DESIGN PATTERNS

Course 10

PREVIOUS COURSE CONTENT

- Applications split on levels
- J2EE Design Patterns
- Intercepting Filters

CURRENT CURSE CONTENT

- Business Delegate
- Service Locator
- Session Facade

APPLICATIONS SPLIT ON LEVELS

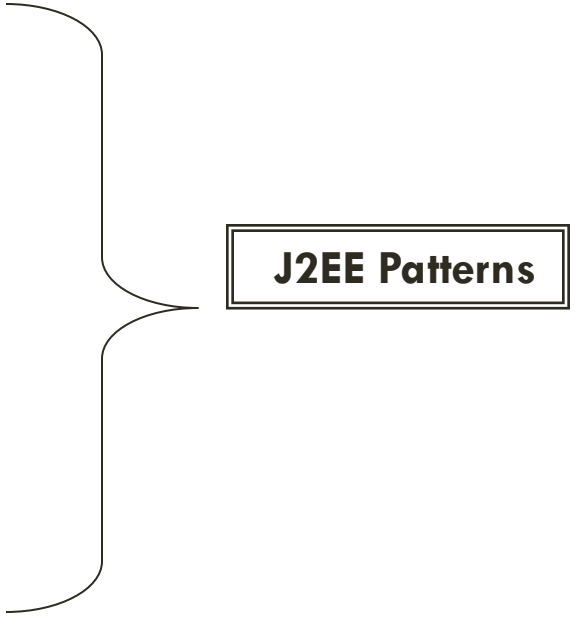
Client Level
Application clients, applets, others GUIs

Presentation level
JSP, Servlets and others UI elements

Business level
EJB and others business resources

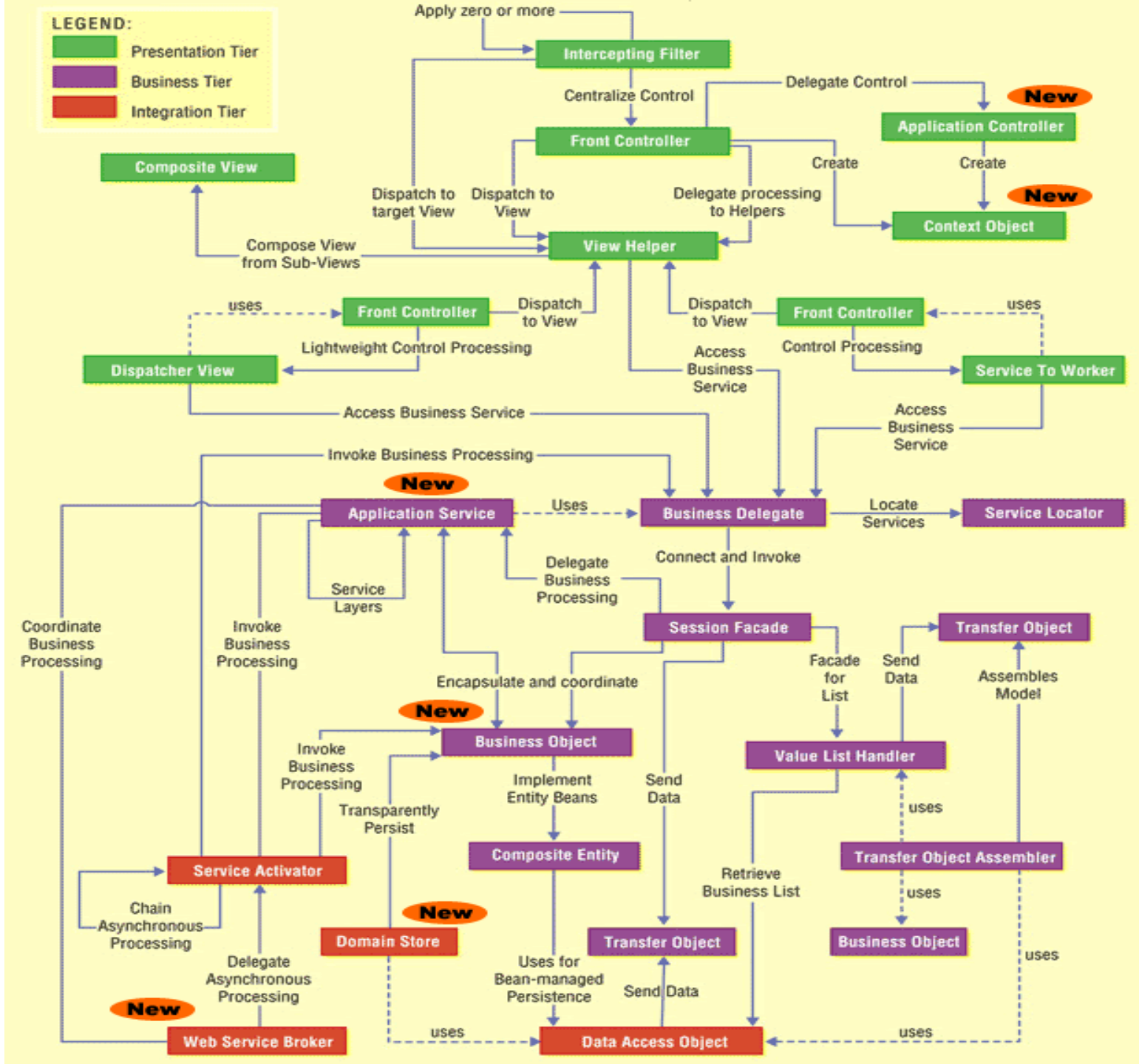
Integration level
JMS, JDBC, Connecters

Resource level
Data bases, external systems, resources



PATTERNS CLASSIFICATION

- Patterns applicable on presentation level
- Patterns applicable on business level
- Patterns applicable on integration level


LEGEND:
Presentation Tier
Business Tier
Integration Tier
Apply zero or more
Intercepting Filter
Centralize Control
Delegate Control
New
Application Controller
Front Controller
Create
Create
New
Context Object
Composite View
Dispatch to target View
Dispatch to View
Delegate processing to Helpers
Compose View from Sub-Views
View Helper
uses
Front Controller
Dispatch to View
Dispatch to View
Front Controller
uses
Lightweight Control Processing
Control Processing
Dispatcher View
Access Business Service
Service To Worker
Access Business Service
Access Business Service
Invoke Business Processing
New
Application Service
Uses
Business Delegate
Locate Services
Service Locator
Service Layers
Delegate Business Processing
Connect and Invoke
Coordinate Business Processing
Invoke Business Processing
Session Facade
Transfer Object
Facade for List
Send Data
Assembles Model
Encapsulate and coordinate
New
Business Object
Value List Handler
Invoke Business Processing
Implement Entity Beans
Send Data
uses
Transparently Persist
Composite Entity
Transfer Object Assembler
Service Activator
Retrieve Business List
Chain Asynchronous Processing
New
Domain Store
Transfer Object
Business Object
uses
Delegate Asynchronous Processing
Uses for Bean-managed Persistence
Send Data
uses
New
Web Service Broker
uses
Data Access Object
uses

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

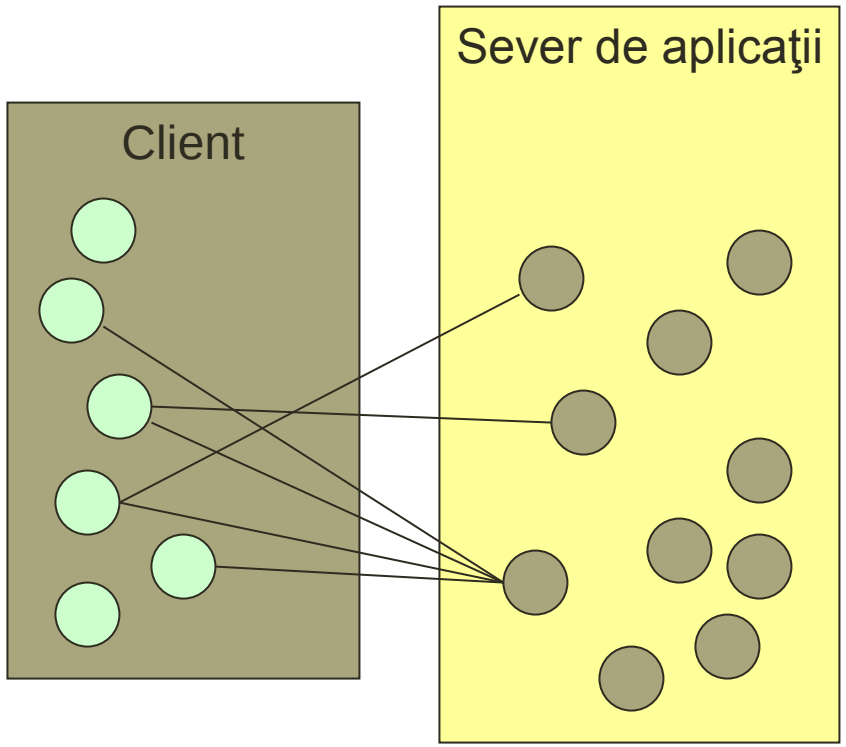
BUSINESS DELEGATE

Solution

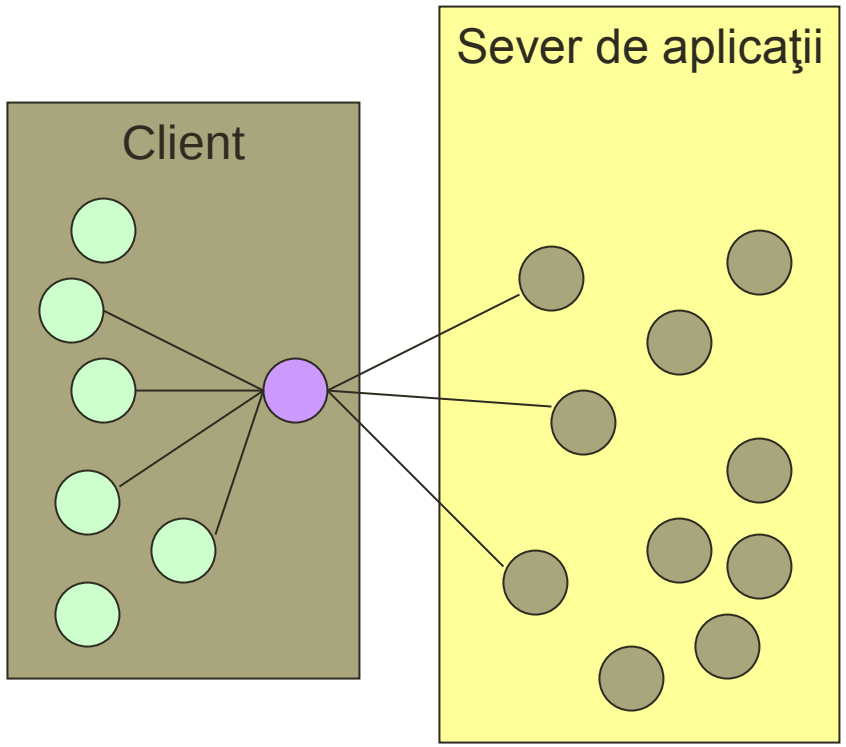
- Use a Business Delegate to encapsulate access to a business service.
- The Business Delegate hides the implementation details of the business service, such as lookup and access mechanisms.

BUSINESS DELEGATE

Without business delegate pattern

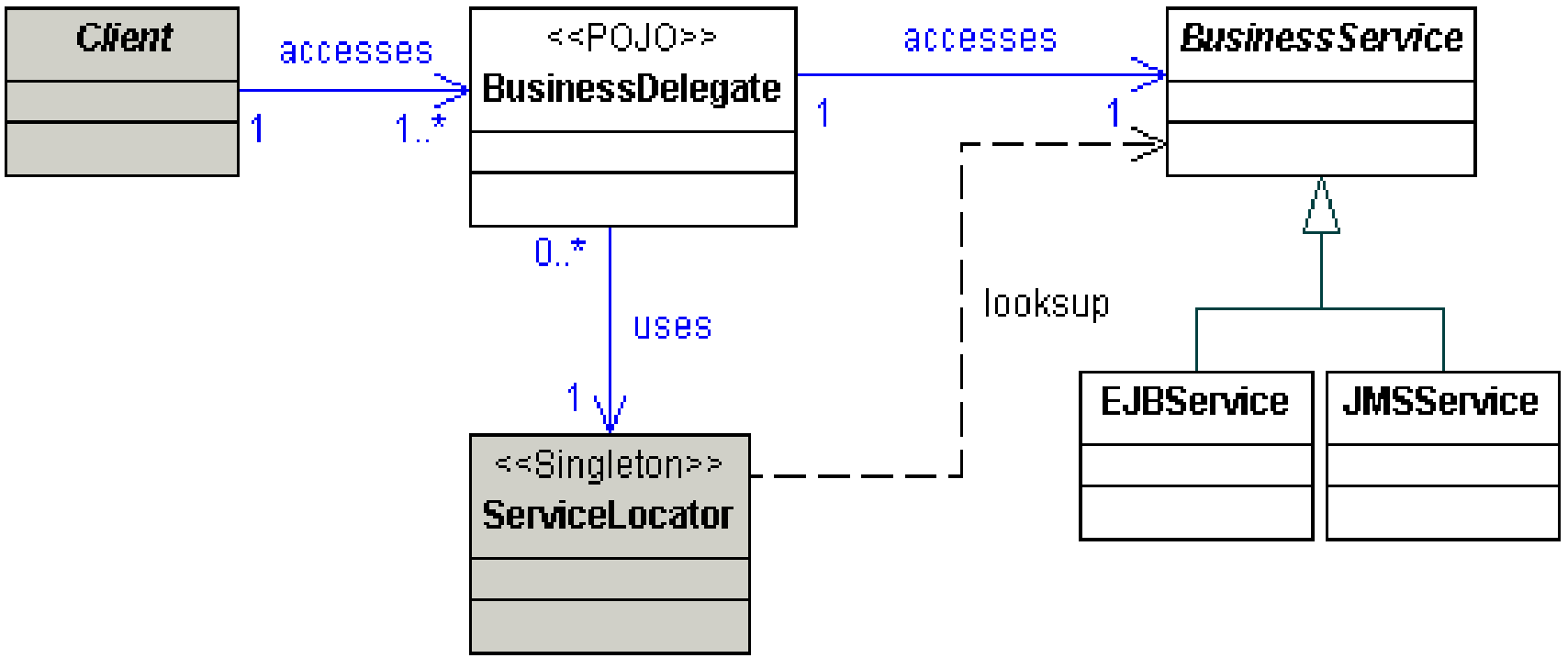


After applaying business delegate pattern



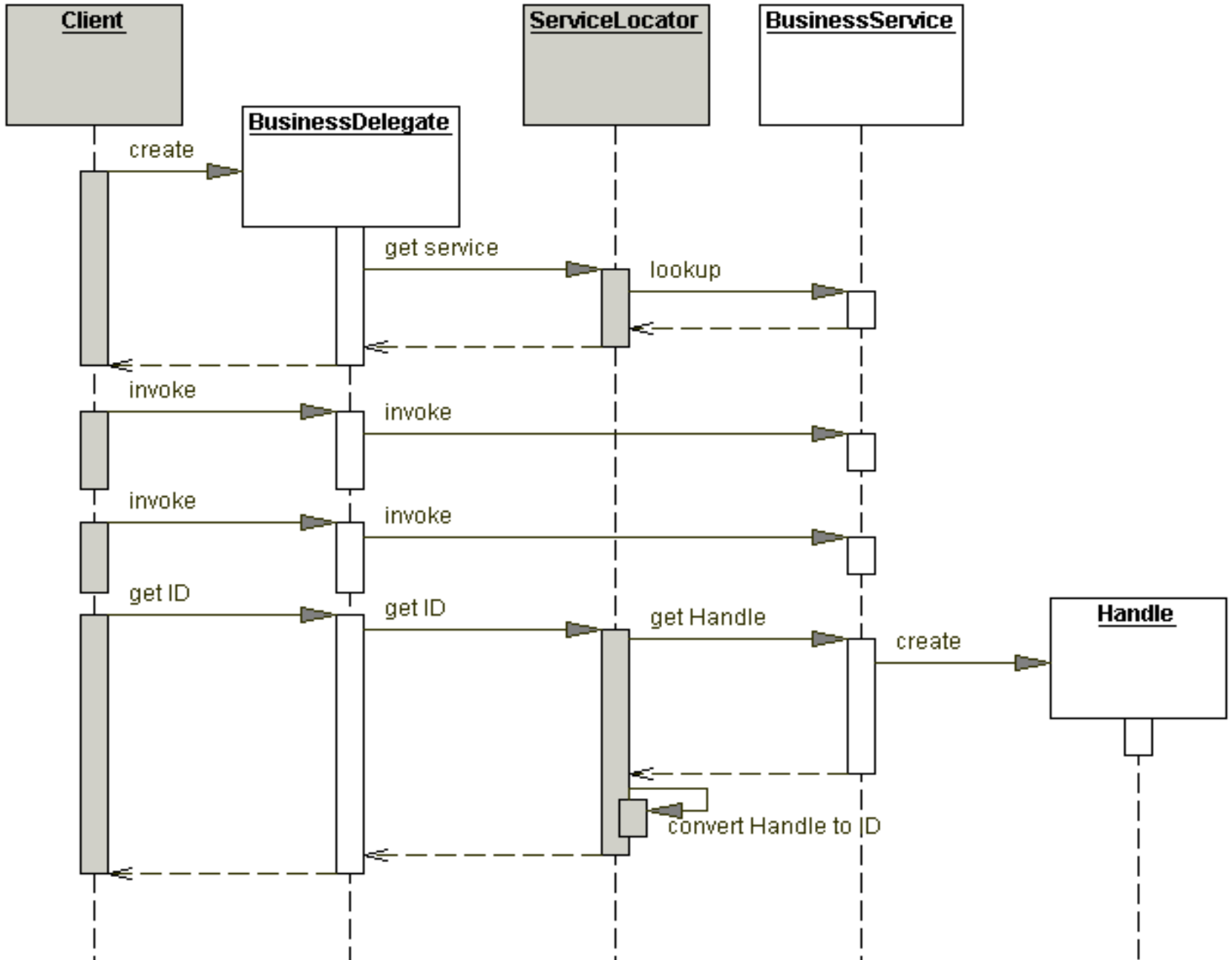
BUSINESS DELEGATE

Class diagram



BUSINESS DELEGATE

Sequence Diagram



BUSINESS DELEGATE

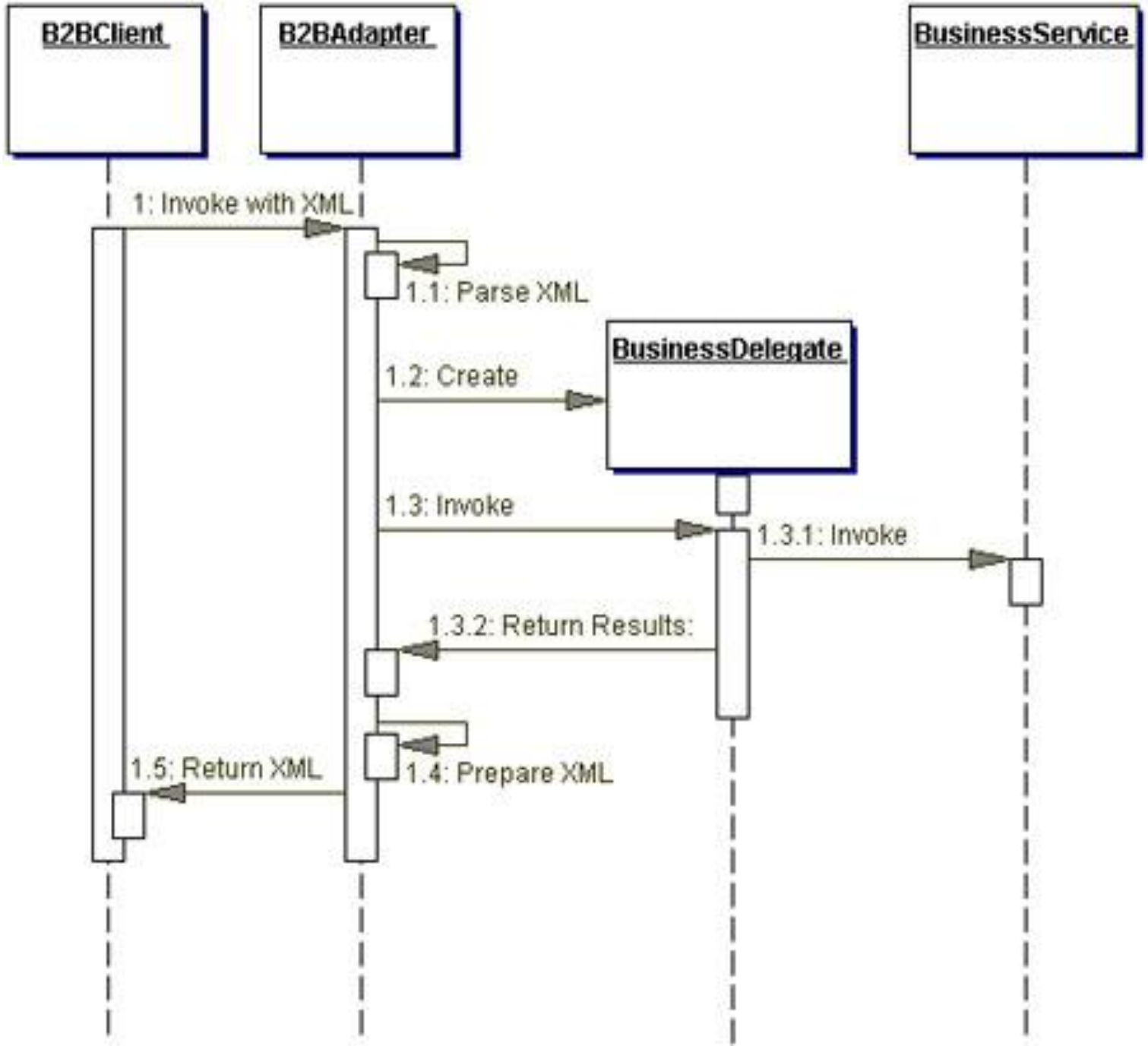
Implementation strategies

- 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

Implementation strategies

- 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

```
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

```
....

    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

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

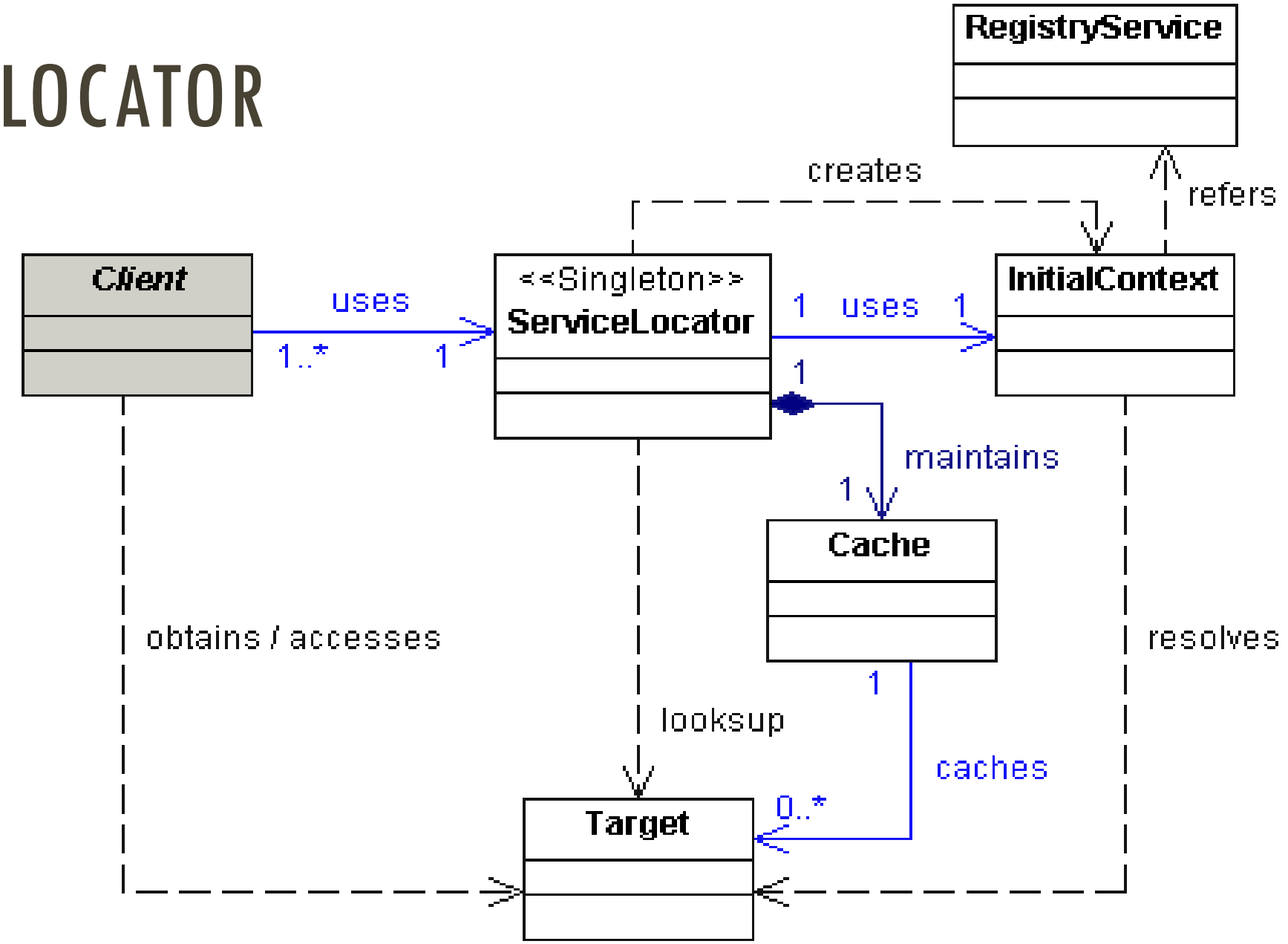
- Use a Service Locator to implement and encapsulate service and component lookup. A Service Locator hides the implementation details of the lookup mechanism and encapsulates related dependencies.

Used With

- Business Delegate
- Session Facade
- Transfer Object Assembler
- Data Access Object

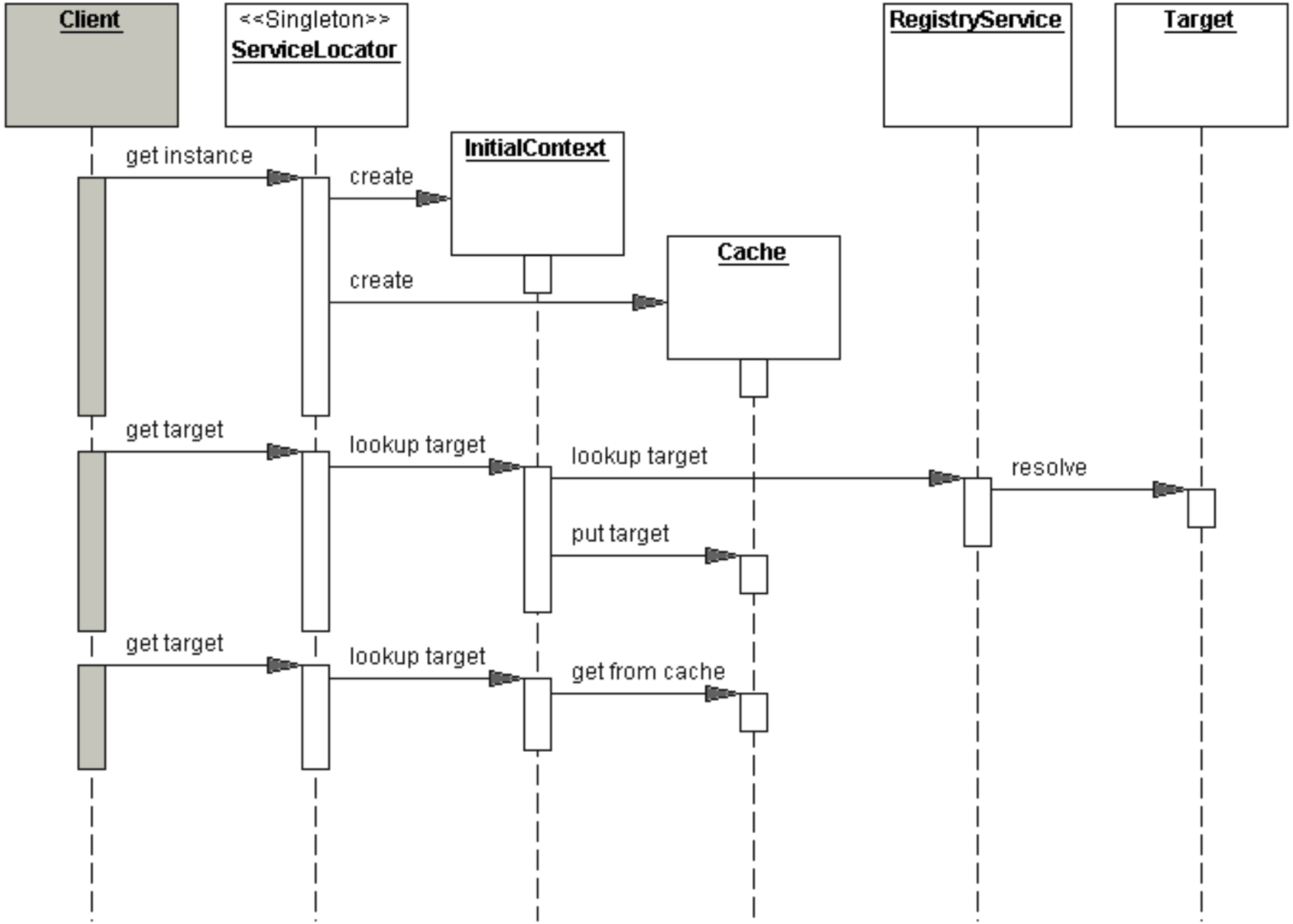
SERVICE LOCATOR

Class diagram



SERVICE LOCATOR

Sequence Diagram



SERVICE LOCATOR

Strategies

- EJB Service Locator
- JMS Queue Service Locator
- JMS Topic Service Locator
- EJB și JMS Service Locator

SERVICE LOCATOR

```
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) {
      throw new ServiceLocatorException(e.getMessage(), e);
    }
  }

  static public EntityManagerServiceLocator getInstance() {
     return _instance;
   }
```

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) {
        throw 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

```
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);
   }
....
```

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 (374) and other business-tier components.
- You want to aggregate and expose your Application Services (357) 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

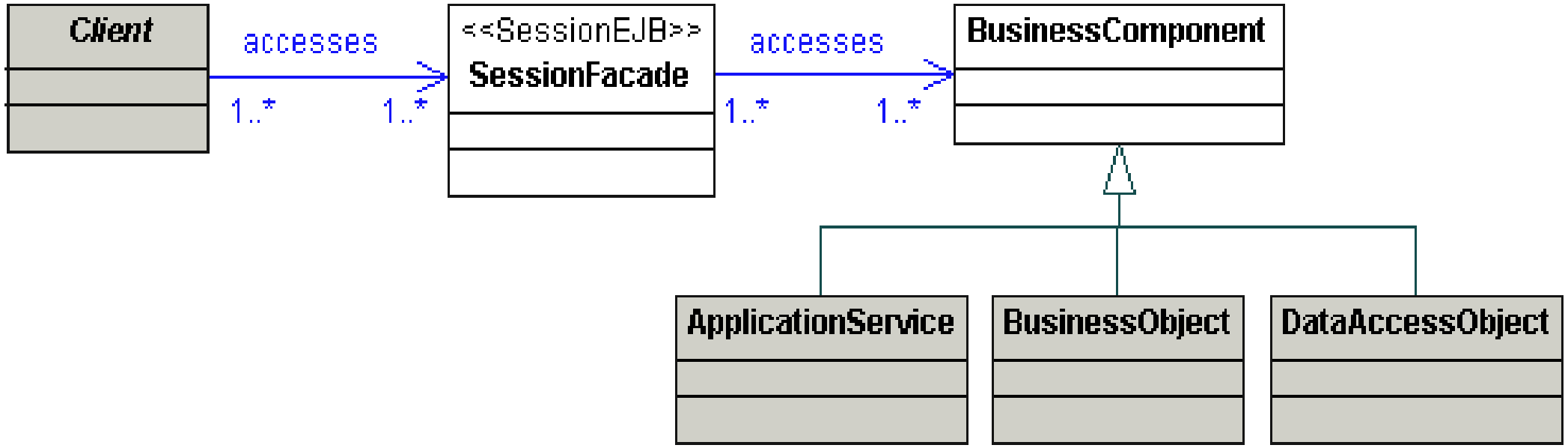
Use a *Session Façade* to encapsulate business-tier components and expose a coarse-grained service to remote clients. Clients access a *Session Façade* instead of accessing business components directly.

Used with

- Business delegate
- Business Object
- Application Service
- Data Acces Object
- Service Locator
- Broker
- Facade

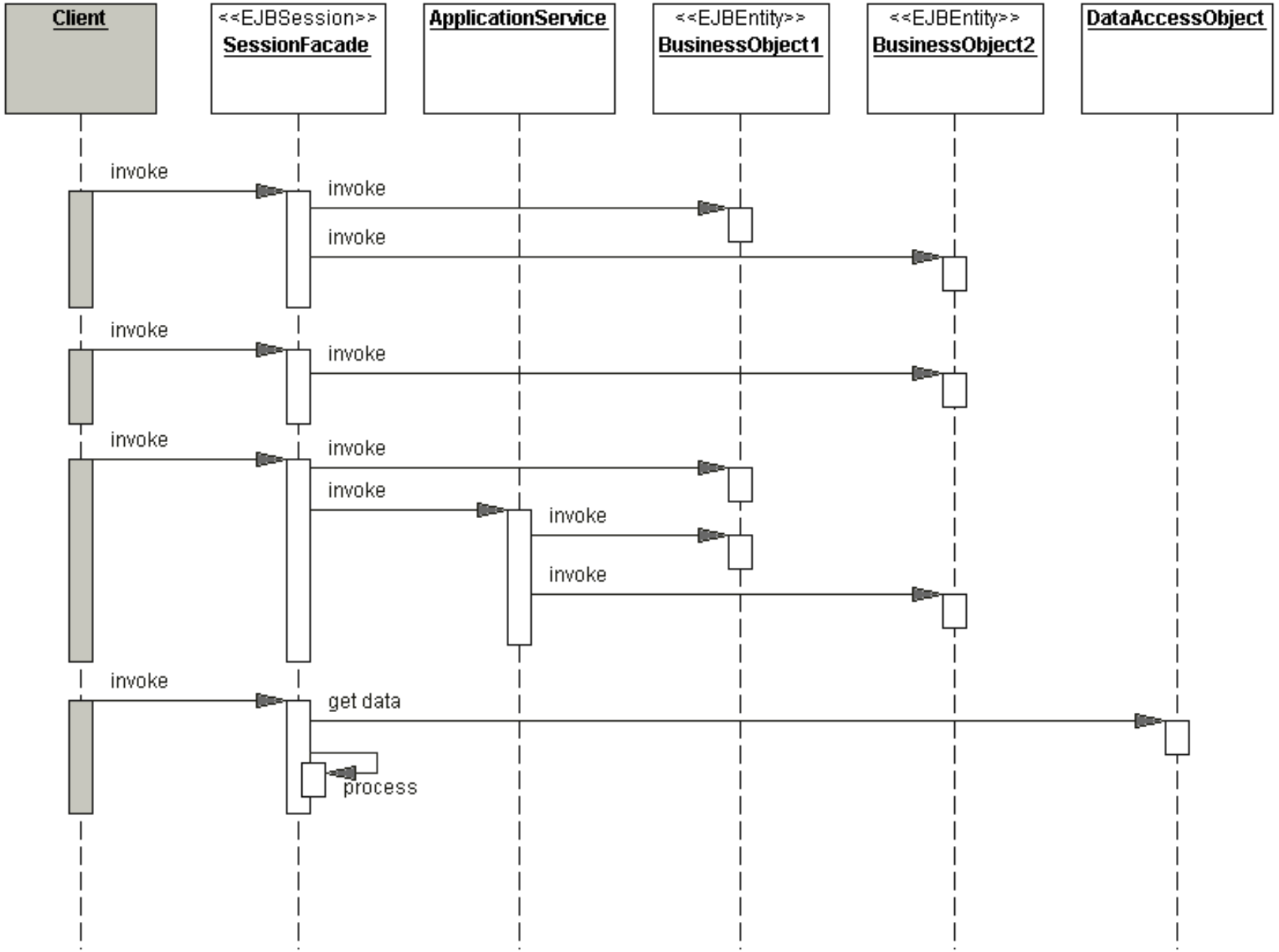
SESSION FACADE

Class diagram



SESSION FACADE

Sequence diagram



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_NUMBER_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

INTEGRATION PATTERNS

Data Access Object

Encapsulate data access and manipulation in a separate layer

Service Activator

Isolates the implementation of persistent storage.

Domain Store

Web Service Broker

Possible clients: Business Object, Session Facade, Application Service. Value List Handler, Transfer Object Assembler

INTEGRATION PATTERNS

Data Access Object

Service Activator

Domain Store

Web Service Broker

Invoke services asynchronously

Can be implemented like a JMS listener that accepts clients requests

INTEGRATION PATTERNS

Data Access Object

Service Activator

Domain Store

Web Service Broker

Separate persistence from object model

Stategies: Custom Persistence Strategy

INTEGRATION PATTERNS

Data Access Object

Service Activator

Domain Store

Web Service Broker

Provide access to one or more services using XML and web protocols