Annexes

Remaniements utilisés

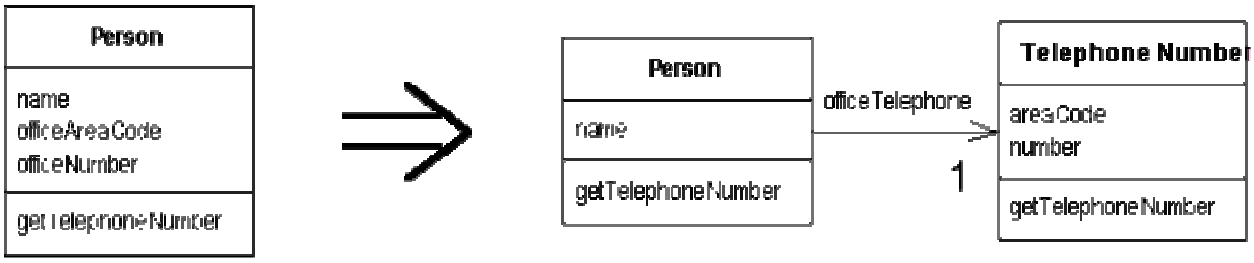
Les remaniements utilisés dans ce livre sont issus du travail de *Martin Fowler*. Vous pouvez consulter la totalité du catalogue dans le livre « *Refactoring – Improving the Design of Existing Code* ». ou sur le site *http://www.refactoring.com/catalog/index.html*.

Extract Class	*You have one class doing work that should be done by two.*
Extract Package	*A package either has too many classes to be easily understandable or it suffers from the 'Promiscuous packages' smell.*
Extract Superclass	*You have two classes with similar features.*
Hide Delegate	*A client is calling a delegate class of an object.*
Move Class	*You have a class that is in a package that contains other classes that it is not related to in function.*
Rename Method	*The name of a method does not reveal its purpose.*
Self Encapsulate Field	*You are accessing a field directly, but the coupling to the field is becoming awkward.*

Extract Class

You have one class doing work that should be done by two.

Create a new class and move the relevant fields and methods from the old class into the new class.



Extract Package

A package either has too many classes to be easily understandable or it suffers from the 'Promiscuous packages' smell.

Extract a sub package depending on their gross dependencies or usages.

Motivation

Dependencies will eventually cause problem in projects of any size, so it make sense to start refactoring sooner rather than later in order to make it clear which part of the code uses what.
This sort of change can make the code more flexible. For example if you are writing a UI tool and then decide a command line variant is required. Then unless the code is properly structured you will have trouble re-using certain components. Packaging is one way of making dependencies explicit.
This refactoring can be useful when a package becomes too large to be easily understood. For example in a diagrammer framework you might like to extra sub packages for important groups such as 'shapes'; 'ui' and 'printing'. This makes it easier to identify the use of a class by its implied association in a package.

The structure produced here is also one that is recommeneded for use with the Abstract Factory pattern. Indeed this is how the example I have provided is structured.

Mechanics

Work out groupings for your classes. Where required use the Extract Superclass to pull together any generic code first.

Create the new package and perform Move Class for each file that needs to be moved. It is often efficient to move groups of classes at once.

Compile the code in the parent package and retest. The code in the sperate package will have been tested as part of the Move Class refactoring.

The refactorings at this point can be considered complete.

You might like to convert the code in the factory to using dynamic class loading by using the Convert Static to Dynamic Construction. This would enable selective inclusion of sub-packages depending on the build environment.

Example

There are no code examples as most of the work in done in Move Class.

Additional Comments

The way I've done this is to move all the classes in one go. (I'm prepared to take the big step first here since all the errors can be found with compiling.)

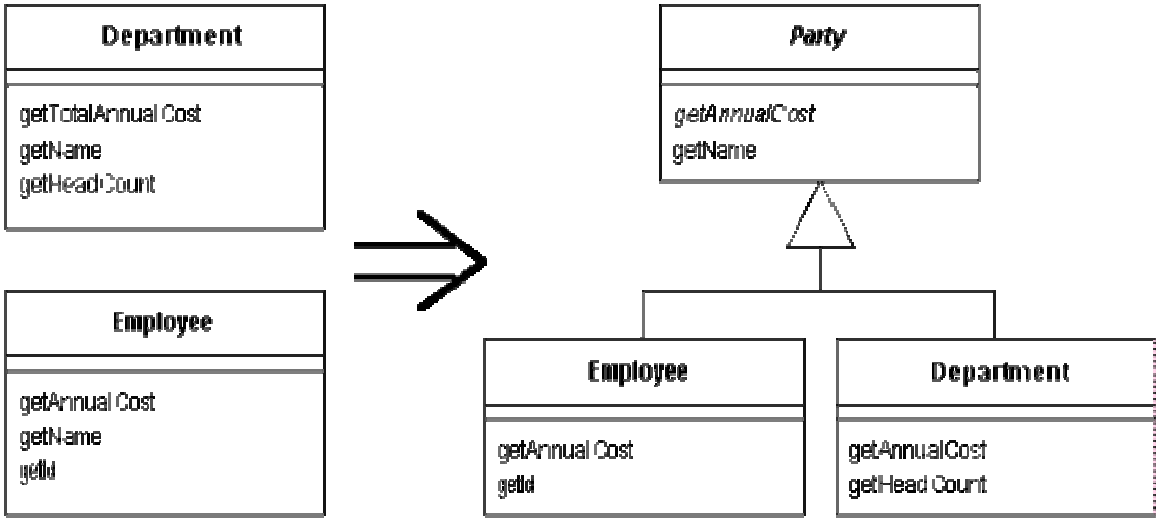
I ensure the original package is dependent on the extracted package but the extracted package is independent of the original. Once I've done the move I compile the original package. The errors here can be fixed with import statements in the offending classes. I fix these errors until the original package compiles

Then I work on the extracted package. This may compile fine, the problem lies if you have a class in the extracted package referring to a class in the original package. I move it back if I can. If not I use Extract Interface to create an interface which captures the way the classes in the extracted pacakge refer to the original class. I then place the extracted interface in the extracted package.

Extract Superclass

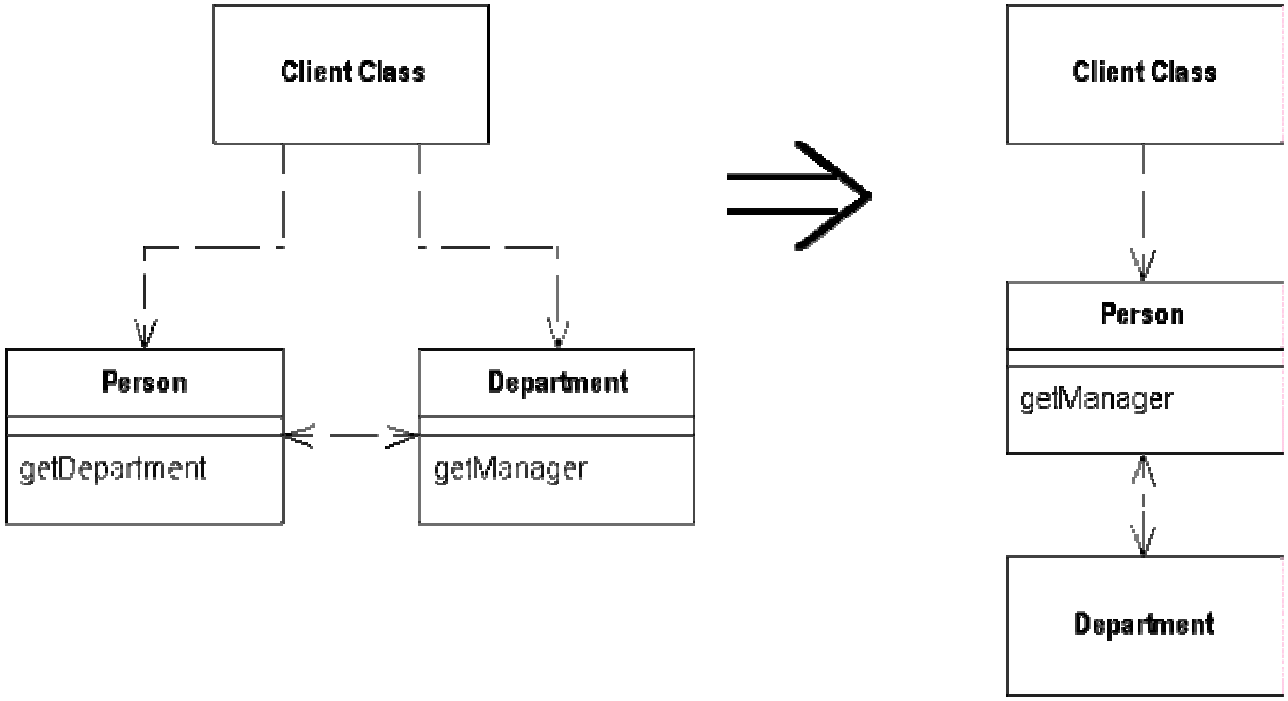
You have two classes with similar features.

Create a superclass and move the common features to the superclass.



Hide Delegate

A client is calling a delegate class of an object.
Create methods on the server to hide the delegate.



Move Class

You have a class that is in a package that contains other classes that it is not related to in function.
Move the class to a more relevant package. Or create a new package if required for future use.

Motivation
Classes are often created in a packages close to where they are being used, this can make sense until the class starts to be re-used by other parts of the product. The package in question might also have just become too big. (I have a preference that my packages never have more than about 10 classes)
It is often better to move to this class to a package more related to it in form or function. This can help remove complex package level dependencies and make it easier for developers to find and re-use classes.

If there are many dependencies for the class within its own package, then Extract Class could be used first to split out the relevant parts.

Another example where is this used often is to move String resource objects into sub a res package to simplify localization compilation.

Mechanics
- Move the class to its new folder on the source tree.
- Remove any class files generated from compiling this class at its old location.
- Alter the package statement in the source file to reflect the new package.
- Create import statements for any dependant classes in original package.
- Compile and test the class in its new package, updating and moving unit tests as required using this same method.
- Alter, and create in some cases, the import statements on any dependee class. Note this is easier if wide imports are not used.
- Check for classes that might dynamically instantiate this class using java.lang.reflect or the ResourceBundle api. Also look for code that might reference this class as an absolute class literal.
- Compile and test all code that was dependant on the original class.
- Consider applying Extract Package when you have many classes of different functionality in a given package.

Example
Lets look at the header of StringUtil

```
package org.davison.ui

// imports

public class StringUtil
```

We move the file and change the package header.

```
package org.davison.util

// imports

public class StringUtil
```

I can now compile and unit test this class in its new package. If the unit tests are in a different package class file then they might need to be updated.

I can now go on to update the dependant classes. For example here I update the import statements, I have not used a wide imports in this example but the principle is the same.

```
package org.davison.log

// imports
import org.davison.ui.StringUtils;

// more imports

public class Logger
```

This becomes:

```
package org.davison.log

// imports
import org.davison.util.StringUtils;

// more imports

public class Logger
```

Again compile and test this new class with the imports.

As mentioned before there are other ways of dynamically loading classes. You may have to look for class literals and strings constants containing the fully qualified name of the old class.

```
// A class literal

public Class m_utils = org.davison.ui.StringUtils.class;

// A dynamically loaded class

public Class m_utils = Class.forName("org.davison.ui.StringUtils");

// A loaded resource bundle

public ResourceBundle m_bundle =ResourceBundle.getBundle("org.davison.ui.StringUtils");
```

These can be fixed using simple find and replaces, these dynamic examples will also be found using the units tests for all classes.

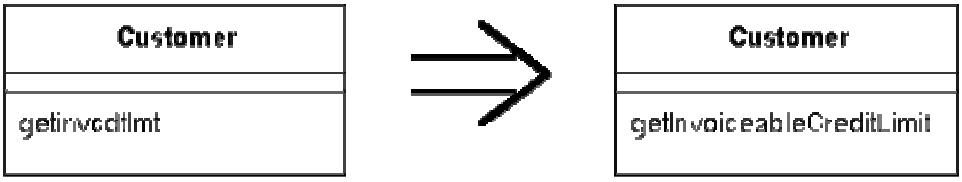
```
// A class literal, this will be found by the compiler.

public Class m_utils = org.davison.util.StringUtils.class;

// A dynamically loaded class

public Class m_utils = Class.forName("org.davison.util.StringUtils");

// A loaded resource bundle

public ResourceBundle m_bundle =ResourceBundle.getBundle("org.davison.util.StringUtils");
```

One all static and dynamic cases have been dealt with, and all unit tests have run. The refactoring is complete.

Rename Method

The name of a method does not reveal its purpose.

Change the name of the method.



Self Encapsulate Field

You are accessing a field directly, but the coupling to the field is becoming awkward.
Create getting and setting methods for the field and use only those to access the field.

```
private int _low, _high;
    boolean includes (int arg) {
    return arg >= _low && arg <= _high;
}
```

This becomes

```
private int _low, _high;
    boolean includes (int arg) {
    return arg >= getLow() && arg <= getHigh();
}

int getLow() {return _low;}
int getHigh() {return _high;}
```

Design Pattern utilisés

Les Design Patterns utilisés dans ce livre proviennent principalement de trois sources différentes. Le fameux livre du Gang of Four « *Design Pattern – Elements of Reusable Object-Oriented Software* » de Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. « *EJB Design Pattern* » de Floyd Marinescu et « *Core J2EE Pattern* » de Deepak Alur, John Crupi et Dan Malks

Business Delegate	Introduces an intermediate class called a business delegate in distributed applications that looks up and handles exceptions for remote business components.
DAO	Separates a data resource's client interface from its data access mechanisms.
DTO	To reduce network traffic, accesses attributs in a corse-grained manner through a remote interface.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Factory	Define an interface for creating an object, but let subclasses decide which class to instantiate.
MVC	Seperates data access code, business logic code, and presentation code.
Service locator	A way to look up the service objects that provide access to distributed components.
Session Facade	The Session Facade pattern defines a higher-level business component that contains and centralizes complex interactions between lower-level business components.
Singleton	Ensure a class has only one instance, and provide a global point of access to it.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses.

Business Delegate

Brief Description
In distributed applications, lookup and exception handling for remote business components can be complex. When applications use business components directly, application code must change to reflect changes in business component APIs.

These problems can be solved by introducing an intermediate class called a business delegate, which decouples business components from the code that uses them. The Business Delegate pattern manages the complexity of distributed component lookup and exception handling, and may adapt the business component interface to a simpler interface for use by views.

Detailed Description
See the Core J2EE Patterns
(http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html)

Detailed Example
Sample application business delegate class `AdminRequestBD` handles distributed lookup and catches and adapts exceptions in the sample application order processing center (OPC).

- **The `AdminRequestBD` business delegate manages distributed component lookup and handles exceptions.**

The structure diagram in Figure 1 shows the `ApplRequestProcessor` servlet using `AdminRequestBD` to find and use distributed business components.

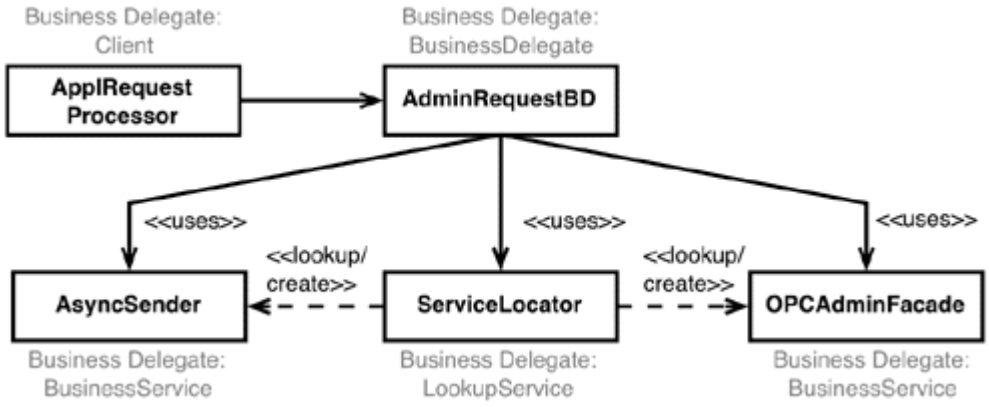


Figure 1. AdminRequestBD locates and adapts other business components

The code sample below shows the constructor for `AdminRequestBD`. It uses class `ServiceLocator` to acquire the remote home interface of session facade `OPCAdminFacade` . (See the Service Locator and Session Facade design patterns.) It uses the remote home interface to create a `OPCAdminFacade` remote component interface, which it maintains in a private field. The block that locates and creates the enterprise bean reference catches exceptions related to finding the home interface and to creating the component interface. Any exception that occurs is then wrapped in an `AdminBDException` , which effectively hides the implementation details of the business delegate from its clients.

```
public AdminRequestBD() throws AdminBDException {
    try {
```

```
        OPCAdminFacadeHome home = (OPCAdminFacadeHome)
ServiceLocator.getInstance().getRemoteHome(OPC_ADMIN_NAME, OPCAdminFacadeHome.class);
        opcAdminEJB = home.create();
    } catch (ServiceLocatorException sle) {
        throw new AdminBDException(sle.getMessage());
    } catch (CreateException ce) {
        throw new AdminBDException(ce.getMessage());
    } catch (RemoteException re) {
        throw new AdminBDException(re.getMessage());
    }
}
```

- **The OPC request processor uses `AdminRequestBD` for simple access to business components components.**

OPC servlet class `ApplRequestProcessor` receives service requests from the admin client in the form of XML messages transmitted using HTTP. One of these request is for statistics about orders that have a given status.

Method `ApplRequestProcessor.getOrders` receives part of an XML DOM tree representing a Web service request. It extracts the status code from the document and uses `AdminRequestBD.getOrdersByStatus` to retrieve a list of order information. The interface to that list is transfer object interface `OrdersTO` . (See the Transfer Object pattern.) The code in the request processor that retrieves the order information appears in the following code sample.

```
public class ApplRequestProcessor extends HttpServlet {
...
    String getOrders(Element root) {
        try {
            AdminRequestBD bd = new AdminRequestBD();
            NodeList nl = root.getElementsByTagName("Status");
            String status = getValue(nl.item(0));
            OrdersTO orders = bd.getOrdersByStatus(status);
...
}
```

Because it has already created the reference to an `OPCAdminFacadeEJB`, the `AdminRequestBD` object can simply forward the call to the enterprise bean's method getOrdersByStatus, as follows:

```
public class AdminRequestBD {
...
    public OrdersTO getOrdersByStatus(String status)
        throws AdminBDException {

        try {
            return opcAdminEJB.getOrdersByStatus(status);
        } catch (RemoteException re) {
            throw new AdminBDException(re.getMessage());
        } catch (OPCAdminFacadeException oafee) {
            throw new AdminBDException(oafee.getMessage());
        }
    }
...
}
```

Notice again that the method catches any exceptions that the enterprise bean may throw and re-throws an exception type that is specific to the business delegate's interface. This hides the business delegate's implementation details from the client.

Data Access Object (DAO)

Brief Description
Code that depends on specific features of data resources ties together business logic with data access logic. This makes it difficult to replace or modify an application's data resources.

The Data Access Object (or DAO) pattern:

- separates a data resource's client interface from its data access mechanisms
- adapts a specific data resource's access API to a generic client interface

The DAO pattern allows data access mechanisms to change independently of the code that uses the data.

Detailed Description
See the Core J2EE Patterns
(http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html)

Detailed Example
The Java Pet Store sample application uses the DAO pattern both for database vendor-neutral data access, and to represent XML data sources as objects.

- **Accessing a database with a DAO.**

A Data Access Object class can provide access to a particular data resource without coupling the resource's API to the business logic. For example, sample application classes access catalog categories, products, and items using DAO interface `CatalogDAO`.

Reimplementing `CatalogDAO` for a different data access mechanism (to use a Connector, for example), would have little or no impact on any classes that use `CatalogDAO`, because only the implementation would change. Each potential alternate implementation of `CatalogDAO` would access data for the items in the catalog in its own way, while presenting the same API to the class that uses it.

The following code excerpts illustrate how the sample application uses the DAO pattern to separate business logic from data resource access mechanisms:

- Interface `CatalogDAO` defines the DAO API. Notice that the methods in the interface below make no reference to a specific data access mechanism. For example, none of the methods specify an SQL query, and they throw only exceptions of type `CatalogDAOSysException` . Avoiding mechanism-specific information in the DAO interface, including exceptions thrown, is essential for hiding implementation details.

```
public interface CatalogDAO {
    public Category getCategory(String categoryID, Locale l)
        throws CatalogDAOSysException;
    public Page getCategories(int start, int count, Locale l)
        throws CatalogDAOSysException;
    public Product getProduct(String productID, Locale l)
        throws CatalogDAOSysException;
    public Page getProducts(String categoryID, int start, int count, Locale l)
        throws CatalogDAOSysException;
    public Item getItem(String itemID, Locale l)
        throws CatalogDAOSysException;
    public Page getItems(String productID, int start, int size, Locale l)
```

```
        throws CatalogDAOSysException;
    public Page searchItems(String query, int start, int size, Locale l)
        throws CatalogDAOSysException;
}
```

- Class `CloudscapeCatalogDAO` implements this interface for the Cloudscape relational database, as shown in the following code excerpt. Note that the SQL to access the Cloudscape database is hard-coded.

```
public class CloudscapeCatalogDAO implements CatalogDAO {
...
public static String GET_CATEGORY_STATEMENT
  = "select name, descn "
  + " from (category a join category_details b on a.catid=b.catid) "
  + " where locale = ? and a.catid = ?";
...
public Category getCategory(String categoryID, Locale l)
    throws CatalogDAOSysException {
      Connection c = null;
      PreparedStatement ps = null;
      ResultSet rs = null;
      Category ret = null;

      try {
        c = getDataSource().getConnection();
        ps = c.prepareStatement(GET_CATEGORY_STATEMENT,
                                ResultSet.TYPE_SCROLL_INSENSITIVE,
                                ResultSet.CONCUR_READ_ONLY);
        ps.setString(1, l.toString());
        ps.setString(2, categoryID);
        rs = ps.executeQuery();
        if (rs.first()) {
          ret = new Category(categoryID, rs.getString(1), rs.getString(2));
        }

        rs.close();
        ps.close();
        c.close();
        return ret;
      } catch (SQLException se) {
        throw new CatalogDAOSysException("SQLException: " + se.getMessage());
      }
  }
...
}
```

- **Implementation strategies**. Designing a DAO interface and implementation is a tradeoff between simplicity and flexibility. The sample application provides examples of several strategies for implementing the Data Access Object pattern.

    - Implement the interface directly as a class. The simplest (but least flexible) way to implement a data access object is to write it as a class. Class `ScreenDefinitionsDAO`, described below and shown in Figure 1 above, is an example of a class that directly implements a DAO interface. This approach separates the data access interface from the details of how it is implemented, providing the benefits of the DAO pattern. The data access mechanism can be changed easily by writing a new class that implements the same interface, and changing client code to use the new class. Yet this approach is inflexible because it requires a code changes to modify the data access mechanism.

    - Improve flexibility by making DAOs "pluggable". A pluggable DAO allows an application developer or deployer to select a data access mechanism with no changes to program code. In this approach, the developer accesses a data source only in terms of an

abstract DAO interface. Each DAO interface has one or more concrete classes that implement that interface for a particular type of data source. The application uses a factory object to select the DAO implementation at runtime, based on configuration information.

For example, the sample application uses factory class `CatalogDAOFactory` to select the class that implements the DAO interface for the catalog. Figure 2 below presents a structure diagram of the Data Access Object design pattern using a factory to select a DAO implementation.

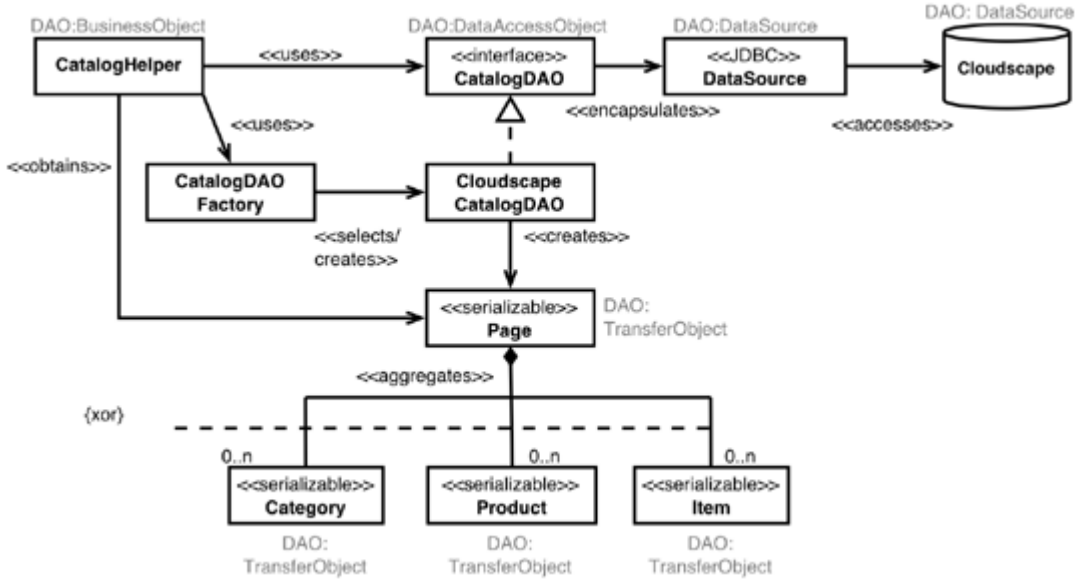


Figure 2. A pluggable DAO

At runtime, the `CatalogHelper` uses the `CatalogDAOFactory` to create an object that implements `CatalogDAO`. The factory looks up the name of the class that implements the DAO interface in environment entry "`param/CatalogDAOClass`". The `CatalogHelper` accesses the catalog data source exclusively using the object created by the factory. In the example shown in the figure, the environment entry was set to the (fully-specified) name of class `CloudscapeCatalogDAO` . This class implements the catalog DAO interface in terms of JDBC data sources, accessing a Cloudscape relational database.

This approach is more flexible than using a hard-coded class. To add a new type of data source, an application developer would simply create a class that implements `CatalogDAO` in terms of the new data source type, specify the implementing class's name in the environment entry, and re-deploy. The factory would create an instance of the new DAO class, and the application would use the new data source type.

- Reduce redundancy by externalizing SQL. Writing a separate class for data source types that have similar APIs can create a great deal of redundant code. For example, JDBC data sources differ from one another primarily in the SQL used to access them. The only differences between the Cloudscape DAO described above and the DAO for a different SQL database are the connection string and the SQL used to access the database.

The sample application reduces redundant code by using a "generic DAO" that externalizes the SQL for different JDBC data sources. Figure 3 below shows how the sample application uses an XML file to specify the SQL for different JDBC data sources.

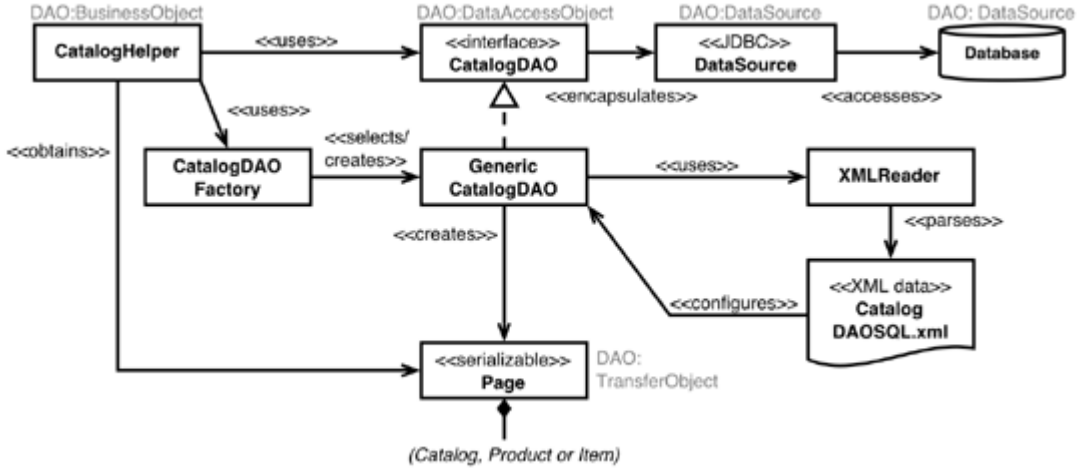


Figure 3. Externalizing DAO SQL

In the figure, the `CatalogDAOFactory` has selected an object of type `GenericCatalogDAO` as the DAO to access the catalog. An XML file called `CatalogDAOSQL.xml` specifies the SQL for each supported operation on each type of database. `GenericCatalogDAO` configures itself by selecting a group of SQL statements from the XML file that correspond to the database type named by environment entry "`param/CatalogDAODatabase`". The code sample below shows the definition of the XML for the `getCategory` operation of the `CatalogDAO`. Different SQL is specified for "cloudscape" and "oracle" database types.

```
<DAOConfiguration>
    <DAOStatements database="cloudscape">
        <SQLStatement method="GET_CATEGORY">
            <SQLFragment parameterNb="2">
                select name, descn
                    from (category a join category_details b
                          on a.catid=b.catid)
                    where locale = ? and a.catid = ?
            </SQLFragment>
        </SQLStatement>
        ...
    <DAOStatements database="oracle">
        <SQLStatement method="GET_CATEGORY">
            <SQLFragment parameterNb="2">
                select name, descn
                    from category a, category_details b
                    where a.catid = b.catid and locale = ? and a.catid = ?
            </SQLFragment>
        </SQLStatement>
       ...
</DAOConfiguration>
```

Method `GenericCatalogDAO.getCategory` chooses the SQL corresponding to the configured database type, and uses it to fetch a category from the database via JDBC. The following code excerpt shows shows the implementation of the method.

```
   public Category getCategory(String categoryID, Locale locale)
     throws CatalogDAOSysException {
     Connection connection = null;
     ResultSet resultSet = null;
     PreparedStatement statement = null;
     try {
       connection = getDataSource().getConnection();
       String[] parameterValues = new String[] { locale.toString(), categoryID };
       statement = buildSQLStatement(connection, sqlStatements,
```

```
            XML_GET_CATEGORY, parameterValues);
        resultSet = statement.executeQuery();
        if (resultSet.first()) {
          return new Category(categoryID, resultSet.getString(1), resultSet.getString(2));
        }
        return null;
      } catch (SQLException exception) {
        throw new CatalogDAOSysException("SQLException: " + exception.getMessage());
      } finally {
        closeAll(connection, statement, resultSet);
      }
    }
```

Notice that the method catches any possible `SQLException` and converts it to a `CatalogDAOSysException`, hiding the implementation detail that the DAO uses a JDBC database.

This strategy supports multiple JDBC databases with a single DAO class. It both decreases redundant code, and makes new database types easier to add. To support a new database type, a developer simply adds the SQL statements for that database type to the XML file, updates the environment entry to use the new type, and redeploys.

The pluggable DAO and generic DAO strategies can be used separately. If you know that a DAO class will only ever use JDBC databases (for example), the generic DAO class can be hardwired into the application, instead of selected by a factory. For maximum flexibility, the sample application uses both a factory method and a generic DAO.

- **Encapsulating non-database data resources as DAO classes.**

A data access object can represent data that is not stored in a database. The sample application uses the DAO pattern to represent XML data sources as objects. Sample application screens are defined in an XML file which is interpreted by the class `ScreenDefinitionDAO`. Specifying screen definitions externally makes access to the screen definitions more flexible. For example, if the application designers (or maintainers) decide to change the application to store screen descriptions in the database, instead of in an XML file, they would need only to implement a single new class (`ScreenFlowCloudscapeDAO`, for example). The code that uses `ScreenDefinitionDAO` would remain unchanged, but the data would come from the database via the new class.

The screen definitions mechanism in the sample application provides an example of a concrete Data Access Object representing an underlying, non-database resource (an XML file).

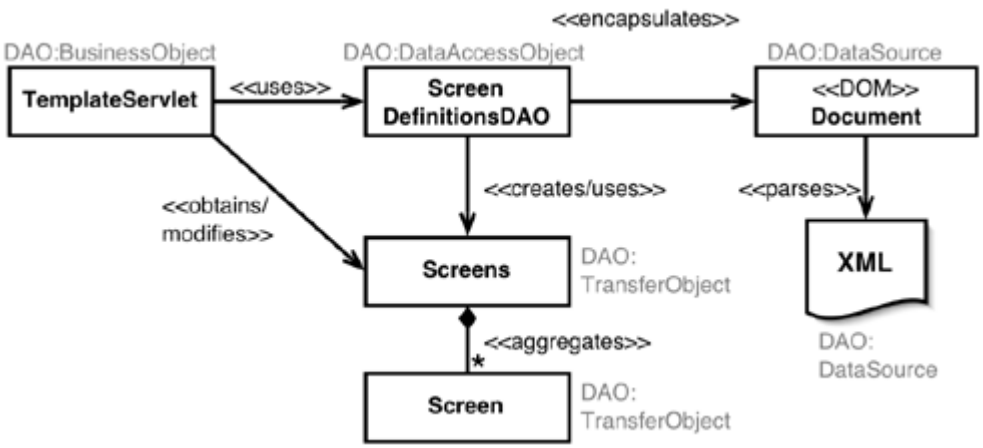


Figure 1. Data Access Object providing access to XML data source

Figure 1 shows a structure diagram of the ScreenDefinitionDAO managing the loading and interpretation of XML data that defines application screens.

- `TemplateServlet` uses the `ScreenDefinitionDAO` to load screen definitions:

```
Screens screenDefinitions =
    ScreenDefinitionDAO.loadScreenDefinitions(screenDefinitionURL);
```

- `ScreenDefinitionDAO` represents screen definitions in an XML file deployed with an application. It uses XML APIs to read the screen definitions from the XML file. Only this class would need to be replaced to support storing screen definitions in some other way. The method that loads screen definitions looks like this:

```
    public static Screens loadScreenDefinitions(URL location) {
        Element root = loadDocument(location);
        if (root != null) return getScreens(root);
        else return null;
    }
   ...
   public static Screens getScreens(Element root) {
    // get the template
    String defaultTemplate = getTagValue(root, DEFAULT_TEMPLATE);
    if (defaultTemplate == null) {
        System.err.println("*** ScreenDefinitionDAO error: " +
                           " Default Template not Defined.");
        return null;
    }

    Screens screens = new Screens(defaultTemplate);
    getTemplates(root, screens);
    // get screens
    NodeList list = root.getElementsByTagName(SCREEN);
    for (int loop = 0; loop < list.getLength(); loop++) {
        Node node = list.item(loop);
        if ((node != null) && node instanceof Element) {
            String templateName = ((Element)node).getAttribute(TEMPLATE);
            String screenName = ((Element)node).getAttribute(NAME);
            HashMap parameters = getParameters(node);
            Screen screen = new Screen(screenName, templateName, parameters);
            if (!screens.containsScreen(screenName)) {
                screens.addScreen(screenName, screen);
            } else {
                System.err.println("*** Non Fatal errror: Screen " + screenName +
                                   " defined more than once in screen definitions file");
            }
        }
    }
    return screens;
    }
```

The code fragment above shows how loadScreenDefinitions loads screen definitions using DOM interfaces, while hiding that fact from clients of the class. A client of this class can expect to receive a Screens object regardless of how those screens are loaded from persistent storage. Method getScreens handles all of the DOM-specific details of loading a screen from an XML file.

Data Transfert Object

Brief Description
Some entities contain a group of attributes that are always accessed together. Accessing these attributes in a fine-grained manner through a remote interface causes network traffic and high latency, and consumes server resources unnecessarily.

A transfer object (previously known as Value Object) is a serializable class that groups related attributes, forming a composite value. This class is used as the return type of a remote business method. Clients receive instances of this class by calling coarse-grained business methods, and then locally access the fine-grained values within the transfer object. Fetching multiple values in one server roundtrip decreases network traffic and minimizes latency and server resource usage.

Detailed Description
See the Core J2EE Patterns
(http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html)

Detailed Example

- **The OrderDetails and OrdersTO transfer objects.**

    - Sample application class `OPCAdminFacade` (see Session Facade) has a method `getChartInfo` that returns an immutable, serializable collection of `OrderDetails` objects, each of which is a transfer object that represents data from an order. The collection returned by method `getChartInfo` is also a transfer object, because the values it contains are always accessed together where they are used. This hierarchical transfer object is implemented by class `OrdersTO`.

    Figure 1 shows the structure of the `OrdersTO` transfer object. Class `MutableOrdersTO`, which extends a serializable `ArrayList`, contains a collection of `OrderDetails` objects. But because `MutableOrdersTO` extends `ArrayList`, it is mutable. Transfer objects should be immutable so that clients do not unintentionally change their contents. Interface `OrdersTO` adapts the `MutableOrdersTO` collection, allowing read-only access to the collection while preventing modifications.

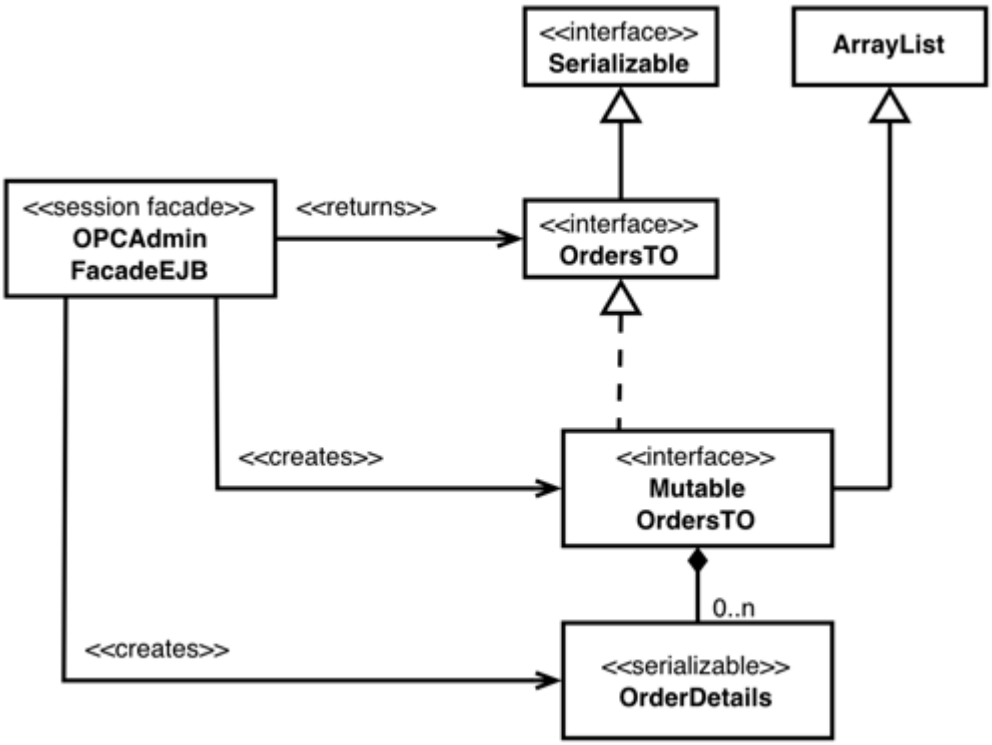


Figure 1. Transfer object OrdersTO is an immutable, serializable collection

The definition of the OrdersTO interface follows.

```
public interface OrdersTO extends Serializable {

    public Iterator iterator();
    public int size();
    public boolean contains(Object o);
    public boolean containsAll(Collection c);
    public boolean equals(Object o);
    public int hashCode();
    public boolean isEmpty();
    public Object[] toArray();
    public Object[] toArray(Object[] a);

    static class MutableOrdersTO extends ArrayList implements OrdersTO {
    }
}
```

The `OrdersTO` interface is clearly a collection, since it defines the Java collection methods. The collection is immutable, because it has no methods that would allow the collection contents to be changed.

An example of using `OrdersTO` appears in method `OPCAdminFacadeEJB.getOrdersByStatus`, shown below. The method constructs a `OrdersTO.MutableOrdersTO` object and populates it with OrderDetails instances. Note that the return type of `getOrdersByStatus` is `OrdersTO`, so callers can access only methods that read the collection contents, and not methods that would change them.

```
public OrdersTO getOrdersByStatus(String status)
   throws OPCAdminFacadeException {

    OrdersTO.MutableOrdersTO retVal = new OrdersTO.MutableOrdersTO();
    PurchaseOrderLocal po;
    ProcessManagerLocal mgr = getProcMgr();
```

```
    try {
        PurchaseOrderLocalHome pohome = getPO();
        Collection orders = mgr.getOrdersByStatus(status);
        Iterator it = orders.iterator();
        while((it!= null) && (it.hasNext())) {
            ... // Access and format data
            retVal.add(new OrderDetails(po.getPoId(), po.getPoUserId(),
                                        podate, po.getPoValue(), status));
        }
    } catch (FinderException fe) {
       ... // process exception
    }
    return(retVal);
}
```

- The details for a single order are contained in an immutable, serializable transfer object of type `OrderDetails`. It is a typical transfer object for a single composite value, containing private fields and public, read-only property accessors, as shown in the following code sample.

```
public class OrderDetails implements java.io.Serializable {

    private String orderId;
    private String userId;
    private String orderDate;
    private float orderValue;
    private String orderStatus;

    public OrderDetails(String oid, String uid, String date, float value,
                        String stat) {
        orderId = oid;
        userId = uid;
        orderDate = date;
        orderValue = value;
        orderStatus = stat;
    }

    public String getOrderId() {
        return(orderId);
    }

    // ...
}
```

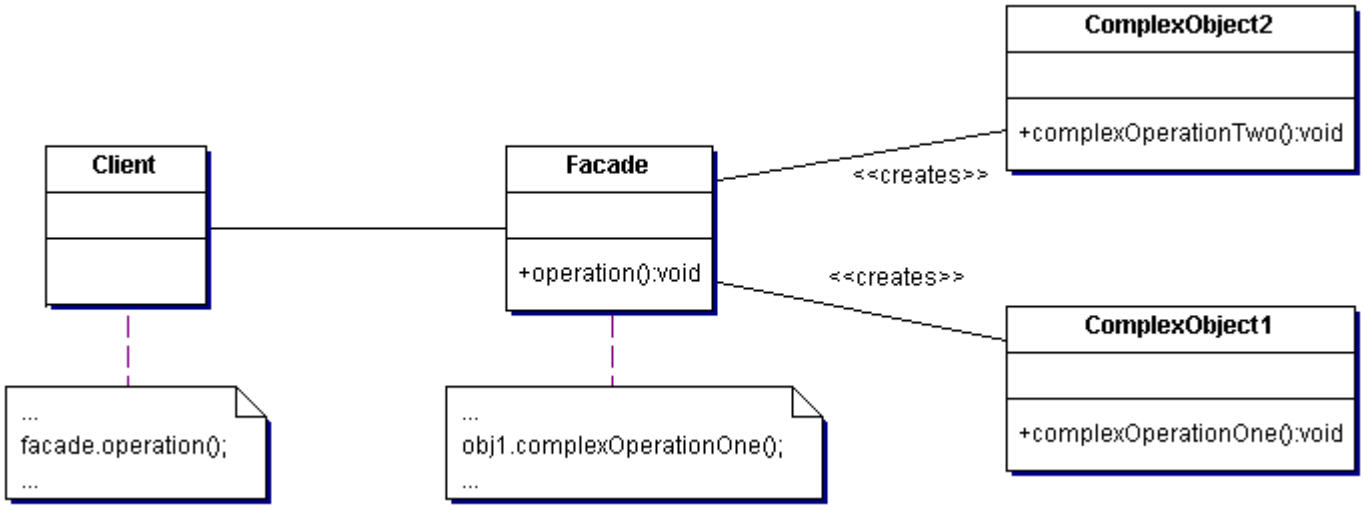
Facade

Intent
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Problem
A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

Discussion
Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.
The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Structure



Example
The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb
Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one. [GOF, p219]

Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem. [GOF, p138]

Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes. [GOF. p193]

Abstract Factory can be used as an alternative to Facade to hide platform-specific classes. [GOF, p193]

Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

Factory

Intent
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Problem
A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Discussion
Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

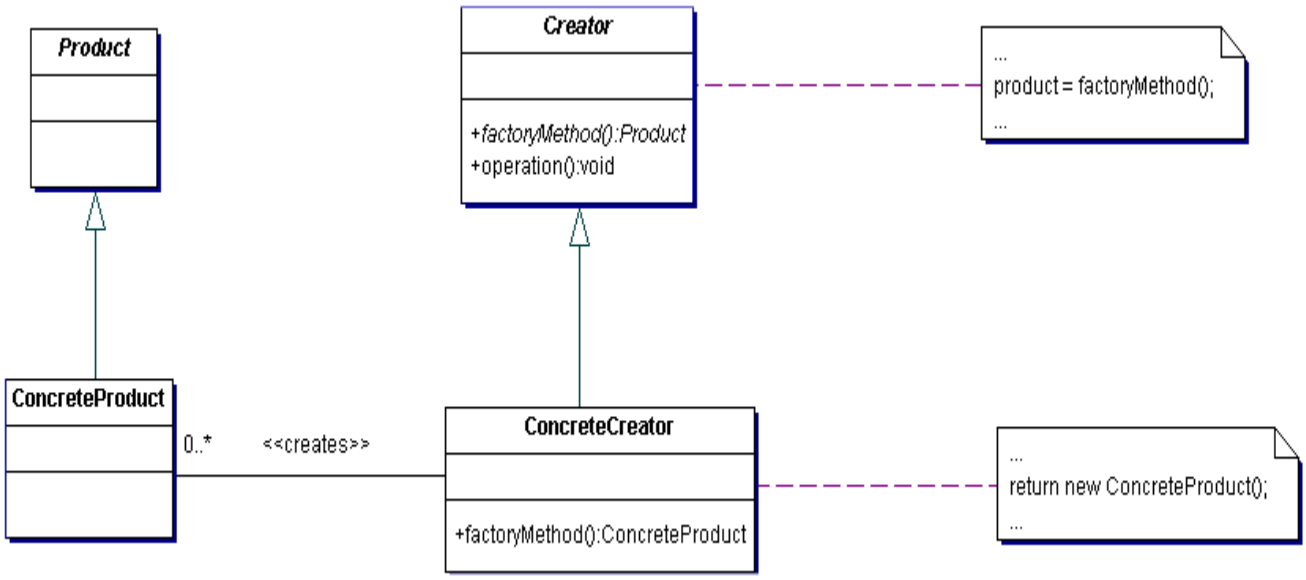
Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation. [GOF, p136]

People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation). [GOF, p136]

Factory Method is similar to Abstract Factory but without the emphasis on families.

Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

Structure



Example
The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The

class of toy (car, action figure, etc.) is determined by the mold. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb

Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype. [GOF, p95]

Factory Methods are usually called within Template Methods. [GOF, p116]

Factory Method: creation through inheritance. Prototype: creation through delegation.

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. [GOF, p136]

Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize. [GOF, p116]

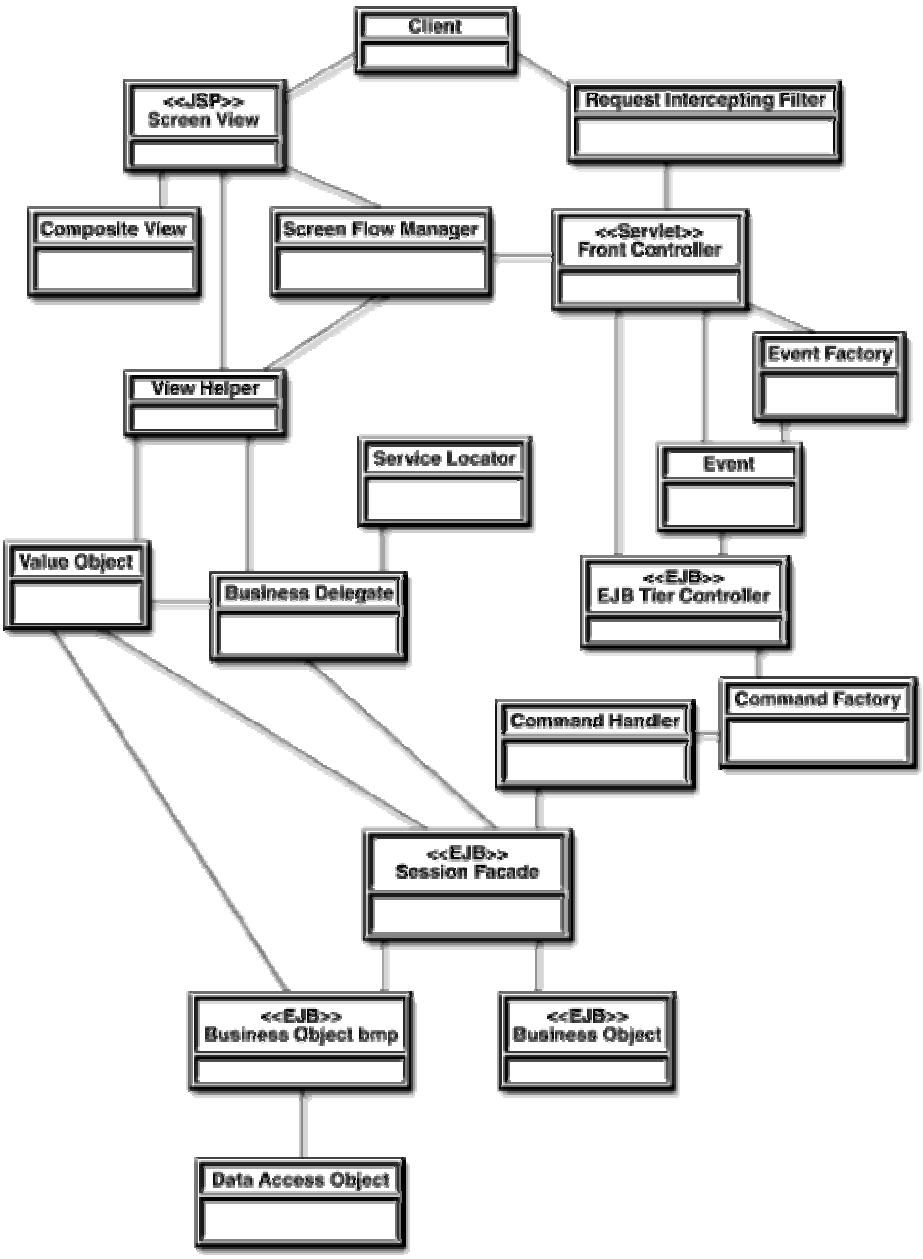
MVC (Model-View-Constoller)

Brief Description
Several problems can arise when applications contain a mixture of data access code, business logic code, and presentation code. Such applications are difficult to maintain, because interdependencies between all of the components cause strong ripple effects whenever a change is made anywhere. High coupling makes classes difficult or impossible to reuse because they depend on so many other classes. Adding new data views often requires reimplementing or cutting and pasting business logic code, which then requires maintenance in multiple places. Data access code suffers from the same problem, being cut and pasted among business logic methods.

The Model-View-Controller design pattern solves these problems by decoupling data access, business logic, and data presentation and user interaction.

Detailed Description
See Model-View-Controller explanation of this architectural pattern
(http://java.sun.com/blueprints/patterns/MVC-detailed.html)

Detailed Example
The architecture of the Java Pet Store sample application website applies the Model-View-Controller design pattern. Other design patterns are combined in the design of the MVC architecture.



Class Diagram Showing Sample Application Architectural Components

The architecture of the Java Pet Store website is described in more detail in the Java BluePrints Program book [SSJ02]

Often, MVC functionality is captured in a framework that is reused by different applications. The sample application Web Application Framework is an extensible framework for creating MVC applications. It is an implementation of MVC to which new data sources, business logic, and data views may be added. The Web Application Framework design is described in more detail at http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html .

Service locator

Brief Description
Enterprise applications require a way to look up the service objects that provide access to distributed components. Java 2 Platform, Enterprise Edition (J2EE) applications use Java Naming and Directory Interface (JNDI) to look up enterprise bean home interfaces, Java Message Service (JMS) components, data sources, connections, and connection factories. Repetitious lookup code makes code difficult to read and maintain. Furthermore, unnecessary JNDI initial context creation and service object lookups can can cause performance problems.

The Service Locator pattern centralizes distributed service object lookups, provides a centralized point of control, and may act as a cache that eliminates redundant lookups. It also encapsulates any vendor-specific features of the lookup process.

Detailed Description
See the Core J2EE Patterns
(http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html)

Detailed Example
The Java Pet Store sample application, v1.3.1 has two service locators: a Web-tier class `ServiceLocator` , and an Enterprise JavaBeansTM (EJB) tier class, also called `ServiceLocator` . Both classes manage lookup and caching of enterprise bean home interfaces, JMS and database connection factories, and environment entries within their respective tiers. The only difference between them is that the Web-tier class is a singleton, and it caches the objects it looks up. The EJB-tier class is not a singleton, and does not cache.

The following code discussion uses examples from the Web-tier ServiceLocator :

- **Clients use ServiceLocator to access services.**

The sample application class `AdminRequestBD` is a business delegate that uses the Web-tier `ServiceLocator` to access the order processing center enterprise bean `OPCAdminFacade` . (See the Business Delegate design pattern for a more detailed description of `AdminRequestBD`.)

Figure 1 is a structure diagram that demonstrates how `AdminRequestBD` uses `ServiceLocator` to find the remote home interface of the `OPCAdminFacade` enterprise bean. The `ServiceLocator` returns a the remote enterprise bean interface `OPCAdminFacadeHome`, by either retrieving it from the cache or looking it up using an internal `InitialContext` instance. The client then uses the `OPCAdminFacade` to find or create a remote component interface to an `OPCAdminFacade`

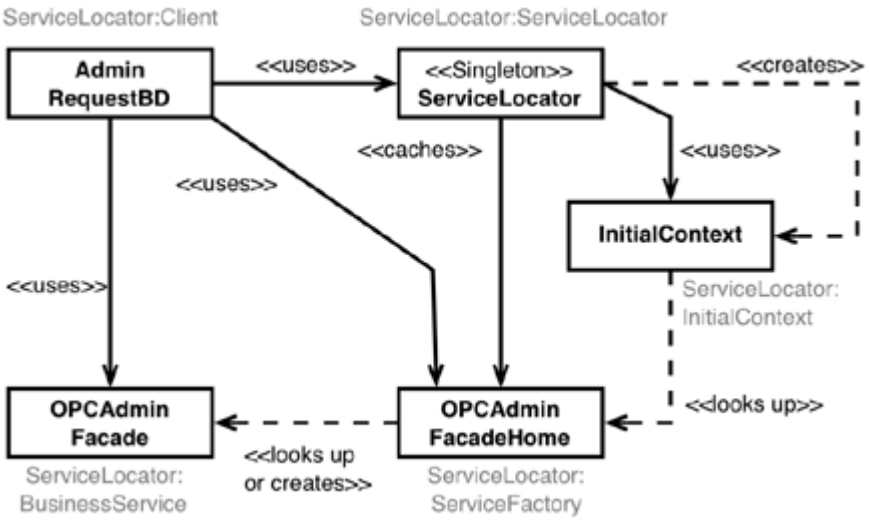


Figure 1. Structure diagram of ServiceLocator sample code

In the following code excerpt, `AdminRequestBD` calls the `ServiceLocator` static method `getInstance` to get the singleton instance of the service locator, then calls `getRemoteHome` to get the remote home interface of the `OPCAdminFacade` enterprise bean. Notice that the caller must typecast the remote home interface to `OPCAdminFacadeHome` because `getRemoteHome` returns type `EJBHome` .

```
public class AdminRequestBD {
...
  public AdminRequestBD() throws AdminBDException {
    try {
      OPCAdminFacadeHome home =
        (OPCAdminFacadeHome) ServiceLocator.getInstance().getRemoteHome(OPC_ADMIN_NAME,
OPCAdminFacadeHome.class);
      opcAdminEJB = home.create();
    } catch (ServiceLocatorException sle) {
    ...
    }
}
```

The service locator greatly simplifies the lookup of the enterprise bean home interface. The singleton and caching strategies (discussed below) also improve performance, because they avoid constructing unnecessary `InitialContext` and enterprise bean home interfaces.

- **Public methods look up distributed resources.**

The public methods of the service locator look up distributed resources by their JNDI names. There are methods that find and return enterprise bean local home interfaces, JDBC data sources, JMS queues and topics, and JMS queue and topic connection factories. There are also convenience methods that look up and perform type conversions on environment entries.

As an example, method `getLocalHome` (for finding enterprise bean local home interfaces) appears below. Each method that locates a particular type of resource returns either a cached reference to the requested resource, or uses JNDI to find the resource, placing a reference in the cache before returning it.

```
    // Enterprise bean lookups
    public EJBLocalHome getLocalHome(String jndiHomeName)
    throws ServiceLocatorException {
      EJBLocalHome home = null;
```

```
      try {
        if (cache.containsKey(jndiHomeName)) {
            home = (EJBLocalHome) cache.get(jndiHomeName);
        } else {
            home = (EJBLocalHome) ic.lookup(jndiHomeName);
            cache.put(jndiHomeName, home);
        }
       } catch (NamingException ne) {
            throw new ServiceLocatorException(ne);
       } catch (Exception e) {
            throw new ServiceLocatorException(e);
       }
       return home;
    }
```

Methods that return enterprise bean home interface references are only type-safe to the platform interface level; for example, `getLocalHome` returns a `EJBLocalHome`, but the client must typecast the result.

Method `getRemoteHome` is similar to `getLocalHome`, except that it returns an enterprise bean remote, instead of local, home interface. It also requires a reference to a class object for the specific remote home interface, because remote home lookups use method `PortableRemoteObject.narrow` to perform the type conversion from the object returned from the JNDI lookup to the actual home interface type. The client that calls `getRemoteHome` must still typecast the result to the remote home interface type, as shown in the first example above.

```
    public EJBHome getRemoteHome(String jndiHomeName, Class className)
    throws ServiceLocatorException {
      EJBHome home = null;
      try {
        if (cache.containsKey(jndiHomeName)) {
            home = (EJBHome) cache.get(jndiHomeName);
        } else {
            Object objref = ic.lookup(jndiHomeName);
            Object obj = PortableRemoteObject.narrow(objref, className);
            home = (EJBHome)obj;
            cache.put(jndiHomeName, home);
        }
       } catch (NamingException ne) {
            throw new ServiceLocatorException(ne);
       } catch (Exception e) {
            throw new ServiceLocatorException(e);
       }

       return home;
    }
```

As mentioned above, the service locator returns JMS resources, JDBC data sources, and performs type conversion on values in environment entries. The table below summarizes the names and return types of these methods.

- **Table 1. Additional ServiceLocator methods**

Method Name	Return Type	Resource Type
`getQueueConnectionFactory`	`QueueConnectionFactory`	JMS
`getQueue`	`Queue`	JMS
`getTopicConnectionFactory`	`TopicConnectionFactory`	JMS
`getTopic`	`Topic`	JMS

`getDataSource`	`DataSource`	JDBC
`getUrl`	`URL`	env-entry
`getBoolean`	`boolean`	env-entry
`getString`	`String`	env-entry

- **Improving performance with the Singleton pattern and caching.**

The Singleton pattern ensures that only a single instance of a class exists in an application. The meaning of the term "singleton" is not always clear in a distributed environment; in `ServiceLocator` it means that only one instance of the class exists per class loader.

The Singleton pattern improves performance because it eliminates unnecessary construction of `ServiceLocator` objects, JNDI `InitialContext` objects, and enables caching (see below).

The Web-tier service locator also improves performance by caching the objects it finds. The cache lookup ensures that a JNDI lookup only occurs once for each name. Subsequent lookups come from the cache, which is typically much faster than a JNDI lookup.

The code excerpt below demonstrates how the `ServiceLocator` improves performance with the Singleton pattern and an object cache.

```
public class ServiceLocator {

    private InitialContext ic;
    private Map cache;

    private static ServiceLocator me;

    static {
      try {
        me = new ServiceLocator();
      } catch(ServiceLocatorException se) {
        System.err.println(se);
        se.printStackTrace(System.err);
      }
    }
   private ServiceLocator() throws ServiceLocatorException  {
      try {
        ic = new InitialContext();
        cache = Collections.synchronizedMap(new HashMap());
      } catch (NamingException ne) {
            throw new ServiceLocatorException(ne);
       }
    }

   static public ServiceLocator getInstance() {
      return me;
    }
```

A private class variable me contains a reference to the only instance of the `ServiceLocator` class. It is constructed when the class is initialized in the static initialization block shown. The constructor initializes the instance by creating the JNDI InitialContext and the `HashMap` that is used a cache. Note that the no-argument constructor is private: only class `ServiceLocator` can construct a `ServiceLocator`. Because only the static initialization block creates the instance, there can be only one instance per class loader.

Classes that use service locator access the singleton `ServiceLocator` instance by calling public method `getInstance`.

Each object looked up has a JNDI name which, being unique, can be used as a cache `HashMap` key for the object. Note also that the `HashMap` used as a cache is synchronized so that it may be safely accessed from multiple threads that share the singleton instance.

Session Facade

Brief Description
Many business processes involve complex manipulations of business classes. Business classes often participate in multiple business processes or workflows. Complex processes that involve multiple business objects can lead to tight coupling between those classes, with a resulting decrease in flexibility and design clarity. Complex relationships between low-level business components make clients difficult to write.

The Session Facade pattern defines a higher-level business component that contains and centralizes complex interactions between lower-level business components. A Session Facade is implemented as a session enterprise bean. It provides clients with a single interface for the functionality of an application or application subset. It also decouples lower-level business components from one another, making designs more flexible and comprehensible.

Fine-grained access through remote interfaces is inadvisable because it increases network traffic and latency. The "before" diagram in Figure 1 below shows a sequence diagram of a client accessing fine-grained business objects through a remote interface. The multiple fine-grained calls create a great deal of network traffic, and performance suffers because of the high latency of the remote calls.

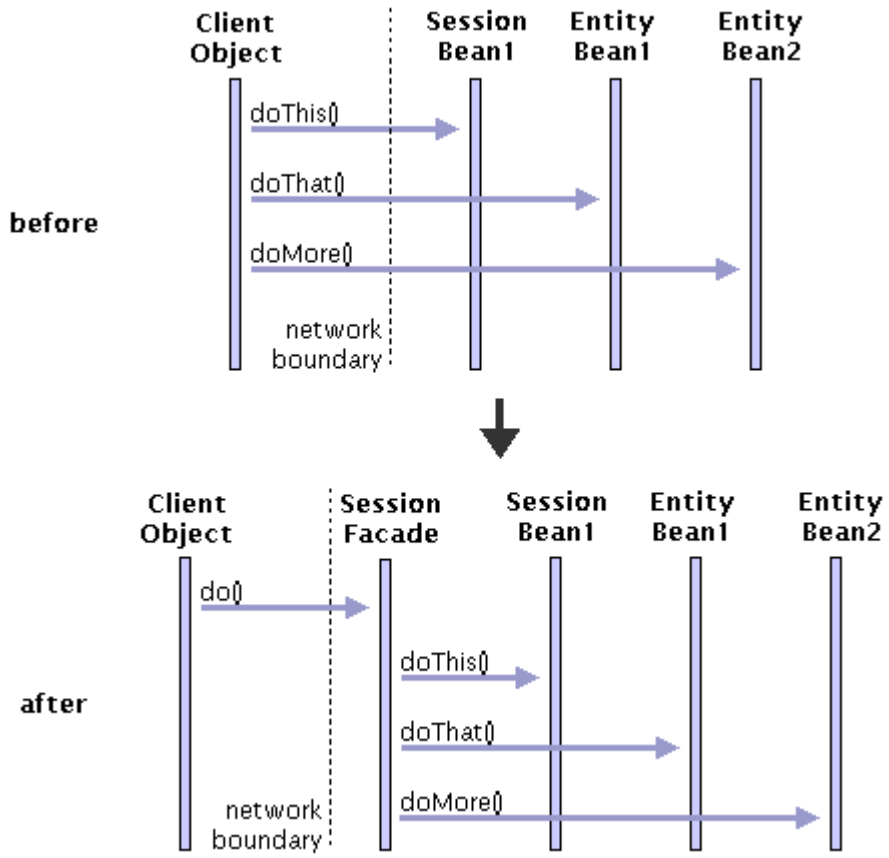


Figure 1. Sequence diagram before and after adding Session Facade

Introducing a Session Facade, as shown in the "after" diagram in Figure 1, decreases network traffic and latency, because all access to fine-grained business objects is local. The Session Facade also acts as a Mediator [GOF] between the business objects, decoupling their APIs from one another.

Detailed Description

See the Core J2EE Patterns
(http://java.sun.com/blueprints/corej2eepatterns/Patterns/SessionFacade.html)

Detailed Example

- **The admin facade.**

The sample admin application of the Pet Store enterprise is responsible for management functionalities of the Pet Store order processing workflow. The admin application is a Swing client with Java Web Start that communicates with the backend order processing center application to approve large purchase orders and view financial data. To interact with the order processing center application, the admin application uses the Session Facade pattern in the interface `OPCAdminFacade` , whose interface appears below.

```
public interface OPCAdminFacade extends EJBObject {

    public OrdersTO getOrdersByStatus(String status)
        throws RemoteException, OPCAdminFacadeException;

    public Map getChartInfo(String request,
                            Date start,
                            Date end,
                            String requestedCategory)
        throws RemoteException, OPCAdminFacadeException;
}
```

The admin facade's implementation class is `OPCAdminFacadeEJB` . The admin application is a client of the `OPCAdminFacade` in the order processing application. A structure diagram of how the Session Facade interacts with other classes in the order processing application appears in Figure 2 below. The greyed-out class in the diagram indicates the transfer object created by the facade class; see the Transfer Object pattern for details.

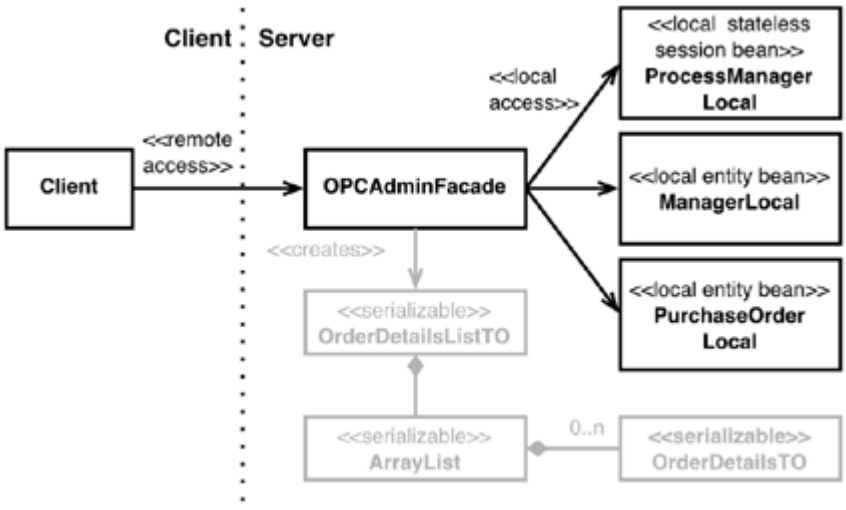


Figure 2. OPCAdminFacade provides an interface to complex interactions between components

The key point to notice here, is that the client only interacts with the `OPCAdminFacade`, and the other Enterprise Beans are hidden behind the facade. So the client has a simple interface and is not exposed to the complexity of the application logic that occurs to process the request. One of `OPCAdminFacade`'s methods, `getOrdersByStatus`, builds a summary of purchase information for display by the sample application's admin client. This method uses a `ProcessManagerLocal`

stateless session bean and `ManagerLocal` local entity beans and iterates over a collection of `PurchaseOrderLocal` entity beans of the requested status. The method creates a serializable `Collection` of `OrderDetails` transfer objects that describe `PurchaseOrderLocal` beans found. (See the Transfer Object pattern for details). The method then returns the collection of objects to the client.

- **The Java Pet Store website sample application shopping facade.**

The Java Pet Store website application also uses the Session Facade pattern. `ShoppingClientFacadeLocal` is a local stateful session bean that centralizes the services that the Web site provides to shoppers. Its local interface appears below:

```
public interface ShoppingClientFacadeLocal extends EJBLocalObject {

    public ShoppingCartLocal getShoppingCart();
    public void setUserId(String userId);
    public String getUserId();
    public CustomerLocal getCustomer() throws FinderException;
    public CustomerLocal createCustomer(String userId);

}
```

The shopping facade provides a single interface to the client for all client shopping functionality: the shopping cart, storing the user's id, and finding and creating the `Customer` data associated with the user. The facade is a stateful session bean because it stores information (such as the user id) that is specific to an individual user in the context of a session.

Singleton

Intent
Ensure a class has only one instance, and provide a global point of access to it.

Problem
Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Discussion
Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Singleton should be considered only if all three of the following criteria are satisfied:

- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for

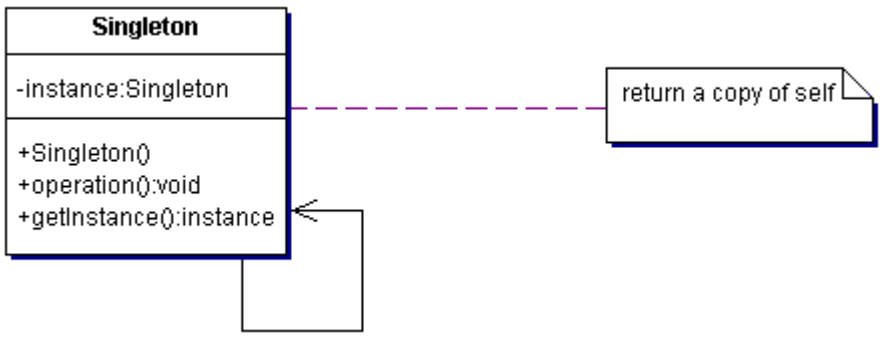
If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

The Singleton pattern can be extended to support access to an application-specific number of instances.

The "static member function accessor" approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.

Deleting a Singleton class/instance is a non-trivial design problem. See "To kill a Singleton" by John Vlissides (C++ Report, Jun 96, pp10-19) for a discussion.

Structure



Example
The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States

Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Opinions

"Singleton is not going to solve global data problems. When I think of Singleton, I think of a pattern that allows me to ensure a class has only one instance. As far as global data, the only benefit I see is reducing the namespace." [Christopher Parrinello]

"This is not really the point, but I always teach Composite, Strategy, Template Method, and Factory Method before I teach Singleton. They are much more common, and most people are probably already using the last two. The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances." [Ralph Johnson]

Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away. The answer to the global data question is not, "Make it a Singleton." The answer is, "Why in the hell are you using global data?" Changing the name doesn't change the problem. In fact, it may make it worse because it gives you the opportunity to say, "Well I'm not doing that, I'm doing this" - even though this and that are the same thing. [Frieder Knauss]

Rules of thumb

Abstract Factory, Builder, and Prototype can use Singleton in their implementation. [GOF, p134]
Facade objects are often Singletons because only one Facade object is required. [GOF, p193]

State objects are often Singletons. [GOF, p313]

Template Method

Intent
Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

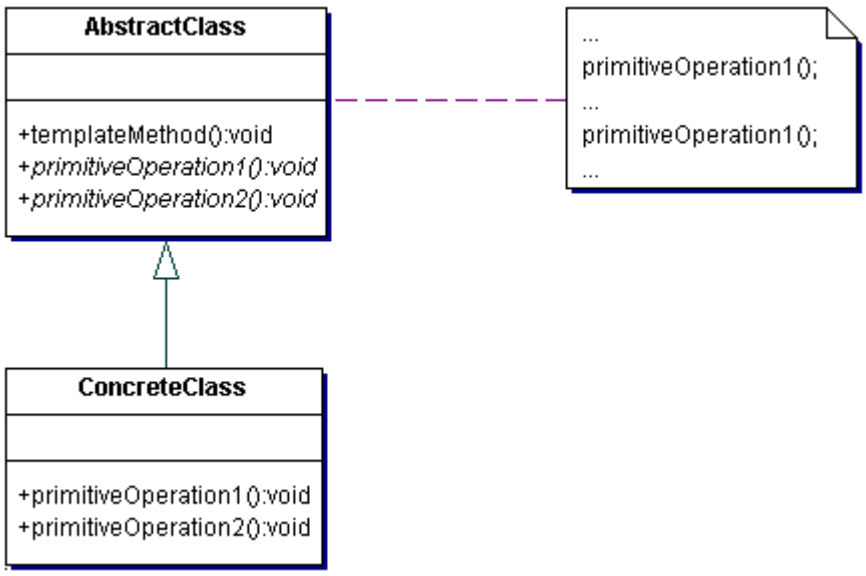
Problem
Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

Discussion
The component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable). The invariant steps are implemented in an abstract base class, while the variant steps are either given a default implementation, or no implementation at all. The variant steps represent "hooks", or "placeholders", that can, or must, be supplied by the component's client in a concrete derived class.

The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

Template Method is used prominently in frameworks. Each framework implements the invariant pieces of a domain's architecture, and defines "placeholders" for all necessary or interesting client customization options. In so doing, the framework becomes the "center of the universe", and the client customizations are simply "the third rock from the sun". This inverted control structure has been affectionately labelled "the Hollywood principle" - "don't call us, we'll call you".

Structure



Example
The Template Method defines a skeleton of an algorithm in an operation, and defers some steps to subclasses. Home builders use the Template Method when developing a new subdivision. A typical subdivision consists of a limited number of floor plans with different variations available for each.

Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house. Variation is introduced in the later stages of construction to produce a wider variety of models. [Michael Duell, "Non-software examples of software design patterns", Object Magazine, Jul 97, p54]

Rules of thumb

Strategy is like Template Method except in its granularity. [Coplien, C++ Report, Mar 96, p88]

Template Method uses inheritance to vary part of an algorithm. Strategy uses delegation to vary the entire algorithm. [GOF, p330]

Glossaire

classe abstraite	Une classe qui ne peut pas être instanciée directement. Par opposition à une classe concrète
agrégation	Une forme spéciale d'association qui spécifie une relation "tout-partie" entre l'agrégat (le tout) et une partie.
association	Relation sémantique entre deux ou plusieurs classes. C'est une connexion, bidirectionnelle par défaut, entre leurs instances. Sous–type de Relation.
checked exception	Exception héritant de java.lang.Exception. Le compilateur oblige le developpeur à les traiter (soit par un catch soit par un throws)
composition	Une forme d'agrégation qui exprime une forte propriété entre le tout et les parties, ainsi qu'une subordination entre l'existence des parties et du tout. Les parties, dont la multiplicité est non figée, peuvent être créées après le composite lui–même, mais une fois créées, elles vivent et meurent avec lui (c'est–à–dire qu'elles partagent sa durée de vie). De telles parties peuvent également être explicitement retirées avant la mort du composite. La composition peut être récursive.
eXtreme Porgramming	Ensemble de pratiques qui couvre une grande partie des activités de la réalisation d'un logiciel : de la planification du projet au développement à proprement dit, en passant par les relations avec le client ou l'organisation de l'équipe.
loose coupling	Couplement lâche
package	Paquetage
refactoring	Remaniement de code
relation	Une connexion sémantique entre les éléments de modélisation. Les relations comprennent les associations et les généralisations.
scalability	Extensibilité
tier	Couche
unchecked exception	Exception héritant de java.lang.RuntimeException. Le compilateur n'oblige pas le developpeur à les traiter (soit par un catch soit par un throws). Il peut cependant le faire.
use case	Cas d'utilisation
Markup Interface	

Acronyme

ADL	Architecture Description Language
Ant	Another Neat Tool
AOP	Aspect Oriented Programming
API	Application Programming Interface
ASP	Active Server Page
AWT	Abstract Windowing Toolkit
B2B	Business to Business
BLOB	Binary Large OBject
BMP	Bean Managed Persistent
CGI	Common Gateway Interface
CLOB	Character Large OBject
CMP	Container Managed Persistent
CMR	Container Managed Relation
CMT	Container Managed Transaction
CORBA	Common Object Request Broker Architecture
CRUD	Create/Read/Update/Delete
DAO	Data Access Object
DOM	Document Object Model
DORAFIA	Design One, Re-implement a Few Interfaces Anywhere
DTD	Document Type Definition
DTO	Data Transfert Object
EAR	Enterprise Archive
EIS	Enterprise Information System
EJB	Enterprise Java Bean
EJBQL	Enterprise Java Bean Query Language
EL	Expression Langage
FYI	For Your Information
GoF	Gang of Four
GPL	General Public License
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure sockets
IDE	Integrated Development Environment
IIOP	Internet Inter-ORB Protocol
IMHO	In My Humble Opinion
IoC	Inversion Of Control
IP	Internet Protocol
JAR	Java Archive
JCA	Java Connector Architecture
JDBC	Java Data Base Connectivity
JDK	Java Development Kit
JDO	Java Data Object
JFC	Java Foundation Classes
JMS	Java Messaging Services
JNDI	Java Naming Directory Interface
JNLP	Java Network Lauching Protocol

JRE	Java Runtime Environment
JRMP	Java Remote Method Protocol
JSP	Java Server Page
JSR	Java Specification Request
JSTL	JSP Standard Tag Library
JVM	Java Virtual Machine
JWS	Java Web Start
LDAP	Lightweight Directory Access Protocol
LGPL	Lesser General Public License
LOC	Line Of Code
MDA	Model Driven Architecture
MDPB	Model-Driven, Pattern-Based
MOM	Message Oriented Middleware
ODBC	Open Database Connectivity
OMA	Object Management Architecture
OMG	Object Management Group
PDA	Personal Digital Assistant
PHP	PHP: Hypertext Preprocessor
POJO	Plain Old Java Object
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
UML	Unified Modelling Language
URL	Uniform Resource Locator
VO	Value Object
WAP	Wireless Application Protocol
WAR	Web Archive
WORA	Write Once Run Anywhere
XML	eXtended Markup Language
XP	eXtreme Programming
XPath	XML Path Language
YAGNI	You Ain't Gonna Need It
YAPS	Yet Another PetStore

Discussion

References

	Effective Use of UML for Software Architecture Design *Christine Hofmeister, Robert Nord, Dilip Soni* http://www.cs.york.ac.uk/uml2000/hofmeister.html
[SAUML]	**Software Architecture and the UML**

	Grady Booch http://www.ecestudents.ul.ie/Course_Pages/MEng_CS/Modules/EE6421/Examples/UML/arch.ppt
	UML and Design Patterns in Final Projects *Asher Sterkin* http://www.hadassah-col.ac.il/cs/staff/asterkin/advCPlusProg/Uml%20and%20Design%20Patterns.ppt
[XP]	**eXtreme Prgramming** http://www.extremeprogramming.org/
[REFACT]	**Refactoring** http://www.refactoring.com/
	Naming Conventions for Enterprise Applications http://java.sun.com/blueprints/code/namingconventions.html
	Code Conventions for the Java Programming Language http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
	Writing Robust Java Code Scott W. Ambler http://www.ambysoft.com/javaCodingStandards.html