



F E D E R A L
S T U D E N T A I D

We Help Put America Through School

FSA Modernization Partner

NSLDS II Reengineering
Application Detailed Design:
Application Architecture

Version 1.1

November 26, 2002

Table of Contents

| | | |
|----------|--|-----------|
| 1 | GENERAL INFORMATION | 4 |
| 1.1 | OBJECTIVE..... | 4 |
| 1.2 | SCOPE..... | 4 |
| 1.3 | DESIGN LAYERS..... | 5 |
| 2 | EXECUTION ARCHITECTURE | 6 |
| 2.1 | IBM WEB SERVER AND APPLICATION SERVER..... | 9 |
| 2.1.1 | <i>Directory Structure</i> | 9 |
| 2.2 | IBM DB2 EEE..... | 10 |
| 2.3 | ORACLE..... | 10 |
| 2.4 | WEB APPLICATION SECURITY..... | 11 |
| 2.4.1 | <i>User Authentication</i> | 11 |
| 2.4.2 | <i>Web Authorization</i> | 11 |
| 2.4.3 | <i>Encryption</i> | 11 |
| 2.5 | REPORTING ENGINE INTEGRATION STRATEGY..... | 12 |
| 3 | BUSINESS LAYER DESIGN | 14 |
| 3.1 | CLASS DIAGRAM..... | 15 |
| 3.2 | OBJECT ACTIONS..... | 25 |
| 3.3 | SEQUENCE DIAGRAMS..... | 30 |
| 3.3.1 | <i>Aid - Update Loan Details</i> | 31 |
| 3.3.2 | <i>Aid - View Loan History</i> | 32 |
| 3.3.3 | <i>Aid - Update Overpayment Details</i> | 33 |
| 3.3.4 | <i>Aid - View Overpayment History</i> | 34 |
| 3.3.5 | <i>Enrollment - Add Reporting Schedule</i> | 35 |
| 3.3.6 | <i>Enrollment - View Summary</i> | 36 |
| 3.3.7 | <i>Logon</i> | 37 |
| 3.3.8 | <i>Organization - View Contact List</i> | 38 |
| 3.3.9 | <i>Organization - Delete Contact from List</i> | 39 |
| 3.3.10 | <i>Student Access – View Financial Aid Review</i> | 40 |
| 3.3.11 | <i>Support – View Contact List</i> | 41 |
| 3.3.12 | <i>Support – Add Contact to List</i> | 42 |
| 3.3.13 | <i>Transfer Monitor – View Transfer List</i> | 43 |
| 3.3.14 | <i>Transfer Monitor – Delete Transfer from List</i> | 44 |
| 3.3.15 | <i>Application Error</i> | 45 |
| 3.3.16 | <i>Java Exceptions</i> | 46 |
| 3.3.17 | <i>Web Conversation Framework ActionForm errors</i> | 47 |
| 4 | ARCHITECTURE LAYER DESIGN | 48 |
| 4.1 | RCS WEB CONVERSATION FRAMEWORK..... | 48 |
| 4.1.1 | <i>Web Conversation implementation of MVC Design Pattern</i> | 48 |
| 4.1.2 | <i>Implementation Example</i> | 51 |
| 4.2 | RCS CONFIGURATION FRAMEWORK..... | 58 |
| 4.3 | RCS JSP CUSTOM TAG LIBRARY FRAMEWORK..... | 59 |
| 4.4 | RCS EXCEPTION HANDLING FRAMEWORK..... | 60 |

| | | |
|-----------|--|-----------|
| 4.4.1 | <i>NSLDS II Exceptions</i> | 60 |
| 4.4.2 | <i>Field-level Validation</i> | 61 |
| 4.4.3 | <i>Application and System Level Exceptions</i> | 62 |
| 4.4.4 | <i>Implementation Example</i> | 63 |
| 4.4.5 | <i>JSP Error Pages</i> | 66 |
| 4.5 | RCS LOGGING FRAMEWORK..... | 67 |
| 4.5.1 | <i>Logging Usage Standards</i> | 68 |
| 4.5.1.1 | DEBUG | 68 |
| 4.5.1.2 | INFO | 69 |
| 4.5.1.3 | ERROR..... | 70 |
| 4.5.1.4 | FATAL | 72 |
| 4.5.2 | <i>Logging Usage Summary</i> | 73 |
| 4.6 | RCS USER SESSION FRAMEWORK..... | 74 |
| 4.6.1 | <i>Session Framework Usage Guidelines</i> | 74 |
| 4.7 | CONTENT DEPLOYMENT (INTERWOVEN TEAMSITE) | 75 |
| 4.7.1 | <i>Content Deployment Approach</i> | 75 |
| 5 | DATA ACCESS LAYER | 76 |
| 5.1 | RCS PERSISTENCE FRAMEWORK | 76 |
| 6 | APPENDIX A - APPLICATION ARCHITECTURE QUESTIONNAIRE | 84 |
| 7 | APPENDIX B - OBJECT-DATA MAPPING MODEL | 85 |
| 8 | APPENDIX C - ITA CODING STANDARDS | 86 |
| 9 | APPENDIX D - ITA BEST PRACTICES GUIDE | 87 |
| 10 | APPENDIX E - CODING STANDARDS REVIEW CHECKLIST | 88 |
| 10.1 | INTRODUCTION..... | 88 |
| 10.1.1 | <i>Purpose</i> | 88 |
| 10.1.2 | <i>References</i> | 88 |
| 10.1.3 | <i>Reviewer Information</i> | 88 |
| 10.2 | JAVA CODING STANDARDS | 88 |
| 10.2.1 | <i>Source Files</i> | 88 |
| 10.2.2 | <i>Code Layout</i> | 89 |
| 10.2.3 | <i>Naming Conventions</i> | 89 |
| 10.2.4 | <i>Programming Style</i> | 90 |
| 10.2.5 | <i>Comments</i> | 90 |
| 11 | APPENDIX F - NSLDSLOGGER PSEUDO CODE | 91 |

Document Control

| Version Number | Description | Release Date | Author |
|-----------------------|--|---------------------|---------------|
| 1.0 | Initial Release | 11/08/2002 | Amy Settle |
| 1.1 | Modifications to Production diagram to include CPS in VDC | 11/20/2002 | Terry Helwig |

1 General Information

1.1 Objective

This Application Architecture detailed design is intended to provide application developers with an understanding of the underlying architecture supporting the NSLDS II reengineering effort. This document will serve as the central architecture design guide to be used in conjunction with the NSLDS II Reengineering Screens Detailed Design.

1.2 Scope

This document focuses on the business, application, and data layers of the NSLDS II web application architecture. An overview of the execution architecture is provided in the beginning as an aid to better understand the interaction of the web application architecture with the other NSLDS II components. This document does not address external systems or interfaces to them. References to external systems can be found in the NSLDS II Reengineering Interface Detailed Design documentation.

The business layer section includes the class diagram, sequence diagrams, and a mapping of the Java objects and attributes to the tables and fields in the Enterprise Data Warehouse (EDW). The architecture layer section includes usage guidelines on how to incorporate the different Reusable Common Services (RCS) components provided by the Integrated Technical Architecture (ITA) initiative into the NSLDS II application architecture. The architecture layer interacts with the database through the data access layer, which is comprised of the ITA RCS Persistence framework.

1.3 Design Layers

A web-based application is comprised of several interconnected layers. This document will focus on the business, architecture, and data access layers. The GUI/presentation layer information is detailed in the NSLDS II Reengineering Screens Detailed Design documentation.

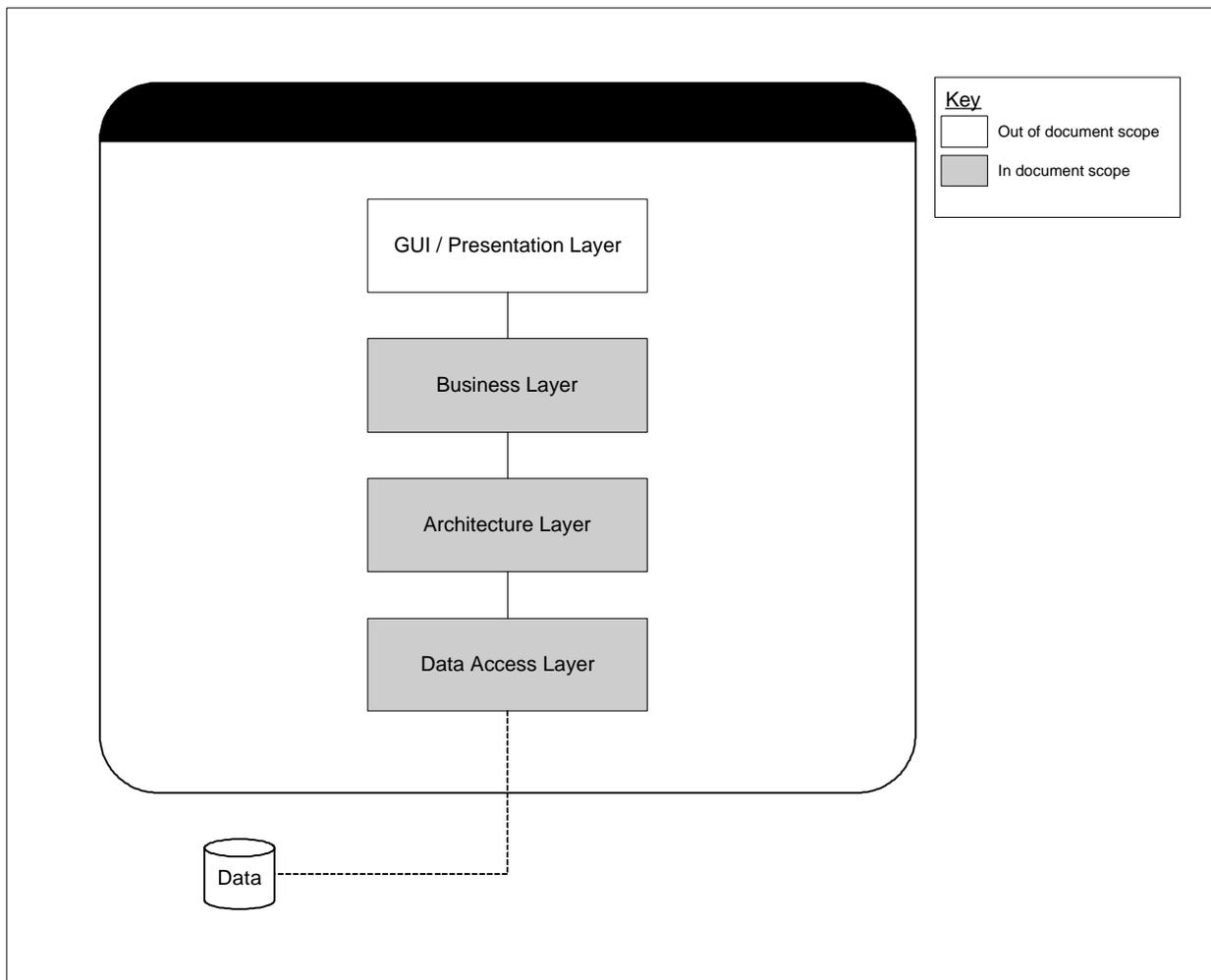


Figure 1, Design Layer

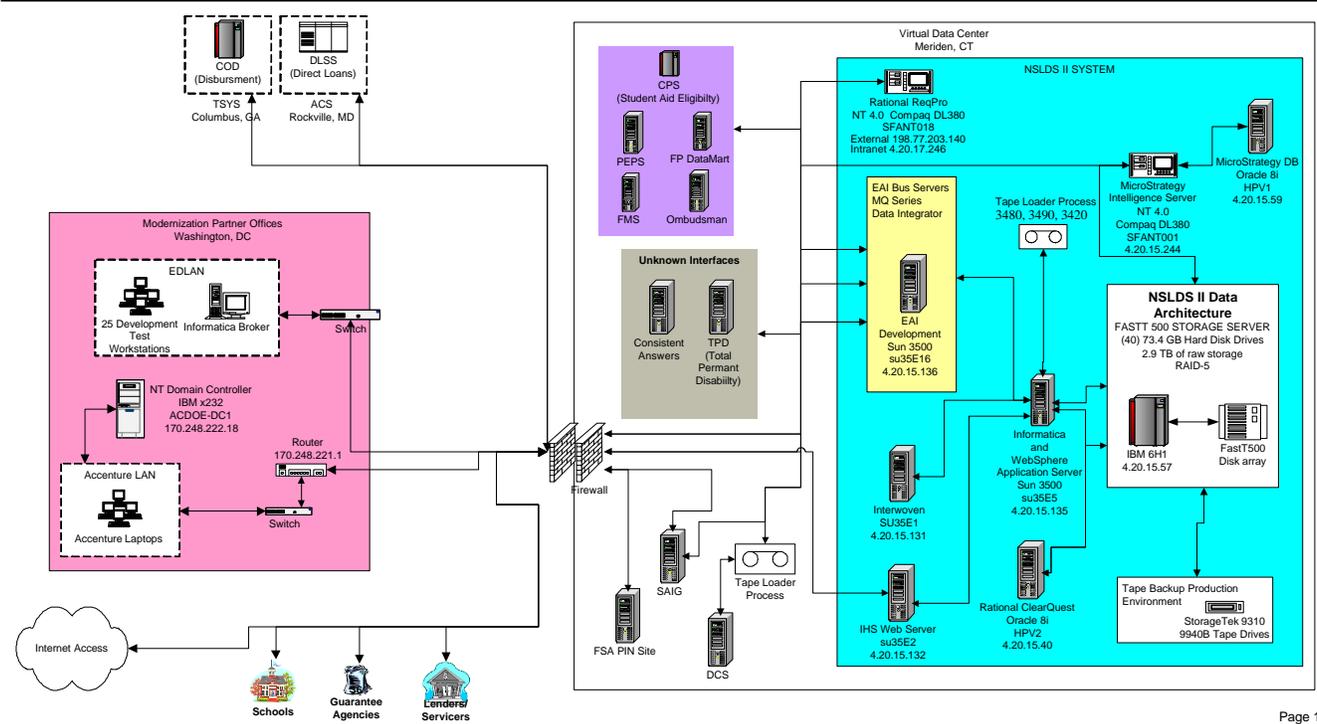
The presentation, business, and architecture layers are based on the Model-View-Controller (MVC) design pattern. This pattern creates a separation of control between different components to ease development and maintenance efforts. Application developers can concentrate on building the business objects and the web site designers can create web pages without having to understand the code needed to access the architecture and data layer. The MVC design pattern will be presented in more detail in the architecture layer section.

2 Execution Architecture

The execution architecture comprises of the internal systems that the core application architecture software components will interact with. It is the foundation of software services that the system relies on. The core environment components are noted here because all major application design decisions have a dependency on them. The diagrams below represent the development and production execution architectures.

NSLDS II Development Environment

Wednesday, November 20, 2002

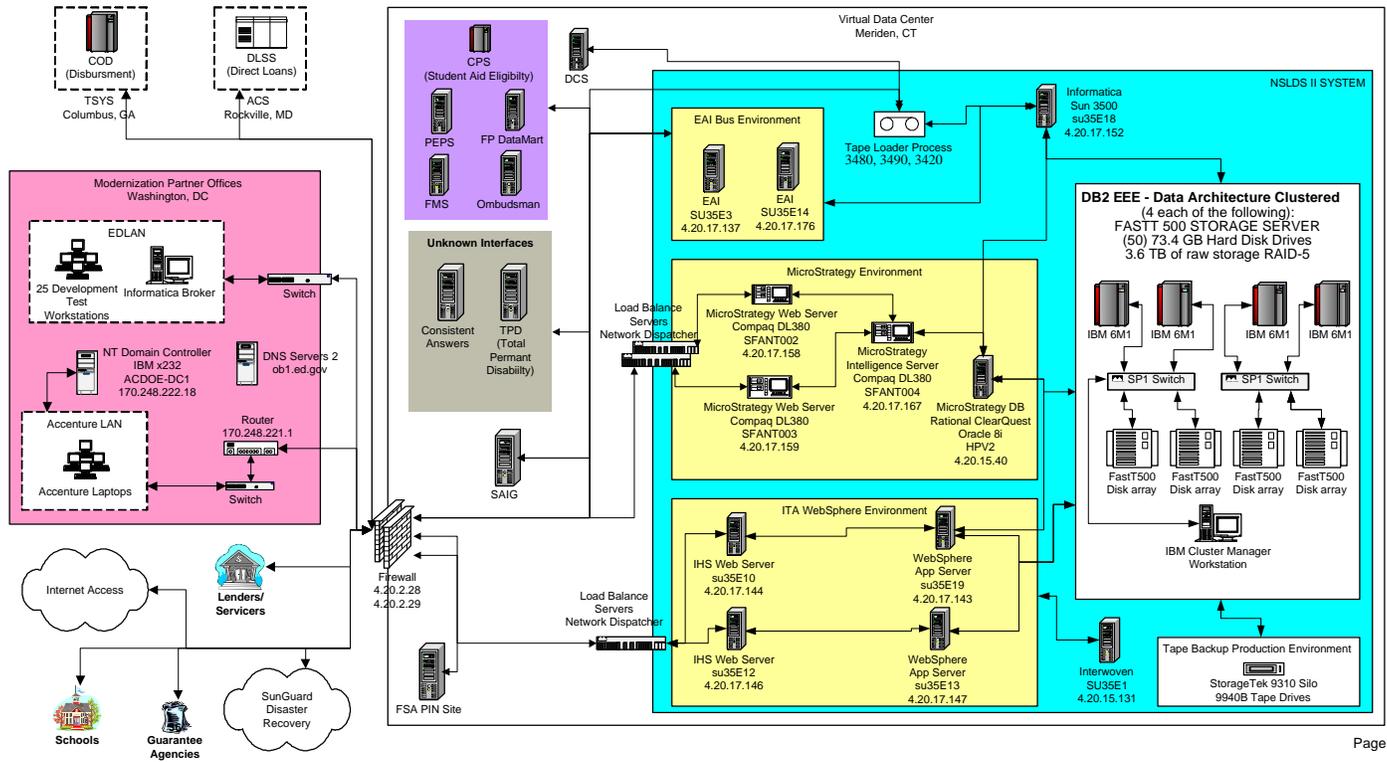


Page 1

Figure 2, NSLDS II Development Execution Architecture

NSLDS II Production Environment

Wednesday, November 20, 2002



Page 1

Figure 3, NSLDS II Production Execution Architecture

2.1 IBM Web Server and Application Server

The IBM Websphere Application Server (WAS v3.5) will be the deployment platform for the NSLDS II web application. It provides the Java 2 Enterprise Edition (J2EE) standards support required for the application architecture, as well as a scalable and robust run-time environment that will enable the application to meet its reliability and performance requirements. It is also the web application environment of choice for FSA.

J2EE standards that WAS support includes:

- JDK 1.2.2 compliance
- JSP 1.1 specification compliance
- Servlet 2.2 specification compliance
- JDBC 2.0 support (including connection pooling)

The IBM HTTP Server (IHS) web server provides for secure HTTP communications via Secure Sockets Layer (SSL) 3.0 and supports up to 128bit encryption.

2.1.1 Directory Structure

ITA provides four standard environments:

- Development (DEV) – used by the application developers for development and unit testing
- Test (TST) – used for component and integration testing
- Staging (STG) – used for performance testing and production readiness review
- Production (PROD) – the fully tested and released application

The tables below show the standard directory structure for ITA supported applications on the shared web server and application server. The development, test, and staging environments all run on the same machines and share a standard directory structure. The production environments are load balanced across multiple machines.

| Directory | Description |
|---|---|
| /www/dev ¹ /nsls/servlets | Location of servlets in the classpath. |
| /www/dev/nsls/web | Location of dynamic content document root and JSPs. |
| /www/dev/nsls/web/WEB-INF | Location of tag library descriptor files. |
| /www/dev/nsls/properties | Location of *.properties files. |
| /www/dev/nsls/jars | Location of jar files in classpath. |
| /opt/dev35 ² /WebSphere/AppServer/logs | Location of NSLDSstdout.log, NSLDSstderr.log, and NSLDS_System.log files. |

Table 1, Development Application Server (su35e5)

¹ This path is the path to the files in the development environment, to access files in staging or test, the path would be: /www/stg/nsls/ or /www/tst/nsls.

² This path is the path to the files in the development environment, to access files in staging or test, the path would be: /opt/stg35/ or /opt/tst35.

| Directory | Description |
|-----------------------|--|
| /www/dev/nslds/htdocs | Location of static content document root, location of HTML files, and PDF files. |

Table 2, Development Web Server (su35e2)

In production, the application will reside on the web servers su35e10 and su35e12, and on the application servers su35e9 and su35e13. The environment identifier will be removed from the directory structure information (e.g. /www/nslds/servlets).

The NSLDS II development web application can be accessed from the following path: http://dev.nslds.ed.gov:8531/NSLDSWebApp/*.jsp. To access other environments, replace dev with stg or tst.

2.2 IBM DB2 EEE

IBM DB2 EEE v7.2 will be the database platform for the NSLDS II web application. It will be the platform used for both the Enterprise Data Warehouse (EDW) and the Data Mart that MicroStrategy will be querying against for reporting purposes. The web application will interact with the EDW while MicroStrategy will query against the Data Mart. Please refer to the NSLDS II Reengineering Data Architecture Detailed Design for additional information.

WAS version 3.5.5 supports DB2 Enterprise Edition 7.2 with FixPaks 4, 5, 6, and 7. FixPak 6 and above is the recommended FixPak to resolve memory leak issues.

In order to configure WAS to access the DB2 databases, all access information (e.g. username, password, port number) must be provided to ITA in the Application Questionnaire. The completed questionnaire will be provided as Appendix A in this document.

2.3 Oracle

While the application data will be stored in the DB2 EDW and Data Mart, a shared Oracle instance provided by ITA will also be used to user store session information.

The session information will be stored in the *Sessions* table of the default Oracle database and will contain the session id and maximum inactive time defined for the application for that session.

| Name | Null | Type | Description |
|-----------------|----------|---------------|--|
| ID | NOT NULL | VARCHAR2(64) | Session ID |
| PROPID | NOT NULL | VARCHAR2(64) | Used by WAS |
| APPNAME | | VARCHAR2(64) | Web application |
| LISTENERCNT | | NUMBER(38) | Used by WAS |
| LASTACCESS | | NUMBER(38) | Last session access time |
| CREATIONTIME | | NUMBER(38) | Time session created |
| MAXINACTIVETIME | | NUMBER(38) | Maximum session inactive time |
| USERNAME | | VARCHAR2(256) | user name associated with the session |
| SMALL | | RAW(2000) | Stores session data less than 2000 bytes |

| Name | Null | Type | Description |
|--------|------|----------|--|
| MEDIUM | | LONG RAW | Stores session data between 2000 bytes and 2MB |
| LARGE | | RAW(1) | Not supported by Oracle |

Table 3, Sessions Table Description

2.4 Web Application Security

This section describes the security functionality of the NSLDS II web application, please reference the Data Architecture Detailed Design for the overall NSLDS II security approach. The web application security details the authentication of users to the web application and authorization of users to view pages and perform functions.

2.4.1 User Authentication

The user logs onto the NSLDS II website by providing a user ID and password on the logon.jsp page. The user ID and password are passed via an encrypted https request using SSL from the client's browser to the IHS server. The username will be stored in an encrypted column in DB2. An algorithm will be used to compare the password entered by the user to the encrypted password stored in the database. The [data access layer](#), described in a later section of this document, will provide the connection to the database and perform the logic to query the database. A logon Java object (an [architecture layer](#) component) will contain the logic to access the data access layer, utilize the algorithm, and compare the user-entered password and the database returned password for authentication.

2.4.2 Web Authorization

User access to specific tabs, pages, and data field elements will be restricted through the use of Location Groups, Function Groups and Web Groups. Location Groups define the specific location that a user is logging in from (e.g. a University's campus). Function Groups define what functionality a user can perform (e.g. support staff functionality). Web Groups map web pages to specific function groups (e.g. access to the Support tab).

A new field will be added into the database to support the ability to add new pages and associate them with web groups. Each new page will have to be created and a menu will exist to create the new page. Then the page will be associated with a web group. On the JSP, permission validation will check if the user belongs to the defined web group.

2.4.3 Encryption

IHS supports 128bit SSL v3 and can be configured to support different variations of encryption level support. The current NSLDS application checks the users' browser to see if it supports 40bit, 56bit, or 128bit encryption. Currently, NSLDS supports all these levels of encryption, but it recommends using a browser that supports 128bit encryption for users within the United States.

When the user logs on to the Student Access page a check on their browser is performed; if the browser does not support 128bit encryption, the user is directed to a warning page specifying how to upgrade their browser. The user is allowed to continue to the PIN site regardless of their encryption level.

For the Financial Aid Professional (FAP) web site, the user must have a supported secure browser (Microsoft Internet Explorer 4.x or higher or Netscape Navigator 4.x or higher) to access the site.

2.5 Reporting Engine Integration Strategy

MicroStrategy will be used as the Reports creation and delivery engine. It will interact with the Data Mart to produce standard and ad hoc reports. The MicroStrategy interface will be accessible when the user selects the Reports tab in the NSLDS II web application. This section defines the integration of the web application with the reporting tool, (please reference the NSLDS II Reengineering Reports Detailed Design for reporting specific information).

When the user clicks on the Reports tab, a new browser window is launched and a new session is started on the MicroStrategy server. The following diagram depicts the way Single Sign On will occur between MicroStrategy and the Web Application. The steps are outlined in more detail in the diagram below.

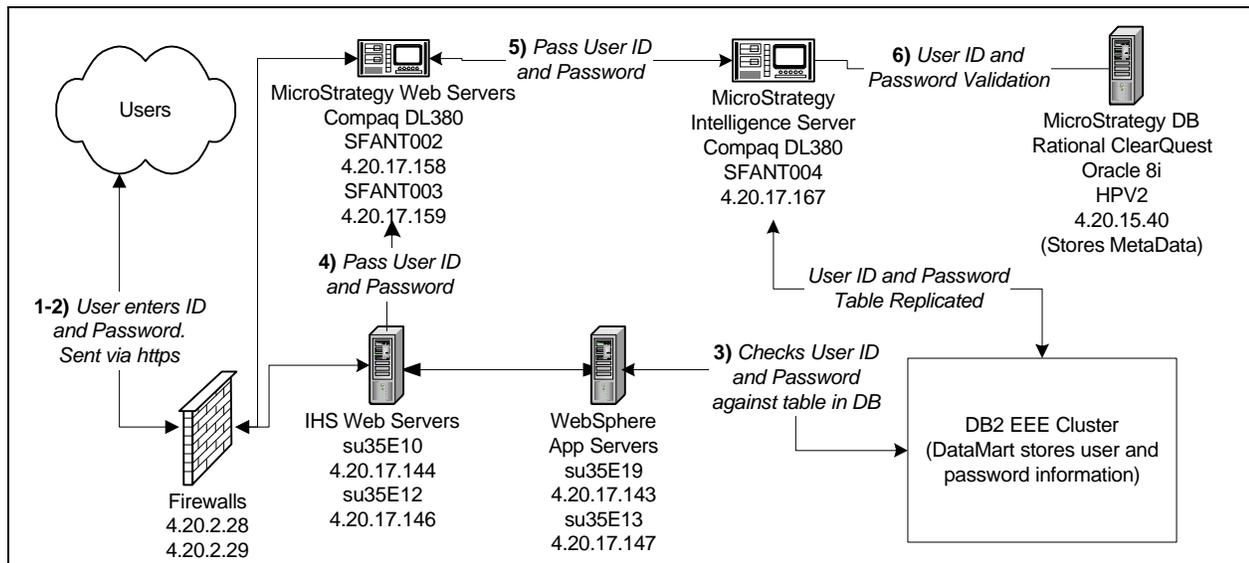


Figure 4: MicroStrategy Single Sign on Authentication

- 1.) Users will enter their user ID and password on the Logon page.
- 2.) The user ID and password will be passed via an https request over SSL to the IHS web Server.
- 3.) WebSphere Application Server has an established connection with the EDW that allows the web application to select the userID and password from the EDW, decrypt the password and authenticate against the parameters entered.
- 4.) When a User clicks on the Reports Tab to create or access a report, the user ID and password will be passed in a form via a JavaScript function to the desktop.asp file on the MicroStrategy

IIS Web Server. (The Report Tab link will reference the MicroStrategy Web Server's internal Network Address Translation IP address. This will remove the need to send information through the firewall and make encrypting the user information unnecessary.)

- 5.) The user ID and password are passed from the ASP file on the MicroStrategy Web Server to the MicroStrategy Intelligence Server.
- 6.) The MicroStrategy Intelligence Server takes the user ID and password and checks it against the MicroStrategy table in its Oracle database. (The user ID and password tables are replicated from the EDW to the MicroStrategy database on a regular basis.)
- 7.) If the user ID and password exist, the MicroStrategy Home page is displayed in a separate browser window. The user is now accessing MicroStrategy ASP files directly from the MicroStrategy IIS Web Server. The requests are encrypted via SSL.

3 Business Layer Design

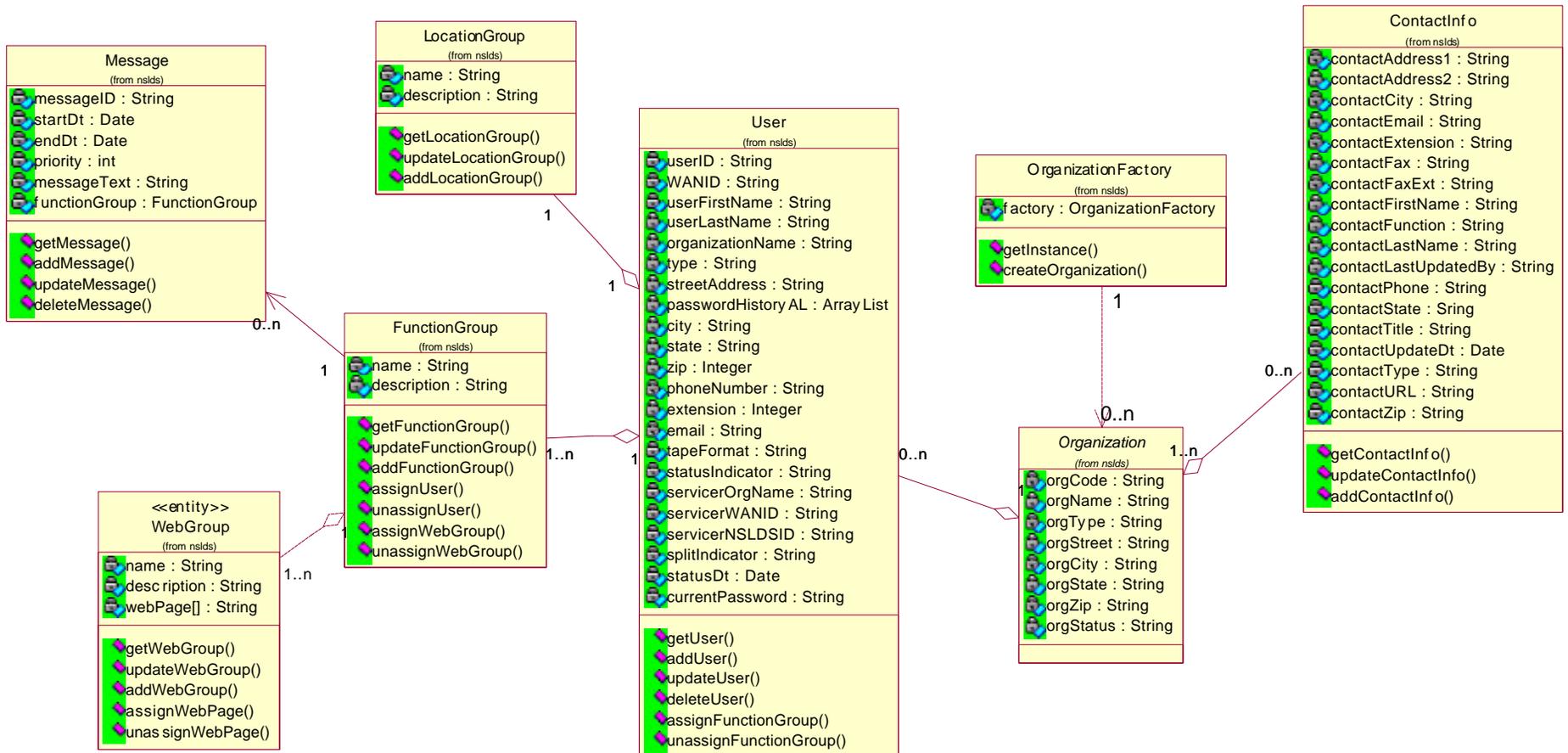
This section is comprised of the class diagram, object definition, sequence diagrams, and object-data mapping model. The class diagram shows the different objects and how they are associated with each other through aggregation, association, or inheritance. It also shows the multiplicity between objects such as one Student can have one or many loans in NSLDS II.

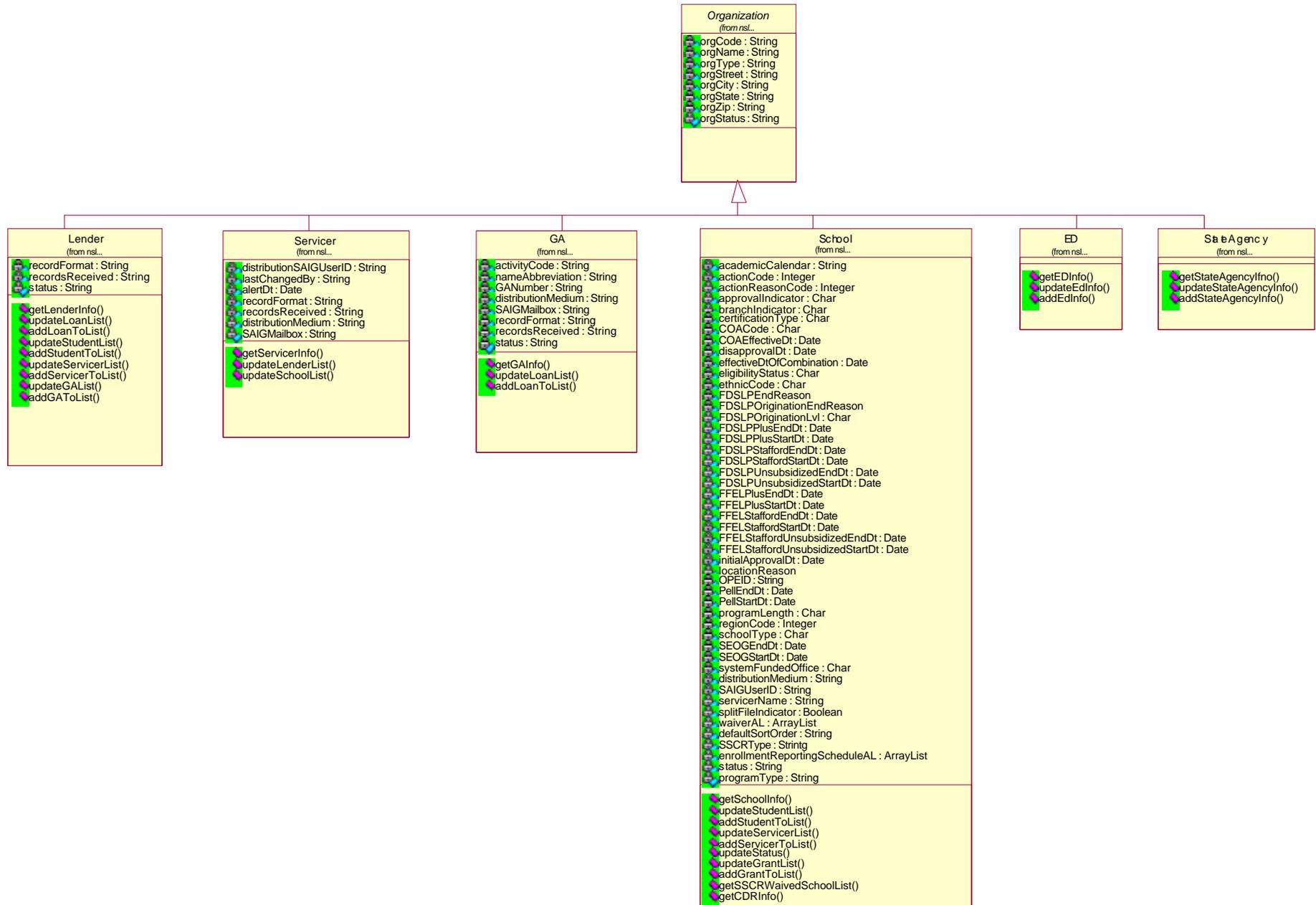
The sequence diagrams illustrate the timing and interaction between methods in the different classes that are called during the execution of a task. Due to time constraints and the numerous interactions between the pages, only select example diagrams from each module are shown. These select examples were chosen to reflect various actions within the system, such as update and add, to provide a more diverse example set.

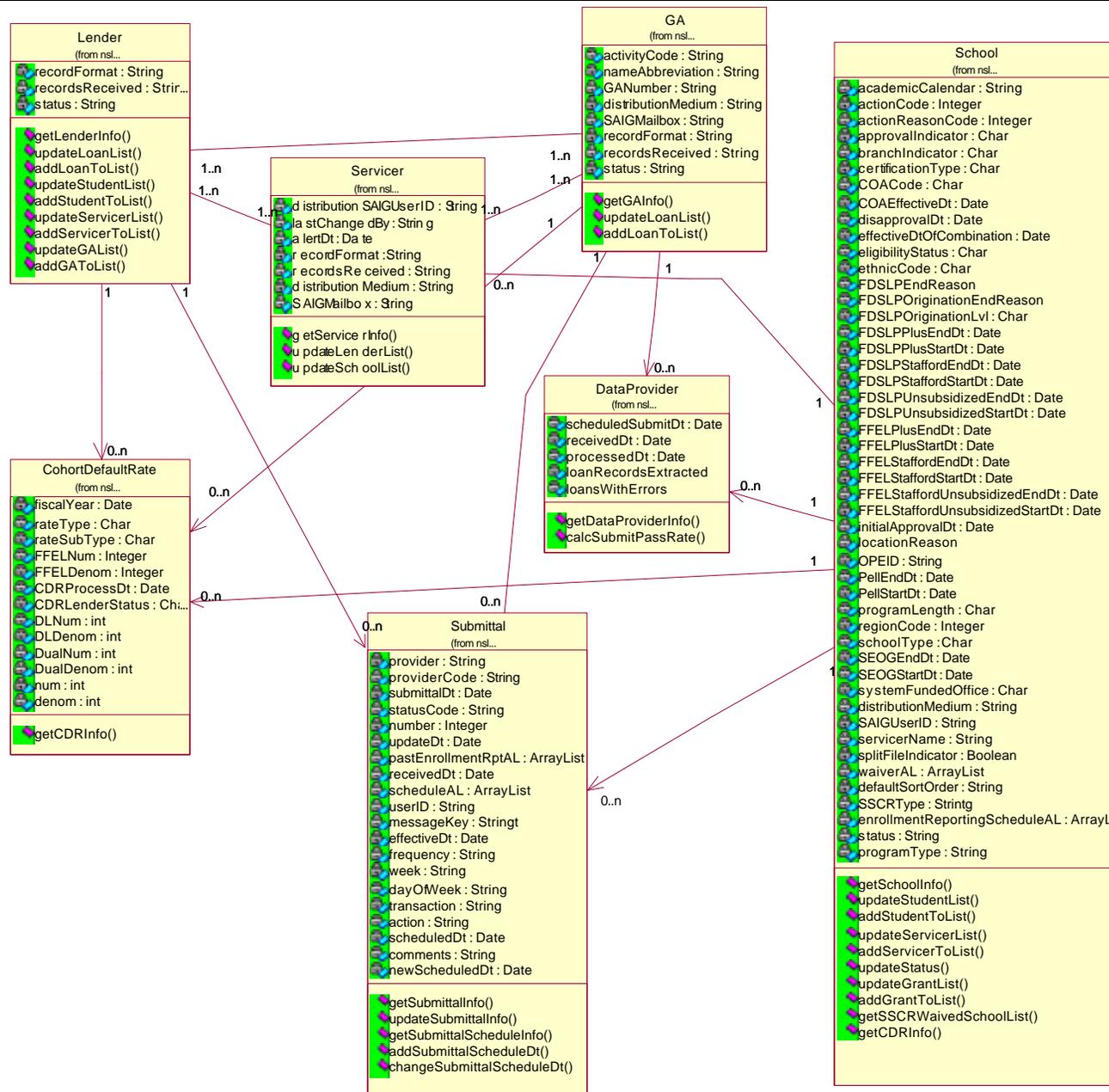
The object-data mapping spreadsheet (provided in Appendix B) maps objects from the system code to the tables and attributes in the database. This spreadsheet contains only the objects and attributes defined in the class diagram and preliminary mapping to the existing EDW data model. The object-data mapping model is not complete, as the source documentation from the legacy system was not provided in sufficient time for this function to be properly documented in this version of the design. However, the complete mapping of objects to the logic present in the legacy COOL:Gen code is included. This mapping provides a complete path from the objects defined in the data-mapping model to the business logic for each screen.

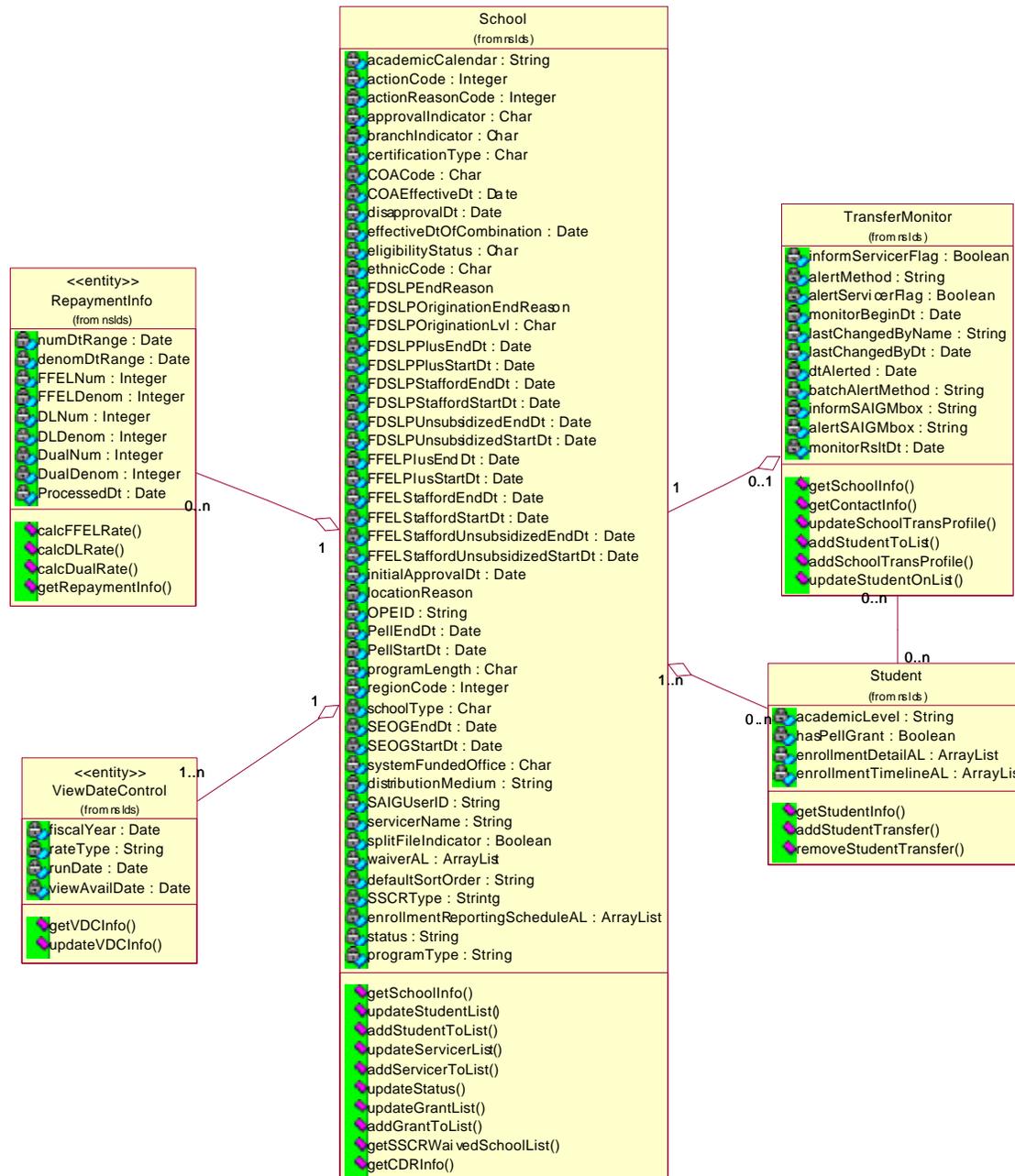
3.1 Class Diagram

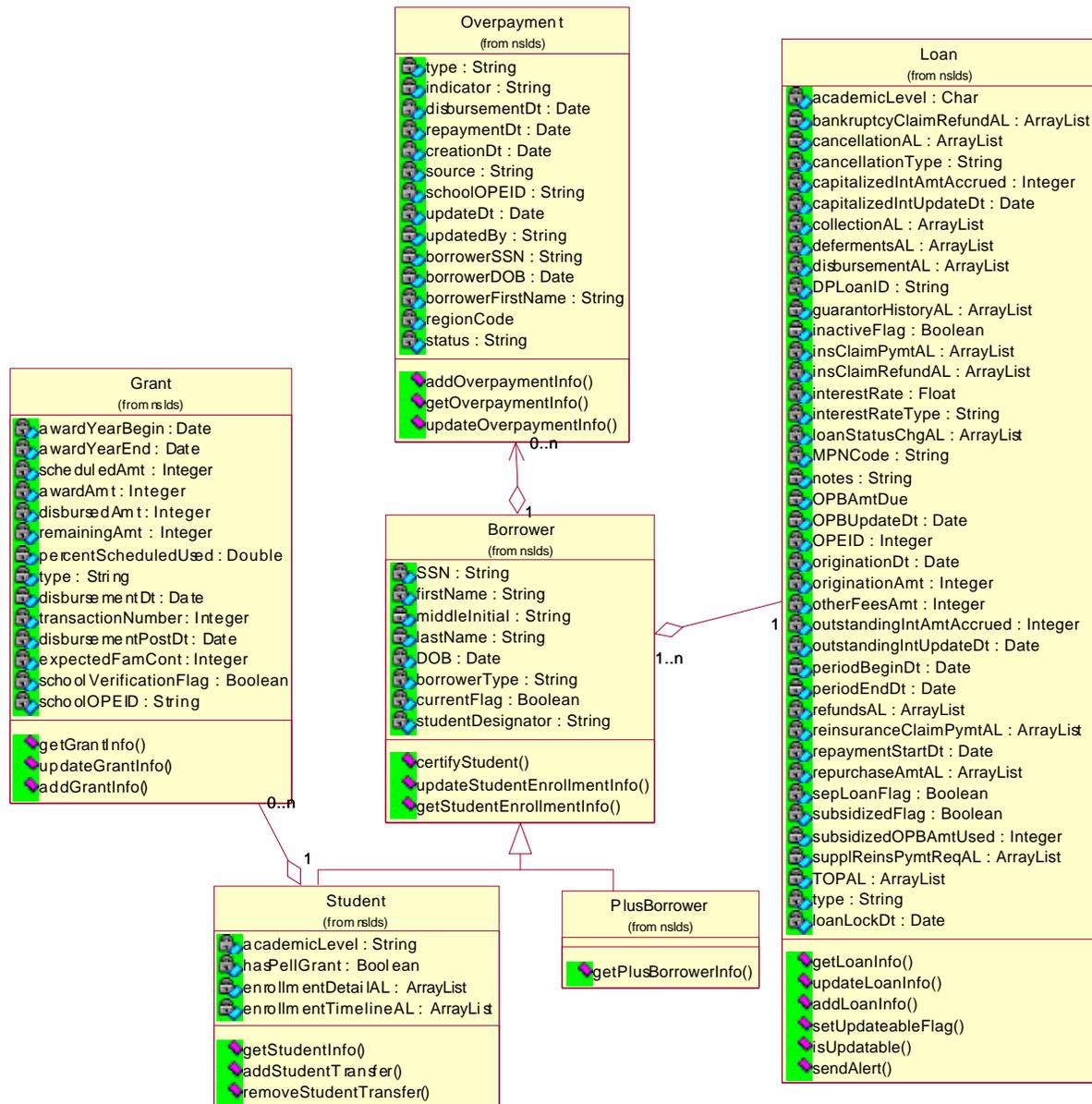
The classes in the diagram have been divided and placed across multiple pages for easier viewing. Some classes may appear in more than one section to show the relationship of that class in relation to other classes on the same page, but only one definition of the class will actually exist.

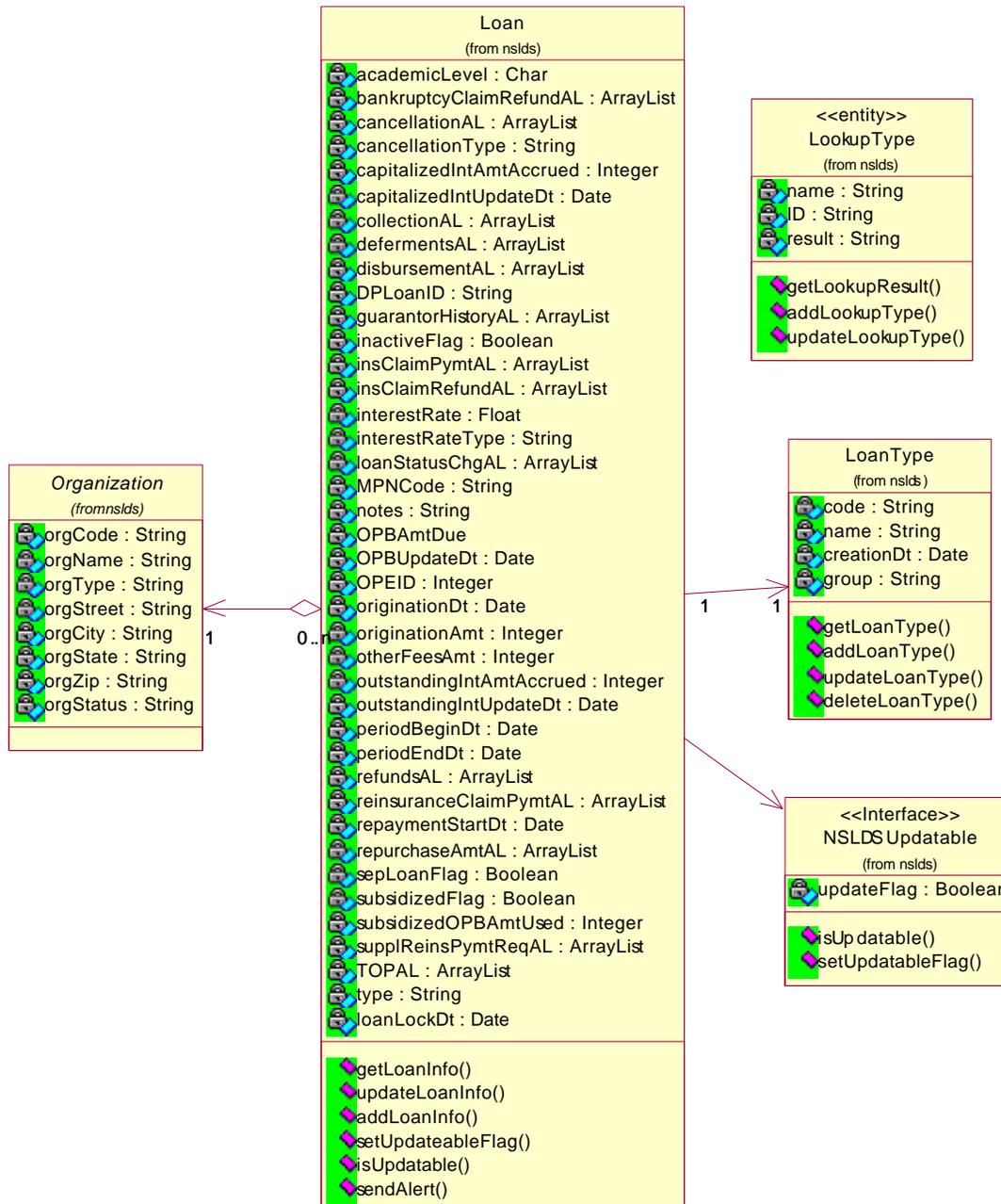


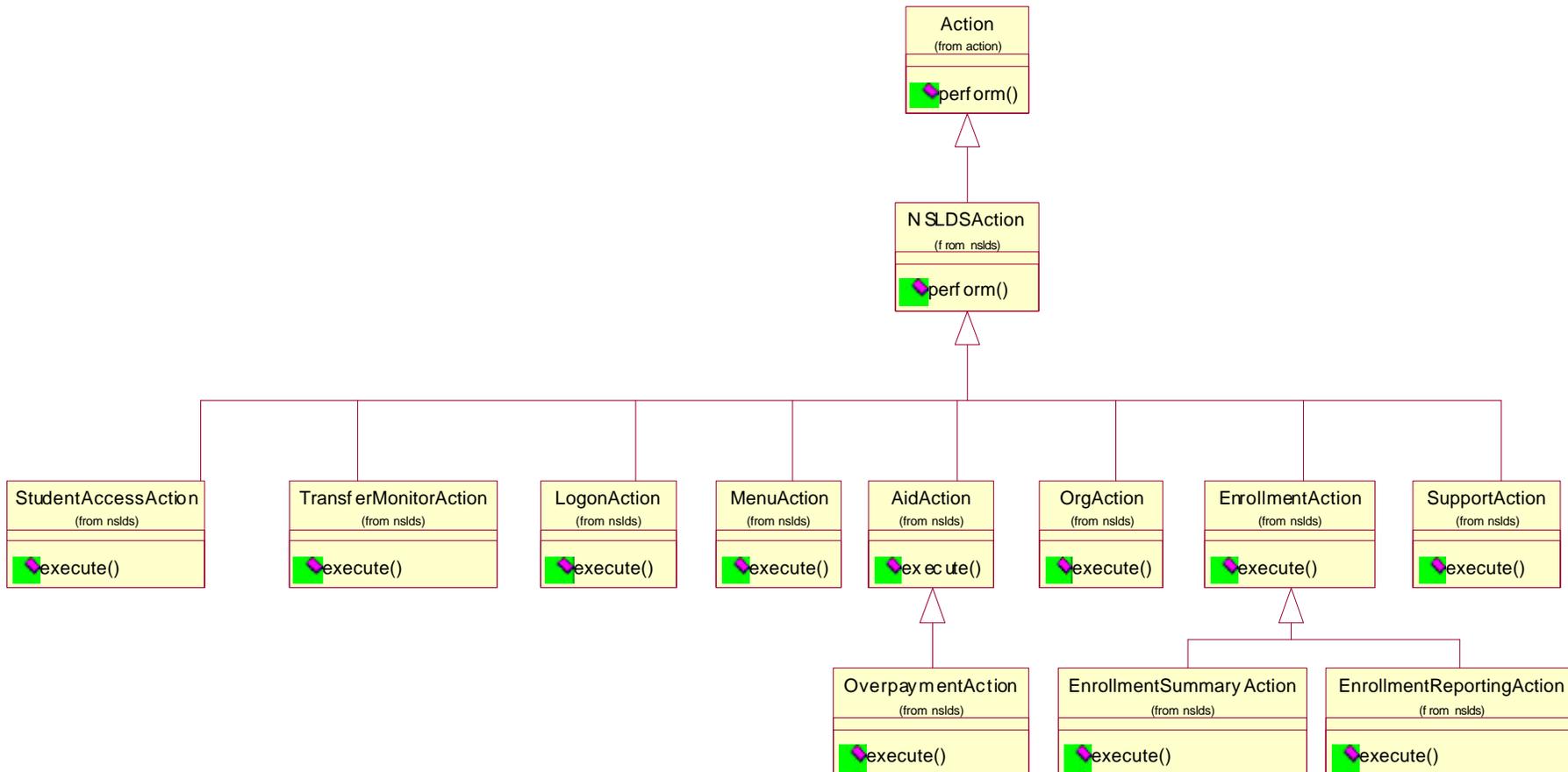




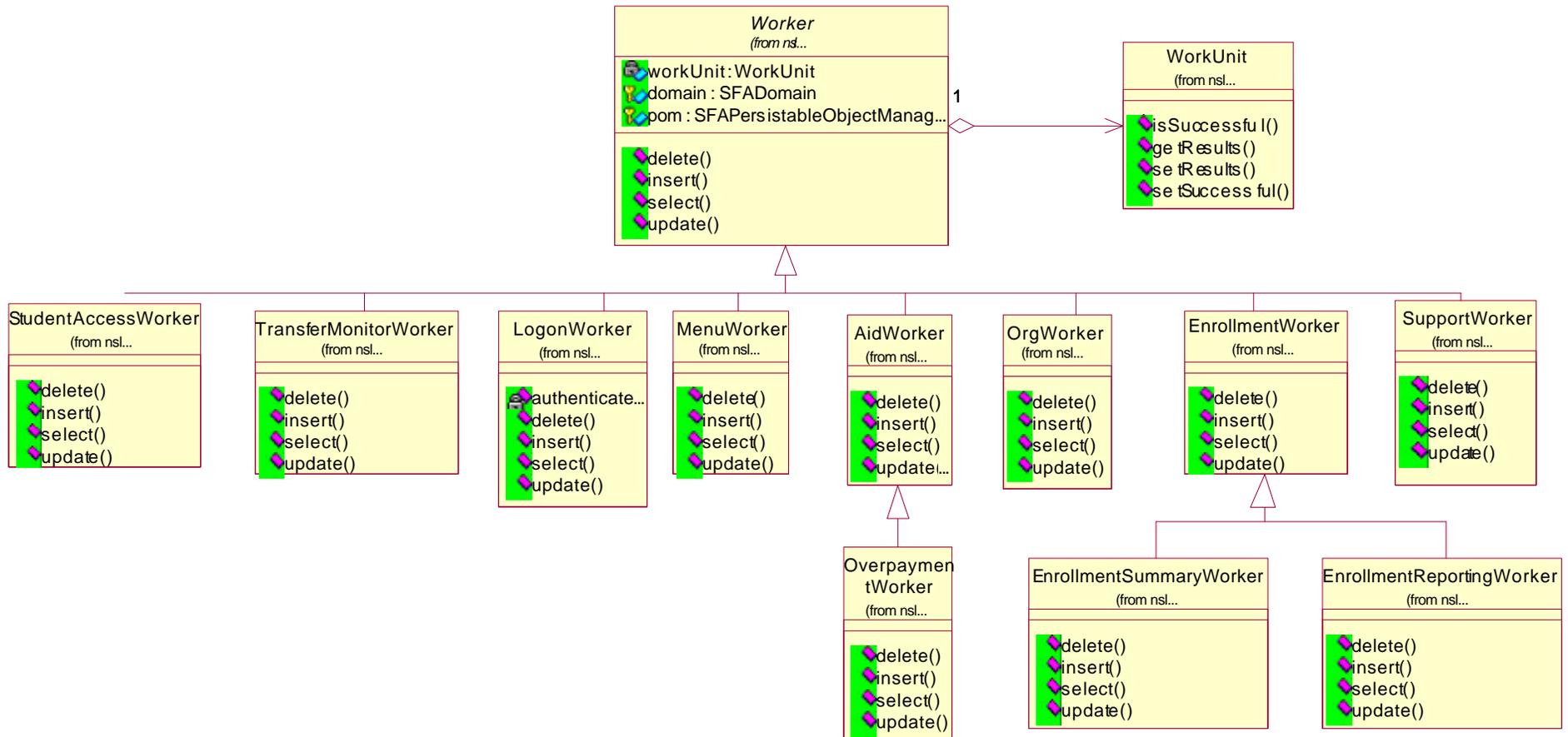




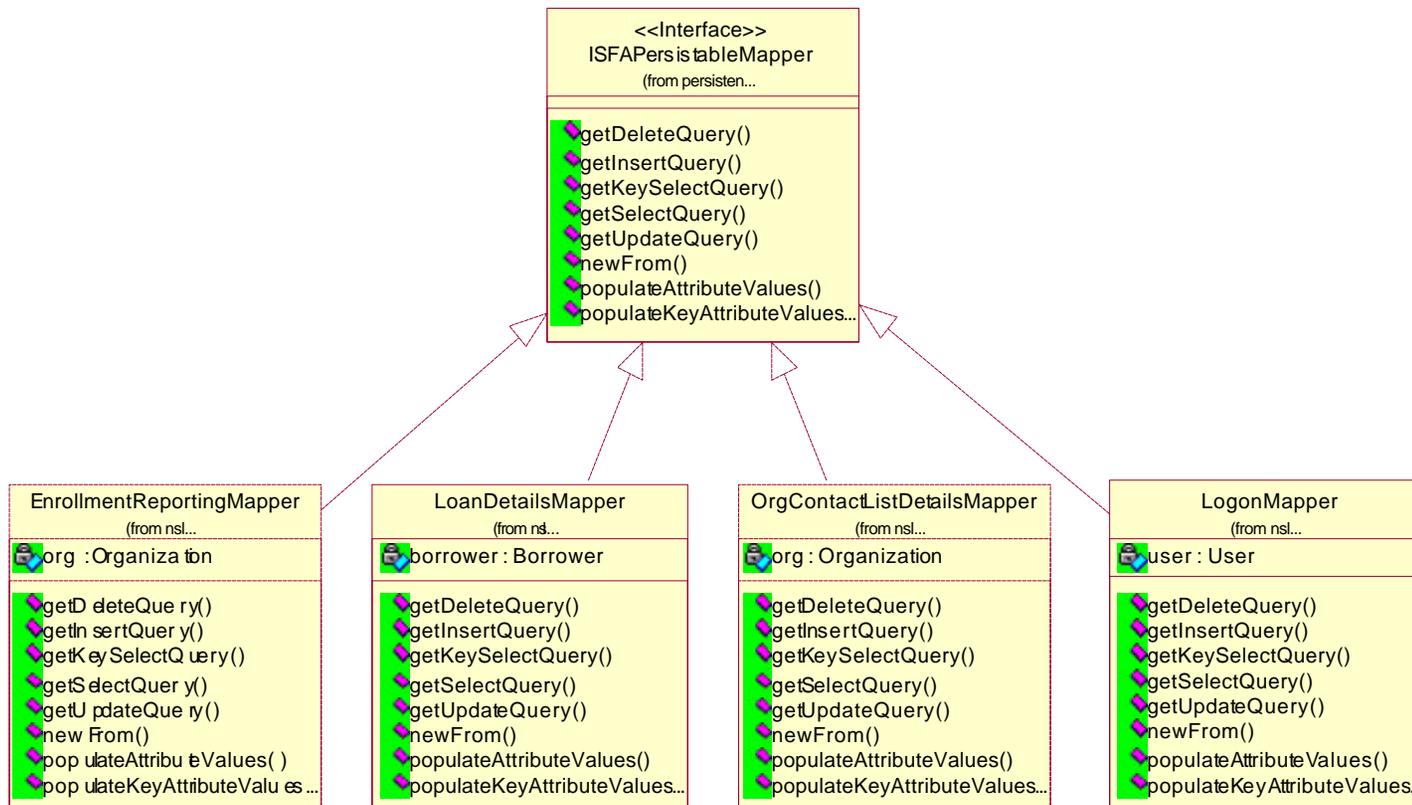
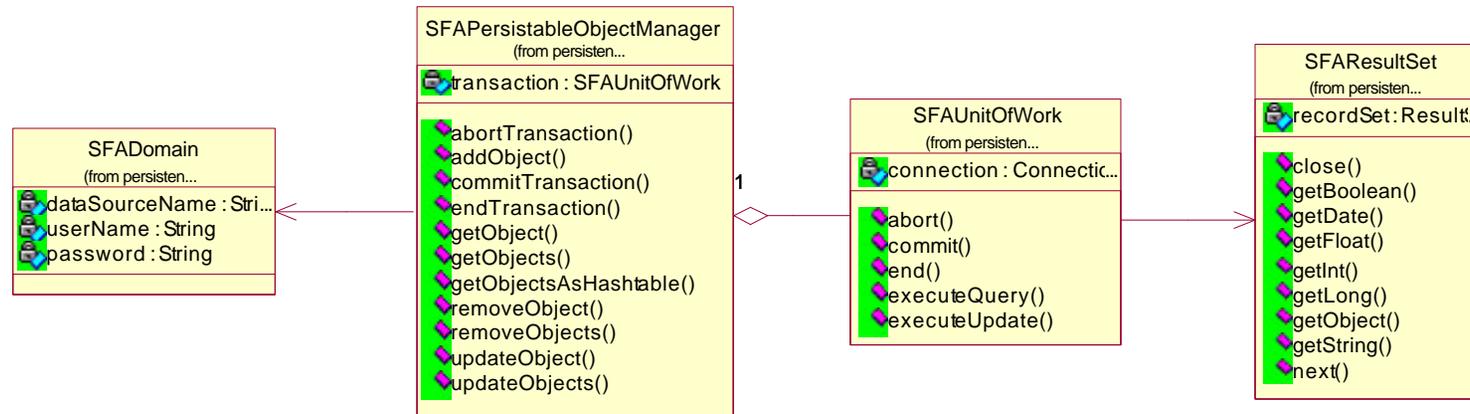


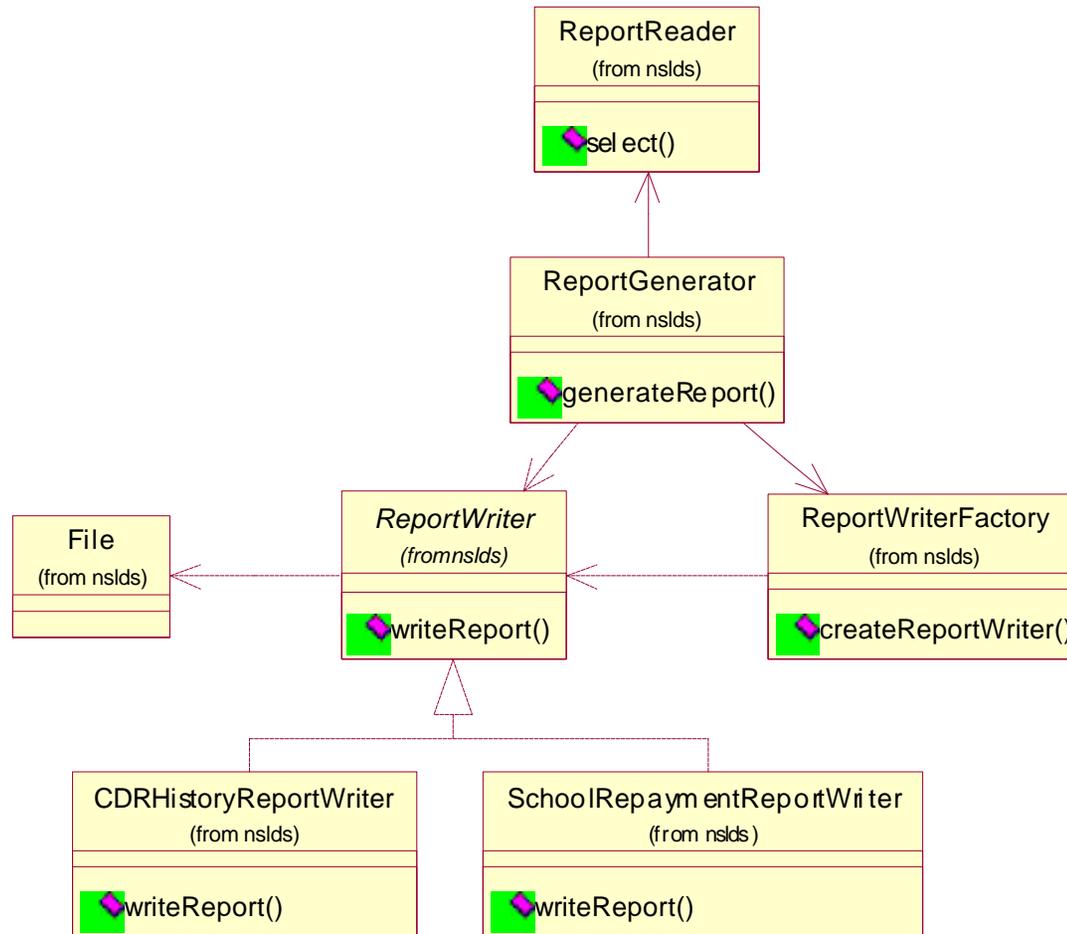


The Action object and NSLDSAction classes are concrete classes that individual business Action subclasses will extend. A business Action subclass will be created for every page in the web site that perform business logic. Due to timing constraints, not all subclasses were displayed; one main subclass for each module is shown to demonstrate the inheritance relationship and act as a placeholder for business actions in that module.



The Worker object is an abstract class implemented by subclassed Worker objects. The Worker class interacts with the Configuration framework and instantiates a protected member variable for the PersistableObjectManager for the Worker subclasses to utilize. For every Action that requires database interaction, there will be a subclassed Worker object. Due to timing constraints, not all subclasses were displayed; one main subclass for each module is shown to demonstrate the inheritance relationship and act as a placeholder for business actions in that module.





These reports objects are for the generation of formatted flat file exception reports which are defined in the NSLDS II Reengineering Reports Detailed Design documentation.

3.2 Object Actions

The table below lists the major actions that will be performed by the objects defined in the class diagram. There are two categories of objects defined by the NSLDS Application Architecture – business objects and business workers.

Business objects contain the informational fields used to hold the NSLDS data. Business worker objects contain specialized methods used to operate on the business objects to perform a NSLDS action, usually in coordination with the database. All worker classes are able to select, insert, update, and delete records from the database. Some workers have methods to perform more specific business logic, such as authentication. The workers use the business objects as placeholders for the NSLDS data when performing actions on the database. The business objects also contain methods that allow them to operate on themselves. Business objects are able to execute calculations and business logic based on the values contained in their fields. The following table lists the actions exposed by the business objects and general worker objects.

| Business Object Name | Description | Action |
|-----------------------------|---|---|
| ContactInfo | Represents an Organization's contact or support personnel's contact information. | getContactInfo updateContactInfo addContactInfo |
| Lender | Represents the entity that lends money to the Borrower. Keeps track of the Students, Loans, Servicers, and GAs for those loans. | getLenderInfo updateLoanList addLoanToList updateStudentList addStudentToList updateServicerList addServicerToList updateGAList addGAToList |
| Servicer | Represent the entity that services enrollment information for Schools and Loan information for Lenders. | getServicerInfo updateLenderList updateSchoolList |
| ED | Represents different Education Regions. | getEDInfo updateEDInfo addEDInfo |
| State Agency | Represents a particular State Agency. | getStateAgencyInfo updateStateAgencyInfo addStateAgencyInfo |
| GA | Entity that guarantees financial backing of a loan. Used for performing online loan updates. | getGAInfo updateLoanList addLoanToList |

| Business Object Name | Description | Action |
|-----------------------------|--|---|
| School | Entity representing a school. Contains a list of current students and grants disbursed. It is also used to monitor transferring students. | getSchoolInfo updateStudentList addStudentToList updateServicerList addServicerToList updateStatus updateGrantList addGrantToList getSSCRWaivedSchoolList getCDRInfo |
| TransferMonitor | Entity used to track transferring students for an associated school. | getSchoolInfo getContactInfo updateSchoolTransProfile addStudentToList addSchoolTransProfile updateStudentOnList |
| RepaymentInfo | Associated with the School class. Able to calculate the rates of repayment for the loans. | calcFFELRate calcDLRate calcDualRate getRepaymentInfo |
| DataProvider | Extracts loan data and populates School and GA objects. | getDataProviderInfo calcSubmitPassRate |
| CohortDefaultRate | Entity to store CDR information. | getCDRInfo |
| Submittal | Data Provider stores schedule information for Schools, GAs, Lenders, and Servicers. | getSubmittalInfo updateSubmittalInfo getSubmittalScheduleInfo addSubmittalScheduleDt changeSubmittalScheduleDt |
| ViewDateControl | Entity to store VDC information. | getVDCInfo updateVDCInfo |
| User | Represents the User on the NSLDS system. User object created during logon action. Object's properties used to authenticate logon process. A User belongs to a function and location group. | getUser addUser updateUser deleteUser assignFunctionGroup unassignFunctionGroup |
| LocationGroup | The physical location of the User. | getLocationGroup addLocationGroup updateLocationGroup |

| Business Object Name | Description | Action |
|-----------------------------|---|---|
| FunctionGroup | Used to authorize User for functionality of the NSLDS site. A function group can have many web groups related to it. | getFunctionGroup updateFunctionGroup addFunctionGroup assignUser unassignUser assignWebGroup unassignWebGroup |
| Message | Represent messages to the User. Can be assigned to different Function groups. User is able to view more than one at a time. | getMessage addMessage updateMessage deleteMessage |
| WebGroup | User's authorization drilled down to the web page level. | getWebGroup updateWebGroup addWebGroup assignWebPage unassignWebPage |
| Borrower | Represents an entity that has borrowed money. It contains the records for Overpayment, Loan, and Grant history. It is used to update and display the financial history. | certifyStudent updateStudentEnrollmentInfo getStudentEnrollmentInfo |
| Student | A subclass of Borrower. A Student belongs to a School. It contains enrollment information for a list of schools and grant details. | getStudentInfo addStudentInfo removeStudentTransfer |
| PlusBorrower | A subclass of Borrower. Represents an entity that has borrowed money on behalf of a Student. | getPlusBorrowerInfo |
| Grant | Entity used to store Grant information. | getGrantInfo updateGrantInfo addGrantInfo |
| Overpayment | Represents an Overpayment by a Borrower on a loan. | getOverpaymentInfo updateOverpaymentInfo addOverpaymentInfo |
| Loan | Contains all of the necessary information for a loan. A loan belongs to a Borrower. A loan is associated with a LoanType. | getLoanInfo updateLoanInfo addLoanInfo setUpdateableFlag isUpdatable sendAlert |

| Business Object Name | Description | Action |
|-----------------------------|--|--|
| LoanType | Describes the type of loan. | getLoanType addLoanType updateLoanType deleteLoanType |
| LookupType | Maintains validation translation relationships. | getLookupType addLookupType updateLookupType |
| NSLDSAction | Subclass of the Action class. Parent class to all NSLDS actions. Entry point for the business action. Contains the centralized exception handling logic. This object will call the execute() method of the other Action objects. | perform |
| Worker | Parent class for all NSLDS business workers. This is an abstract class and cannot be instantiated. Defines the required methods for all Worker classes. Contains a WorkUnit object that is used to return the results of the work. | delete insert select update |
| WorkUnit | Used to record the results of the operation of a Worker class. | isSuccessful setSuccessful getResults setResults |
| ISFAPersistableMapper | Part of the Persistence framework. Interface that all business mappers must implement. Contains logic for the object-data model mapping. | getDeleteQuery getInsertQuery getKeySelectQuery getSelectQuery getUpdateQuery populateAttributeValues populateKeyAttributeValues |
| SFADomain | Part of the Persistence framework. Placeholder class for database's data source name, user ID, and password which it pulls from the Configuration framework. | |

| Business Object Name | Description | Action |
|-----------------------------|---|--|
| SFAPersistableObjectManager | Part of the Persistence framework. Defines the different database operations exposed to the client objects. Used to select, update, insert, and delete records with the database. Provides transactional monitoring through abort and commit methods. | abortTransaction addObject commitTransaction endTransaction getObject getObjects getObjectsAsHashtable removeObject removeObjects updateObject updateObjects |
| SFAUnitOfWork | Part of the Persistence framework. This class is not exposed to the developer. Used internally in the Persistence Framework to record a roundtrip transaction. | abort commit end executeQuery executeUpdate |
| SFAResultSet | Part of the Persistence framework. This class is not exposed to the developer. Used internally in the Persistence framework to record results from database query. | close getBoolean getDate getFloat getInt getLong getObject getString next |
| ReportGenerator | Control class to generate a report File. Interacts with ReportReader and ReportWriter. | generateReport |
| ReportReader | Reads from a database table all of the report fields corresponding to a Username + timeStamp +Report ID label and stores the data in a ResultSet. | select |
| ReportWriter | Generates the output File. While iterating through the ResultSet from the ReportReader, it writes the report data and report headers. | writeReport |
| ReportWriterFactory | Creates the specific ReportWriter based on the Report ID. | createReportWriter |

Table 4, Object Description and Actions

3.3 Sequence Diagrams

The Normal Processing category includes examples from the seven different business modules. The Exception Processing has three diagrams that are examples of the different types of exceptional conditions in the NSLDS system and how they are handled by the application architecture.

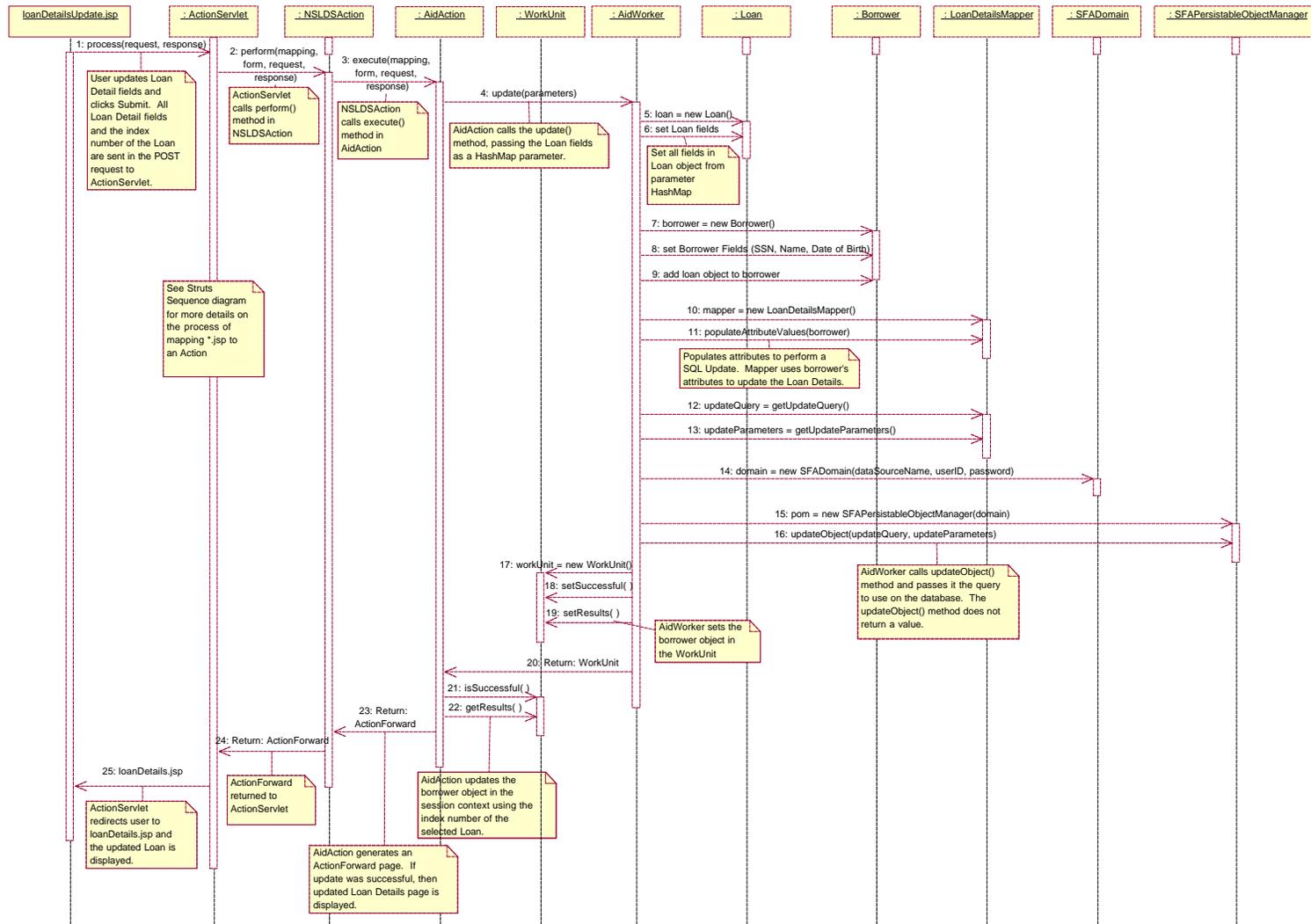
Normal Processing – Business Modules

- Aid - Update Loan Details
- Aid - View Loan History
- Aid - Update Overpayment Details
- Aid - View Overpayment History
- Enrollment - Add Reporting Schedule
- Enrollment - View Summary
- Logon
- Organization - View Contact List
- Organization - Delete Contact from List
- Student Access - View Financial Aid Review
- Support - View Contact List
- Support - Add Contact to List
- Transfer Monitor - View Transfer List
- Transfer Monitor - Delete Transfer from List

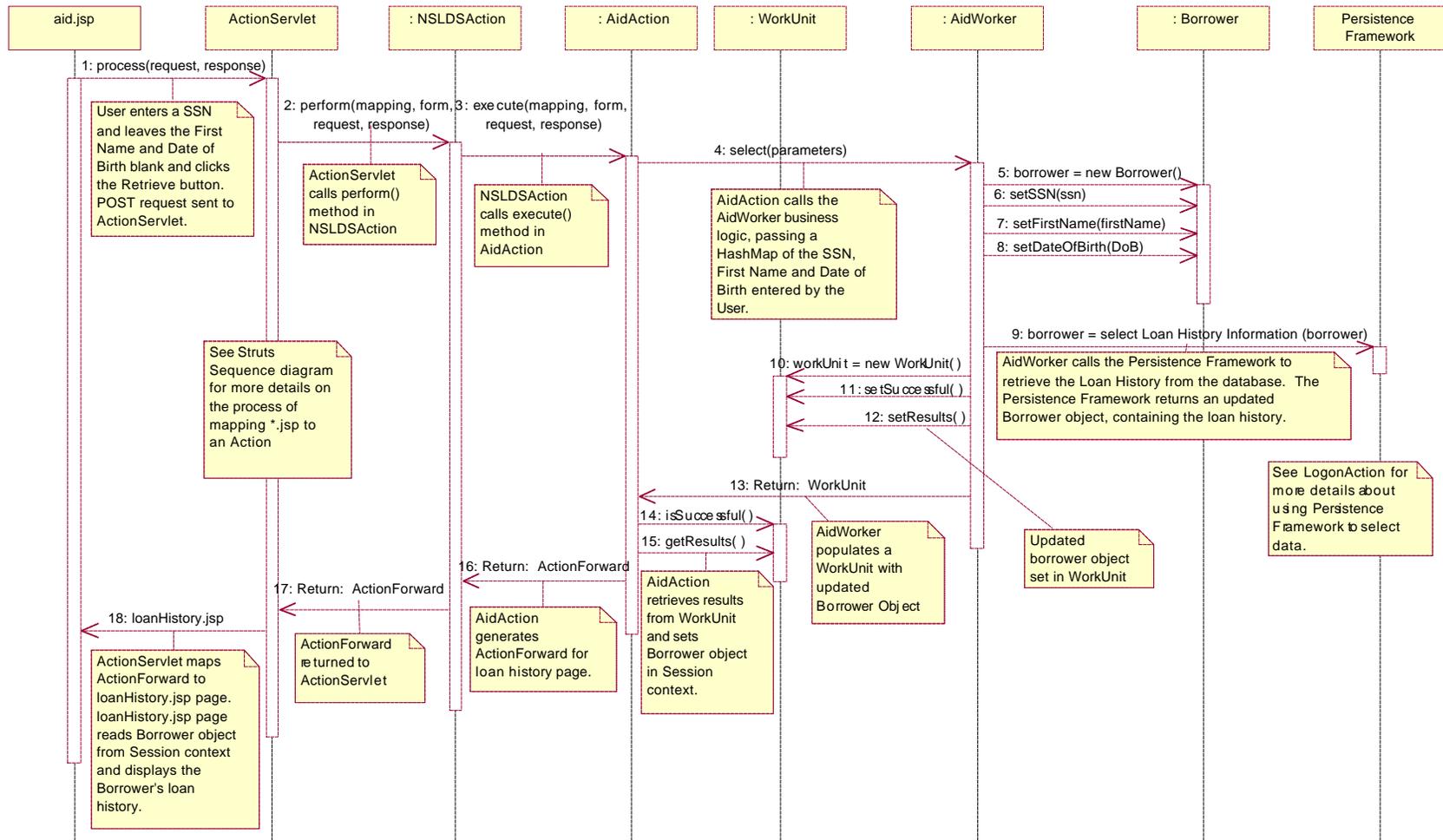
Exception Processing – Example sequences

- Application errors (i.e. Authentication error)
- Exceptions thrown by Java code (i.e. SQL Exception)
- Web Conversion framework errors (i.e. Form validation error)

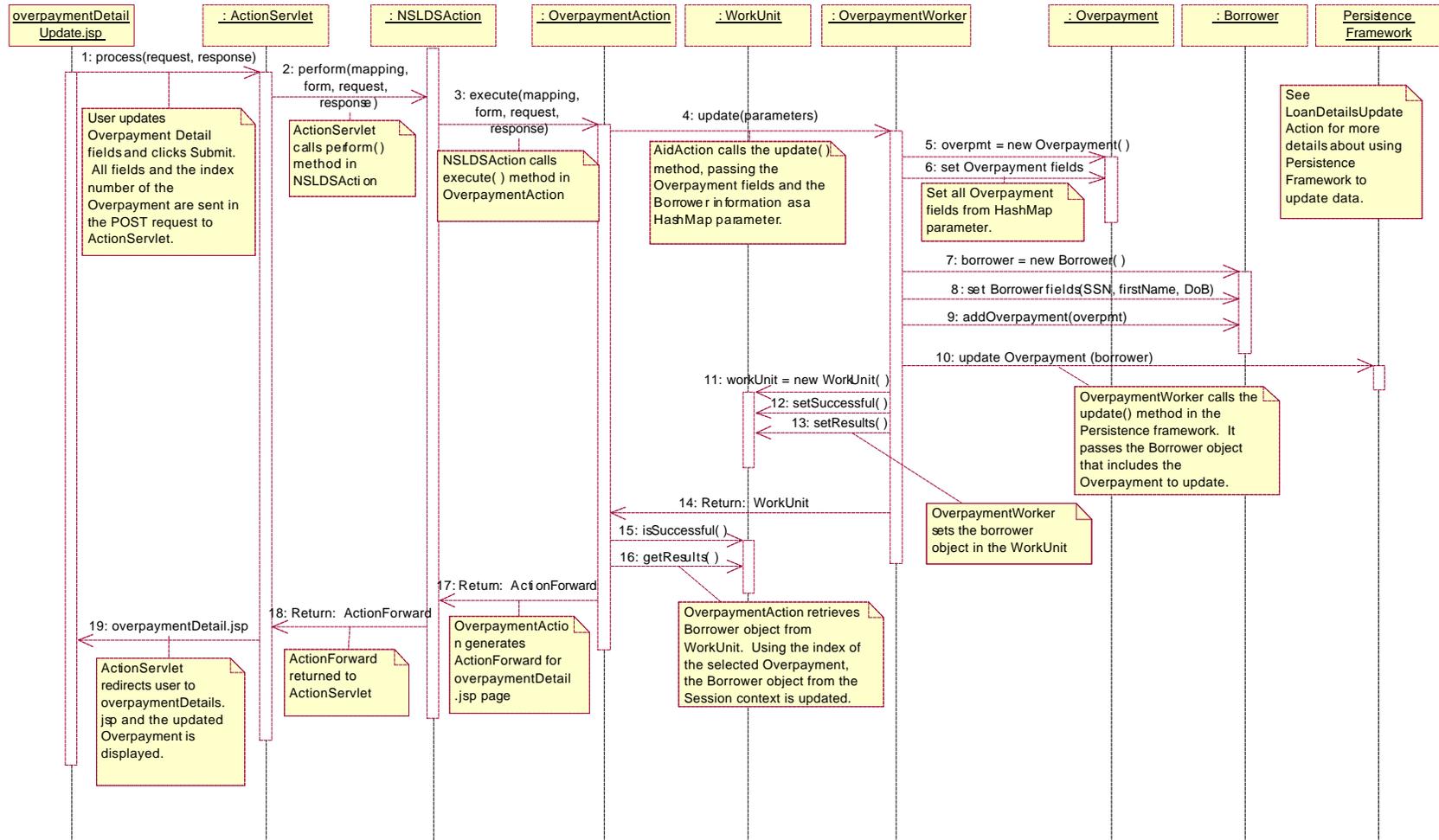
3.3.1 Aid - Update Loan Details



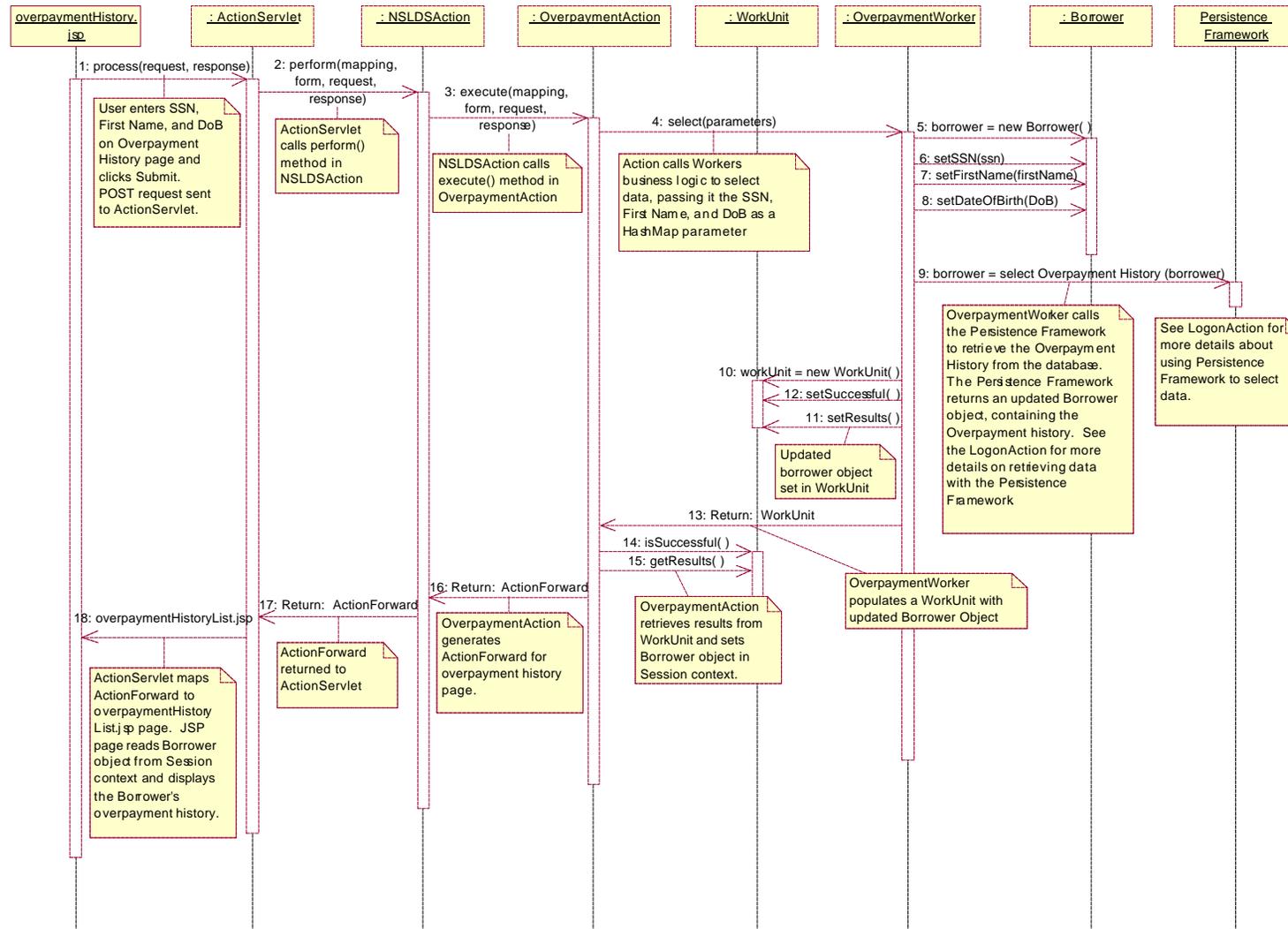
3.3.2 Aid - View Loan History



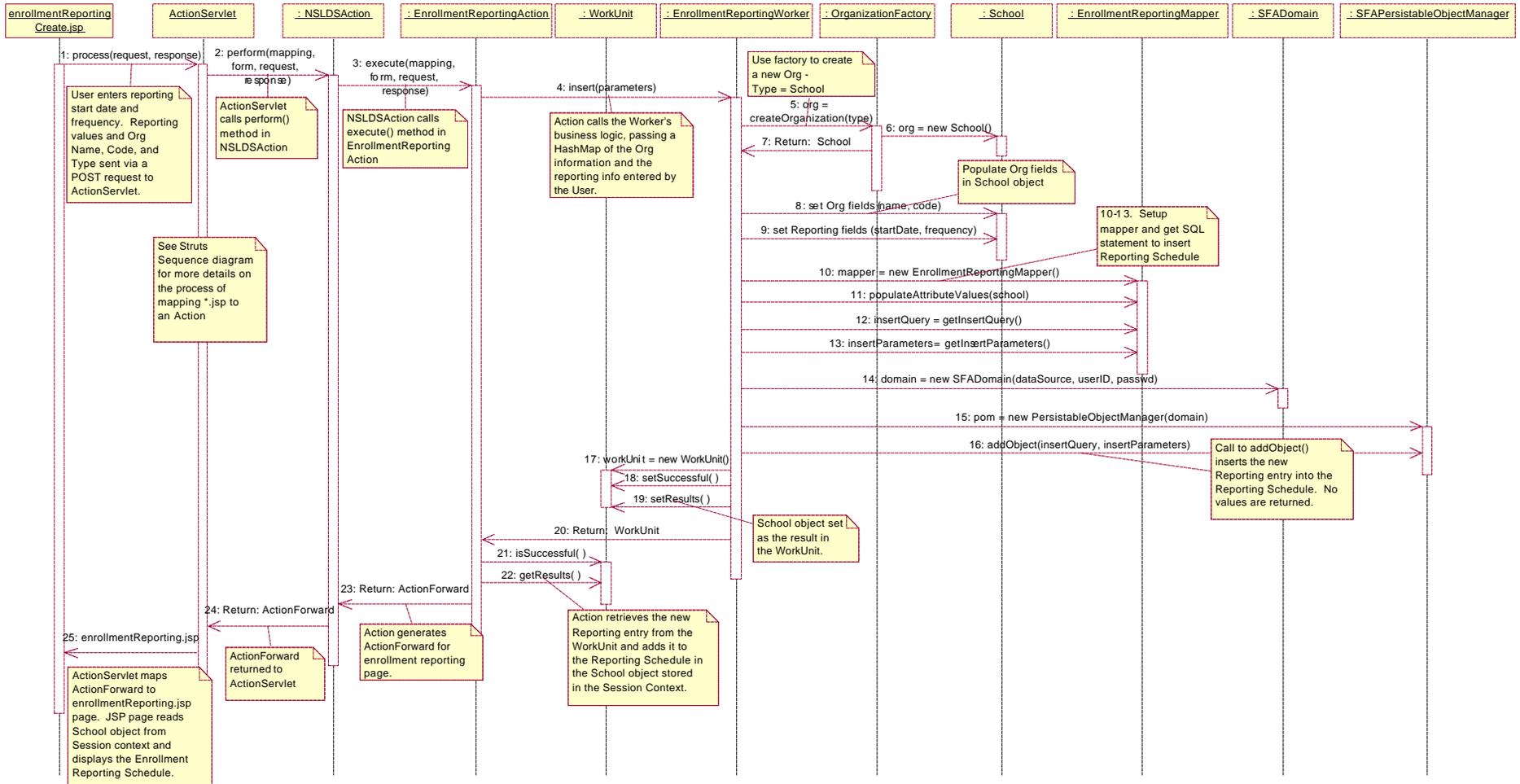
3.3.3 Aid - Update Overpayment Details



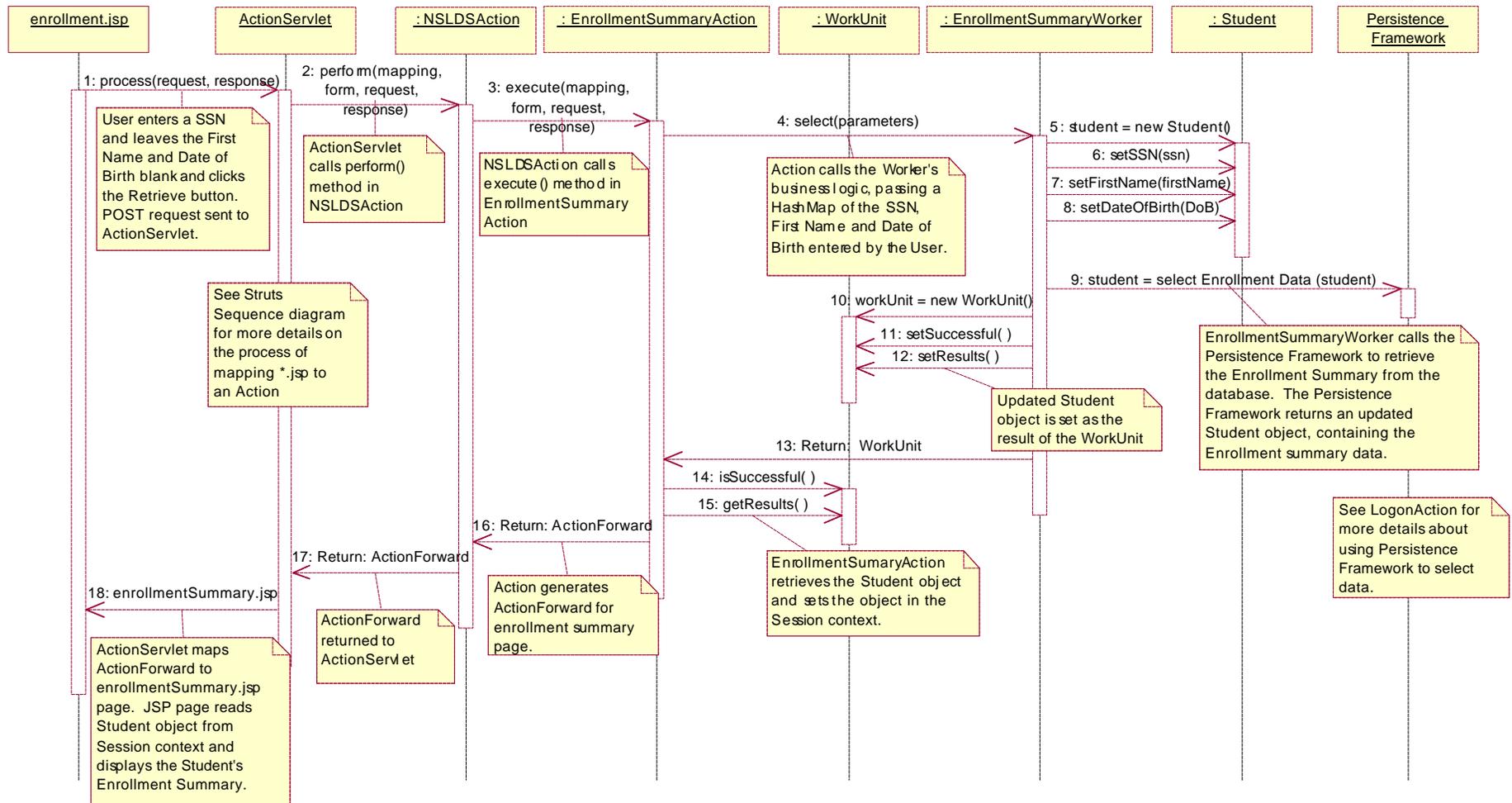
3.3.4 Aid - View Overpayment History



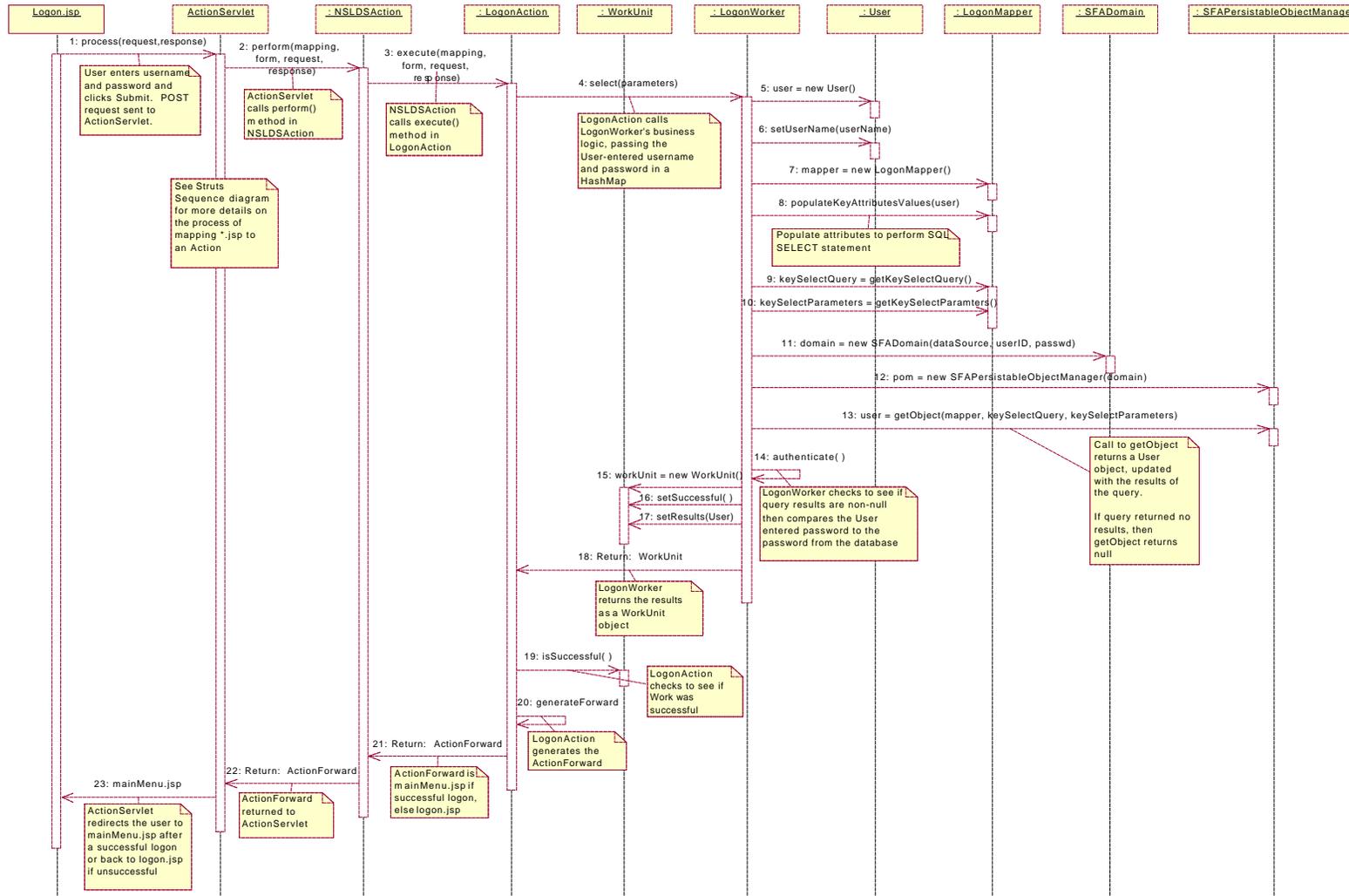
3.3.5 Enrollment - Add Reporting Schedule



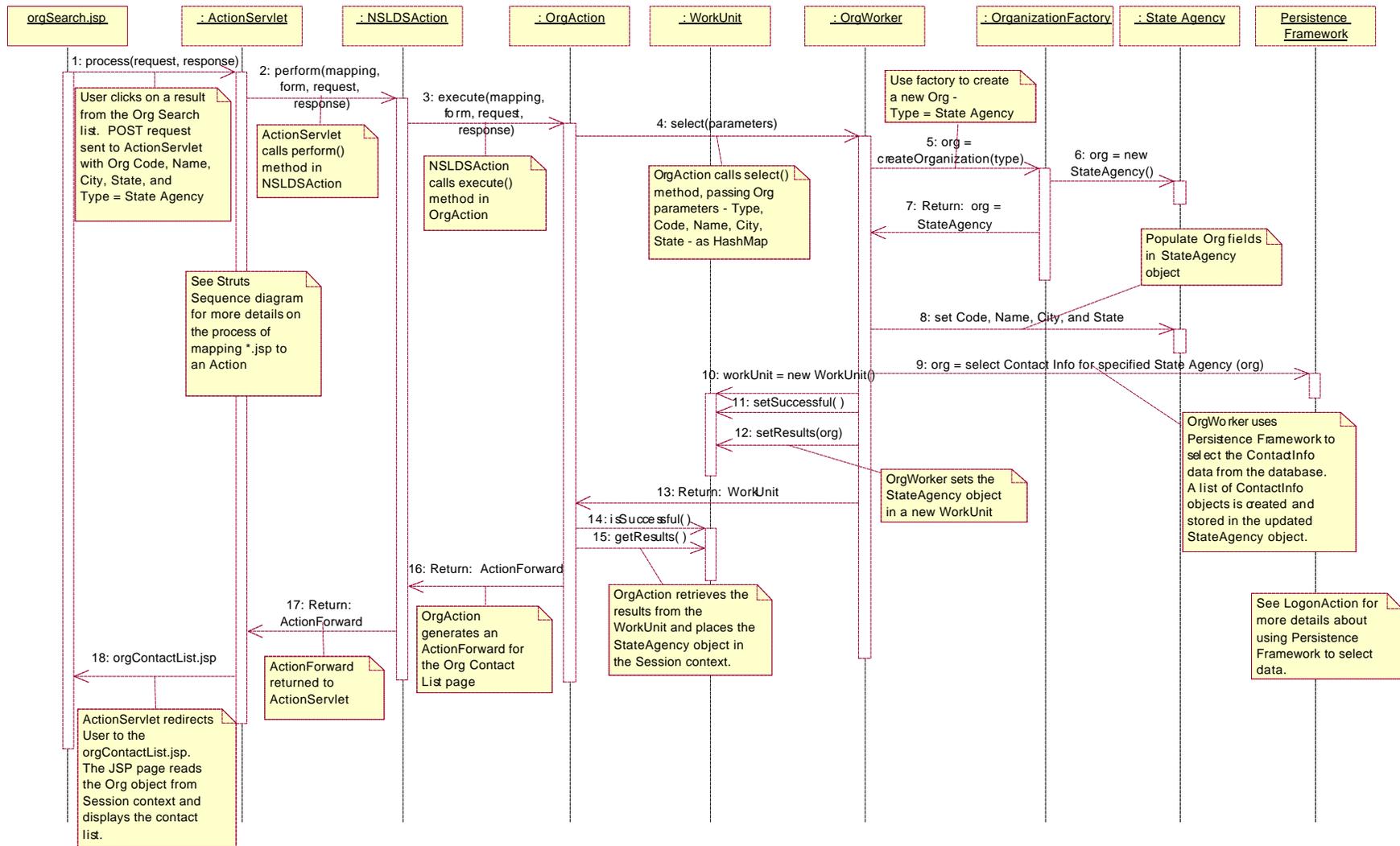
3.3.6 Enrollment - View Summary



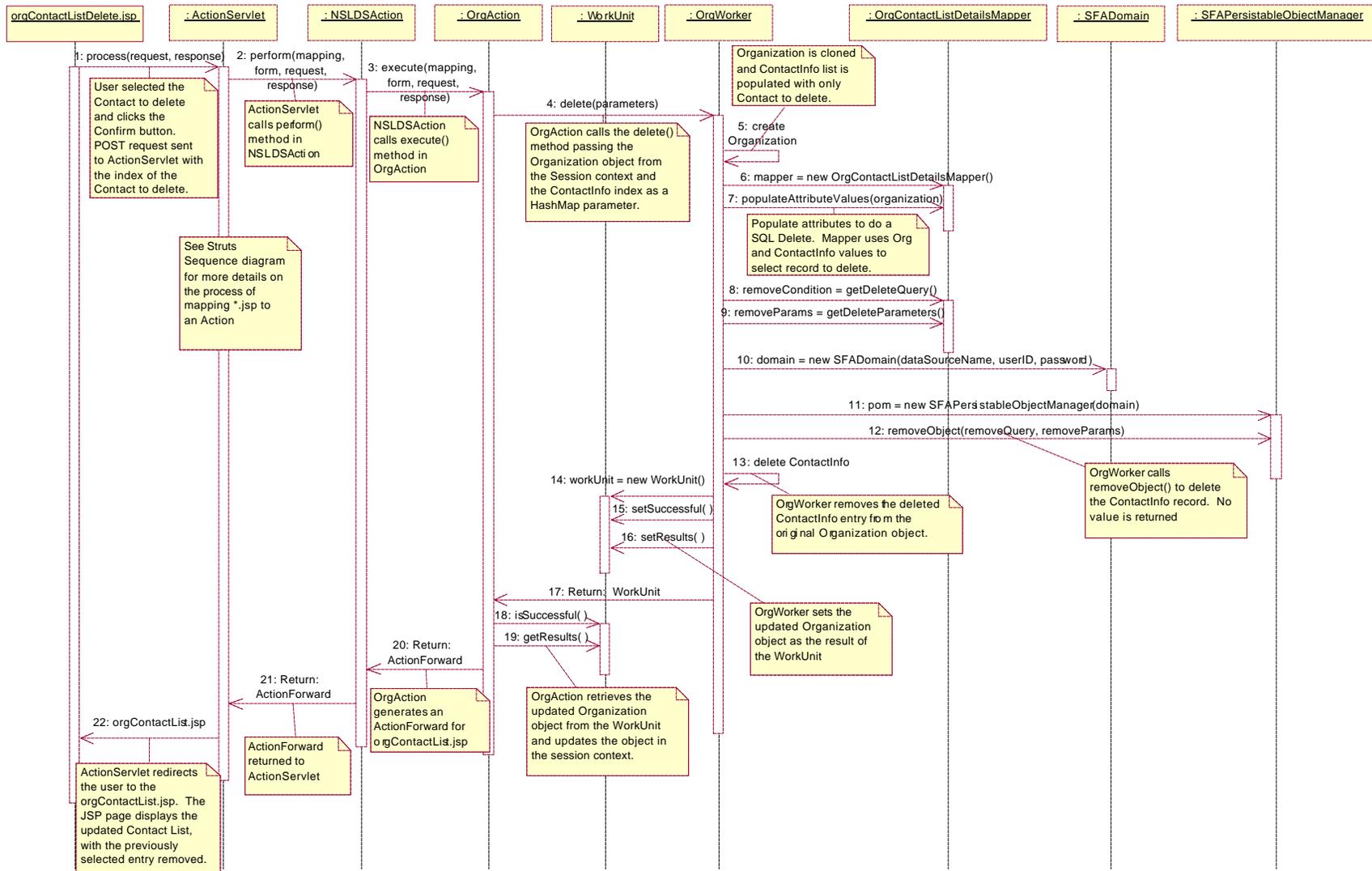
3.3.7 Logon



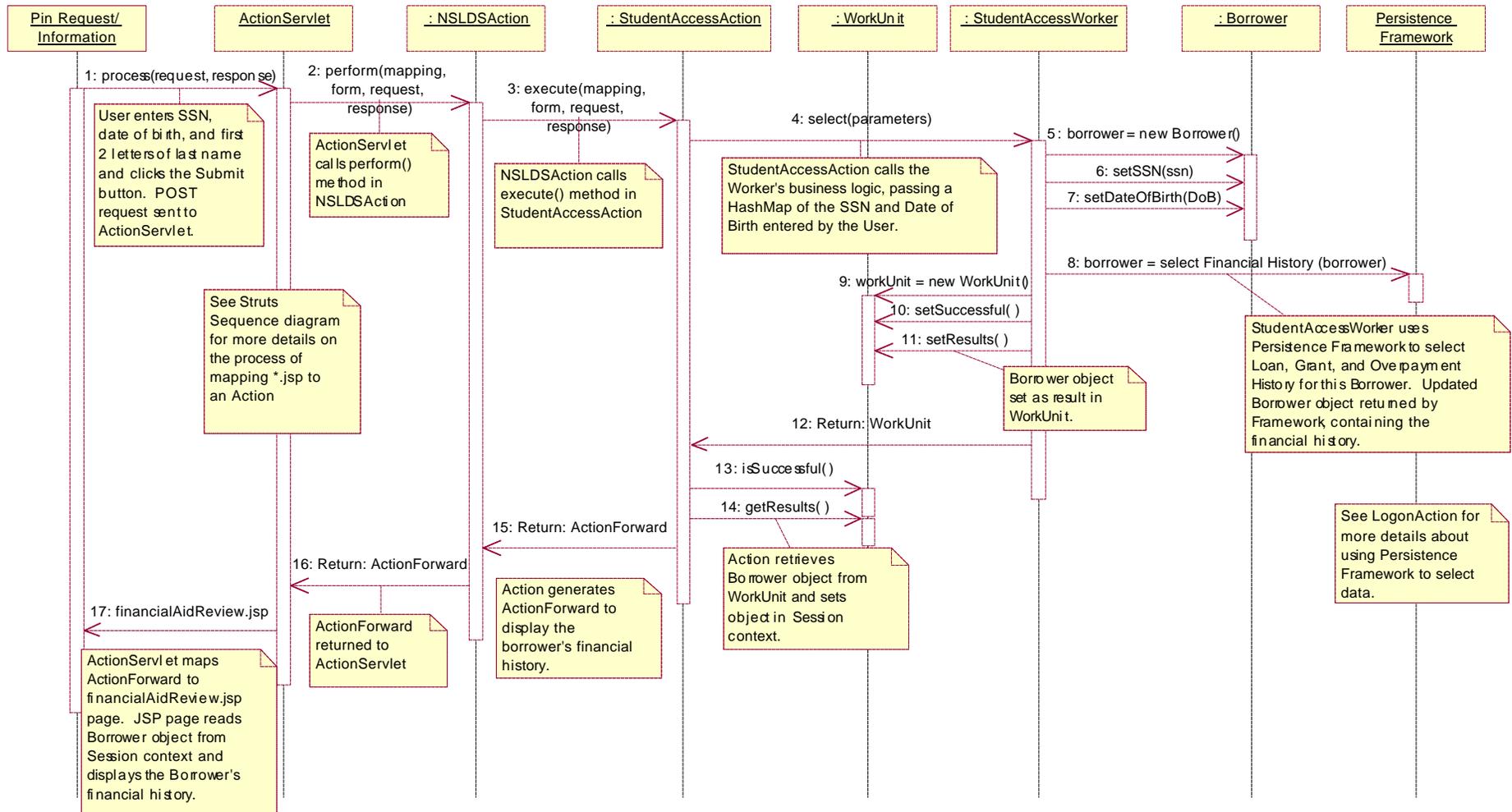
3.3.8 Organization - View Contact List



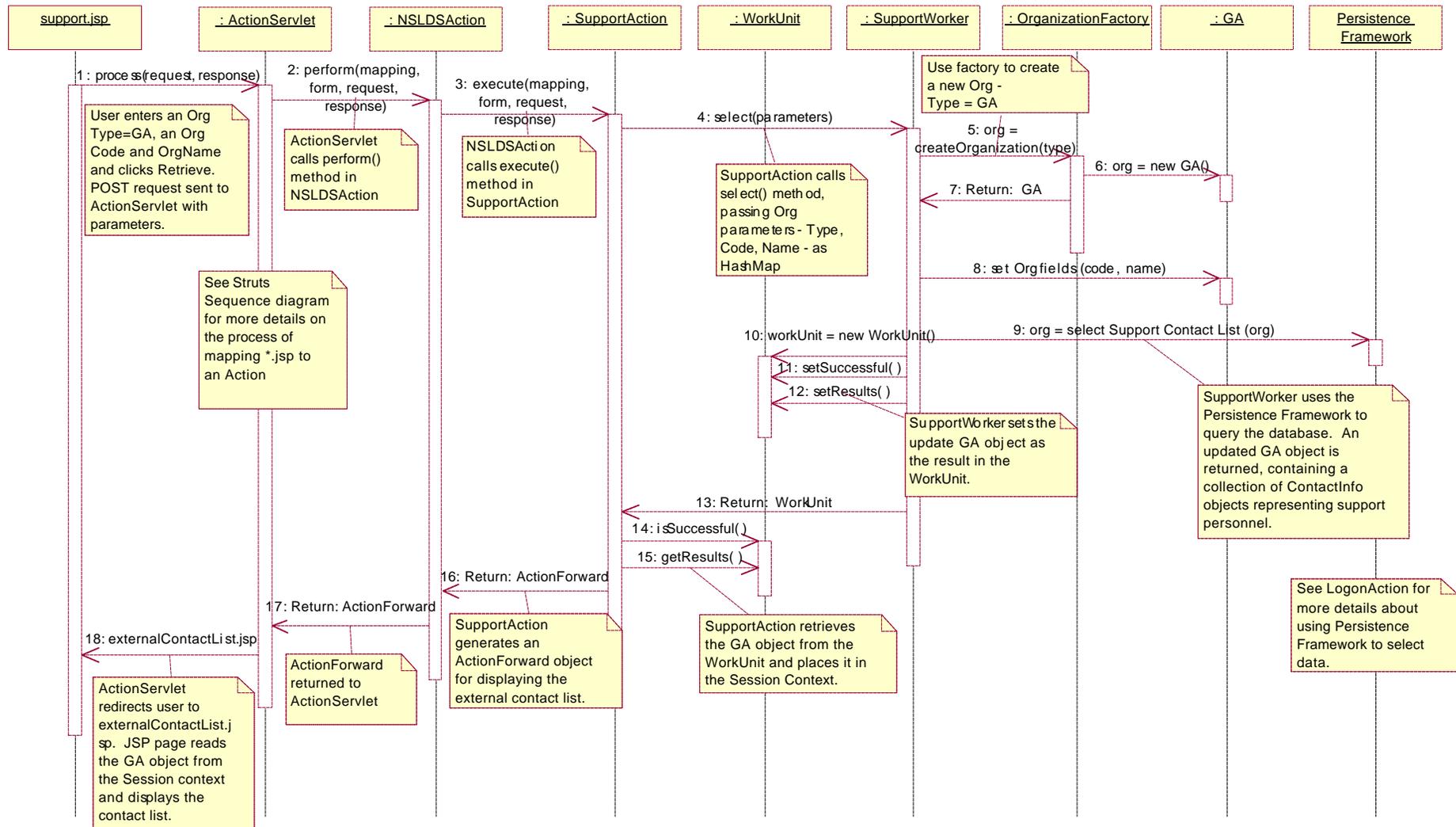
3.3.9 Organization - Delete Contact from List



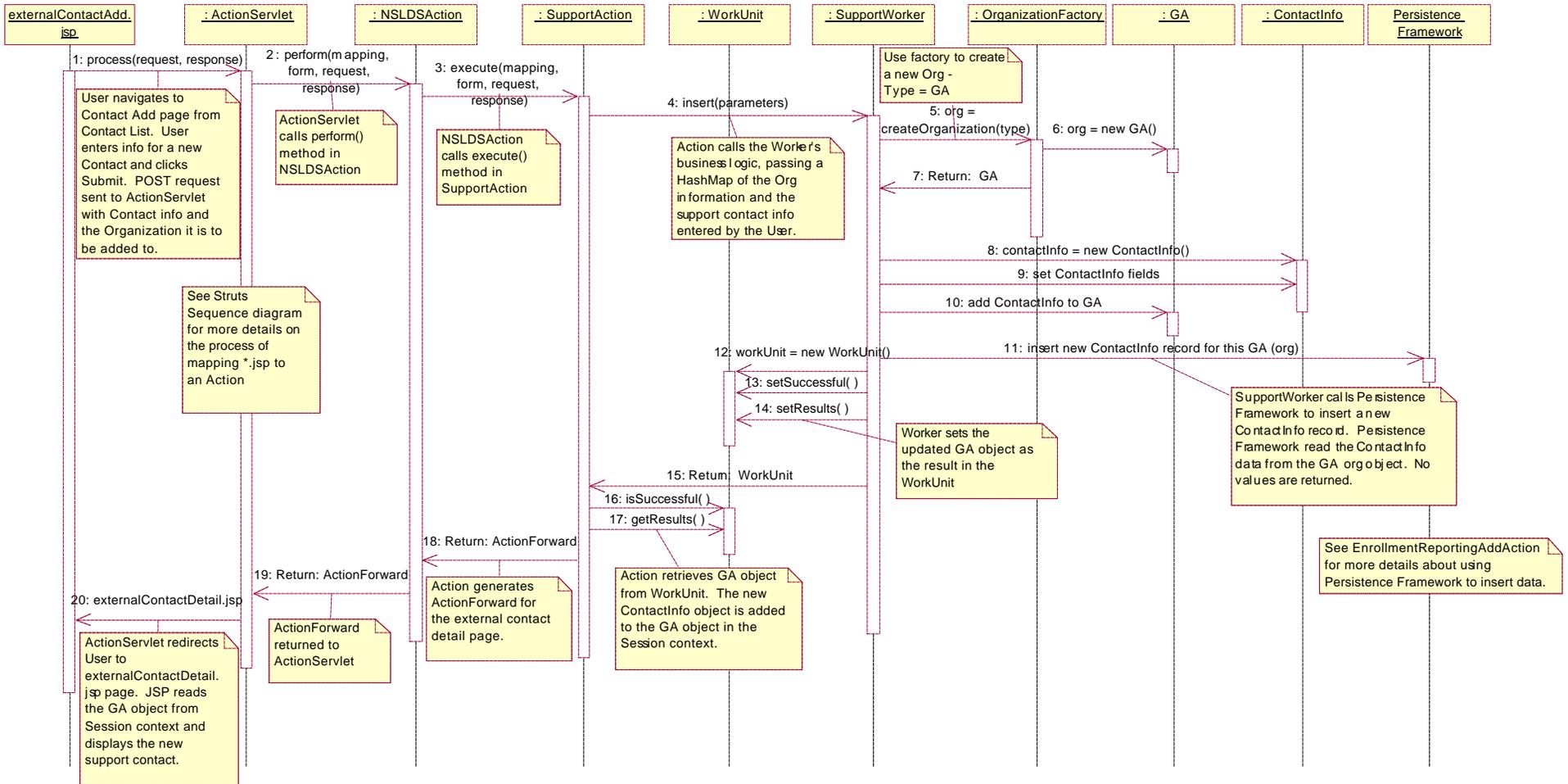
3.3.10 Student Access – View Financial Aid Review



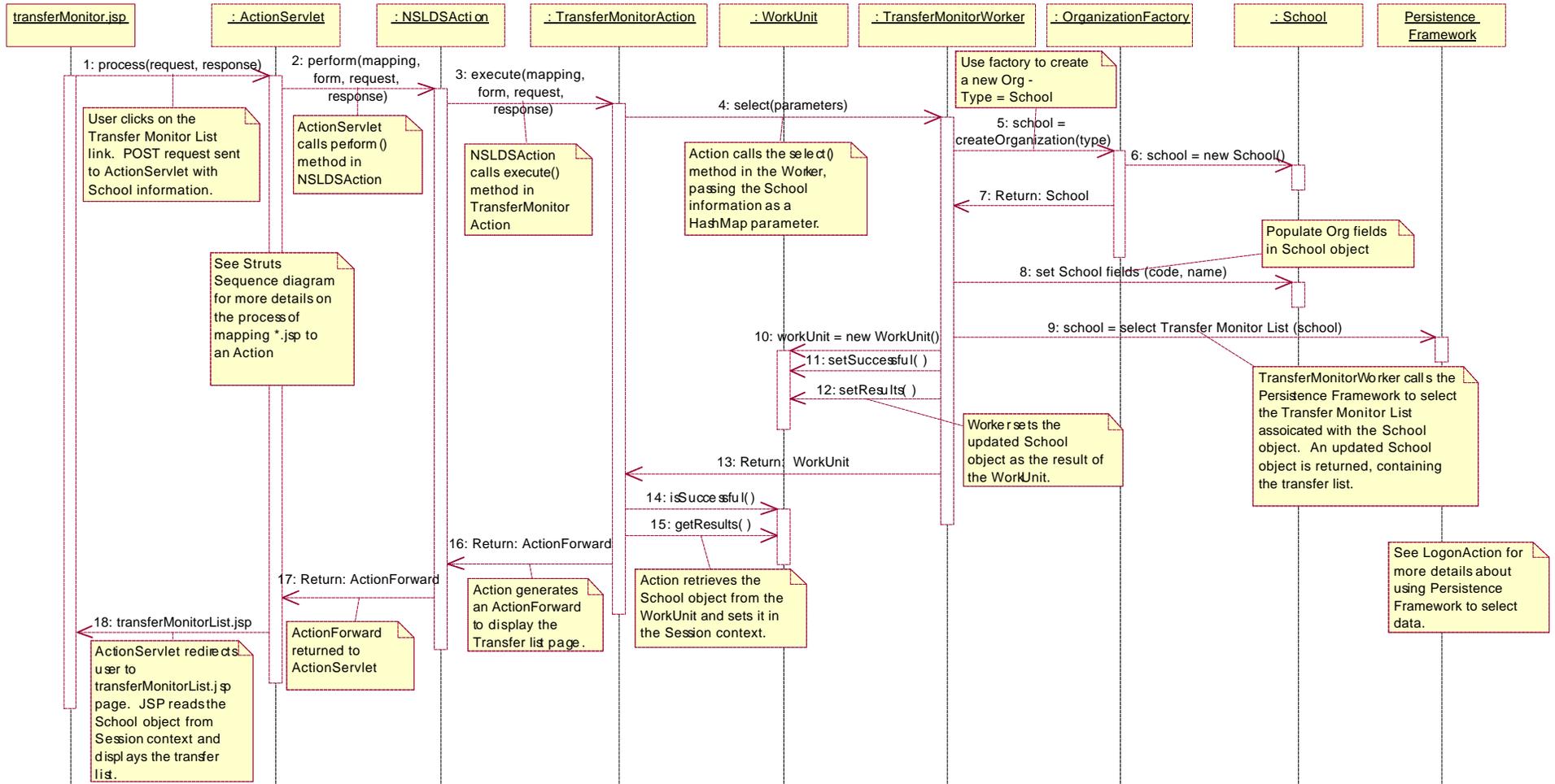
3.3.11 Support – View Contact List



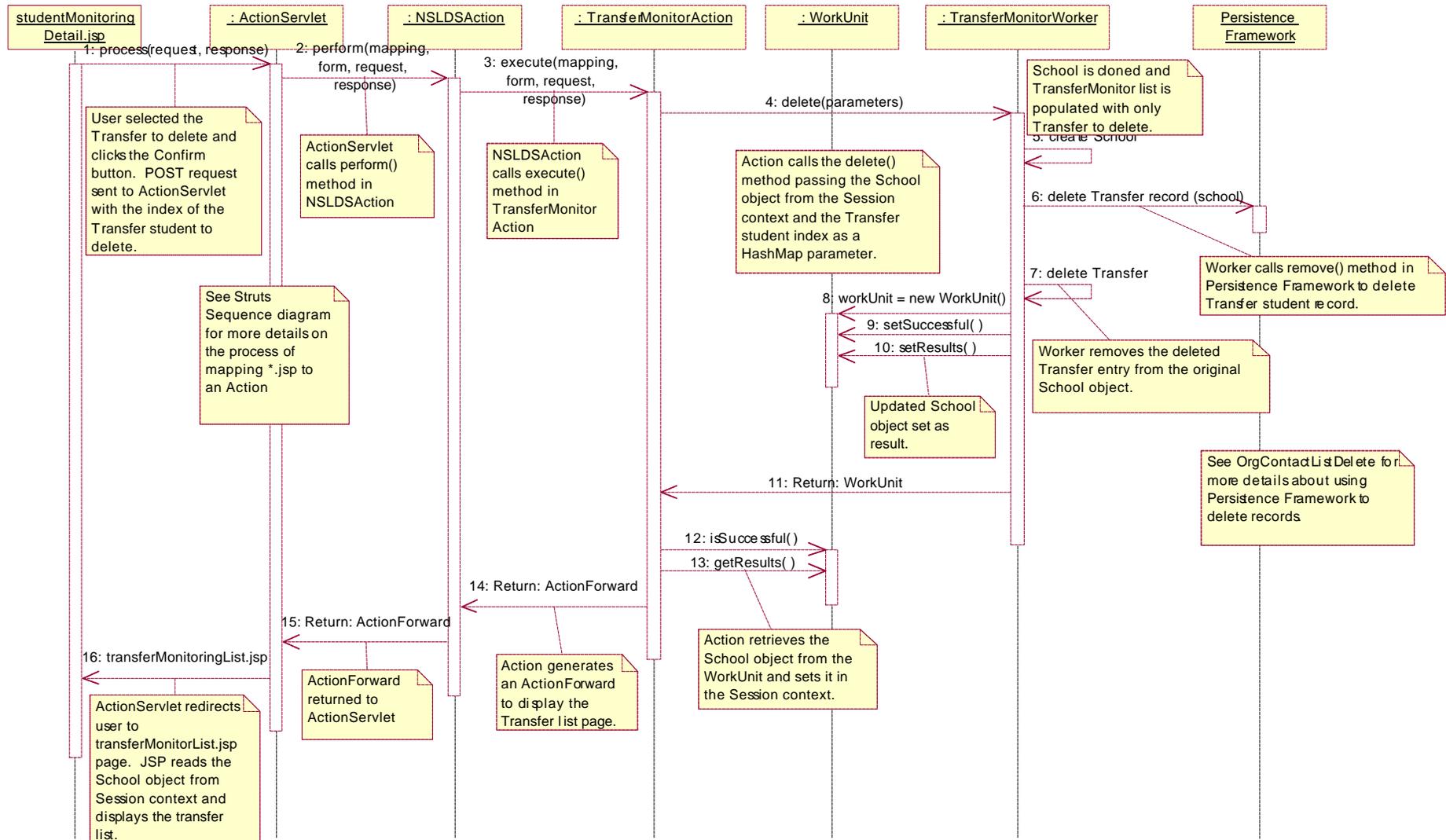
3.3.12 Support – Add Contact to List



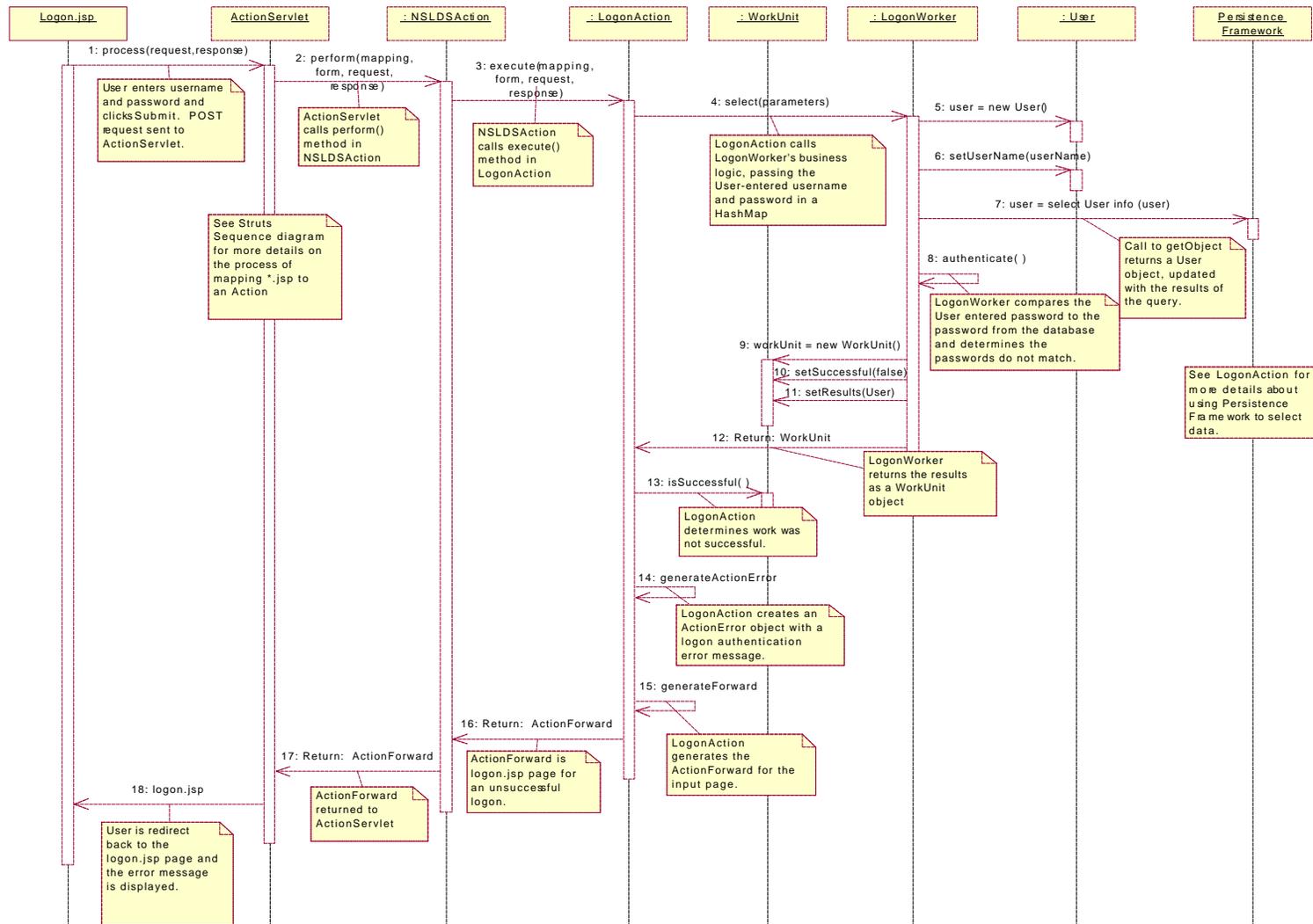
3.3.13 Transfer Monitor – View Transfer List



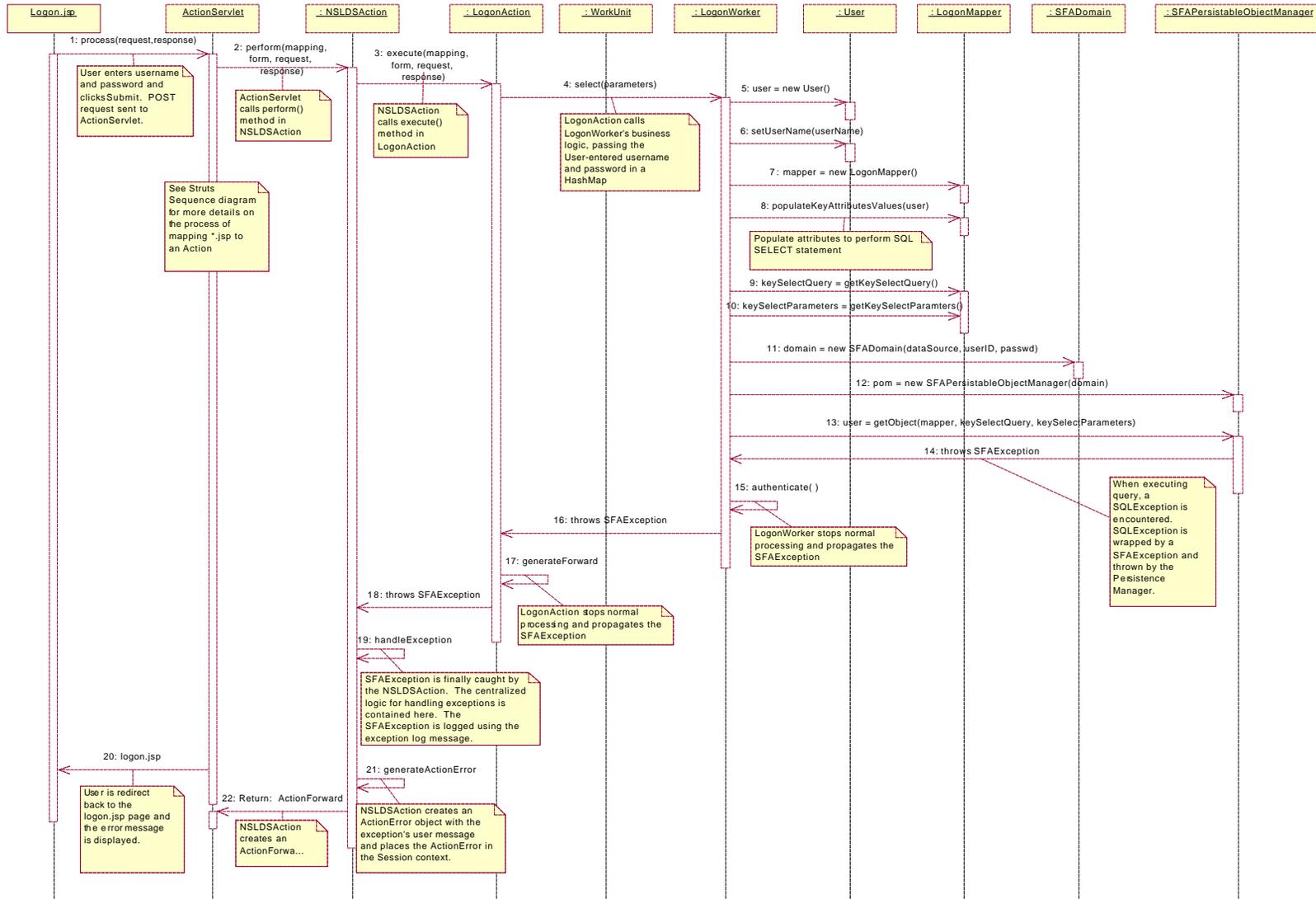
3.3.14 Transfer Monitor – Delete Transfer from List



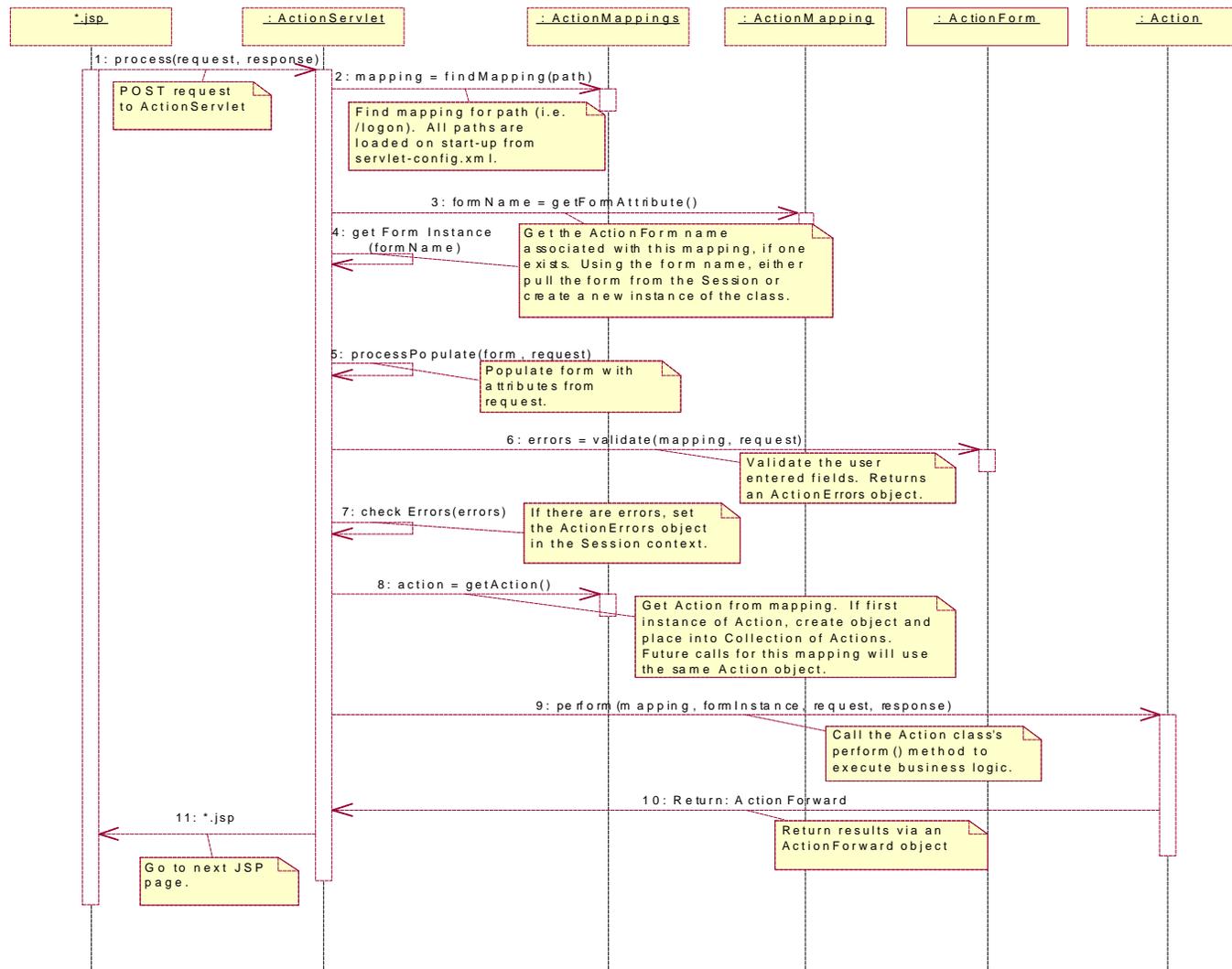
3.3.15 Application Error



3.3.16 Java Exceptions



3.3.17 Web Conversation Framework ActionForm errors



4 Architecture Layer Design

The goal of the ITA initiative is to promote code reuse, standardization of development, and application of best practices across all FSA system development projects. Reusable Common Services (RCS) components were created as part of this initiative to deliver reusable common functionality. The ITA Coding Standards and Best Practices guide are provided as Appendix C and Appendix D respectively. The Coding Standards Review Checklist is provided as Appendix E of this document.

NSLDS II application level uses RCS to provide the core application structure and common services for the NSLDS II system. This section will concentrate only on the application architecture parts that require configuration or customization to serve the purpose of building the system.

The following ITA RCS components will be used in NSLDS II web application:

- Web Conversation framework
- Exception Handling framework
- Logging framework
- User Session framework
- JSP Custom Tag Library framework
- Configuration framework
- Persistence framework

Specific implementation and detail usage of these frameworks can be found in the ITA component detailed design and user guide documents for each framework. These documents are available through the ITA.

4.1 RCS Web Conversation Framework

The purpose of the ITA Web Conversation framework is to provide a standard for developers and designers to apply the Model-View-Controller (MVC) design pattern for developing web applications. This framework allows different tiers of the web application to be created independently of one another, provide the ability to update the static strings displayed without updating and recompiling code, and facilitates internationalization of web pages. The Web Conversation framework enables developers to combine the use of JavaServer Pages (JSP) and Java Servlets to create dynamic pages.

The ITA Web Conversation framework provides for:

- Separation of control and business logic
- Automated form validation
- Multiple page forms (e.g. wizard steps lasting multiple pages).

4.1.1 Web Conversation implementation of MVC Design Pattern

In the Model-View-Controller software design pattern, the business logic/state is separated from the user interface, and from the program flow control. This separation allows web designers to create the web pages without having to know how to write application code, and vice versa. The separation of the presentation and logic also promotes code reuse.

The following diagram shows the Web Conversation framework components of the MVC model.

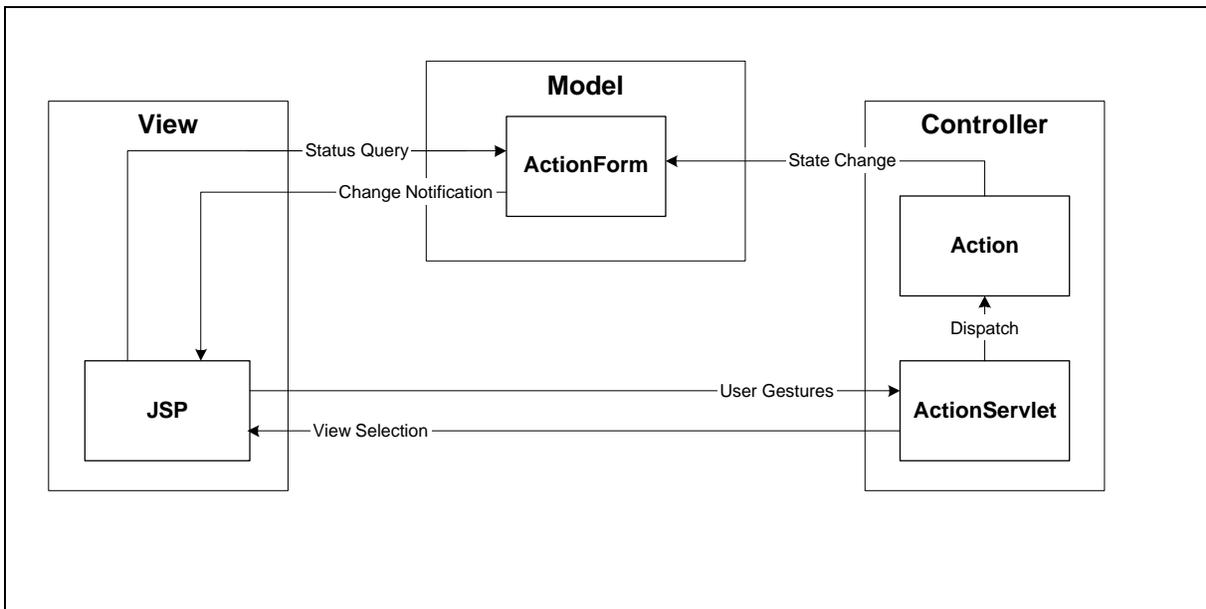


Figure 5, Mapping of Web Conversation framework to MVC Design Pattern

Model

The model encapsulates the business logic and represents the data in the system. A Java Object (typically a Java Bean) usually represents the Model component in the Web Conversation framework. The bean will represent details of the internal state of the system. The Model classes typically extend the abstract class ActionForm, which is a standard JavaBean with *getter* and *setter* methods that are used to access its state. All form classes are automatically populated and validated within this framework by the ActionServlet

If an error in processing occurs, an ActionErrors object can be created to hold all errors from the page and return those errors to the original requesting page so a user can correct the problem.

View

The view is represented by the JSP or HTML page presented to the user. The framework provides custom tag libraries that allow the JavaServer Pages to be displayed without embedding Java code in the page. The JSPs contain static HTML and offers authors the ability to insert dynamic content based on the interpretation (at page request time) of special action tags. There are also custom tag libraries that facilitate creating user interfaces to interact with ActionForm beans (part of the Model component). Custom tag libraries also allow the JSP developer to access and display business objects that are stored within the Session context.

Controller

The controller is responsible for controlling the flow of the program by processing updates from the model to the view (and vice versa). The controller is comprised of two components the `org.apache.struts.action.ActionServlet` object and classes that extend `org.apache.struts.action.Action`. The `ActionServlet` is loaded upon startup of the WebSphere Application Server and it will read all of the action mappings defined in the `struts-config.xml` file. The `ActionServlet` will act as a switchboard by reading incoming Uniform Resource Identifier (URIs) requests and matching that against an action mapping (defined in a `struts-config.xml` configuration file to find a controller (Action) class that will handle the request. The Actions are then responsible for interacting with the Model and forwarding control to the logic within the business objects.

For web-based applications, the `vhosts.properties`, `rules.properties`, and `queues.properties` files work in conjunction to direct all incoming requests for a particular application pointing to certain extensions to the appropriate application server, port number, and working directory. The `rules.properties` file is used to direct all incoming requests for web pages with an extension of `*.activity` to be forwarded to its associated `ActionServlet`. The `ActionServlet` will inspect the incoming URI and match it to an Action class listed in the action mapping in the `struts-config.xml` file. The Action object handles the request and forwards control to a view defined in the `struts-config.xml` file.

Aside from the `ActionServlet`, all other controller classes must extend the `org.apache.struts.action.Action` class. The Action class has a `perform` method that is used to process incoming requests and direct results to the next page. It is passed an `ActionMapping` object (loaded with values for this controller from the `struts-config.xml` file) that the `findForward()` method uses to direct the results to the appropriate 'next' page based on the action performed.

The following diagram details the interaction of the framework within the application server.

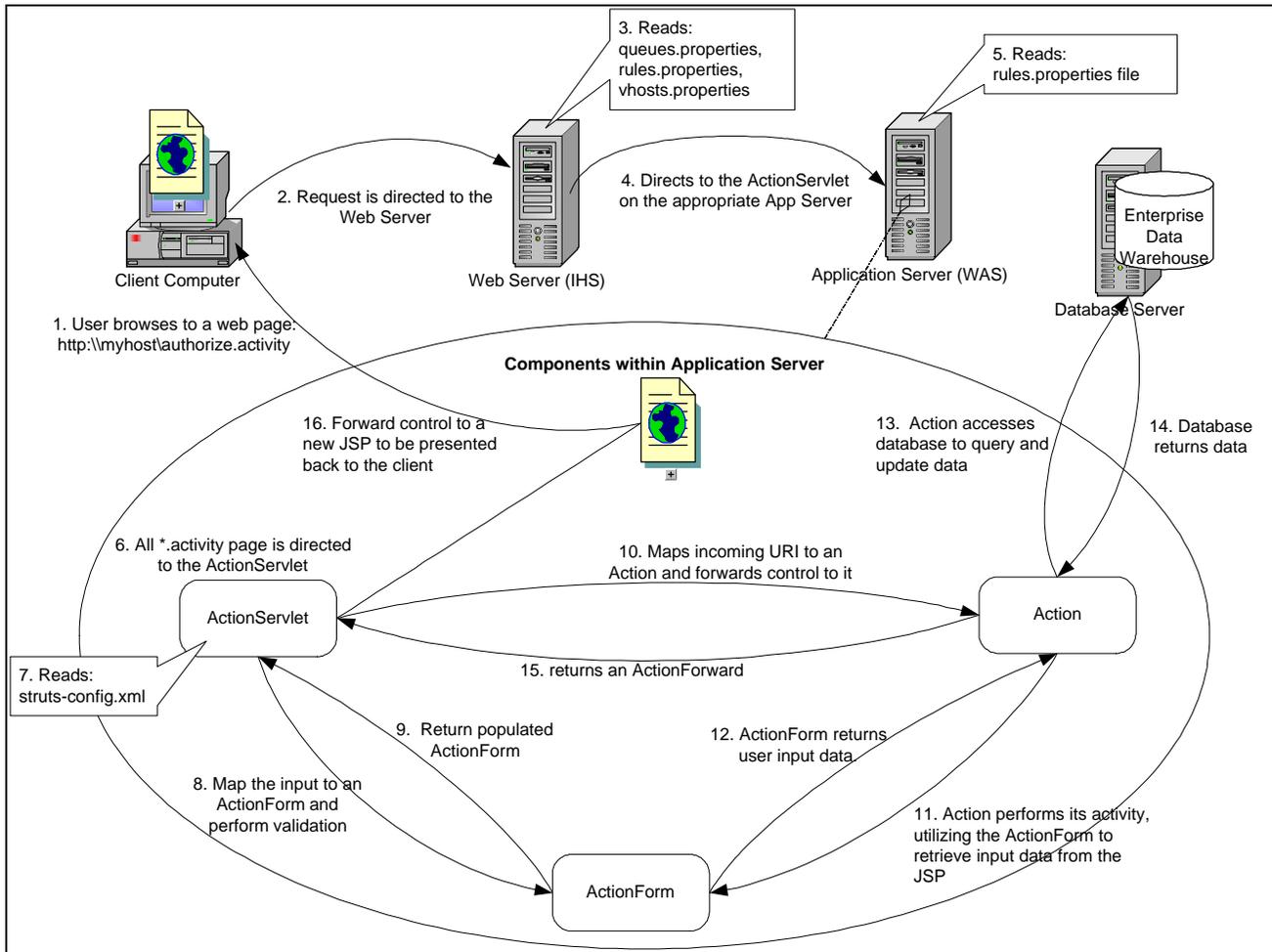


Figure 6, Web Conversation framework implemented in WebSphere Application Server container

4.1.2 Implementation Example

The following section provides example code for the logon functionality of the NSLDS II web application. The important parts of each component have been bolded to highlight the main focus of each component. More specific interaction points between these components and the web conversation framework architecture can be determined by referring to the sequence diagrams for the [Logon](#) section and for the Web Conversation framework provided earlier in this document.

The example code provided throughout the rest of this document will map to the following steps and the Figure 7 below:

- 1) When the user clicks submit, the logon.jsp file is directed to the ActionServlet.
- 2) The ActionServlet references the struts-config.xml file.
 - a) The struts-config.xml file maps the logon.jsp page to a LogonForm Java Bean.

- 3) The LogonForm acts as both a temporary holding place and error checker for web form data. The User ID and Password are passed via the Post method from the logon.jsp page to the LogonForm, via the ActionServlet.
 - a) The LogonForm performs minor error checking (e.g. valid length, characters) and returns the results to the ActionServlet.
- 4) If the User ID and Password are in the appropriate format, the ActionServlet passes the parameters to a LogonAction.
- 5) Within the LogonAction, the LogonWorker object is instantiated.
- 6) The LogonWorker object calls the Persistence framework.
- 7) The framework builds and runs a SQL query against the EDW for valid user information. The query results are returned to the LogonWorker and checked against the user's User ID and password for final validation. If the user was successfully authenticated, the LogonAction returns an ActionForward containing the next JSP page to be displayed, back to the ActionServlet. If logon was unsuccessful, the ActionForward refers to the input page, which in this case would be the logon.jsp page.

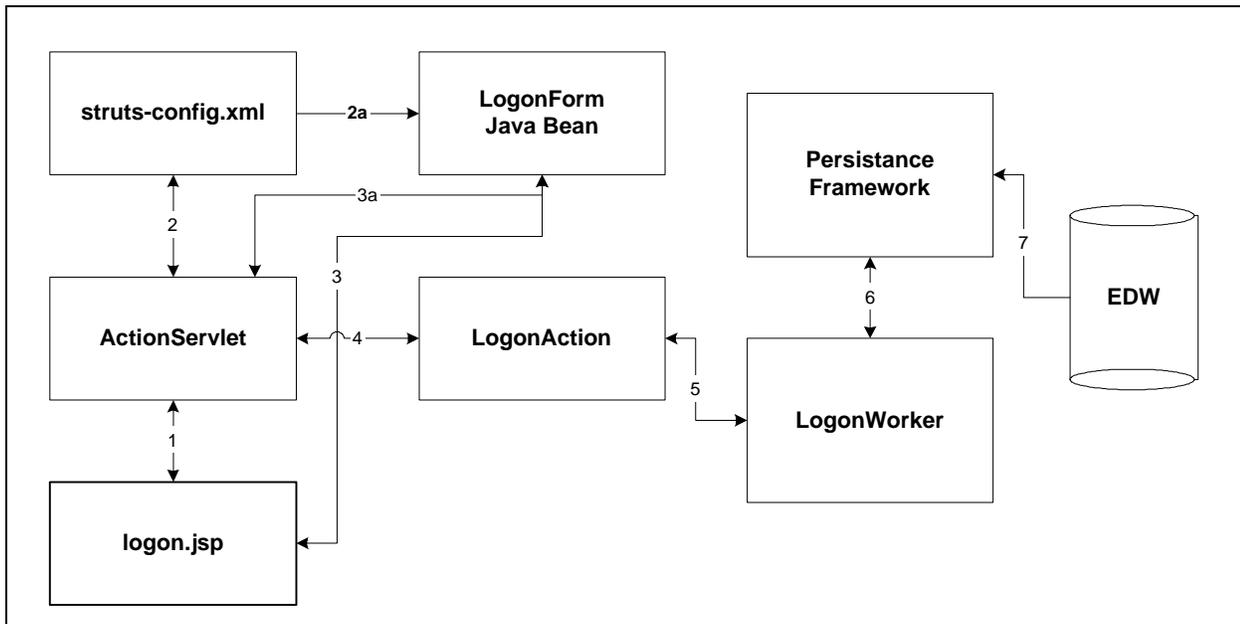


Figure 7: Logon Example

Examples for steps 5 - 7 will be provided in the [Data Access Layer section](#) (RCS Persistence framework) of this section. The ActionServlet code is not included as it is a standard class that is not modified and is a servlet loaded on the Application Server upon start up.

Step 1) logon.jsp (View):

This is an example JavaServer Page that will display the username and password input boxes and will interact with the form bean to display the form fields and submit button to the user. The `<html:form action="/logon" focus="username">` tag uses the struts-html custom tag library developed to bridge the gap between a JSP view and the other components of the web application.

On startup, the ActionServlet reads the struts-config.xml file and loads all of the mappings between the JSP actions (such as /logon) and the Struts Action and ActionForm objects. The action="/logon" directs the ActionServlet to look for the mapping <action path="/logon" to find the path to the ActionForm associated with this form. The JSP action "/logon" is mapped to a LogonAction and LogonForm.

The <bean:message> tag directs the JSP to read the static text from the NSLDSApplicationResource.properties file to find a match for the key. For example, <bean:message key="prompt.username"/> will have the resulting JSP display **User ID:** in that field. The <html:text> and <html:password> tags are renders as HTML <input> element of type text and password respectively. The property attribute for both tags will be the name used to populate the FormBean associated with this form. Refer to the model section for details on the interaction with these attributes.

```
<%@ page language="java" %>
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
<%@ taglib uri="/WEB-INF/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/struts-template.tld" prefix="template" %>

<!-- insert the template -->
<template:insert template="/templates/maintemplate.jsp">
  <!--insert the system restriction header -->
  <template:put name='restrictHeader' content='/template/restrictHeader.html' direct='true'/>
  <!--insert the links to pages that do not require log-on -->
  <template:put name="infoHeader" content="/template/infoHeader.html" direct='true'/>
</template:insert>

<html:html locale="true">
<head>
<title><bean:message key="logon.title"/></title>
<html:base/>
</head>
<body bgcolor="white">

<html:errors/>

<html:form action="/logon" focus="username">

<table border="0" width="100%">
  <tr align="middle">
    <td>
      <bean:message key="prompt.username"/> <html:text property="username" size="30"
                                     maxlength="30"/>
      <bean:message key="prompt.password"/> <html:password property="password" size="10"
                                     maxlength="8" redisplay="false"/>
    </td>
  </tr>
</table>

</html:form>
```

```
</body>  
</html:html>
```

Figure 8, logon.jsp pseudo code

NSLDSApplicationResources.properties (View):

This is an example properties file that can be customized with messages to be displayed in JSPs for the web application. Utilizing a properties file allows the developer to maintain all of the static messages from one location for easier updates. Properties files are also useful for internationalization purposes. Text strings could be converted into other languages so that one JSP can display text that international readers can understand.

```
prompt.username=<B>User ID:</B>  
prompt.password=<B>Password:</B>  
error.username.required=<li>Username is required</li>  
error.password.required=<li>Password is required</li>  
error.ssn.required=<li>Please enter either all 3 identifiers or SSN.</li>
```

Figure 9, NSLDSApplicationResources.properties pseudo code

Step 2) struts-config.xml (Controller):

The ActionServlet is the main component of the Controller and directs the flow of interactions in the framework. When the JSP containing the form is submitted, control is directed to the ActionServlet, which reads the struts-config.xml file to find the mapping for the form. Once the form has passed validation, the ActionServlet directs the request to an Action subclass, which will then process the request. Each JSP submits its form to *.activity, which is configured in the Application Server to direct the request to the ActionServlet. Then struts-config.xml file, which defines the form bean information, action mappings, and forwarding information, is read for the mappings. This is a sample portion of the file. The struts-config.xml file is read to produce the ActionMapping object passed to the perform method in the Action controller class.

For each Action, a mapping will have to be defined with the: path to the page, path to the Action class for this Action, path to the JSP for this Action. Forward elements will have to be defined that directs where the control of the application should be forwarded after the Action has completed. The forward elements include the name used to map to a forward and the path of the JSP to forward to. Global forward elements are defined with the name referenced by the application and the path to forward it to. The form-bean information is also defined in this file.

```
<action-mappings>  
  <action path="/logon"  
    type="gov.ed.fsa.nsls.LogonAction"  
    name="LogonForm"  
    scope="request"  
    input="/logon.jsp">  
  </action>  
  <forward name="success" path="/menu.jsp"/>
```

```
</action-mappings>

<global-forwards>
  <forward name="logoff" path="/logoff.activity"/>
  <forward name="logon" path="/logon.jsp"/>
</global-forwards>
```

Figure 10, struts-config.xml pseudo code

Step 3) LogonForm (Model):

This is an example LogonForm bean that will be used to capture the current state of the user information. The bean contains *getters* and *setters* to access the current state, validation functionality, and the ability to reset the forms to a default state.

```
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;

public final class LogonForm extends ActionForm
{
  // ----- Instance Variables
  private String username = null;
  private String password = null;

  // ----- Properties
  /** Return the username. */
  public String getUsername()
  {
    return (this.username);
  }

  /** Set the username.
   * @param ssn : The new ssn */
  public void setUsername(String username)
  {
    this.username = username;
  }

  /** Return the password. */
  public String getPassword()
  {
    return (this.password);
  }

  /**Set the password.
   * @param password The new password */
  public void setPassword(String password)
  {
    this.password = password;
  }
}
```

```
}  
  
// ----- Public Methods  
  
/**Reset all properties to their default values.  
 * @param mapping The mapping used to select this instance  
 * @param request The servlet request we are processing */  
public void reset(ActionMapping mapping, HttpServletRequest request)  
{  
    this.username = null;  
    this.password = null;  
}  
  
/**Validate the properties that have been set from this HTTP request,  
 * and return an ActionErrors object that encapsulates any  
 * validation errors that have been found. If no errors are found, return  
 * null or an ActionErrors object with no recorded error messages.  
 * @param mapping The mapping used to select this instance  
 * @param request The servlet request we are processing */  
public ActionErrors validate(ActionMapping mapping,  
    HttpServletRequest request)  
{  
    ActionErrors errors = new ActionErrors();  
  
    if ( (username == null) || (username.length() < 1) )  
    {  
        errors.add("username", new ActionError("error.username.required"));  
    }  
  
    if ( (password == null) || (password.length() < 1) )  
    {  
        errors.add("password", new ActionError("error.password.required"));  
    }  
  
    if ( (password.length() < 6) || (password.length() > 8) )  
    {  
        errors.add("password", new ActionError("error.password.length"));  
    }  
  
    // Enter additional password validation here  
  
    return errors;  
}  
}
```

Figure 11, LogonForm.java pseudo code

4) LogonAction (Controller):

This is an example LogonAction object that extends an NSLDSAction (see [RCS Exception framework](#) section for details on NSLDSAction) object to interact with a LogonWorker object to access the

persistence layer to obtain the requested information from the database. The LogonWorker example will be defined in the [Data Access Layer](#) section of this document.

```
// Package Statement
// Import Statements

public class LogonAction extends NSLDSAction
{
    public ActionForward execute(ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response)
    {
        // cast the form to a LogonForm object
        LogonForm myForm = (LogonForm) form;

        // create a HashMap of parameters to send to the LogonWorker
        // send a HashMap instead of the LogonForm to hide the details of the Web Conversation
        // Framework (i.e. LogonForm) from the LogonWorker
        HashMap params = new HashMap();
        params.put("userName", myForm.getUserName());
        params.put("password", myForm.getPassword());
        LogonWorker worker = new LogonWorker();
        // execute the select() method in the LogonWorker, passing the userName and password
        WorkUnit workUnit = worker.select(params);

        // check to see if the work the LogonWorker performed was successful

        // if it was successful then forward the user to the Main Menu JSP page
        if (workUnit.isSuccessful())
        {
            return mapping.findForward("mainMenu");
        }
        // else if the work was not successful, record the error and send the user back to the Logon
page
        else
        {
            ActionError error = new ActionError();
            // add logic to add logon error to list
            // go back to logon page
            return new ActionForward(mapping.getInput());
        }
    }
}
```

Figure 12, LogonAction.java pseudo code

4.2 RCS Configuration Framework

The RCS Configuration framework allows configuration data to be stored in the form of properties files, xml files, database tables, or any combination of the three. The framework creates one common interface for application developers.

The Configuration framework provides the following services:

- Configuration Data Load: the configuration data files are loaded into a static initializer that on system start-up loads the data from properties files, xml files, or database tables into one singleton object for central data access.
- Configuration Data Retrieval: returns the value based on the domain name and the tag/key name.

The Configuration framework will be used in the NSLDS II system to load the domain values for connecting to the database. The data source name, database user ID, and database password will be stored in the `dbDataSource.properties` file.

The Configuration framework will also be used to load the lookup values from the database. This will provide a standard programming interface to access the lookup values and will make sure the values are loaded only one time from the database. The configuration framework utilizes the singleton design pattern so that all class access the same instance of the framework. All values from the database and properties files will be loaded in the Configuration framework on initialization of the NSLDS II web application service.

The following is an example of the master properties file containing the database connection fields.

```
dataSource="java:comp/env/jdbc/NSLDSWebSiteDB"  
userID="system"  
password="123456789"
```

Figure 13, `dbDataSource.properties`:

4.3 RCS JSP Custom Tag Library Framework

The tag library framework provides a collection of commonly used JSP custom tag libraries for JSP developers to access. A JSP tag library is a collection of custom tags created by developers to aid the separation of control between the presentation and the application logic. Custom tags allow JSP developers to focus on the design of the page and presentation issues without having to know how to code to access enterprise services.

Custom tags are beneficial in that:

- They have access to all the objects available to JSP pages.
- They can modify the response generated by the calling page.
- They can be nested within one another, allowing for complex interactions within a JSP page.

The NSLDS II web application will leverage the following tag libraries:

| Custom Tag Library | Usage |
|------------------------|--|
| Logging Taglib | The logging tag library will allow JSPs to access the functionalities of the logging framework and will be used to provide logging functionality from the JSP. |
| Input Taglib | The input tag library combined with the struts-html tag library to ease form creation and usage. |
| Struts-html taglib | |
| Struts-bean Taglib | The struts-bean and logic tag libraries will allow the JSP developer to easily manipulate beans and perform logic operations in the web application. |
| Struts-logic taglib | |
| Struts-template taglib | The struts-template tag library will be used for creating a template for the entire web application for the header, footer, menu, and message areas. |

Table 5, Leveraged Custom Tag Libraries

4.4 RCS Exception Handling Framework

The ITA Exception Handling framework is designed to provide a mechanism for trapping and dealing with any erroneous or unexpected actions which applications may encounter during runtime. The Exception Handling framework also allows error message formatting standards to be developed and enforced to ensure proper information is gathered if a program failure occurs. This framework was created as part of ITA Release 2.0 before the name change from SFA to FSA; therefore, the framework's Java classes' name contains SFA.

The ITA Exception Handling framework enhances the ability of an application to define and trap errors that may arise as the application executes. The Exception Handling framework includes the following key components:

- SFAExceptionFactory
- SFAException

The best way to customize the SFAException for use in an application is by defining application-specific exception codes. The SFAException class is the only exception class instantiated, but by supplying it with different exception codes, different types of exceptions are created. By customizing it this way, the SFAExceptionFactory does not have to be extended to create different exception objects other than SFAException.

The SFAExceptionFactory is the class used to create exceptions. It should work in any application without modifications. This class is designed to work as a singleton (i.e. there is only one instance of it in the application server's memory). Therefore, a reference to the factory is obtained via the `getInstance()` method. It also contains some exception-centered utility methods, such as `getNestedException()`, which can be used to obtain the nested exception for all standard Java exception types. The SFAExceptionFactory exposes a method called `getMessage()` that when passed a SFAException will return the log message associated with that exception. The log message is pulled from a properties file containing mappings of exception codes to messages.

The SFAException class is the base exception, which can be customized on an application level. It has *setters* and *getters* methods for each of its main properties, such as `methodName` and `className`. The developer will use the SFAException and SFAExceptionFactory classes when catching exception thrown by Java components. The original Java exception will be caught, then a SFAException is created using the SFAExceptionFactory. The exception code depends on the type of exception of the original Java Exception.

4.4.1 NSLDS II Exceptions

There are different categories of exceptional conditions that may occur during the execution of the NSLDS II web application. The different categories where exceptions could occur are:

- Field-Level validation
- Application level exceptions
- System level exceptions

- JSP error pages

Two of the exception categories, Field-Level Validation and JSP Error Pages, do not use the Exception Handling framework levels, but rely on the Web Conversation framework for proper handling of system and application errors. The fundamental idea of exception handling for all categories is the centralization of logic to catch the exceptions, produce meaningful log and user messages, and redirect control to the appropriate response.

4.4.2 Field-level Validation

In using the Web Conversion framework, all field-level validation of the JSP pages will be executed within the validate method of the associated ActionForm Java object (i.e. a logon.jsp page will have a LogonForm to validate the web form). This validation consists of checking each field on a web form to ensure the proper data is being sent by the user and all required fields are populated. The ActionForms use an ActionError object to record the validation errors that occur in the validate method. Every validation error gets recorded and an ActionErrors collection object is sent back to the ActionServlet. The error messages displayed to be displayed will be read from the NSLDSIIApplicationResources.properties file and show to the user.

The following is a diagram of the sequence for validating form fields and displaying the appropriate information to the user when an error is encountered.

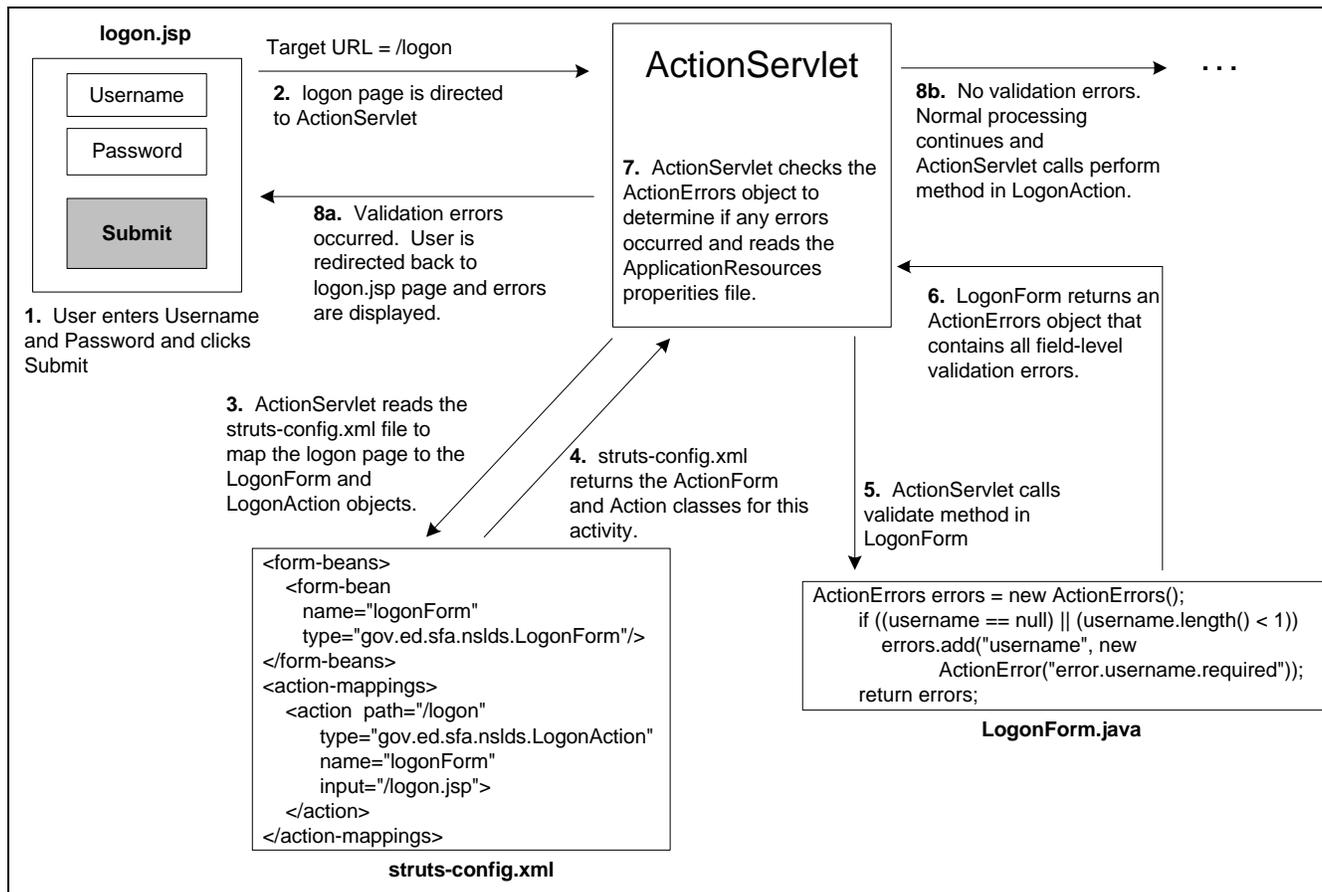


Figure 14, Field-Level Validation

4.4.3 Application and System Level Exceptions

Application and system level exceptions in NSLDS II can occur during the life of the application. These exceptions will occur at well-defined places and will therefore be caught and wrapped with the `SFAException` class. Java Runtime exceptions are unchecked exceptions that could not be predicted during the design phase or can occur at any place in the code, such as a `java.lang.NullPointerException`. A `NullPointerException` is thrown when an action is performed on a null object.

Application and system level exceptions will differ by their exception code value used. The exception code value is used to define the type of exception when a `SFAException` is created. It also maps to the appropriate properties files and is used to retrieve the log and user messages associated with the exception.

The following tables list sample application and system level exceptions and the possible log messages and exception codes for each. A complete list of existing exception messages was not provided and will have to be defined in a future version.

| Exception Name | Code | Example Message |
|--------------------------|------|---|
| Authentication Exception | 1001 | The username and password could not be verified. |
| Add Exception | 1002 | The disbursement amount could not be added to the loan details. |
| Delete Exception | 1003 | The loan record could not be deleted. |
| Update Exception | 1004 | The loan contact information could not be updated. |

Table 6, Sample Application Exception Code

| Exception Name | Code | Example Message |
|-------------------------|------|---|
| SQL Exception | 2001 | A SQL error occurred. |
| JNDI Exception | 2002 | A JNDI error occurred. |
| Communication Exception | 2003 | Could not connect to external resource. |
| FileNotFoundException | 2004 | Could not locate resource file. |
| Undefined Exception | 3001 | A general error occurred. |

Table 7, Sample System Exception Code

The difference between these two types of exceptions is very important since they are logged as different levels and indicate different severity of the exceptions. Usually, application exceptions occur due to normal behavior of the system. They are used to indicate an action cannot be completed, but allow the system to exit gracefully. System exceptions, on the other hand, often indicate that something in the execution environment has been changed or no longer working. They tend to be much more severe and may affect not only the current action executing, but also subsequent actions. Application level exceptions will be logged at the ERROR level, and System level exceptions will be logged at the FATAL level. Please refer to the next section, [RCS Logging framework](#) on the different logging levels available.

4.4.4 Implementation Example

The NSLDSAction class will handle all Java exceptions, including SFAExceptions and Runtime exceptions. This class extends the Web Conversation framework's Action class and is the base class for all actions in the NSLDS II application. The NSLDSAction class overrides the perform method of the Action class. When the ActionServlet calls the perform method in an application Action (such as LogonAction) the call gets forwarded to the NSLDSAction, which in turn calls the execute method of the LogonAction. The NSLDSAction provides the central exception handling logic around the call to the LogonAction and will trap and log all exceptions that propagate back to the NSLDSAction. The following is pseudo-code for the NSLDSAction.

NSLDSAction (Controller):

There will be two properties files used to retrieve the exception messages. Each properties file will be a listing of the same exception code keys, matched to the appropriate exception messages. The logmessages.properties file will include the messages used by the Logging framework to record the exceptions as either ERROR or FATAL. The NSLDSIIApplicationResource.properties file, which is defined by the Web Conversation framework, contains the user messages.

The NSLDSIIApplicationResources.properties file is used in conjunction with the ActionError object, which will hold the exception code used to retrieve the error message from the properties file. The user messages will be displayed on the page represented by the ActionForward object. As an example, if a SFAException with the exception code equal to 2001 (SQL Exception) is encountered, the exception handling code will read from both the logmessages.properties file and the NSLDSIIApplicationResource.properties file.

The [RCS Logging framework](#) (see next section) will record the log message into the log file and the user message will be displayed using the ActionError object. There will also be general messages in the log and user message properties files for undefined exceptions that are not of type SFAException, such as the NullPointerException, but encountered during runtime.

```
public ActionForward perform(ActionMapping mapping, ActionForm form,
                            HttpServletRequest request, HttpServletResponse response)
{
    // ActionForward is returned to redirect user
    // If exception occurs, then redirect back to input page or error page
    ActionForward forward = null;

    // try/catch around call to LogonAction execute() method
    try
    {
        // call the execute method in LogonAction – returns ActionForward
        forward = execute(mapping, form, request, response);
    }
    catch (SFAException sfae)
    {
        // determine if application or system exception
        // log SFAException
        handleException(sfae);
        forward = Input Page with exception message
    }
    finally
    {
        // always return a forward page, even when handling exceptions
        return forward;
    }
}
```

Figure 15, NSLDSAction.java pseudo code

The diagram shows the handling of an exception thrown by the database. The exception is originally wrapped by a SFAException in the Persistence framework and then propagated back to the NSLDSAction, where it is handled.

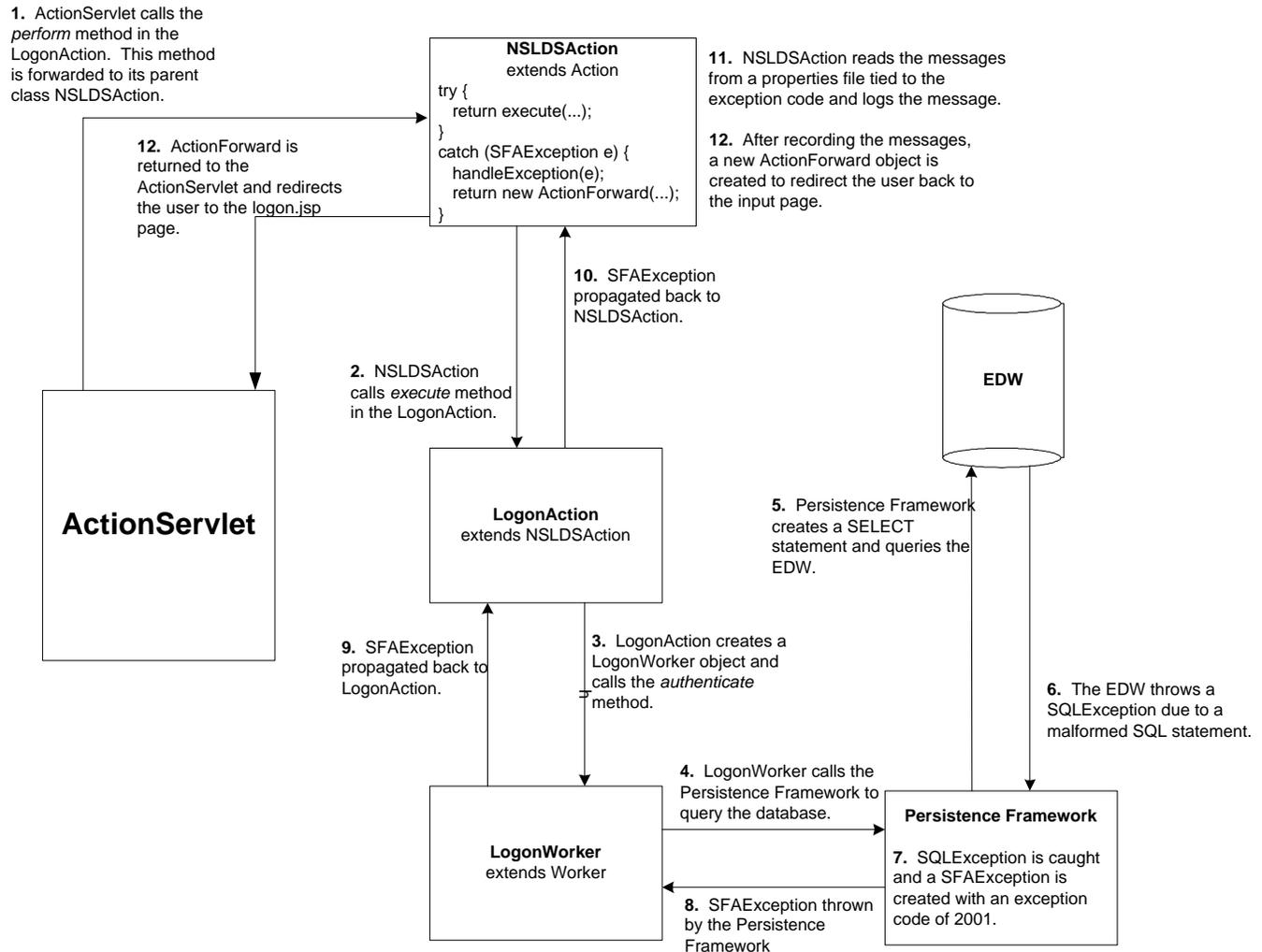


Figure 16, SFAExceptions Usage

4.4.5 JSP Error Pages

In addition to field-level validation and SFAExceptions, there may also be exceptions that occur in the ActionServlet code, JavaServer Pages, and the connection between the two. Any exceptions that occur in the ActionServlet and in the connection to it would be system level exceptions that cannot be caught by NSLDSAction. NSLDSAction cannot capture these exceptions as the failure will have occurred and the activity stopped before the call to NSLDSAction has been made.

The JSP specification allows the developer to define error pages that are included in the header of the JSP page. If an exception is thrown by the JSP or code the JSP calls, then control will be directed to the error page. The error page will handle the exception appropriately, displaying a message to the user and logging the exception. There will be one global errorPage.jsp file that will contain the centralized error handling code. There is only one errorPage.jsp file as it is a requirement of the current system. errorPage.jsp will utilize the Logging custom tag library (please refer to the [RCS JSP Custom Tag Library framework](#) section) to call the NSLDSLogger and log the exception stack trace (at the FATAL level – please refer to the [RCS Logging framework](#)) read from the JSP implicit object "exception" that is available to JSP error pages.

The "exception" object has *page* scope so if there are exceptions thrown by the ActionServlet for multiple users, each user will see a different exception object. Since the error page will not be able to determine the cause of the exception, the message displayed to the user will be a generic system exception message that is stored statically in the errorPage.jsp file. The following diagram details an exception thrown by the ActionServlet and handled by the errorPage JSP.

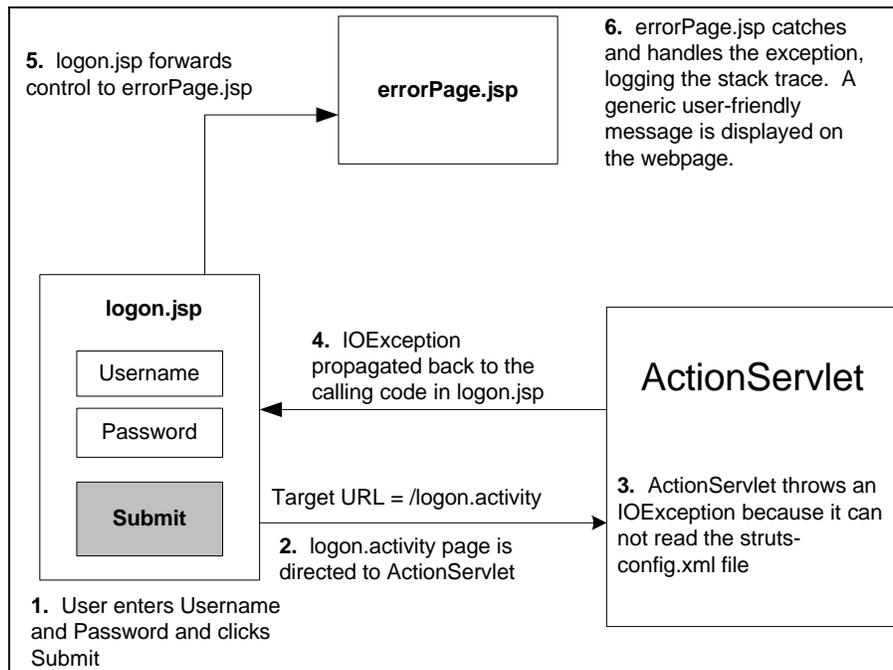


Figure 17, JSP Error Page

4.5 RCS Logging Framework

The ITA Logging framework enhances the current logging capabilities of the WebSphere Application Server, by allowing programmers to dynamically set logging and tracing functionality without modifying source code. The Logging framework also allows for formatting standards to be developed and enforced for log files to ensure proper information is gathered if a failure occurs.

The ITA RCS Logging framework provides the following features:

- Simple logging API
- Background logging (Configurable)
- Multiple message severities
- Configuration can be modified on-the-fly while the system is running

The Logging framework allows the programmer to log messages easily through a simple API. It contains one main object called Syslog that provides a set of calls to log in a variety of manners. Most of the operation of Syslog is set through the rcs.xml configuration file the component uses. The logging level and output location are defined in the rcs.xml file and the file can be configured while the Application Server is running. Any changes made to the configuration file will not take effect though until the instance of the NSLDS service has been restarted.

There will be four message levels of the Logging framework used in the NSLDS II web application (in order of severity): DEBUG, INFO, ERROR, and FATAL. These levels allow the Logging framework to be used as a debugging tool during development and as a troubleshooting tool when the application goes into production. During development and testing, the messages may be simply sent to a single log file. When the system is moved into production, log messages can be split up by severity levels (Fatal messages may result in someone being paged, for instance) and log files may be rotated every night and archived. These kinds of configuration changes do not require changes to the code and can even be made while the system is running.

In the Production environment, the rcs.xml file will define the location of the NSLDS web application's system log file. The file will be stored in: /opt/WebSphere/AppServer/logs/ and the file name will be NSLDS_System.log. The storage location is configured in the WebSphere Administration Console. The log file is rotated regularly and CSC's Unicenter TNG tool can integrate with the RCS Logging framework by monitoring the log file. When the monitoring tool encounters a FATAL message in the log, it should trigger an escalation action to CSC personnel.

rcs.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Syslog defaultMask="ERROR" backgroundLogging="false">
<Logger class="com.protomatter.syslog.FileLog" name="NSLDS_System">
  <fileName>/opt/WebSphere/AppServer/logs/NSLDS_System.log</fileName>
  <autoFlush>true</autoFlush>
  <stream>System.out</stream>
  <Policy class="com.protomatter.syslog.SimpleLogPolicy">
    <channels>ALL_CHANNEL</channels>
    <logMask>INHERIT_MASK</logMask>
```

```
</Policy>
<Format class="com.protomatter.syslog.SimpleSyslogTextFormatter">
  <showChannel>>false</showChannel>
  <showThreadName>>false</showThreadName>
  <showHostName>>false</showHostName>
  <dateFormat>HH:mm:ss MM/dd</dateFormat>
  <dateFormatCacheTime>1000</dateFormatCacheTime>
  <dateFormatTimeZone>America/New_York</dateFormatTimeZone>
</Format>
</Logger>
</Syslog>
```

4.5.1 Logging Usage Standards

This section will detail the levels and scenarios for using the ITA Logging framework in the NSLDS II application. NSLDS II JavaObjects and JSPs will be utilizing the four logging levels defined above. These four levels provide a mechanism to differentiate the degree of importance of the information logged. The DEBUG and INFO levels are typically used during development and testing and turned off when the application is in production. While this will reduce the size of the log file created, the log files will still need to be rotated on a nightly basis. Log files will be kept for a minimum of five days for any troubleshooting purposes before being deleted. This section will define which phase of the application lifecycle the logging level is required and provide guidelines as to the type of information to be captured in the log message.

The usage guidelines of the Application Programming Interfaces (API) below uses a custom NSLDSLogger described in [Appendix F](#). This logger is an application specific wrapper around the Syslog class provided by the Logging framework. The developer would call *NSLDSLogger.getLogger().logLEVEL("short message", "detailed message");* where LEVEL would be the logging level (logDebug, logInfo, logError, logFatal).

Refer back to the [Exception Handling](#) section for more details on system and application level exceptions, which utilize the ERROR and FATAL logging levels.

4.5.1.1 DEBUG

The DEBUG logging level is primarily used by the developers to replace System.out.println() statements used to test and debug code. These log statements will only be recorded during the development and test phases of the application lifecycle and should be used thoroughly when writing the Java code. The key advantage of the DEBUG log statements over System.out.println() statements is they do not need to be removed when migrating to a production system. The DEBUG level can simply be turned off when it is no longer necessary to view these statements.

The DEBUG logging level will be used in the following environments:

- Development
- Unit Test
- Assembly Test

- System Integration Test
- Training

Usage Guidelines

The developers should use the DEBUG log statements wherever they feel it is necessary. As stated before, there is minimal overhead in using DEBUG statements and the more information provided via the logging process will greatly reduce the amount of time in completing bug fixes. The following table is a sample list of scenarios where the DEBUG log statements could be included:

| Example Scenario | Example Message |
|--|--|
| Entry/Exit points of complex methods | Entering Loan Details update method. |
| Test conditions used in logical statements | Start Date = 9/23/1998 and End Date = 2/15/2003 |
| Before and after data transformations | SSN = 123456789 SSN = 123-45-6789 |
| Results of calculations | Subsidized Loans Pending disbursements = (LOAN_AMOUNT - LOAN_TOTAL_DISBURSEMENT_AMOUNT - LOAN_TOTAL_CANCELLATION_AMOUNT) = 250 |
| SQL statements | LogonWorker - password query = "SELECT password FROM table WHERE user = userID" |
| Load external resources | Loading ApplicationResources.properties file |

Table 8, Debug Usage Scenarios

Usage Example

```
NSLDSLogger.getLogger().logDebug("This is a short debug message", "This is a more detailed debug message");
```

The developer can log information that may be useful to debug later for checking to see what the application had processed. While coding LogonWorker, the developer would be able to put in `NSLDSLogger.getLogger().logDebug("In LogonWorker", "LogonWorker will access persistence framework to query = 'SELECT password FROM table WHERE user = userID'");`

4.5.1.2 INFO

The INFO logging level should be used to record major points in the application. The main purpose of the INFO level is to record the roundtrip of requests made to the system. For example, when a user clicks on a submit button for a web form, a request is generated to process the form and sent to the Application Server. This request should be logged using the INFO level. When the request is finished processing and a response returned to the user, the response should also be logged using the INFO level. The INFO level is utilized mostly by application framework classes and Java Objects and will not be used heavily by the JSP developers.

The INFO logging level will be used in the following environments:

- Development
- Unit Test
- Assembly Test
- System Integration Test
- Training

Usage Guidelines

The INFO level may be utilized to record any changes made to the system through field updates, especially those involving monetary adjustments. If a system properly utilizes INFO level logging, it will be possible to trace through the major events of an actions as it executes during runtime. INFO level messages should be written so that they may be understood by more than just the Java developer. The idea is to have a high-level snapshot of the system from input to output. The following is a sample list of scenarios where the INFO log statements could be included:

| Example Scenario | Example Message |
|-------------------------------------|--|
| Input requests and Output responses | User trying to logon with username = test and password = 123456789 |
| Any changes to monetary fields | Loan principal amount changed from 2,300 to 3,500. |
| Updates to user/system fields | Student contact name update to John Smith |
| Creation/Deletion of records | User added disbursement payment of 150 |

Table 9, INFO Usage Scenarios

Usage Example

```
NSLDSLogger.getLogger().logInfo("This is a short info message", "This is a more detailed info message");
```

The developer can log activities that occur in the system later for checking to see what the application had processed. While coding LogonAction, the developer would be able to put in

```
NSLDSLogger.getLogger().logInfo("In LogonAction", "User trying to logon with username = xyz");
```

4.5.1.3 ERROR

The ERROR and FATAL logging levels are closely linked together in that they are both used when exceptions occur during runtime. The main difference is the ERROR level should be used to log application exceptions where as the FATAL level should be used to record system level exceptions. Application exceptions will mostly be generated from field-level validation exceptions and conditions that break the normal execution. These circumstances, while exceptions, are mostly normal behavior of the system. They usually do not require escalation to support personnel, but do require proper logging and robust messaging to inform the user of the condition. Please refer to the Exception Handling framework which implements the messaging functionality.

The ERROR logging level will be used in the following environments:

- Development
- Unit Test
- Assembly Test
- System Integration Test
- Performance Test
- User Acceptance Test
- Production
- Training

Usage Guidelines

The ERROR level is used to record erroneous situations when exceptions arise in application code during runtime. The log message should include the circumstances behind the exception and the location the exception occurred, as well as any additional information provided by the application developer. However, the message should not be logged by the developer in the application code. The developer will only be responsible for throwing the exception (again, see the Exception Handling section). The exception will be propagated to a central location in the system, the NSLDSAction, where it will be handled by the NSLDSAction and the exception message logged. All application exceptions should be logged with the ERROR logging level.

| Exception Name | Custom Message |
|--------------------------|--|
| Authentication Exception | User not authenticated with username=test and password=1233456 |
| Add Exception | New disbursement amount record could not be added on the Loan Activities. Amount=2,500 and Date=10/21/2002 |
| Delete Exception | The insurance claim payment could not be deleted. Amount=1,000, Date=10/30/2002, Reason Code=DF |

Table 10, ERROR Usage Scenarios

Usage Example

```
NSLDSLogger.getLogger().logException("This is a custom exception message.", new SFAException(...));
```

The developer would be able to log an error message if the logon was unsuccessful. In the LogonAction, if the WorkUnit's isSuccessful() method returned false, the LogonAction would know the authentication was unsuccessful and log a message using the ERROR level. The java statement would look like NSLDSLogger.getLogger().logException("Authentication unsuccessful with User=" + userName + " and Password=" + password, sfaAuthenticationException) where sfaAuthenticationException is a SFAException created with the AuthenticationException exception code.

4.5.1.4 FATAL

As stated before, the FATAL logging level should be used to log system level exceptions. These are exceptions when a system component has failed unexpectedly or a system resource was unavailable. In most cases, support personnel should be notified by the CSC's Unicenter TNG monitoring tool during these circumstances.

The FATAL logging level will be used in the following environments:

- Development
- Unit Test
- Assembly Test
- System Integration Test
- Performance Test
- User Acceptance Test
- Production
- Training

Usage Guidelines

Similar to ERROR level logging, all logging of FATAL level messages will be constrained to one central location. This allows for a standardization of exception logging and reduced code duplication. Application developers do not need to worry about logging FATAL messages. All FATAL messages should be the result of system exceptions or Java runtime exceptions (such as `java.lang.NullPointerException`).

| Exception Name | Custom Message |
|-----------------------|---|
| SQL Exception | Malformed SQL statement. "SELECT * from " |
| FileNotFoundException | Could not load resource file NSLDSApplicationResource.properties |
| NullPointerException | java.lang.NullPointerException. A general error occurred. |

Table 11, FATAL Usage Scenarios

Usage Example

```
NSLDSLogger.getLogger().logException("This is a custom exception message", new SFAException(...));
```

The FATAL log level would only be used by the developer coding the NSLDSAction class. The NSLDSAction handles all exception cases, which should be logged with the FATAL log level. In handling the exceptions, the developer should include a log statement like the following:

```
NSLDSLogger.getLogger().logException("A System exception has occurred", sfaException),
```

 where `sfaException` is a system `SFAException`

4.5.2 Logging Usage Summary

The table below summarizes the different environments and the logging level that should be captured for it.

| Environment | DEBUG | INFO | ERROR | FATAL |
|-------------------------|--------------|-------------|--------------|--------------|
| Development | x | x | x | x |
| Unit Test | x | x | x | x |
| Assembly Test | x | x | x | x |
| System Integration Test | x | x | x | x |
| Performance Test | | | x | x |
| User Acceptance Test | | | x | x |
| Production | | | x | x |
| Training | x | x | x | x |

Table 12, Logging Level Usage Summary

Although DEBUG and INFO are typically turned off during performance testing, user acceptance testing, and production; those messages could be turned back on if inexplicable system errors occur and need to be traced through the system.

4.6 RCS User Session Framework

The session framework provides a mechanism for the retrieval and manipulation of user session and context data stored in cookies, web server session variables, or in a data store. The framework also creates one common interface for application developers to access the session via any of these methods.

The session framework will provide context management service that stores a user's temporary data during their HTTP session. The session framework will provide for the following services on both the client-side and server-side contexts:

- A common interface to all HTTP variables (request, cookie, session)
- Storing temporary data that should persist across each web page presented to the user

4.6.1 Session Framework Usage Guidelines

The WebSphere Administration Console contains a section for defining the session settings for the web application. For NSLDS II, session id will be stored on a session cookie on the client in the browser's memory. The actual session information will be stored in the *Sessions* table in the Oracle database. Session information is persisted to the database to ensure that session information will be maintained and accessed in a load-balanced environment.

- Users must have cookies enabled on their browsers. If it is not enabled, the user will be directed to a session error page. NSLDS II may not use persistent cookies as they are against federal regulations. Since cookies are required, URL rewriting will not have to be supported.
- In the session manager should have a timeout setting of 30 minutes.
- In the session manager, manual update will be set to false. This means that the database will be written to at the end of execution of every servlet's `service()` method.

4.7 Content Deployment (Interwoven TeamSite)

4.7.1 Content Deployment Approach

Interwoven's TeamSite is a powerful content management tool that can be used for version control, virtualization, templating, workflow, and deployment. For NSLDS II, TeamSite will be used for version control and deployment of the PDF help files to different environments. TeamSite will be used only for deploying the static documents, such as the PDF help files and monthly newsletters.

Its version control utility will be used to monitor the changes and differences in versions of PDF files published to the site. It also provides the capability to 'archive' how the PDF file looked at a specific time.

Interwoven's Open Deploy technology allows users to deploy content housed in TeamSite to development, test, and production environments. Typically, there is a two-week change control request process to update any web sites in the Production environment. Using TeamSite and Open Deploy to deploy PDF documents to production bypasses this process and negates the wait time. NSLDS II monthly newsletters can be created and published in a matter of days instead of having to wait weeks for the process to be completed.

5 Data Access Layer

5.1 RCS Persistence Framework

The ITA persistence framework provides a transparent and flexible mapping of the business objects to relational database tables. It is transparent in that once the business objects and their mappings are defined, application developers do not need to have any knowledge of the underlying relational database tables. It is flexible in that if the underlying relational database model changes, the business object model does not have to change with it – a change in the mapping layer is all that should be required. This framework represents the Data Access Layer that the Architecture Layer will use to access the database.

The table below shows the components that work together to make up the Persistence framework:

| Component | Description |
|--------------------------------------|--|
| ISFAPersistableMapper | Interface that all business mappers must implement. Contains logic for the object-data model mapping. |
| Domain Component | Placeholder class for database's data source name, user ID, and password. |
| Unit of Work Component | This class is not exposed to the developer. Used internally in the Persistence framework to record a roundtrip transaction. |
| Persistable Object Manager Component | Defines the different database operations exposed to the client objects. Used to select, update, insert, and delete records with the database. Provides transactional monitoring through abort and commit methods. |
| Result Set Component | This class is not exposed to the developer. Used internally in the Persistence framework to record results from database query. |
| Business Mapper Component | Individual mappers that implement the ISFAPersistableMapper and contains the actual mappings to the database. |
| Business Object Component | Individual business objects such as User, Student, School with attributes that store the parameters for the queries and also the results of the queries. |

Table 13. Persistence Framework Component Description

The Worker subclasses provide the centralized access to the Persistence framework. The related Worker subclass is called by its Action object. The Worker creates the appropriate Mapper class and populates the attributes of the business object the work is being performed on.

The WorkUnit has methods for the LogonAction to determine if the action was successful and, if so, retrieve the results. The WorkUnit returns the results as an Object, so the Action and Worker classes will have to know how each other expects the results. In the Logon example, both the Action and Worker use the User object to perform the work.

Example usage of the Persistence framework as it applies to NSLDS II:

Business Object Component:

The Business Object components are for the most part simply Java data beans. They contain *getter* and *setter* methods to access the attributes of the component. In the NSLDS system, the business objects may also contain some business logic to operate on their attributes. All example of a Business Object would be the Loan class. The Loan class has many different attributes, each with a get and set method corresponding to the attribute name (i.e. *getInterestRate* and *setInterestRate* for the *interestRate* attribute).

```
//package statement
//import statement(s)

public class User
{
    private String userName;
    private String currentPassword;
    // . . . other attributes included here

    public User()
    {

    }

    public String getUserName()
    {
        return userName;
    }

    public void setUserName(String userName)
    {
        this.userName = userName;
    }

    public String getCurrentPassword()
    {
        return currentPassword;
    }

    public void setCurrentPassword(String currentPassword)
    {
        this.currentPassword = currentPassword;
    }

    // . . . other getter/setter methods included here
```

}

Figure 18, User.java pseudo code

Business Mapper Component:

A Business Mapper component is defined for each Business Object that is mapped to a database table. The Business Mappers implement the logic necessary to select, insert, update, and delete database records about the Business Objects. The Mappers use the Business Object components to generate the SQL queries and update the business objects with the results returned by the queries. The Business Mappers expose methods to retrieve the SQL queries, which are executed by the SFAPersistableObjectManager.

```
// package statement
// import statements

public class LogonMapper implements ISFAPersistableMapper
{
    private User user;
    private String tableName;

    public LogonMapper()
    {
        tableName = "USER";
    }

    public String getDeleteQuery()
    {
        return "DELETE FROM " + tableName + " WHERE userName = ?userName?";
    }

    // returns a collection of parameters to insert into the query (i.e. userName)
    public Vector getDeleteParameters()
    {
        Vector params = new Vector();
        params.addElement(new SFAPParameter("userName", user.getUserName(),
            SFAPParameter.PTSTRING));
        return params;
    }

    public String getInsertQuery()
    {
        return "INSERT INTO " + tableName + " (USERNAME, CURRENTPASSWORD)
            VALUES(?userName?, ?currentPassword?)";
    }

    // returns a collection of parameters to insert into the query (i.e. userName, currentPassword)
    public Vector getInsertParameters()
    {
        Vector params = new Vector();
        params.addElement(new SFAPParameter("userName", user.getUserName(),
            SFAPParameter.PTSTRING));
    }
}
```

```
        params.addElement(new SFAPParameter("currentPassword", user.getCurrentPassword(),
                                           SFAPParameter.PTSTRING));
    }
    return params;
}

public String getKeySelectQuery()
{
    return "CREATE VIEW clear_auth (currentPassword) as SELECT ?userName?, decrypt_char(currentPassword)
          FROM NSLDS_USER_AUTH Set encryption password = 'test123' SELECT userName,
          currentPassword FROM clear_auth";
}

// returns a collection of parameters to insert into the query (i.e. userName)
public Vector getKeySelectParameters()
{
    Vector params = new Vector();
    params.addElement(new SFAPParameter("userName", user.getUserName(),
                                       SFAPParameter.PTSTRING));
    return params;
}

public String getSelectQuery(String selectCondition)
{
    return "SELECT userName, currentPassword FROM " + tableName + " WHERE " + selectCondition;
}

public String getUpdateQuery()
{
    return "UPDATE " + tableName + " SET currentPassword = ?currentPassword? WHERE
          userName = ?userName?";
}

// returns a collection of parameters to insert into the query (i.e. userName, currentPassword)
public Vector getUpdateParameters()
{
    Vector params = new Vector();
    params.addElement(new SFAPParameter("userName", user.getUserName(),
                                       SFAPParameter.PTSTRING));
    params.addElement(new SFAPParameter("currentPassword", user.getCurrentPassword(),
                                       SFAPParameter.PTSTRING));
    return params;
}

// Updates a User object with the results of the query and returns the User object
public Object newFrom(SFAResultSet resultSet) throws SFAException
{
    User user = new User();
    user.setUserName(resultSet.getString("userName"));
    user.setCurrentPassword(resultSet.getString("currentPassword"));
    return user;
}

// Used to retrieve the query parameters for the insert, update, and selectCondition queries
public void populateAttributeValues(Object obj)
```

```

{
    this.user = (User)obj;
}

// Used to retrieve the query parameters for the key select query
public void populateKeyAttributeValues(Object obj)
{
    this.user = (User)obj;
}

public void setTableName(String tableName)
{
    this.tableName = tableName;
}
}

```

Figure 19, LogonMapper.java pseudo code

Worker (To Retrieve Data from the Database):

The Worker abstract class makes use of the Configuration framework (please refer to the [RCS Configuration framework](#) for details), to access the data source name, database user ID, and database password necessary for creating the SFADomain object and connecting to the database. These values are stored in a properties file, read by the Configuration framework so they may be easily maintained and changed without recompiling the code base.

```

// package statement
// import statement(s)

public abstract class Worker
{
    // domain and pom are available to Worker subclasses
    protected SFADomain domain;
    protected SFAPersistableObjectManager pom;

    public Worker()
    {
        // Use Configuration framework to obtain database dataSourceName, userID, and password
        String dataSourceName = FSAConfigurationSI.getProperty("master", null, "dataSource");
        String userID = FSAConfigurationSI.getProperty("master", null, "userID");
        String password = FSAConfigurationSI.getProperty("master", null, "password");
        // Create SFADomain
        domain = new SFADomain(dataSourceName, userID, password);
        // Create PersistableObjectManager with this domain
        pom = new SFAPersistableObjectManager(domain);
    }

    public abstract WorkUnit select(HashMap parameters);
    public abstract WorkUnit insert(HashMap parameters);
    public abstract WorkUnit delete(HashMap parameters);
    public abstract WorkUnit update(HashMap parameters);
}

```

Figure 20, Worker.java pseduo code

LogonWorker (To Retrieve Data from the Database):

For example, a LogonWorker is called by a LogonAction that performs work on a User object. The LogonWorker creates a LogonMapper and populates the mapper's attributes with values from the User object. The LogonMapper is responsible for mapping the User object to the database, through the PersistableObjectManager component. The LogonMapper will return the results to the LogonWorker as an updated User object. The LogonWorker wraps the updated User object in a WorkUnit and returns the WorkUnit to the LogonAction.

```
// package statement
// import statement(s)

public class LogonWorker extends Worker
{
    public LogonWorker()
    {
    }

    public WorkUnit select(HashMap parameters)
    {
        // get the userName and password string from the hashmap, passed in by the LogonAction
        String userName = (String)parameters.get("userName");
        String userPassword = (String)parameters.get("password");

        // create a User object and populate userName attribute
        User user = new User();
        user.setUserName(userName);

        // create LogonMapper object and populate attributes
        LogonMapper mapper = new LogonMapper();
        mapper.populateKeyAttributeValues(user);

        // Use the PersistableObjectManager, pom, instantiated by the Worker super class
        // retrieve the updated user object from the database – will contain currentPassword
        user = (User)pom.getObject(mapper, mapper.getKeySelectQuery(),
            mapper.getKeySelectParameters());

        // create WorkUnit to return results
        WorkUnit workUnit = new WorkUnit();

        // check to see if user entered password matches password from database
        // if yes, then set the workunit success to true, else false
        workUnit.setSuccessful(user.getCurrentPassword().equals(userPassword));

        // set updated User object as the results in the workUnit
        workUnit.setResults(user);
        return workUnit;
    }
}
```

```
public WorkUnit insert(HashMap parameters)
{
    // ...
}

public WorkUnit delete(HashMap parameters)
{
    // ...
}

public WorkUnit update(HashMap parameters)
{
    // ...
}
}
```

Figure 21. LogonWorker.java pseudo code

WorkUnit (To Retrieve Data from the Database):

The WorkUnit exposes two methods dealing with the success of work - `isSuccessful()` and `setSuccessful(boolean)`. The `setSuccessful()` method is used by the Worker subclasses to notify the Action subclasses if this piece of work has completed successfully. This method is called after the work has completed and before the WorkUnit is returned to the Action subclass. When the Action subclass receives the WorkUnit from the Worker subclass, the first thing it does is call `isSuccessful()` to check if the work done by the Worker was successful or not. It uses this information to determine the next course of action and the type of ActionForward to return to the ActionServlet.

```
// package statement
// import statement(s)

public class WorkUnit
{
    private boolean success;
    private Object result;

    public WorkUnit ()
    {
    }

    public boolean isSuccessful()
    {
        return success;
    }

    public void setSuccessful(boolean success)
    {
        this.success = success;
    }
}
```

```
public Object getResults()
{
    return result;
}

public void setResults(Object result)
{
    this.result = result;
}
}
```

Figure 22, WorkUnit.java pseudo code

6 Appendix A - Application Architecture Questionnaire

7 Appendix B - Object-Data Mapping Model

8 Appendix C - ITA Coding Standards

9 Appendix D - ITA Best Practices Guide

10 Appendix E - Coding Standards Review Checklist

10.1 Introduction

10.1.1 Purpose

This document is provided by the ITA for code and peer reviews of Java coding standards. Please document the "Reviewer Information" section and complete all questions and comments where necessary when reviewing code.

10.1.2 References

The ITA deliverables 16.1.3 Java Coding Standards and the 46.1.5 ITA Best Practices documents served as references for this document. (ITA deliverables, provided separately.)

10.1.3 Reviewer Information

Modules/ Packages Reviewed _____

Developer _____

Reviewer _____

Date Reviewed _____

10.2 Java Coding Standards

10.2.1 Source Files

Source files are important because they import the files necessary for the program to run correctly. A source file should contain one public class or interface (not including inner classes). When private classes and interfaces are associated with a public class, they can be placed in the same source file as the public class. The public class should be the first class or interface in a source file.

Each source file should include the following source files at the top of the code, in the following order:

- Beginning comments stating the file information
- A Package statement utilizing the **gov.ed.fsa** name base for all FSA custom packages
- **import** statements that are listed explicitly

| Category | Yes | No | Comments |
|---|-----|----|----------|
| Are the source files accurate and correctly stated? | | | |

10.2.2 Code Layout

Code layout is important for making the code easily readable and easy to maintain. In addition, good code layout facilitates debugging and locating errors. Ensure each source file follows the code layout guidelines listed below:

- Class headers that are declared on one line
- Method headers that are on one line, if possible
- Using curly braces, even for one block statements

| Category | Yes | No | Comments |
|---|-----|----|----------|
| Does the code follow the code layout guidelines?? | | | |

10.2.3 Naming Conventions

The naming conventions are important because they make programs more understandable and easier to read. They also give information about the function of the identifier. Regarding naming conventions, each source file should include:

- Custom-developed packages for FSA that utilize the **gov.ed.fsa** name convention
- The prefix of a unique package name in all-lowercase
- Class names that are nouns and their first letter is capitalized
- Variables that are in mixed case with a lowercase first letter
- Method names that are in proper case, with initial letter in lower-case
- Constants that are all uppercase
- Exception names that follow class-naming conventions, with the additional requirement that the name end in “*Exception*”

| Category | Yes | No | Comments |
|--|-----|----|----------|
| Are the naming conventions correctly utilized? | | | |

10.2.4 Programming Style

This section describes the layout and style of programming. Ensure that good programming styles listed below are followed in each source file:

- A method that is generally not more than 50 “real” lines of code
- A method should generally do just one thing, and the method name should reflect this
- Variables should be declared one line at a time, unless directly related to another variable
- All variables should be declared explicitly
- Numerical constants (literals) should not be explicitly coded
- Use condition variables when necessary
- Use a **for** loop whenever possible, instead of a **while**, **do/while**, or other type
- Name all threads explicitly

| Category | Yes | No | Comments |
|--|-----|----|----------|
| Is the programming style clear and understandable? | | | |

10.2.5 Comments

Comments are important for walking through the code, and especially for debugging. Each source file should contain:

- A single line comment, consisting of two front slashes //
- A multiple line comment, consisting of a front slash followed by an asterisk to begin the comment /*, and an asterisk and front slash to end the comment */
- A JavaDoc comment, consisting of a front slash followed by an asterisk, /** to begin the comment and an asterisk and front slash to end the comment */. A JavaDoc comment should only be used if the user wants those comments extracted to the JavaDoc HTML file.

| Category | Yes | No | Comments |
|---|-----|----|----------|
| Does the source file have the correct comments? | | | |

11 Appendix F - NSLDSLogger Pseudo Code

```
import gov.ed.sfa.ita.logging.Syslog;

/**
 * The NSLDSLogger is a wrapper around the Syslog class provided by the RCS Logging framework.
 * This class provides NSLDS-specific implementations of methods to execute the logging for the 4
 * desired logging levels. The ERROR and FATAL logging levels are both implemented by the
 * logException method.
 */
public class NSLDSLogger
{
    private NSLDSLogger logger;

    private NSLDSLogger()
    {
        super();
    }

    public static synchronized NSLDSLogger getLogger()
    {
        if (logger == null)
        {
            logger = new NSLDSLogger();
        }

        return logger;
    }

    // Log a short message and detailed message to the DEBUG level.
    public void logDebug(Object shortMsg, Object detailedMsg)
    {
        Syslog.log(getCallingClass(), LOG_CHANNEL, shortMsg, detailedMsg, Syslog.DEBUG);
    }

    // Log a short message and detailed message to the INFO level.
    public void logInfo(Object shortMsg, Object detailedMsg)
    {
        Syslog.log(getCallingClass(), LOG_CHANNEL, shortMsg, detailedMsg, Syslog.INFO);
    }

    // Logs a Java Exception. If it is an SFAException, the method checks to see if it is an Application Exception
    // or a System exception to determine the level of logging. Application Exceptions are logged as ERROR
while
    // System exceptions are FATAL.
    public void logException(Object customMsg, Throwable t)
    {
        // retrieve the Class that called the NSLDSLogger
        Class clazz = getCallingClass();

        // if t is of type SFAException then use the Exception Handling framework to locate the desc
        if (t instanceof SFAException)
        {
```

```
// use the SFAExceptionFactory to get the log message associated with this exception code
shortMsg = SFAExceptionFactory.getMessage(((SFAException)t).getCode());

// check to see if the Throwable t is an Application Exception by determining it's exception code
if (t is an ApplicationException)
{
    // if t is an Application Exception, log it at the ERROR level
    Syslog.log(clazz, LOG_CHANNEL, shortMsg, customMsg, Syslog.ERROR);
}
else
{
    // t is a System Exception so log at the FATAL level
    Syslog.log(clazz, LOG_CHANNEL, shortMsg, customMsg, Syslog.FATAL);
}
}
// log Java Runtime exceptions as FATAL
else
{
    Syslog.log(clazz, LOG_CHANNEL, t.toString(), customMsg, Syslog.FATAL);
}
}

// Determines the calling class of the NSLDSLogger by examining the stack trace.
private Class getCallingClass() {
    // Record stack trace
    // Find class name of the class calling the NSLDSLogger
    // Create a new class using this class name
    // Return the new Class object
}
}
```