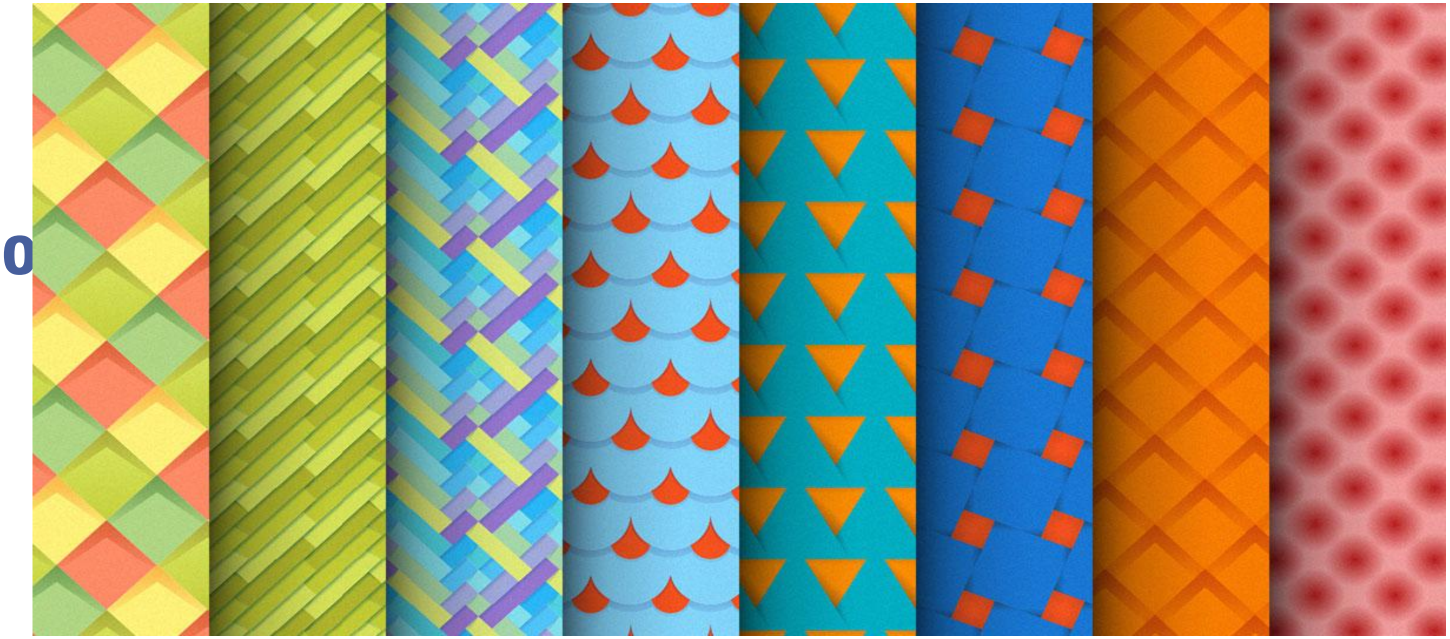
DESIGN PATTERNS

COURSE 10

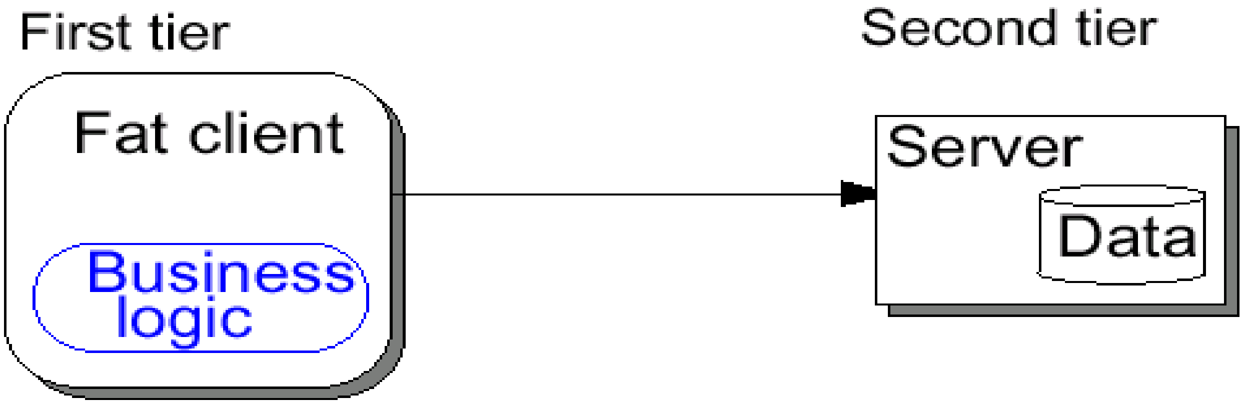


CONTENT

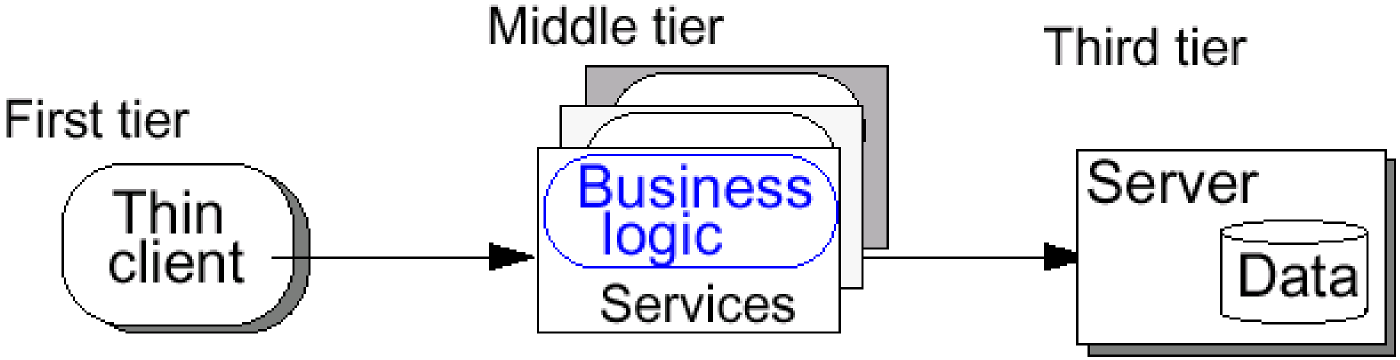
- ❑ J2EE Design Patterns

APPLICATION SERVERS

- In the 90’s, systems should be *client-server*



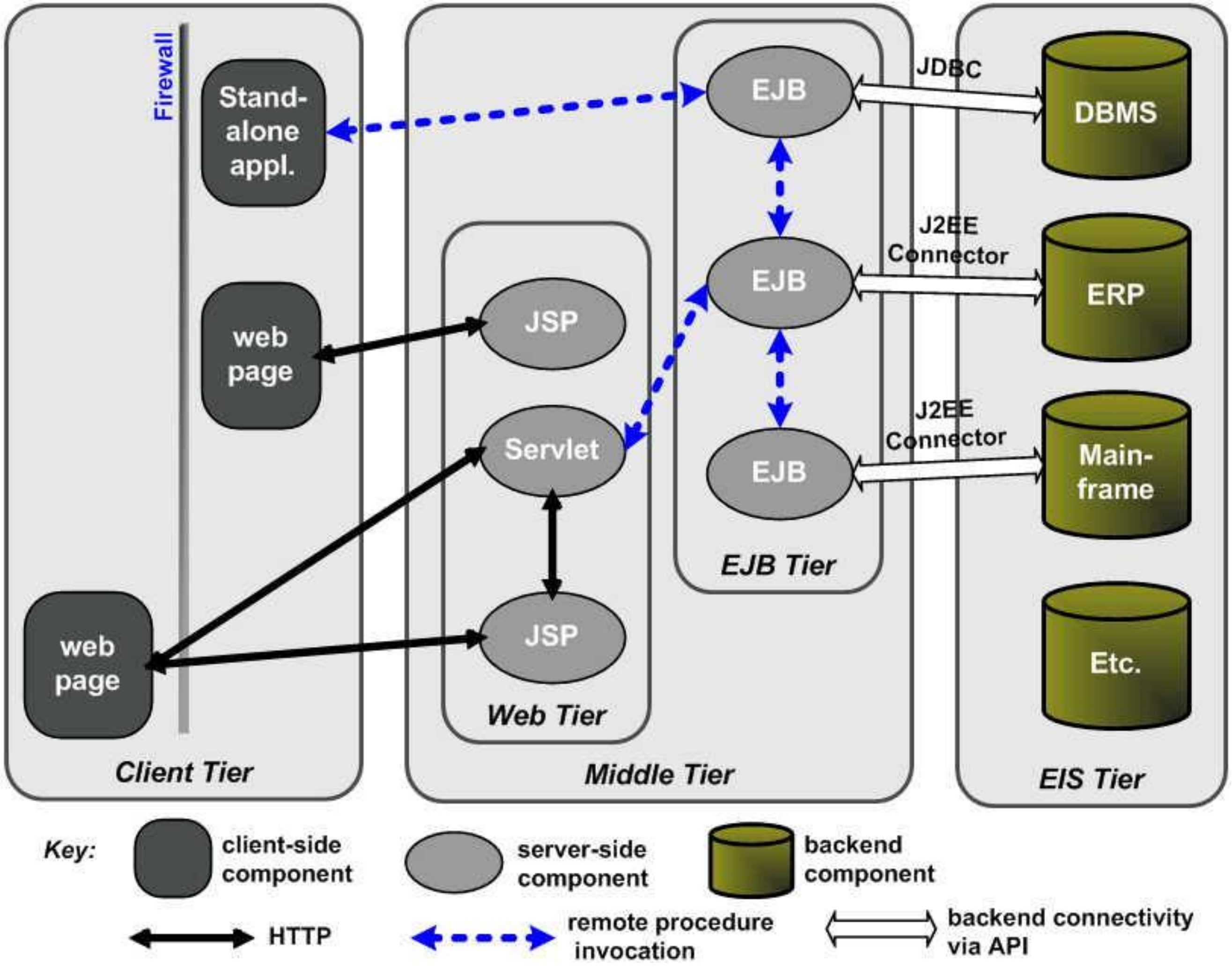
- Today, enterprise applications use the *multi-tier* model



J2EE

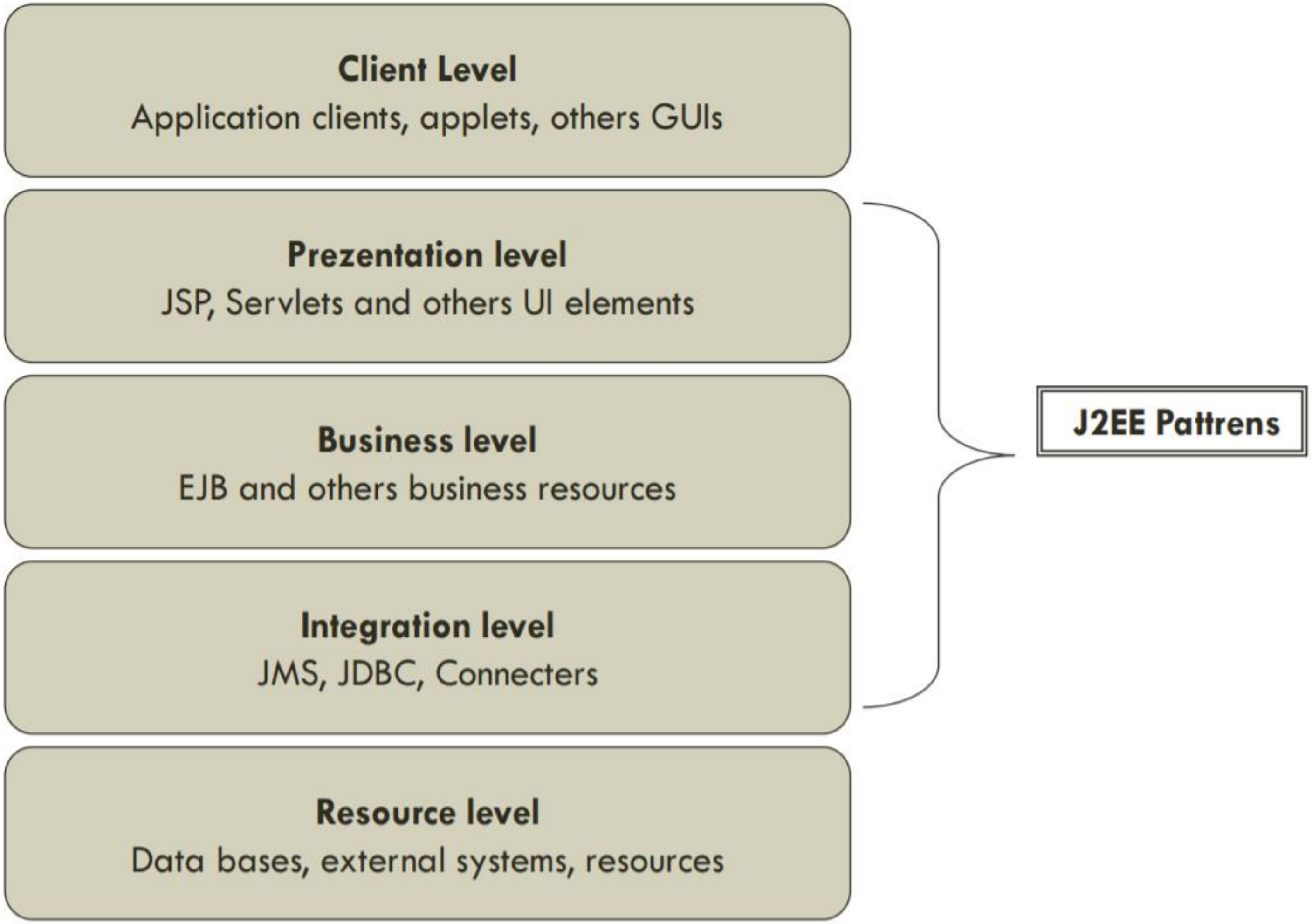
- **Main Components**
  - JavaServer Pages (JSP)
    - Used for web pages with dynamic content
    - Processes HTTP requests (non-blocking call-and-return)
    - Accepts HTML tags, special JSP tags, and scriptlets of Java code
    - Separates static content from presentation logic
    - Can be created by web designer using HTML tools
  - Servlet
    - Used for web pages with dynamic content
    - Processes HTTP requests (non-blocking call-and-return)
    - Written in Java; uses print statements to render HTML
    - Loaded into memory once and then called many times
    - Provides APIs for session management
  - Enterprise JavaBeans (EJB)
    - EJBs are *distributed components* used to implement business logic (no UI)
    - Developer concentrates on business logic
    - Availability, scalability, security, interoperability and integrability handled by the J2EE server
    - Client of EJBs can be JSPs, servlets, other EJBs and external aplications
    - Clients see *interfaces*

J2EE



Firewall
Stand-alone appl.
web page
web page
JSP
Servlet
JSP
Web Tier
EJB
EJB
EJB
EJB Tier
JDBC
J2EE Connector
J2EE Connector
DBMS
ERP
Main-frame
Etc.
Client Tier
Middle Tier
EIS Tier
Key:
client-side component
server-side component
backend component
HTTP
remote procedure invocation
backend connectivity via API

APPLICATIONS SPLIT ON LEVELS



Client Level
Application clients, applets, others GUIs

Prezentation level
JSP, Servlets and others UI elements

Business level
EJB and others business resources

Integration level
JMS, JDBC, Connecters

Resource level
Data bases, external systems, resources

J2EE Pattrens

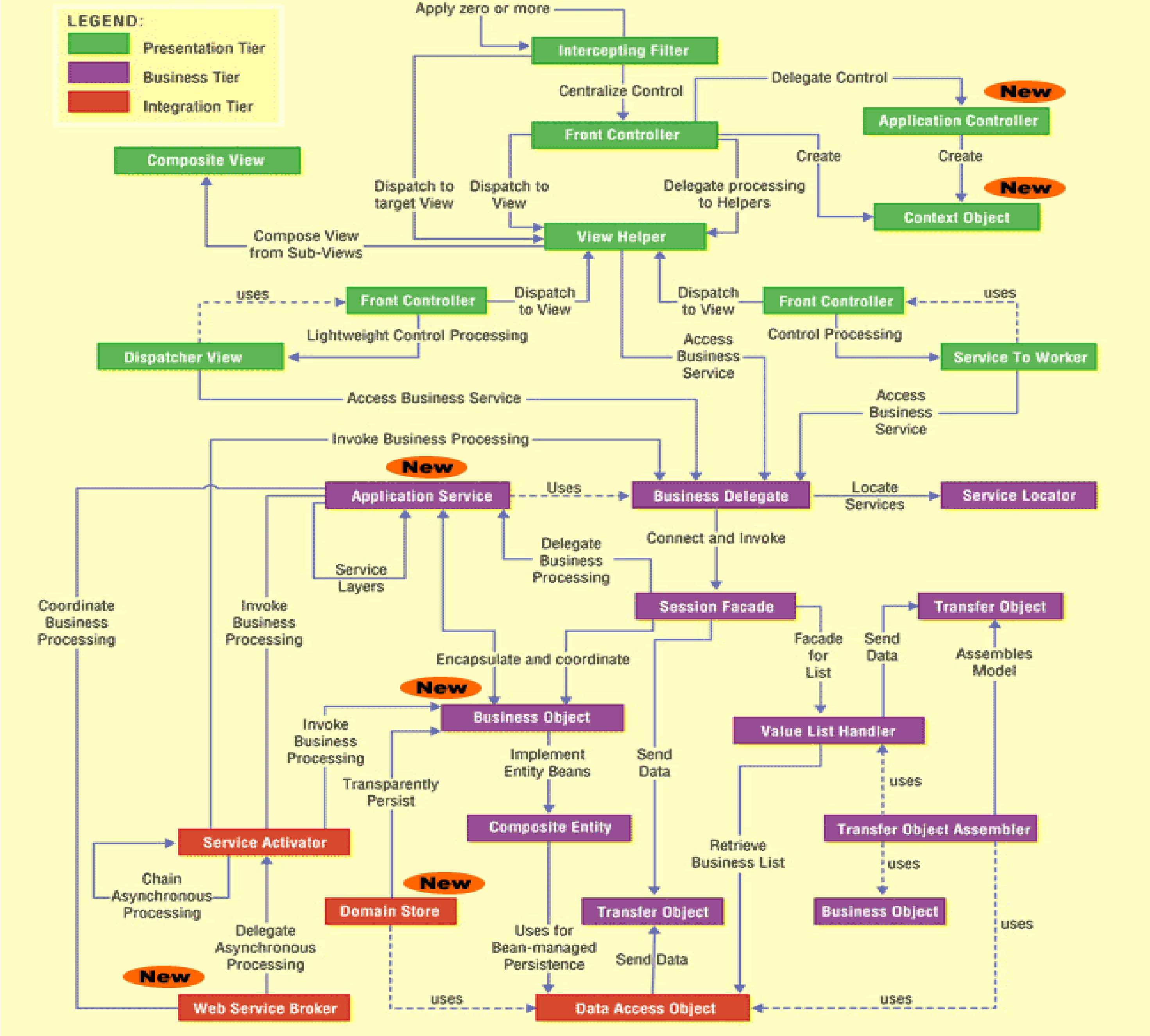
PATTERNS CLASSIFICATION

- Patterns applicable on presentation level

- Patterns applicable on business level

- Patterns applicable on integration level

Core J2EE Patterns, 2nd Edition



BUSINESS PATTERNS

- ❑ ***Business Delegate***
- ❑ **Service Locator**
- ❑ **Session Facade**
- ❑ Application Service
- ❑ Business Object
- ❑ Composite Entity
- ❑ Transfer Object
- ❑ Transfer Object Assembler
- ❑ Value List Handler

BUSINESS DELEGATE

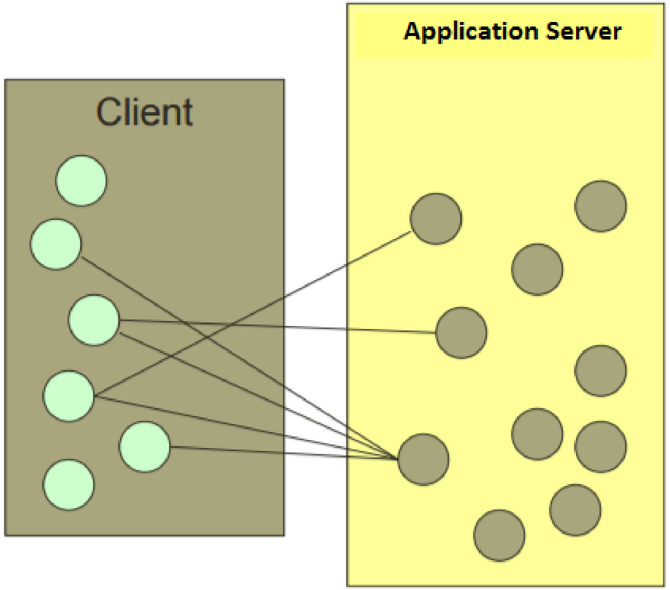
- **Problem**
  - You want to hide clients from the complexity of remote communication with business service components.
- **Forces**
  - You want to access the business-tier components from your presentation-tier components and clients, such as devices, web services, and rich clients.
  - You want to minimize coupling between clients and the business services, thus hiding the underlying implementation details of the service, such as lookup and access.
  - You want to avoid unnecessary invocation of remote services.
  - You want to translate network exceptions into application or user exceptions.
  - You want to hide the details of service creation, reconfiguration, and invocation retries from the clients

BUSINESS DELEGATE

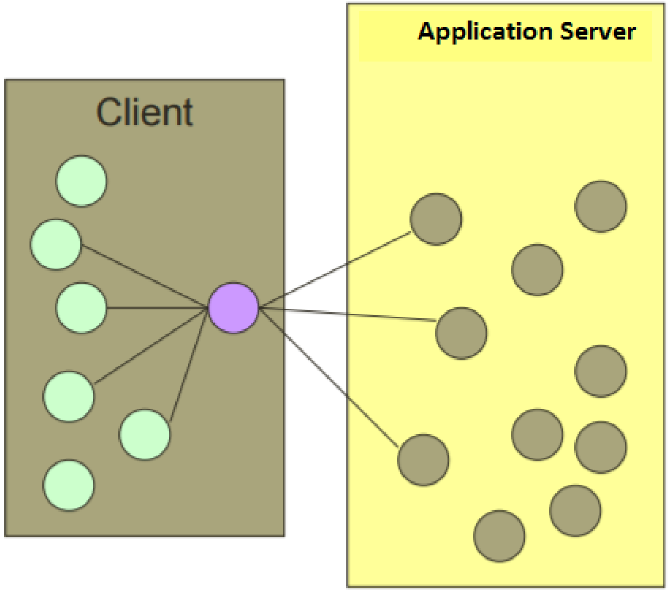
- **Solution**
  - Use a Business Delegate to encapsulate access to a business service.
  - The Business Delegate hides the implementation details of the business service, such as lookup and access mechanisms.

BUSINESS DELEGATE

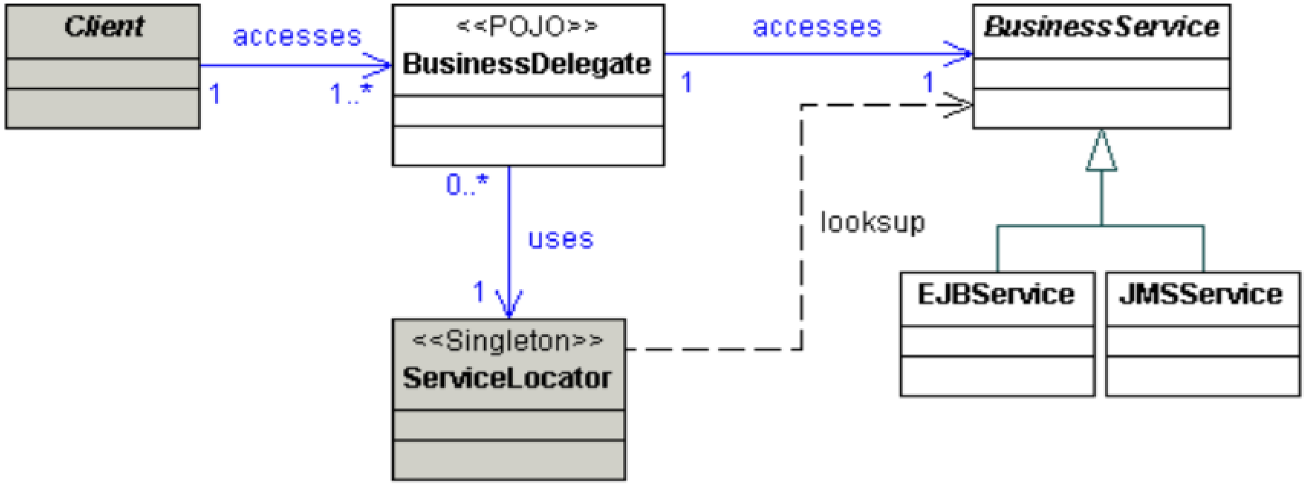
WITHOUT APPLAING THE PATTERN



AFTER BUSINESS DELEGATE PATTERN IS APPLIED

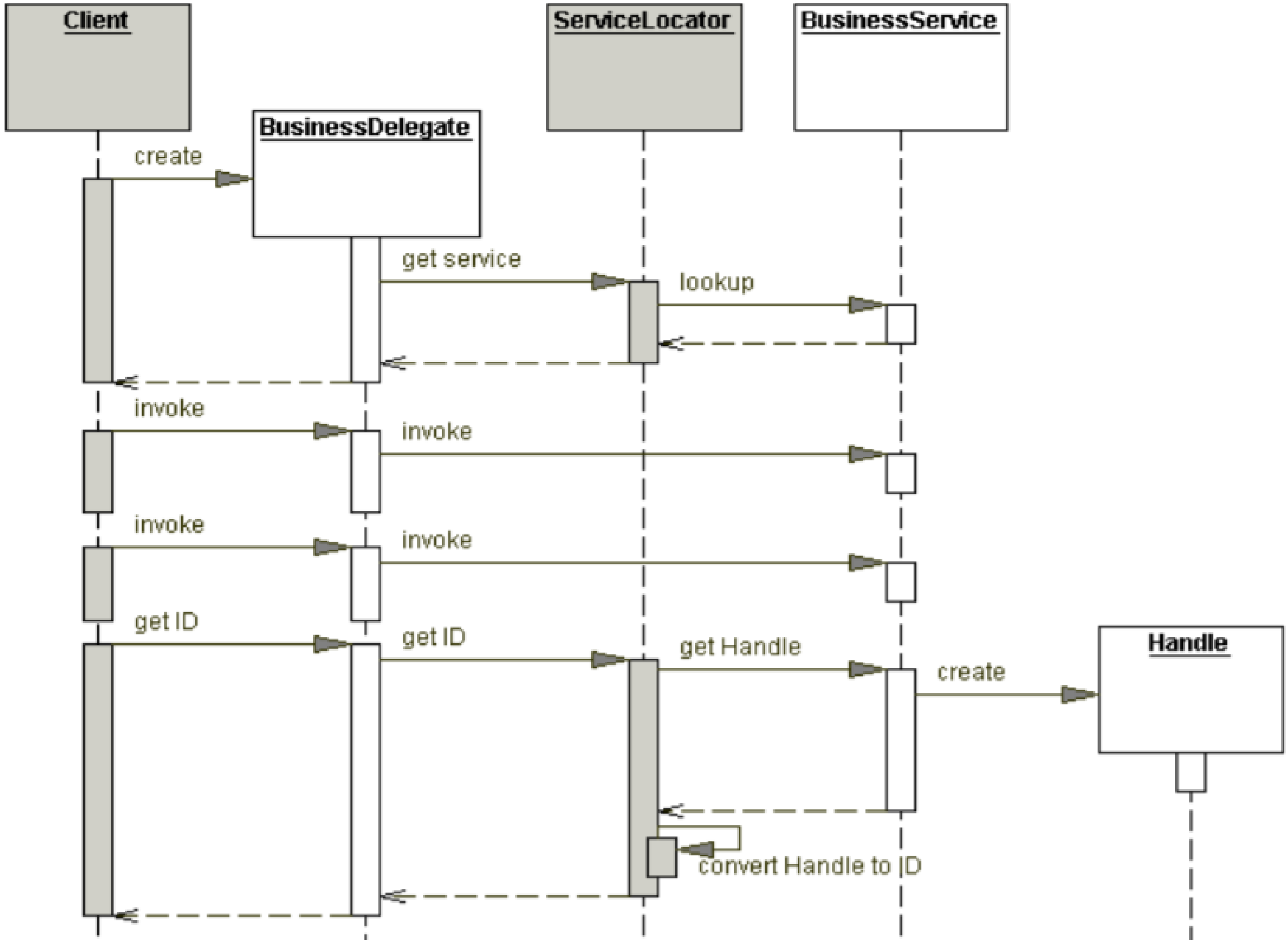


BUSINESS DELEGATE. STRUCTURE



- ❑ **Client**
  - ❑ Presentation tier code may be JSP, servlet or UI java code.
- ❑ **Business Delegate**
  - ❑ A single entry point class for client entities to provide access to Business Service methods.
- ❑ **LookUp Service**
  - ❑ Lookup service object is responsible to get relative business implementation and provide business object access to business delegate object.
- ❑ **Business Service**
  - ❑ Business Service interface. Concrete classes implement this business service to provide actual business implementation logic.

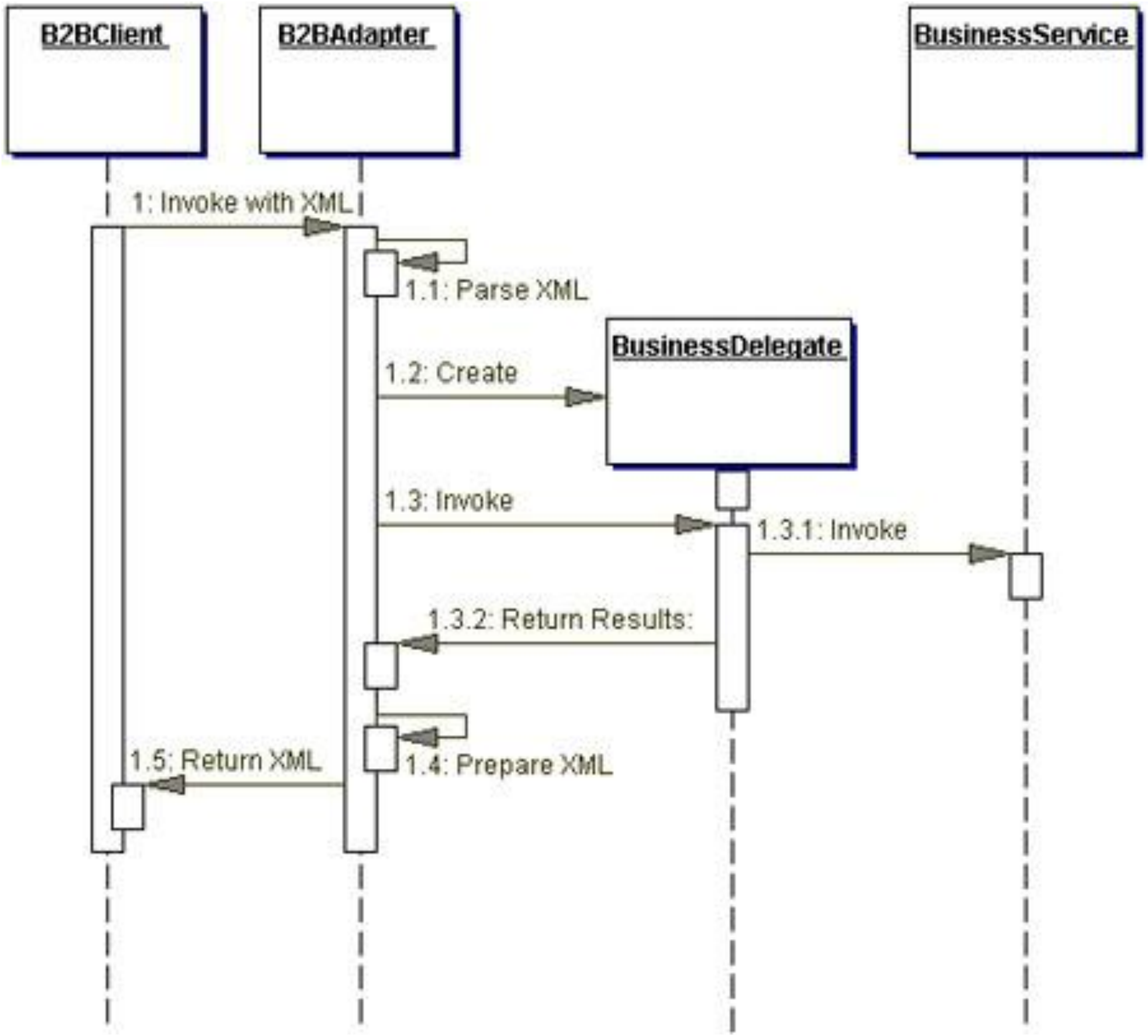
BUSINESS DELEGATE



BUSINESS DELEGATE

- **Implementation strategies**
  - Delegate Adapter
    - The Business Delegate proves to be a nice fit in a B2B environment when communicating with Java 2 Platform, Enterprise Edition (J2EE) based services.
    - Disparate systems may use an XML as the integration language.
    - Integrating one system to another typically requires an Adapter to meld the two disparate system
  - Delegate Proxy

BUSINESS DELEGATE. DELEGATE ADAPTER



BUSINESS DELEGATE

- **Implementation strategies**
  - Delegate Adapter
  - Delegate Proxy
    - The Business Delegate exposes an interface that provides clients access to the underlying methods of the business service API.
    - In this strategy, a Business Delegate provides proxy function to pass the client methods to the session bean it is encapsulating.
    - The Business Delegate may additionally cache any necessary data, including the remote references to the session bean's home or remote objects to improve performance by reducing the number of lookups.
    - The Business Delegate may also convert such references to String versions (IDs) and vice versa, using the services of a Service Locator.

BUSINESS DELEGATE

- **Implementation strategies**
  - Delegate Proxy
    - Business Delegate

```
public class LibraryDelegate {
   private BookDaoBase library;
    public LibraryDelegate()
                      throws ApplicationException {
         init();
     }
     public void init() throws ApplicationException {
      // Look up and obtain our session bean
     try {
              library = (BookDaoBase) ServiceLocator.getInstance().
              getInterface("BookDao/remote");
      } catch (ServiceLocatorException e) {
              throw new ApplicationException(e);
      }
 }
...
```

BUSINESS DELEGATE

- **Implementation strategies**
  - Delegate Proxy
    - Concrete service

```
....
public List<Book> getBooks()
            throws ApplicationException {
           return library.queryAll();
 }
public Book getBook(String isbn)
        throws  ApplicationException {
    try {
        return library.getBook(isbn);
     } catch (NoSuchBookException e) {
        new ApplicationException(e);
     }
}
...
```

BUSINESS DELEGATE

- **Consequences**
  - Reduces coupling, improves maintainability
  - Translates business service exceptions
  - Improves availability
  - Exposes a simpler, uniform interface to the business tier
  - Improves performance
  - Introduces an additional layer
  - Hides remoteness

BUSINESS DELEGATE

- **Related patterns**
  - Service Locator
  - Session Facade
  - Proxy
  - Adapter
  - Broker

BUSINESS PATTERNS

- **Business Delegate**
- ***Service Locator***
- **Session Facade**
- Application Service
- Business Object
- Composite Entity
- Transfer Object
- Transfer Object Assembler
- Value List Handler

SERVICE LOCATOR

- **Problem**
  - You want to transparently locate business components and services in a uniform manner.
- **Forces**
  - You want to use the JNDI API to look up and use business components, such as enterprise beans and JMS components, and services such as data sources.
  - You want to centralize and reuse the implementation of lookup mechanisms for J2EE application clients.
  - You want to encapsulate vendor dependencies for registry implementations, and hide the dependency and complexity from the clients.
  - You want to avoid performance overhead related to initial context creation and service lookups.
  - You want to reestablish a connection to a previously accessed enterprise bean instance, using its Handle object.

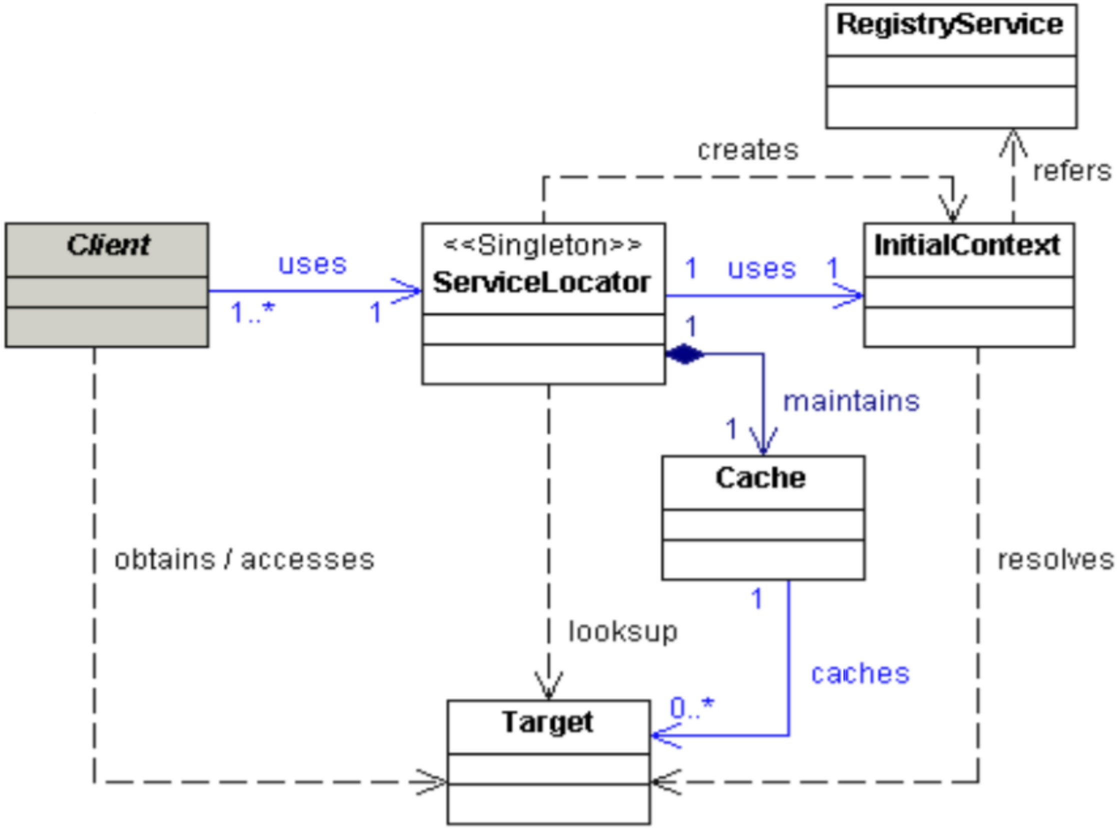
SERVICE LOCATOR

- **Solution**
  - 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.

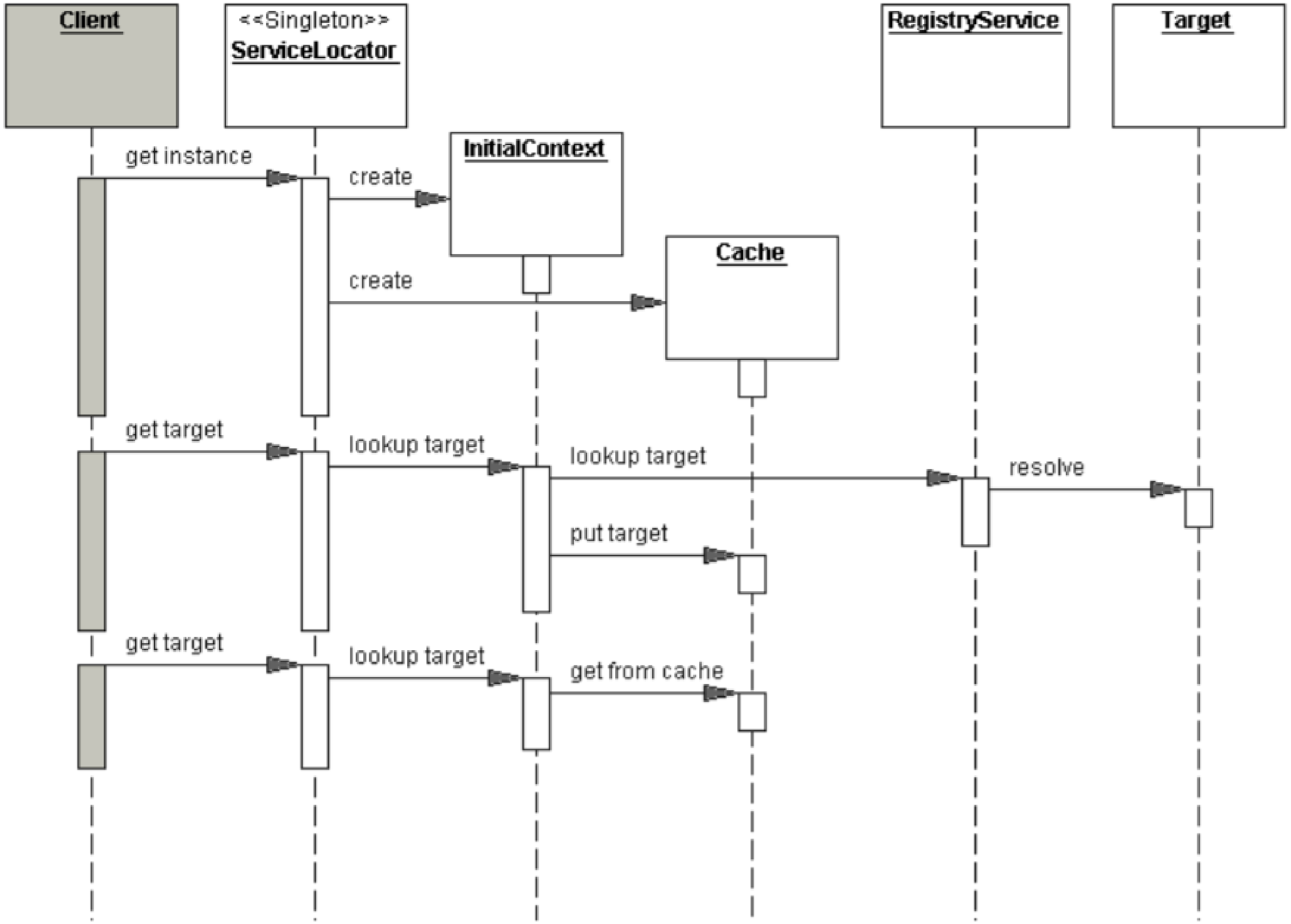
- **Used with**
  - Business Delegate
  - Session Facade
  - Transfer Object Assembler
  - Data Access Object

SERVICE LOCATOR. STRUCTURE

- ❑ **Service**
  - ❑ Actual Service which will process the request. Reference of such service is to be looked upon in JNDI server.
- ❑ **Context / Initial Context**
  - ❑ JNDI Context carries the reference to service used for lookup purpose.
- ❑ **Service Locator**
  - ❑ Service Locator is a single point of contact to get services by JNDI lookup caching the services.
- ❑ **Cache**
  - ❑ Cache to store references of services to reuse them
- ❑ **Client**
  - ❑ Client is the object that invokes the services via ServiceLocator.



SERVICE LOCATOR



SERVICE LOCATOR

- **Strategies**
  - EJB Service Locator
  - JMS Queue Service Locator
  - JMS Topic Service Locator
  - EJB şi JMS Service Locator

SERVICE LOCATOR. EXAMPLE

```
public class EntityManagerServiceLocator {
    private InitialContext initialContext;
    private Map<String, EntityManager> cache;
    private static EntityManagerServiceLocator _instance;
    static {
           try {
                _instance = new EntityManagerServiceLocator();
           } catch (ServiceLocatorException se) { }
    }
    private EntityManagerServiceLocator() throws ServiceLocatorException {
         try {
              initialContext = new InitialContext();
              cache = Collections.synchronizedMap(new HashMap<String, EntityManager>());
         } catch (NamingException ne) {  throw new ServiceLocatorException(ne.getMessage(), ne);
         } catch (Exception e) {  trow new ServiceLocatorException(e.getMessage(), e);  }
    }
    static public EntityManagerServiceLocator getInstance() {     return _instance;   }
```

SERVICE LOCATOR

- **Consequences**
  - Abstracts complexity
  - Provides uniform service access to clients
  - Facilitates adding EJB business components
  - Improves network performance
  - Improves client performance by caching

SERVICE LOCATOR

- **EJB 3.0 Depency Injection**
  - @Resource
  - @Ejb
  - It does not replace the JNDI mechanism, it just replace the way in witch a reference is obtain to JNDI
  - Example

```
 public class BookDao implements BookDaoRemote {
     @PersistenceContext(unitName = "libraryDS")
      private EntityManager em;
      public void delete(int id) {
      Book b = em.find(Book.class, new Long(id));
      em.remove(b);
  }
 ....
```

BUSINESS PATTERNS

- ❑ **Business Delegate**
- ❑ **Service Locator**
- ❑ ***Session Facade***
- ❑ Application Service
- ❑ Business Object
- ❑ Composite Entity
- ❑ Transfer Object
- ❑ Transfer Object Assembler
- ❑ Value List Handler

SESSION FACADE

- **Problem**
  - You want to expose business components and services to remote clients.

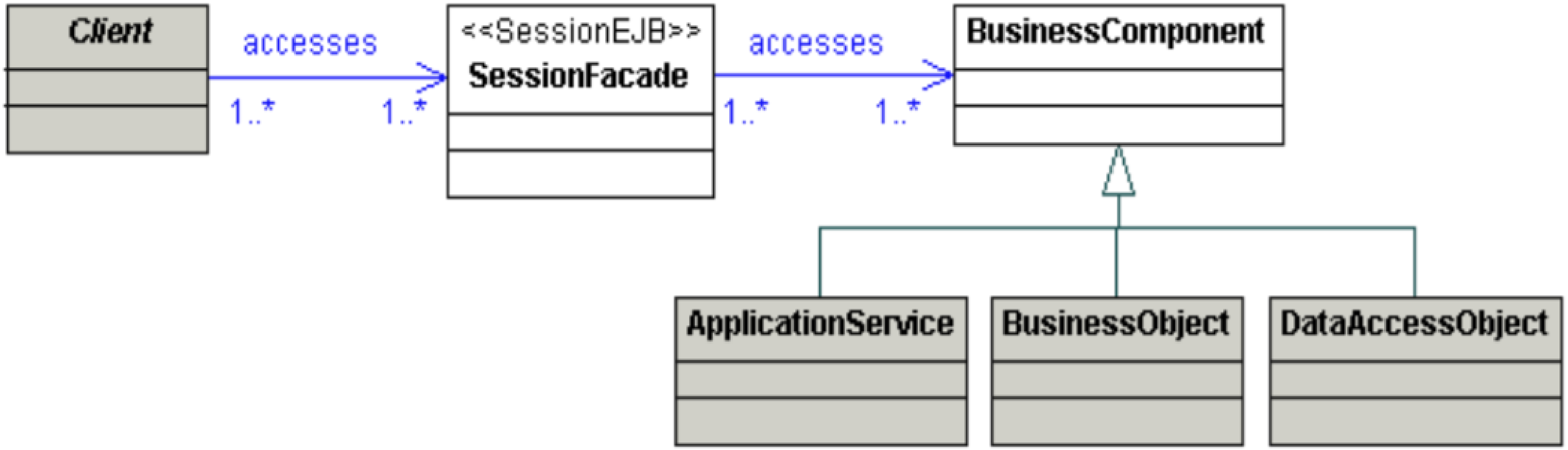
- **Forces**
  - You want to avoid giving clients direct access to business-tier components, to prevent tight coupling with the clients.
  - You want to provide a remote access layer to your Business Objects and other business-tier components.
  - You want to aggregate and expose your Application Services and other services to remote clients.
  - You want to centralize and aggregate all business logic that needs to be exposed to remote clients.
  - You want to hide the complex interactions and interdependencies between business components and services to improve manageability, centralize logic, increase flexibility, and improve ability to cope with changes.

SESSION FACADE

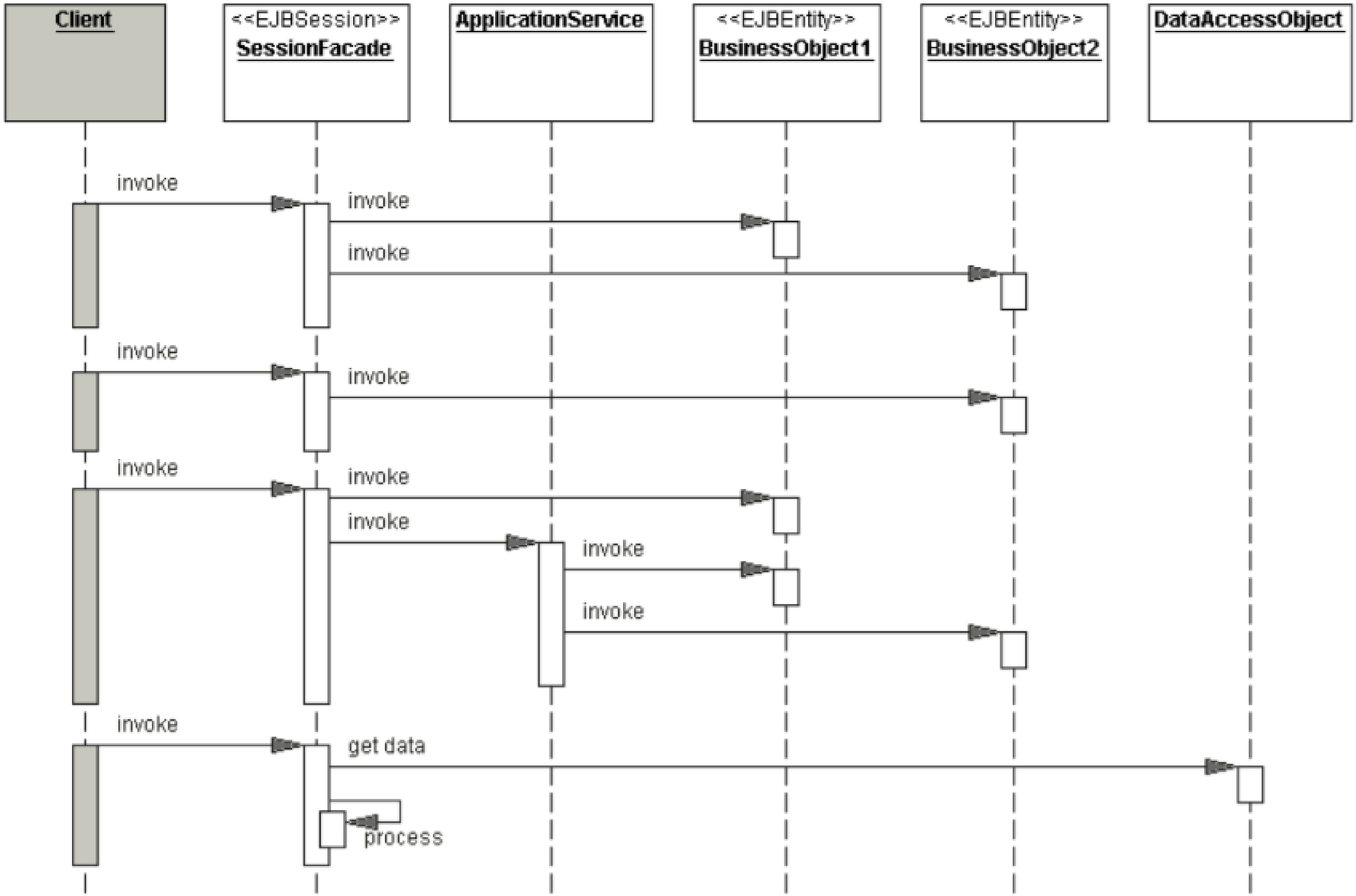
- **Solution**
  - Use a Session Façade to encapsulate business-tier components and expose a coarsegrained service to remote clients. Clients access a Session Façade instead of accessing business components directly.

- **Used with**
  - Business delegate
  - Business Object
  - Application Service
  - Data Acces Object
  - Service Locator
  - Broker
  - Facade

SESSION FACADE. STRUCTURE



SESSION FACADE



SESSION FACADE

- **Stategies**
  - Stateless session beans
    - A process that needs a single call to a business component
  - Stateful session beans
    - A business process that needs to maintain a conversation with multiple business components

SESSION FACADE

```
public class LibraryFacadeBean implements LibraryFacade {
    @EJB(beanName = "BookDao")
    private BookDaoRemote bookEntity;

     @EJB(beanName = "BookClientDao")
     private BookClientDaoRemote bookClientEntity;

     public boolean takeBook(final String isbn,
          final int clientId) throws Exception {
         boolean status = true;
         Book book = bookEntity.getBook(isbn);
         if (book != null && !book.isStatus()) {
            status = false;
            throw new Exception("The book is not available!");
         }
         if (bookClientEntity.numberOfBorrowedBooks(clientId) >
           Constants.MAX_NUMBER_OF_BOOKS_TO_BE_BORROWED) {
            status = false;
            throw new Exception ("The client has borrowed “ +
              “already the maximum amount of books"
               + Constants.MAX_NR_OF_BOOKS_TO_BE_BORROWED
               + "!");
         }
         book.setStatus(false);
         BookClientTO bc = new BookClientTO();
         bc.setBookId(book.getId());
         bc.setClientId(clientId);
         bc.setBorrowDate(new Date());
         bookClientEntity.insert(bc.translateToBookClient());
         return status;
     }
```

SESSION FACADE

- **Consequences**
  - Introduces a layer that provides services to remote clients
  - Exposes a uniform coarse-grained interface
  - Reduces coupling between the tiers
  - Promotes layering, increases flexibility and maintainability
  - Reduces complexity
  - Improves performance, reduces fine-grained remote methods
  - Centralizes security management
  - Centralizes transaction control
  - Exposes fewer remote interfaces to clients

NEXT COURSES

- Refactoring

- Anti-patterns