Core J2EE Patterns - Business Delegate

Context

A multi-tiered, distributed system requires remote method invocations to send and receive data across tiers. Clients are exposed to the complexity of dealing with distributed components.

Problem

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: When the implementation of the business services change, the exposed implementation code in the presentation tier must change too.

Additionally, there may be a detrimental impact on network performance because presentation-tier components that use the business service API make too many invocations over the network. This happens when presentation-tier components use the underlying API directly, with no client-side caching mechanism or aggregating service.

Lastly, exposing the service APIs directly to the client forces the client to deal with the networking issues associated with the distributed nature of Enterprise JavaBeans (EJB) technology.

Forces

Presentation-tier clients need access to business services.
Different clients, such as devices, Web clients, and thick clients, need access to business service.
Business services APIs may change as business requirements evolve.
It is desirable to minimize coupling between presentation-tier clients and the business service, thus hiding the underlying implementation details of the service, such as lookup and access.
Clients may need to implement caching mechanisms for business service information.
It is desirable to reduce network traffic between client and business services.

Solution

Use a Business Delegate to reduce coupling between presentation-tier clients and business services. The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.

The Business Delegate acts as a client-side business abstraction; it provides an abstraction for, and thus hides, the implementation of the business services. Using a Business Delegate reduces the coupling between presentation-tier clients and the system's business services. Depending on the implementation strategy, the Business Delegate may shield clients from possible volatility in the implementation of the business service API. Potentially, this reduces the number of changes that must be made to the presentation-tier client code when the business service API or its underlying implementation changes.

However, interface methods in the Business Delegate may still require modification if the underlying business service API changes. Admittedly, though, it is more likely that changes will be made to the business service rather than to the Business Delegate.

Often, developers are skeptical when a design goal such as abstracting the business layer causes additional upfront work in return for future gains. However, using this pattern or its strategies results in only a small amount of additional upfront work and provides considerable benefits. The main benefit is hiding the details of the underlying service. For example, the client can become transparent to naming and lookup services. The Business Delegate also handles the exceptions from the business services, such as java.rmi.Remote exceptions, Java Messages Service (JMS) exceptions and so on. The Business Delegate may intercept such service level exceptions and generate application level exceptions instead. Application level exceptions are easier to handle by the clients, and may be user friendly. The Business Delegate may also transparently perform any retry or recovery operations necessary in the event of a service failure without exposing the client to the problem until it is determined that the problem is not resolvable. These gains present a compelling reason to use the pattern.

Another benefit is that the delegate may cache results and references to remote business services. Caching can significantly improve performance, because it limits unnecessary and potentially costly round trips over the network.

A Business Delegate uses a component called the Lookup Service. The Lookup Service is responsible for hiding the underlying implementation details of the business service lookup code. The Lookup Service may be written as part of the Delegate, but we recommend that it be implemented as a separate component, as outlined in the Service Locator pattern (See "Service Locator" on page 368.)

When the Business Delegate is used with a Session Facade, typically there is a one-to-one relationship between the two. This one-to-one relationship exists because logic that might have been encapsulated in a Business Delegate relating to its interaction with multiple business services (creating a one-to-many relationship) will often be factored back into a Session Facade.

Finally, it should be noted that this pattern could be used to reduce coupling between other tiers, not simply the presentation and the business tiers.

Structure

Figure 8.1 shows the class diagram representing the Business Delegate pattern. The client requests the BusinessDelegate to provide access to the underlying business service. The BusinessDelegate uses a LookupService to locate the required BusinessService component.

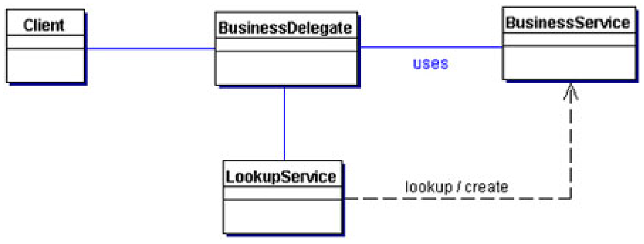

Figure 8.1 BusinessDelegate class diagram

Participants and Responsibilities

Figure 8.2 and Figure 8.3 show sequence diagrams that illustrate typical interactions for the Business Delegate pattern.

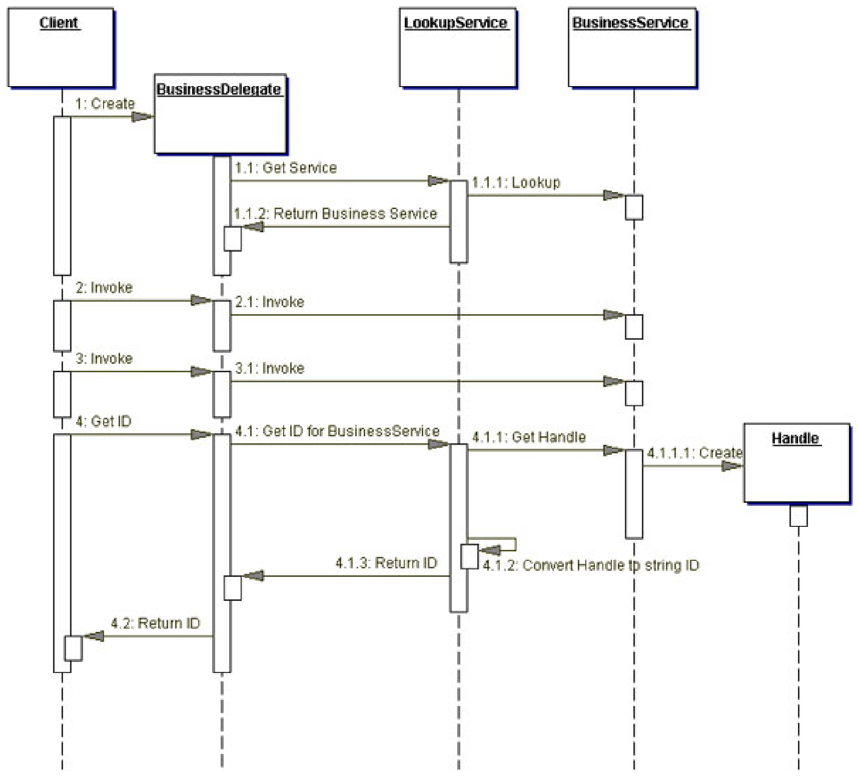

Figure 8.2 BusinessDelegate sequence diagram

The BusinessDelegate uses a LookupService for locating the business service. The business service is used to invoke the business methods on behalf of the client. The Get ID method shows that the BusinessDelegate can obtain a String version of the handle (such as EJBHandle object) for the business service and return it to the client as a String. The client can use the String version of the handle at a later time to reconnect to the business service it was using when it obtained the handle. This technique will avoid new lookups, since the handle is capable of reconnecting to its business service instance. It should be noted that handle objects are implemented by the container provider and may not be portable across containers from different vendors.

The sequence diagram in Figure 8.3 shows obtaining a BusinessService (such as a session or an entity bean) using its handle.

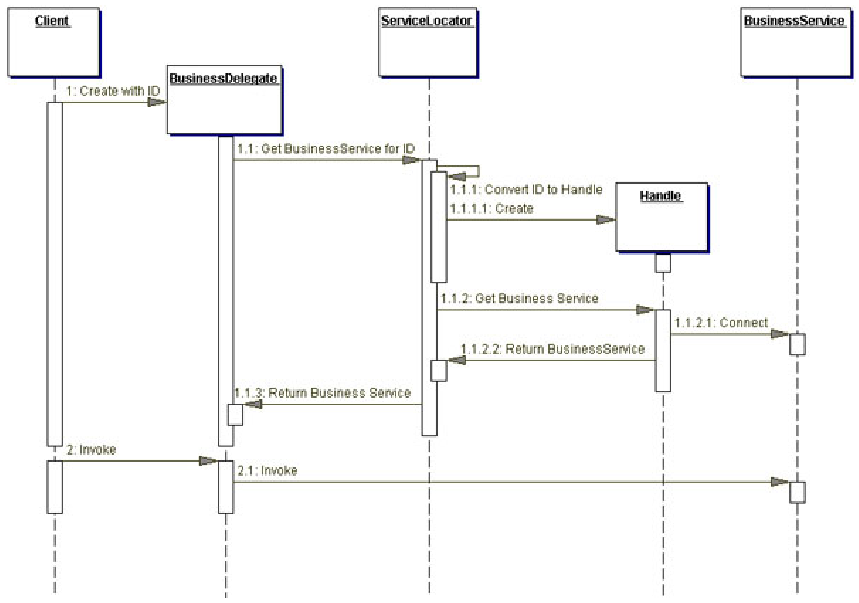
Client
BusinessDelegate
ServiceLocator
BusinessService
Handle
1: Create with ID
1.1: Get BusinessService for ID
1.1.1: Convert ID to Handle
1.1.1.1: Create
1.1.2: Get Business Service
1.1.2.1: Connect
1.1.2.2: Return BusinessService
1.1.3: Return Business Service
2: Invoke
2.1: Invoke

Figure 8.3 BusinessDelegate with ID sequence diagram

BusinessDelegate

The BusinessDelegate's role is to provide control and protection for the business service. The BusinessDelegate can expose two types of constructors to clients. One type of request instantiates the BusinessDelegate without an ID, while the other instantiates it with an ID, where ID is a String version of the reference to a remote object, such as EJBHome or EJBObject.

When initialized without an ID, the BusinessDelegate requests the service from the Lookup Service, typically implemented as a Service Locator (see "Service Locator" on page 368), which returns the Service Factory, such as EJBHome. The BusinessDelegate requests that the Service Factory locate, create, or remove a BusinessService, such as an enterprise bean.

When initialized with an ID string, the BusinessDelegate uses the ID string to reconnect to the BusinessService. Thus, the BusinessDelegate shields the client from the underlying implementation details of BusinessService naming and lookup. Furthermore, the presentation-tier client never directly makes a remote invocation on a BusinessSession; instead, the client uses the BusinessDelegate.

LookupService

The BusinessDelegate uses the LookupService to locate the BusinessService. The LookupService encapsulates the implementation details of BusinessService lookup.

BusinessService

The BusinessService is a business-tier component, such as an enterprise bean or a JMS component, that provides the required service to the client.

Strategies

Delegate Proxy Strategy

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.

The example implementation for this strategy is discussed in the "Sample Code" section of this pattern.

Delegate Adapter Strategy

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 [GoF] to meld the two disparate systems. Figure 8.4 gives an example.

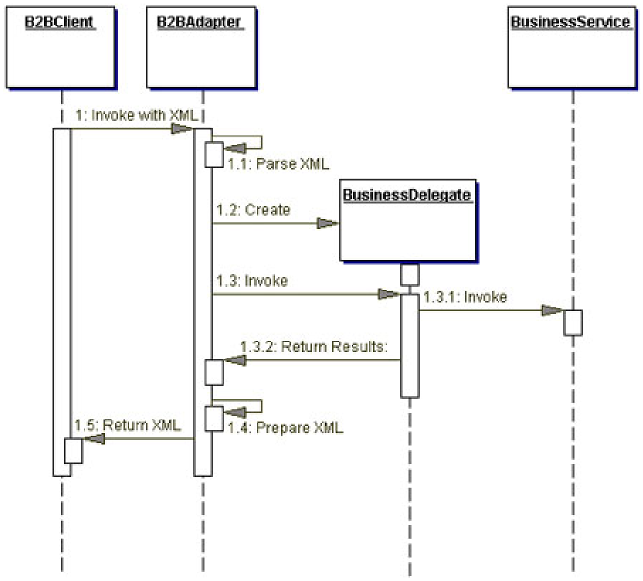

Figure 8.4 Using the Business Delegate pattern with an Adapter strategy

Consequences

Reduces Coupling, Improves Manageability

The Business Delegate reduces coupling between the presentation tier and the business tier by hiding all business-tier implementation details. It is easier to manage changes because they are centralized in one place, the Business Delegate.

Translates Business Service Exceptions

The Business Delegate is responsible for translating any network or infrastructure-related exceptions into business exceptions, shielding clients from knowledge of the underlying implementation specifics.

Implements Failure Recovery and Thread Synchronization

The Business Delegate on encountering a business service failure, may implement automatic recovery features without exposing the problem to the client. If the recovery succeeds, the client need not know about the failure. If the recovery attempt does not succeed, then the Business Delegate needs to inform the client of the failure. Additionally, the business delegate methods may be synchronized, if necessary.

Exposes Simpler, Uniform Interface to Business Tier

The Business Delegate, to better serve its clients, may provide a variant of the interface provided by the underlying enterprise beans.

Impacts Performance

The Business Delegate may provide caching services (and better performance) to the presentation tier for common service requests.

Introduces Additional Layer

The Business Delegate may be seen as adding an unnecessary layer between the client and the service, thus introducing added complexity and decreasing flexibility. Some developers may feel that it is an extra effort to develop Business Delegates with implementations that use the Delegate Proxy strategy. At the same time, the benefits of the pattern typically outweigh such drawbacks.

Hides Remoteness

While location transparency is one of the benefits of this pattern, a different problem may arise due to the developer treating a remote service as if it was a local one. This may happen if the client developer does not understand that the Business Delegate is a client side proxy to a remote service. Typically, a method invocations on the Business Delegate results in a remote method invocation under the wraps. Ignoring this, the developer may tend to make numerous method invocations to perform a single task, thus increasing the network traffic.

Sample Code

Implementing the Business Delegate Pattern

Consider a Professional Services Application (PSA), where a Web-tier client needs to access a session bean that implements the Session Facade pattern. The Business Delegate pattern can be applied to design a Delegate class ResourceDelegate, which encapsulates the complexity of dealing with the session bean ResourceSession. The ResourceDelegate implementation for this example is shown in Example 8.1, and the corresponding remote interface for the Session Facade bean ResourceSession is shown in Example 8.2.

Example 8.1 Implementing Business Delegate Pattern - ResourceDelegate

```
// imports
...

public class ResourceDelegate {

  // Remote reference for Session Facade
  private ResourceSession session;

  // Class for Session Facade's Home object
  private static final Class homeClazz =
  corepatterns.apps.psa.ejb.ResourceSessionHome.class;

  // Default Constructor. Looks up home and connects
  // to session by creating a new one
  public ResourceDelegate() throws ResourceException {
    try {
      ResourceSessionHome home = (ResourceSessionHome)
        ServiceLocator.getInstance().getHome(
          "Resource", homeClazz);
      session = home.create();
    } catch(ServiceLocatorException ex) {
      // Translate Service Locator exception into
      // application exception
      throw new ResourceException(...);
    } catch(CreateException ex) {
      // Translate the Session create exception into
      // application exception
      throw new ResourceException(...);
    } catch(RemoteException ex) {
      // Translate the Remote exception into
      // application exception
      throw new ResourceException(...);
    }
  }

  // Constructor that accepts an ID (Handle id) and
  // reconnects to the prior session bean instead
  // of creating a new one
  public BusinessDelegate(String id)
    throws ResourceException {
    super();
    reconnect(id);
  }

  // Returns a String ID the client can use at a
  // later time to reconnect to the session bean
  public String getID() {
    try {
      return ServiceLocator.getId(session);
    } catch (Exception e) {
      // Throw an application exception
    }
 }

  // method to reconnect using String ID
  public void reconnect(String id)
    throws ResourceException {
    try {
      session = (ResourceSession)
                ServiceLocator.getService(id);
    } catch (RemoteException ex) {
      // Translate the Remote exception into
      // application exception
      throw new ResourceException(...);
    }
  }

  // The following are the business methods
  // proxied to the Session Facade. If any service
  // exception is encountered, these methods convert
  // them into application exceptions such as
  // ResourceException, SkillSetException, and so
  // forth.

  public ResourceTO setCurrentResource(
    String resourceId)
    throws ResourceException {
    try {
      return session.setCurrentResource(resourceId);
    } catch (RemoteException ex) {
      // Translate the service exception into
      // application exception
      throw new ResourceException(...);
    }
  }

  public ResourceTO getResourceDetails()
    throws ResourceException {

    try {
      return session.getResourceDetails();
    } catch(RemoteException ex) {
      // Translate the service exception into
      // application exception
      throw new ResourceException(...);
    }
  }

  public void setResourceDetails(ResourceTO vo)
    throws ResourceException {
    try {
      session.setResourceDetails(vo);
    } catch(RemoteException ex) {
      throw new ResourceException(...);
    }
  }

  public void addNewResource(ResourceTO vo)
    throws ResourceException {
    try {
      session.addResource(vo);
    } catch(RemoteException ex) {
      throw new ResourceException(...);
    }
  }

  // all other proxy method to session bean
  ...
}
```

Example 8.2 Remote Interface for ResourceSession

```
// imports
...
public interface ResourceSession extends EJBObject {

  public ResourceTO setCurrentResource(
    String resourceId) throws
    RemoteException, ResourceException;

  public ResourceTO getResourceDetails()
     throws RemoteException, ResourceException;

  public void setResourceDetails(ResourceTO resource)
     throws RemoteException, ResourceException;

  public void addResource(ResourceTO resource)
      throws RemoteException, ResourceException;

  public void removeResource()
      throws RemoteException, ResourceException;

  // methods for managing blockout time by the
  // resource
  public void addBlockoutTime(Collection blockoutTime)
      throws RemoteException, BlockoutTimeException;

  public void updateBlockoutTime(
    Collection blockoutTime)
      throws RemoteException, BlockoutTimeException;

  public void removeBlockoutTime(
    Collection blockoutTime)
      throws RemoteException, BlockoutTimeException;

  public void removeAllBlockoutTime()
      throws RemoteException, BlockoutTimeException;

  // methods for resource skillsets time by the
  //resource
  public void addSkillSets(Collection skillSet)
      throws RemoteException, SkillSetException;

  public void updateSkillSets(Collection skillSet)
      throws RemoteException, SkillSetException;

  public void removeSkillSet(Collection skillSet)
      throws RemoteException, SkillSetException;

  ...
}
```

Related Patterns

Service Locator
The Service Locator pattern may be used to create the Business Delegate's Service Locator, hiding the implementation details of any business service lookup and access code.
Proxy [GoF]
A Business Delegate may act as a proxy, providing a stand-in for objects in the business tier.
Adapter [GoF]
A Business Delegate may use the Adapter pattern to provide coupling for disparate systems.
Broker [POSA1]
A Business Delegate performs the role of a broker to decouple the business tier objects from the clients in other tiers.

Java SDKs and Tools

Java SE

Java EE and Glassfish

Java ME

Java Card

NetBeans IDE

Java Mission Control

Java Resources

Java APIs

Technical Articles

Demos and Videos

Forums

Java Magazine

Java.net

Developer Training

Tutorials

Java.com

E-mail this page Printer View

ORACLE CLOUD
Learn About Oracle Cloud Computing
Get a Free Trial
Learn About DaaS
Learn About SaaS
Learn About PaaS
Learn About IaaS
Learn About Private Cloud
Learn About Managed Cloud

JAVA
Learn About Java
Download Java for Consumers
Download Java for Developers
Java Resources for Developers
Java Cloud Service
Java Magazine

CUSTOMERS AND EVENTS
Explore and Read Customer Stories
All Oracle Events
Oracle OpenWorld
JavaOne

EMAIL SUBSCRIPTIONS
Subscribe to Oracle Communications
Subscription Center

COMMUNITIES
Blogs
Discussion Forums
Wikis
Oracle ACEs
User Groups
Social Media Channels

SERVICES AND STORE
Log In to My Oracle Support
Training and Certification
Become a Partner
Find a Partner Solution
Purchase from the Oracle Store

CONTACT AND CHAT
US Sales: +1.800.633.0738
Global Contacts
Oracle Support
Partner Support