J2EE AntiPatterns

Bill Dudney

Object Systems Group

bill@dudney.net

Agenda

- What is an AntiPattern?
- What is a Refactoring?
- AntiPatterns & Refactorings
  - Persistence
  - Service Based Architecture
  - JSP & Servlet
  - EJB Entity
  - EJB Session
  - Message Driven Beans
  - Web Service

What Is an AntiPattern?

- Recurring Solution with negative outcome
  - *i.e.* "I've done the wrong thing lots of times, don't repeat my mistakes"
- Consists of:
  - Name
  - Catalog Items
    - Also Known As
    - Refactorings
    - Anecdotal Evidence
  - Background
  - General Form

What Is an AntiPattern? *(Continued)*

- Symptoms & Consequences
- Typical Causes
- Known Exceptions
- Refactorings
- Variations
- Example
- Related Solutions

AntiPattern – Covered Items

- Name
- General Form
- Symptoms & Consequences
- Refactorings
- Example

What Is Refactoring?

- A means to improve the design of existing software without breaking (*i.e.* rewriting) every piece of code that uses the refactored code.
- Consists Of:
  - Before and After Avatar
    - Sometimes UML
    - Sometimes Code
  - Motivation
    - To get out of the AntiPatterns
  - Mechanics
  - Example

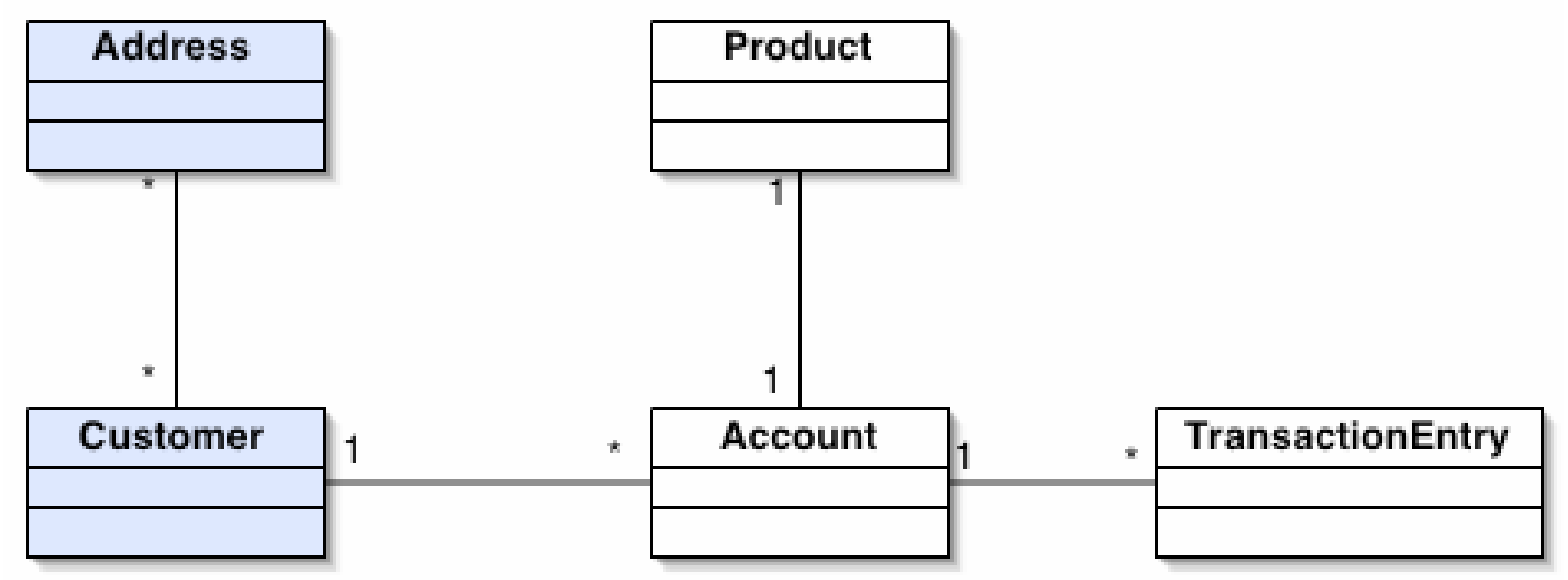
Persistence AntiPatterns

- Dredge – Don't fetch the whole database
- Stifle – Don't ignore JDBC performance techniques

AntiPattern: Dredge

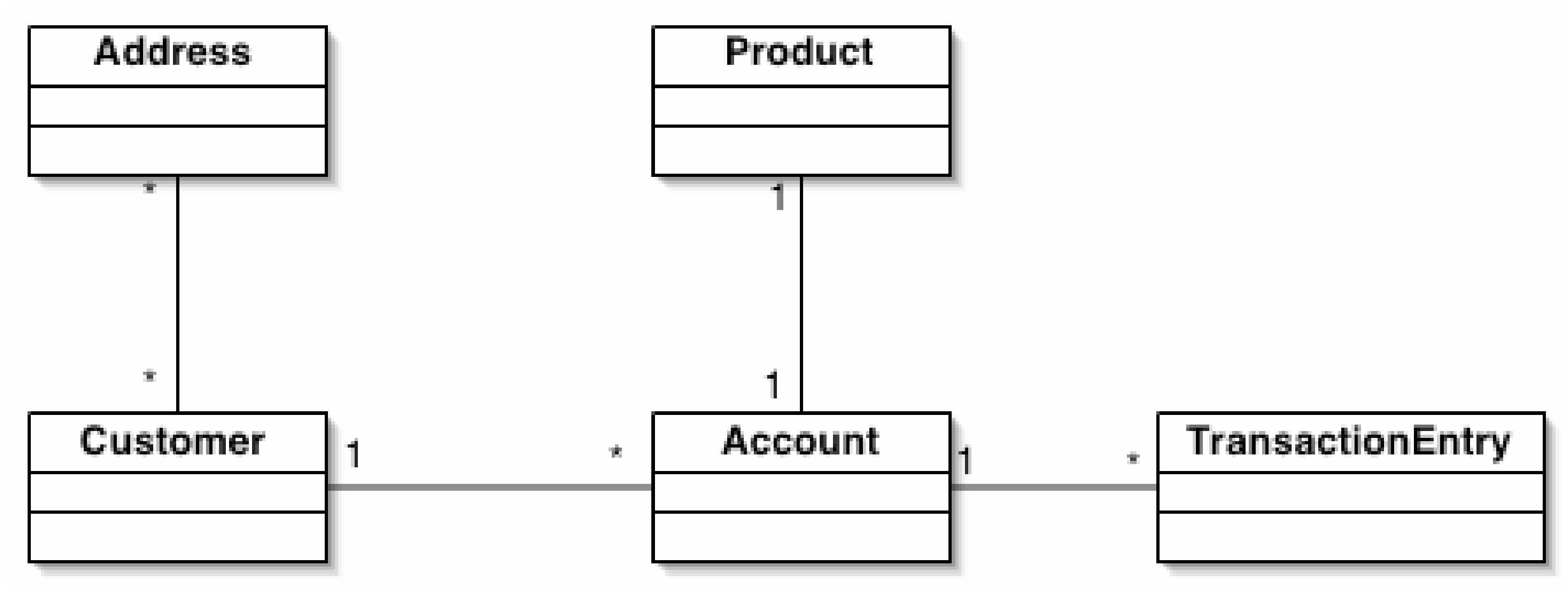
- General Form
  - Long Lists of EJB Entities
  - Deep Graphs of EJB Entities
- Symptoms & Consequences
  - Huge Memory Footprint
  - Poor Performance
- Refactorings
  - Light Query

Dredge – Example



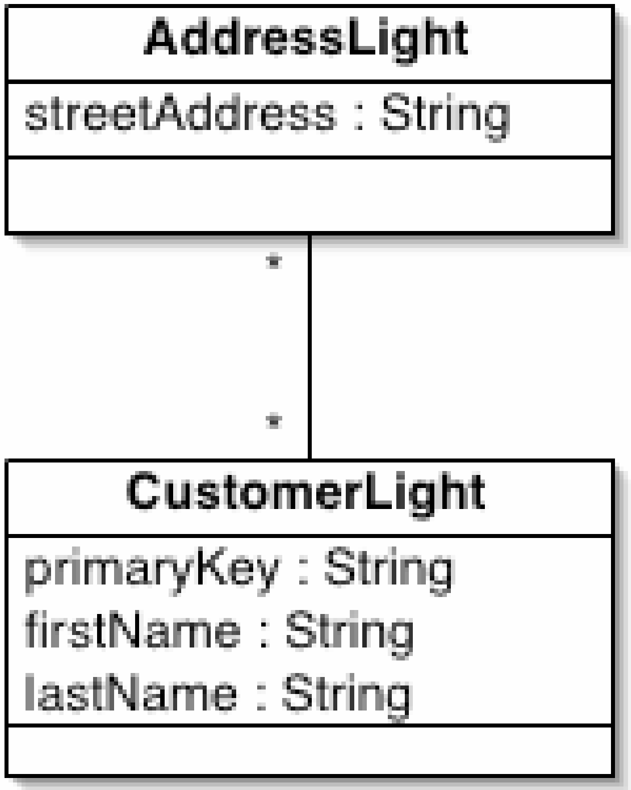
Refactoring: Light Query

- Before



Refactoring: Light Query

- After



Light Query – Mechanics

- Identify the lists your application must display
  - Its usually best to start with a simple one, a list that displays a single entity
  - It might make sense to start with a more complex list if it is causing serious performance problems
- Locate the existing logic that generates the list

Light Query – Mechanics

- Introduce a light DTO to represent the custom row.
- Introduce or modify DTO and/or Session Façade
  - Make sure to use a light weight mechanism to get the data such as JDBC or your R/O mapping tools mechanism for light weight queries to populate the light DTOs

AntiPattern: Stifle

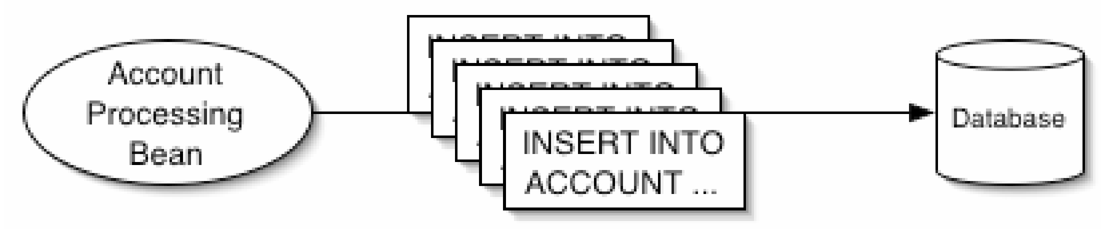
- General Form
  - Lack of JDBC batch processing
- Symptoms & Consequences
  - Poor Database Performance
  - Unhappy Users – loss of confidence
- Refactorings
  - Pack

Stifle – Example

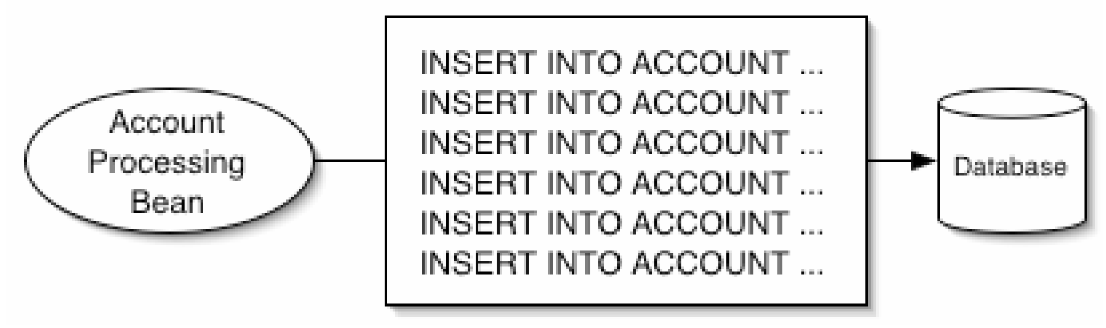
```
// Extract and loop through account data
while(accountIter.hasNext())
{
...
Statement statement = conn.createStatement();
int rowsAffected =
   statement.executeUpdate("UPDATE ACCOUNT SET
   ...");
...
}
```

Refactoring: Pack

- Before



- After



Pack – Mechanics

- Change your looped statement execution to addBatch calls
  - ➢Remember to set a batch size and execute the batch ever size steps
- Call executeBatch on the statement
  - ➢Make sure to execute the batch on a regular basis so that it does not get too big
- Deploy & Test

Service-Based Architecture AntiPatterns

- Stove Pipe – Don’t rebuild the technical details for every service
- Client Completes Service – Don’t build services that are incomplete

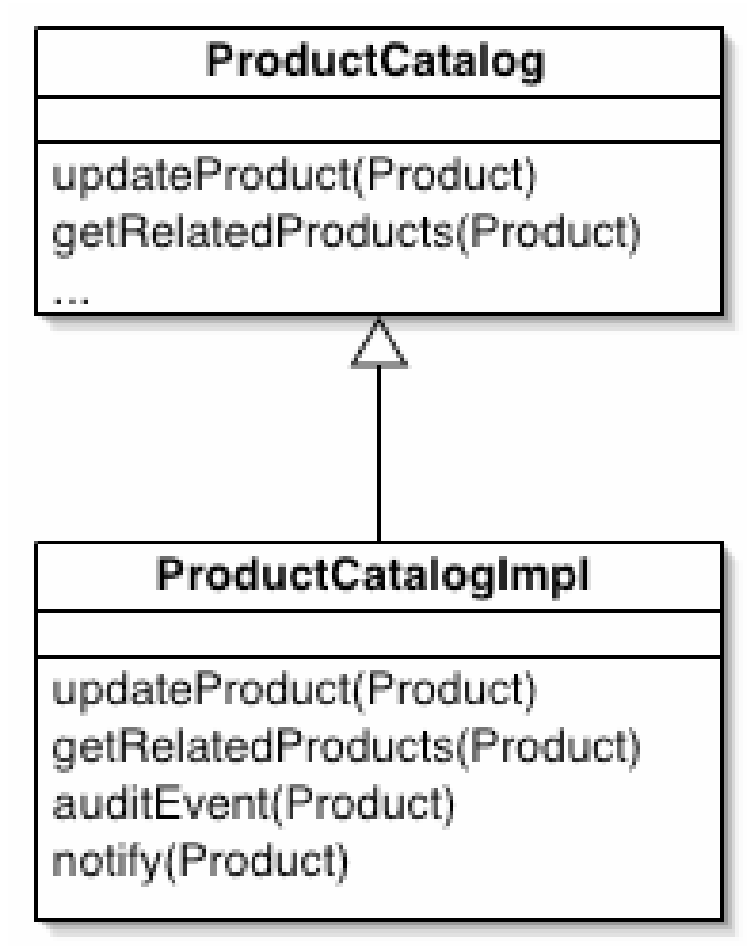
AntiPattern: Stove Pipe

- General Form
  - Lots of private technical services oriented methods
  - Duplicated implementation effort across services
- Symptoms & Consequences
  - Service is large with many methods not directly related to the interface
  - Inconsistent implementations across various services of the technical services
  - Development time is negatively impacted

AntiPattern: Stove Pipe

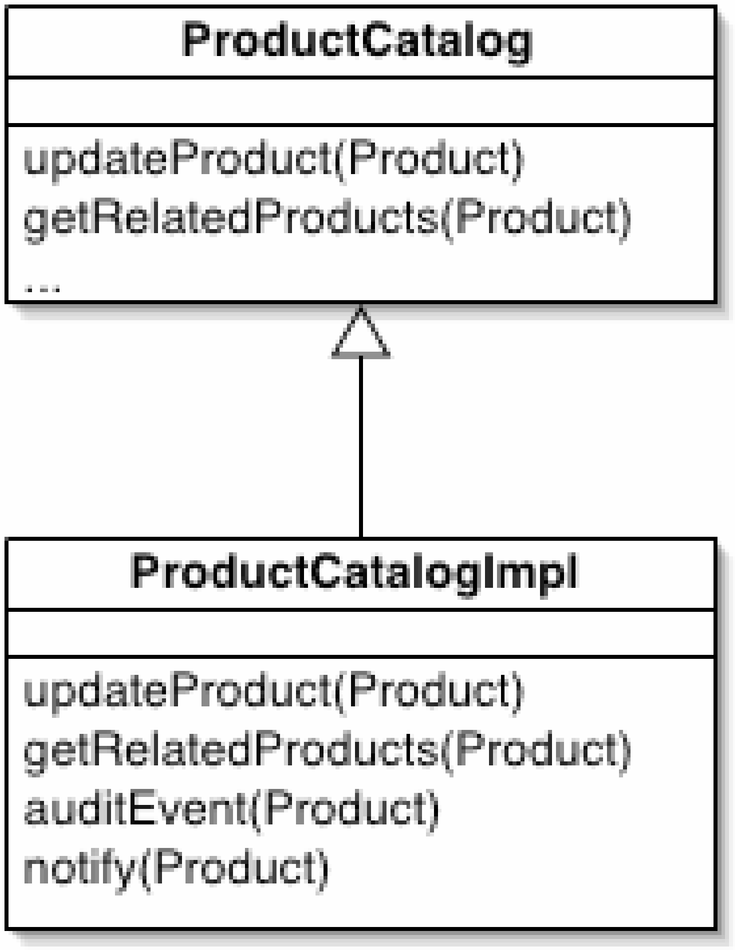
- Refactorings
  - Build Technical Services Layer

Stove Pipe – Example



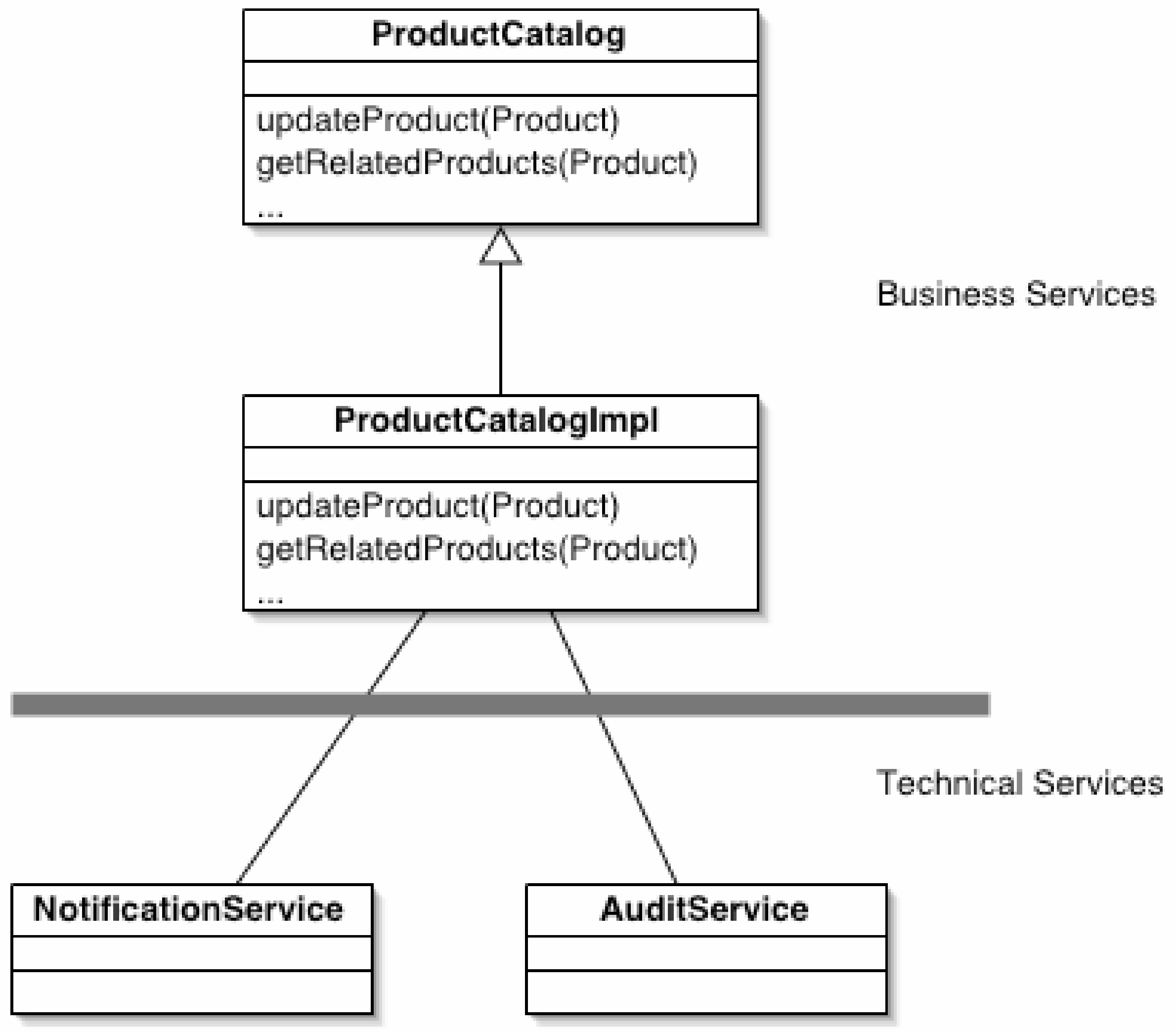
Refactoring: Build Technical Services Layer

- Before



Refactoring: Build Technical Services Layer

- After



Build Technical Services Layer – Mechanics (1 of 3)

- Review current services for duplicate private methods.
  - This can be very difficult especially if the services were implemented by different groups
  - Look for similar names
  - Look for similar functionality
- Start with the simplest functionality that is duplicated
- Apply Fowler's Extract Interface refactoring
  - Instead of making your service implement the interface, use it, you should use the new interface as a replacement for the duplicate code.

Build Technical Services Layer – Mechanics (2 of 3)

- Implement the newly defined service interface
  - Start with moving the method from the business service's implementation to the technical service's implementation
  - You can use Fowler's Move Method here
  - Many any necessary changes to get the business service to use the new technical service
- Deploy and Test

Build Technical Services Layer – Mechanics (3 of 3)

- After all tests pass, review the other business services with implementations of the technical service and refactor them to use the new technical service
  - This is a modified version of Move Method. Instead of physically moving the code, you will comment it out, then use the technical service.
  - Some adjustment may need to be made to the technical service to accommodate the various implementations – remember that you are striving for a uniform implementation that all services share.

AntiPattern: Client Completes Service

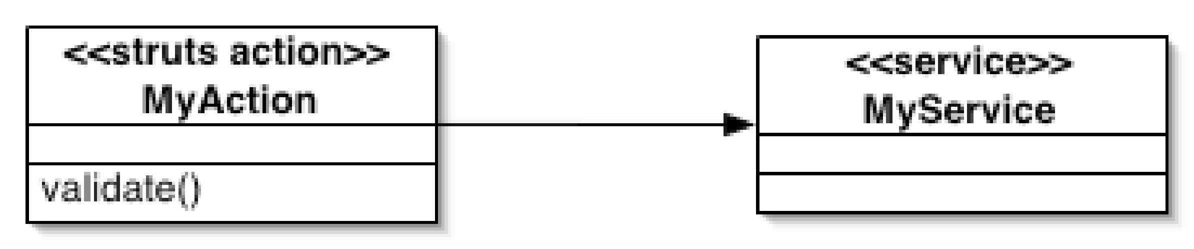
- General Form
  - Client Code includes service functionality
    - This can include items such as data validation, security checking or things related to technical services covered in the last AntiPattern
- Symptoms & Consequences
  - Some client side artifacts (JSPs, front controllers *etc.*) contain server-side code
  - Potentially different behavior when invoking a service *via* a Web-service interface and a user interface
- Refactorings
  - Move Method Cross Tier

Client Completes Service – Example

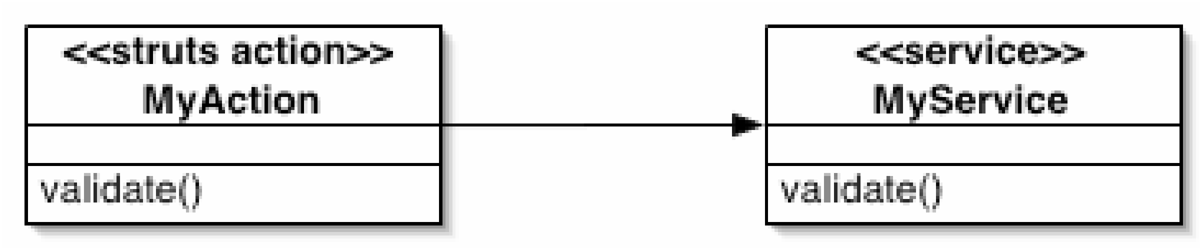
```
<%!
  List errors = null;
  if(value.intValue() > 5) {
    errors = (List)
        session.getAttribute("errors");
    errors.add("Invalid value");
  }
%>
```

Refactoring: Move Method Cross Tier

- Before



- After



Move Method Cross Tier – Mechanics

- Locate server side code in client artifacts
  - The artifacts can range from simple Java Beans to JavaScript in a JSP or HTML page
- Move code to Service Implementation
  - This can be difficult because of the widely varying client side artifacts that the implementation can be in.
  - For JavaBeans and Servlets you can use Fowler's Move Method
  - For JSPs you can use a modified Move Method
    - The code in the JSP has to be consolidated into a method first.
- Deploy & Test

JSP AntiPattern

- Too Much Data in Session – Not sure? Stick it in the session.

AntiPattern: Too Much Data in Session

- General Form
  - Lots of calls to getAttribute and setAttribute
  - Treatment of the Session as a global data space
- Symptoms & Consequences
  - Bugs related to different types being under the same key
  - Maintenance Headaches
- Refactorings
  - Beanify

Too Much Data in Session – Example

```
<% session.getAttribute("firstName"); %>
...
<% session.getAttribute("lastName"); %>
...
<% session.getAttribute("middleInitial"); %>
```

Refactoring: Beanify

- Before

```
<%
Boolean validUser = (Boolean)session.
getAttribute("validUser");
String buttonTitle = "Login";
String url = "Login.jsp";
if(null != validUser && validUser.booleanValue()) {
  buttonTitle = "Logout";
  url = "Logout.jsp";
}
%>
```

Refactoring: Beanify

- After

```
<jsp:useBean id="userCtx" class="ibank.web.UserContext"/>
...
<a class="BorderButton" href="${userCtx.nextNav}">
${userCtx.loginTitle}
</a>
```

Beanify – Mechanics (1 of 2)

- Create a JavaBean to hold the data
- Add an attribute to the bean for every unique key used in setAttribute or getAttribute
- Add a jsp:useBean to the JSP
- Remove all calls to getAttribute and replace them with expression language statements

Beanify – Mechanics (2 of 2)

- Remove all calls to setAttribute
  - If you are using the Delegate Controller pattern place the state change logic into your controller
  - If you are not using Delegate Controller consider refactoring to include this pattern and in the mean time use the jsp:setProperty tag
- Deploy and Test

Servlet AntiPattern

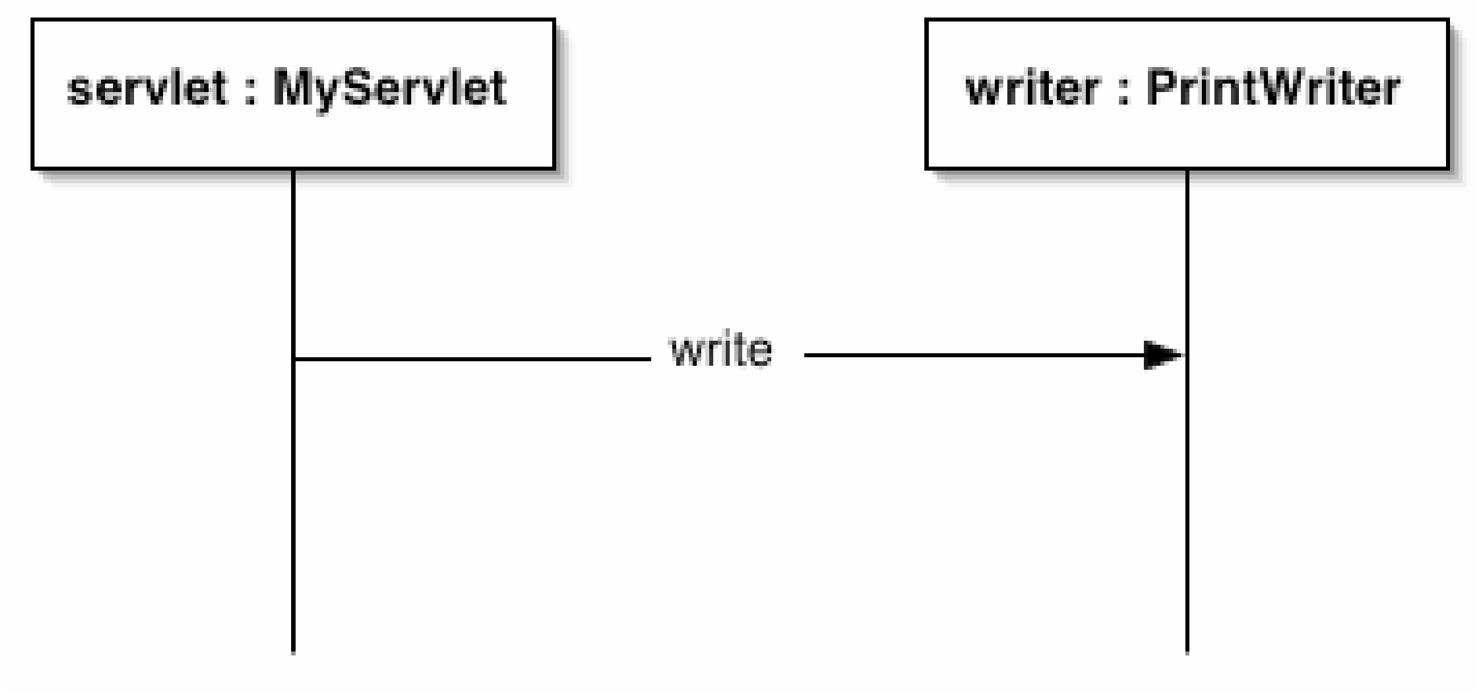
- Template Text In Servlet – looked like a good idea at the time…

AntiPattern: Template Text in Servlet

- General Form
  - Large Servlet classes with lots of static HTML in the form of strings
- Symptoms & Consequences
  - Low ratio of business logic to HTML
  - Maintenance Headaches
- Refactorings
  - Use JSPs

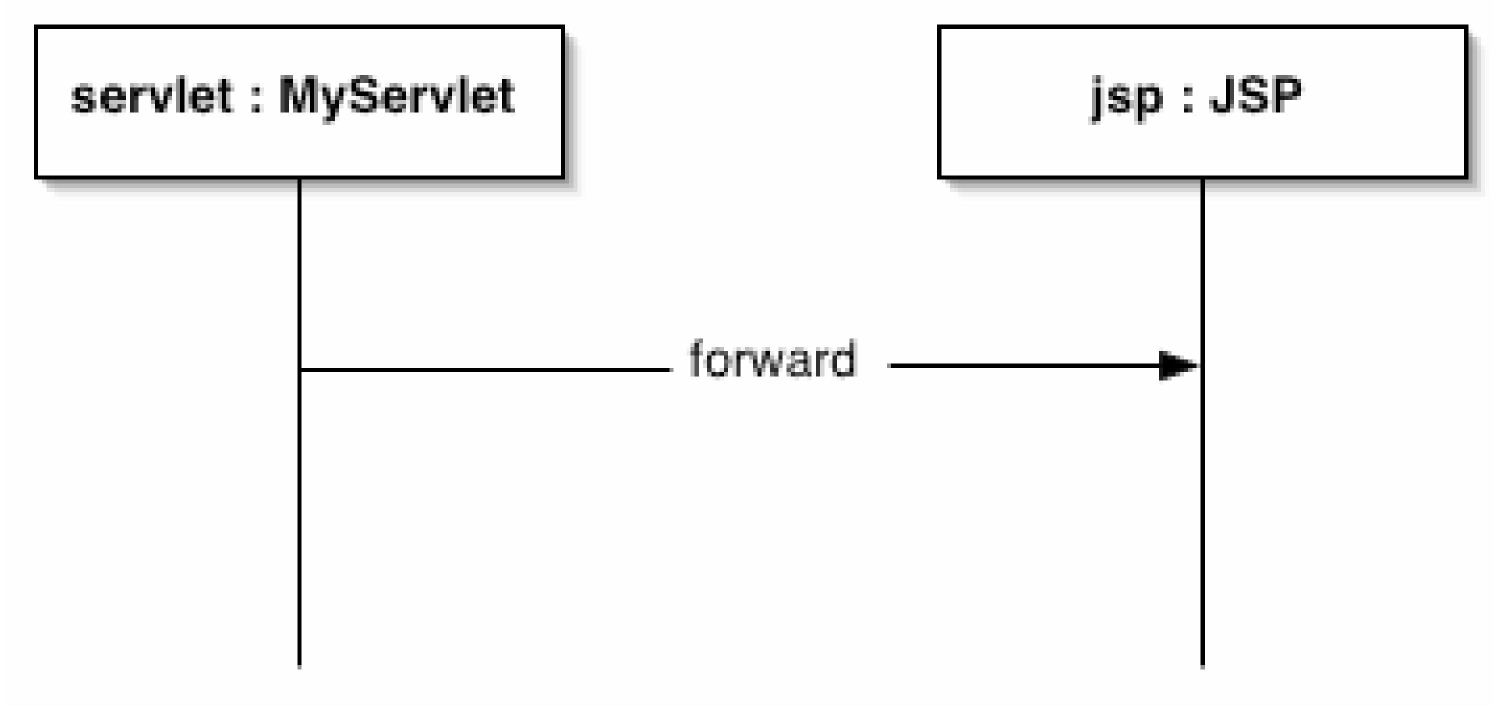
Refactoring: Use JSPs

- Before



Refactoring: Use JSPs

- After



Use JSPs – Mechanics (1 of 3)

- Save a copy of the HTML output from your Servlet
  - You can skip this step if you have a good set of tests
- Create a new JSP and copy all the obviously static HTML out of the Servlet and paste it into the JSP
  - Make note of dynamic content creation as you proceed. This dynamic behavior will have to be melded with the JSP *via* a JavaBean.

Use JSPs – Mechanics (2 of 3)

- Define a JavaBean to be populated by the Servlet and used by the JSP.
  - This bean will hold the data and possibly some of the behavior from the Servlet
  - You might have to apply Fowler's Move Method to get some of the functionality in the Servlet into the bean
- Add a jsp:useBean action to the new JSP to use the freshly created bean.
- Change the Servlet to create and populate the bean and place it under request scope in the session.

Use JSPs – Mechanics (3 of 3)

- Comment out the static generation code from the Servlet
- Change the Servlet to forward to the JSP
- Deploy and Test
  - You can use the copy of the HTML output you saved earlier as a visual guide to the validity of your refactoring

EJB Entity AntiPatterns

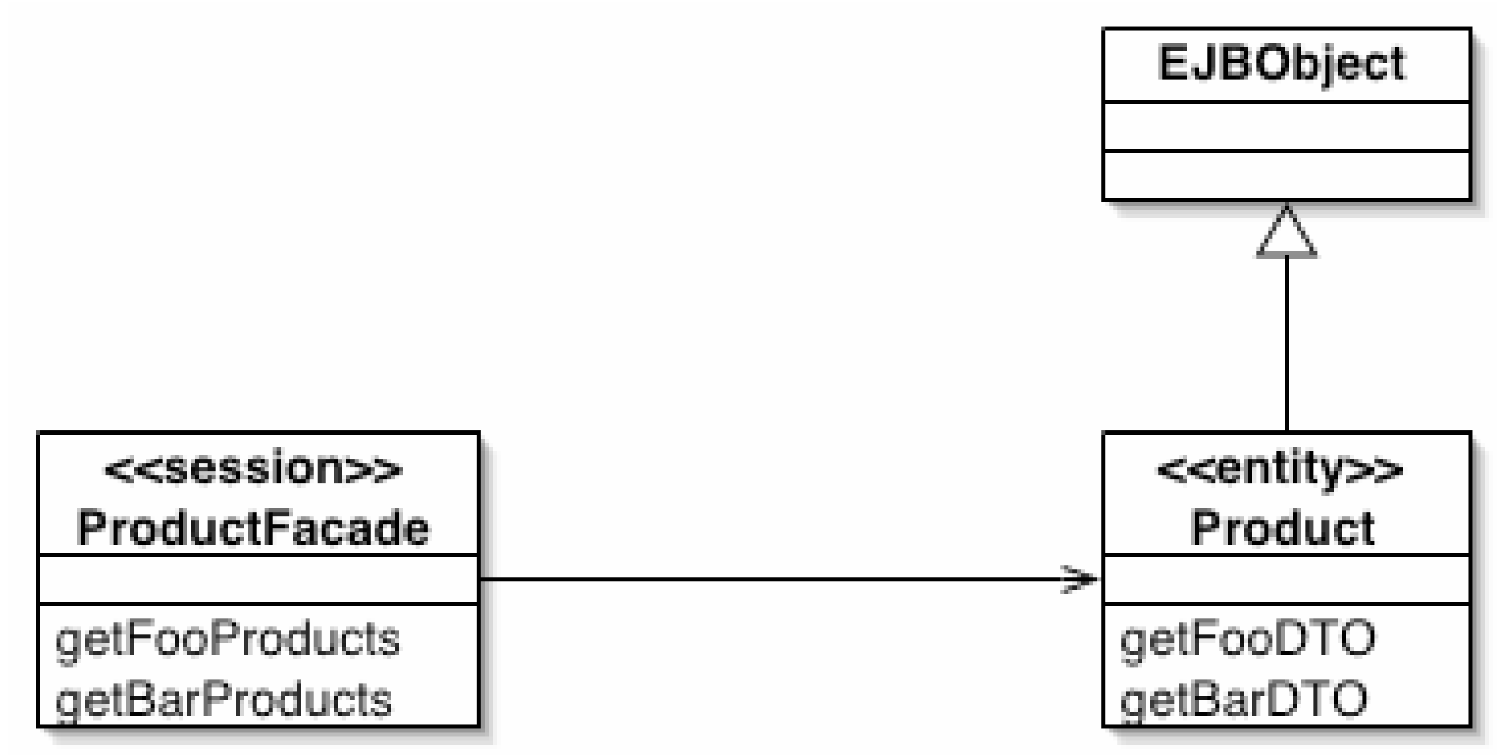
- DTO Explosion – A DTO for every occasion
- Coarse Behavior – Too many abstractions in one place

AntiPattern: DTO Explosion

- General Form
  - EJB Entities providing more than one DTO
    - Usually one for each view or use case the Entity is involved in
  - Many many DTOs
- Symptoms & Consequences
  - Huge maintenance overhead in synchronizing the various DTOs with Entity changes.
  - Reduced usability of Entities because they are tied to a particular view types
- Refactorings
  - Localize Access

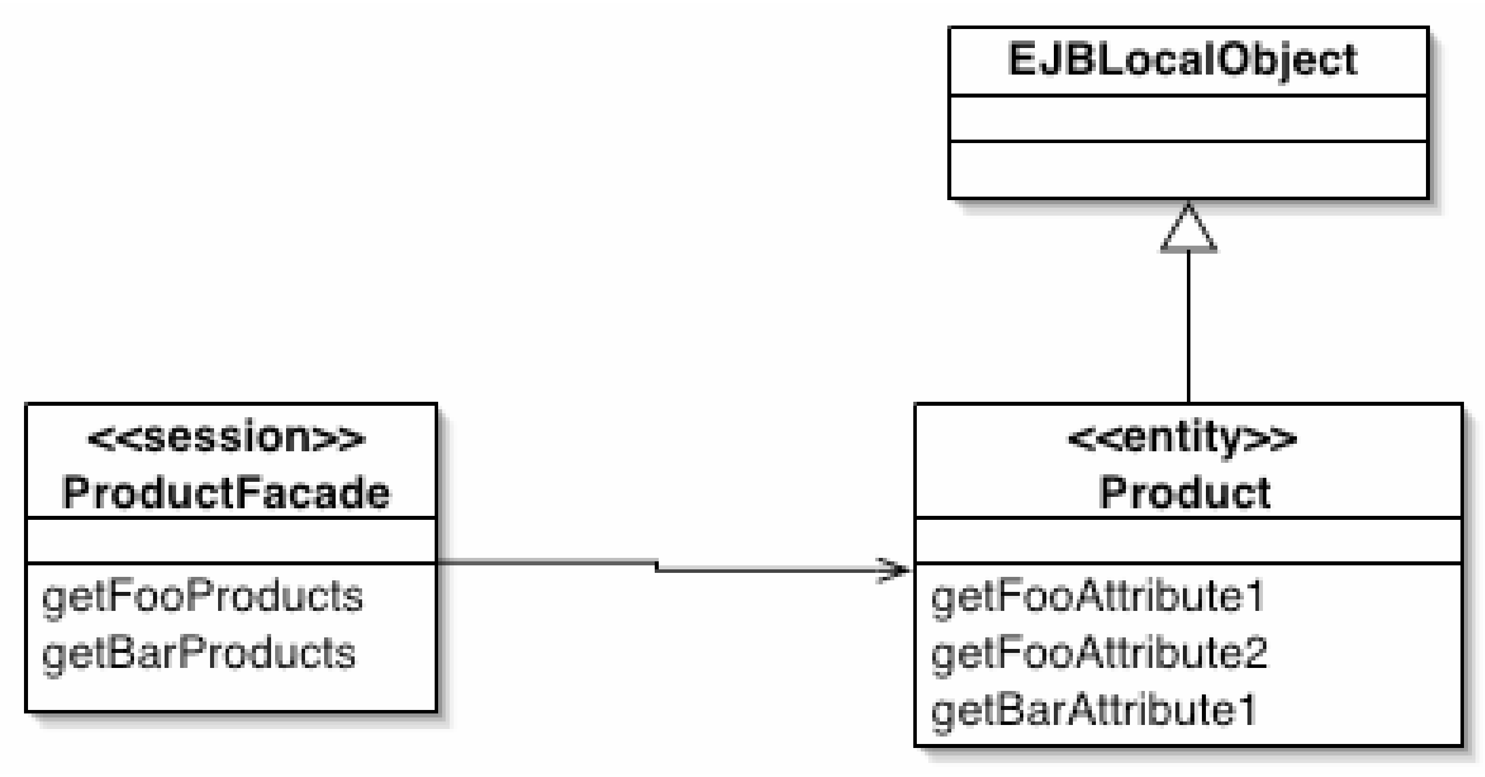
Refactoring: Localize Access

- Before



Refactoring: Localize Access

- After



Localize Access – Mechanics

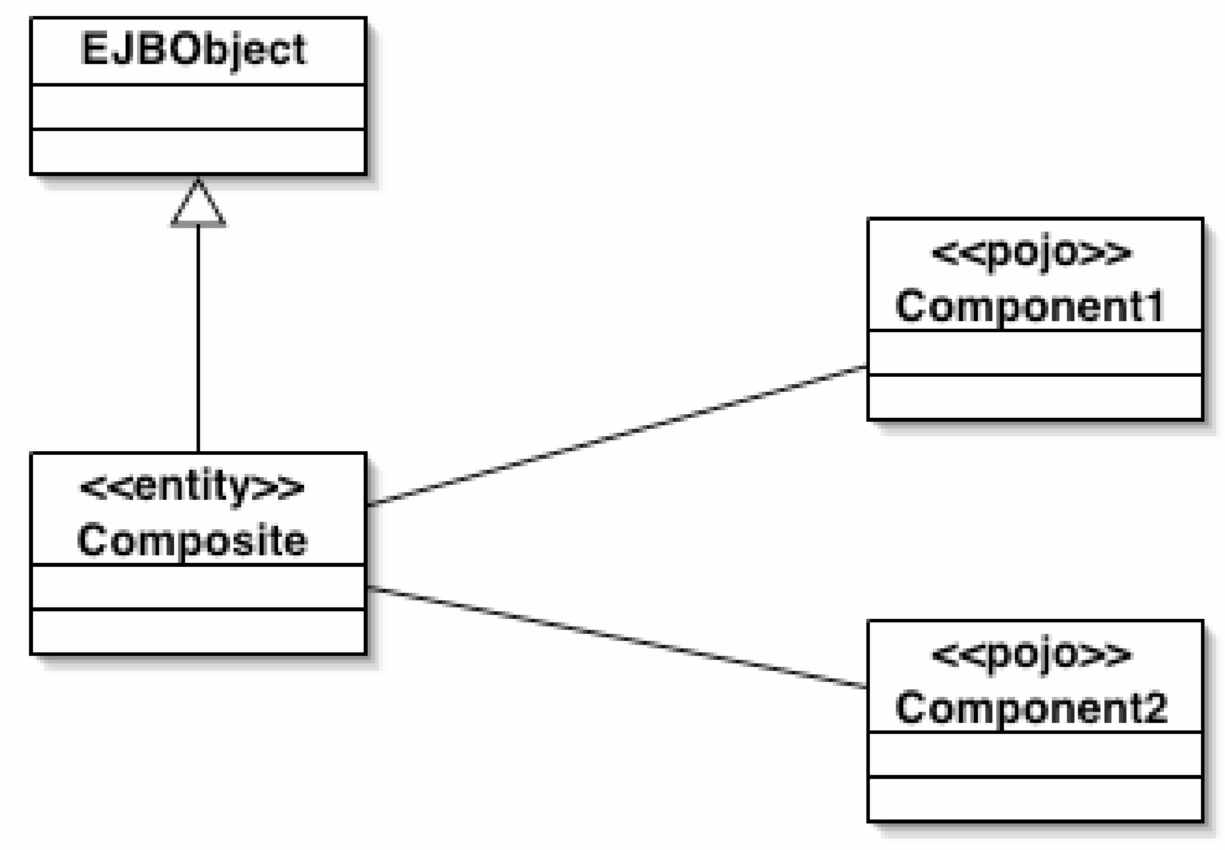
- Identify EJB Entities making view oriented DTO
- Change the Entities to local
- Use Move Method to move the DTO creation code to your session façade
  - This method will have to be updated to work with an instance of the Entity
  - You might also consider creating a DTO factory
- Deploy and Test

AntiPattern: Coarse Behavior

- General Form
  - Huge bloated EJB Entities following old style (EJB 1.x) patterns like Composite Entity
- Symptoms & Consequences
  - Increased Complexity
    - Difficult Maintenance
    - Increased Development Time
- Refactorings
  - Extract Entity

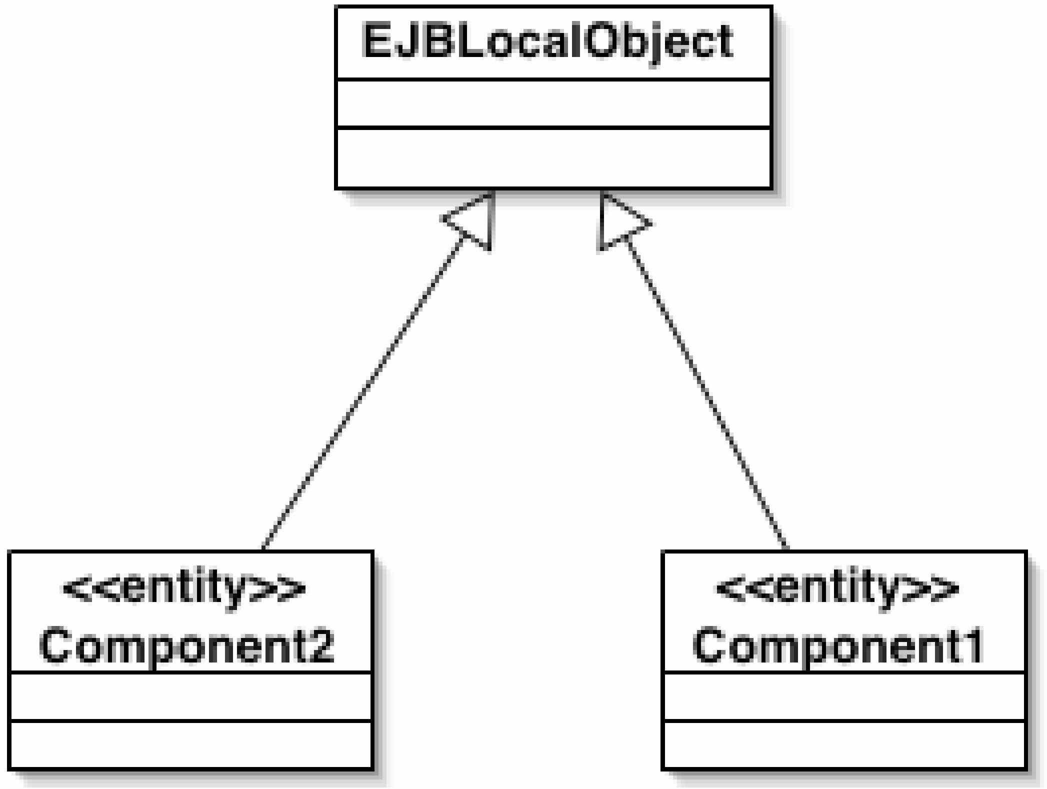
Refactoring: Extract Entity

- Before



Refactoring: Extract Entity

- After



Extract Entity – Mechanics (1 of 2)

- Apply Extract Interface for each POJO your composite Entity aggregates
  - These interfaces become your new EJBs interfaces
- Update each POJO to become a local EJB Entity
  - Use CMP where ever possible
  - If you can't use CMP, you will have to move the JDBC code from the existing composite to each of the newly created Entities
  - If you were using a relational object mapping tool (R/O) and you can't go to CMP, then you have to integrate the R/O persistence with your containers CMP, or roll your own with BMP

Extract Entity – Mechanics (2 of 2)

- Modify or create a session façade to provide clients with the existing functionality
  - If you have to create a session façade, then you should look at the Façade refactoring in Chapter 6 of the *J2EE AntiPatterns* book & the Session Façade pattern in the *Core J2EE* patterns book.
- Deploy and Test

EJB Session AntiPatterns

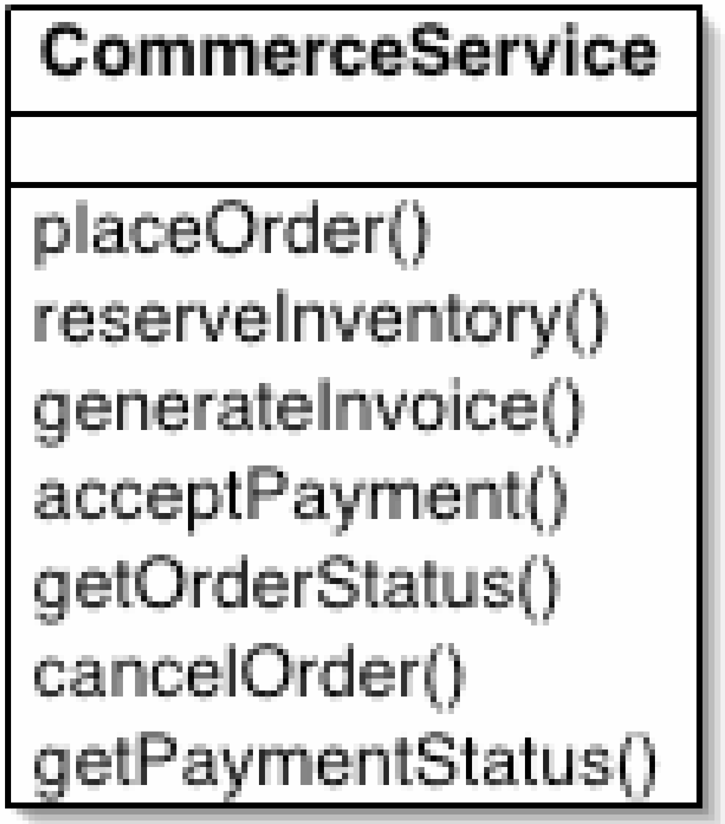
- Bloated Session – The kitchen sink
- Transparent Façade – Straight to the entity source

AntiPattern: Bloated Session

- General Form
  - Large API with many methods
- Symptoms & Consequences
  - Methods acting on different abstractions
    - *i.e.* part of the API works on orders, another works on accounts
  - Hard to understand and use API
    - Increased maintenance
- Refactorings
  - Interface Partitioning

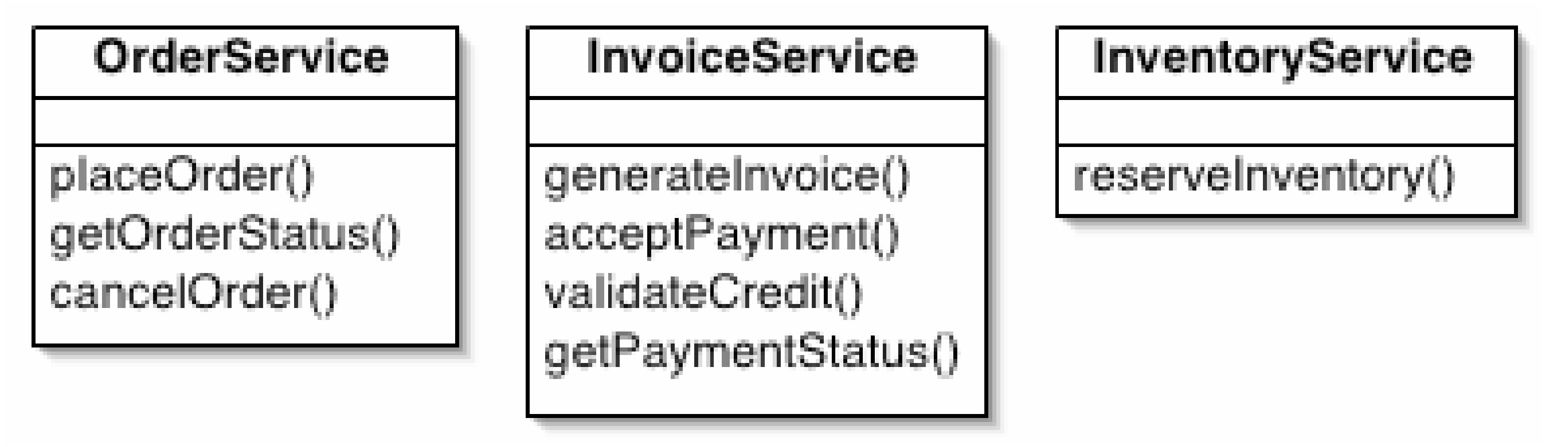
Refactoring: Interface Partitioning

- Before



Refactoring: Interface Partitioning

- After



Interface Partitioning – Mechanics (1 of 2)

- Identify each abstraction the service is acting on
  - Group the methods related to these other services together
  - You can start with the method names as a possible grouping mechanism, *i.e.* placeOrder, getOrderStatus, *etc.*
- Apply Extract Interface for each group

Interface Partitioning – Mechanics (2 of 2)

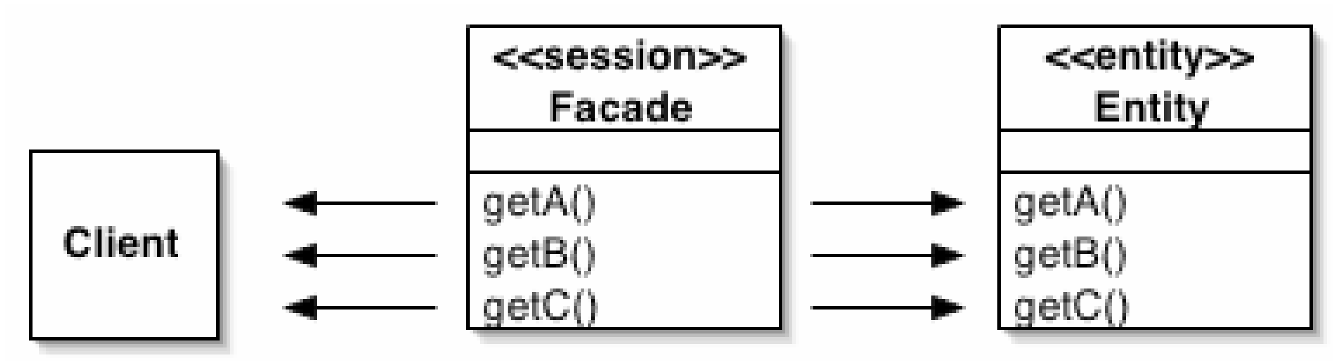
- Build a service around each new interface
  - Start with the simplest interface
- Refactor the original service to delegate the new service
- Refactor Clients to use the new service
  - This step should be done but is not required.
- Deploy and Test

AntiPattern: Transparent Façade

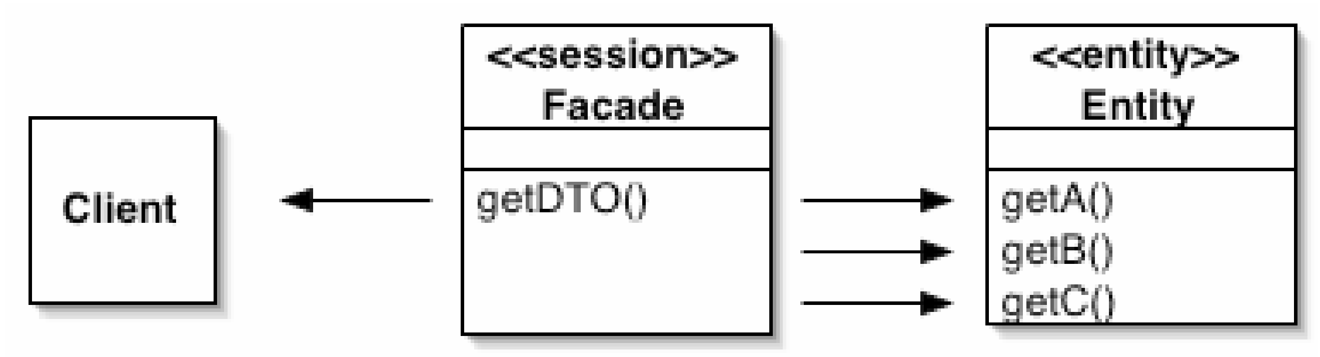
- General Form
  - Session directly delegates to underlying entity
- Symptoms & Consequences
  - Tight Coupling between Session and Entity
  - Poor performance
  - Increased Maintenance
- Refactoring
  - Right-size Session Façade

Refactoring: Right-size Session Façade

- Before



- After



Right-size Session Façade – Mechanics

- Determine what coarse grained behavior belongs on the Session Façade
  - You can start with clients of the existing façade: What methods do they use, and what do the clients do with the data they get back?
- Move the functionality from the clients to the façade
  - You might be able to apply Move Method here
- Refactor all clients to use the coarse grained behavior
  - Some of the functionality might have been implemented more than once
- Deploy and Test

Message Driven EJB AntiPattern

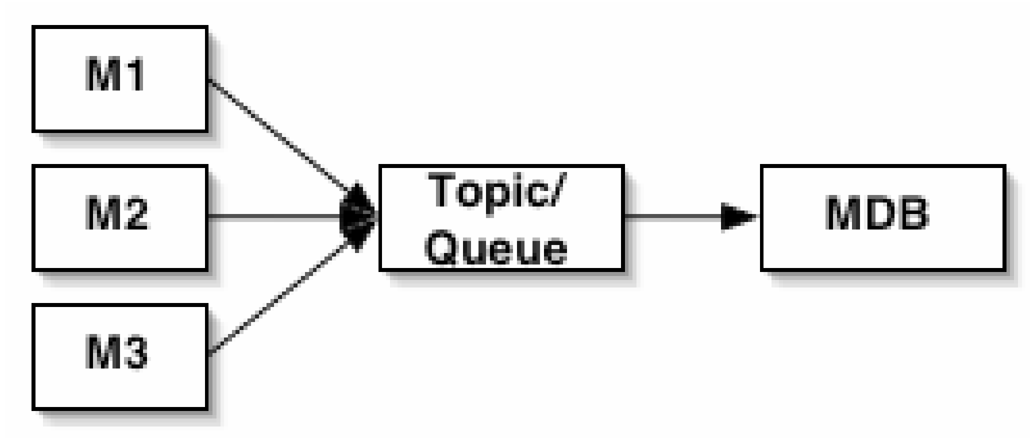
- Overloading Destinations – Why go through the trouble of another destination?

AntiPattern: Overloading Destinations

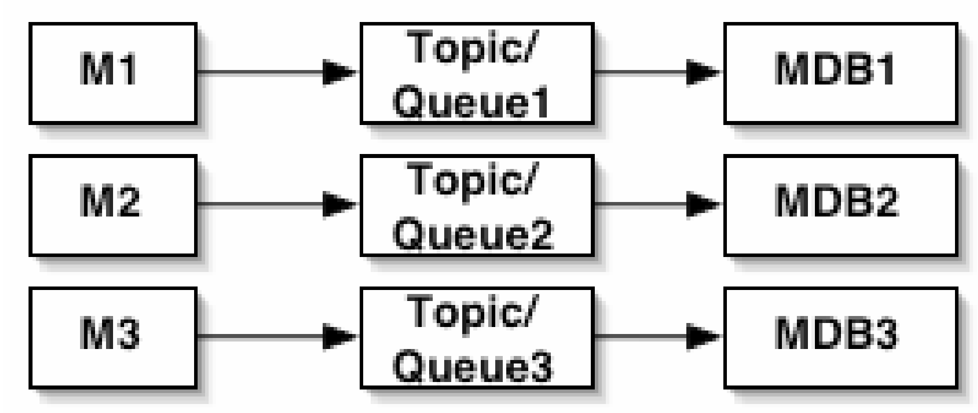
- General Form
  - Message Driven Bean that processes more than one type of message in its onMessage method
- Symptoms & Consequences
  - Poor performance
  - Difficult Maintenance
  - Bloat over time
- Refactorings
  - One Message One Bean

Refactoring: One Message One Bean

- Before



- After



One Message One Bean – Mechanics (1 of 2)

- For each message type your bean is processing, introduce a new bean
- Move each block of code that is dealing with each message type into the various beans
  - You can use Move Method here, with the change that you are not moving a whole method, just the content of an `if` block
- Modify deployment descriptor to deploy the new beans
  - In this step you will have to introduce all the new topics and/or queues requires as well

One Message One Bean – Mechanics (2 of 2)

- Refactor clients to use the new beans
  - This step will involve changing the topic/queue posted to
- Deploy and Test
  - Any unit tests for the old message driven bean can likely have Move Method applied to them to move the test to a different test class

Web Service AntiPatterns

- Omniscient Object – Everything to everyone
- Single Schema Dream – We'll make all the clients conform to this one schema

AntiPattern: Omniscient Object

- General Form
  - Large service implementation that spans business abstractions
  - Very similar to the Bloated Session AntiPattern
- Symptoms & Consequences
  - Multiple Document Types Exchanged
    - Increased complexity and thus maintenance
  - Reuse more difficult
- Refactorings
  - Interface Partitioning

Refactoring from Omniscient Object

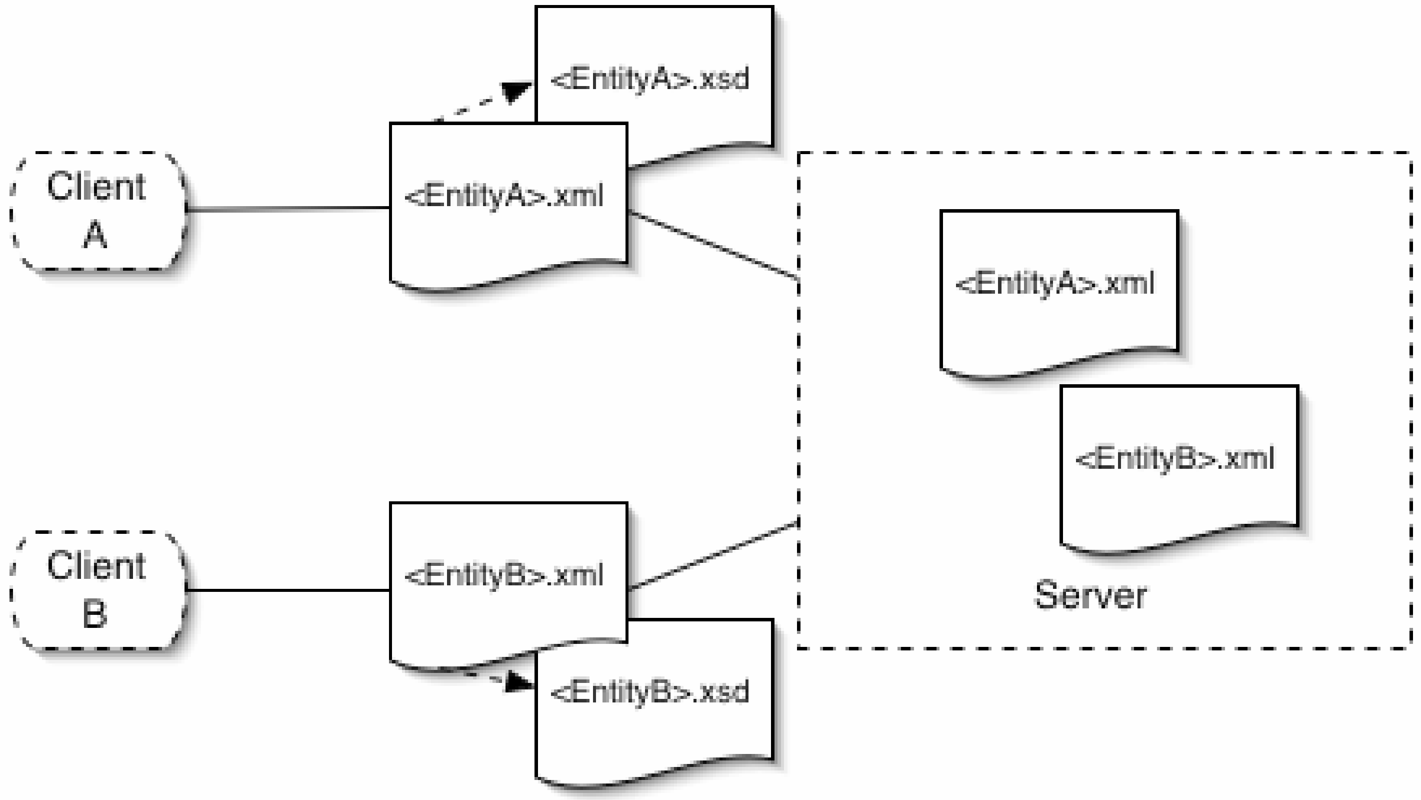
- Slightly modified, since the WSDL in addition to the interfaces and implementation will have to be modified
- The idea is the same, but the details will differ because of the additional artifacts associated with the Web service

AntiPattern: Single Schema Dream

- General Form
  - Schema changes often to accommodate new clients
  - Large `if...else if` blocks
- Symptoms & Consequences
  - Increased complexity in the service
  - Frequent client breakage
- Refactorings
  - Introduce Schema Adaptor

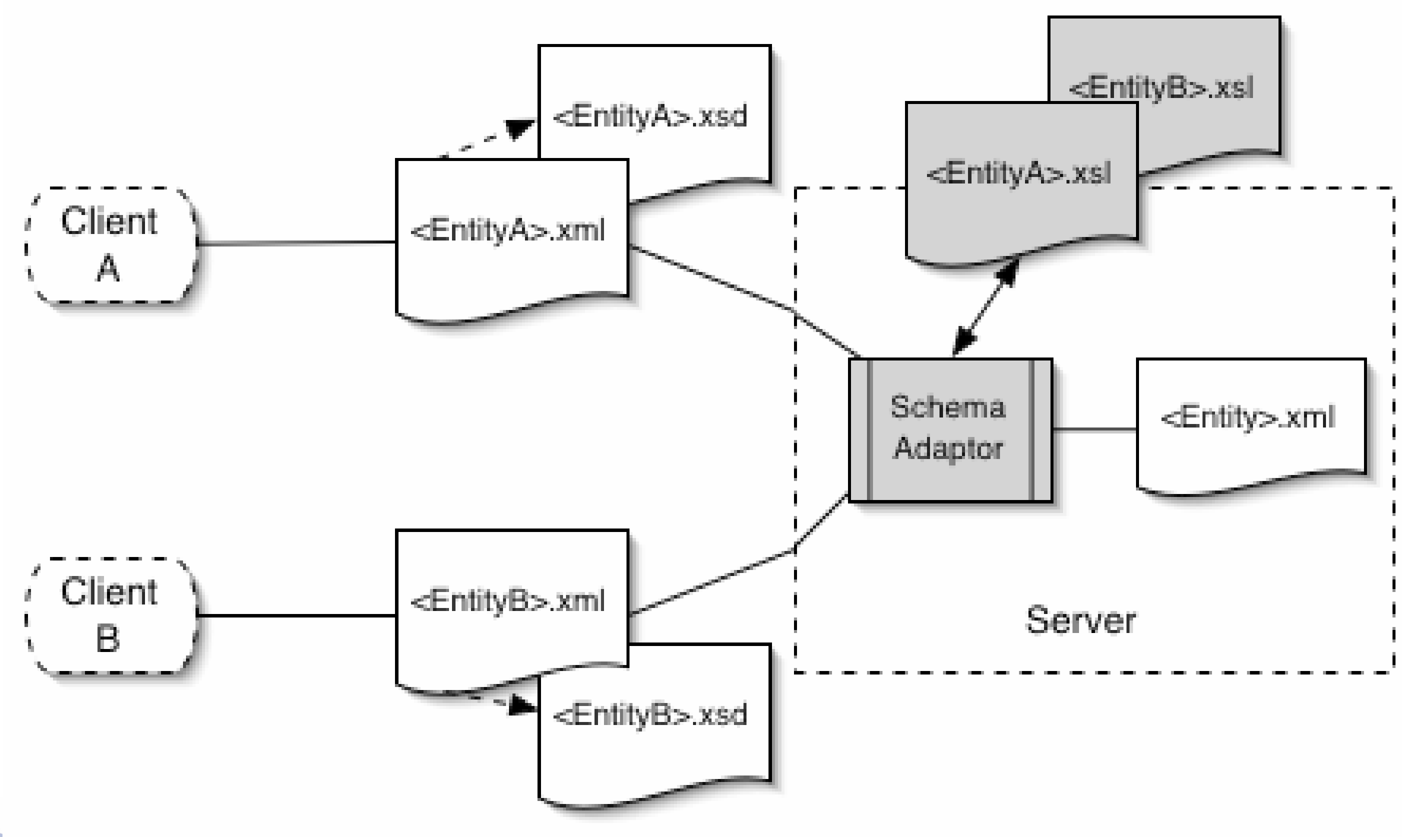
Refactoring: Introduce Schema Adaptor

- Before



Refactoring: Introduce Schema Adaptor

- After



Introduce Schema Adaptor – Mechanics (1 of 2)

- Implement the Schema Adaptor
  - All this really has to do is find the client specific XSL file and invoke the JAXP API
- Define and organize client specific transformations
  - The organization needs to be such that you can retrieve the client specific transformations from the adaptor

Introduce Schema Adaptor – Mechanics (1 of 2)

- Define client specific schemas
  - The schema adaptor will need access to the schemas as well, in order to validate the documents
- Update the Web service to use the schema adaptor.

References

- *J2EE AntiPatterns*
  Bill Dudney, Stephen Asbury, Joseph Krozak, Kevin Wittkopf
  John Wiley & Sons; First edition (August 11, 2003)
  ISBN: 0-47114-615-3
- *Jakarta Pitfalls: Time-Saving Solutions for Struts, Ant, JUnit, and Cactus (Java Open Source Library)*
  Bill Dudney, Jonathan Lehr
  John Wiley & Sons; (July 2003)
  ISBN: 0-47144-915-6