Introduction to Design Patterns

Dr Ian Bayley[1]

Oxford Brookes

Advanced O-O Prog

[1] with thanks to David Sutton

Outline

1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

Example: Storing Don Giovanni's Girlfriends

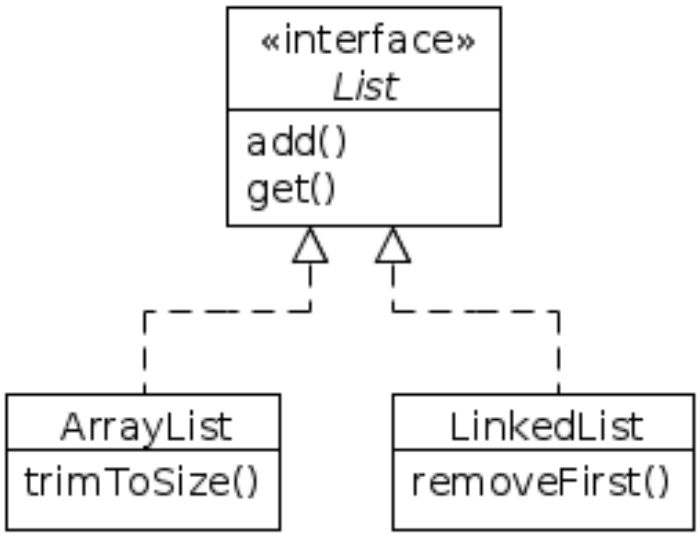
- consider two collections: Italian girlfriends and Spanish girlfriends

One Representation

```
ArrayList <String> italians;
LinkedList <String> spaniards;
```

A Better Representation

```
List italians;
List spaniards;
```



- if you treat variables as lists,
  you don't expose their concrete types
  so code is simpler and easier to change

Fundamentals of Object-Orientation

Abstraction Ability to access objects in a way that obscures details that we do not want to be dependent on and which would unnecessarily complicate code

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 they control access to those properties
e.g. public and private in Java

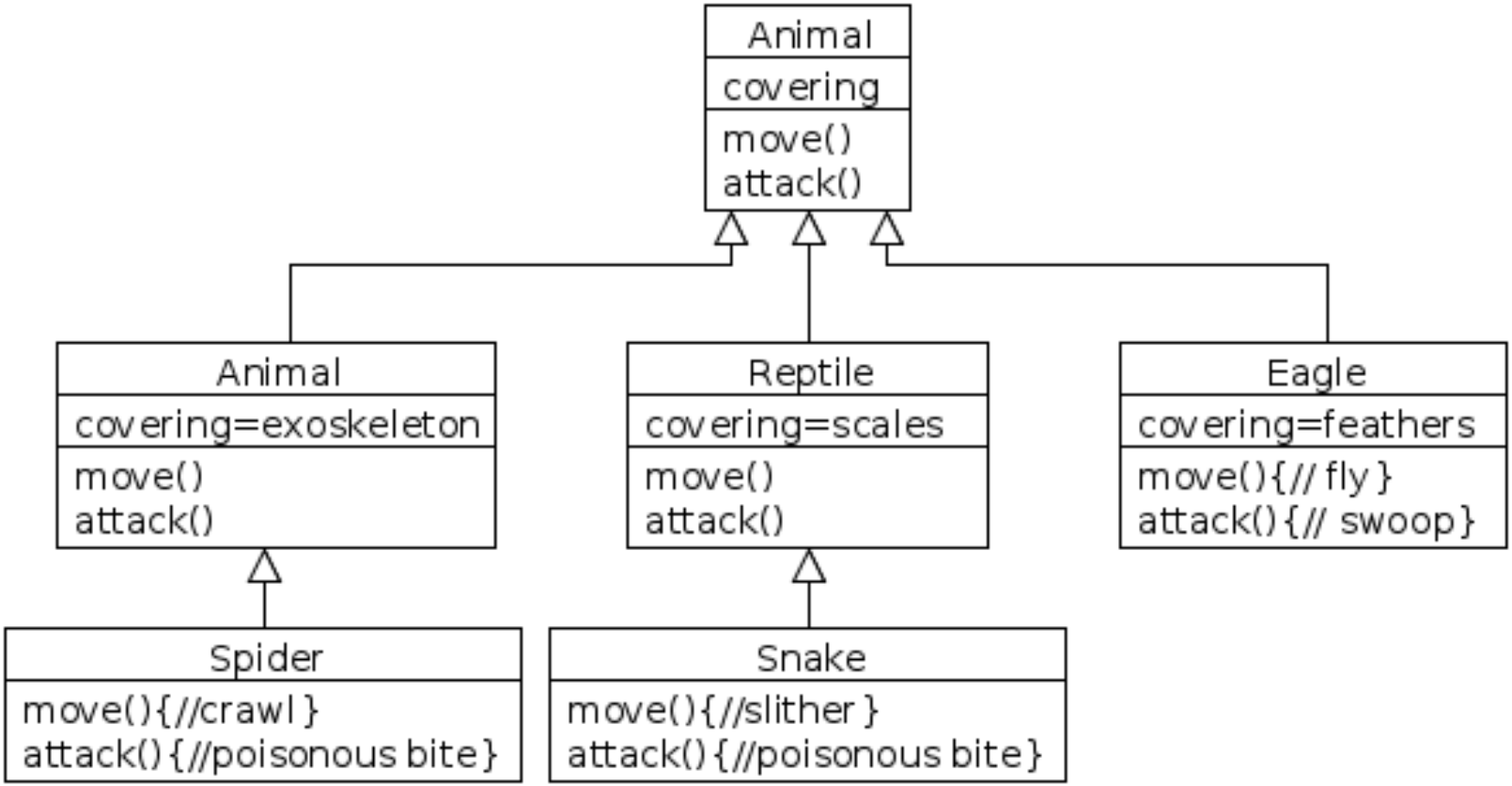
- is that all there is to know?

Outline

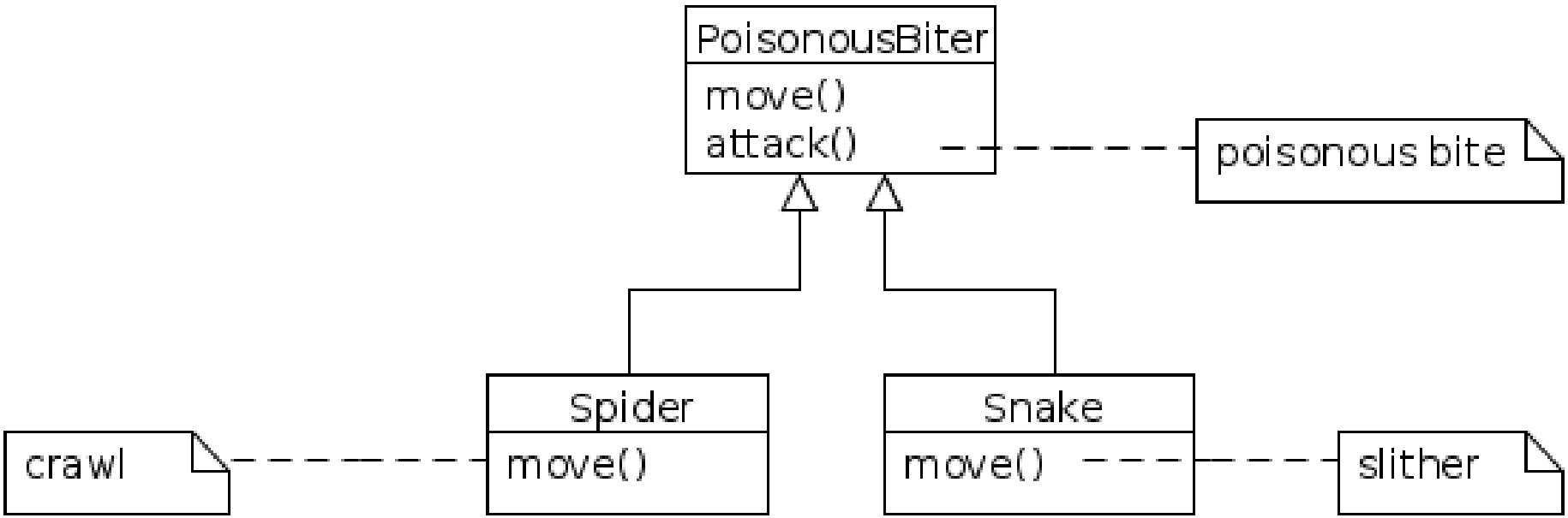
1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

Class Diagram for the Game of Survival

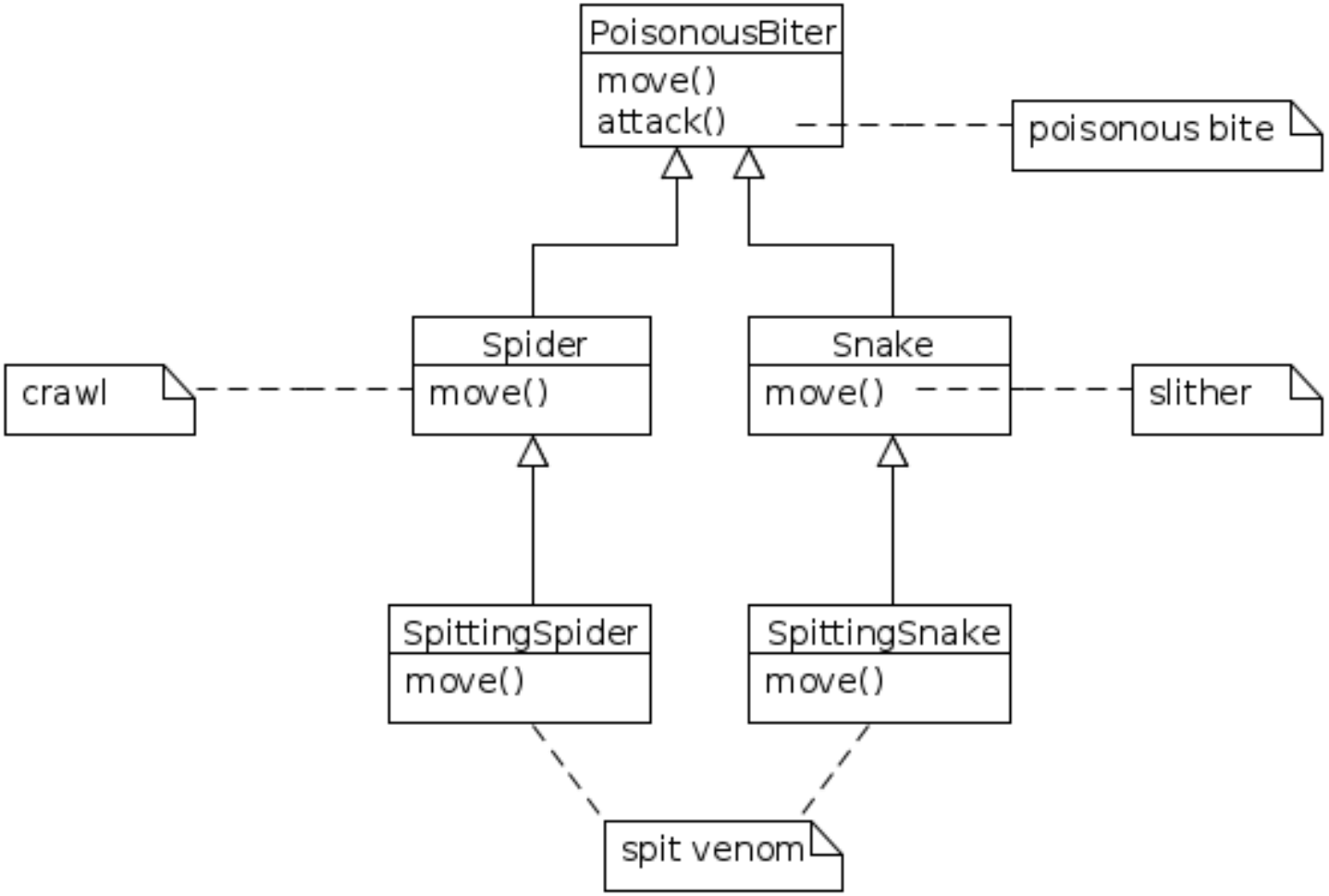
- consider a multi-player game where the playable characters are animals
- each one implements methods that allow it to move, attack, eat etc



Sharing Poisonous Bite Code



Failing to Share Spitting Code



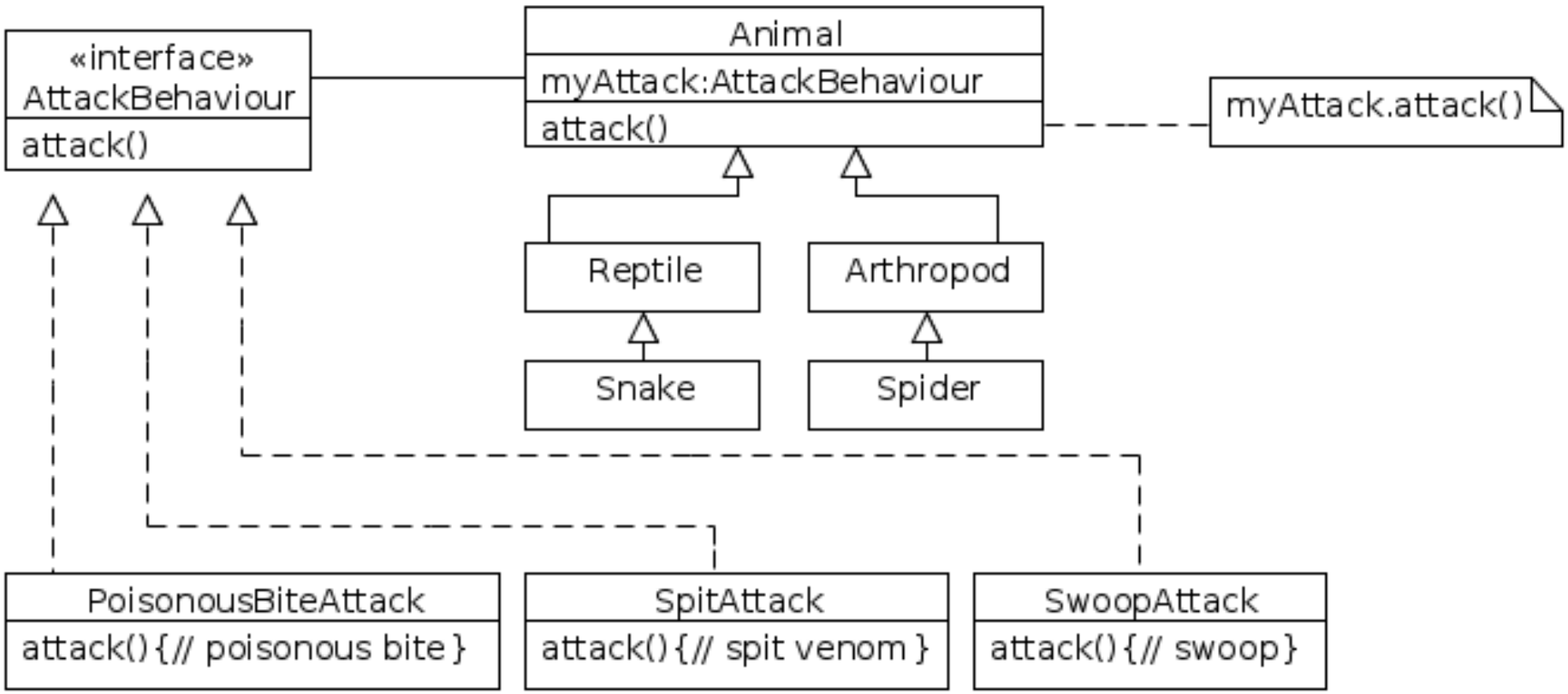
- we have already used inheritance once to share poisonous bite code
- we can't use inheritance for a second time to share spitting code

Outline

1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

A Better Solution: Encapsulate the Behaviour

- put the attack code into a separate class hierarchy
- move code could also be moved to a separate class hierarchy



Code for Animal

```
public abstract class Animal {
    String name;
    String description;
    String covering;
    AttackBehaviour myAttack;
    // myAttack is object that performs the attack
    public Animal (String name) {
        this.name = name;
    }
    abstract void move();
    void performAttack() {
        myAttack.attack();
    }
}
```

Code for Eagle (subclass of Animal)

```
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");
    }
}
```

Code for Reptile (subclass of Animal)

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

Code for Snake (subclass of Reptile)

```
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");
    }
}
```

Code for AttackBehaviour Class Hierarchy

```
public interface AttackBehaviour {
    void attack();
}
public class PoisonBiteAttack implements AttackBehaviour {
    public void attack() { // usually more complicated!
        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!");
    }
}
```

Code for 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);
    }
    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();
    }
}
```

Output from Testing

```
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!
```

Outline

1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

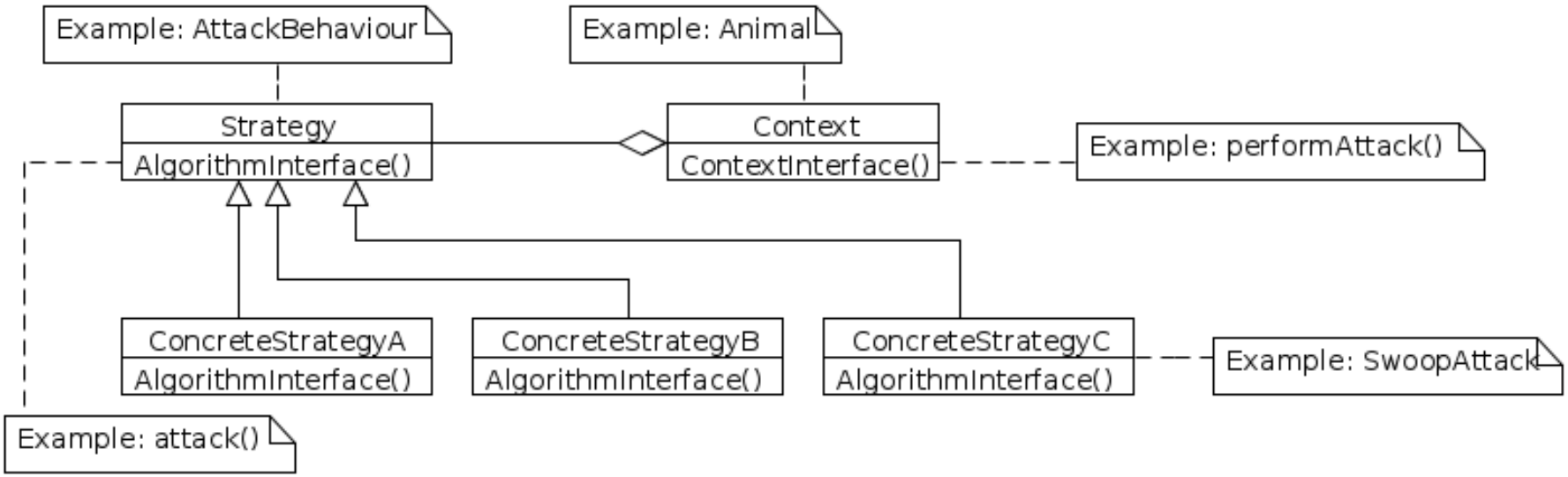
Introduction to Patterns

- A pattern is a solution to a problem that occurs quite frequently, and has been used more than once in the past
- The Gang of Four book is the most well-known pattern catalogue
- Patterns can be classified as:

behaviour	describe how objects interact and distribute responsibility
creational	concern the process of creating objects
structural	deal with composition of objects into larger systems

- Patterns also embody Design Principles. For Strategy they are
  - favour composition over inheritance
  - encapsulate what varies (=lists of methods)
  - program to an interface not to an implementation

Class Diagram for the Strategy Pattern



How to Describe a Pattern

Name: so that the pattern becomes part of design vocabulary

Classification: what kind of pattern is it

Intent: short description of what pattern does and problem it solves

Motivation: concrete example of the use of the pattern

Applicability: when to use the pattern

Structure: diagram showing the classes and their relationships

Participants: describes the classes that participate in the above structure

Collaborations: describes how patterns work together

Consequences: good and bad effects of patterns

Implementation: techniques and issues you should be aware of when implementing the pattern

Sample code: fragments to illustrate how you may implement the pattern

Known uses: at least two or three

Related paterns

The Gang of Four Patterns

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

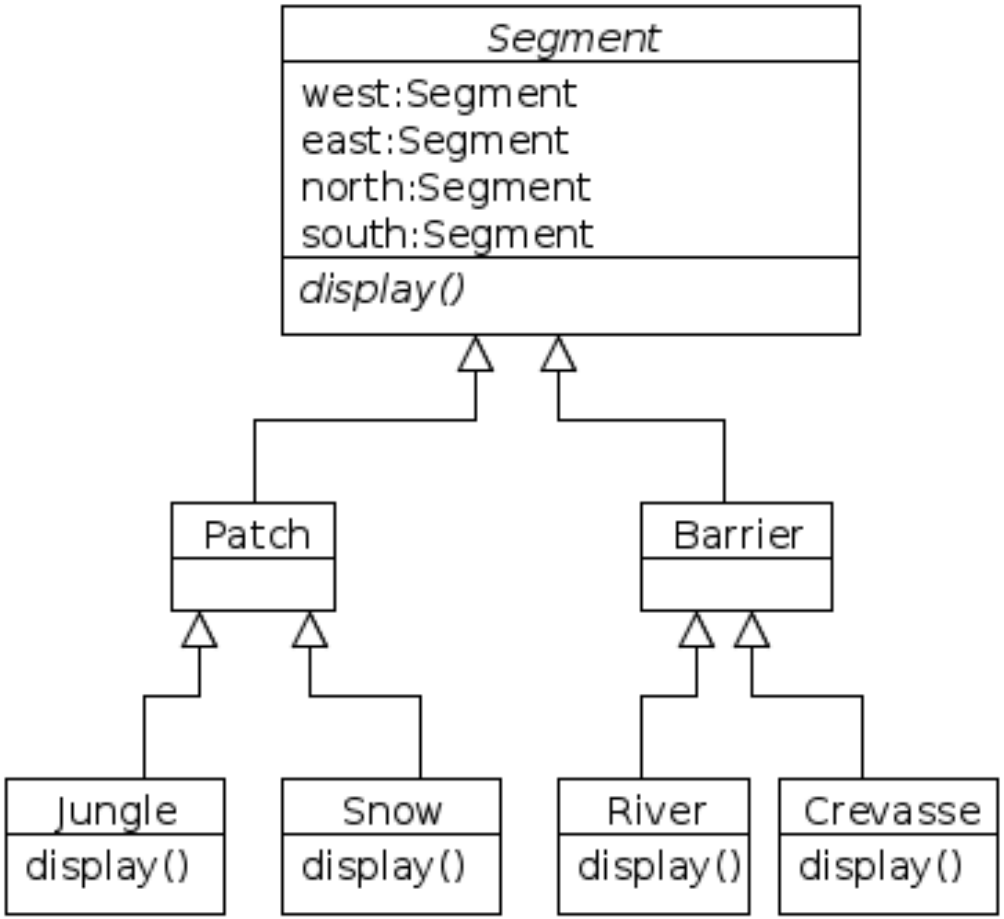
Outline

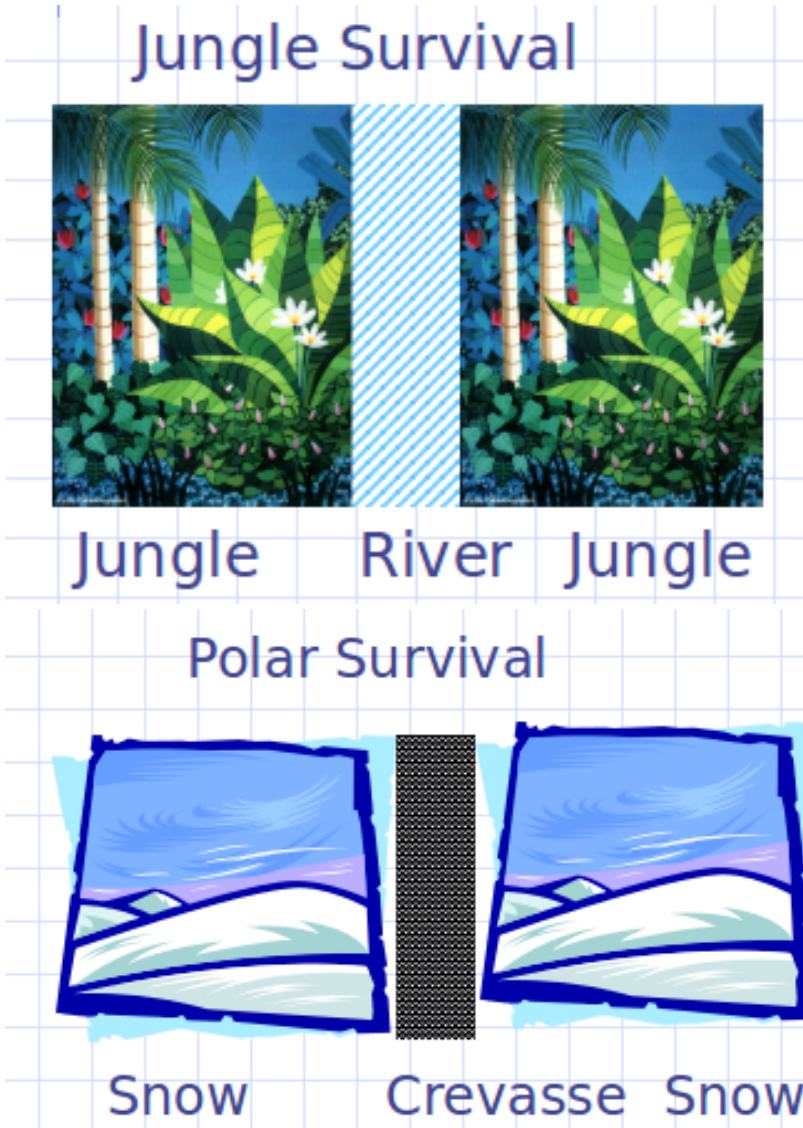
1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

Creating Survival Environments

```
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();
    }
}
```

Two Types of Survival Environments





Code for Polar Survival Environment (Bad Idea 1)

```
//class JungleSurvival { // modifying original file
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);
        // and so on ...
    }
}
```

- two different copies to maintain

Code for JungleOrPolar Survival Environment (Bad Idea 2)

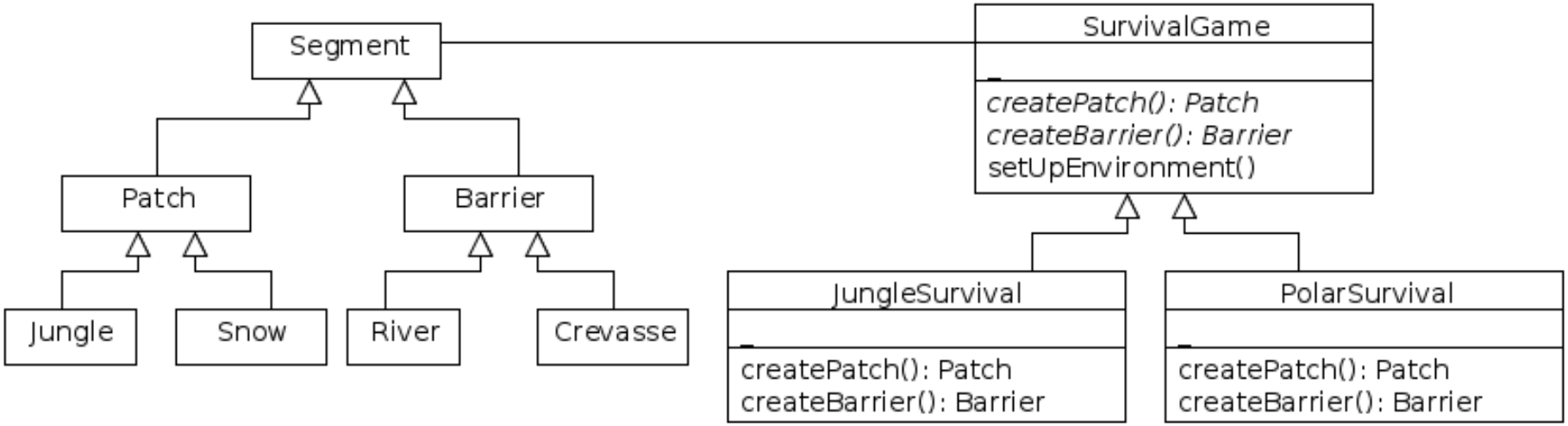
```
class JungleOrPolarSurvival {
    String type;
    setUpEnvironment {
    Patch westPatch;
    if (type == “jungle”)
        westPatch = new Jungle();
    else if (type == “polar”)
        westPatch = new Snow();
    Barrier barrier;
    if (type == “jungle”)
        barrier = new River();
    else if (type == “polar”)
        barrier = new Crevasse();
// and so on
}
```

- if there is a new type like Savannah then class must change

Outline

1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

Class Diagram for Survival Environments



- `JungleSurvival` creates `Jungle` and `River`
- `PolarSurvival` creates `Snow` and `Crevasse`

Code for SurvivalGame

```
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();
}
```

Code for 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();
    }
}
```

Code for Segment

```
abstract class Segment {
    Segment west, east, north, south; //neighbouring
    Segment() {west= east= north = 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();
}
```

Code for Barriers and Patches

```
abstract class Patch extends Segment{ /* more later */}
abstract class Barrier extends Segment { /* more later */}

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");}
}
```

Code for 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);
    }
}
```

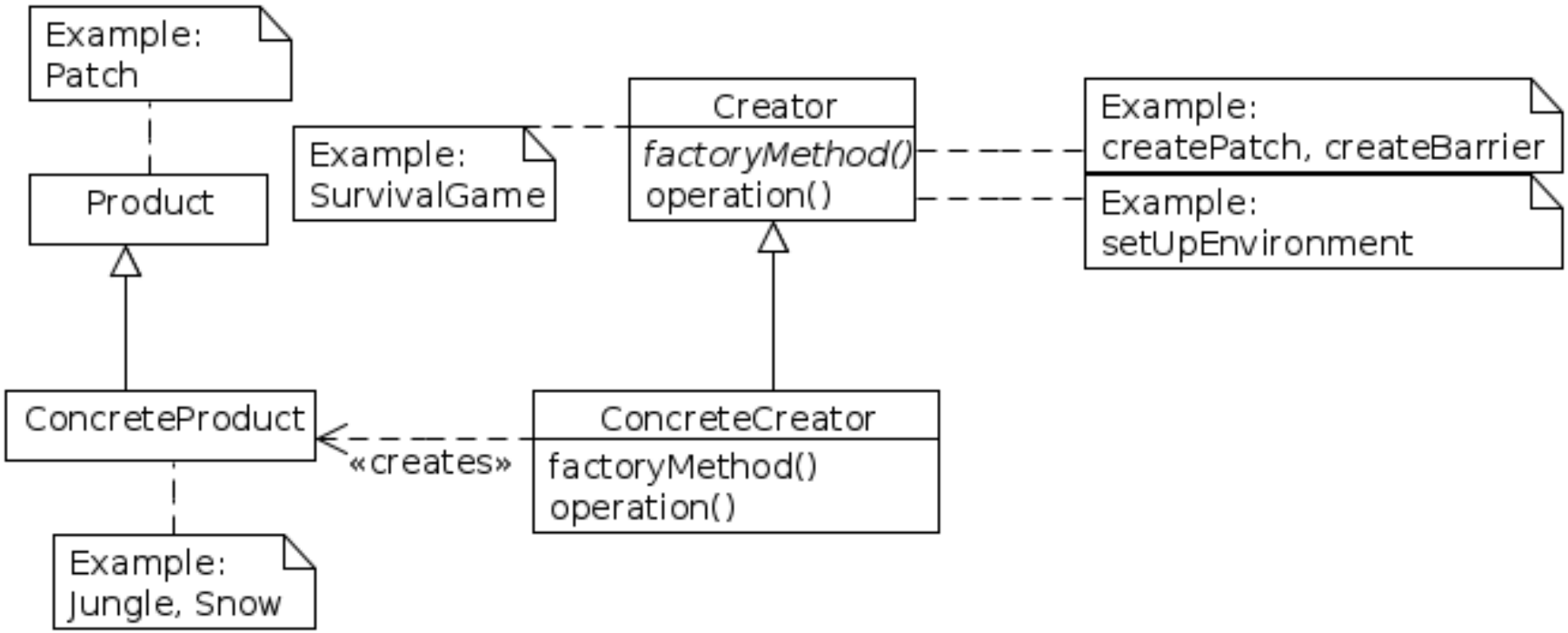
Code for Testing the Classes (More)

```
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();
}
```

Result of Testing the Classes

```
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
```

Class Diagram for Factory Method Pattern



Design Principles within Factory Method Pattern

The Open-Closed Principle

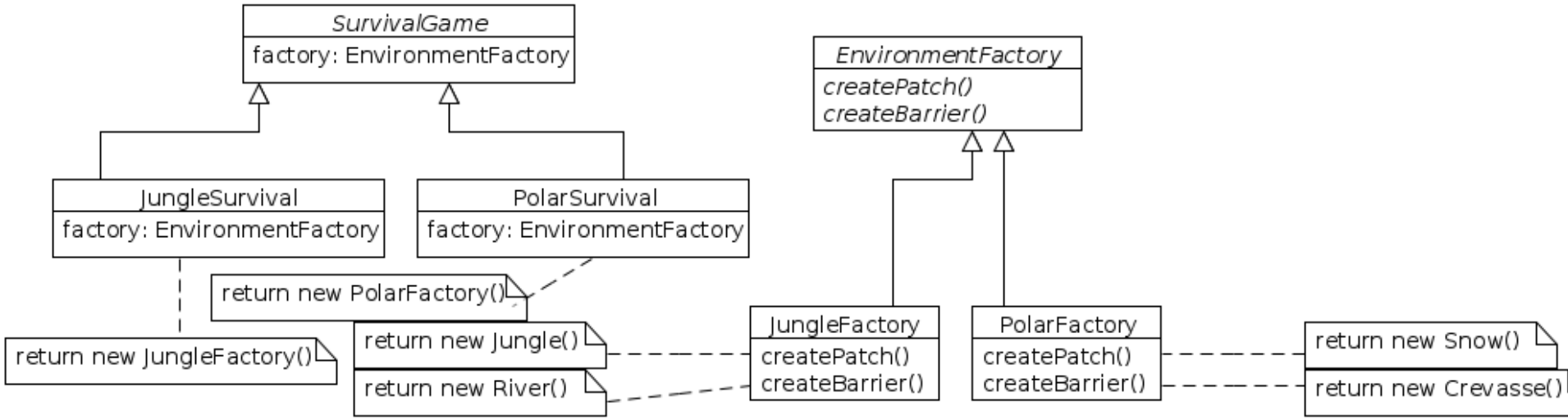
Classes should be open to extension but closed to modification

- this is considered good because existing classes should already be tested so it is best not to modify them
- in other words, "if it ain't broke don't change it"
- note that the ConcreteProduct classes are hidden from the Creator class

Outline

1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

Class Diagram Using Abstract Factory Pattern for Survival



- Abstract Factory uses composition instead of inheritance and overriding

Code for Environment Factory Class Hierarchy

```
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();}
}
```

Code for 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;
    }
}
```

Code for SurvivalGame concrete Subclasses

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

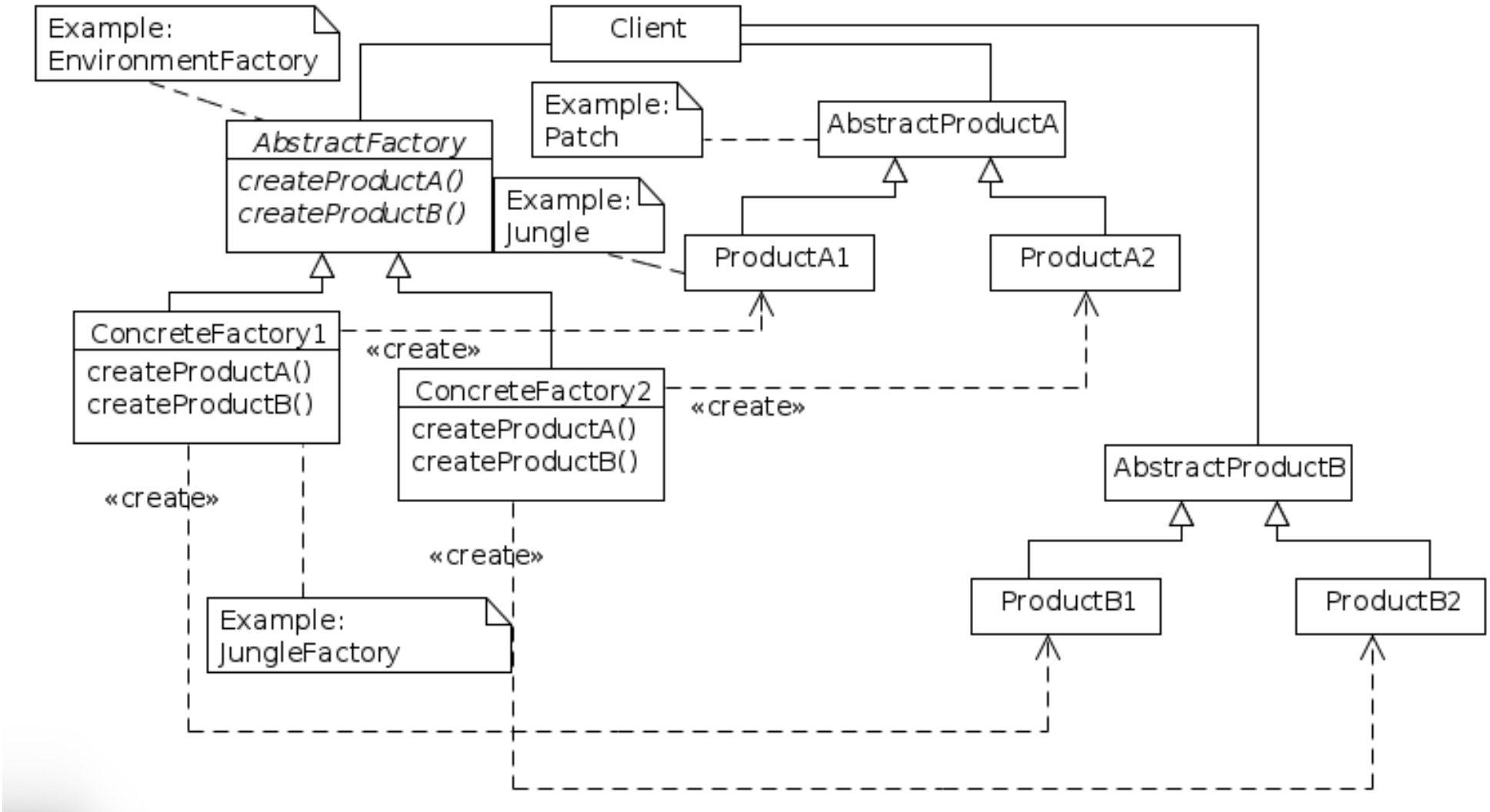
Code for Testing the Classes

```
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
}
```

Result of Testing the Classes

```
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
```

Class Diagram for Abstract Factory Pattern



How Abstract Factory is different from Factory Method

- creation is delegated to separate factory classes
- products of the same family are grouped together
- composition is used rather than inheritance
- so the choice of factory can be made at run-time

Outline

1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

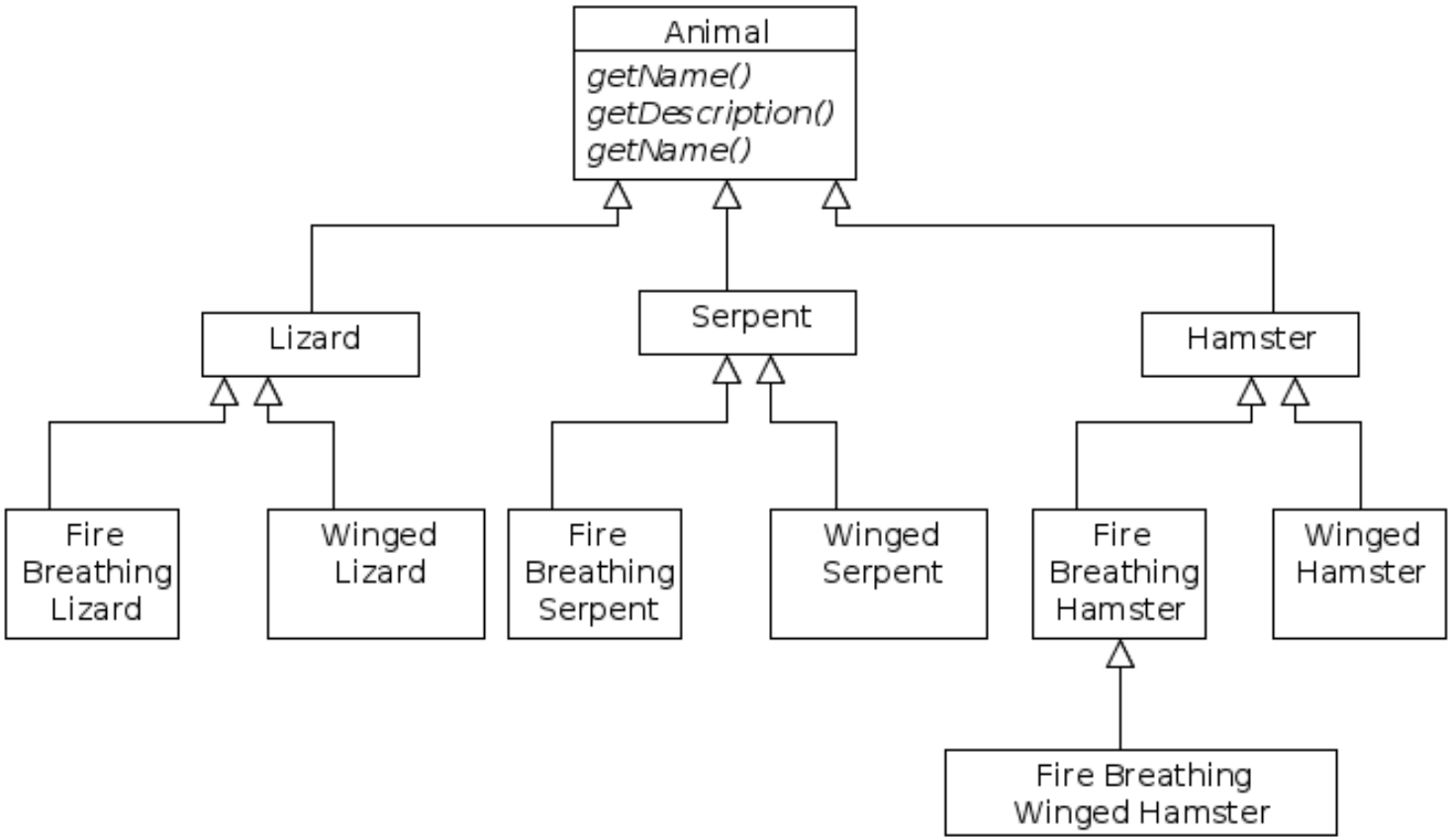
A Modification to Survival

- suppose we want a fantasy version of the Survival Game
- mythical animals created by adding properties to conventional animals

Examples of Adding Properties to Animals

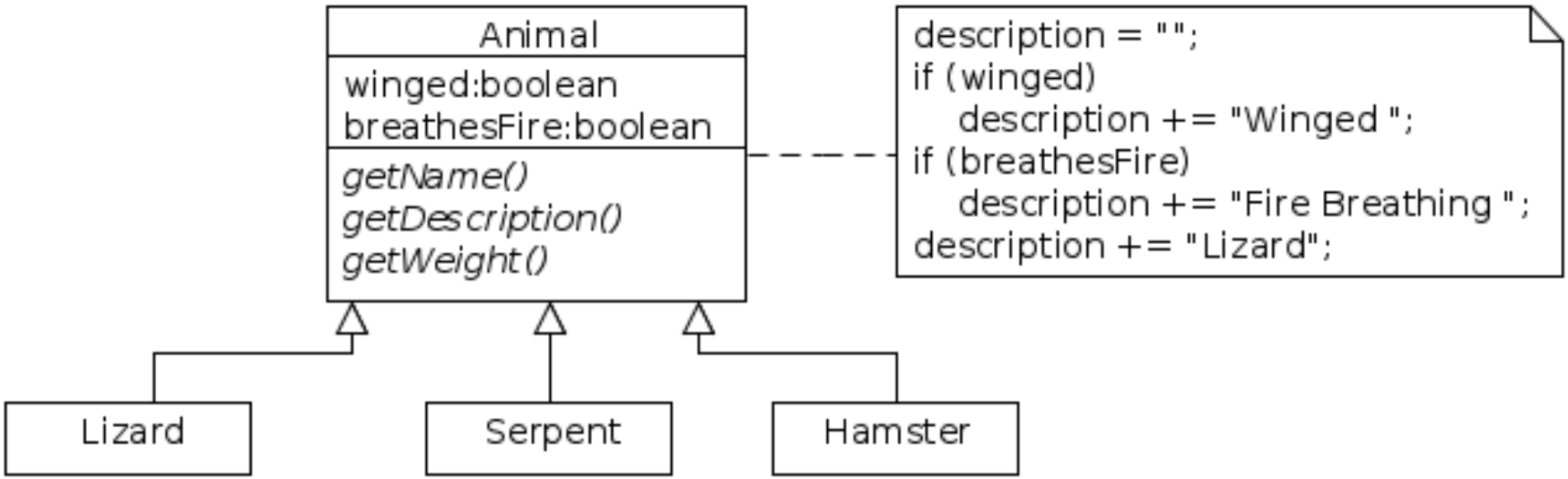
- Serpent $\Rightarrow$ Winged Serpent
- Lizard $\Rightarrow$ Fire-Breathing Lizard
- Hamster $\Rightarrow$ Winged, Fire-Breathing Hamster

Class Diagram for Enhancements



- each new property doubles the number of classes
- cannot change an Animal's properties at run-time

Another Class Diagram for Enhancements

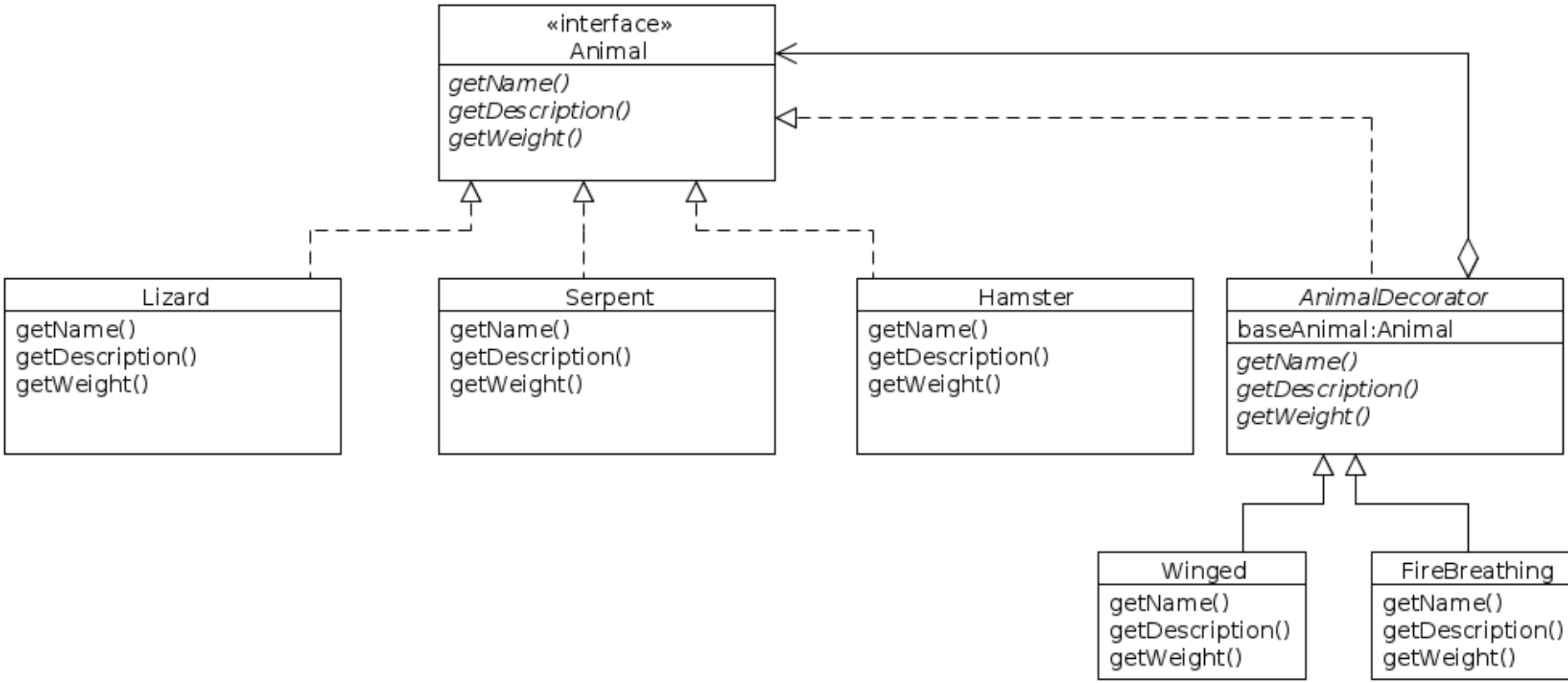


- but this violates the Open-Closed principle, because if you add a new enhancement then `getName` must changed

Outline

1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

Class Diagram Using Decorator Pattern



Code for Animal Interface

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

Code for AbstractAnimal

```
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;}
}
```

Code for Hamster, Serpent (concrete classes)

```
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);
    }
}
```

Code for AnimalDecorator

```
abstract class AnimalDecorator implements Animal {
    private Animal baseAnimal; // which we will 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();
}
```

Code for Winged Decorator Class

```
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();
    }
}
```

Code for FireBreathing Decorator Class

```
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();
    }
}
```

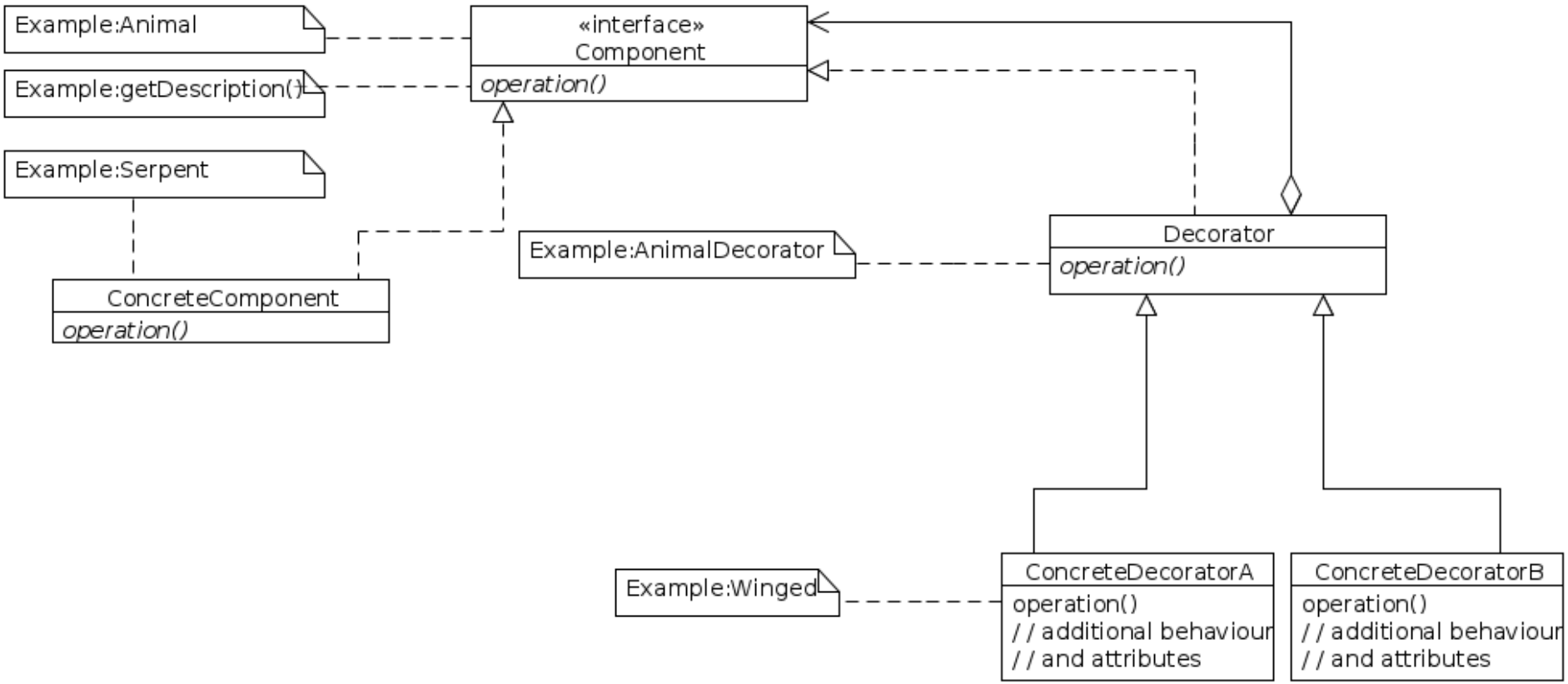
Code for Testing the Classes

```
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.");
    }
}
```

Result of Testing the Classes

```
I am Q the winged serpent.  I weigh 2.2 Kg.
I am H the fire breathing winged hamster.  I weigh
0.08800000000000002 Kg.
```

Class Diagram of Decorator Pattern



Outline

1. What is Object-Orientation?
2. Without the Strategy Pattern
3. With the Strategy Pattern
4. The Concept of Patterns
5. Without the Factory Method Pattern
6. With the Factory Method Pattern
7. Abstract Factory Pattern
8. Without Decorator Pattern
9. With Decorator Pattern
10. Conclusion

Design principles

- favour composition over inheritance eg Strategy, Abstract Factory
- encapsulate what varies eg Strategy
- program to an interface, not an implementation
- classes should be open to extension but closed to modification

The Last Word from a Former U08186 student

"I feel that it would be beneficial to future students to understand just how crucial a concept this is to any aspiring developer. The technologies I work with are at the cutting edge and introduce many complex concepts of their own, however these are all principally based on the work done by the gang of four. I think its fair to say that without your lectures I would not have the skills needed to carry out my work."