Design Patterns

Programmeertechnieken, Tim Cocx

Universiteit
Leiden
The Netherlands

A nice interface example

```
public class datastructure {
    public void temp(){
        Collection<Object> someCollection = new LinkedList<>();

        someCollection.add(new Object());
        someCollection.add(new Object());

        Iterator itr = someCollection.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next()); //here as well: nice abstraction!
        }
    }
}
```

A nice interface example

```
public class datastructure {
   public void temp(){
      Collection<Object> someCollection = new TreeSet<>();

      someCollection.add(new Object());
      someCollection.add(new Object());

      Iterator itr = someCollection.iterator();
      while(itr.hasNext()){
         System.out.println(itr.next()); //here as well: nice abstraction!
      }
   }
}
```

A nice interface example

```
public class datastructure {
    public void temp(){
        Collection<Object> someCollection = new HashSet<>();

        someCollection.add(new Object());
        someCollection.add(new Object());

        Iterator itr = someCollection.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next()); //here as well: nice abstraction!
        }
    }
}
```

Definition and purpose of design patterns

Definition

- **Design pattern** is a general reusable solution to a commonly occurring problem in software design.
- It is a description or template for how to solve a problem that can be used in many different situations.
- A design pattern is not a finished design that can be transformed directly into code.

Purpose:

Reuse of previously invented solutions (not specific code)
Establishing common terminology to improve communication between developers.
Improving maintainability, reusability and changability

Pitfalls

Design patterns are not a goal in itself
Increases the complexity.

Classification of Design Patterns

Creational patterns	Structural patterns	Behavioral patterns
Creates objects; Abstracts the Instantiation process	Composes classes or objects in larger structures	Concerns about communication between objects; distribution of responsibilities
Abstract Factory	Adapter	Command
Builder	Bridge	Interpreter
Factory Method	Composite	Iterator
Singleton	Decorator	Observer
etc	Facade	Strategy
	etc	Template (Method)
		etc

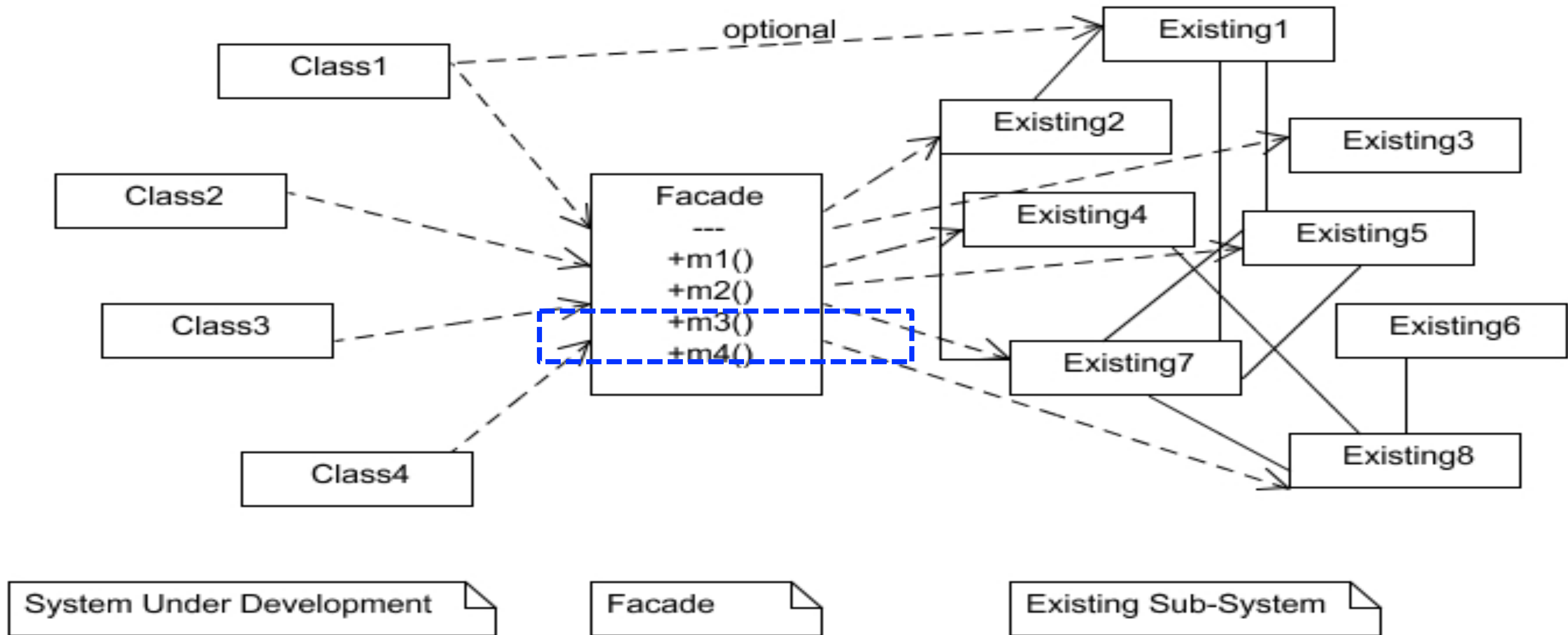
Principles and strategies for design patterns

- **Open-Closed Principle**
  - Modules, classes and methods should be open for extension, but closed for modification
  - Design your software so, that you can extend its capabilities without changing it.
- **Design From Context**
  - First create the big picture, before designing the details
  - First design the abstraction (interface, abstract class), then the implementation of the subclasses (details = specialization).
- **Dependency Inversion Principle**
  - High level modules depend on abstractions (*not on low level modules*)
  - Details (low level modules) depend on abstractions (*abstractions should not depend on details - low level modules*)

Pattern Description - Template

item		
Name		Unique name
	classification	[creational, structural, behavioral]
Intent		The purpose of the pattern
Problem		The problem that the pattern is trying to solve
	context	and the **context** in which it shows up
Solution		Description of the solution
	participants	The objects/classes participating in the pattern
	collaborators	how the classes/objects collaborate
	responsibilities	and their responsibilties
Consequences		The effects that the use of the pattern may have. The **good** and the **bad** effects. They are necessary for evaluating design alternatives and for understanding the cost and the benefits of applying the pattern
Generic structure		A standard diagram that shows the typical structure for the pattern
Implementation		How the pattern can be implemented (example)

Facade pattern



Participants: System under development, Facade, Existing sybsystem

Facade pattern

Name	Facade pattern (structural pattern)
Intent	You want to simplify the complexity of an existing system. You need to define your own interface.
Problem	You need to use only a subset of an existing complex system. or: You need to interact with the system in a particular way or: You don’t want (have the time) to learn the whole system.
Solution	The Facade presents a new (easier) interface for the existing system, so the client of the existing system, or a new client/system, can use this new interface.
Consequences	The facade simplifies the use of the required subsystem. However, because the facade is not complete, certain functionality may be unavailable for the client.

Façade Pattern

Use the façade pattern when:

- You don’t need to use all the functionality of a complex system.
  - Create a new class that contains all the rules for accessing that subsystem.
  - This API (interface) must be much simpler then the original system.

- You want to encapsulate or hide the original system.

- You want to use the functionality of the original system and want to add some new functionality as well.

- The cost of writing the new class is less than the cost of everybody learning the original system.

Facade

```
public class Facade {
  //declarations of the necessary objects of the existing system
  //Facade class will be written by "experts".
    public void m1() {
    //all calls to the existing system
    }
    public String m2(){
    //all calls to the existing system
        return rval;
    }
}
```

Facade

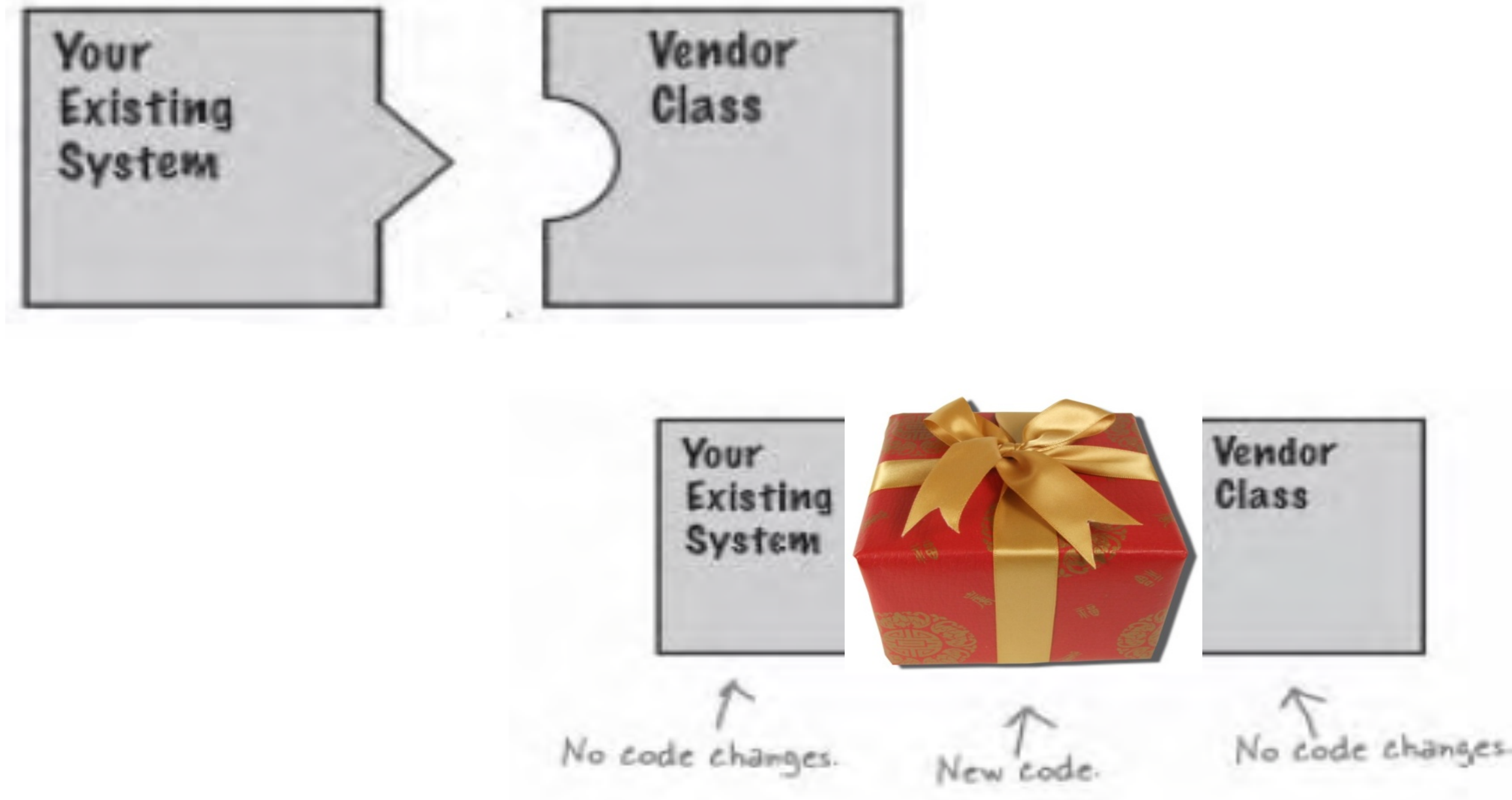
//An object of the class Facade is already available for the client

```
Facade facade = new Facade();
```

//or

```
import Facade;


public class Client1 {
  …
  facade.m1();
  System.out.println (facade.m2());
}
```

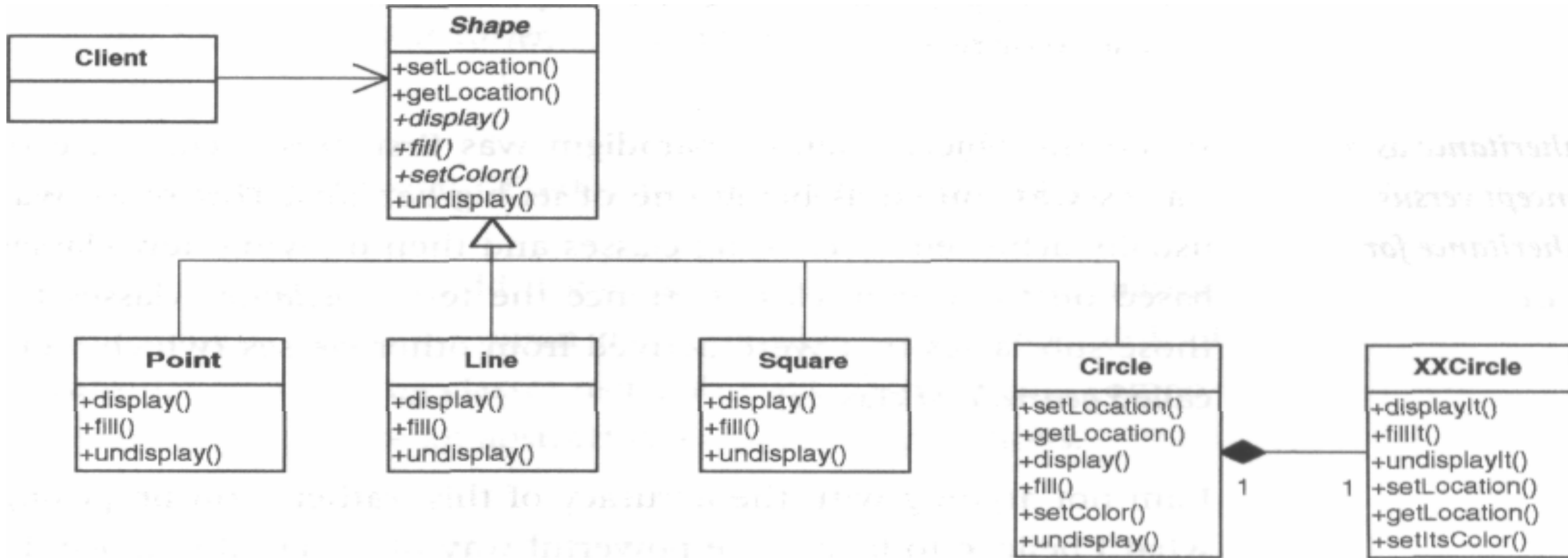
Adapter Pattern



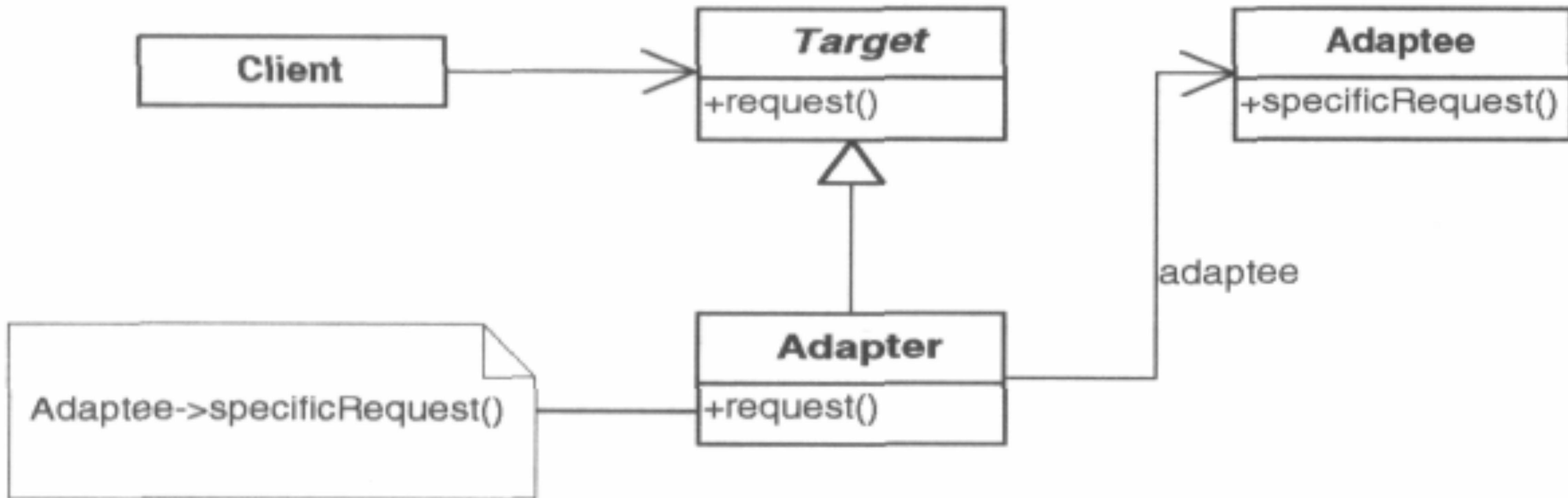
Adapter pattern

Name	**Adapter pattern (structural pattern)**
Intent	Changing an interface of an existing object. Matching an existing object (beyond your control) to a particular interface
Problem	A system has the right **data and behavior** but the wrong interface. You want to **reuse the behavior** of a class, but the class hasn't the right interface. Typically used when you want to make a class a derivative of an abstract class
Solution	The Adapter provides a wrapper with the desired interface. The adapter converts the interface of a class into another interface the clients expect. Adapter lets classes cooperate, that couldn't otherwise due to incompatible interfaces.
Consequences	Reuse of existing behavior. The adapter allows for existing objects to fit into new class structures without being limited by their interfaces. Needed functionality not present in the existing class, must be implemented in the adapter. Considering the use of an adapter against the savings by reusing the "adaptee".

Adapter



(Object) Adapter



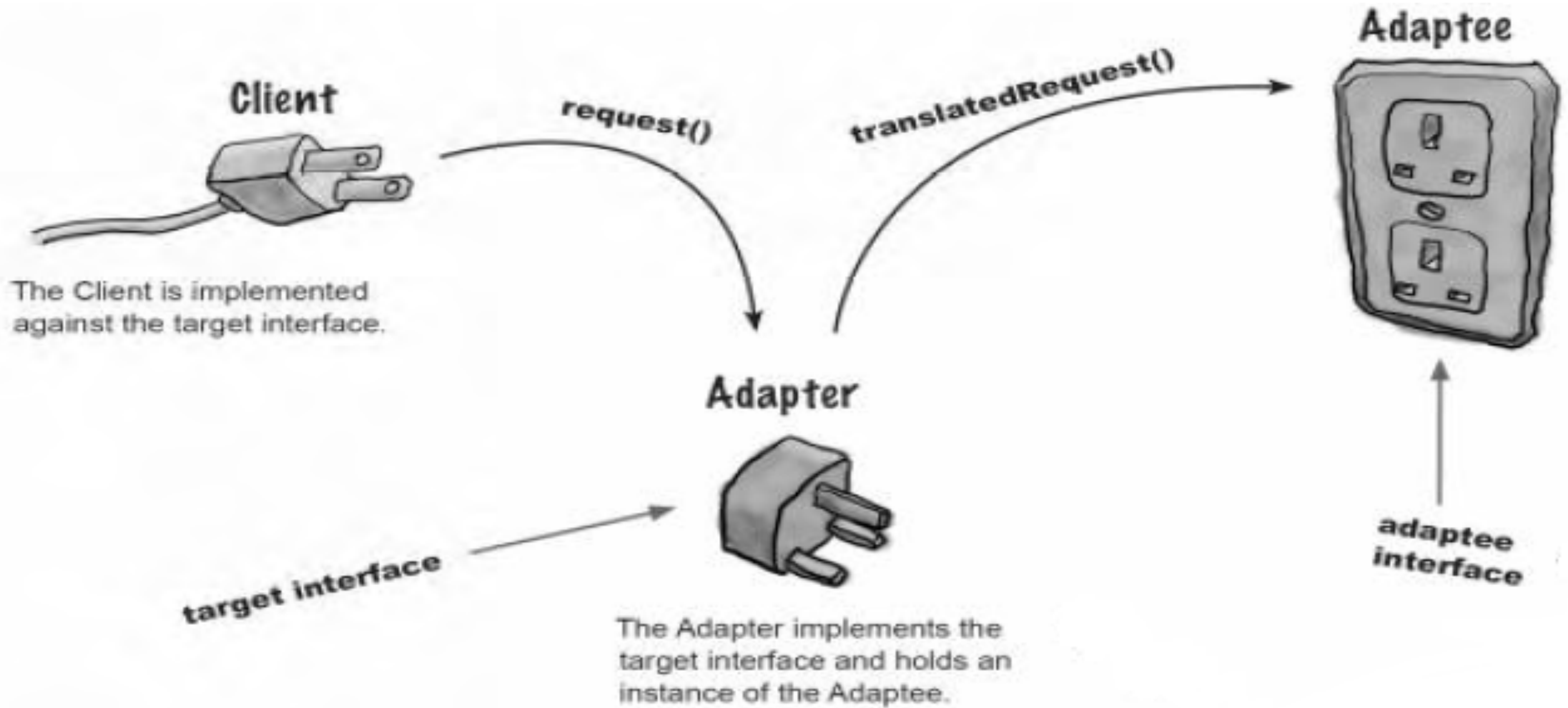
Participants:
Client
Target
Adaptee
Adapter

(Object) Adapter

```
public class Adapter extends Target {
  private Adaptee myAdaptee;

  public Adapter (Adaptee adaptee) {
      myAdaptee = adaptee;
  }

  public setAdapter (Adaptee adaptee) {
      myAdaptee = adaptee;
  }

  void public request() {
      myAdaptee.specificRequest();
  }
}
```

"Target" can be an interface, abstract class or standard class

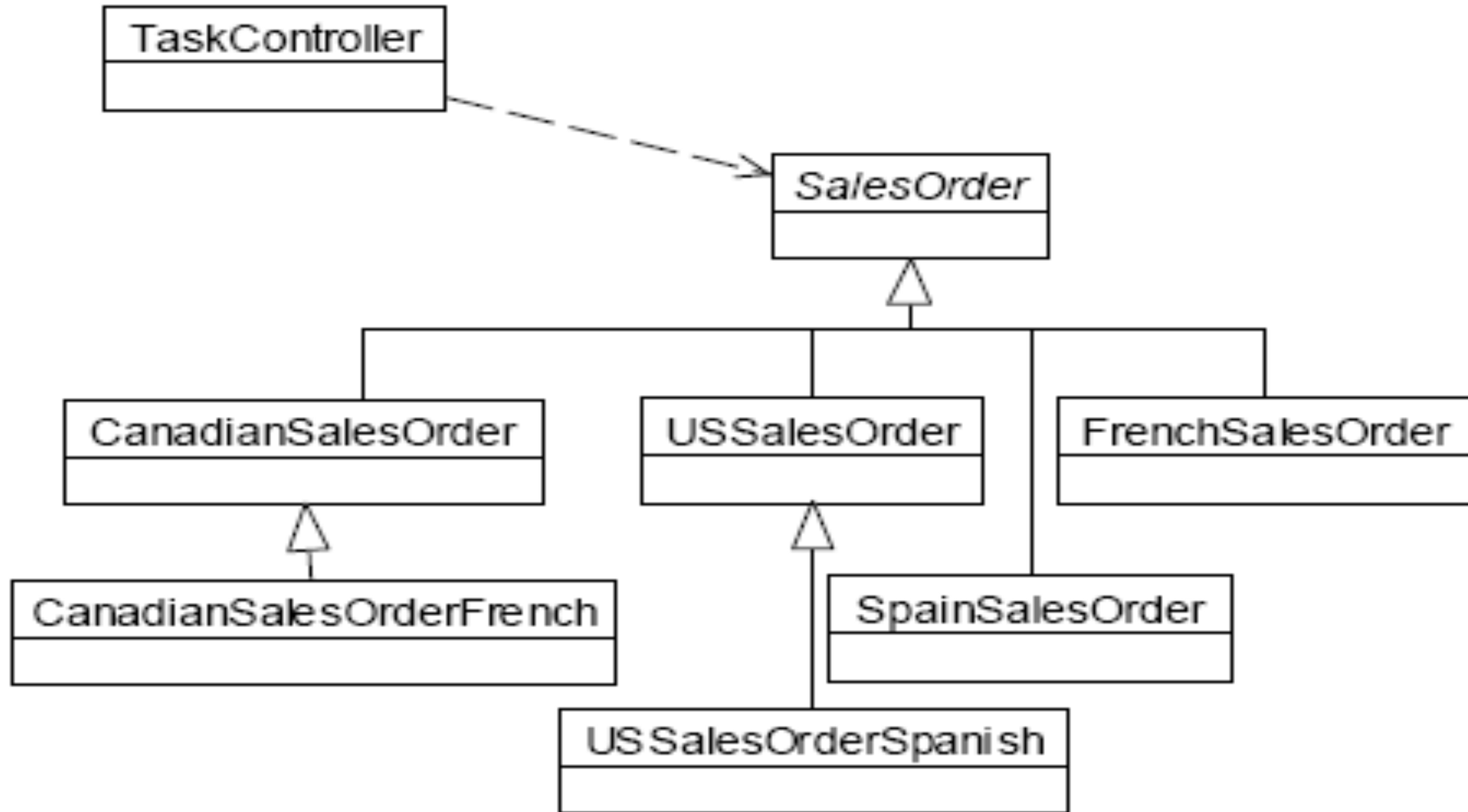
Adapter



Adapter Pattern

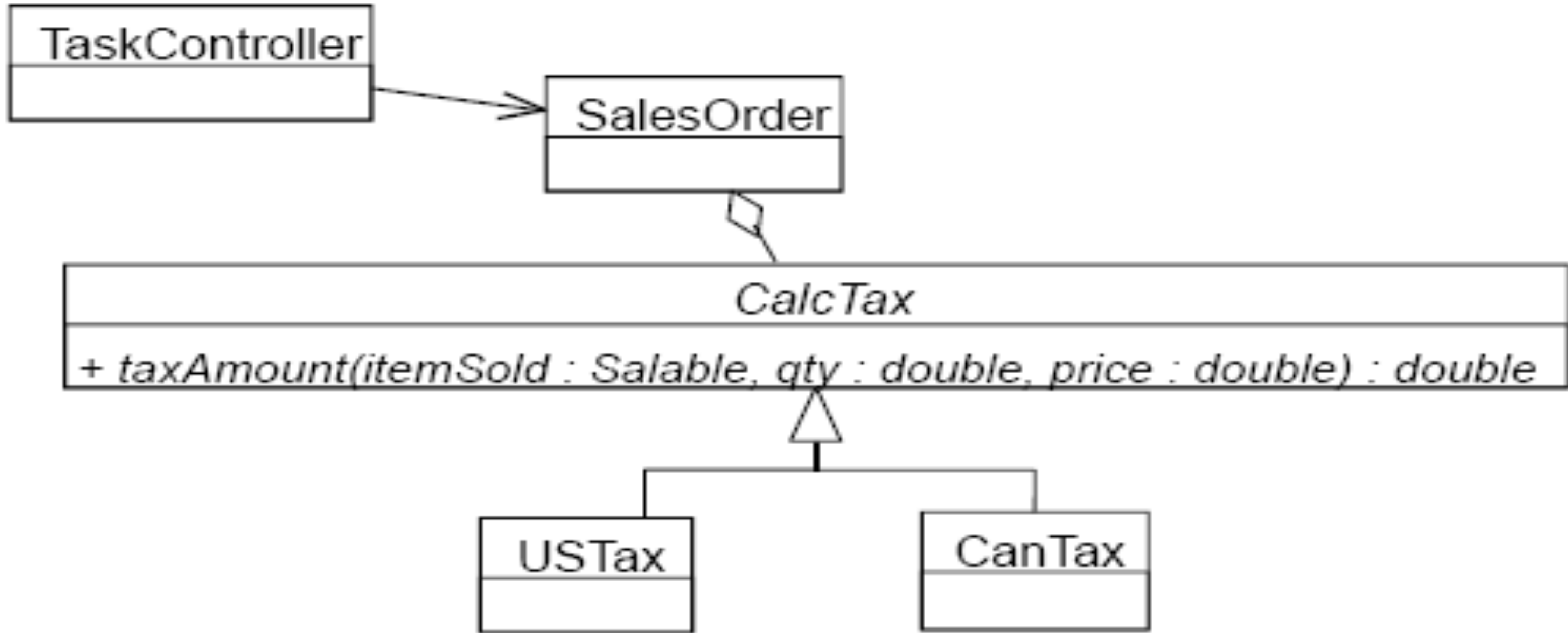
- *With the adapter pattern you don't have to worry about the interfaces of the existing classes during design.
  When you have a class with the right functionality, you can use the adapter to give the right interface.*
- *Many patterns require certain classes to derive from the same class.
  If there are preexisting classes the adapter pattern can be used to match this class to appropriate abstract class.*

Strategy Pattern

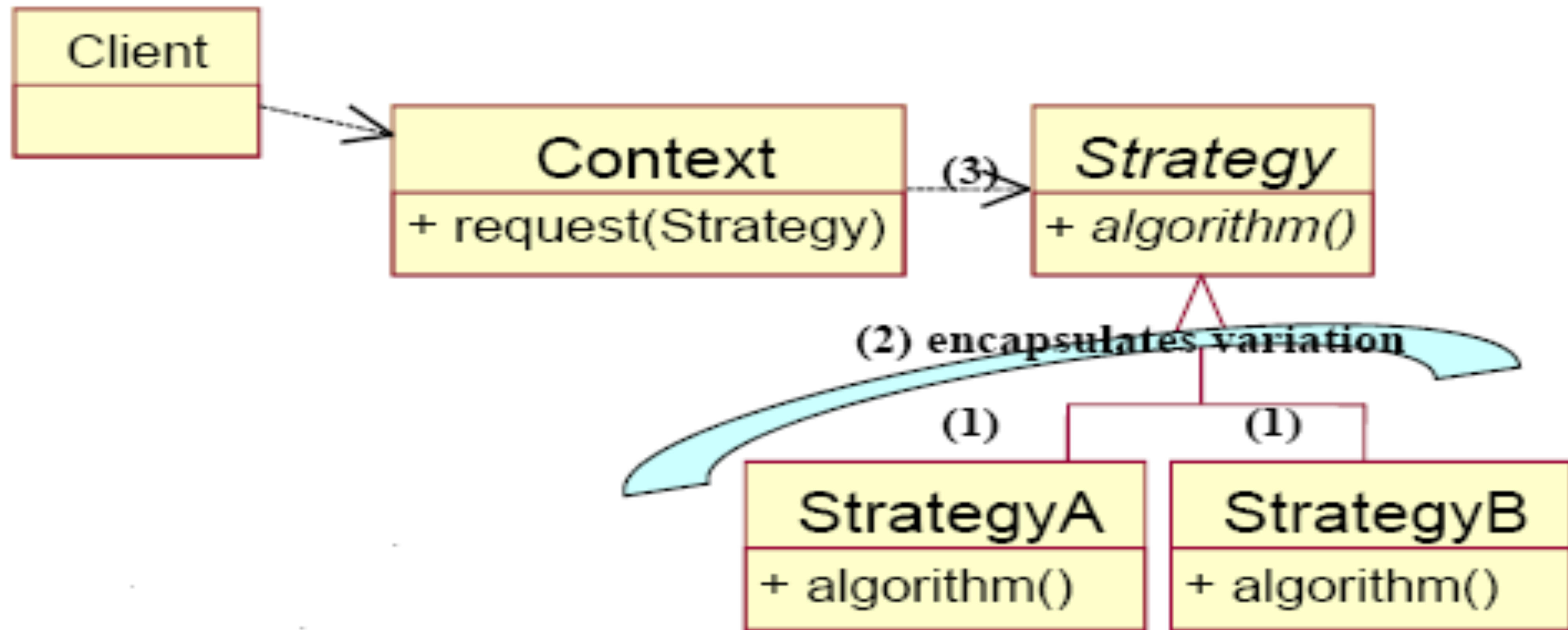


Unwanted subclasses

Strategy Pattern



Strategy



Strategy declares an interface (how the different algorithms are used)
StrategyA and StrategyB implement this different algorithms
Context uses the interface to call the concrete algorithm and maintains a reference tot the strategy object (concrete algorithm)
The Client passes the specific Strategy to the Context by a parameter

strategy implementation

```
public interface Strategy {
 public void algorithm();
}
public class StrategyA implements Strategy {
 public void algorithm(){
      // implement algorithm A
 }
}

public class StrategyB implements Strategy {
 public void algorithm(){
      // implement algorithm B
 }
}
```

strategy implementation

```
public class Context{
 private Strategy strategy;

 public Context(Strategy strategy) {
      this.strategy = strategy;
 }
 public setStrategy(Strategy strategy) {
      this.strategy = strategy;
 }
 public request() {
      strategy.algorithm();
 }
}
```

strategy implementation

```
public class Client{

 public static void main(String args[]) {
     . . . . . . . . . .
     Context context = new Context(new StrategA());
     context.request();
     . . . //algorithm A will be executed
     context.setStrategy(new StrategB());
     context.request();
      . . //algorithm B will be executed
 }
}
```

Strategy pattern

Name	Strategy pattern (behavioral pattern)
Intent	Enables you to use different algorithms, depending on the context in which they occur. You want to change behaviour (algorithms) run-time
Problem	The selection of the algorithm that need to be applied depends on the Client. Different behaviour depends on conditions in the Client class.
Solution	Separate the selection of the algorithm from the implementation. Define a family of algorithms, encapsulate each one and make them interchangeable.
Consequences	The pattern defines a family of algorithms. The algorithms vary independantly from clients that use it. Switches or conditionals can be eliminated. All the algorithms must have the same interface The client must have knowledge of the different strategies to select the required one. Increasing number of classes; Context class increases complexity When behaviour doesn’t change: don’t use the strategy pattern

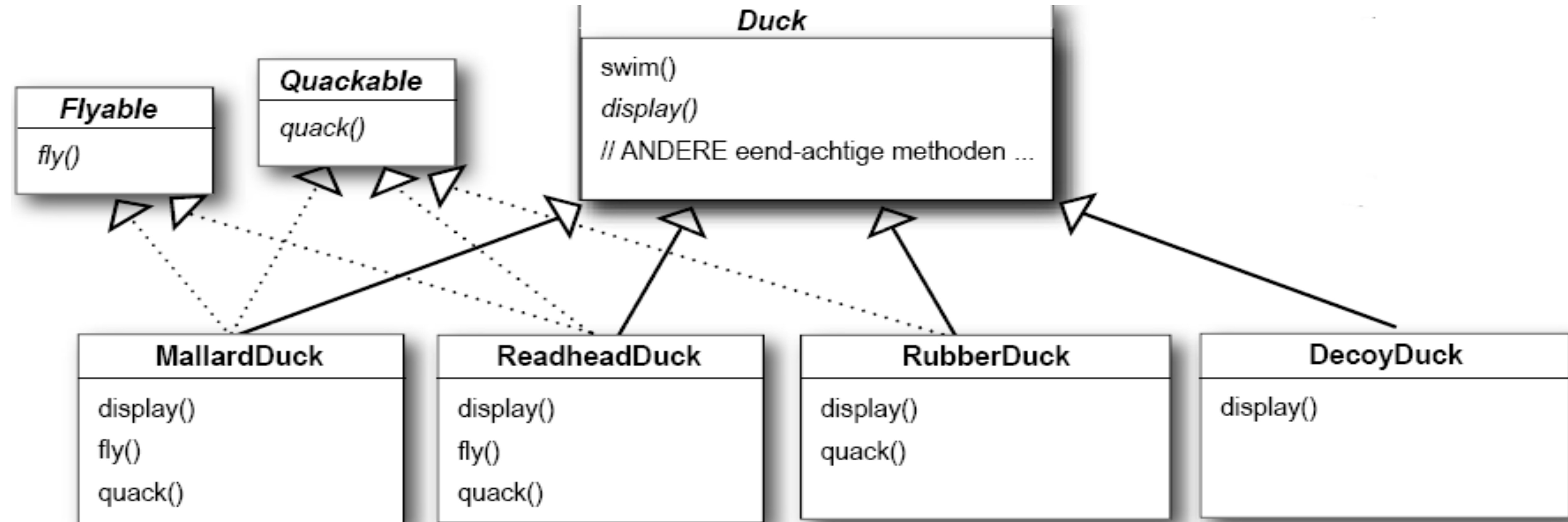
Strategy Pattern

- *Strategy pattern simplifies unit testing, because each algorithm is in its own class and can be tested through its interface alone*
- *Strategy pattern increases cohesion (each object worries only about one function)*
- *Strategy pattern decreases coupling (independency of different algorithms)*

Sorting exercise – strategy pattern

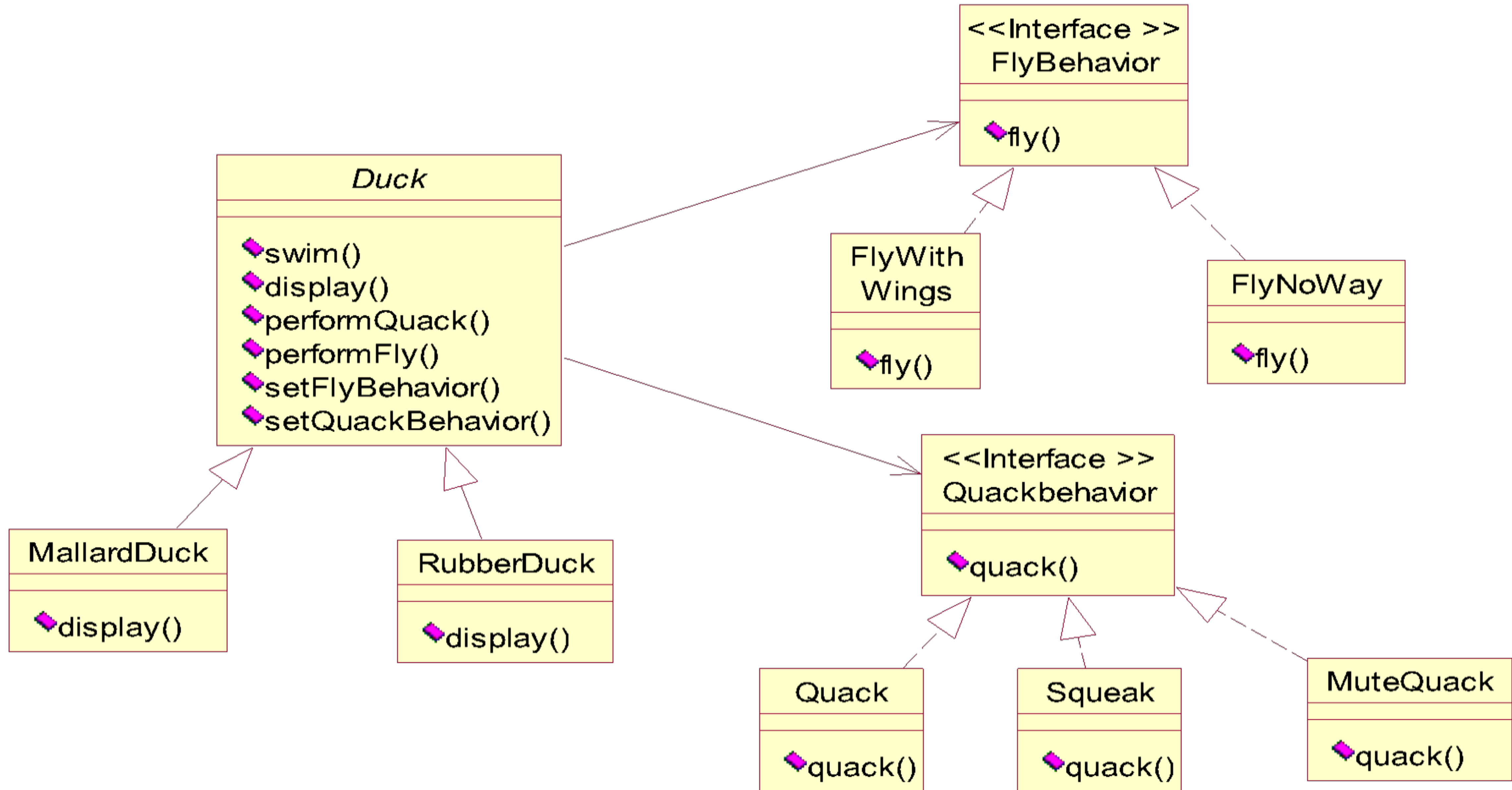
```
public interface SortInterface {
 public void sort(double[ ] list);
}
public class QuickSort implements SortInterface {
 public void sort(double[ ] list){
      // implement Quicksort algorithm
 }
}

public class BubbleSort implements SortInterface {
 public void sort(double[ ] list){
      // implement Bubblesort algorithm
 }
}
```

Duck-exercise – design with traditional inheritance



Source Duck-exercise: Freeman, E & E., Head First, Design Patterns

Duck-exercise – strategy pattern



Duck-exercise– strategy pattern

```
public abstract class Duck {
   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;

   public Duck() {
   }
   public void setFlyBehavior (FlyBehavior fb) {
               flyBehavior = fb;
   }
   public void setQuackBehavior(QuackBehavior qb) {
               quackBehavior = qb;
   }
   public void performFly() {
               flyBehavior.fly();
   }
   public void performQuack() {
               quackBehavior.quack();
   }
}
```

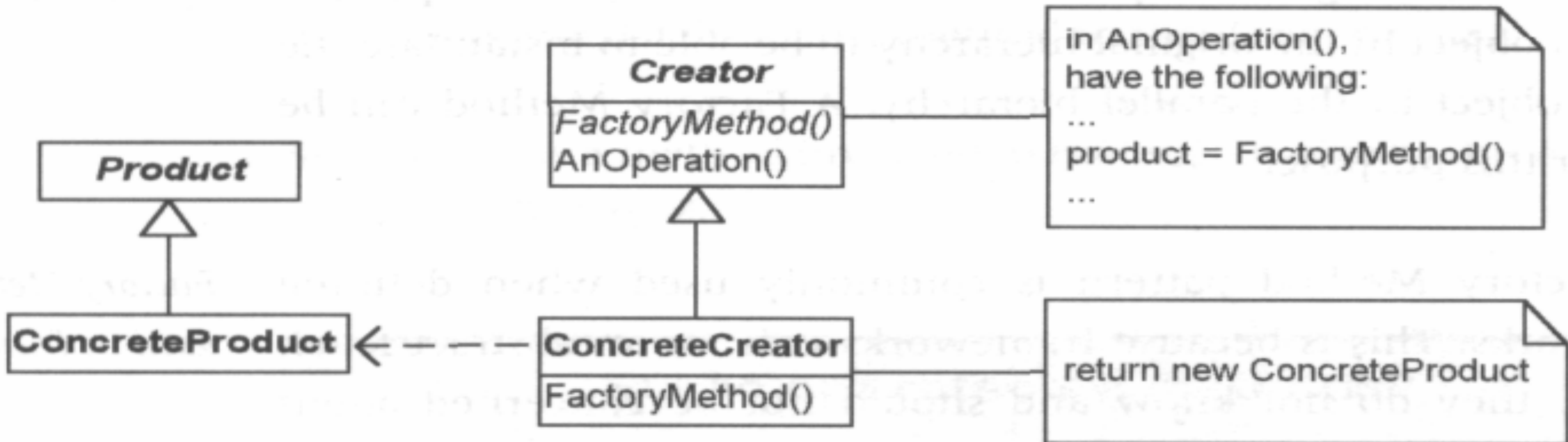
Duck-exercise – strategy pattern

```
public class DecoyDuck extends Duck {
  public DecoyDuck() {
        setFlyBehavior(new FlyNoWay());
        setQuackBehavior(new MuteQuack());
  }
  public void display() {
        System.out.println("I'm a duck Decoy");
  }
}
```

Factories

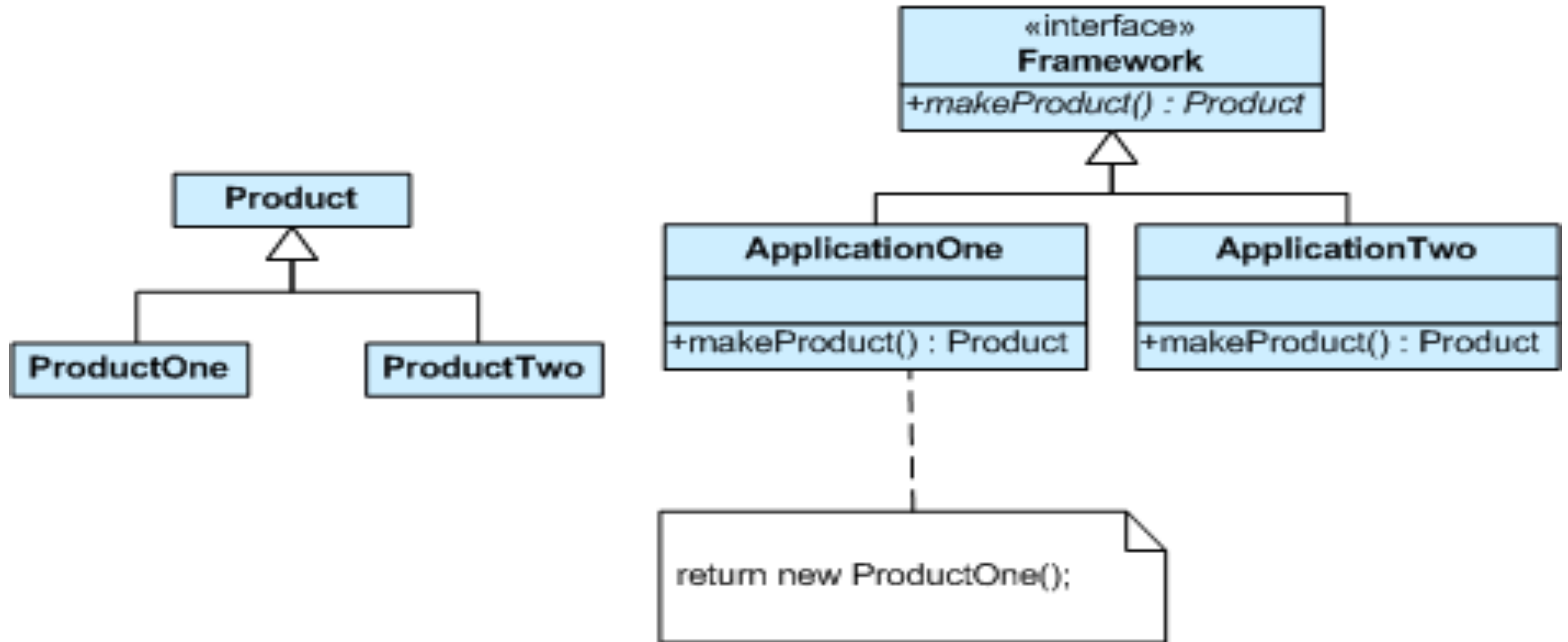
- Creational patterns
- Een factory verbergt het creëren van objecten. Deze code heeft veel kennis van de objecten (parameters e.d.) nodig.
- De client kan zich concentreren op waar het echt om gaat (de functionaliteit).
- Het deel van de code dat objecten creëert (de factory) bemoeit zich niet met de functionaliteit.
- Conclusie: een object of creëert andere objecten
of gebruikt andere objecten
- Goede cohesie
- Verbergt wat varieert.
- Factories zijn een goed voorbeeld van open closed principe (open voor uitbreiding, gesloten voor verandering)

Factory Method pattern



Product: definieert de interface van de objecten die de factory method creëert
ConcreteProduct: implementeert de Product interface
Creator: wordt ook wel Factory genoemd; geeft een product terug.
ConcreteCreator: implemeteert de factoryMethod(), de methode die in feit alle producten aanmaakt; deze klasse weet als enig hoe de producten te maken

Factory Method pattern



Factory Method pattern

Name	Factory Method pattern (creational pattern)
Intent	Het definiëren van een interface voor het creëren van een object. De subklassen beslissen welke klasse er geïnstantieerd wordt. De subklassen zijn verantwoordelijk voor de instantiatie.
Problem	Een klasse moet een object van een subklasse instantiëren, maar weet niet welke subklasse.
Solution	Definieer een interface met de factory Method. Laat de subklasse van deze interface beslissen welk type object geïnstantieerd wordt en op welke manier.
Consequences	De Client is niet op de hoogte welk object gecreëerd wordt en is dus onafhankelijk van het te creëren object. Soms moet er speciaal hiervoor een hiërarchie van klasse – subklasse opgezet worden (als er maar 1 klasse is).

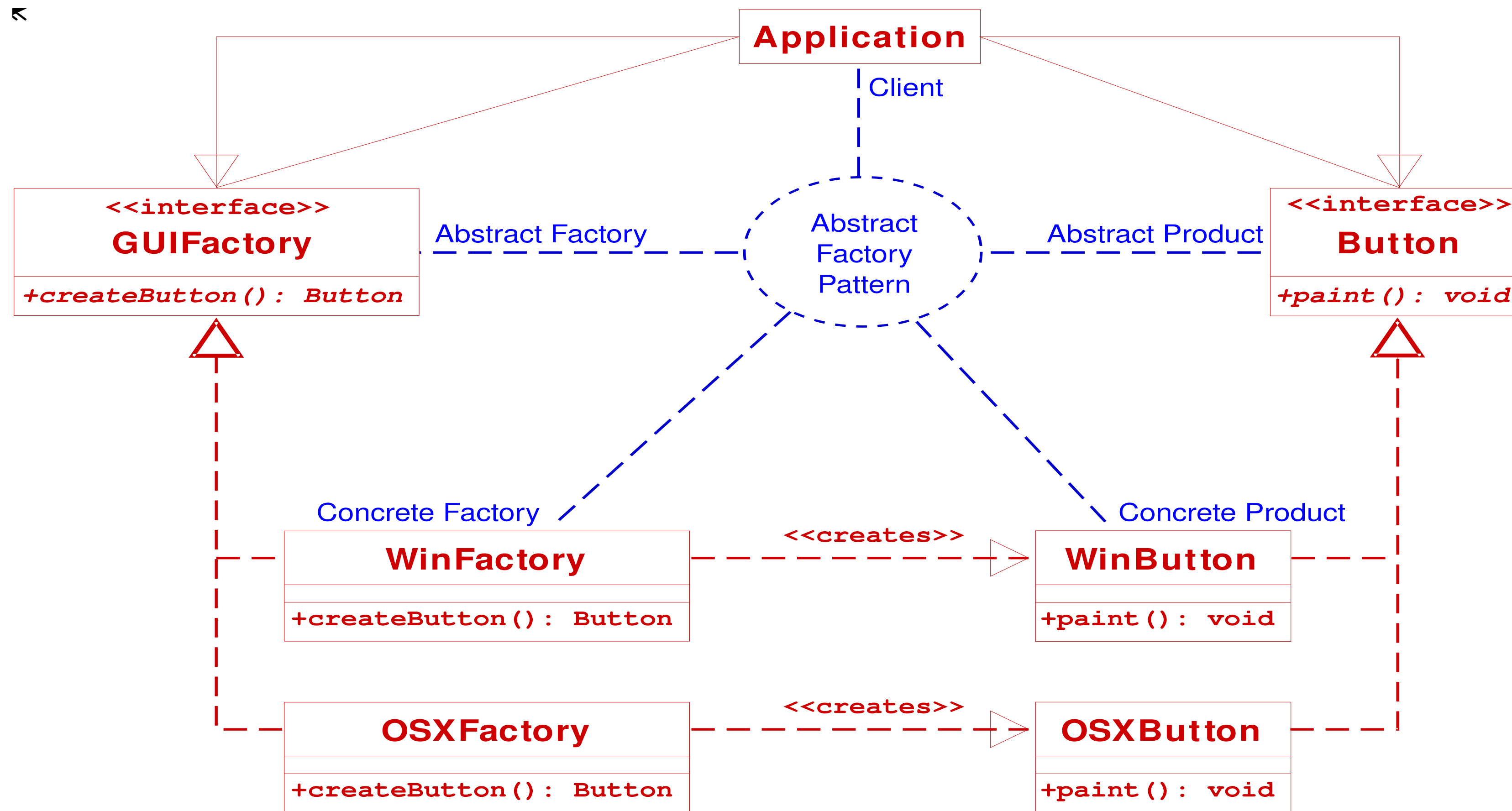
Factory Method Implementatie

```
public interface Product{ □ }
public class ConcreteProductA implements Product { □ }

public abstract class Creator {
 public void anOperation() {
      Product product = factoryMethod();
 }
 protected abstract Product factoryMethod();
}
public class ConcreteCreatorA extends Creator {
 protected Product factoryMethod() {
      return new ConcreteProductA();
 }
}
```

factory Method implementatie

```
public class Client{

 public static void main(String args[]) {
     . . . . . . . . . .
     Creator creator = new ConcreteCreatorA();
     creator.anOperation();
     . . . //concreteProductA wordt gecreerd
 }
}
```

Factory Example

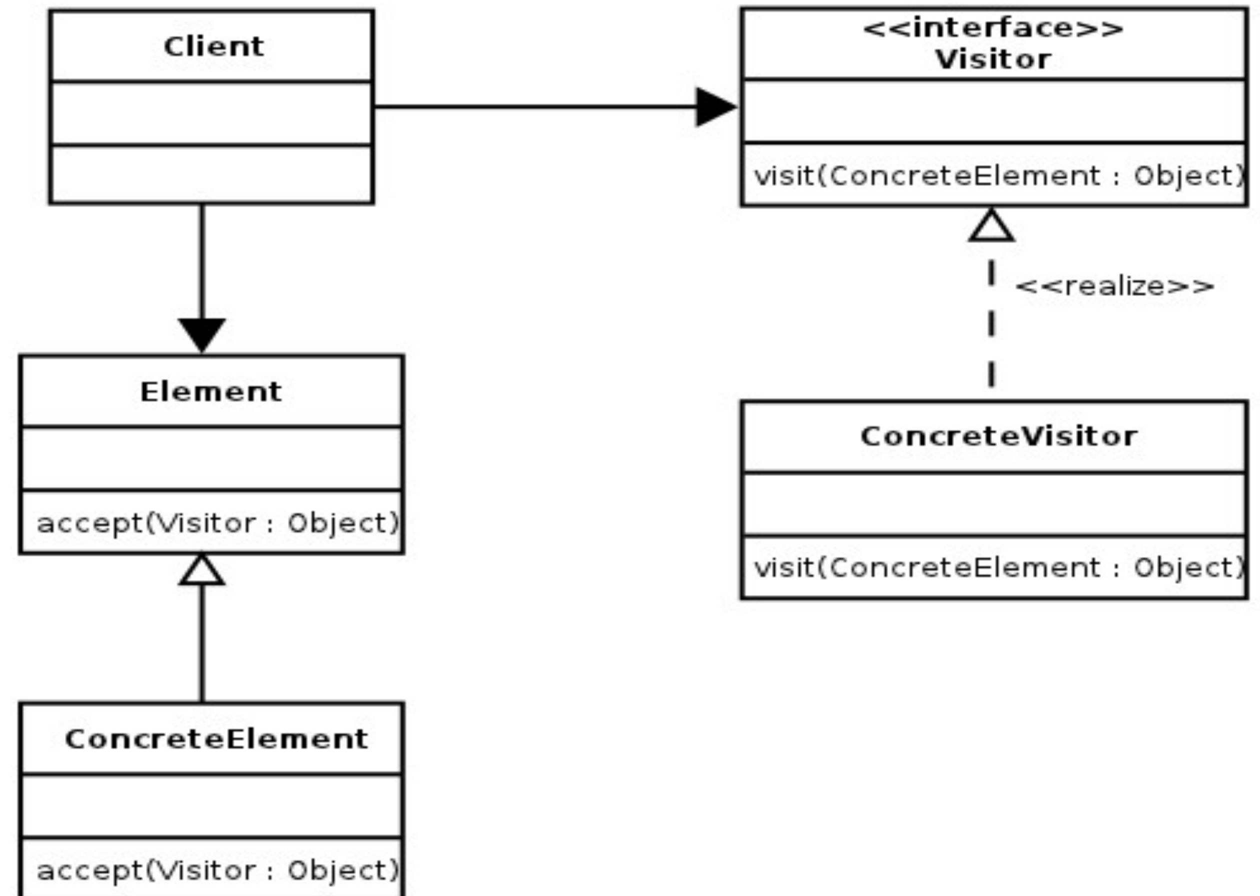


Factory Method: voorbeeld

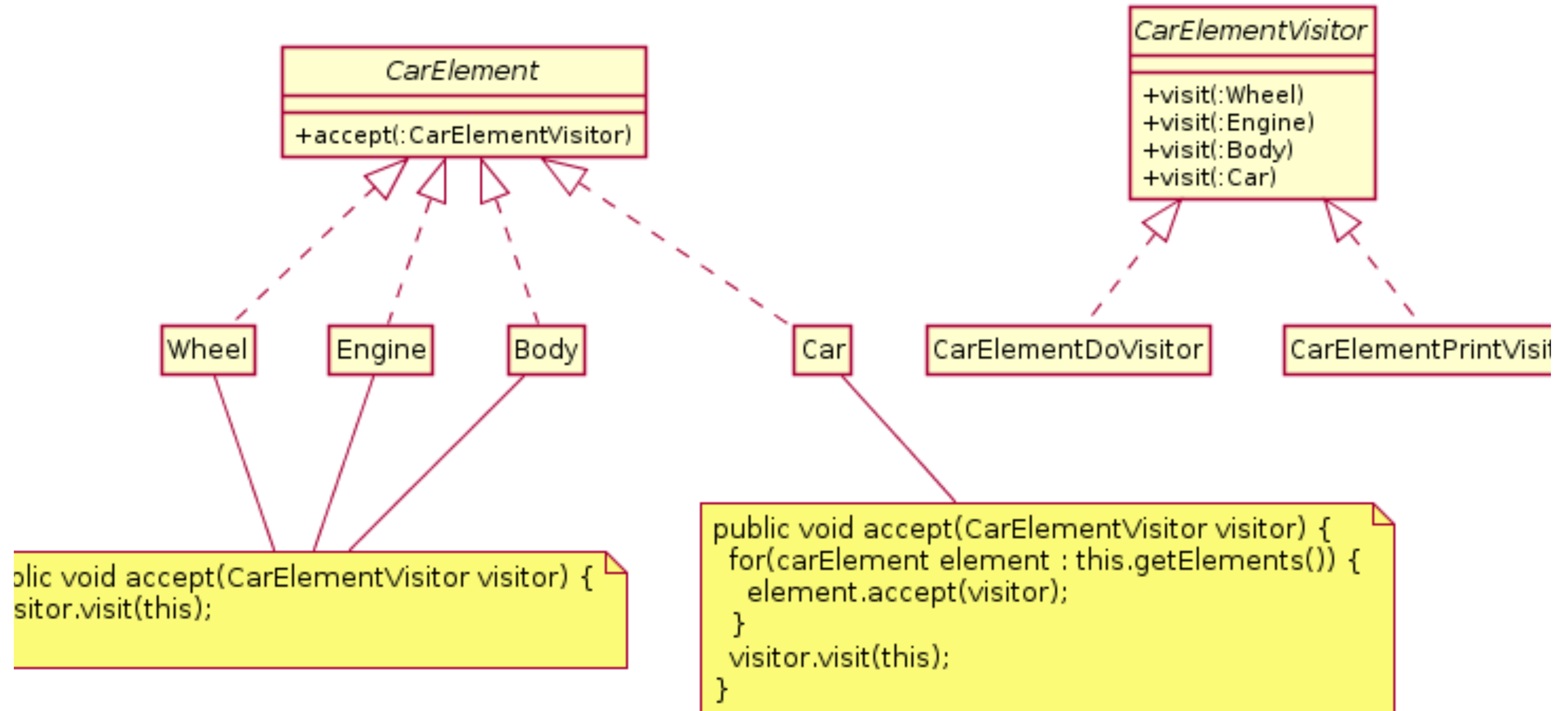
Voorbeeld factory method: iterator() in Java

Visitor Design Pattern

- **Problem**: Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid “polluting” the node classes with these operations. And, you don’t want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.
- **Solution**: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



Visitor Example



Next up...

- Continue with assignment 3