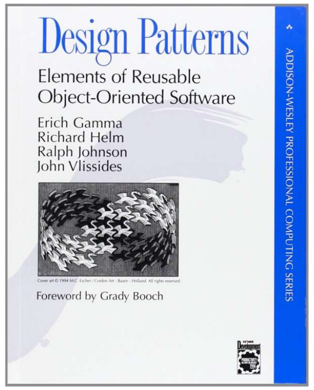
# Design Patterns – Structural

UA.DETI.PDS
José Luis Oliveira

UNIVERSIDADE
DE AVEIRO

## Resources

- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- *Design Patterns Explained Simply (sourcemaking.com)*

## Structural patterns

- They simplify software design by identifying a simple way to build relationships between entities.

UNIVERSIDADE DE AVEIRO

## Structural design patterns

Class

- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

UNIVERSIDADE DE AVEIRO

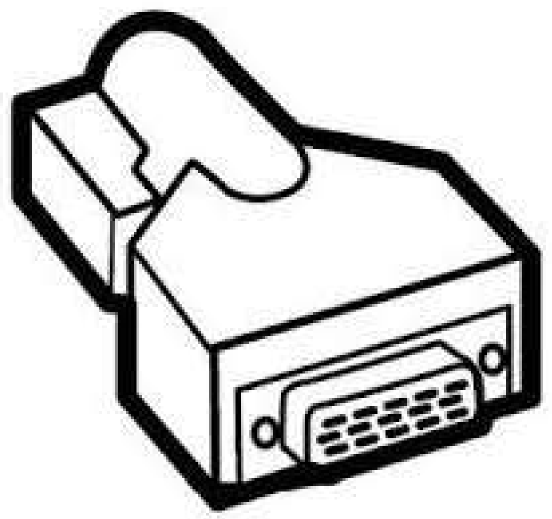
## Class Adapter

Class

- Adapter

Object

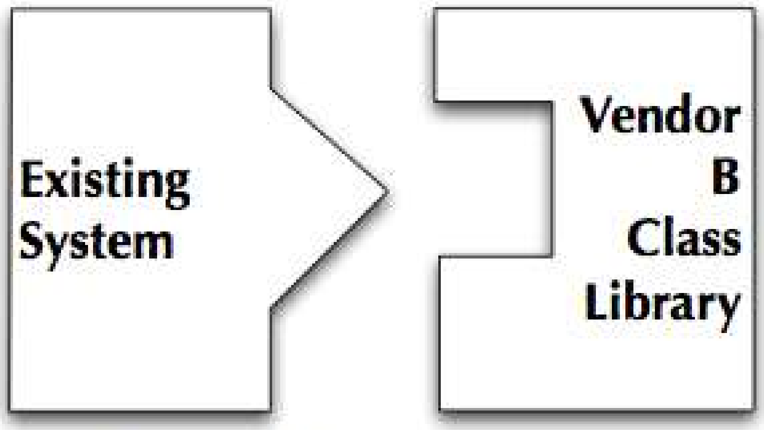
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy



## Software Adapters (I)

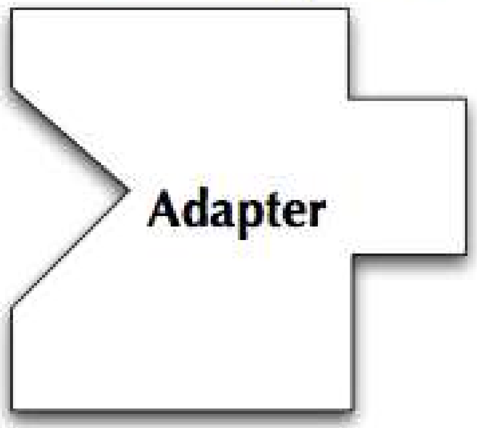
- Pre-condition
  - You are maintaining an existing system that makes use of a third-party class library from vendor A
- Stimulus
  - Vendor A goes belly up and corporate policy does not allow you to make use of an unsupported class library
- Response
  - Vendor B provides a similar class library but its interface is completely different from the interface provided by vendor A
- Assumptions
  - You don't want to change your code, and you can't change vendor B's code
- Solution?:
  - Write new code that adapts vendor B's interface to the interface expected by your original code

## Software Adapters (II)



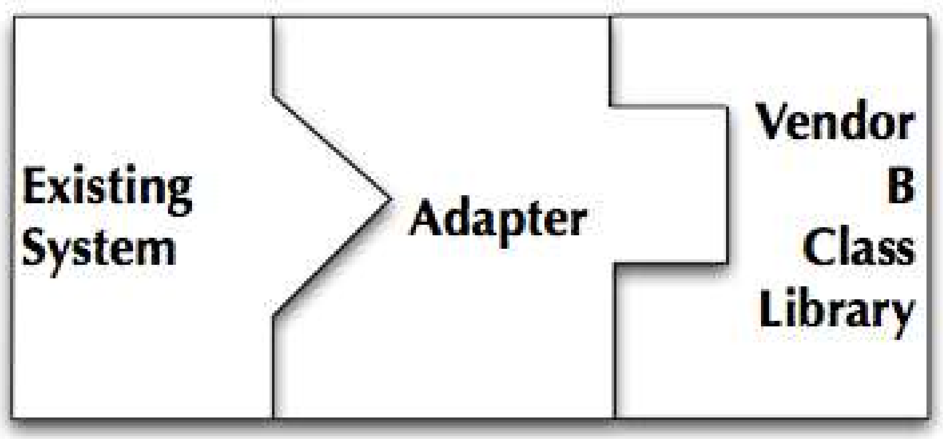
Interface Mismatch
Need Adapter

Create Adapter



And then...

## Software Adapters (III)

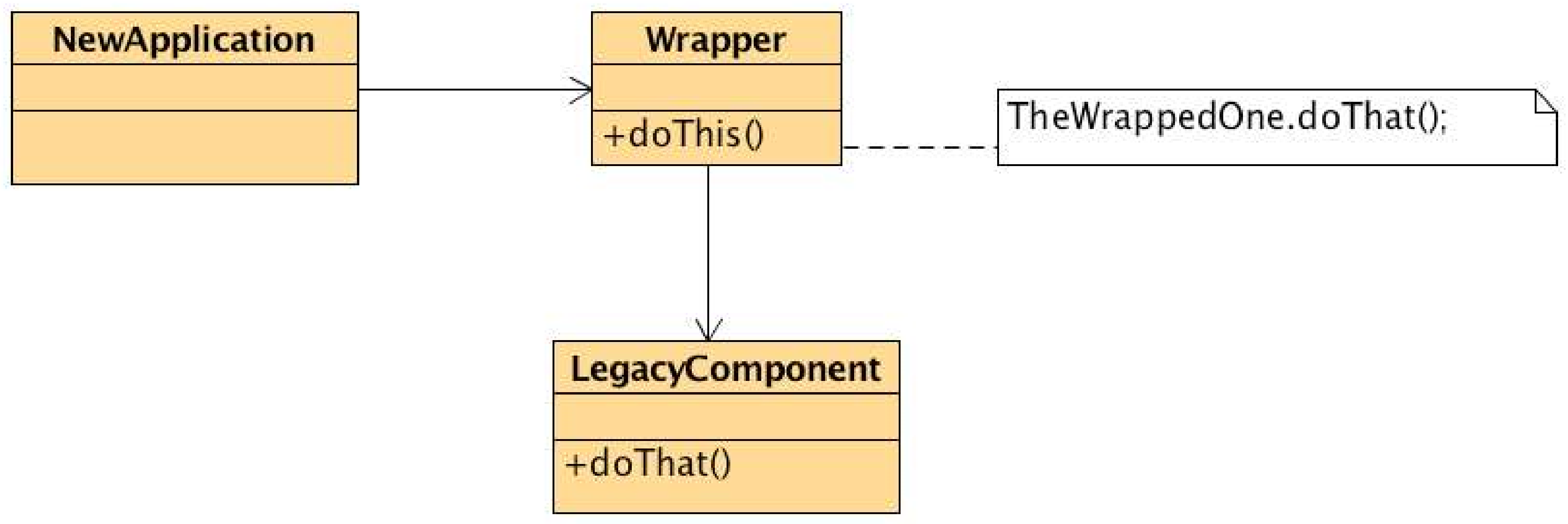


- ...plug it in
- Benefit: Existing system and new vendor library do not change - new code is isolated within the adapter

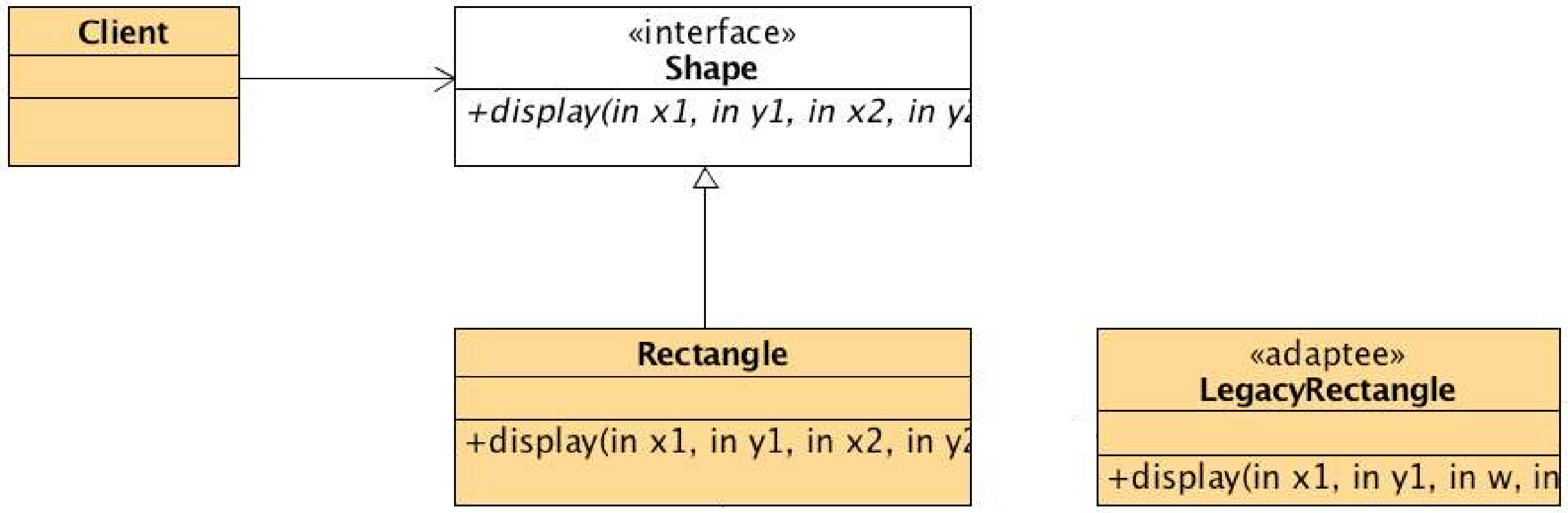
## Motivation

- **Intent**
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
- **Problem**
  - An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

## Solution


NewApplication
Wrapper
+doThis()
TheWrappedOne.doThat();
LegacyComponent
+doThat()

## Problem – Solution?


Client
«interface»
Shape
+display(in x1, in y1, in x2, in y
Rectangle
+display(in x1, in y1, in x2, in y
«adaptee»
LegacyRectangle
+display(in x1, in y1, in w, in

## Example – the problem

```java
interface Shape {
  void draw(int x1, int y1, int x2, int y2);
}

class Rectangle implements Shape {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("rectangle from (" + x1 + ',' + y1 + ") to (" +
x2
                + ',' + y2 + ')');
    }
}

class LegacyRectangle {
    public void draw(int x, int y, int w, int h) {
        System.out.println("old format rectangle at (" + x + ',' + y
                + ") with width " + w + " and height " + h);
    }
}
```

## Example – the problem

```java
public class NoAdapterDemo {
    public static void main(String[] args) {
        Object[] shapes = { new Rectangle(), new LegacyRectangle() };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
           if (shapes[i].getClass().getSimpleName().equals("Rectangle"))
                ((Rectangle) shapes[i]).draw(x1, y1, x2, y2);
           else if (shapes[i].getClass().getSimpleName()
                   .equals("LegacyRectangle"))
                ((LegacyRectangle) shapes[i]).draw(Math.min(x1, x2),
                    Math.min(y1, y2), Math.abs(x2 - x1), Math.abs(y2 - y1));
    }
}
```

```
rectangle from (10,20) to (30,60)
old format rectangle at (10,20) with width 20 and height 40
```

UNIVERSIDADE DE AVEIRO

## Example – the Adapter solution

```java
class OldRectangle implements Shape {
    private LegacyRectangle adaptee = new LegacyRectangle();

    public void draw(int x1, int y1, int x2, int y2) {
        adaptee.draw(Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2 - x1),
                Math.abs(y2 - y1));
    }
}

public class AdapterDemo2 {
    public static void main(String[] args) {
        Shape[] shapes = { new Rectangle(), new OldRectangle() };
        // A begin and end point from a graphical editor
        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
           shapes[i].draw(x1, y1, x2, y2);
    }
}
```

## Another example

```
interface Rectangle {
  void scale(int factor); //grow or shrink by factor
  void setWidth();
  float getWidth();
  float area(); …
}

class Client {
  void clientMethod(Rectangle r) {
    // ...
    r.scale(2);
  }
}

class NonScalableRectangle {
  void setWidth(); …
  // no scale method!
}
```

How to use this rectangle in Client?

UNIVERSIDADE DE AVEIRO

## Another example: via subclassing

- Class adapter adapts via subclassing

```
class ScalableRectangle1
        extends NonScalableRectangle
        implements Rectangle {

    void scale(int factor) {
        setWidth(factor*getWidth());
        setHeight(factor*getHeight());
    }
}
```

## Another example: via delegation

- Object adapter adapts via delegation:
  - it forwards work to delegate

```
class ScalableRectangle2 implements Rectangle {
   NonScalableRectangle r; // delegate
   ScalableRectangle2(NonScalableRectangle r) {
     this.r = r;
   }

   void scale(int factor) {
     setWidth(factor * r.getWidth());
     setHeight(factor * r.getHeight());
   }

   float getWidth() { return r.getWidth(); }
   // ...
}
```

## Subclassing versus delegation

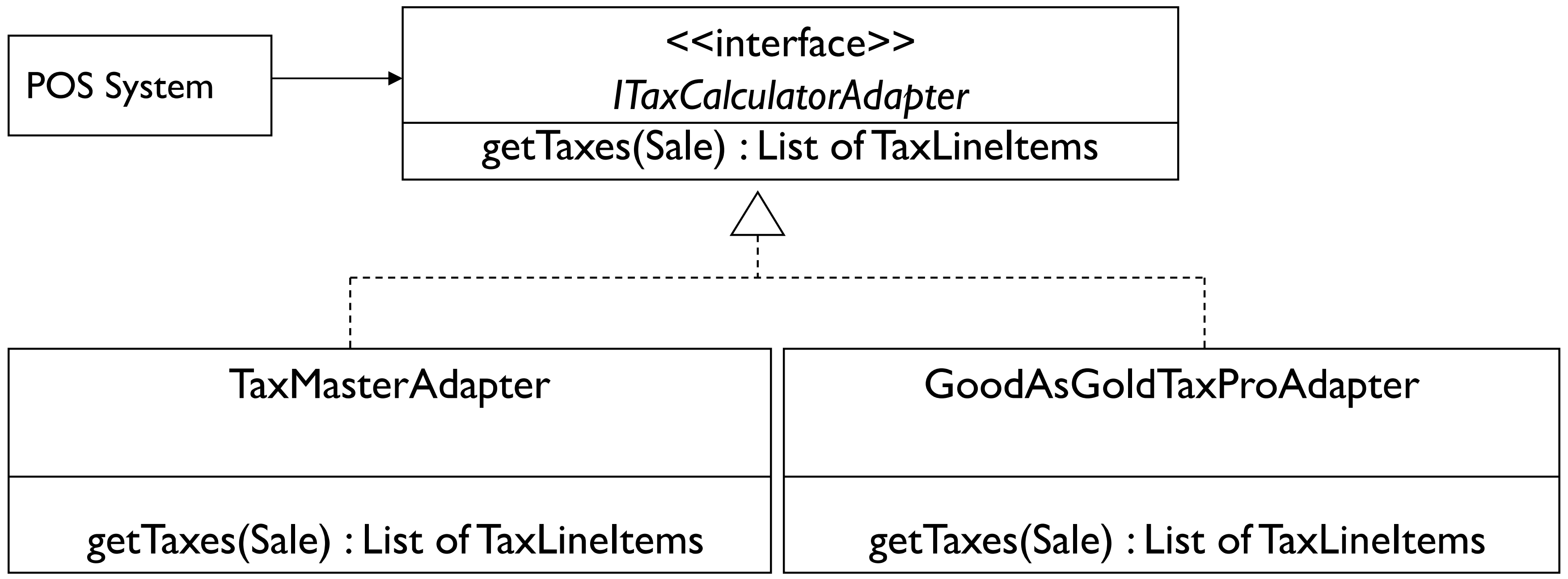
- Subclassing
  - Automatically gives access to all methods in the superclass
  - More efficient
- Delegation
  - Permits removal of methods
  - Wrappers can be added and removed dynamically
  - Multiple objects can be composed
  - Bottom line: more flexible

UNIVERSIDADE DE AVEIRO

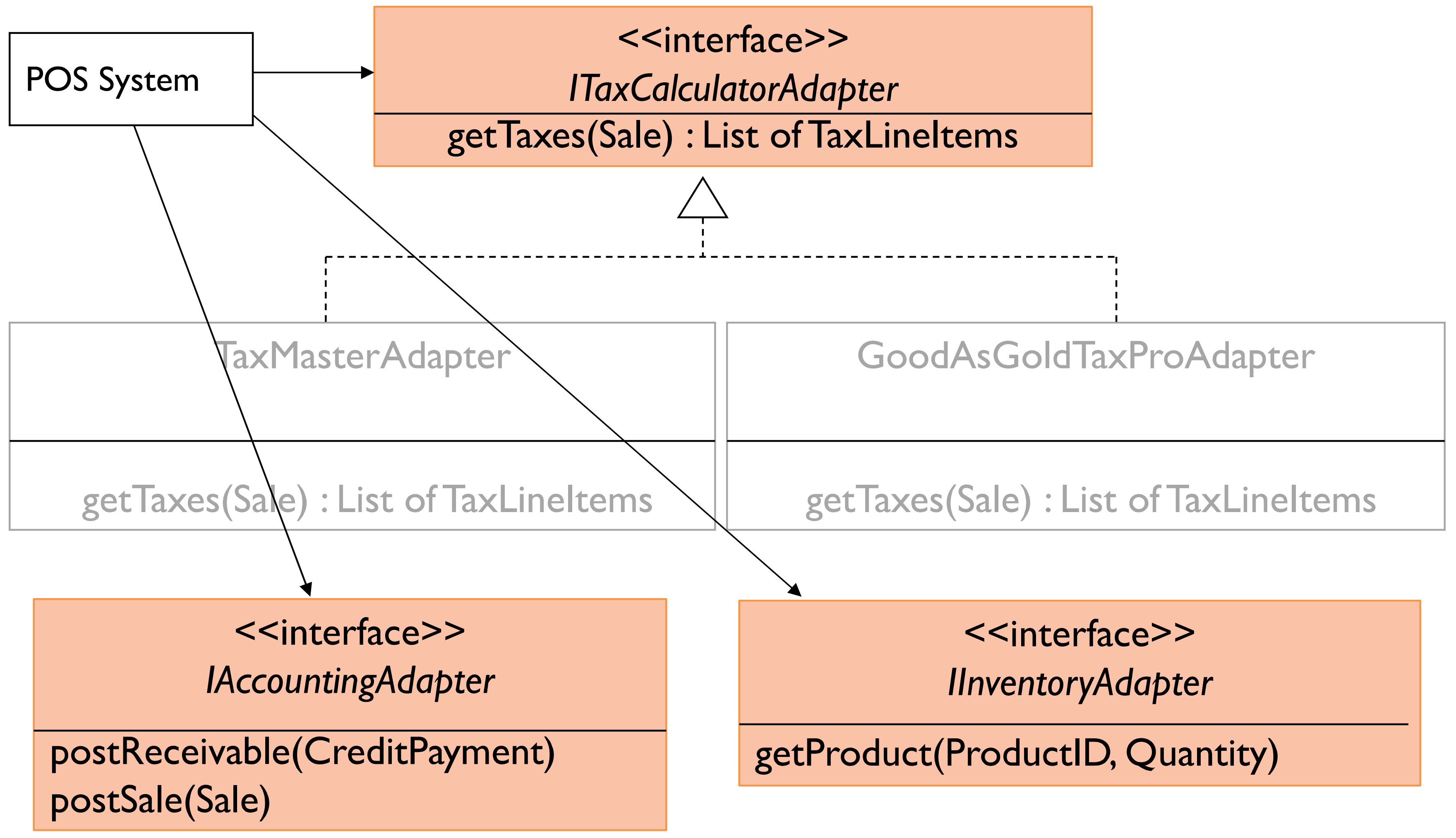
## Exercise

- A Point-of-Sale system needs to support services from different third-party vendors:
  - Tax calculator service from different vendors
  - Credit authorization service from different vendors
  - Inventory systems from different vendors
  - Accounting systems from different vendors
- Each vendor service has its own API, which can't be changed
- What design pattern solves this problem?

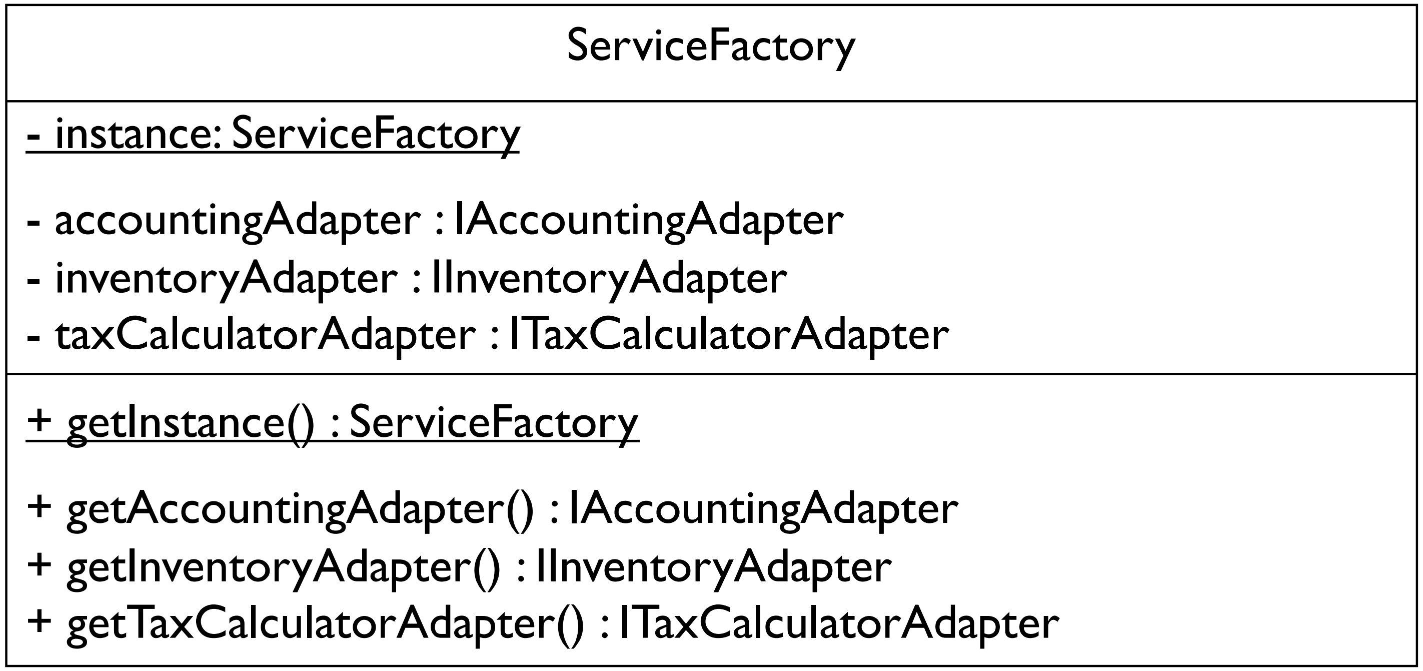
## The Solution: Object Adapter


POS System
<<interface>>
*ITaxCalculatorAdapter*
getTaxes(Sale) : List of TaxLineItems
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxProAdapter
getTaxes(Sale) : List of TaxLineItems

## Extending the problem/solution


POS System
<<interface>>
*ITaxCalculatorAdapter*
getTaxes(Sale) : List of TaxLineItems
TaxMasterAdapter
getTaxes(Sale) : List of TaxLineItems
GoodAsGoldTaxProAdapter
getTaxes(Sale) : List of TaxLineItems
<<interface>>
*IAccountingAdapter*
postReceivable(CreditPayment)
postSale(Sale)
<<interface>>
*IInventoryAdapter*
getProduct(ProductID, Quantity)

## The Solution


ServiceFactory
- instance: ServiceFactory
- accountingAdapter : IAccountingAdapter
- inventoryAdapter : IInventoryAdapter
- taxCalculatorAdapter : ITaxCalculatorAdapter
+ getInstance() : ServiceFactory
+ getAccountingAdapter() : IAccountingAdapter
+ getInventoryAdapter() : IInventoryAdapter
+ getTaxCalculatorAdapter() : ITaxCalculatorAdapter

- Single instance of ServiceFactory ensures single instance of adapter objects.
  - underline means static. *instance* and *getInstance* are static.

## Check list

- Decide if "platform independence" and creation services are the current source of pain.
- Map out a matrix of "platforms" versus "products".
- Define a factory interface that consists of a factory method per product.
- Define a factory derived class for each platform that encapsulates all references to the new operator.
- The client should retire all references to new, and use the factory methods to create the product objects.

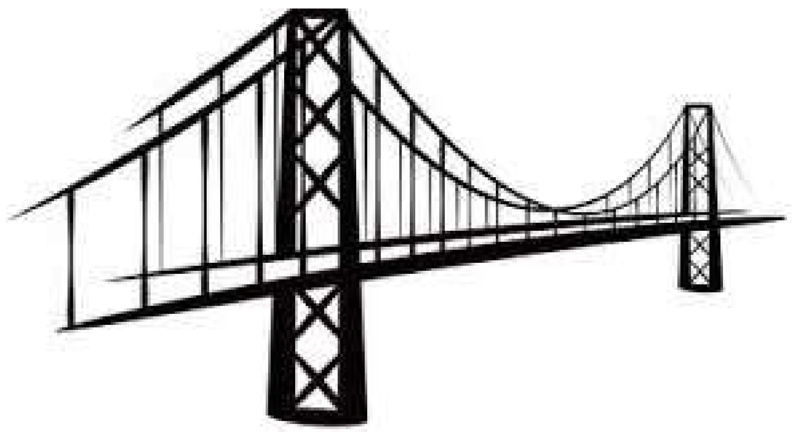
## Structural design patterns

Class

- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy



UNIVERSIDADE DE AVEIRO

## Motivation

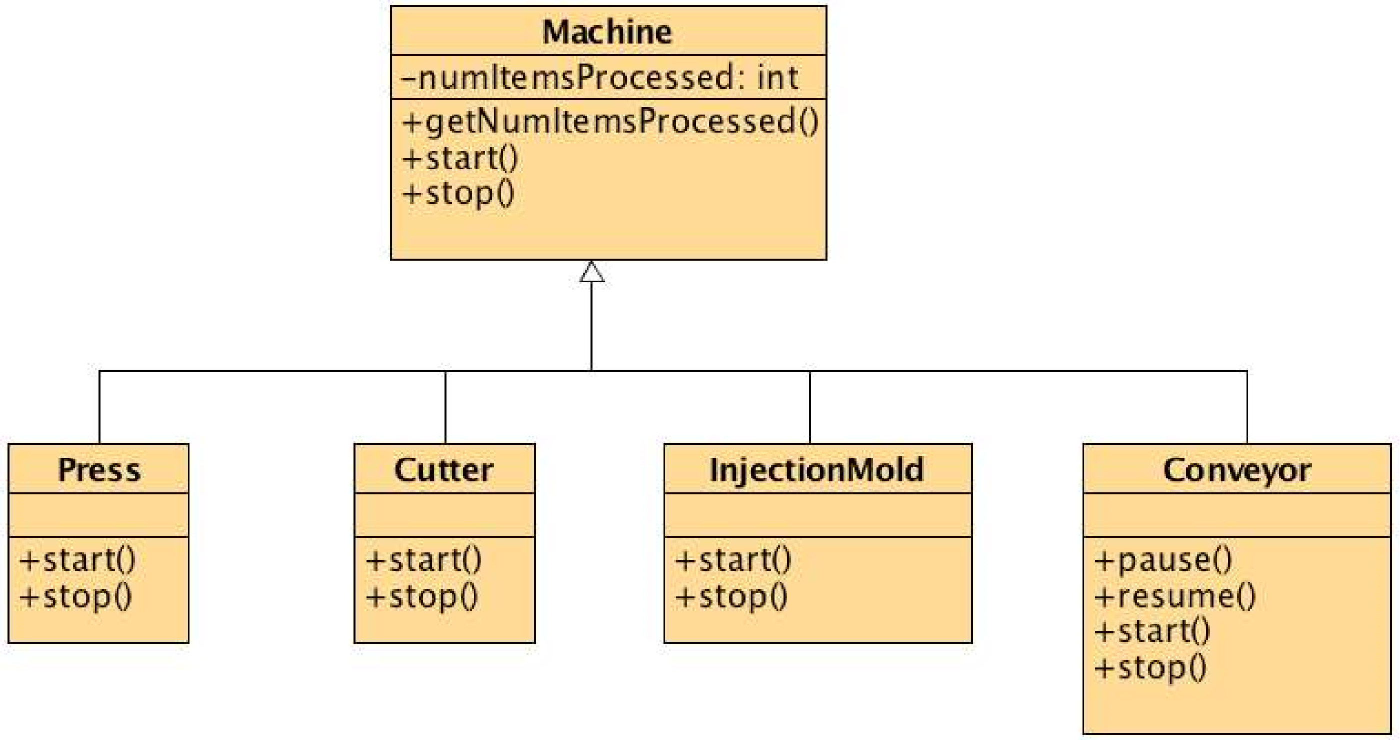
- **Intent**
  - Decouple an abstraction from its implementation so that the two can vary independently.
  - Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.

- **Problem**
  - "Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

UNIVERSIDADE DE AVEIRO

## Bridge

- An abstraction, *Machine*, has one of several possible implementations, which may override or extend *start()* and *stop()*

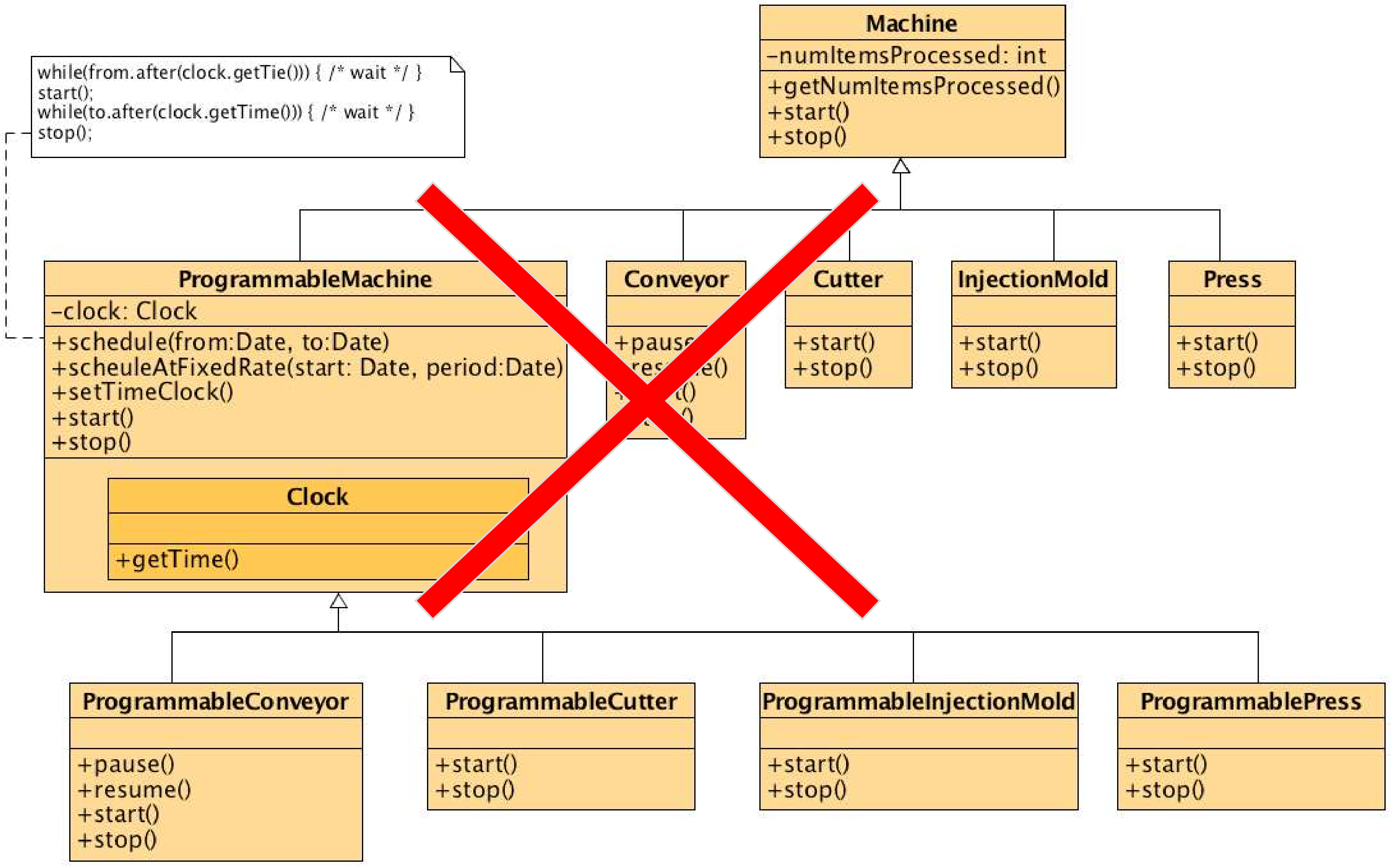


## Bridge – the problem

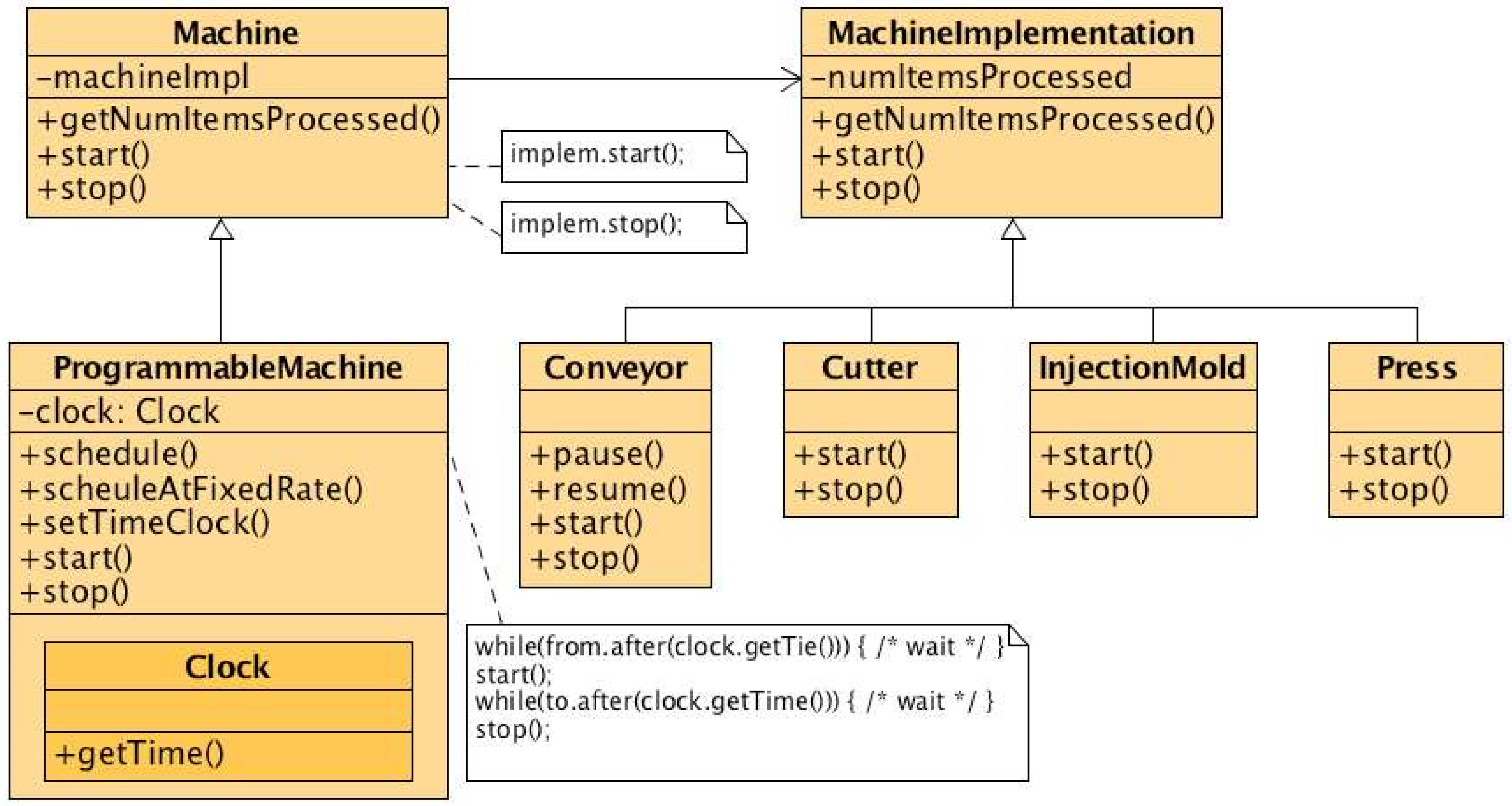
- Later on, machines are bought which can be programmed to start and stop at given times, and even to do it periodically
  - We are forced to add many new classes: every combination of {non-programmable, programmable} × {press, cutter, injection molding, conveyor belt} because start/stop are different and may have or not schedule() capability
- Inheritance binds an implementation to the abstraction permanently
  - makes it difficult to modify, extend, and reuse abstractions and implementations independently

UNIVERSIDADE DE AVEIRO

## Bridge – the solution?

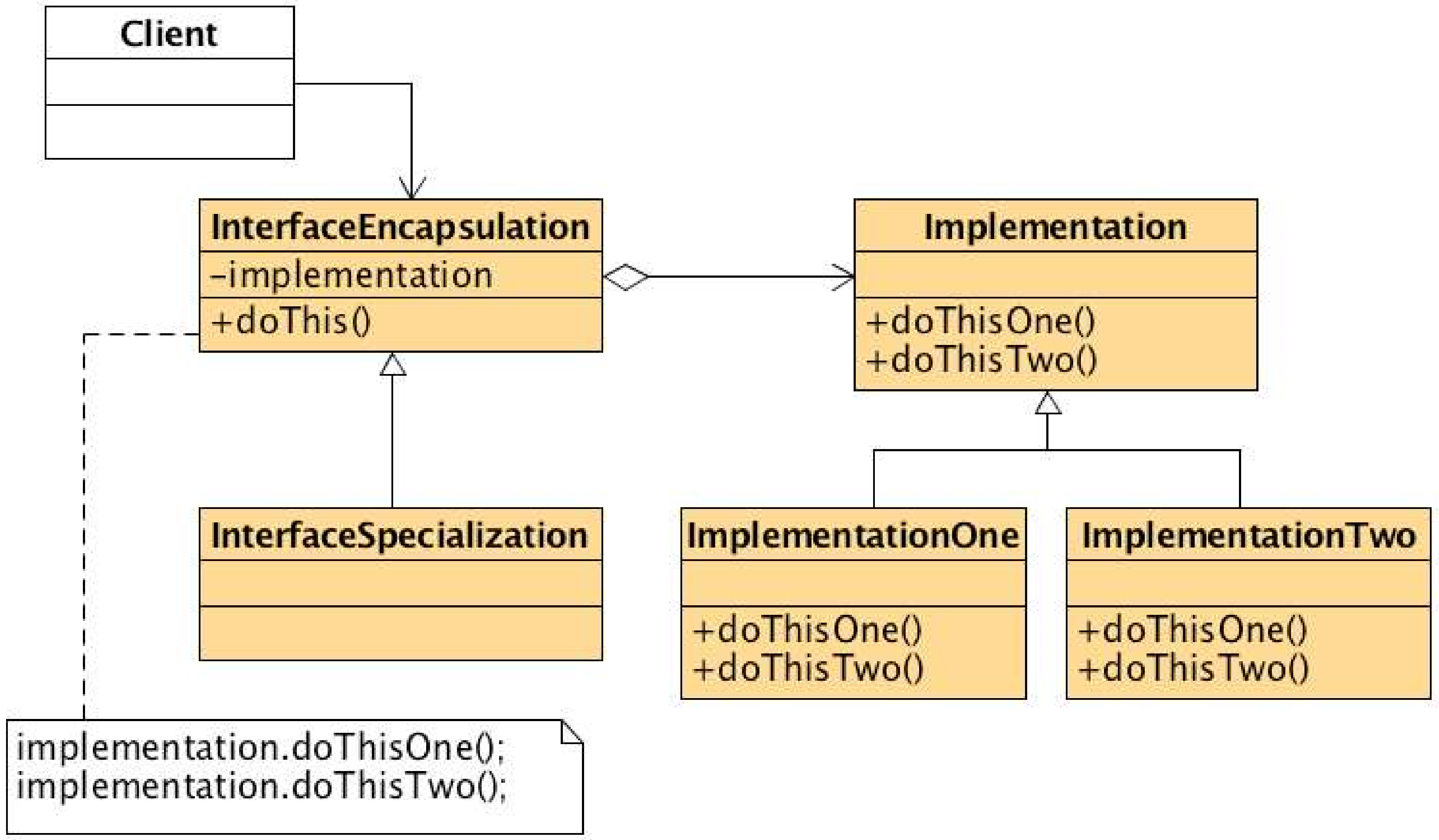


# Bridge – the solution

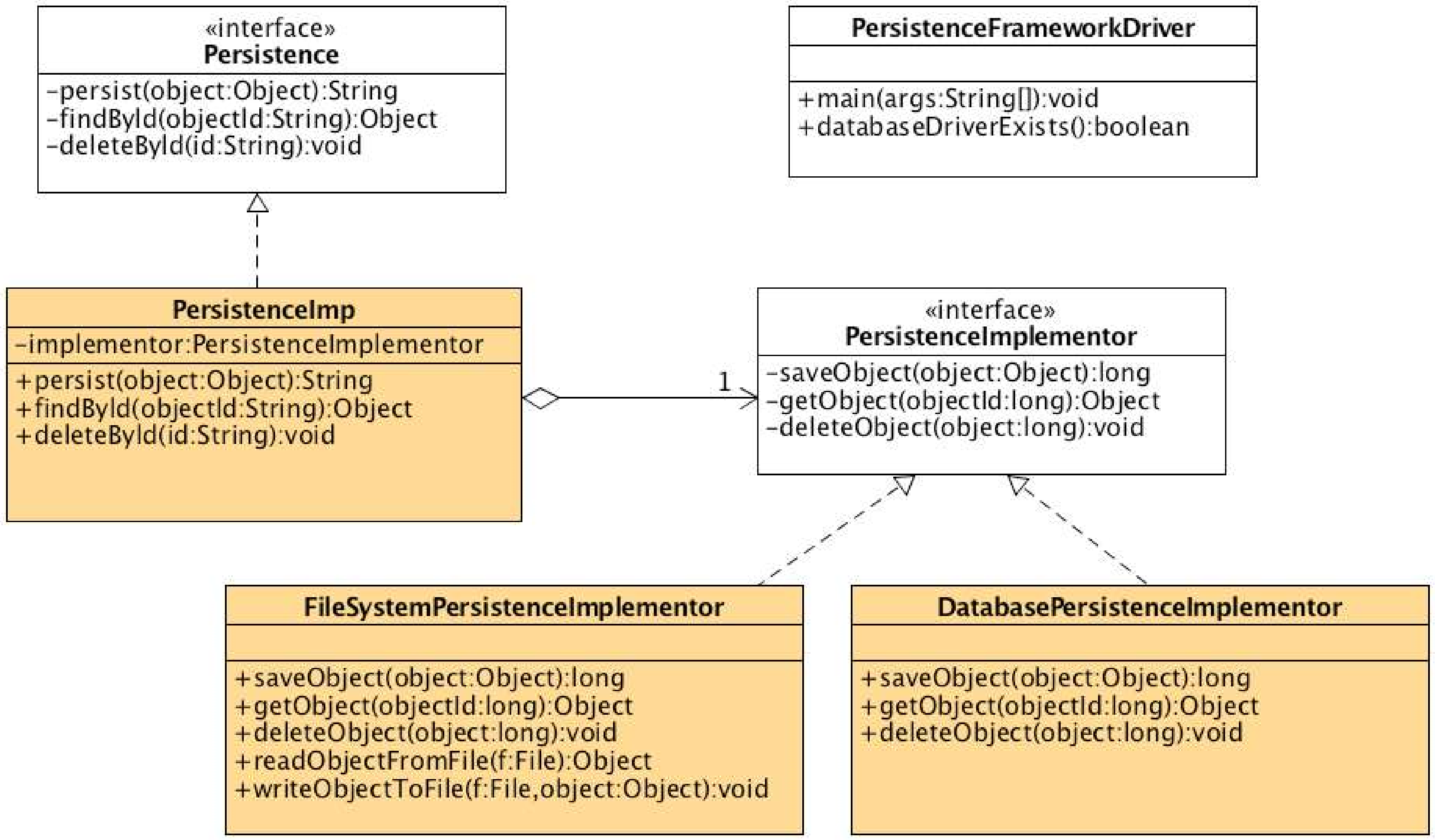


```
while(from.after(clock.getTie())) { /* wait */ }
start();
while(to.after(clock.getTime())) { /* wait */ }
stop();
```

## Structure


Client
InterfaceEncapsulation
-implementation
+doThis()
Implementation
+doThisOne()
+doThisTwo()
InterfaceSpecialization
ImplementationOne
+doThisOne()
+doThisTwo()
ImplementationTwo
+doThisOne()
+doThisTwo()
implementation.doThisOne();
implementation.doThisTwo();

# Example (1)


«interface»
Persistence
-persist(object:Object):String
-findById(objectId:String):Object
-deleteById(id:String):void
PersistenceFrameworkDriver
+main(args:String[]):void
+databaseDriverExists():boolean
PersistenceImp
-implementor:PersistenceImplementor
+persist(object:Object):String
+findById(objectId:String):Object
+deleteById(id:String):void
1
«interface»
PersistenceImplementor
-saveObject(object:Object):long
-getObject(objectId:long):Object
-deleteObject(object:long):void
FileSystemPersistenceImplementor
+saveObject(object:Object):long
+getObject(objectId:long):Object
+deleteObject(object:long):void
+readObjectFromFile(f:File):Object
+writeObjectToFile(f:File,object:Object):void
DatabasePersistenceImplementor
+saveObject(object:Object):long
+getObject(objectId:long):Object
+deleteObject(object:long):void

## Example (1) – Client

```java
public class PersistenceFrameworkDriver {
    public static void main(String[] args) {

        PersistenceImplementor implementor = null;
        if(databaseDriverExists()) {
            implementor = new DabatasePersistenceImplementor();
        } else {
            implementor = new FileSystemPersistenceImplementor();
        }
        Persistence persistenceAPI = new PersistenceImp(implementor);
        Object o = persistenceAPI.findById("12343755");
        // do changes to the object ... then persist
        persistenceAPI.persist(o);
        // can also change implementor
        persistenceAPI = new PersistenceImp(
                new DabatasePersistenceImplementor());
        persistenceAPI.deleteById("2323");
    }
}
```

## Example (2) – Client

```java
public class BridgeDemo {

    public static void main(String[] args) {

        Vehicle vehicle = new BigBus(new SmallEngine());
        vehicle.drive();
        vehicle.setEngine(new BigEngine());
        vehicle.drive();

        vehicle = new SmallCar(new SmallEngine());
        vehicle.drive();
        vehicle.setEngine(new BigEngine());
        vehicle.drive();

    }

}
```

Vehicle and Engine can evolve independently!
How to model this?

## Check list

- Decide if **two orthogonal dimensions** exist in the domain (e.g. abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation).
- Design the **separation of concerns**: what does the client want, and what do the platforms provide.
- Design a platform-oriented interface that is minimal, necessary, and sufficient.
- Define a derived class of that interface for each platform.
- Create the **abstraction base class** that "has a" platform object and delegates the platform-oriented functionality to it.
- Define **specializations** of the abstraction class if desired.

UNIVERSIDADE DE AVEIRO

## Structural design patterns

Class

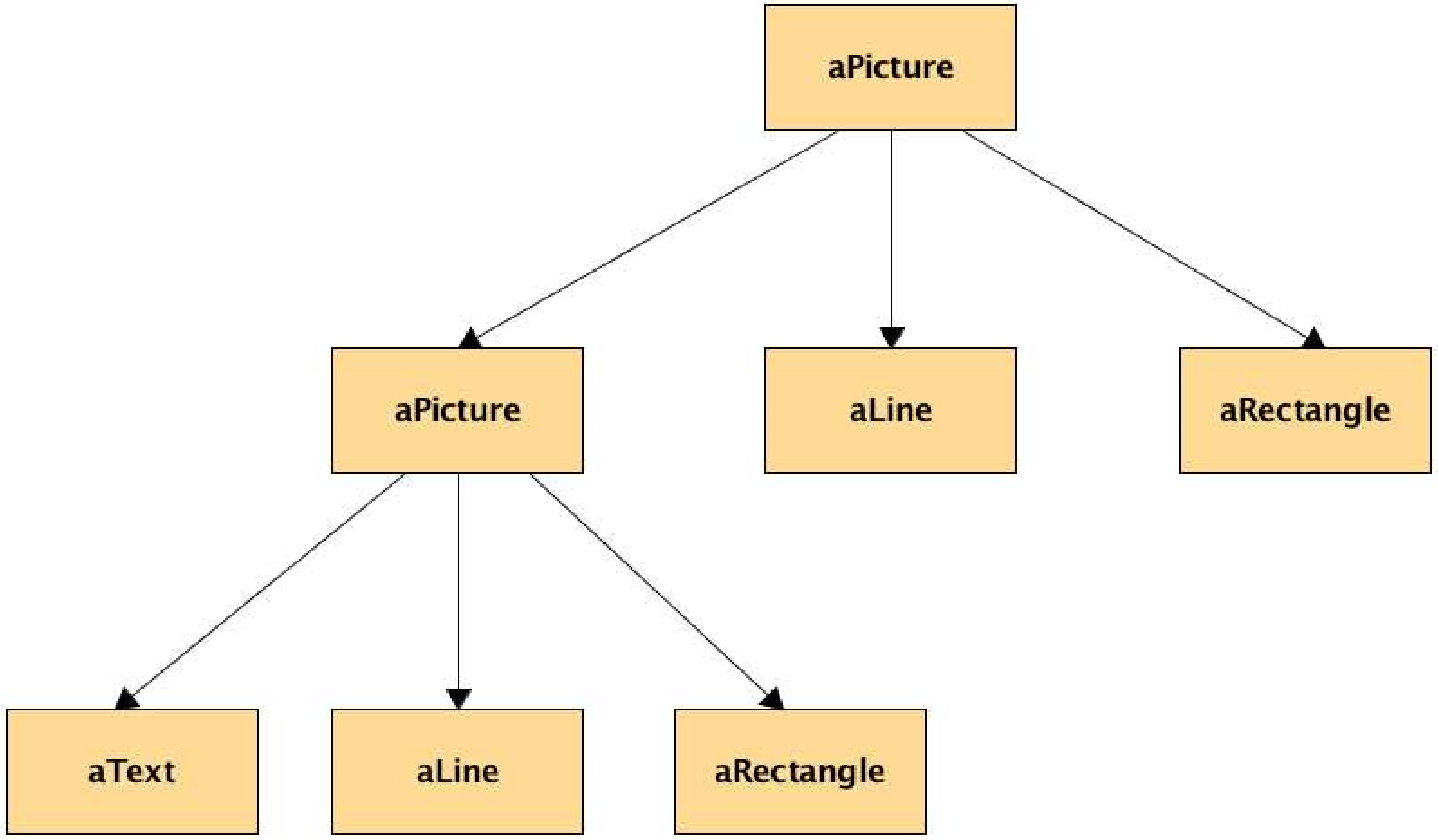
- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

UNIVERSIDADE DE AVEIRO

## Motivation



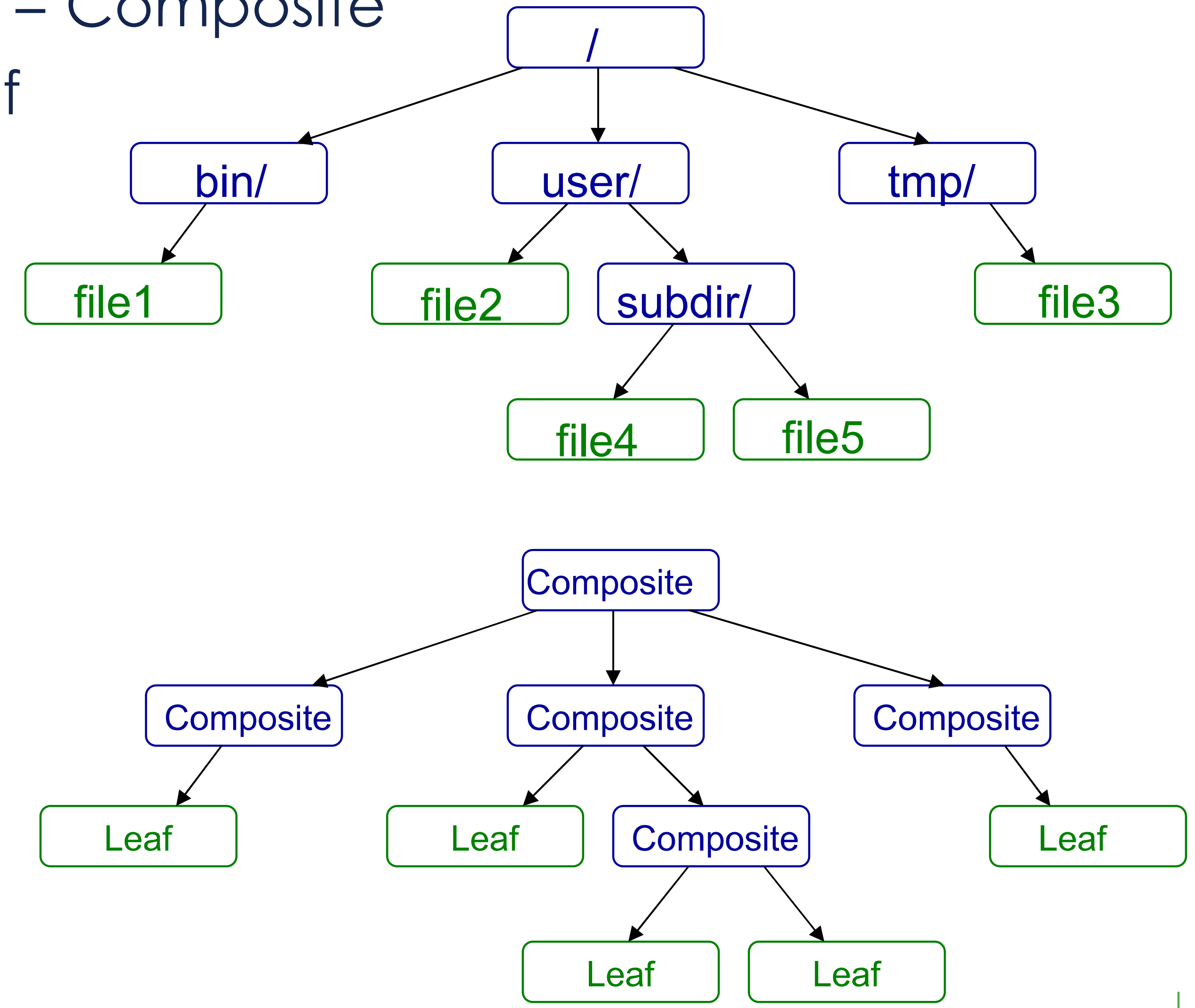
UNIVERSIDADE DE AVEIRO

## Motivation

- **Intent**
  - Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  - Recursive composition
  - "Directories contain entries, each of which could be a directory."
  - 1-to-many "has a" up the "is a" hierarchy
- **Problem**
  - Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

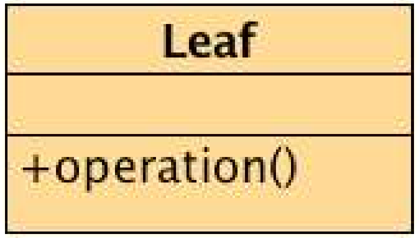
## Directory / File Example

- Directory = Composite
- File = Leaf

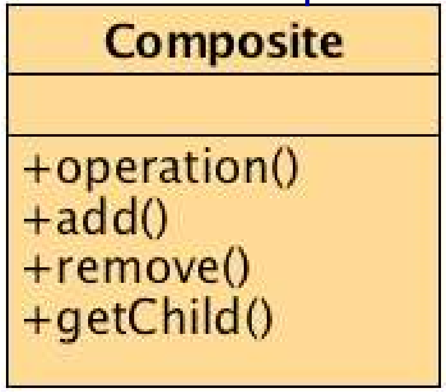


## Directory / File Example – Classes

Leaf Class: File

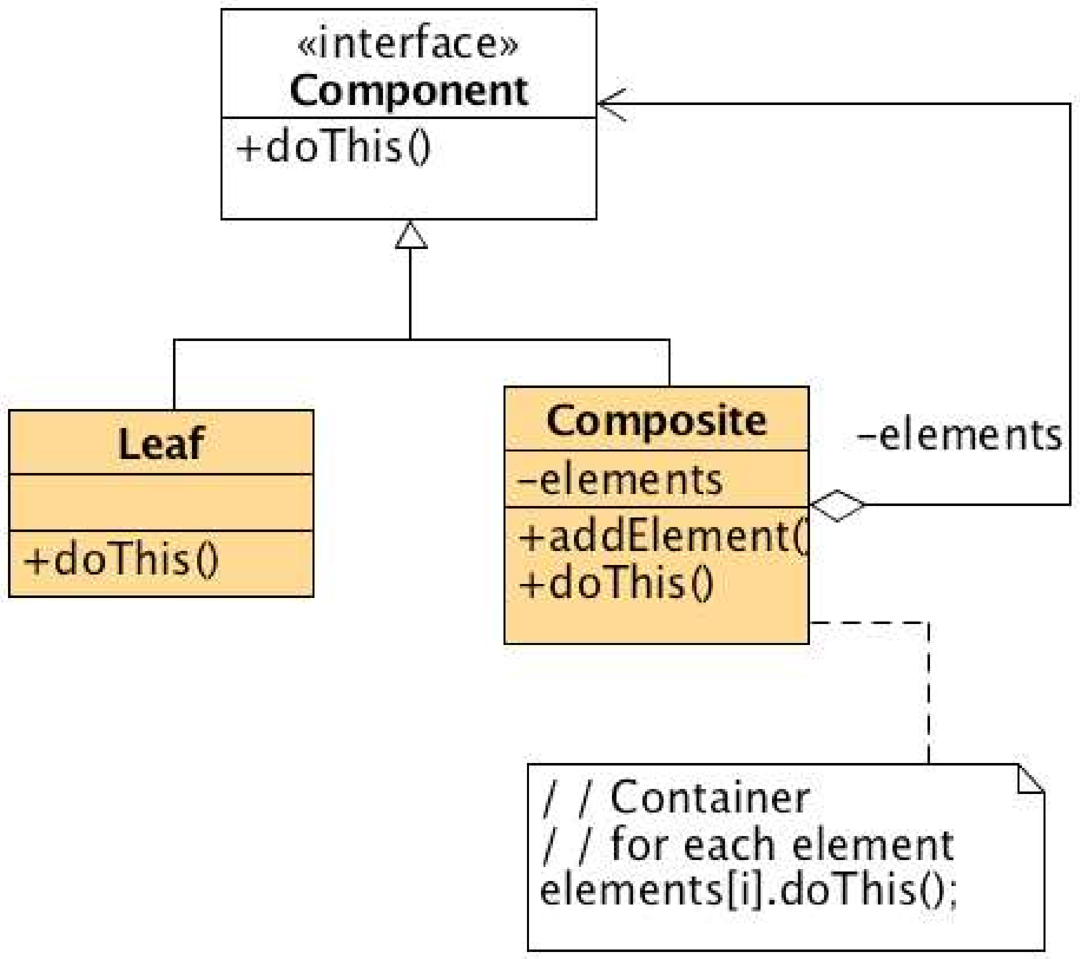


Composite Class: Directory



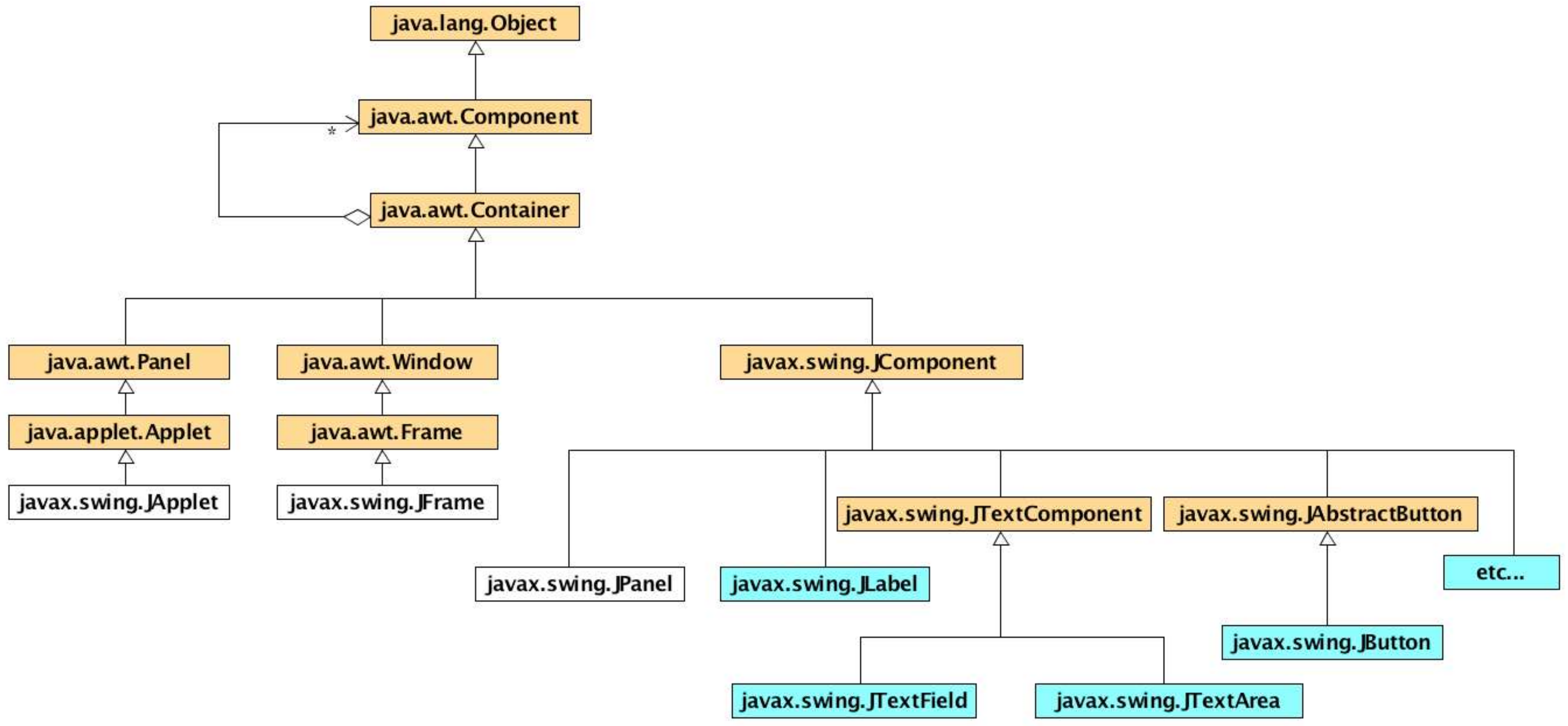
- One class for Files (Leaf nodes)
- One class for Directories (Composite nodes)
  - Collection of Directories and Files
- How do we make sure that Leaf nodes and Composite nodes can be handled uniformly?
  - Derive them from the same abstract base class

## Structure



«interface»
Component
+doThis()
Leaf
+doThis()
Composite
-elements
+addElement(
+doThis()
-elements
/ / Container
/ / for each element
elements[i].doThis();

# Java Swing – where is the composite?



## Example – Entity/Product/Box

```java
abstract class Entity {
    protected static StringBuffer indent = new StringBuffer();
    public abstract void traverse();
}

class Product extends Entity {
    private int value;

    public Product(int val) {
        value = val;
    }

    public void traverse() {
        System.out.println(indent.toString() + value);
    }
}
```

## Example – Entity/Product/Box

```java
class Box extends Entity {
    private List<Entity> children = new ArrayList<>();
    private int value;

    public Box(int val) {
        value = val;
    }

    public void add(Entity c) {
        children.add(c);
    }

    public void traverse() {
        System.out.println(indent.toString() + value);
        indent.append("   ");
        for (int i = 0; i < children.size(); i++)
            children.get(i).traverse();
        indent.setLength(indent.length() - 3);
    }
}
```

## Example – Entity/Product/Box

```java
public class CompositeLevels {
    public static void main(String[] args) {
        Box root = initialize();   root.traverse();
    }
    private static Box initialize() {
        Box[] nodes = new Box[7];
        nodes[1] = new Box(1);
        int[] s = { 1, 4, 7 };
        for (int i = 0; i < 3; i++) {
            nodes[2] = new Box(21 + i);
            nodes[1].add(nodes[2]);
            int lev = 3;
            for (int j = 0; j < 4; j++) {
                nodes[lev - 1].add(new Product(lev * 10 + s[i]));
                nodes[lev] = new Box(lev * 10 + s[i] + 1);
                nodes[lev - 1].add(nodes[lev]);
                nodes[lev - 1].add(new Product(lev * 10 + s[i] + 2));
                lev++;
            }
        }
        return nodes[1];
    }
}
```

```
1
   21
      31
      32
         41
         42
            51
            52
               61
               62
               63
            53
         43
      33
   22
      34
      35
         44
         45
            54
            55
               64
               65
               66
            56
         46
      36
   23
      37
      38
         47
         48
            57
            58
               67
               68
               69
            59
         49
      39
```

## Check list

- Ensure that your problem is about representing "**whole-part" hierarchical relationships**.
- Consider the heuristic, "Containers that contain containees, each of which could be a container."
- Create a "**lowest common denominator**" interface that makes your containers and containees interchangeable.
- All container and containee classes declare an "is a" relationship to the interface.
- All container classes declare a one-to-many "has a" relationship to the interface.
- Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class.

UNIVERSIDADE DE AVEIRO

## Structural design patterns

Class

- Adapter

Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

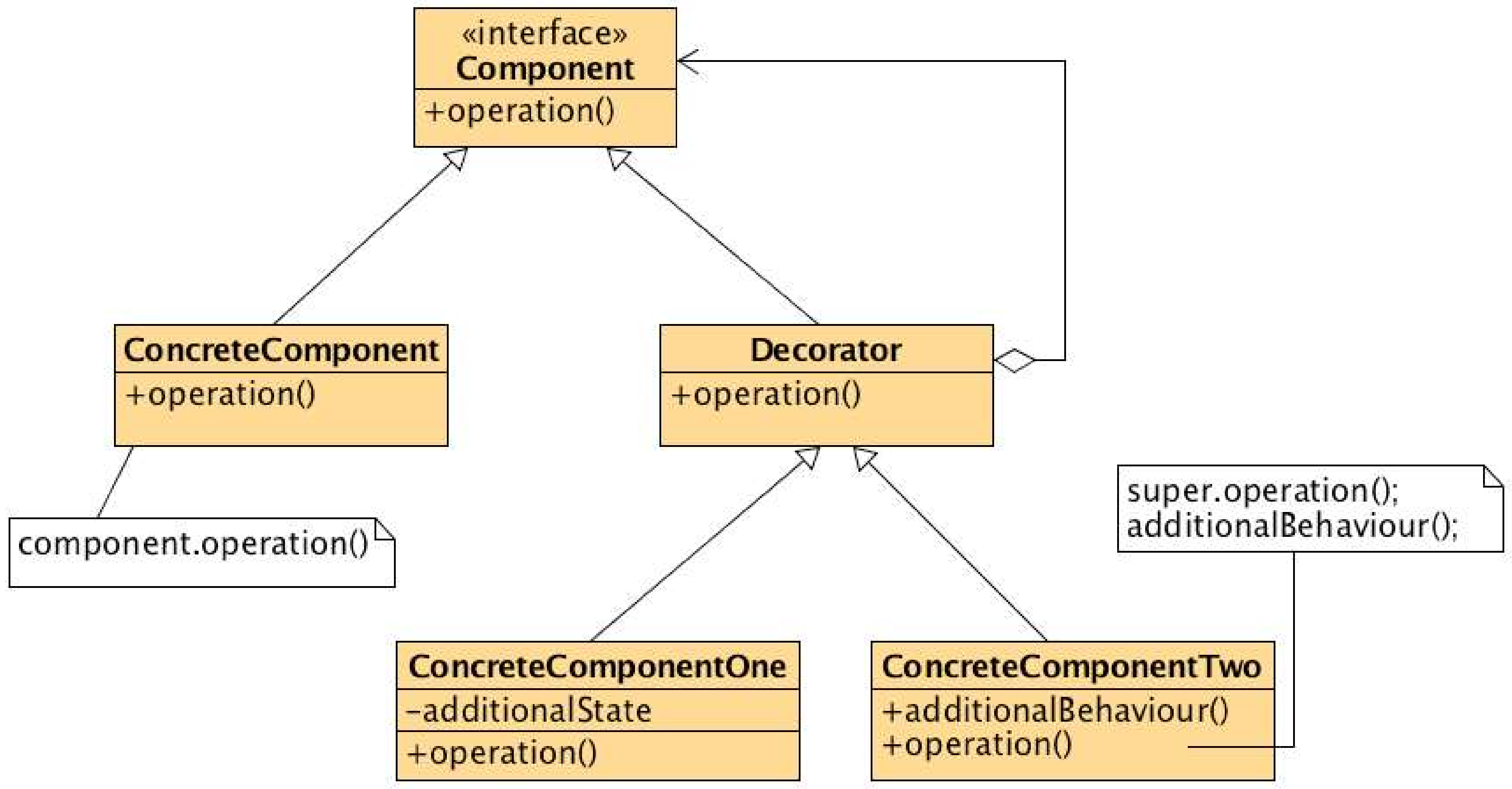
## Decorator

- **Intent**
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
  - Client-specified embellishment of a core object by recursively wrapping it.
  - Wrapping a gift, putting it in a box, and wrapping the box.
- **Problem**
  - You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

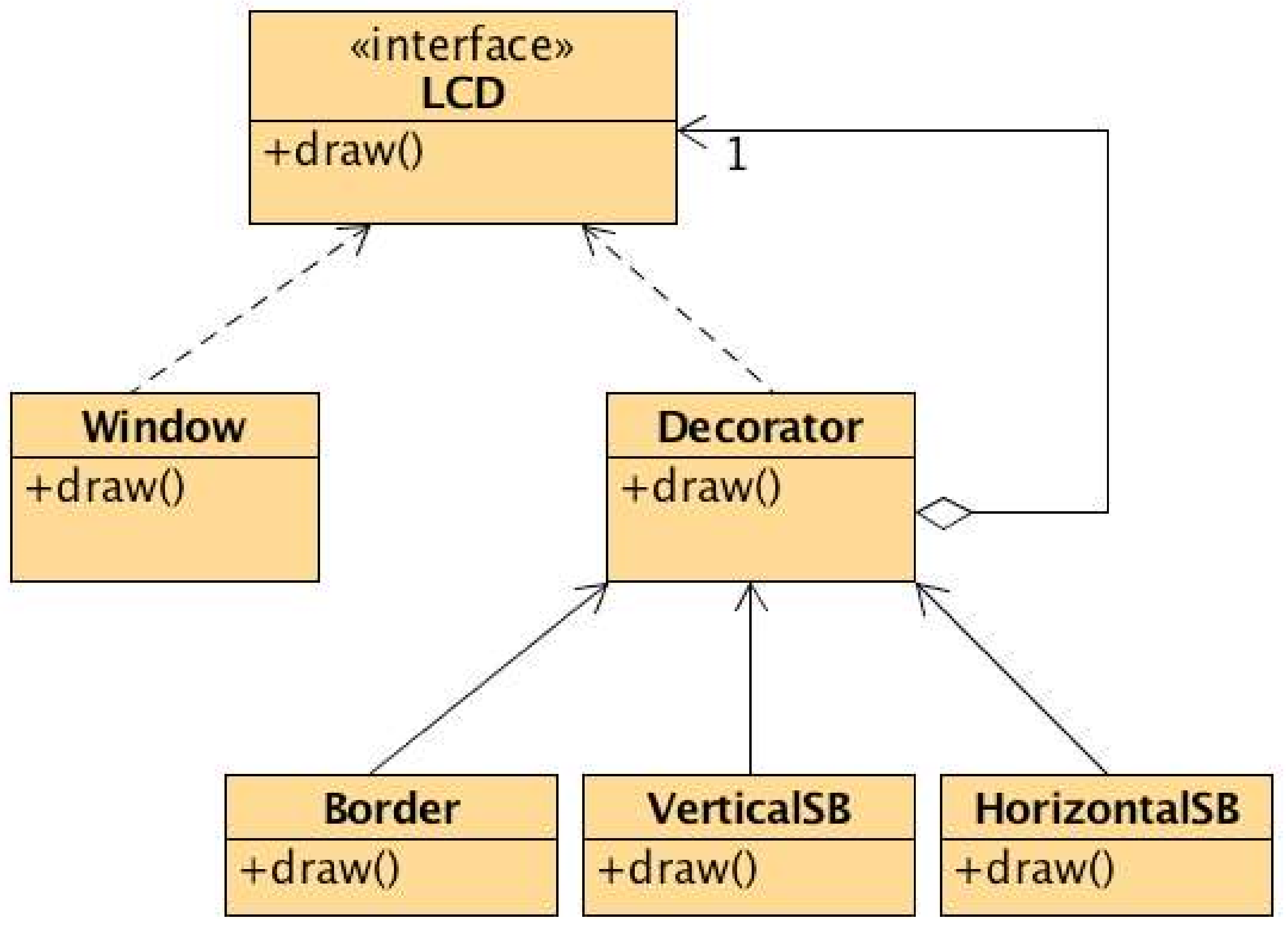
## Motivation

- Consider the following entities:
  - Futebolista (joga, passa, remata),
  - Tenista (joga, serve),
  - Jogador (joga)
- Let's complicate:
  - O Rui joga Basquete e Futebol
  - A Ana joga Badminton e Basquete
  - O Paulo joga Xadrez, Futebol e Basquete
- .... Solution?

## Structure


«interface»
Component
+operation()
ConcreteComponent
+operation()
Decorator
+operation()
component.operation()
super.operation();
additionalBehaviour();
ConcreteComponentOne
-additionalState
+operation()
ConcreteComponentTwo
+additionalBehaviour()
+operation()

# Structure – Example



## Example

```java
interface JogadorInterface {
    void joga();
}

class Jogador implements JogadorInterface {
    private String name;
    Jogador(String n) { name = n; }
    @Override public void joga()
        { System.out.print("\n"+name+" joga "); }
}

abstract class JogDecorator implements JogadorInterface {
    protected JogadorInterface j;
    JogDecorator(JogadorInterface j) { this.j = j; }
    public void joga()       { j.joga(); }
}
```

## Example

```java
class Futebolista extends JogDecorator {
    Futebolista(JogadorInterface j) { super(j); }
    @Override public void joga()
        { j.joga(); System.out.print("futebol ");  }
    public void remata()  { System.out.println("-- Remata!"); }
}

class Xadrezista extends JogDecorator {
    Xadrezista(JogadorInterface j) { super(j); }
    @Override public void joga()  { j.joga();
        System.out.print("xadrez ");  }
}

class Tenista extends JogDecorator {
    Tenista(JogadorInterface j) { super(j); }
    @Override public void joga()
        { j.joga(); System.out.print("tenis ");  }
    public void serve()  { System.out.println("-- Serve!"); }
}
```

## Example

```java
public class PlayTest{
    public static void main(String args[]) {
        JogadorInterface j1 = new Jogador("Rui");
        Futebolista f1 = new Futebolista(new Jogador("Luis"));
        Xadrezista x1 = new Xadrezista(new Jogador("Ana"));
        Xadrezista x2 = new Xadrezista(j1);
        Xadrezista x3 = new Xadrezista(f1);
        Tenista t1 = new Tenista(j1);
        Tenista t2 = new Tenista(
                new Xadrezista(
                        new Futebolista(
                                        new Jogador("Bruna"))));
        JogadorInterface lista[] = { j1, f1, x1, x2, x3, t1, t2 };
        for (JogadorInterface ji: lista)
            ji.joga();
    }
}
```

```
Rui joga
Luis joga futebol
Ana joga xadrez
Rui joga xadrez
Luis joga futebol xadrez
Rui joga ténis
Bruna joga futebol xadrez ténis
```

## Decorator example: Java I/O

- *InputStream* class has only public *int read()* method to read one letter at a time
- decorators such as *BufferedReader* or *Scanner* add additional functionality to read the stream more easily

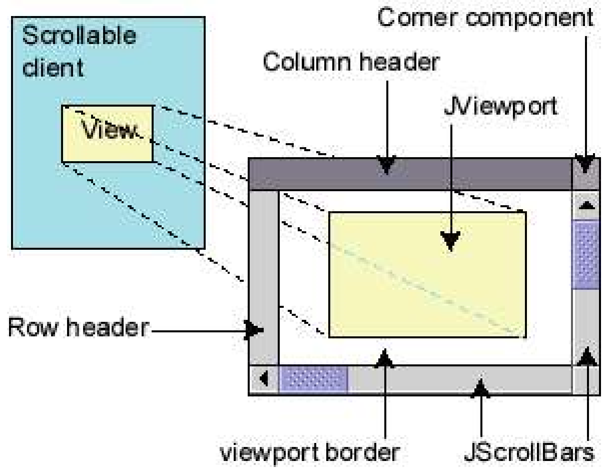
```
// InputStreamReader/BufferedReader decorate InputStream
InputStream in = new FileInputStream("hardcode.txt");
InputStreamReader isr = new InputStreamReader(in);
BufferedReader br = new BufferedReader(isr);

// because of decorator streams, we can read an
// entire line from the file in one call
// (InputStream only provides public int read() )
String wholeLine = br.readLine();
```

## Decorator example: GUI

- Common GUI components don't have scroll bars
- *JScrollPane* is a container with scroll bars to which we can add any component to make it scrollable

```
// JScrollPane decorates GUI components
JTextArea area = new JTextArea(20, 30);
JScrollPane scrollPane =
  new JScrollPane(area);
contentPane.add(scrollPane);
```



## Exercise

**Create the required classes to the following main function**

```
public class TestDecorator {

   public static void main(String args[]) {
        Icecream icecream =
          new HoneyDecorator(
             new NuttyDecorator(
                   new SimpleIcecream()));
        System.out.println(icecream.makeIcecream());
   }

}
```

```
Base Icecream + cruncy nuts + sweet honey
```

## Check list

- Ensure the context is: a single **core** (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all.
- Create a "Lowest Common Denominator" **interface** that makes all classes interchangeable.
- Create a second level base class (**Decorator**) to support the optional wrapper classes.
- The Core class and Decorator class inherit from the interface.
- The Decorator class declares a composition relationship to the interface, and this data member is initialized in its constructor.
- Define a Decorator derived class for each optional embellishment.

## Structural design patterns

Class

- Adapter

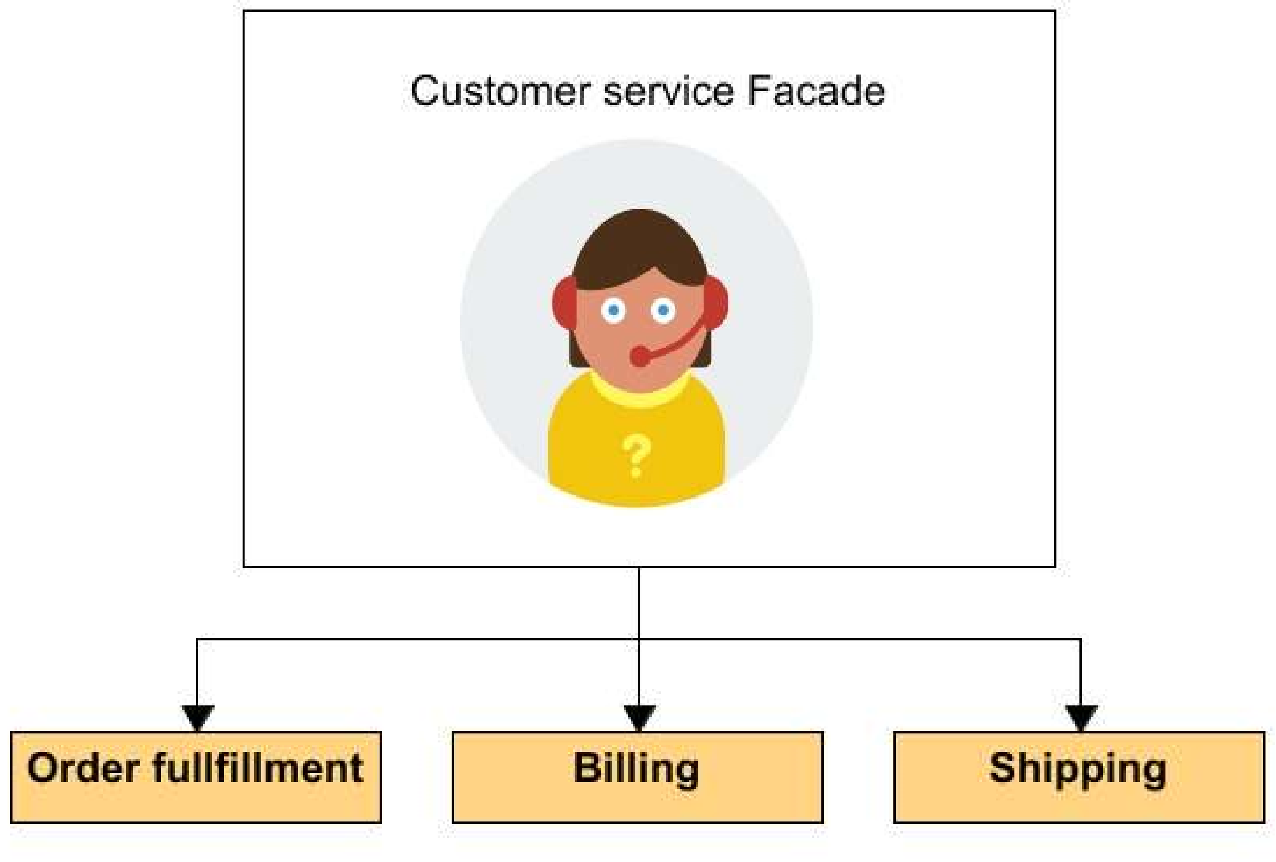
Object

- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

## Motivation

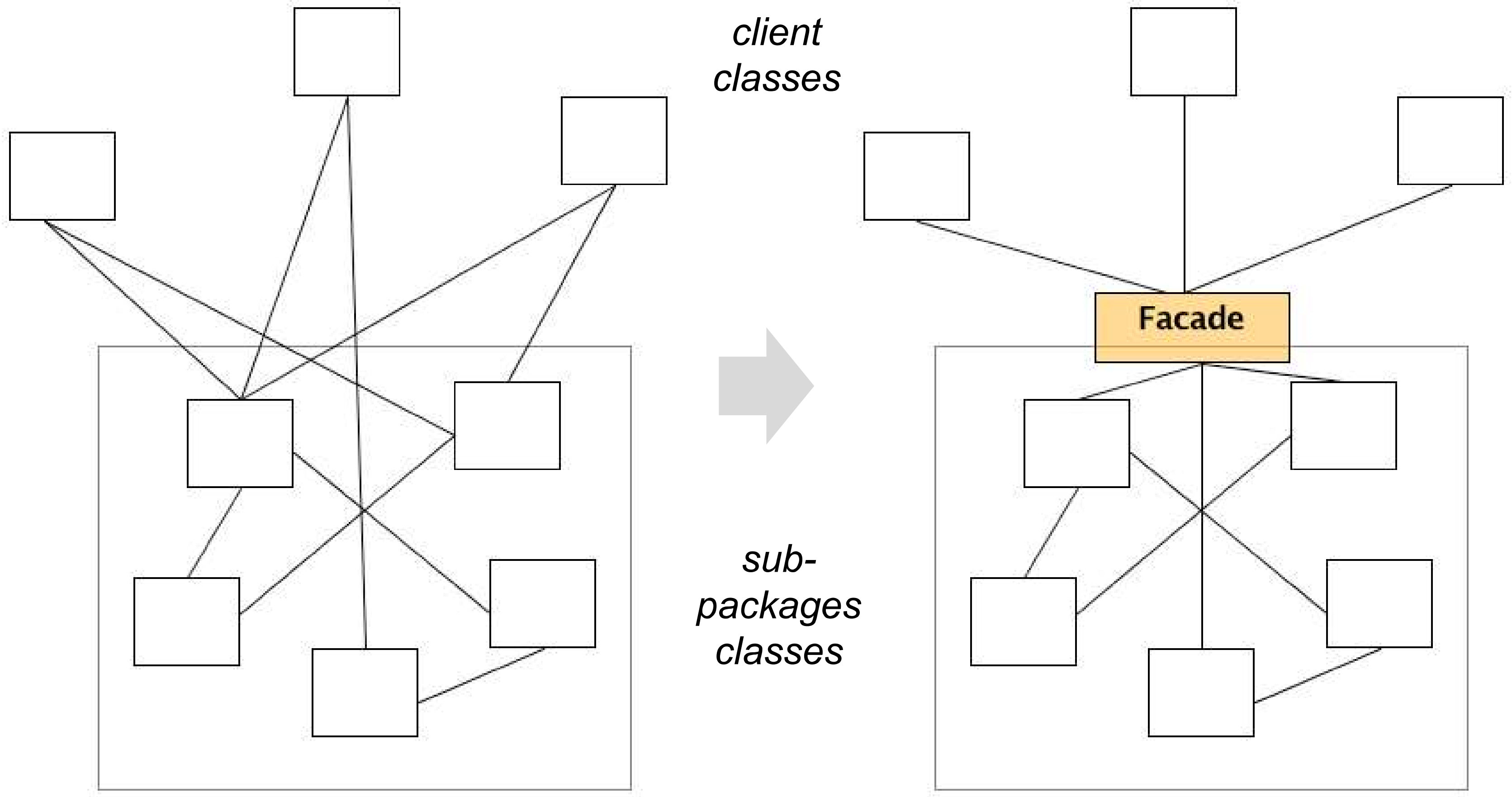
- **Problem**
  - A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.
- **Intent**
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
  - Wrap a complicated subsystem with a simpler interface.

## Motivation

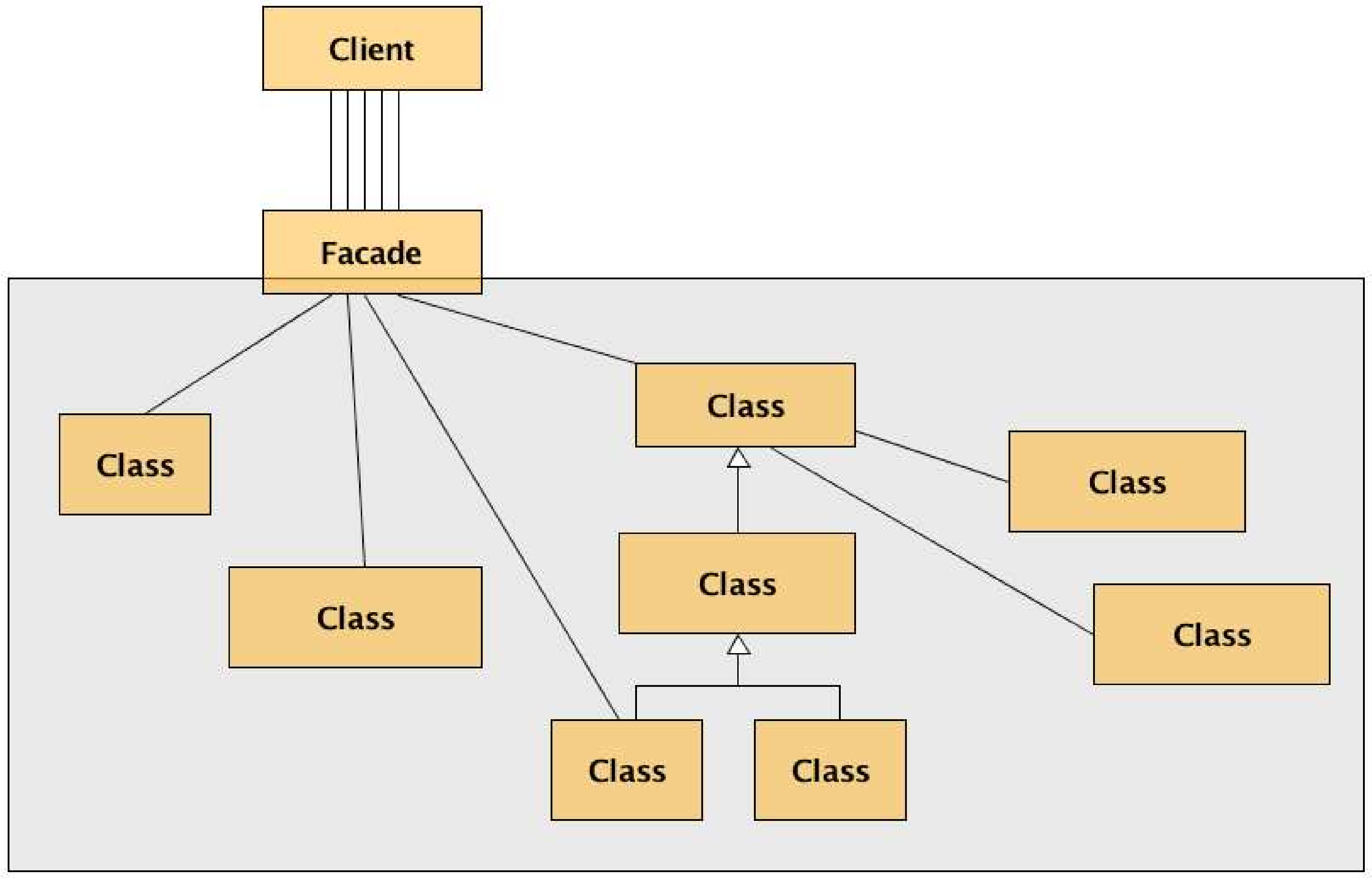


UNIVERSIDADE DE AVEIRO

# Motivation



## Structure



Client
Facade
Class
Class
Class
Class
Class
Class
Class
Class
Class

## Example

```java
// ...
class TravelFacade {
    private HotelBooker hotelBooker;
    private FlightBooker flightBooker;
    private LocalTourBooker tourBooker;
    public void getFlightsAndHotels(City dest, Date from, Data to) {
        List<Flight> flights = flightBooker.getFlightsFor(dest, from, to);
        List<Hotel> hotels = hotelBooker.getHotelsFor(dest, from, to);
        List<Tour> tours = tourBooker.getToursFor(dest, from, to);
        // process and return
    }
}

public class FacadeDemo {
    public static void main(String[] args) {
        TravelFacade facade = new TravelFacade();
        facade.getFlightsAndHotels(destination, from, to);
    }
}
```

UNIVERSIDADE DE AVEIRO

## Consequences

- Benefits
  - It hides the implementation of the subsystem from clients, making the subsystem easier to use
  - It promotes weak coupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.
  - It reduces compilation dependencies in large software systems
- It does not add any functionality, it just simplifies interfaces
- It does not prevent clients from accessing the underlying classes

## Structural design patterns

Class

- Adapter

Object

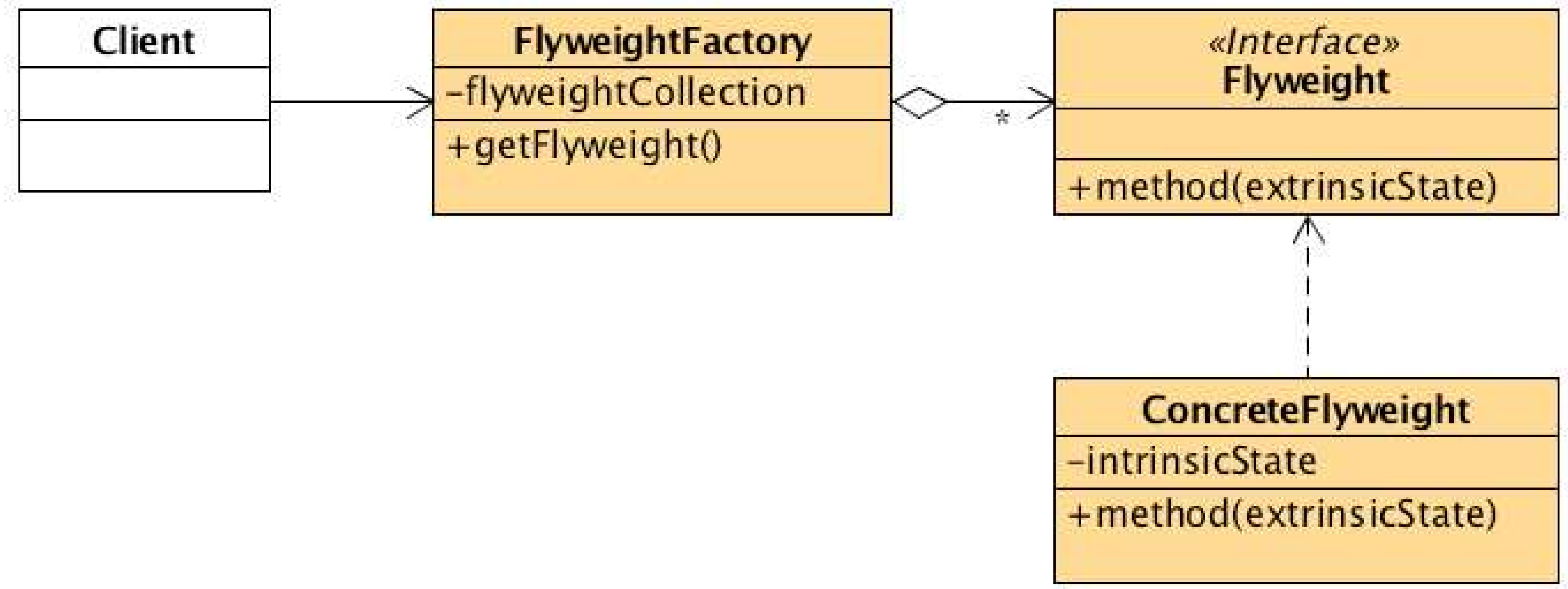
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

UNIVERSIDADE DE AVEIRO

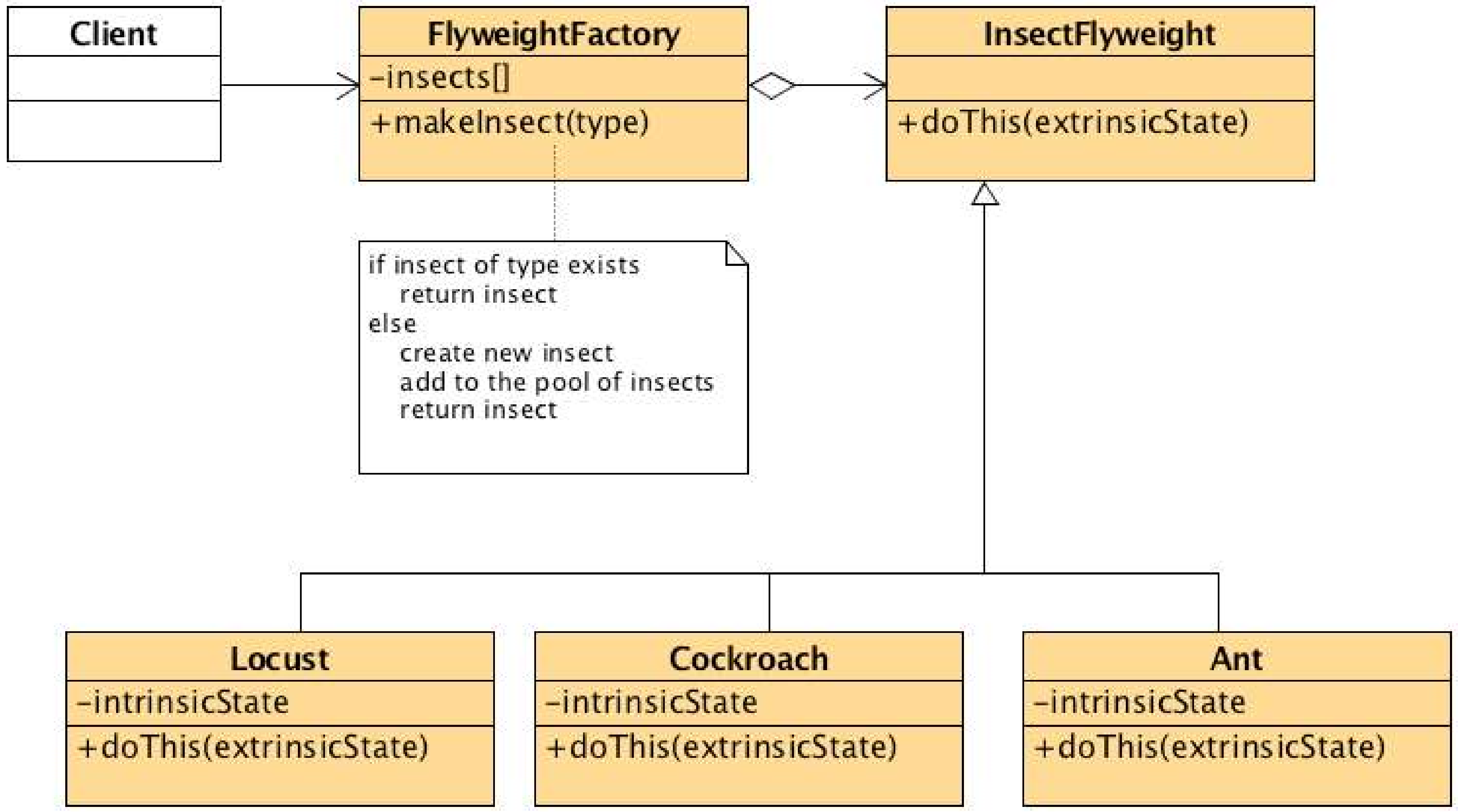
## Motivation

- **Intent**
  - Use sharing to support large numbers of fine-grained objects efficiently.
  - The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.
- **Problem**
  - Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

## Structure


Client
FlyweightFactory
-flyweightCollection
+getFlyweight()
*
«Interface»
Flyweight
+method(extrinsicState)
ConcreteFlyweight
-intrinsicState
+method(extrinsicState)

## Structure - example



```
if insect of type exists
    return insect
else
    create new insect
    add to the pool of insects
    return insect
```

## Example (java.lang.Integer:valueOf)

```
public final class Integer extends Number implements Comparable<Integer> {

// ...
public static Integer valueOf(int i) {
    final int offset = 128;
    if (i >= -128 && i <= 127) { // must cache
        return IntegerCache.cache[i + offset];
    }
    return new Integer(i);
}

// ...
private static class IntegerCache {
     static final Integer cache[] = new Integer[-(-128) + 127 + 1];
     static {
        for(int i = 0; i < cache.length; i++)
             cache[i] = new Integer(i - 128);
     }
}
```

## Example – web browser cache

Browser loads images just once and then reuses them from pool:

## Check list

- Ensure that object overhead is an issue needing attention.
- Divide the target class's state into: shareable (intrinsic) state, and non-shareable (extrinsic) state.
- Remove the non-shareable state from the class attributes, and add it the calling argument list of affected methods.
- Create a Factory that can cache and reuse existing class instances.
- The client must use the Factory instead of the new operator to request objects.

## Structural design patterns

Class

- Adapter

Object

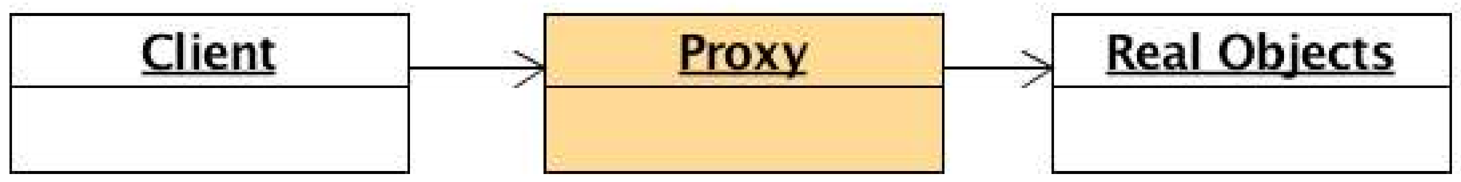
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

## Motivation

- **Problem**
  - You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.
- **Intent**
  - Provide a surrogate or placeholder for another object to control access to it.
  - Use an extra level of indirection to support distributed, controlled, or intelligent access.
  - Add a wrapper and delegation to protect the real component from undue complexity.

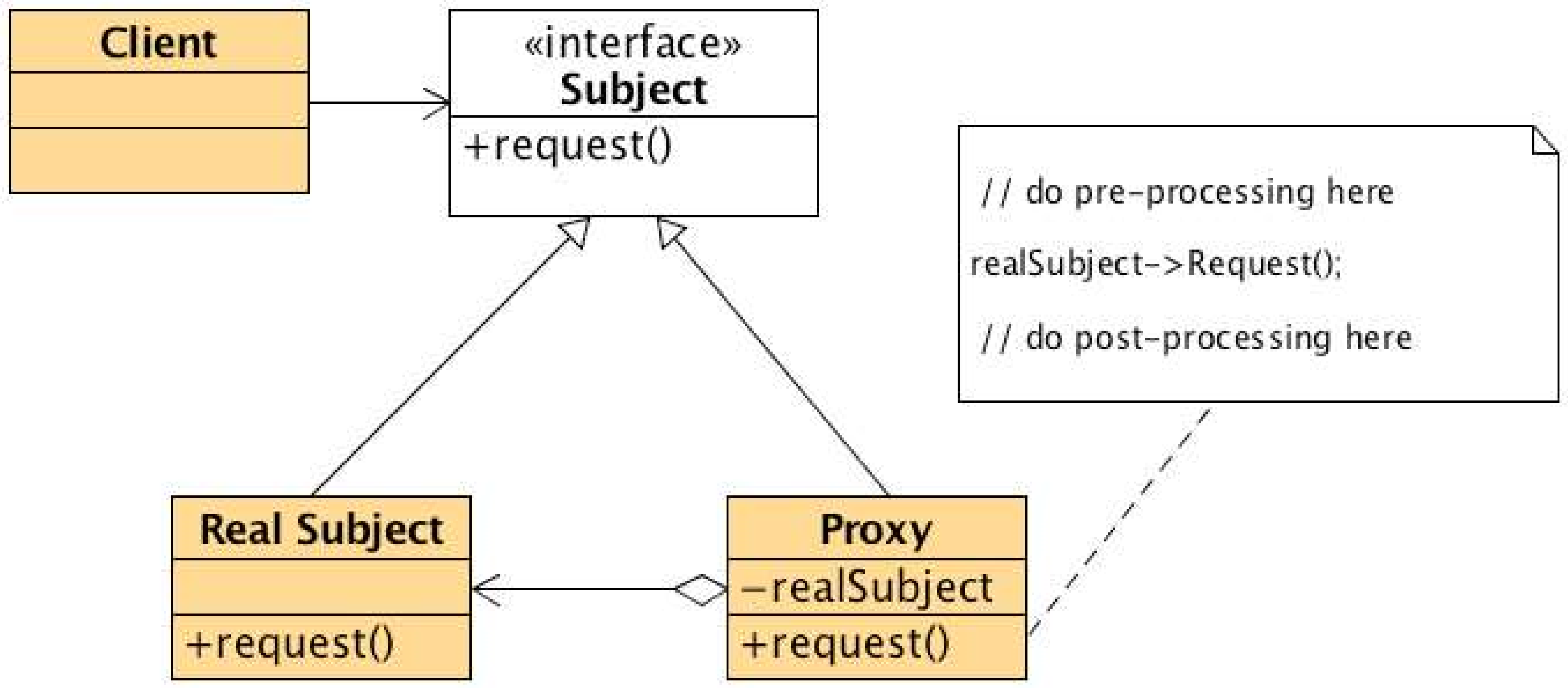
## Solution

- Create a Proxy object that implements the same interface as the real object
  - The Proxy object (usually) contains a reference to the real object
  - Clients are given a reference to the Proxy, not the real object
  - All client operations on the object pass through the Proxy, allowing the Proxy to perform additional processing



UNIVERSIDADE DE AVEIRO

## Structure


Client
«interface»
Subject
+request()
// do pre-processing here
realSubject->Request();
// do post-processing here
Real Subject
+request()
Proxy
−realSubject
+request()

## Consequences

- Provides an additional level of indirection between client and object that may be used to insert arbitrary services
- Proxies are invisible to the client, so introducing proxies does not affect client code

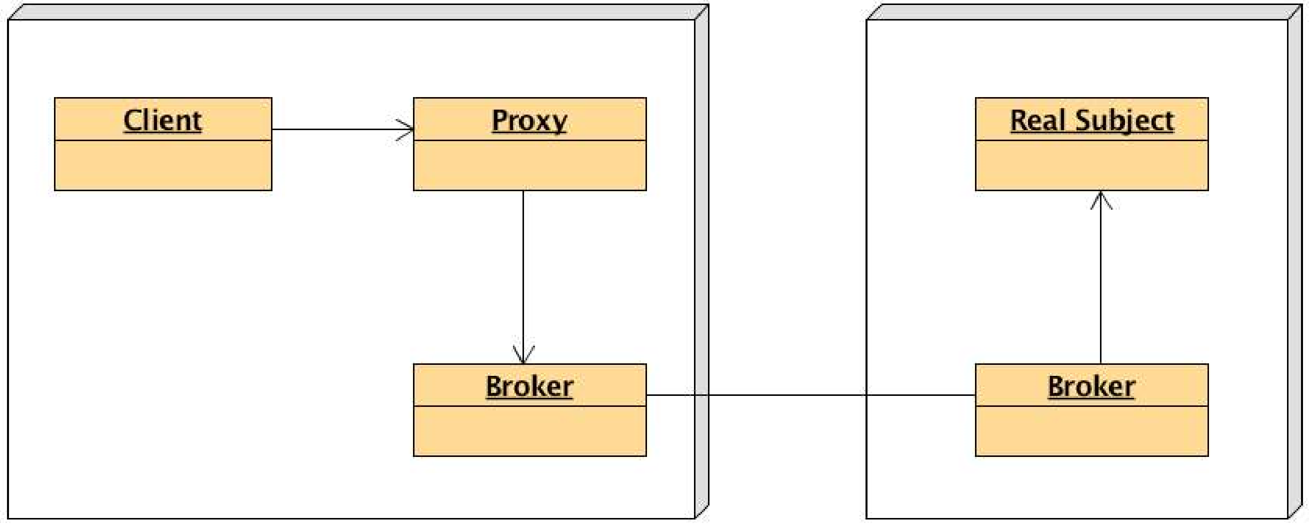
UNIVERSIDADE DE AVEIRO

## Known Uses: Java Collections

- Read-only Collections
  - Wrap collection object in a proxy that only allows read-only operations to be invoked on the collection
  - All other operations throw exceptions
  - List Collections.unmodifiableList(List list);
    - Returns read-only List proxy
- Synchronized Collections
  - Wrap collection object in a proxy that ensures only one thread at a time is allowed to access the collection
  - Proxy acquires lock before calling a method, and releases lock after the method completes
  - List Collections.synchronizedList(List list);
    - Returns a synchronized List proxy

## Known Uses: Distributed Objects

- The Client and Real Subject are in different processes or on different machines, and so a direct method call will not work
- The Proxy's job is to pass the method call across process or machine boundaries, and return the result to the client (with Broker's help)



## Known Uses: Secure Objects

- Different clients have different levels of access privileges to an object
- Clients access the object through a proxy
- The proxy either allows or rejects a method call depending on what method is being called and who is calling it (i.e., the client's identity)

UNIVERSIDADE DE AVEIRO

## Known Uses: Lazy Loading

- Some objects are expensive to instantiate (i.e., consume lots of resources or take a long time to initialize)
- Create a proxy instead, and give the proxy to the client
  - The proxy creates the object on demand when the client first uses it
  - Proxies must store whatever information is needed to create the object on-the-fly (file name, network address, etc.)

## Known Uses: Copy-on-Write

- Multiple clients share the same object as long as nobody tries to change it
- When a client attempts to change the object, they get their own private copy of the object
- Read-only clients continue to share the original object, while writers get their own copies
- Allows resource sharing, while making it look like everyone has their own object
- When a write operation occurs, a proxy makes a private copy of the object on-the-fly to insulate other clients from the changes

## Check list

- ❖ Identify the functionality that is best implemented as a wrapper or surrogate.
- ❖ Define an interface that will make the proxy and the original component interchangeable.
- ❖ Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
- ❖ The wrapper class holds a pointer to the real class and implements the interface.

## Structural design patterns

Class

- Adapter

Object

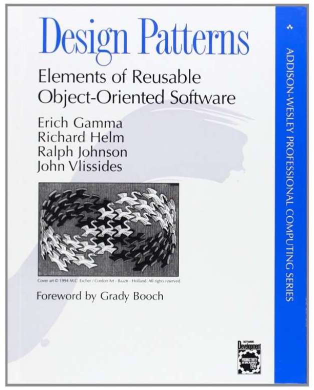
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

## Structural patterns – Summary

- Adapter
  - Match interfaces of different classes
- Bridge
  - Separates an object’s interface from its implementation
- Composite
  - A tree structure of simple and composite objects
- Decorator
  - Add responsibilities to objects dynamically
- Facade
  - A single class that represents an entire subsystem
- Flyweight
  - A fine-grained instance used for efficient sharing
- Proxy
  - An object representing another object

## Resources

- Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

- *Design Patterns Explained Simply (sourcemaking.com)*

UNIVERSIDADE DE AVEIRO