




More design patterns

Alexandre Bergel

Nancy Hitschfeld

30/11/2020



Roadmap

- 1.Template
- 2.Composite
- 3.Null-Object
- 4.Factory
- 5.Singleton
- 6.Flyweight

Template Method Pattern

How do you implement a generic algorithm, deferring some parts to subclasses?

Define it as a Template Method

A Template Method factors out the common part of similar algorithms, and delegates the rest to:

hook methods that subclasses *may extend*, and

abstract methods that subclasses *must implement*

Template Method Pattern

Example

The method `init()` in the `AbstractBoardGame` that is defined in subclasses

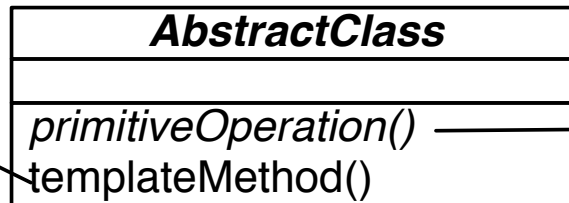
`TestCase.runBare()` is a template method that calls the hook method `setUp()`

Consequences

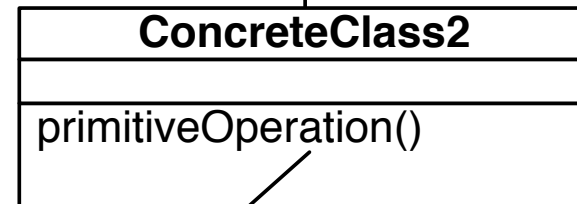
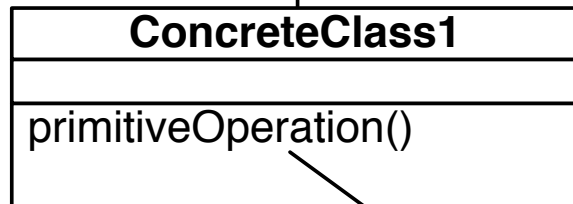
Template methods lead to an *inverted control structure* since a parent class calls the operations of a subclass and not the other way around.

Template Method is used in most frameworks to allow application programmers to easily extend the functionality of framework classes.

```
/** Define the
skeleton of the algorithm
*/
...
this.primitiveOperation()
...
```



Hook method
May be abstract or
simply empty



Overrides the base
class method

Template Method Pattern - Example

Subclasses of TestCase are expected to *override hook method* `setUp()` and possibly `tearDown()` and `runTest()`

Composite Pattern

How do you manage a part-whole hierarchy of objects in a consistent way?

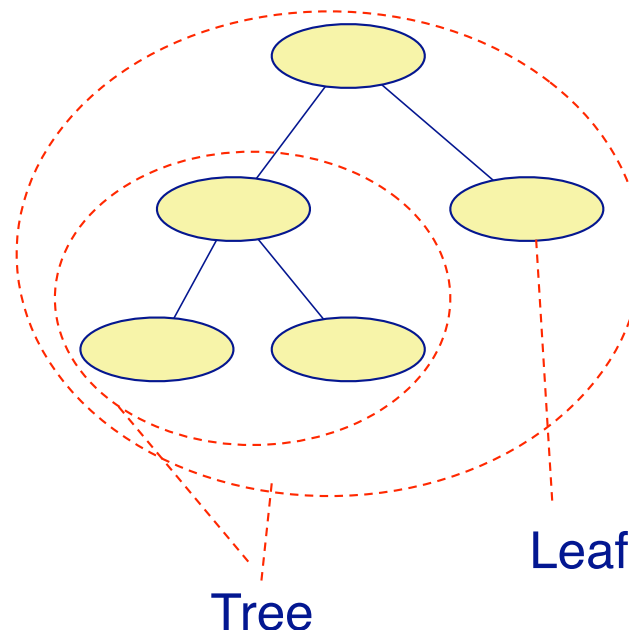
Define a common interface that both parts and composites implement

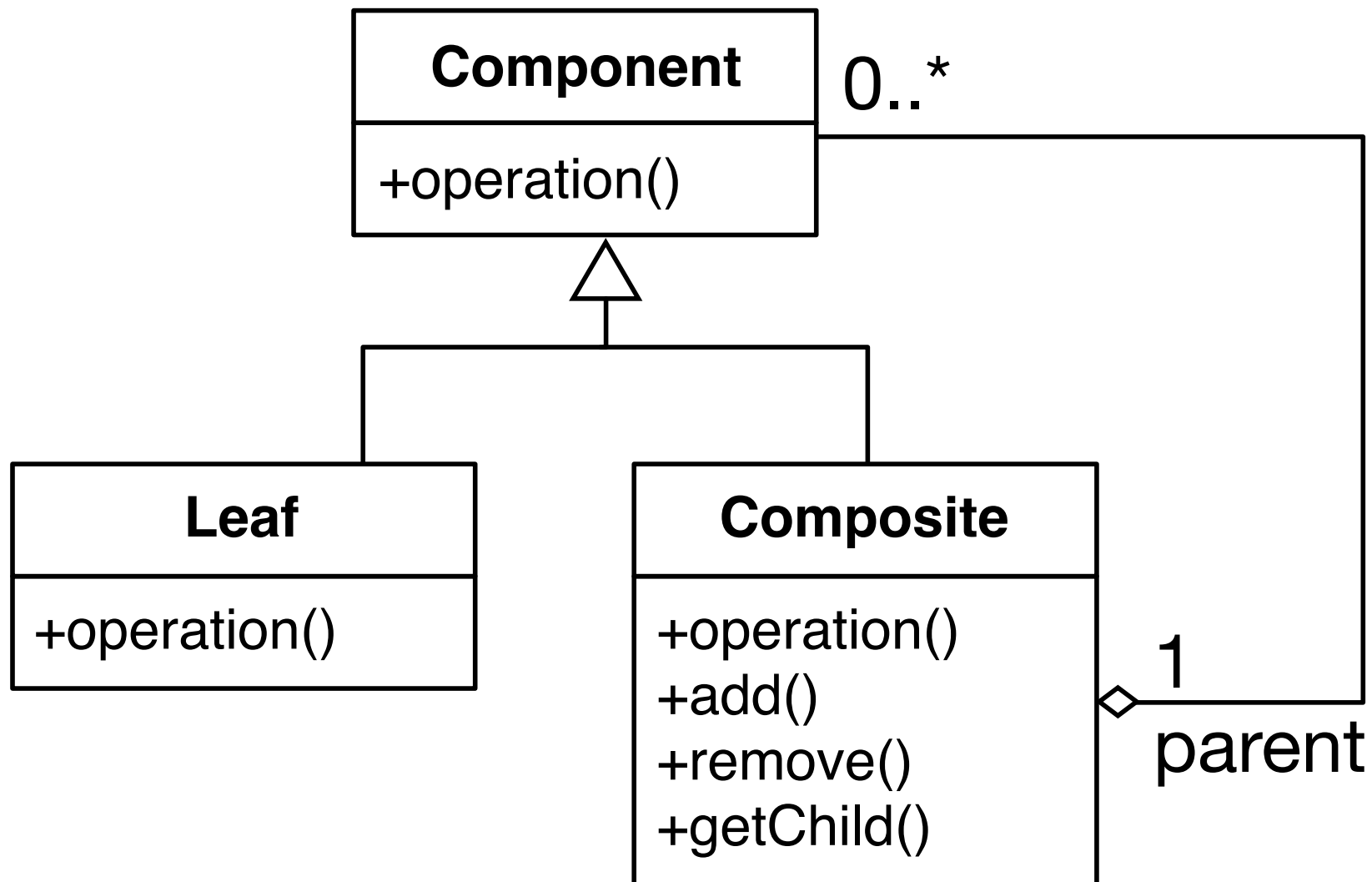
Typically composite objects will implement their behavior by *delegating to their parts*

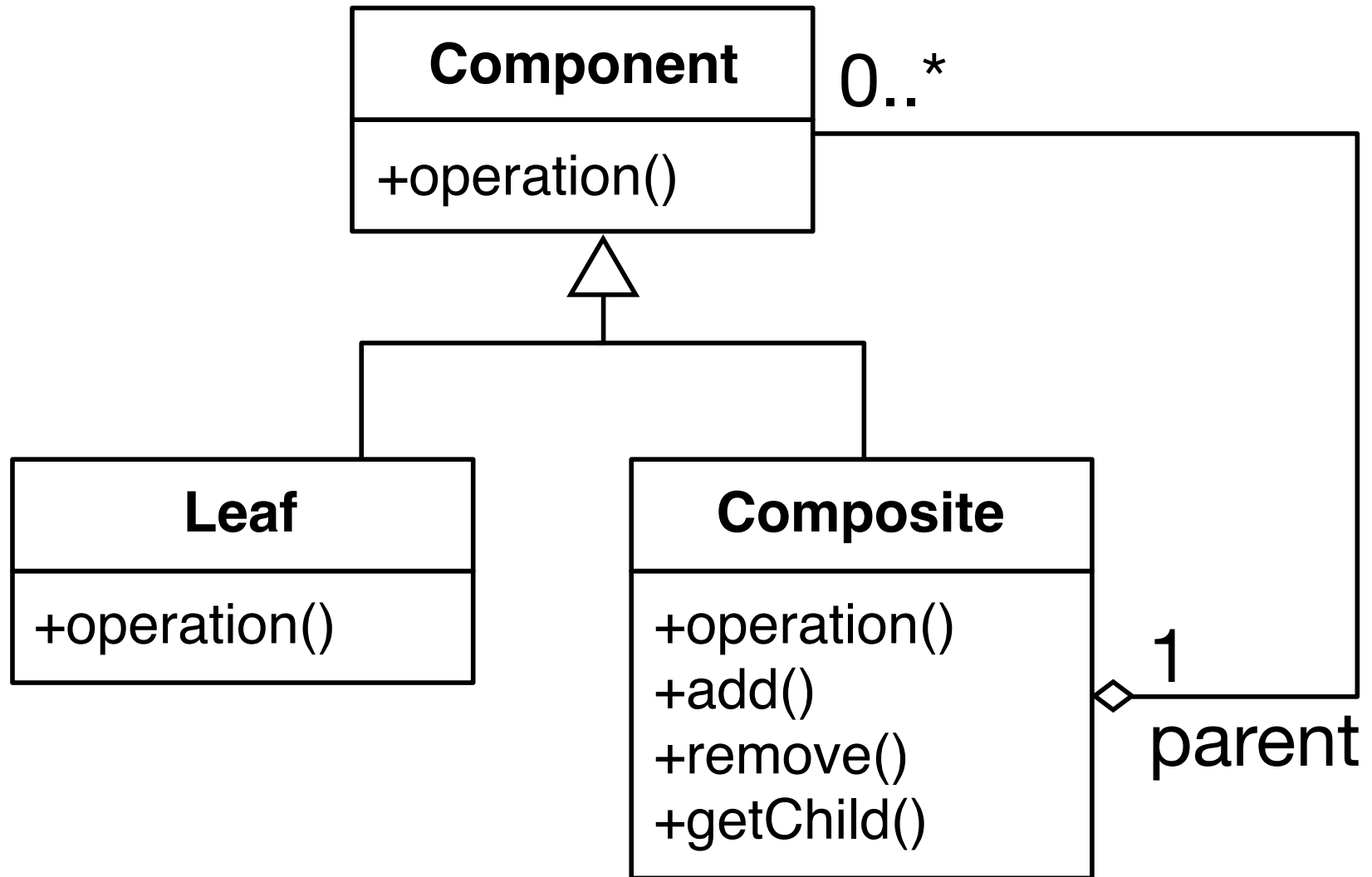
Composite Pattern Example

Composite allows you to treat a single instance of an object the same way as a *group* of objects.

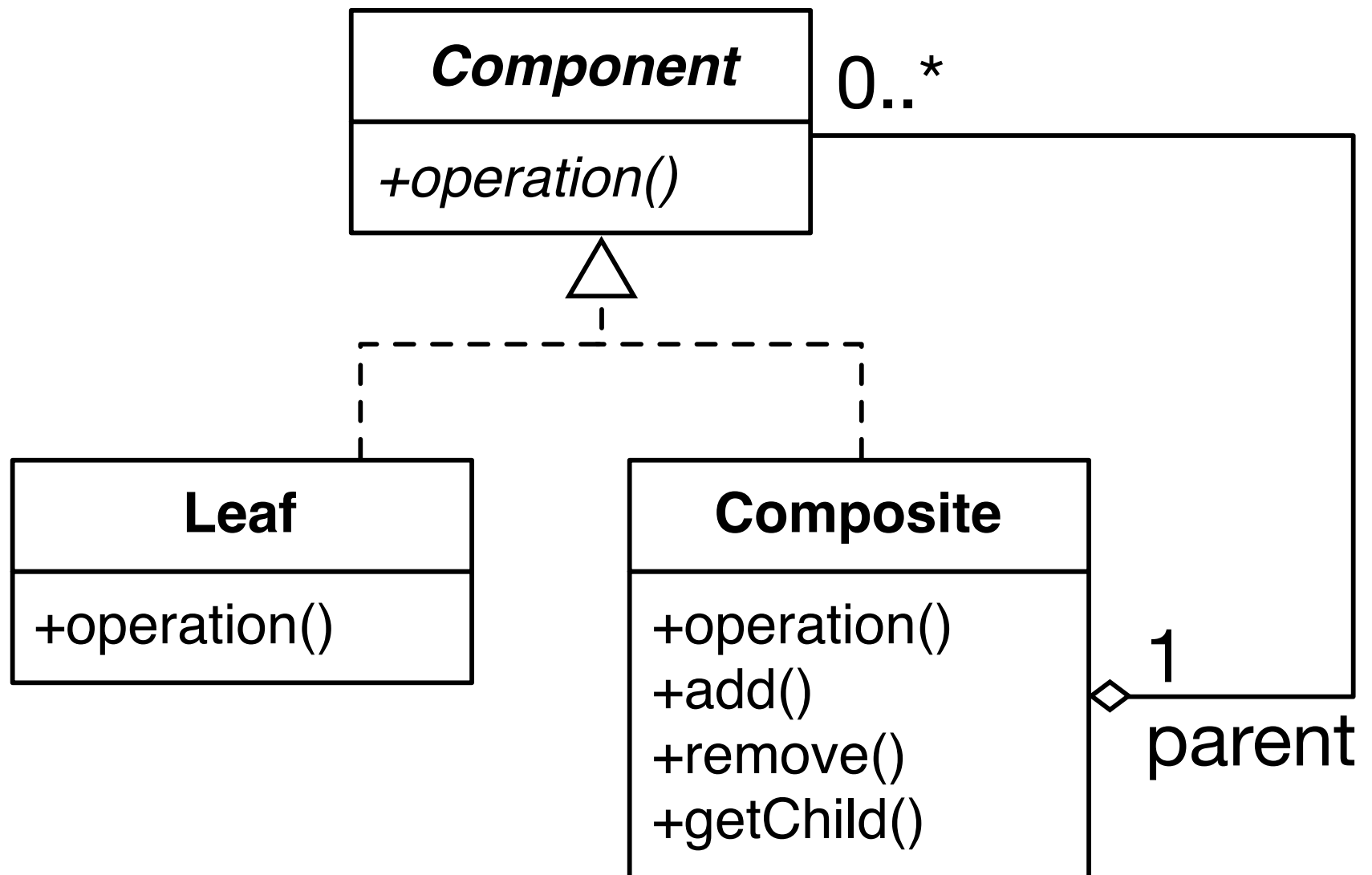
Consider a *Tree*. It consists of Trees (subtrees) and *Leaf* objects.







Using a class as root



Using an interface as root

```
public interface Component {  
    int getValue();  
}
```

```
public class Composite implements Component {  
    private List<Component> components = new ArrayList<Component>();  
  
    @Override  
    public int getValue() {  
        int s = 0;  
        for (Component c : components) s += c.getValue();  
        return s;  
    }  
  
    public void add(Component component) {  
        components.add(component);  
    }  
}
```

```
public class City implements Component {  
    private int inhabitant;  
  
    public City(int inhabitant) { this.inhabitant = inhabitant; }  
  
    @Override  
    public int getValue() { return inhabitant; }  
}
```

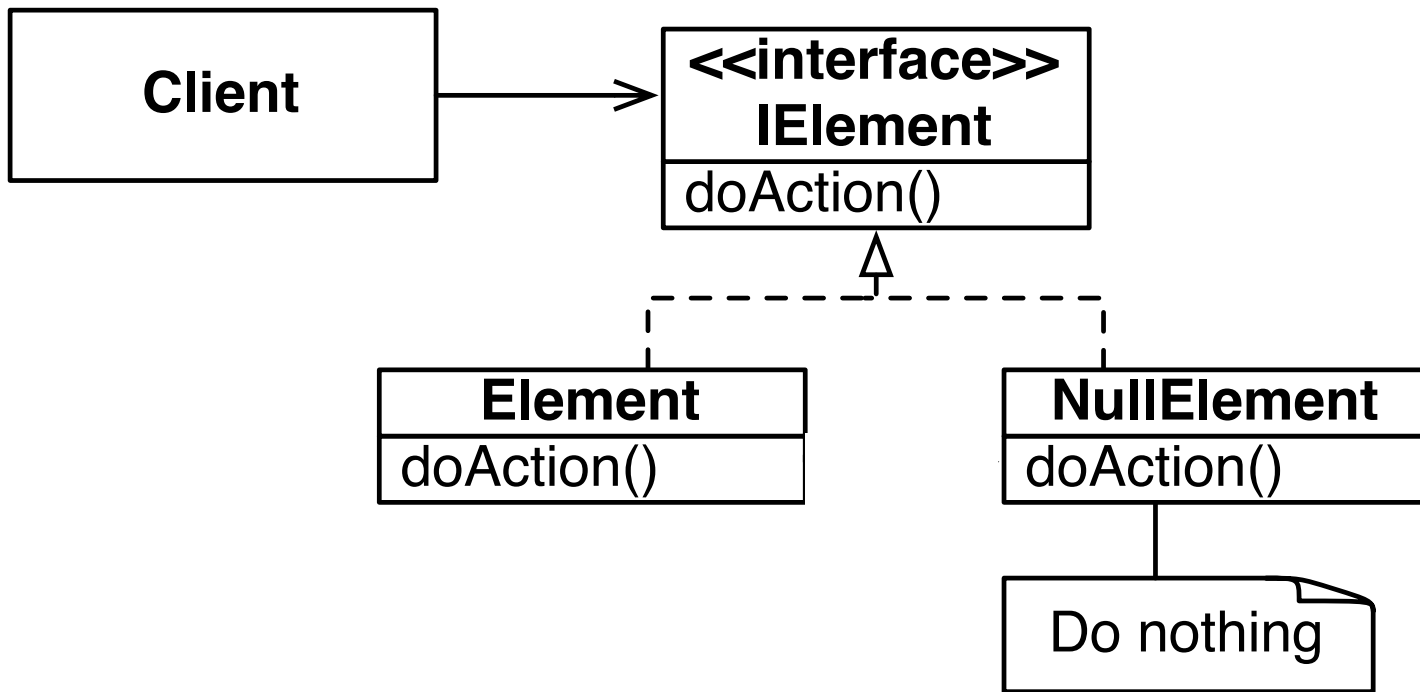
```
public class Example {  
    public static void main(String[] arg) {  
        Composite chile = new Composite();  
        City santiago = new City(6300000);  
        City serena = new City(201000);  
        City vina = new City(289000);  
  
        chile.add(santiago);  
        chile.add(serena);  
        chile.add(vina);  
  
        Composite southAmerica = new Composite();  
        southAmerica.add(chile);  
        System.out.println(southAmerica.getValue());  
    }  
}
```

Null Object Pattern

How do you avoid cluttering your code with *tests* for null object pointers?

Introduce a Null Object that implements the interface you expect, but does nothing

Null Objects may also be Singleton objects, since you never need more than one instance



Null Object

Examples

`NullOutputStream` extends `OutputStream` with an empty `write()` method

Consequences

Simplifies client code

Not worthwhile if there are only few and localized tests for null pointers

Factory Pattern - Example



Lara can have
different weapons:
bow, guns, ...

How would you make Lara fire arrows, bullets and so on?

Factory Pattern

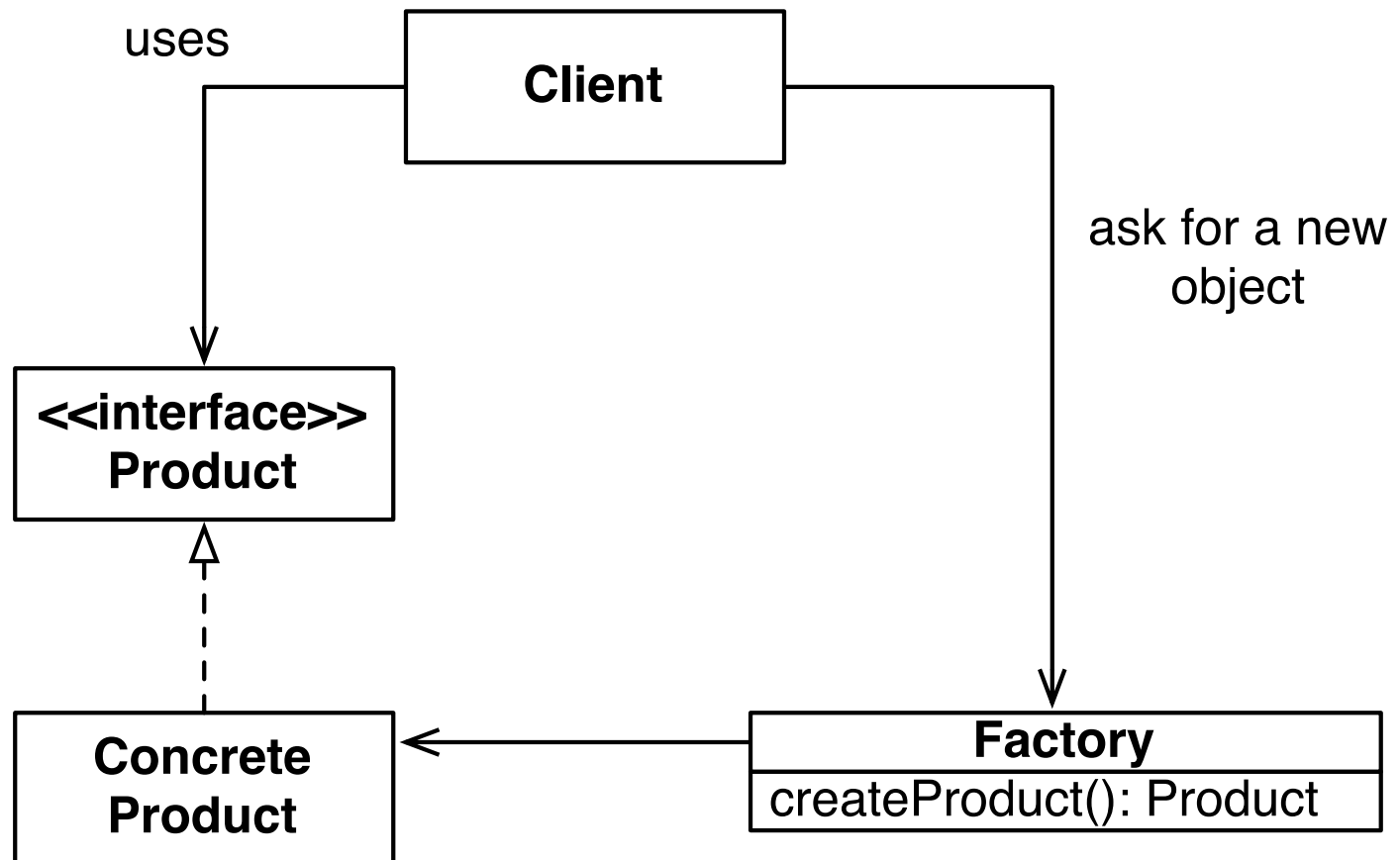
How do you externalize the creation of multiple objects?

Use a factory class to build customized objects

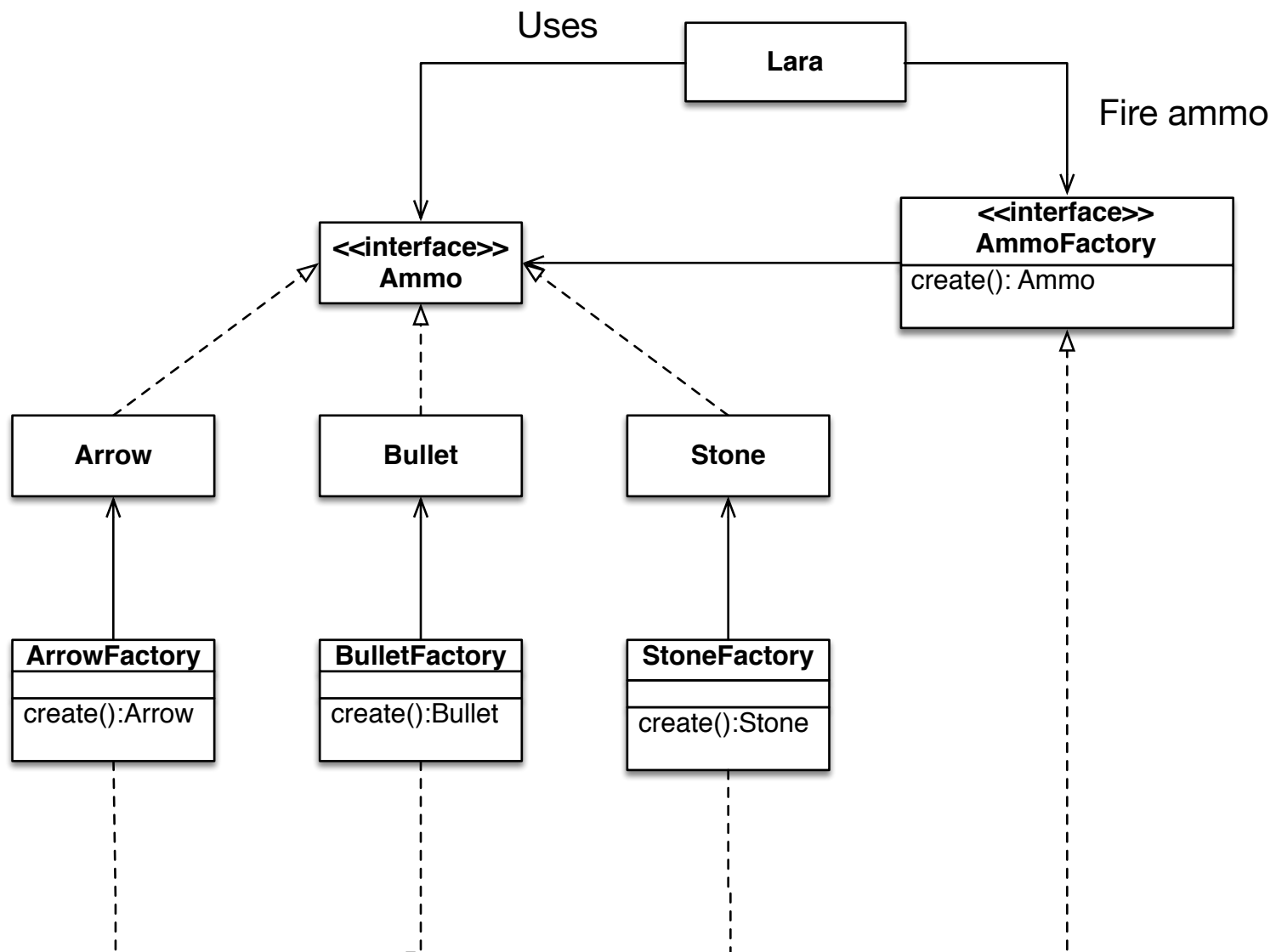
A factory creates objects without exposing the instantiation logic to the client

All the burden to initialize objects is hidden

Factory Pattern - UML



Factory Pattern - UML



Factory Pattern - Example

```
public class Lara {  
  
    private AmmoFactory ammoFactory;  
  
    public Ammo fire() {  
        return ammoFactory.create();  
    }  
  
    public void ammoFactory(AmmoFactory anAmmoFactory) {  
        ammoFactory = anAmmoFactory;  
    }  
}
```

Factory Pattern - Example

```
public class ArrowFactory implements AmmoFactory {  
    @Override  
    public Arrow create() {  
        return new Arrow();  
    }  
}
```

```
public class Arrow implements Ammo {  
}
```

```
public interface Ammo {  
}
```

```
public interface AmmoFactory {  
    Ammo create();  
}
```

Lara is attacking

```
@Test
public void test() {
    Lara lara = new Lara();
    lara.ammoFactory(new ArrowFactory());
    assertEquals(lara.fire().getClass(), Arrow.class);
}
```

A factory has to be set

Factory and Object Initialization

Having constructor accepting many arguments reduces readability

A factory may greatly simplify initialization of objects

Factory and Object Initialization

Consider the following class:

```
public class Arrow implements Ammo {  
    /* Many arguments */  
    public Arrow(int pikeSize, Color color, int arrowSize, ArrowMaterial m) {  
        ...  
    }  
    public Arrow(int arrowSize, ArrowMaterial m) {  
        this(5, Color.BROWN, arrowSize, m);  
    }  
    ...  
}
```

Factory and Object Initialization

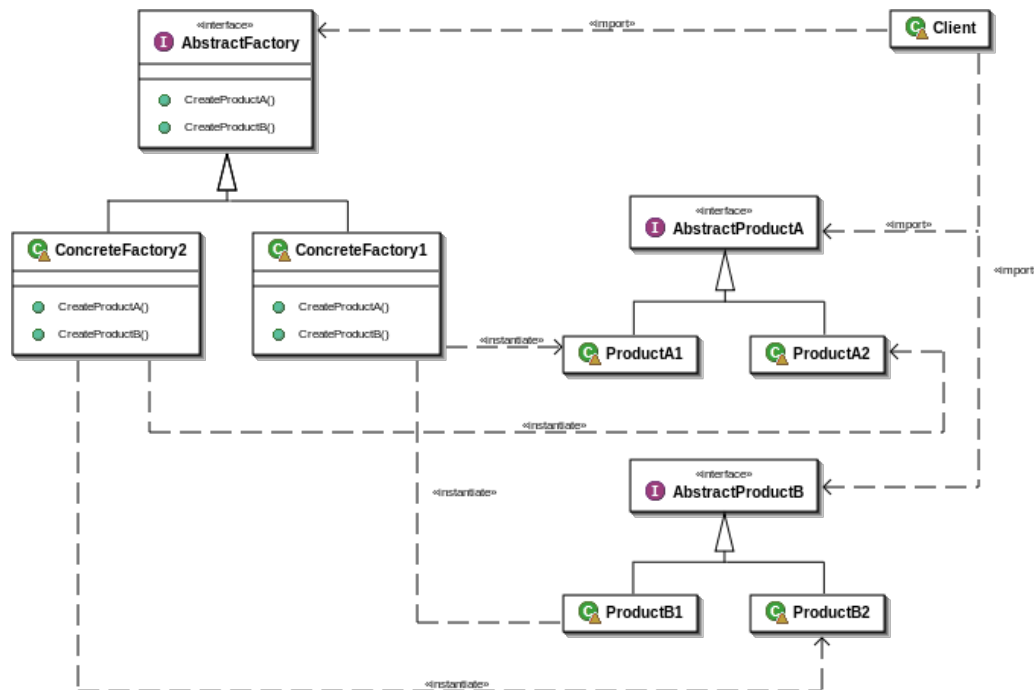
Consider the following class:

```
public class Arrow implements Ammo {  
    /* Many arguments */  
    public Arrow(int pikeSize, Color color, int arrowSize, ArrowMaterial m) {  
        ...  
    }  
    public Arrow(int arrowSize, ArrowMaterial m) {  
        this(5, Color.BROWN, arrowSize, m);  
    }  
    ...  
}
```

A factory may simplify the initialization:

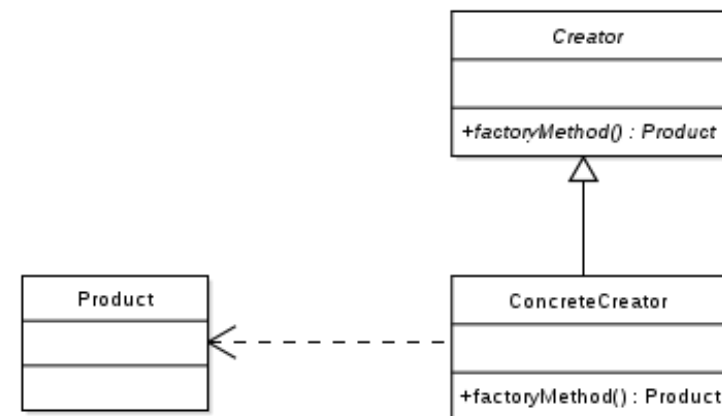
```
ArrowFactory factory = new ArrowFactory();  
factory.setSize(10);  
factory.setColor(Color.BLUE);  
factory.create();  
...
```

Terminology



Abstract Factory Pattern

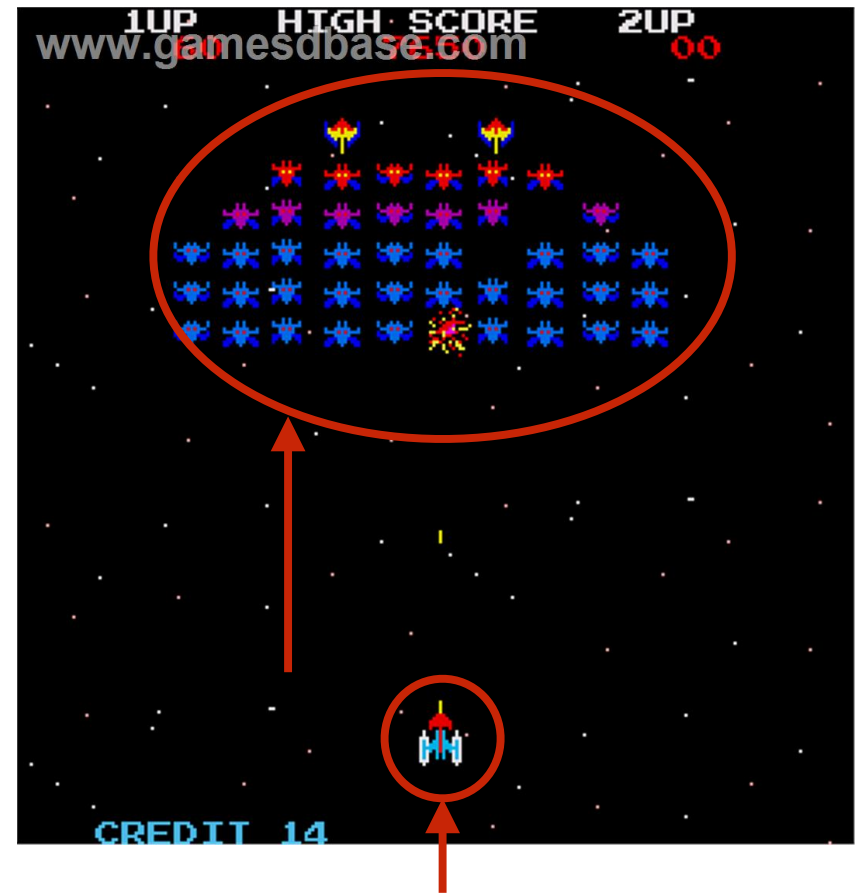
Factory Method Pattern



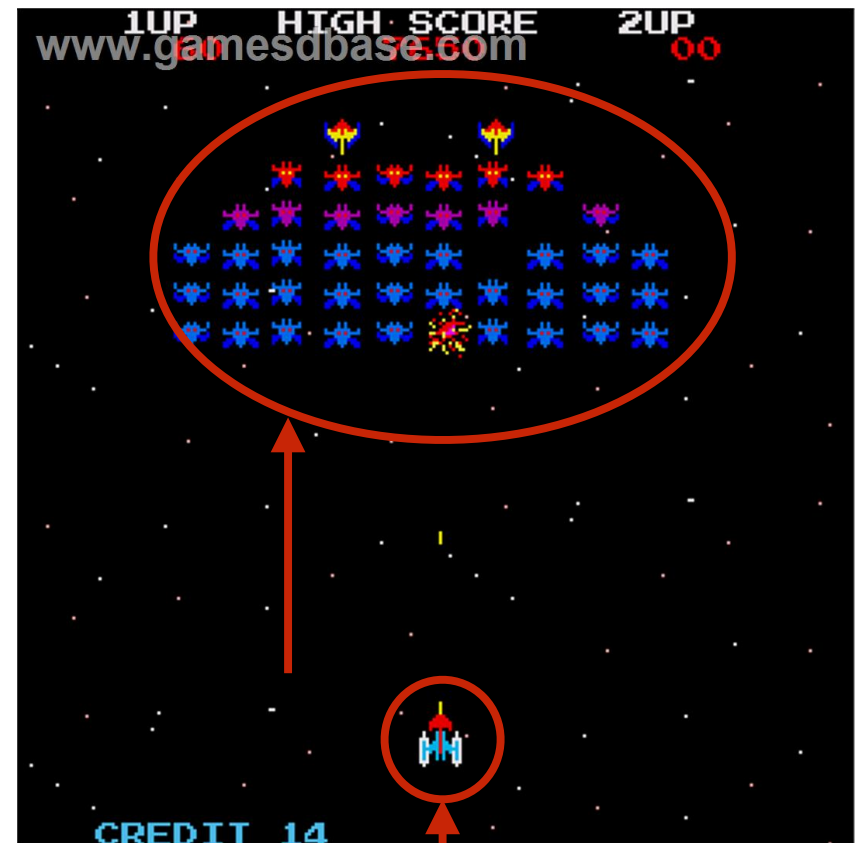
Perspectiva cultural



Perspectiva cultural



Perspectiva cultural



How to prohibit the creation of more than one object?

Singleton Pattern

How to forbid more than one instance from a particular class?

A singleton pattern makes sure no more than one instance can be obtained from a class

Has to be use with care since it introduces a global state

Singleton Pattern - Example

How many Lara Croft? No more than one

```
public class Lara {  
    private static Lara uniqueInstance;  
  
    private Lara () { }  
  
    public static Lara uniqueInstance() {  
        if(uniqueInstance == null) {  
            uniqueInstance = new Lara();  
        }  
        return uniqueInstance;  
    }  
    ...  
}
```

Singleton
- instance : Singleton = null
+ getInstance() : Singleton
- Singleton()

Flyweight Pattern

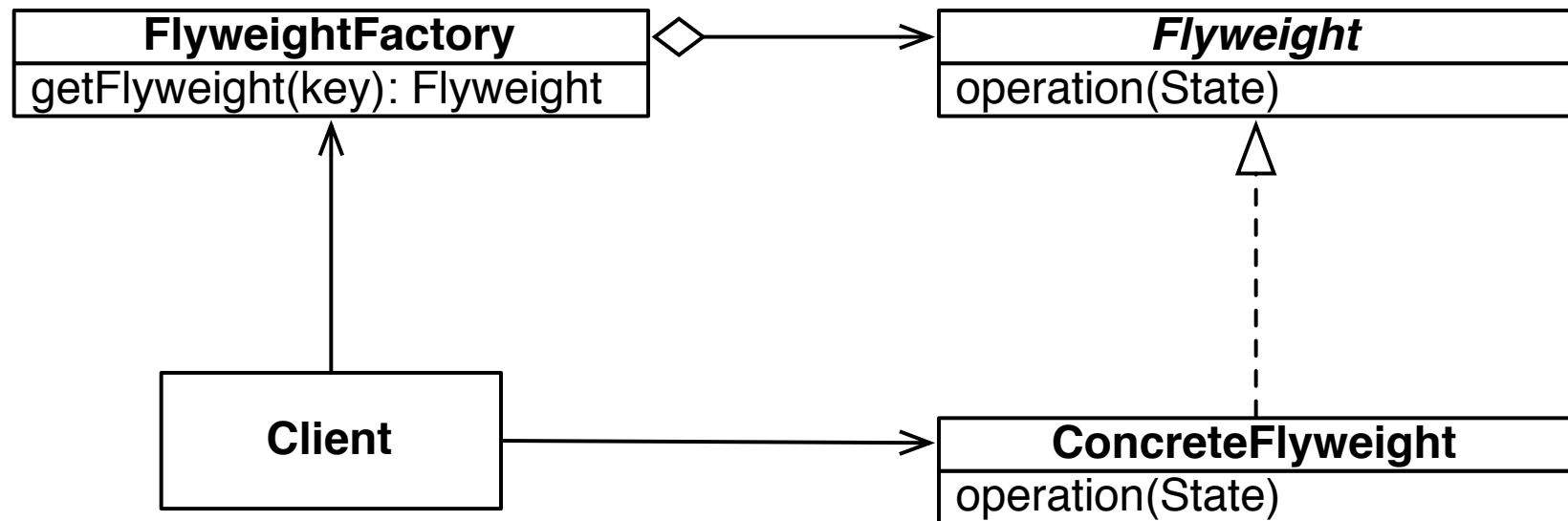
How to support a large number of individual fine-grained objects efficiently?

A flyweight pattern enables reusing objects that are *qualified* as unnecessary

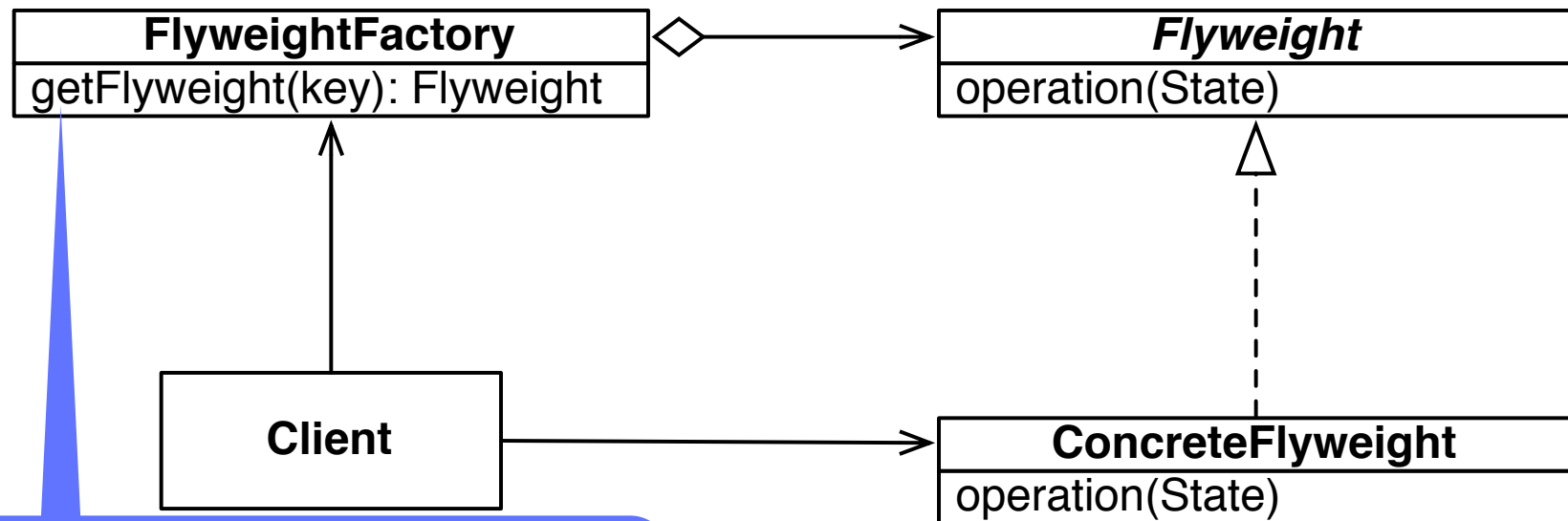
Creating many objects may be the cause of poor memory performance

Storing short living objects into a table enable one to easily reuse them

Flyweight Example - UML



Flyweight Example - UML



```
obj = hashtable.get(key);
if(obj != null)
    return obj;
else {
    obj = new ConcreteFlyweight();
    hashtable.put(key, obj);
    return obj; }
```

Example without Flyweight

```
public class ColorFactory {  
  
    public Color getColor(int red, int green, int blue) {  
        return new Color(red, green, blue);  
    }  
}  
  
public class ColorfulTest {  
    @Test  
    public void testFactory(){  
        ColorFactory f = new ColorFactory();  
        Color c1 = f.getColor(255, 0, 0);  
        Color c2 = f.getColor(255, 0, 0);  
        assertEquals(c1, c2);  
        assertTrue(c1 == c2);  
    }  
}
```

Example without Flyweight

```
public class ColorFactory {  
  
    public Color getColor(int red, int green, int blue) {  
        return new Color(red, green, blue);  
    }  
}
```

```
public class ColorfulTest {  
    @Test  
    public void testFactory(){  
        ColorFactory f = new ColorFactory();  
        Color c1 = f.getColor(255, 0, 0);  
        Color c2 = f.getColor(255, 0, 0);  
        assertEquals(c1, c2);  
        assertTrue(c1 == c2);  
    }  
}
```

junit.framework.AssertionFailedError

Example without Flyweight

```
public class ColorFactory {
```

It does not look like to be a serious problem
But if you create millions of colors, then it will be a
serious problem

```
public void testFactory() {  
    ColorFactory f = new ColorFactory();  
    Color c1 = f.getColor(255, 0, 0);  
    Color c2 = f.getColor(255, 0, 0);  
    assertEquals(c1, c2);  
    assertTrue(c1 == c2);  
}  
}
```

junit.framework.AssertionFailedError

Example with Flyweight

```
public class ColorFactory {  
    private Hashtable<List<Integer>,Color> hashtable = new Hashtable<>();  
  
    public Color getColor(int red, int green, int blue) {  
        Integer[] array = { red, green, blue };  
        List<Integer> key = Arrays.asList(array);  
        Color color = hashtable.get(key);  
        if(color != null)  
            return color;  
        else {  
            color = new Color(red, green, blue);  
            hashtable.put(key, color);  
        }  
        return color;  
    }  
}
```

Example with Flyweight (another possibility)

```
public class ColorFactory {  
    private static HashMap<Integer,Color> colorCache = new HashMap<>();  
  
    public static Color getColor(int red, int green, int blue) {  
        int key = red * 256 * 256 + green * 256 + blue;  
  
        if(colorCache.containsKey(key))  
            return colorCache.get(key);  
        else {  
            Color c = new Color(red, green, blue);  
            colorCache.put(key, c);  
            return c;  
        }  
    }  
}
```


Example with Flyweight

```
public class ColorfulTest {  
    @Test  
    public void testFactory(){  
        ColorFactory f = new ColorFactory();  
        Color c1 = f.getColor(255, 0, 0);  
        Color c2 = f.getColor(255, 0, 0);  
        assertEquals(c1, c2);  
        assertTrue(c1 == c2);  
    }  
}
```

The test now passes
The exact same object is reused

Classification

Creational patterns

abstract factory, builder, factory method, lazy initialization, singleton

Structural patterns

adapter (wrapper), bridge, composite, decorator, façade, flyweight, proxy

Behavioral patterns

command, interpreter, iterator, null object, observer, state, template method, visitor

Classification

Concurrency patterns

Active objects, balking, lock, monitor object, thread pool, scheduler, ...

These designs are specific to the concurrency domain, and do not appear in the GoF book

What you should know!

How to apply the singleton pattern on a class?

Can you give examples for which it is beneficial to use a flyweight pattern?

Can you answer these questions?

How can a Template Method help to eliminate duplicated code?

When do I use a Composite Pattern? Do you know any examples from the Frameworks you know?

How would you adapt the Singleton pattern to produce not more than a particular number of instances?

Can you describe situations where the Flyweight pattern is not appropriate?

What is the effect of the Flyweight on the garbage collector?

License



Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- Share: copy and redistribute the material in any medium or format
- Adapt: remix, transform, and build upon the material for any purpose, even commercially

The licensor cannot revoke these freedoms as long as you follow the license terms



Attribution: you must give appropriate credit



ShareAlike: if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original

Complete license: <https://creativecommons.org/licenses/by-sa/4.0/>