Stateful beans

Let's play a bit...

```
package session;
import javax.ejb.Stateful;
@Stateful
public class StatefulSessionBean implements
StatefulSessionBeanRemote {
    int counter=0;

    @Override
    public String ping() {
        counter++;
        return "SF hits ="+counter;
    }
}
```

Let's play a bit...

```
package session;
import javax.ejb.Stateless;
@Stateless
public class StatelessSessionBean implements
StatefulSessionBeanRemote {
    int counter=0;

    @Override
    public String ping() {
        counter++;
        return "SL hits ="+counter;
    }
}
```

A client with stateful and stateless

```
package _client;
public class _Client {
public static void main(String[] args) throws NamingException {
 Properties jndiProps = new Properties();
 jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
        "org.jboss.naming.remote.client.InitialContextFactory");
 jndiProps.put(Context.URL_PKG_PREFIXES,"org.jboss.ejb.client.naming");
        jndiProps.put(Context.PROVIDER_URL, "remote://localhost:4447");
 jndiProps.put(Context.SECURITY_PRINCIPAL, "user");
 jndiProps.put(Context.SECURITY_CREDENTIALS, "pw");
 jndiProps.put("jboss.naming.client.ejb.context", true);
 Context ctx=new InitialContext(jndiProps);
```

A client with stateful and stateless

```
StatelessSessionBeanRemote bean = (StatelessSessionBeanRemote)
    ctx.lookup("_Server/_Server-ejb/StatelessSessionBean!
    session.StatelessSessionBeanRemote");
StatefulSessionBeanRemote sf_bean = (StatefulSessionBeanRemote)
    ctx.lookup("_Server/_Server-ejb/StatefulSessionBean!
    session.StatefulSessionBeanRemote");
StatelessSessionBeanRemote bean1 = (StatelessSessionBeanRemote)
    ctx.lookup("_Server/_Server-ejb/StatelessSessionBean!
    session.StatelessSessionBeanRemote");
StatefulSessionBeanRemote sf_bean1 = (StatefulSessionBeanRemote)
    ctx.lookup("_Server/_Server-ejb/StatefulSessionBean!
    session.StatefulSessionBeanRemote");
```

A client with stateful and stateless

```
        System.out.println(bean.ping());
        System.out.println(bean.ping());
        System.out.println(bean.ping());
        System.out.println(sf_bean.ping());
        System.out.println(sf_bean.ping());
        System.out.println(sf_bean.ping());
        System.out.println(bean1.ping());
        System.out.println(bean1.ping());
        System.out.println(bean1.ping());
        System.out.println(sf_bean1.ping());
        System.out.println(sf_bean1.ping());
        System.out.println(sf_bean1.ping());
    }
}
```

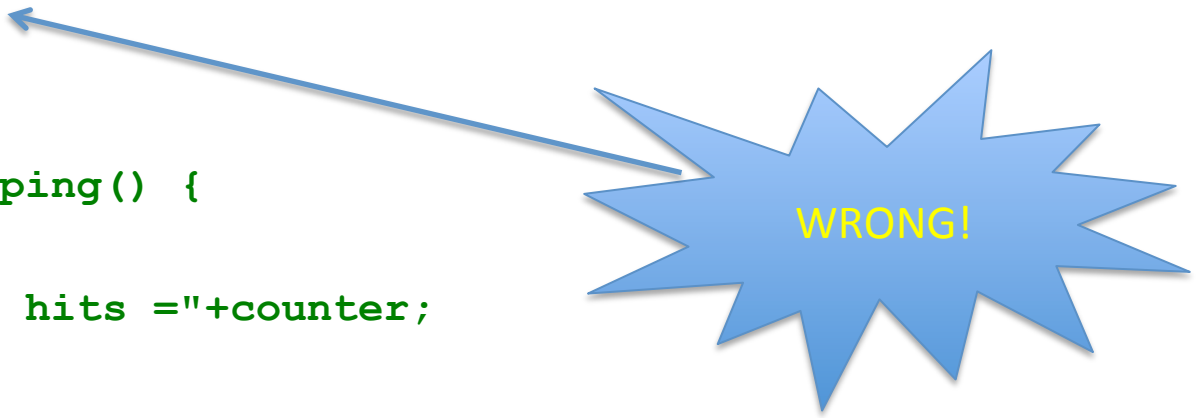
Execution results

SL hits =1
SL hits =2
SL hits =3
SF hits =1
SF hits =2
SF hits =3
SL hits =4
SL hits =5
SL hits =6
SF hits =1
SF hits =2
SF hits =3

No instance variables in stateless!

```
package session;
import javax.ejb.Stateless;
@Stateless
public class StatelessSessionBean implements
StatefulSessionBeanRemote {
    int counter=0;

    @Override
    public String ping() {
        counter++;
        return "SF hits ="+counter;
    }
}
```



EJB Patterns

What is a pattern?

The best solution to a recurring problem"

Recurring software design problems

identified and catalogued in a standard way

so as to be accessibile to everybody and usable in any programming language.

Singleton

- Ensure a class has only one instance and provide a global point of access to it.

```
class Referee{
    static Referee instance= null;
    private Referee() {
      String s = "";
    }
    public static Referee getReferee() {
      if (instance ==null) instance=new Referee();
      return instance;
    }
    public void whistle() {
      //...
    }
}
```

Singleton usage

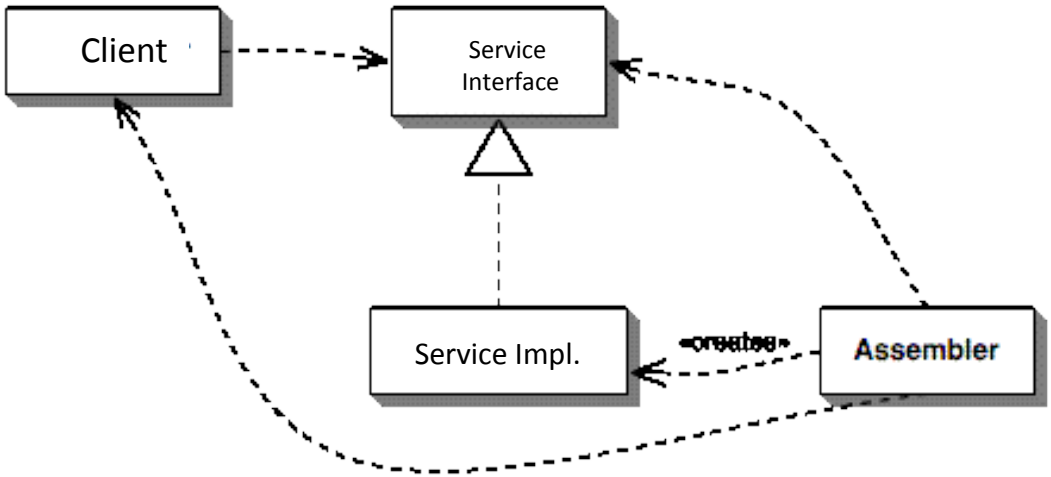
```
package myPackage;

public class Game{
      public static void main(String a[]) {
    new Game ();
  }

  Game () {
    //Referee a=new Referee (); // would give an error!
    Referee b=Referee.getReferee();
    Referee c=Referee.getReferee();
    System.out.println(b==c);
  }
}
```

Factory

Factories are used to encapsulate instantiation.

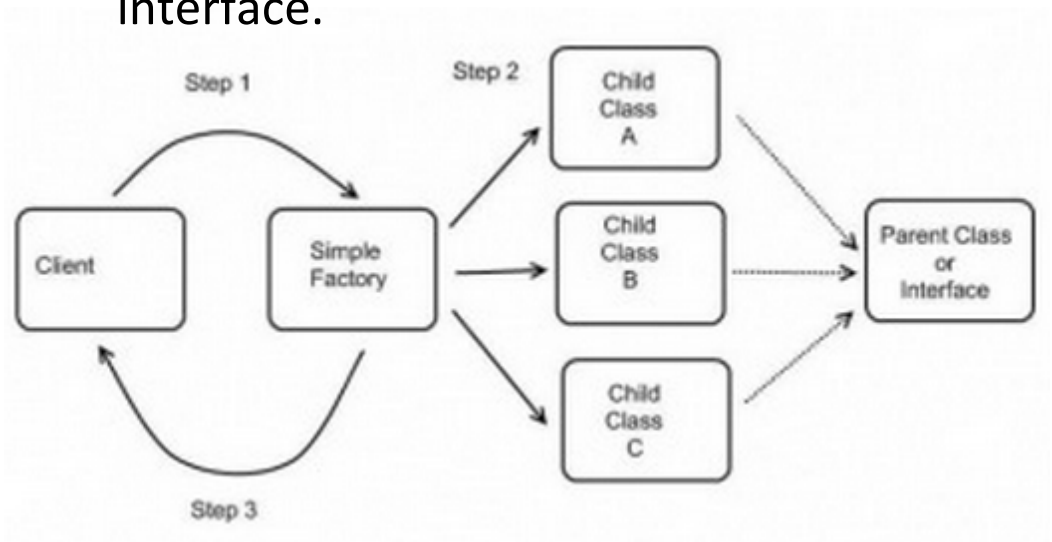


using a Simple Factory

1) you call a (possibly static) method in the factory. The call parameters tell the factory which class to create.

2) the factory creates your object. All the objects it can create either have the same parent class, or implement the same interface.

3) factory returns the object, the client expect is it to match the parent class / interface.

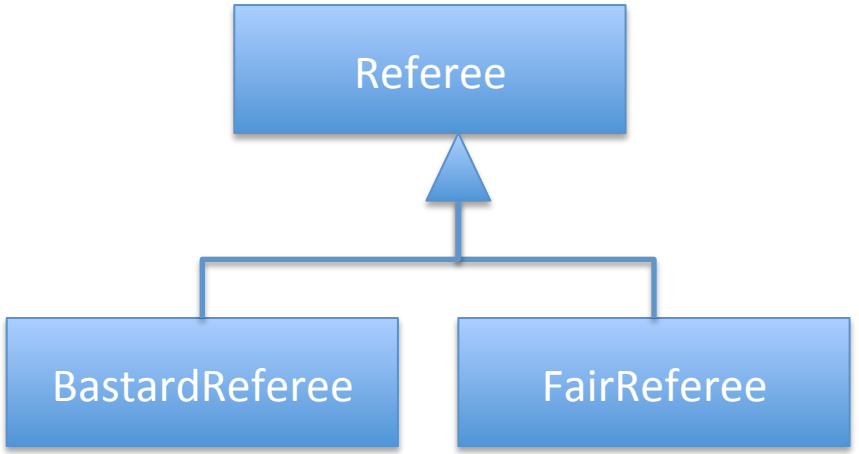



```
Parent x=Factory.create(p);


class Factory{
        static Parent create(Param p) {
                if (p…) return new ChildA();
                else return new ChildB();
        }
}
```

SimpleFactory: isolate the code from the concrete implementating class

```
Referee x=new BastardReferee();
```




```
If (bastardnesslevel==0)
    Referee x=new FairReferee();
else
    Referee x=new BastardReferee();
```

```
Referee x=RefereeFactory.getReferee(bastardnessLevel);
```

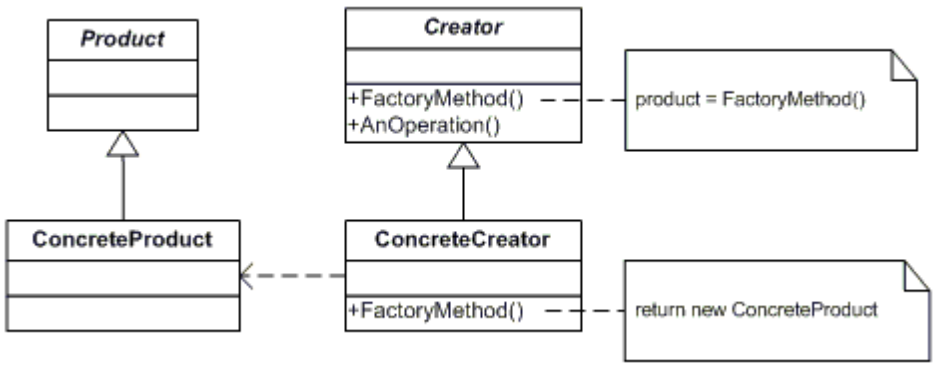
Example

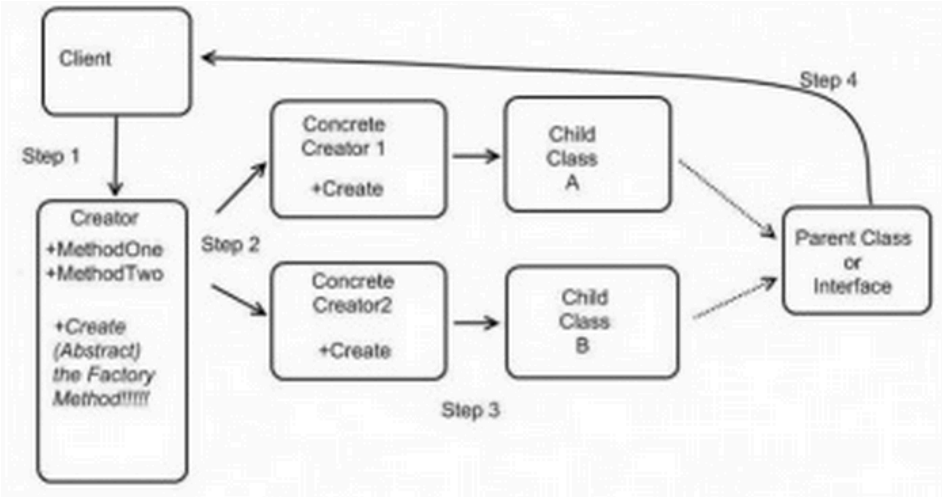
```
SAXParserFactory factory = SAXParserFactory.newInstance();  // singleton
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();  // simple factory
```

Factory method

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Factory Method lets a class defer instantiation to subclasses.





Factory method - example

The products:

```
abstract class Document{…}
class Report extends Document{…}
class Resume extends Document{…}
```

The factories:

```
abstract class DocumentCreator{
        abstract Document create();
}
class ReportCreator extends DocumentCreator {
        Document create() return new Report();
}
class ResumeCreator extends DocumentCreator {
        Document create() return new Resume();
}
```

The client:

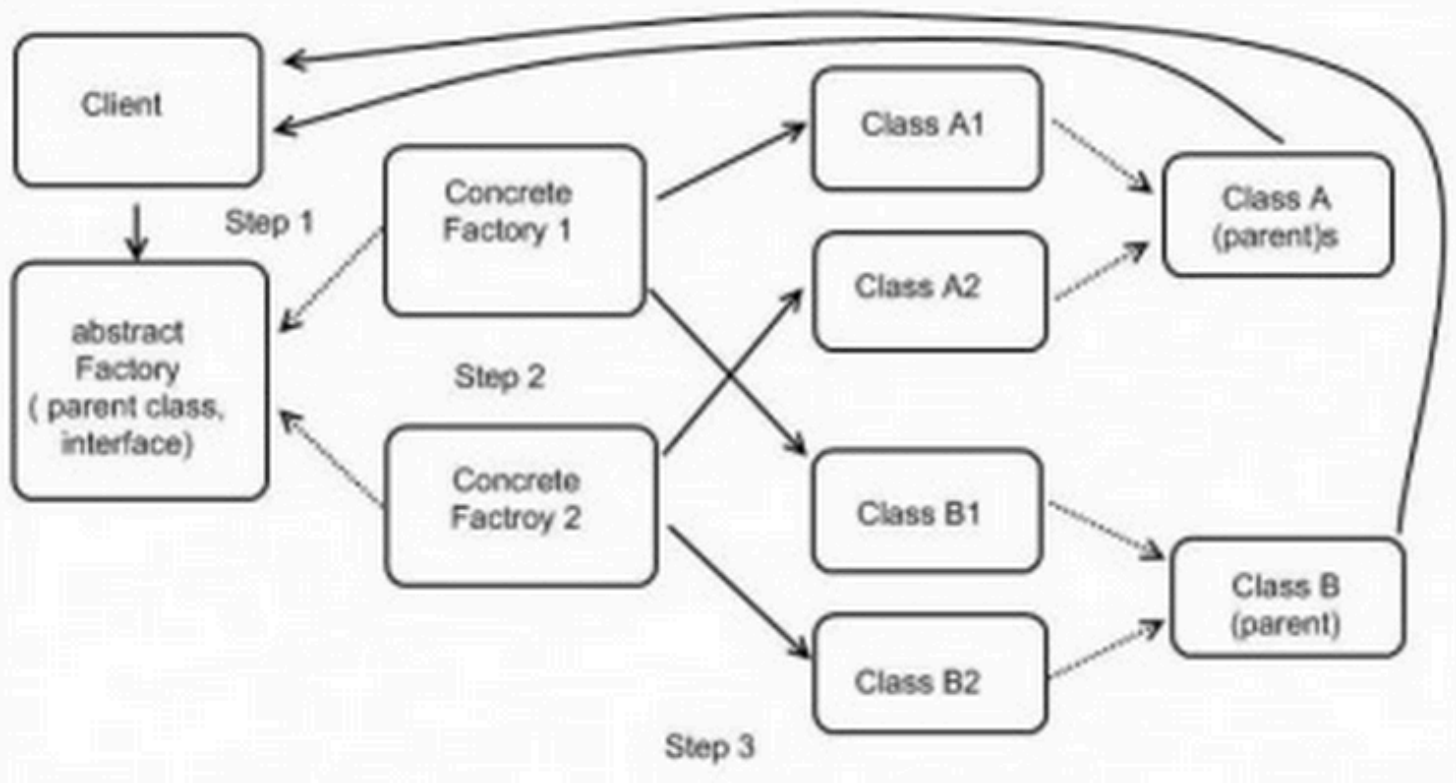
```
Document x=null;
String choice=JOptionPane.showInputDialog("Choose Report (1) or Resume (2)", null);
if (choice.equals("1") x=ReportCreator.create();
if (choice.equals("2") x=DocumentCreator.create();
```

AbstractFactory

Provide an interface for creating

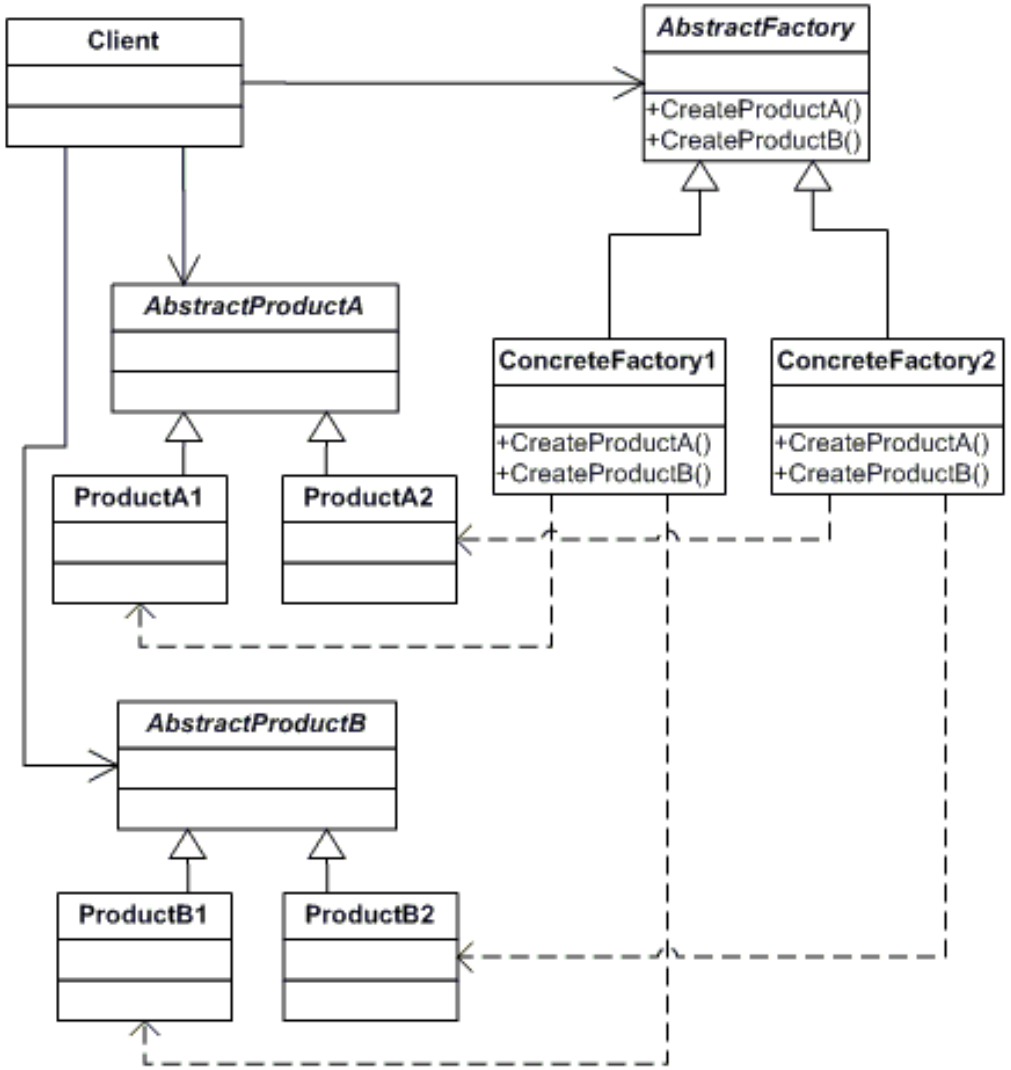
families of related or dependent objects

without specifying their concrete classes.



AbstractFactory

The big difference is that by its own definition, an Abstract Factory is used to create a family of related products, while Factory Method creates one product.



Summary of Factory types

- A Simple Factory is normally called by the client via a static method, and returns one of several objects that all inherit/implement the same parent.
- The Factory Method design is really all about a "create" method that is implemented by sub classes.
- Abstract Factory design is about returning a family of related objects to the client. It normally uses the Factory Method to create the objects.

Service Locator

Have an object that knows how to get hold of all of the services that an application might need.

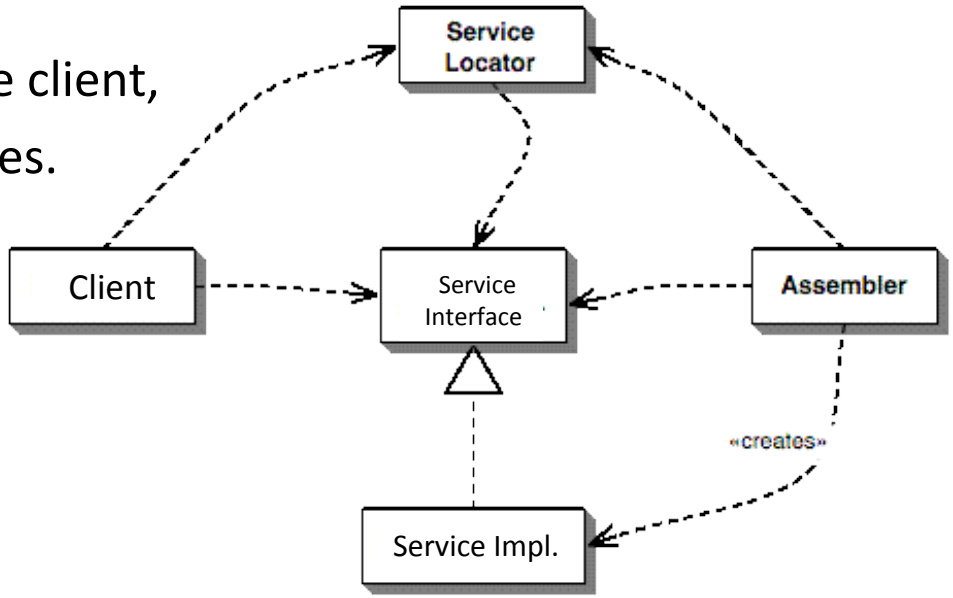
A service locator has a method that, given a key value, returns the implementation of a service when one is needed.

Of course this just shifts the burden:

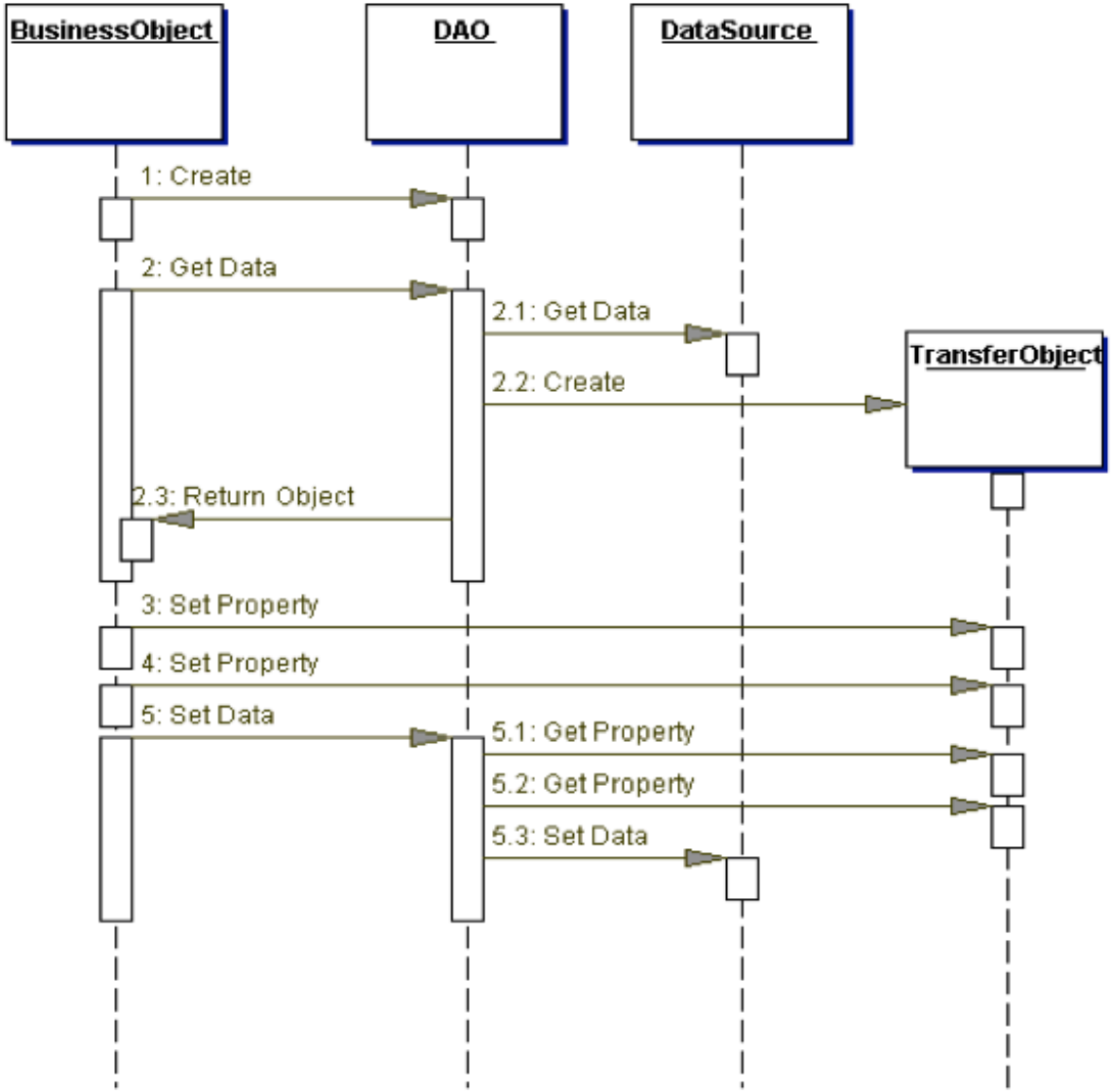
we still have to get the locator into the client,

but this scales well for multiple services.

Example: the rmi registry



DAO – Data Access Object



DTO – Data Transfer Object

also known as **Value Object** or **VO**,

used to transfer data between software application subsystems.

DTO's are often used in conjunction with DAOs to retrieve data from a database.

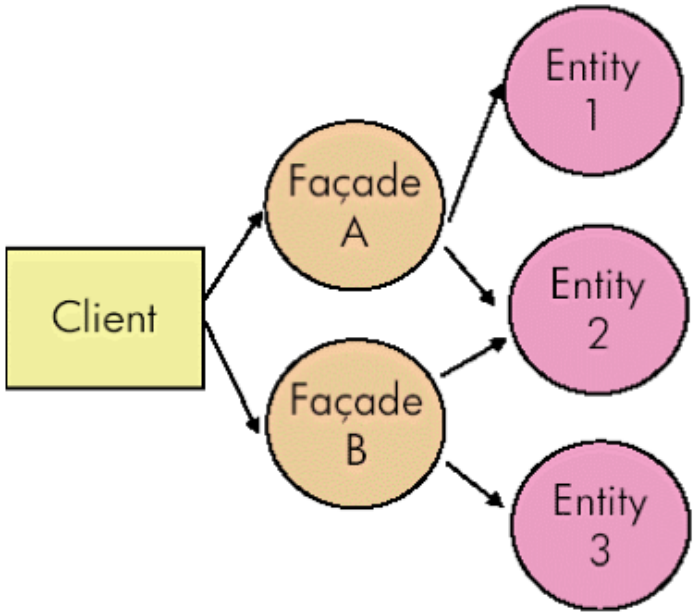
DTOs do not have any behaviour except for storage and retrieval of its own data (mutators and accessor).

Session Facade

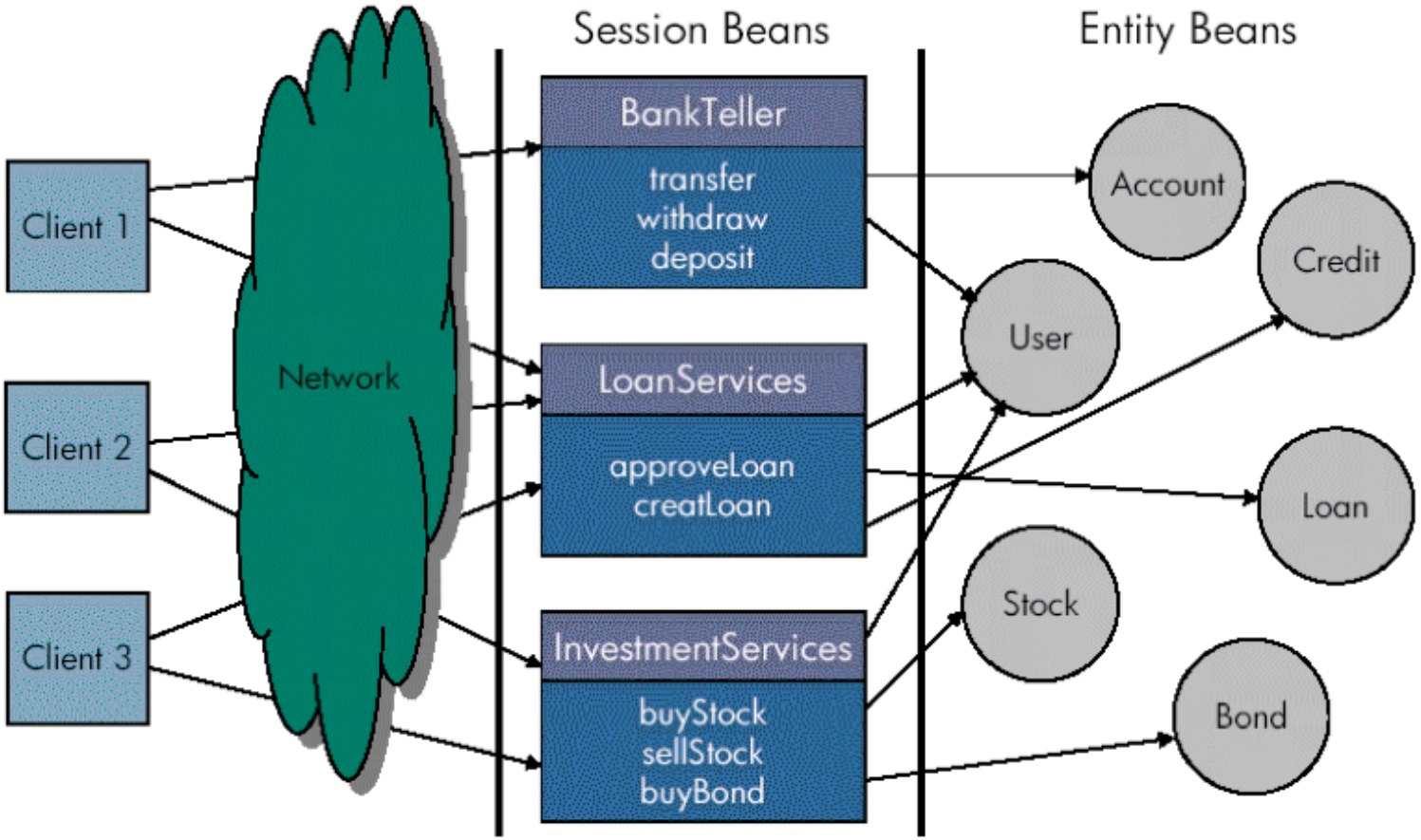
Uses a session bean to encapsulate the complexity of interactions between the business objects participating in a workflow.

Manages the business objects, and provides a uniform coarse-grained servic

access layer to clients



Mapping Session Facade on use cases



Business Delegate Pattern

Use a BusinessDelegate to

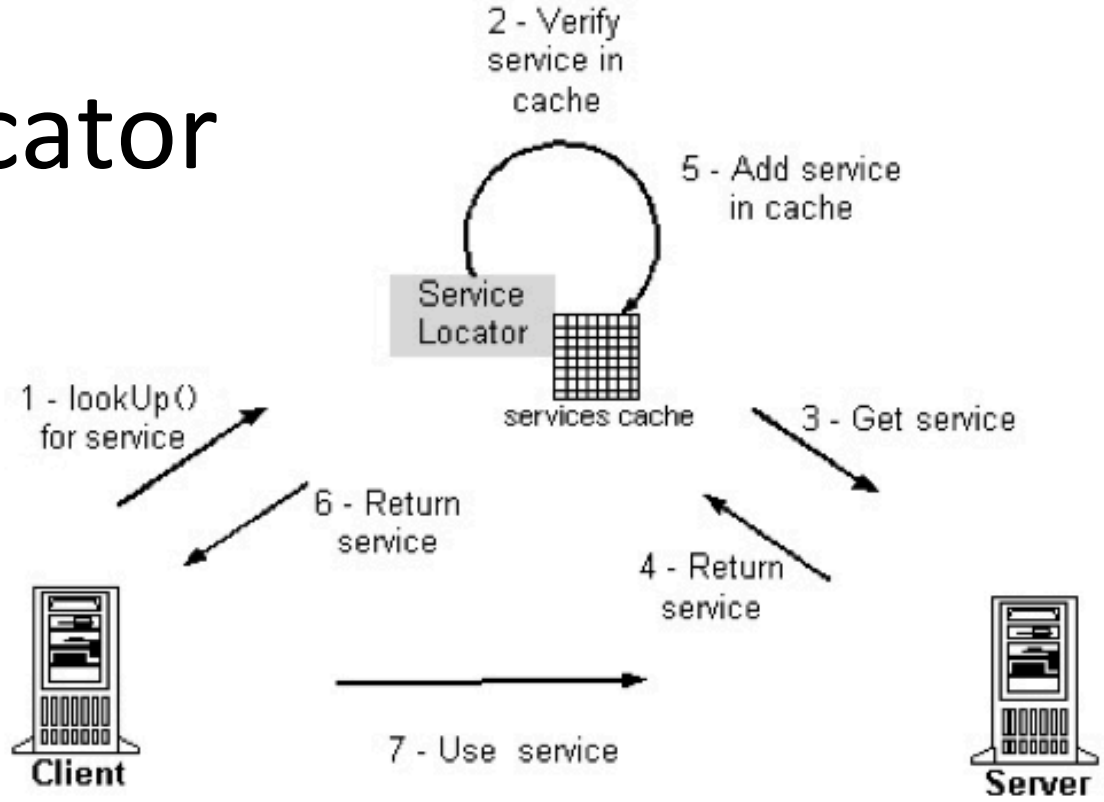
– Reduce coupling between presentation-tier and business service components

– Hide the underlying implementation details of the business service components

– Cache references to business services components
– Cache data
– Translate low level exceptions to application level exceptions – Transparently retry failed transactions
– Can create dummy data for clients

Business Delegate is a plain java class

Service Locator



Use a ServiceLocator to

– Abstract naming service usage

– Shield complexity of service lookup and creation

– Promote reuse

– Enable optimize service lookup and creation functions

• Usually called within BusinessDelegate or Session Facade object

Service Locator

```
package ...; import ...;
public class ServiceLocator throws Exception {
   private static ServiceLocator serviceLocator;
   private static Context context;
   private ServiceLocator() { context =  getInitialContext(); }
   private Context getInitialContext(){
      Hashtable environment = new Hashtable();
      environment.put(..);
      return new InitialContext(environment);
   } public static synchronized ServiceLocator getInstance(){
     if (serviceLocator == null) {
         serviceLocator = new ServiceLocator(); }
     return serviceLocator;
   }
   public Object getBean(…) {return context.lookup(…)}
}
```

Overall view