Design Document for PIMS

1. Overview

After reviewing the Use Case analysis, following are the basic classes and actions that emerge out:-

Classes: (Basic building blocks of PIMS)

Sl no.	Class	Principle Responsibility
1	Investment	Manages computations regarding total investment.
2	Portfolio	Manages computations regarding a Portfolio.
3	Security	Manages computations related to a security.
4	Transaction	Manages computations and stores attributes related to a transaction.
5	GUI	Manages the Graphical User Interface.
6	NetLoader	Manages downloading current prices of shares from the Internet.
7	Current Value System	Manages current value of shares.
8	Alerts	Manages alerts.
9	SecurityManager	Manages user validation.
10	DataRepository	Manages all file operations. It is an interface between the main modules and the database (which in our case is done using file system.)

Note: *Other subsidiary classes may get added to the list in course of implementation for the purpose of load balancing and modularity.*

Actions:

Sl. no.	Action
1	Create/Delete/Rename Portfolio/Security.
2	Create/Delete/Edit Transactions.
3	Calculate *Net Worth* of Investment/Portfolio/ Security.
4	Calculate *Rate of Investment* of a security.
5	Load Current Prices from the Internet.
6	Check/Set/Delete Alerts.
7	Validate User.

Note: *There are other minor actions that does not play major role in modeling.*

2. System Structure

Here we describe the final structure. It should, however, be kept in mind that the obtaining the final structure is an iterative exercise – an initial structure is refined as the design progresses. In particular, the dynamic modeling has an impact on the structure.

2.1. Inheritance Structure

There does not seem to be any inheritance structure because of the lack of commonality between the classes. In some places inheritance seems intuitive, for example in specializing Security into BankSecurity and ShareSecurity and specializing Transaction into Buy and Sell. The figure below shows the inheritance structures.

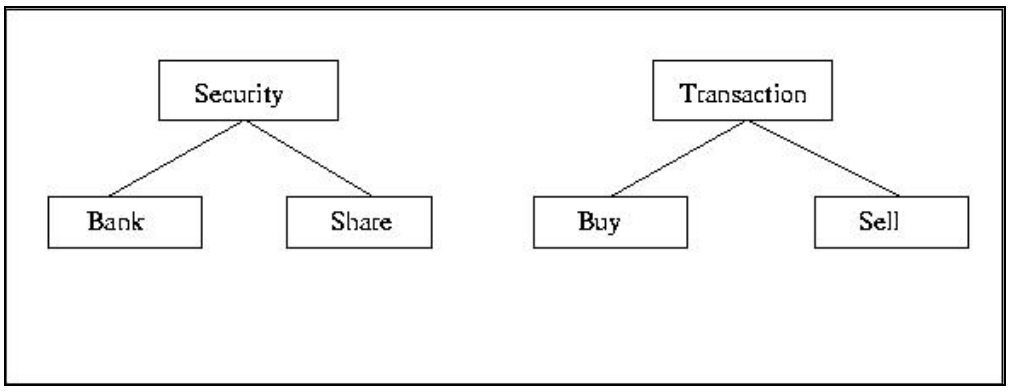

Fig 2.1.1: Possible Inheritance

However these inheritance structures are not necessary. We can model them using an extra attribute securityType and transactionType in the classes Security and Transaction respectively.

2.2. Aggregations

The logical structure of Investment suggests the following aggregation between the classes.

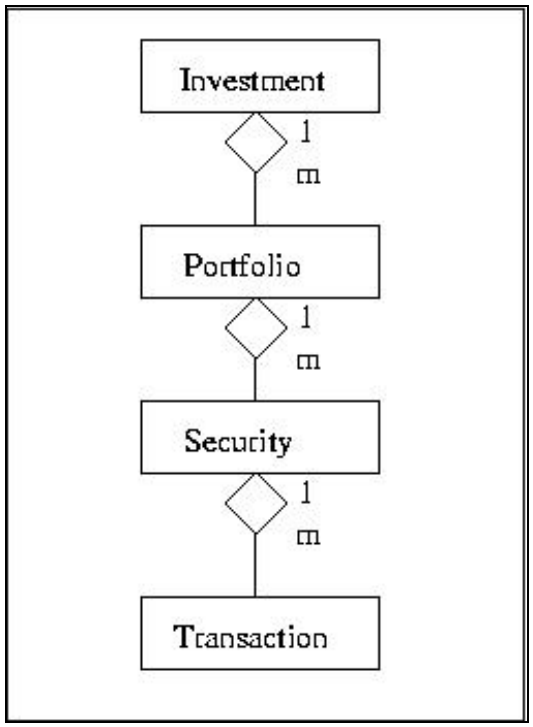

Fig 2.2.1: Aggregation Structure

2.3. Associations

We figure out the association between classes in the process of modeling the principle actions.

Example: Classes (with aggregations and associations) involved in the principle action Create/Delete/Edit Transactions

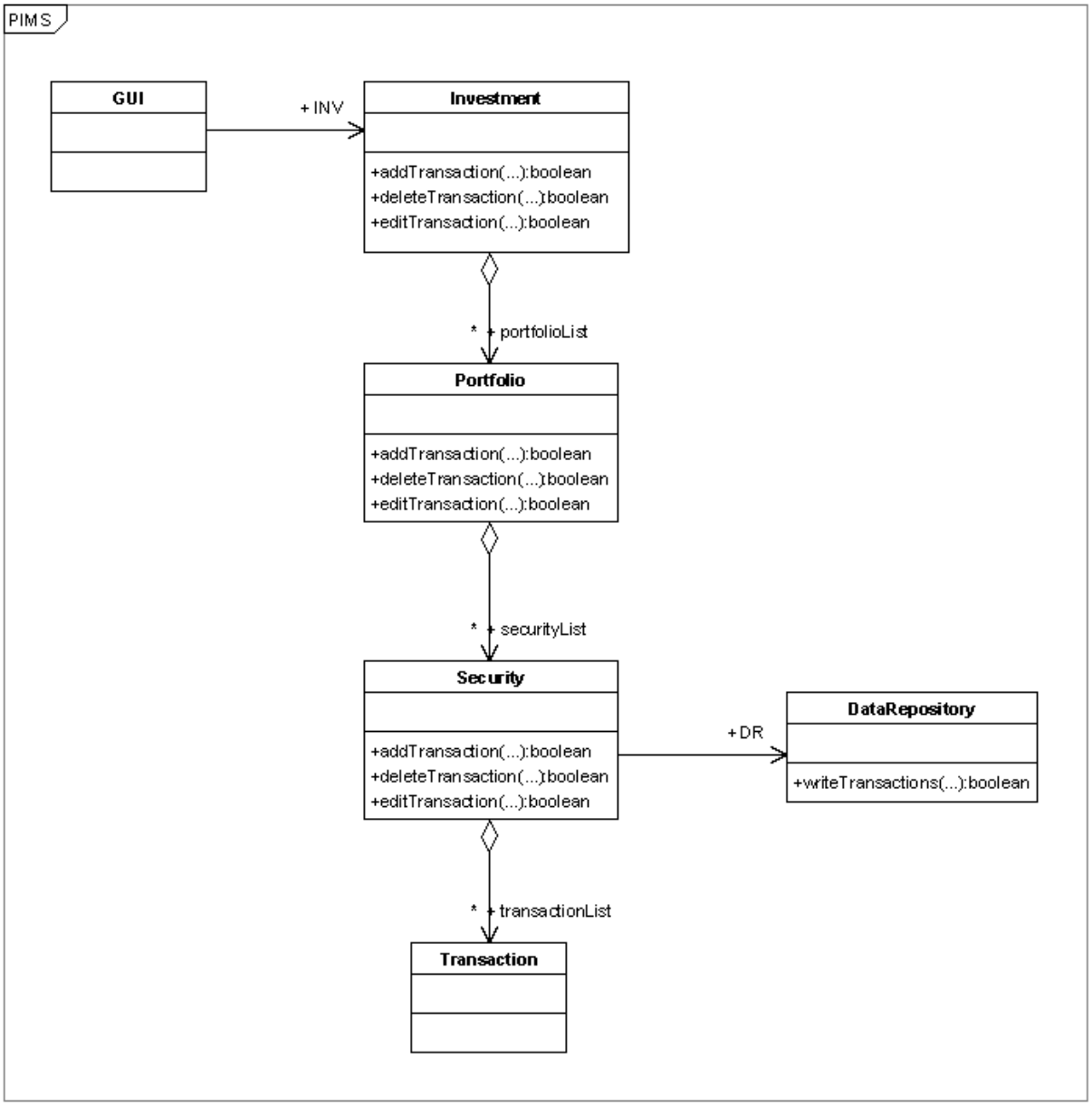

Fig 2.3.1: Class diagram showing associations for action Create/Delete/Edit Transaction

2.4. Complete class diagram

Finally after considering all the major actions the complete association + aggregation structure is arrived at.

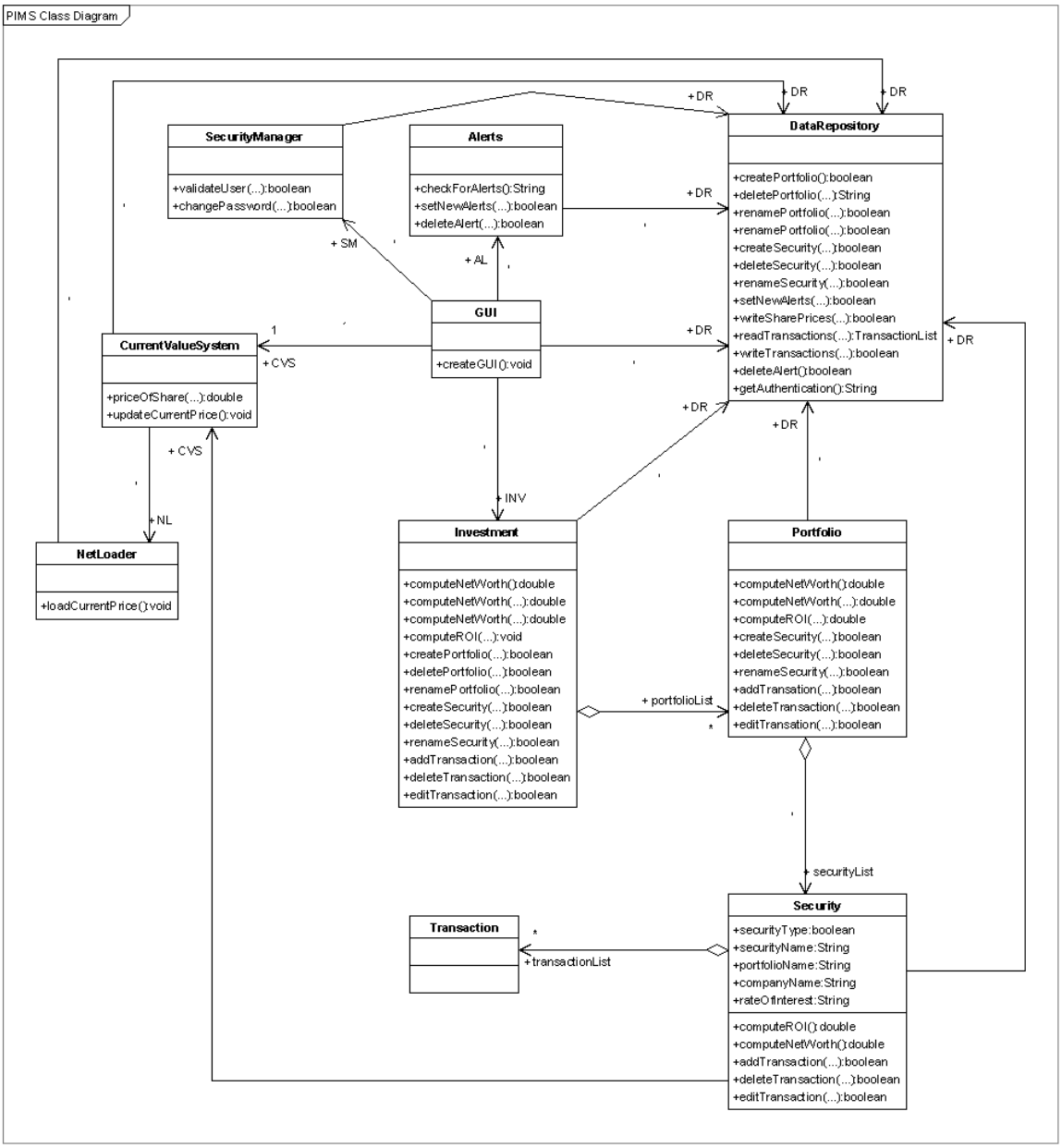

Fig 2.4.1: Class diagram showing all classes and associations in the system

3. System behavior

The dynamic behavior of the system is modeled by figuring out the interactions between the classes involved in each principal action. We are showing the final diagrams here. It should be remembered that these models have an impact in refining and enhancing the class diagrams – we are not discussing these aspects here.

3.1. Principle Action: Create/Delete/Rename Portfolio/Security.

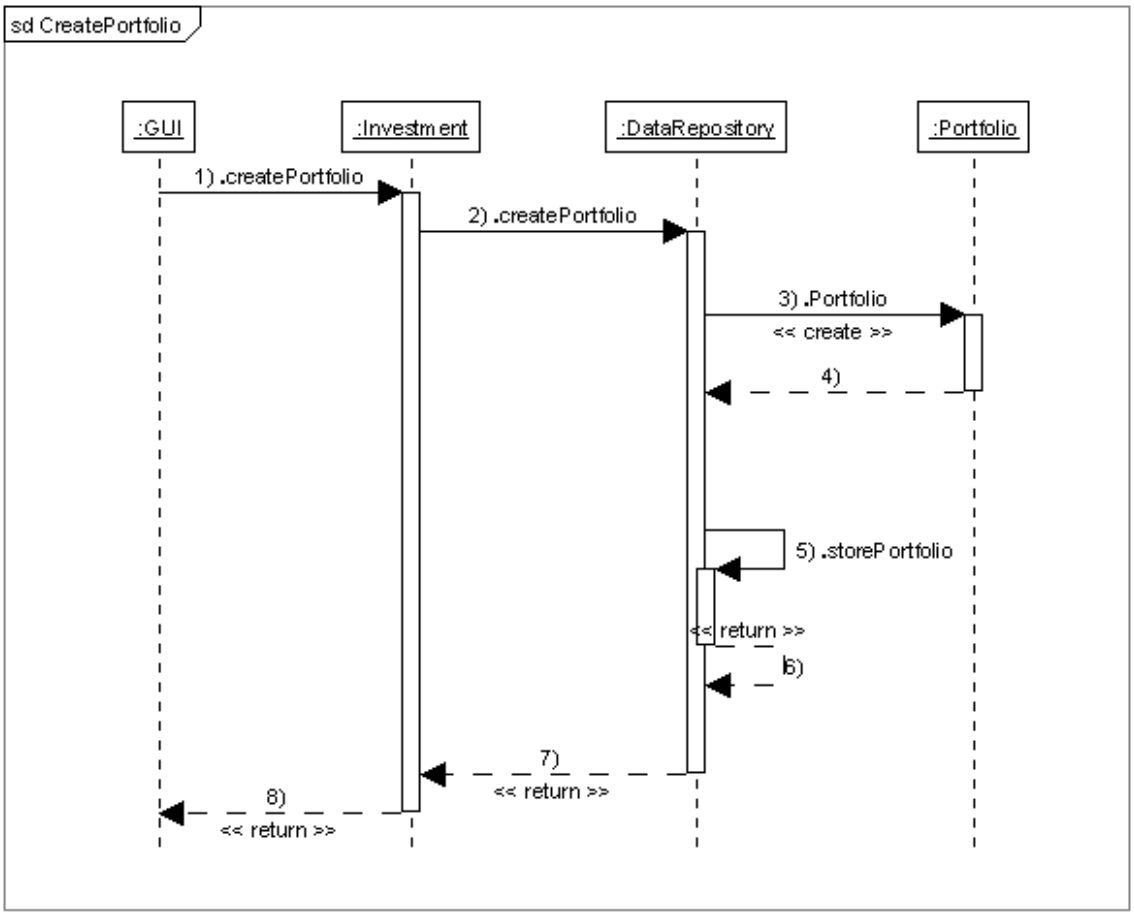

Fig 3.1.1: Sequence diagram for principle action Create Portfolio

3.2. Principle Action: Create/Delete/Edit Transactions.

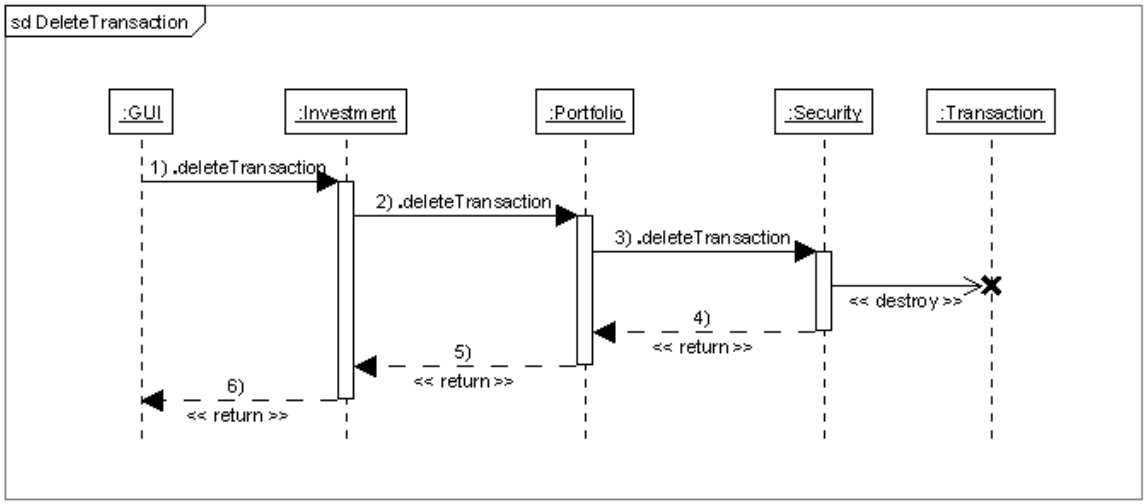

Fig 3.2.1: Sequence diagram for principle action Delete Transaction

3.3. Principle Action: Calculate *Net Worth* of Investment/Portfolio/Security

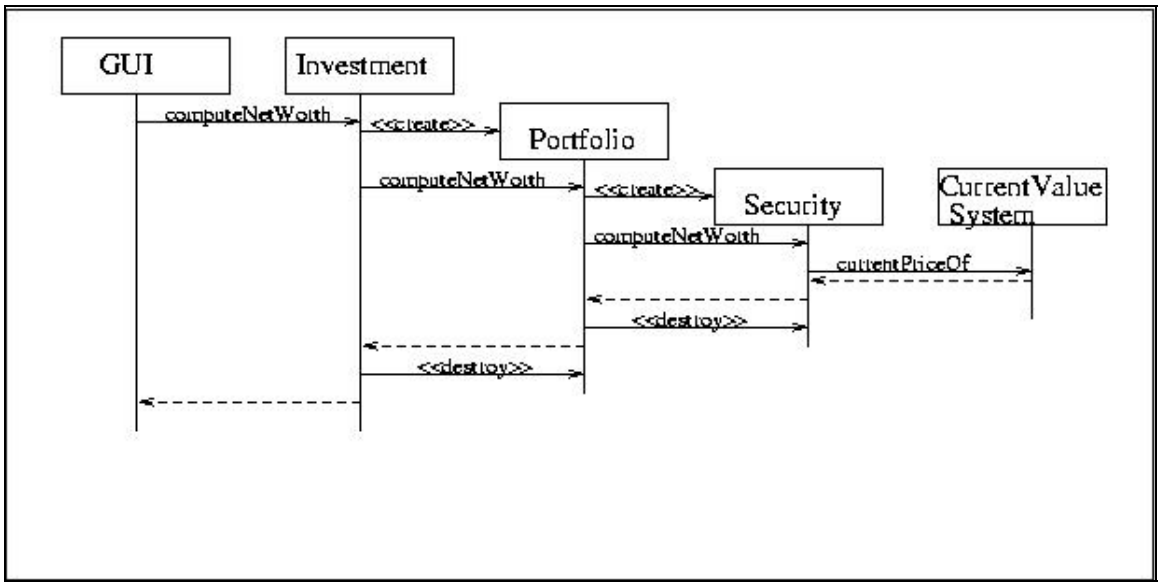

Fig 3.3.1: Sequence diagram for action Compute Net Worth of Investment/Portfolio/Security

3.4. Principle Action: Calculate *Rate of Investment* of a security.

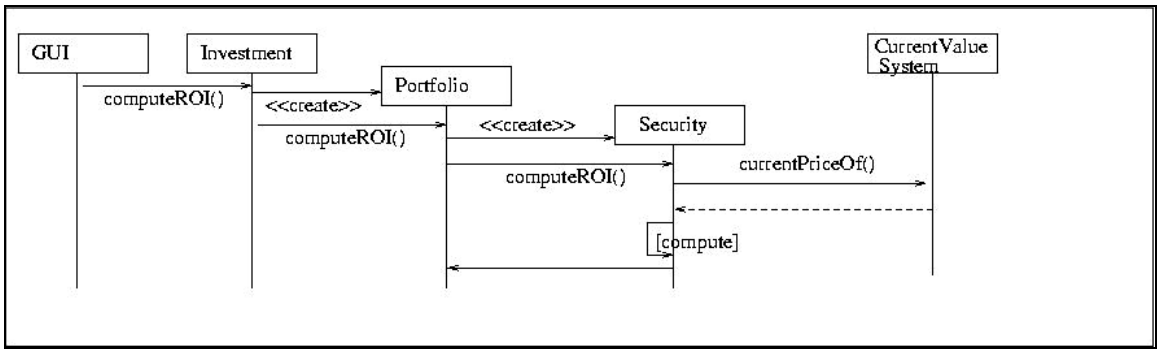

Fig 3.4.1: Sequence diagram for action Compute ROI

3.5. Principle Action: Load Current Prices from the Internet.

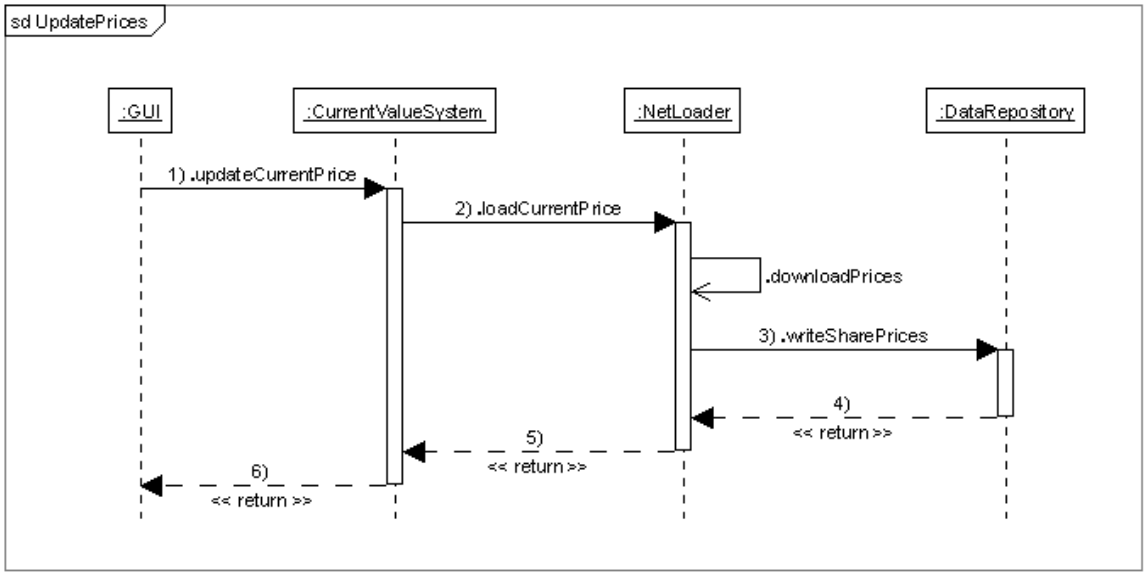

Fig 3.5.1: Sequence diagram for action Load current prices from the Internet

3.6. Principle Action: Check/Set/Delete Alerts.

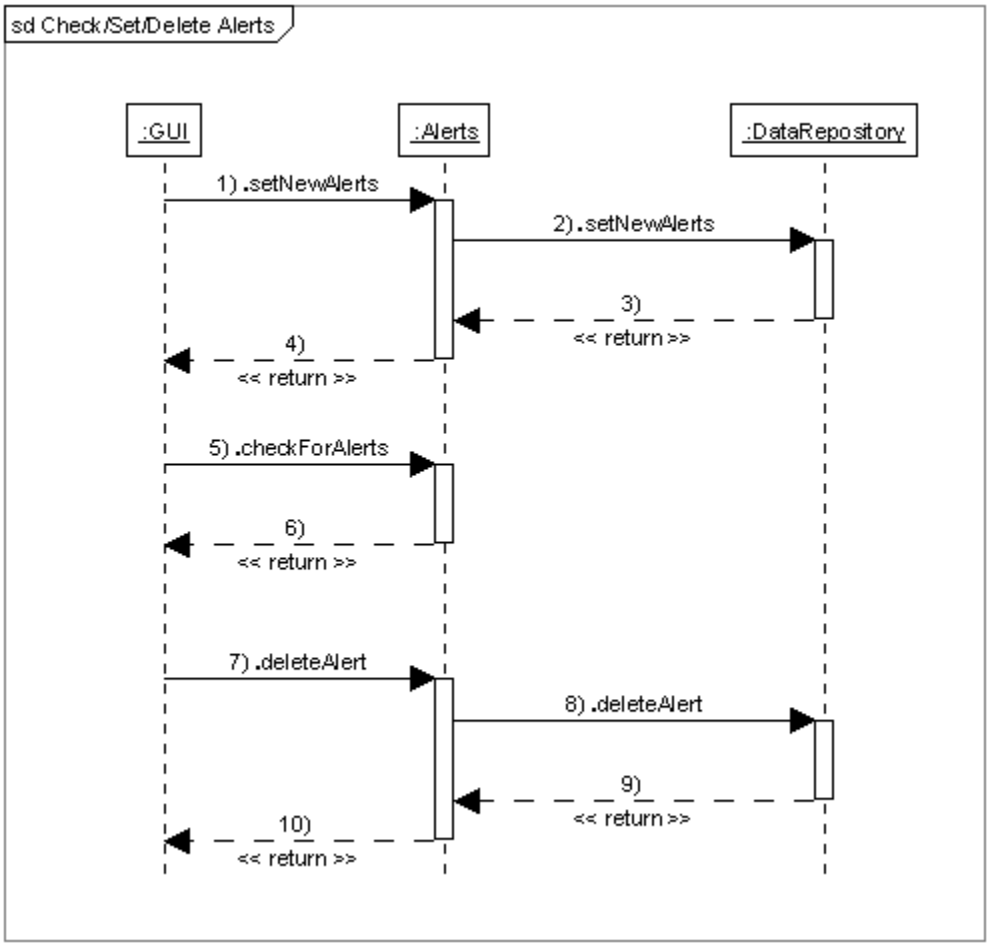

Fig 3.6.1: Sequence diagram for action Set/Check/Delete Alerts

3.7. Principle Action: Validate User.

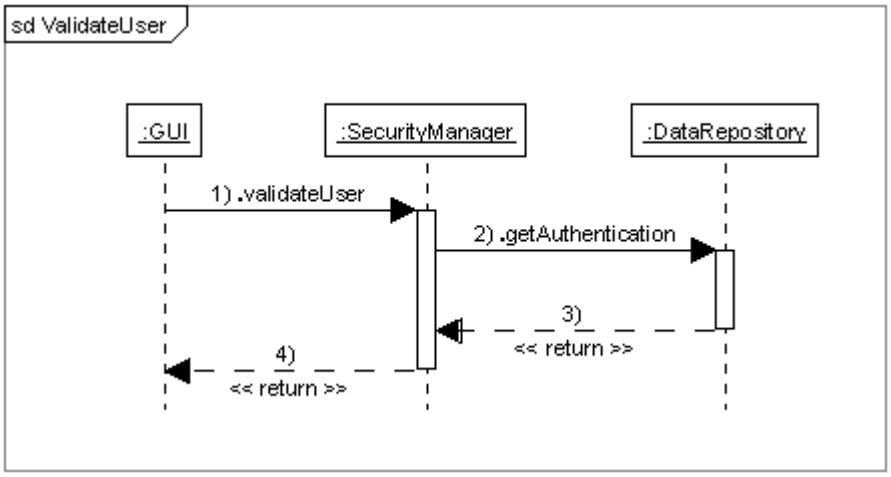

Fig 3.7.1: Sequence diagram for action Validate User

Now we are in a position to start with the design specification as we have all the attributes and methods of all the classes.

4. Detail Design Specification:

It consists of a list of main classes and their attributes and methods with proper comments.

1. class **GUI**{
   //**attributes**//
   CurrentValueSystem CVS; *//Object of the class CurrentValue.*
   Alerts AL; *//Object of the class Alerts.*
   Investment INV; *//Object of the class Investment.*
   DataRepository DR;*//Object of the DataRepository class*
   //**methods**//
   void createGUI(); *//creates the Graphical Interface.*
   }
2. class **Alerts**{
   //**attributes**//
   String alertList[N][2]; *//list containing date and details of all the alerts.*
   //**method**//
   String [] checkForAlerts(); *//check and return all the pending alerts.*
   boolean setNewAlerts(Date date, String details); *//set a new alert as specified by the user.*
   boolean deleteAlert(String Alert);*//Deletes a specified alert*
   }
3. class **NetLoader**{
   //**method**//
   void loadCurrentPrice(); *//Downloads the page from the internet parses it and updates the database.*
   }
4. class **CurrentValueSystem**{
   //**attributes**//
   NetLoader NL;*//Net loader object used to call the loadCurrentPrice() method*
   String sharePrices[N][2]; *//list of current price of shares.*
   //**method**//
   double priceOfShare(String security_name); *//returns the current price of a security.*
   void updateCurrentPrice(); *//This method invokes the NetLoader which updates the share prices,by downloading the current price from the remote database.*
   }
5. class **SecurityManager**{
   //**attributes**//
   String username; *//stores the user name of the investor.*
   String Password; *//stores the password of the user.*
   //**methods**//

boolean validateUser(String user_name, String password); *//checks for the validity of the user.*

boolean changePassword(String oldPassword, String newPassword); *// Changes the password of the authorized user*

}

6. class **Investment**{

//**attributes**//

String PortfolioList[]; *//llist of names of all the portfolios.*

//**methods**//

double computeNetWorth(); *//computes net worth of the investment.*

double computeNetWorth(String portfolio_name);*//computes and returns the net worth of a specified portfolio*

double computeNetWorth(String portfolio_name, String security_name); *//computes and returns the net worth of a specified security in a specified portfolio*

double computeROI(String portfolio_name, String security_name); *//computes the ROI of a specified security in a specified portfolio*

boolean (create/delete/rename)Portfolio(String portfolio_name); *//creates /deletes/renames a portfolio.*

boolean (create/delete/rename)Security(String portfolio_name, String security_name); *// creates/deletes/renames a security.*

boolean (add/delete/edit)Transaction(String portfolio_name, String security_name, Transaction trans);*// adds/deletes/edits a transaction*

}

7. class **Portfolio**{

//**attributes**//

String SecurityList[]; *//list of securities in this particular portfolio.*

String PortfolioName;*//Name of this portfolio*

//**methods**//

double computeNetWorth(); *//returns the net worth of this portfolio.*

double computeNetWorth(String security_name); *//returns the net worth of a specified security*

double computeROI(String security_name); *//computes the ROI of a specified security in this portfolio*

boolean (create/delete/rename)Security(String security_name); *// creates/deletes/renames a security in this portfolio*

boolean (add/delete/edit)Transaction(String portfolio_name, String security_name, Transaction trans);*// adds/deletes/edits a transaction of a specified security*

}

8. class **Security**{

//**attributes**//

Transaction transactionList[]; *//list of transaction objects.*

boolean securityType; *//stores the type of security, bank or share*

String SecurityName;*//Name of this security*
String PortfolioName;*//Name of the portfolio to which it belongs*
String CompanyName;*//Name of the company if share type*
double RateOfInterest;*//Rate of Interest if bank type*
//**methods**//
double computeROI(); //*computes the rate of returns of the security.*
double computeNetWorth(); *//computes the net worth of this security.*
boolean (add/delete/edit)Transaction(Transaction trans);*//Adds/Deletes/ Edits a transaction of this security*

}

9. class **Transaction**{

//**attributes**//
Date date; *//stores the date of the transaction.*
String details; *//stores details of the transaction.*
double TransactionAmount; *//stores the amount of money exchanged.*
boolean Transtype; *//stores the type of transaction buy/sell.*
int numShares; *//stores the number of shares exchanged..*
double CostOfShare;*//stores the cost of share exchanged*

}

10. class **DataRepository**{

//**methods**//
//all these methods do file operations.
boolean createPortfolio(); *//creates a portfolio.*
boolean deletePortfolio(String portfolio_name); *//deletes a portfolio.*
boolean renamePortfolio(String portfolio_name); *//renames a portfolio.*
boolean createSecurity(String portfolio_name, String security_name); *//creates a security.*
boolean deleteSecurity(String portfolio_name, String security_name); *//deletes a security.*
boolean renameSecurity(String portfolio_name, String security_name); *//renames a security.*
boolean setNewAlerts(String alertList[][]); *//set a new alerts as specified by the user.*
void deleteAlert(int index); *// Deletes an alert from the alerts file.*
boolean writeSharePrices(String currentValues[][]); *//sets the new values of the securities.*
TransactionList readTransactions(String portfolio_name, String security_name); //*reads the transactions and returns a list of transaction objects.*
boolean writeTransactions(TransactionList list, String portfolio_name, String security_name); *//writes the transactions into a specified file.*
String getAuthentication(); *//Returns <login>:<password> by reading from the login file*

}

Note: *The Investment class has the list of portfolio* ***names*** *as the attribute and not the list of portfolio objects. This is done to put less pressure on the RAM, keeping all the objects of portfolios, securities and transactions live means that we have the whole database in RAM this might severely effect the efficiency. The portfolio object can be made on the run as and when it is needed. Similar thing has been done for portfolios.*