1. Introduction

1.1 Overview

This case study demonstrated how UML is used in an application. The requirements for an application are described with use cases and possibly domain analysis. These are analyzed, producing an object-oriented analysis model. It is then expanded into a design model that describes a technical solution. Finally, it is programmed in Java to create an application that is ready to run.

Most case studies discuss an application and include a few examples but don’t include additional models and diagrams. This case study is a Web application that automates something with which everyone is familiar: a library. As one might imagine, it provides capabilities to search, reserve, and borrow items, as well as supporting capabilities to manage inventory and the users of the library. It is a relatively simple application, walking a fine line of good design versus a more minimalist approach, but it attempts to address the areas of opportunity to enhance the design based on common practices in the industry, such as the use of certain design patterns.

The purpose of the case study is twofold:

• To show the usage of UML in a complete application, tracing the models from requirements to analysis to design to the actual code and then into a running application.
• To give you an opportunity to invoke UML in an existing model by going through the exercises at the end. The exercises provide opportunities to extend and improve the application. You can choose to do the exercises only in analysis and/or design, or if you know how to program in Java, you can evolve the actual code to incorporate the changes.

Before getting started, we need to discuss some terms related to Web development with Java that might not be familiar to you:

• JavaServer Pages (JSP). The JSP specification provided by Sun Microsystems provides the presentation mechanism that allows you to combine HTML and Java code to present information to the user dynamically.
• Servlets. Information appearing on java.sun.com states that “Servlets provide a component-based, platform-independent method for building Web-based applications, without the performance limitations of CGI programs.”
• Taglibs. Sun Microsystems indicates on java.sun.com that JSP tag libraries define reusable functionality. Per Sun, “Tag libraries reduce the necessity to embed large amounts of Java code in JSP pages by moving the functionality provided by the tags into tag implementation classes.”

Just as in a real-world solution, we utilize a number of tools commonly used in developing Java Web applications today, such as the following:

• Eclipse. This is the integrated development environment (IDE) used for editing Java code and managing all of the development of the application.
• MySQL. An open-source relational database management system.
• J2SE. As Sun Microsystems indicates on java.sun.com, the Java 2 Standard Edition (J2SE) provides a “solution for rapidly developing and deploying mission-critical, enterprise applications,... compiler, tools, runtimes, and APIs for writing, deploying, and running applets and applications in the Java programming language.”
• Log4j. Part of the Apache Jakarta project. It is used to log information for development and
runtime environments
- Ant. Part of the Apache project. It is a build tool, similar to make, for the purpose of building and deploying code in a platform-independent manner.
- J2EE. The Java 2 Enterprise Edition. Provides enterprise services such as Servlets, JSP, and Java Database Connectivity (JDBC).
- Tomcat. Part of the Apache Jakarta project. It is an open-source Web container that supports the Servlet and JSP specifications provided by Sun Microsystems.
- Struts Framework. Part of the Apache Jakarta project. It is used to build J2EE Web applications by providing the presentation and control mechanisms to handle common patterns in Web development.

The primary focus of this case study is to demonstrate the use of UML, so we do not favor one implementation over others. We chose not to use Enterprise JavaBeans (EJB) in the example so as not to get hung up in too many implementation details. In practice, we could create a better, more elegant J2EE solution for the Web; however, for this case study it was more important that we focus on demonstrating how to use the UML diagrams to implement a J2EE Web solution.

Note: What is shown is only one possible solution: there is no right solution for all circumstances. The goal is to produce a system that satisfies the requirements and that works well, not to produce diagrams that are perfect in all their details.

1.2 Requirements

Various means to elicit, capture, and communicate the requirements of the system exist. Among the more prominent means are the system vision (communicated in a document) and the system’s use cases communicated in UML as a use-case model, and in a series of use-case specifications, that detail the core functionality of the system in a language and manner that emphasizes user-to-system interaction.

Other common requirements artifacts include things like a glossary of terms and supplementary specifications (where we put requirements that transcends use cases) but, for this case study, we limited the robustness of the requirements artifacts.

In a real project, it is essential to engage the system’s stakeholders to support the production, approval, and inevitable evolution of the requirements. In addition to the vision and use cases we also demonstrate a simple domain model, often used to define the key domain classes and their relationships. The domain model, like the requirements artifacts, should limit itself to terms that are understood by users of the system and others who understand the system’s domain, but who are not necessarily programming experts.

1.3 Developing a Vision

Before delving into detailed requirements that prescribe what the system shall do, it is a good idea to gather a set of higher-level requirements that assist in scoping the effort. These high-level requirements constitute the vision of the system to be developed and assist in achieving agreement on what will be developed.

A lightweight set of features for the first version of the library application might look like this:
- It is a support system for a library.
- The library lends books and magazines to borrowers, who are registered in the system, as
are the books and magazines.

- The library handles the purchase of new titles for the library. Popular titles are bought in multiple copies. Old books and magazines are removed when they are out of date or in poor condition.
- The librarian is an employee of the library who interacts with the customers (borrowers) and whose work is supported by the system.
- A borrower can reserve a book or magazine that is not currently available in the library, so that when it’s returned or purchased by the library, that borrower is notified. The reservation is canceled when the borrower checks out the book or magazine or through an explicit canceling procedure.
- The librarian can easily create, update, and delete information about the titles, borrowers, loans, and reservations in the system.
- The system can run on all popular Web browser platforms.
- The system is easy to extend with new functionality.

To give some business context, we explain portions of the library inside and outside the application. In the first version, the borrower is able to access the application from any of the PCs within the library. In later versions, enhancements are made to the business processes and the application so that borrowers can access the application from anywhere they have access to a supported Web browser. The borrowers are able to check out items and pick them up at a later time.

The first version of the system doesn’t have to handle the message that is sent to the borrower when a reserved title becomes available, nor does it have to check that a title has become overdue. Additional requirements for future versions are available in the exercises at the end of this chapter.

### 2. Modeling Use Cases

#### 2.1 Establishing the Use Cases

The first step in use-case modeling is to define the actors that represent those who interact with the system and the use cases that describe what the library system provides in terms of functionality to those actors - the functional requirements of the system. While identifying the use cases, you must read and analyze all specifications, as well as discuss the system with potential users of the system and all stakeholders.

The actors in the library are identified as the librarians and the borrowers, because both are users of the system. The librarians have management capability to add borrowers, titles, and items. The borrowers are people who check out and reserve books and magazines. Occasionally a librarian or another library can be a borrower. Finally, we have a Master Librarian actor - this role is capable of managing the librarians as well. It is possible to add a title to the system before the library has a copy (an item), to enable borrowers to make reservations.

The use cases in the library system are as follows:

- Login
- Search
- Browse
- Make Reservation
• Remove Reservation
• Checkout Item
• Return Item
• Manage Titles
• Manage Items
• Manage Borrowers
• Manage Librarians
• Assume Identity of Borrower

Note the difference in concepts between “title” and “item.” Because a library often has several copies of a popular title, the system must separate the concept of the title - the name of a book and the book’s author - and a separate physical copy of the same title, which is an item.

To pragmatically limit the complexity of this case study, we do not perform an implementation for some of the “Manage” use cases such as Manage Librarians and Manage Borrowers. As in a real-world effort, the scope of what can reasonably be built using available resources must be weighed against what is needed to have the software succeed. For example, rather than implement Manage Librarians, we expect those designated as Master Librarians to be able to manage librarians by editing the database directly. As potentially dangerous as that is, it is a risk worth taking until a foolproof implementation can be provided.

The library functional requirements are documented in a UML use-case diagram as shown in Figure 1. Each of the use cases is documented with text, describing the use case and its interaction with the actor in more detail.

The outline of the basic flow for the use case Checkout Item (which means that a Borrower can check out an Item) is described as follows:

1. The borrower chooses to perform a “Search” for desired titles.
   2. The system prompts the borrower to enter Search criteria.
   3. The borrower specifies the search criteria and submits the search.
   4. The system locates matching titles and displays them to the borrower.
   5. The borrower selects a title to check out.
   6. The system displays the details of the title, as well as whether or not there is an available item to be checked out.
   7. The borrower confirms that he or she wishes to checkout the item.
   8. The system checks out the item.
   9. Steps 1 to 8 can be repeated as often as desired by the borrower.
  10. The borrower completes checkout
  11. The system notifies a librarian that the borrower has concluded the checkout item session and displays instructions for the borrower to collect the contents.

Several alternative and exception flows for this use case exist. The description of this use case and others are included in more formal detail in the use-case model on the CD-ROM.

A team implements use cases throughout the development of the system to provide descriptions of the functional requirements of the system. They are used in the analysis to check whether the appropriate analysis classes (discussed later) have been defined, and they are used during the design process to confirm that the technical solution is sufficient to handle the
required functionality. The use cases can be visualized in sequence diagrams, which detail their realization.

## 2.2 Establishing the Domain Model

An early domain model is useful to establish a core set of classes that represents the things in the problem space of the system to be built. For example, in the case of the library, you see such things as Title, Item, Reservation, and Borrower as well as their relationships.

To build a domain model, you typically need to have a reasonable grasp of the business domain of the system under development. You read the specifications and the use cases and look at which “concepts” should be handled by the system. You might also organize a brainstorming session with users and domain experts to try to identify all the key concepts that must be handled, along with their relationships to each other.

The domain classes in the library system are as follows: Borrower, Title, Book Title, Magazine Title, Item, Reservation, and Loan. They are documented in a class diagram along with their relationships, as shown in Figure 2. Everyone’s practices vary slightly, but you can formally model a business model or simply perform some domain analysis with the intent of identifying entities of the system. We chose to use the stereotype “entity” for each of our domain classes. Typically, you use the stereotype “entity” to illustrate that a class is likely in the domain and is likely persistent (that is, instances are stored in a database).
It is important to note that the detail of classes modeled evolves over time. You might be pleased to see good names, with clear definitions, and only a hint of relationships, operations, and attributes during the early stages of a project. But, as you conclude other supporting steps, or even complete iterations of development, you see the classes evolve, become refined, even explode into numerous classes or disappear as you move further into understanding of both the problem and solution space.

For those classes with interesting states one can build UML state machine diagrams. These diagrams show the different states that objects of those classes can have, along with the events that make them change their state. The state machine diagram for the Title class is shown in Figure 3.

3. Analysis

As you move closer to developing a solution to your problem, you begin to exploit more of UML to represent unambiguously and consistently your understanding of the problem space in ways that easily translate into proposed solutions. You can perform analysis in several ways, and as a matter of fact, the lines between analysis and design are often blurred, particularly the further into a project you get. Often, it is simply too resource-intensive to manage both an analysis model and a design model, so many teams would simply stop maintaining their analysis model as development proceeds. Regardless of how much (or little) emphasis you place on formal analysis, a technique known as use-case analysis is an excellent way to get beyond an empty canvas and begin to see the formation of a rich enough object model so that design can proceed with a degree of confidence.

3.1 Performing Use-Case Analysis

Assuming that you have done some domain analysis, you have a simple set of classes that begin to demonstrate the analysis classes involved in your system. That is a good start, but to really enrich your model you perform use-case analysis to refine the classes, add additional
Performing Use-Case Analysis

To describe the dynamic behavior of the analysis classes, a variety of the dynamic UML diagrams can be used: sequence, communication, or activity. The basis for the sequence diagrams are the use cases, where each use case has been described with its impact on the

classes, and add operations and attributes to these classes, so that each helps to represent what goes on during the execution of the various paths through your use cases.

You use static diagrams (for example, class diagrams) and dynamic diagrams (for example, sequence diagrams) to illustrate how classes and their instances collaborate to perform the actions required by the use cases. Analysis classes are typically organized as one of the following:

- «entity». As you have already seen, entity classes represent those classes that have meaning in the domain and are likely have long lifetimes in your system. The reality of long lifetime is that they require some form of persistence so that they can exist beyond the execution of a single use case or session with a user. In the majority of applications today, you can infer that some form of database is used to support the persistence of entities.

- «boundary». Boundary classes are used to represent those things that are necessary to provide a means for actors to communicate with the system. In the case of an interactive user, boundary classes represent their user interface mechanisms (for example, a form or page). In the case of noninteractive users, like an external system or device, you can typically expect some protocol-based interface to shuttle communication between the system and the nonhuman actor. During use-case analysis, you generally create a single boundary class for every actor-system relationship found in the use-case model.

- «control». Control classes are used as the placeholder of the coarsegrained logic of a use case. You usually start out by creating a single control class for each use case you are analyzing.
analysis classes, to illustrate how the analysis classes collaborate to perform the use case inside the system. When modeling these sequence diagrams, you discover new operations and add them to the classes. Though there are now more precise operations, some even with hints of arguments, you are far from a fully defined set of classes. You complete things like operation signatures, and review classes for completeness, sufficiency, and primitiveness during individual class design.

How much is enough? When performing use-case analysis, typically look for a class diagram representing the View of Participating Classes (VOPC) for each use case modeled. Each VOPC depicts the classes, relationships, operations, and attributes necessary to support its associated use case. If you can derive that class diagram without once modeling a sequence or communication diagram, then good for you. However, most of you will like to do a few interaction diagrams just to ensure you can trace through the various flows of a use case.

As such, we recommend developing a sequence diagram for each primary (or basic) flow of a use case and supporting those with occasional diagrams for interesting or potentially ill-understood alternative or exceptional flows. These can be supplemented by communication diagrams that better demonstrate the links required between the objects to support the use case. Ultimately, these diagrams might be treated as mere scaffolding, but it is certainly a reasonable check during the early stages of development. You should be able to see clearly that the VOPC diagram is robust enough to support all the other dynamic diagrams. Figure 5 shows a communication diagram that is semantically equivalent to the sequence diagram shown in Figure 4.

Figure 6 shows a View of Participating Classes diagram that shows which classes participate in the Checkout Item use case and which relations and operations are exercised. We have dropped off the actor because that relationship is not part of the final technical design, but otherwise one can clearly see that this diagram parallels Figure 5.

In addition to the classes and operations obvious from the communication diagram, we have added some fidelity to the associations modeled. Multiplicity has been specified on the associ-
Based on the directions of the messages in the communication diagram, the associations are predominantly unidirectional. We have left the association between Title and Item—already in our model from the domain modeling effort—as bidirectional; we are confident we will need to traverse in each direction as we model the rest of the system. On the other hand, to minimize dependencies, we have changed the association between Borrower and item to be unidirectional. This is a natural way a model is refined over time.
3.2 Not UML? - How can that be?

We all love UML and want to think it is all one ever needs to model a software system properly. However, the cold hard reality (the normative power of facts) is, that there are other tools one can use which are extremely helpful as well.

Somewhat between analysis and design, for example, one usually wants to develop “User Experience” models that illustrate in a very direct and visual way what users can expect when they interact with the system. These user experiences come in a variety of forms and generally are presented in support of use cases. For example, Fowler [Fow3] introduces the “screen flow diagram” to illustrate how user control and information flows from one screen window to another.

Storyboards of screen content, action, and navigation are very useful in ensuring that both the use cases and the subsequent use-case analysis are grounded in tangible reality. Additionally, simply having an occasional, tangible screen shot, double-edged sword that it is, serves to move the understanding and agreement process. (It is a double-edged sword because on the one side it gives something the users can easily grasp, but on the other it can give a false sense of progress to an unmanaged client.) Figures 7 and 8 show examples of a storyboard and comp (cool user interface lingo short for “composition”), respectively.
WARNING The use of such client-friendly diagrams must be tempered with a good deal of management of customer expectations. In other words, once the clients see nifty stuff, they'll want more, and it is up to the development team to keep the focus on productive and semantically valuable stuff.
4. Design

The application of the design discipline and the resulting UML model expands and details the analysis model by taking into account all technical implications and restrictions. The purpose of the design is to specify a working solution that can be easily translated into programming code.

The classes defined in the analysis are detailed, and new classes are added to handle technical areas such as database, user interface, communication, devices, and more. The design can be divided into two segments:

- Architectural design: This is the high-level design where you define the packages (sub-systems), including the dependencies and primary communication mechanisms between the packages. Naturally, a clear and simple architecture is the goal, where the dependencies are few and bi-directional dependencies are avoided if at all possible.
- Detailed design: This part details the contents in the packages, so that all classes are described in enough detail to give clear specifications to the programmer who codes the class. Use dynamic models from the UML to demonstrate how objects of the classes behave in specific situations.

4.1 Designing the Architecture

A well-designed architecture is the foundation for an extensible and easily maintainable system. Arguably, the most critical technical role on most projects is that of the architect. To be an architect is to be a rock star (in the software world), and the time when architects earn their pay most is when they are performing design activities of architectural breadth and depth.

Such things as key package and layer structure, selecting or designing large-scale reusable components, committing to key patterns for critical and repeated functionality (like the persistence mechanism or security model), and simply driving the design and coding standards to be upheld during the life of the project are each examples of critical architectural responsibilities.

4.1.1 System Structure

In this section, we put on the hat of architect and try to separate the system at hand into large chunks (packages) that serve to both layer and partition the system under development. A simple class diagram representing the essence of these packages is depicted in Figure 9. Figure 10 illustrates to another level of detail how those packages expand both internally and through relationships with implementation mechanisms (in this case, several preexisting standard Java and J2EE packages). Notice that the static architecture has no bidirectional dependencies between packages (avoiding packages’ becoming forever coupled) and an implied build dependency exists between the packages. This example is a simple case of the sort of implicit rule by which architects abide.

The packages that provide layering for the library system are as follows:

- presentation. This contains classes for the entire user interface, to enable the user to view data from the system and to enter new data. The standard for presenting this information to the user in a Java Web application is to use JSP. This package cooperates with the business objects (via the Value Objects) and the controller packages. The user interface package uses the Value Objects and posts data so that it can be sent to the Controller package.
controller. The classes in this package are responsible for most of the application control and also provide “traffic cop” mechanisms to help delegate dependencies to and separate them between the presentation and domain packages.

• business. This includes the domain classes from the analysis model such as Borrower, Title, Item, Loan, and so on. These classes are detailed in the design so that their operations are completely defined. The business object package cooperates with the database package via an associative relationship.

• dao. Named dao because it contains a Data Access Object (DAO) that represents a commonly used design pattern that encapsulates all data-related activity into discrete classes. This is the means to firewall or limit the knowledge of how things are persisted.

• vo. The vo package is so named as it contains simple Value Objects, which are lightweight representations of domain-related things. The Value Object (VO) is yet another common pattern used to limit the passing of heavyweight objects and traffic across enterprise systems.

4.1.2 Architectural Mechanisms

• Architectural mechanisms describe the techniques to deal with, among other things, persistence, security, error handling, and system interfaces. Architectural mechanisms must be modeled to give the architect(s) and developers a view on common mechanisms that transcend any one-use case. The application must have objects stored persistently; therefore, a database layer, called the dao package, must be added to provide this service. The solution for a complex application is to use a commercial database; typically this database is a relational database management system (RDBMS) such as Oracle or SQL Server. However, because this case study application is intended to be portable, we chose to use an open-source database tool you may be familiar with called MySQL. The data is manipulated

Figure 9 An easy to grasp class diagram showing an architectural overview with the library system packages and their dependencies.
using DAOs, implementing a common interface to call common operations such as create(), store(), remove(), and load() on the objects. Figure 11 illustrates a pattern specified by the architect to be used for data storage and retrieval. The diagram reflects both how DAO and VO patterns are blended to work together. Figure 12 is a variant on the 11 sequence diagram, using a class diagram instead.

4.1.3 Design Patterns

Design patterns are, among other things, well-proven, reusable solutions used to solve a particular problem. In designing the case study, we made a decision to use many of the patterns applied to the design good Web applications. Among other patterns, we are using these patterns in the design of this application:

- Model-view-controller (MVC). The MVC pattern is more of an architectural pattern that uses design patterns to enforce its high-level pattern. Nicholas Kassem and the Enterprise
Figure 11  A sequence diagram that highlights VO and DAO use on the library system.

Figure 12  A class diagram depicting a VOPC for the key VO and DAO pattern usage.
Team at Sun Microsystems state in Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition, “The MVC architecture allows for a clean separation of business logic, data, and presentation logic.” The model contains the business data, the view (that is, JSP) renders content of the model, and the controller defines the application behavior [Kass2k].

- Front controller. This design pattern is provided by the Struts framework that supports the MVC pattern. As Alur and his coauthors describe in CoreJZEE Patterns: Best Practices and Design Strategies, the front controller provides the first point of contact for handling incoming requests to the application (Alur et al., 2001).

- Data Access Object (DAO). Alur and his coauthors also state that a DAO is used to “abstract and encapsulate all access to a data source.” [Aluk]. This allows you to go to one place to make data changes and does not have an impact on the business or presentation layers. You use a DAO Factory so that you can conditionally access data from multiple datasources, such as a Properties file or the MySQL database, thus promoting the tenets of polymorphism.

- Value Object (VO). According to Alur and his coauthors, a VO is “used to encapsulate business data”. In the Unified Library application, VOs are used by the business objects, DAOs, and the JSP.

4.1.4 Design Policies

We establish certain policies on the project to enforce the principles of the architectural design. You can see many of these identified in the architectural detail diagram in Figure 10. Here, you identify that the following rules are established:

- Loose coupling between the architectural layers:
  - A change to one layer does not necessitate a change elsewhere in the application.
  - Separation of concerns. This allows developers and designers to focus on what they know best. All calls to DAOs are made only through the appropriate business object. For example, to access the BorrowerDao, you call the Borrower business object.

- JSP only access the model (in the MVC architectural pattern) via a Value Object. JSPs do not access any business-related objects directly.

- JSP contain only presentation-related code. No business functionality is contained in the JSP.

- JSPs do not contain any Java scriptlet code. Instead, they use taglibs to conditionally show or hide certain presentation-related data.

- Action classes supported by Struts framework contain only code related to extracting the data from the request objects, passing data to the appropriate business object, and returning to the appropriate JSP. There is no business code in these action classes. This enables you to support more than one type of client (other than a Web browser).

- Since FormBeans are an implementation mechanism of the Struts framework, we chose to link the FormBeans to the Value Object and then act upon the Value Object elsewhere in the application. This way if you need to use a different framework, you can do so with min-
Performing Detailed Design

The purpose of the detailed design is to employ the strategic architectural design and all of its associated implicit guidance (like key patterns for frequently performed or critical functionality) and adhere to it while evolving the results of analysis, or even earlier iterations, into a precise and semantically rich representation of what you want to build in code. This gets pretty detailed, including such things as refining individual class specs to ensure that they are complete, meeting the policies of the project for exception propagation, reporting, general handling, and so on. After a few iterations, the design model should be detailed enough so that you can hand the model to a tool, and an appropriate compilable specification so that its classes can be generated with a click of a button. Clearly, the visual parts of UML are insufficient to depict all that is needed; however, the combination of graphical elements and nongraphical UML elements are sufficient to achieve the goal. The following sections describe the nature of detailed design considerations and activities on a key package-by-package basis. As a part of this discussion, we focus on the business and presentation packages described in some of the detailed design decisions based on original analysis.