II. ARCHITECTURAL AND DESIGN REPRESENTATION

1. Technical constraints

The following are constraints that have to be considered when designing the architecture for the system:

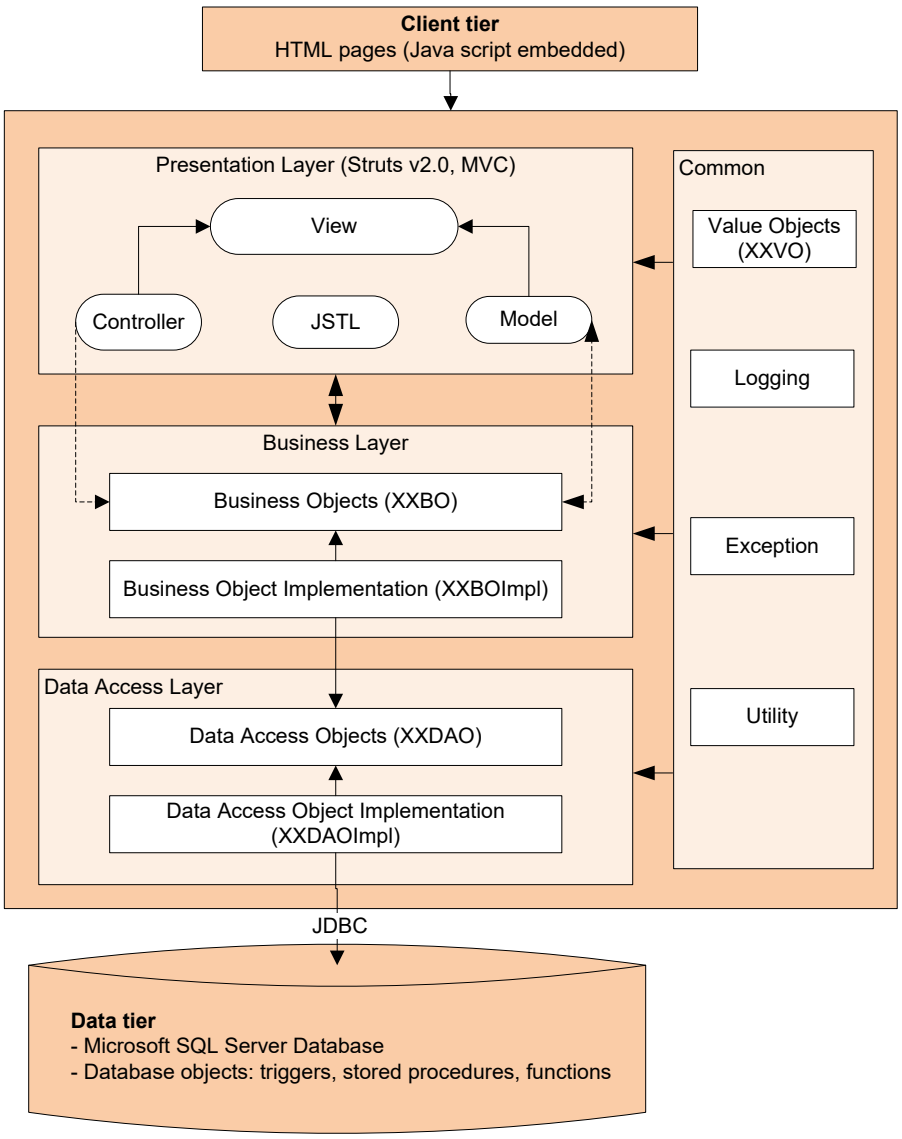
- System Framework: Use MVC2 (Struts v2.0)
- Client Framework: Use HTML, JavaScript
- Web support: IE 5.0 or later, Firefox 2.0 or later
- Database: Microsoft SQL Server
- Web Application server: Apache Tomcat

2. Logical view

This section describes the technologies & framework to be used in all the layers of this application

The architecture of the system is designed following the industry standard n-tier and J2EE approach architecture.

The diagram below shows the main logical layers of the architecture and how these logical layers interact with each other



a. Presentation Layer

This layer controls the display to the end user;

The Struts 2 framework is chosen as the implementation for the MVC (Model View Controller) model in this layer. The Struts framework is responsible for:

- Managing requests/responses from/to the clients.
- Controlling display of UI to the end user.
- Assembling a model that can be presented in a view.
- Performing UI validation.
- Providing a controller to delegate calls to business logic layer.
- Handling exceptions from other layers that throw exceptions to a Struts Action.

b. Business Layer

This layer manages the business processing rules and logic. It consists of a set of Business Objects to support business logic implementation of the system. The presence of this layer is to add flexibility between the presentation and the persistence layer so they do not directly communicate with each other.

- Handling application business logic and business validation.
- Managing transactions.
- Allowing interfaces for interaction with other layers.
- Managing dependencies between business level objects.
- Adding flexibility between the presentation and the persistence layer so they do not directly communicate with each other.
  - Exposing business services provided by this layer to the presentation layer.
  - Managing implementations from the business logic to the data access layer.

c. Data Access Layer

This layer manages access to persistent storage via JDBC. It manages reading, writing, updating, and deleting stored data.

The primary reason to separate data access layer from the Business Layer is to make sure the business implementation and the data access are loosely coupled. With this separation, it will be easier to switch data sources and share Data Access Objects (DAOs) between applications later, increasing the reusability, maintainability, and flexibility of the application.

d. Common classes

Value Objects: contain lightweight structures for related business information. A value object (VO – sometimes calls Transfer Object) is a lightweight, serializable object that structures groups of data items into a single logical construct. (Value objects always implement java.io.Serializable).

- A VO is intended to minimize network traffic between enterprise beans and their callers (because each argument passed initiates a network transmission).
- A VO is designed to improve the performance of enterprise beans by minimizing the number of method arguments, and thus network transmissions, needed to call them.
- In addition, VOs are useful in communication among all layers of the application

Utility Package: contains utility classes which provide common functionalities such as: string processing, date conversion, etc.

Logging Package:

This package contains classes for logging information to the log file for debugging/auditing purpose. The commons logging framework will be used with the Log4j implementation.

- The log4j component from Apache (http://logging.apache.org/log4j/docs/index.html) will be used to implement this package because it is possible to enable logging at runtime without modifying the application binary. The log4j package is designed so that these statements can remain in shipped code without incurring a heavy performance cost.
- Editing a configuration file, without touching the application binary, can control logging behavior.
- There are four levels of log: Error, Info, Debug, Warning.
  - **DEBUG**, this is the finest grain of information. The use of DEBUG statements is encouraged but production servers should not be left in DEBUG mode as this will slow down the performance of the servers and fill up the log files.

- **INFO**, this is less fine grained than the DEBUG level. Informational messages should highlight the progress of an application. E.g. INFO level could be used to highlight when a system batch is starting and ending. Note, INFO level should not be used in such a way that log files will expand rapidly.
- **WARN**, indicates a potentially harmful situation.
- **ERROR** designates error events. E.g. any exceptions within the application should be logged at this level.

In Test environment, all level of log can be turned on (by editing the configuration file) to provide the full error message. In production environment, log Error level can be turned on to provide only the friendly error message.

Exception Package:

This package will include all general exceptions that will typically used by more than one package. Where possible the Struts Exception Handler should be used. The try-catch clauses should be kept to a minimum. Where an exception is thrown the original exception should be included in the constructor.

There are two kinds of exception:

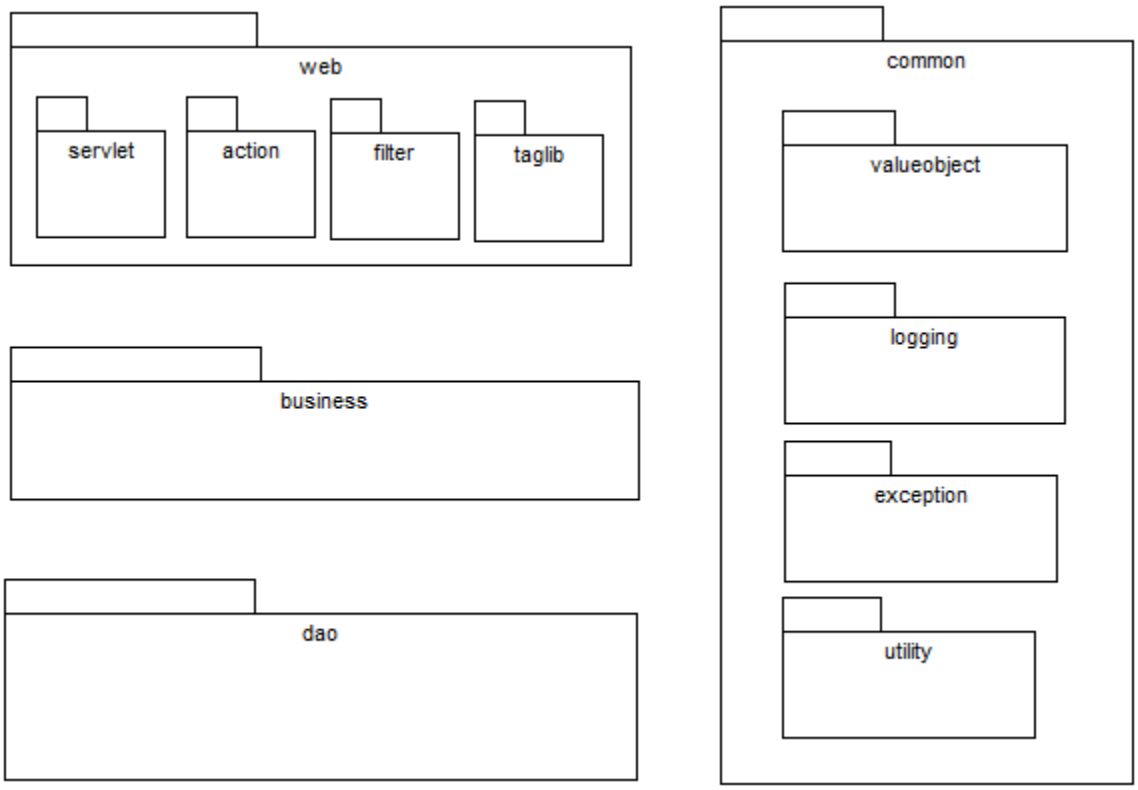
- *System Exception*: an exception will be thrown in case an error system occurs such as error connection to database, error reading file system…
- *Functional Exception*: an exception related to the business logic, for example the validation input data, authentication …

As a minimum the following details must always be captured and displayed in error messages:

- What happened when the error was identified. This should include what Class, Function, Method was being called and from where.
- The values of any Variables at the time the error was identified.
- Why the error occurred. Any system indication of why the error occurred should be reported

3. Architecturally significant design packages

a. Package structure



The package name for the application is mock.appcode (where: appcode is the code or short name of the relevant chosen application). Each component is divided into sub-packages as described below:

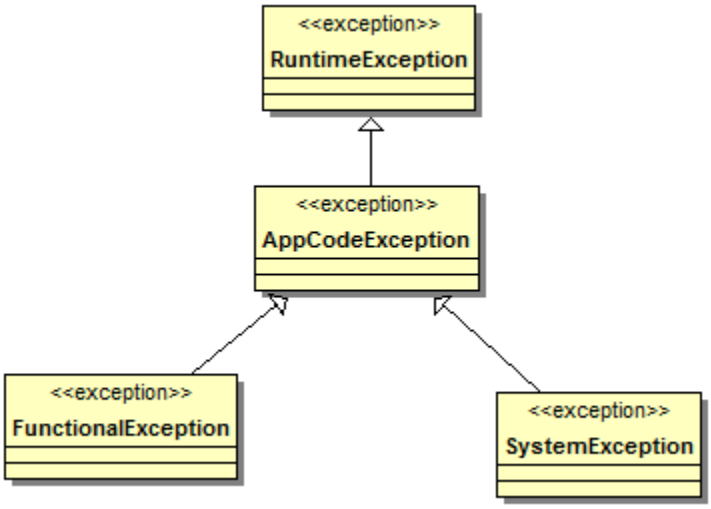
Package Name	Description	Naming Convention
mock.appcode.web.servlet	Contains all related servlet	xxServlet
mock.appcode.web.action	Contains all Struts action	xxAction
mock.appcode.web.filter	Contains filter classes	xxFilter
mock.appcode.web.taglib	Contains all taglibs used in the application	xxTag
mock.appcode.business	Contain all business interface & implementation	xxBO xxBOImpl
mock.appcode.dao.daointerface	Contain all data access interface & implementations	xxDAO xxDAOImpl
mock.appcode.common.valueobjects		xxVO
mock.appcode.common.utility	Contains utility classes	xxUtility
mock.appcode.common.exception	Contains exception classes	AppCodeException FunctionalException SystemException
mock.appcode.common.logging	Contains logging classes	

Among the packages of the system, the business package is not trivial and will be prepared by the trainees (via class diagrams, sequence diagrams, and pseudo codes).

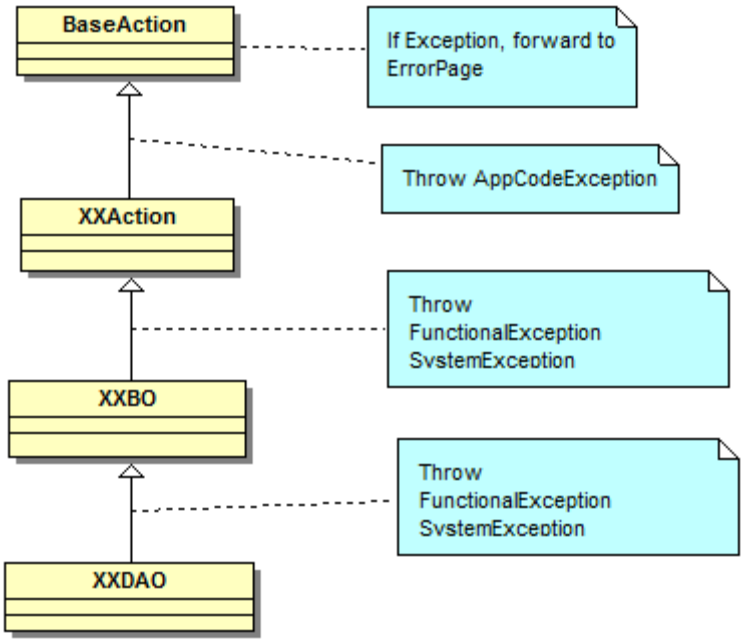
b. Error and exception handling

The application will use unchecked Exception mechanism that all exceptions thrown by AppCode will be extended from RuntimeException. The below class diagram shows the exception package

Class Diagram



Usage mechanism



FunctionalException: are logical exceptions, they are thrown in cases parameters transfers between method are inadequate, wrong type, missing data (data or business validation) etc.

SystemException: are RuntimeException, any unexpected errors like out of memory, database interruption etc.

c. Log, trace, and debug

The application uses log4j as standard logging package. We divide to 4 levels of logging:

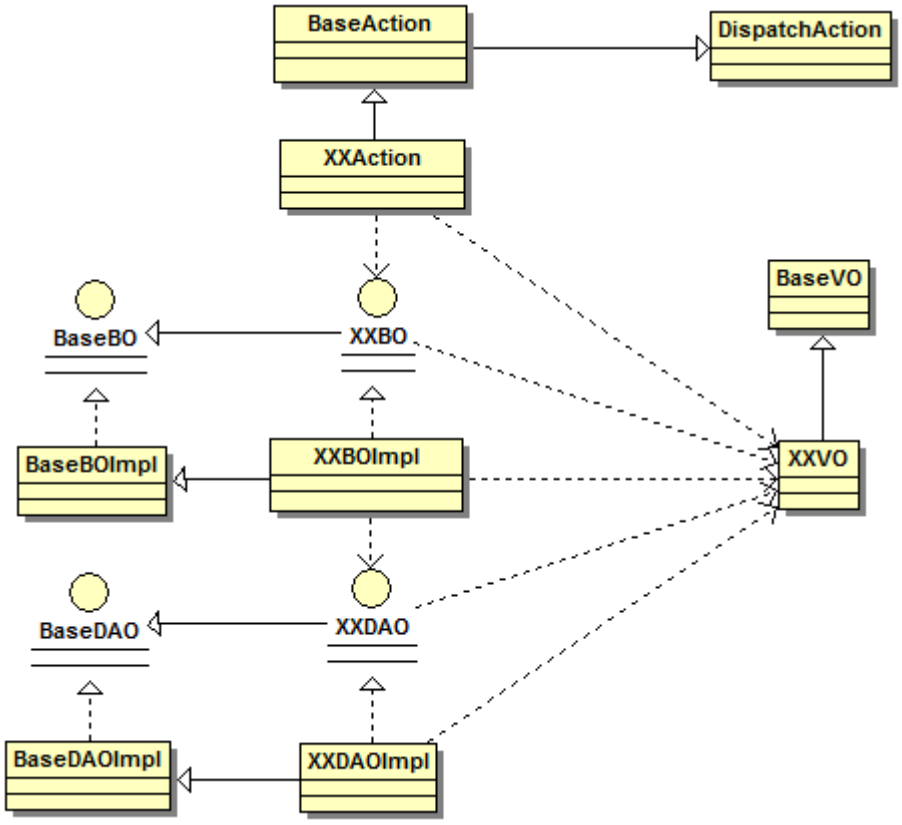
- **Info**: this is less fine grained than the DEBUG level. Informational messages should highlight the progress of an application. E.g. INFO level could be used to highlight when a system batch is starting and ending. Note, INFO level should not be used in such a way that log files will expand rapidly.
- **Warn**: indicates a potentially harmful situation
- **Debug**: this is the finest grain of information. The use of DEBUG statements is encouraged but production servers should not be left in DEBUG mode as this will slow down the performance of the servers and fill up the log files
- **Error**: designates error events. E.g. any exceptions within the application should be logged at this level.

d. Common flow

This section demonstrates the common code flow by class and sequence diagrams of one common use case.

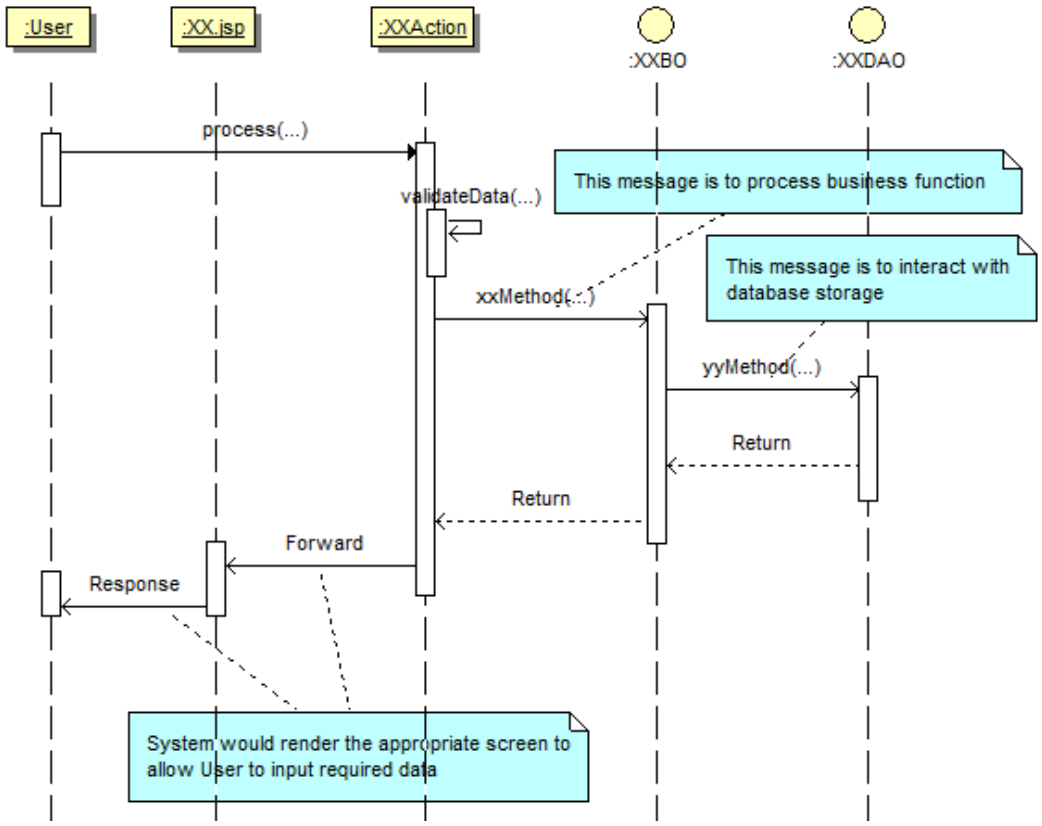
Class diagram

The typical class diagram for the use case is represented below



Sequence diagram

The typical sequence diagram for the use case is represented below



4. Software and deployment architecture

Application Server	
J2EE Application server	Apache Tomcat
Operating Systems	Windows Server
JDBC driver	TBD
Application Logging	Apache Jakarta Log4j (Deployed with the Application)
HTTP Server	
Web Server	Apache
Operating Systems	Windows Server
Database Server	
Operating Systems	Windows Server
DB Management System	Microsoft SQL Server