Session: 15

EJB Design Patterns

Objectives

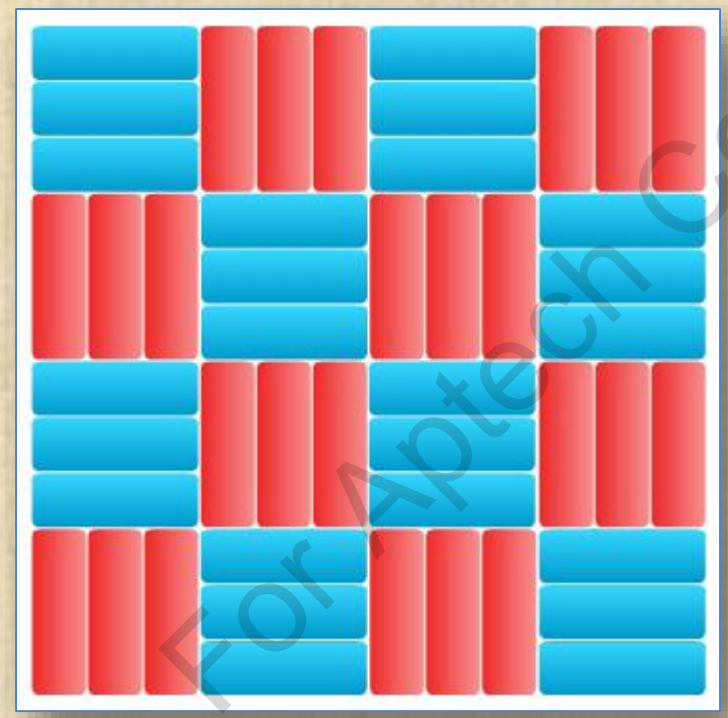
- Describe design patterns
- Explain different types of design patterns
- Explain mostly commonly used Java EE design patterns
- Describe best practices in selecting EJB component in the enterprise applications

Introduction 1-2

- In the world of programming, a previously used solution to a problem can be applied to a new problem by making relevant changes.
- This solution which has been used effectively in an earlier problem is described as a pattern.

Introduction 2-2

- A pattern is a shape, model, or structure appearing again and again to create a design.
- Following figure shows a pattern of tiles arrangement in real world:



- The tiles are printed in such a way that they need to be arranged correctly, so that the print falls in the right place and a pattern is formed.
- Once a pattern is formed out of the combination of first set of tiles, it becomes a pattern or a solution for the rest of the floor.

Design Patterns

- The concept of design patterns was proposed by **Erich Gamma**, **Richard Helm**, **Ralph Johnson**, and **John Vlissides**, popularly known as **Gang of Four**.

- **Design patterns**
  - Are language independent strategies for solving object oriented design problems.
  - Are proven techniques for implementing robust, reusable, and extensible object-oriented software.
  - Helps in describing structures at a higher level of abstraction of classes, components, and instances.

Types of Design Patterns

- There are three types of design patterns:

Creational patterns	• Defines the object creation process in an application.
Structural patterns	• Defines the organization of classes and objects in an application.
Behavioral patterns	• Defines the interaction between various objects in an application.

Creational Patterns 1-2

- Used in scenarios where object composition is crucial to the application.
- Allow developers to configure a system with objects which vary widely in structure and functionality. Following table shows the list of creational patterns:

Pattern	Description
Singleton	Singleton pattern restricts the number of instances of a class to only one. It is used to implement tasks such as application logging and so on.
Factory	Factory pattern is used to create multiple instances of a class. This is used in a scenario where there is a super class with multiple subclasses. Based on the user input an instance of appropriate sub class is created.

Creational Patterns 2-2

Pattern	Description
Abstract Factory	Abstract Factory pattern is similar to factory pattern. It is used to provide an interface for creating instances of sub classes in an inheritance structure without defining concrete classes.
Builder	Builder pattern is used when the instances of the class have large number of optional parameters.
Prototype	Prototype pattern is used when creating an instance of the class is expensive and resource intensive.

Structural Patterns 1-3

- Structural patterns define the structure of multiple classes and objects to function together.
- They define how classes and objects in the application combine to form a large structure.
- These patterns are used to realise the relationships among the classes of the application through object-oriented mechanisms such as abstraction and inheritance.
- Following table shows the list of structural patterns:

Pattern	Description
Adapter	Adapter pattern is used to bridge the structural gap between two interfaces.

Structural Patterns 2-3

Pattern	Description
Composite	Composite patterns are used when composite objects should be treated like primitive objects.
Proxy	Proxy pattern is used when the application wants to provide a controlled access to certain functionality of the application. This pattern allows an object to provide a place holder for another object so that it can access it in a controlled manner.
Flyweight	Flyweight pattern is used when the application has to create large number of objects of a particular class. This is applied when the application runs on low memory devices as it allows sharing of memory.
Façade	Facade pattern is used for defining the interaction of client systems with the application.

Structural Patterns 3-3

Pattern	Description
Bridge	Bridge pattern separates the interfaces from implementations. It is a structural pattern which aims at hiding the implementation details from the clients.
Decorator	Decorator design pattern enables modifying the functionality of an object at runtime. It also isolates other instances of the same class from being modified.

Behavioral Patterns 1-3

- Behavioral patterns determine the interaction of the objects.
- They simplify the complex behavior by specifying the responsibilities of the objects and their communication method.
- Following table shows the list of behavioral patterns:

Pattern	Description
Template Method pattern	Template method pattern is a behavioral design pattern which defers the implementation of behavior of objects to sub classes in the hierarchy.

Behavioral Patterns 2-3

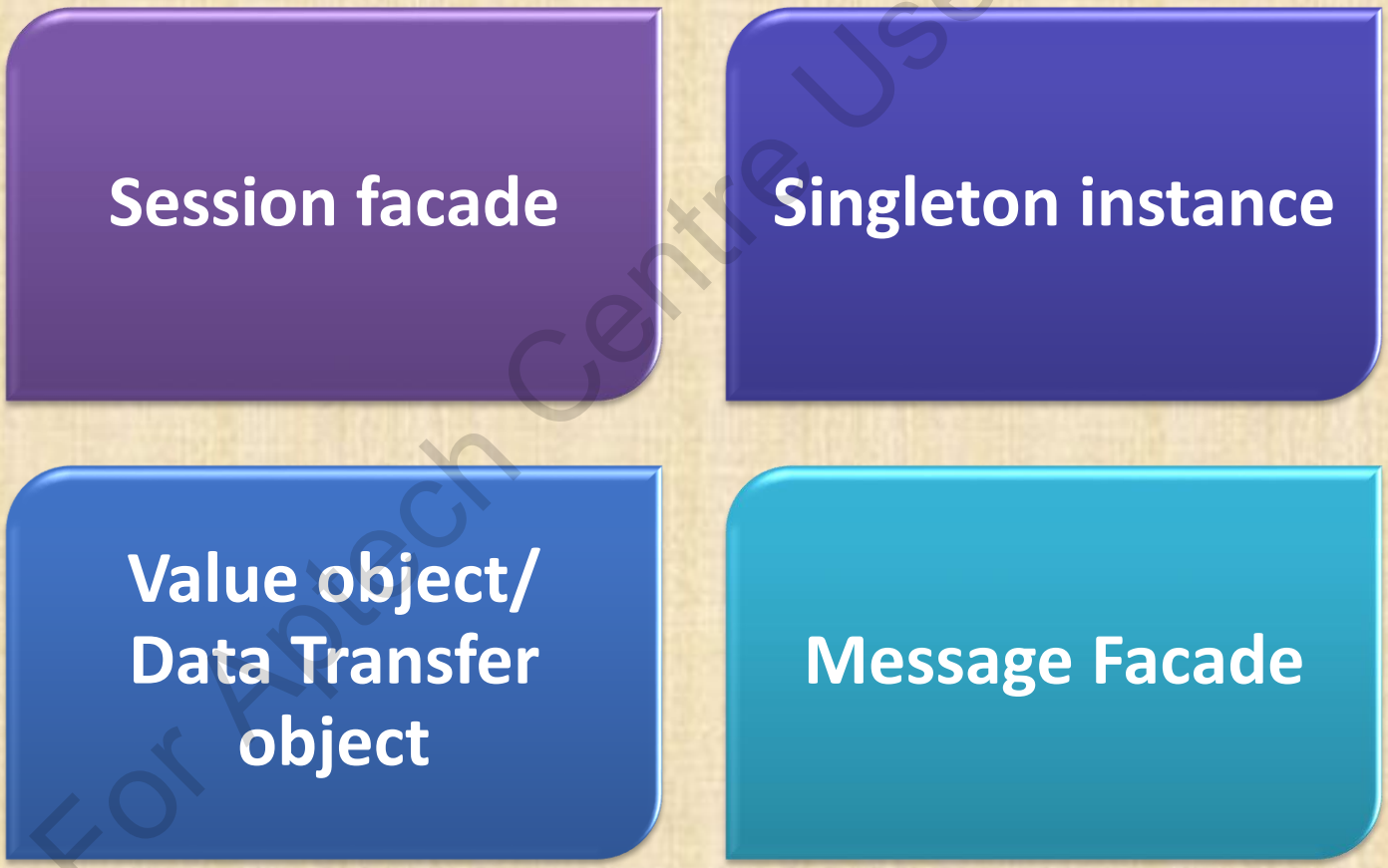
Pattern	Description
Mediator pattern	Mediator pattern is used to establish a communication mechanism among different objects in the application. It implements loose coupling among the objects in the application.
Chain of responsibility pattern	Chain of responsibility pattern defines the method of handling the client request in the application. The client request is passed through a chain of objects while processing it.
Observer pattern	Observer pattern is used to track the state of an object in the application. The object which keeps track of the state is known as an Observer.
Strategy pattern	Strategy pattern enables the application to have multiple algorithms which can be used on certain event. The algorithm to be used is decided at runtime.
Command pattern	Command pattern is a behavioral pattern which implements loose coupling of the components. The application components execute in a request response model.

Behavioral Patterns 3-3

Pattern	Description
State pattern	State pattern is used when the object changes its behavior based on its internal state.
Visitor pattern	Visitor pattern enables separating the operational logic to a separate class. It is used when an operation has to be performed on a group of similar objects.
Interpreter pattern	Interpreter pattern is used when the application has to analyze the syntax and semantics of input.
Iterator pattern	Iterator pattern is used when the application has to provide a standard way to traverse through a group of objects.
Memento pattern	Memento design pattern is used when the application has to save the state of the object and restore it for later use.

Java EE Design Patterns

- Following are some of the useful design patterns in Java EE applications:

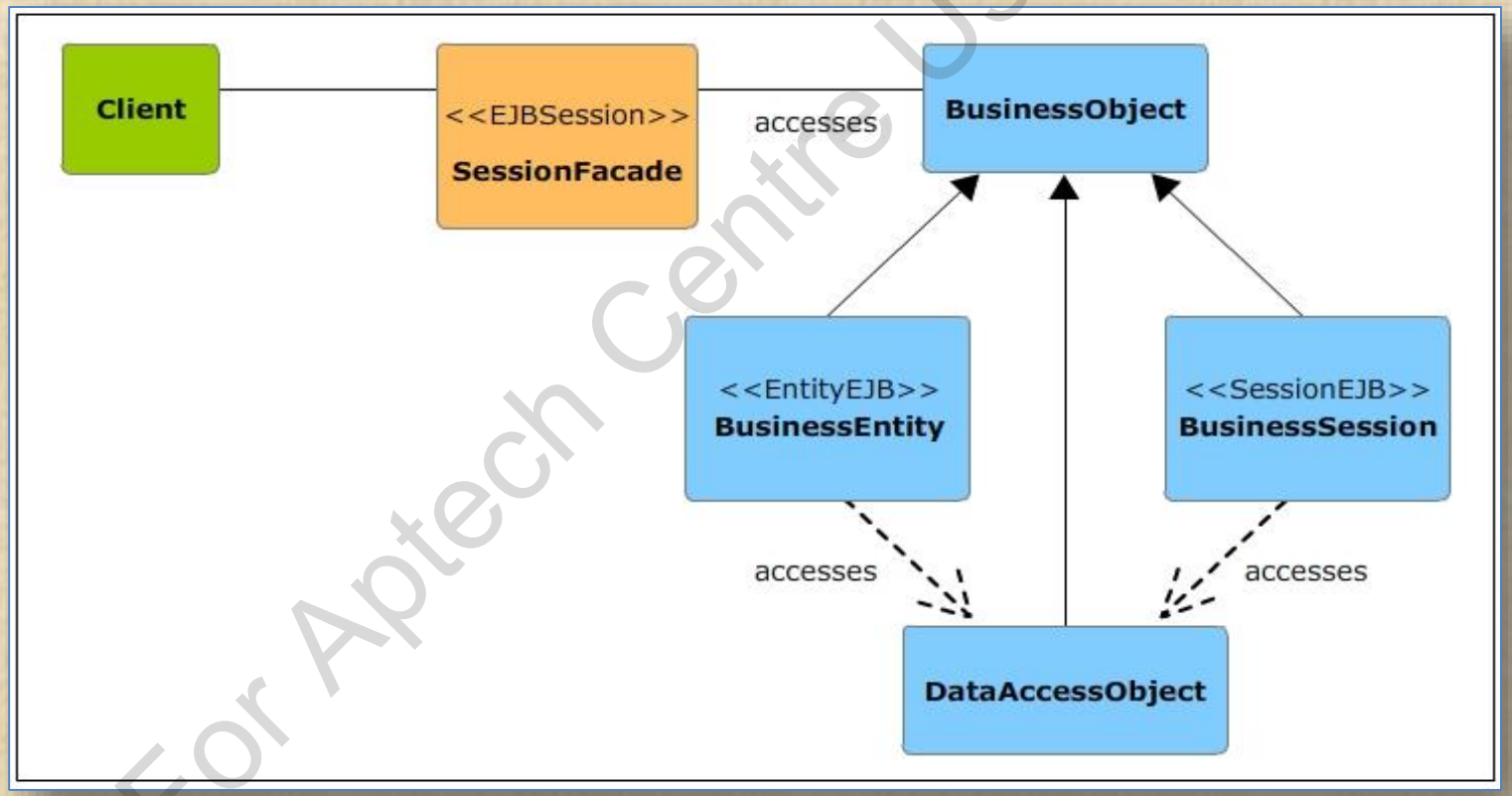


Session Façade 1-13

- Session façade abstracts the interaction of business objects by providing a service layer.
- The service layer exposes the required interfaces to the client.
- The session façade manages the interactions between the business data and business service objects in a workflow.
- Session façade enables decoupling of all the lower-level application components.
- Java EE applications implement session facade through session beans.

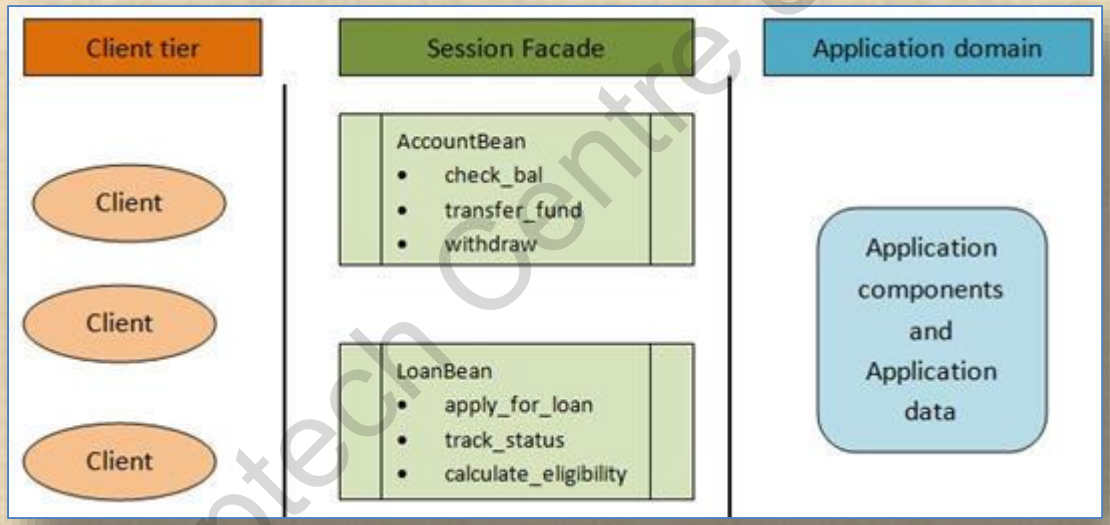
Session Façade 2-13

- Following figure shows the class diagram of a Session façade pattern:



Session Façade 3-13

- Following figure shows an example on implementation of session façade in enterprise applications:



Session Façade 4-13

- Following code snippet shows an entity class **`Message`** which is later accessed through session bean according to session façade pattern:

```
@Entity
public class Message implements Serializable {
private static final long serialVersionUID = 1L;
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
private String message;
// Getter and Setter methods
. . .
@Override
public int hashCode() {
```


Session Façade 5-13

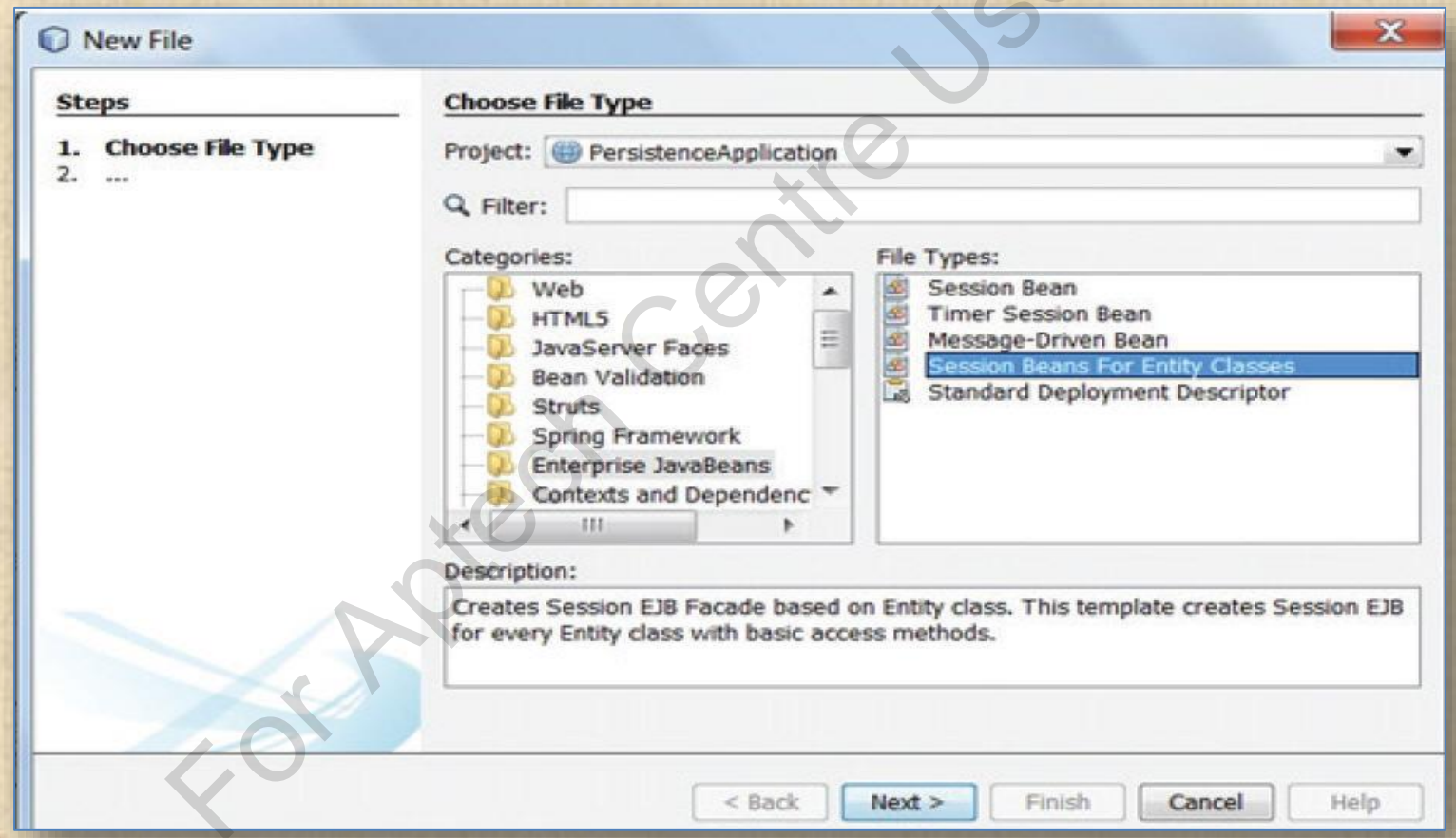
```
int hash = 0;
hash += (id != null ? id.hashCode() : 0);
return hash;
}
@Override
public boolean equals(Object object) {
if (!(object instanceof Message)) {
return false;
}
Message other = (Message) object;
if ((this.id == null && other.id != null) || (this.id !=
null && !this.
id.equals(other.id))) {
return false;
}
return true;
}
```


Session Façade 6-13

```
@Override
public String toString() {
return “entities.Message[ id=” + id + “ ]”;
}
}
```


Session Façade 7-13

- Following figure demonstrates how to create session façade for **`Message`** entity:



Session Façade 8-13

- Select **`entities.Message`** from the Available Entity Classes and click **Add**. The **`Message`** entity will be added to the Selected Entity Classes.
- Create the session bean in a package '**`façade`**', this package name is based on the developer's choice.
- This creates a session bean which will be exposed to a local managed bean using a no-interface view.

Session Façade 9-13

- Following code snippet shows the code for **`AbstractFacade.java:`**

```
. . .
public abstract class AbstractFacade<T> {
private Class<T> entityClass; @GeneratedValue(strategy
= GenerationType.AUTO)
public AbstractFacade(Class<T> entityClass) {
   this.entityClass = entityClass;
}
protected abstract EntityManager getEntityManager();
public void create(T entity) {
    getEntityManager().persist(entity);
}
public void edit(T entity) {
    getEntityManager().merge(entity);
}
```


Session Façade 10-13

```
public void remove(T entity) {
getEntityManager().remove(getEntityManager().merge(entity));
}
public T find(Object id) {
return getEntityManager().find(entityClass, id);
}
public List<T> findAll() {
  javax.persistence.criteria.CriteriaQuery cq =
 getEntityManager().getCriteriaBuilder().createQuery();
cq.select(cq.from(entityClass));
return getEntityManager().createQuery(cq).getResultList();
}
```


Session Façade 11-13

```
public List<T> findRange(int[] range) {
javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().
getCriteriaBuilder().createQuery();
cq.select(cq.from(entityClass));
javax.persistence.Query q =
getEntityManager().createQuery(cq);
q.setMaxResults(range[1] - range[0] + 1);
q.setFirstResult(range[0]);
return q.getResultList();
}

public int count() {
javax.persistence.criteria.CriteriaQuery cq =
getEntityManager().
getCriteriaBuilder().createQuery();
```


Session Façade 12-13

```
javax.persistence.criteria.Root<T> rt =
cq.from(entityClass);
cq.select(getEntityManager().getCriteriaBuilder().cou
nt(rt));
javax.persistence.Query q =
getEntityManager().createQuery(cq);
return ((Long) q.getSingleResult()).intValue();
}
}
```


Session Façade 13-13

- Following code snippet shows the **`MessageFaçade`** class:

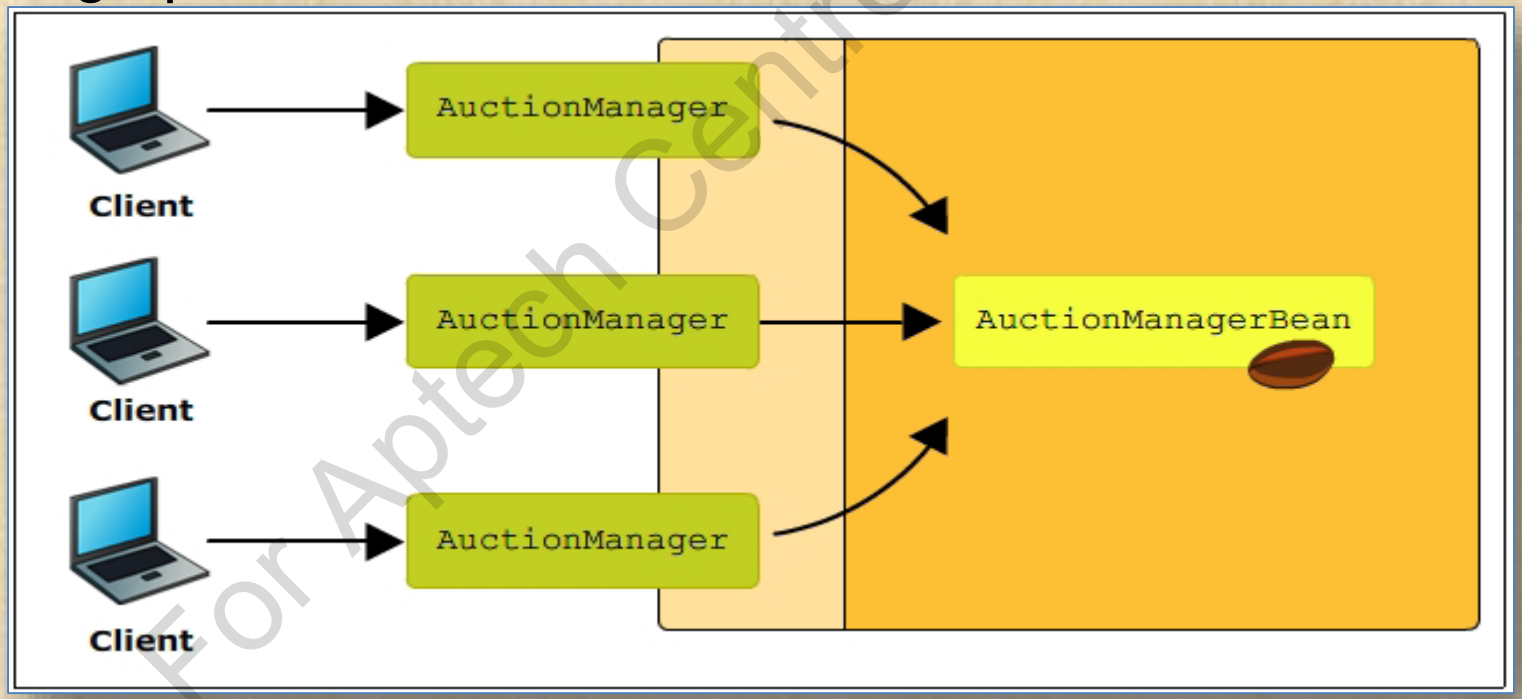
```
. . .
@Stateless
public class MessageFacade extends AbstractFacade<Message> {
@PersistenceContext(unitName = "PersistenceApplicationPU")
private EntityManager em;

@Override
protected EntityManager getEntityManager() {
re t u r n em;
}
 public MessageFacade() {
      super(Message.class);
 }
}
```

- The code injects the resource of the Persistence unit through an annotation and then invokes an entity manager for this context.

Singleton Design Pattern 1-3

- Singleton design pattern ensures that there is only one instance of a class in the application.
- Following figure shows the implementation of a Singleton design pattern:



Singleton Design Pattern 2-3

Singleton pattern can be implemented in Java applications in one of the following ways:

- Using constructor in the Java class it can be declared **`private`**
- It can also be implemented as a **`static`** instance of a class

Singleton instance can be created according to one of the following strategies:

- Eager initialization
- Static block initialization
- Lazy initialization
- Thread safe singleton

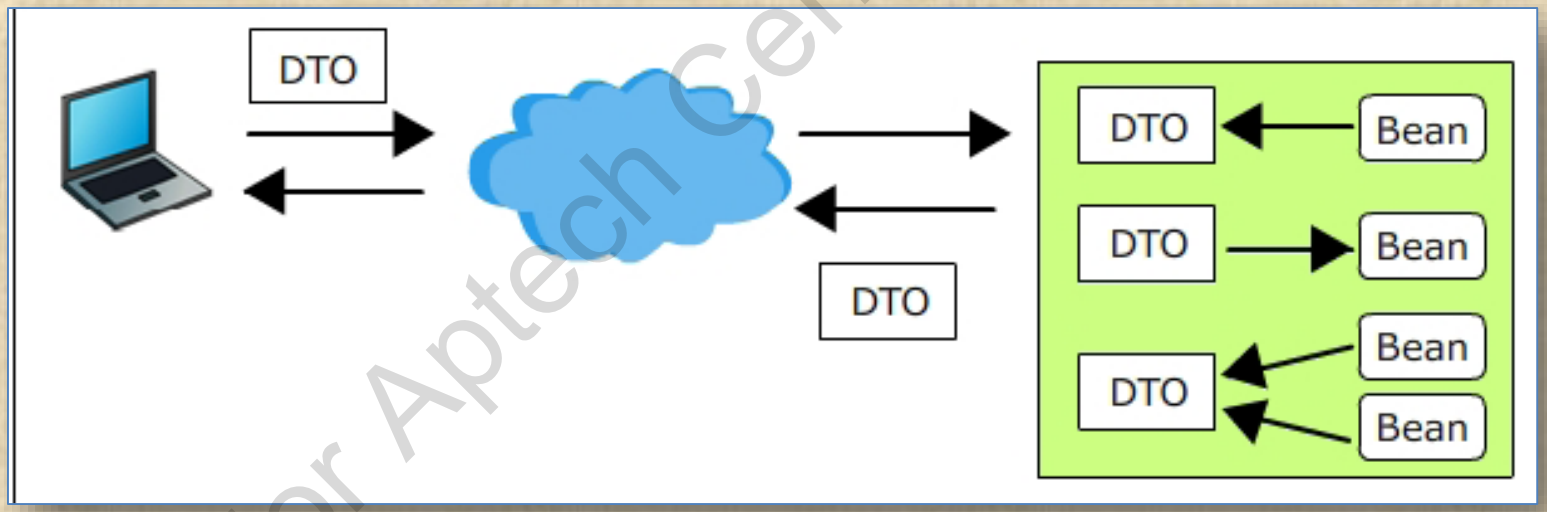
Singleton Design Pattern 3-3

- Following code snippet shows the instantiation of an object according to singleton pattern:

```
public class SingleInstance {
// Declare an instance of the class
private static SingleInstance inst = null;
// set constructor as private
private SingleInstance() {}
// Define a synchronized method for creating the instance
public static synchronized SingleInstance getInstance() {
// check for existence of instance
if (inst == null) {
// create instance if it does not exist
inst = new SingleInstance ();
}
   return inst; // return the instance
}
}
```

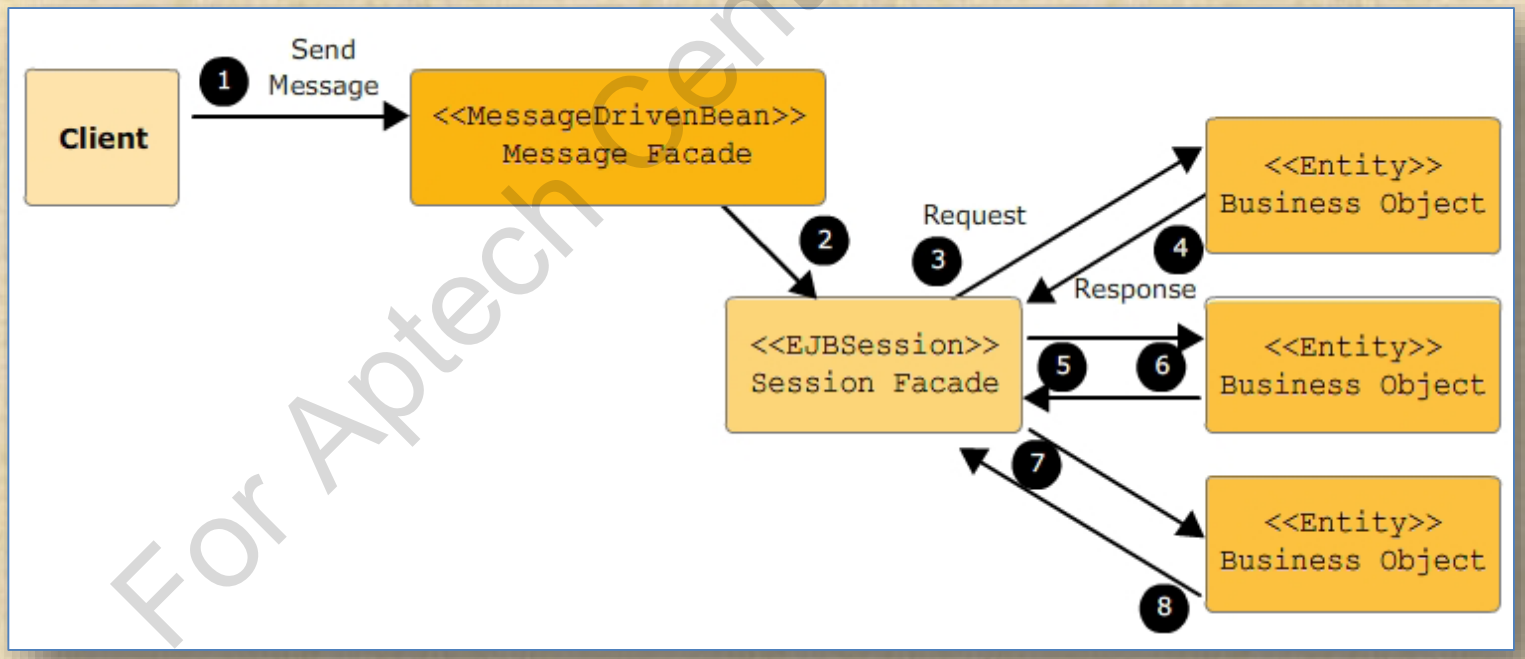
Value Object/Transfer Object Pattern

- The value object pattern is used while implementing communication among different entities in the application.
- Data Transfer Objects are defined to minimize the number of method calls in a distributed system.
- Following figure shows the data transfer object pattern:



Message Façade Pattern

- Message Façade pattern proposes asynchronous handling of method calls.
- Introduces an additional component such as Java Messaging Service (JMS).
- Following figure shows message façade pattern:



EJB Best Practices 1-3

- Enterprise applications developed with EJBs should have the following characteristics:
  - Remote access to the application components
  - Distributed transactions in the application domain
  - Security implementation to different components
  - Data persistence
  - Application scalability

EJB Best Practices 2-3

- Every EJB application should have at least four files - home interface, remote interface, EJB implementation, and deployment descriptor.
- It is a good practice to write the application using JSPs and Servlets. These Web components can then be refactored and expanded to include the EJB components in the application.
- Design patterns must be used whenever possible in the applications, as they provide a reliable, tried, and tested solution for the problem.
- The complete design of the application should be available before actually implementing the application.
- Container Managed Persistence should be used whenever possible in the application as the J2EE container is capable of efficiently managing the application persistence.

EJB Best Practices 3-3

- An interface must be defined through which an EJB can be accessed. Even if the EJB is accessed locally, a local interface must be defined to access the EJB.
- Appropriate exception handling code must be written for all the exceptions that might occur in the EJB code.
- Lazy loading of database tables should be used for optimized performance of the application.

Summary

- Design patterns in software are strategies which are language-independent and used for solving commonly encountered object-oriented design problems.
- The concept of design patterns was proposed by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. They are popularly known as the ‘Gang-of-Four’.
- There are three primary categories of design patterns defined by GOF. They are namely, creational, structural, and behavioral patterns.
- Creational patterns enable creating objects, deciding the properties, and defining the methods of the objects based on the situation.
- Structural patterns are meant for enabling multiple classes and objects to function together. Behavioral patterns are those that determine the interaction of objects.
- There are number of patterns identified by Sun Java center for simplifying the development of enterprise applications.
- Some of the Java EE design patterns are namely, session facade, Singleton Instance, Value Object/Data Transfer Object, and Message Facade.
- The session façade manages the relationships between various business objects and provides a higher-level abstraction to the client.
- A Singleton design pattern ensures that there is only one instance of a class in the application.
- The value object pattern is used while implementing communication among different entities in the application.
- The Message Façade pattern proposes asynchronous handling of method calls.