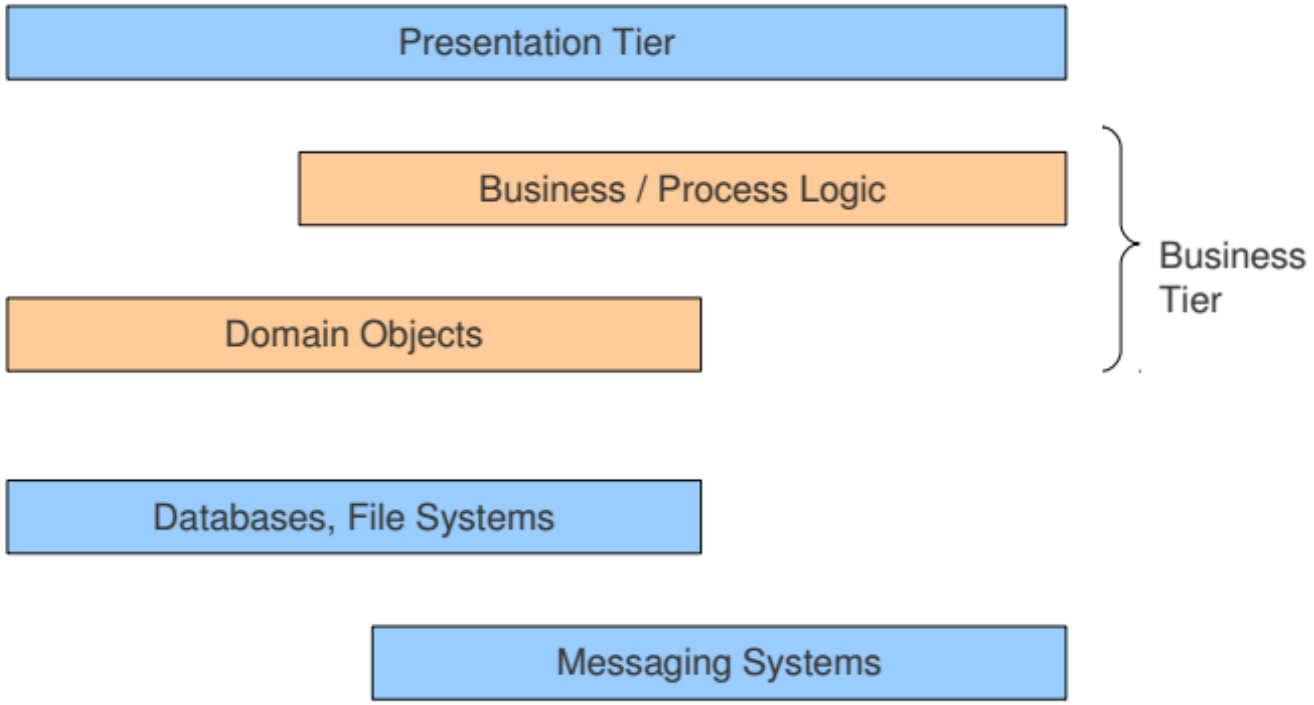
Business Layer Patterns

Lecture 04

by Udara Samaratunge

Three Tiers

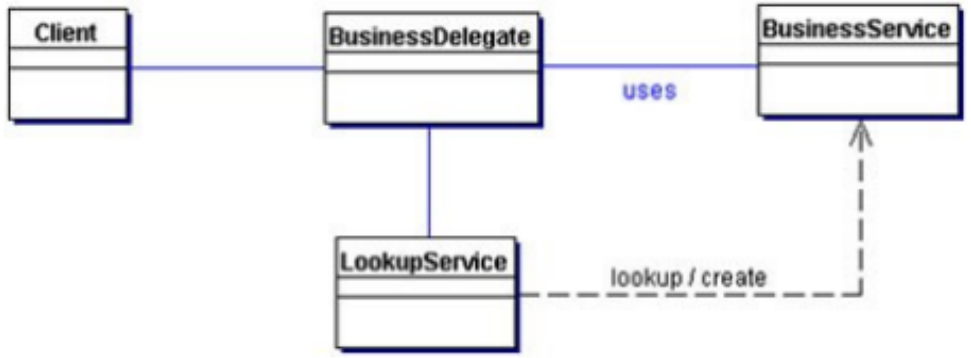
Business Tier



Domain / Business Layer Design Patterns

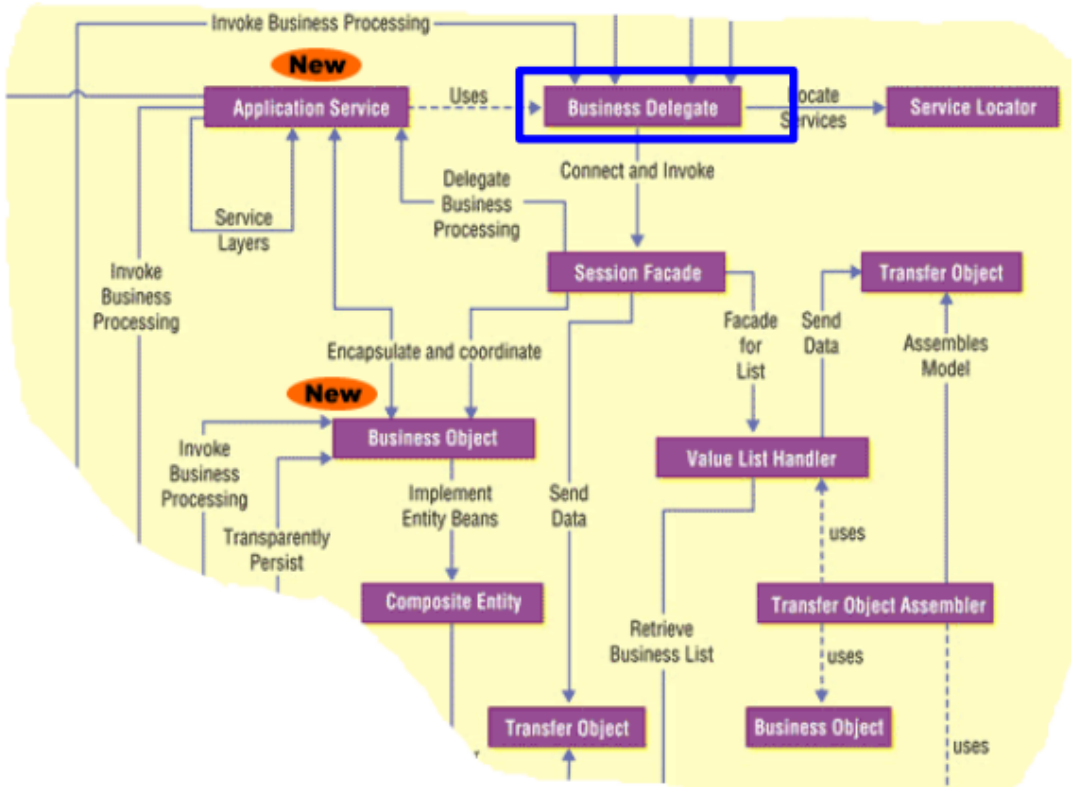
- Business Delegate
- Service Locater
- Session Facade
- Transfer Object
- Value List Handler
- Composite Entity
- Transfer Object Assembler

Business Delegate Pattern



The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of business services

Business Delegate Pattern

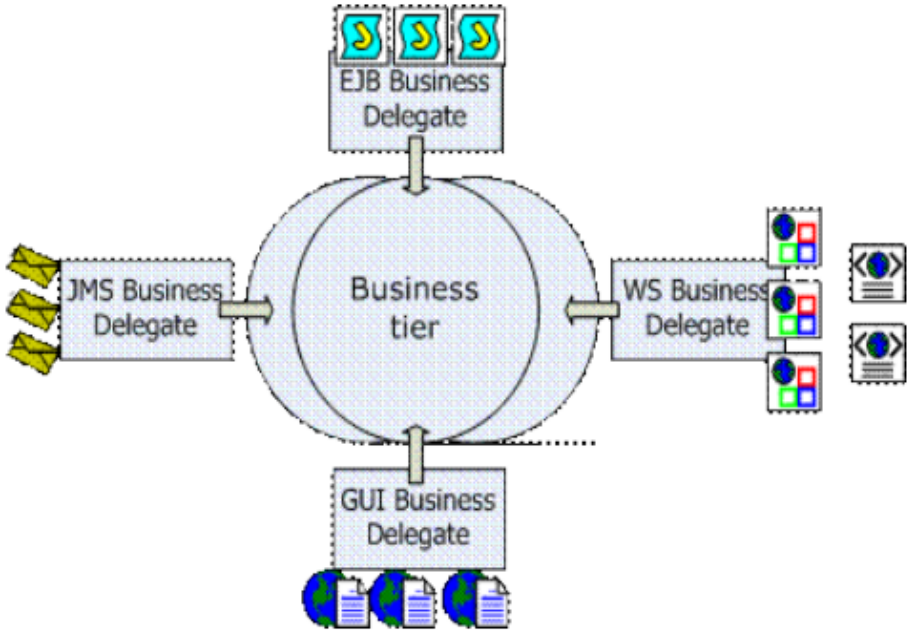


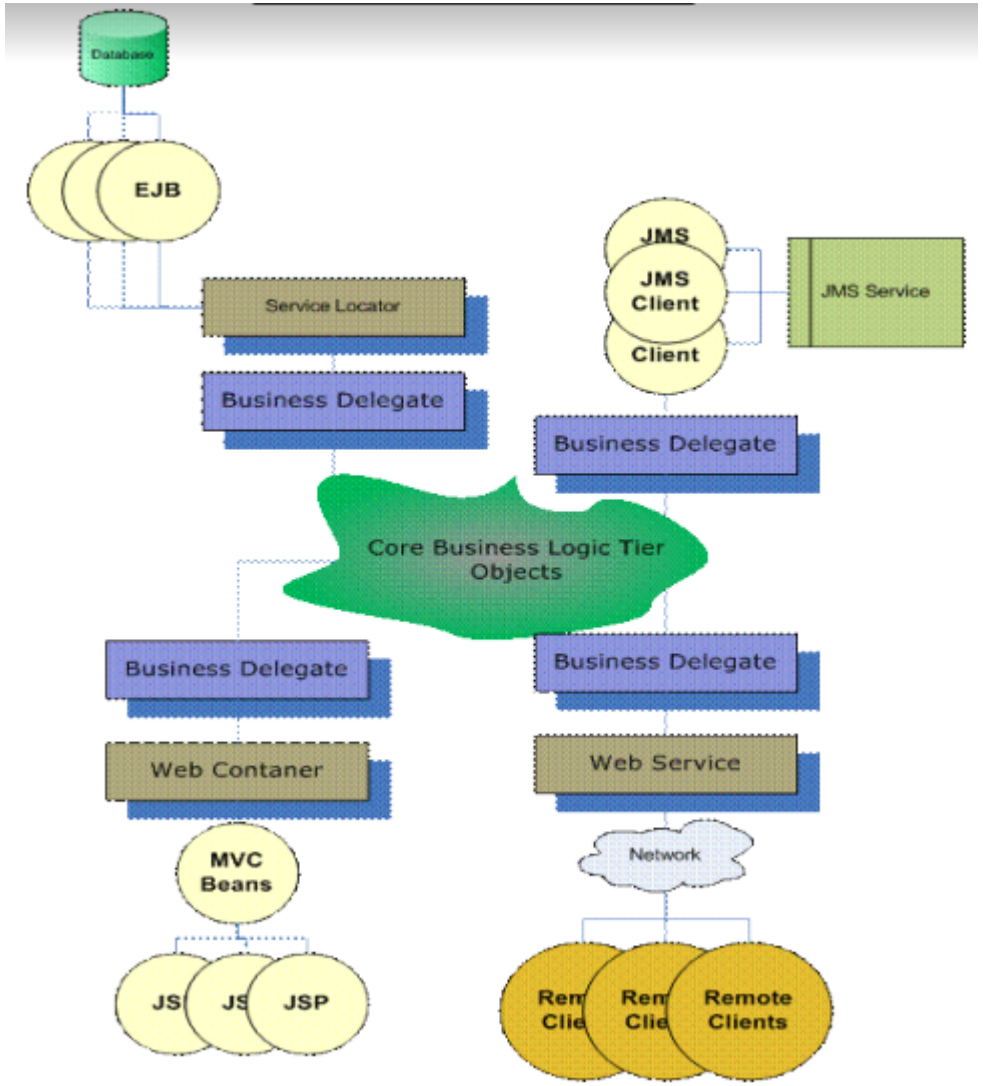
Business Delegate Pattern

- There can be a lot of invocations to Business Service APIs from the presentation layer. That can create a heavy network traffic

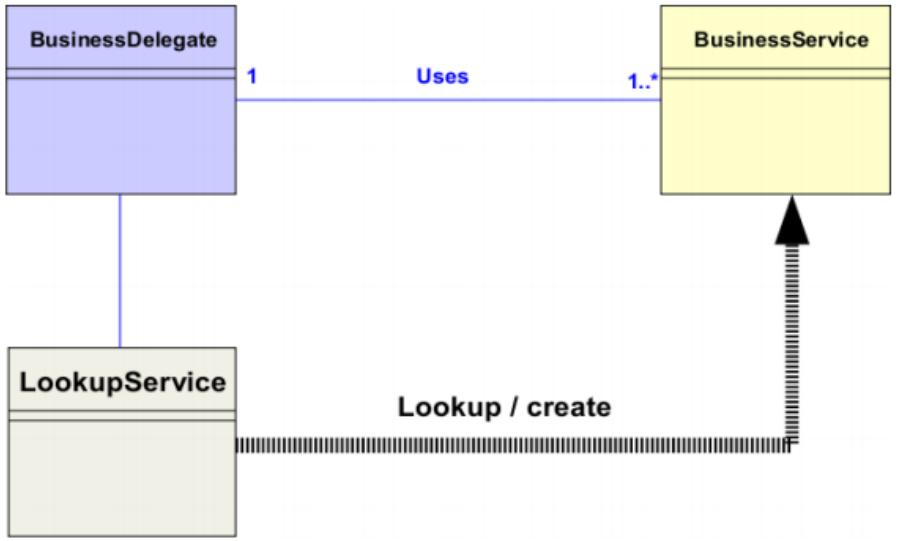
- **Without this pattern..**
  - Presentation-tier components interact directly with business services
  - This direct interaction exposes the underlying implementation details of the business service application program interface (API) to the presentation tier
  - As a result, the presentation-tier components are vulnerable to changes in the implementation of the business services (If the business services changed, the presentation tier must be changed too)

Business Delegate Pattern

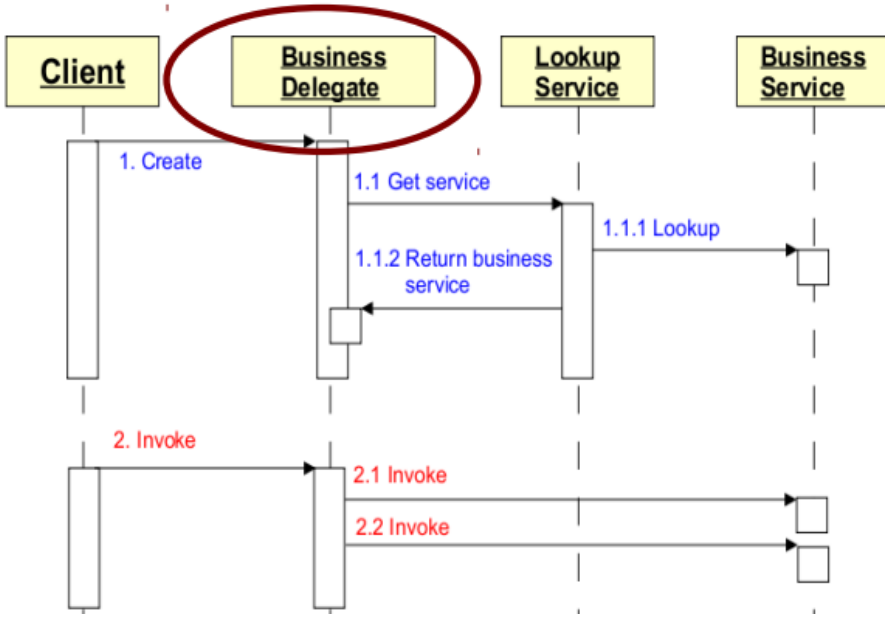




Business Delegate Pattern



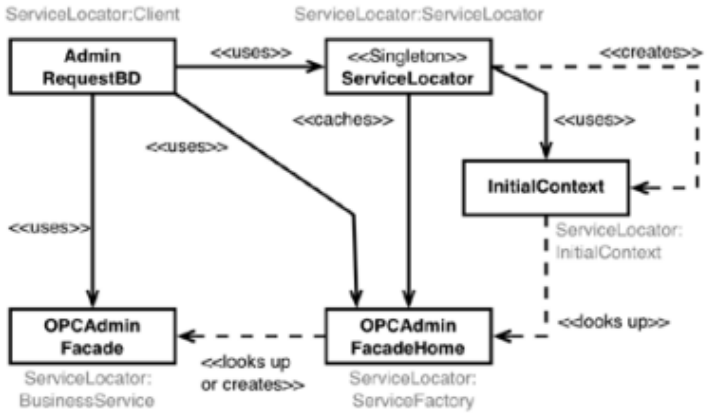
Business Delegate Pattern



Business Delegate Pattern

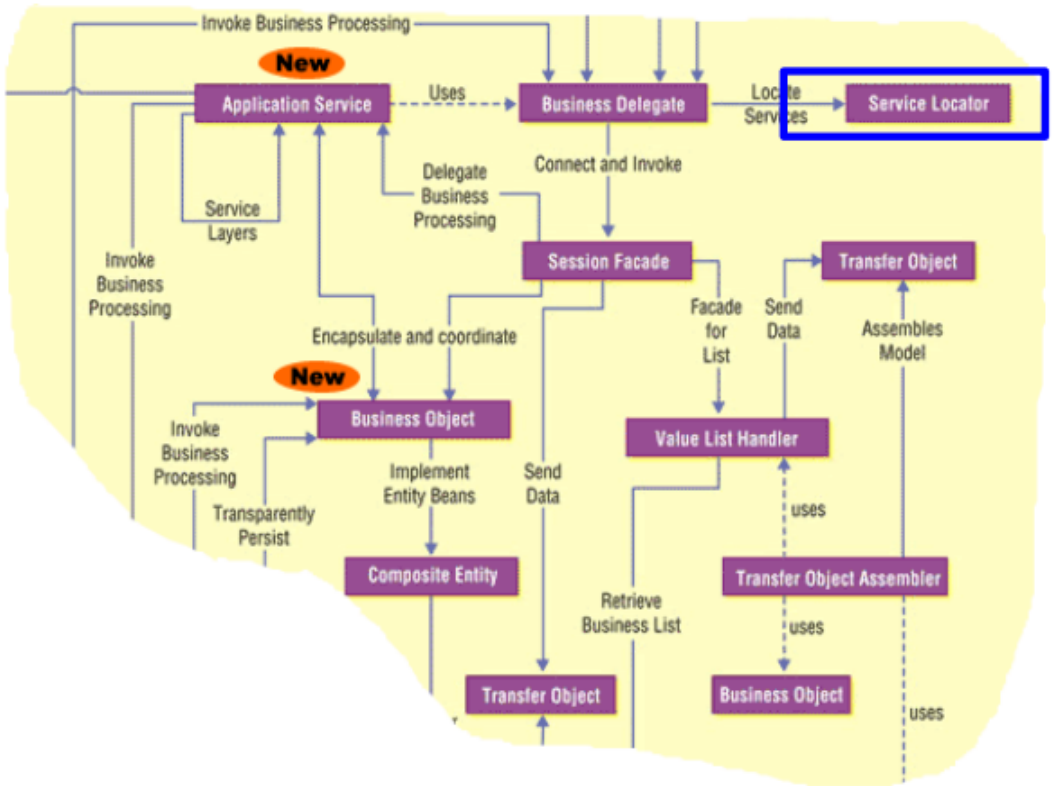
- The **Business Delegate** uses a **Lookup Service** for locating the business service
- The **Business Service** is used to invoke the business methods on behalf of the client
- The role of this pattern is to provide control and protection for the business service

Service Locator Pattern



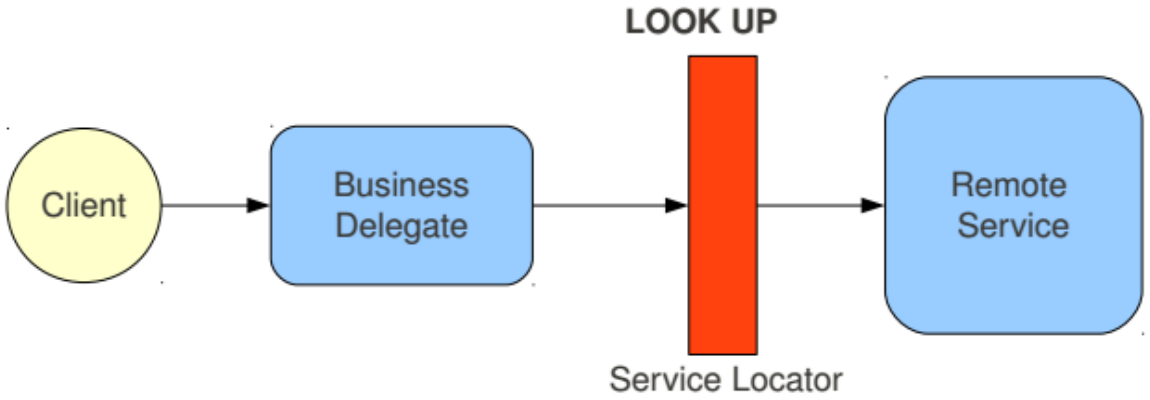
Use a Service Locator to implement and encapsulate service and component lookup. A Service Locator hides the implementation details of the lookup mechanism and encapsulates related dependencies

Service Locator Pattern



Service Locator Pattern

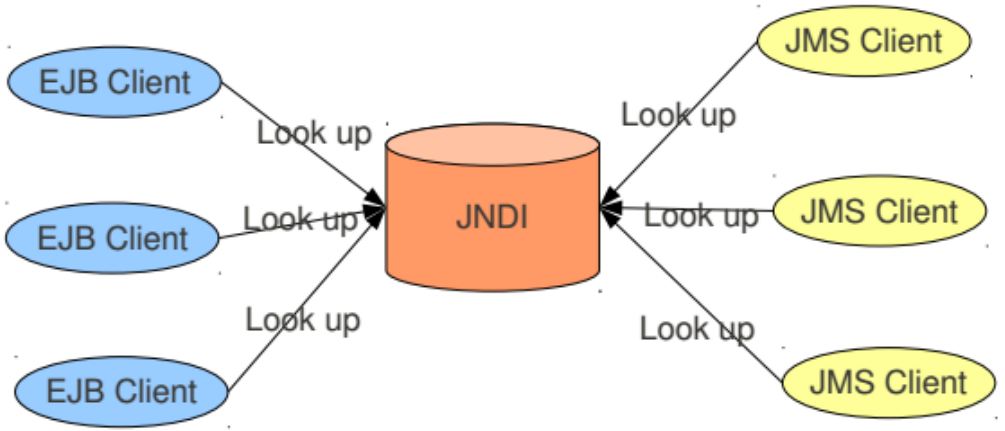
- Enterprise applications require a way to look up the service objects that provide access to distributed components



Service Locator Pattern

Problem

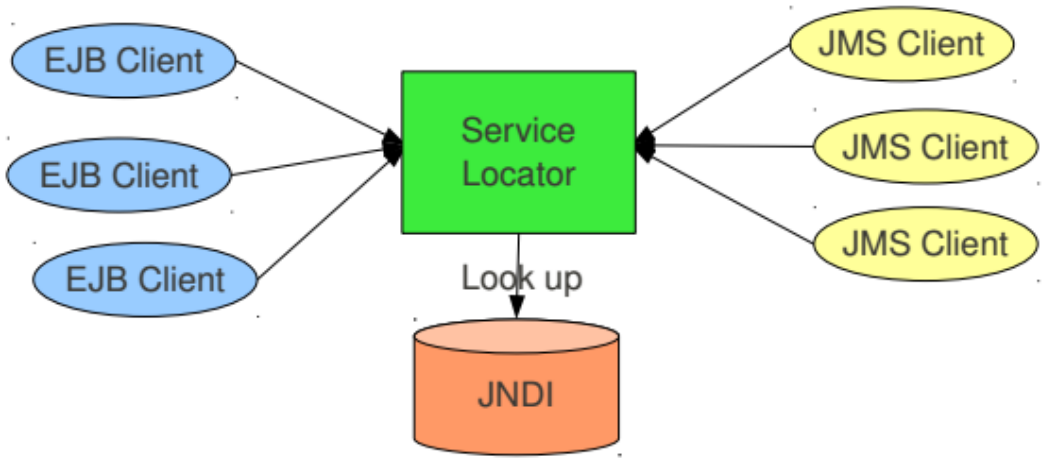
- Without the Service Locator, (In J2EE)
  - J2EE specification mandates the usage of JNDI (Java Naming and Directory Interface) to access different resources/services



Service Locator Pattern

Solution

- Using the Service Locator, (In J2EE)
  - The solution is to cache those service objects when the client performs JNDI lookup first time and reuse that service object from the cache second time onwards for other clients.

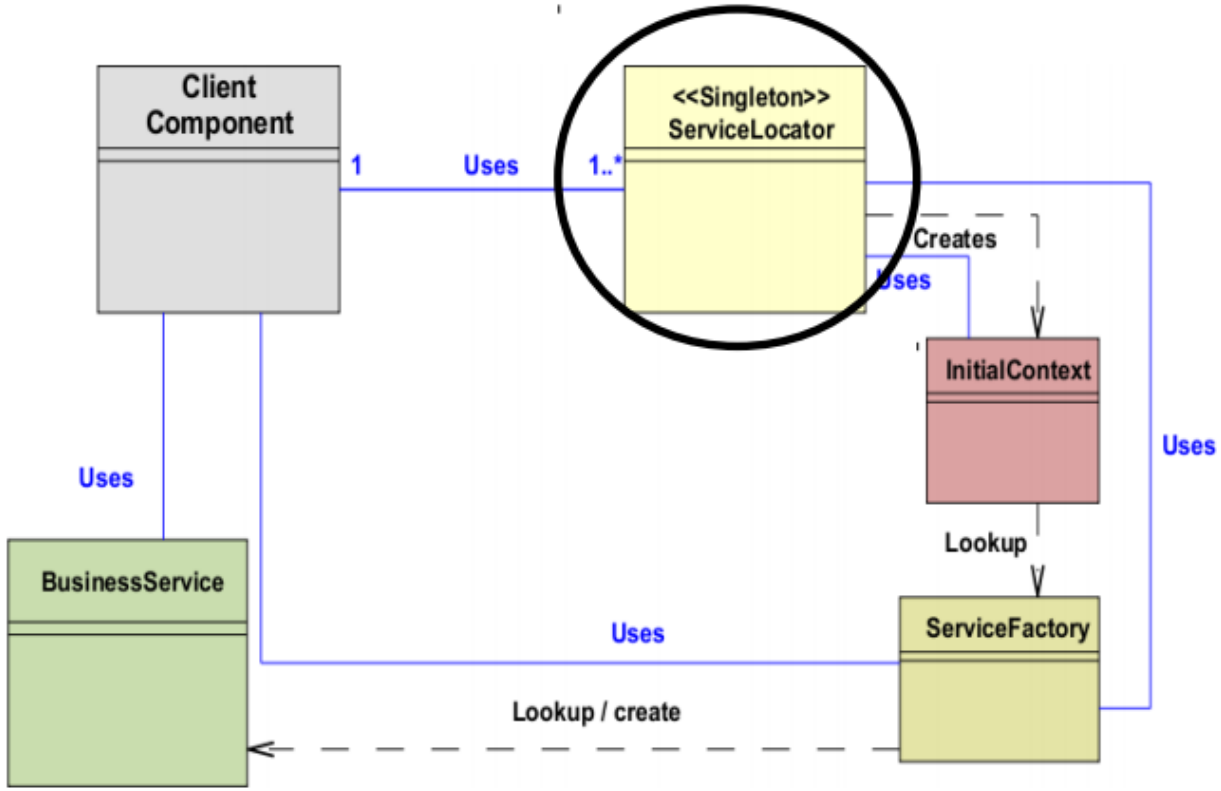


Service Locator Pattern

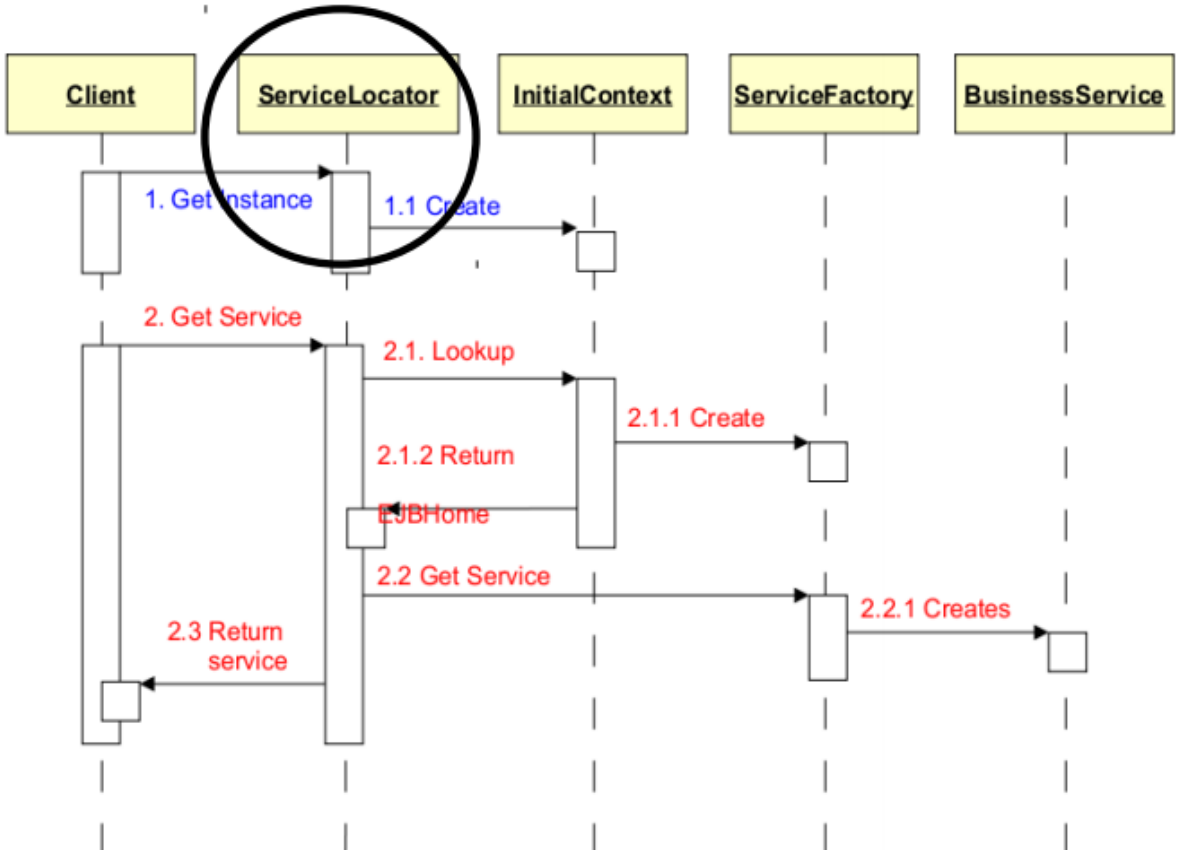
Solution

- Using the Service Locator, (In J2EE) cont..
  - Here the ServiceLocator class intercepting the client request and accessing JNDI once and only once for a service object.
  - Here the clients call ServiceLocator class to get a service object rather than calling JNDI directly. ServiceLocator acts as interceptor between client and JNDI.

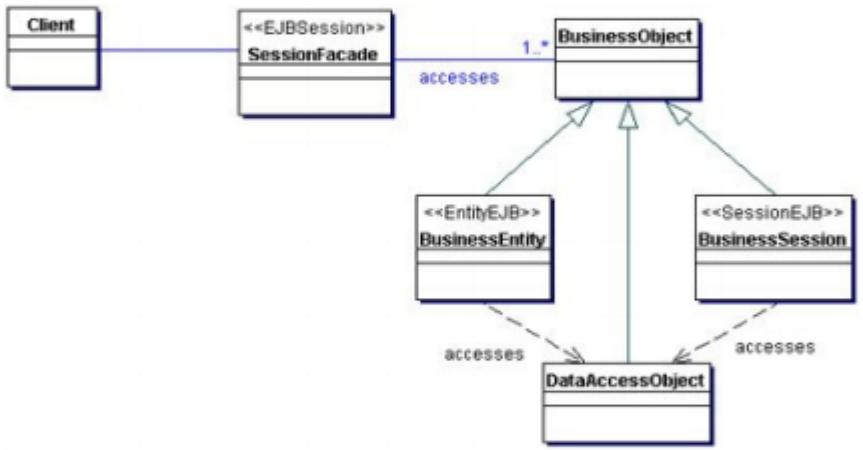
Service Locator Pattern



Service Locator Pattern

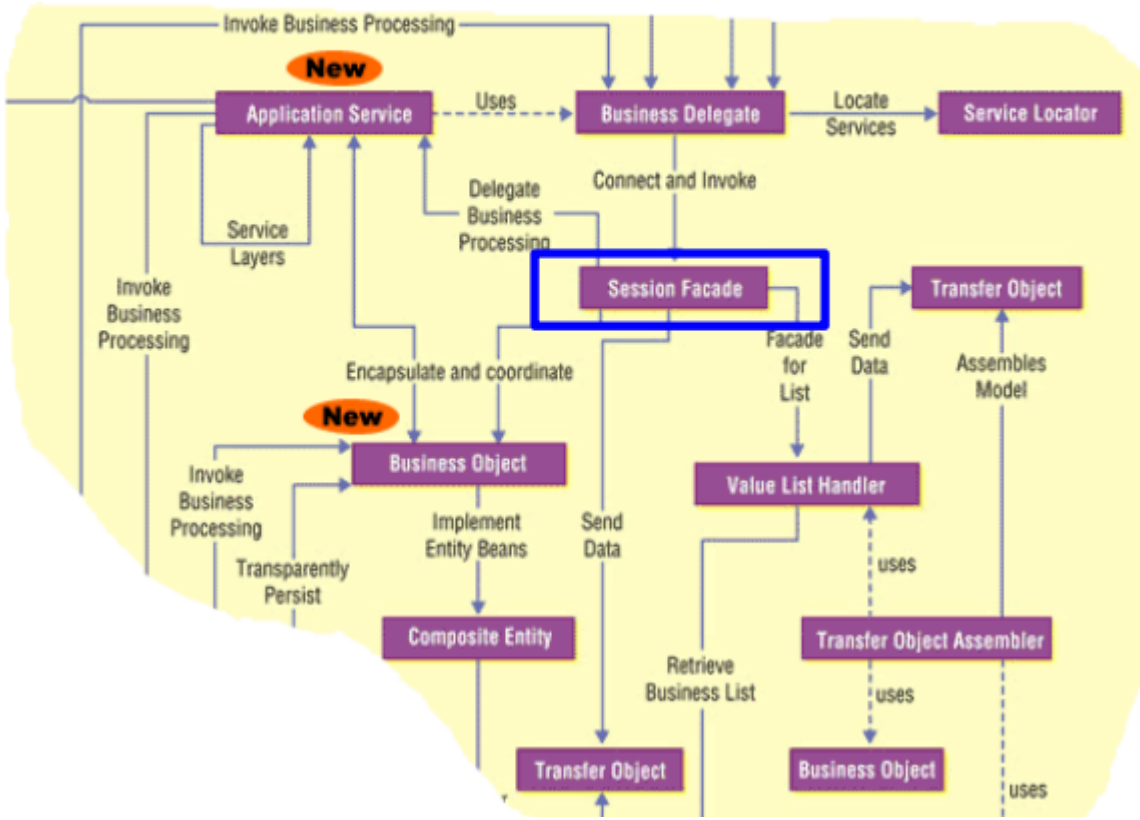


Session Facade Pattern

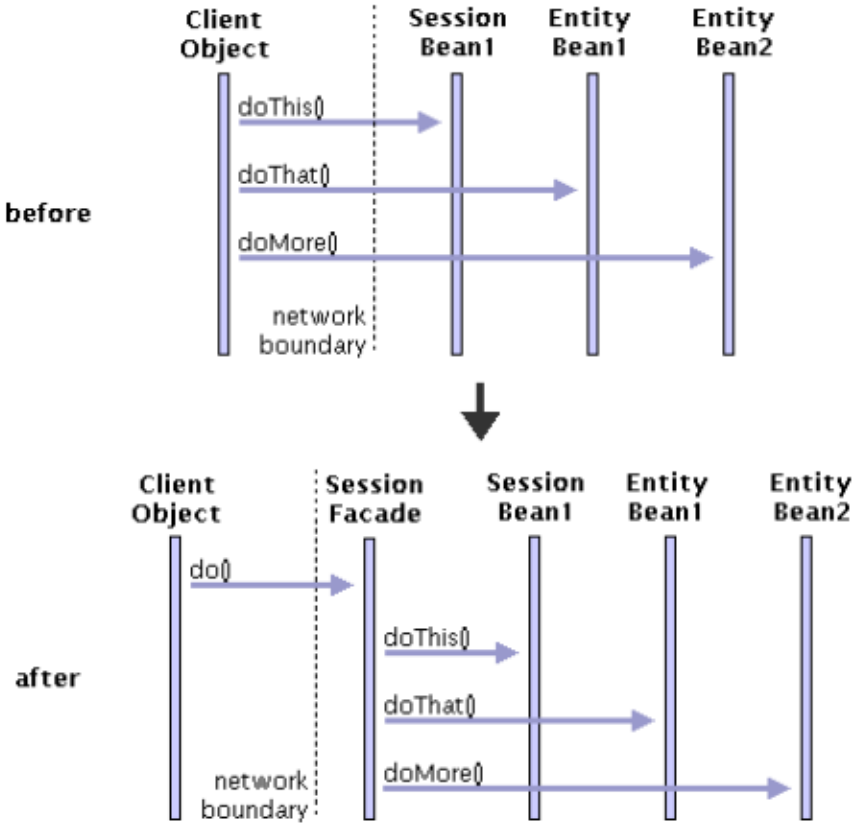


The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients

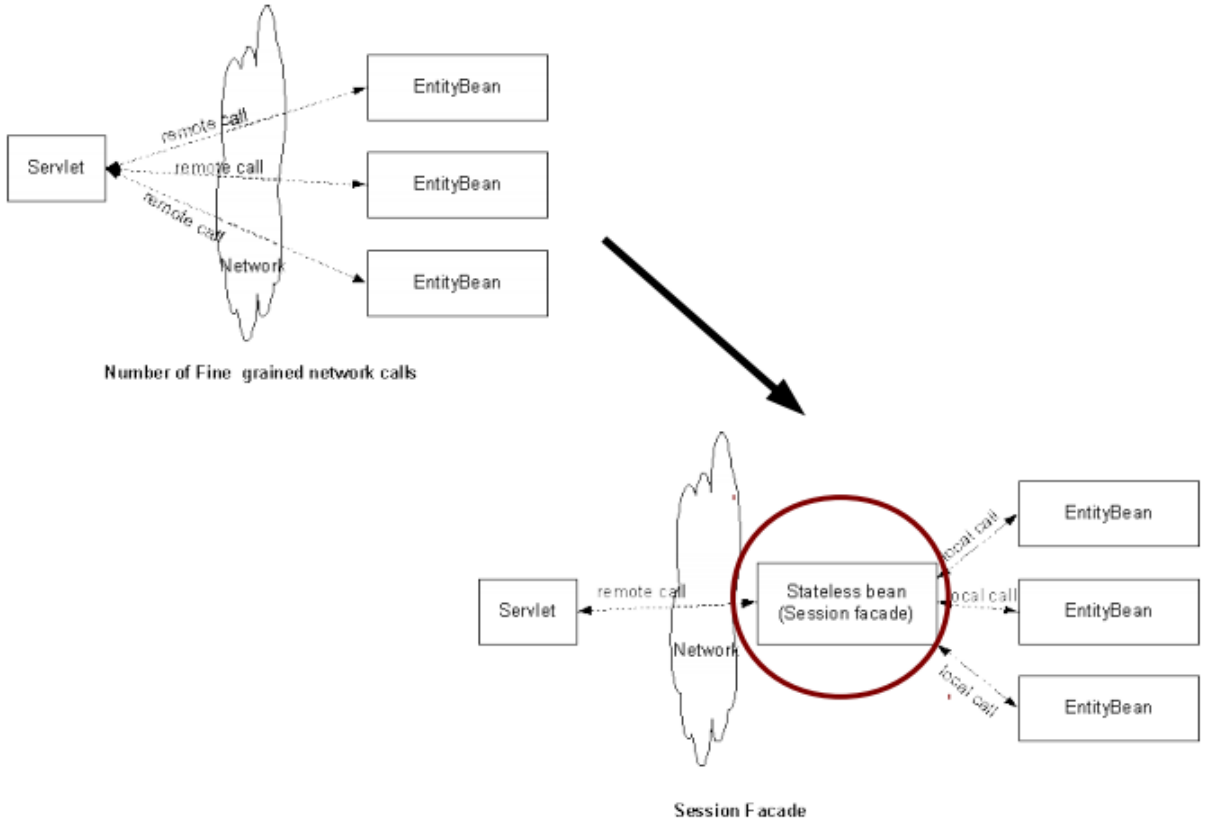
Session Facade Pattern



Session Facade Pattern



Session Facade Pattern



Session Facade Pattern

- It is implemented as a higher level component (i.e.: Session EJB), and it contains all the interactions between low level components (i.e.: Entity EJB).
- Provides a single interface for the functionality of an application or part of it
- It decouples lower level components simplifying the design

Session Facade Pattern

- **Benefits**
  - Acts as a business-tier controller layer
  - Exposes a uniform interface
    - *Reduces the complexity of underlying business components*
  - Reduces coupling, introduce manageability
  - Improves performance, reduces fine-grained methods
  - Provides coarse grained access
    - *Analyze the interaction between the client and the application services, using use cases and scenarios to determine the coarseness of the facade*

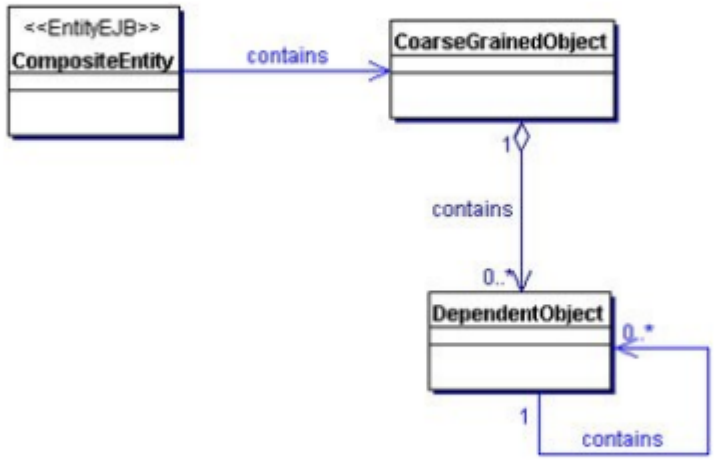
Domain Object Model

- The Domain Object Model defines an object-oriented representation of the underlying data of an application.
- It is a portion of the business tier that defines the actual data and processes managed and implemented by the system

- **Purpose:**
  - The business tier becomes more simpler and are decoupled from the implementation of the business tier.
  - Allows changes to the resource layer level implementation without affecting the UI components.

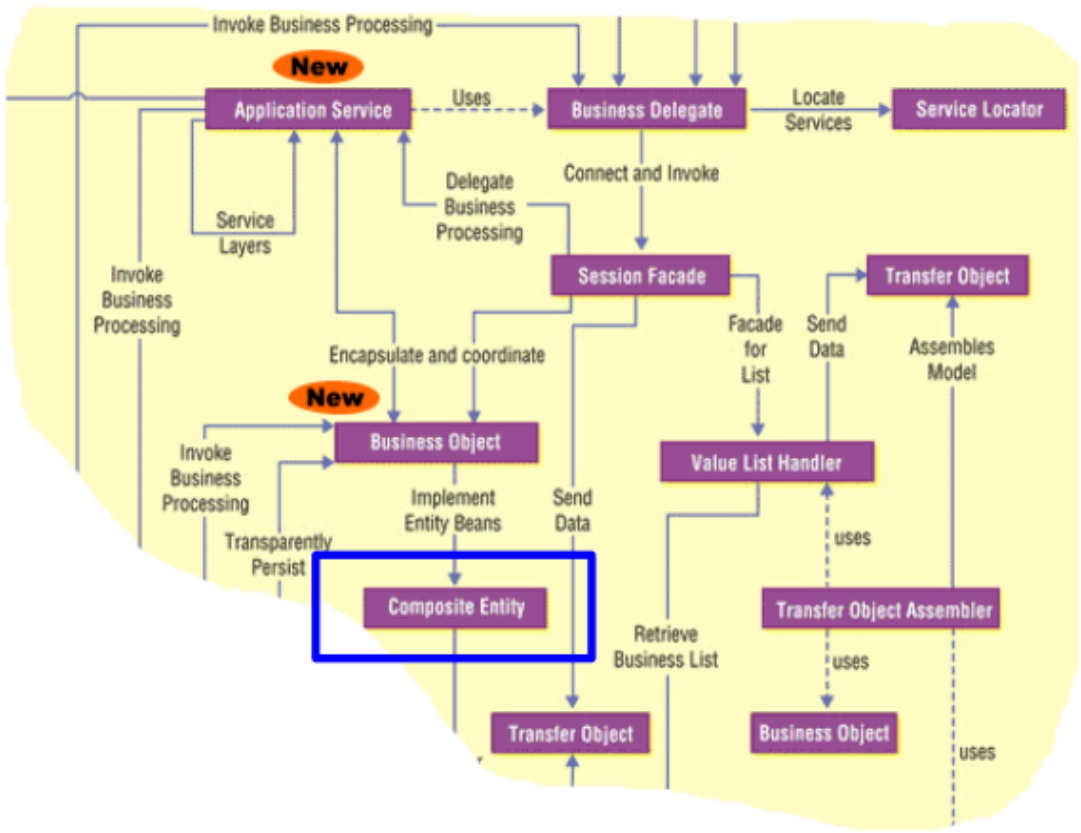
- The presentation tier can access those plain objects instead of manipulating the database directly
- In Java,
  - Two types of domain object models.
    - Enterprise Java Beans (EJBs)
    - POJOs (Plain Old Java Objects)
  - EJBs often make sense when you need a coarse-grained entity model, remote access, and integral transaction support

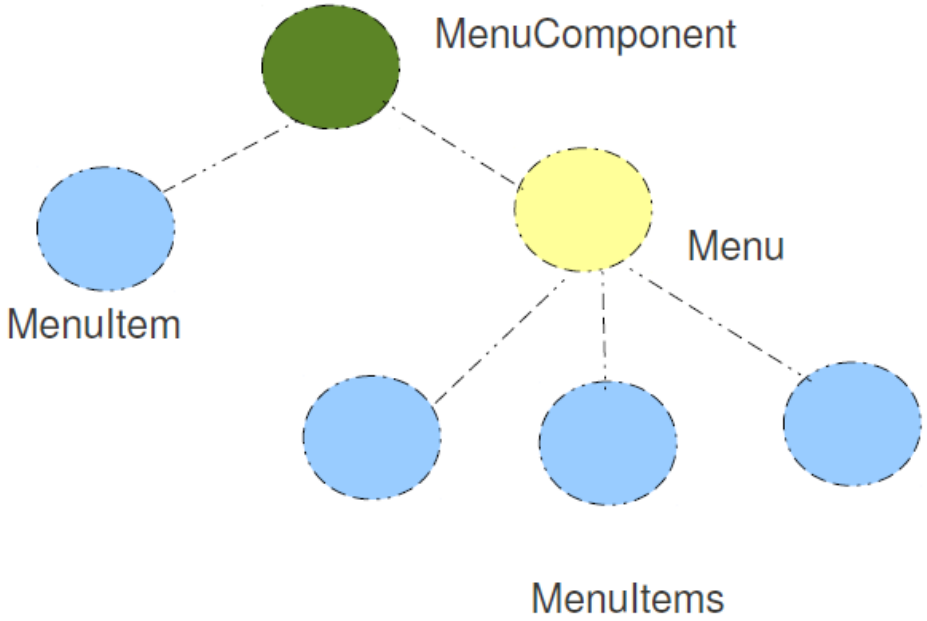
Composite Entity Pattern

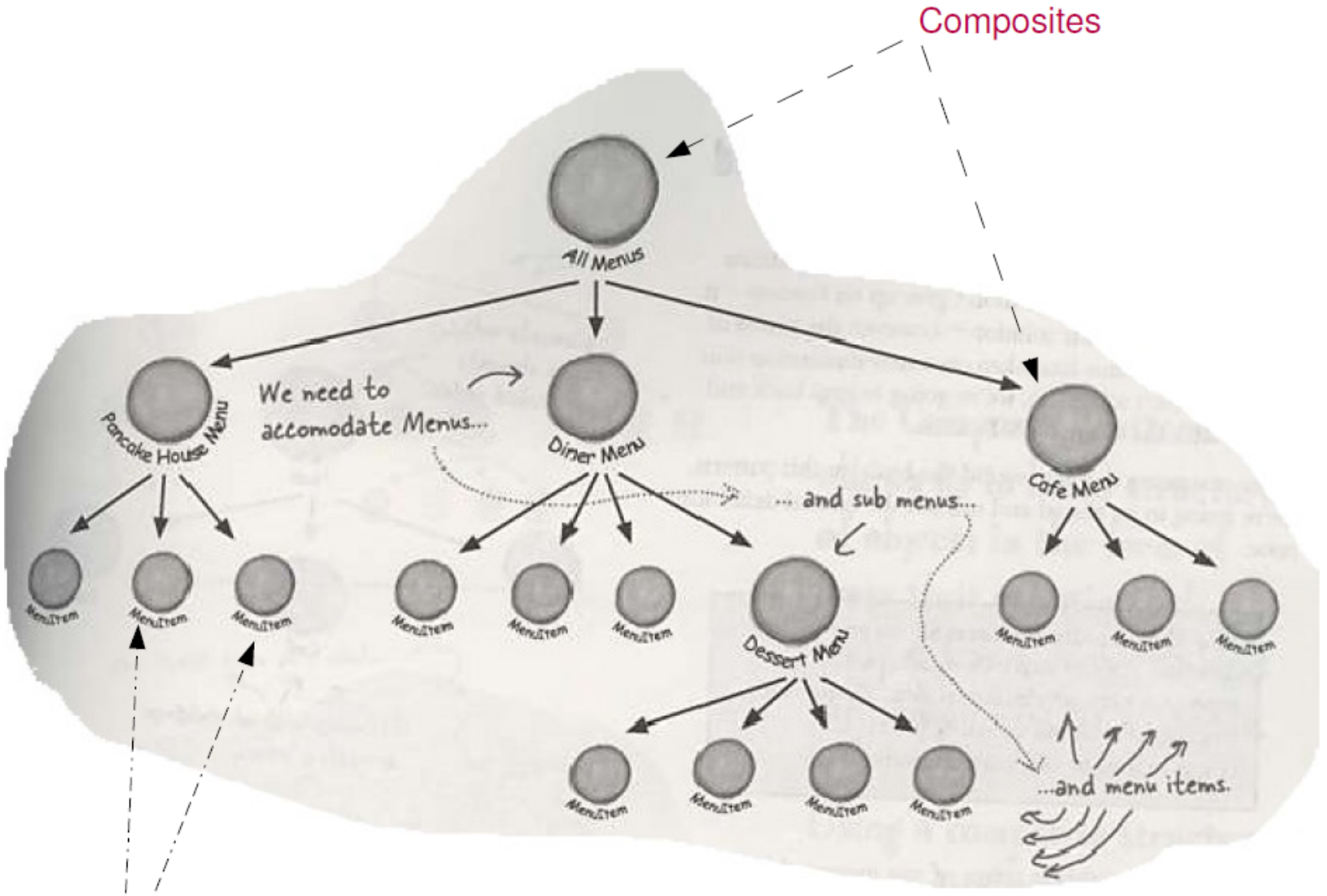


Use Composite Entity to model, represent, and manage a set of interrelated persistent objects rather than representing them as individual fine-grained objects.

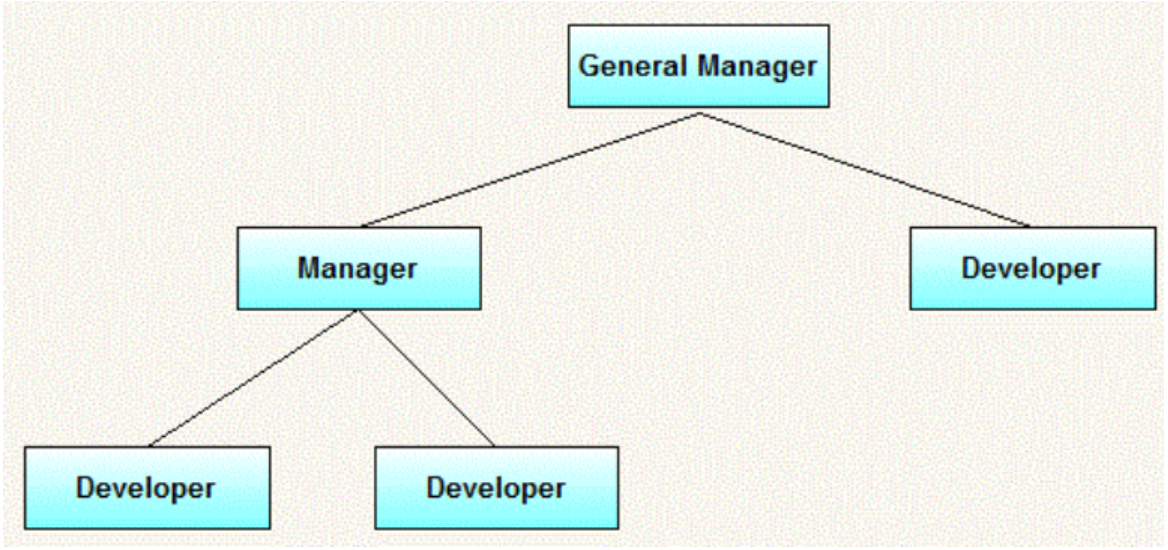
Composite Entity Pattern

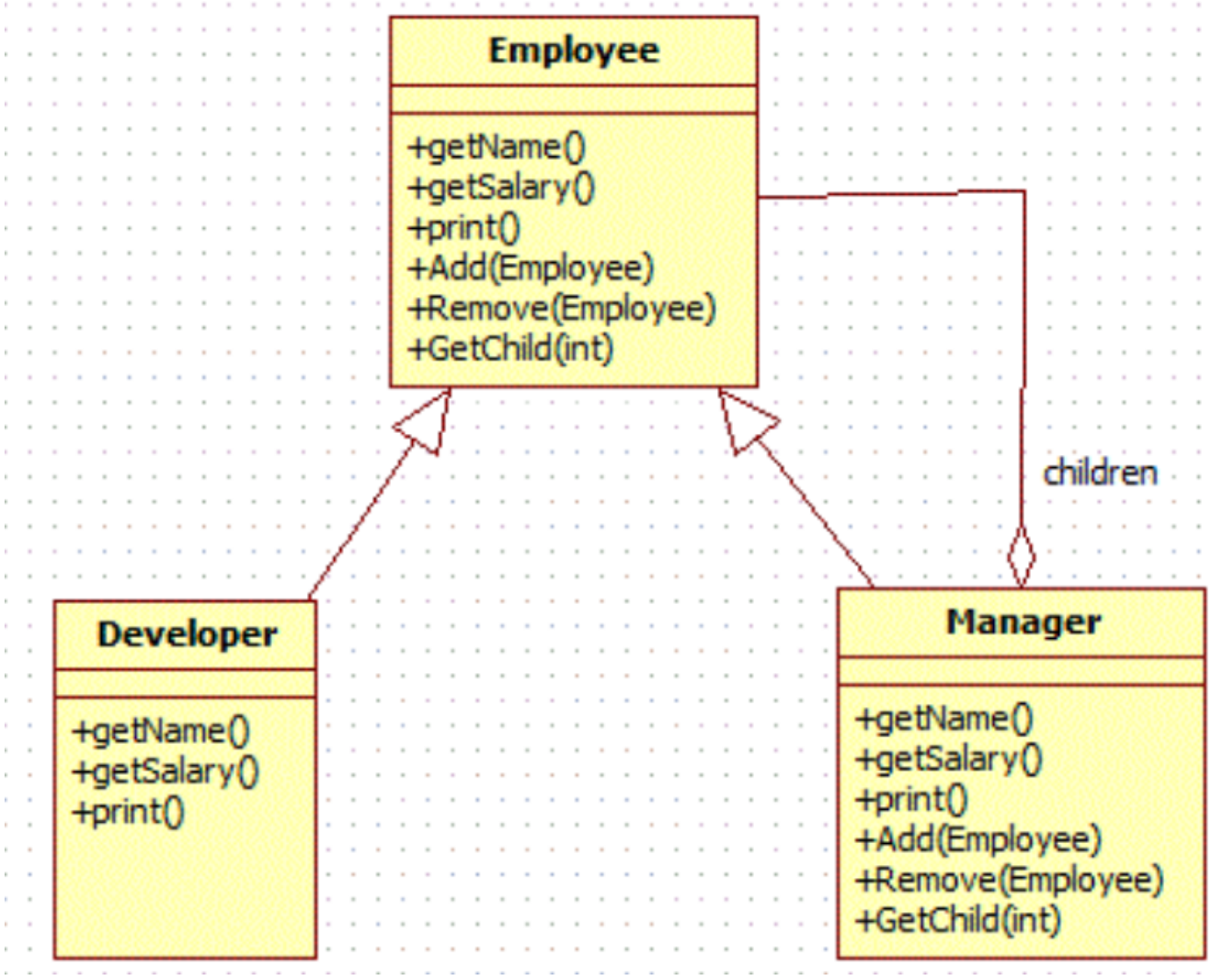



MenuComponent
Menu
MenuItem
MenuItems


Composites
All Menus
Pancake House Menu
We need to
accomodate Menus...
Diner Menu
... and sub menus...
Cafe Menu
Dessert Menu
...and menu items.
MenuItem
Leaf Nodes

In-class Activity for Composite Entity Patterns





Test class for the Output

ymentController.java | Employee.java | CompositeDesignPatternMain.java | Manag

```
public class CompositeDesignPatternMain {

    public static void main(String[] args) {

        Employee emp1 = new Developer("John", 10000);
        Employee emp2 = new Developer("David", 15000);
        Employee manager1 = new Manager("Daniel", 25000);
        manager1.add(emp1);
        manager1.add(emp2);
        Employee emp3 = new Developer("Michael", 20000);
        Manager generalManager = new Manager("Mark", 50000);
        generalManager.add(emp3);
        generalManager.add(manager1);
        generalManager.print();

    }

}
```

Console | Prob

```
<terminated> Compos
-------------
Name =Mark
Salary =50000.0
-------------
-------------
Name =Michael
Salary =20000.0
-------------
-------------
Name =Daniel
Salary =25000.0
-------------
-------------
Name =John
Salary =10000.0
-------------
-------------
Name =David
Salary =15000.0
-------------
```

Composite Entity Pattern

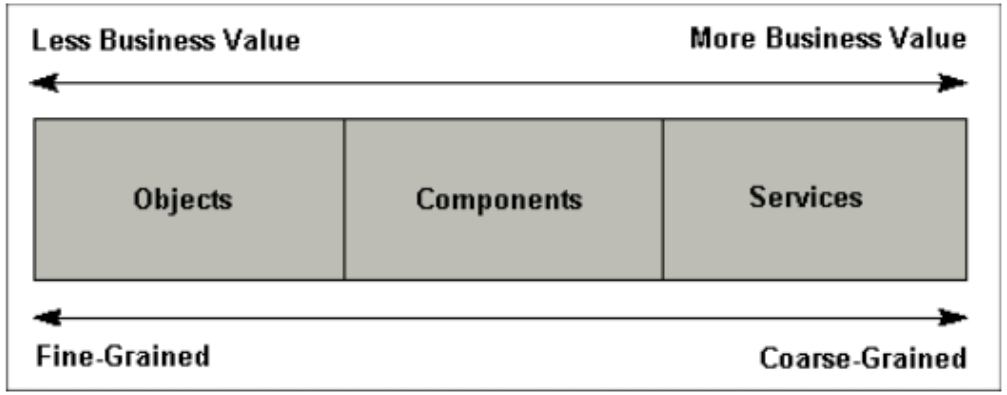
- **Why this pattern?**
  - One common mistake is mapping each table in the underlying database to a class, and each row in that table to an object instance
    - Examples: *CMP, other O/R Mapping Tools*
    - Issues:
      - *Servers must do more to maintain the integrity of each bean*
      - *Communication overhead causes performance degradation*

Persistence of Objects

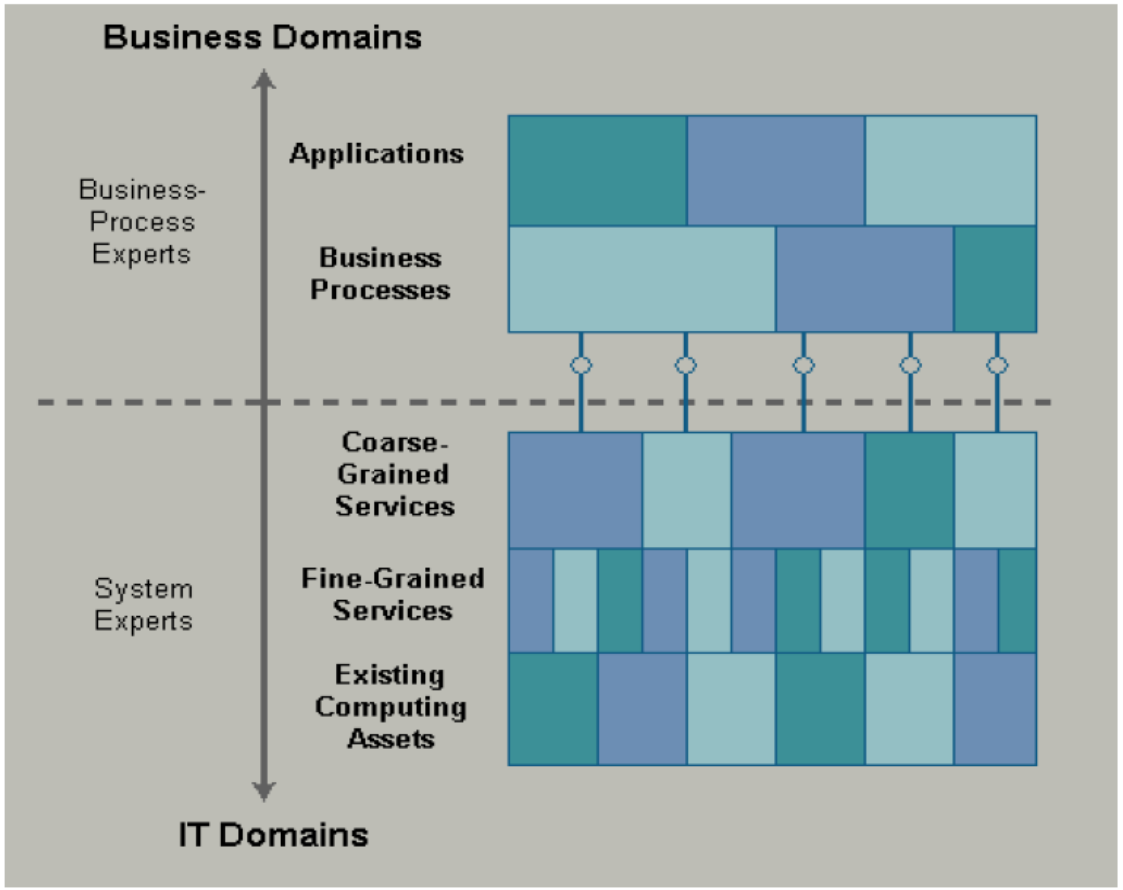
- A persistent object is an object that is stored in some type of a data store.
- Multiple clients usually share persistent objects.
- Persistent objects can be classified into two types:
  - Coarse-grained objects
  - Dependent objects

Service Composition

Degrees of Granularity
(Fine-Grained/ Coarse-Grained)



Degrees of Granularity



Service Composition

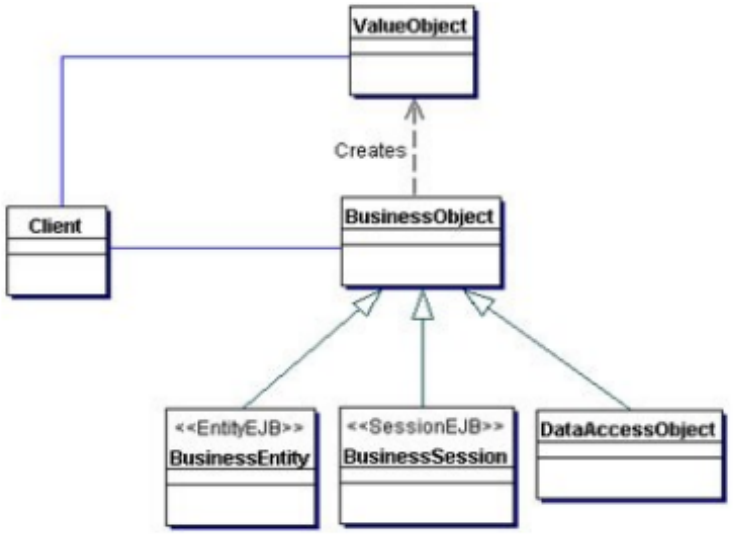
Lazy Loading

- What is it?
  - Lazy loading is a characteristic of an application when the actual loading and instantiation of a class is delayed until the point just before the instance is actually used.
  - The goal is to only dedicate memory resources when necessary by only loading and instantiating an object at the point when it is absolutely needed .

Lazy Loading

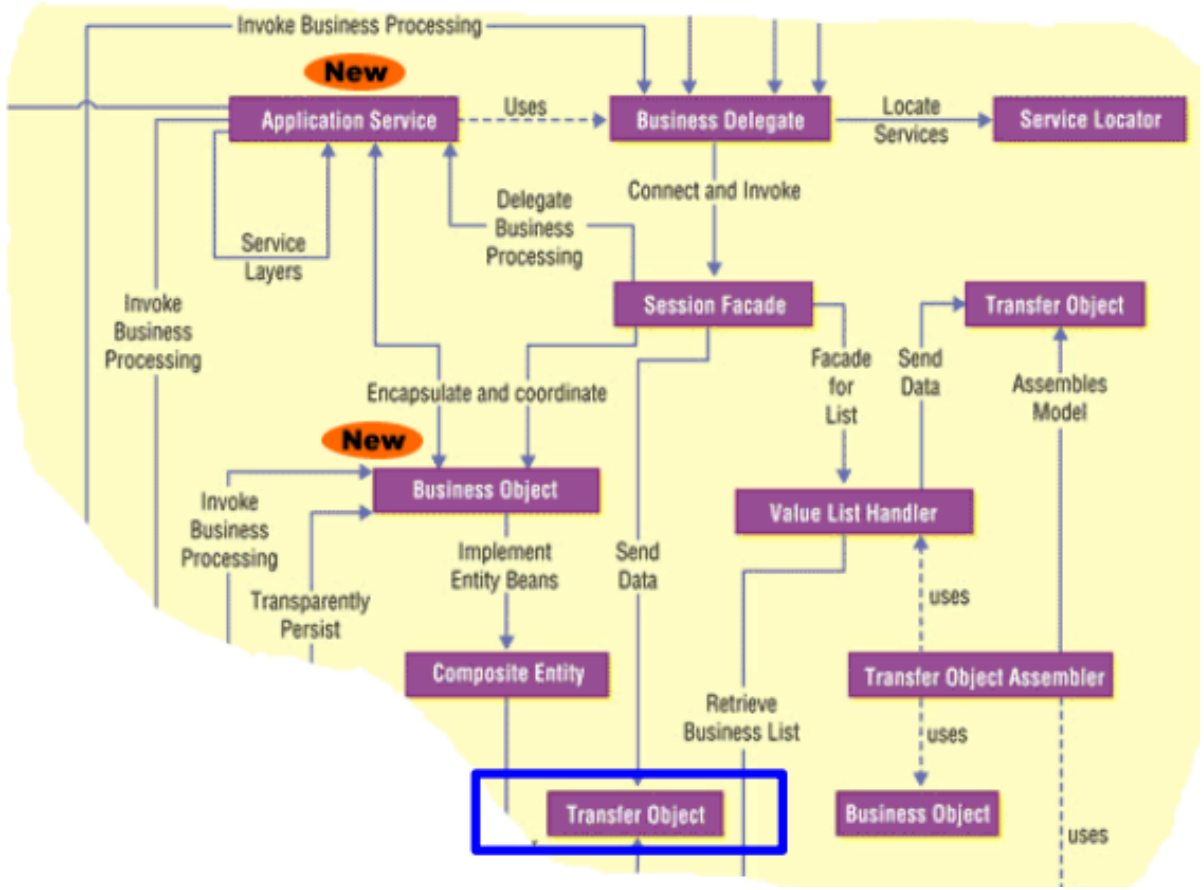
- Loading all the dependent objects of the Composite Entity may take considerable time and resources (i.e. ejbLoad() in EJB spec)
- One way to optimize this is by using a lazy loading strategy for loading the dependent objects
- When at first only load those dependent objects that are most crucial to the Composite Entity clients.
- Subsequently, when the clients access a dependent object that has not yet been loaded from the database, the Composite Entity can perform a load on demand

Transfer Object Pattern



Use a Transfer Object to encapsulate the business data. A single method call is used to send and retrieve the Transfer Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Transfer Object, populate it with its attribute values, and pass it by value to the client

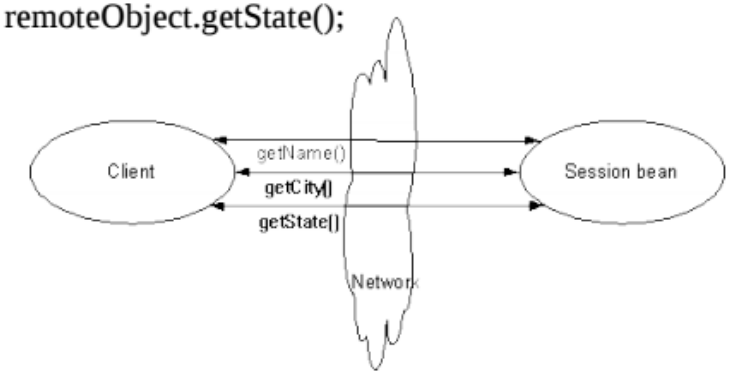
Transfer Object Pattern



Transfer Object Pattern

Problem

- Need to avoid fine-grained method calls. This imposes a overhead on the network due to the number of calls
  - remoteObject.getName(); remoteObject.getCity(); remoteObject.getState();

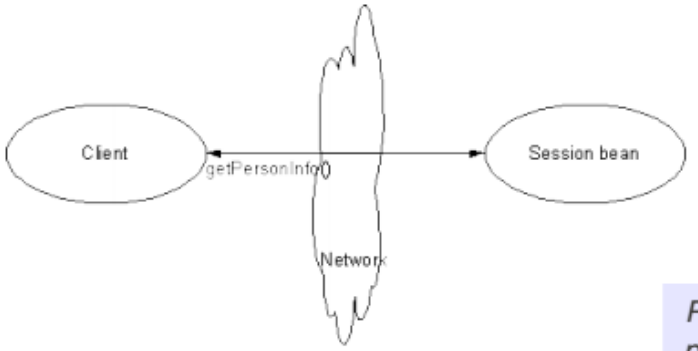


Fine grained method calls

Transfer Object Pattern

Solution

- Use the coarse-grained approach



Coarse grained method call
(returns Value Object)

```
PersonInfo person = new PersonInfo();
person.setName("Amal");
person.setCity("Colombo");
person.setState("WP");
person.zipCode("11600");

// send Value Object through network
remoteObject.getPersonInfo(person);
```

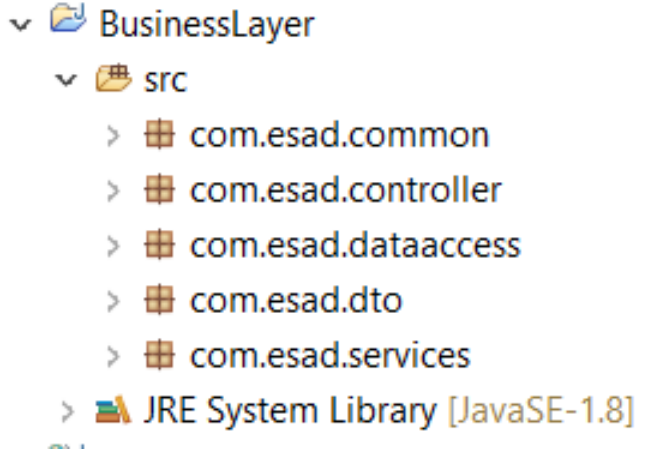
In-class Activity: Exercise 02 for Business Layer Patterns Implementation

- Implement the **Business Layer** assuming the request comes from **Presentation Layer (Controller class)** and invoke the **business delegator (ServiceDelegator)** class that fetches required **business Services (PaymentService, ReservationService)**. You can implement common Service Interface that uses in all business services. Then the required request should be forwarded to the **DataAccess Layer** to execute necessary SQL query. You can apply **Transfer Object Pattern** to send the request among layers and in **DataAccess Layer** you can get those values to map to the Database Entities. Create suitable package hierarchy and Controller implementation should be as following piece of code.

Controller Implementation

```
PaymentDTO paymentDTO = new PaymentDTO();
paymentDTO.setAmount(120000.00);
paymentDTO.setDescription("Electricity Bill");

((IPaymentService)(ServiceDelegate.getService("payment"))).addPayment(paymentDTO);
```

Package Hierarchy



Solution

```java
public interface IService {
}
```

```java
public interface IPaymentService extends IService{

    public void addPayment(PaymentDTO paymentDTO);
}
```

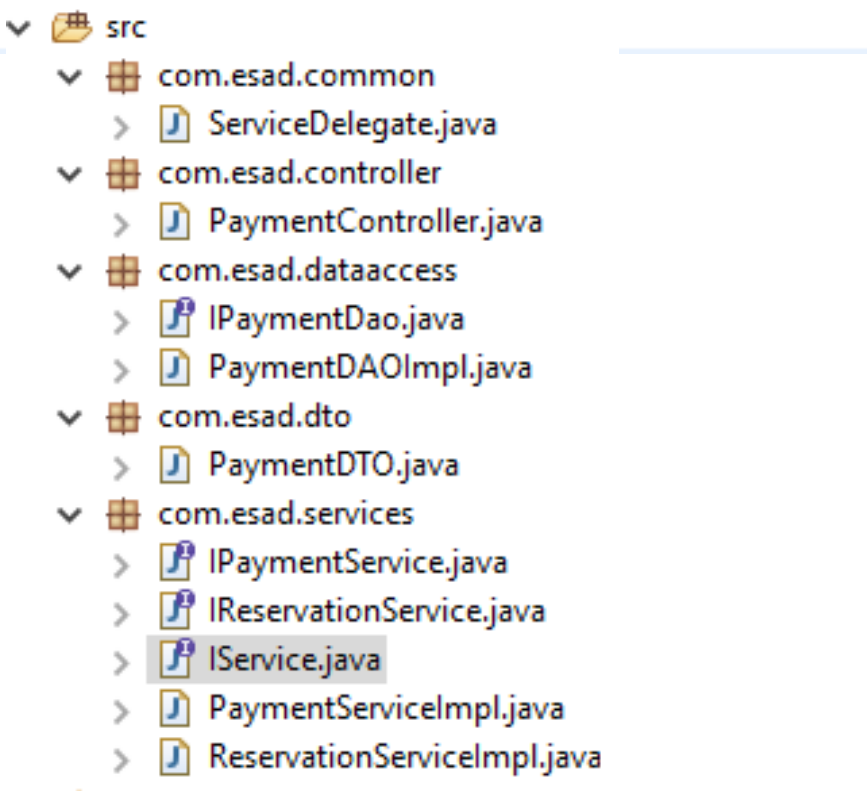
```java
public class PaymentServiceImpl implements IPaymentService {

    @Override
    public void addPayment(PaymentDTO paymentDTO) {
        double amount = paymentDTO.getAmount();
        String description = paymentDTO.getDescription();
        System.out.println("Insert for Payment Table " +
        description  + " = " + amount);
    }
}
```

```java
public class ServiceDelegate {

    private static final String PAYMENT = "payment";

    private static final String RESERVATION = "reservation";

    public static IService getService(String serviceType){

        if(serviceType.equals(PAYMENT)){
            return new PaymentServiceImpl();
        }else if(serviceType.equals(RESERVATION)){
            return new ReservationServiceImpl();
        }else{
            return null;
        }
    }
```

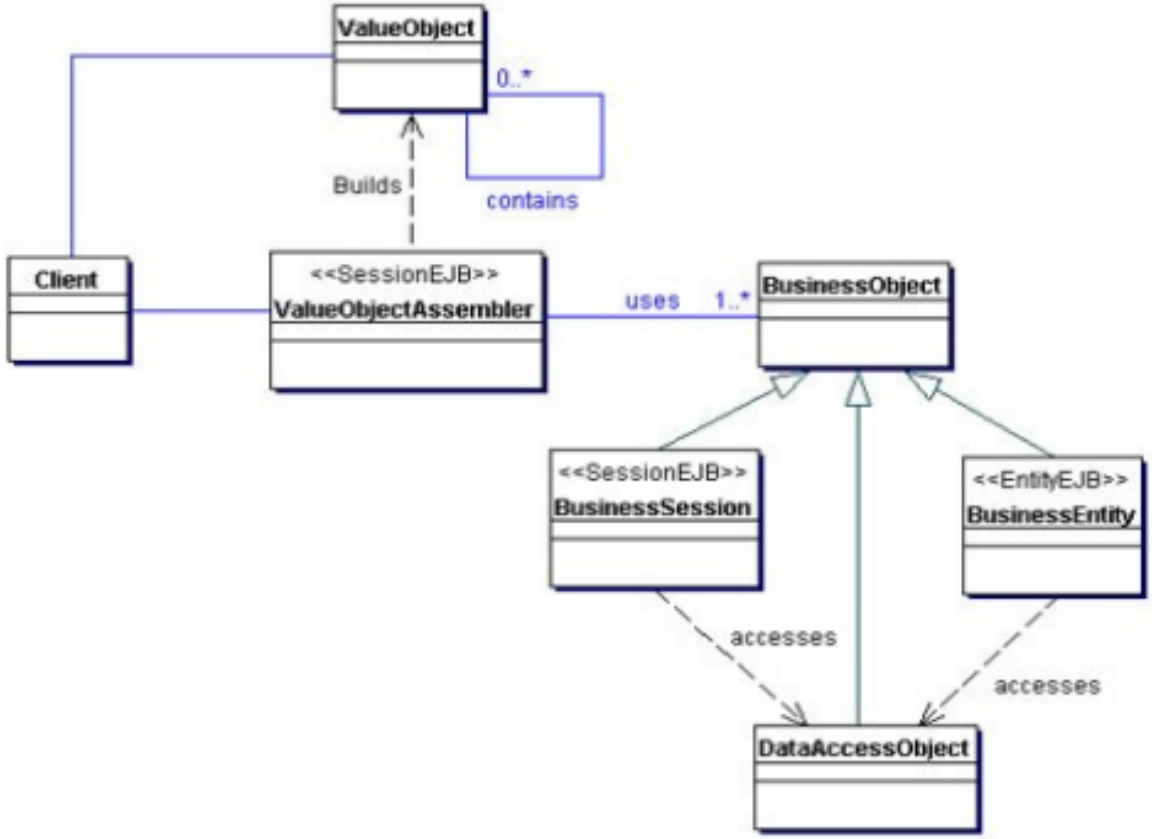
```java
public class PaymentDTO {

    private String ID;
    private String description;
    private double amount;

    public String getID() {
        return ID;
    }

    public void setID(String iD) {
        ID = iD;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```

```java
public interface IPaymentDao {

    public boolean addPayment(Object obj);
}
```

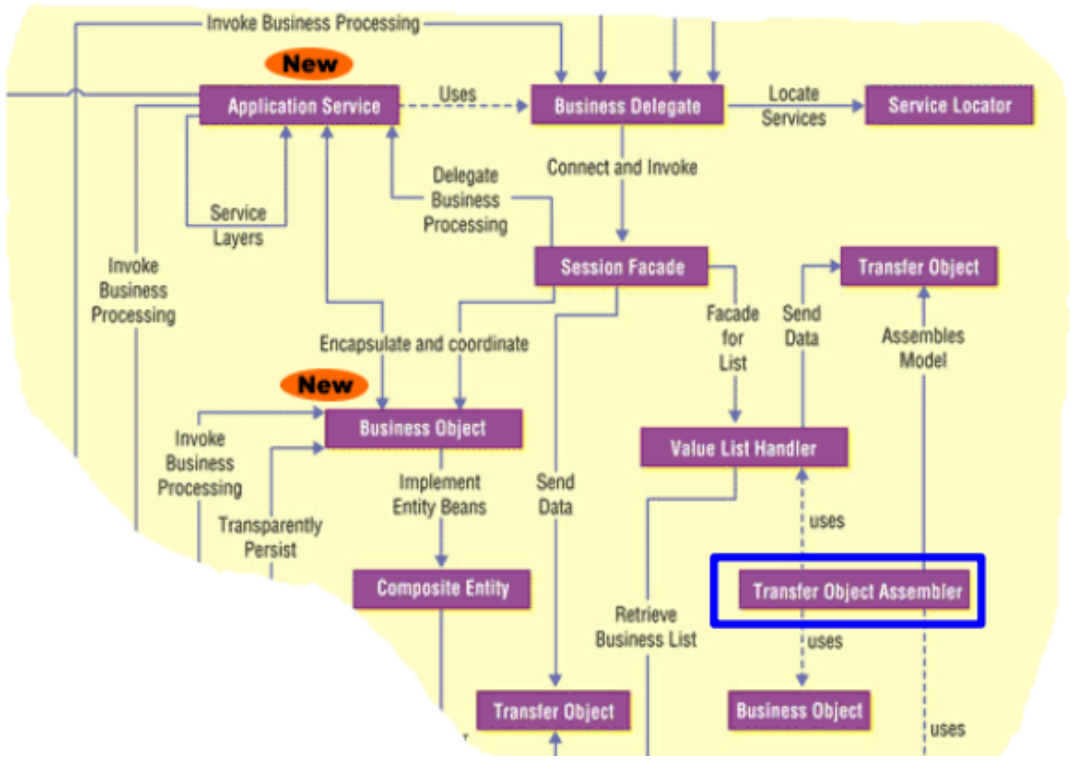
```java
public class PaymentDAOImpl implements IPaymentDao{
    @Override
    public boolean addPayment(Object obj) {
        return false;
    }
}
```



Transfer Object Assembler Pattern



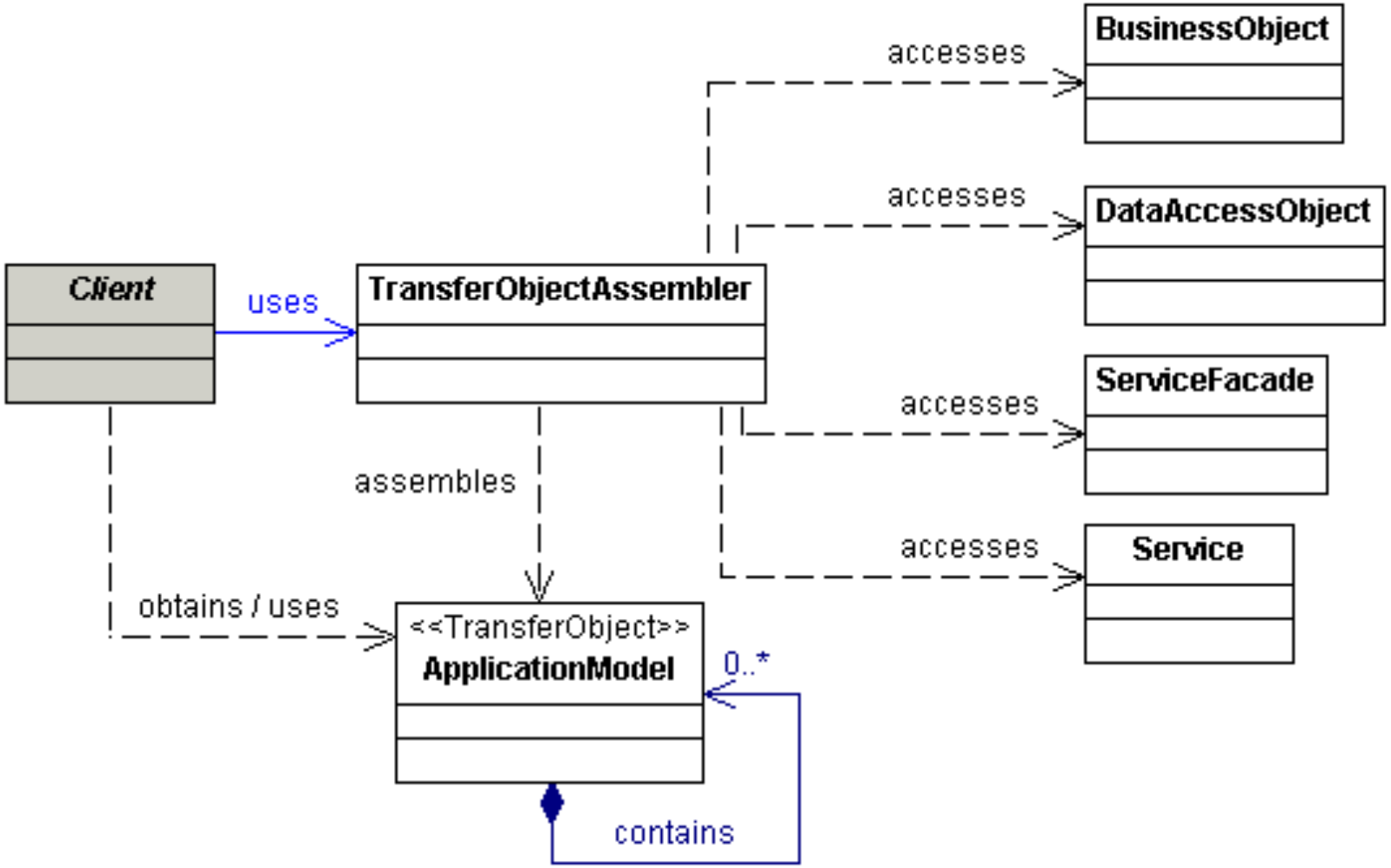
Transfer Object Assembler Pattern



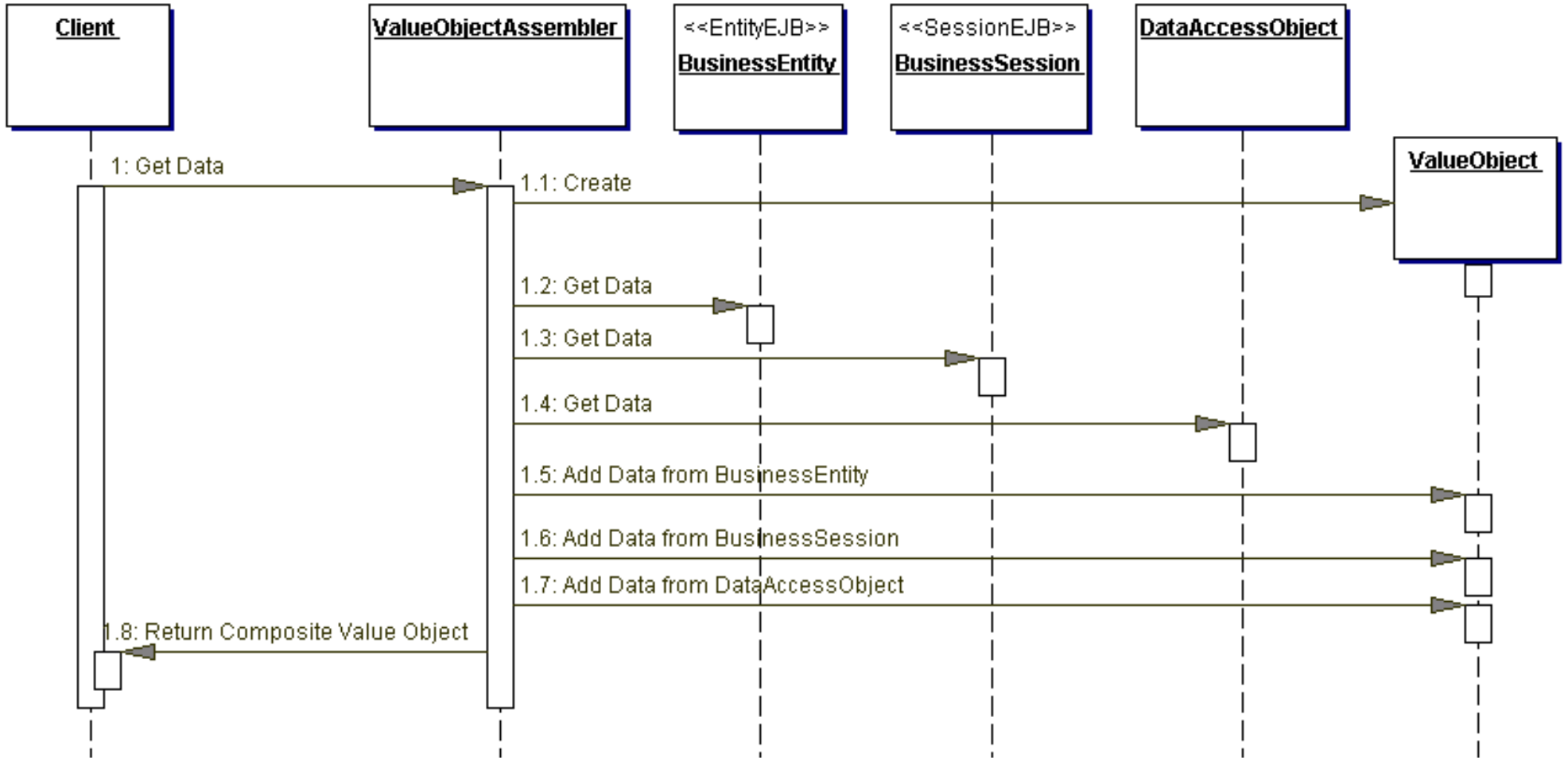
Use a Transfer Object Assembler to build the required model or sub-model. This uses Transfer Objects to retrieve data from various business objects and other objects that define the model or part of the model.

- Use a Transfer Object Assembler to build an application model as a composite Transfer Object.

- The **Transfer Object Assembler aggregates multiple Transfer Objects** from various business components and services and returns it to the client.



Transfer Object Assembler Pattern



In-class Activity: Exercise 04 for Transfer Object Assembler

Refer the below TO classes (ResourceTO, TaskTO, ProjectTO, ProjectManagerTO, and TaskResourceTO)

```java
public class ResourceTO {
    private String resourceId;
    private String resourceName;
    private String resourceEmail;
    public String getResourceId() {
        return resourceId;
    }
    public void setResourceId(String resourceId) {
        this.resourceId = resourceId;
    }
    public String getResourceName() {
        return resourceName;
    }
    public void setResourceName(String resourceName) {
        this.resourceName = resourceName;
    }
    public String getResourceEmail() {
        return resourceEmail;
    }
    public void setResourceEmail(String resourceEmail) {
        this.resourceEmail = resourceEmail;
    }
}
```

```java
public class TaskTO {
 private String projectId;
 private String taskId;
 private String name;
 private String description;
 private Date startDate;
 private Date endDate;
 private String assignedResourceId;
 public String getProjectId() {
  return projectId;
 }
 public void setProjectId(String projectId) {
  this.projectId = projectId;
 }
 public String getTaskId() {
  return taskId;
 }
public void setTaskId(String taskId) {
 this.taskId = taskId;
}
public String getName() {
 return name;
}
public void setName(String name) {
 this.name = name;
}
```

```java
public class TaskResourceTO {
    private String projectId;
    private String taskId;
    private String name;
    private String description;
    private Date startDate;
    private Date endDate;
    private TaskResourceTO assignedResource;
    public String getProjectId() {
        return projectId;
    }
    public void setProjectId(String projectId) {
        this.projectId = projectId;
    }
    public String getTaskId() {
        return taskId;
    }
     public void setTaskId(String taskId) {
         this.taskId = taskId;
     }
     public String getName() {
         return name;
     }
     public void setName(String name) {
         this.name = name;
     }
```

```java
public class ProjectManagerTO {

    private String name;
    private String address;
    private String projects;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getAddress() {
        return address;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public String getProjects() {
        return projects;
    }
    public void setProjects(String projects)
        this.projects = projects;
    }

}
```

```java
public class ProjectTO {

    private String projectId;
    private String projectName;
    private String projectDesc;
    public String getProjectId() {
        return projectId;
    }
    public void setProjectId(String projectId) {
        this.projectId = projectId;
    }
    public String getProjectName() {
        return projectName;
    }
    public void setProjectName(String projectName) {
        this.projectName = projectName;
    }
    public String getProjectDesc() {
        return projectDesc;
    }
    public void setProjectDesc(String projectDesc) {
        this.projectDesc = projectDesc;
    }
}
```

A Transfer Object Assembler pattern can be implemented to assemble this composite transfer object.

```
public class ProjectDetailsData {
    private ProjectTO projectData;
    private ProjectManagerTO projectManagerData;
    private Collection < TaskResourceTO > listOfTasks;

    public ProjectDetailsData(ProjectTO projectData, ProjectManagerTO projectManagerData,
        Collection < TaskResourceTO > listOfTasks) {
        super();
        this.projectData = projectData;
        this.projectManagerData = projectManagerData;
        this.listOfTasks = listOfTasks;
    }
}
```

In-class Activity: Exercise 04

- Create Transfer Object Assembler class to aggregate Project data, Project Manager Data, and List of tasks using classes (ProjectTO, ProjectManagerTO, and collection of ResourcesTO) Refer the provided transfer Object classes and implement the **Project Details Assembler** class

Solution

```
public class ProjectDetailsAssembler {

    public ProjectDetailsData getData(String projectId) {

        // Construct the composite transfer object
        // get project related information from database and set to ProjectDetailsData class object.
        ProjectTO projectData = new ProjectTO();

        // get ProjectManager info and add to ProjectDetailsData
        ProjectManagerTO projectManagerData = new ProjectManagerTO();

        // construct a new TaskResourceTO using Task and Resource data.
        //get the Resource details from database.
        // construct a list of TaskResourceTOs
        Collection < TaskResourceTO > listOfTasks = new ArrayList < > ();
        // Add Project Info to ProjectDetailsData
        // Add ProjectManager info to ProjectDetailsData
        ProjectDetailsData pData = new ProjectDetailsData(projectData, projectManagerData, listOfTasks)
        return pData;
    }
}
```

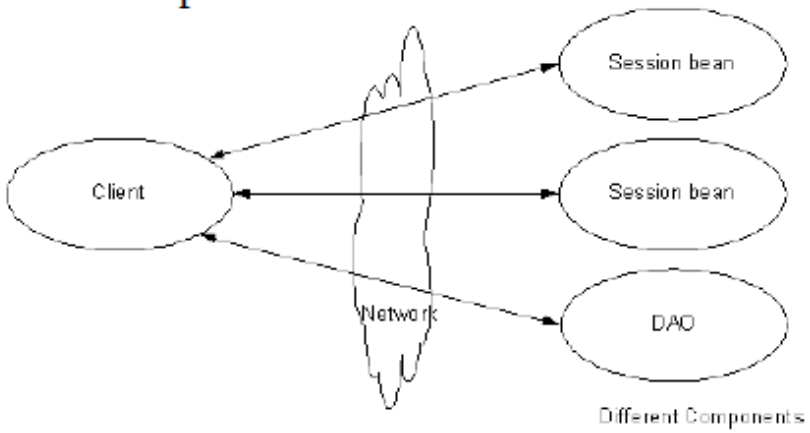
Transfer Object Assembler Pattern

Problem

- The application model is an *abstraction* of the business data and business logic implemented on the server side as business components.
- In a J2EE application, the application model is a distributed collection of objects such as *session beans*, *entity beans*, or *DAOs* and *other objects*
- For a client to obtain the data for the model, such as to display to the user or to perform some processing, it must access individually each distributed object that defines the model

Problem

- The client must reconstruct the model after obtaining the model's parts from the distributed components. The client therefore needs to have the necessary business logic to construct the model. If the model construction is complex and numerous objects are involved in its definition, then there may be an additional performance overhead on the client due to the construction process



Transfer Object Assembler Pattern

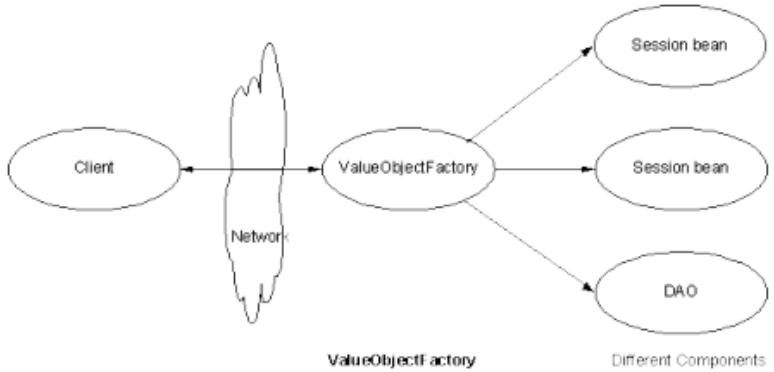
Problem

- When the client constructs the application model, the construction happens on the client side. Complex model construction can result in a significant performance overhead on the client side for clients with limited resources
- Because the client is tightly coupled to the model, changes to the model require changes to the client

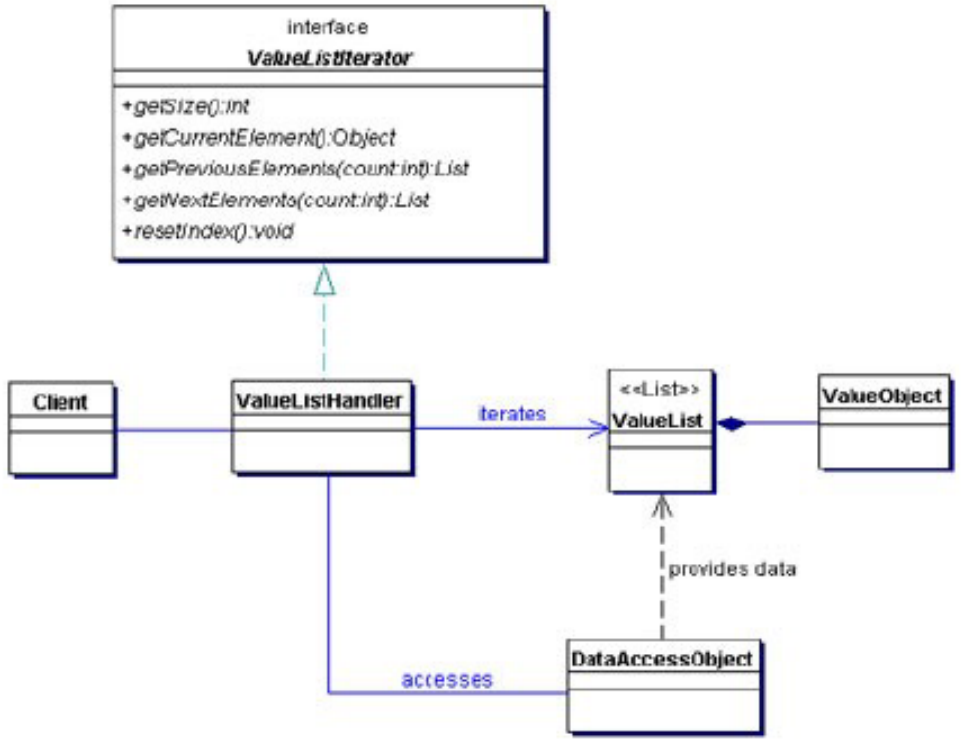
Transfer Object Assembler Pattern

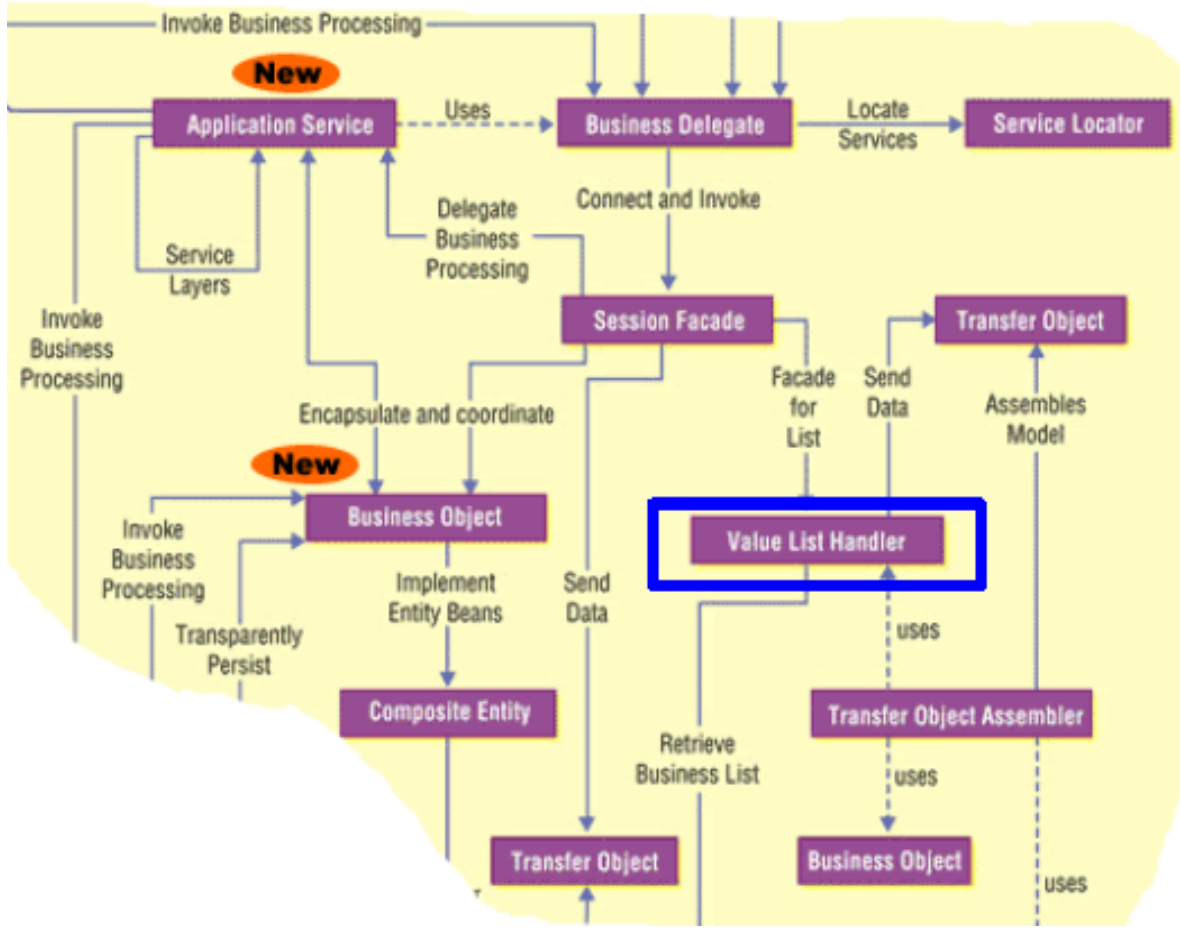
Solution

- Using the Transfer Object Assembler
  - To reduce the network traffic due to accessing multiple components by a client for a single request, this holds different ValueObjects as place holders and respond with a single ValueObject for a client request



Value List Handler Pattern



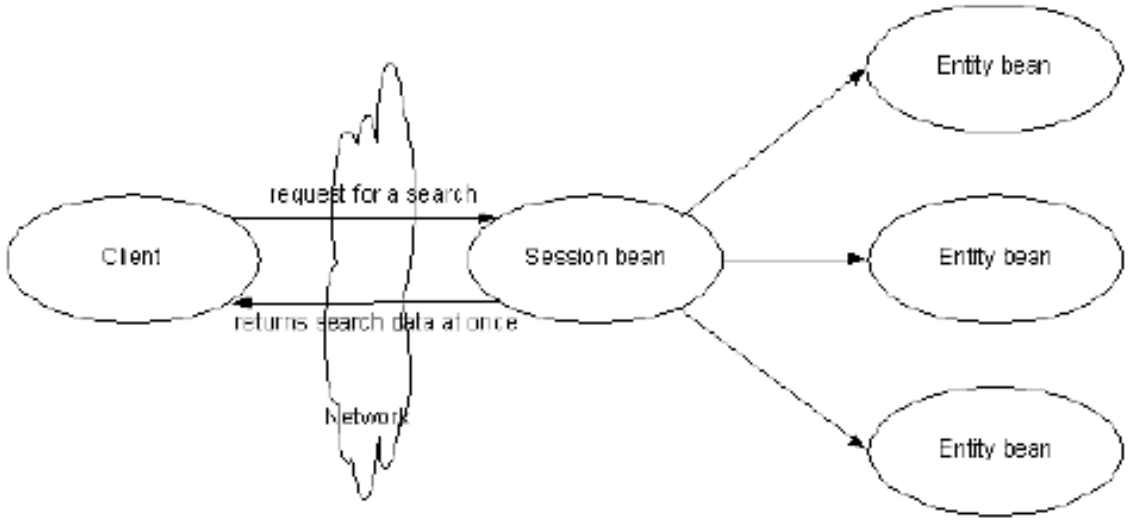


Use a Value List Handler to control the search, cache the results, and provide the results to the client in a result set whose size and traversal meets the client's requirements

Value List Handler Pattern

Problem

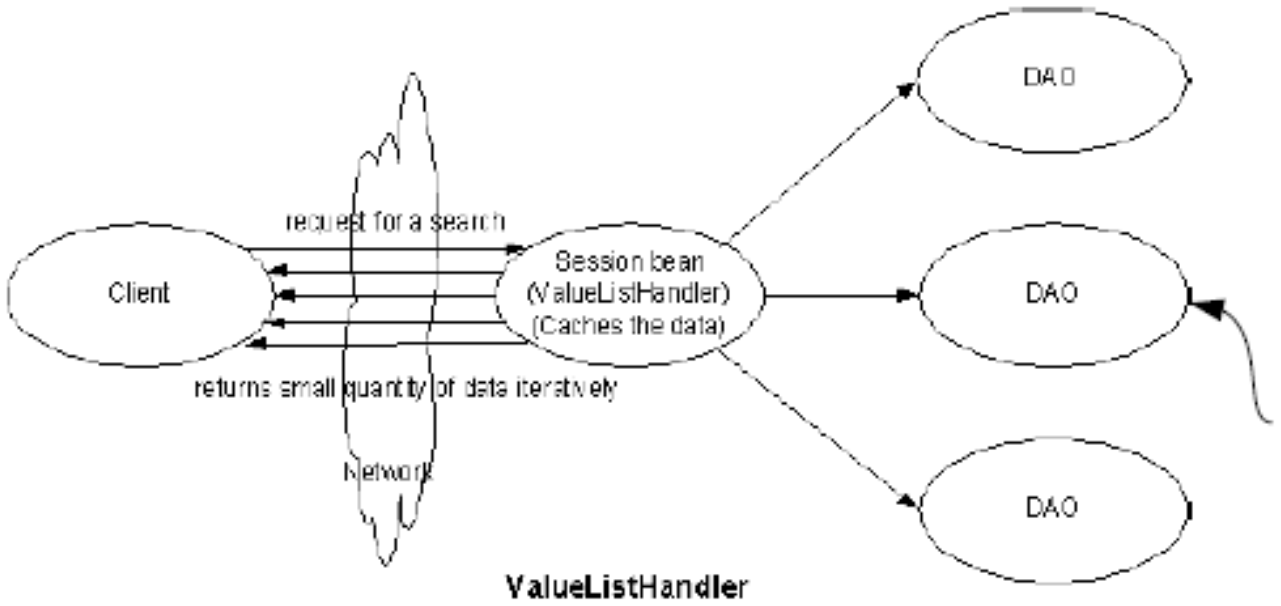
- Enterprise applications generally have the **search facility** and have to search huge data and retrieve results
- If an application returns huge queried data to the client, the client takes long time to retrieve that large data



Value List Handler Pattern

Solution

- Using the Value List Handler
  - Return **small quantity of data** *multiple times iteratively* rather than returning large amount of data at once to the client



Using DAO instead of Entity Beans

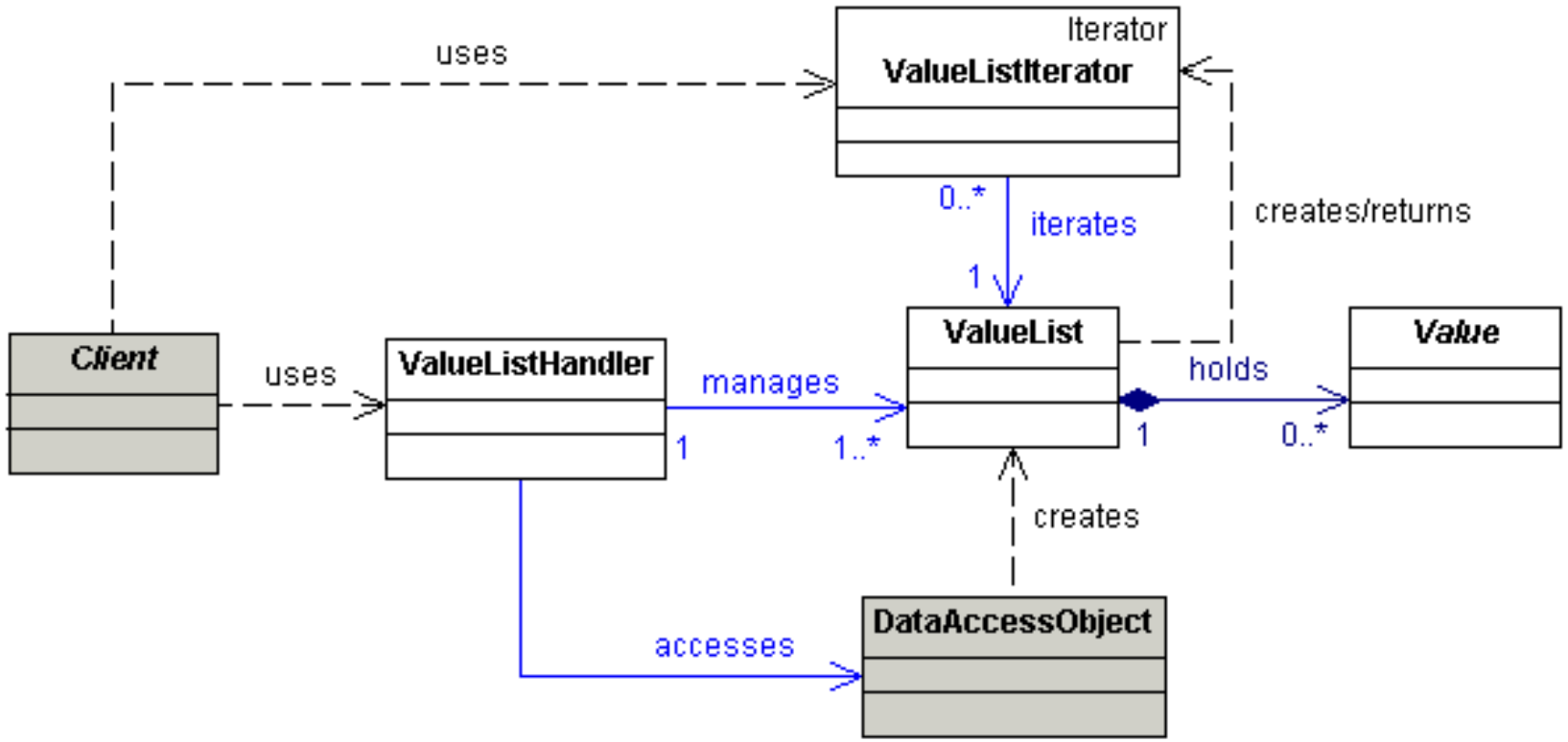
Value List Handler Pattern

Solution

- This pattern creates a ValueListHandler to control query execution functionality and results caching.
- This directly accesses a **DAO** that can execute the required query.
- Then it stores the results obtained from the DAO as a collection of Transfer Objects.
- The client requests the ValueListHandler to provide the query results as needed.
- This implements an Iterator pattern [GoF] to provide the solution

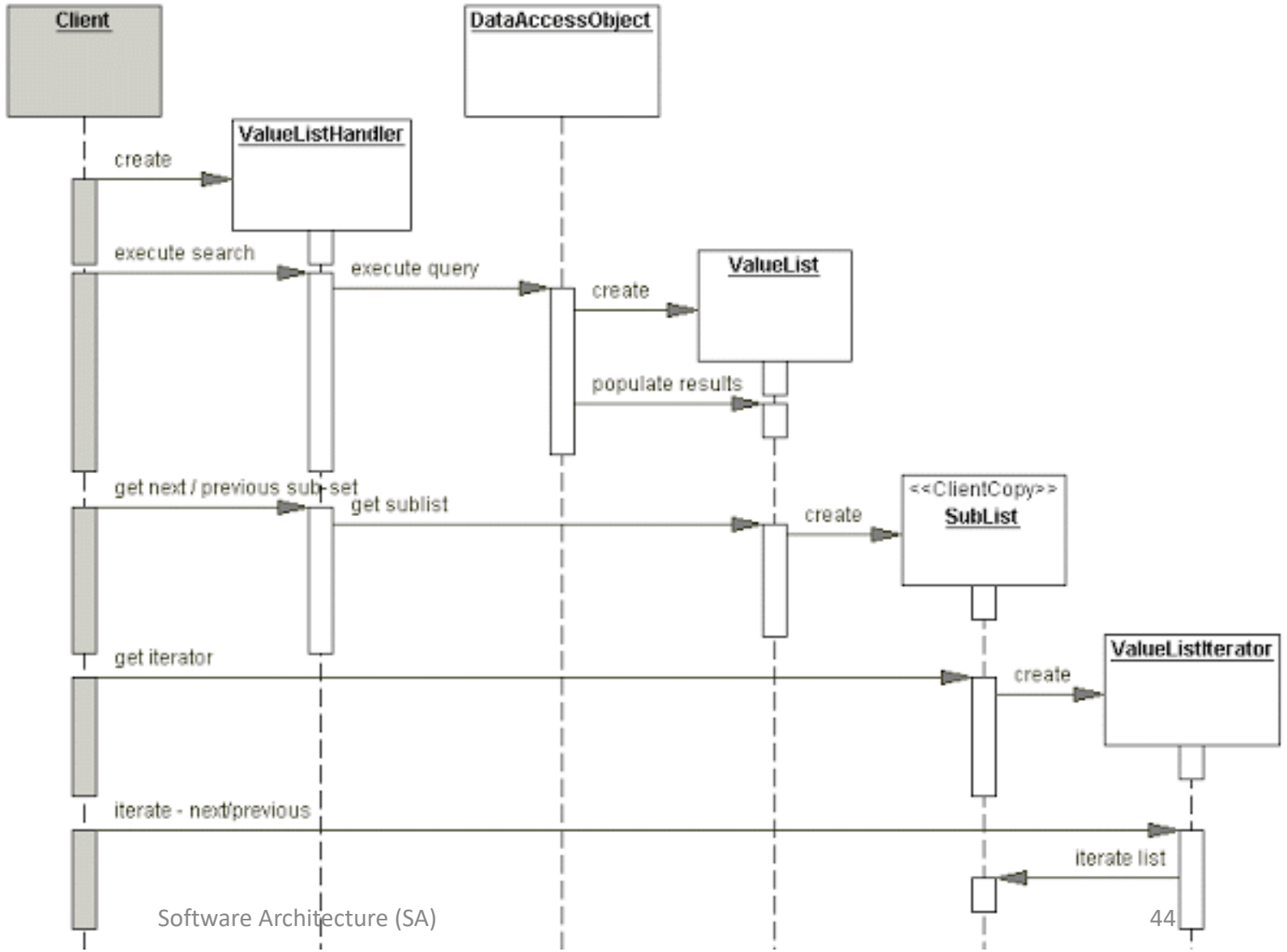
Value List Handler Pattern

Class Diagram



Value List Handler Pattern

Sequence Diagram



Value List Handler Pattern

```java
public class ProjectTO {
    private String projectId;
    private String projectName;
    private String managerId;
    private Date startDate;
    private Date endDate;
    private boolean started;
    private boolean completed;
    private boolean accepted;
    private Date acceptedDate;
    private String customerId;
    private String projectDescription;
    private String projectStatus;

    public String getProjectId() {
        return projectId;
    }

    public void setProjectId(String projectId) {
        this.projectId = projectId;
    }
```

```java
public interface ValueListIterator {
    public int getSize() throws IteratorException;

    public Object getCurrentElement() throws IteratorException;

    public List getPreviousElements(int count) throws IteratorException;

    public List getNextElements(int count) throws IteratorException;

    public void resetIndex() throws IteratorException;

    // other common methods as required
    // ...
}
```

```
public class ValueListHandler implements ValueListIterator {

    protected List list;
    protected ListIterator listIterator;
    public ValueListHandler() {}

    protected void setList(List list) throws IteratorException {
        this.list = list;
        if (list != null)
            listIterator = list.listIterator();
        else
            throw new IteratorException("List empty");
    }

    public Collection getList() {
        return list;
    }

    public int getSize() throws IteratorException {
        int size = 0;
        if (list != null)
            size = list.size();
        else
            throw new IteratorException("No data found"); //No Data
        return size;
    }
    public Object getCurrentElement() throws IteratorException {

        Object obj = null;
        // Will not advance iterator
        if (list != null) {
            int currIndex = listIterator.nextIndex();
            obj = list.get(currIndex);
        } else
            throw new IteratorException("");
        return obj;
    }
```

```
public List getPreviousElements(int count) throws IteratorException {
    int i = 0;
    Object object = null;
    LinkedList list = new LinkedList();
    if (listIterator != null) {
        while (listIterator.hasPrevious() && (i < count)) {
            object = listIterator.previous();
            list.add(object);
            i++;
        }
    }
    else
        throw new IteratorException("No data found");
    return list;
}

public List getNextElements(int count) throws IteratorException {
    int i = 0;
    Object object = null;
    LinkedList list = new LinkedList();
    if (listIterator != null) {
        while (listIterator.hasNext() && (i < count)) {
            object = listIterator.next();
            list.add(object);
            i++;
        }
    } else
        throw new IteratorException("No data found");
    return list;
}
    public void resetIndex() throws IteratorException {
        if (listIterator != null) {
            listIterator = list.listIterator();
        } else
            throw new IteratorException("No data found");
    }
}
```

References

- Patterns of Enterprise Application Architecture (PoEAA): *Martin Fowler*
- J2EE Design Patterns: *William Crawford & Jonathan Kaplan*
- Core J2EE Patterns: *Deepak Alur, John Crupi, Dan Malks*
- http://articles.techrepublic.com.com/5100-10878_11-5064520.html
- http://www.precisejava.com/javaperf/j2ee/Patterns.htm#Patterns103