Design Patterns

Charlotte Pierce

SWINBURNE UNIVERSITY OF TECHNOLOGY

Many problems have been solved before

Many solutions share a common need

Over years programmers have identified general classes of problems to solve

Use Design Patterns to incorporate best practice into your OO design

Design patterns provide optimised, reusable templates to solve classes of problems

Structural Design Patterns deal with relationships between objects, making it easier for these entities to work together.

Creational patterns provide instantiation mechanisms, making it easier to create objects in a way that suits the situation.

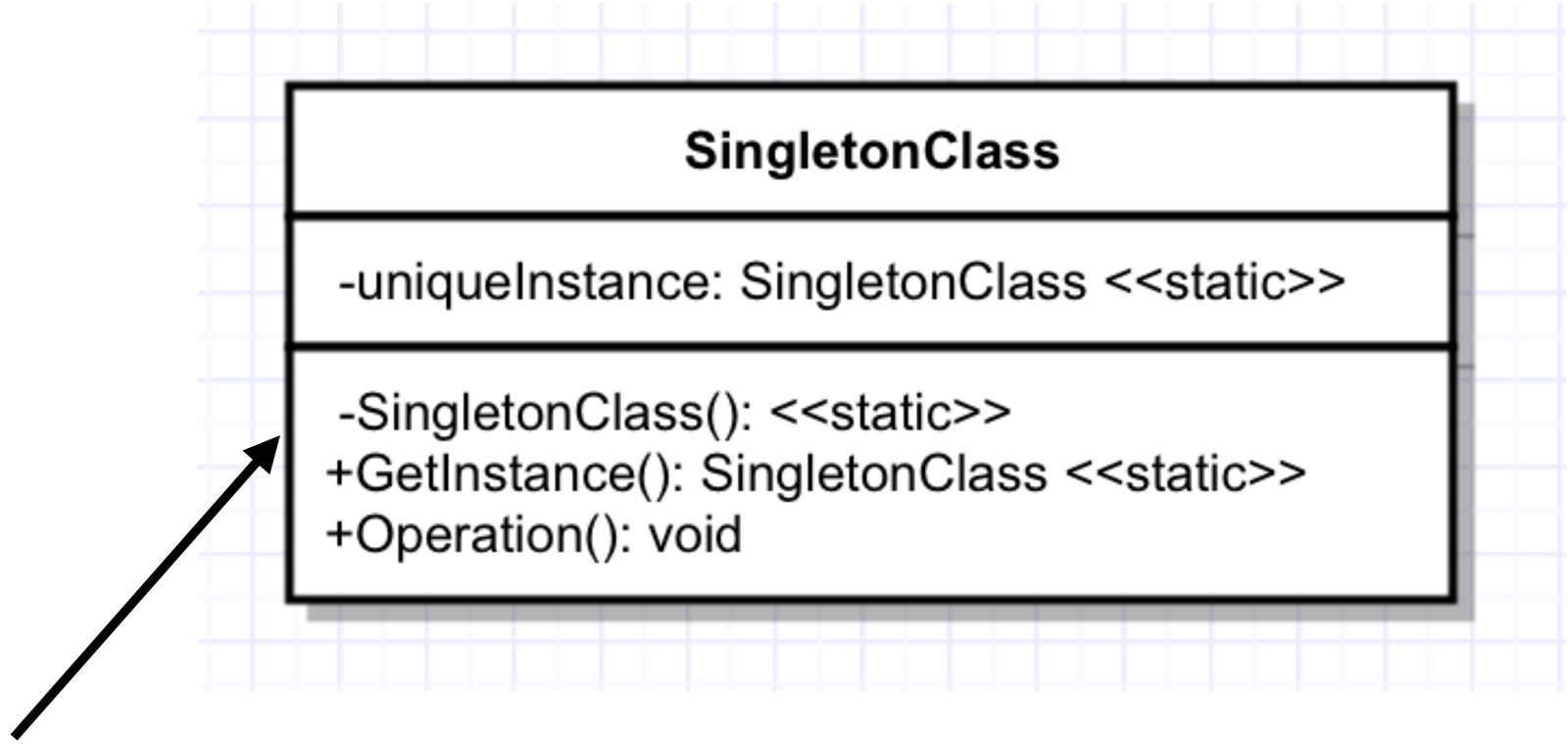
Behavioural patterns dictate communications between objects, increasing the ease and flexibility of object communication.

Structural Design Patterns

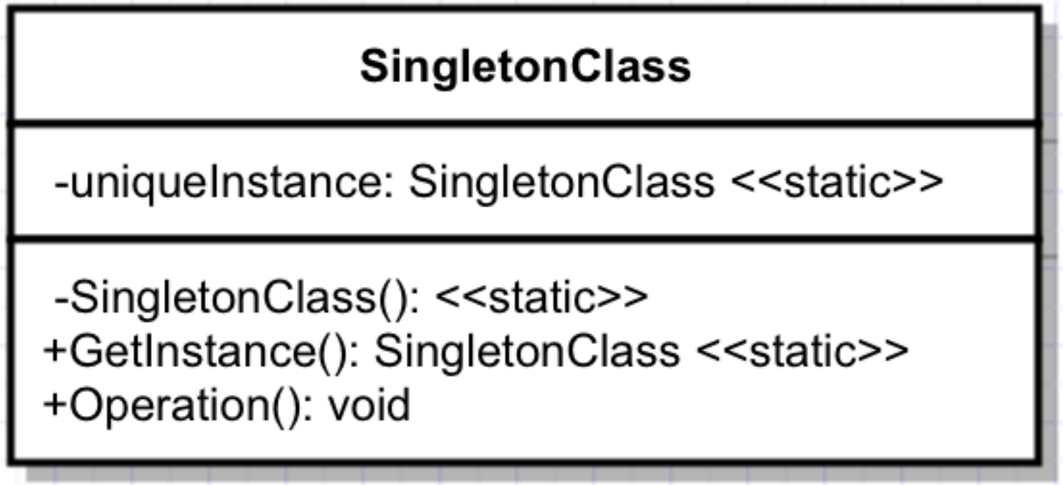
You want a class to be the global point of coordination.

You only want one instance of it.

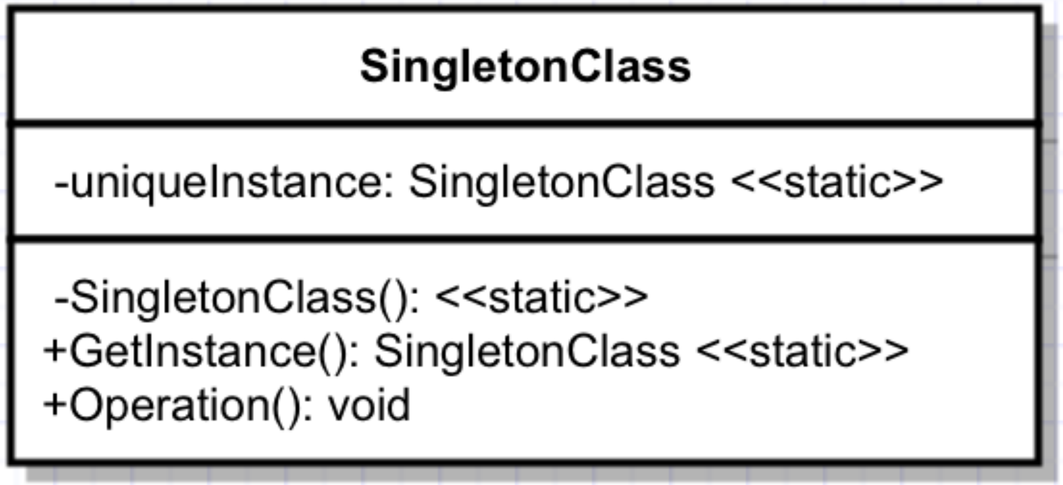
Use the Singleton Design Pattern to create a single object to rule them all



Private constructor disallows direct instantiation of class externally




```
public class SingletonClass
{
    private static SingletonClass instance;

    private SingletonClass()
    {
            // init class
    }

    public static  SingletonClass GetInstance()
    {
        if (instance == null)
            instance = new SingletonClass();

        return instance;
    }

    public void Operate()
    {
        //perform SingletonClass task
    }
}
```




```
public class SingletonClass
{
    private static SingletonClass instance;

    private SingletonClass()
    {
            // init class
    }

    public static  SingletonClass GetInstance()
    {
        if (instance == null)
            instance = new SingletonClass();

        return instance;
    }

    public void Operate()
    {
        //perform SingletonClass task
    }
}
```

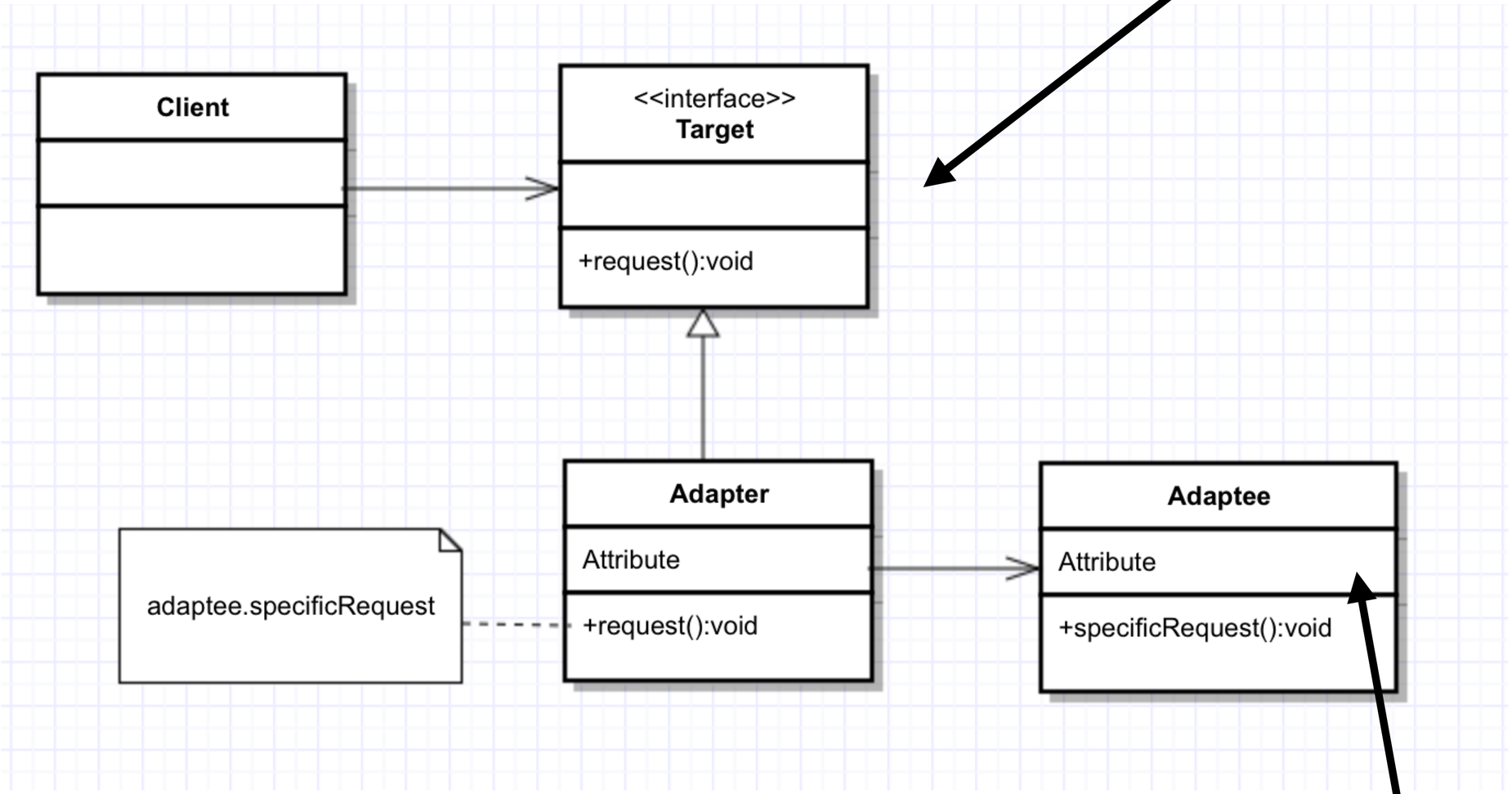
SingletonClass.GetInstance().Operate()

You want the functionality of a particular class

But you have a different interface.

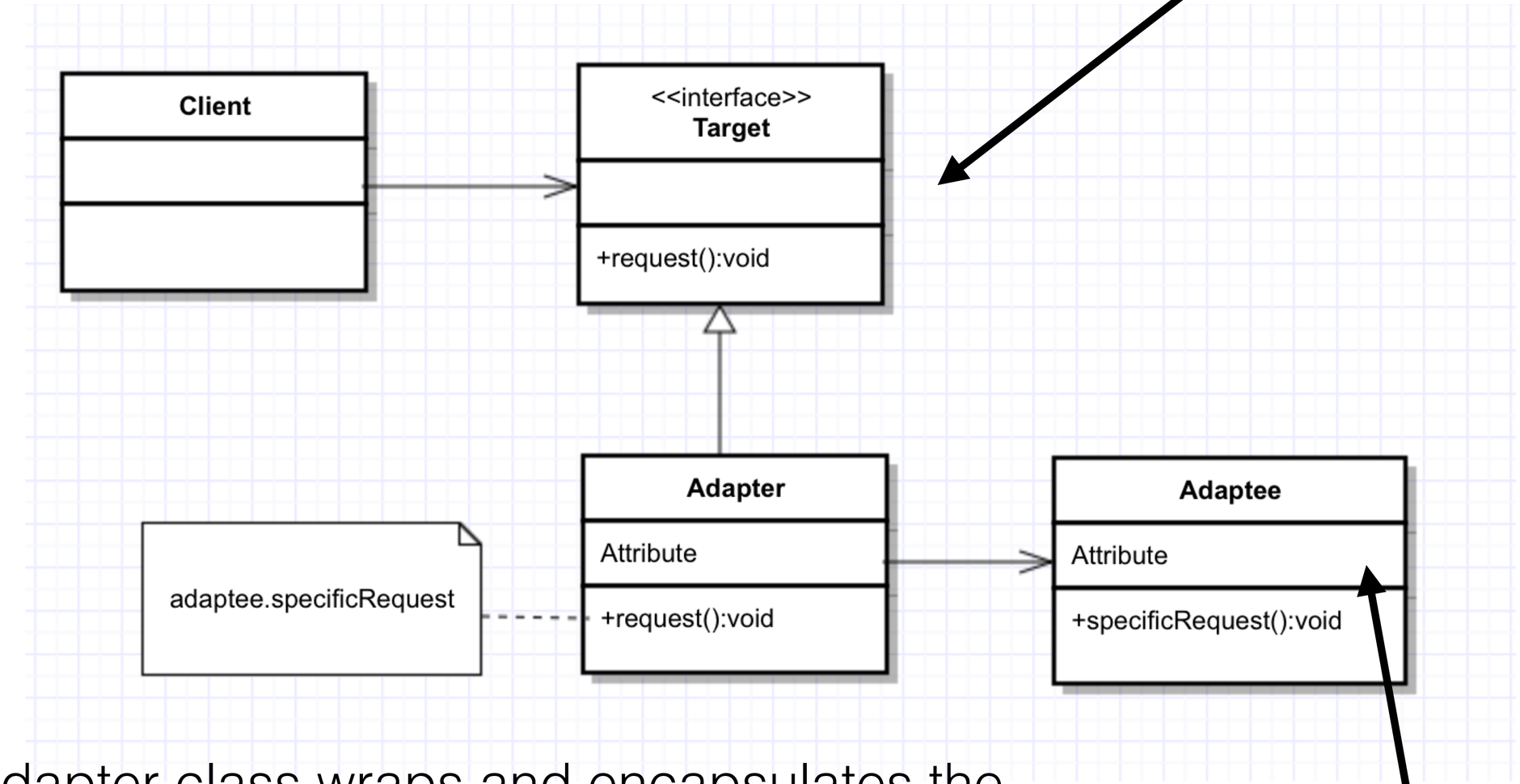
Repurpose a class with a new interface using the Adapter Pattern

Domain specific interface



Existing interface

Domain specific interface

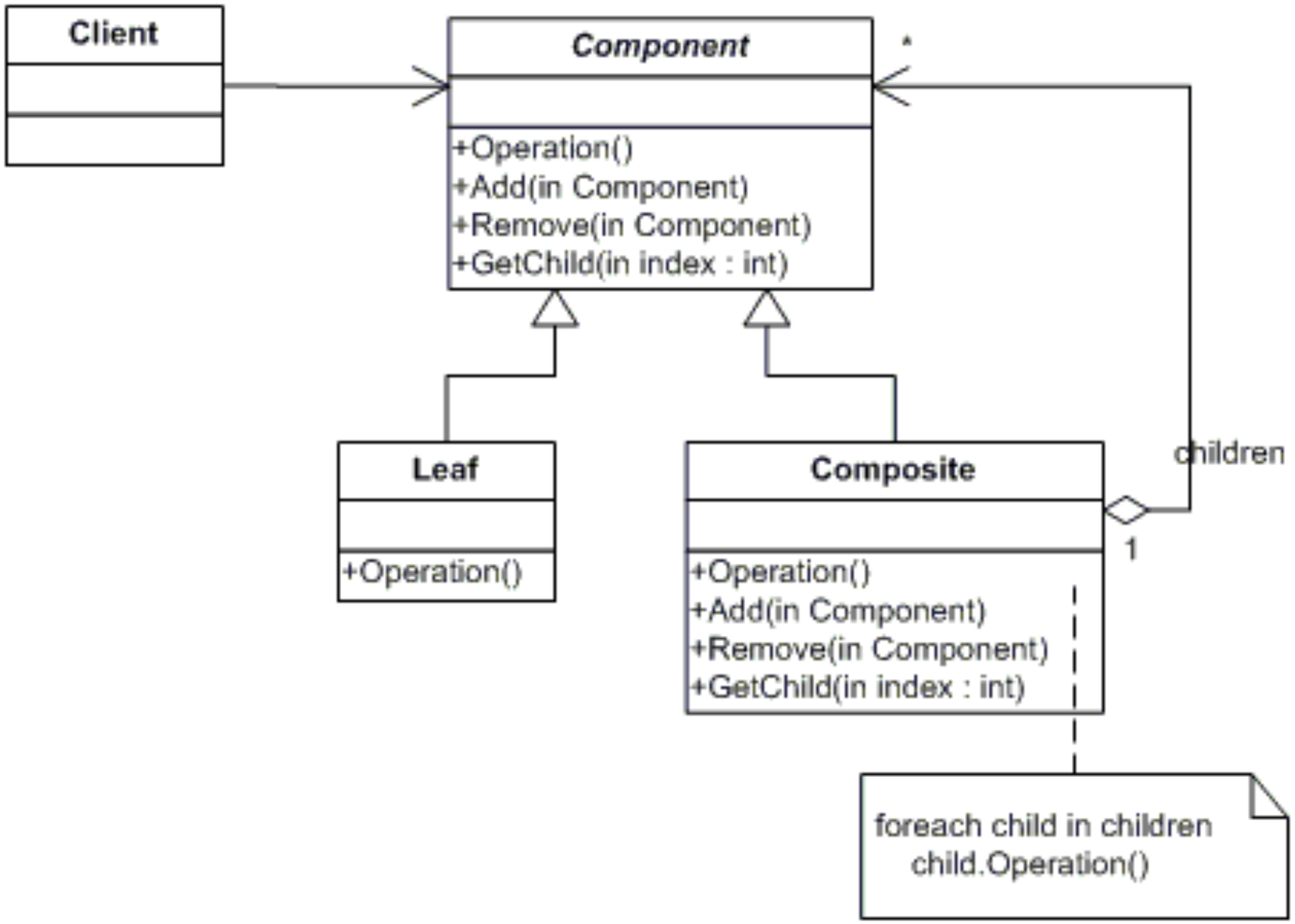


The Adapter class wraps and encapsulates the relevant methods of the adaptee

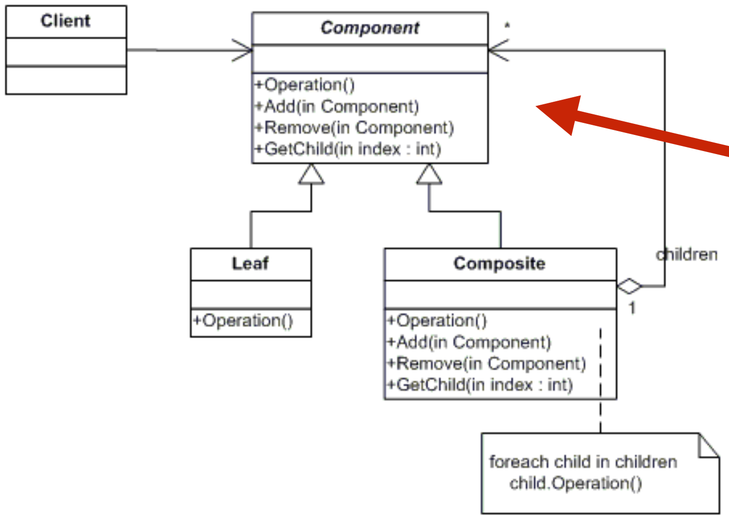
Existing interface

You want the benefits of polymorphism, but don’t want the rigidity of a deep inheritance hierarchy

Composite design patterns provide shallow inheritance and flexible class extension

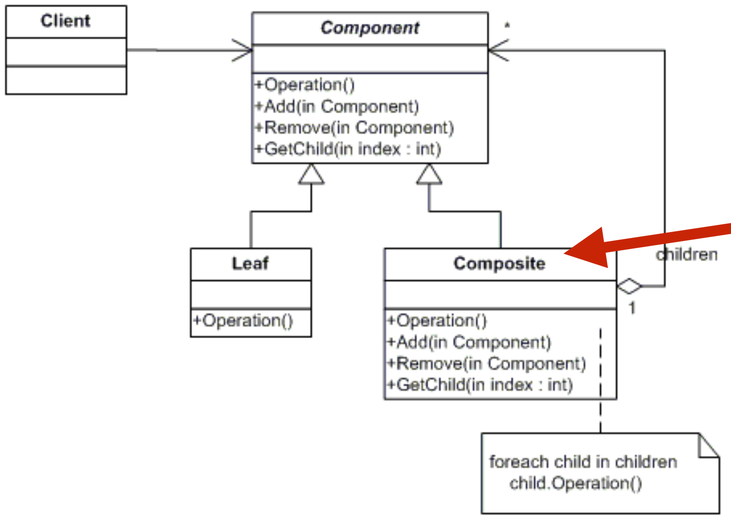


http://www.dofactory.com/net/composite-design-pattern



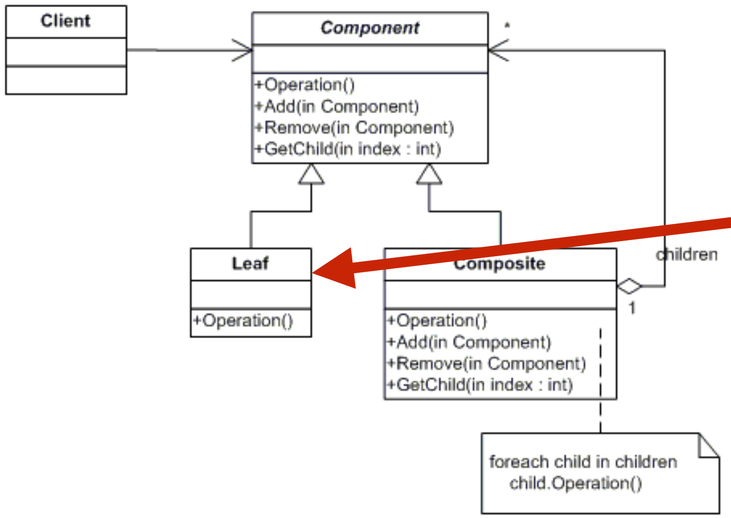

```
abstract class Component
{
    protected string name;

    // Constructor
    public Component(string name)
    {
        this.name = name;
    }

    public abstract void Add(Component component);
    public abstract void Remove(Component component);
    public abstract void Display(int depth);
}
```

http://www.dofactory.com/net/composite-design-pattern




```
class Composite : Component
{
    private List<Component> _children = new List<Component>();

    // Constructor
    public Composite(string name): base(name)
    {
    }

    public override void Add(Component component)
    {
        _children.Add(component);
    }

    public override void Remove(Component component)
    {
        _children.Remove(component);
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);

        // Recursively display child nodes
        foreach (Component component in _children)
        {
            component.Display(depth + 2);
        }
    }
}
```

http://www.dofactory.com/net/composite-design-pattern



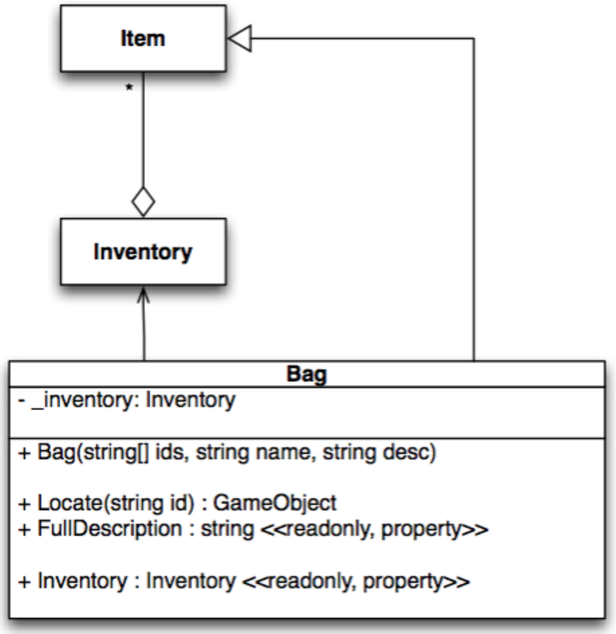

```
class Leaf : Component
{
    // Constructor
    public Leaf(string name)
      : base(name)
    {
    }

    public override void Add(Component c)
    {
        Console.WriteLine("Cannot add to a leaf");
    }

    public override void Remove(Component c)
    {
        Console.WriteLine("Cannot remove from a leaf");
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
    }
}
```

http://www.dofactory.com/net/composite-design-pattern

You have seen composition before

Iteration 3 - Bags

In this iteration you will add a Bag class to make it possible to have items that contain other items.



The Bag abstraction is a special kind of item, one that contains other items in its own Inventory. This is a version of the **composite pattern**, which allows flexible arrangements of bags and items, for example a bag to contain another bag.

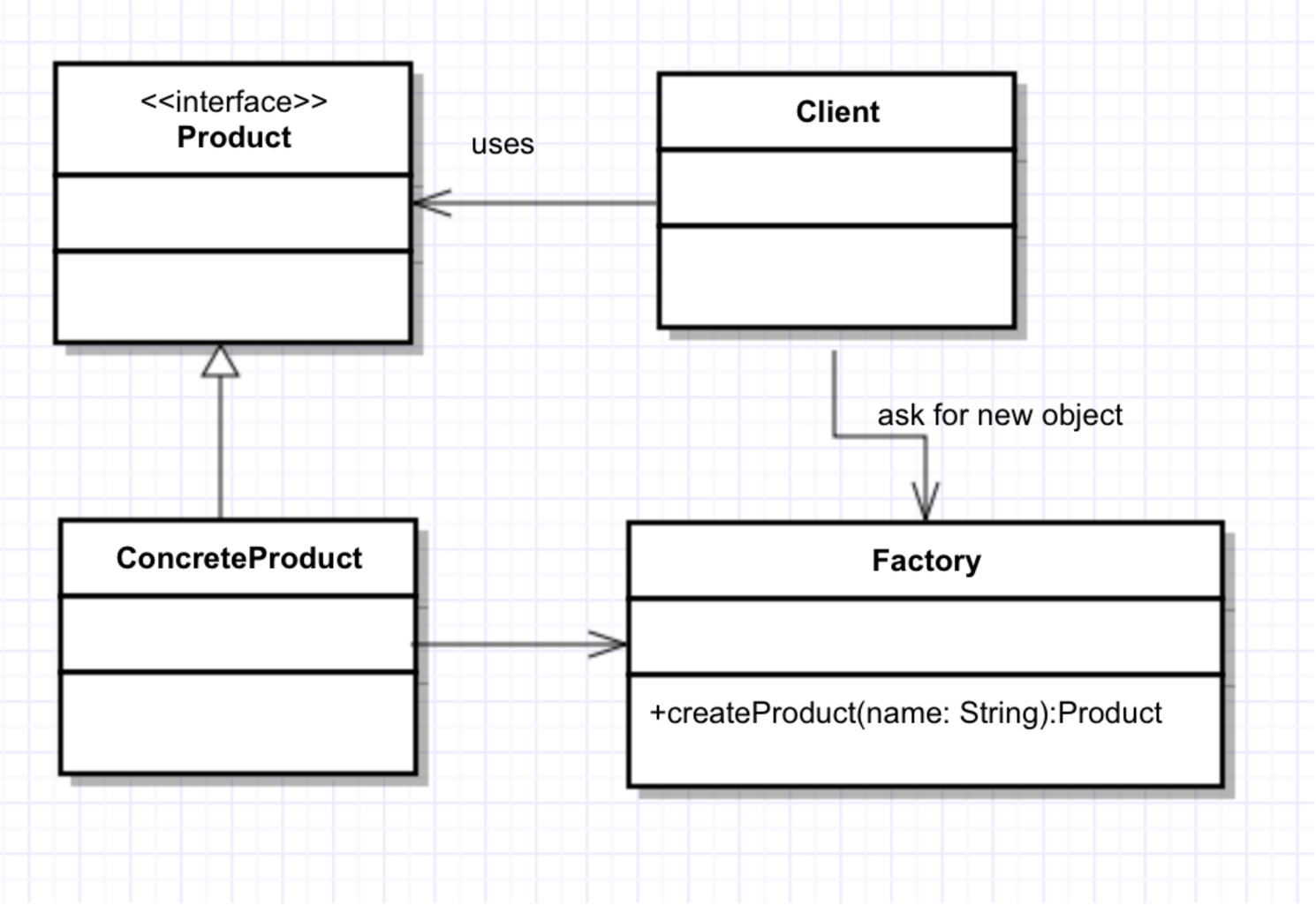
Creational Patterns

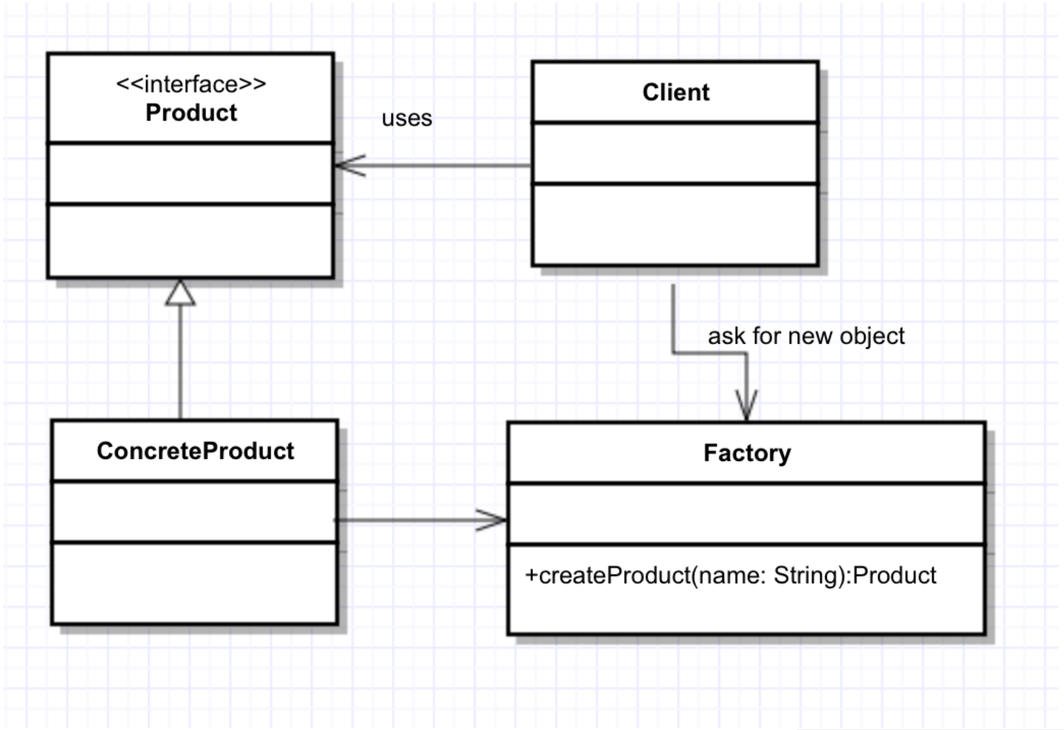
You want to create multiple variations of a particular class.

You don’t want to expose the instantiation logic

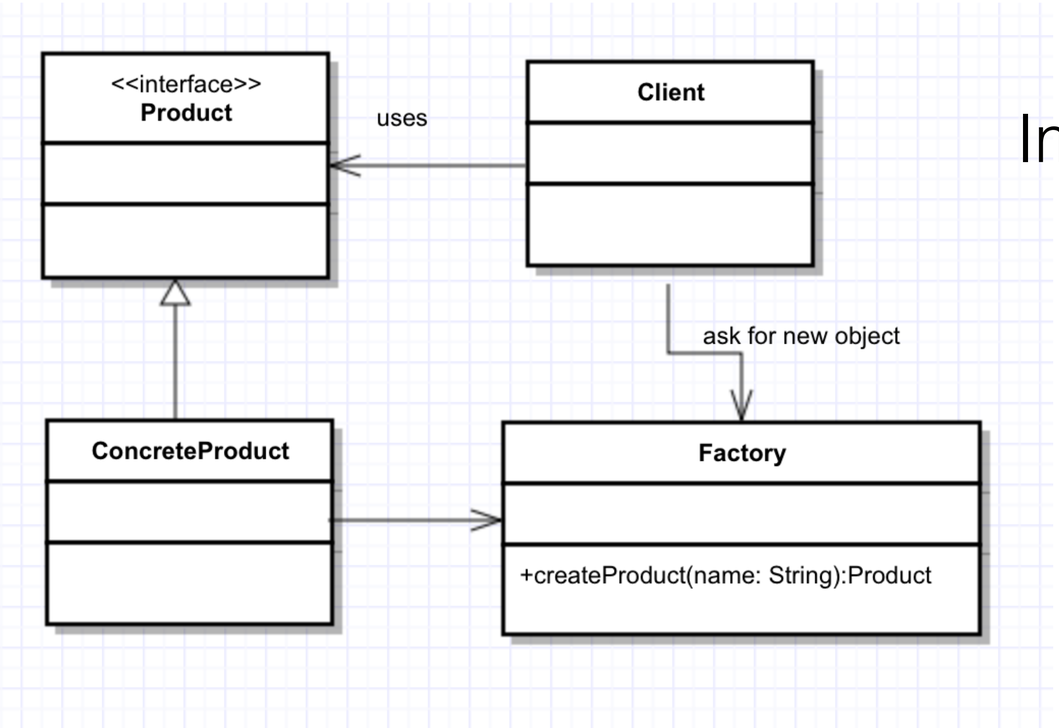
Use Factory Patterns to encapsulate the generation of object instances

Image source: http://www.lankanewspappers.com


<<interface>>
Product
uses
Client
ask for new object
ConcreteProduct
Factory
+createProduct(name: String):Product




```
public class ProductFactory{
        public Product createProduct(String ProductID){
                if (id==ID1)
                        return new OneProduct();
                if (id==ID2) return
                        return new AnotherProduct();
                ... // so on for the other Ids

        return null; //if the id doesn't have any of the expected values
    }
    ...
}
```



Improve this further using class registration in a Dictionary

```
public class ProductFactory{
        public Product createProduct(String ProductID){
                if (id==ID1)
                        return new OneProduct();
                if (id==ID2) return
                        return new AnotherProduct();
                ... // so on for the other Ids

        return null; //if the id doesn't have any of the expected values
    }
    ...
}
```

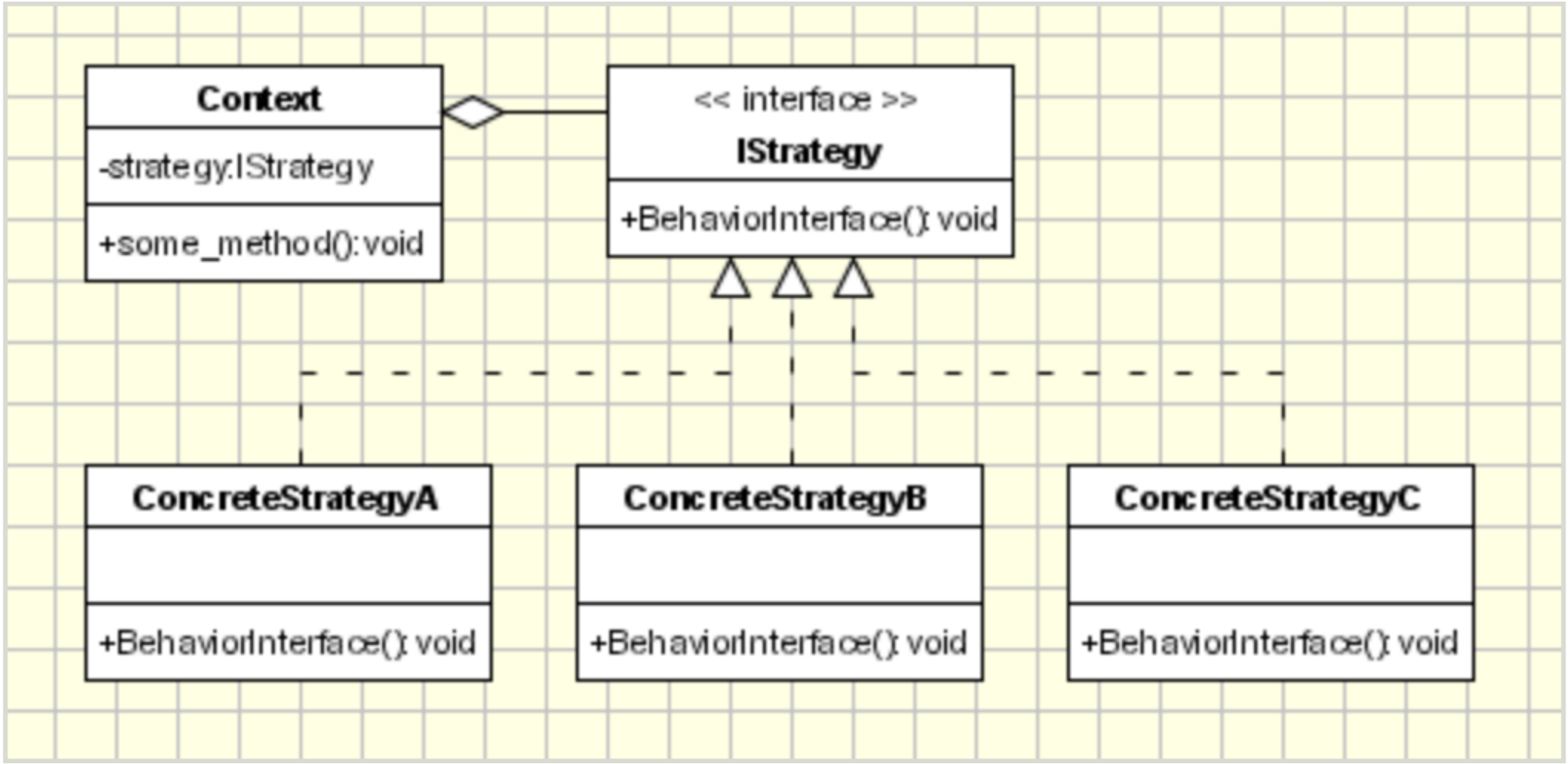
Behavioural Patterns

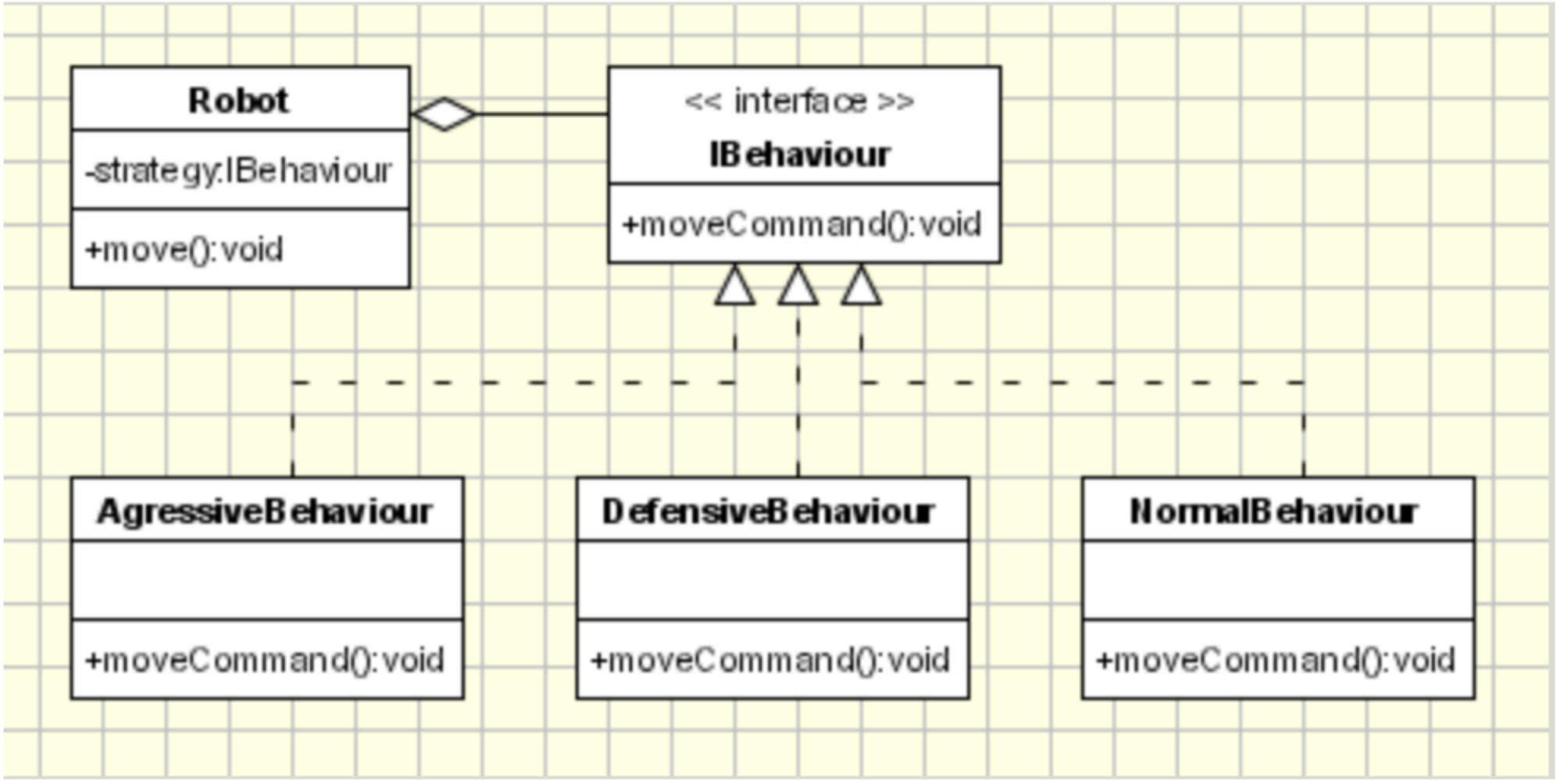
Your plan of attack depends on the current situation

Use a Strategy Pattern to organise and encapsulate your different approaches

The Strategy Pattern is a behavioural design pattern.

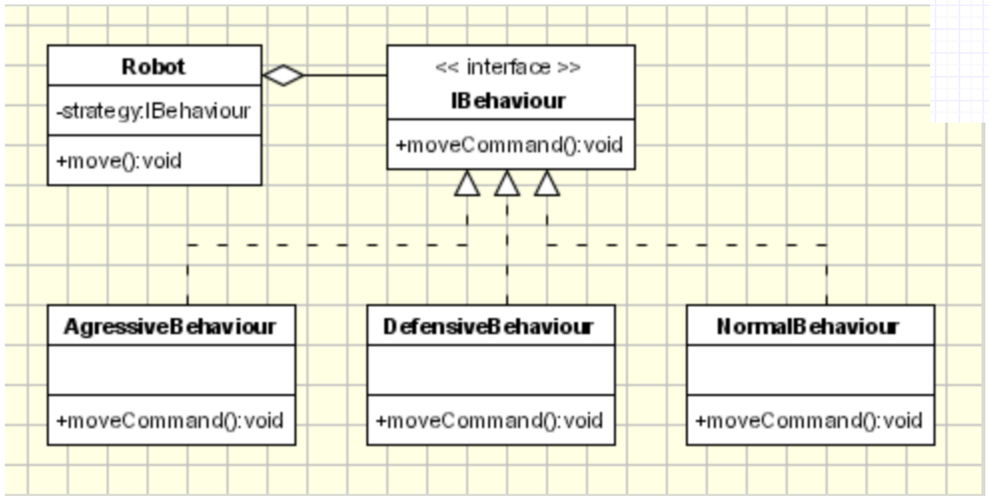
Behavioural Patterns are concerned with the communication between objects at runtime

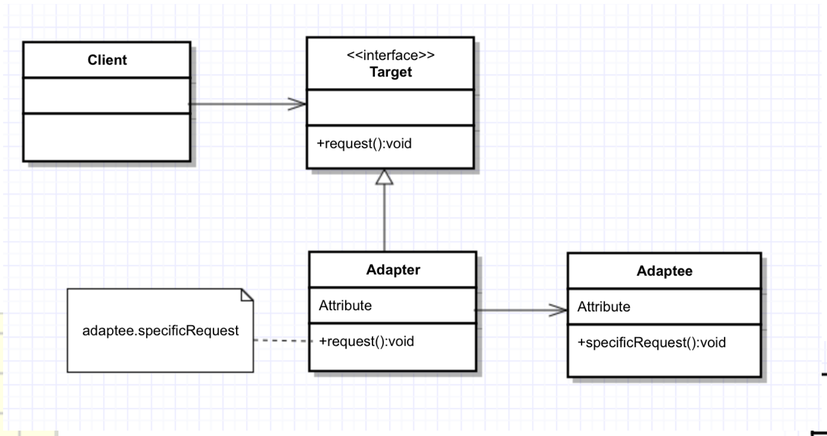

Context
-strategy:IStrategy
+some_method():void
<< interface >>
IStrategy
+BehaviorInterface(): void
ConcreteStrategyA
+BehaviorInterface(): void
ConcreteStrategyB
+BehaviorInterface(): void
ConcreteStrategyC
+BehaviorInterface(): void

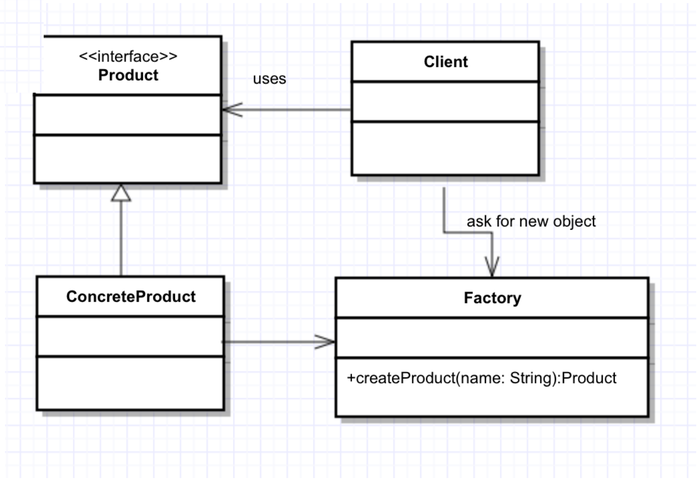

Robot
-strategy:IBehaviour
+move():void
<< interface >>
IBehaviour
+moveCommand():void
AgressiveBehaviour
+moveCommand():void
DefensiveBehaviour
+moveCommand():void
NormalBehaviour
+moveCommand():void

Design Patterns often promote composition over inheritance

System behaviours are composed via Interfaces.. not inheritance







Choose the Design Pattern that best fits, and if needed, adapt it.

Don’t make implementing the pattern distract from solving the problem

Explore Design Patterns and see how they might help

Use Design Patterns to communicate your design

Look for the common problems to solve

Design Patterns may help