CPE 545

Middleware Principles and Basic Patterns

Software component - terminology

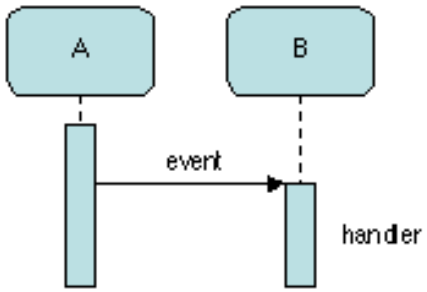
- A software system is organized as a set of parts, or components
- A software component is a unit of composition with contractually specified interfaces and explicit context dependencies.
- In order to provide its service, a component usually relies on services provided to it by other components.
- A provided service is usually embodied in a set of interfaces, each of which represents an aspect of the service.
- Software components can be used to offer a modular approach to products

Interaction Mechanisms between Components

- Components interact through an underlying communication system by means of:
  - Asynchronous transient event.
  - Asynchronous persistent message passing
  - Synchronous call

Interaction Mechanisms

Asynchronous transient event

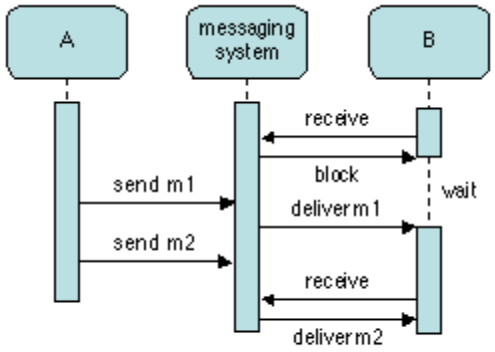
- A thread executing in component *A* produces an event to a specified set of recipients and continues execution.
- The "transient" attribute means that the message is lost if no recipient is waiting for it.
- Reception of the event by component B starts the execution of a program (the handler) associated with that event.
- Usage:
  - It may be used by *A* to request a service from *B*, when no result is expected;
  - It may be use 

tor the activity of *A*.

Interaction Mechanisms

Asynchronous persistent messages (Buffered messages)

- A message is transmitted by a sender to a receiver over a the communication system provides a buffering function ("persistent" attribute)
  - If the receiver is waiting for the message, the communication system delivers it;
  - If not, the message remains available until the receiver attempts to read it.

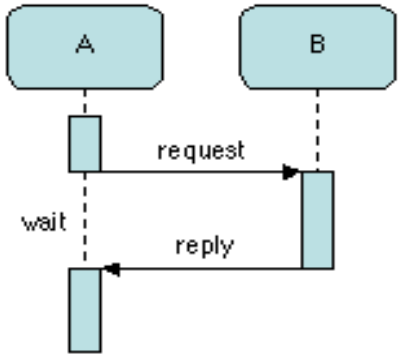


(b) Buffered messages

Interaction Mechanisms

Synchronous transient call

- *A* (the client of a service provided by *B*) sends a request message to *B* (server) and waits for a reply (e.g. RPC).



(c) Synchronous call

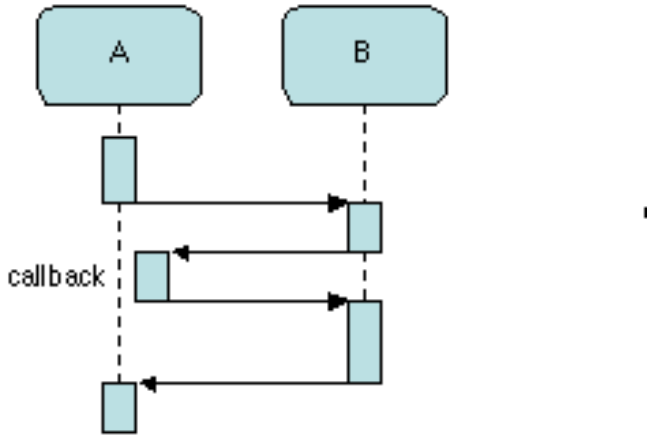
Combination

- Synchronous and asynchronous interactions may be combined to allow the service requester to continue execution after issuing the request.
  - The provider may inform the requester, by an *asynchronous event*, that the results are available. Requester still needs to request (pull) to get the results.
  - or the requester may call (*poll*) the provider at a later time to find out about the state of the execution.

Interaction Mechanisms

Synchronous call with callback

- It may happen that execution of the call from *A* to *B* relies on a *callback* from *B* to a function provided by *A*.
- The callback is executed by a new thread, while the original thread keeps waiting for the completion of the initial call.

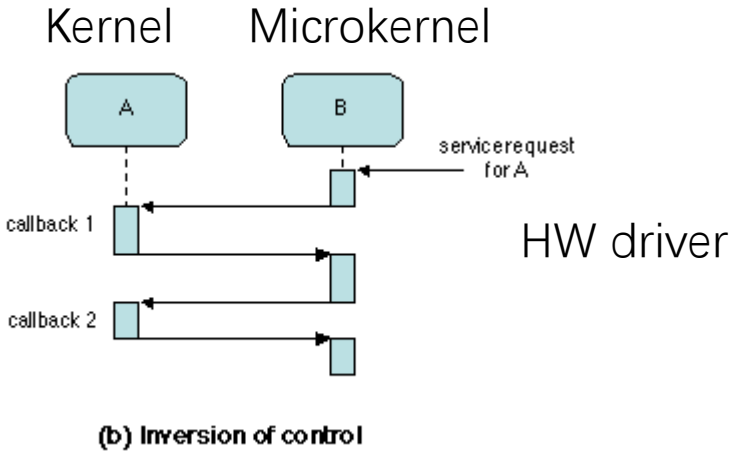


(a) Synchronous call with callback

Interaction Mechanisms

Inversion of control (triggered by asynchronous event)

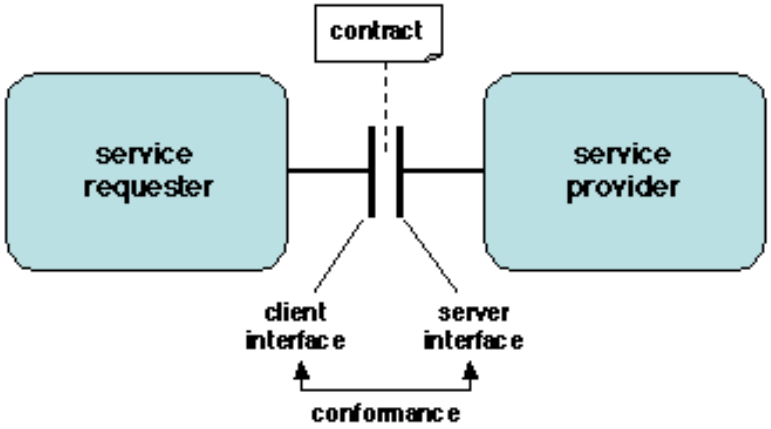
- The service provided by *B* to *A* may be requested from an outside source, with *A* still providing callback to *B*. This interaction pattern is called *inversion of control*, because the flow of control is from *B* (the provider) to *A* (the requester).
  - It typically occurs when *B* is "controlling" *A*, and the request for service originates from the outside, triggered by an external event such as a timing signal.



(b) Inversion of control

Interfaces

- An interface is a concrete description of the interaction between the requester (client) and the provider (server) of the service.
- The server interface should be "conformant" with the client interface;



Interfaces

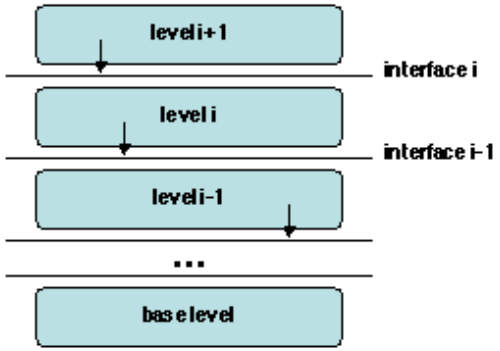
- The concrete representation of an interface consists of a set of operations, which may take a variety of forms:
  - Synchronous procedure or method call, with parameters and return value;
  - asynchronous procedure call;
  - Access to an attribute, i.e. a data structure (this can be converted into the previous form by means of "getter" or "setter" functions on the elements of the data structure);
  - event source (sender) e.g. clicking a button -- or sink (receiver) e.g. how to handle clicked button);
  - data stream provider (output channel) or receiver (input channel);

Interfaces - characteristics

- A number of notations, known as Interface Description Languages (IDL), have been designed to formally describe interfaces.
- Neither the requester nor the provider should make any assumption on the other party, beyond the information explicitly specified in the interface.
- Anything beyond the client or server interface is seen by the other party as a "black box". This rule is known as the *encapsulation principle.*
- The encapsulation principle ensures independence between interface and implementation, and allows a system to be modified by "plug and play"; hence the rest of the system remain compatible.

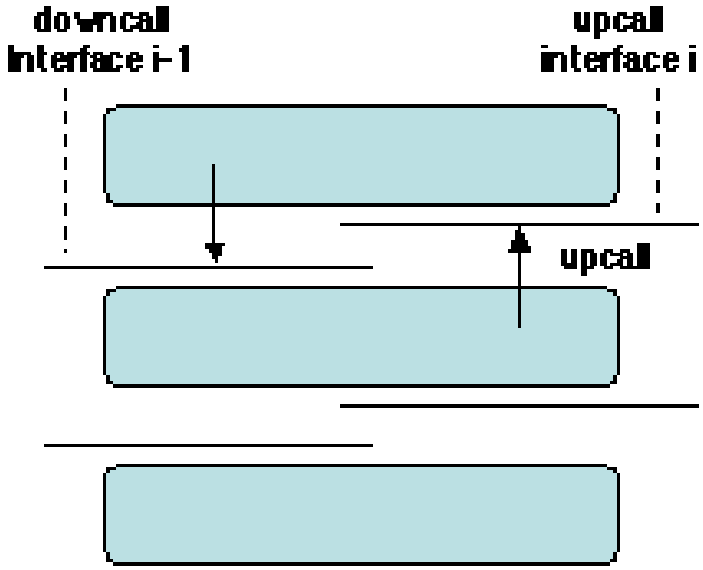
Architectural Patterns examples

- **Multilevel Architectures (Layered Architecture)**
  - A complex system may be described at different levels of abstraction.
  - Each level *i* defines its own entities, which provide an interface to the upper level (*i*+1).
  - These entities are implemented using the interface provided by the lower level (*i*-1), down to a predefined base level
    - An alternative view is to consider each level as a virtual machine, instruction set is defined by its interface. By virtue of the encapsulation principle, a virtual machine hides the implementation details of all the lower levels.



Architectural Patterns examples

- Each layer receives synchronous calls from the upper layer, and asynchronous calls from the lower layer.
  - The upper (application) layer activates the kernel (and hardware-provided functions) through synchronous **downcalls**.
  - On the other hand, the hardware typically activates the kernel through asynchronous interrupts (**upcalls**), which trigger the execution of handlers.



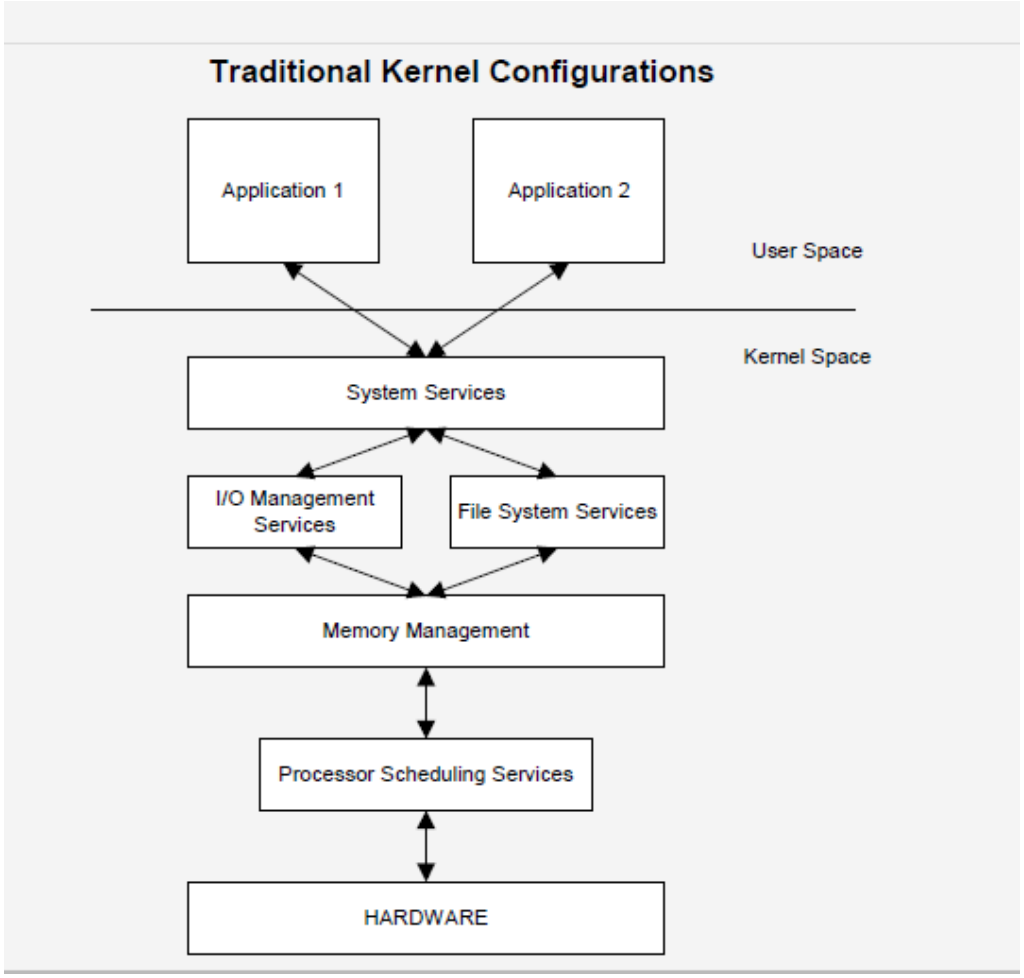
Architectural Patterns examples

- A microkernel-based operating system consists of two layers.
  - The microkernel proper, which manages the hardware resources (processor, memory, I/O, network communication), and provides an abstract resource management API to the upper level.
  - The kernel, which implements a specific operating system using the API of the microkernel.

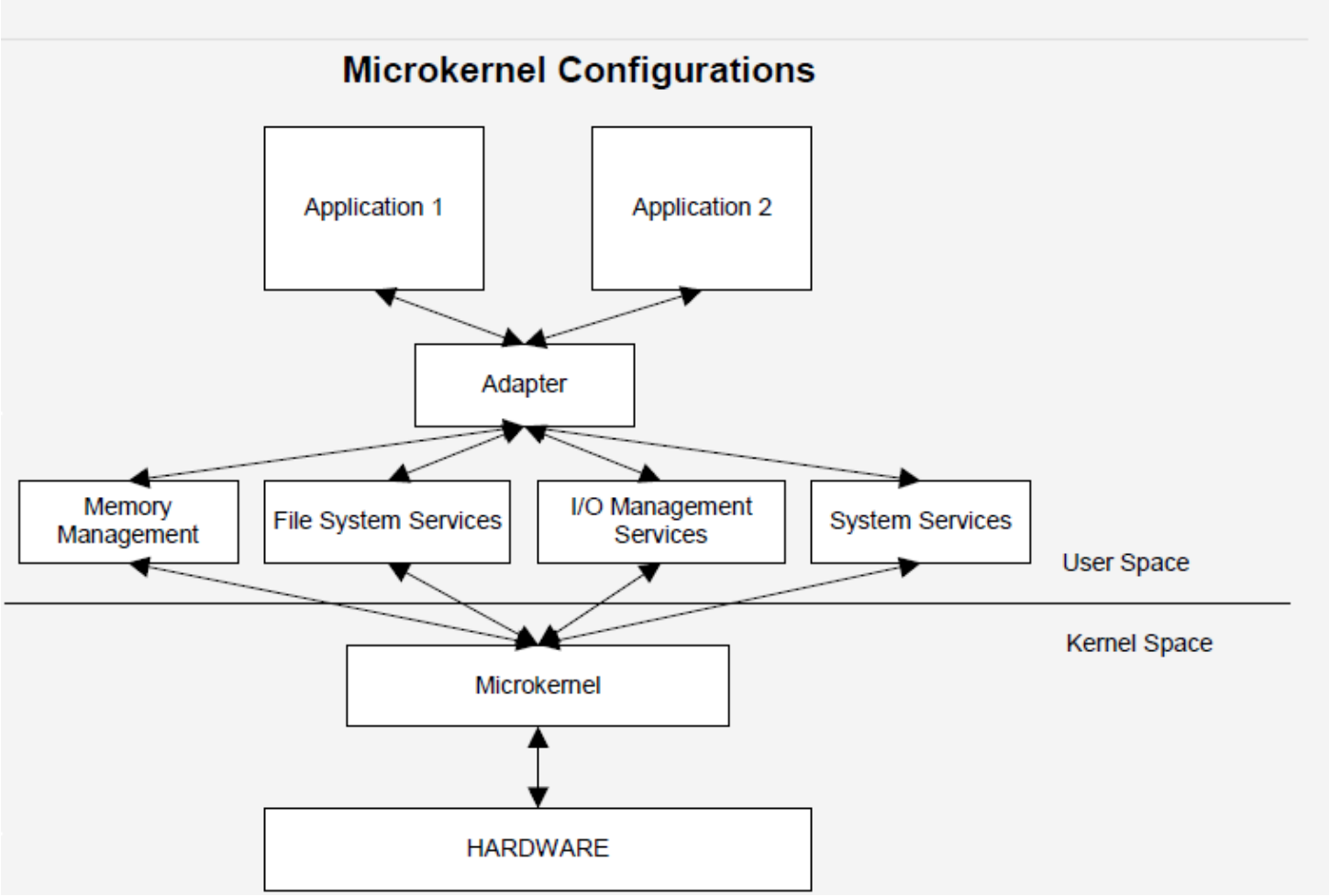
Microkernel

- A Microkernel is a highly modular subsystem composed of OS-neutral abstractions, providing only essential services such as process abstractions, threads, IPC, and memory management primitives.
- All device drivers, etc., which are normally part of an OS kernel, run on the microkernel as just another user process
- Multiple operating systems can then be layered *on top of* these abstractions, and are thus viewed as simply another *application.*
- This focus on modularity allows for scalability, extensibility and portability not found in monolithic operating systems (Unix, Linux, DOS, etc.)
- Because of its highly modular nature, many of the services commonly found in "kernel space" are found in "user space" on a microkernel

Microkernel



Microkernel

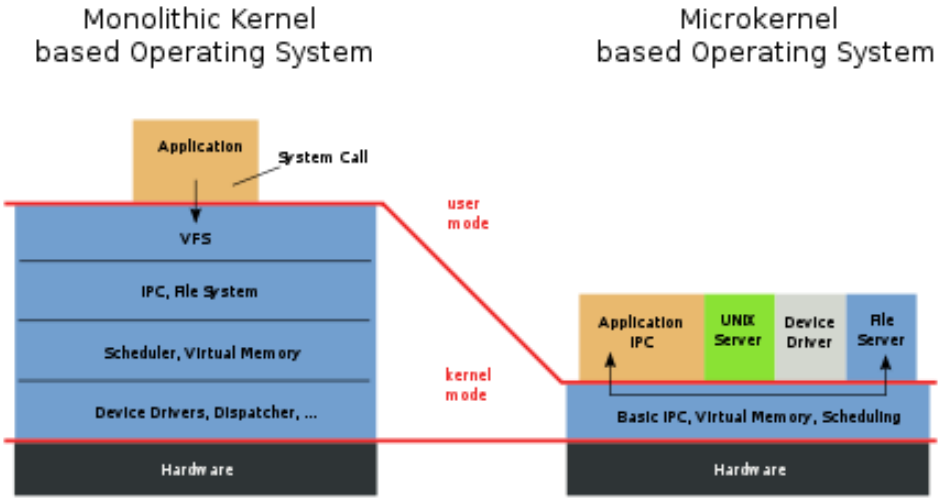


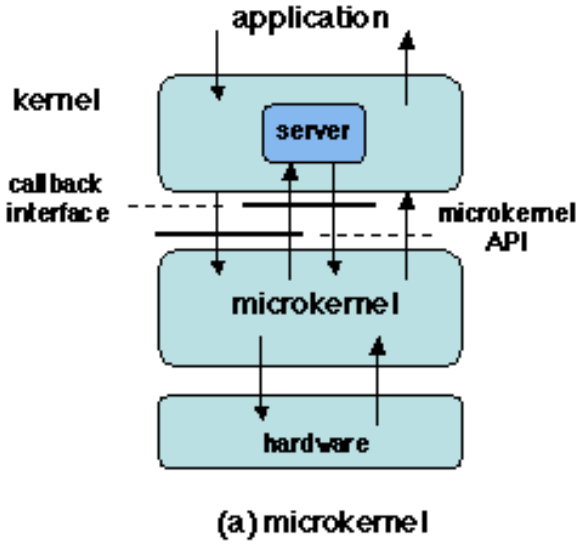
Microkernel

- In most microkernel-based organizations, an operating system kernel is structured as a set of *servers*, each of which is in charge of a specific function .
- A typical system call issued by an application is processed as follows:
  - The kernel analyzes the call and downcalls the microkernel using the appropriate function of its API.
  - The microkernel upcalls a server in the kernel and may interact with the hardware.
  - The microkernel returns to the kernel, which completes the work and returns to the application.

Microkernel

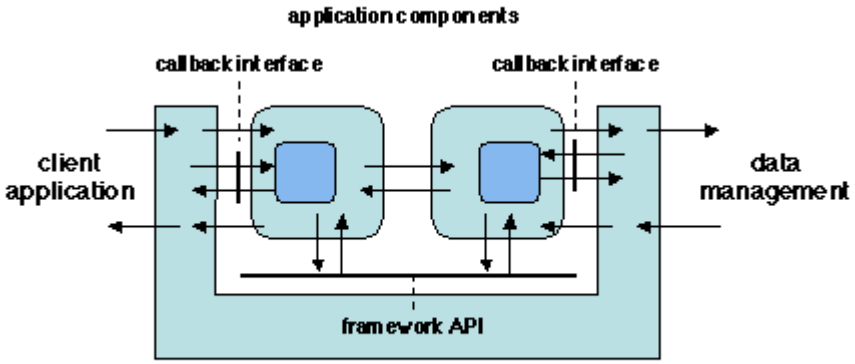
- Adding a new function to a kernel is done by developing and integrating a new server.





Frameworks

- The middle tier framework – interacts with both the *client* and the *data management* tiers, and mediates the interaction between these tiers and the *server application program*.
- A Client application program uses the *API provided* by the framework and supplies a set of *callback* interfaces (via callback registration).
- A client application request is handled by the framework, which activates an appropriate application component (via callbacks), interacts with it using its own API and the component's callback interface, and finally returns to the client.



(b) middle tier framework

Frameworks

- Both above examples illustrate inversion of control.
  - To provide its services, the framework uses callbacks to externally supplied software modules
- These modules in turn must respect the framework contract
  - by providing a specified callback interface,
  - by using the framework API.

Distributed Objects

- An *object* is a software representation of a real-world entity which consist of *state* and related *behavior*. An object stores its state in *fields* and exposes its behavior through *methods*.

Distributed Objects

Encapsulation

- An object has an interface, which comprises a set of methods and states (attributes).
- The only way of accessing an object is through its *interface*.
- No part of the state is visible from outside the object, other than those explicitly present in the interface, and the user of an object should not rely on any assumption about its implementation.
- *Encapsulation* achieves independence between interface and implementation. Changing the implementation of an object is invisible to its users, as long as the interface is preserved.

Distributed Objects- properties

Classes and instances

- A *class* is a generic description that is common to a set of objects (the instances of the class).
- The instances of a class have the same interface (hence the same type), and their state has the same structure; they differ by the value of that state.
- Each instance is identified as a distinct entity.
- Instances of a class are dynamically created, through an operation called *instantiation*; they may also be dynamically deleted, either explicitly or automatically (by garbage collection) depending on the specific implementation of the object model.

Distributed Objects- properties

Inheritance

- A class may be derived from another class by specialization, i.e. by defining additional methods and/or additional attributes, or by redefining (*overloading*) existing methods.
- The derived class is said to *extend* the initial class (or base class) or to *inherit* from it.
- Some models also allow a class to inherit from more than one class (multiple inheritance).

Inheritance - Java

- When we talk about inheritance, the most commonly used keyword would be **extends** and **implements**. These words would determine whether one object IS-A type of another.
- IS-A Relationship:
  - IS-A is a way of saying : This object is a type of that object.

Inheritance - Java

- Let us see how the **extends** keyword is used to achieve inheritance:

```
public class Animal {

}

public class Mammal extends Animal {

 }

public class Reptile extends Animal {

}

public class Dog extends Mammal {

}
```

Inheritance - Java

- In the example above, the following are true:
  - Animal is the superclass of Mammal class.
  - Animal is the superclass of Reptile class.
  - Mammal and Reptile are subclasses of Animal class.
  - Dog is the subclass of both Mammal and Animal classes.
- Considering the IS-A relationship, we can say:
  - Mammal IS-A Animal
  - Reptile IS-A Animal
  - Dog IS-A Mammal
  - Hence : Dog IS-A Animal as well

Inheritance - Java

- With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.
- We can assure that Mammal is actually an Animal with the use of the instance operator.

```
public class Dog extends Mammal {
   public static void main(String args[]){
      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();
      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

Inheritance - Java

- An **interface** is a collection of abstract methods.
  The **implements** keyword is used by classes to inherit from interfaces.
  Interfaces can never be extended by the classes.

```
interface Animal
{
    public void eat();
    public void travel();
}
```

```
public class MammalInt implements Animal
{
    public void eat()
    {
        System.out.println("Mammal eats");
    }
    public void travel()
    {
        System.out.println("Mammal travels");
    }
    public int noOfLegs()
    {
      return 0;
    }
    public static void main(String args[])
    {
      MammalInt m = new MammalInt();
      m.eat(); m.travel();
    }
}
```

Inheritance - Java

- HAS-A relationship:
  - This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.
  - This shows that class Van HAS-A Speed.
  - By having a separate class for Speed, we do not have to put the entire code that belongs to speed inside the Van class., which makes it possible to reuse the Speed class in multiple applications.

```
public class Vehicle{}

public class Speed{}

public class Van extends Vehicle {
   private Speed sp;
}
```

Class & Interface - Java

- An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.
- An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

Class & Interface - Java

- Extending Interfaces:
  - An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.
  - The following Sports interface is extended by Hockey and Football interfaces.

Class & Interface - Java

```
//Filename: Sports.java
public interface Sports {
    public void setHomeTeam(String name);
    public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports {
    public void homeTeamScored(int points);
    public void visitingTeamScored(int points);
    public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports {
    public void homeGoalScored();
    public void visitingGoalScored();
    public void endOfPeriod(int period);
    public void overtimePeriod(int ot);
}
```

Class & Interface - Java

- When overriding methods defined in interfaces there are several rules to be followed:
  - The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
  - An implementation class itself can be abstract and if so interface methods need not be implemented.
- When implementation interfaces there are several rules:
  - A class can implement more than one interface at a time.
  - A class can extend only one class, but implement many interfaces.
  - An interface can extend another interface, similarly to the way that a class can extend another class.

Class & Interface - Java

- An interface is similar to a class in the following way:
  - An interface can contain any number of methods.
- An interface is different from a class in several ways, including:
  - You cannot instantiate an interface.
  - An interface does not contain any constructors.
  - All of the methods in an interface are abstract.
  - An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
  - An interface is not extended by a class; it is implemented by a class.
  - An interface can extend multiple interfaces.

Class & Interface - Java

- Interfaces have the following properties:
  - An interface is implicitly abstract.
  - Each method in an interface is also implicitly abstract.
  - Methods in an interface are implicitly public.

- When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

Class & Interface - Java

```
interface Animal
{
    public void eat();
    public void travel();
}
```

```
/* File name : Mammal.java */
public class Mammal implements Animal {
      public void eat() {
      System.out.println("Mammal eats");
}

public void travel() {
     System.out.println("Mammal travels");
}

public int noOfLegs() {
     return 0;
}

public static void main(String args[]){
     Mammal m = new Mammal();
     m.eat();
     m.travel(); }
}
```

Distributed Objects- properties

Polymorphism

- *Polymorphism* is the ability, for a method, to accept parameters of different types and to have a different behavior for each of these types. Thus an object may be replaced, as a parameter of a method, by a "*compatible*" object.
- The notion of *compatibility*, or conformance is expressed by a relationship between types, which depends on the specific programming model or language being used.

Polymorphism - Java

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

- A Deer IS-A Animal
- A Deer IS-A Vegetarian
- A Deer IS-A Deer
- A Deer IS-A Object

- When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

Remote Objects

- *Encapsulation:* a powerful tool in a *heterogeneous* environment.
  - The user of an object only needs to know an *interface* for that object, which may have *different implementations* on *different locations*.

- *Dynamic creation* of object *instances* allows different objects to be created with the same interface, possibly at different remote locations –
  - middleware must again provide a mechanism for remote object creation, in the form of factories

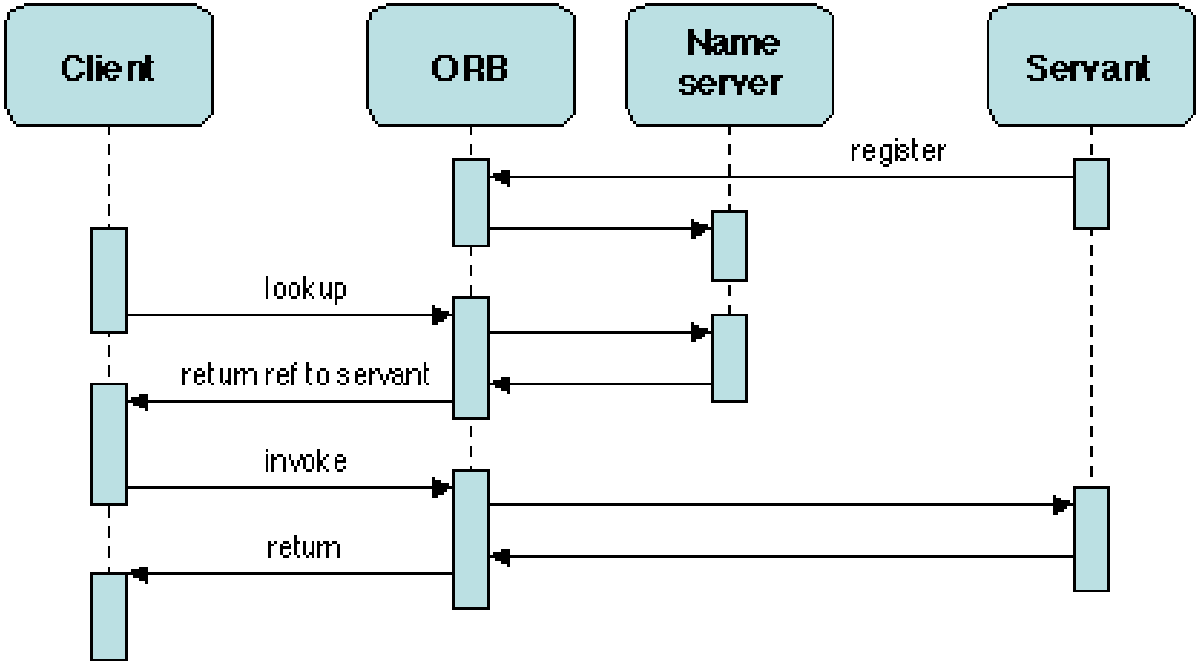
- *Inheritance* and *polymorphism* is useful for distributed applications developers, who are confronted with a *changing environment* and have to define new classes to deal with new situations.
  - A generic (base) class is first designed to capture a set of object features that are common to a wide range of expected situations. Specific, more specialized, classes are then defined by extending the base class

Remote Objects

- Objects that make up an application may be located on distributed sites.
- A client application may use an *object located on a remote site* by calling a method of the object's interface, as if the object were local.
  - Objects used in this way are called *remote objects*, and a method call on a remote object is called *Remote Method Invocation*.
- Middleware must:
  - locate an implementation of the servant object on a remote site,
  - send the parameters to the object's location,
  - perform the call,
  - and return the results to the caller.
- A middleware that performs these tasks is an Object Request Broker, or ORB.

Remote Objects

- The remote object must first be located, which is usually done by means of a name server or trader; then the call itself is performed. Both the lookup and the invocation are mediated through the ORB.



Patterns for Distributed Object Middleware - Proxy

- The PROXY pattern is one of the first design patterns identified in distributed programming
- **Context**. Applications organized as a set of objects in a distributed environment
  - A client requests a service provided by some possibly remote object (the servant).
- **Problem**. Define an access mechanism that:
  - does not involve hard-coding the location of the servant into the client code,
  - does not necessitate deep knowledge of the communication protocols by the client

Patterns for Distributed Object Middleware - Proxy

- **Desirable Properties**. Transparency. Access should be efficient at run time. Programming should be simple for the client.

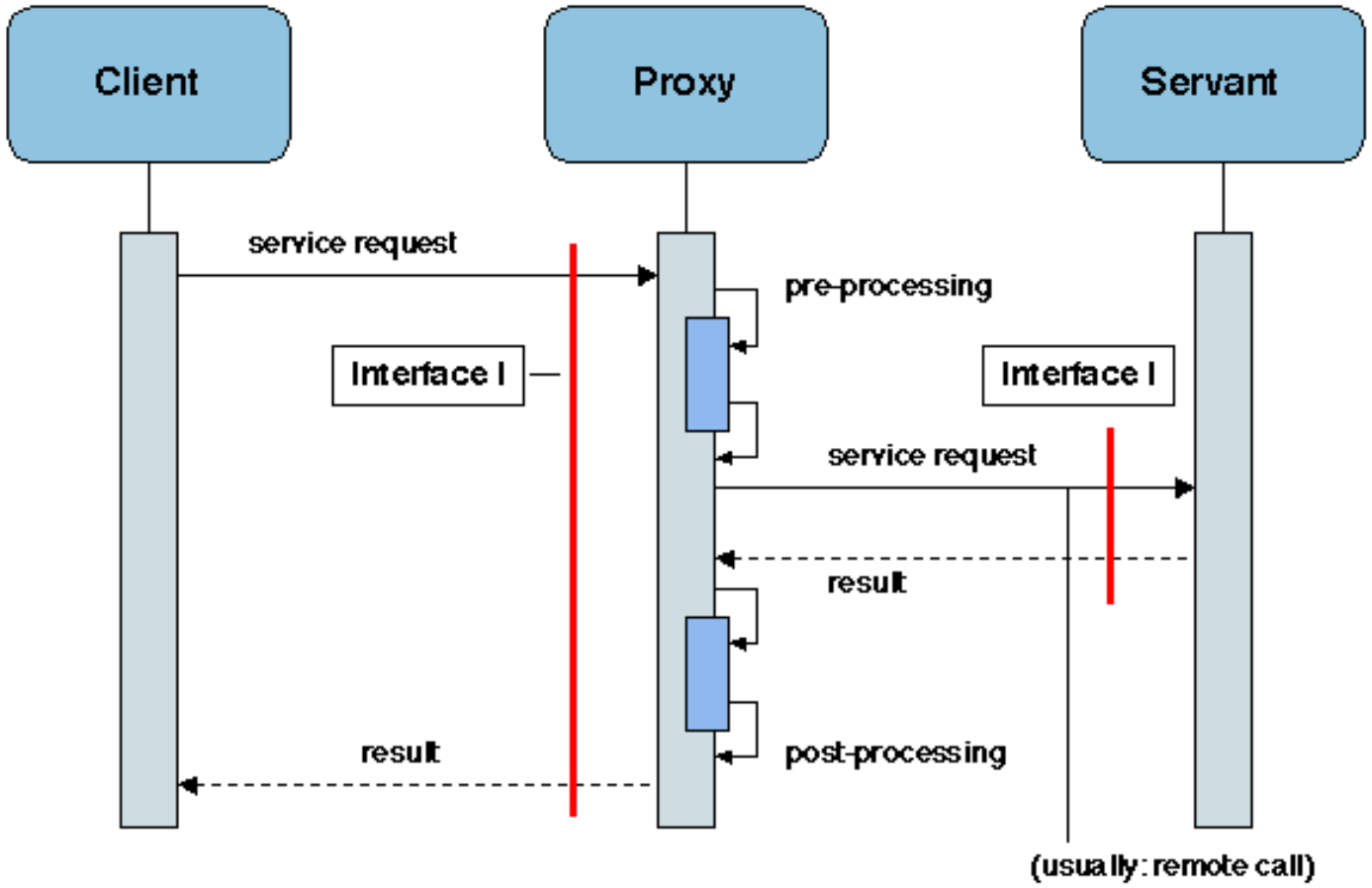
- **Constraints**. The client and the server are in different address spaces.

- **Solution**. Use a local representative of the server on the client site. All information related to the communication system and to the location of the servant is hidden in the proxy, and thus invisible to the client.

Patterns for Distributed Object Middleware - Proxy

- A pre-processing phase, which essentially consists of marshalling the parameters and preparing the request message.
- The actual invocation of the servant, using the underlying communication protocol to send the request and to receive the reply.
- A post-processing phase, which essentially consists of unmarshalling the return values.

Patterns for Distributed Object Middleware - Proxy



Patterns for Distributed Object Middleware - Factory

- **Context**. Applications organized as a set of objects in a distributed environment
- **Problem:** Dynamically create families of related objects (without specifying the exact class of object that will be created), while allowing some decisions to be deferred to run time (such as choosing a concrete subclass to implement a given interface).
- The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects

Patterns for Distributed Object Middleware - Factory

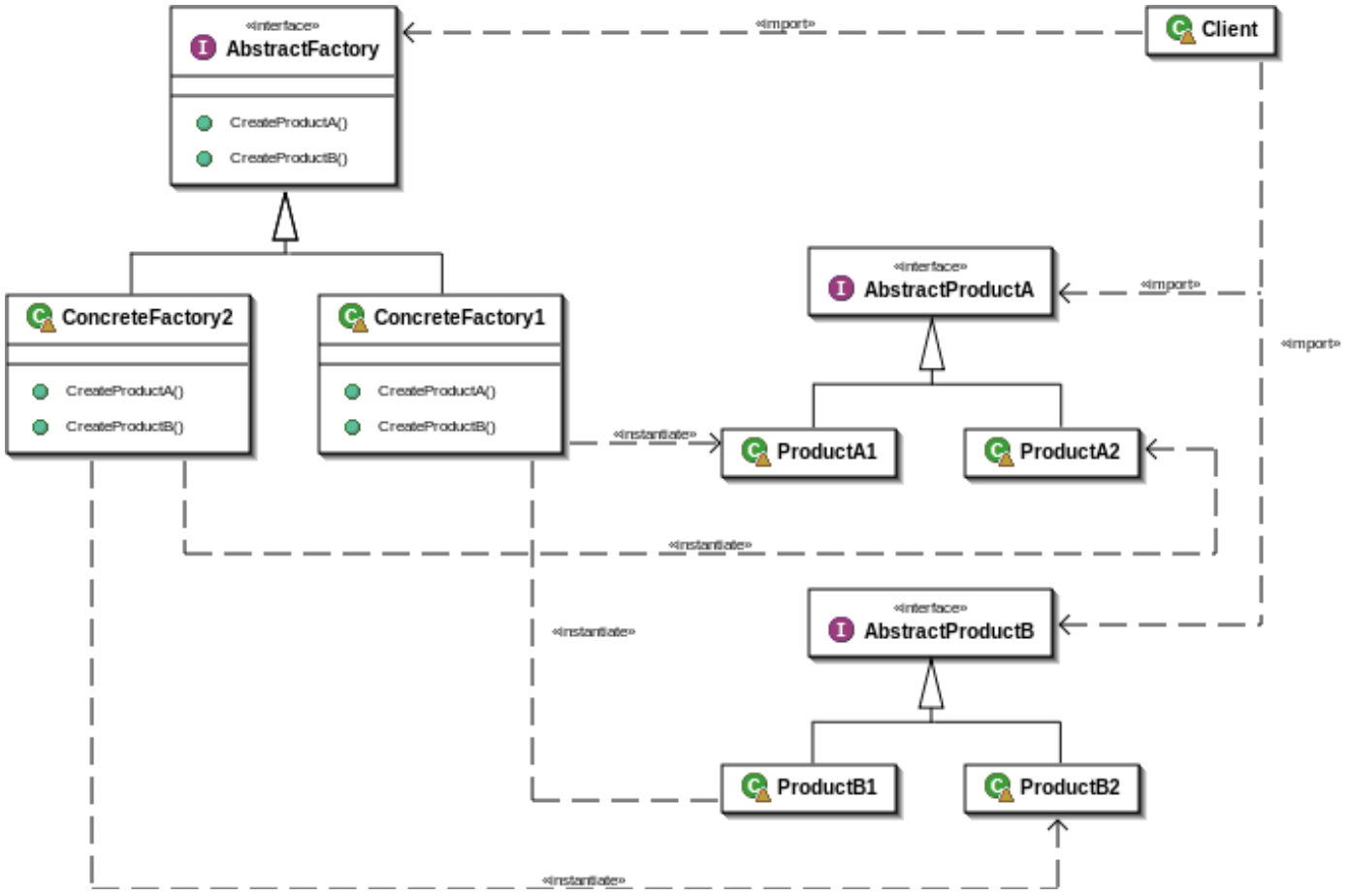
- **Desirable Properties**.
  - The implementation details of the created objects should be abstracted away.
  - The creation process should allow parameters.
  - Evolution of the mechanism should be easy (no hard-coded decisions).

- **Constraints**. The main constraint results from the distributed environment: the client (requesting object creation) and the server (actually performing creation) are in different address spaces.

Patterns for Distributed Object Middleware - Factory

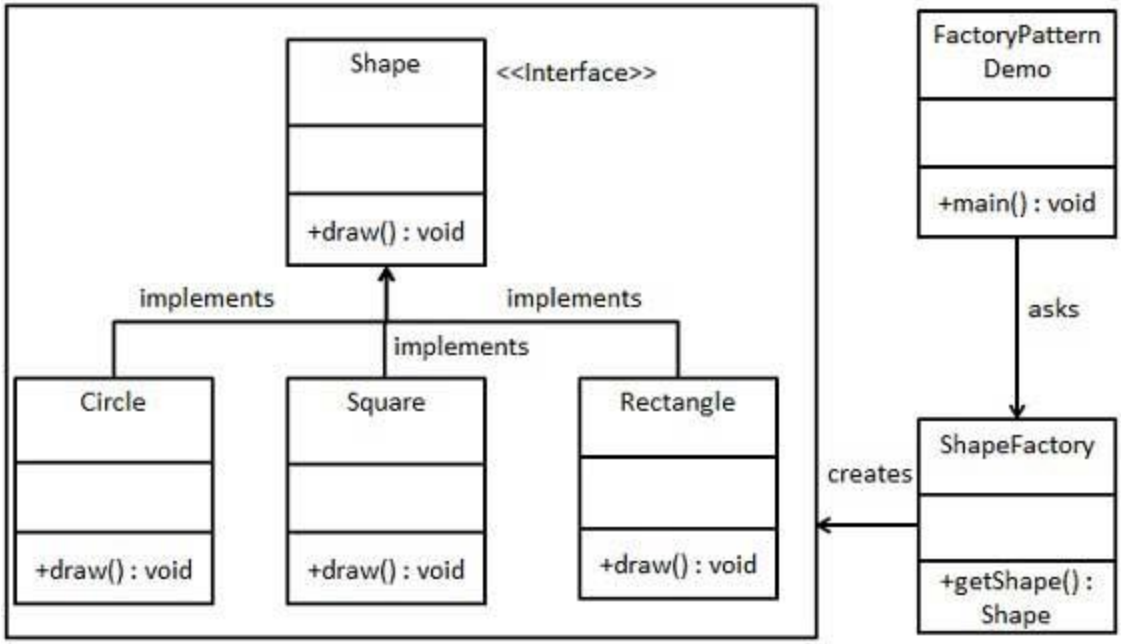
- **Solution**:
- Use two related patterns: An Abstract Factory defines a generic interface for creating objects; A Concrete Factory performs the actual creation of the object for that interface.
  - Abstract Factory provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes
  - The client code has no knowledge whatsoever of the concrete type, not needing to include any header files or class declarations related to it. The client code deals only with the abstract type.
  - Objects of a concrete type are created by the factory, but the client code accesses such objects only through their abstract interface.

Patterns for Distributed Object Middleware - Factory



Patterns for Distributed Object Middleware - Factory

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined:



Patterns for Distributed Object Middleware - Factory

Step 1

Create an interface.

Shape.java

```
public interface Shape {
   void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}
```

Square.java

```
public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}
```

Circle.java

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```

Patterns for Distributed Object Middleware - Factory

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
}
```

Patterns for Distributed Object Middleware - Factory

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

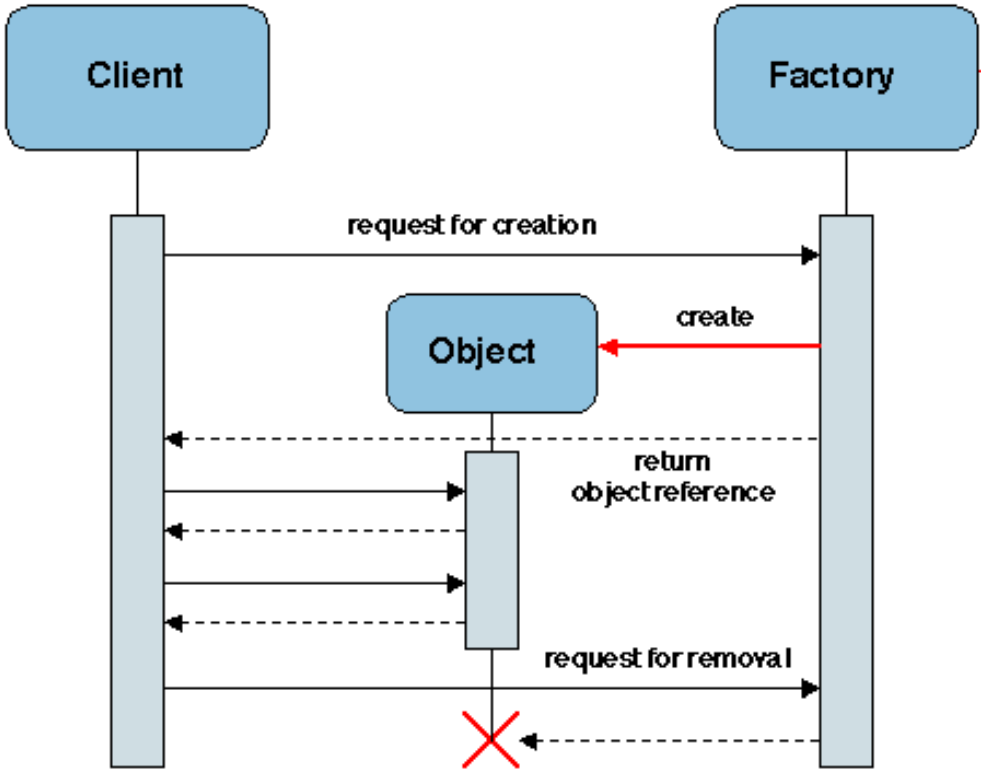
FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

Patterns for Distributed Object Middleware - Factory

- A Factory may also be used as a manager of the objects that it has created
- A Factory may thus implement a method to look up an object (returning a reference for it), and to remove an object upon request.

Patterns for Distributed Object Middleware - Factory



Patterns for Distributed Object Middleware - Adapter

- **Context**. Service *provision*, in a distributed environment:
  - a service is defined by an interface;
  - clients request services;
  - servants located on remote servers, provide services.

- **Problem**. Reuse an existing servant by providing a different interface for its functions in order to comply to the interface expected by a client (or class of clients).

Patterns for Distributed Object Middleware - Adapter

- **Desirable Properties**.
  - The interface conversion mechanism should be run-time efficient.
  - It should also be easily adaptable, in order to respond to unanticipated changes in the requirements.
- **Constraints**. No specific constraints.
- **Solution**. Provide a component (the adapter, or wrapper) that screens the servant by intercepting method calls to its interface.

  Each call is prefixed by a prologue and followed by an epilogue in the adapter.

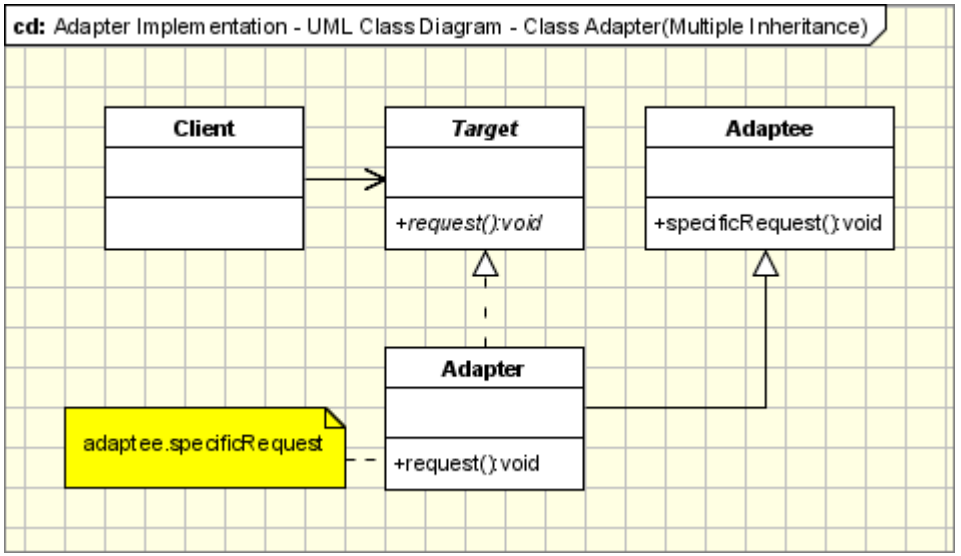
Examples: Portable Object Adapter (POA) of CORBA

Java Connector Architecture (JCA).

Patterns for Distributed Object Middleware - Adapter

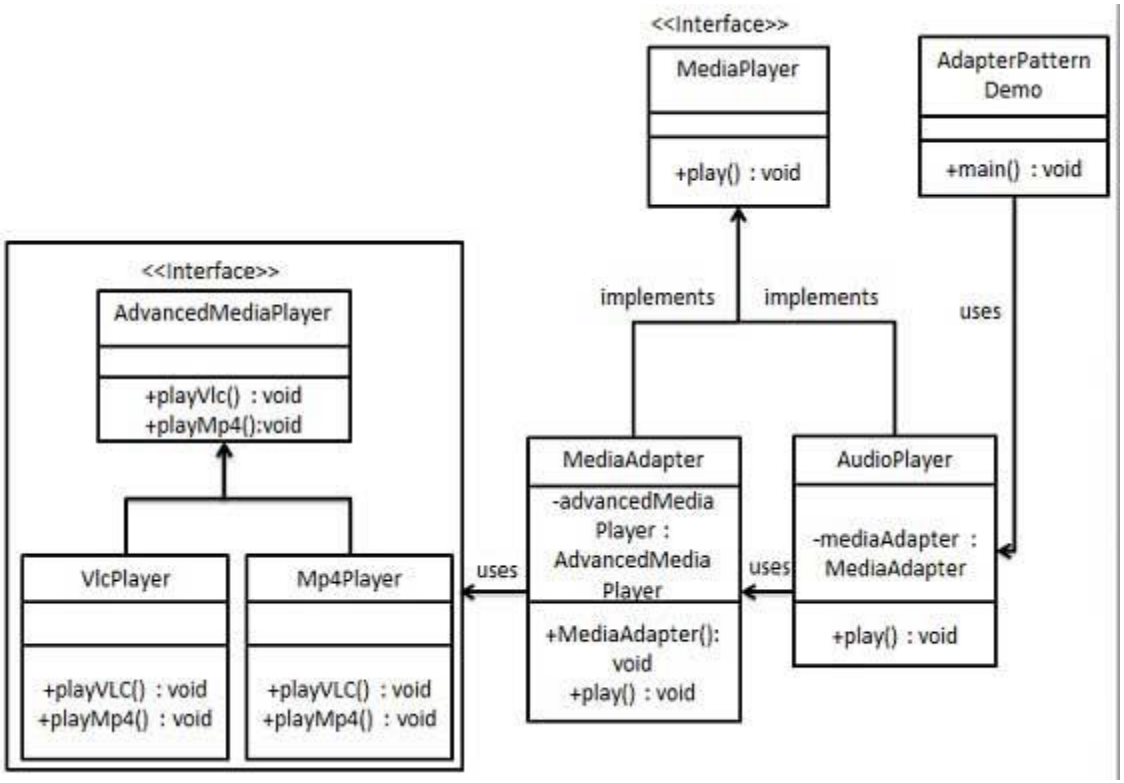
- Adapter Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together, that could not otherwise because of incompatible interfaces.

- **Target** - defines the domain-specific interface that Client uses.
- **Adapter** - adapts the interface Adaptee to the Target interface.
- **Adaptee** - defines an existing interface that needs adapting.



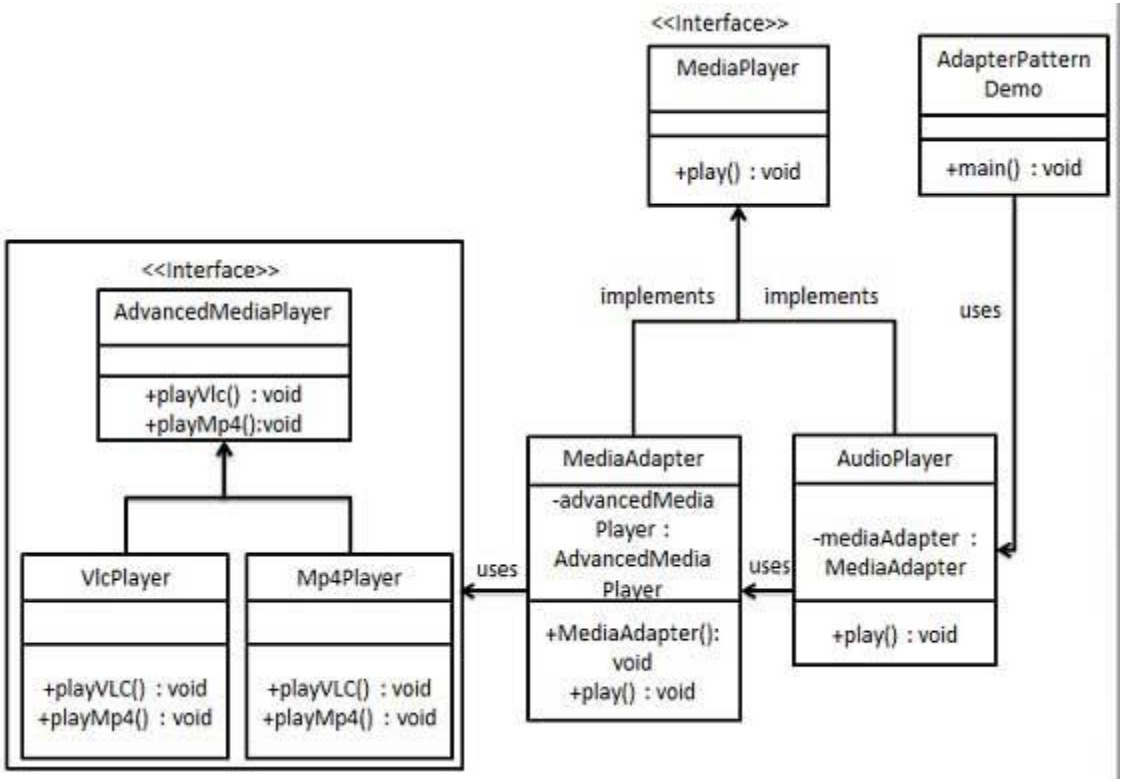
Patterns for Distributed Object Middleware - Adapter

- We've an interface *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the*MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.
- We're having another interface *AdvancedMediaPlayer* and concrete classes implementing the*AdvancedMediaPlayer* interface. These classes can play vlc and mp4 format files.



Patterns for Distributed Object Middleware - Adapter

- We want to make *AudioPlayer* to play other formats as well. To attain this, we've created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer*objects to play the required format.
- *AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use*AudioPlayer* class to play various formats.



Patterns for Distributed Object Middleware - Adapter

Step 1

Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

Patterns for Distributed Object Middleware - Adapter

Step 2

Create concrete classes implementing the *AdvancedMediaPlayer* interface.

VlcPlayer.java

```
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
```

Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //do nothing
   }

   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file. Name: "+ fileName);
   }
}
```

Patterns for Distributed Object Middleware - Adapter

Step 3

Create adapter class implementing the *MediaPlayer* interface.

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {

   AdvancedMediaPlayer advancedMusicPlayer;

   public MediaAdapter(String audioType){
      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();
      } else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }

   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

Patterns for Distributed Object Middleware - Adapter

Step 4

Create concrete class implementing the *MediaPlayer* interface.

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @Override
   public void play(String audioType, String fileName) {

      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: "+ fileName);
      }
      //mediaAdapter is providing support to play other file formats
      else if(audioType.equalsIgnoreCase("vlc")
         || audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }
      else{
         System.out.println("Invalid media. "+
            audioType + " format not supported");
      }
   }
}
```

Patterns for Distributed Object Middleware - Adapter

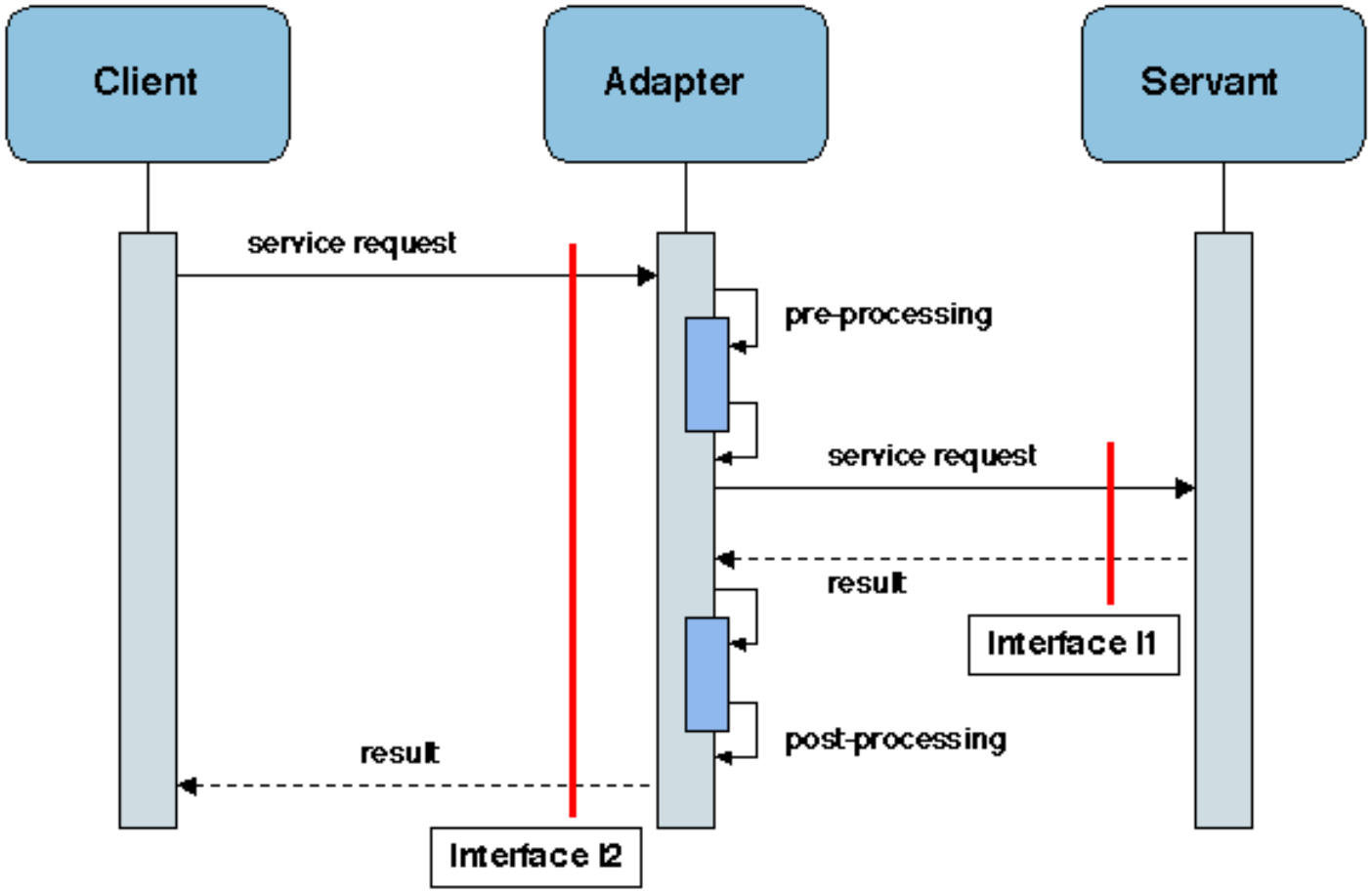
Step 5

Use the AudioPlayer to play different types of audio formats.

AdapterPatternDemo.java

```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

Patterns for Distributed Object Middleware - Adapter



References

- Middleware Architecture with Patterns and Frameworks, Sacha Krakowiak
- Designing Embedded Communications Software, by T. Sridhar, ISBN: 157820125x, CMP Books
- IT Architectures and Middleware – 2nd edition. Chris Britton, Peter Bye. Addison-Wesley
- Tanenbaum & van Steen Distributed Systems: Principles and Paradigms, 2nd ed. ISBN: 0-132-39227-5. [Schmidt et al. 2000]Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. 666 pp.
- [Gamma et al. 1994]Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley.
- [Buschmann et al. 1995]Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1995). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons
- http://www.tutorialspoint.com/java/java_inheritance.htm
- http://www.tutorialspoint.com/java/java_interfaces.htm
- http://www.tutorialspoint.com/design_pattern/factory_pattern.htm