

ECP Design Workflow Walkthrough

version 1.1

Introduction

The aim of this document is to walk you through part of the design workflow for the ECP example from the point of view of the OO analyst/designer. This example should be read in conjunction with the book “UML and the Unified Process” [Arlow].

References

[Arlow] - UML and the Unified Process, Jim Arlow and Ila Neustadt, Addison Wesley, 2002, ISBN0201770601

[Alur] - Core J2EE™ Patterns, Deepak Alur et al, Sun Microsystems Press, 2002, ISBN0130648841

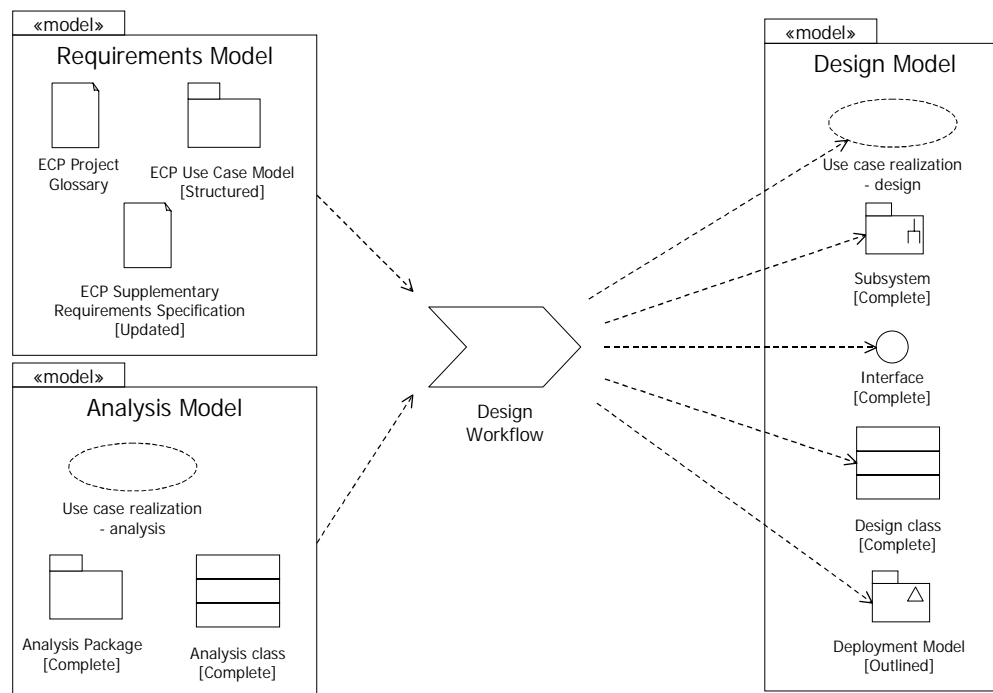
[ECPRequirements] - ECP Requirements Walkthrough, Clear View Training, 2002

[ECPAnalysis] - ECP Analysis Walkthrough, Clear View Training, 2002

Inputs and outputs

Here are the inputs and outputs to the UP Design Workflow for the ECP:

Figure 1



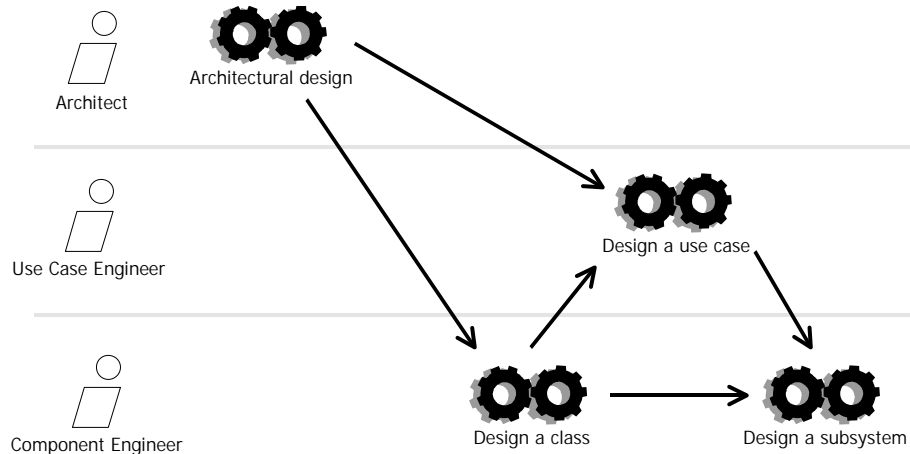
Scope

We will be designing a vertical slice through the ECP from the user interface down to the database. We are going to do this because we wish to mitigate technical risks around the use of Java 2 Enterprise Edition (J2EE) and the MySQL database. Creating this vertical slice will allow us to validate our choice of technology choices and also give the development team the opportunity to familiarize themselves with all the technology.

The Design Workflow

The Unified Process (UP) Design Workflow is shown below:

Figure 2



This workflow is made up of a number of activities:

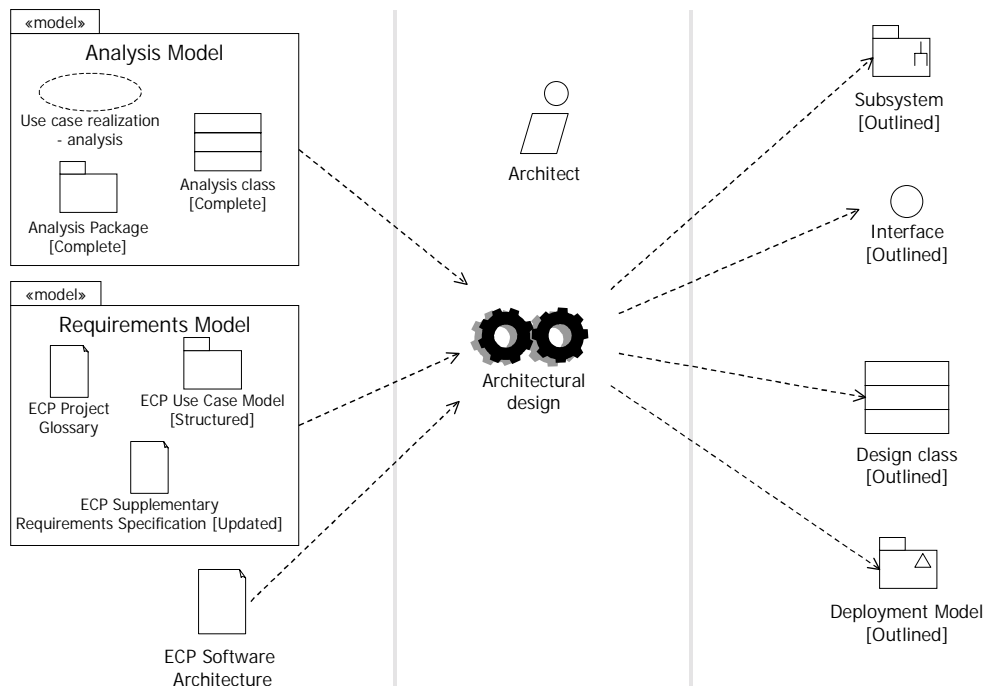
Table 1

Activity	Purpose
Architectural design	Outline the design and deployment models.
Design a class	Identify the attributes, operations and relationships of an design class based on its role in the interaction diagrams and on non-functional requirements.
Design a subsystem	Identify the interfaces to each subsystem and reduce coupling between subsystems.
Design a use case	This is about identifying the design classes and subsystems needed to realise the behaviour specified in the use case.

Activity: Architectural design

The figure below shows the inputs and outputs of this activity for the ECP project:

Figure 3

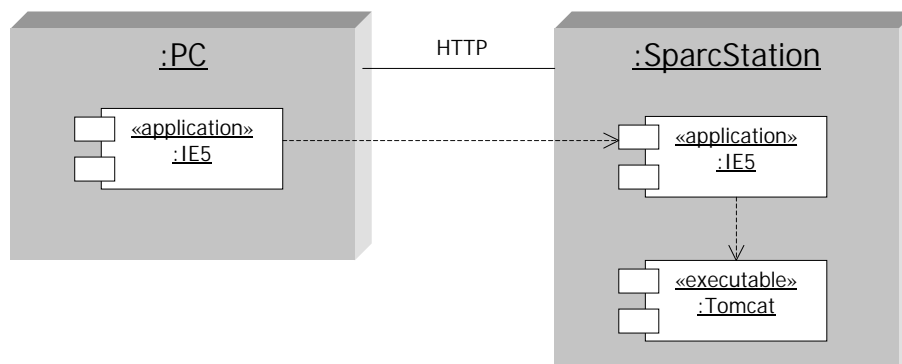


The first step is to review the “ECP Software Architecture Document” as this contains information about the physical architecture of the system.

From this, we see that we are using Java Servlets and JavaServer Pages running on Tomcat which is itself hosted by an Apache web server. The server machines will be SparcStations and the clients will typically be PCs running IE4 or IE5. However, the client hardware itself is not an issue as long as it is running a reasonably up to date web browser.

Here is the outlined deployment diagram for this architecture:

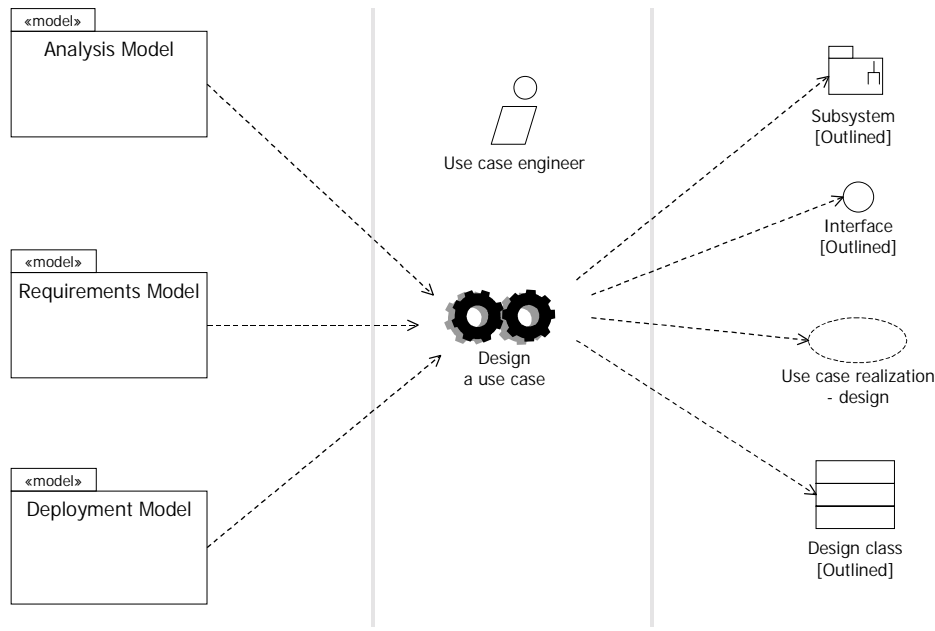
Figure 4



Activity: Design a use case

This activity involves taking all of our models to date and using the information in them to help us create a design that realizes the behavior specified in a use case. The inputs and outputs for the activity is shown below for the ECP.

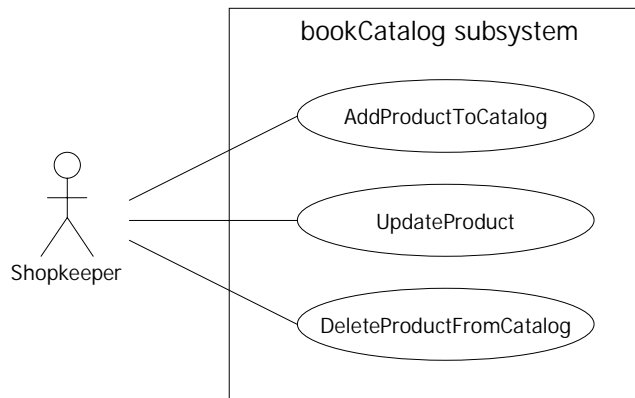
Figure 5



Obviously, the key step in this activity is first of all to figure out exactly which classes we will need to design. We do this by considering the use case that we are going to work with.

In fact, in this walkthrough, we will look at three use cases that are closely related:

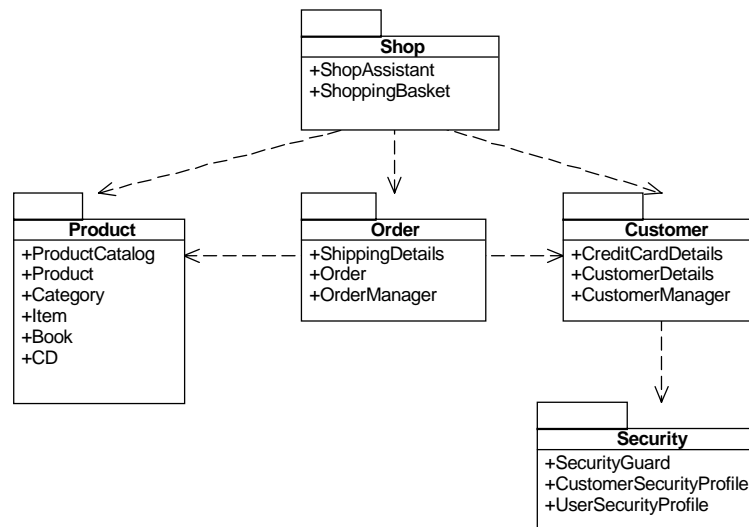
Figure 6



This set of use cases is a cohesive subsystem that is simple enough to be workable, yet complex enough to be interesting. It is ideal for this walkthrough!

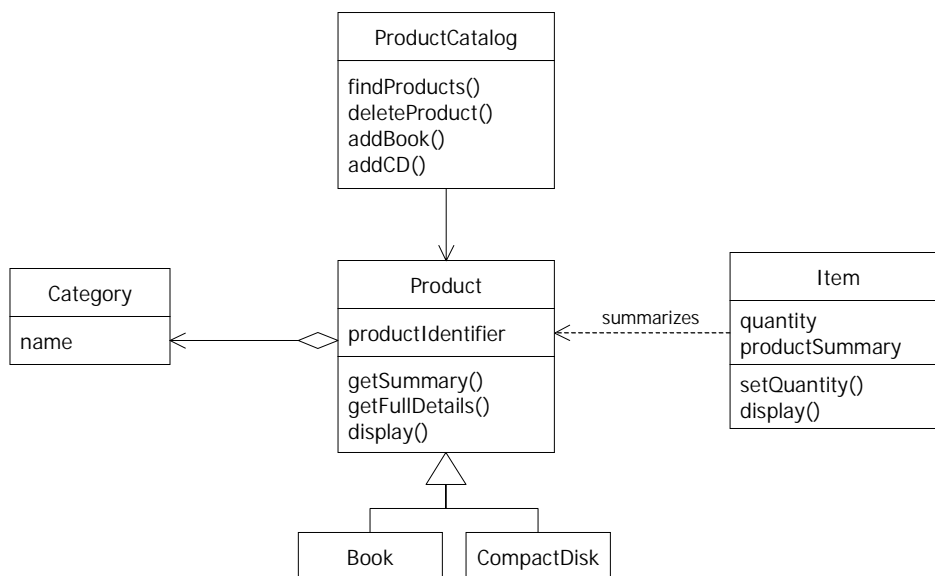
In the analysis model (shown below), these three use cases are realized by the classes in the Product package:

Figure 7



Here are the analysis classes in the Product package:

Figure 8

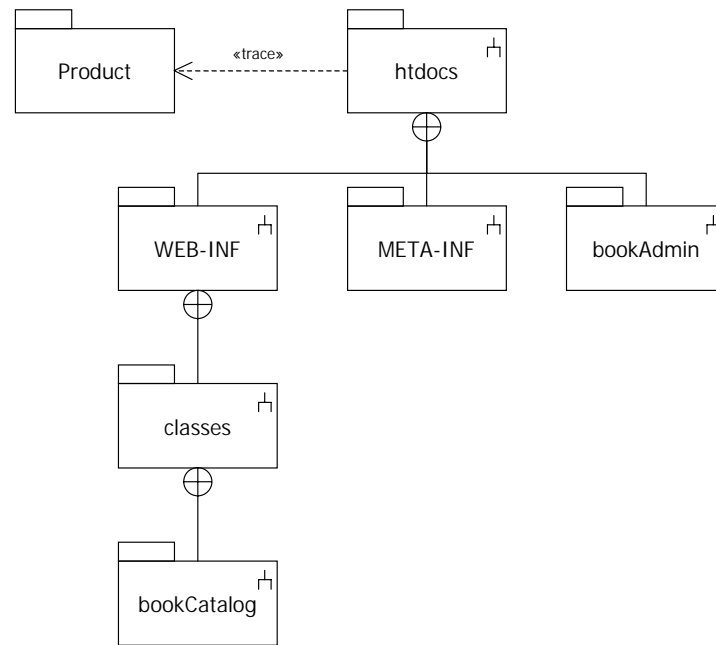


As we are not working with orders at this point in time, we can ignore the Item class. We will also make another simplification - we will initially only model one type of product - books.

Now that we have worked out which analysis packages and classes that we're going to work with, we can begin to design a subsystem and transform the analysis classes to design classes.

Because we are creating a web application using J2EE and Tomcat, we have tight constraints on how we can design this subsystem. This is because all J2EE web applications have the same directory structure as defined in the J2EE specification. We can refine our Product analysis package into subsystems as follows:

Figure 9



Each subsystem represents a directory on the server. Here is a summary of the role of each subsystem in the web application:

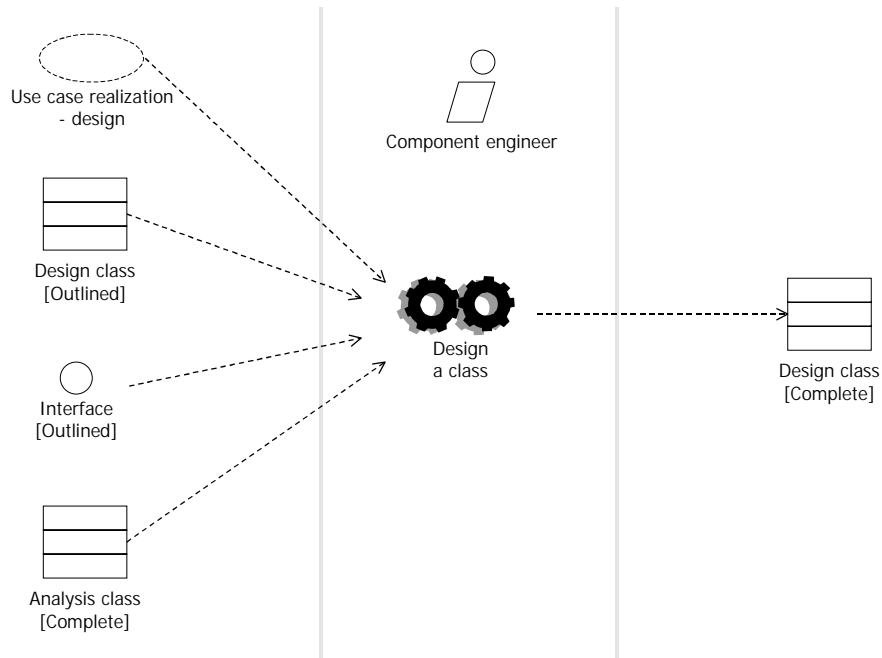
Table 2

Subsystem	Role
htdocs	This is the root directory of our web application on the particular Tomcat installation we are using. HTML files and JavaServer Pages (JSPs) may be placed in this directory and its subdirectories.
bookAdmin	We will place the HTML and JSP files that provide the UI for managing the book catalog in here. We can use Apache security to ensure that users must provide a username and password to access this directory.
WEB-INF	This is a necessary part of a J2EE web application. It is not used in this case.
META-INF	This contains the tag library descriptors for custom tag libraries.
classes	This directory and its subdirectories may contain Java servlets, JavaBeans and custom tag libraries.
bookCatalog	We will put all of the Java class files related to managing the book catalog in here.

Activity: Design a class

In this activity, we need to work out how express our analysis classes using the implementation technology we are using. We take the artefacts shown below as inputs to this design process:

Figure 10

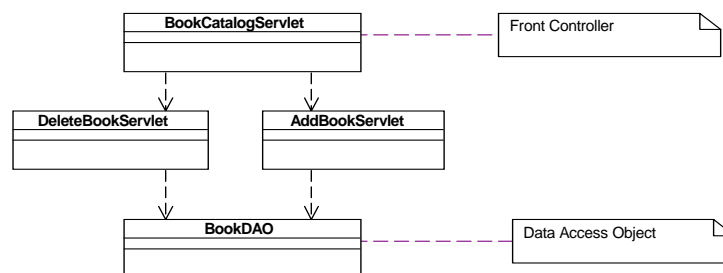


In order to be able to design a class, we need a deep understanding of the target implementation technology. It is impossible to do class design otherwise! To keep our design simple, and make it more understandable, we will also apply design patterns wherever it seems appropriate to do so.

Firstly, we will apply the Front Controller pattern [Alur] to the ProductCatalog analysis class. This pattern tells us that we will have a single class (a servlet or JSP) that controls and provides an interface to several other classes. We will use one servlet as the Front Controller, one servlet to update books in the database and one servlet to delete books in the database. We will abstract all database access by these servlets by applying the Data Access Object pattern [Alur]. This pattern localizes all database access into one or more data access classes. It prevents database dependencies spreading throughout the Java code.

Our design is modeled below:

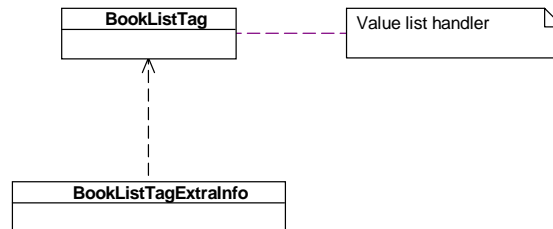
Figure 11



We have servlets to add and delete single books from the database. However, we also need some way to obtain lists of books. To achieve this we apply the Value List Handler pattern [Alur]. This patterns specifies how lists of objects are handled.

We could implement the Value List Handler pattern as one or more JavaBeans that could be used within JavaServer Pages. However, this approach would not separate HTML from Java code and this is one of the goals stated in the ECP Software Architecture Document. Instead, we will use a custom tag to implement the Value List Handler pattern. This will allow the JSPs to be almost completely free of Java code. The custom tag and its helper class are shown below:

Figure 12

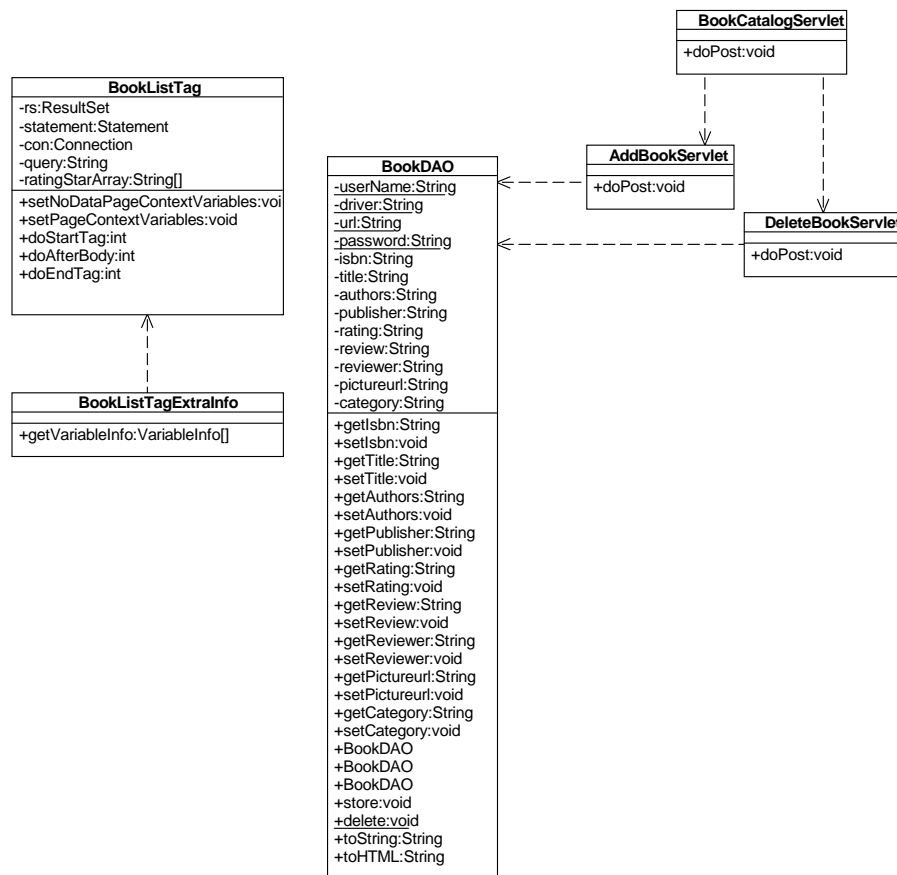


To deploy this web application on our installation of Tomcat, we have to put all of the class files in `htdocs/WEB-INF/classes` or one of its subdirectories. We will package all the class files together in the `bookCatalog` subsystem which we deploy as follows:

`htdocs/WEB-INF/classes/bookCatalog`

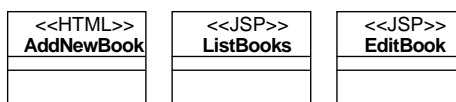
Here is the final class diagram for the book catalog subsystem after adding all the attributes and methods to the classes:

Figure 13



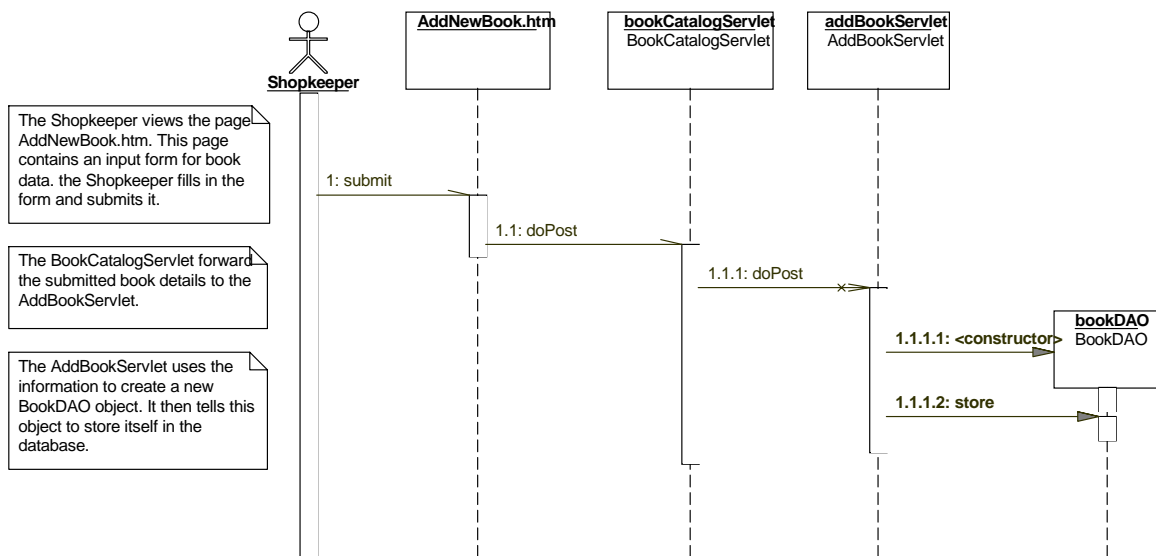
As well as having the data access servlets and tag libraries, we also need some sort of user interface to our book catalog. In fact, we need one HTML file and two JSPs as shown below:

Figure 14



Here is a sequence diagram that shows how a book may be added to the database:

Figure 15

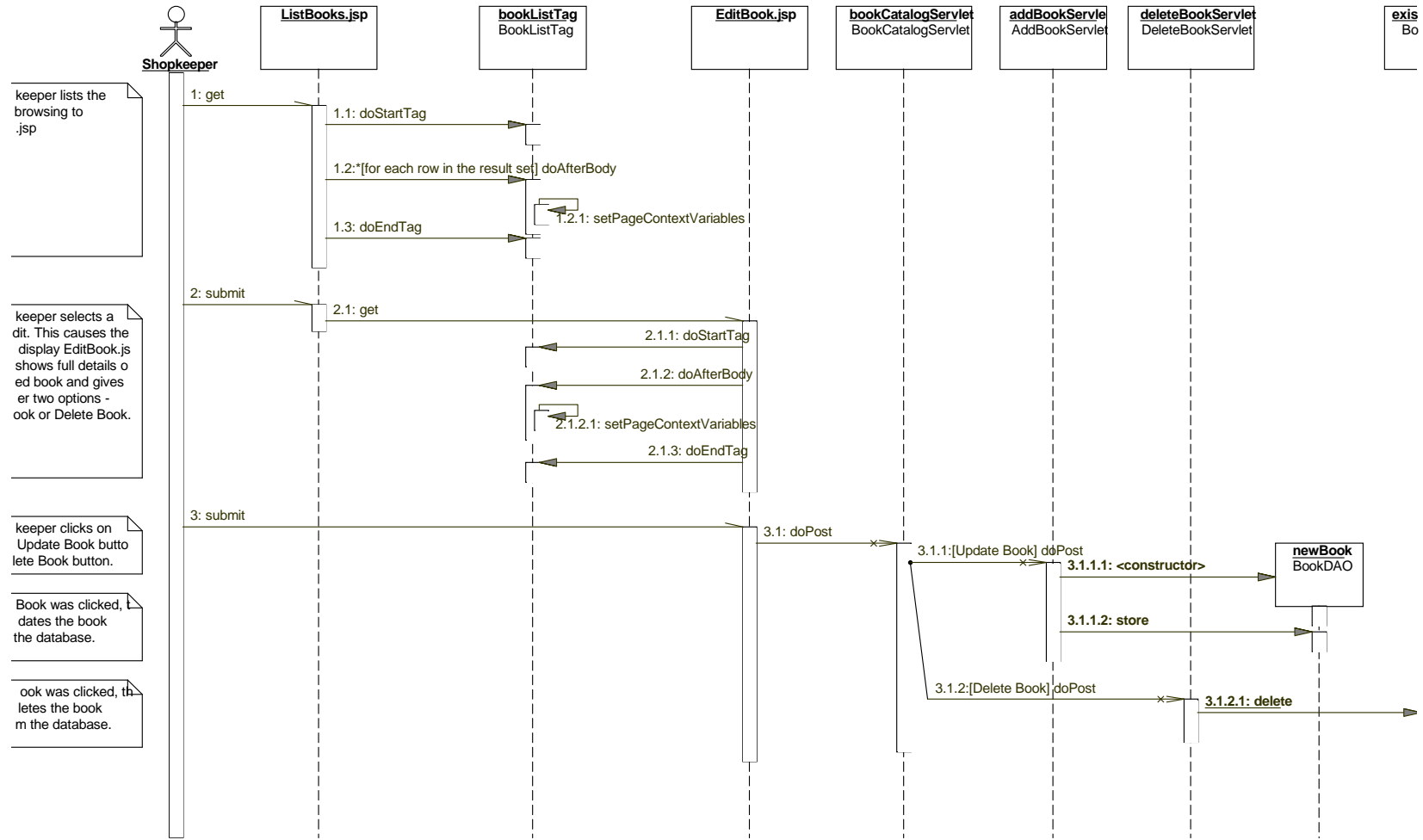


To simplify the sequence diagram we have not shown either Apache or Tomcat. In reality the Shopkeeper sends HTTP requests to Apache and these are forwarded to Tomcat for processing. Showing this extra level of interaction would not really add anything to the sequence diagram from the design perspective, and so we have omitted it.

We should point out another small error in Together UML syntax - the class names in the objects should really be prefixed by a colon and underlined.

Here is a sequence diagram that shows how the Shopkeeper may view a list of books, select a book and then update or delete it

Figure 16 (on next page).



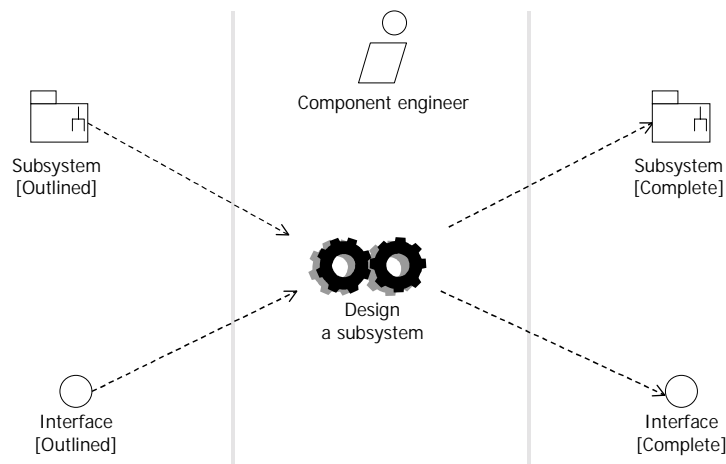
:Activity: Design a subsystem

The purpose of this activity is to ensure that the subsystem is:

- Is as independent as possible from other subsystems or interfaces.
- Provides the right interfaces.
- Provides a correct realization of all interfaces it defines.

The inputs and outputs of this activity are shown below for the ECP:

Figure 17



In this simple example, we have kept all of the classes related to book data in the same subsystem - so we have achieved high cohesion. Although we have not used Java interfaces explicitly (there was no need in this case), this subsystem has two well-defined logical interfaces. One logical interface is defined by the Front Controller class (BookCatalogServlet) and another is defined by the custom tag library BookListTag. Having two interfaces to book data is necessitated in this case by having to work with both individual books (BookCatalogServlet) and with lists of books (BookListTag). These well-defined interfaces mean that this subsystem has low coupling.

Summary

This concludes our brief walkthrough of the design workflow. The complete design model may be found on our website at www.clearviewtraining.com/example1/index.htm.