CSCI E-97
Lecture 8 – Structural Patterns

October 23, 2014

Announcements

- Assignment 2
  - Grades returned tuesday night
- Assignment 3
  - Peer design reviews should be complete
  - Implementation under way
  - Post questions to discussion forum (and review what is there already)
  - Due Thursday at midnight
  - Questions?

Structural Patterns

- Introduce the category of Structural Patterns
- Cover the following Structural Patterns
  - Adapter Pattern
  - Bridge Pattern
  - Decorator Pattern
  - Façade Pattern
  - Proxy Pattern

Structural Patterns

- Often a single object can have many parts, or be an entry point into a network of related objects
- Structural Patterns address ways to compose classes and objects to form larger structures
- The Composite Pattern is a Structural Pattern

Examples of Structural Design Questions

- How can I design my Human Interface Objects so that I can implement them to use one or more database management systems (see the Bridge Pattern)
- I have a set of classes that adhere to our standard interface guidelines. How can I add a third-party class with desirable functionality so that it fits into these interface guidelines (see the Adapter Pattern)

Examples of Structural Design Questions – 2

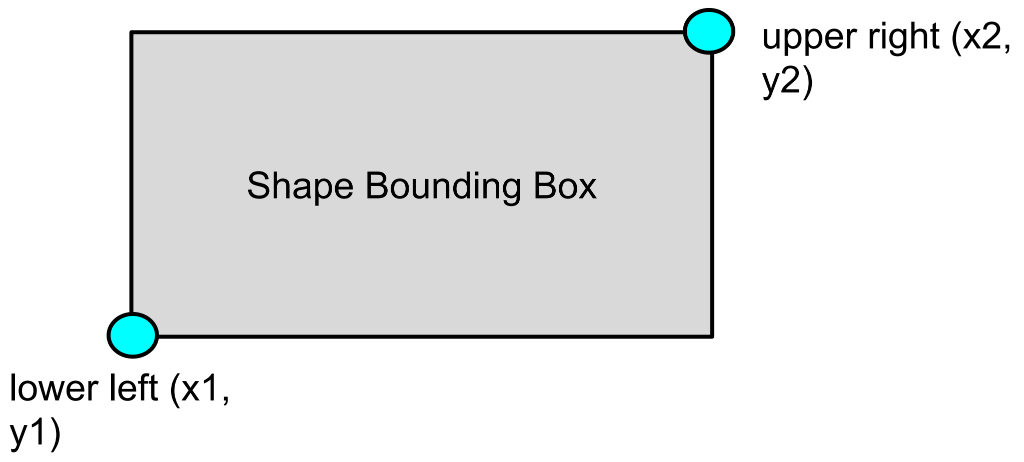
- I have to partition my system so that the business objects (`Customer`, `Account`, etc) will execute in a process in a server machine remote from the user's workstations. How can I provide remote access to these objects (See the Proxy Pattern)
- The classes that make up my business object subsystem are too complex to expose to a client. How can I provide a simpler interface that encapsulates that complexity (see the Façade Pattern)

The Adapter Pattern

- The Problem: We want to convert the interface of a class into another interface that clients expect
- This will enable class interaction that would be inconvenient otherwise

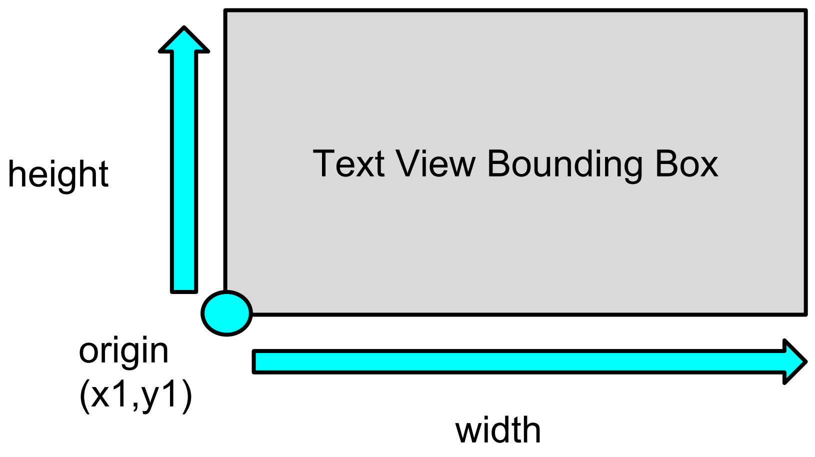
Example – Adding a Class to a Package

- Suppose we have developed a set of classes that support a drawing package
- The base for all interfaces in the package is the `Shape` interface
- Our package describes the bounding box of a shape in terms of its lower left and upper right corners



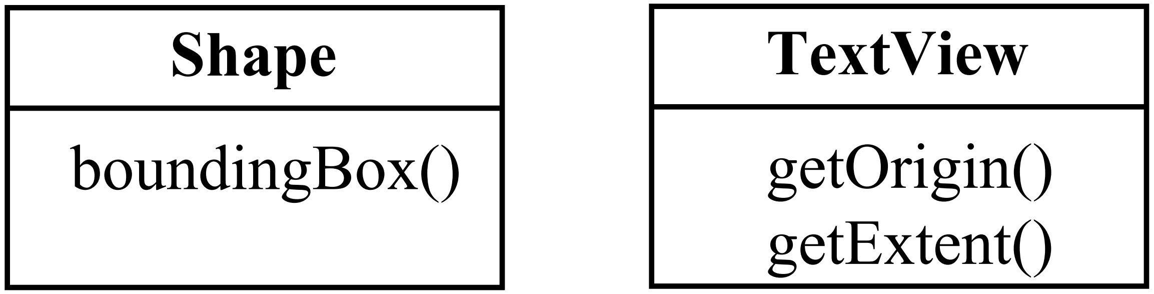
Adapting TextView to Shape

- We want to incorporate a newly purchased class, `TextView`, which has functionality we need in our package of shapes
- However, `TextView` describes its bounding box in terms of its origin (the lower left corner) and its extent (the height and width)



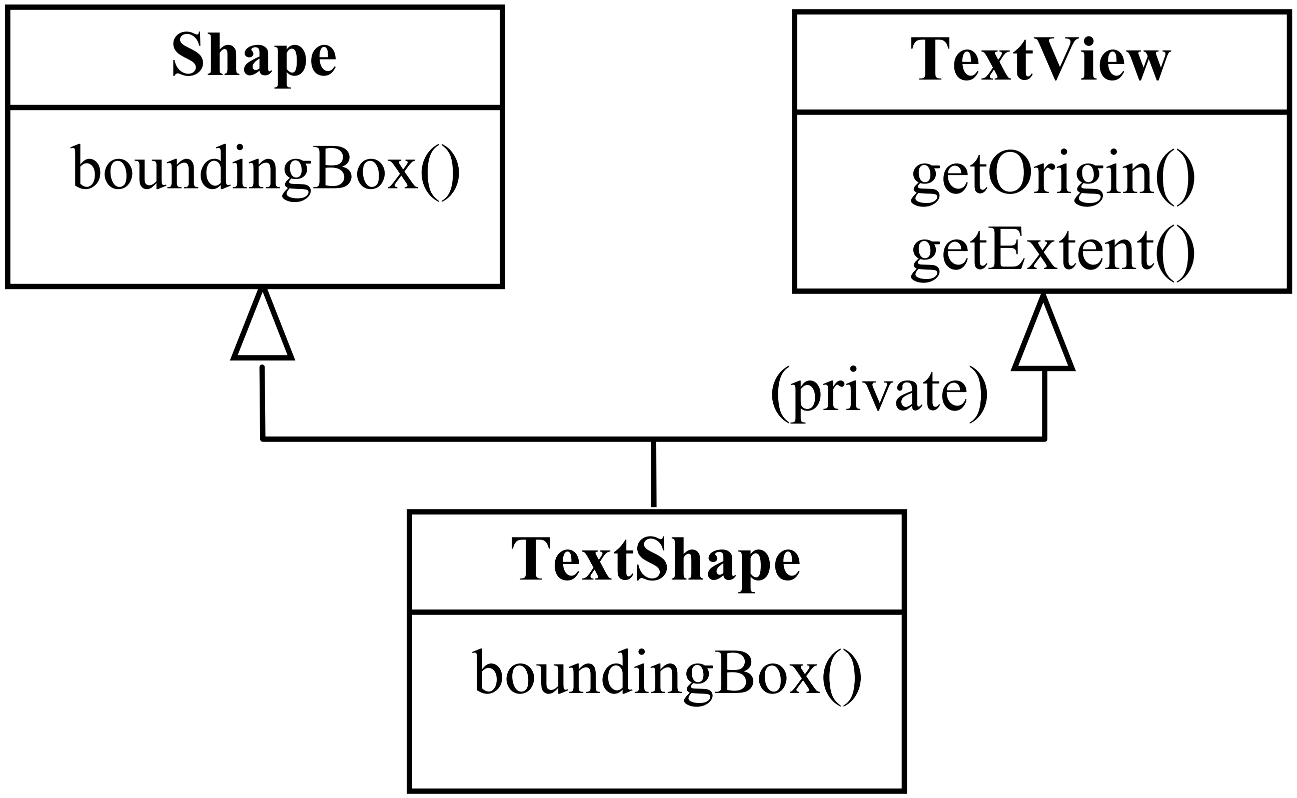
Shape and TextView Interfaces

- `Shape` and `TextView` provide different interfaces for returning the bounding box information about a shape, as shown below
- We want to create a new class, `TextShape`, which provides all the functionality of `TextView` through an interface that's compatible with the `Shape` interface



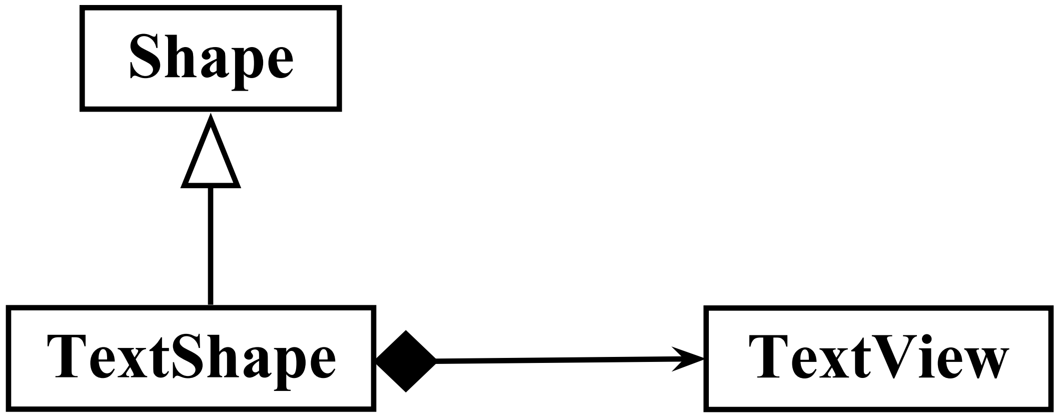
Approach 1 – Using Multiple Inheritance

- In this version, `TextShape` inherits publicly from `Shape` and privately from `TextView`



Approach 2 – Using Composition

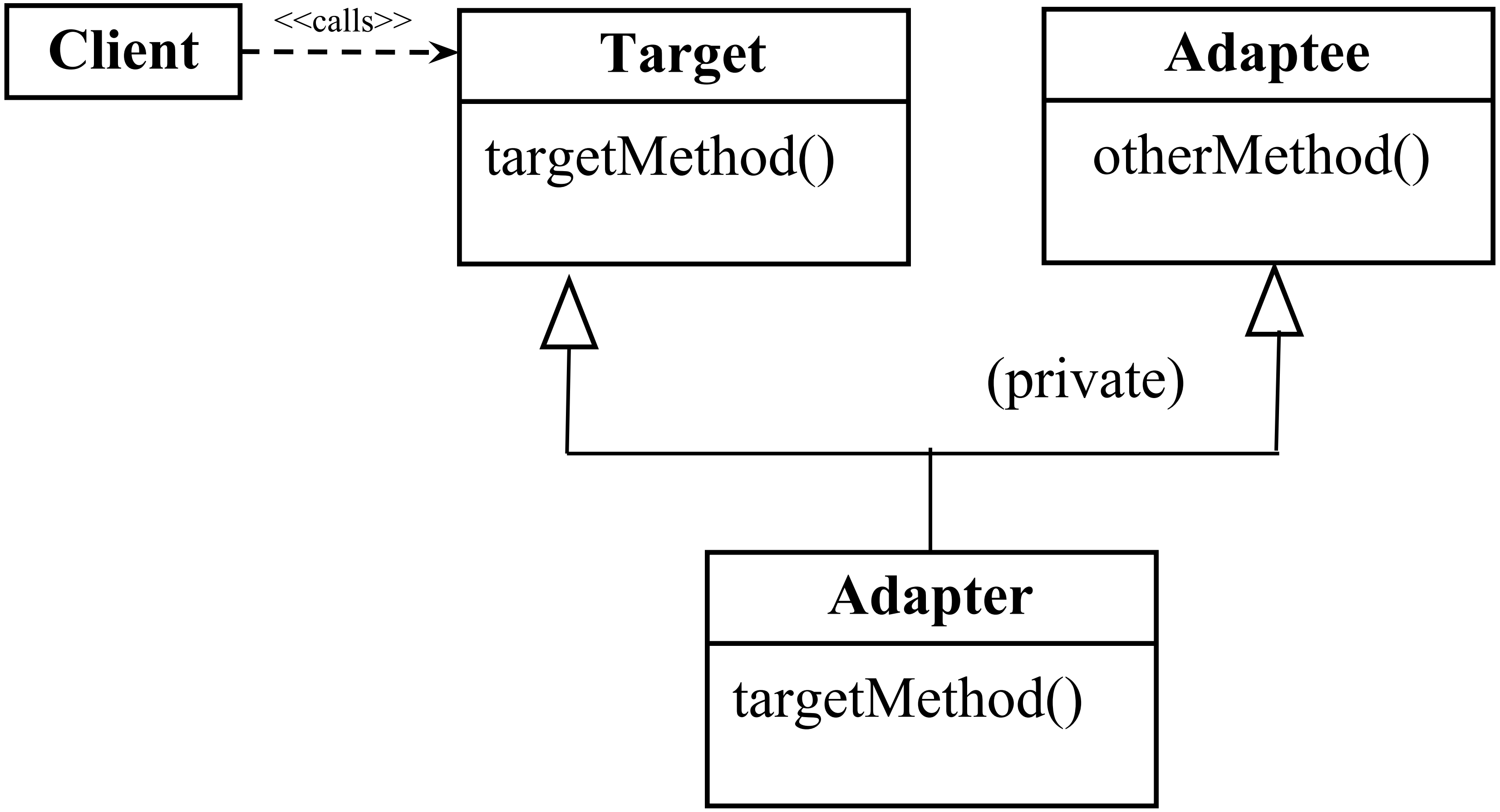
- In this version, `TextShape` contains a `TextView` object (exactly one)



Two Forms of the Adapter Pattern

- The *class adapter* uses multiple inheritance for the mapping
- The *object adapter* uses object composition for the mapping

A Generic Model for the Class Adapter



Sample Code – Class `Shape`

```
// C++ code
class Shape {
public:
// other methods …

  virtual void boundingBox(
    Point &lowerLeft, Point &upperRight) const;

// private data omitted
};
```

Sample Code – Class `TextView`

```
class TextView {
public:
// other methods …

 void getOrigin(Coord &x, Coord &y) const;

 void getExtent(Coord &width, Coord &height)const;

// private data omitted
};
```

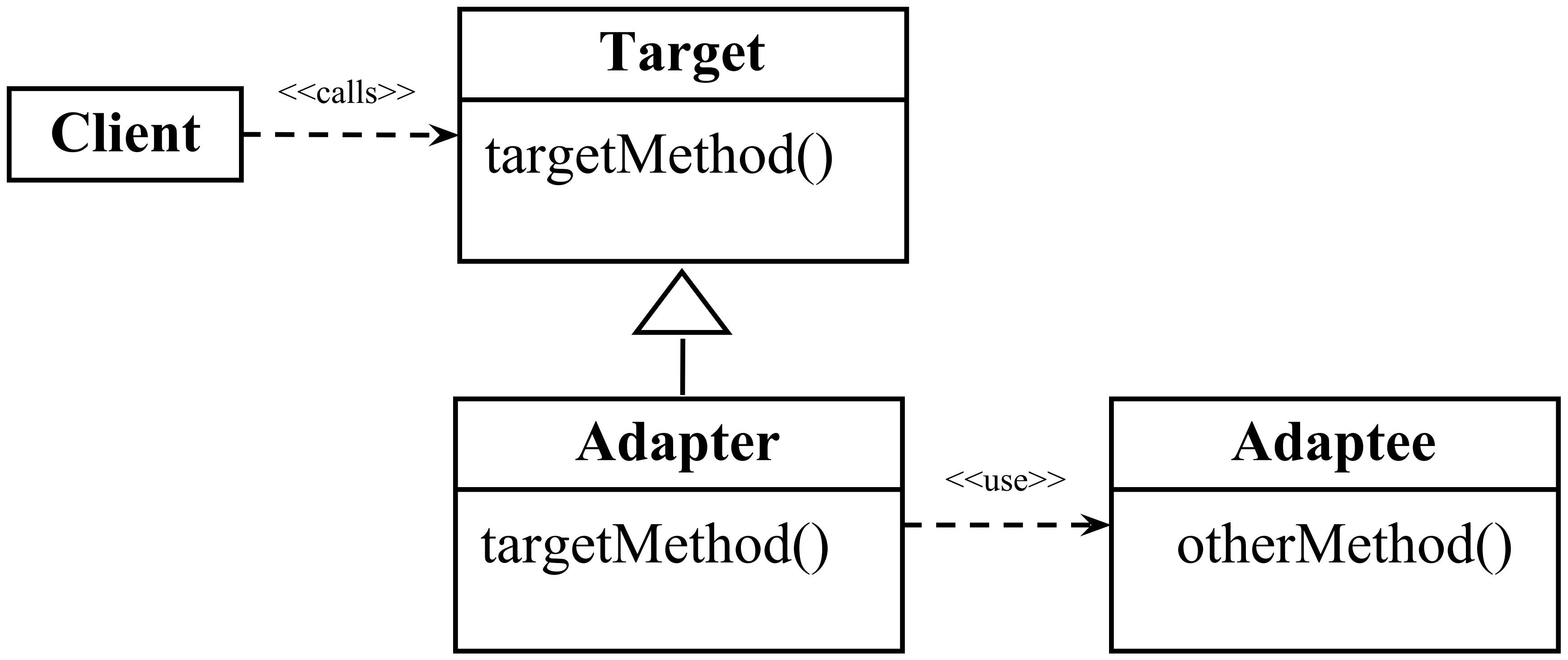
Sample Code –`TextShape` as Class Adapter

```
class TextShape: public Shape, private TextView {
public:
// other methods ...

  virtual void boundingBox(
      Point &lowerLeft, Point &upperRight) const;

};
```

Sample Code – `TextShape::boundingBox()`

```
TextShape::boundingBox(
      Point &lowerLeft, Point &upperRight) const{
   Coord bottom, left, width, height;
   getOrigin(bottom, left); // call TextView
   getExtent(width, height); // call TextView
   lowerLeft = Point(bottom, left);
   upperRight = Point(bottom+height,left+width);
}
```

The Generic Model for the Object Adapter



The `TextShape` Object Adapter in Java

- Some of the syntax we had in C++ for TextShape will not carry forward to Java. For instance, in C++, we had the following method in TextView

```
    void getOrigin(Coord &x, Coord &y) const;
```

- This passes the `Coord` objects by reference so they can be returned as out parameters
- Here we make two assumptions:
  - both `TextView` and `TextShape` utilize the class `java.awt.Point` to define a location on the screen
  - both `TextView` and `TextShape` utilize the class `java.awt.Dimension` for defining the extent (width and height) of a rectangle

Sample Code – Class `Shape` in Java

```
// First we defined the Java version
// of Shape as a Java interface
interface Shape {

// other methods …

  public Box boundingBox();

// private data omitted
}
```

Sample Code – Class `TextView` in Java

```
public class TextView {
  private Point origin;
  private Dimension extent;
  public Point getOrigin() {
    return origin;
  }
  public Dimension getExtent() {
    return extent;}
  }
  // other methods
}
```

Sample Code – `TextShape`

```
public class TextShape implements Shape {
  private TextView myTextView;
  public TextShape () {
      myTextView = new TextView();
  }
  public Box boundingBox()  {
    Point lowerLeft;
    Point upperRight;
    Dimension ext;
    // continued on next slide
```

Sample Code – Class `TextShape` – 2

```
  // now we call TextView to get the
  // origin and extent
    lowerLeft = myTextView.getOrigin();
    ext = myTextView.getExtent();
    upperRight = new Point(
     (int)lowerLeft.getX()+
                    (int)ext.getWidth(),
     (int)lowerLeft.getY()+
                    (int)ext.getHeight());
    return new Box(lowerLeft,upperRight);
  }
} // end TextShape
```

When to Use the Adapter Pattern

- When we have a coherent, consistent set of classes and want to add in a new class (say, from a third-party) that has a very different style of interface
- This could happen when you have a set of GUI classes and buy a specialized class from another vendor

Consequences of Using the Adapter Pattern

- Whether we use the Class Adapter or the Object Adapter
  - `Adapter` implements `targetMethod()` in terms of `otherMethod()` in `Adaptee`
  - The `Client` object will have only the `Adapter` interface available to it
- The Class Adapter approach fails if we want to adapt a class *and* all of its subclasses, while the Object Adapter allows the `Adapter` object to deal with any subclass of the `Adaptee` class

The Bridge Pattern

- The Problem: We want to decouple an abstraction from its implementation so that the two can vary independently

Example – Encapsulating Data Access

- Most interesting problem domain objects have state which is stored in external databases or files (these objects are referred to as *persistent* objects)
- It is desirable to separate the functionality of the problem domain object (e.g., Customer or Account) from the services required to read/write the object’s state from/to the external data source

Example – Encapsulating Data Access – 2

- The first layer of separation comes in defining, for each object, such as Customer, a data management object, say, CustomerDM, which interacts with the actual database systems and could cache the data for the object
- When there are many potential kinds of data stores (multiple relational DBMS systems, flat files, etc) we can encapsulate the data access so that all the data management objects have a uniform means of reading and writing data

Example – Encapsulating Data Access – 3

- We want to define an interface that allows us to read/write data from several kinds of data sources
- This will provide a bridge between the data management objects and disparate back-end systems

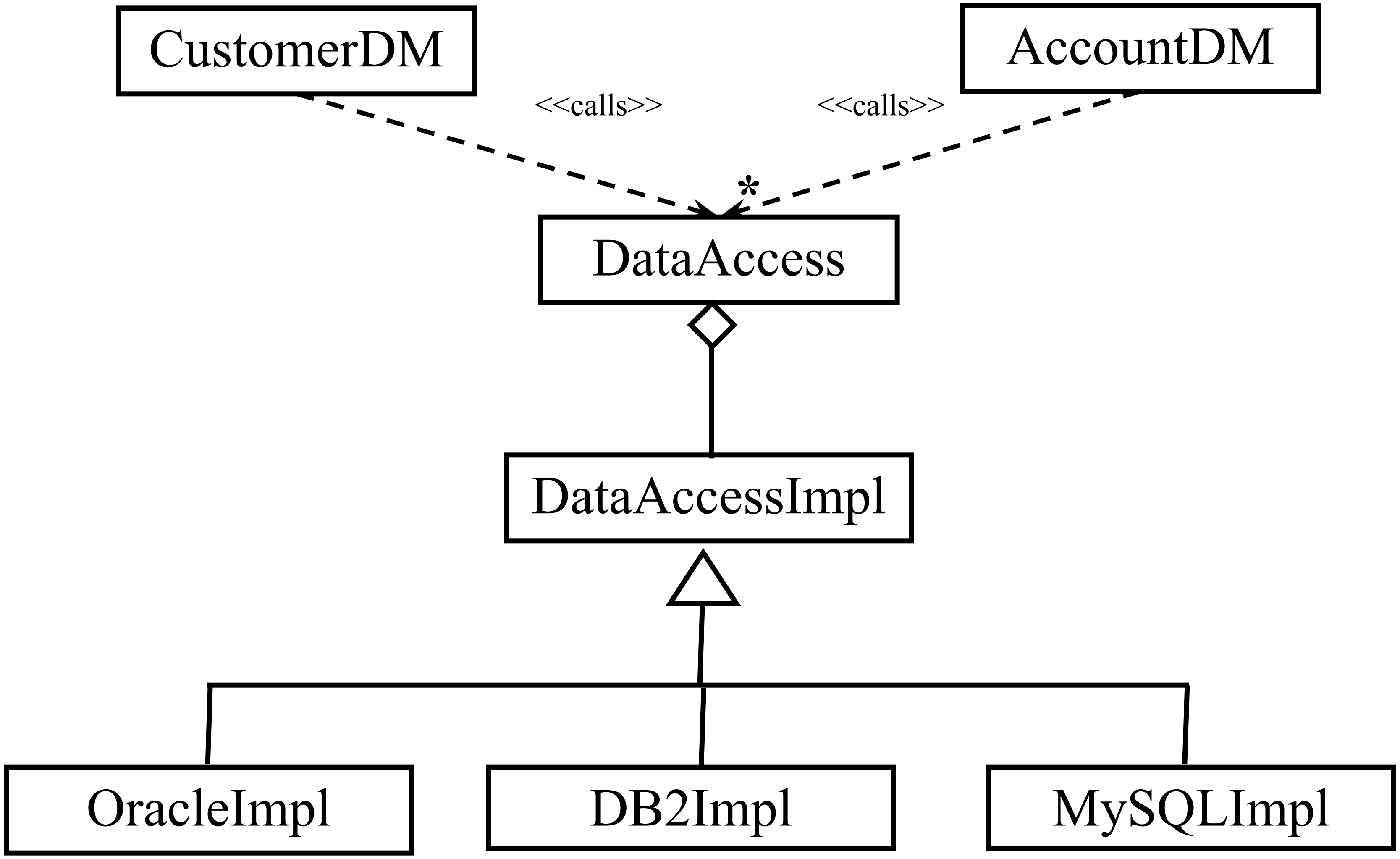
A Generic Data Access Class

- We define an abstract interface `DataAccess` with the following methods:
  - `DBId Connect(DBDescriptor desc)`
  - `void CloseConnection(DBId id)`
  - `ParameterList IssueRequest(DBId targetDB, RequestDescriptor request, ParameterList input)`

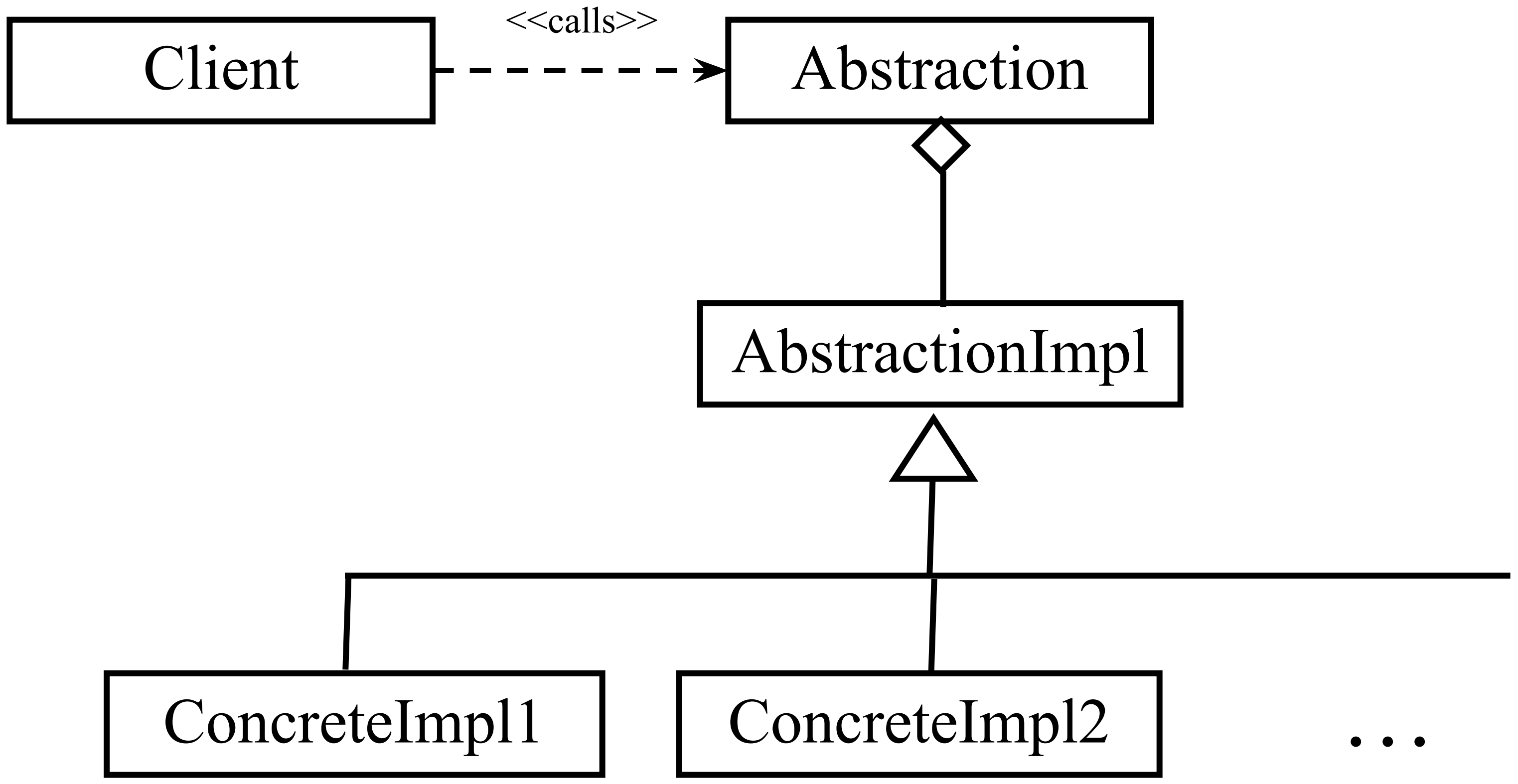
Handling Multiple Data Stores

- Having defined the generic interface, we don’t want to have to write code with lots of switch statements, such as “if it’s Informix, send the data this way, else if it’s CICS, send the data this other way, etc”)
- We will provide a separate implementation class for each of the kinds of data stores we must deal with, and have the generic interface delegate its requests to the right instance of an implementation class

Diagram for Data Access Encapsulation



The Generic Model for the Bridge Pattern



When to Use the Bridge Pattern

- Use the Bridge Pattern when you want to vary the implementation of an interface or set of interfaces at run-time, independently of the interface

Tradeoffs for the Bridge Pattern

- Building a complete bridge, such as the data access interface or an abstract windowing interface is a complex task
- It requires a deep understanding of the kinds of implementations to be bridged and an understanding of clients needs so that a useful generic interface can be defined
- The positive gains from insulating a client from many nasty details of multiple kinds of implementations must be weighed against the cost of building and maintaining the bridge itself

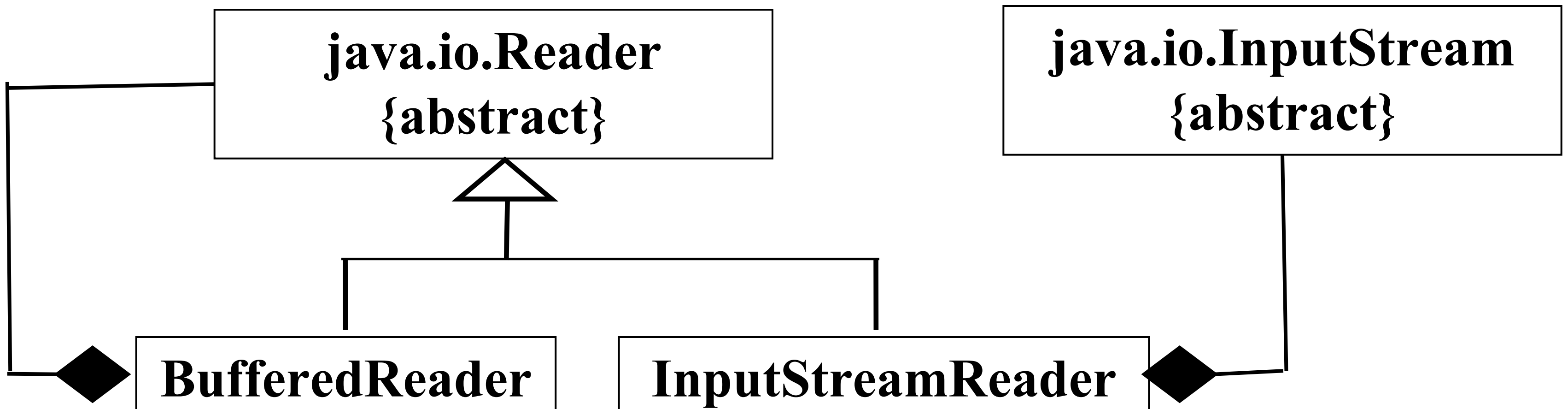
The Decorator Pattern

- The Problem: You have a set of basic objects with many options for adding behavior at run-time

Example: java.io

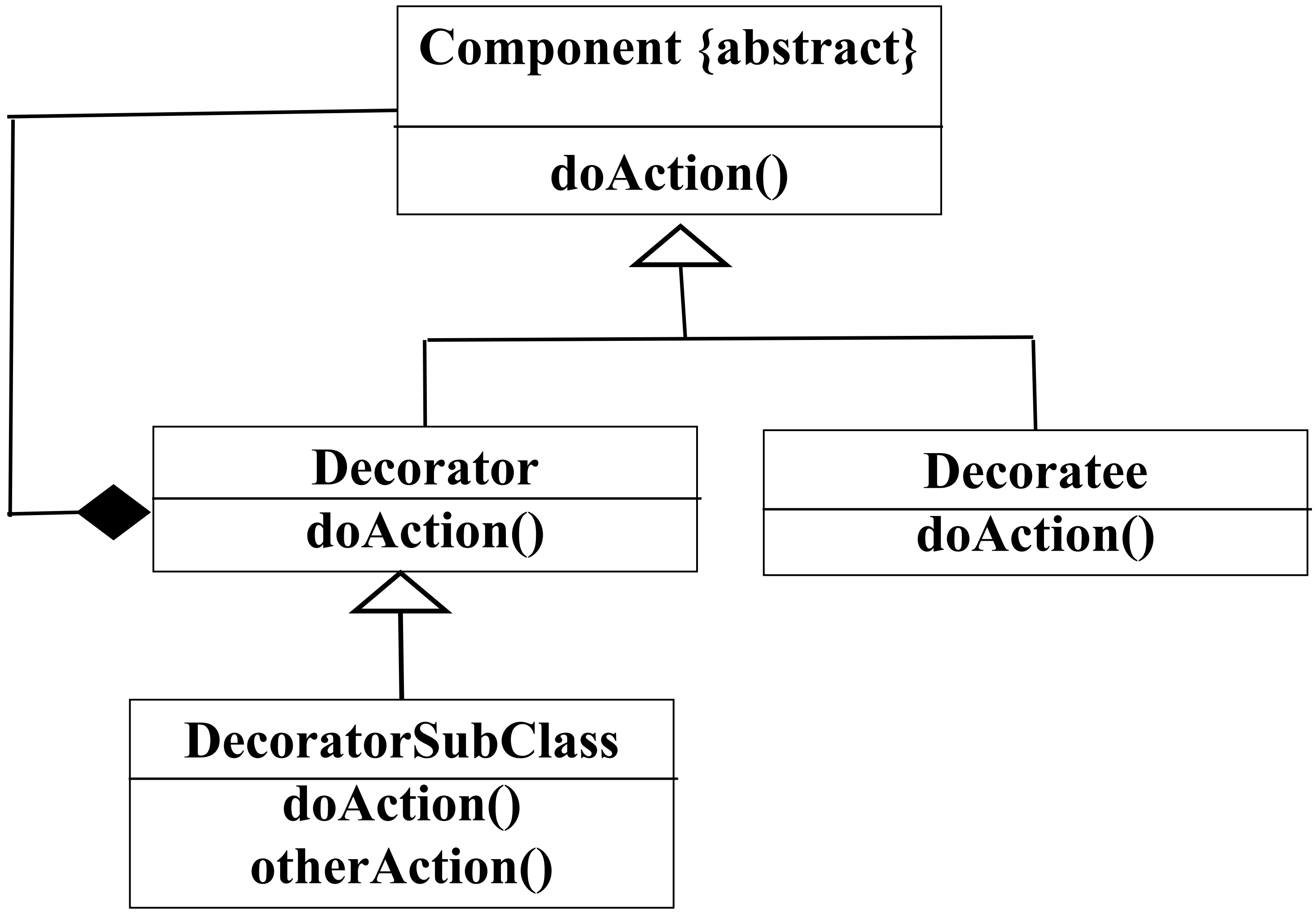
- The original java.io package had a limited view of reading or writing files in that it was (byte) character-based as opposed to Unicode (multi-byte) character-based
- This was modified in JDK 1.1 by providing higher-level readers and writers that could add functionality
- A Unicode reader, for instance, can be constructed by using a lower-level byte-character reader as the initial input device

Decorating Byte InputStreams



```
BufferedReader in = new BufferedReader(
                 new InputStreamReader (System.
                 in));
```

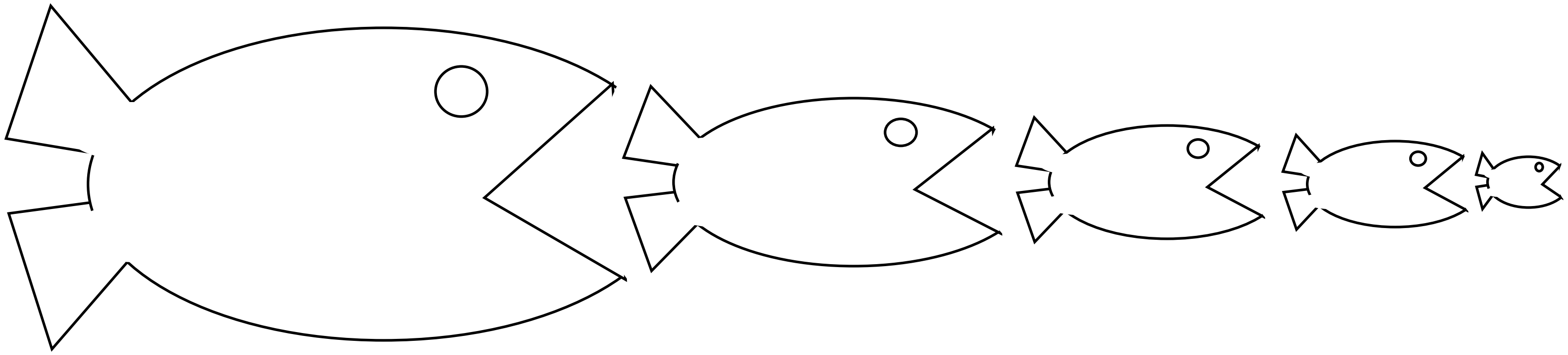
The Generic Decorator Model



When to Use the Decorator Pattern

- When you have lots of additional behavioral combinations to add, but using subclassing would create too complex a class hierarchy

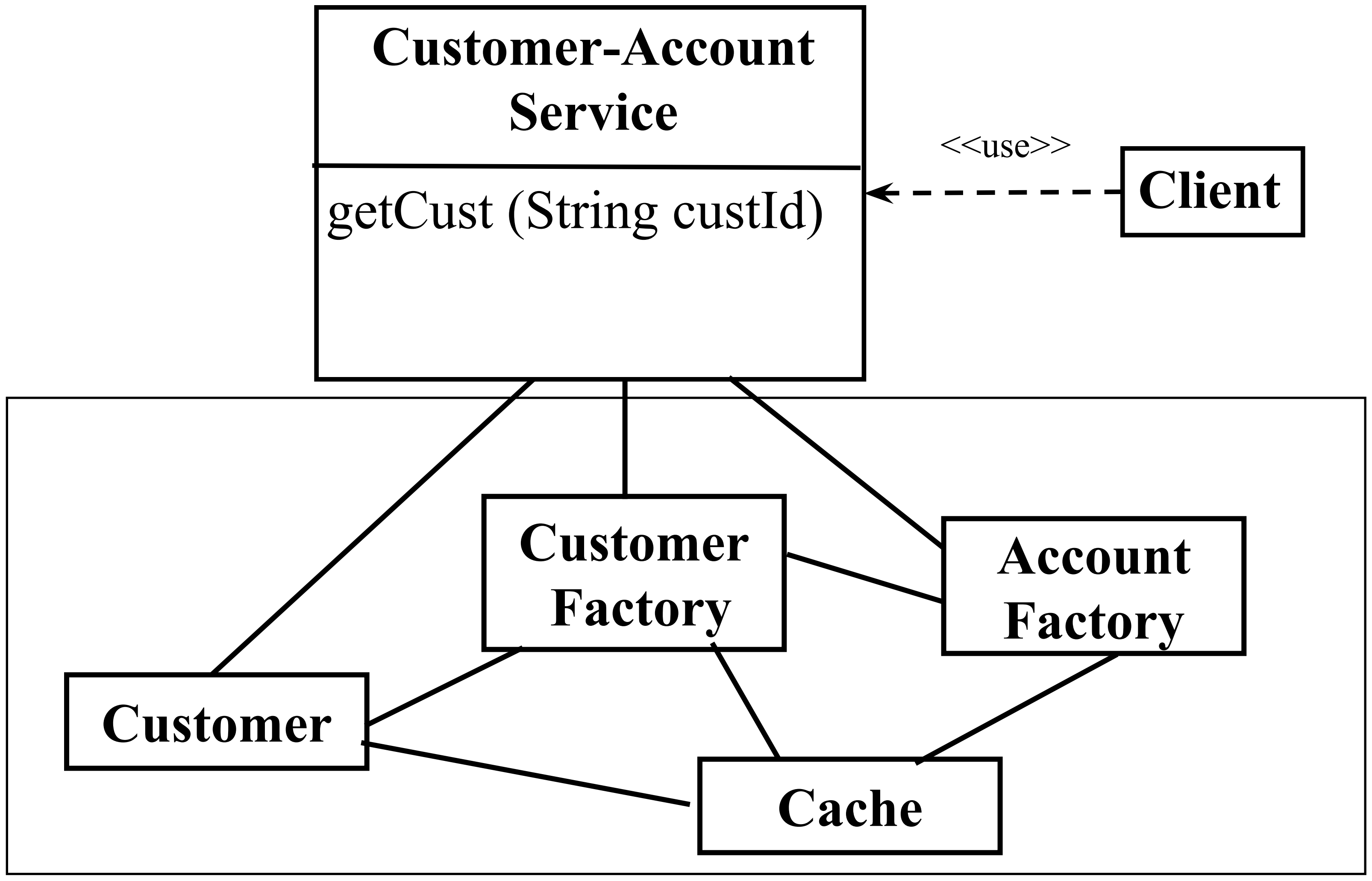
The 'food-chain' View of Decorator



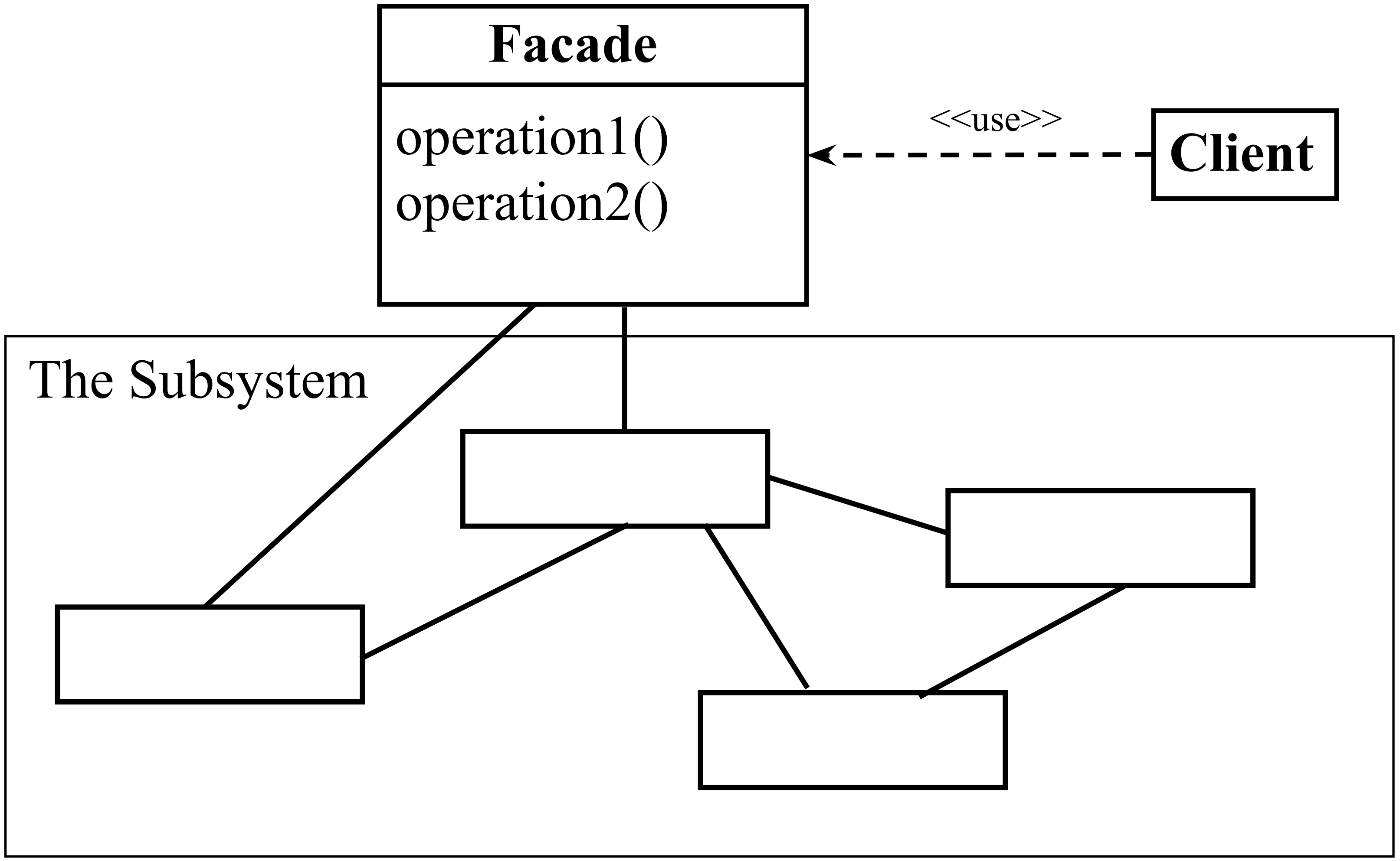
The Façade Pattern

- The Problem: You have a complex set of objects that together provide a useful service. However, it is not appropriate to make the entire set of objects visible to clients, nor is it desirable to expose the way in which objects within the service are related and interact

Example –The Customer-Account Service



The Generic Model for the Facade Pattern



Sample Code – `CustomerAccountService`

```
public class CustomerAccountService {
  // other methods. . .
  public Customer getCust(String custID) {
    // do something to look up the Customer
    // and return it
 }

  public Account  getAccount(String accountID){
    // do something to look up the Account
    // and return it;
   }
}// end CustomerAccountService
```

Sample Code – Using `CustomerAccountService`

```
// somewhere in the client's code
// 1. Create an instance of the service
 CustomerAccountService cas =
                   new CustomerAccountService();
// 2. somehow, get a Customer ID
 String customerID = getCustID();
// 3. Use getCust to fetch a Customer object
 Customer newCustomer = cas.getCust(customerID);
// 4. Now ask for the account balance
 double balance = newCustomer.accountBalance();
```

When to Use the Façade Pattern

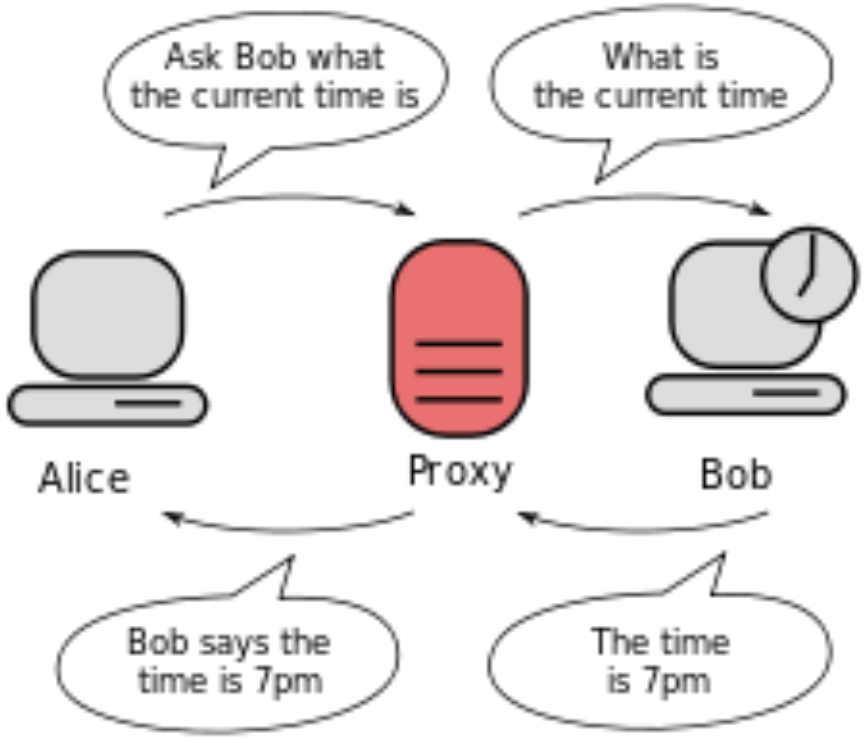
- Use the Façade Pattern when you want to provide a simple interface to a complex subsystem, thereby exposing fewer of the classes in the subsystem
- This can occur when you want to layer your subsystems, so that each layer uses the services of the layer below and provides support to the next higher layer

Consequences of Using the Façade Pattern

- The Façade Pattern can decouple clients from explicit knowledge of the relationships among the objects in a subsystem used by a client
- The Façade should also be flexible enough to allow clients access to those objects that are necessary (for instance, in the Customer-Account Service, the access methods will likely return `Customer` and `Account` objects, but not the related objects such as `CustomerFactory` and `AccountFactory`)

The Proxy Pattern

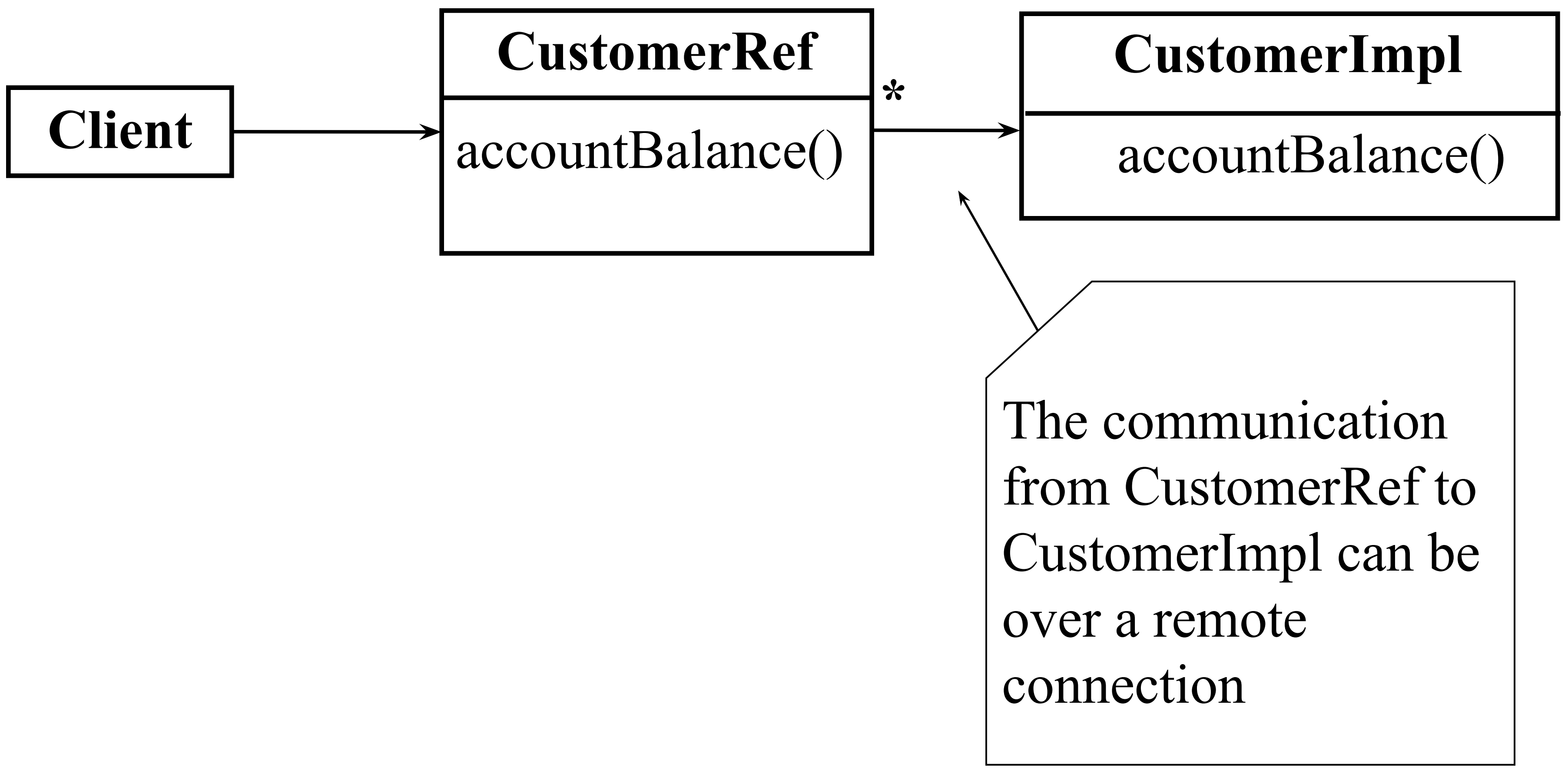
- The Problem: We want to provide a surrogate or a placeholder for an object that allows us to control access to it



A Proxy for a Remote Object

- Suppose we've built a `Customer` class that works fine in a single JVM, and we would like it to be available over the network; that is, clients in one process could interact with an instance of class `Customer` running in a second process
- We'll consider only the `getBalance()` method of class `Customer`

Initial Diagram for the `Customer` Proxy


Client
CustomerRef
accountBalance()
*
CustomerImpl
accountBalance()
The communication from CustomerRef to CustomerImpl can be over a remote connection

The Generic Model for the Proxy Pattern

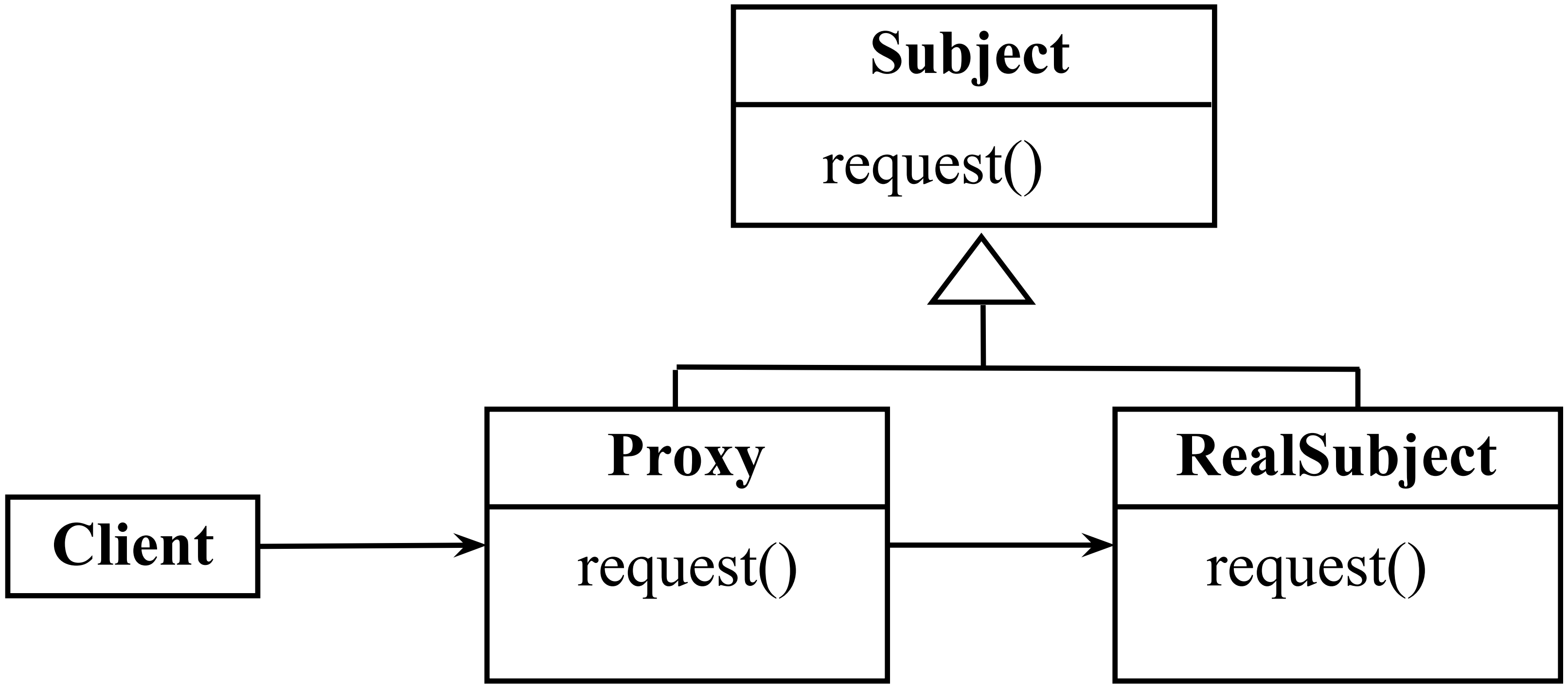
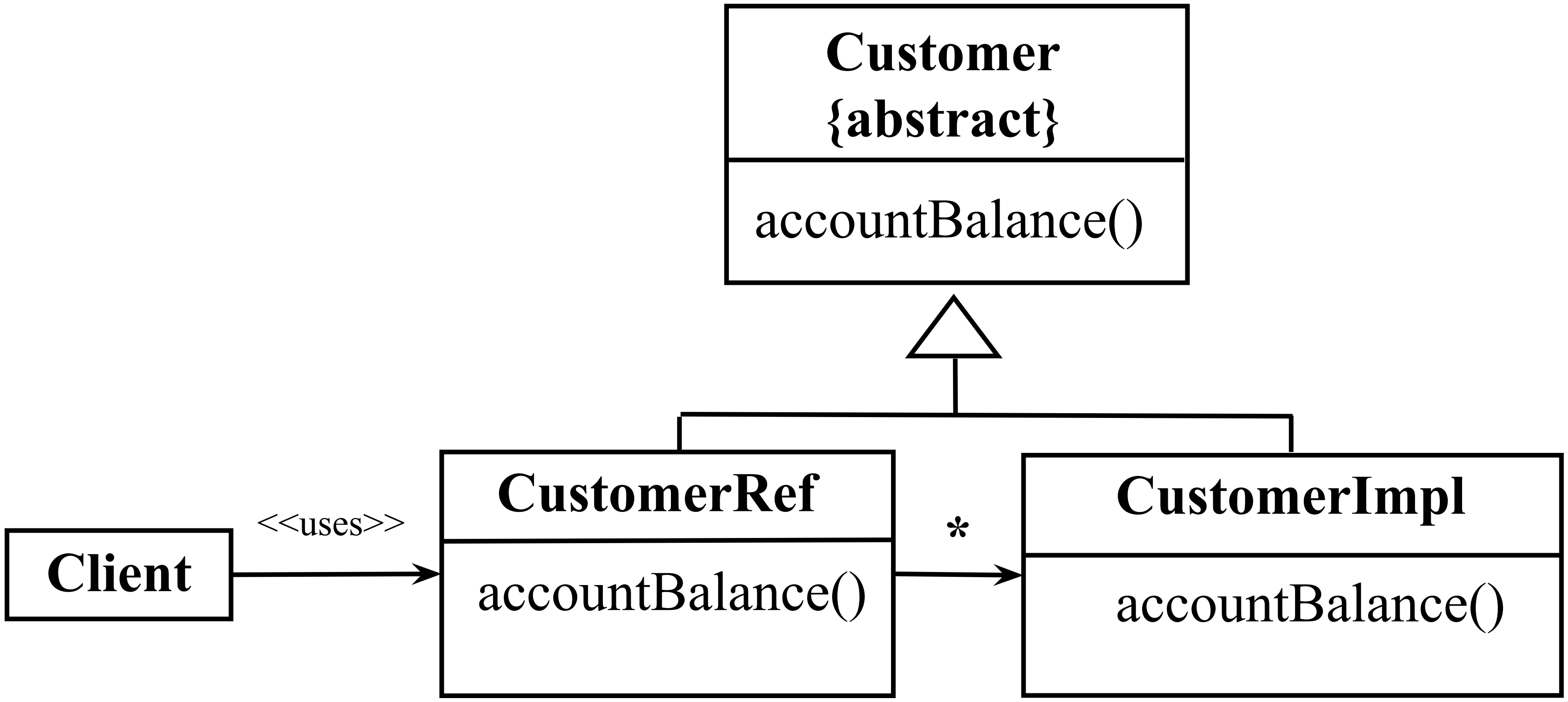


Diagram for the Customer Proxy



Sample Code – Interface `Customer`

```
public interface Customer {
    public double accountBalance();
    // other useful stuff ...
};
```

Sample Code – Class `CustomerRef`

```
public class CustomerRef implements Customer {
  public CustomerRef(String customerID){
/**
 * the constructor of the proxy has to work to find the real
 * object, CustomerImpl, and have code established to send
 * requests to CustomerImpl.  This usually involves low-level
 * stuff, such as sockets, that you never want to see.
*/
}
```

Sample Code – Class `CustomerImpl`

```
class CustomerImpl implements Customer {
  public CustomerImpl(String customerID ) {}
  public double accountBalance() {
    // add code to calculate the account balance here
  }
// other useful stuff ...
}
```

Sample Code – A `Customer` Client

```
// Assume there is a CustomerAccountService
// instance called CustService which
// returns a Customer object that is actually
// of type CustomerRef in its getCust() method

Customer custObj = CustService.getCust("1234");

// now we can get the account balance from custObj
double currentBalance = custObj.accountBalance();
```

When to Use the Proxy Pattern

- When you want to hide details of how to invoke methods on an object
- When you want to provide security checks on the real object
- When you want to separate the interface from the implementation so they can be in separate processes or on separate machines

Consequences of Using the Proxy Pattern

- The Proxy Pattern is useful for enhancing memory management and insulating clients from details of the implementation of the real object
- Proxies have become pervasive in distributed systems
- Writing a proxy for many methods can be tedious
- Proxies are often generated (e.g. JAX-WS and JAX-RS for generating SOAP and REST proxies respectively)

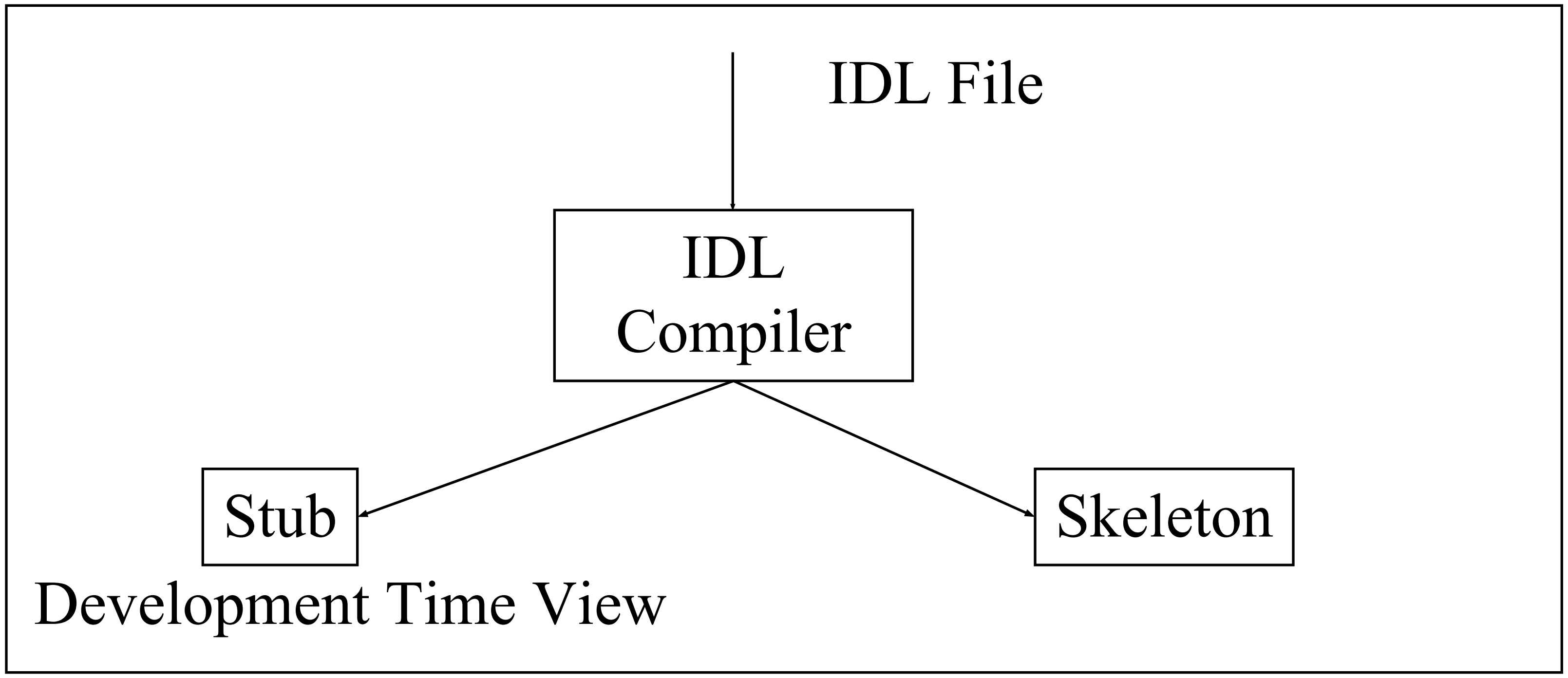
Tradeoffs for the Proxy Pattern

- The `Proxy` implementation needs to present the same interface as the original `Subject`, including error returns and exceptions thrown
- A `Proxy` used for remote access might have to introduce new error returns or throw additional exceptions

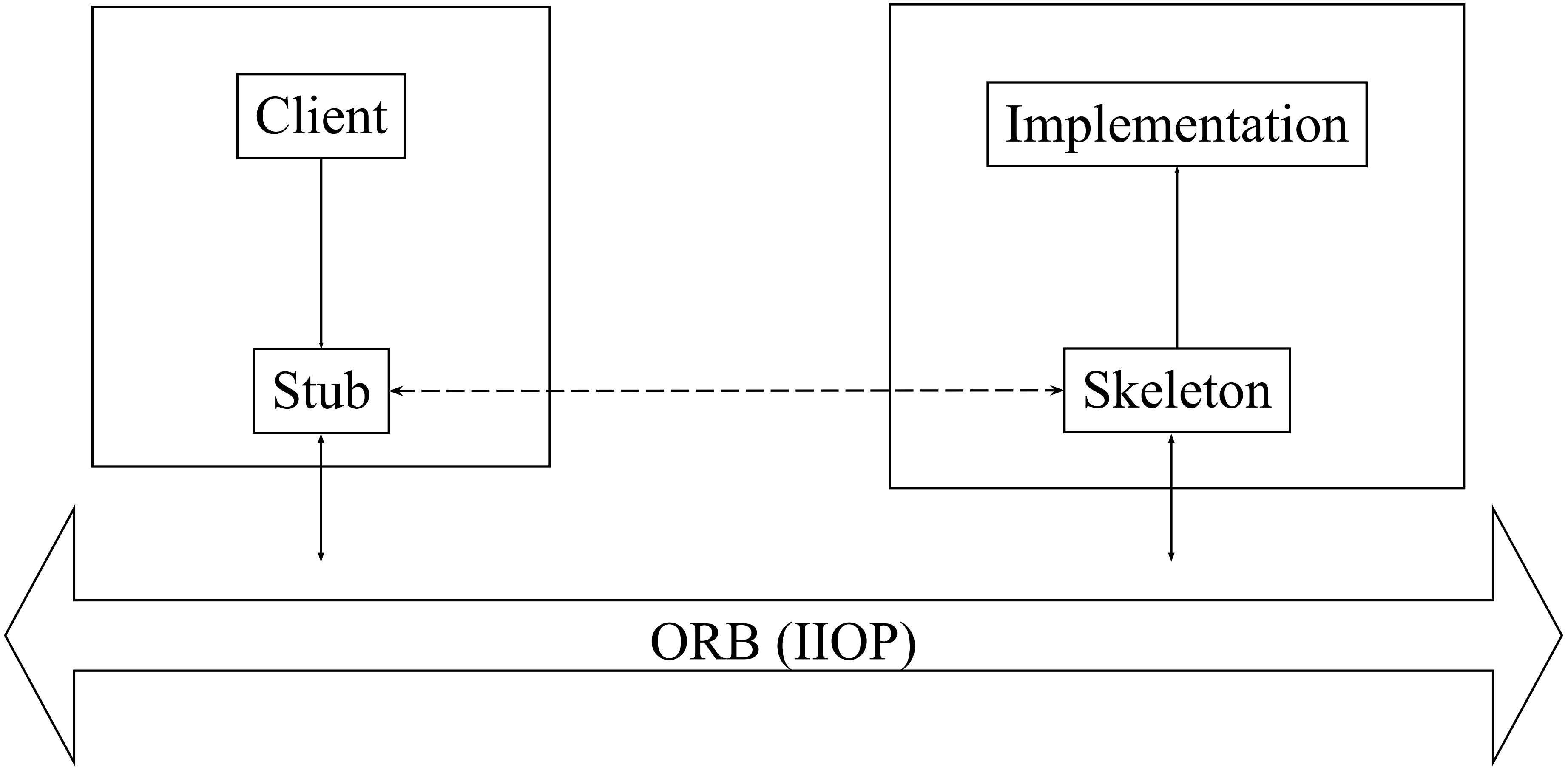
How CORBA Generates Proxies

- The Object Management Group released the definition of the Common Object Request Broker Architecture (CORBA) in 1990
- With CORBA, one defines an interface in the OMG Interface Definition Language (IDL)
- Then one feeds the IDL into a language specific IDL compiler to generate a proxy (called a *stub*) and a corresponding server side class (called a *skeleton*)

CORBA Proxy Generation



CORBA Run-time View



Other ORB-Like Creatures

- There are several ORB-like mechanisms available. All provide similar functionality to the ORB, with varying degrees of additional services (e.g., Naming Services, Event Services, …)
- DCOM
- RMI
- Web Services (using WSDL to describe the interface)
- .NET