Transactions in EJB

Declarative Transactions

Who begins a transaction?

Who begins a transaction? *Who* issues either a commit or abort? This is called *demarcating transactional boundaries* .

There are three ways to demarcate transactions:

- *programmatically:*
  you are responsible for issuing a *begin* statement and either a *commit* or an *abort* statement.
- *declaratively,*
  the EJB container *intercepts* the request and starts up a transaction automatically on behalf of your bean.
- *client-initiated.*
  write code to start and end the transaction from the client code outside of your bean.

Programmatic vs. declarative

programmatic transactions:

your bean has full control over transactional boundaries.*For instance,you can use programmatic transactions to run a series of minitransactions within a bean method.*

When using programmatic transactions,always try to complete your transactions in the same method that you began them.Doing otherwise results in spaghetti code where it is difficult to track the transactions;the performance decreases because the transaction is held open longer.

declarative transactions:

your entire bean method must either run under a transaction or not run under a transaction*.*

Transactions are simpler! (just declare them in the descriptor)

Client-initiated

Client initiated transactions:

A non-transactional remote client calls an enterprise bean that performs its own transactions The bean succeeds in the transaction,but the network or application server crashes before the result is returned to a remote client.The remote client would receive a Java RMI RemoteException indicating a network error,but would not know whether the transaction that took place in the enterprise bean was a success or a failure.

With client-controlled transactions, if anything goes wrong,the client will know about it.

*The downside to client-controlled transactions is that if the client is located far from the server, **transactions are likely to take a longer time and the efficiency will suffer**.*

Transactional Models

A *flat transaction* is the simplest transactional model to understand.A flat transaction is a series of operations that are performed atomically as a single *unit of work* .

A *nested transaction* allows you to embed atomic units of work within other units of work.The unit of work that is nested within another unit of work can roll back without forcing the entire transaction to roll back. (*subtransactions can independently roll back without affecting higher transactions in the tree*)
(Not currently mandated by the EJB specification)

Other models: *chained transactions* and *sagas*.
(Not supported by the EJB specification)

EJB Transaction Attribute Values

Required

You want your method to always run in a transaction.

If a transaction is already running, your bean joins in on that transaction. If no transaction is running, the EJB container starts one for you.

Never

Your bean cannot be involved in a transaction.

If the client calls your bean in a transaction, the container throws an exception back to the client (java.rmi.RemoteException if remote, javax.ejb.EJBException if local).

EJB Transaction Attribute Values

Supports

***The method runs only in a transaction if the client had one running already** —it joins that transaction.*

If the client does not have a transaction,the bean runs with no transaction at all.

Mandatory

***a transaction must be already running when your bean method is called**. If a transaction isn ′t running, javax.ejb.TransactionRequiredException is thrown back to the caller (or javax.ejb.TransactionRequiredLocalException if the client is local).*

EJB Transaction Attribute Values

NotSupported

your bean cannot be involved in a transaction at all.

For example,assume we have two enterprise beans,A and B.Let ′s assume bean A begins a transaction and then calls bean B. If bean B is using the NotSupported *attribute,the transaction that A started is suspended. None of B′s operations are transactional,such as reads/writes to databases. When B completes,A ′s transaction is resumed.*

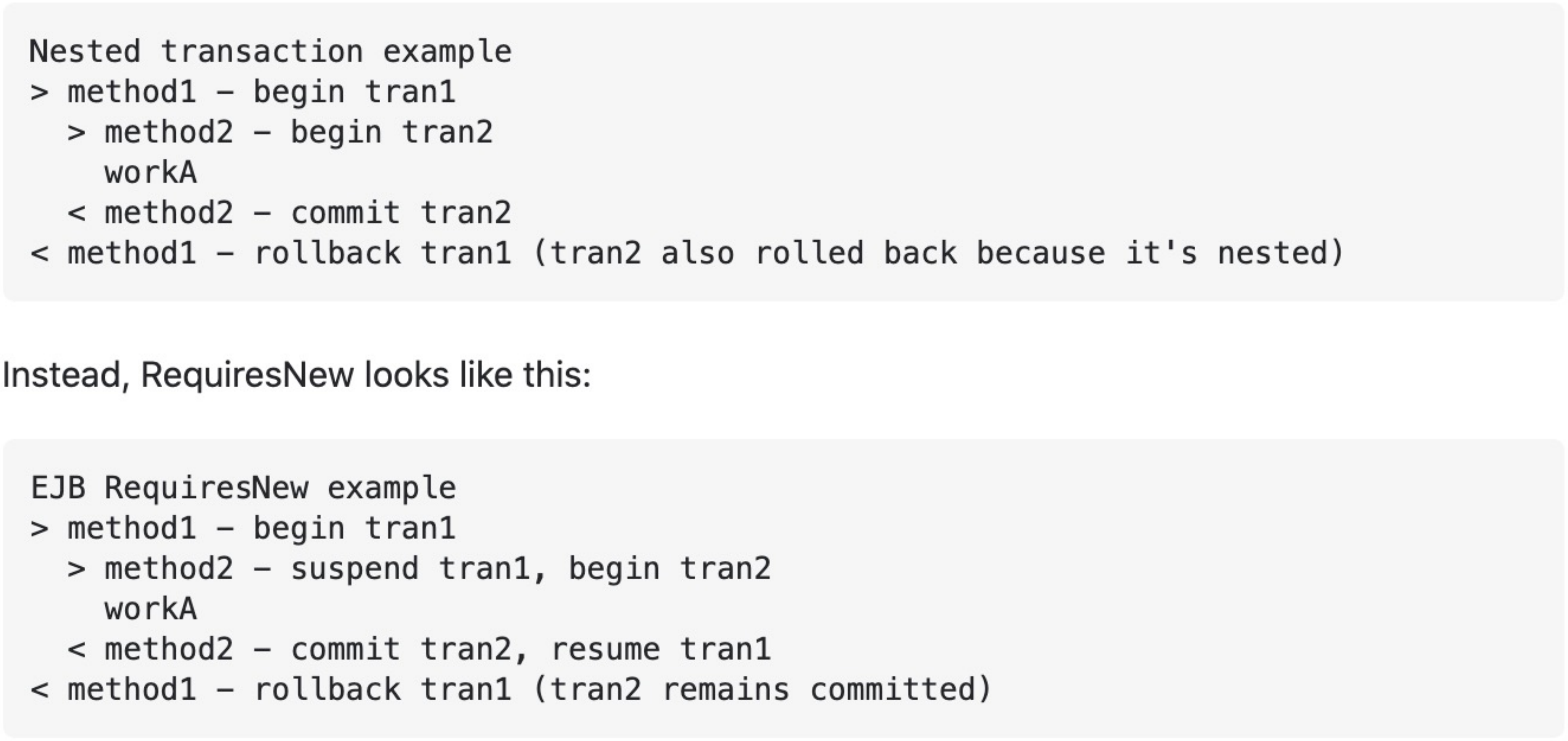
EJB Transaction Attribute Values

RequiresNew

You should use the RequiresNew *attribute if you* ***always want a new transaction to begin*** *when your bean is called. If a transaction is already underway when your bean is called,that transaction is suspended during the bean invocation.*

The container then launches a new transaction and delegates the call to the bean.The bean performs its operations and eventually completes.The container then commits or aborts the transaction and finally resumes the old transaction. If no transaction is running when your bean is called,there is nothing to suspend or resume.

Difference between RequiresNew and Nested



```
Nested transaction example
> method1 - begin tran1
  > method2 - begin tran2
    workA
  < method2 - commit tran2
< method1 - rollback tran1 (tran2 also rolled back because it's nested)
```

Instead, RequiresNew looks like this:

```
EJB RequiresNew example
> method1 - begin tran1
  > method2 - suspend tran1, begin tran2
    workA
  < method2 - commit tran2, resume tran1
< method1 - rollback tran1 (tran2 remains committed)
```

from https://stackoverflow.com/questions/10817838/ejb-3-0-nested-transaction-requires-new

EJB Transaction Attribute Values

TYPE	PRECONDITION	POSTCONDITION
Required	NO transaction	NEW
	PRE-EXISTING	PRE-EXISTING
RequiresNew	NO transaction	NEW
	PRE-ESISTENTE	NEW
Supports	NO transaction	NO transaction
	PRE-EXISTING	PRE-EXISTING
Mandatory	NO transaction	error
	PRE-EXISTING	PRE-EXISTING
NotSupported	Nessuna transazione	NO transaction
	PRE-ESISTENTE	NO transaction
Never	NO transaction	NO transaction
	PRE-EXISTING	error

Annotations

- @Stateless
- @TransactionManagement(javax.ejb.TransactonManagementType.CONTAINER)
- public class Mybean implements Myinterface{
- @PersistencyContext private EntityManager em;
- @Resource private SessionContext ctx;
- …
- @TransactionAttribute(javax.ejb.TransactonManagementType.REQUIRED)
- public void myTransactedMethod(){…
- if (…) ctx.setRollbackOnly;
- }
- }

EJB Transaction Attribute Values

```
<assembly-descriptor>
    <container-transaction>
   <method>
      <ejb-name>Employee</ejb-name>
      <method-name>*</method-name>
   </method>
   <trans-attribute>Mandatory</trans-attribute>
    </container-transaction>
    <container-transaction>
   <method>
      <ejb-name>Employee</ejb-name>
      <method-name>setName</method-name>
      <method-param>String</method-param>
   </method>
   <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

Dooming container-managed transactions

call *setRollbackOnly()* on your *EJB context* object.
If the transaction participant is not an Container Managed EJB component, you can doom a transaction by looking up the JTA and calling the JTA 's *setRollbackOnly()* method,

Container-managed transactional beans can detect doomed transactions by calling the *getRollbackOnly()*method on the EJB context object. If this method returns *true* ,the transaction is doomed.

Isolation levels in EJB

BMT:

you specify isolation levels with your resource manager API (such as JDBC).

For example,you could call java.sql.Connection.SetTransactionIsolation(...).

CMT:

there is no way to specify isolation levels in the deployment descriptor.

You need to either use resource manager APIs (such as JDBC),or rely on your container ′s tools or database ′s tools to specify isolation.

Isolation portability problems

Unfortunately, there is no way to specify isolation for container-managed transactional beans in a portable way—you are reliant on container and database tools.

This means if you have written an application, you cannot ship that application with built-in isolation. The deployer now needs to know about transaction isolation when he uses the container's tools, and the deployer might not know a whole lot about your application's transactional behavior.

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

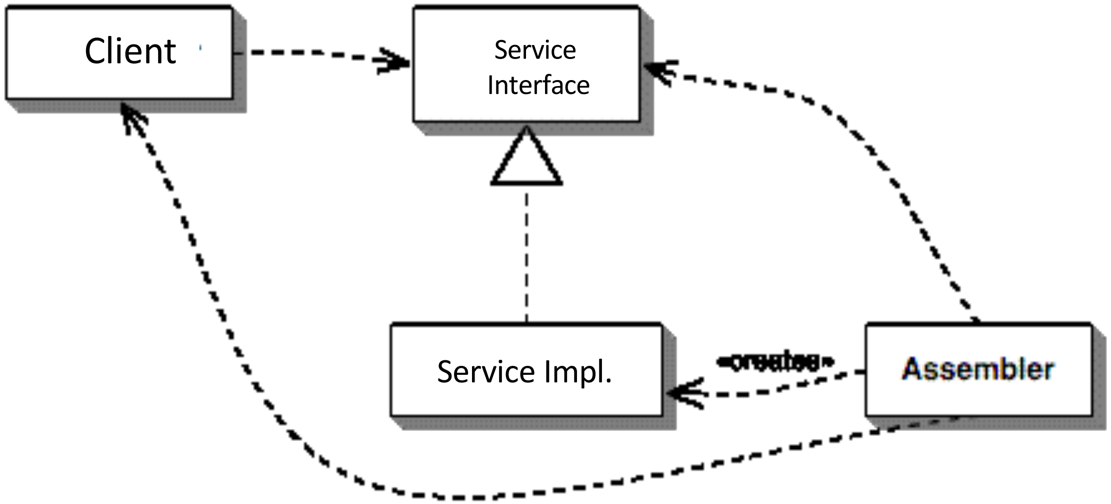
```
package myPackage;

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.



Client

Service Interface

Service Impl.

«creates»

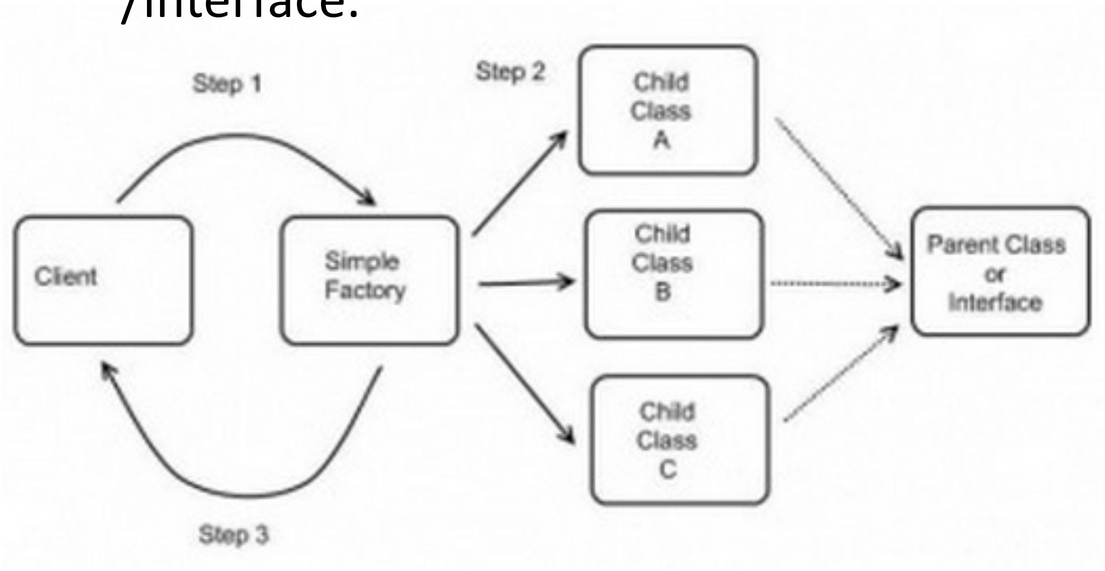
Assembler

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.

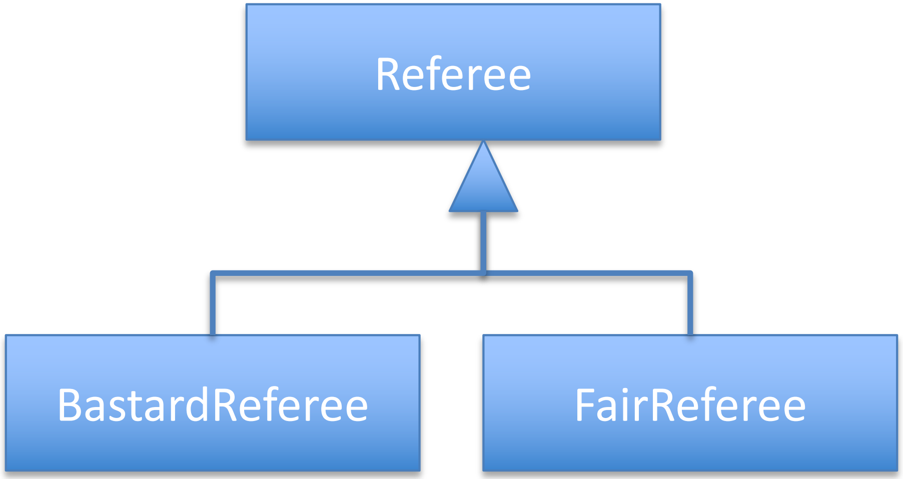
3) factory returns the object, the client expect is it to match the parent class /interface.



```
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);
```

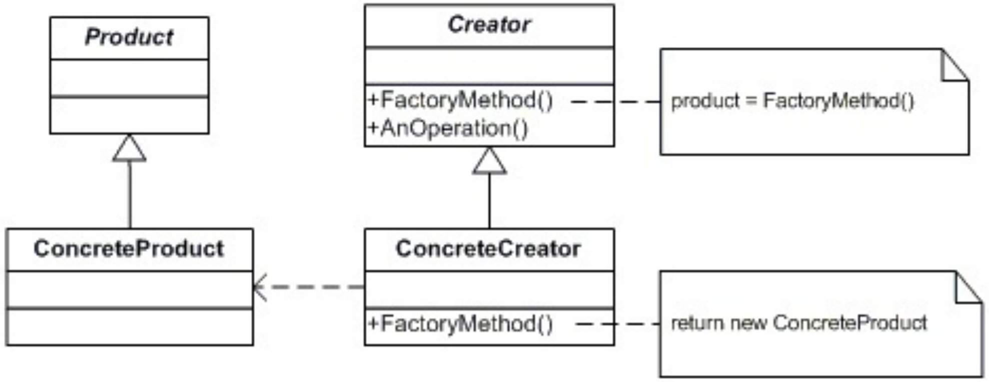
Example

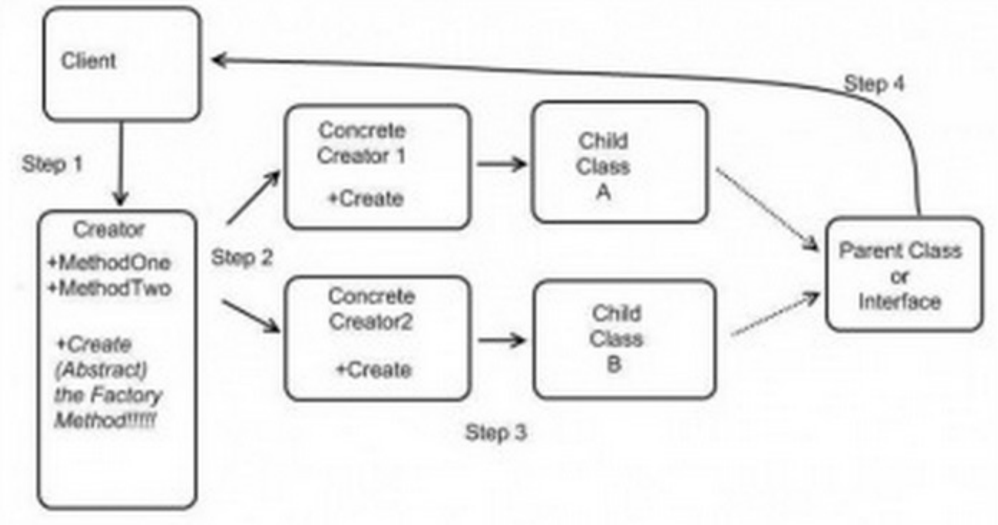
```
SAXParserFactory factory = SAXParserFactory.newInstance();  // singleton
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();  // simple factory
```

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:

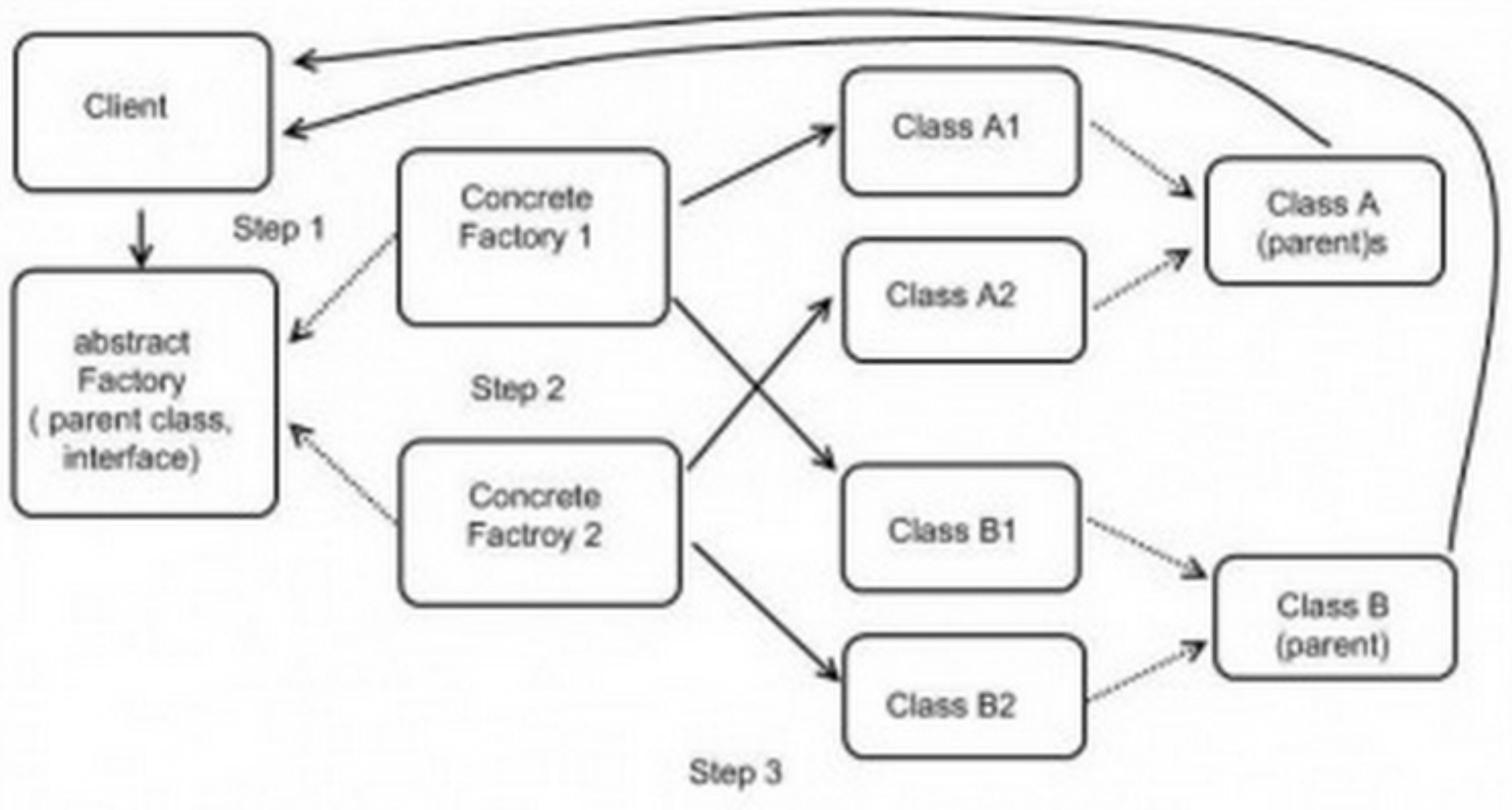
```
Document x=null;
String choice=JOptionPane.showInputDialog("Choose Report (1) or Resume (2)", null);
if (choice.equals("1") x=ReportCreator.create();
if (choice.equals("2") x=DocumentCreator.create();
```

AbstractFactory

Provide an interface for creating

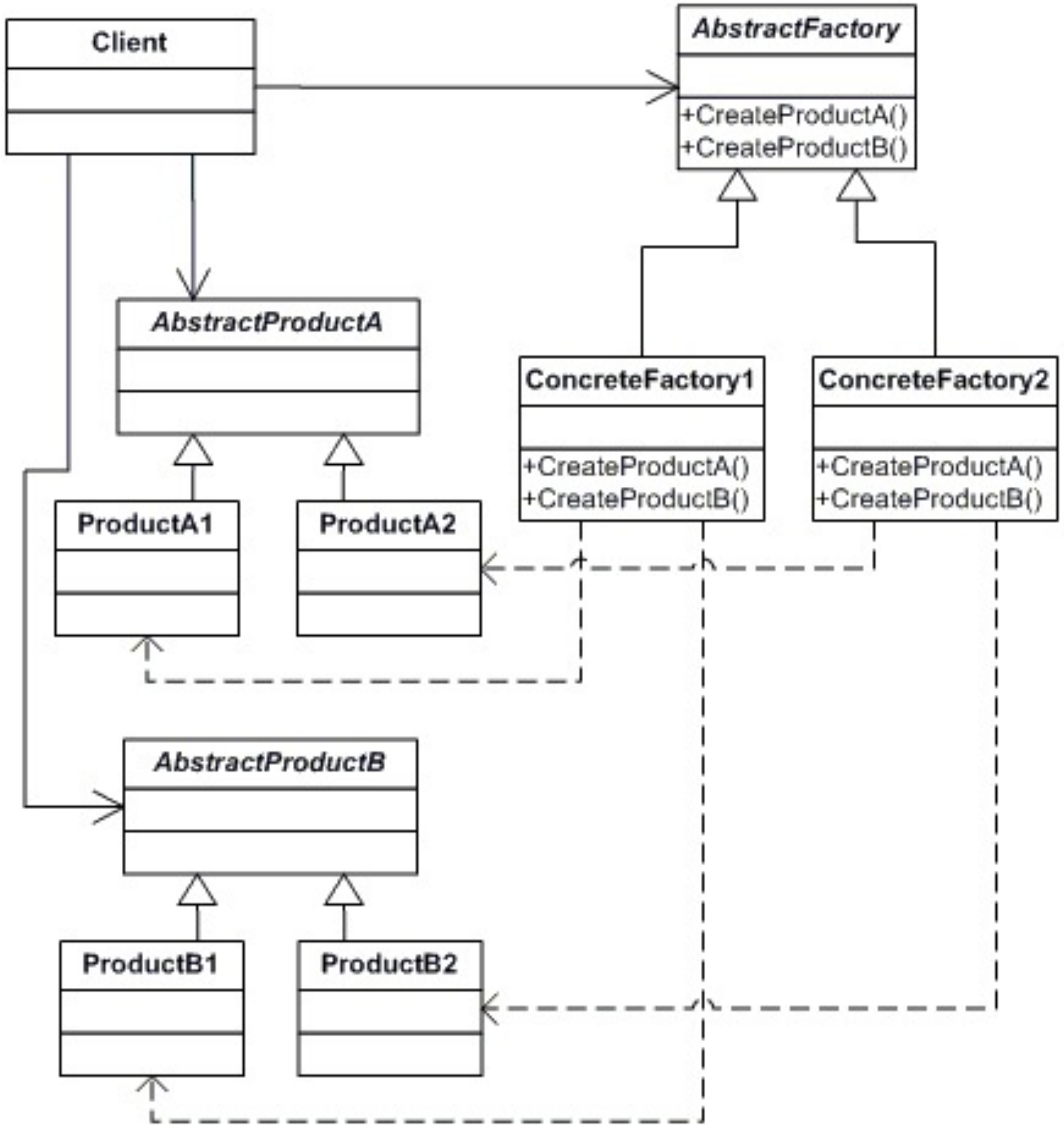
families of related or dependent objects

without specifying their concrete classes.



AbstractFactory

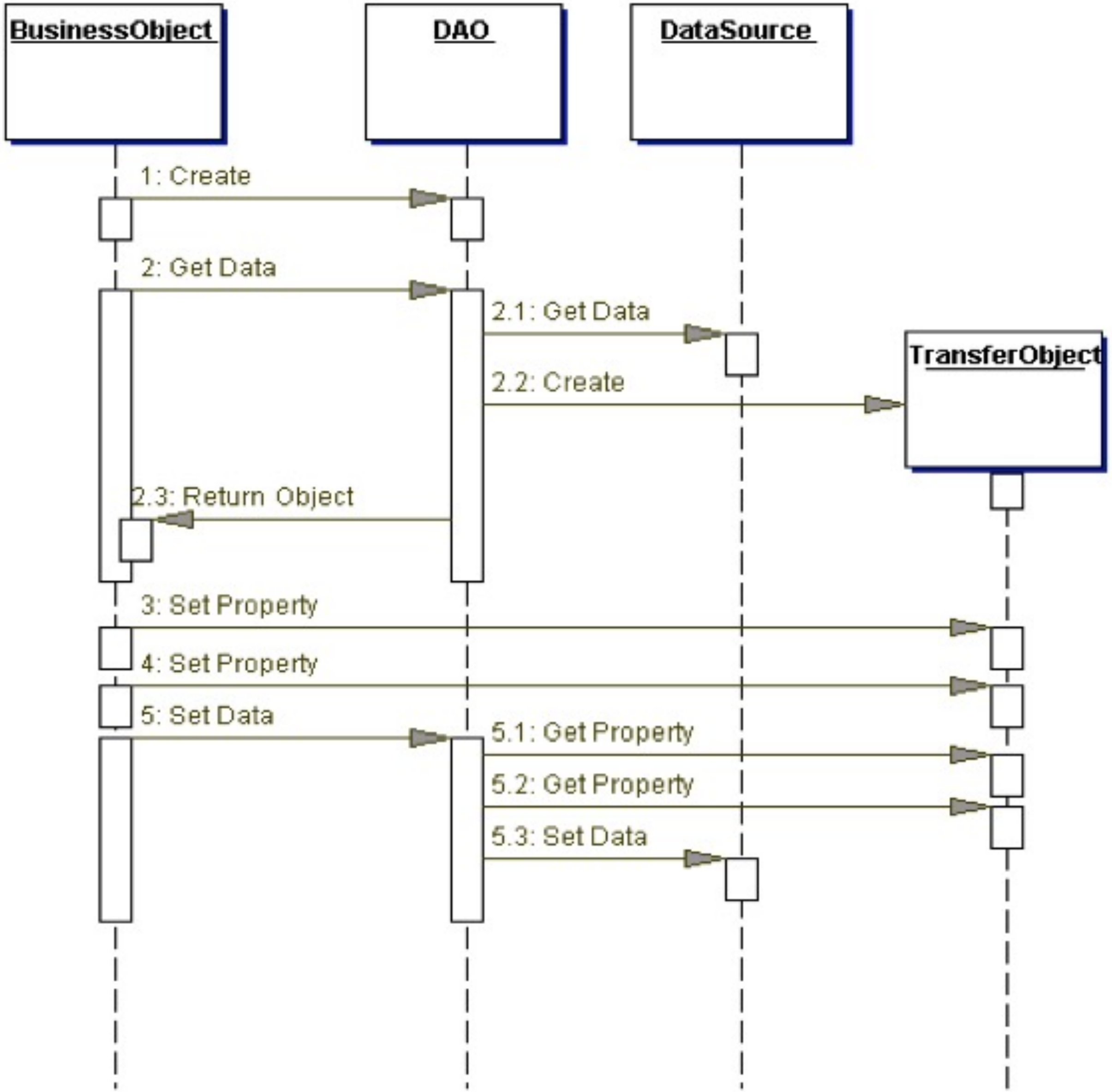
The big difference is that by its own definition, an Abstract Factory is used to create a family of related products, while Factory Method creates one product.



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.

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.

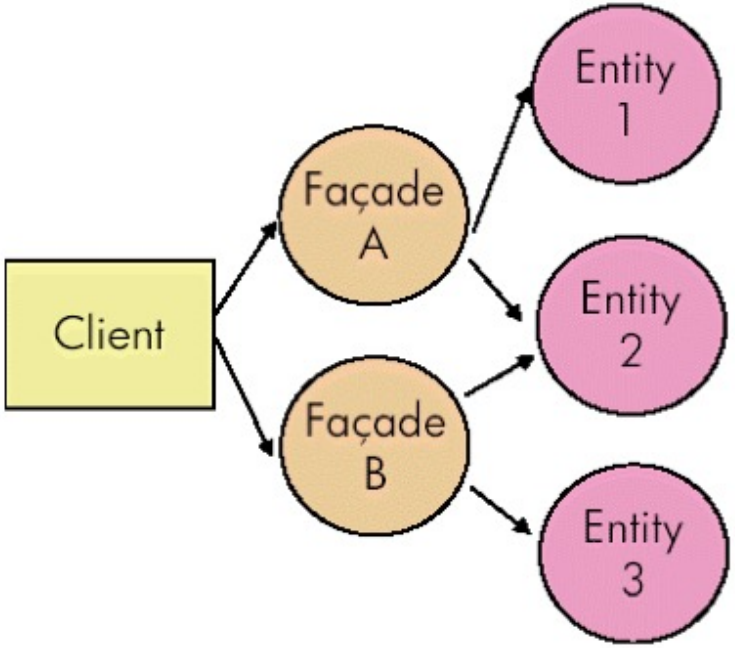
DTOs do not have any behaviour except for storage and retrieval of its own data (mutators and accessor).

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



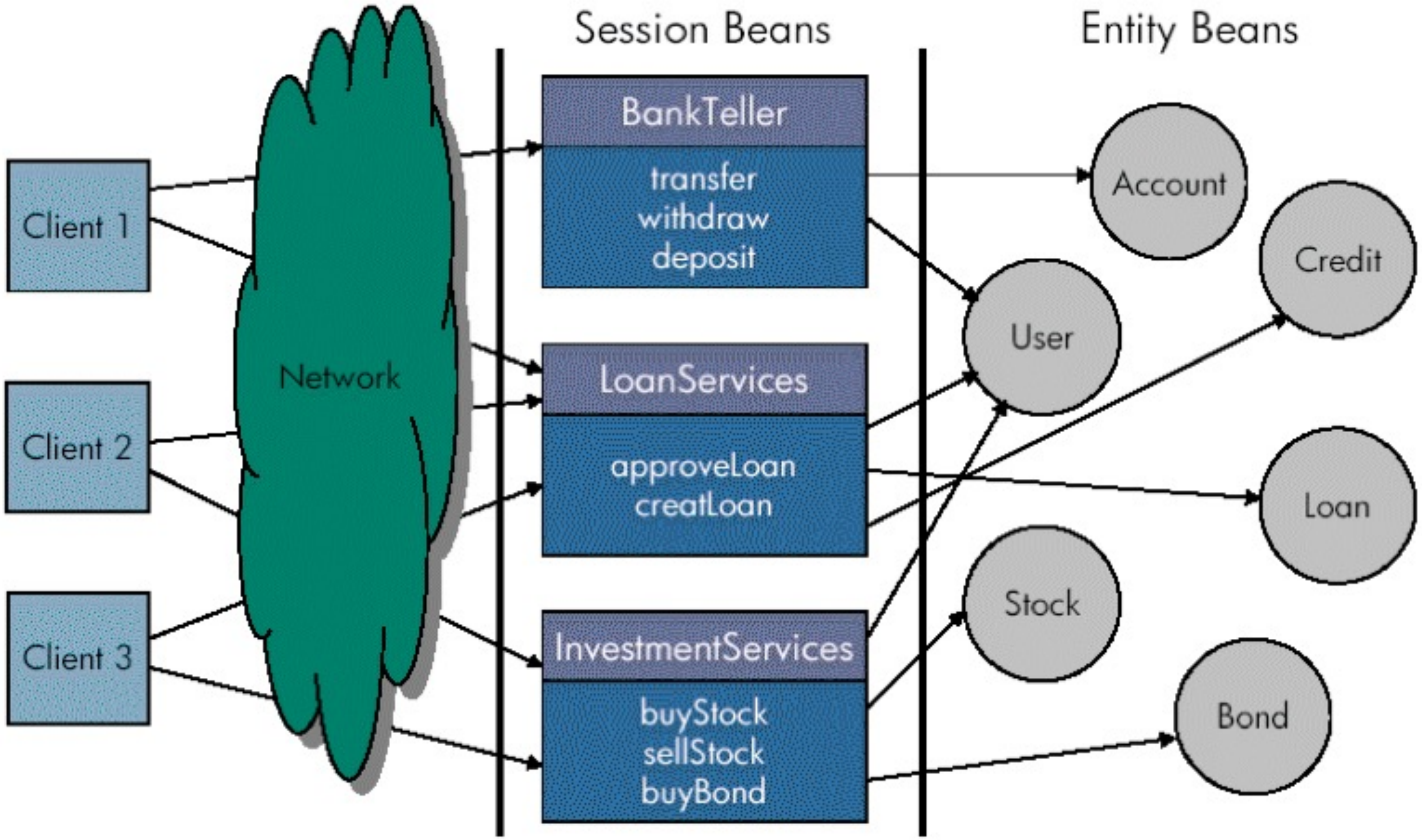
Local beans

`@Local`

Used to specify Local interface(s) of a session bean. This local interface states the business methods of the session bean (which can be stateless or stateful).

This interface is used to expose the business methods to local clients, which are running in the same deployment/application as EJB.

Mapping Session Facade on use cases



Business Delegate Pattern

Use a BusinessDelegate to

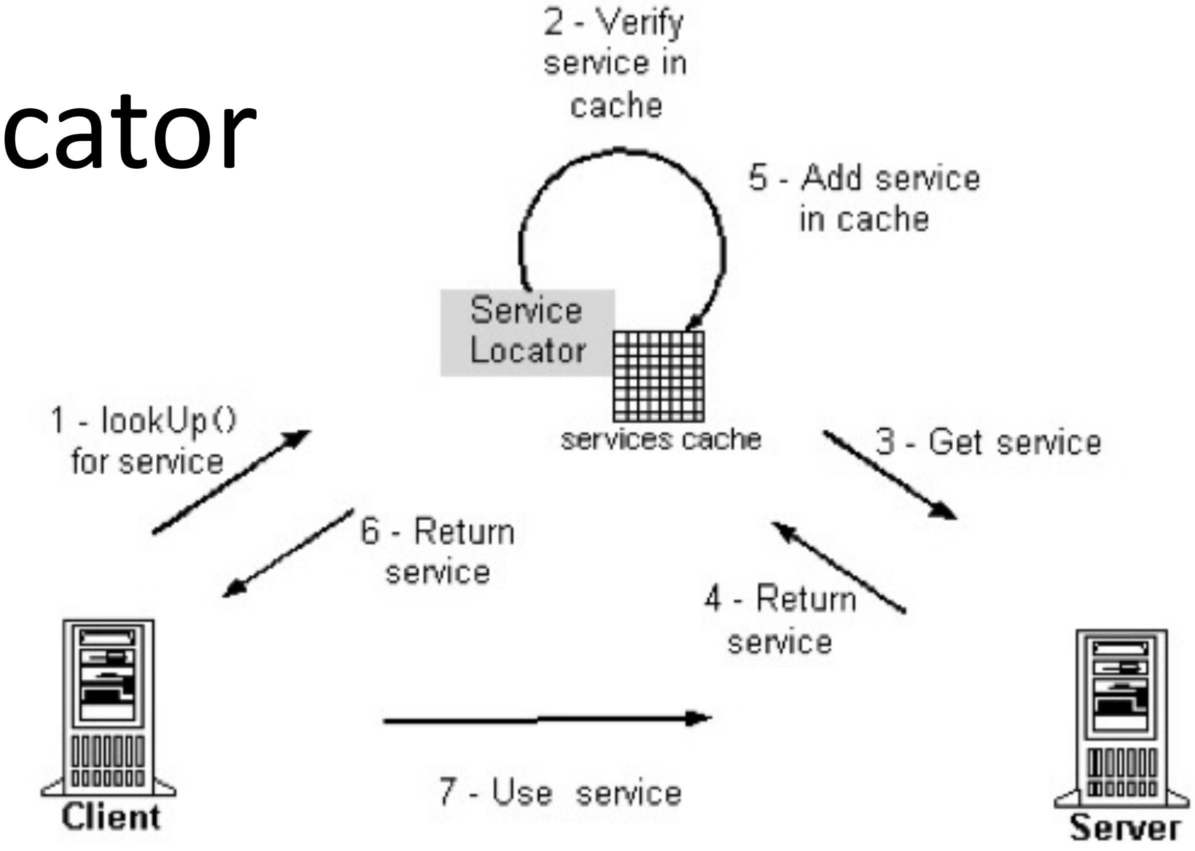
– Reduce coupling between presentation-tier and business service components

– 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

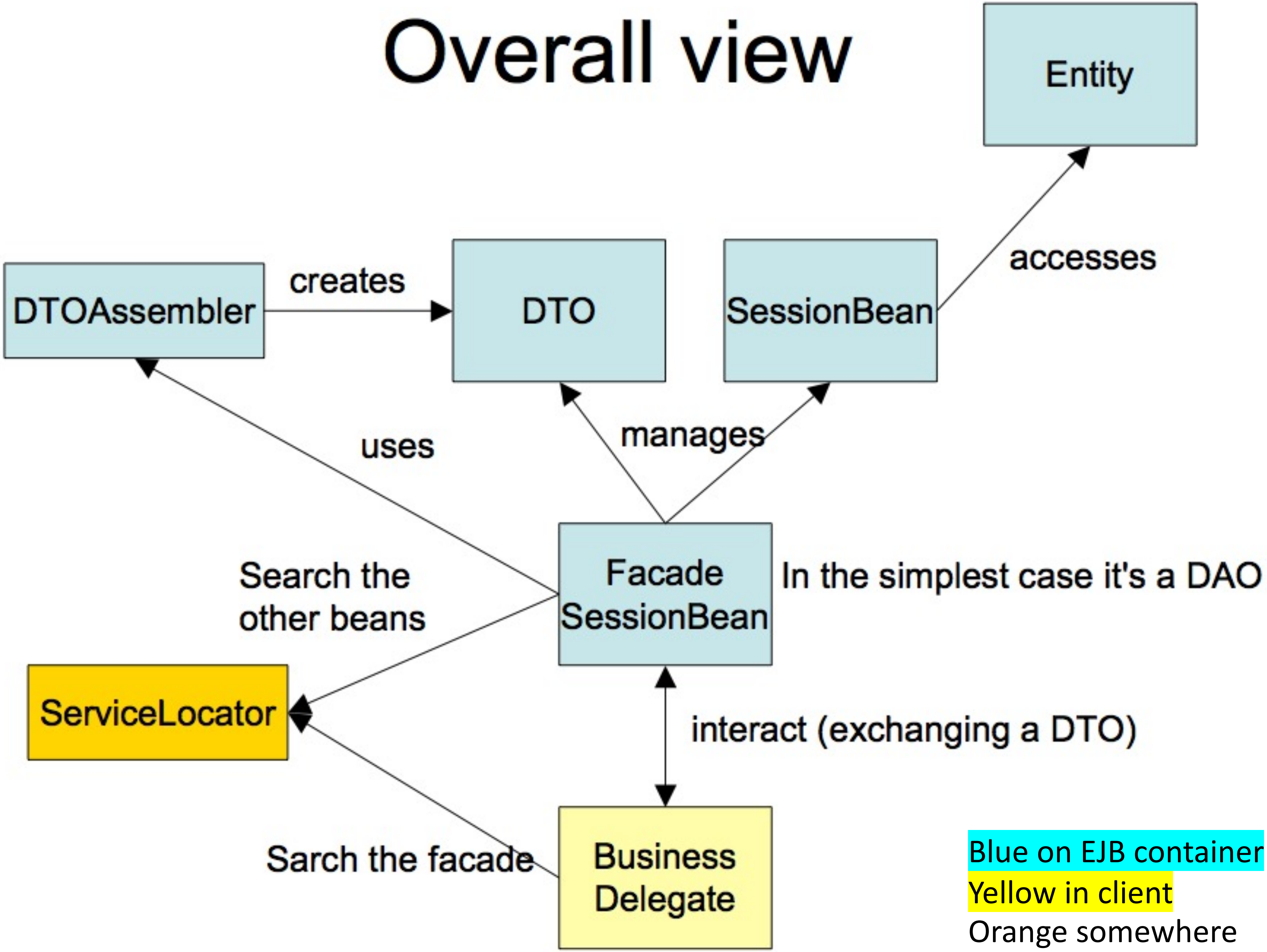
– Enable optimize service lookup and creation functions

- 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



Blue on EJB container

Yellow in client

Orange somewhere