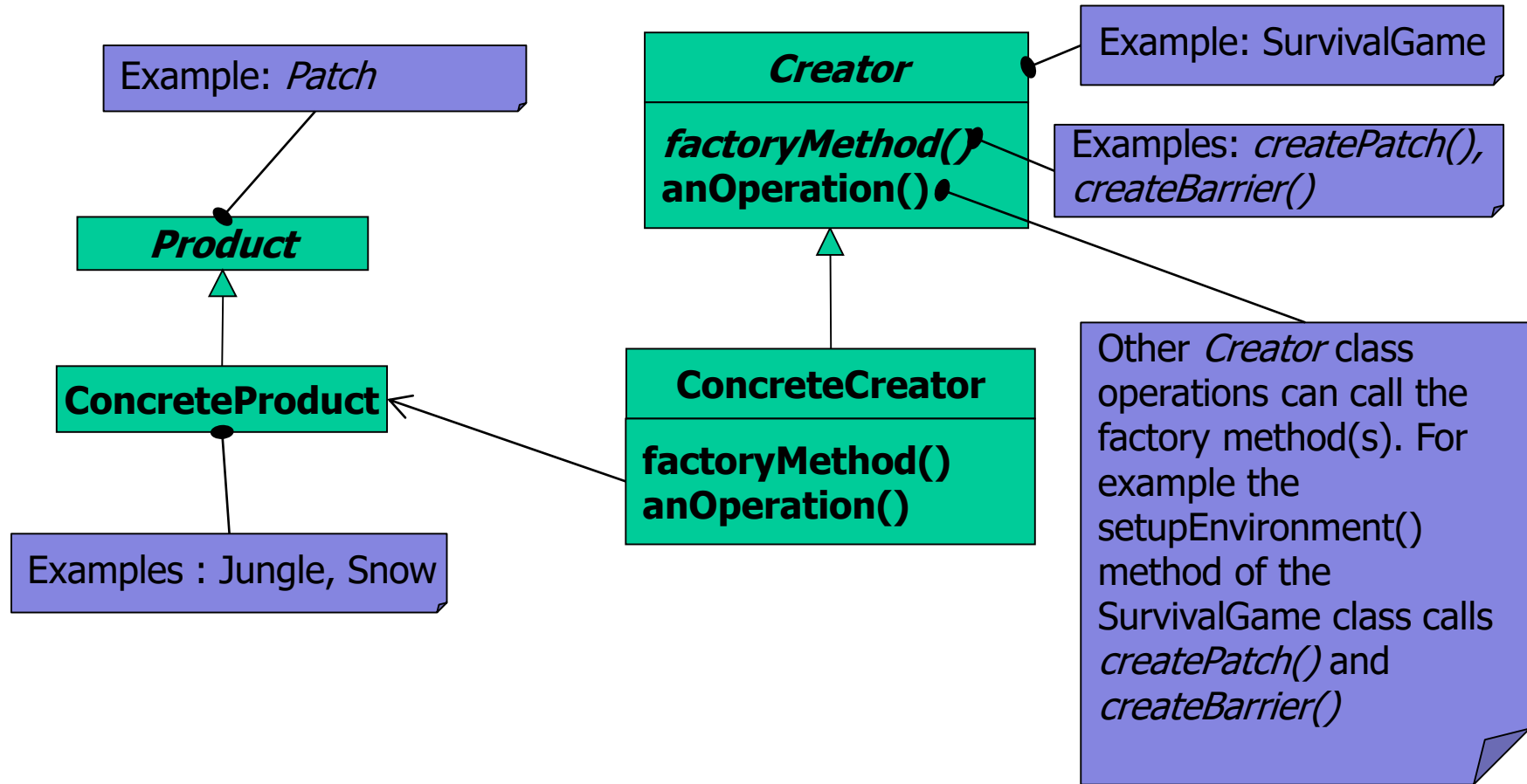
JUNGLE

You are in a river. To the west is a patch of jungle. To the east is a patch of jungle. To the north is nothing. To the south is nothing

POLAR

You are in a crevasse. To the west is a patch of snow. To the east is a patch of snow. To the north is nothing. To the south is nothing

The Factory Method Pattern



The Factory Method Pattern

- Disadvantages
 - Added complexity
- Advantages
 - Concrete product classes hidden from Creator class.
 - System can be extended by adding new classes, without modifying existing ones.

Design Principles

The Open-Closed Principle

Classes should be open to extension but closed to modification.

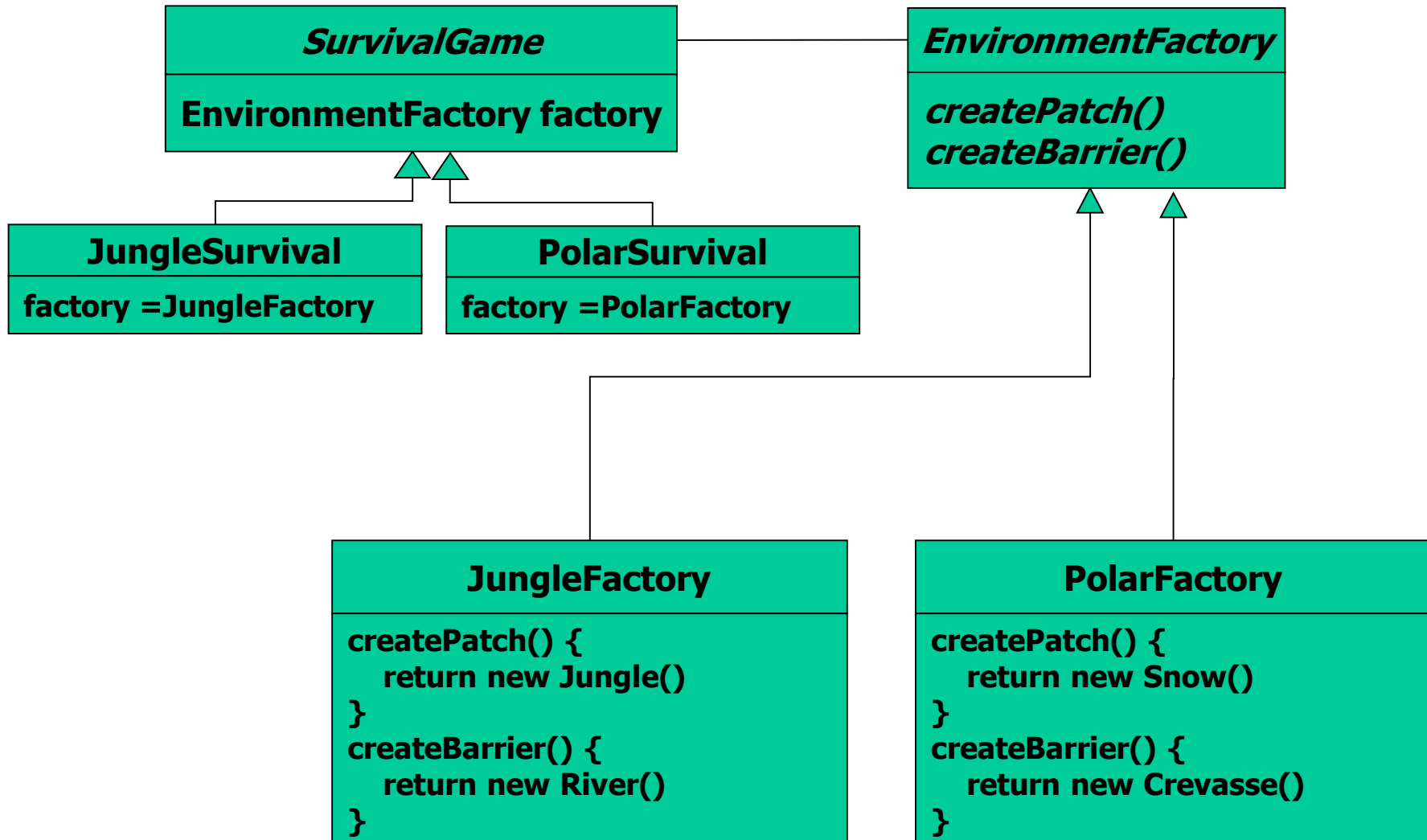
Why is that a good idea?

- The classes that we have already written are, we hope, well tested and extensively used. If we modify these classes we run the risk of regression (i.e. something that currently works might stop working).
- Extending existing classes allows us to add functionality without the risk of regression.
- In other words, if it ain't broke, extend it!

The Abstract Factory Pattern

- When we use the factory method pattern we use inheritance and overriding to determine which set of concrete products gets created.
- Another way of approaching the problem is to use composition, here's how it works.

The Abstract Factory Pattern



Environment Factories

```
interface EnvironmentFactory {  
    Patch createPatch();  
    Barrier createBarrier();  
}
```

```
class JungleFactory implements EnvironmentFactory {  
    public Patch createPatch() {return new Jungle();}  
    public Barrier createBarrier() {return new River();}  
}
```

```
class PolarFactory implements EnvironmentFactory {  
    public Patch createPatch() {return new Snow();}  
    public Barrier createBarrier() {return new Crevasse();}  
}
```

The SurvivalGame Class

```
abstract class AFSurvivalGame {  
  
    EnvironmentFactory factory;  
  
    Segment currentPosition;  
  
    void setupEnvironment() {  
        Patch westPatch = factory.createPatch();  
        Barrier barrier = factory.createBarrier();  
        Patch eastPatch = factory.createPatch();  
  
        westPatch.setEast(barrier);  
        barrier.setWest(westPatch);  
        barrier.setEast(eastPatch);  
        eastPatch.setWest(barrier);  
  
        currentPosition = barrier;  
    }  
}
```

Concrete Game Classes

```
class AFJungleSurvival extends AFSurvivalGame {  
    AFJungleSurvival() {  
        factory = new JungleFactory();  
    }  
}
```

```
class AFPolarSurvival extends AFSurvivalGame {  
    AFPolarSurvival() {  
        factory = new PolarFactory();  
    }  
}
```

Testing

```
class TestAFGames {  
  
    public static void main(String [] args) {  
        AFSurvivalGame jungleGame = new AFJungleSurvival();  
        jungleGame.setupEnvironment();  
        System.out.println("JUNGLE");  
        describe(jungleGame.currentPosition);  
  
        AFSurvivalGame polarGame = new AFPolarSurvival();  
        polarGame.setupEnvironment();  
        System.out.println("\nPOLAR");  
        describe(polarGame.currentPosition);  
    }  
    //describe is defined as in previous example
```



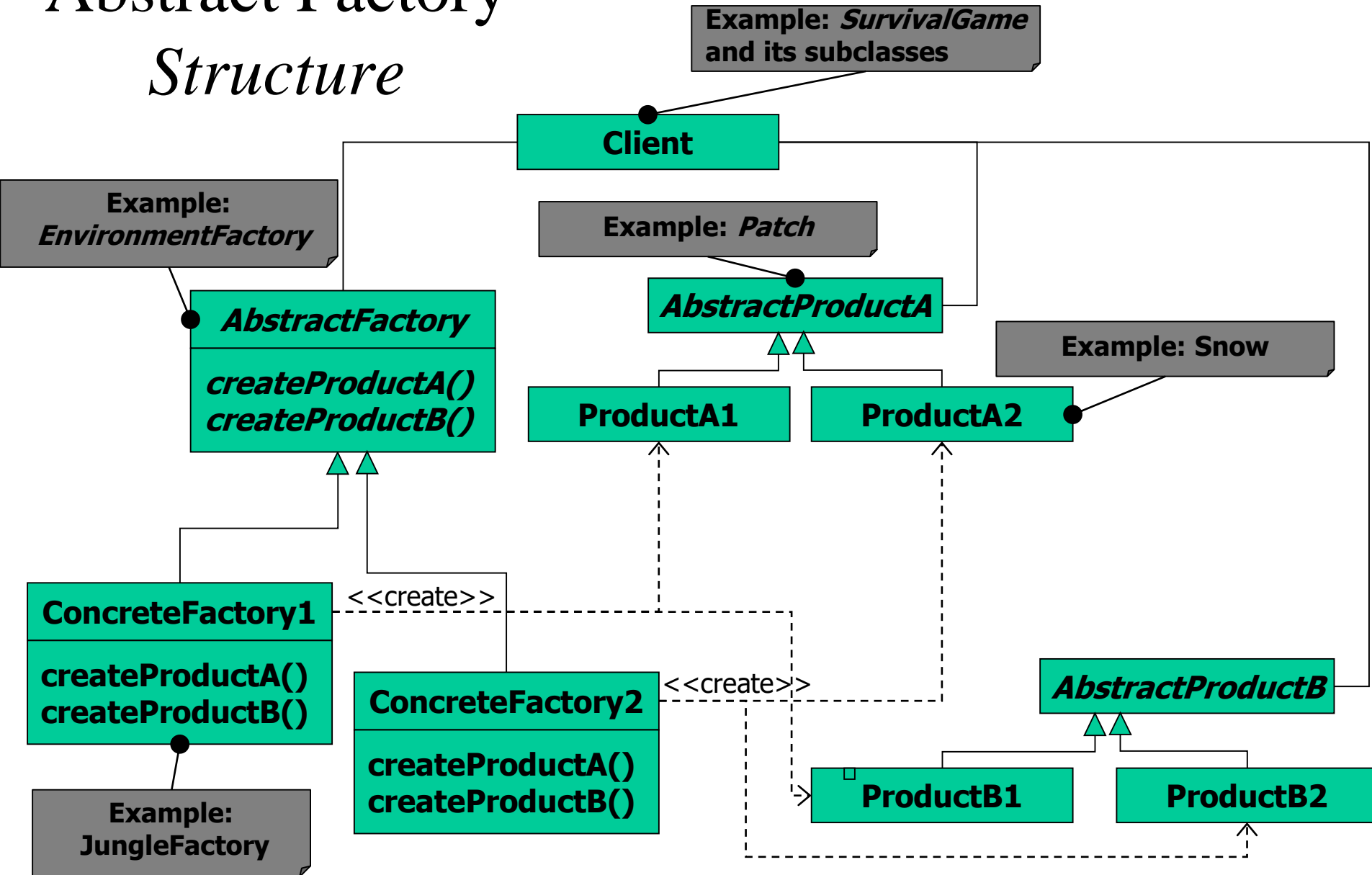
JUNGLE

**You are in a river. To the west is a patch of jungle. To the east is a patch of jungle.
To the north is nothing. To the south is nothing**

POLAR

**You are in a crevasse. To the west is a patch of snow. To the east is a patch of
snow. To the north is nothing. To the south is nothing**

Abstract Factory Structure



Abstract Factory

compared and contrasted with Factory Method

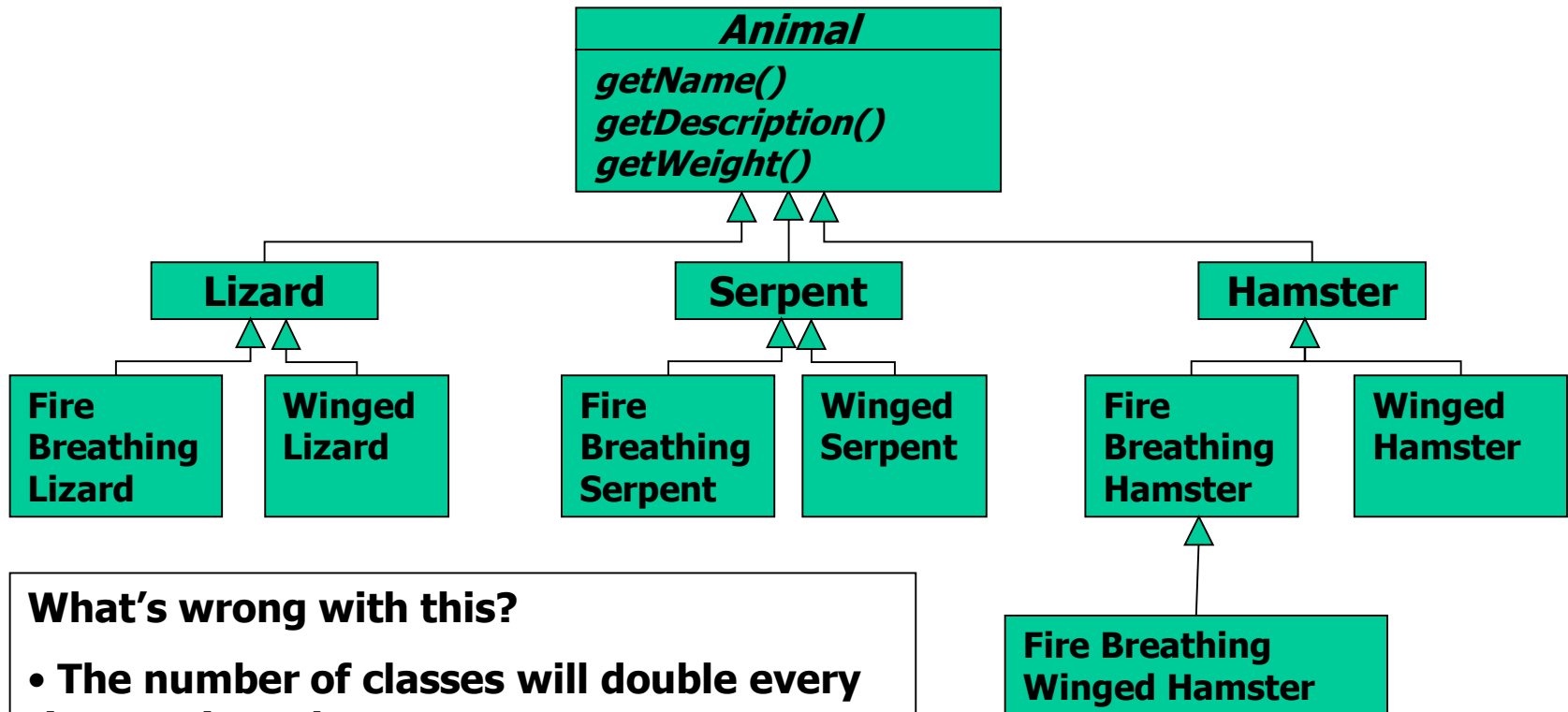
- In the Factory Method pattern the Creator class creates (and potentially uses) products, in the Abstract Factory Pattern creation is delegated to a separate factory class.
- Abstract Factory groups together families of related products.
- Abstract Factory uses composition rather than inheritance.
- Abstract Factory allows factory to be changed at run time.

A Structural Pattern

The Decorator Pattern

- Suppose that we want to develop a fantasy version of the survival game in which players can create mythical animals by adding magical properties to real animals
- Examples:
 - serpent → winged serpent
 - lizard → fire breathing lizard
 - hamster → winged, fire breathing hamster

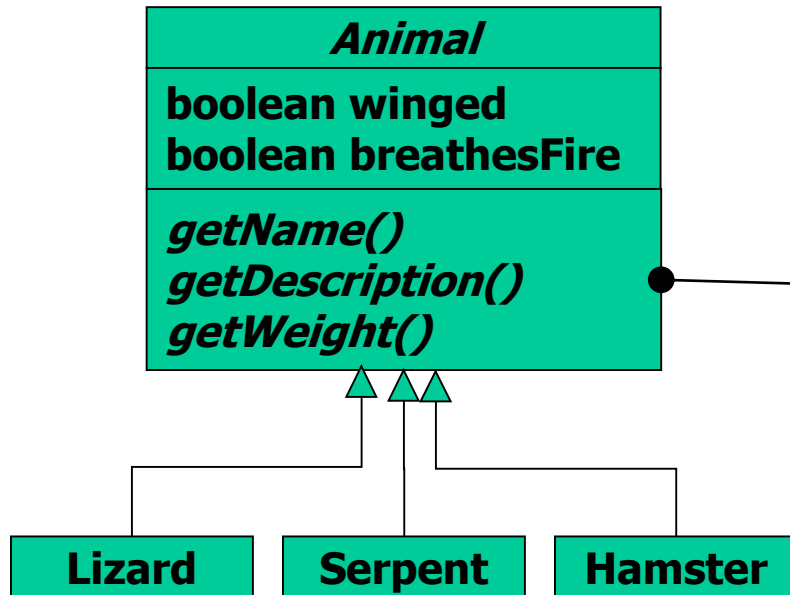
A Possible Class Hierarchy



What's wrong with this?

- The number of classes will double every time we introduce a new property. For instance if we introduce talking animals to our menagerie we will have to add classes for talking, fire breathing lizards; talking winged lizards, etc.
- Can't change an animal's properties at run time.

Another Possible Hierarchy



Implementations of methods will be designed so as to behave differently depending on whether the boolean attributes are true or false.

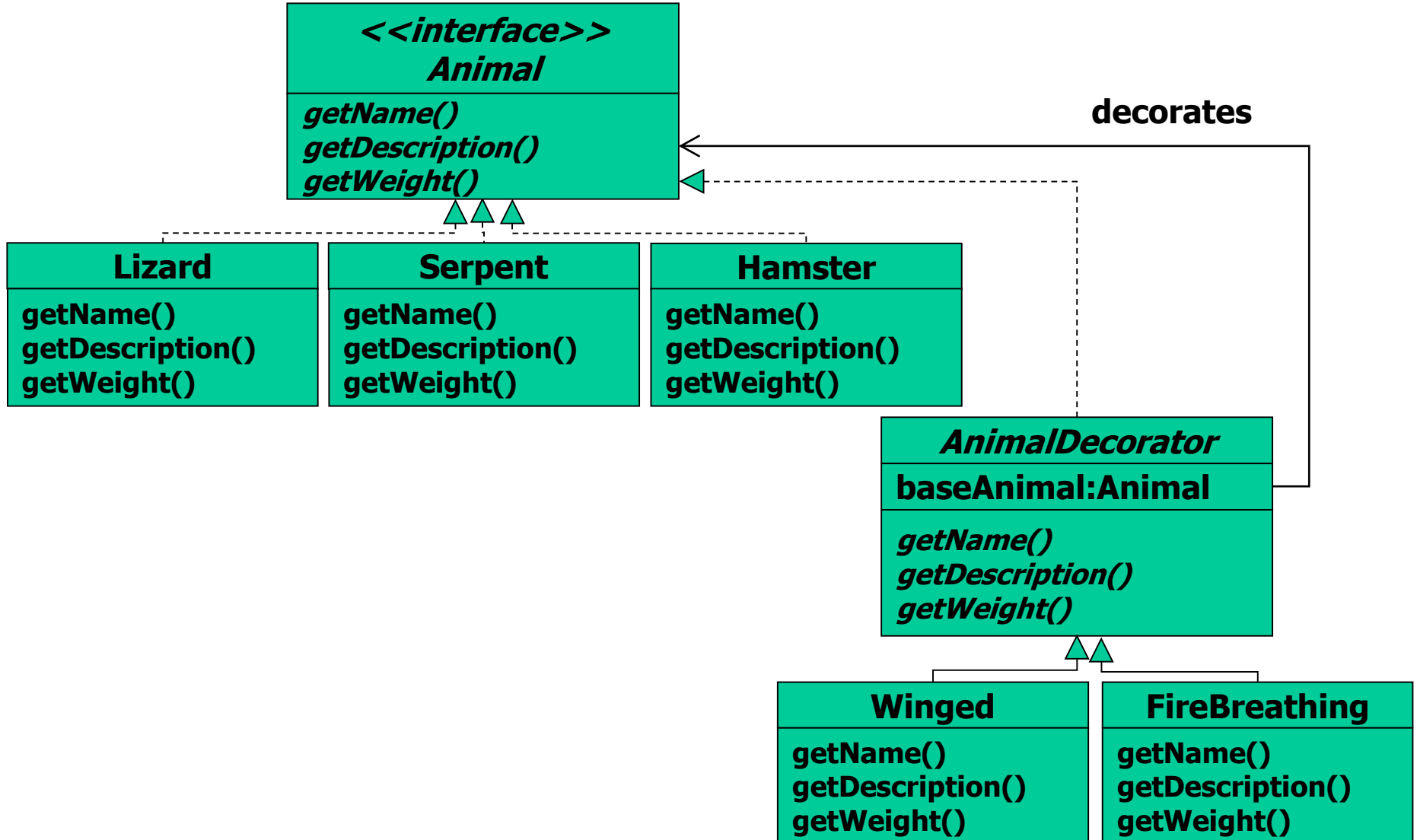
Example:

```
String getDescription() {
    return
        winged?"Winged ":"" +
        breathesFire?"Fire Breathing ":""
        + "Lizard";
}
```

Now what's the problem?

- If we want to introduce a new property (e.g. ability to talk) we will have to *modify* existing code. That violates the "open-closed" principle!

The Decorator Pattern



The Animal Interface

```
interface Animal {  
    String getName();  
    String getDescription();  
    double getWeight();  
}
```

Abstract Animal

```
abstract class AbstractAnimal implements Animal {  
    private String name;  
    private String description;  
    private double weight;  
  
    public AbstractAnimal(String name, String description, double weight) {  
        this.name = name;  
        this.description = description;  
        this.weight = weight;  
    }  
  
    public String getName() {return name;}  
    public String getDescription() {return description;}  
    public double getWeight() {return weight;}  
}
```

Concrete Animals

```
class Hamster extends AbstractAnimal {  
    public Hamster(String name) {  
        //The British Standard Hamster weighs 0.1 Kg  
        super(name, "hamster", 0.1);  
    }  
}  
  
class Serpent extends AbstractAnimal {  
    public Serpent(String name) {  
        super(name, "serpent", 2);  
    }  
}
```

AnimalDecorator

```
abstract class AnimalDecorator implements Animal {  
  
    private Animal baseAnimal; //the animal we are going to decorate  
  
    public AnimalDecorator(Animal baseAnimal) {  
        this.baseAnimal = baseAnimal;  
    }  
  
    public Animal getBaseAnimal() {return baseAnimal;}  
  
    public String getName() {return baseAnimal.getName();}  
    public abstract String getDescription();  
    public abstract double getWeight();  
}
```

Winged Animals

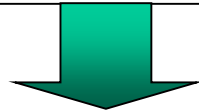
```
class Winged extends AnimalDecorator {  
    public Winged(Animal animal) {  
        super(animal);  
    }  
    public String getDescription() {  
        return "winged " + getBaseAnimal().getDescription();  
    }  
    public double getWeight() {  
        //wings add 10% to an animal's weight  
        return 1.1*getBaseAnimal().getWeight();  
    }  
}
```

Fire Breathing Animals

```
class FireBreathing extends AnimalDecorator {  
    public FireBreathing(Animal animal) {  
        super(animal);  
    }  
    public String getDescription() {  
        return "fire breathing " + getBaseAnimal().getDescription();  
    }  
    public double getWeight() {  
        //all that hydrogen reduces the weight by 20%  
        return 0.8* getBaseAnimal().getWeight();  
    }  
}
```


Test Harness

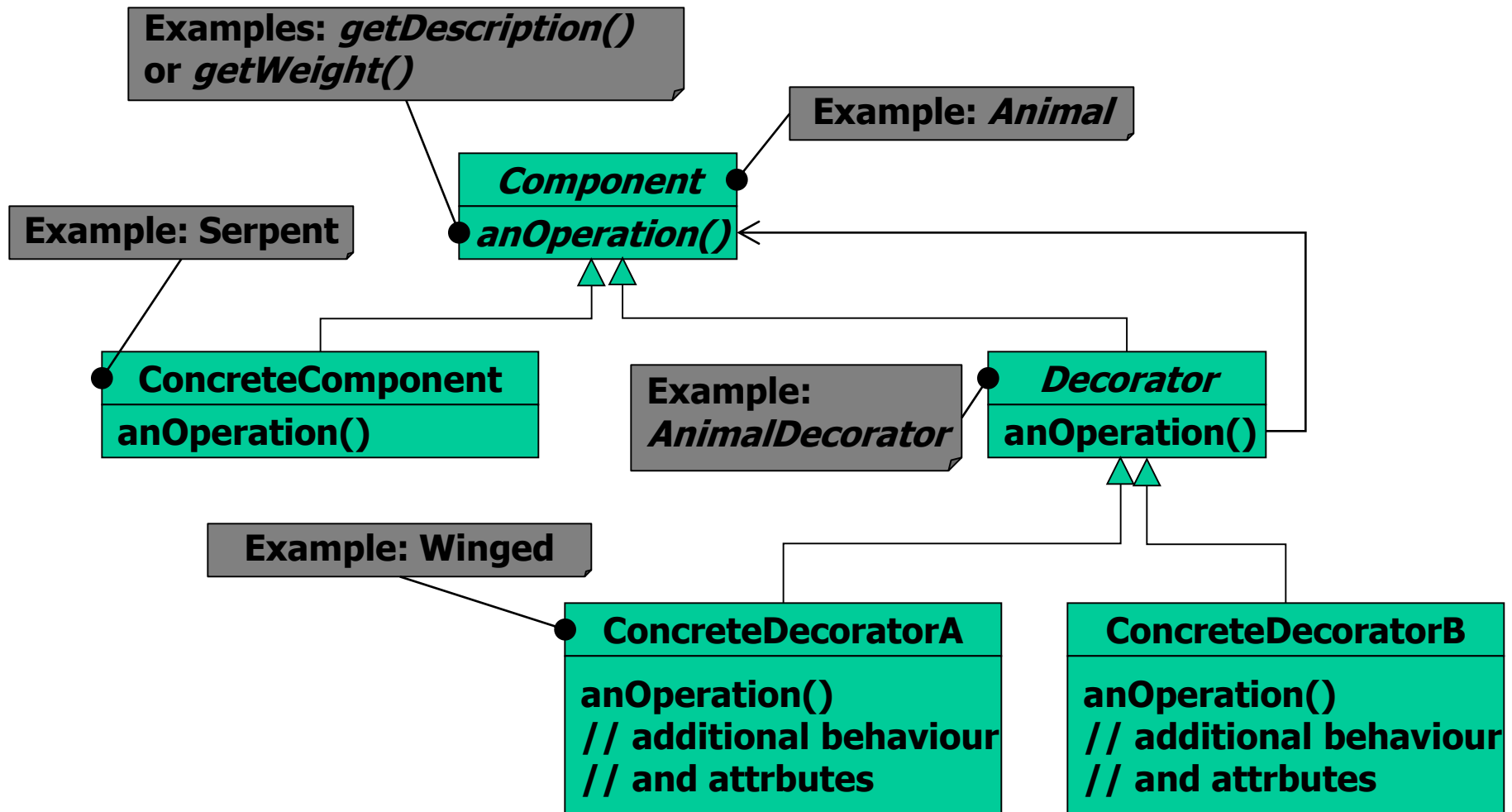
```
public class TestDecorator {  
  
    public static void main(String[] args) {  
        Animal q = new Winged (new Serpent("Q"));  
        describe(q);  
        Animal h = new FireBreathing(new Winged (new Hamster("H")));  
        describe(h);  
    }  
  
    public static void describe(Animal animal)  
    {  
        System.out.print("I am " + animal.getName()  
                        + " the " + animal.getDescription() + ". ");  
        System.out.println("I weigh " + animal.getWeight() + " Kg.");  
    }  
}
```



I am Q the winged serpent. I weigh 2.2 Kg.
I am H the fire breathing winged hamster. I weigh 0.088000000000000002 Kg.

The Decorator Pattern

Structure



Decorator Pattern

Points to Note

- The *Decorator* classes implement the same *Component* interface as the *ConcreteComponent* classes. However a *Decorator* object is also *contains* a *Component* object.
- When we add a new property we simply add one decorator class. No other classes need to be added or modified.

Summary

Design Principles

- Favour composition over inheritance.
- Encapsulate what varies.
- Program to an interface not an implementation.
- Classes should be open to extension but closed to modification.

Summary

Design Patterns

- **Strategy:** *encapsulates behaviour which varies between objects into a separate class.*
- **Factory method:** *object creation is handled by factory methods which are overridden by concrete creator classes.*
- **Abstract factory:** *object creation handled by separate factory classes that create families of related products.*
- **Decorator:** *functionality added to objects by composition with decorator classes that have the same interface as the objects that they decorate.*