Constructing Enterprise Applications 4

Learning Goals

After completing the chapter, you will be able to:

- Outline construction readiness.
- Create software construction map.
- Develop application framework components.
- Develop application components for corresponding layers.
- Perform code review and static code analysis.
- Formulate build process.
- Perform unit testing.
- Perform dynamic code analysis.

The construction of an enterprise application involves translating the design into code components to build an application, and fulfill the business perspective by leveraging architecture, development techniques, modeling, frameworks, tools and technologies. The translation of design to code typically involves the following activities:

- Achieving construction readiness
- Constructing solution layers—application framework and application components
- Code review and static code analysis
- Compilation and creating deployable packages
- Unit testing
- Code profiling and code coverage analysis

In general, the bigger picture for constructing an enterprise application involves sharing the common practices, procedures, processes and guidelines, irrespective of the technologies and tools used. In this chapter, however, we will be primarily using Java Enterprise Edition (Java EE), and related frameworks, tools and technologies to illustrate the principles, practices and paradigms of construction. Java EE provides the platform to build distributed and multi-tier enterprise applications.

We have explored technical architecture layers and its design components in the previous chapter. Let us now look at the key activities required to start the construction phase, followed by creation, review, verification and analysis of code components, as envisaged by the architecture and design.

4.1 Construction Readiness

The development blueprint of an enterprise application is comprehensive after completion of requirements engineering and design phase in a particular iteration. A few of the activities that are necessary before getting into the construction activities are as follows.

4.1.1 Defining a Construction Plan

A construction plan will consist of the sequence and schedule of tasks required to be carried out in the construction phase, their interdependencies, technical risks, and resources required to implement the task. The primary objective of the plan is to complete the construction phase in the shortest time possible, while not compromising on functionality and quality requirements. To achieve this, tasks are parallelized as much as possible. The ability to parallelize is limited due to interdependencies among tasks, in which case they need to be sequenced appropriately. Further, resources and schedule conflicts also need to be taken into account. There are a lot of activities that can be done in parallel to coding such as test case preparation, setting up the test environments, planning review activities, planning version control, release planning, etc. A setup for continuous integration is also planned to facilitate ongoing integration of code as and when it is developed.

4.1.2 Defining a Package Structure

The code units that are created during construction have to be structured in a meaningful and logical way. This can be based on considerations such as loose coupling, tight cohesion, reusability, use of third-party libraries, etc. The package structure reflects the physical organization of the units of code. It provides a clear picture to the developer about the location of classes they develop relative to rest of the code. The technology and tools used to raise an enterprise application may also influence the package structure.

Package structures should follow mutually exclusive and collectively exhaustive (MECE) principle, while grouping different application components. Each node of the package structure should have a clear and meaningful name to help in better organization and management of the components.

The package structure followed in LoMS is shown in Figure 4-1. We will further expand the package structures corresponding to the layers in the following sections.

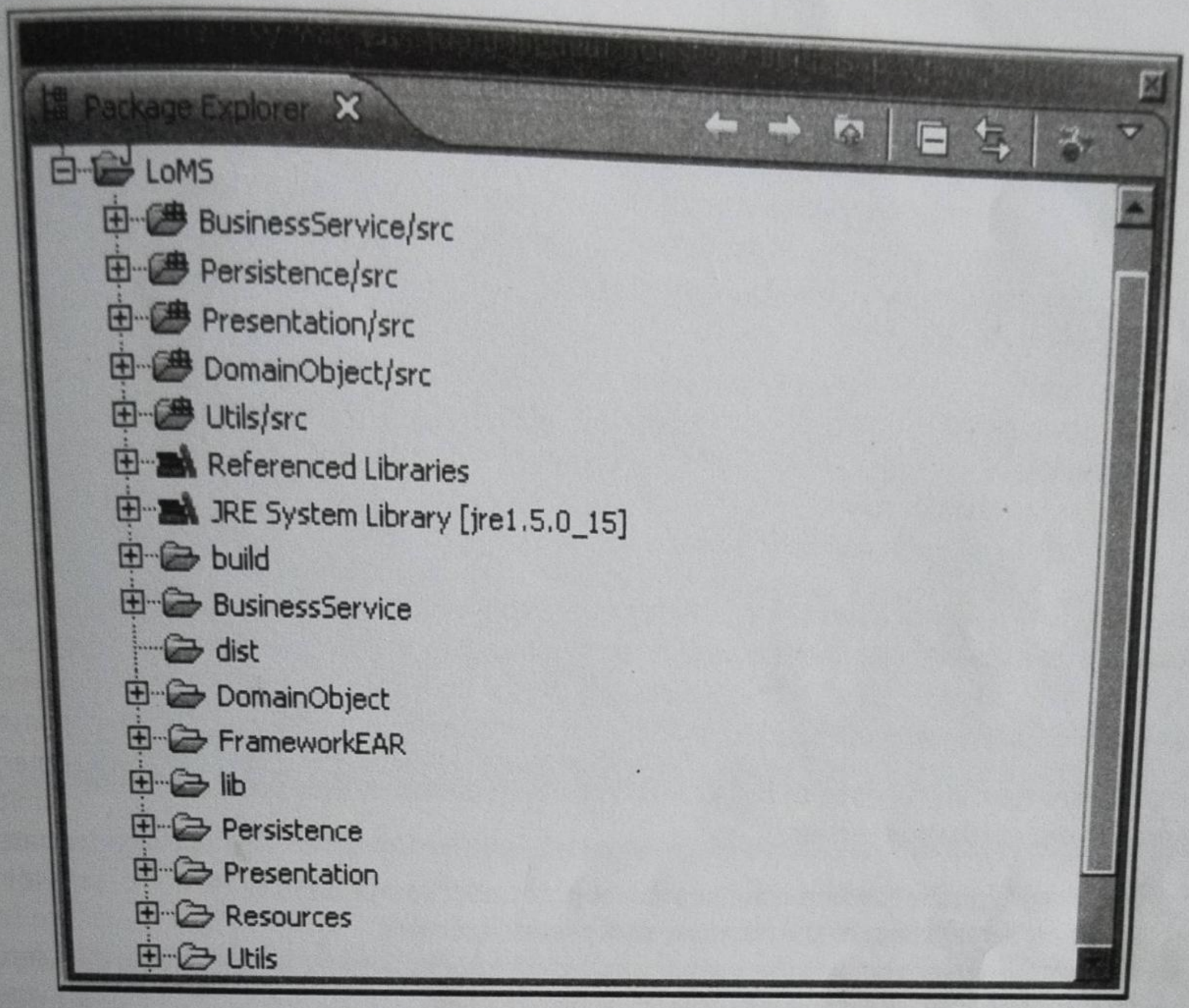


Figure 4-1 LoMS package structure.

4.1.3 Setting Up a Configuration Management Plan

The evolution of artifacts through several versions has to be tracked in order to ensure the integrity and consistency of the artifacts. This process is referred to as *configuration management*, and the artifacts that are managed are referred to as *configuration objects*. This is also required for controlled change management. For example, changes may occur due to a requirement change, or an artifact may have to be rolled back to a previous version. The configuration management plan provides the basis for managing the configuration objects in a consistent and meaningful manner by specifying the tool to be used for the purpose, defining a repository structure, which is aligned to the package structure defined for the project, and specifying appropriate access rights to parts of the repository for the different users.[1]

4.1.4 Setting Up a Development Environment

A development environment is the construction ecosystem that facilitates developers to develop, build, review, verify and analyze code components of an application under development. This environment typically comprises of infrastructure elements (hardware, operating systems and servers), processes

[1] For more information on configuration management plan, you may refer to IEEE Std. 828-1990.

and construction tools, which need to be installed, initialized and configured in order to start the construction. The following is the list of the typical elements that are set up as part of a development environment.

1. Hardware including operating systems
2. Servers such as application, Web, database, directory and portal servers
3. Integrated development environment
4. Third-party libraries
5. Build tools
6. Unit testing tool
7. Profiling tool
8. Static code analysis tool
9. Licenses for all the required software

Assuming that required hardware and operating systems are in place, the following key activities are followed to set up a development environment.

Installation and configuration of servers

Servers that are typically required to be installed in a development environment are file servers, database servers and application servers:

- *File server* typically contains configuration objects under source control.
- *Database server* contains the database with required schema.
- *Application server* contains the various application components. It is typically configured with the required JDBC, JNDI and JMS objects, along with the generic configurations such as object pool size, ports and security settings.

Setting up the integration development environment (IDE)

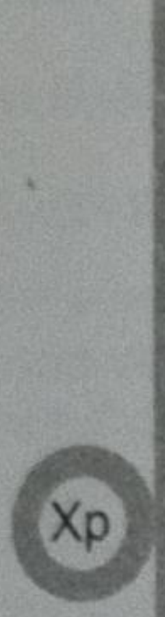
An IDE is configured with relevant libraries, plug-ins, database and server configurations. A project is set up in the IDE as per the package structure that will be used by the construction team. Members of the development team replicate this IDE configuration in their desktop, and may customize it further based on their role and preferences.

Setting up frameworks

An enterprise application is typically built on top of an application framework which, in turn, may consist of several other layer-specific frameworks. These frameworks are typically extended, configured and instantiated to build the application components. Setting up the framework typically involves importing and configuring the libraries of the framework as part of the project in the IDE.

In addition to all of the above, it should be kept in mind that the construction phase is most intensive in terms of number of people involved and the simultaneous activities in progress. This requires a lot of coordination and communication within the team, besides careful planning and meticulous implementation on the part of the project leadership.

4.2 Introduction to Software Construction Map



- A developer gets a designer perspective of the software solution from several design output such as relational models, UML sequence diagrams and class diagrams. To understand the design and translate it into code, a developer may often need some more information on the design.
- The common dilemma of a developer while coding may include the following:
  - How does the overall picture of components look like?
  - What is the relationship between the solution components?
  - How do the components interact and interface within and outside the layer?
  - How to visualize the flow of functionality across various technical architecture layers?
- Based on our experience, we felt a need to represent the design elements in a different form for better understanding of components, their interactions, and to help visualize the overall solution from a developer perspective. This brought out the new form of representation of various components which we christen as *software construction map* (SCM).
- A SCM coupled with existing design models, brings out several benefits to the develoment team such as visualizing the big picture of solution, inter and intra relationships among various components across the layers, and multiple hand offs between the development team.
- Readers will be introduced to the SCM in the following section.

Let us now explore the construction of code components which corresponds to the technical architecture layers.

4.3 Constructing the Solution Layers

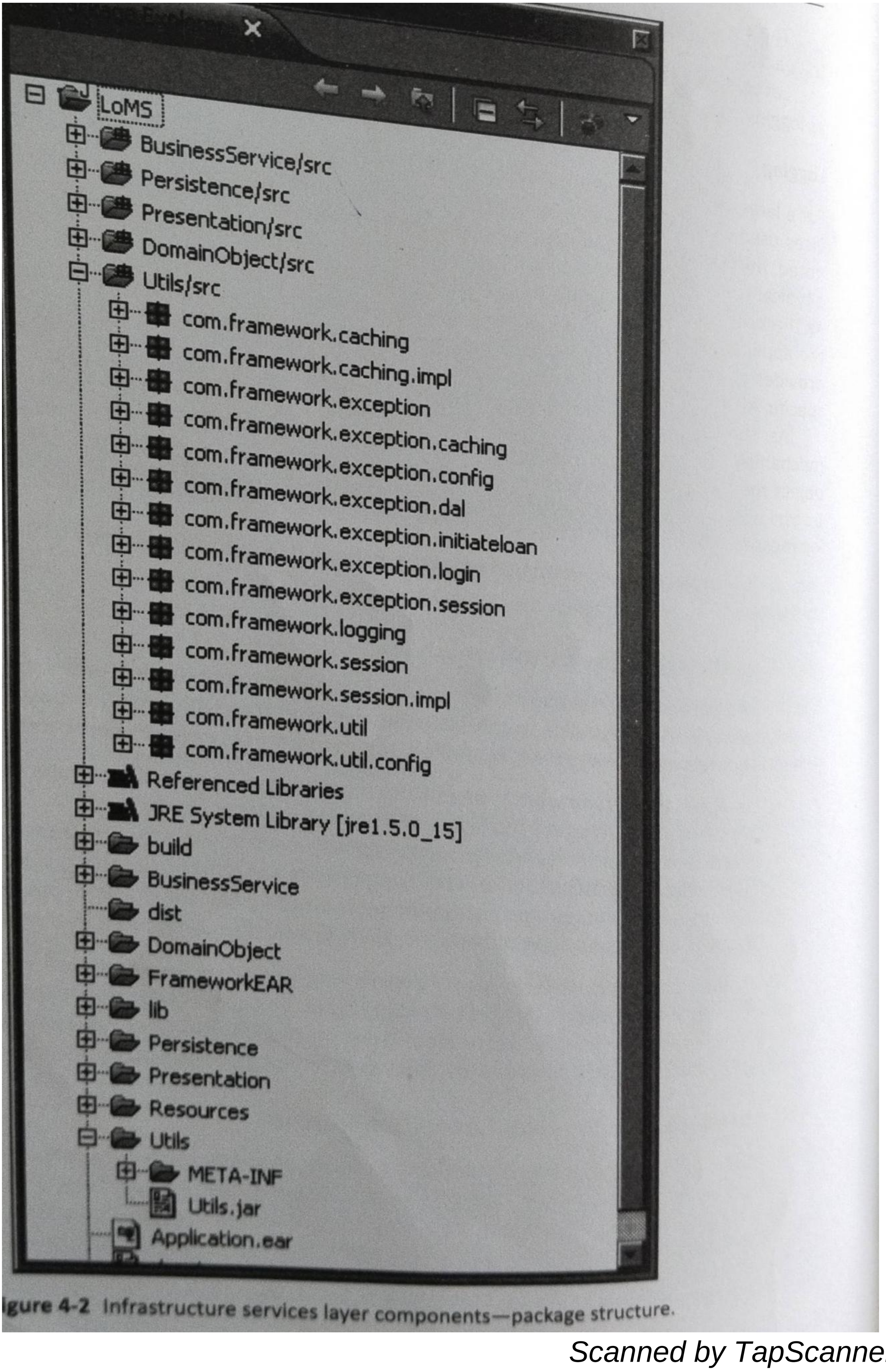
Construction of an enterprise application typically does not start from scratch, even for a greenfield development project. Every organization usually has a reusable assets repository and frameworks to ensure that the wheel is not reinvented every time. Reusable assets repository may have different components as follows:

- An application framework is one of the most important reusable components. It provides the plumbing code across all the layers of an enterprise application.
- A repository of technical components typically has components such as workflow component, rules engine, infrastructure services components and generic reporting components.
- A repository of domain specific components typically has components related to business such as *check credit rating* component and *quote generation* component in the banking domain.

Use cases captured in requirements engineering phase are implemented using these reusable components by either directly using or extending them. These components provide a standard way for the entire development team to construct an enterprise application. This section will take the readers through the construction of technical architecture layers of an enterprise application.

4.3.1 Infrastructure Services Layer Components

As described in Chapter 3, infrastructure services layer of an enterprise application comprises of general purpose components such as logging, session management, security, exception handling, auditing, caching, notification and reporting, which are typically used across all the layers. These components are usually designed and constructed as part of the application framework. A common scenario is that such components are readily available off the shelf, and are plugged into the framework with enhancements and proper configuration as required.



igure 4-2 Infrastructure services layer components—package structure.

The elements of infrastructure services layer in LoMS are grouped in the *Utils* folder, as depicted in the package structure diagram (Figure 4-2).

Let us now explore the construction of some key components of infrastructure services layer such as logging, session management, exception handling and caching.

Logging

For a large distributed application development, logging is an important element of infrastructure services used by almost all the solution layers, as explained in the previous chapter. As you may have noted, third party packages, such as Log4J, are generally used to implement the logging functionality in a typical Java enterprise application, and calls to the package APIs are inserted in the application code. As the best practice to improve the maintainability and flexibility of code, it is important to insulate the application from a specific logging package. For example, in LoMS, the `LoggingManager` class provides an encapsulation of the Log4J package, and helps in decoupling the application from package specific APIs.

As shown in Figure 4-3, the `LoggingManager` class provides a facade for the logging mechanism. It has static methods for logging information into centralized logging store/files. Any object may be passed to `LoggingManager` as long as it implements the `ILoggable` interface. `LoggingManager` instantiates the Logger of Log4J package and invokes the appropriate method to log messages.

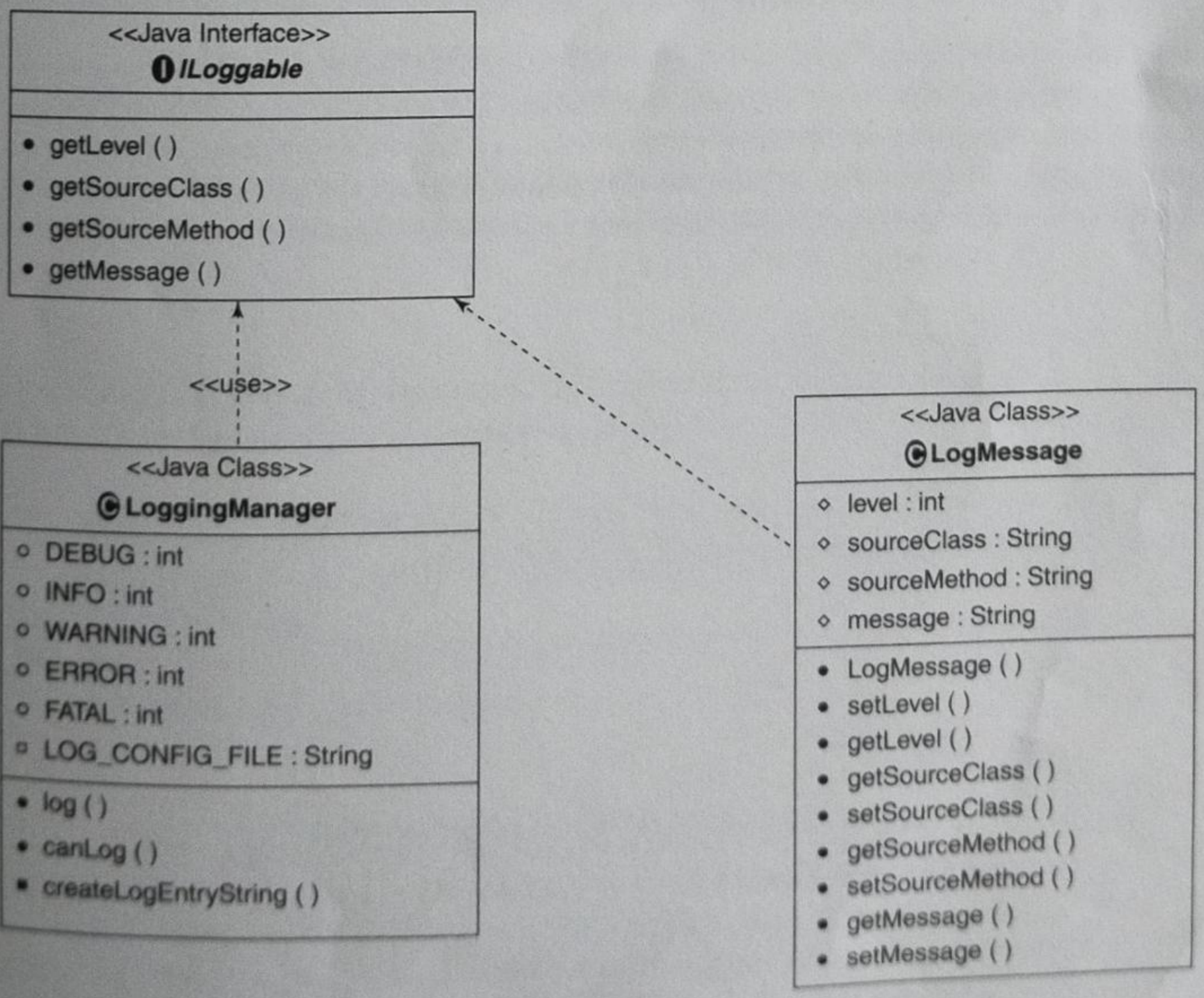


Figure 4-3 Logging framework component.

Code Listing 4.1 depicts the partial implementation of the `LoggingManager`.

```
public class LoggingManager
{
      static public final int DEBUG = 0;
      static public final int INFO = 1;
      static public final int WARNING = 2;
      ----------------------------------------
      /** Name of the Log4j config file **/
      private static final String LOG_CONFIG_FILE = "config/Log4jConfig.xml";
      /**Provides a static initializer for log4J initialization**/
      static
      {
        ClassLoader classLoader =LoggingManager.class.getClassLoader();
        try {
          URL fileUrl = classLoader.getResource(LOG_CONFIG_FILE);
          if (fileUrl != null)
              {
                DOMConfigurator.configure (fileUrl);
              }
          else
              {
                Logger rootLogger =
                Logger.getRootLogger();
                rootLogger.setLevel(Level.OFF);
                System.err.println("Log4j not Initialized : Could not find
                any Log4j Config File.");
              }
          catch(Exception ex) {
            System.err.println("Exception Occcured in Log4j initialization");
          }
        }
      /** Log method for logging the application messages**/
      public static void log(Loggable itemToLog)
      {
        String sourceClass = itemToLog.getSourceClass();
        if(sourceClass == null)
        {
          sourceClass = "";
        }
        String sourceMethod = itemToLog.getSourceMethod();
        if(sourceMethod == null)
        {
          sourceMethod = "";
        }
```

```
String message = itemToLog.getMessage();
if(message == null)
{
  message = "";
}
Logger logger = Logger.getLogger(sourceClass);
String logEntryString = createLogEntryString(sourceClass,
sourceMethod, message);
int loggingLevel = itemToLog.getLevel();
switch (loggingLevel)
{
  case DEBUG : logger.debug(logEntryString);break;
  case INFO : logger.info(logEntryString);break;
  ----------------------------------------------
  ----------------------------------------------
}
}
}
```

Code Listing 4.1 `LoggingManager`—façade to the Logging package API.

`LogMessage` class takes care of setting the log level, source class, source method and logging message. As and when required, as depicted in Code Listing 4.2, enterprise application layers create the `log` messages and call the `log` method of `LoggingManager` to write the statements into a log file. It is always better to check whether the logger is enabled for the specific level before logging the message. In LoMS, the `canLog` method of `LoggingManager` checks whether the level passed as input is enabled to log the message.

```
// Log entry
if(LoggingManger.canLog(LoggineManger.DEBUG, CLASS_NAME))
{
    String methodName = "getCustomerDetails";
    LogMessage logMessage = new LogMessage(LoggingManager.DEBUG,
CLASS_NAME, methodName,
   "Viewing the customer details");
    LoggingManager.log(logMessage);
}
```

Code Listing 4.2 Calling logging operation.

The log file name and path are configured in the `Log4J` configuration file. Loggers can assign different types of log levels. Typical log levels are `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR` and `FATAL`. In LoMS, `DEBUG` log level is used. The logging framework exposes the `log` method to all other application layer components, and internally `LoggingManager` will call the printing methods of a logger instance depending on the log level.

A logging request can be enabled by setting the log level equal to, or higher than, the level of its logger. The typical order of level is defined as `DEBUG < INFO < WARN < ERROR < FATAL`.

Large-scale enterprise applications usually generate thousands of log requests that may indirectly hit the performance of the application. The deployment team has to appropriately set the log level at the time of deployment to control the volume of log generated. Readers should note that the above represents one of the several ways of implementing the logging strategy for an enterprise application. Another approach to implement logging in an enterprise application is to use *aspect-oriented programming* (AOP) to achieve better modularity. AOP is a programming paradigm that enables separation of concerns into *aspects*. An aspect is a feature that cuts across the core features of an application, e.g., logging and error handling.

Exception handling

Exception handling is a mechanism used by an application component to take appropriate corrective or preventive measures, due to occurrence of an exception at runtime of the application. Chapter 3 has presented the readers with various important points that need to be taken care of while designing an exception handling strategy. Exception handling is one of the mandatory infrastructure components that typically gets fabricated as part of the application framework components. Figure 4-4 depicts exception handling as implemented in LoMS. Various categories of exceptions and their interrelationships are also shown in Figure 4-4. The implementation of the exception handling components is available for reference in the LoMS codebase.

Session management

In Chapter 3, you have familiarized with the session management concepts, and approaches, which need to be considered while designing a session management strategy. As mentioned earlier, LoMS uses "in-memory" storage technique for session management, and JBoss caching solution to implement it. In the session management implementation of LoMS, the application creates a unique `SessionID` using session management components after successful authentication of a user. The `SessionID` gets stored in an `HttpSession` object, along with the application state information of that user. Whenever an application component requires information from a session object, it retrieves the `SessionID` from the `HttpSession` object, and passes it to the caching component to further fetch the related session details.

Figure 4-5 depicts the overall session management component, which has been designed and implemented as part of the LoMS application framework. The `Session` class contains the logic for creation of a session, invalidating a session, retrieval of information, and adding and removing the information details from `Session`.

Code Listing 4.3 depicts the `Session` object constructor, which ensures the creation of a new `SessionInfo` object, for every session, which gets stored in cache with `SessionID`.

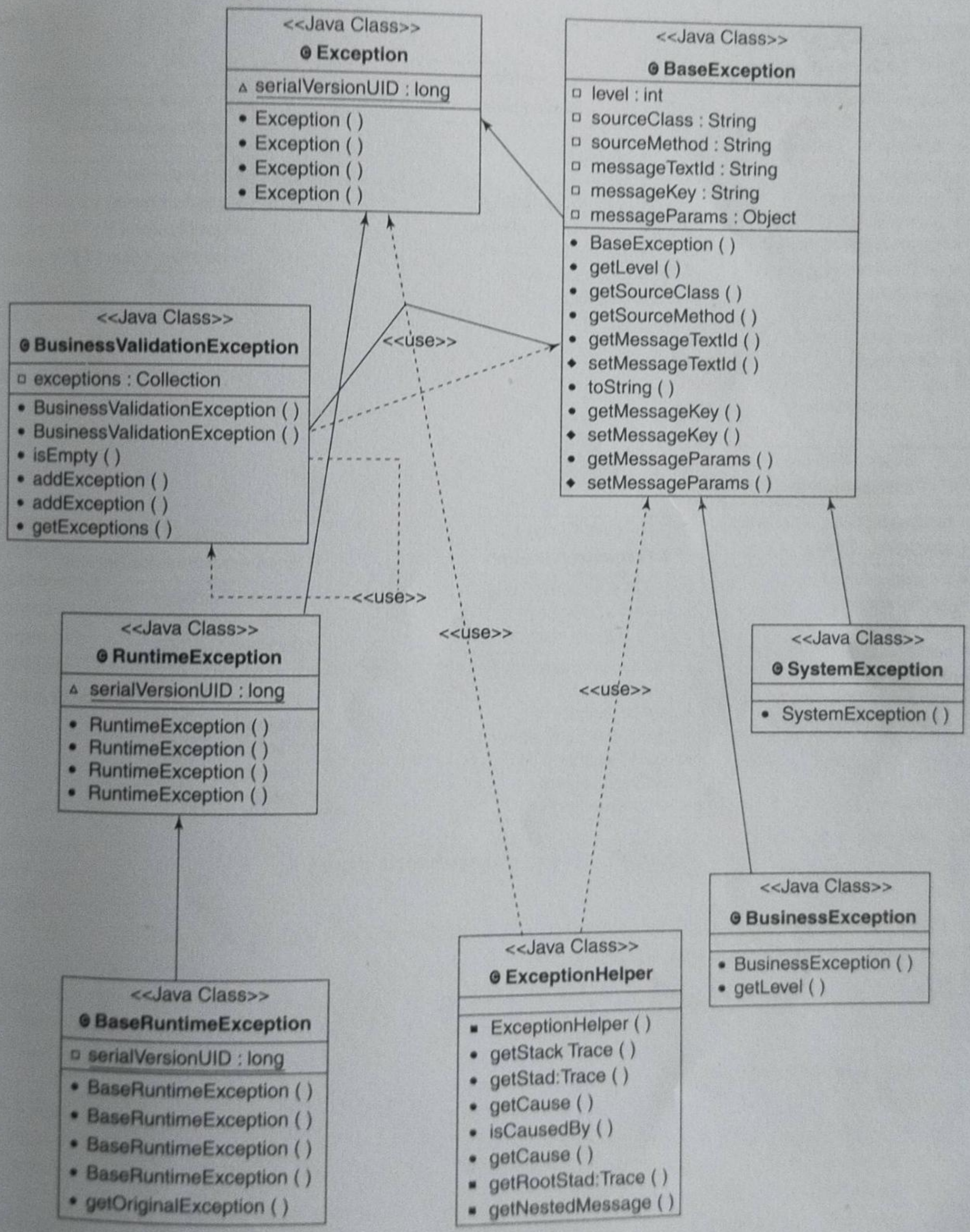


Figure 4-4 Exception handling component.

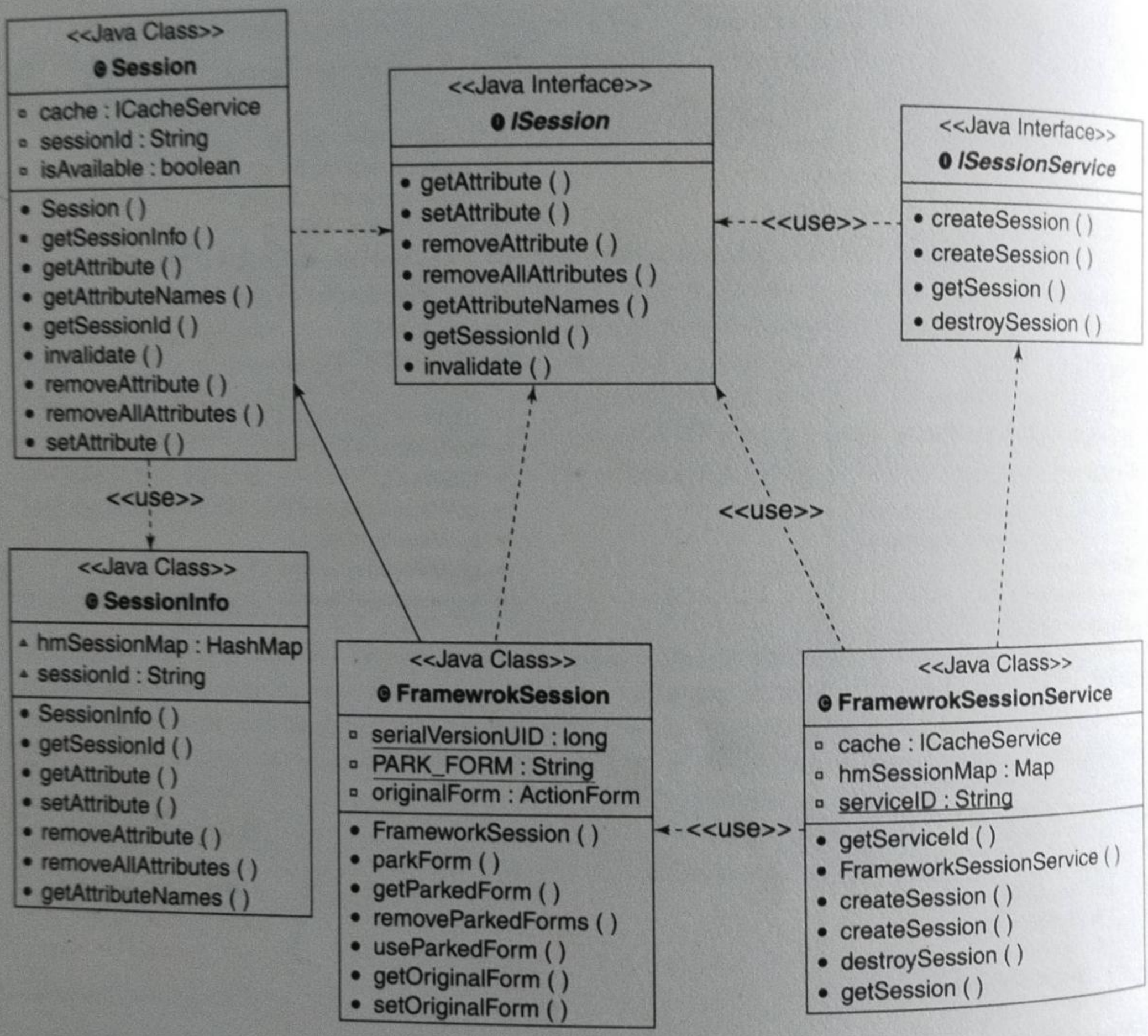


Figure 4-5 Session management component.

```
public Session (String sessionId, ICacheService cache) throws
SessionServiceException {

LogMessage logMessage = new LogMessage(LoggingManager.DEBUG,
CLASS_NAME,"Session","Session() Constructor");

LoggingManager.log(logMessage);

try {
      this.sessionId = sessionId;
      this.cache = cache;
      SessionInfo objSessionInfo = new SessionInfo(sessionId);
      cache. put(sessionId, objSessionInfo);
      isAvailable = true;
     }
catch (Exception ex) {
```

```
throw new SessionServiceException ("Session","Session Constructor","Error
in creating a new session with Id" +ex.getMessage(), null);
    }
}
```

Code Listing 4.3 Creation of `SessionInfo` object.

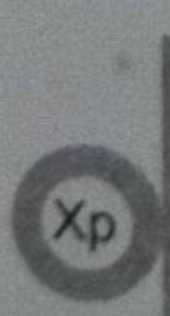
`SessionInfo` constructor creates a `HashMap` object to store session attribute values. `SessionService` class has the functionality for storing all the session objects based on the `SessionID`.

When it comes to the requirement of clustering enterprise applications, it requires handling the http session in a reliable fashion. Session attributes must be serializable, if they are to be used in a clustered environment. In case of large session objects, session data, which are not required to get replicated, can be declared as transient that means that this data will not be replicated across the clustered environment.

Enterprise applications optimally take the advantage of built-in session management features of the application server. Any commercial application server typically has the features, as shown in Table 4-1.

Table 4-1 Session management features

Features	Description
Session affinity	*Session affinity* is a mechanism, where all requests that are part of a session are directed to a particular node of the clustered environment. This helps enhance the application performance by caching the session objects.
Session tracking	*Session tracking* is a mechanism to use cookies for session management. The application server generates a session ID and returns this ID in a cookie to the browser. The browser stores the cookie and includes in it every request, which enables the server to associate the request with the session. The cookie is destroyed in the events such as termination of the session, browser closure, expiry of cookie, or session time out.
In-memory session and persistent session	*In-memory session* and *persistent session* are mechanisms wherein the session attributes are stored in the memory or persisted in a database respectively.



One of the other very important features of session management is managing synchronized access. Consider the case of two servlets changing the same session properly at the same time, or one servlet invalidating the session while another one is still trying to use it; data inconsistencies may occur as a consequence. To avoid such issues, the session component has to synchronize access following the `request.getSession()` call, along with checking the validity of the session.

Caching

Chapter 3 provided a primer on caching and related tools to implement it as a framework component. In LoMS, the caching component is implemented as a tree cache, where the cache is organized as a tree. Each node in the tree is implemented as a map for storing key-value pairs.

As shown in Figure 4-6, `ICacheService` is an interface that exposes the methods to get, put and remove object from a cache. Lazy loading mechanism is used for populating the cache. The entire static data loading is defined in a configuration service. When there is a request for data, the configuration service queries the cache for the object. If the object is not present in the cache, the configuration service accesses the data source to load the data into cache. The configuration service starts retrieving the data from cache for subsequent requests for the same data.

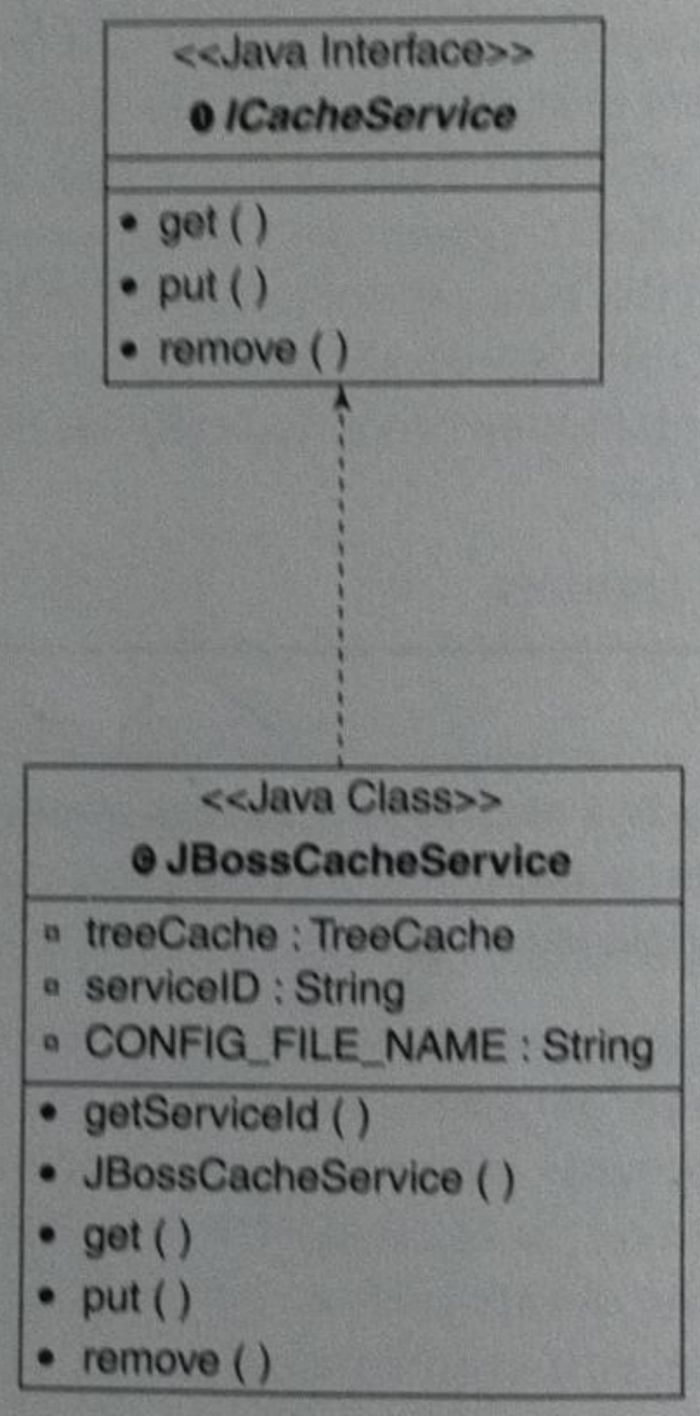


Figure 4-6 Caching—infrastructure component.

An instance of `treeCache` is created within the constructor of `JBossCacheService`, as shown in Code Listing 4.4.

```
public JBossCacheService() throws CacheServiceException {
LogMessage logMessage = new LogMessage(LoggingManager.DEBUG, CLASS_NAME,
"JBossCacheService","Entering Constructor of JBossCacheService");
LoggingManager.log(logMessage);
try {
      treeCache = new TreeCache();
      PropertyConfigurator config = new PropertyConfigurator();
      config.configure(treeCache, CONFIG_FILE_NAME);
      treeCache.setUseInterceptorMbeans(true);
      treeCache.createService();
      treeCache.startService();
    }
catch (Exception ex) {
```

```
        throw new CacheServiceException("JBossCachingService"," Cache
        Constructor", ","Error in Creating Cache Service"+ex.getMessage(),null);
    }
}
```

Code Listing 4.4 `treeCache` instantiation.

Caching modes, locking level and transaction isolation level properties are configured in the JBoss caching service configuration file. Caching mode is required to declare whether caching is required locally or across the cluster. Readers can delve into the code and configuration files provided in the reference code for more implementation details.

An applications' data may be cached at the client-side or at the server-side. Client-side caching is required when there is a strict requirement to conserve network bandwidth. Server-side caching is useful when multiple users connecting to the server require access to the same data. Server-side caching is applicable to all the layers of an *n*-tier enterprise application.

Tt JBoss Cache, JCS (Java Caching System), OSCache are a few of the server-side caching frameworks. Using any of them to implement the caching functionality saves significant development efforts and time and, in general, results in a more robust solution.

To decide whether to opt for data caching, designers have to look into some key questions—Is the amount of cached data manageable? Is the cached data read-only? What are the consequences of stale data? Caching, only if used optimally, provides the expected performance and scalability improvements.

4.3.2 Presentation Layer Components

Construction of presentation components can be done in a variety of ways. Simplest of the options is to construct the application using JSP, Servlet and plain old Java object (POJO) components. These are enough to build a simple presentation layer. But enterprise application developers prefer to use the frameworks to develop the presentation layer components, as they come with loads of out-of-the-box benefits.

The design team has to take care of various things to construct a successful presentation layer. They have to select the right design patterns, decide the right look and feel, give due consideration to input validation, achieve maximum reuse, ensure ease of navigation, session management and caching of data, internationalization and several other issues.

Frameworks and technologies

There are a variety of Java based frameworks available for all of the technical layers of an enterprise application—so also for the presentation layer. This provides flexibility in terms of choice, but also complicates the choice of the framework. Selecting the right framework is the key to ensure performance, maintainability and extensibility of an enterprise application. It also brings standardization and structure to the construction.

Tt Presentation layer construction can be done using frameworks like Struts, JSF, Tapestry, Stripes and many more.

Selection of a framework typically happens during the architecture phase of an enterprise application development as part of laying down the technical architecture. The selection of the presentation layer

framework usually depends on features such as internationalization support, navigation support, request processing, asynchronous JavaScript and XML (AJAX) support, data validation support, testing support and multichannel support. The selection of a framework may not be always based purely on technical reasons. It might be influenced by several other factors such as availability of framework documentation, maturity of the framework, community/vendor support, usability, QoS considerations, availability of skilled workforce, compliance to industry standards and company policies. To start with, let us explore a few of the popular presentation frameworks and evaluate them on the features offered by them.

Struts and JSF are the two most popular presentation layer contenders among the Java presentation frameworks. They both exhibit a lot of similarities. Both are MVC based frameworks, and provide support for managing the request life cycle, request validation, configurability of page navigation and many more things. Let us explore how they differ, as shown in Table 4-2.

Table 4-2 JSF vs. struts

Features	JSF	Struts
Acceptance	JSF is governed by JCP and is specified in JSR 127 as a Java EE standard	Struts, by far, is the most mature and popular framework. But it is not part of the Java EE specifications
Working	Follows an event-driven model, where components of a JSF page can trigger appropriate events	Follows an action-driven model, where a page request is treated as a single event
Support	Relatively new, but gaining maturity rapidly	Very mature and stable, supported by a large developer base and many IDEs
User Interface	Based on a standard UI component model	Does not have a standard UI component model
Client support	JSF supports the non-HTML rendering kit, which makes it easier to plug-in support for WML or XML based clients also	Struts tags generate only HTML
Development	JSF being based on a standard UI component model, supports RAD and reusable components	Struts is based on custom tag libraries and do not support RAD and reusable components

JSF is surely the future, as it is governed by the JCP as a standard specification. But it is yet to be a universal choice for building the presentation layer. If application presentation layer requires features, such as sophisticated presentation components, use of portlets and support for multiple clients, then JSF may be the preferred choice, otherwise Struts is also a good choice.[2]

Package structure

In LoMS, Struts framework is used along with custom application framework components for laying the foundation of the presentation layer. Application specific components are created on top of these

[2] You can learn more about Struts and JSF at http://struts.apache.org/ and http://java.sun.com/javaee/javaserverfaces/, respectively.

framework components. The application framework components and the Struts framework components help increase the reusability of components and standardization of construction. You will encounter a few of these components in the next section. To start with, let us take a look at Figure 4-7 that depicts the package structure of the presentation layer components used in LoMS.

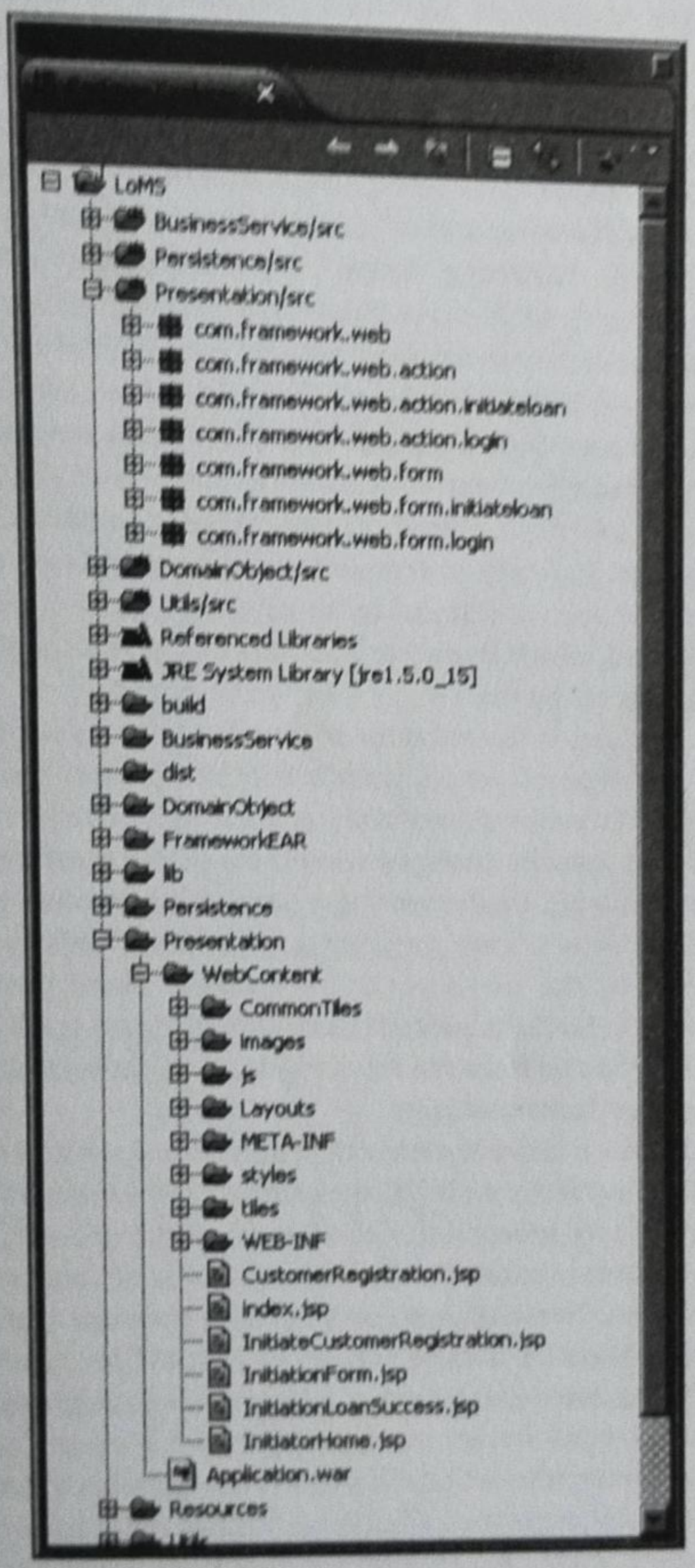


Figure 4-7 Package structure of presentation layer.

It is evident from Figure 4-7 that the presentation layer has broadly two types of artifacts—Web content and source code for the presentation layer. Web content typically comprises of static images,

tiles, style sheets and related configuration files. The source code has the JSPs and other Java artifacts related to the application framework components and the application components.

Application framework components

In LoMS, presentation layer application framework components extend the Struts framework. Figure 4-8 shows the static view of the presentation layer related application framework components.

The application framework has `FrameworkBaseAction`, which is inherited from Struts `Action` class, and it is the mother of all types of application actions and specific classes required for a generic application development. `FrameworkBaseAction` is inherited by specialized actions. The specialized actions are categorized as "viewing action" (`FrameworkTilesAction`), "submitting action" (`FrameworkSubmitAction`), "updating action" (`FrameworkUpdateAction`) and "parking form action" (`ParkFormAction`). All of these base classes utilize the template pattern to define the overall process and provide specific hooks for subclasses to plug in their custom logic.

`FrameworkTilesAction` is the ancestor of all action classes, which are created for displaying a page. When a display action is executed, the required business data is retrieved from the related business service, and then loaded to a form bean to render the contents of the page. Subclasses only need to be concerned with the `retrieveData` method, which is used to retrieve all business data required by this action to render the page. Typically, the required domain objects are retrieved through the business service and stored in the `context` object to be used by `loadForm`. Every display action class needs to implement this method, which is used to populate the form used by this action with business data stored in the `context` object by the `retrieveData` method.

`FrameworkSubmitAction` is the ancestor of all action classes which are created for processing the POST requests. In this type of action, data is submitted from the browser to the server to fulfill some business function. To avoid unnecessary processing, `FrameworkSubmitAction` tries to detect whether the user has actually changed something in the form before it invokes the worker method to process the business logic. Implementing a post action involves two methods. The method `unloadForm` transfers data from a form to criteria or domain objects, and stores them in the `context` object to be used by the `persistData` method. Every post-action class implements the `persistData` method to submit the posted data to the business service for persistence. The data should have been previously collected from the form and put in a context object by the `unloadForm` method in the form of criteria or domain objects.

`FrameworkUpdateAction` is the ancestor of action classes which are created for handling the actions that only update the view locally without repopulating the form with fresh data from the backend, or submitting the form data for processing. For example, the dropdown list of a selection box may need to be rebuilt to show different options in response to a user action, or certain page components may need to be shown or hidden. These changes only occur in the view and do not affect the underlying model. This action should always be invoked with an HTTP POST request in order to retain the state of the form. To implement an update view action, a subclass needs to provide the specific logic in the `process` method, which is the main worker method declared in `FrameworkBaseAction`.

`ParkFormAction` is the concrete action class that is used to park a form. It is used in the place of regular post actions that extend `FrameworkSubmitAction`. Actions that use `ParkFormAction` normally redirect to the display action that renders the next page. Form parking means saving a form in a Web session temporarily, instead of submitting the form data to the business tier to be processed immediately. This might be required in a situation where the current unit of work spans more than one page (and therefore the intermediate steps must be temporarily cached in the session). The business delegate is only invoked from the post action on the last page in the group to process the data

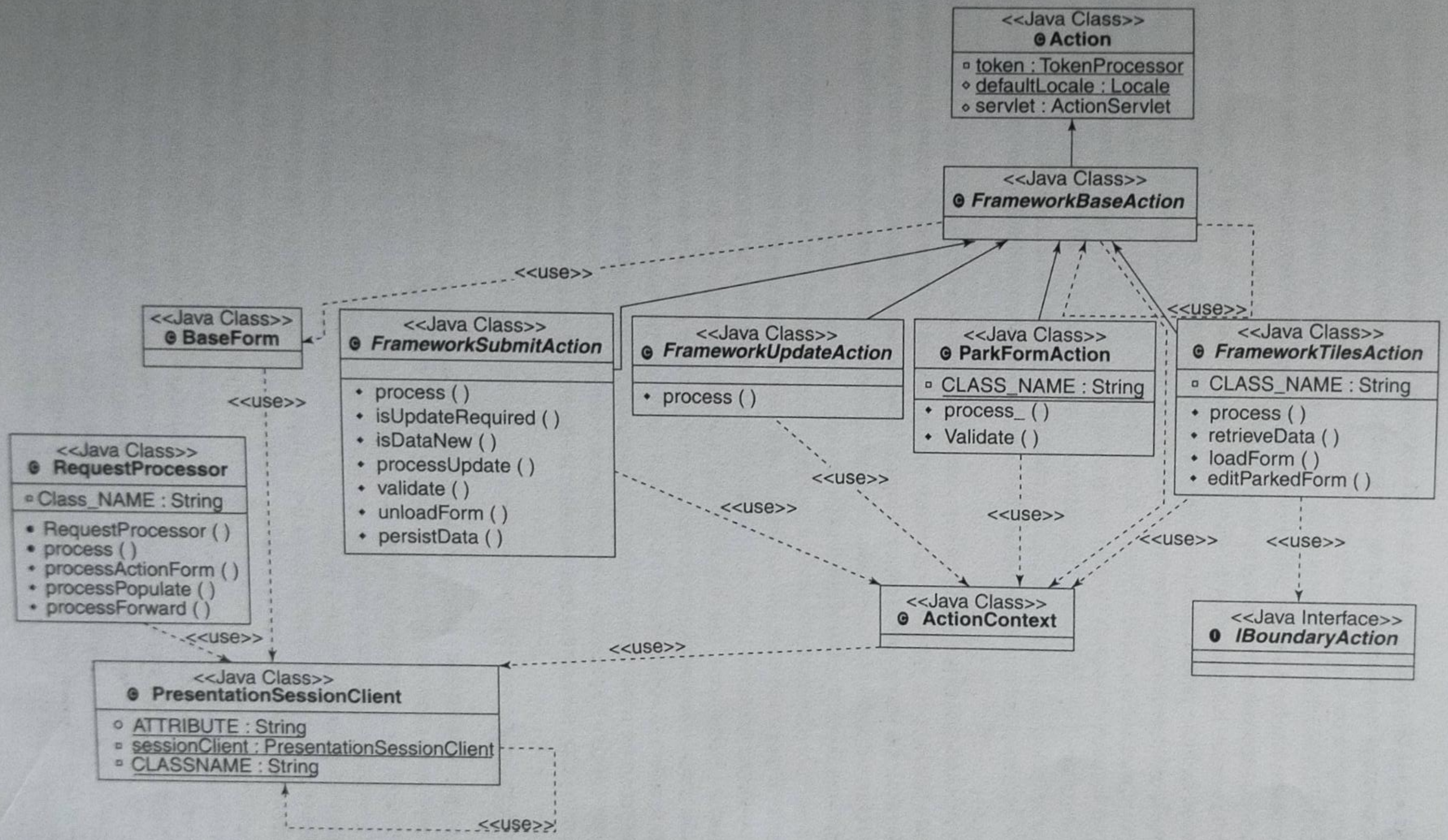


Figure 4-8 Presentation layer—application framework components.

collected by all pages in a group. There can be a situation wherein the user leaves a primary page to go to a secondary page from which he may return to resume working on the primary page. The form for the primary page is parked before the secondary page is displayed. When the user returns to the primary page, the parked form is used to restore the page back to the state it had when the user left it. In both these cases, the form should still be declared as having request scope in the action mapping. The framework is responsible for managing the cache of parked forms.

`ActionContext` is a presentation layer framework class that is used to organize the standard parameters used by Action methods into a single structure to simplify method signatures. This helps in achieving simplicity for developers in terms of writing the code. It also provides a single-point of access to the `PresentationSessionClient`. It also provides transient storage of any action-specific data during the scope of the current action. An instance of `ActionContext` is created by the framework at the entry-point of an action. This instance is then passed to all the worker methods of the action. `ActionContext` has `getSession` method to return the `FrameworkSession` object. The `ActionContext` instance is created by the `execute` method of the `FrameworkBaseAction` class.

The only method that may need to be overridden in `FrameworkBaseAction` is `getForward`, which provides the logic for determining where control should be forwarded upon the completion of the action. It returns a string representing a logical forward name defined in the struts configuration file. If null is returned, either failure or success is substituted by the framework depending on whether error messages are present in the request object or not.

All `ActionForm` classes in the application are based on `BaseForm`, which extends `ActionForm` in Struts. Every JSP page will have its own form that is basically used as the form bean.

This section has presented the description of a few of the key presentation layer components of the application framework. You may refer to the accompanying code for further details on implementation. All these framework components of presentation layer are designed considering Struts as the base presentation tier framework. These components may not hold well, partly or completely, in using some other presentation tier framework like JSF being used. But whatever be the choice of layer-specific framework, the application framework components are designed to achieve reusability, simplicity, robustness and maintainability among other desired things in an application framework.

Application components

Presentation layer application components are the core application components that can be traced back to the use cases captured in the requirement phase. These components are designed and implemented by reusing the application framework components. The application framework and application components are stitched together to construct an enterprise application.

The sequence diagram (Figure 4-9) represents how the application components work together in the presentation layer for the realization of the "initiate loan" use case in the presentation layer.

When the actor bank customer invokes "Initiate Loan", a request is raised for the action path `initiationHome.do`. The `RequestProcessor` looks into the memory structure, created by the `ActionServlet` by parsing through the `struts-config.xml` for the mapping details of `initiationHome.do` action path. Code Listing 4.5 depicts an excerpt from the `struts-config.xml`, which is the configuration file of the Struts framework. The `RequestProcessor` populates the `com.framework.web.form.initiateloan.InitiateLoanForm` form bean and invokes the `execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)` of `com.frame work.web.action.initiateloan.ViewInitiationFormAction` action class.

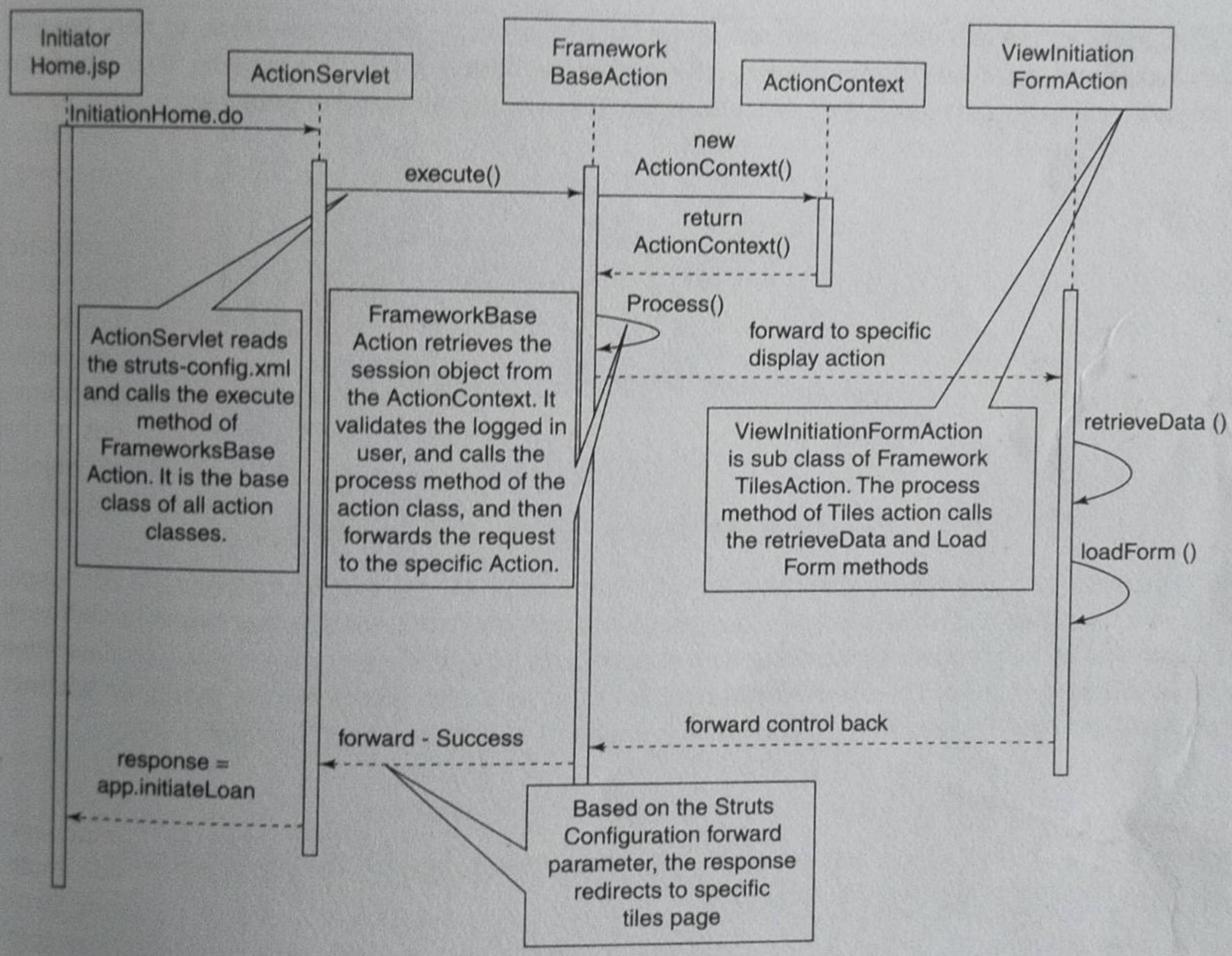


Figure 4-9 View initiation form sequence.

```
...
<form-bean name="initiationForm"
      type="com.framework.web.form.initiateloan.InitiateLoanForm" />
...
<action path="/initiationHome" name="initiationForm" scope="request"
input="InitiatorHome.jsp"
type="com.framework.web.action.initiateloan.ViewInitiationFormAction"
            validate="false">
                  <forward name="Success" path="app.initiateLoan" />
                  <forward name="Failure" path="/index.jsp" />
</action>
```

Code Listing 4.5 Excerpt from `struts-config.xml`.

The `execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)` method of `com.framework.web.action.FrameworkBaseAction`, which is the super class of `com.framework.web.action.initiateloan.ViewInitiationFormAction` action class, instantiates the `ActionContext` populating the `ActionMapping`, `ActionForm`, `HttpServletRequest`, `HttpServletResponse` and `Locale`.

The method checks whether the customer information is already available in the session, which would be set while authenticating the customer during login. If available, it invokes the `process(ActionContext context)` method. These steps are listed in Code Listing 4.6.

```
…
if (null != user) {
                    if (LOGIN_PATH.equals(requestPath) &&
(context.getSession().getAttribute("SESSION_EXIST") == null)) {
                            // …
                    } else {
                            forward = process(context);
                    }
            }
```

Code Listing 4.6 Excerpt from the `execute` method.

The `execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)` method determines whether any error is associated with the request object by invoking `errorsExist(ActionContext context)`, as shown in Code Listing 4.7. If so, the method returns `ActionForward` object for the mapping "Failure". Otherwise, the method returns `ActionForward` object for the mapping "Success".

```
…
if (forward == null) {
forward = errorsExist(context) ? FAILURE_FORWARD :SUCCESS_FORWARD;}
return mapping.findForward(forward);
```

Code Listing 4.7 The `execute` method continued …

The corresponding view, `InitiationForm.jsp` is rendered to the customer. The above example illustrates how the presentation layer's application components are built by leveraging application framework components. Let us now conclude the presentation layer components section by looking at some sample software construction maps that are self-explanatory. They capture the big picture of several application and application framework components, participating in a specific scenario in an easy-to-understand fashion. Developers can complement them with other design diagrams, as necessary, to aid the construction task.

Figure 4-10 is a software construction map to capture the AJAX implementation in "validate SSN" scenario of the "Initiate Loan" usecase. In this scenario, a user-supplied SSN is validated in order to fetch the preapproved loan amount for a customer. The "validate SSN" scenario is implemented using Web services.[3]

The next software construction map, as shown in Figure 4-11, captures the interactions among the session and caching management components in LoMS application.

The software construction map in Figure 4-12 illustrates the login and concurrent user session validation usecase, which is another crucial requirement in any multiuser enterprise application.

---

[3] You will explore more on this in the integration layer components section in this chapter.

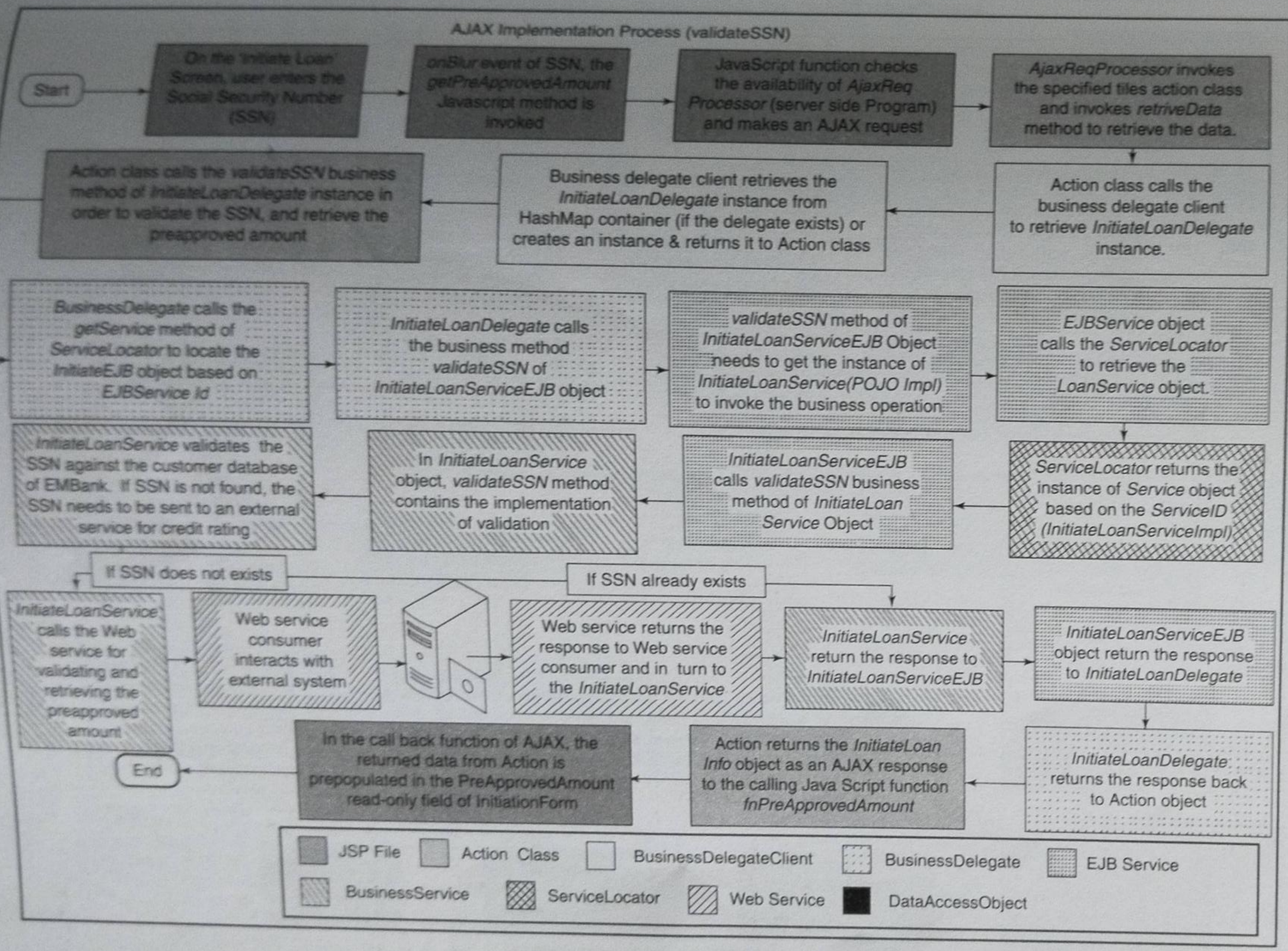


Figure 4-10 Software construction map for AJAX implementation.

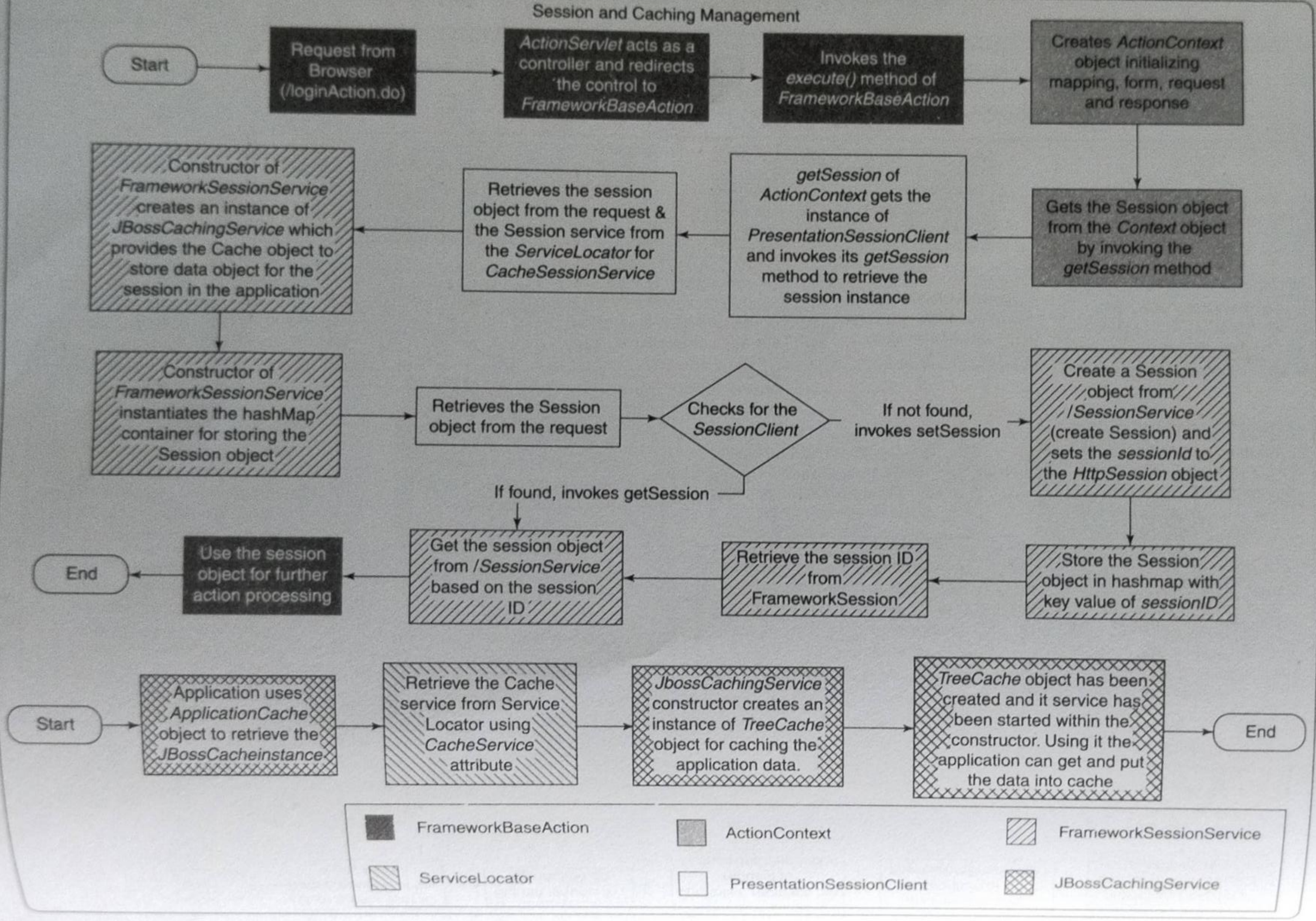


Figure 4-11 Software construction map for session and caching management.

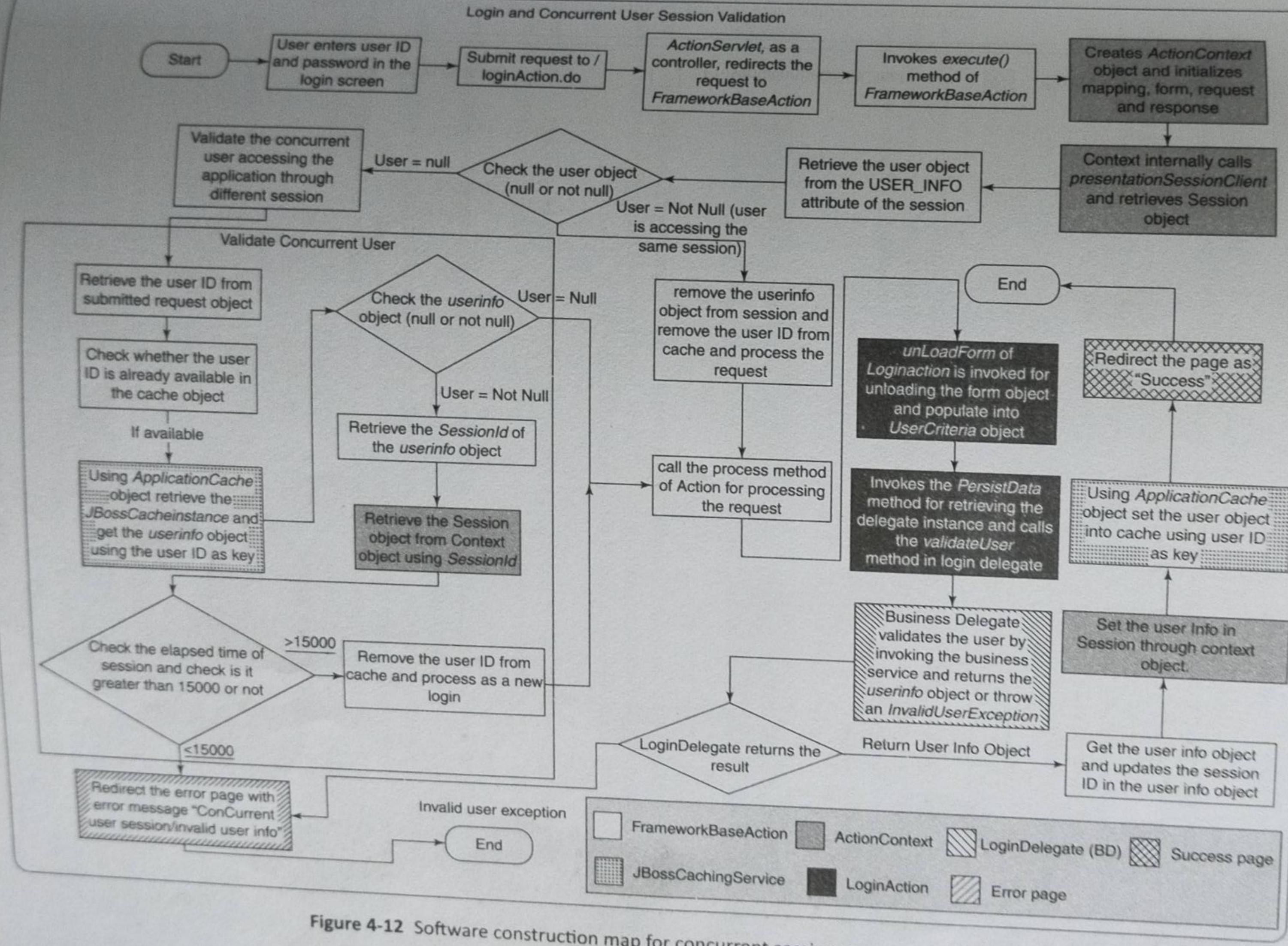


Figure 4-12 Software construction map for concurrent session management.

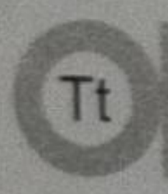
4.3.3 Business Layer Components

Like other layers, the construction of business layer components can also be done in several ways. Implementation using POJO is among the simplistic of the options available. The other usual choices for implementation are extending readily available frameworks.

Design team has to consider several aspects to ensure the construction of a robust and flexible business layer such as applying the right design patterns, designing optimized and appropriate transaction management and concurrency control mechanisms, designing business workflows and business rules repositories, and many more things.

Frameworks and technologies

As discussed in the previous section, enterprise application developers usually prefer to use frameworks to develop application layer components to leverage framework's easy-to-use and out-of-the-box capabilities. The developer community has multiple choices to select from the available business layer frameworks/technologies.



There are two strong players in the area of developing business layer components: Spring framework and EJB 3.0.

Spring framework is an open-source application framework. This framework is not just limited to the design and implementation of the business layer. It comprises of several sub frameworks to provide various services across the layers of a typical *n*-tier enterprise application:

- For Web/presentation tier, it provides an MVC framework with rich UI support. It extends support for JSP tag libraries and other popular presentation tier frameworks such as Struts and JSF.
- Spring framework's transaction management framework provides support for both programmatic and declarative transaction management mechanisms.
- Spring framework's data access framework extends support for almost all of the popular data access frameworks including Hibernate, TopLink, JDO, JPA and iBatis.
- Spring framework also supports batch processing and remoting by using spring batch and spring remote access framework, respectively. The remoting feature of Spring framework supports technologies such as RMI, RMI-IIOP and SOAP.

Two important features of Spring framework, which need special mention, are *inversion of control* (IoC) and AOP. The concept of IoC is the reverse of the traditional programming model, where application components have the control, and they maintain the flow and dependencies among themselves. In IoC, the control and dependency management is with the framework, which calls the application code, rather than the application code calling the framework components. It is something like doing class wiring by using configuration, instead of coding. The configuration files contain the dependencies among the Java objects. The Spring framework IoC container uses this dependency information and "injects" the referenced objects on demand, and is referred to as *dependency injection* (DI).

Spring framework's AOP framework provides the basic support for aggregating in one place the cross-cutting concerns of an enterprise application by using aspects. Other AOP frameworks, such as AspectJ, are relatively more comprehensive.[4]

EJB 3.0 API is one of the core APIs of the Java Enterprise platform meant for distributed computing. The basic idea of using EJB is to simplify the development process by providing services for scalability, security, life cycle management and transaction management to the application so that its focus can be limited to the business logic. The previous versions of EJB were complex, but EJB 3.0 has been introduced as a lean and simpler API. The introduction to metadata annotations has significantly reduced complexity by removing the need for creating deployment descriptors, home interfaces and remote/local object interfaces, which are now generated by the EJB container itself. Classes and interfaces of EJB are now created using POJO and plain old Java interface (POJI), and use the dependency injection mechanism through which the container makes the bean instances.[5]

Both EJB and Spring framework have their own pros and cons. But the comparison between the two is always a moving target, as they are getting simpler and richer version to version. They are not mutually exclusive, however, and can be combined to get the best of both the worlds. In the LoMS application framework, both EJB 3.0 and Spring framework are used in an integrated manner. Table 4-3 provides a comparative analysis of EJB 3.0 and the Spring framework.

Table 4-3 EJB 3.0 vs. Spring framework

Features	EJB 3.0	Spring framework
Acceptance	EJB 3.0 is governed by JCP and is specified in JSR 220 as Java EE standard	Spring framework is an open-source framework, and is not a Java standard. It is a popular alternative to EJB 3.0
Transaction support	EJB 3.0 uses transaction support provided by the EJB container. It only supports JTA transactions	Spring framework supports transactions using Spring AOP's transaction aspects
State management	An EJB 3.0 application maintains state by using a Stateful Session EJB. Stateful Session EJB is one of three different kinds of EJBs in the JEE technology stack	Spring framework maintains state by using non-singleton (or *prototype* beans) and scoped beans
Dependency injection	EJB 3.0 has primitive support for dependency injection	Spring framework has exhaustive support for several types of dependency injections

Package structure

In LoMS, both the Spring framework and EJB components are extended to create a custom application framework for laying the foundation of business layer components. Application specific components

[4] For further information on Spring framework and AspectJ, you can refer to the reference documentation on http://static.springframework.org/spring/docs/2.0.x/reference/index.html and http://www.eclipse.org/aspectj/, respectively.

[5] For further information on EJB, you may refer to http://java.sun.com/products/ejb/.

e created on top of these components. For understanding application-specific components, excerpts om the "Initiate Loan" use case are provided below. To start with, let us look at Figure 4-13, showing e package structure of the business layer components.

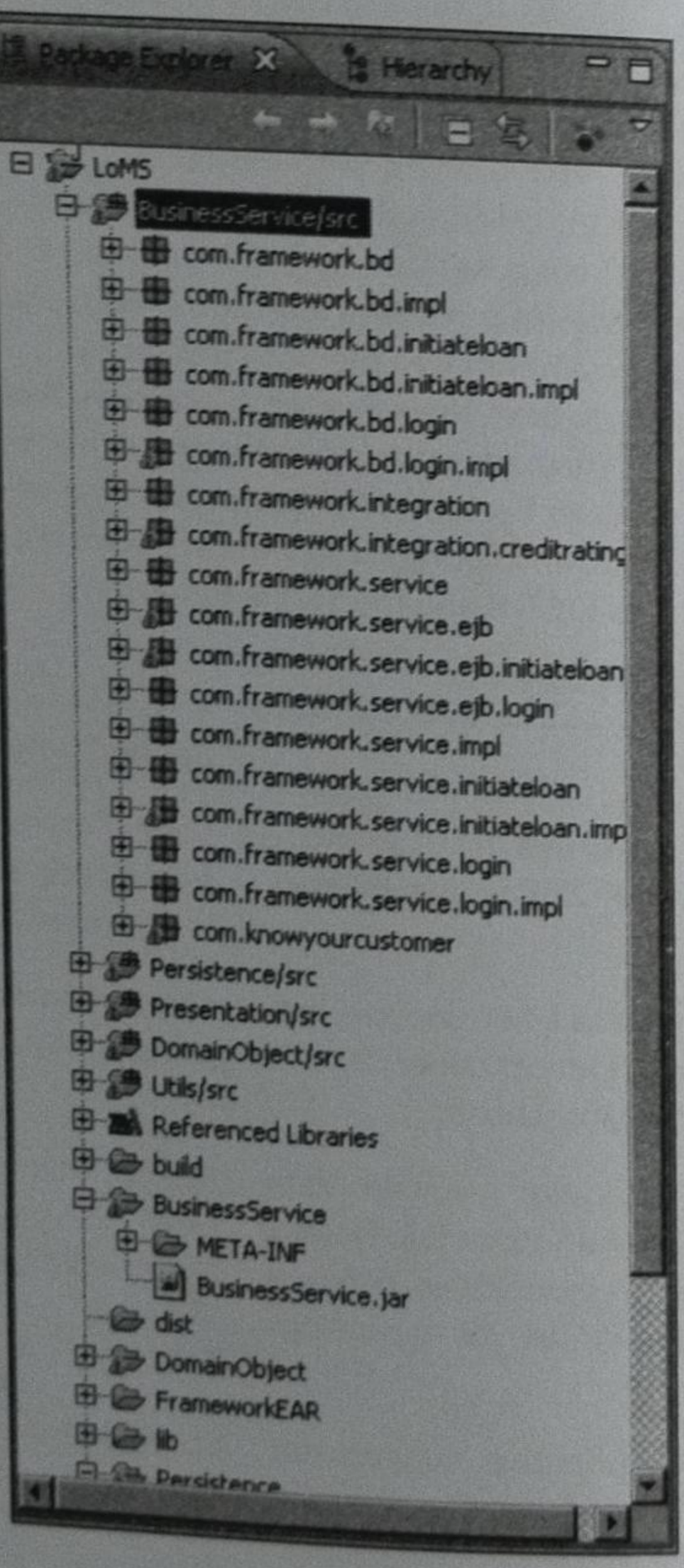


Figure 4-13 Package structure of business layer.

Application framework components

In LoMS, the business layer application framework extends EJB and Spring framework components along with several other Java components. Figure 4-14 depicts the static view of the business layer related application framework components. In this example, the business layer framework of the application comprises of Business Delegate components, session façade components, business service components and business model components.

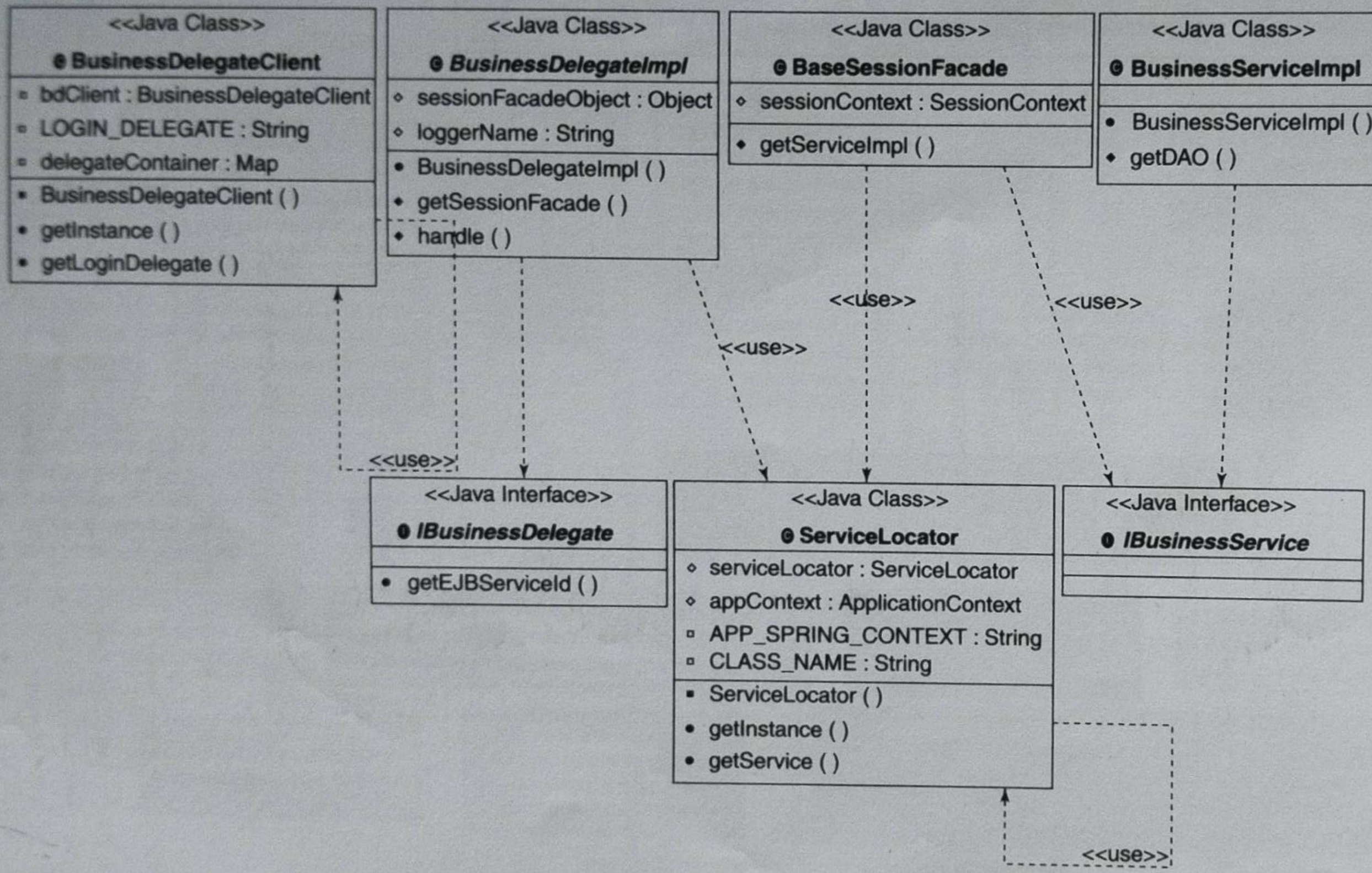


Figure 4-14 Business layer—application framework components.

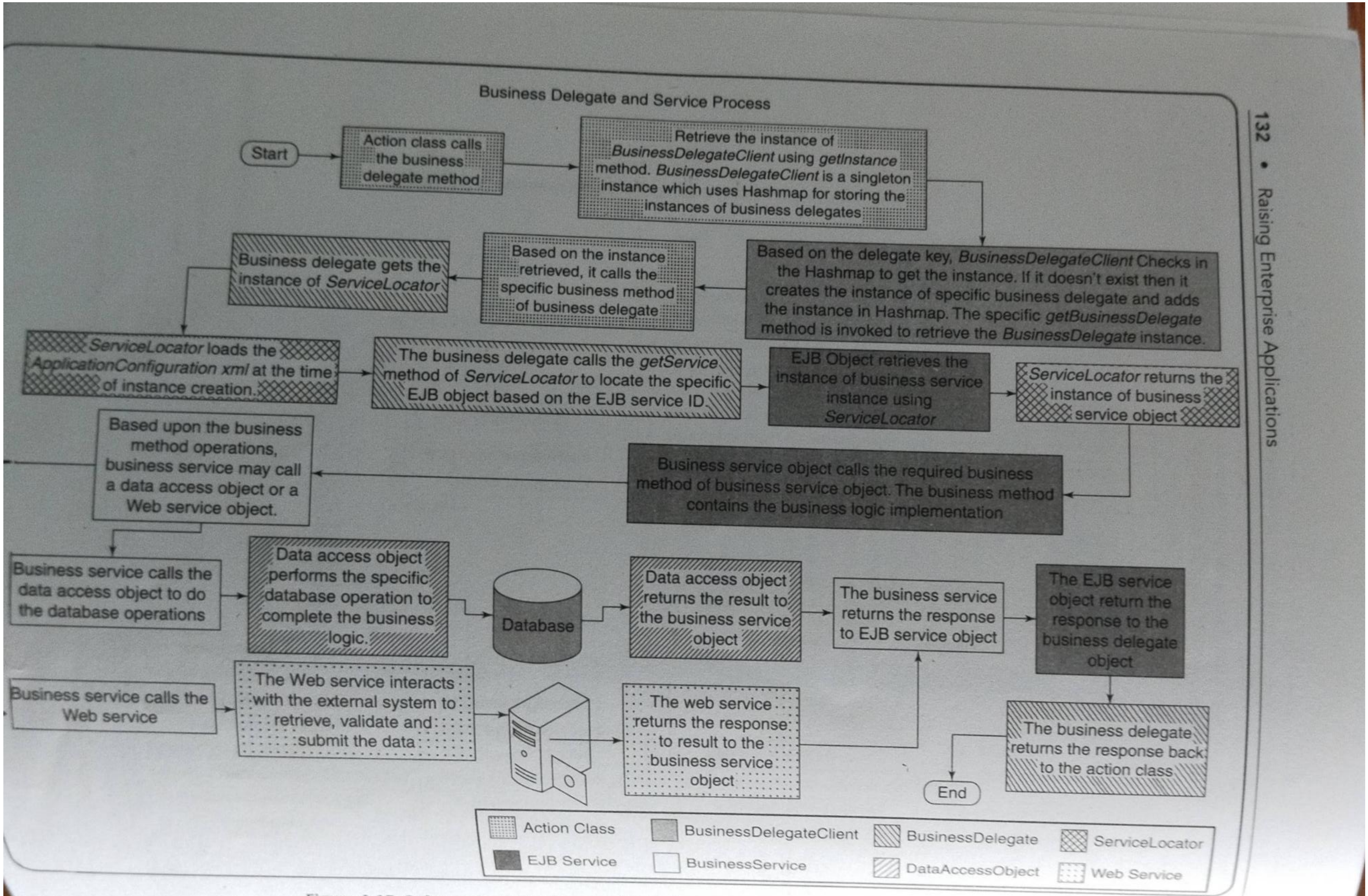


Figure 4-15 Software construction map for Business Delegate and Service Process.

There are three elements of Business Delegate components:

- `IBusinessDelegate` is the business delegate interface. All business delegate interfaces extend this base interface. This interface has `getEJBServiceId` method to return the EJB Service ID name.
- `BusinessDelegateImpl` is an implementation of `IBusinessDelegate`. All business delegate classes extend this base delegate. It has the implementation of `getSessionFacade` method to get the session bean object through `ServiceLocator`.
- `BusinessDelegateClient` class is used by the presentation layer to get the Business Delegate objects, which in turn are used to invoke business methods.

There are two elements of Business Service components:

- `IBusinessService` is the business service interface. All the framework application business service interfaces extend this interface.
- `BusinessServiceImpl` is the implementation of the `IBusinessService` interface. It contains `getDAO` method implementation to get the specific data access object.

Figure 4-15 depicts the big picture of the business delegate and the service process using a software construction map. There are three important elements of Business Model components, as shown in Figure 4-16:

- `IBusinessModel` is the business model interface. This interface extends the serializable interface.
- `BusinessModel` is an implementation of the business model interface. All the framework application business model implementation classes extend this class.
- `ICriteria` is the criteria interface. The objects that implement this interface act as a request object containing the request parameter information to perform business operations.

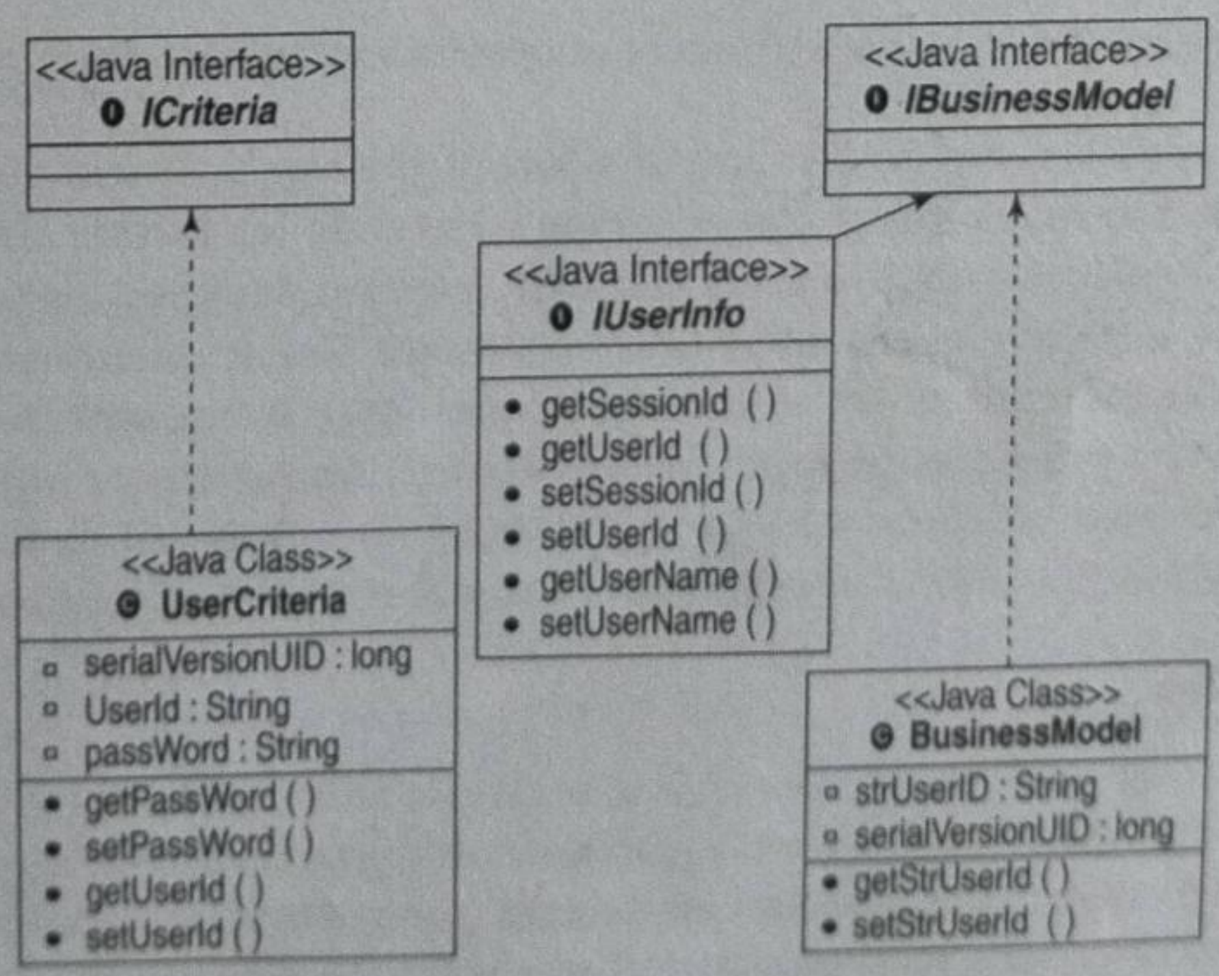


Figure 4-16 Business layer—business model application framework components.

The package structure of the criteria and domain objects is shown in Figure 4-17.

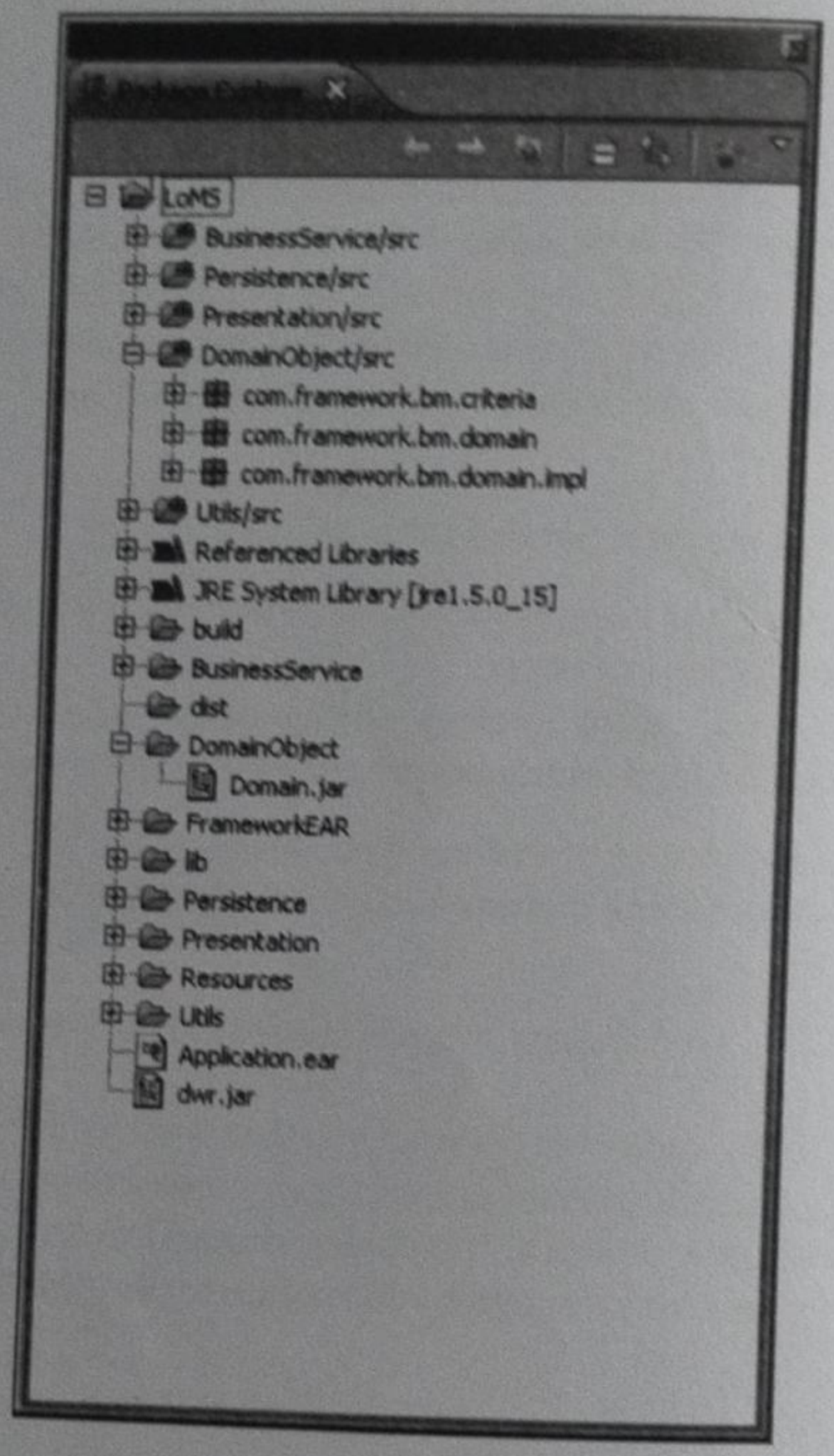


Figure 4-17 Package structure of criteria and domain objects.

This section has presented the description of a few of the key business layer components of the application framework. You may refer to the accompanying code for further details about the implementation. All these framework components of business layer are designed, considering EJB and Spring framework as the base business layer framework/technology. These components may not hold well, partly or completely, in the case of some other business layer framework/technology being used. As explored before in the presentation layer framework section, whatever be the choice of a layer-specific framework, the application framework components are designed to achieve reusability, simplicity, robustness and maintainability among other desirable properties in an application framework.

Application components

Application components in the business layer are the components that form the core of an application, and can be traced back to the use cases captured in the requirements phase. As you already know, as is the case with any other layer, the application framework components and application components are stitched together to construct the business layer of an enterprise application.

Let us further consider the "initiate loan" use case to understand how the business layer application components work together. The sequence diagram, as in Figure 4-18, represents the

sequence of interactions among the components that implements the "initiate loan" use case in the business layer.

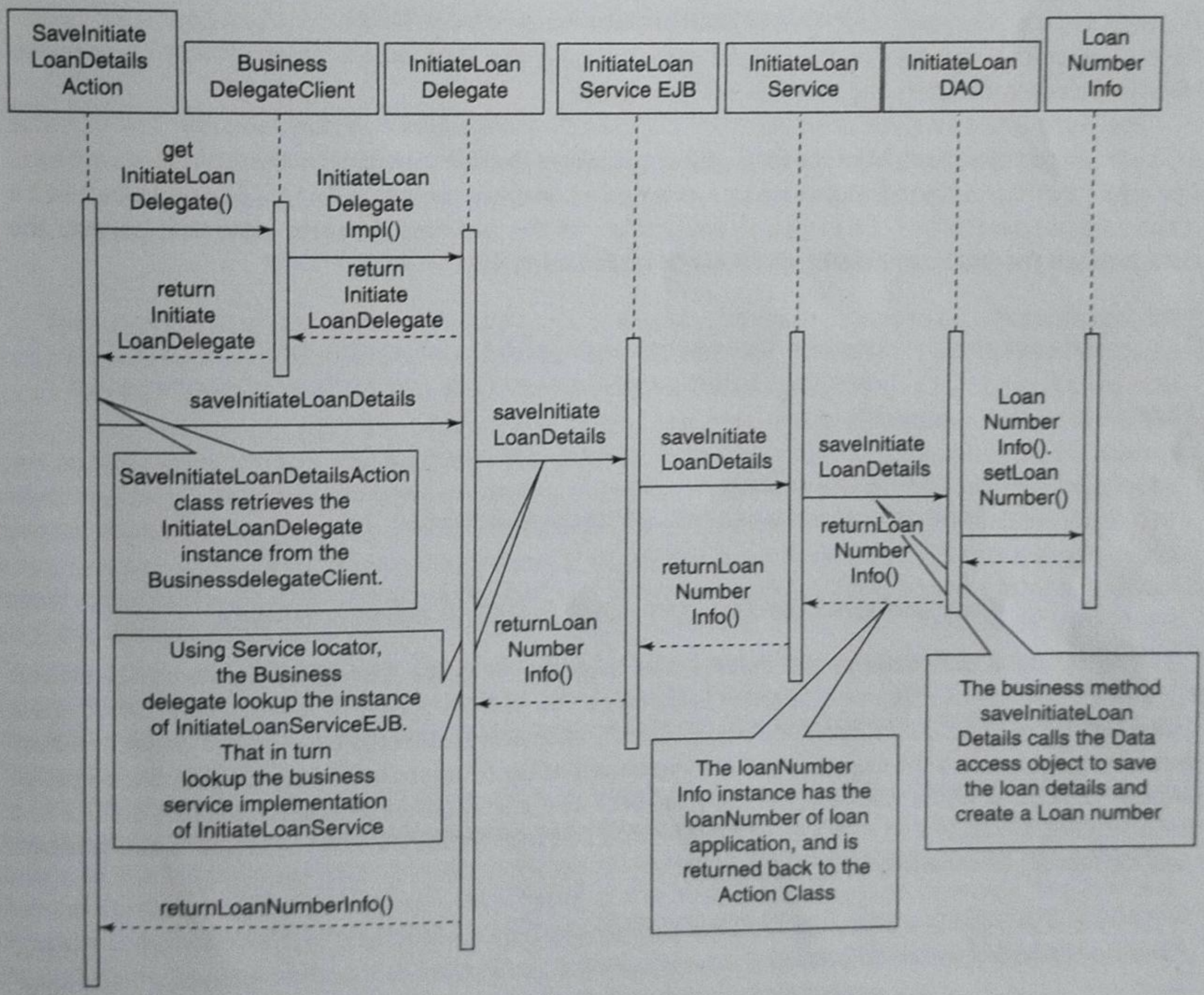


Figure 4-18 Initiate loan service—submit and save loan initiation.

The `InitiationForm.jsp` raises a request for the action path `initiationAction` when the form is submitted. The `RequestProcessor` of the Struts framework instantiates and populates the the `com.framework.web.form.initiateloan.InitiateLoanForm`, and invokes the `execute(ActionMappingmapping, ActionFormform, HttpServletRequestrequest, HttpServletResponse response)` method of the `SaveInitiateLoanDetailsAction` action class. The method instantiates the `ActionContext` and retrieves the session. The method invokes the `isUpdateRequired(context)` to verify whether the form has been modified, as shown in Code Listing 4.8.

```
if (isUpdateRequired(context)) {
    processUpdate(context);
}
```

Code Listing 4.8 Verifying the form for modifiability.

This validation ensures that no form is submitted more than once without modification. The `processUpdate(ActionContext context)` unloads the form by invoking `unloadForm(ActionContext context)`, and invokes the `persistData(ActionContext context)` to persist the data in the database by invoking the data access object (DAO) components through business delegate and business session façade.

The `unloadForm(ActionContext context)` instantiates and populates the `InititateLoanInfo` domain object from the form object and stores the domain object in the `ActionContext`. The `persistData(ActionContext context)` invokes the `saveInitiateLoanDetails(InitiateLoanInfo initiateLoanInfo)` of the business delegate class that persists the data through the DAO component, as shown in Code Listing 4.9.

```
try {
        loanNumber = context.getSessionClient().getDelegateClient()
                    .getInitiateLoanDelegate().saveInitiateLoanDetails
(info);
   }
 catch (BusinessException ex) {
            addError(context, ex.getMessageTextId());
  }
```

Code Listing 4.9 Dataflow from action to DAO via business delegate.

The Struts action component invokes the business delegate through `BusinessDelegateClient`. The `getInitiateLoanDelegate()` of the `BusinessDelegateClient` class instantiates the `InitiateLoanDelegateImpl`, and returns the reference to it. Further, the business delegate invokes the business service implementation component through the business session façade. `InitiateLoanDelegateImpl` invokes the `saveInitiateLoanDetails(InitiateLoanInfo initiateLoanInfo)` of `InitiateLoanServiceEJB`, which is the business session façade, as shown in Code Listing 4.10.

```
IInitiateLoanEJBLocal initiateLoanEJB = (IInitiateLoanEJBLocal)
super.sessionFacadeObject;
```

Code Listing 4.10 Business delegate implementation.

As shown in Code Listing 4.11, the `sessionFacadeObject` is initialized to the `InitiateLoanServiceEJB` business session façade, as `getEJBServiceId()` returns `InitiateLoanServiceEJB`.

```
sessionFacadeObject =
ServiceLocator.getInstance().getService(getEJBServiceId());
```

Code Listing 4.11 Business session façade.

The business session façade invokes the `saveInitiateLoanDetails(InitiateLoanInfo criteria)` of `InitiateLoanServiceImpl`, which is the business service implementation component for loan initiation, as shown in Code Listing 4.12.

```
IInitiateLoanService    initiateLoanService    =    (IInitiateLoanService)
super.getServiceImpl(serviceID);
      return initiateLoanService.saveInitiateLoanDetails(criteria);
```

Code Listing 4.12 Session façade invocation.

The `super.getServiceImpl(serviceID)` instantiates the `InitiateLoanServiceImpl` business service implementation component, as shown in Code Listing 4.13.

```
businessService = (IBusinessService)
ServiceLocator.getInstance().getService(serviceID);
```

Code Listing 4.13 Business service component initiation.

The `saveInitiateLoanDeatils(InitiateLoanInfo criteria)` instantiates the `InitiateLoanDAO` through service locator, and invokes the `saveInitiateLoanDetails (criteria)` of `InitiateLoanDAO`. The `saveInitiateLoanDetails(InitiateLoan Info criteria)` of `InitiateLoanDAO` persists the data into the database and retrieves the loan number generated by the database. The method instantiates the `LoanNumberInfo` domain object and populates the loan number into the object. The method returns the `LoanNumberInfo` domain object to the `persistData(ActionContext context)` of the Action class. The `persistData(ActionContext context)` stores the `LoanNumberInfo` into the session. In case if either of the delegate components, business components or DAO components raise any exception, the exception is added on to the request object using the `addError` method defined in the `FrameworkBaseAction` class. The `execute(ActionMapping mapping, ActionForm form, HttpServletRequest request, HttpServletResponse response)` method of the Action class determines whether an error is associated. The method returns the corresponding `ActionForward` object.

In the overall flow of the business layer components, readers have noticed that without the business delegate, presentation-tier components have direct exposure to the underlying implementation details of the business services. It might necessitate changing the implementation of the presentation-tier components in case the implementations of the business services get changed. The Business Delegate component in turn interacts with service components using service locator, which abstracts the lookup mechanism. In the LoMS framework, a service locator is used for all kinds of look up services, e.g. business service lookup, EJB lookup and data access components lookup.

In case where the Spring framework is used, the IoC container of framework takes care of providing all the dependencies for a given components, and the nitty-gritty of locating and instantiating the dependencies are transparently managed by the container. LoMS leverages these features of the Spring framework in its implementation.

4.3.4 Data Access Layer Components

Like other layers of an enterprise application, construction of the data access layer components can be done in several ways. Developers have multiple choices ranging from POJOs to the available persistence layer frameworks/technologies. In an enterprise application, the data access layer should be designed in a manner that it is decoupled not only from the business components but also from the underlying relational tables or any other data source.

Frameworks and technologies

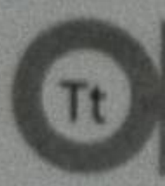
The simplest of the data access layer technologies is *Java Database Connectivity* (JDBC), wherein SQL queries are directly embedded in Java code. In such a scenario, there is a tight coupling between Java objects and the underlying relational data stores. The concept of *object relation mapping* (ORM) has come into practice to avoid such coupling. ORM provides a bridge between the Java object world and the data store's relational world by providing mappings between the two through the XML configuration files or annotations.

The ORM approach enables developers to think in terms of objects rather than tables or relations, when it comes to persistence of objects. This enables developers to apply object-oriented best practices in implementing a data access layer.

Hibernate, Java Persistence API (JPA), EJB, TopLink and iBatis are some of the popular frameworks/technologies, which provide ORM based implementation of the data access layer.

Hibernate framework is an open-source ORM framework that allows developers to implement data access layer classes using object oriented features such as polymorphism, inheritance and encapsulation. Hibernate also supports Hibernate Query Language (HQL), which is like a primitive OO extension to the SQL. It also provides native SQL support. Hibernate framework is currently the most popular ORM framework for Java based persistence solutions.

JPA is a specification, which has been introduced as part of Java EE 5 platform, standardizes the way persistence layers are developed. JPA has Java persistence query language (JPQL) with which developers can query the persistent entity objects.



The Spring framework provides Spring DAOs and Hibernate DAOs to build the data access layer.

Let us now explore a few differences between JPA and other data access layer frameworks, as presented in Table 4-4. Some best practices, which are required to construct a robust and scalable data access layer, are as follows:

- **Identifying callbacks**. Every entity, which needs to be persisted, has a life cycle associated with it. The entity may undergo certain operations or manipulations during the life cycle. For example, a prerequisite for the phone number of a customer to be persisted is to take off the hyphens and the brackets from the phone number field. Persistence frameworks provide support to capture such manipulations via the "callback" methods. JPA and Hibernate framework provide them as annotations.
- **Identifying embedded objects**. Identification of embedded entity objects can result in improving reusability. For example, an address entity can be embedded as customer's communication address, loan guarantor's address, customer's permanent address, etc. JPA has @Embedded annotation to do this.

Package structure

In LoMS, POJO based implementation of the DAO pattern is used to design both the framework components and the application components of the data access layer. Readers have already seen the application specific components of the data access layer in the previous section, where excerpts from the "Initiate Loan" use case were discussed.

Table 4-4 JPA vs. other data access layer frameworks

Features	JPA	Other data access layer frameworks
Acceptance	JPA is a Java standard specification (JSR 220) and has been introduced in Java EE 5 platform	Other popular data access layer frameworks are Hibernate, iBatis and Oracle TopLink
Control on queries	Control on queries is limited as it generates the query automatically	A strong control over queries is possible as the onus is on the developer to map Java objects and queries
Applicability	Usage of JPA is preferred in applications, where control on queries including their fine tuning is not the main criteria	Frameworks such as iBatis are used in database centric applications where tight control on queries is required

In this section, readers will see the package structure of the data access layer used in LoMS, and later explore the application framework components specific to the data access layer. Figure 4-19 depicts the package structure of the data access layer components.

Application framework components

In LoMS, the data access layer framework is designed using POJOs. Figure 4-20 shows the static view of the framework components of the data access layer.

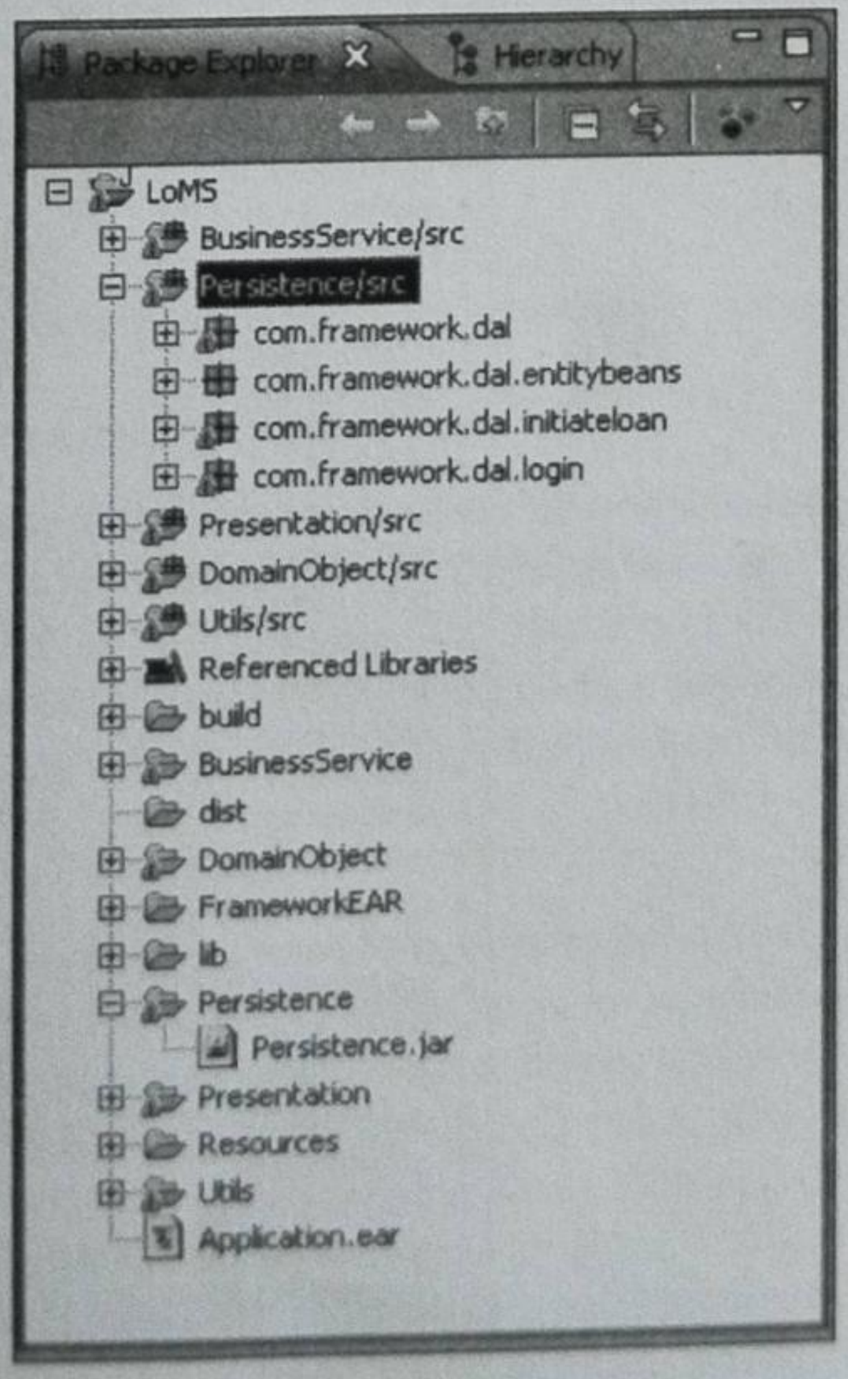


Figure 4-19 Package structure of data access layer.

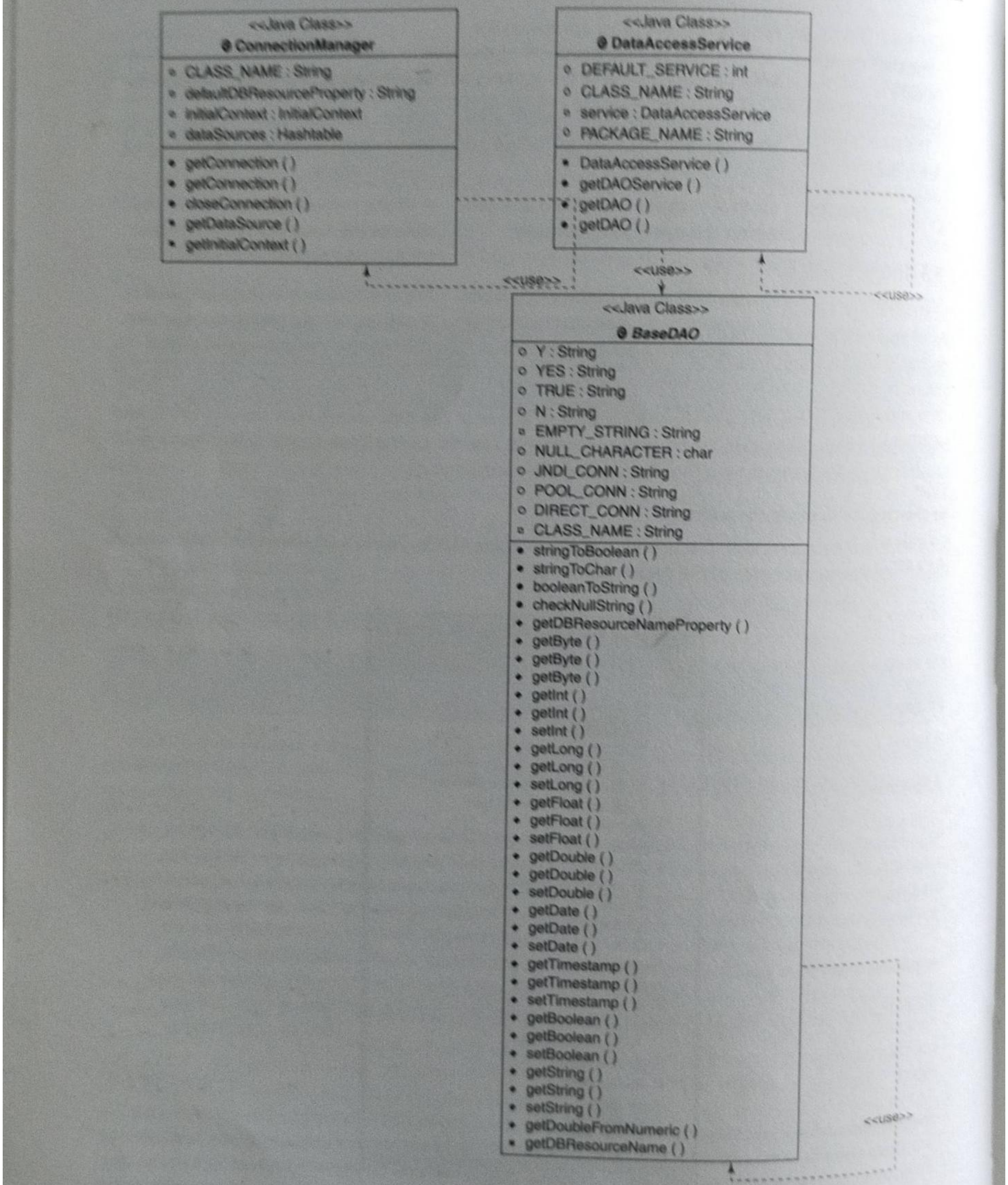


Figure 4-20 Data access layer—application framework components.

The DAO implementation encapsulates the mechanisms used to access the underlying data stores. If the data source is a relational database, as in LoMS, simple JDBC, JPA or other ORM frameworks can be used to implement the mechanism to access data stores.

The DAO implementation may also encapsulate a service that integrates with another application to fetch data required by the enterprise application. In such a case, either a low-level socket based mechanism can be used or a high-level API (such as RMI or JMS) can be used to implement the underlying access mechanism.

4.3.5 Integration Layer Components

Construction of integration layer components can be done in a variety of ways, and at different levels (of application layers), as discussed in Chapter 3, where you have learnt about several integration mechanisms and technologies along with the design elements that facilitate the integration of applications by realizing a robust integration layer. In this section, you will go through an integration scenario and understand how the application framework components and application components work together to implement it.

The sample integration scenario is as follows: LoMS has to determine the preapproved amount for a loan applicant as part of the "Loan Initiation" use case. The preapproved amount is determined based on the credit rating of the customer. An external credit rating agency provides the credit rating of an individual based on their transaction history and liabilities. LoMS has to access the KnowYourCustomer Web service hosted by this agency to retrieve the credit rating of the customer.

Integration layer of LoMS uses JAX-WS API to implement the SOAP-based Web services integration between LoMS and credit rating system. As readers have learnt in Chapter 3, the JAX-WS API is used to expose a Web service as a POJO interface. When the Web service consumer invokes the service method of the POJO interface, it automatically generates a SOAP request message to communicate with the provider.

Package structure

Figure 4-21 shows the package structure of the integration layer components, as used in LoMS.

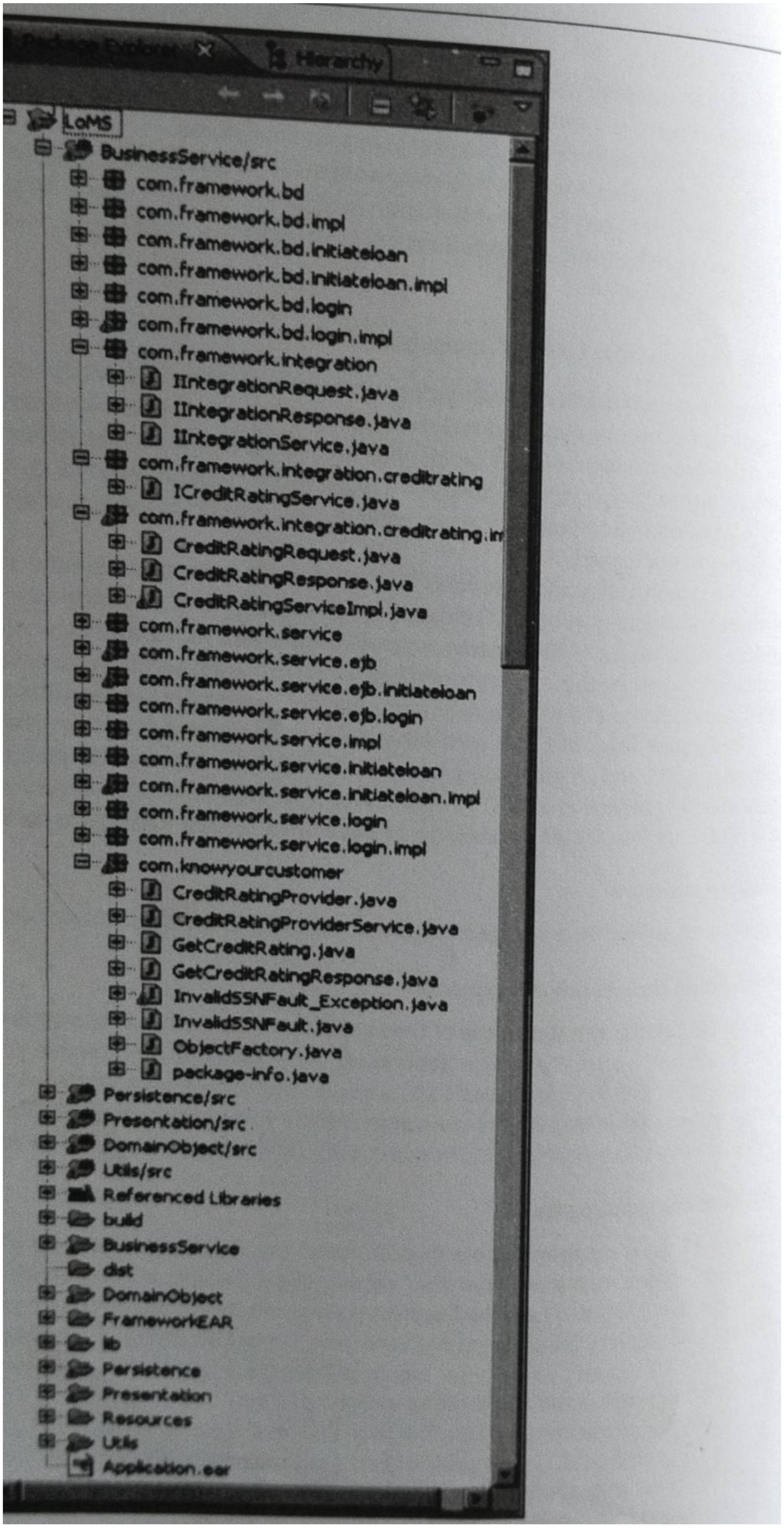
Application framework components

Figure 4-22 shows the static view of the application framework components related to integration layer. The `IIntegrationService` generalizes the underlying Web service consumers' implementation. `IIntegrationRequest` and `IIntegrationResponse` hold the parameters that get communicated between the Web service consumer and the Web service provider. The consumer uses the JAX-WS API to invoke the `KnowYourCustomer` Web service that is provided by the credit rating agency.

Application components

Integration layer components are built on top of the application framework components specific to the integration layer. Let us consider the "validate social security number (SSN)" scenario in "initiate loan" use case to understand how the business layer application component integrates with the Web service. Figure 4-23 depicts the sequence of retrieving the credit rating of a customer based on his SSN. The `InitiateLoanServiceImpl` has to invoke the `KnowYourCustomer` Web service to validate the SSN and retrieve the credit rating of the customer.

Once the user enters the SSN, the form invokes the `com.framework.web.action.intiateloan.ValidateInitiationAction` class through the AJAX processor. The `ValidateInitiationAction` class, through the `InitiateLoanDelegate` and `InitiateLoanServiceEJB`, invokes the `validateSSN(InitiateLoanInfo initiateLoanInfo)` of `InitiateLoanService`.



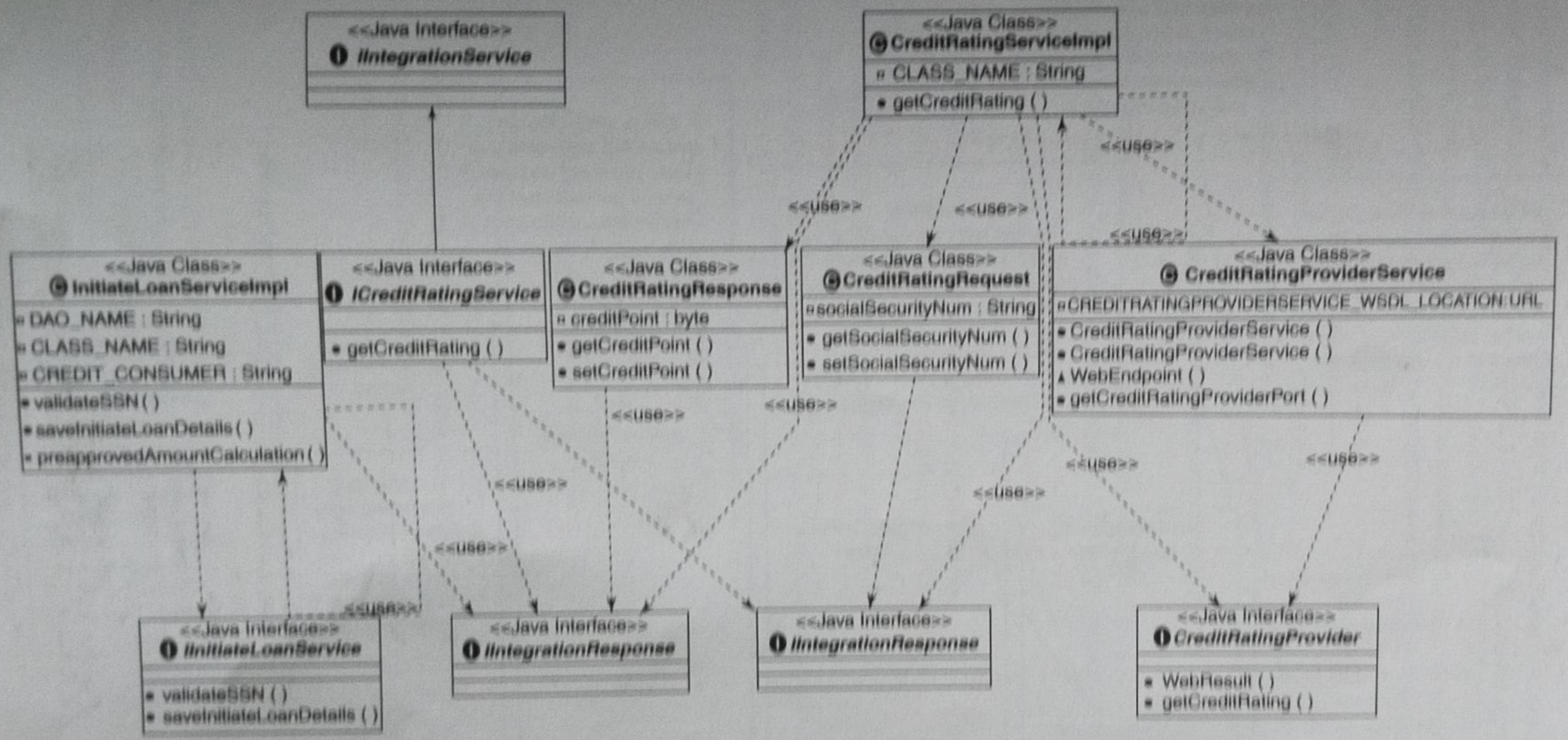


Figure 4-22 Integration layer—application framework and application components.

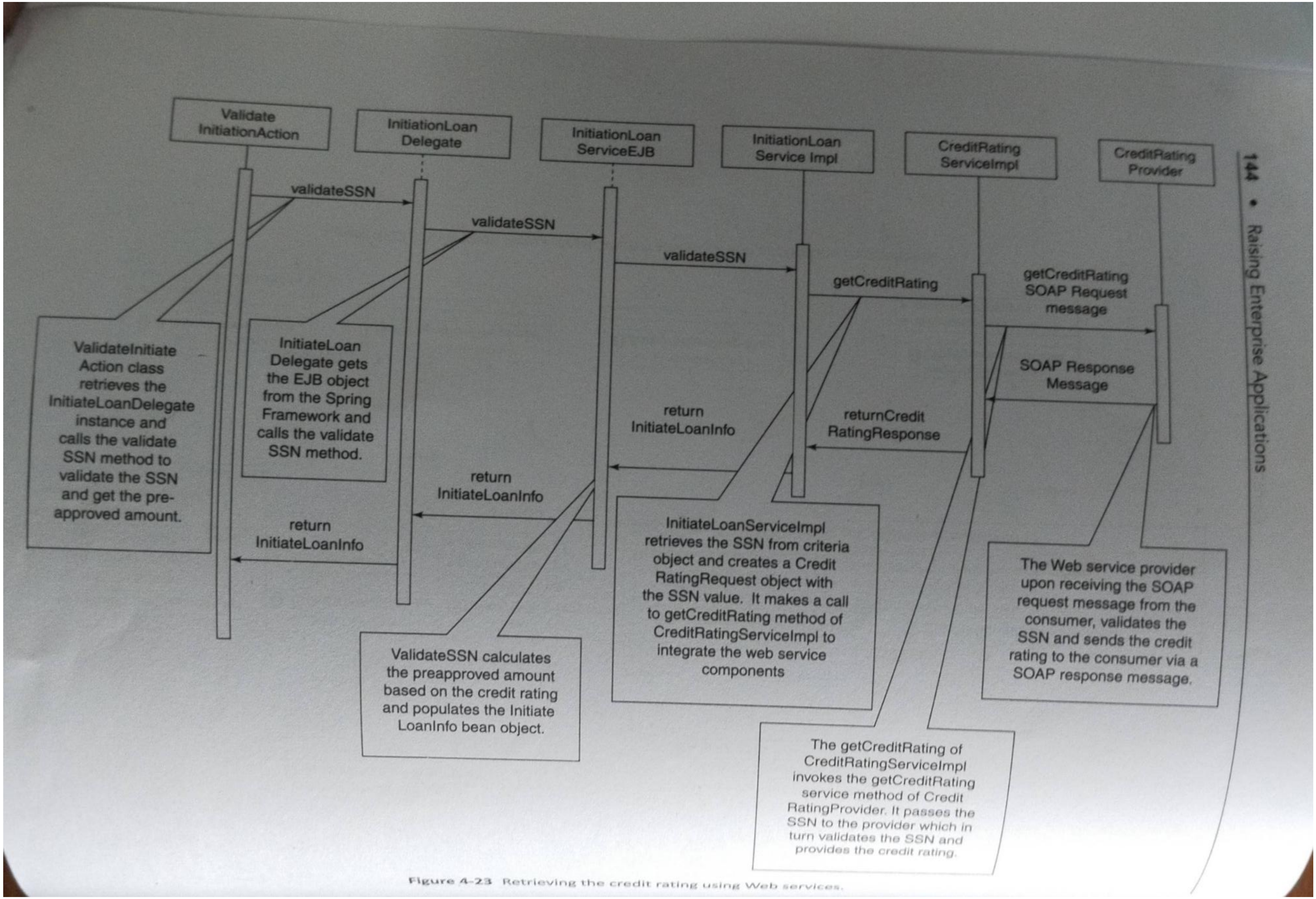


Figure 4-23 Retrieving the credit rating using Web services.

The `InitiateLoanServiceImpl` business service implementation class invokes the Web service consumer component `CreditRatingServiceImpl` to validate the SSN and retrieve the credit rating of the customer. The `validateSSN(InitiateLoanInfo criteria)` of `InitiateLoanServiceImpl` class instantiates the `CreditRatingRequest` object, which has the properties (the applicant's SSN, in this case) to be sent to the Web service consumer, as shown in Code Listing 4.14.

```
InitiateLoanInfo loanInfo = criteria;
CreditRatingRequest request = new CreditRatingRequest();
request.setSocialSecurityNum(loanInfo.getSsn());
```

Code Listing 4.14 Request for SSN.

The method then invokes the `getCreditRating` method of the `CreditRatingServiceImpl` Web consumer class passing the "request" as a parameter. The code snippet for the same is shown in Code Listing 4.15.

```
ICreditRatingService consumer = null;
      try {
            consumer = (CreditRatingServiceImpl)
            ServiceLocator.getInstance().
getService(CREDIT_CONSUMER);
      } catch(ServiceNotAvailableException e) {
            logMessage = new
LogMessage(LoggingManager.Error,CLASS_NAME,"InitiateLoanInfo","Credit
Rating Consumer Exception"+e);
            LoggingManager.log(logMessage);
            throw new BaseRuntimeException(e);
         }

      …
         IIntegrationResponse response = null;
         try {
         response = consumer.getCreditRating(request);
         catch(InvalidSSNException ex){
         logMessage .setMessage("Exception Occurred due to Invalid SSN"+ex);
         LoggingManager.log(logMessage);
         throw ex;
         }
```

Code Listing 4.15 Passing the request as a parameter.

The `getCreditRating(IIntegrationRequest request)` of `CreditRating ServiceImpl` class retrieves the SSN, as shown in Code Listing 4.16.

```
CreditRatingRequest creditRequest = (CreditRatingRequest)request;
String ssn = creditRequest.getSocialSecurityNum();
```

Code Listing 4.16 Retrieval of SSN.

The `getCreditRating(IIntegrationRequest request)` invokes the Web service provider, as shown in Code Listing 4.17.

```
com.knowyourcustomer.CreditRatingProviderService service = new
      com.knowyourcustomer.CreditRatingProviderService();
com.knowyourcustomer.CreditRatingProvider port =
      service.getCreditRatingProviderPort();
```

Code Listing 4.17 Invoking the Web service provider.

The `getCreditRating (IIntegrationRequest request)` invokes the `getCredit Rating(String ssn)` Web method of the service provider. The provider returns the credit rating for the given SSN. If the SSN is found to be invalid, the provider throws the `InvalidSSNFault_ Exception` to the consumer. The `getCreditRating(IIntegrationRequest request)` method handles the exception and throws the `InvalidSSNException`. The code snippet for the same is shown in Code Listing 4.18.

```
try{
    creditRate = port.getCreditRating(ssn);
}
catch(InvalidSSNFault_Exception e){
InvalidSSNFault fault = e.getFaultInfo();
throw new InvalidSSNException(CLASS_NAME, "validateSSN", fault.
getErrorDescription(), null);
}
```

Code Listing 4.18 Verification of SSN.

The `getCreditRating(IIntegrationRequest request)` of `CreditRating ServiceImpl` populates the credit rate to the `CreditRatingResponse` and returns the same to the business service implementation class. The code snippet for the same is shown in Code Listing 4.19.

```
CreditRatingResponse creditResponse = new CreditRatingResponse();
creditResponse. setCreditPoint (creditRate);
return creditResponse;
```

Code Listing 4.19 Populating the response object.

The `validateSSN(InitiateLoanInfo criteria)` of `InitiateLoanServiceImpl` business service implementation class handles the response and retrieves the credit rating. The method invokes the private `preapprovedAmountCalculation(creditRate)` to determine the amount the customer is eligible for, as depicted in Code Listing 4.20.

```
CreditRatingResponse creditResponse = (CreditRatingResponse)response;
```

```
double preapprovedAmount = preapprovedAmountCalculation(creditResponse.
  getCreditPoint ());
loanInfo.setPreapprovedamount(preapprovedAmount);
```

Code Listing 4.20 Retrieval of pre-approved amount.

The method then populates the preapproved loan amount in `loanInfo` and returns the same which will be rendered in the `InitiateForm.jsp`.

So far, we have explored the construction in a layer-by-layer manner—both application framework components and the application components. In the course of this discussion, you have noticed that how an individual component interacts with other components—within the layer and outside the layer. You have also noticed that how these interactions—both static and dynamic—are captured using class diagrams, sequence diagrams and software construction maps.

As mentioned earlier, coding and integrating these units of code to build the system is the core of the construction phase. Other key construction phase activities that we will explore in the following sections are code review, unit testing, build process, static code analysis and dynamic code analysis.

4.4 Code Review

The review process plays a pivotal role in measuring the degree of compliance with the specifications provided, and verifying the quality of artifacts produced during development. The code artifacts are the primary target of review in the construction phase.

4.4.1 Objectives

The objectives of code review are manifold—completeness, correctness, consistency, logic, maintainability, traceability and robustness of code, as represented in Figure 4-24.

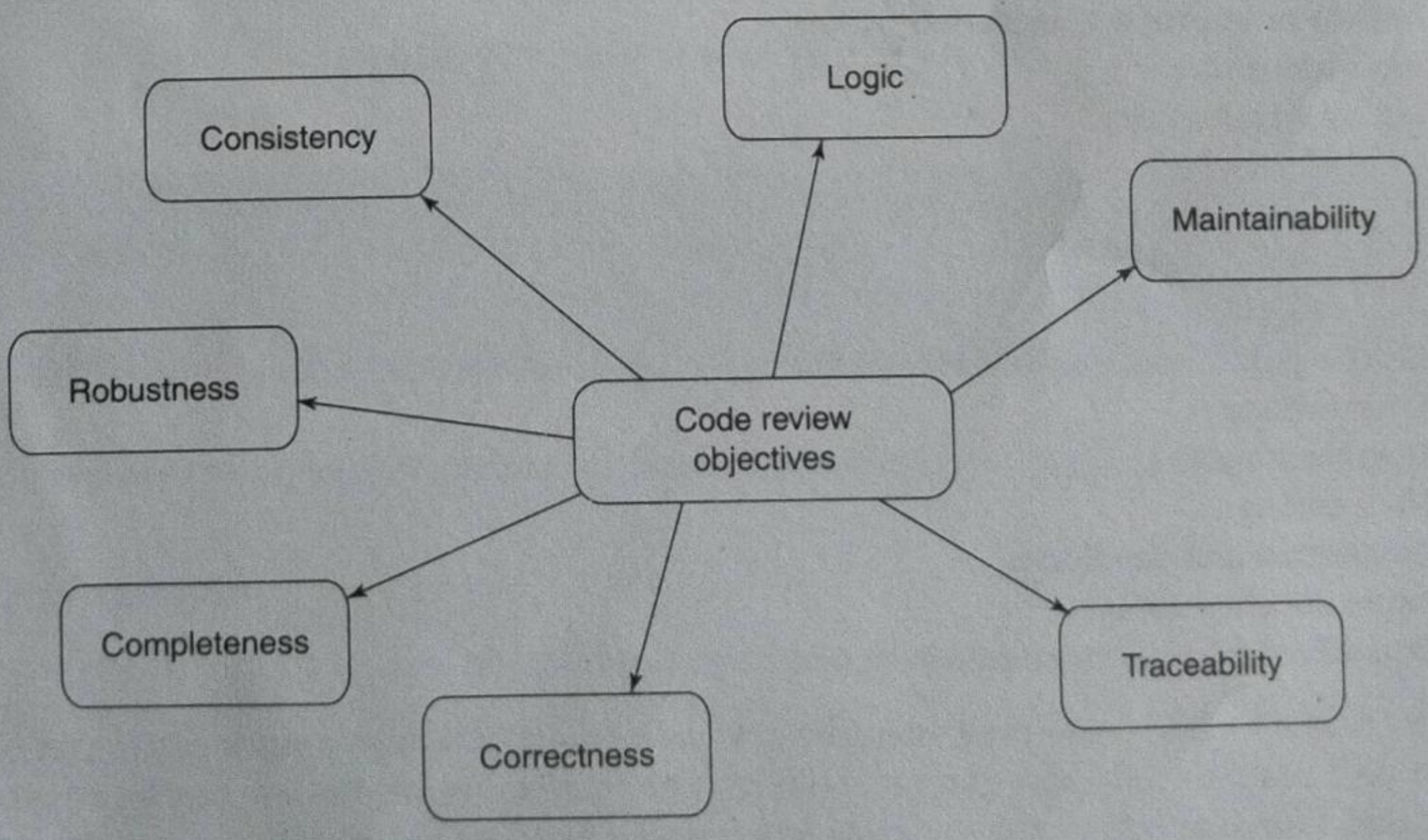


Figure 4-24 Code review objectives.

We outline these code review aspects below:

- *Completeness* review ensures that code is in line with the design of a software component and meets all key requirements.
- *Correctness* review typically vouches for conformance to coding best practices such as use of language-specific idioms, avoiding hard coding and eliminating unused variables.
- *Consistency* review ensures uniformity in coding, commenting, error and exception handling.
- Review of code from *logical* perspective is to ensure that the code results in expected behavior.
- *Maintainability* review is performed to ensure that the code is easily understood from the perspective of maintenance, i.e., readability of the code and is supported by adequate documentation. For example, use of descriptive identifiers enhances the readability significantly.
- *Traceability* review takes care of finding out whether any requisite functionality is missing, and whether the source code of the software unit can be traced back to its functional requirements and to corresponding design elements.
- Reviewing code from the *robustness* perspective is to ensure that the code is able to handle errors and unexpected events at runtime.

4.4.2 Process

Code review starts with identification of right set of reviewers and assigning ample time to perform it with the help of appropriate checklists and review guidelines. There are several things that a reviewer has to look for, which are captured in the review checklist. A few important ones are as follows:

- Code naming conventions
- Computational errors
- Comparison errors
- Control flow errors
- Infinite or improperly terminated loops
- Interface errors
- Input/output errors
- Unused variables
- Wrong initialization and default values
- Data reference errors
- Data declaration errors
- Unhandled exceptions and error conditions
- Error messages
- Help messages
- Hard coding
- Contention and deadlocks
- Inline documentation
- Beautification, e.g., indentation of code

Code reviews may be performed manually or in an automated fashion against certain set of guidelines and best practices. Reviews can be conducted in multiple forms. Review can be structured or unstructured, formal or informal, a self review or a peer review, or a code walkthrough driven by the developer or by the reviewer.

4.5 Static Code Analysis

Static code analysis is an activity of analyzing the code, in source or byte-code form, to identify various issues related to non-runtime aspects of the code. As depicted in Figure 4-25, static code analysis is a wide area that focuses on analysis of a codebase from several perspectives such as coding style, bug finding, finding security related vulnerabilities and code quality analysis. Static code analysis is an automated activity, which is performed using tools.

Tt Open source tools, such as Findbugs, CheckStyle and PMD, are used for static code analysis. While these tools can also be configured through rules to identify certain types of security vulnerabilities, specialized tools such as Fortify 360 and CodeSecure are specifically intended for identifying a vast range of security vulnerabilities.

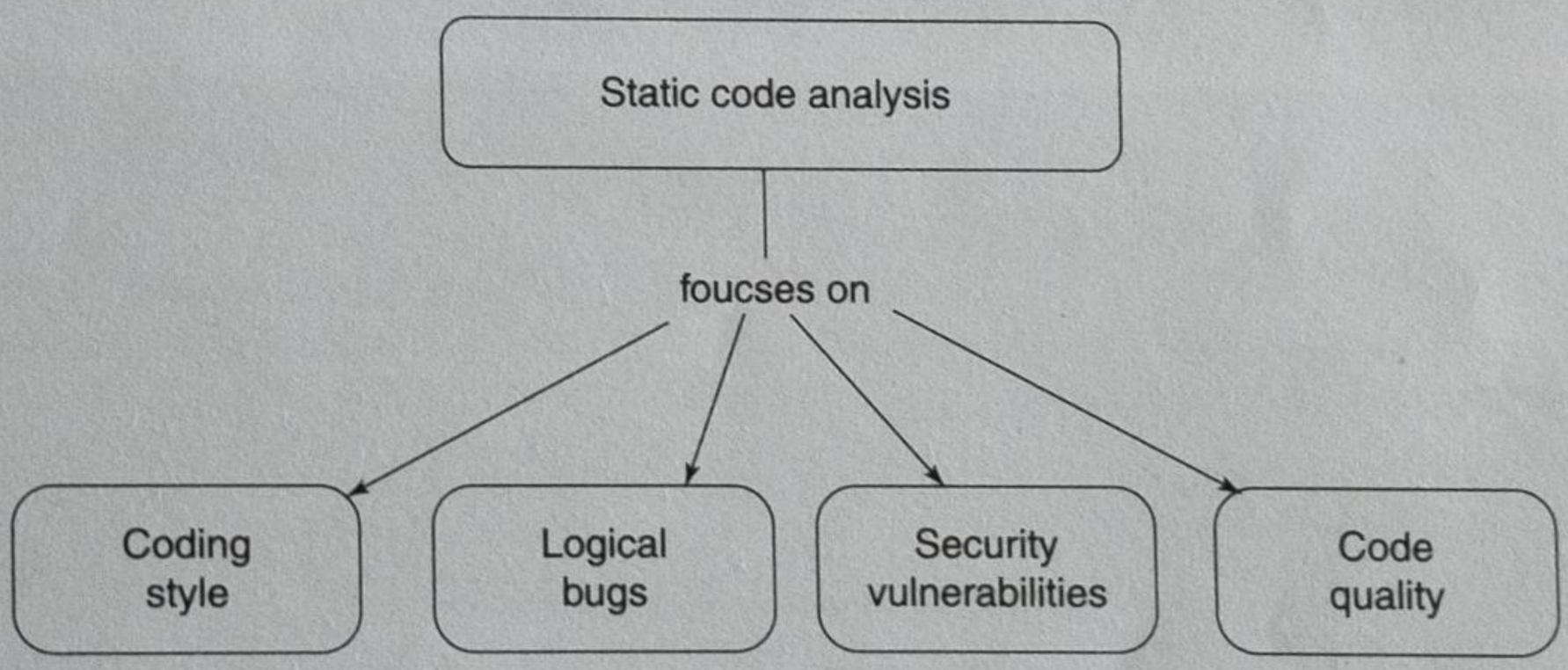


Figure 4-25 Static code analysis objectives.

In Two keywords, *false positives* and *false negatives*, are usually encountered druing automated static code analysis. False positive is a reported problem by a tool that is actually not a problem. False negative represents a problem which actually exists, but is not reported by a tool.

Let us now explore the primary areas that are focused on while performing the static code analysis.

4.5.1 Coding Style

Coding style check focuses on use of right language features such as use of apt control constructs, appropriate language idioms and right type of data structures. It also checks whether the code is correctly formatted for readability. It reports inconsistencies based on the preconfigured rules for style specification. Inconsistencies may be related to naming the identifiers, improper use of whitespace or the structural aspects of the program. Rules related to style are typically identified in the very beginning of the construction phase, and are enforced with the help of tools.

4.5.2 Logical Bugs

While finding *logical bugs* is essentially the endeavor of manual code review activity, a small subset of bugs may also be identified through encoded rules verified as part of static code analysis. These include identification of defects such as not releasing resources when they are no longer required, not using `StringBuffer` for string concatenation and unhandled potential `NullPointer` exceptions.

4.5.3 Security Vulnerabilities

The growing security threats are making the Internet facing enterprise applications more and more vulnerable, and necessitate a dire need to ensure secure code. Security code review and code analysis from a security perspective both aim to unearth security vulnerabilities in code.

There are myriads of security vulnerabilities which give hackers a chance to exploit them to abuse enterprise resources. Figure 4-26 depicts an application security vulnerability cloud. OWASP, an open community which focuses on improving the security of Web application software, has defined the top 10 security vulnerabilities that are prevalent in Web applications.[6]

Figure 4-26 Potential security vulnerabilities in an enterprise application.

Security is an essential component of enterprise applications, and starts from its inception. Developers and reviewers also have to follow various secure coding practices to ensure the development of secure code. Let us now look at a few of the vulnerabilities and secure coding practices to avoid them:

- **Cross-site scripting.** Popularly known as XSS, it is a vulnerability which allows an attacker to pass malicious scripts (potentially harmful Javascript code) through the form fields in

[6] You may refer to http://www.owasp.org/ to know more about OWASP and top security vulnerabilities.

the UI of an application. When a user requests for the HTML page containing the form, the values (malicious script) corresponding to these fields are passed back to the user's browser, the malicious script gets executed on the client machine. Code Listing 4.21 depicts an XSS prone code.

```
response.write(request.getParameter("customerName"));
```

Code Listing 4.21 XSS vulnerable code.

In the above code, the string value for the parameter customerName can potentially be replaced with a malicious script. A proper input validation should be done to prevent any malicious input from the front end.

To prevent propagation back to the user of malicious scripts that may have somehow made it into the application, the presentation logic on the server side that retrieves values of the form fields will scan for special characters such as "<" and ">" symbols, and replace them with their corresponding escape sequences "<" and ">". These characters are typically used in a HTML page to mark the beginning of a script. Typically, frameworks such as Struts and Spring have support for such encoding, which neutralize the potential malicious script. This technique is illustrated in Code Listing 4.22.

```
response.write( encoder(request.getParameter("customerName")));
```

Code Listing 4.22 Secure code—XSS fixed.

The method encoder takes care of reading the input string and converting the special characters to their corresponding escape sequences.

- **SQL injection.** *SQL injection* is one of the most dreaded application security vulnerabilities. This allows the attacker to change the logic of an underlying SQL query by injecting malicious SQL from the form fields of the front end of an application. This may result in unauthorized access or modification of precious data. To avoid SQL injection, input provided to form fields should be validated for size, type, format, range, etc. Use of bind variable also helps in avoiding this vulnerability. Code Listing 4.23 presents an example of code vulnerable to SQL injection.

```
PreparedStatement pstmt =
conn.prepareStatement("insert into CUSTOMER (CUSTNAME) values ('"
  +custname + "')");
```

Code Listing 4.23 Code vulnerable to SQL injection.

A hacker may insert malicious SQL code in the above code through the variable `custname`. This can be avoided by rewriting the code using bind variables as presented in Code Listing 4.24. The bind variables use the input value exclusively, and do not allow interpreting the input in any way.

```
PreparedStatement pstmt =
conn.prepareStatement ("insert into CUSTOMER (CUSTNAME) values (?)");
```

```
String name = request.getParameter("custname");
pstmt.setString (1, custname);
```

Code Listing 4.24 Secure code—SQL injection fixed.

- **Improper session handling.** Developers and reviewers have to take care of session related requirements and design such as whether the session time out has been set, whether the session of a user request has been validated before giving him/her access to a particular resource, etc.
- Authentication and authorization mechanisms should be checked for any inherent weakness.
- Internal error messages should not be leaked because of inefficient exception management.
- Reviewers and developers have to ensure that sensitive data like user credentials are not present in the code. Further, they should be stored or communicated in the encrypted form.

There are several other kinds of vulnerabilities that may exist in enterprise applications. The list is ever increasing, and organizations have to be on the guard against such dreaded vulnerabilities which may compromise application security.

The security code review can be automated or performed manually. Comprehensive manual code review can be quite exhausting, and requires a lot of time and patience from reviewer. Typically, for mundane tasks such as finding XSS or SQL injection pitfalls, automation is the best choice. Automated security code review is performed with the help of static code analyzers. These tools are quite effective and save time significantly. Both of these are "white box" approaches to security code review. Usually, a mix of manual and automated approach is used to analyze the code from the security perspective.

Security code review is a very important task during construction phase to ensure the detection and fixing security vulnerabilities, as early as possible. Deferring it to the end of software development life cycle (SDLC) may considerably increase the costs of remediation or may delay the application going live. This activity is different from "black box" security testing, more popularly known as *penetration testing*, which is typically performed just before the rollout of an application to the production environment. Penetration testing is done much after security code review in a typical SDLC.[7]

4.5.4 Code Quality

Code quality analysis is important to ensure that code being delivered is of the highest quality in terms of its modularity, extensibility, maintainability, reusability, testability and performance. Some of the key metrics that are used to measure the code quality are as follows:

- **Class size**. This metric is the measure of the size of a class in terms of lines of code. Big classes, say more than 1000 lines of code, have poor readability and maintainability. This may result in a higher cost of maintenance and introduction of more defects in the maintenance process.
- **Cyclomatic complexity**. This metric is the measure of number of linearly independent paths through a method. High value of this metric typically results in more testing effort to cover all the possible paths. Loops should be minimized, unnecessary conditions should be eliminated, and nested loops should be avoided to lower this metric. Typically, the maximum range of this metric should be between 8 and 12.

[7] You will learn more about penetration testing in Chapter 5.

- **Comments-to-code ratio**. This metric is the measure of ratio between comments to total lines of code. A lower comment ratio usually results in difficulty of understanding the code, and in turn, impacts the maintainability of code.
- **Number of attributes**. This metric is the measure of number of attributes in a class. A higher number may indicate that the class represents more than one entity, and may adversely impact its maintainability. If the functionality of a class can be logically broken down, then the class should be split into two or more classes. Otherwise, if the functionality of the class cannot be logically broken down, an abstraction level can be added.
- **Coupling between objects**. This metric represents the number of other classes to which a class is coupled. The count is based on number of reference types used in attribute declarations, formal parameters, local variables, return types, and throws declarations. More coupling between objects typically leads to less maintainability and more testing effort.

4.6 Build Process and Unit Testing

As in the case of construction of an enterprise application (where multiple teams are working on the code), it is a good practice to run unit test cases (along with the build process) to ensure the health of the increment at each step of construction phase. Continuous integration and unit testing are prominent practices of iterative software development to avoid last minute surprises. Let us first explore the build process, followed by unit testing of an enterprise application. Thereafter, you will explore how they work together during the construction of a typical enterprise application.

4.6.1 Build Process

Every enterprise application requires an accurate, automated and maintainable build process in order to effectively and efficiently build an application. The build process is not limited only to the compilation of source code, but also comprises of other key activities such as retrieving the appropriate set of code units from source code control systems, packaging application components and libraries, component specific compilation, and deploying the application into a development environment.

Though the build process can be executed manually, an automatic build is almost invariably preferred for building an enterprise application. A manual process may lead to confusions, errors, and a relatively longer duration to build applications. Automation usually helps in saving time and improving quality by standardizing the entire build process.

Tt Tools such as Apache Ant and Maven are used to automate the build process of Java based enterprise applications.

A build utility, such as Ant, defines build scripts using XML and performs common build operations such as accessing source code control repositories to extract source code units, definition of build *classpaths* and source compilation. Figure 4-27 presents a partial snapshot of a sample ant build script—`build.xml`. It represents the target for the final deployable unit of a JEE based application, i.e., an enterprise archive (EAR) file, which comprises of all required Java archive (JAR) and Web archive (WAR) files of an enterprise application. Similar build scripts are also required to create the individual JARs/WARs for application layers. Dependencies among application components are also taken care of by build scripts by appropriately defining the build dependencies in the XML script.[8]

[8] To know more about Apache Ant, you may refer to the Apache Jakarta Web site, http://ant.apache.org/

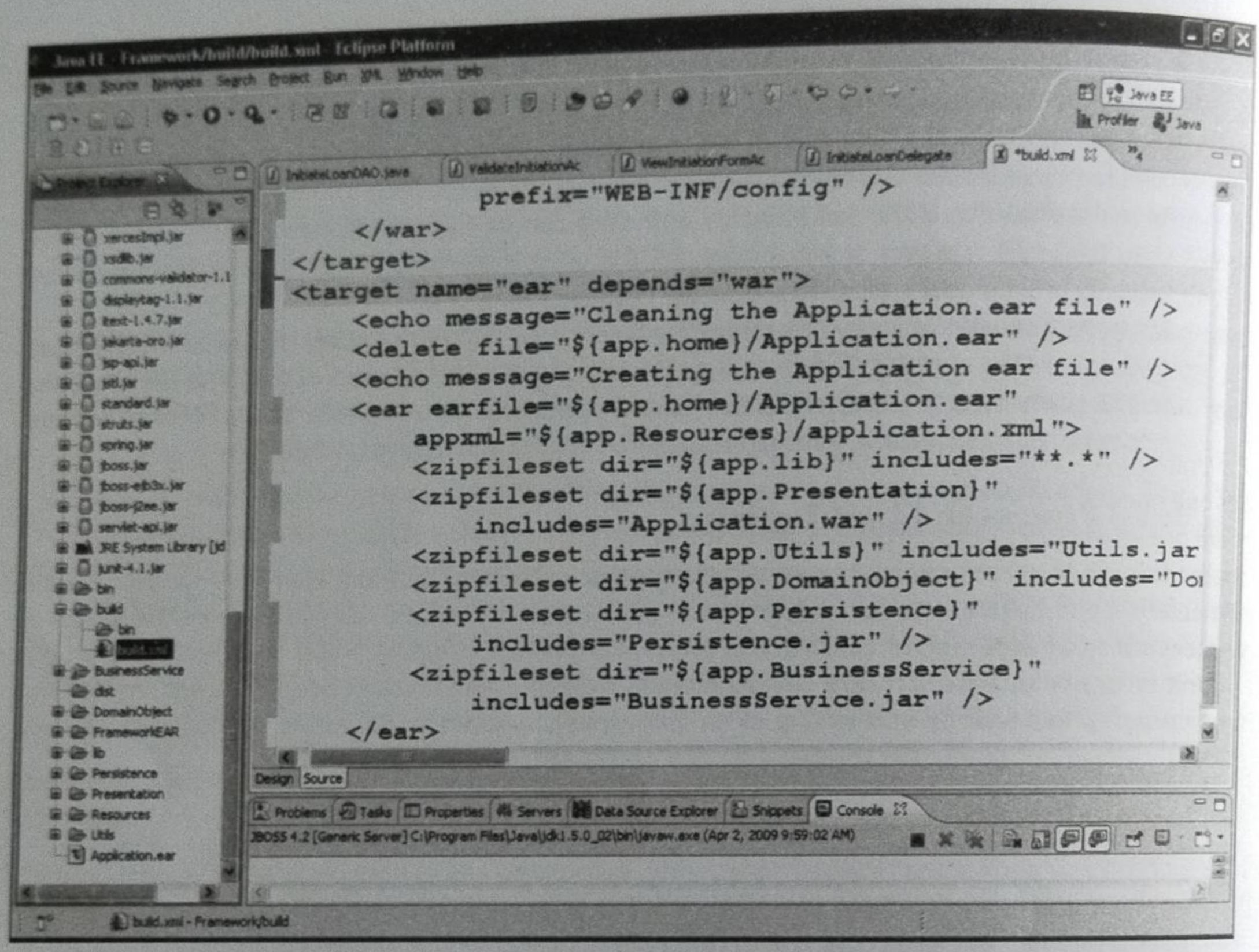


Figure 4-27 Sample `build.xml` (partial) file.

In Build scripts can be executed as commands on a need basis, or they can be scheduled based on a trigger condition or a designated time. The scheduled build processes are more appropriate in an open source development scenario, where several individuals or teams across geographies are checking in code into a central source code repository

The build process does not normally conclude after compiling and packaging the code. It is usually followed by running test cases and deploying the application. All this happens along with other peripheral work around this activity, which typically includes documentation activities such as generating Javadocs and release notes.

Tt For creating sophisticated build processes, the Jython scripting language can be used to extend Ant tasks. A right mix of Ant and Jython provides the power of Ant coupled with sophisticated scripting capabilities of Jython.

A build utility, such as Ant, can also run automated unit test cases as part of the build process, which helps in automating the entire process with minimal human intervention. Let us now understand unit testing—the first level of testing of an individual component/module of an application.

4.6.2 Unit Testing

After the development of a unit of code, it needs to be tested for various scenarios covering all the aspects of the functionality of the unit. Unit testing is the first formal level of testing in a SDLC, where individual units of software are tested, independent of other part(s) of an enterprise application. Units can be modules, procedures, methods or classes based on the paradigm of software development. As the code evolves, it is much easier and economical to find and eliminate the bugs early. Usually, it is the developer's responsibility to deliver unit-tested code.

Tt

Unit testing can be a manual process or is automated by using testing frameworks/tools like JUnit.[9]

Unit testing brings in a lot of confidence with respect to the test coverage of the application. But what does one need to test during unit testing? For example, in the case of presentation components, one has to ensure presence of all form fields, and correct translation and localization in case the application is internationalized. For the business and integration components, control flows and looping constructs in a software unit need to be verified to ensure the total coverage of the unit. Unit testing is not only about testing positive cases but also negative ones. Erroneous and exceptional situations also need to be unit tested.

Unit testing is an elaborate process which spans across the life cycle of application development. Three primary steps typically followed in the unit testing process are as follows:

1. **Test planning.** *Test planning* is the first step, which is typically carried out towards the end of the requirements engineering phase. It involves creation of test plans which document the steps to be followed in order to carry out tests. It also includes other aspects of test planning that typically include what needs to be tested, whose responsibility is to test, what the prerequisites are and assumptions for testing, which criteria pass or fail the test, etc.
2. **Test cases and test data preparation.** Test cases documents need to be tested in order to ensure a correctly functioning unit of an enterprise application. This typically comprises of testing the interfaces and properties exposed by the software unit along with the internal structure of the unit in terms of control flows and application logic.

Table 4-5 represents typical elements, which need to be captured in a test case. It may be noted that these test case elements are not specific only to unit testing, and they can be used for other types of testing as well.

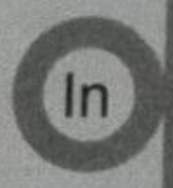
Table 4-5 Test case elements

Test case elements	Description of elements
Test case description	Describe the steps to execute the test case
Input test data	Data with which a given test case has to be executed
Expected result	The expected result after execution of the test case
Actual result	The actual result of execution of a test case
Pass/fail flag	Flag the outcome of a test case as "pass" or "fail"
Iteration number	Denote the iteration number of a test case

[9] For more details about JUnit, you can refer to http://www.junit.org/

Test data is the sample data with which test cases are executed to validate functionality. Test data should be selected very carefully to ensure complete coverage of functionality of the unit under test. Unit testing tools come with handy features such as support to input test data sources such as spreadsheet, providing flexibility during the execution of test cases.

3. **Execution of test cases and fixing the bugs.** Execution of test cases with selective test data is the third step that is performed after the construction of a unit of software. The actual results of test cases are compared with expected results, and test cases are designated as pass or fail. Bug fixing is done for all the failed test cases. The magnitude of change in the unit of software varies based on deviation between expected results and actual results. It may be as simple as fixing a simple logical error, or refactoring the code, or may lead to change in the underlying software design. This cycle is iterative in nature, and is repeated till a unit of software passes respective test cases and is flagged bug free.



It is worthwhile to mention that there is one more school of thought/paradigm that is related with testing — the *test driven* development. This is a technique to develop a software, which involves writing automated test cases before writing actual code. These test cases usually contain assertions that result in true or false to indicate the success or to failure of a test case. Typical Java IDEs like Eclipse comes with built in JUnit framework to facilitate the test driven development. To know more about test driven development you can refer http://www.testdriven.com.

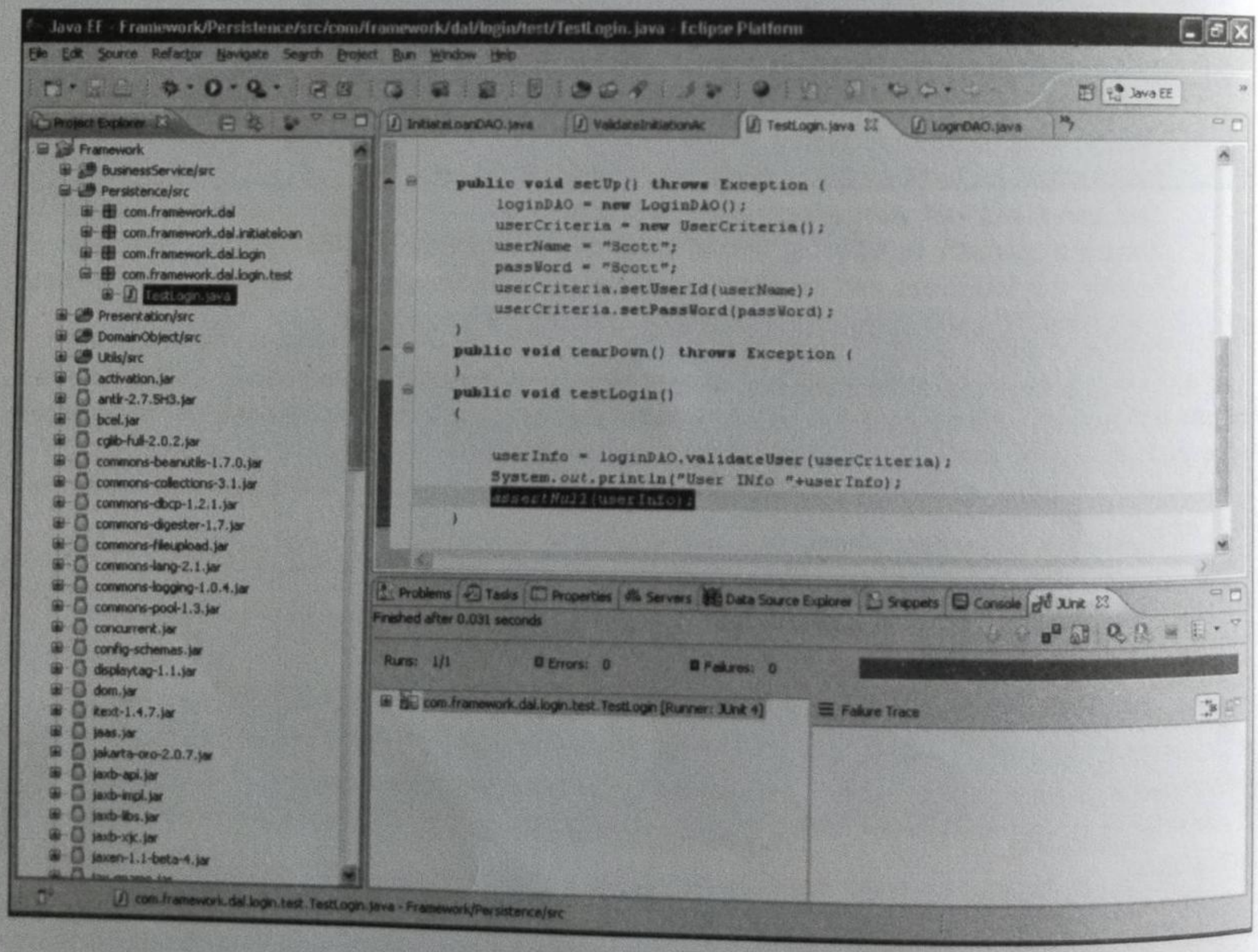


Figure 4-28 Unit test case run using JUnit—passed iteration.

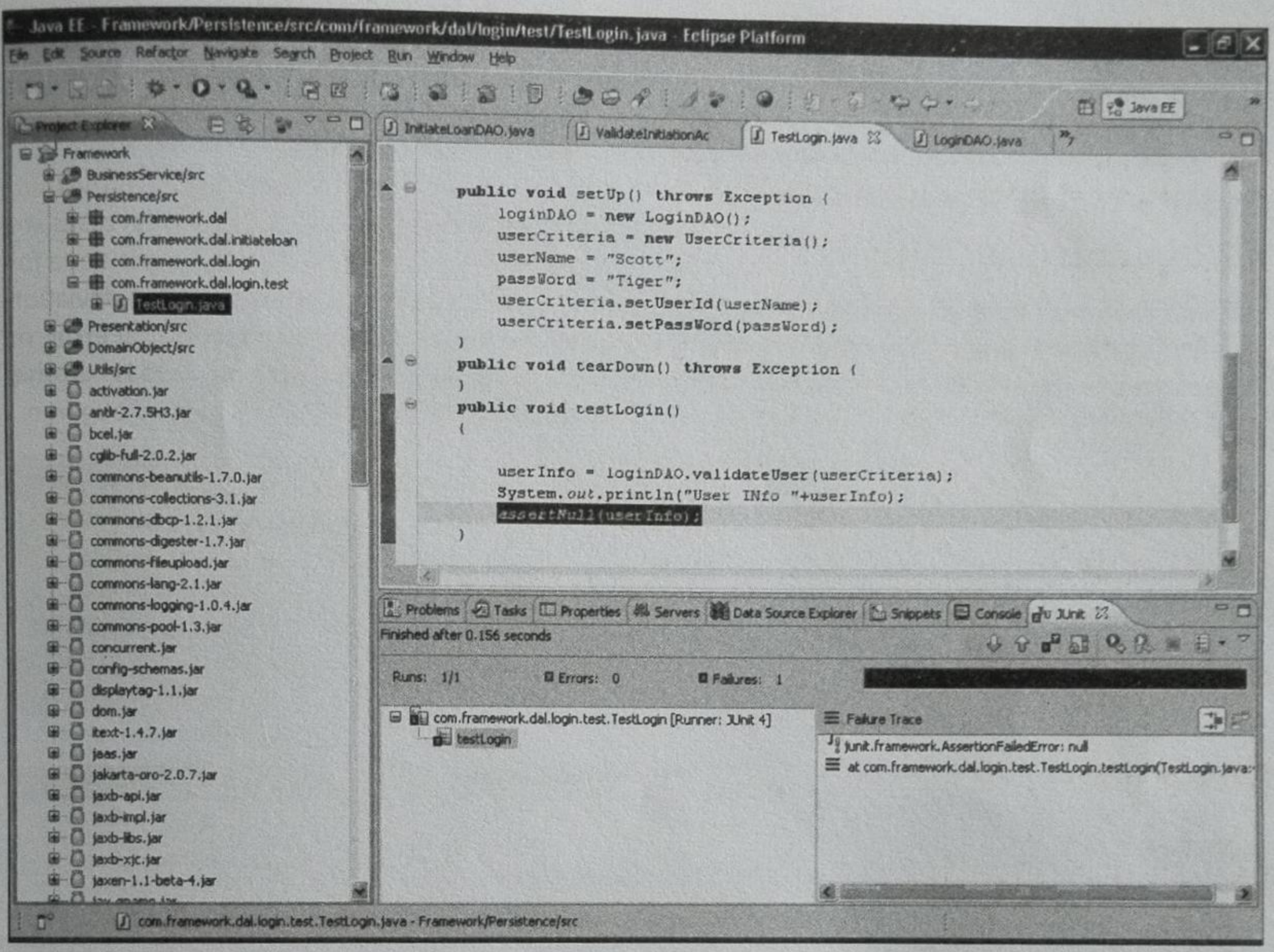


Figure 4-29 Unit test case run using JUnit—failed iteration.

Let us now see an example of automated unit testing using JUnit. The requirement is to test the `validateUser()` method. `validateUser()` is a method in the `loginDAO` class that accepts the `userCriteria` object comprising of username and password attributes. If the username and password combination is valid, then `validateUser()` returns an object of `userInfo` type, otherwise it returns null. Returning null means either the username or password, or both, are invalid.

To automate the testing of `validateUser()`, a unit test case is designed using JUnit. As shown in Figure 4-28, the test case is designed as the method `testLogin()`. This test case uses JUnit's `assertNull()` functionality to Figure out whether the test case succeeded (see Figure 4-28). A failed iteration of the same test case is depicted in Figure 4-29.

A misconception prevails that unit testing carried out during coding consumes a lot of time and can be deferred till integration testing. But it is not at all a good idea, as it induces delay in testing the units of software and proves to be more expensive and time consuming in the long run. Unit testing should be considered as one of the major action items in the construction phase to curb defects right in the early stages.

Ultimately, in whatever way build and unit testing steps are followed, automation is the key to standardize and speed up the entire process.

Tt

Tools like CruiseControl can be used to automate build and unit testing processes. Such tools are typically used in projects where there is a requirement to support continuous integrations of software units.

4.7 Dynamic Code Analysis

Dynamic code analysis enables a development team to obtain a holistic view of an enterprise application in the running state, contrary to the static code analysis that involves analyzing code without even compiling it. As illustrated in Figure 4-30, *code profiling* and *code coverage* are the focus areas of dynamic code analysis.

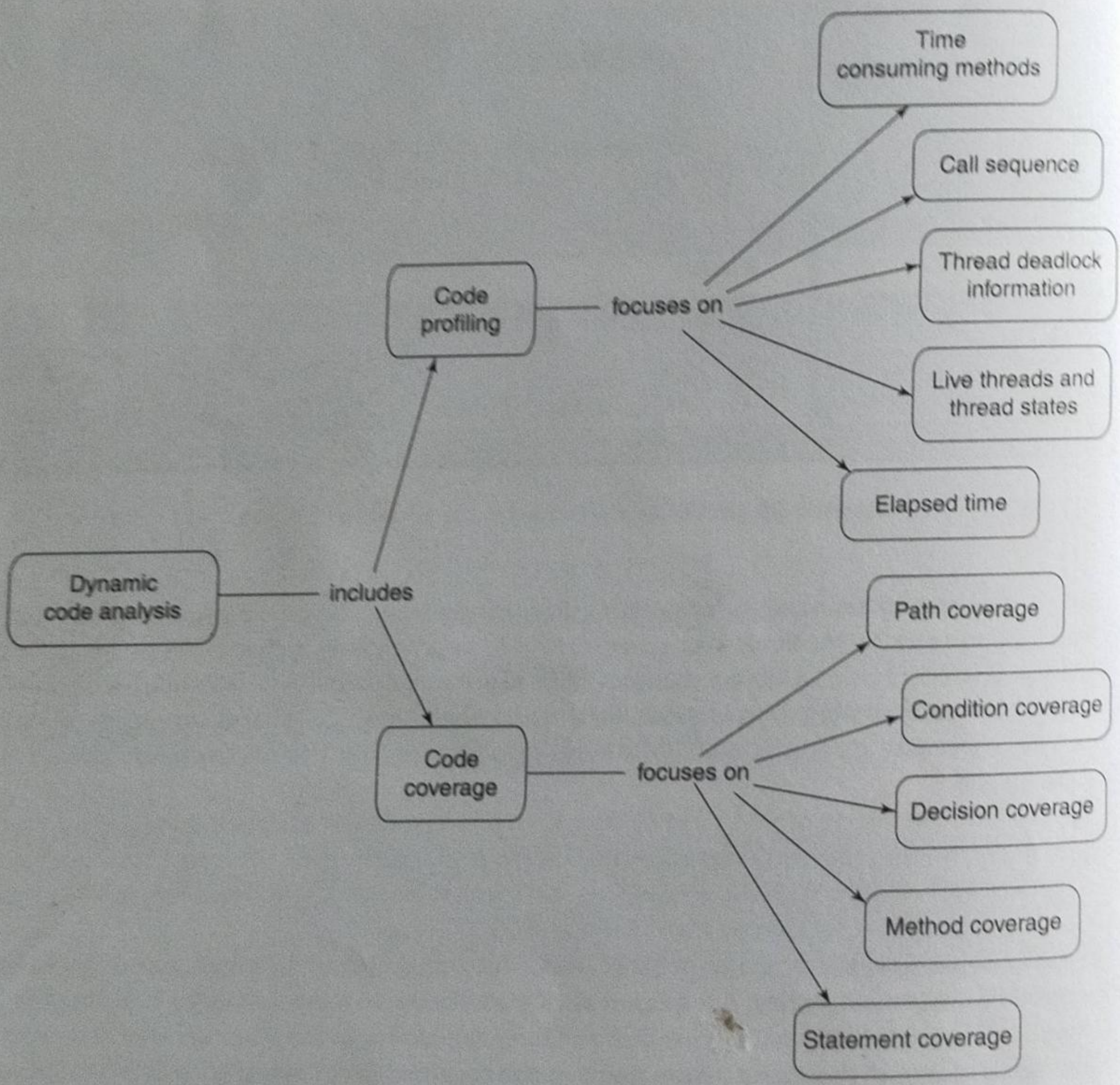


Figure 4-30 Dynamic code analysis objectives.

4.7.1 Code Profiling

Code profiling is one of the focus areas of dynamic code analysis, which is primarily concerned with the performance analysis of an application that enables the development team to quantify the performance aspects of an enterprise application. Code profiling captures various run-time attributes of the code, with which development team can identify the performance bottlenecks in an enterprise application. A few of the attributes, as shown in Figure 4-30, are described below:

- **Time consumed by methods.** This metric captures the time taken by the CPU to complete the execution of methods. It helps in identifying the most time-consuming methods and analysis of performance bottlenecks with respect to computational time.
- **Call sequence.** This attribute lists the actual sequence of method calls in a particular execution of the program. This helps in analysis of paths traversed during the code execution.

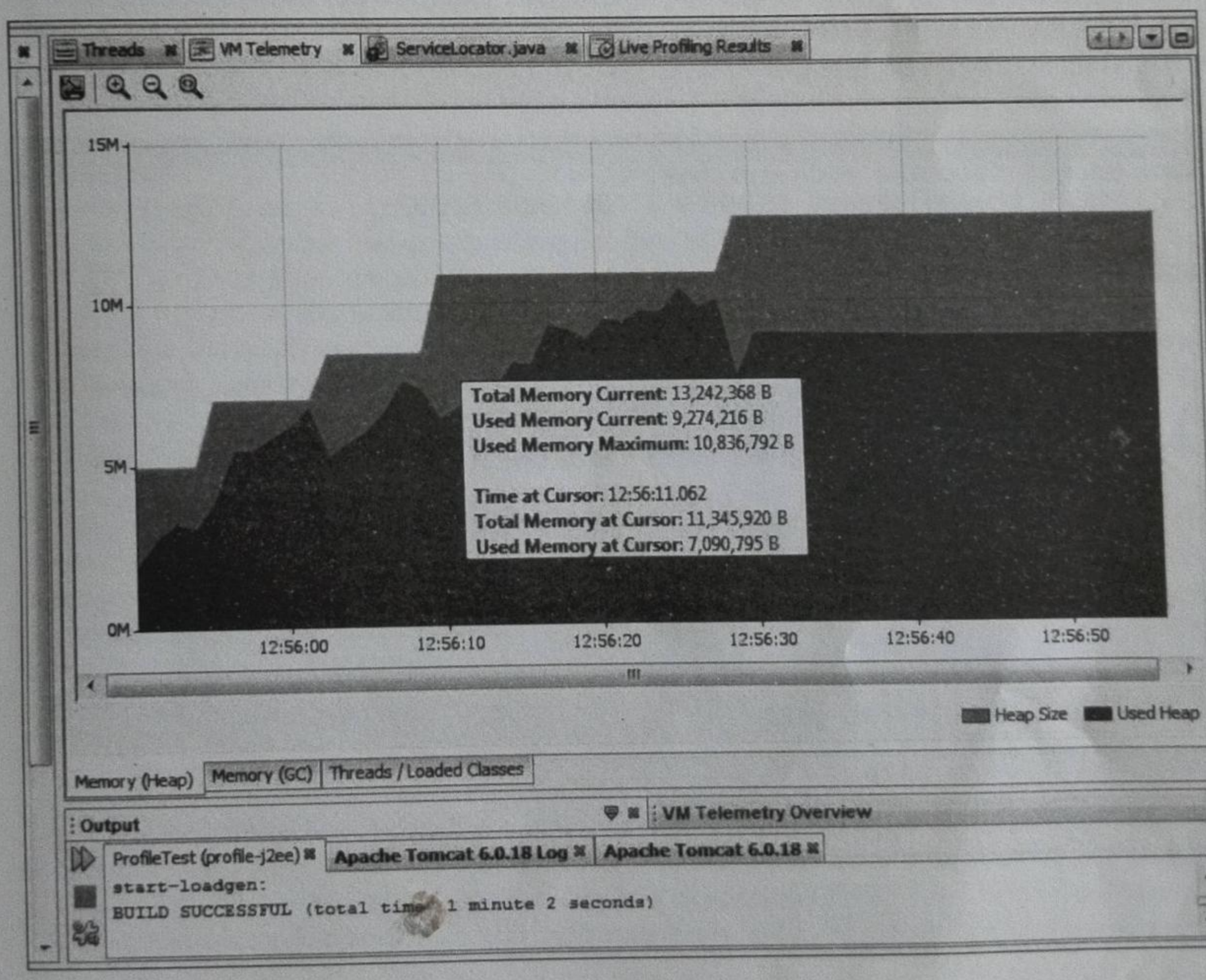


Figure 4-31 A snapshot from Java Profiler.

- **Thread deadlock information.** This data provides information on deadlocks due to resource contention or any other reason. This helps in identifying problematic sequences or scenarios which are potential sources of performance bottlenecks.
- **List of live threads and thread states.** This data provides information on live threads in a given analysis scenario, and the various states in which threads are running.
- **Elapsed time.** This metric, which includes sleep periods, program pauses and I/O waits, provides an indication of the impact on end user's experience. It helps in understanding if there are bottlenecks in execution of those result in perceptible time lags by end users.

Java Profiler tools typically provide a graphical "call graph" depicting the selected methods under analysis that provides an easy way to navigate through the code. This eases the task of analyzing the performance bottlenecks.

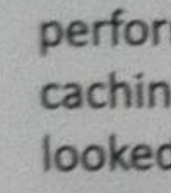
As shown in Figure 4-31, memory profiling provides information such as object instance counts on the heap, size of objects, etc. These statistics helps in finding issues such as memory leaks, loitering objects, or heap memory utilization that, in turn, provide ways to optimize memory usage.

Code profiling of an enterprise application is not just limited to the application code for business logic. The underlying database may also have several application components such as stored

```
C:\WINDOWS\system32\cmd.exe - sqlplus
SQL> SELECT custname FROM customer
  2  WHERE preapploanamt BETWEEN 370000 AND 500000;

10 rows selected.


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CUSTOMER'
   2    1     INDEX (RANGE SCAN) OF 'CUSTOMER_A_INDX_2' (NON-UNIQUE)



Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          4  consistent gets
          0  physical reads
          0  redo size
        589  bytes sent via SQL*Net to client
        503  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         10  rows processed
```

Figure 4-32 A database profiler's output.

procedures and triggers. These components are typically written using languages such as PL/SQL and T-SQL, which also may need to be analyzed to further fine tune the performance.



Commercial databases usually come with built-in profilers which can profile SQL queries and other database objects such as stored procedures and functions.

Figure 4-32 represents a sample output from the *Trace* utility of Oracle, which is used to profile a SQL query (depicted in the first two lines of the output). The output depicts the query execution plan that is a snapshot of how the database optimizer is running the query. It depicts which tables and indexes are scanned to retrieve the required data. It also provides statistics such as physical reads, database blocks affected and network roundtrips. Dynamic analysis greatly helps the development team identify and iron out any performance bottlenecks. The profiler utilities of databases typically have the capability to display metrics both in textual and graphical format.

What are the benefits for an application development team in capturing the code profiling metrics? These metrics provide clues to identify and plot the user behavior in terms of the frequently-used use cases. More tuning efforts can be focused on such use cases. These metrics also provide data on computational time, deadlocks and time consumed by the methods under analysis, which can be used for performance tuning. For example, if an operation to generate some data is expensive then the option of caching that data can be looked at. In some cases, the option of pre-computing some results can also be looked at to enhance performance. The metrics are not only used to find performance bottlenecks, but also used to benchmark the code. The next set of code releases can be compared with this benchmark to find further performance bottlenecks, if any.

One should remember that code profiling only provides objectivity to the performance analysis and helps in nailing down the performance bottlenecks. It does not add wings to your code. One has to take a cue from this analysis and spend effort to do performance tuning. It is also important to remember that performance tuning is not just about the code, but there are also network settings, operating system settings, application and database server configurations, JVM/container settings, etc., which, as a whole, determine the performance of an enterprise application. One has to consider a holistic view of all these elements and act upon the performance tuning of an enterprise application.

Even before all these tuning efforts, developers can really make a difference to the performance of an application by practicing a few basic good coding practices. Moving the code which is not required in the loop to outside the loop, avoiding method calls in the loop termination test, using primitive types in loop conditions and using "static final" for algebraic expressions so that the compiler can pre-compute the constant expressions, are a few examples of good coding practices to enhance the performance at the code level.[10]

4.7.2 Code Coverage

Code coverage is defined as the level, or percentage, to which the source code of an enterprise application is tested. This does not check the functionality of an enterprise application, but finds out what percentage of the codebase is not being executed as part of execution of test cases. This helps in ensuring that the test cases created for checking the functionality are adequate. Development teams may create

---

[10] To know more about for Java related performance issues and solutions, you may visit http://java.sun.com/docs/performance/

more test cases, if required, to increase the coverage level. The process of code coverage, as shown in Figure 4-30, primarily furnishes following metrics:

- **Statement coverage.** This measures the degree to which the statements of the application codebase under test have been covered during test execution.
- **Method coverage.** This measures the degree to which the functions or methods of an application codebase under test have been covered during test execution.
- **Decision coverage.** This measures the degree to which the Boolean conditions of all decision and looping constructs of an application codebase under test have been covered during test execution.
- **Condition coverage.** This measures the degree to which the Boolean conditions of all Boolean expressions of an application codebase under test have been covered during test execution. This is not similar to decision coverage, which is to measure whether all branches of decision and looping constructs are covered.
- **Path coverage.** This measures the degree to which the permutations of all possible logical flows in a given codebase under test have been covered during test execution.

The above list represents the primary metrics that capture code coverage. Apart from this, there exist several other metrics such as data flow coverage, race condition coverage that may also be captured. These metrics are very helpful to improve testing productivity and ensure that the application is tested for all conditions, branches, methods, etc.

Xp

At least 80% coverage of the underlying codebase is usually considered as a good level of code coverage.

Typically, both code profiling and code coverage tools use the same engine to perform dynamic code analysis, and hence often come bundled together. The enterprise application under analysis is run-on application servers, with which these tools are integrated and configured to perform dynamic code analysis.

Tt

- IBM Rational PurifyPlus suite comes with three tools, namely, IBM Rational Purify, IBM Rational Quantify and IBM Rational PureCoverage. Rational Purify can identify problems related to memory leaks, un-initialized memory, bugs in third-party libraries, etc. Rational Quantify has the ability to collect performance data from code, and identify performance bottlenecks. Rational PureCoverage indicates the amount of code tested and reports the level of coverage.
- Quest JProbe suite comes with JProbe Profiler, JProbe Memory Debugger and JProbe Coverage. JProbe Profiler is used to identify the performance problems, JProbe Memory Debugger identifies memory usage problems, and JProbe Coverage is used to identify the untested code.

Dynamic code analysis complements static code analysis, both of them together form a crucial part of analyzing software components developed during the construction phase. The testing and analysis activities should be done with a right mix of automated and manual modes to establish the best possible and optimized code in order to take it further in the next phase of raising enterprise applications—the testing phase that helps certify an enterprise application as "ready for service".

Summary

This chapter started with crucial activities related to construction readiness such as preparation of construction plan, definitions of package structures, establishing a configuration management mechanism, and eventually setting up a development environment for the construction team. It also introduced the concept and importance of software construction maps and presented some examples of them for better understanding of the concept. Here, you have learnt the construction of the application framework components and application components of the different layers of an enterprise application. The chapter also provided examples from the LoMS case study to illustrate the construction of infrastructure, presentation, business, data access and integration layer components. You have understood how to craft the infrastructure layer as part of the application architecture framework. We also explored class diagrams, sequence diagrams, software construction maps and relevant code across all layers of an enterprise application with specific examples from LoMS.

This chapter also introduced the code review process and static code analysis. In addition, coding style, logical bugs, application security vulnerabilities and code quality analysis have been introduced. This chapter also explained the concept of unit testing and build process, and how these processes work together. The concept of dynamic code analysis, which primarily involves code profiling and code coverage, is also discussed in this chapter.

Review Questions

1. List the prerequisites for starting the construction phase.
2. Explore the features of Java enterprise application development IDEs.
3. What is a software construction map?
4. What is aspect-oriented programming?
5. What are the typical components of a presentation layer package structure?
6. Describe one framework per layer used to build the application layers.
7. What is a dependency injection?
8. What are the objectives of code review?
9. What are the differences between a static code analysis and a dynamic code analysis?
10. Describe the three most dreaded application security vulnerabilities.
11. Explore JUnit framework.
12. Explore Jython and Ant scripts.
13. List the metrics of code coverage.
14. How does code profiling help in tuning the performance of an application?
15. List the tools used for static and dynamic code analysis.

Further Readings

Apache Ant: http://ant.apache.org/
Apache Log4J: http://logging.apache.org/log4j
Application security by OWASP: http://www.owasp.org/
AspectJ: http://www.eclipse.org/aspectj/
Dependency injection: http://www.martinfowler.com/articles/injection.html
EJB: http://java.sun.com/products/ejb/
IBM Rational PurifyPlus : http://www-01.ibm.com/software/awdtools/purifyplus/