A Four-Tier Model of
A Web-Based Book-Buying System

6.826 Final Project
May 14, 2004

Joy Forsythe, Xian Ke, Leah Oats
{forsyjoy, xke, loats} @ mit.edu

Table of Contents

1 Introduction

In this document, we provide the specifications and high-level implementations for a Web-based book-buying system. We consider the various requirements involved in designing a system similar to Amazon.com, and formulate this design in terms of the layers of a four-tier architecture.

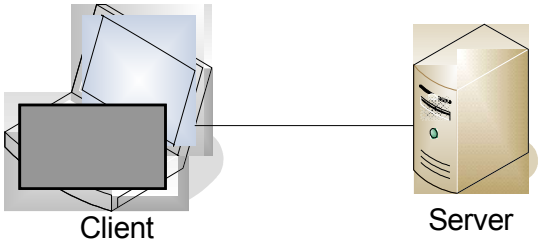


Figure 1: Traditional client-server model of a distributed system

The need for an N-tier architecture arises due to requirements in scalability, flexibility, and maintainability that becomes increasingly difficult to meet in distributed applications. The standard model for distributed computing is a client-server model, which can be viewed as a two-tier architecture (see Figure 1). However, this structure has many drawbacks which can be alleviated by distributing resources in a more systematic way. One problem is that one of the two layers tends to end up being a large monolithic application. These applications must be carefully written because of limited system resources. The two-tier architecture also results in heavy network traffic when data is mainly processed on the client side. This traffic is costly to both the server and client. Updating the logic that processes the information is extremely difficult because it must be updated on every client machine. N-tier architectures are designed to avoid these problems by creating layers within the client and server that more efficiently distribute these tasks (see Figure 2).

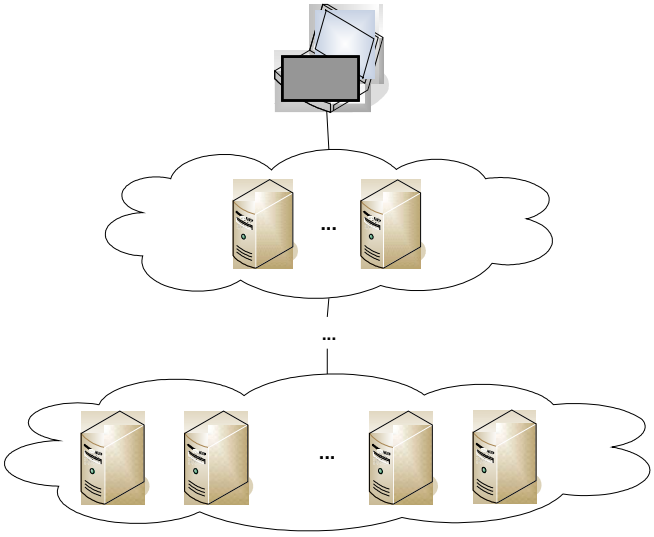


Figure 2: Multi-tier model of a distributed system

Applications on the World Wide Web must be able to handle surges in requests and failures of servers with minimal interruptions in service. One important constraint for a Web-based application is that the client, the Web browser, is specified to be essentially stateless. The only exception is "cookies" that the browser user can clear at any time. The HTTP protocol that the browser uses to communicate with the Web server is also stateless, making it impossible for any logic to be performed on the client side. Without layering, it is clear that a monolithic server application suffers from a severe lack in scalability, flexibility, and maintainability.

A three-tier system is a common paradigm in enterprise Web applications. The three standard tiers are as follows:

1. The client machine on which a user browses the Web

2. Stateless servers that provide the logic to do front-end processing on information and sends the results to the client browser. This tier is otherwise known as middleware.

3. Stateful machines that store information, such as in a database, and has functionality to send data to the front-end servers.

In our book-buying application, we consider each of the above tiers, but we further divide the middleware into a logic layer and a presentation layer. The resulting four layers are as follows:

Client Layer	Displays the content sent from the server, storing and retrieving any cookie state in the process

Presentation Layer	Processes and delivers display content to browser (e.g. HTML), along with any cookie state
Logic Layer	Specifies the business objects and rules of the application, and handles interfacing between the presented information and the stored data.
Database Layer	Stores the nonvolatile state of the application, and exposes ways to access this state

2 Background

2.1 N-Tier Architecture

An N-tier architecture divides computer hardware and software resources into multiple logical layers, where N is any natural number. Because each layer can be dealt with independently of other layers, the result is increased flexibility in developing, managing, and deploying a system. Developers only have to modify or add specific layers rather than disturb the entire system if technologies change, use scales up, or problems occur.

Abstraction, modularity, and separation of control are key concepts in computer science. As such, considering a system in terms of tiers is really nothing new. However, the term N-tier architecture implies a greater degree of separation than typically happens within an application. It is not uncommon for two tiers to run on different platform environments, different machines, and across geographical boundaries. Furthermore, each tier that makes up a distributed application can be a cluster of physical machines, with each machine doing a share of the tier's processing work. Replication within the cluster can provide fault tolerance due to failure of individual machines. A switch or load balancer handles sending requests to the right machines in the tier.

A central rule in an N-tier system is that only resources on adjacent tiers can communicate with each other. Such a restriction simplifies reasoning of the application by minimizing the dependencies needed to achieve functionality. One always has an understanding of how the state of a tier can be changed, because only a maximum of two other tiers can change it. Again, this concept of reducing dependency by decoupling modules is nothing new. It is widely regarded and taught as a best practice in software engineering. However, while it is often tempting to closely couple modules in a local application for code efficiency, doing so in a physically distributed application is much more expensive. Each dependency requires adding an additional channel of communication, not to mention the complexities and overhead of maintaining these connections in a distributed environment.

2.2 Book Buying

The process of buying a book from start to finish has a number of steps. Users must first visit the book vendor's home page, and then must browse or search for a book. After having picked one or more to purchase, she or he must sign in (which may involve creating a new account with the

site) and then enter shipping and payment information before finally confirming her or his order. All in all, a barebones specification of a book-buying Web site takes at least 14 different web pages as illustrated in Figure 3, and can easily take more as new features are added to the site. The specification diagram in Figure 3 charts not only the Web pages needed to create a book-buying site but also how to navigate between these pages.

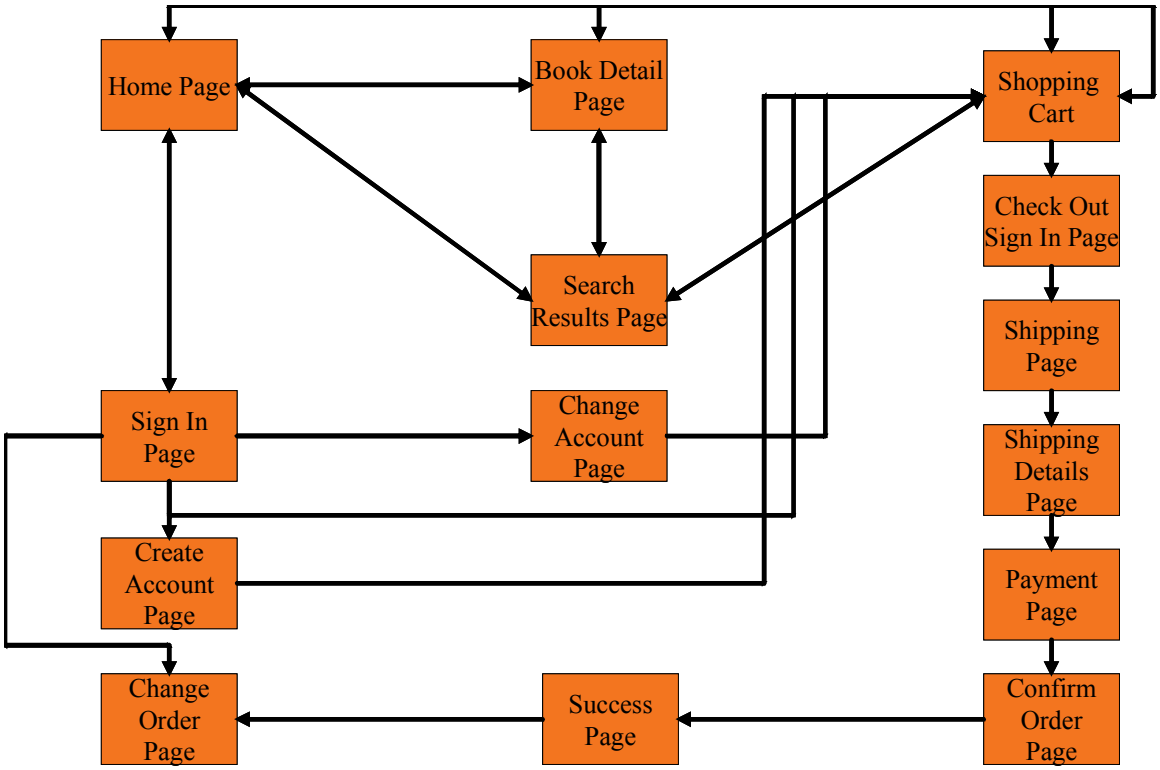


Figure 3: Specification diagram of a bare-bones book-buying site

Some details have been omitted from Figure 3 for the sake of clarity. First, the user can begin navigation at any of the pages by simply directing their browser to the appropriate address. It may then be necessary to request account login information prior to displaying certain pages. For instance, a sign in must have occurred before the "Change Order Page" can display a list of unshipped orders. Another omitted detail is the fact that every page can navigate back to the home page.

It is interesting to note that while the majority of the pages are very accessible from each other, those involved in the purchasing process form a fairly rigid line. Closing the browser at any step in the purchasing process requires starting over from the beginning of the process.

3 Specifications

We now proceed to describe the functionality for each layer in the book-buying system from the bottom up. We represent each layer as a module that exposes a set of functions and procedures to a higher layer.

3.1 Global Type Definitions

We begin by defining global types that can be used in any layer. Most of these types are records that allow information to more easily be passed from one layer to another, serving to make the specification code more understandable.

Listing 1: Type definitions for user accounts and orders

```
TYPE UserInfo = [firstName : String, middleInitial : Char, lastName : String,
                      email : String]

       Address = [AddressLine1 : String, AddressLine2 : String,
                  City : String, State : String, ZipCode : String
                  Country : String]

   PaymentInfo = [CreditCardNumber : String, ExpirationDate : Date,
                        SecurityNumber : String]

  ShippingInfo = [CarrierName : String, TrackingNumber : String,
                        ShippingDate : Date]
```

Listing 2: Type definitions for books

```
TYPE Book = [title : String,
             author : String,
             isbn : ISBN,
             retailPrice : Price
             numberPages : Nat
             editionNumber : Nat
             publicationDate : Date
             description : String
             categories : SEQ String
             keywords : SEQ String
             outOfPrint : Boolean]

 TYPE
     CountryID : String  % number assigned by ISBN.org
     Country : CountryID -> String
     PublisherID : String % number assigned by ISBN.org
     Publisher : PublisherID -> String % one publisher can have multiple ID's
     ItemID : String

     ISBN = [countryID, publisherID, itemID, checksum : Int]

     % invariants on ISBN:
     % -------------------
     % (CountryID + PublisherID + ItemID).size = 9
```

```
    % /\ checksum_temp = +: {i :IN (1..9) ||
    %    (CountryID + PublisherID + ItemID)(i) * i} // 11
    % /\ IF checksum_temp = 10 => checksum = 'X'
    %    [*] checksum = checksum_temp

    Price : [whole : Nat, decimal : 1..99]
    Date : [month : 1..12, day : 1..31, year : Nat]
    BookID : Int
```

Listing 3: Type definitions for stocks of books

```
TYPE Location = [locationName : String, locationZipcode : String]
     Quantity = Int
    StockInfo = (Location -> Quantity)
```

3.2 Database Layer

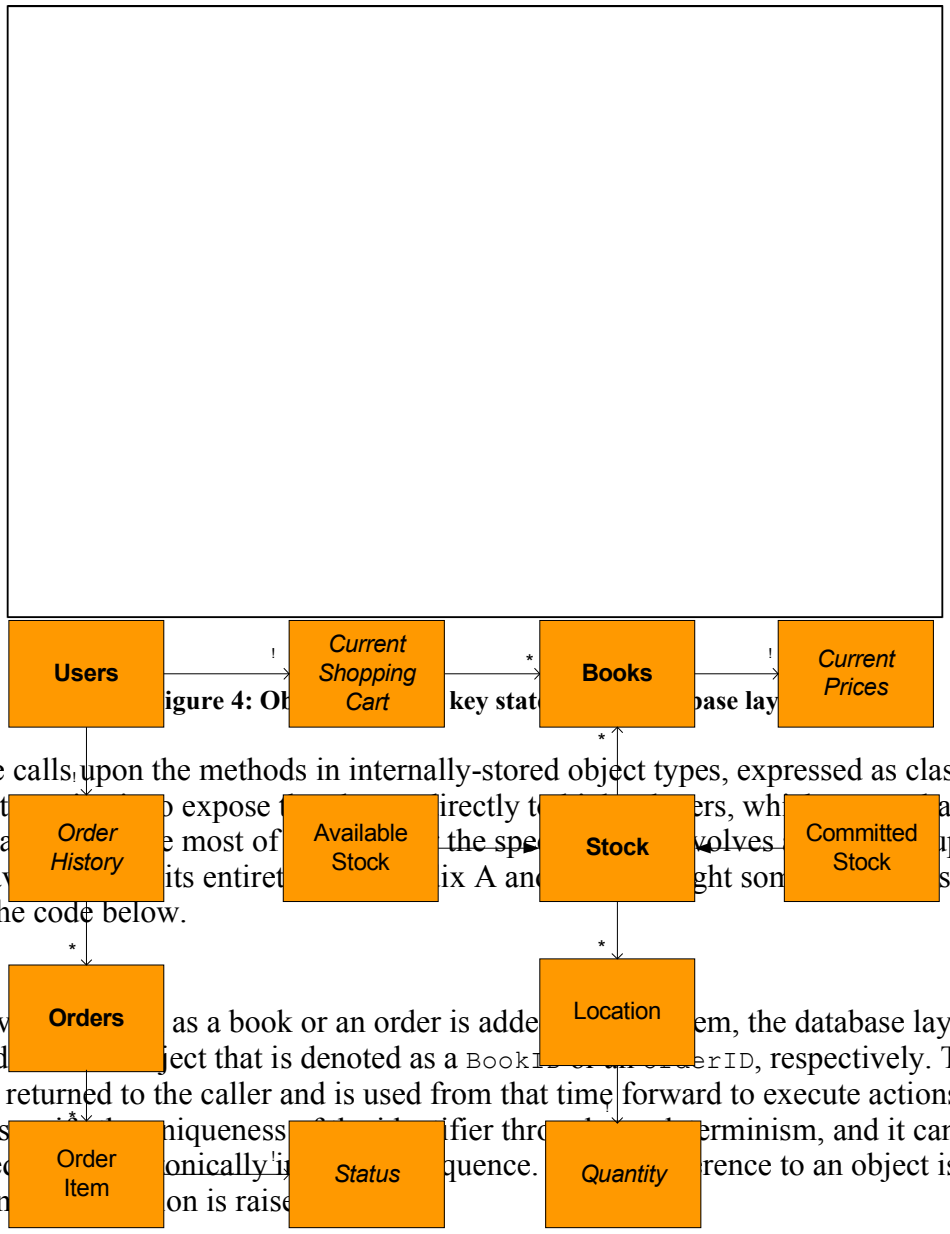
The database layer maintains the stable state that the system relies on for saving and displaying information. This layer is typically implemented by a relational database management system (RDBMS), where classes of objects are held within tables and instances of objects are stored in the tables as records. Higher layers are able to interface with the database system through the SQL query language.

It is not our goal to even begin to attempt to reinvent a database system. Rather, we will abstract the complexities of transactional processing in order to focus on the key functionality that the database layer should expose for book buying. These functionalities are represented as methods in a module, `DatabaseLayer`.

Listing 4: States maintained in the database layer

```
MODULE DatabaseLayer EXPORT .... =
   VAR
      books : BookID -> Book                % the catalog of books
      prices : BookID -> Price              % the current prices of the books
      availableStock := stock.new();        % stock that can be purchased
      committedStock := BookID -> quantity; % stock that has not been shipped, but has
been ordered
      orders : OrderID -> Order             % the orders in the system

      % the users of the system
      users : UserID -> [userName : String, userInfo, password : String,
                         currentShoppingCart : ShoppingCart,
                         orderHistory : SEQ OrderID := {} ]
   ..
 END DatabaseLayer
```



igure 4: Ob key stat base lay

The module calls upon the methods in internally-stored object types, expressed as classes in Spec. An alt o expose t irectly t ers, wh ance clarity and understa e most of the spe volves updating state, we ha its entiret ix A an ght son st notable aspects of the code below.

When a new as a book or an order is adde em, the database layer assigns a unique hand ject that is denoted as a `BookID` or an `OrderID`, respectively. This identifier is returned to the caller and is used from that time forward to execute actions on that object. We s niqueness of identifier thro erminism, and it can be implemente onically i quence. rence to an object is not found in the systen on is raise

Spec provides several alternatives to represent object contents. In our code we simplify things by representing object types as records for the most part. These types can be constructed and passed around easily. However, for accuracy and flexibility, the database layer should represent types by functions (see Listing 5).

Listing 5: Illustration of how the `Book` record can be converted to a function representation

```
TYPE DBRecord = String -> (String + Date + Int + Boolean + Decimal)

FUNC pad(str, padNum) = <<
  DO str.size < padNum =>
    str = "0" + str;
  OD
  RET str;
```

```
  >>

  FUNC convertToDBRecord(book) -> DBRecord = <<
    VAR r : DBRecord |
      r("title") = book.title;
      r("author") = book.author;
      r("isbn") = book.ISBN.countryID + book.ISBN.publisherID
                    + book.ISBN.itemID + book.checksum;
      r("retailprice") = book.retailPrice.whole + "."
                       + book.retailPrice.decimal;
      r("numberPages") = book.numberPages;
      r("editionNumber") = book.editionNumber;
      r("publicationDate") = book.publicationDate.year + "/" +
                             pad(book.publicationDate.month, 2) + "/"
                             pad(book.publicationDate.day, 2);
      ...
    RET r;
  >>
```

Representing database objects with functions allows us to refer to and return specific fields while still adhering to the rules of the Spec language. However, due to the overhead of parsing the representation, we still prefer to refer to records whenever possible.

An example of how the function-based representation is useful is in the `searchBooks` function to return fields of books according to some matching criteria and a sort order (see Listing 6). Matching constraints can be placed on multiple field names, and the match is a partial order so as to not limit the implementation to a strict equality. The field names to return are indicated explicitly.

Listing 6: Searching for books in the database

```
TYPE ordering = SEQ (fieldName : String, (ASCENDING + DESCENDING))

% returns the book ID and a sequence of field values for each book matching
% the criteria
FUNC searchBooks(fieldName : String, fieldValue : String, ordering,
                 fieldNamesToReturn : SEQ String) -> SEQ (BookID, SEQ String) = <<
  VAR bookIDs := { bookID :IN books.dom |
                   fieldValue <<= books(bookID)(fieldName) | bookID },
      sortedBookIDs := {ALL t | bookIDs.count(t) = sortedBookIDs.count(t))
                               /\ sorted(sortedBookIDs, ordering)},
      i := 0, returnValues : SEQ(BookID, SEQ String) := {} |

    DO i < sortedBookIDs.size =>
      VAR j := 0, fieldValues : SEQ String := {} |
        DO j < fieldNamesToReturn.size =>
          fieldValues := fieldValues +
                         books(sortedBookIDs(i))(fieldNamesToReturn(j));
          j := j + 1;
        OD
      returnValues := returnValues + (sortedBookIDs(i), fieldValues);
      i := i + 1;
    OD;
    RET returnValues;
>>

FUNC inSortedOrder(sortOrder : (ASCENDING + DESCENDING),
                   value1 : String, value2 : String) -> Bool =
```

```
  ((sortOrder = ASCENDING /\ bookIDs(i-1)(fieldName)
     <= bookIDs(i)(fieldName))
   \/
   (sortOrder = DESCENDING /\ bookIDs(i-1)(fieldName)
     >= bookIDs(i)(fieldName)))

FUNC sorted(bookIDs : SEQ BookID, ordering) -> Bool =
  VAR o := 0 |
   RET (ALL i :IN bookIDs - {0}, ALL o :IN 0..ordering.size-1 |
          VAR fieldName := ordering(o)(0), sortOrder := ordering(o)(1) |
          IF o > 0 =>
             VAR prevFieldName := ordering(o-1)(0) |
             % sort order of secondary fields apply only if
             % previous field is equal
                bookIDs(i-1)(prevFieldName) = bookIDs(i)(prevFieldName) =>
                  inSortedOrder(book, bookIDs(i-1)(fieldName),
                                bookIDs(i)(fieldName))
          [*]
              % sort according to primary field
              inSortedOrder(book, bookIDs(i-1)(fieldName),
                            bookIDs(i)(fieldName))
          FI
       )
```

There are two different stocks in the system, available and committed. The latter is used to keep track of stock that has already been purchased, but has not yet been shipped. Both stock are instances of a single type (see Listing 7). In our specification, we have not placed any constraints on the number of locations from which a book can be shipped. Sites such as Amazon.com often attempt to minimize shipping time and cost by distributing popular items in multiple locations.

Listing 7: Stock abstracts the quantity and locations of books

```
CLASS Stock EXPORT add, edit, getQuantity =
  VAR stock : BookID -> StockInfo
  PROC add(bookID, location, quantity) = <<
    IF stock!bookID /\ stock(bookID)!location =>
      stock(bookID)(location) += quantity;
    [*]
      stock(bookID)(location) = quantity;
    FI
  >>
  PROC remove(bookID, location, quantity) = <<
    IF stock!bookID /\ stock(bookID)!location =>
      stock(bookID)(location) -= {stock(bookID)(location), quantity}.min;
    [*] SKIP FI
  >>
  FUNC getQuantity(bookID) -> Int = <<
    RET +:(stock(bookID).rng) + 0;
  >>
  FUNC getStock(bookID) -> StockInfo RAISES {NotFound} = <<
    IF ~stock!bookID => RAISE NotFound
                      [*] RET stock(bookID);
    FI
  >>
  % returns zip codes of where the books are stored
  FUNC getLocations(bookID) -> SET String RAISES {NotFound} <<
    IF ~stock!bookID => RAISE NotFound
       [*] RET {VAR location :IN stock(bookID).dom | | location.locationZipcode};
    FI
  >>
```

```
END Stock
```

In specifying orders, we have chosen to allow users to cancel items in an order, and to allow the book seller to ship a single item at a time. We do, however, impose the constraint that partial quantities of a single item cannot be shipped or cancelled.

Listing 8: Order contains shipping, billing, cost, and item information

```
CLASS Order EXPORT cancelItem, shipItem,
                   editBillingAddress, editShippingAddress, editPaymentInfo,
                   editShippingMethod, editShippingCost, editTax =
  % states of the order item
  TYPE ACTIVE = 0 % this is the default
       CANCELLED = 1
       SHIPPED = 2
       ItemStatus = (ACTIVE + CANCELLED + SHIPPED)
  VAR orderDate : Date
      billingAddress : Address
      shippingAddress : Address
      paymentInfo : PaymentInfo % this is basically credit card
      items : SEQ (bookID, quantity, unitPrice)
      status : BookID -> ItemStatus := (\ bookId | ACTIVE )
      % can ship each item separately (but not partial quantities of items)
      shippingInfo : BookID -> ShippingInfo
      % single shipping method and cost
      shippingMethod : String
      shippingCost : Price
      tax : Price
  % initialize the order
  APROC new(billingAddress, shippingAddress, paymentInfo, items) -> Order = << ... >>
  FUNC getOrderItems(itemStatus) -> SET BookID = <<
    RET {VAR bookID :IN status.dom | | status(bookID) = itemStatus}
  >>
  PROC cancelItem(bookID) = <<
    status(bookID) = CANCELLED;
  >>
  PROC shipItem(bookID, shippingInfo) = <<
    status(bookID) = SHIPPED;
    shippingInfo(bookID) = shippingInfo;
  >>
  PROC editBillingAddress(billingAddress : Address) = << ..  >>
  PROC editShippingAddress(shippingAddress : Address) = << .. >>
  PROC editPaymentInfo(paymentInfo) = << .. >>
  PROC editShippingMethod(shippingMethod : String) = << .. >>
  PROC editShippingCost(shippingCost : Price) = << .. >>
  PROC editTax(tax : Price) = << .. >>

END Order
```

3.3 Logic Layer

The logic tier of the system acts as the intermediary between the presentation and database layers; it uses the information stored in the database to respond to the requests of the presentation layer. It has no persistent state, just a table associating session IDs with a shopping cart and optional user information. This table is flushed periodically and no guarantee is made about the stability of the data.

Listing 9: States maintained in the logic layer

```
Module LogicLayer EXPORT ... =

CONST maxSID

TYPE SID          = [0 .. maxSID]
     Session      = [uID: UserID + Null, username: String + Null,
                     firstname: String + Null,
                     cart: ShoppingCart + Null, loggedIn: Bool]

VAR  sessions      : SID -> Session    := {}
```

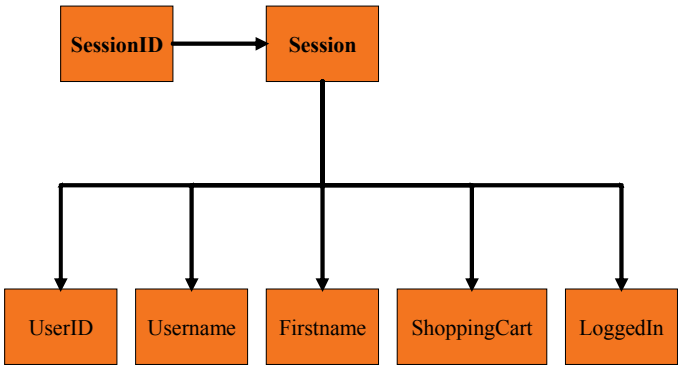


Figure 5: Object model of states in the logic layer

All requests from the presentation layer must contain a session ID, unless it is a request for a new session (see Listing 10). If a user ID is available, it is included in the session request and used to retrieve information from the database layer that is then stored with that session ID. We choose to load the user name, first name, and shopping cart. If no user ID is provided, it leaves user ID, user name, and first name blank and creates a new shopping cart.

Listing 10: Creating a new session

```
PROC NewSession(userID : UserID + Null) RAISES {NotFound} -> SID  =
  VAR sID := NewSID()
     session := Session{uID := userID, username := nil, firstname := nil
                        cart := new ShoppingCart, loggedIn := false}
     user |
  IF userID # nil =>
    user := DatabaseLayer.getUserInfo(userID);
    session := session{username := user.userName, firstName := user.userInfo.firstName
                       cart := user.currentShoppingCart}
  FI
  sessions := sessions{sID -> session};
  RET sID

PROC NewSID() -> SID =
  VAR sID | !(sID IN sessions.dom()) |
  RET sID;
```

When a user logs in, their session cart, stored in the logic layer, is merged with persistent cart, stored in the database layer, and then migrated to the persistent layer. Thereafter, all cart actions are performed on the persistent cart stored in the database layer. All actions that change the state of the database layer require the user to be logged in. The only actions that can be performed without logging in are searching, viewing book details, and editing the shopping cart. Logging

out ends the session in all cases. If the user was logged in, the shopping cart is kept on the database layer. If the user was not, the shopping cart is lost.[1]

Listing 11: Maintaining the shopping cart

```
PROC NewUser(sID, uname : String, password : String) RAISES {NoSuchSession},
{UserExistsAlready} -> UserId =
 IF sID IN sessions.dom() =>
  IF !DatabaseLayer.isUserNameInUse(uname) =>
    <<VAR userID := addUser(uname, password)
                session := sessions(sID)
        items := session.cart.getItems() |
     DO items.size() > 0 =>
       VAR item : BookID | item IN items |
       DatabaseLayer.addShoppingCartItem(userID, item);
       DatabaseLayer.setShoppingCartQuantity(userID, item, session.cart.getQuantity
(item));
       items - := SEQ {item}; OD
        sessions(sID) := session{uID := userID, username := uname, cart := null,
loggedIn := true};
        RET userID; >>
    [x] RAISE {UserExistsAlready} ; FI
[x] RAISE {NoSuchSession} FI
PROC SignIn(sID, uname : String, pass : String) -> UserID RAISES {NoSuchSession},
{NotFound}, {WrongPassword}  =
IF sID IN sessions.dom() =>
  VAR userID := getUserID(uname),
    password := DatabaseLayer.getUserPassword(userID) |
  IF pass = password =>
     VAR session := sessions(sID),
         items := session.getItems(),
         lnum,
         dnum |
     <<DO items.size() > 0 =>
        VAR item: BookID | item IN items |
            lnum := session.cart.getQuantity(item);
            dnum := DatabaseLayer.getShoppingCartQuantity(userID, item);
          DatabaseLayer.setShoppingCartQuantity(userID, item, {lnum, dnum}.max);
          items - := {item}; OD
          user := DatabaseLayer.getUserInfo(userID)
          sessions(sID) := session{uID := userID, username := uname,
                                                 firstname := user.firstName,
                                                 cart := nil,loggedIn := true};
          RET userID >>
  [x] RAISE {WrongPassword}; FI
[x] RAISE {NoSuchSession} FI
%==========Cart Actions==================
APROC AddToCart(sID, bookID) RAISES {NotFound}, {NoSuchSession} =
IF sID IN sessions.dom() =>
  VAR session := sessions(sID) |
  IF session.loggedIn =>
    addShoppingCartItem(session.uID, bookID);
  [x] session.cart.addItem(bookID) FI
[x] RAISE {NoSuchSession} FI

APROC ChangeQuantity(sID, bookID, newQuantity : Int) RAISES {NotFound} =
IF sID IN sessions.dom() =>
```

[1] Note that this is not the behavior of Amazon.com, which saves any items added to the shopping cart even when user has not logged in. Doing so probably increases sales, but at the expense of annoying users who must delete items they never intended to purchase (e.g. simply viewing the price of some items now require adding them to the shopping cart).

```
  VAR session := sessions(sID) |
  IF session.loggedIn =>
    DatabaseLayer.setShoppingCartQuantity(session.uID, bookID, newQuantity);
    [x] session.cart.setQuantity(bookID, newQuantity); FI
[x] RAISE {NoSuchSession} FI

PROC GetCart(sID) -> SEQ (BookID, Int) RAISES {NoSuchSession} =
IF sID IN sessions.dom(); =>
  VAR session := sessions(sID),
       rCart : SEQ (BookID, Int) := {},
        cart : ShoppingCart |
  IF session.loggedIn =>
    cart := DatabaseLayer.getShoppingCartItems(session.uID);
  [x]
    cart := session.cart.getItems()
  FI
  DO cart.size() > 0 =>
  VAR item : BookID | item IN cart |
     IF session.loggedIn =>
       rCart + := {(item, DatabaseLayer.getShoppingCartQuantity(session.uID, item))};
       [x] rCart +:= {(item, session.cart.getQuantity(item))}; FI
            cart - := {item}; OD
         RET rCart;
[x] RAISE {NoSuchSession} FI
```

The shopping cart presents an area where functionality may vary greatly. It is not obvious how the shopping cart should change when users log in and out. We choose to migrate the cart at login and keep the cart on the database at log out.

Another area where questions arise is the ordering process. A user should be able to edit an order that has not been shipped, change addresses and shipping methods, or cancel the order. The question is whether you should be able to edit this information by item or just by order. The logic layer is also responsible for calculating the tax and shipping, which is complicated when an order can be shipped to multiple addresses. A book seller may also wish to ship from different locations to minimize costs. This affects shipping and the timing; a whole order cannot be shipped at the same time if there are multiple addresses. We choose to allow multiple shipping origins but only one shipping destination, since a user can always simply place multiple orders.

Listing 12: Managing an order

```
PROC CreateOrder(sID, billingAddress : Address, shippingAddress : Address,
                 paymentInfo : PaymentInfo, shippingMethod) RAISES {NoSuchSession},
{NotLoggedIn}, {EmptyCart} -> OrderID =
IF sID IN sessions.dom() =>
     VAR session := sessions(sID) |
     IF session.loggedIn =>
        VAR cart := DatabaseLayer.getShoppingCartItems(userID),
            orderItems : SEQ (BookID, Int, Price) := {} |
        <<DO cart.size() > 0 =>
                  orderItems + := {(cart.head(),
DatabaseLayer.getShoppingCartQuantity(userID, cart.head()),
                                    DatabaseLayer.getPrice(cart.head()))}; OD
             RET DatabaseLayer.newOrder(session.uID, billingAddress, shippingAddress,
paymentInfo, shippingMethod,
                                getTax(sID, shippingAddress), getShipping
(sId,shippingMethod, shippingAddress),
                                orderItems);>>
```

```
        [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

APROC CancelItem(sID, orderID, bookID) RAISES {NoSuchSession}, {NotValidOrder},
{NotActiveItem},{NotLoggedIn} =
IF sID IN sessions.dom() =>
     VAR session := sessions(sID) |
     IF session.loggedIn =>
       VAR orders := DatabaseLayer.getOrderHistory(session.uID) |
       IF orderID IN orders =>
         VAR activeItems := DatabaseLayer.getOrderItems(orderID, "ACTIVE") |
         IF bookID IN activeItems.size =>
              DatabaseLayer.cancelItem(orderID, bookID);
            [x] RAISE {NotActiveItem} ; FI
         [x] RAISE {NotValidOrder}; FI
       [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI
```

Revisiting the question of persistence, several procedures in the specification for the logic layer model the potential loss of data. In the event of a crash, all of the sessions are lost. This is allowable because if the user was logged in, all the data can be easily recovered by looking it up with the user ID. If the user was not logged in, they will lose their shopping cart. We expect this layer to run on a reasonably stable machine which will make crashes infrequent enough to render this loss negligible. We also need to flush sessions periodically. The implementation would need a specific policy for this. One possibility is flushing transactions that are inactive for a set period of time. We modeled this by having a function that deletes an arbitrary session.

Listing 13: State maintenance

```
PROC Flush()
     %Implementation will have a housecleaning routine that flushes sessions that have
expired
     VAR sID | sID IN sessions.dom() |
     sessions := sessions{sID -> };


PROC Crash() =
     %This layer lacks persistent state and will lose all sessions in the event of a
crash
     sessions := {* -> };
```

Please see Appendix B for the full code to the logic layer.

3.4 Presentation Layer

The presentation layer of our book-buying system is responsible for communicating between the logic layer and the end-user's computer. As such it receives requests to load a new page, for instance, or to append some information to a user's account. The presentation layer retrieves whatever information is necessary for it to complete this action from the logic layer, and passes the content back to the end user's computer. All the while, it maintains no state whatsoever.

Our specification for the presentation layer is divided into modules that each represent a page. Typically, modules contain a load function to return whatever information is needed to display

the page properly. It is presumed that another process generates actual HTML code from this information, a feature that is also part of the presentation layer.

Each module also contains functions that allow the user to jump to a different page. This format is successful because user interactions with any given page will load a new page; for instance, if someone is looking at the detail page for a book she or he is thinking of buying, possible interactions include adding the book to the shopping cart, which would cause the shopping cart page to load, typing in a new search to look for something else, which would cause the search results page to load, or jumping back to the homepage to start browsing books again. As such, the layout of each page is fairly simple, and is consistent across the fourteen web pages specified here with few exceptions. Listing 14 uses the Search Results page to illustrate this basic formatting of modules in the presentation layer.

Listing 14: Sample Web Page

```
MODULE SearchResultsPage

  PROC load(userID: UserID + Null, sessionID : SID + Null,
            searchField: String, searchValue: String) RAISES {NoSuchSession},
{NotFound} -> SEQ (BookID, SEQ String) =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    VAR searchResults := LogicLayer.Search(sessionID, searchField, searchValue,
ASCENDING, ("title", "price", "shippingSpeed", "edition"))
    RET searchResults;

  PROC goToShoppingCartPage(userID: UserID + Null, sessionID : SID + Null) RAISES
{NoSuchSession}, {NotFound} -> SEQ (BookID, Int)=
    RET ShoppingCartPage.load(userID, sessionID);

  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
{NoSuchSession}, {NotFound} -> String =
    RET HomePage.load(userID, sessionID);

END SearchResultsPage
```

When the presentation layer is confronted with an error, it handles it by sending the client layer to a page saying that the error has been encountered rather than attempting to load the page the client requested. We have represented this in our specification simply by raising errors.

Please see Appendix C for the full code to the presentation layer.

3.5 Client Layer

The client layer is the Web browser used to display site content. Browsers keep semi-persistent data in the form of cookies, and it is this cookie information that allows the book-buying system to remember whether or not a user has previous visited the page. Cookies can also be used to keep track of the session ID. The system uses the session ID to associate necessary session-

related information such as shopping carts with potential buyers and often to monitor and track user behavior.

We say that the client layer only keeps semi-persistent information because at any time the user can delete cookie files, disable cookie functionality, or log on from another browser where no cookie state exists. There is also no way to ascertain that the cookie data actually belongs to the user currently surfing the page. Privacy concerns are in fact the very reason why many users periodically flush their browser cookies or turn off the use of cookies. The system minimizes display of information that may violate the privacy of the user named in the cookie data by always requiring explicit sign on whenever orders are placed or financial transactions take place. The system can also function fully when cookies are turned off by using an alternate mechanism for storing session ID's such as URL rewriting or keeping hidden fields within forms.

4 Implementation Issues

4.1 Load Balancing

A major issue with all web-based systems is load balancing. To scale up a system, we replicate one part of it and use many machines to perform the function of one. This creates a new problem of how to distribute the work among the different machines, since not doing so leaves resources unused and defeats the purpose of replication. Work distribution is typically performed by placing a load balancing or switching application between the client and the replicated servers. This switch can be placed on a separate machine or on one of the servers. It should be fault-tolerant, or otherwise the switching application will be a single point of failure for the entire system.

In our four-tier architecture, it is useful to implement load balancing for at least two of the layers: the presentation and logic layers.[2] We can distribute the load in the logic layer by allocating sessions to different machines. Each session interacts with one server in the layer, which will locally associate the session ID with the necessary information. It is also possible to distribute data for a single session among multiple machines, but this approach is not worthwhile because the data is not persistent. The load-balancing application for the logic layer will be responsible for determining which server to send a new session request to and for ensuring that all future requests get forwarded to the correct server. If a server goes down, all of its sessions are lost and a new session request is necessary.

In the presentation layer, work distribution is simpler because there is no state to be maintained. Any request from the client can be processed by any presentation layer server. The load-balancing application will be responsible for determining where each request should be sent based on previous traffic and the current load on all of the servers.

[2] Replication of the database server could necessitate load balancing for that layer as well, but the implementation concept would be the same as that of the presentation layer.

The biggest difficulty of the implementation is moving the assignment and flushing of the session IDs. Our implementation assigns and flushes the IDs in the load-balancing application of the logic layer. The load-balancing application maintains a table of IDs and migrates changes to the servers. If a server crashes, the load balancer flushes all of the sessions assigned to that server. Another issue is determining when the server has crashed. Our implementation queries the server and updates its status based on the response.

Further details would be necessary for a fully-functional implementation. When routing a new packet to the presentation layer or a new session to the logic layer, we choose the server in a non-deterministic manner. A strategy for this choice would be an important component of the load-balancing application. Choosing a server with less load than most of the others is an obvious choice, though a random scheme is a simple and common alternative.

Listing 15: Implementing load balancing in the presentation and logic layers

```
MODULE PresentationLoadBalancer EXPORTS RoutePacket =

  TYPE Request
       Server
       ServerID

  VAR servers : ServerID -> Server
      alive : ServerID -> Bool

  Thread Alive() =
    VAR serverID | serverID IN servers.dom() |
    IF !servers(serverID).alive() => alive(serverID) := false FI

  PROC RouteRequest(request) =
    VAR serverID | serverID IN servers.dom() /\ alive(serverID) |
    servers(serverID).process(request);

END MODULE

MODULE LogicLoadBalancer EXPORTS RouteRequest, NewSID, FlushSession =

  TYPE Request
       Server
       ServerID

  VAR servers : ServerID -> Server
      alive : ServerID -> Bool

      sessions : SID -> ServerID

  THREAD Alive() =
    VAR serverID | serverID IN servers.dom() |
    IF !servers(serverID).alive() =>
       alive(serverID) := false;
       DO (exists sID | sessions(sID) = serverID) =>
          VAR sID | sessions(sID) = serverID |
          FlushSession(sID); OD FI

  PROC NewSID() -> SID =
    VAR sID | !(sID IN sessions.dom()) |
    RET sID;
```

```
  PROC RouteRequest(request) RAISE {NoSuchSession} =
    IF request.getSID = null =>
       VAR newSID := NewSID()
            serverID | serverID IN servers.dom() /\ alive(serverID) |
       request.getSID := NewSID();
       sessions := sessions(newSID -> serverID);
       servers(sessions(newSID)).process(request);
       [x] IF alive(sessions(request.getSID)) =>
              servers(sesions(request.getSID)).process(request);
              [x] RAISES {NoSuchSession} FI FI
  PROC FlushSession(sID) =
    servers(sessions(sID)).logout(sID);
    sessions := sessions{sID ->};

  PROC Flush() =
    VAR sID | sID IN sessions.dom() |
    FlushSession(sID);

=====================================================================
%In both the Logic and Presentation layer we add this proc:
PROC Alive() =
  RET true;

=====================================================================
%All of the procedures, other than NewSession, must check that the SID exists
IF !(sID IN sessions.dom()) => RAISE {NoSuchSession} FI

%In the logic layer, need to remove flush and NewSID

%No longer need to make new SID in the logic layer
PROC NewSession(sID, userID : UserID + Null), RAISES {NotFound} -> SID  =
     session := Session{uID := userID, username := nil, firstname := nil
                        cart := new ShoppingCart, loggedIn := false}
     user |
     IF userID # nil =>
        user := DatabaseLayer.getUserInfo(userID);
        session := session{username := user.userName, firstName :=
user.userInfo.firstName
     cart := user.currentShoppingCart} FI
     sessions := sesions{sID -> session};
     RET sID
```

4.2 Fault-Tolerance

In a multi-tiered system, faults can occur in a single machine of a layer, or faults can occur across an entire layer. Of course, these types of faults are not mutually exclusive, especially if there is only one machine in a layer.

Presumably, in a typical bookselling website implementation, there will be multiple machines in each layer, and these machines will be configured in such a way that some degree of load balancing will occur between them. Thus if a single machine in the presentation layer, which maintains no state, fails, then fault tolerance and load balancing are synonymous concepts: redistributing the load between the functioning machines solves the problem of fault tolerance completely. As for the logic layer, whose maintained state is not persistent, the simplest option might be to allow the layer not to be fault tolerant at all (beyond redistributing load between

remaining machines): return some sort of error statement and lose whatever non-persistent state had been maintained. Customers might be annoyed at having to sign in again or refill their shopping carts, but no real damage has been done, especially if these failures are relatively infrequent. As for the database layer, loss of state is a far graver situation. Thus replication strategies must be employed to ensure that state is maintained to whatever degree of confidence is needed.

Perhaps the simplest fault tolerance strategy to employ in the database layer is to simply replicate everything in the database layer, possibly through a RAID machine configuration. On the other hand, a strategy that offers less fault tolerance but greater efficiency might be to partition the information stored in the database. This information naturally takes three forms: books, stock, and users. A vendor might choose to partition the books across multiple servers and employ less replication or none at all. In this case, if a server were to fail, users who were searching for a specific title stored on that server might notice the failure in that the book they were searching for could not be found. If, on the other hand, a user was browsing the site without a specific title in mind, the loss of a small subset of books from that site might not be noticed at all. In this way, a weaker form of fault-tolerance than pure replication is employed because the severity of failures is lessened. Our vendor might also choose to partition the user information across several servers. In this instance, some form of replication might be necessary after the partitioning takes place since losing user information is more serious than losing books. However, partitioning the servers by user has the side benefit that it provides a natural form of load-balancing because requests from the logic layer keyed to a user whose name starts with a particular letter of the alphabet can be directed only to the appropriate server.

A final consideration in discussing the fault tolerance of this system is that the load-balancing application we have specified creates a single point of failure. As such, load balancing servers must be replicated as well.

4.3 Caching

Caching takes advantage of locality of reference to speed up retrieval of data. We can implement a software cache in the logic layer to avoid the latency of accessing the database layer for frequently-requested information. The performance savings of caching can be dramatic, especially when the bandwidth between the database and logic layers is low or a great deal of traffic is regularly transferred.

While conceptually similar, we do not consider implementing caching in the presentation layer since the potential speed up may be significantly offset by implementation complexity and performance hits from having to synchronize across multiple layers. However, we expect that any static content in the presentation layer would automatically cached in memory.

The specifics of the caching implementation is similar to the way caching is performed for disk drives. A replacement policy such as least recently used can be implemented. Read-ahead functionality can cache content that will likely be requested by the logic layer in the future.

An important difference between caching for disk drives and caching in this environment is that there may be changes made to the database independent of the logic layer, and these changes must be propagated to the higher layer. For instance, if a separate administrative interface is available for editing database contents, or if a rollback occurs in the database. One solution is to send a message to the logic layer to flush any associated cache objects. When there are multiple machines in the logic layer, a less efficient broadcast message may be replaced by a directory scheme whereby the database layer keeps track of which logic layer machines may have relevant cached content.

A write-through scheme for cache updates is the most simple option for the system. Supporting a delayed update scheme requires dealing with significant complexities; for instance, if the logic layer machine containing the update fails, or if another session accesses the updated content through a different logic layer machine before the database layer reflects the change. In the latter scenario, the cache must be implemented in a distributed manner, through either a broadcast scheme or a directory scheme that keeps track of read and write locks on the object.

Lost updates may not be catastrophic in many circumstances, since a Web user who does not immediately see changes reflected will likely reenter the information. Hence, it may be a worthwhile consideration to batch update commands from the logic layer to amortize connection costs. To minimize lost updates in case of failures, one should choose a small time slice from which to batch and write unsent updates to stable storage in the logic layer so that they can be executed when the machine is resurrected[3].

Conceivably, we can cache any and all objects that the logic layer retrieves from the database layer and simply use a replacement policy when the cache is full. However, we can probably do better by prioritizing objects to cache according to type. For example, caching frequently-accessed books displayed on the front page should be more beneficial than caching order details from the order history. Book information is relevant to both anonymous and logged in users, and includes the minimal fields displayed within search results or "spotlight" sections as well as per-book details displayed on a single page. Caching the persistent shopping cart would speed up the display of cart contents. Caching updates to that shopping cart would minimize delays experienced by the user when he/she clicks on the button to add a book. Similarly, caching the action to finalize the order would prevent users from having to wait to receive a confirmation, though at the expense of possible lost orders if the update does not get executed.

[3] The updates should be idempotent in case the user has already made the update.

4.4 Security

Concerns about the privacy of customer information must be addressed in order to maintain a solid customer base. In this section, we address several security issues for our book-buying system.

Customer information is saved in the database layer, a layer which in principle should not be accessible to any computers except those in the logic layer. A typical security measure employed by companies using a multi-tiered system is to place firewalls between each tier of a Web site to be as certain as possible that no unauthorized communication can occur between any two tiers of the system, or between a tier and the outside world. Thus an attacker would need to first break into a machine in the presentation layer, use that machine to break into one in the logic layer, and then use the machine in the logic layer to break into the database layer. Three separate, non-trivial attacks would need to be waged to reach the database layer.

Sensitive information such as passwords and credit card numbers are often stored within the database layer. While many measures can be taken to keep this information private, the only truly secure option is to not store the information at all. Instead, the database would store a cryptographically strong hashing of the contents, which it would compare with the hash of actual information entered during the signing in or ordering process. When information is passed between the customer and the server, for instance while the customer is logging in, a common security measure is to use a well-established cryptographic protocol, like SSL, to thwart eavesdroppers.

Breaches of security are not limited to attacks on the server side or during transmission between the client and the server. Since browser cookies keep track of session and user state, it is possible for an attacker to gain account access by obtaining or forging the cookie information. Consider a scenario where the session ID's are assigned sequentially and saved "as is" to the client's browser. Anyone can forge someone else's previously unexpired session by simply decrementing the session ID in their local cookie file. Even if session ID's are assigned randomly, there may not be enough entropy to thwart forgery. One solution is for the server to encrypt the cookie data with a secret key before sending it to the client. The server would then need to decrypt the cookie before making use of its contents. By taking this measure, only those able to access or find the secret key could gain information about session or user state -- even information about their own session ID. Additional protections like encrypting expiration information along with the usual contents can minimize replay attacks from someone who has managed to obtain another's encrypted cookie data.

5 Conclusion

In this paper, we analyzed and specified the functionality of a Web-based book-buying system. We formulated our specification in terms of a four-tier architecture consisting of the database, logic, presentation, and client layers so as to take advantage of the modularity and flexibility

provided by a multi-tier system. In the process, we have addressed multiple questions that any electronic commerce site should consider. For example, we have considered whether shopping carts should be stored as persistent state, and the extent to which changes to orders are supported.

Given that it was beyond the scope of our project to actually implement our specification, we instead considered four interesting implementation issues that are central to most computer systems: caching, security, load balancing, and fault tolerance. We analyzed these issues in the context of our four-tier book-buying architecture, describing various options for the implementation and weighing implementation complexity against potential benefits and the needs of the system.

6 References

[1] Amazon.com, Inc. http://www.amazon.com
[2] Carnegie Mellon Software Engineering Institute. “Three Tier Software Architectures,” February, 2000. Available Online: http://www.sei.cmu.edu/str/descriptions/threetier.html
[3] Handout 4: Spec Reference Manual
[4] Handout 21: Distributed Systems
[5] Handout 27: Distributed Transactions

7 Appendix A: Database Layer Specification Code

```
MODULE DatabaseLayer EXPORT .... =
  VAR
      books : BookID -> Book % the catalog of books
      prices : BookID -> Price % the current prices of the books
      availableStock := stock.new();
      committedStock := BookID -> quantity;
      orders : OrderID -> Order % the orders in the system
      % the users of the system
      users : UserID -> [userName : String, userInfo, password : String,
                         currentShoppingCart : ShoppingCart,
                         orderHistory : SEQ OrderID := {} ]
  %%%%%%%%%% cataloging %%%%%%%%%%
  FUNC addBook(book) -> BookID = <<
    VAR id | ~books!id =>
      books(id) := book;
      storeBookInDB(book);
    RET id
  >>
  PROC editBook(bookID, book) RAISES {NotFound} = <<
    IF ~books!bookID => RAISE NotFound
                      [*] books(bookID) = book;
    FI
  >>
  FUNC getBook(bookID) -> Book RAISES {NotFound} = <<
    IF ~books!bookID => RAISE NotFound
                      [*] RET books(bookID);
    FI
  >>
  TYPE ordering = SEQ (fieldName : String, (ASCENDING + DESCENDING))


  % returns the book ID and a sequence of field values for each book
  matching the criteria
  FUNC searchBooks(fieldName : String, fieldValue : String, ordering,
                   fieldNamesToReturn : SEQ String) -> SEQ (BookID, SEQ
  String) = <<
    VAR bookIDs := { bookID :IN books.dom | fieldValue <<= books(bookID)
  (fieldName) | bookID },
        sortedBookIDs := {ALL t | bookIDs.count(t) = sortedBookIDs.count
  (t)) /\ sorted(sortedBookIDs, ordering)},
        i := 0, returnValues : SEQ(BookID, SEQ String) := {} |
        DO i < sortedBookIDs.size =>
          VAR j := 0, fieldValues : SEQ String := {} |
            DO j < fieldNamesToReturn.size =>
              fieldValues := fieldValues + books(sortedBookIDs(i))
  (fieldNamesToReturn(j));
              j := j + 1;
            OD
          returnValues := returnValues + (sortedBookIDs(i),
  fieldValues);
          i := i + 1;
        OD;
        RET returnValues;
  >>

  FUNC inSortedOrder(sortOrder : (ASCENDING + DESCENDING), value1 :
  String, value2 : String) -> Bool =
    ((sortOrder = ASCENDING /\ bookIDs(i-1)(fieldName) <= bookIDs(i)
  (fieldName)) \/
     (sortOrder = DESCENDING /\ bookIDs(i-1)(fieldName) >= bookIDs(i)
  (fieldName)))

  FUNC sorted(bookIDs : SEQ BookID, ordering) -> Bool =
    VAR o := 0 |
     RET (ALL i :IN bookIDs - {0}, ALL o :IN 0..ordering.size-1 |
            VAR fieldName := ordering(o)(0), sortOrder := ordering(o)(1)
  |
            IF o > 0 =>
               VAR prevFieldName := ordering(o-1)(0) |
                  % sort order of secondary fields apply only if
  previous field is equal
                  bookIDs(i-1)(prevFieldName) = bookIDs(i)
  (prevFieldName) =>
                    inSortedOrder(book, bookIDs(i-1)(fieldName), bookIDs
  (i)(fieldName))
            [*]
                % sort according to primary field
                inSortedOrder(book, bookIDs(i-1)(fieldName), bookIDs(i)
  (fieldName))
            FI
         )

  %%%%%%%%%% stocking %%%%%%%%%%
  PROC addAvailableStock(bookID, location, quantity) = <<
    availableStock.add(bookID, location, quantity);
  >>
  PROC removeAvailableStock(bookID, location, quantity) = <<
    availableStock.remove(bookID, location, quantity);
  >>
  FUNC getAvailableStock(bookID) -> StockInfo RAISES {NotFound} = <<
    RET availableStock.getStock(bookID);
  >>
  FUNC getAvailableQuantity(bookID) -> Int = <<
    RET availableStock.getQuantity(bookID);
  >>
  PROC commitStock(bookID, quantity) = <<
    IF ~committedStock!bookID =>
      committedStock(bookID) = quantity;
    [*]
      committedStock(bookID) += quantity;
    FI
  >>
  PROC unCommitStock(bookID, quantity) = <<
    IF committedStock!bookID =>
      committedStock(bookID) -= {committedStock(bookID), quantity}.min;
    [*] SKIP FI
  >&gt;
  FUNC getCommittedQuantity(bookID) -> Int = <<
```

```
  RET committedStock(bookID) + 0;
>>
%%%%%%%%%% pricing %%%%%%%%%%
PROC setPrice(bookID, price) <<
  prices(bookID) = price;
>>
FUNC getPrice(bookID) -> Int RAISES {NotFound} = <<
  IF ~prices!bookID => RAISE NotFound
                    [*] RET prices(bookID);
  FI
>>
%%%%%%%%%% shopping cart %%%%%%%%%%
PROC emptyShoppingCart(userID) RAISES {NotFound} =  <<
  IF ~users!userID => RAISE NotFound
   [*] users(userID).currentShoppingCart.empty();
  FI
>>
PROC addShoppingCartItem(userID, bookID) RAISES {NotFound} =  <<
  IF ~users!userID => RAISE NotFound
   [*] users(userID).currentShoppingCart.addItem(bookID);
  FI
>>
PROC removeShoppingCartItem(userID, bookID) RAISES {NotFound} =  <<
  IF ~users!userID => RAISE NotFound
   [*] users(userID).currentShoppingCart.removeItem(bookID);
  FI
>>
FUNC getShoppingCartItems(userID) -> SEQ BookID RAISES {NotFound} =
<<
  IF ~users!userID => RAISE NotFound
   [*] RET users(userID).currentShoppingCart.getItems();
  FI
>>
FUNC getShoppingCartQuantity(userID, bookID) -> Int RAISES {NotFound}
=  <<
  IF ~users!userID => RAISE NotFound
   [*] RET users(userID).currentShoppingCart.getQuantity(bookID);
  FI
>>
PROC setShoppingCartQuantity(userID, bookID, quantity) RAISES
{NotFound} =  <<
  IF ~users!userID => RAISE NotFound
   [*] users(userID).currentShoppingCart.setQuantity(bookID,
quantity);
  FI
>>
%%%%%%%%%% ordering %%%%%%%%%%
FUNC newOrder(userID, billingAddress : Address,
                      shippingAddress : Address,
                      paymentInfo : PaymentInfo,
                      items : SEQ (bookID, quantity, unitPrice),
                      shippingMethod : String,
                      shippingCost : Price,
                      tax : Price) -> OrderID RAISES {NotFound}= <<
  IF ~users!userID => RAISE NotFound
  [*] VAR id | ~orders!id =>
    orders(id) = order.new(billingAddress, shippingAddress,
paymentInfo, items);
    editShippingMethod(id, shippingMethod);
    editShippingCost(id, shippingCost);
    editTax(id, tax);
    users(userID).orderHistory := users(userID).orderHistory + id;
  FI
  RET id
>>
FUNC getOrderItems(orderID, itemStatus) -> SET BookID RAISES
{NotFound} = <<
  IF ~orders!orderID => RAISE NotFound
   [*] orders(orderID).getOrderItems(itemStatus);
  FI
>>
PROC cancelItem(orderID, bookID) RAISES {NotFound} = <<
  IF ~orders!orderID => RAISE NotFound
   [*] orders(orderID).cancelItem(bookID);
  FI
>>
PROC shipItem(orderID, bookID, shippingInfo) RAISES {NotFound} = <<
  IF ~orders!orderID =&gt; RAISE NotFound
   [*] orders(orderID).shipItem(bookID, shippingInfo);
  FI
>>
PROC editBillingAddress(orderID, billingAddress : Address) RAISES
{NotFound} = <<
  IF ~orders!orderID => RAISE NotFound
   [*] orders(orderID).editBillingAddress(billingAddress);
  FI
>>
PROC editShippingAddress(orderID, ShippingAddress : Address) RAISES
{NotFound} = <<
  IF ~orders!orderID => RAISE NotFound
   [*] orders(orderID).editShippingAddress(shippingAddress);
  FI
>>
PROC editPaymentInfo(orderID, paymentInfo : PaymentInfo) RAISES
{NotFound} = <<
  IF ~orders!orderID => RAISE NotFound
   [*] orders(orderID).editPaymentInfo(paymentInfo);
  FI
>>
PROC editShippingMethod(orderID, shippingMethod : String) RAISES
{NotFound} = <<
  IF ~orders!orderID => RAISE NotFound
   [*] orders(orderID).editShippingMethod(shippingMethod);
  FI
>>
PROC editShippingCost(orderID, shippingCost : Price) RAISES {NotFound}
= <<
  IF ~orders!orderID => RAISE NotFound
   [*] orders(orderID).editShippingCost(shippingCost);
  FI
>>
PROC editTax(orderID, tax: Price) RAISES {NotFound} = <<
  IF ~orders!orderID => RAISE NotFound
```

```
    [*] orders(orderID).editTax(tax);
    FI
  >>
  FUNC getOrderInfo(orderID) => (billingAddress : Address,
                                 shippingAddress : Address, PaymentInfo)
                                 RAISES {NotFound} = <<
    IF ~orders!orderID => RAISE NotFound
     [*] RET (orders(orderID).billingAddress, orders(orderID).
  shippingAddress, orders(orderID).paymentInfo);
    FI
  >>
  FUNC getOrderHistory(userID) => (SEQ OrderID)
                                 RAISES {NotFound} = <<
    IF ~users!userID=> RAISE NotFound
     [*] RET users(userID).orderHistory;
    FI
  >>
  %%%%%%%%%% user mgmt %%%%%%%%%%
  FUNC isUserNameInUse(userName : String) -> Boolean = <<
    RET (ALL userID :IN users.dom | users(userID).userName = userName |
  userName}.count > 0
  >>
  FUNC getUserID(userName : String) -> UserID = <<
    % should only return one userID, but have a choose just in case
  something is messed up
    RET {ALL userID :IN users.dom | users(userID).userName = userName |
  userID}.choose
  >>
  FUNC addUser(userName : String, password : String) -> UserID = <<
    VAR id | ~users!id =>
      users(id) := {userName := userName, userInfo := nil, password :=
  password,
                  currentShoppingCart := ShoppingCart.new(),
                  orderHistory := {}};
    RET id
  >>
  FUNC getUserInfo(userID) RAISES {NotFound} -> UserInfo = <<
    IF ~users!userID => RAISE NotFound
                    [*] RET users(userID).userInfo;
    FI
  >>
  PROC editUserInfo(userID, userInfo) RAISES {NotFound} = <<
    IF ~users!userID => RAISE NotFound
                    [*] users(userID).userInfo = userInfo;
    FI
  >>
  FUNC getUserPassword(userID) RAISES {NotFound} -> String = <<
    IF ~users!userID => RAISE NotFound
                    [*] RET users(userID).password;
    FI
  >>
  PROC editPassword(userID, password) RAISES {NotFound} = <<
    IF ~users!userID => RAISE NotFound
                    [*] users(userID) = password;
    FI
  >>
END DatabaseLayer
```

```
================================================================================
  ==

% Stock abstracts the quantity and locations of the books
CLASS Stock EXPORT add, edit, getQuantity =
  VAR stock : BookID -> StockInfo
  PROC add(bookID, location, quantity) = <<
    IF stock!bookID /\ stock(bookID)!location =>
      stock(bookID)(location) += quantity;
    [*]
      stock(bookID)(location) = quantity;
    FI
  >>
  PROC remove(bookID, location, quantity) = <<
    IF stock!bookID /\ stock(bookID)!location =>
      stock(bookID)(location) -= {stock(bookID)(location), quantity}.
  min;
    [*] SKIP FI
  >>
  FUNC getQuantity(bookID) -> Int = <<
    RET +:(stock(bookID).rng) + 0;
  >>
  FUNC getStock(bookID) -> StockInfo RAISES {NotFound} = <<
    IF ~stock!bookID => RAISE NotFound
                        [*] RET stock(bookID);
    FI
  >>
  % returns zip codes of where the books are stored
  FUNC getLocations(bookID) -> SET String RAISES {NotFound} <<
    IF ~stock!bookID => RAISE NotFound
       [*] RET {VAR location :IN stock(bookID).dom | |
  location.locationZipcode};
    FI
  >>
END Stock

================================================================================
  ==
CLASS ShoppingCart EXPORT addItem, removeItem, getItems
                          getQuantity, setQuantity =
  VAR items : SEQ BookID % express as sequence in case order is
  important
  % add one book
  PROC addItem(bookID) = <<
    items := items + bookID;
  >>
  % remove all books of a certain ID
  PROC removeItem(bookID) = <<
    DO items.count(bookID) > 0 =>
      items := items - bookID;
    OD
  >>
  % get the books in the cart (avoid repeats)
  FUNC getItems() -> SEQ BookID = <<
    VAR books : SEQ BookID, i := 0 |
      DO i < items.size =>
        IF books.count(items(i)) = 0 =>
```

```
        books := books + items(i);
      [*] SKIP FI
      i := i + 1;
    OD
    RET books;
  >>
  % get the quantity ordered for a certain book
  FUNC getQuantity(bookID) -> Int = <<
    RET items.count(bookID);
  >>
  % get the quantity ordered for a certain book
  FUNC setQuantity(bookID, quantity) = <<
    DO items.count(bookID) < quantity =>
      addItem(bookID);
    OD
    DO quantity > 0 /\ items.count(bookID) > quantity =>
      items := items - bookID;
    OD
    IF quantity = 0 =>
      removeItems(bookID);
    [*] SKIP FI
  >>
  % empty the shopping cart
  FUNC empty() = <<
    items = {};
  >>
END ShoppingCart
=============================================================================
  ==
CLASS Order EXPORT cancelItem, shipItem,
                   editBillingAddress, editShippingAddress,
  editPaymentInfo,
                   editShippingMethod, editShippingCost, editTax =
  % states of the order item
  TYPE ACTIVE = 0 % this is the default
       CANCELLED = 1
       SHIPPED = 2
       ItemStatus = (ACTIVE + CANCELLED + SHIPPED)
  VAR orderDate : Date
      billingAddress : Address
      shippingAddress : Address
      paymentInfo : PaymentInfo % this is basically credit card
      items : SEQ (bookID, quantity, unitPrice)
      status : BookID -> ItemStatus := (\ bookId | ACTIVE )
      % can ship each item separately (but not partial quantities of
  items)
      shippingInfo : BookID -> ShippingInfo
      % single shipping method and cost
      shippingMethod : String
      shippingCost : Price
      tax : Price
  % initialize the order
  APROC new(billingAddress, shippingAddress, paymentInfo, items) ->
  Order = <<
    self := StdNew();
    orderDate := CURRENT_DATE;
    self.billingInfo := billingInfo;
    self.shippingInfo := shippingInfo;
    self.paymentInfo := paymentInfo;
    self.items := items;
    RET self
  >>
  FUNC getOrderItems(itemStatus) -> SET BookID = <<
    RET {VAR bookID :IN status.dom | | status(bookID) = itemStatus}
  >>
  PROC cancelItem(bookID) = <<
    status(bookID) = CANCELLED;
  >>
  PROC shipItem(bookID, shippingInfo) = <<
    status(bookID) = SHIPPED;
    shippingInfo(bookID) = shippingInfo;
  >>
  PROC editBillingAddress(billingAddress : Address) = <<
    self.billingAddress = billingAddress;
  >>
  PROC editShippingAddress(shippingAddress : Address) = <<
    self.shippingAddress = shippingAddress;
  >>
  PROC editPaymentInfo(paymentInfo) = <<
    self.paymentInfo = paymentInfo;
  >>
  PROC editShippingMethod(shippingMethod : String) = <<
    self.shippingMethod = shippingMethod;
  >>
  PROC editShippingCost(shippingCost : Price) = <<
    self.shippingCost = shippingCost;
  >>
  PROC editTax(tax : Price) = <<
    self.tax = tax;
  >>
END Order
```

8 Appendix B: Logic Layer Specification Code

```
Module LogicLayer EXPORT ... =

CONST maxSID

TYPE SID          = [0 .. maxSID]
     Session      = [uID: UserID + Null, username: String + Null,
    firstname: String + Null,
                     cart: ShoppingCart + Null, loggedIn: Bool]

VAR  sessions      : SID -> Session    := {}

%====================Account and User Actions================
PROC NewSession(userID : UserID + Null) RAISES {NotFound} -> SID  =
  VAR sID := NewSID()
     session := Session{uID := userID, username := nil, firstname := nil
                        cart := new ShoppingCart, loggedIn := false}
     user |
  IF userID # nil =>
    user := DatabaseLayer.getUserInfo(userID);
    session := session{username := user.userName, firstName :=
    user.userInfo.firstName
                       cart := user.currentShoppingCart}
  FI
  sessions := sessions{sID -> session};
  RET sID

PROC NewUser(sID, uname : String, password : String) RAISES
    {NoSuchSession}, {UserExistsAlready} -> UserId =
 IF sID IN sessions.dom() =>
  IF !DatabaseLayer.isUserNameInUse(uname) =>
    <<VAR userID := addUser(uname, password)
                session := sessions(sID)
        items := session.cart.getItems() |
     DO items.size() > 0 =>
       VAR item : BookID | item IN items |
       DatabaseLayer.addShoppingCartItem(userID, item);
       DatabaseLayer.setShoppingCartQuantity(userID, item,
    session.cart.getQuantity(item));
       items - := SEQ {item}; OD
        sessions(sID) := session{uID := userID, username := uname, cart :=
    null, loggedIn := true};
        RET userID; >>
    [x] RAISE {UserExistsAlready} ; FI
[x] RAISE {NoSuchSession} FI

FUNC GetUsername(sID) RAISES {NoSuchSession}, {NoID} -> String =
IF sID IN sessions.dom() =>
  VAR session := sessions(sID) |
  IF session.username # null =>
    RET session.username;
    [x] RAISE {NoID}; FI
[x] RAISE {NoSuchSession} FI

FUNC GetName(sID) RAISES {NoSuchSession}, {NoID} -> String =
IF sID IN sessions.dom() =>
  VAR session := sessions(sID) |
  IF session.firstName # null =>
    RET session.firstName;
    [x] RAISE {NoID}; FI
[x] RAISE {NoSuchSession} FI

FUNC GetUserInfo(sID) RAISES {NoSuchSession}, {NoID} -> UserInfo =
IF sID IN sessions.dom() =>
    VAR session := sessions(sID) |
    IF session.userID # null =>
        RET DatabaseLayer.getUserInfo(UserID);
    [x] RAISE {NoID}; FI
[x] RAISE {NoSuchSession} FI

PROC ChangeUserInfo(sID, fName : String, mInitial : Char, lName : String,
    eMail : String)
                                                          RAISES
    {NoSuchSession}, {NotLoggedIn}-> String =
IF sID IN sessions.dom() =>
  VAR session := sessions(sID) |
  IF session.loggedIn =>
     <<DatabaseLayer.EditUserInfo(session.uID, UserInfo{firstName :=
    fName,
                                middleInitial := mInitial, lastName :=
    lName,
                                email := eMail});
       sessions(sID).firstName := fName;
       RET fName; >>
  [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

PROC ChangePassword(sID, oldPass : String, newPass : String) RAISES
    {NoSuchSession}, {NotLoggedIn}, {WrongPassword} =
IF sID IN sessions.dom() =>
  VAR session := sessions(sID) |
  IF session.loggedIn =>
     VAR password := DatabaseLayer.getUserPassword(session.uID) |
     IF password = oldPass =>
        <<DatabaseLayer.editPassword(session.uID, newPass);  >>
     [x] RAISE {WrongPassword}; FI
  [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

PROC SignIn(sID, uname : String, pass : String) -> UserID RAISES
    {NoSuchSession}, {NotFound}, {WrongPassword}  =
IF sID IN sessions.dom() =>
  VAR userID := getUserID(uname),
    password := DatabaseLayer.getUserPassword(userID) |
  IF pass = password =>
     VAR session := sessions(sID),
         items := session.getItems(),
         lnum,
         dnum |
     <<DO items.size() > 0 =>
        VAR item: BookID | item IN items |
```

```
            lnum := session.cart.getQuantity(item);
            dnum := DatabaseLayer.getShoppingCartQuantity(userID, item);
          DatabaseLayer.setShoppingCartQuantity(userID, item, {lnum,
    dnum}.max);
          items - := {item}; OD
          user := DatabaseLayer.getUserInfo(userID)
          sessions(sID) := session{uID := userID, username := uname,
                                                  firstname :=
    user.firstName,
                                                  cart := nil,loggedIn :=
    true};
          RET userID >>
  [x] RAISE {WrongPassword}; FI
[x] RAISE {NoSuchSession} FI

PROC LogOut(sID) RAISES {NoSuchSession} =
IF sID IN sessions.dom() =>
  sessions := sessions{sID -> };
[x] RAISE {NoSuchSession}; FI

%==========Cart Actions==================
APROC AddToCart(sID, bookID) RAISES {NotFound}, {NoSuchSession} =
IF sID IN sessions.dom() =>
  VAR session := sessions(sID) |
  IF session.loggedIn =>
    addShoppingCartItem(session.uID, bookID);
  [x] session.cart.addItem(bookID) FI
[x] RAISE {NoSuchSession} FI

APROC ChangeQuantity(sID, bookID, newQuantity : Int) RAISES {NotFound} =
IF sID IN sessions.dom() =>
  VAR session := sessions(sID) |
  IF session.loggedIn =>
    DatabaseLayer.setShoppingCartQuantity(session.uID, bookID,
    newQuantity);
    [x] session.cart.setQuantity(bookID, newQuantity); FI
[x] RAISE {NoSuchSession} FI

PROC GetCart(sID) -> SEQ (BookID, Int) RAISES {NoSuchSession} =
IF sID IN sessions.dom(); =>
  VAR session := sessions(sID),
       rCart : SEQ (BookID, Int) := {},
        cart : ShoppingCart |
  IF session.loggedIn =>
    cart := DatabaseLayer.getShoppingCartItems(session.uID);
  [x]
    cart := session.cart.getItems()
  FI
  DO cart.size() > 0 =>
  VAR item : BookID | item IN cart |
     IF session.loggedIn =>
       rCart + := {(item, DatabaseLayer.getShoppingCartQuantity
    (session.uID, item))};
       [x] rCart +:= {(item, session.cart.getQuantity(item))}; FI
           cart - := {item}; OD
        RET rCart;
[x] RAISE {NoSuchSession} FI
```

```
%==========Order Actions==================

PROC CreateOrder(sID, billingAddress : Address, shippingAddress : Address,
                 paymentInfo : PaymentInfo, shippingMethod) RAISES
    {NoSuchSession}, {NotLoggedIn}, {EmptyCart} -> OrderID =
IF sID IN sessions.dom() =>
     VAR session := sessions(sID) |
     IF session.loggedIn =>
        VAR cart := DatabaseLayer.getShoppingCartItems(userID),
            orderItems : SEQ (BookID, Int, Price) := {} |
        <<DO cart.size() > 0 =>
                orderItems + := {(cart.head(),
    DatabaseLayer.getShoppingCartQuantity(userID, cart.head()),
                                    DatabaseLayer.getPrice(cart.head
    ())))}; OD
          RET DatabaseLayer.newOrder(session.uID, billingAddress,
    shippingAddress, paymentInfo, shippingMethod,
                                getTax(sID, shippingAddress), getShipping
    (sId,shippingMethod, shippingAddress),
                                orderItems);>>
        [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

FUNC getShipping(sID, shippingMethod, shippingAddress : Address) RAISES
    {EmptyCart}, {NoSuchSession}
                        -> Price =
   VAR session := sessions(sID)
         items : SEQ BookID
   shippingCost : Price |
   IF session.loggedIn =>
     items := DatabaseLayer.getShoppingCartItems(session.uID);
   [x] items := session.cart.getItems(); FI
   IF items.size > 0 =>
     RET getShippingCost(items, shippingMethod, shippingAddress);
   [x] RAISE {EmptyCart}; FI
[x] RAISE {NoSuchSession} FI

FUNC getChangeShipping(sID, orderID, shippingMethod, shippingAddress :
    Address)
   RAISES {NoSuchSession}, {NotValidOrder}, {NotActiveOrder},
    {NotLoggedIn} -> Price =
IF sID IN sessions.dom() =>
   VAR session := sessions(sID) |
   IF session.loggedIn =>
   VAR orders := DatabaseLayer.getOrderHistory(session.uID) |
   IF orderID IN orders =>
     VAR activeItems := DatabaseLayer.getOrderItems(orderID, "ACTIVE") |
     IF activeItems.size > 0 =>
        RET getShippingCost(items, shippingMethod, shippingAddress);
        [x] RAISE {NotActiveOrder} ; FI
     [x] RAISE {NotValidOrder}; FI
   [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession}

FUNC getShippingCost(books : SEQ BookID, shippingMethod, shippingAddress :
    Address) RAISES {NotEnoughStock}=
```

```
   VAR bookLocs : SEQ (BookID, SET Locations) := {} |
   DO books.size() > 0 =>
     VAR book : BookID | book in books |
          bookLocs + := {(book, DatabaseLayer.getLocations(book))};
          books - := {book}; OD
   RET CalculateShipping(bookLocs, shippingMethod, shippingAddress);
   %This should choose a location for each book to ship from then
   %calculate the shipping between that location and the shipping address
   %then return the total shipping.  It should also raise an error if
    there is not
   %enough stock of all the books.

FUNC getTax(sID, shippingAddress : Address) RAISES {NoSuchSession},
    {EmptyCart} -> Price =
IF sID IN sessions.dom() =>
   VAR session := sessions(sID)
    items := session.cart.getItems() |
   IF items.size = 0 => RAISE {EmptyCart} FI
   RET getTaxCost(items, shippingAddress);
[x] RAISE {NoSuchSession}

FUNC getChangeTax(sID, orderID, shippingAddress)
   RAISES {NoSuchSession}, {NotValidOrder},{NotActiveOrder},{NotLoggedIn}
    -> Price =
   VAR session := session(sID) |
   IF session.loggedIn =>
   VAR orders := DatabaseLayer.getOrderHistory(session.uID) |
   IF orderID IN orders =>
     VAR activeItems := DatabaseLayer.getOrderItems(orderID, "ACTIVE") |
     IF activeItems.size > 0 =>
        RET getTaxCost(activeItems, shippingAddress);
        [x] RAISE {NotActiveOrder} ; FI
     [x] RAISE {NotValidOrder}; FI
   [x] RAISE {NotLoggedIn}; FI
[x] RAISE  {NoSuchSession} FI

FUNC getTaxCost(books : SEQ BookID, shippingAddress : Address) -> Price =
   VAR taxItems : SEQ Price := {};
    taxRate := getTaxRate(shippingAddress) |
      % This should find the rate for that region of the country
   DO books.size() > 0 =>
   VAR book : BookID | book IN books |
     taxItems + := {taxRate * DatabaseLayer.getPrice(book)};
     books - := {book}; OD
   RET (+: taxItems);

APROC GetShippingAddress(sID, orderID)
   RAISES {NoSuchSession}, {NotValidOrder}, {NotLoggedIn} =
IF sID IN sessions.dom() =>
   VAR session := sessions(sID) |
   IF session.loggedIn =>
    VAR orders := DatabaseLayer.getOrderHistory(session.uID) |
    IF orderID IN orders =>
                  RET DatabaseLayer.getShippingAddress(orderID);
    [x] RAISE {NotValidOrder}; FI
   [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

PROC ChangeOrderShippingAddress(sID, orderID, newShippingAddress :
    Address)
   RAISES {NoSuchSession}, {NotFound}, {NotActiveOrder}, {NotValidOrder},
    {NotLoggedIn}, {Shipped} =
IF sID IN sessions.dom() =>
   VAR session := sessions(sID)
    shippingMethod := DatabaseLayer.getOrderShippingMethod(orderID)
    newCost := getChangeShipping(sID, orderID, shippingMethod,
    newShippingAddress)
    newTax := getChangeTax(sID, orderID, shippingAddress) |
   <<DatabaseLayer.editShippingAddress(orderID, newShippingAddress);
        DatabaseLayer.editShippingCost(orderID, newCost);
        DatabaseLayer.editTax(orderID, newTax);>>
RAISE {NoSuchSession} FI

APROC ChangeOrderBillingAddress(sID, orderID, newBillingAddress : Address)
   RAISES {NoSuchSession}, {NotValidOrder}, {NotActiveOrder},
    {NotLoggedIn} =
IF sID IN sessions.dom() =>
   VAR session := sessions(sID) |
   IF session.loggedIn =>
    VAR orders := DatabaseLayer.getOrderHistory(session.uID) |
    IF orderID IN orders =>
       VAR activeItems := DatabaseLayer.getOrderItems(orderID, "ACTIVE") |
       IF activeItems.size > 0 =>
                  DatabaseLayer.editBillingAddress(orderID,
    newBillingAddress);
          [x] RAISE {NotActiveOrder} ; FI
    [x] RAISE {NotValidOrder}; FI
   [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

APROC ChangePaymentInfo(sID, orderID, newPaymentInfo : PaymentInfo)
   RAISES {NoSuchSession}, {NotValidOrder}, {NotActiveOrder} ,
    {NotLoggedIn} =
IF sID IN sessions.dom() =>
   VAR session := sessions(sID) |
   IF session.loggedIn =>
   VAR orders := DatabaseLayer.getOrderHistory(session.uID) |
   IF orderID IN orders =>
     VAR activeItems := DatabaseLayer.getOrderItems(orderID, "ACTIVE") |
     IF activeItems.size > 0 =>
        DatabaseLayer.editPaymentInfo(orderID, newPaymentInfo);
        [x] RAISE {NotActiveOrder} ; FI
     [x] RAISE {NotValidOrder}; FI
   [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

APROC CancelItem(sID, orderID, bookID) RAISES {NoSuchSession},
    {NotValidOrder},{NotActiveItem},{NotLoggedIn} =
IF sID IN sessions.dom() =>
     VAR session := sessions(sID) |
     IF session.loggedIn =>
       VAR orders := DatabaseLayer.getOrderHistory(session.uID) |
       IF orderID IN orders =>
```

```
         VAR activeItems := DatabaseLayer.getOrderItems(orderID, "ACTIVE")       sessions := {* -> };
    |
         IF bookID IN activeItems.size =>
              DatabaseLayer.cancelItem(orderID, bookID);
            [x] RAISE {NotActiveItem} ; FI
         [x] RAISE {NotValidOrder}; FI
       [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

FUNC getCompletedOrders(sID) -> SEQ (OrderID, OrderInfo) RAISES
    {NoSuchSession}, {NotLoggedIn} =
IF sID IN sessions.dom() =>
     VAR session := sessions(sID) |
     IF session.loggedIn =>
        VAR orders := DatabaseLayer.getOrderHistory(session.uID)
            orderLog : SEQ (OrderID, OrderInfo) := {} |
        DO orders.size > 0 =>
           VAR order : OrderID | order IN orders |
           orderLog + := {(order, DatabaseLayer.getOrderInfo(order))};
           orders - := {order}; OD
        RET orderLog;
     [x] RAISE {NotLoggedIn}; FI
[x] RAISE {NoSuchSession} FI

%============Search and Display Actions=========

FUNC Search(sID, fieldName : String, fieldValue : String, ordering,
    fieldsToReturn)
                                                          -> SEQ (BookID,
    SEQ String) RAISES {NoSuchSession} =
IF sID IN sessions.dom() =>
     RET DatabaseLayer.searchBooks(fieldName, fieldValue, ordering,
    fieldsToReturn)
[x] RAISE {NoSuchSession} FI

FUNC Detail(sID, bookID) -> (BookID, Book, Price) RAISES {NoSuchSession} =
IF sID IN sessions.dom() =>
    VAR book := DatabaseLayer.getBook(bookID)
         price := DatabaseLayer.getPrice(bookID) |
     RET (bookID, book, price);
[x] RAISE {NoSuchSession} FI

%============State Actions====================
PROC NewSID() -> SID =
     VAR sID | !(sID IN sessions.dom()) |
     RET sID;

PROC Flush()
     %Implementation will have a housecleaning routine that flushes
    sessions that have expired
     VAR sID | sID IN sessions.dom() |
     sessions := sessions{sID -> };


PROC Crash() =
     %This layer lacks persistent state and will lose all sessions in the
    event of a crash
```

9 Appendix C: Presentation Layer Specification Code

```
MODULE BookDetailPage

  PROC load(userID: UserID + Null, sessionID : SID + Null) -> (BookID,
    Book, Price) RAISES {NotFound}, {NoSuchSession} =
    IF sessionID = Null =>
       sessionID := LogicLayer.NewSession(userID) FI
    RET LogicLayer.Detail(sessionID, bookID);

  PROC goToSearchPage(userID: UserID + Null, sessionID : SID + Null,
    searchField: String, searchValue: String) RAISES {NoSuchSession},
    {NotFound} -> SEQ (BookID, SEQ String)  =
    RET SearchResultsPage.load(userID, sessionID, searchField,
    searchValue);

  PROC goToShoppingCartPage(userID: UserID + Null, sessionID : SID + Null)
    RAISES {NoSuchSession}, {NotFound} -> SEQ (BookID, Int) =
    RET ShoppingCartPage.load(userID, sessionID);

  PROC addToShoppingCart(userID: UserID + Null, sessionID : SID + Null,
    bookID)  RAISES {NoSuchSession}, {NotFound} -> SEQ (BookID, Int) =
    RET ShoppingCartPage.add(userID, sessionID, bookID);

  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
    RET HomePage.load(userID, sessionID);

END BookDetailPage
================================================================================
    ========================================
MODULE ChangeAccountPage

 PROC load(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NotFound}, {NoSuchSession} -> (String, String, String, String) =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID)
    [*] VAR userInfo := LogicLayer.getUserInfo(sessionID);
            listOfUserInfo := (userInfo.firstName, userInfo.lastName,
    userInfo.middleInitial, userInfo.email)
        RET listOfUserInfo;

  PROC changeUserInfoAndGoToHomePage(userID: UserID + Null, sessionID :
    SID + Null,
                                     firstName: String, middleInit: Char,
    lastName: String, email: String)
                     RAISES  {NoSuchSession}, {NotLoggedIn}, {NotFound}
    -> String =
     LogicLayer.ChangeUserInfo(sessionID, firstName, middleInit, lastName,
    email);
     RET HomePage.load(userID, sessionID);

  PROC changePasswordAndGoToHomePage(userID: UserID + Null, sessionID :
    SID + Null, oldPass: String, newPass: String) RAISES {NoSuchSession},
    {NotFound}, {NotLoggedIn}, {WrongPassword} -> String =
     LogicLayer.ChangePassword(sessionID, oldPass, newPass);
     RET HomePage.load(userID, sessionID);

  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
     RET HomePage.load(userID, sessionID);

END ChangeAccountPage
================================================================================
    ========================================
MODULE ChangeOrderPage

  PROC load(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotLoggedIn} -> SEQ(OrderID, OrderInfo) =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    RET LogicLayer.getCompletedOrders(sessionID);

  PROC changeShippingMethod(userID: UserID + Null, sessionID : SID + Null,
    orderID: Int, shippingMethod) RAISES {NoSuchSession}, {Not
    ValidOrder}, {NotActiveOrder}, {NotLoggedIn} -> (OrderID,
    ShippingMethod, Price) =
    VAR newShipping := LogicLayer.getChangeShipping(sessionID, orderID,
    shippingMethod, LogicLayer.getShippingAddress(sID, orderID)) |
    RET load(orderID, shippingMethod, newShipping);

  PROC changeOrderShippingAddress(userID: UserID + Null, sessionID : SID +
    Null, orderID: Int, address: Address) RAISES {NoSuchSession},
    {NotFound}, {NotActiveOrder}, {NotLoggedIn} ->  SEQ(OrderID,
    OrderInfo)=
       LogicLayer.changeOrderShippingAddress(userID, sessionID, orderID,
    address);
       RET load(userID, sessionID)

  PROC changeOrderBillingAddress(userID: UserID + Null, sessionID : SID +
    Null, orderID: Int, address: Address) RAISES {NoSuchSession},
    {NotValidOrder}, {NotActiveOrder}, {NotLoggedIn} -> SEQ(OrderID,
    OrderInfo) =
       LogicLayer.changeOrderBillingAddress(userID, sessionID, orderID,
    address);
       RET load(userID, sessionID);

  PROC changePaymentInfo(userID: UserID + Null, sessionID : SID + Null,
    orderID: Int, newPaymentInfo: PaymentInfo) RAISES {NoSuchSession},
    {NotLoggedIn}, {NotValidOrder}, {NotActiveOrder} -> SEQ(OrderID,
    OrderInfo)=
       LogicLayer.changePaymentInfo(userID, sessionID, orderID,
    newPaymentInfo);
       RET load(userID, sessionID);

  PROC cancelItem(userID: UserID + Null, sessionID : SID + Null, orderID:
    Int, bookID: Int) RAISES {NoSuchSession}, {NotLoggedIn},
    {NotValidOrder}, {NotActiveOrder} -> SEQ(OrderID, OrderInfo)=
       LogicLayer.cancelItem(userID, sessionID, orderID, bookID);
       RET load (userID, sessionID);
```

```
  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
       RET HomePage.load(userID, sessionID);

END ChangeOrderPage
================================================================================
    ========================================
MODULE CheckOutSignInPage

  PROC load(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    RET LogicLayer.getUserName(sessionID);

  PROC goToShippingPage(userID: UserID + Null, sessionID : SID + Null,
    username: String, password: String) RAISES {NoSuchSession},
    {NotFound}, {WrongPassword} -> Int =
    LogicLayer.SignIn(sessionID, username, password);
    RET ShippingPage.load(userID, sessionID);

  PROC goToCreateAccountPage(userID: UserID + Null, sessionID : SID +
    Null) RAISES
    RET CreateAccountPage.load(userID, sessionID);

End CheckOutSignInPage
================================================================================
    ========================================
MODULE ConfirmOrderPage

  PROC load(userID: UserID + Null, sessionID : SID + Null, paymentInfo:
    PaymentInfo, shippingMethod: String, address: Address) -> (Price,
    Price) RAISES {NoSuchSession}, {NotFound}, {EmptyCart},
    {NotEnoughStock} -> (Price, Price, PaymentInfor, ShippingMethod) =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    VAR shippingPrice := LogicLayer.getShipping(sessionID, shippingMethod,
    address),
        tax := LogicLayer.getTax(sessionID, address) |
    RET (shippingPrice, tax, paymentInfo, shippingMethod);

  APROC goToSuccessPage(userID: UserID + Null, sessionID : SID + Null,
    paymentInfo: PaymentInfo, shippingMethod: String, address: Address)
    RAISES {NoSuchSession}, {NotLoggedIn}, {EmptyCart} -> OrderID =
    VAR order := LogicLayer.CreateOrder(sessionID, address, address,
    paymentInfo, shippingMethod) |
    RET SuccessPage.load(userID, sessionID, order);

  APROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String=
    RET HomePage.load(userID, sessionID);

END ConfirmOrderPage

================================================================================
    ========================================
MODULE CreateAccountPage

  PROC load(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI

  PROC createAccountAndGoToHomePage(userID: UserID + Null, sessionID : SID
    + Null, name: String, userName1: String, userName2: String, password1:
    String, password2: String) RAISES {NotFound}, {NoSuchSession},
    {UserExistsAlready} -> (String, String, String, String) =
   IF userName1 = userName2 /\ password1 = password2 =>
     LogicLayer.NewUser(sessionID, userName1, password1);
     RET HomePage.load(userID, sessionID)
   [*] RET CreateAccountPage.load(sessionID) FI

  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
   RET HomePage.load(userID, sessionID);

  PROC goToShoppingCartPage(userID: UserID + Null, sessionID : SID + Null)
    RAISES {NoSuchSession}, {NotFound} -> SEQ (BookID, Int)=
   RET ShoppingCartPage.load(userID, sessionID)

END CreateAccountPage

================================================================================
    ========================================
MODULE HomePage

  PROC load(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    VAR name := LogicLayer.getName(sessionID)
              EXCEPT NoID => name := "";
    RET name;

  PROC goToSearchResultsPage(userID: UserID + Null, sessionID: SID + Null,
    searchField: String, searchValue: String) RAISES {NoSuchSession},
    {NotFound} -> SEQ (BookID, SEQ String) =
    RET SearchResultsPage.load(userID, sessionID, searchField,
    searchValue);

  PROC goToBookDetailPage(userID: UserID + Null, sessionID: SID + Null,
    bookID: Int) RAISES {NotFound}, {NoSuchSession} -> (BookID, Book,
    Price) =
    RET BookDetailPage.load(userID, sessionID, bookID);

  PROC goToShoppingCartPage(userID: UserID + Null, sessionID: SID + Null)
    RAISES {NoSuchSession}, {NotFound} -> SEQ (BookID, Int)  =
    RET ShoppingCartPage.load(userID, sessionID);

END HomePage

================================================================================
    ========================================
MODULE PaymentPage =
```

```
  PROC load(sessionID: SID, shippingMethod: String, address: Address)
    RAISES {NotFound} -> (String, Address) =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    RET (shippingMethod, address)

  PROC goToConfirmOrderPage(sessionID: SID, paymentInfo: PaymentInfo,
    shippingMethod: String, address: Address) RAISES {NoSuchSession},
    {NotFound}, {EmptyCart}, {NotEnoughStock} -> (Price, Price,
    PaymentInfor, ShippingMethod)
    RET ConfirmOrderPage.load(sessionID, paymentInfo, shippingMethod,
    address);

END PaymentPage

================================================================================
    ========================================
MODULE SearchResultsPage

  PROC load(userID: UserID + Null, sessionID : SID + Null,
            searchField: String, searchValue: String) RAISES
    {NoSuchSession}, {NotFound} -> SEQ (BookID, SEQ String) =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    VAR searchResults := LogicLayer.Search(sessionID, searchField,
    searchValue, ASCENDING, ("title", "price", "shippingSpeed",
    "edition"))
    RET searchResults;

  PROC goToShoppingCartPage(userID: UserID + Null, sessionID : SID + Null)
    RAISES {NoSuchSession}, {NotFound} -> SEQ (BookID, Int)=
    RET ShoppingCartPage.load(userID, sessionID);

  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
    RET HomePage.load(userID, sessionID);

END SearchResultsPage
================================================================================
    ========================================
Module ShippingDetailsPage

  PROC load(userID: UserID + Null, sessionID : SID + Null, address:
    Address) RAISES {NoSuchSession}, {NotFound} -> (SEQ (BookID, Int) ,
    Address) =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    VAR shoppingCart := LogicLayer.GetCart(sessionID) |
    RET (shoppingCart, address);

  PROC goToPaymentPage(userID: UserID + Null, sessionID : SID + Null,
    shippingMethod: String, address: Address) RAISES {NotFound} ->
    (String, Address) =
    RET PaymentPage.load(userID, sessionID, shippingMethod, address);

End ShippingDetailsPage
```

```
================================================================================
    ========================================
MODULE ShippingPage

  PROC load(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NotFound} -> Int =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    RET sessionID

  PROC goToShippingDetailsPage(userID: UserID + Null, sessionID : SID +
    Null, address: Address) RAISES {NoSuchSession}, {NotFound} -> (SEQ
    (BookID, Int) , Address) =
    RET ShippingDetailsPage.load(userID, sessionID, address);

END ShippingPage

================================================================================
    ========================================
MODULE ShoppingCartPage

  PROC load(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> SEQ (BookID, Int) =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    VAR shoppingCart := LogicLayer.getCart(sessionID) |
    RET shoppingCart;

  PROC addToCart(userID: UserID + Null, sessionID : SID + Null, bookID)
    RAISES {NoSuchSession}, {NotFound} -> SEQ (BookID, Int) =
    LogicLayer.addToCart(bookID);
    RET load(userID, sessionID);

  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
    RET HomePage.load(userID, sessionID);

  PROC goToSearchPage(userID: UserID + Null, sessionID : SID + Null,
    searchField: String, searchValue: String) RAISES {NoSuchSession},
    {NotFound} -> SEQ (BookID, SEQ String) =
    RET SearchResultsPage.load(userID, searchField, searchValue);

  PROC goToBookDetailPage(userID: UserID + Null, sessionID : SID + Null,
    bookID: Int) RAISES {NotFound}, {NoSuchSession} -> (BookID, Book,
    Price) =
    RET BookDetailPage.load(userID, sessionID, bookID);

  PROC checkOutSignInPage(userID: UserID + Null, sessionID : SID + Null)
    RAISES {NoSuchSession}, {NotFound} -> String =
    RET CheckOutSignInPage.load(userID, sessionID);

END ShoppingCartPage
================================================================================
    ========================================
MODULE SignInPage
```

```
  PROC load(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    RET LogicLayer.getName(sessionID);

  PROC signInAndGoToHomePage(userID: UserID + Null, sessionID : SID +
    Null, username: String, password: String) RAISES {NoSuchSession},
    {NotFound}, {WrongPassword} -> String =
    LogicLayer.signIn(sessionID, username, password);
    RET HomePage.load(userID, sessionID);

  PROC signInAndGoToChangeAccountPage(userID: UserID + Null, sessionID :
    SID + Null, username: String, password: String) RAISES {NotFound},
    {NoSuchSession}, {WrongPassword} -> (String, String, String, String) =
    LogicLayer.signIn(sessionID, username, password);
    RET ChangeAccountPage.load(userID, sessionID);

  PROC signInAndGoToChangeOrderPage(userID: UserID + Null, sessionID : SID
    + Null, username: String, password: String) RAISES {NoSuchSession},
    {NotLoggedIn}, {WrongPassword} -> SEQ(OrderID, OrderInfo) =
    LogicLayer.signIn(sessionID, username, password);
    RET ChangeOrderPage.load(userID, sessionID);

  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
    RET HomePage.load(userID, sessionID);

  PROC goToShoppingCartPage(userID: UserID + Null, sessionID : SID + Null)
    RAISES {NoSuchSession}, {NotFound} -> SEQ (BookID, Int) =
    RET ShoppingCartPage.load(userID, sessionID);

  PROC goToCreateAccountPage(userID: UserID + Null, sessionID : SID +
    Null) RAISES {NoSuchSession}, {NotFound} =
    RET CreateAccountPage.load(userID, sessionID);

END SignInPage

================================================================================
    ========================================
MODULE SuccessPage

  PROC Load(sessionID: SID, order: OrderID) RAISES {NotFound} -> OrderID =
    IF sessionID = Null =>
       sessionID := LogicLayer.newSession(userID) FI
    RET order;

  PROC goToHomePage(userID: UserID + Null, sessionID : SID + Null) RAISES
    {NoSuchSession}, {NotFound} -> String =
    RET HomePage.load(userID, sessionID);

  PROC goToChangeOrderPage(userID: UserID + Null, sessionID : SID + Null)
    RAISES {NoSuchSession}, {NotLoggedIn} -> SEQ(OrderID, OrderInfo) =
    RET ChangeOrderPage.load(userID, sessionID);

END SuccessPage
```