Electronic Asset Trading Platform

Major Project CAB 302

Milestone 1

Group 2:

Name	**Student ID**
Sam Carran	
Jian Sheng Lee	
Nicholas Meurant	
Sophia Politylo	N10489045

Table of Contents

Requirements:

The Requirements have been broken down into 3 separate categories representing the importance of feature. With the ‘Necessary Requirements’ representing the functions that are need for a functioning trading platform. The client’s requirements are the second most important requirements representing functions that the client specifically requested, that are not critical to a trading platform, but should still be implanted, even if they are not critical to the operation of the program. The quality-of-life requirements represents functions and features that have been requested to be implanted if the time and resources permit it, with these features proving enhanced functionality for the user, while not being critical to the operation of a trading platform.

Necessary Requirements

- The platform must track the quantity of credits owned by a particular organisational unit (OU).
- The platform must track the number of assets owned by a particular OU.
- The platform must allow for different levels of system access based on the type of user.
- The platform must be interacted with through a GUI on the client side.
- All general users will be assigned to a OU.
- All members of a given OU must be able to trade using the assets and credits associated with their OU.
- All members of a given OU must be able to remove offers from any other member of the same OU.
- Assets that have been put into a SELL order must be locked from being placed into any other SELL orders.
- Credits that have been put into a BUY order must be locked from being placed into any other BUY orders.
- Offers that have been removed must unlock the frozen assets or credits associated with that offer.
- Outstanding trades must be periodically reconciled, through a FIFO queue, implemented through the sever.
- A member of the IT Administration team must be able to create new OUs.
- A member of the IT Administration team must be able to create new asset types.
- A member of the IT Administration team must be able to edit the number of credits a given OU has.
- A member of the IT Administration team must be able to edit the number of an asset a given OU has.
- A member of the IT Administration team must be able to add new users to the database and assign passwords from the client.
- A member of the IT Administration team must be able to change an existing user’s password.

- A member of the IT Administration team must be able to grant access up to the current level of access that member has.
- A user must be able to place BUY orders for a certain quantity of a particular asset at a particular price.
- The platform must stop users from placing BUY orders for more credits than that users OU has.
- The platform must not complete transactions where the BUY order price is less than the SELL order price.
- A user must be able to place SELL orders for a certain quantity of a particular asset at a particular price.
- The platform must stop users from placing SELL orders of more of the given asset than that OU has.
- The platform must not complete orders where the SELL order price is higher than the BUY order price.
- The platform must complete transactions where the SELL order price is lower than the BUY order price at the SELL orders’ price.
- The user’s client must interact with a single server (MariaDB, PostgreSQL, or SQLite3).
- The server must store all user’s information (the username, password, account type, and OU).
- The server must store all OU information (OU name, credits, assets, and quantity of assets).
- The server must store all asset types (asset name).
- The server must store all current trades (BUY/SELL, OU, asset name, quantity, price, date).
- The server must store all trade history (BUY/SELL, OU, asset name, quantity, price, date).
- All users must have a unique username.
- All users must have a password.
- The system must hash passwords before sending them from the client to the server.
- The server database must only store hashed passwords.
- The system must require all users to enter their correct username and password into the client to gain access to the platform.

Client's Requirements

- The platform should have a view displaying the current BUY and SELL offers for a given asset.
- The platform should not impose artificial limits on the number of commodities stored.
- The platform should not impose artificial limits on the number of trades that can be listed.
- The platform should not impose artificial limits on the number of users using the system.
- All users should be able to change their own passwords, regardless of system access.
- The platform should display the price history of a given asset, ordered chronologically.
- The client should read from a config file to get the server IP address and port to connect to.
- The server should read from a config file to get its port number.

Quality of Life Requirements

- The platform could have a view displaying the price history on a graph, with the X axis notating the date and the Y axis notating the price.
- The platform could notify users, who have the client running on their computer, when a trade involving their OU has been reconciled.

Detailed Design

Java Classes

Class Diagrams

Client-Side Java Class

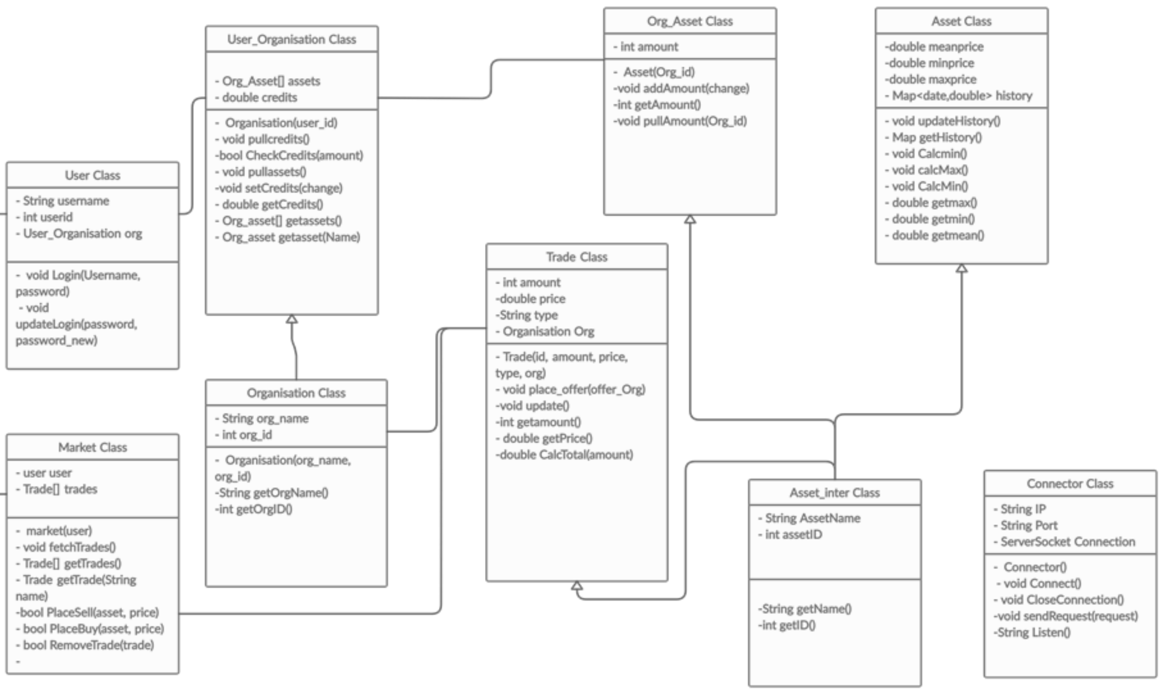


Figure 1 - Class diagram for the client-side class showing inheritance and dependent classe

Server-side Java Class Diagram

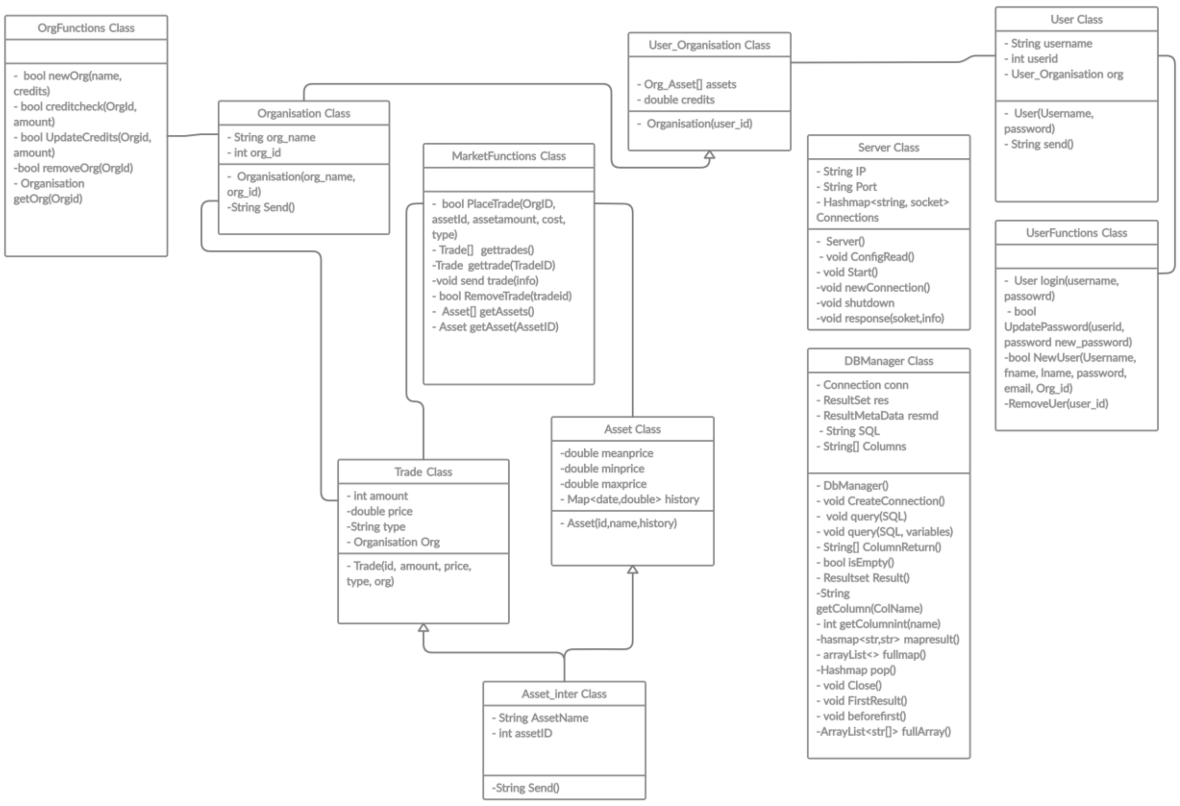
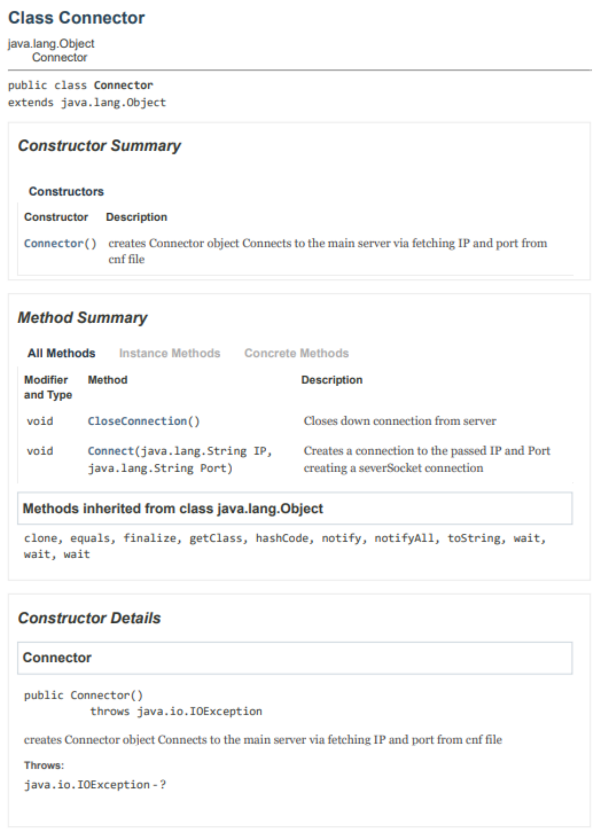


Figure 2 - Class diagram for the server-side class showing inheritance and dependent classes

Java Classes

Client Classes



Class Connector

java.lang.Object
    Connector

```
public class Connector
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor	Description
`Connector()`	creates Connector object Connects to the main server via fetching IP and port from cnf file

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
void	`CloseConnection()`	Closes down connection from server
void	`Connect(java.lang.String IP, java.lang.String Port)`	Creates a connection to the passed IP and Port creating a severSocket connection

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Details

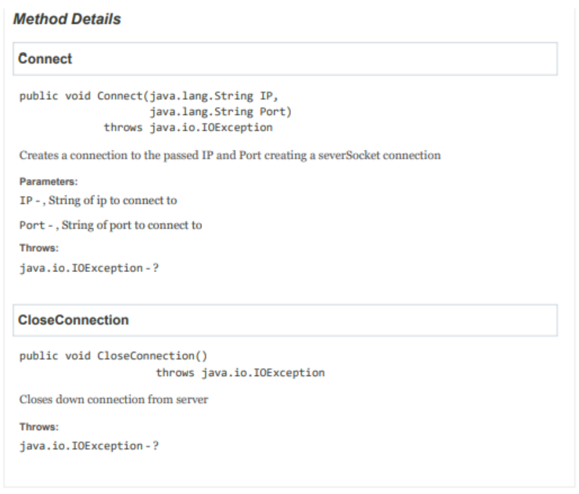
Connector

```
public Connector()
          throws java.io.IOException
```

creates Connector object Connects to the main server via fetching IP and port from cnf file

Throws:

`java.io.IOException` - ?



Method Details

Connect

```
public void Connect(java.lang.String IP,
                    java.lang.String Port)
             throws java.io.IOException
```

Creates a connection to the passed IP and Port creating a severSocket connection

Parameters:

`IP` - , String of ip to connect to

`Port` - , String of port to connect to

Throws:

`java.io.IOException` - ?

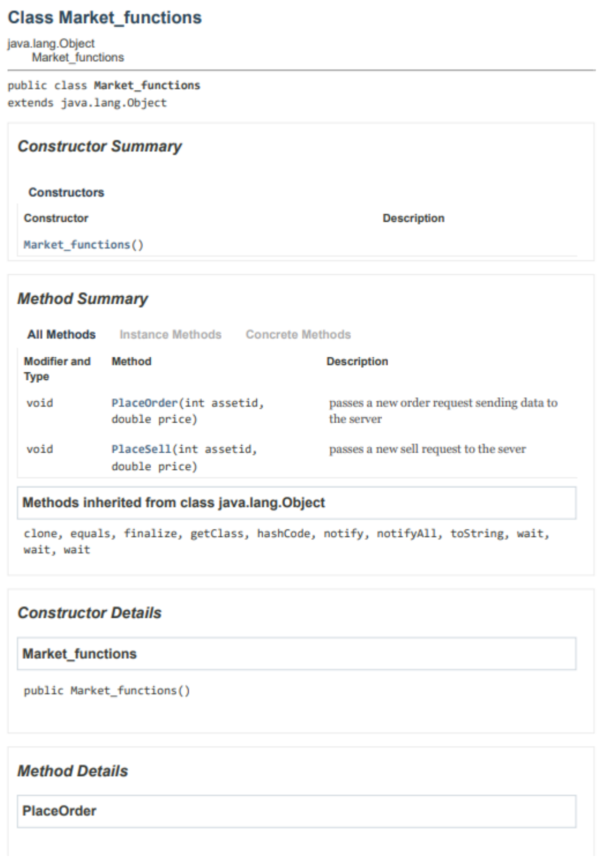
CloseConnection

```
public void CloseConnection()
                     throws java.io.IOException
```

Closes down connection from server

Throws:

`java.io.IOException` - ?



Class Market_functions

java.lang.Object
    Market_functions

```
public class Market_functions
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor	Description
`Market_functions()`	

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
void	`PlaceOrder(int assetid, double price)`	passes a new order request sending data to the server
void	`PlaceSell(int assetid, double price)`	passes a new sell request to the sever

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

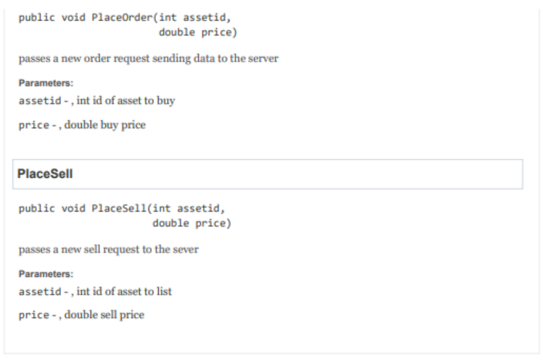
Constructor Details

Market_functions

```
public Market_functions()
```

Method Details

PlaceOrder



```
public void PlaceOrder(int assetid,
                       double price)
```

passes a new order request sending data to the server

Parameters:

`assetid` - , int id of asset to buy

`price` - , double buy price

PlaceSell

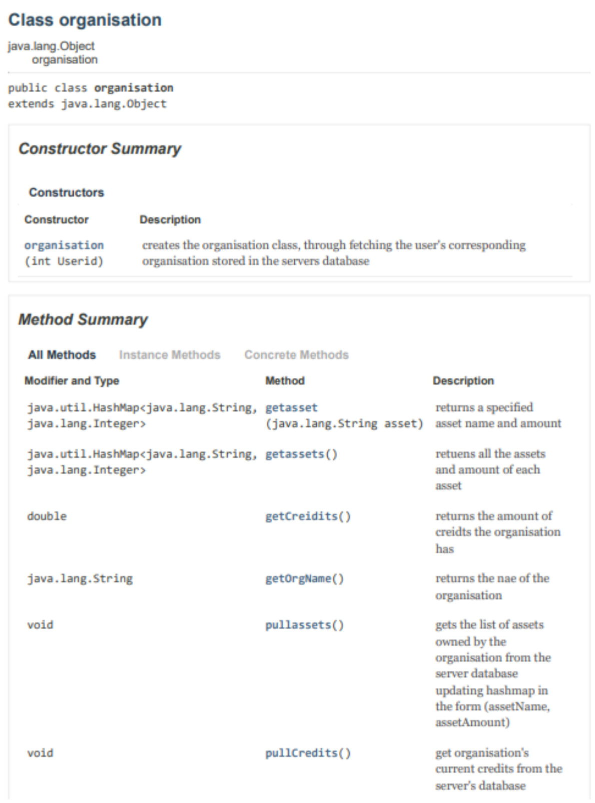
```
public void PlaceSell(int assetid,
                      double price)
```

passes a new sell request to the sever

Parameters:

`assetid` - , int id of asset to list

`price` - , double sell price



Class organisation

java.lang.Object
  organisation

```
public class organisation
extends java.lang.Object
```

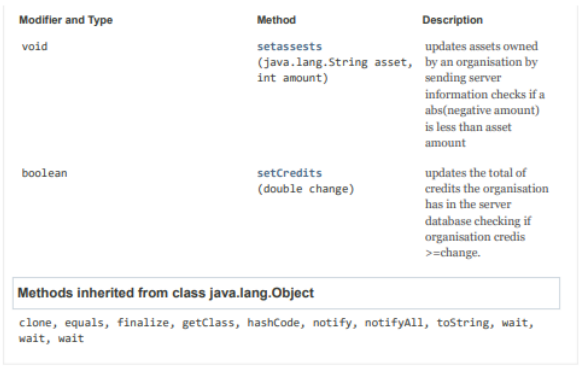
Constructor Summary

Constructors

Constructor	Description
`organisation (int Userid)`	creates the organisation class, through fetching the user's corresponding organisation stored in the servers database

Method Summary

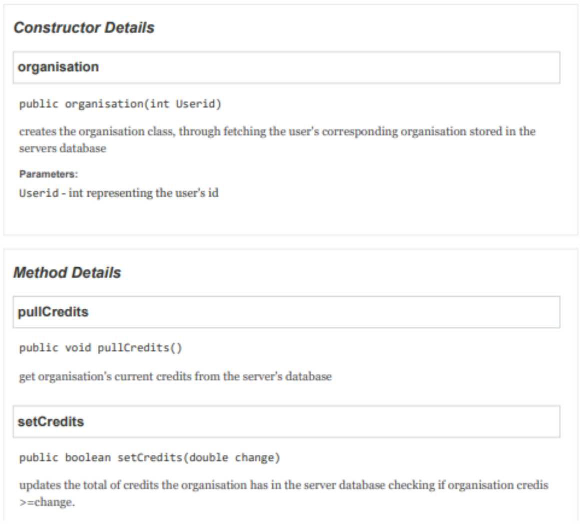
All Methods Instance Methods Concrete Methods



Modifier and Type	Method	Description
`java.util.HashMap<java.lang.String, java.lang.Integer>`	`getasset (java.lang.String asset)`	returns a specified asset name and amount
`java.util.HashMap<java.lang.String, java.lang.Integer>`	`getassets()`	retuens all the assets and amount of each asset
`double`	`getCreidits()`	returns the amount of creidts the organisation has
`java.lang.String`	`getOrgName()`	returns the nae of the organisation
`void`	`pullassets()`	gets the list of assets owned by the organisation from the server database updating hashmap in the form (assetName, assetAmount)
`void`	`pullCredits()`	get organisation's current credits from the server's database
`void`	`setassests (java.lang.String asset, int amount)`	updates assets owned by an organisation by sending server information checks if a abs(negative amount) is less than asset amount
`boolean`	`setCredits (double change)`	updates the total of credits the organisation has in the server database checking if organisation credis >=change.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`



Constructor Details

organisation

```
public organisation(int Userid)
```

creates the organisation class, through fetching the user's corresponding organisation stored in the servers database

Parameters:
`Userid` - int representing the user's id

Method Details

pullCredits

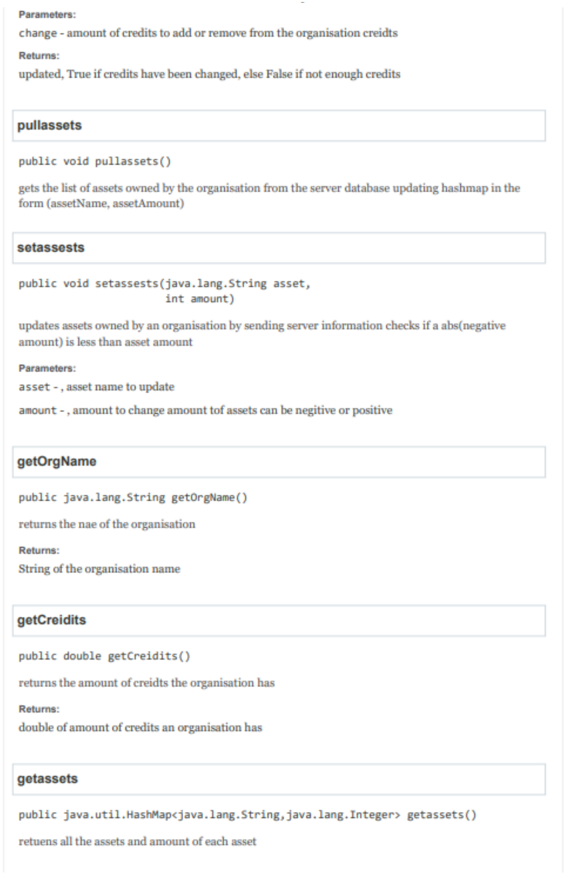
```
public void pullCredits()
```

get organisation's current credits from the server's database

setCredits

```
public boolean setCredits(double change)
```

updates the total of credits the organisation has in the server database checking if organisation credis >=change.



Parameters:
`change` - amount of credits to add or remove from the organisation creidts

Returns:
updated, True if credits have been changed, else False if not enough credits

pullassets

```
public void pullassets()
```

gets the list of assets owned by the organisation from the server database updating hashmap in the form (assetName, assetAmount)

setassests

```
public void setassests(java.lang.String asset,
                       int amount)
```

updates assets owned by an organisation by sending server information checks if a abs(negative amount) is less than asset amount

Parameters:
`asset` - , asset name to update
`amount` - , amount to change amount tof assets can be negitive or positive

getOrgName

```
public java.lang.String getOrgName()
```

returns the nae of the organisation

Returns:
String of the organisation name

getCreidits

```
public double getCreidits()
```

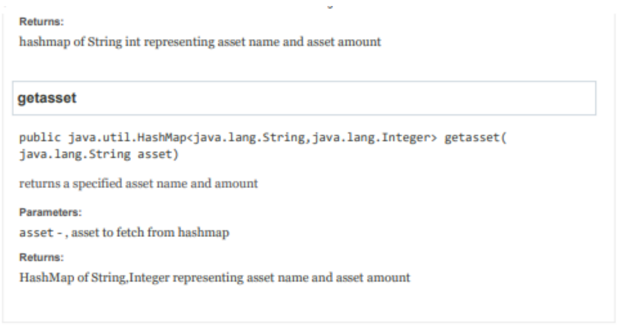
returns the amount of creidts the organisation has

Returns:
double of amount of credits an organisation has

getassets

```
public java.util.HashMap<java.lang.String,java.lang.Integer> getassets()
```

retuens all the assets and amount of each asset



Returns:
hashmap of String int representing asset name and asset amount

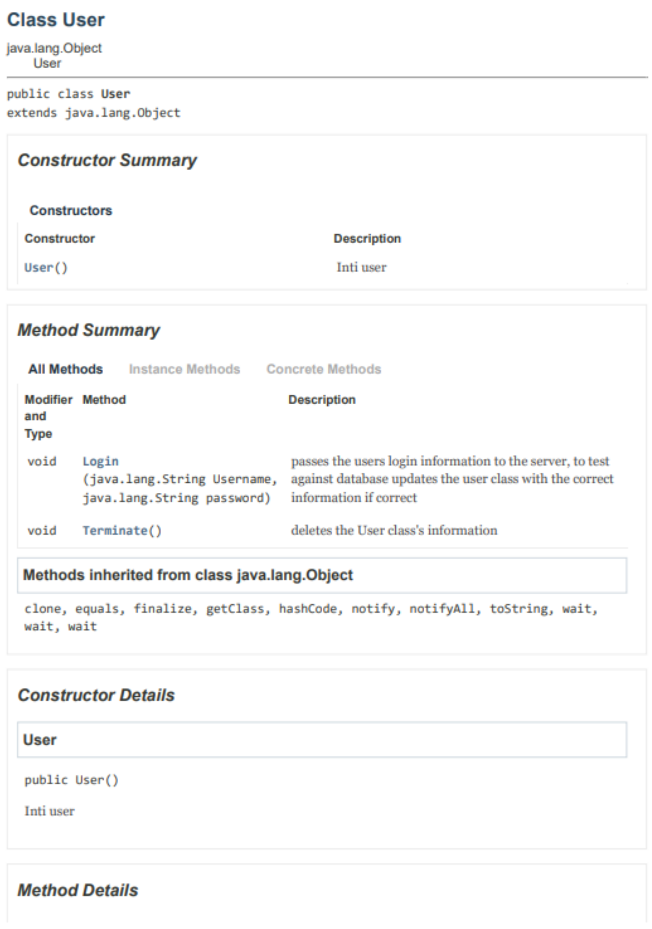
getasset

```
public java.util.HashMap<java.lang.String,java.lang.Integer> getasset(
java.lang.String asset)
```

returns a specified asset name and amount

Parameters:
`asset` - , asset to fetch from hashmap

Returns:
HashMap of String,Integer representing asset name and asset amount



Class User

java.lang.Object
    User

```
public class User
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor	Description
`User()`	Inti user

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method	Description
`void`	`Login(java.lang.String Username, java.lang.String password)`	passes the users login information to the server, to test against database updates the user class with the correct information if correct
`void`	`Terminate()`	deletes the User class's information

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

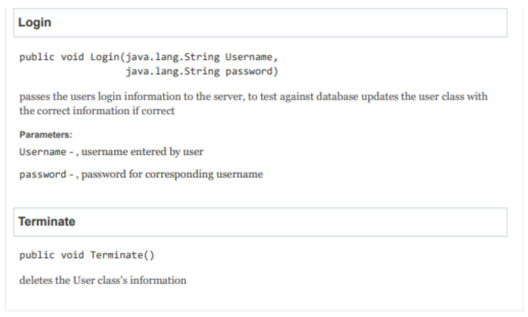
Constructor Details

User

```
public User()
```

Inti user

Method Details



Login

```
public void Login(java.lang.String Username,
                  java.lang.String password)
```

passes the users login information to the server, to test against database updates the user class with the correct information if correct

Parameters:

`Username` - , username entered by user

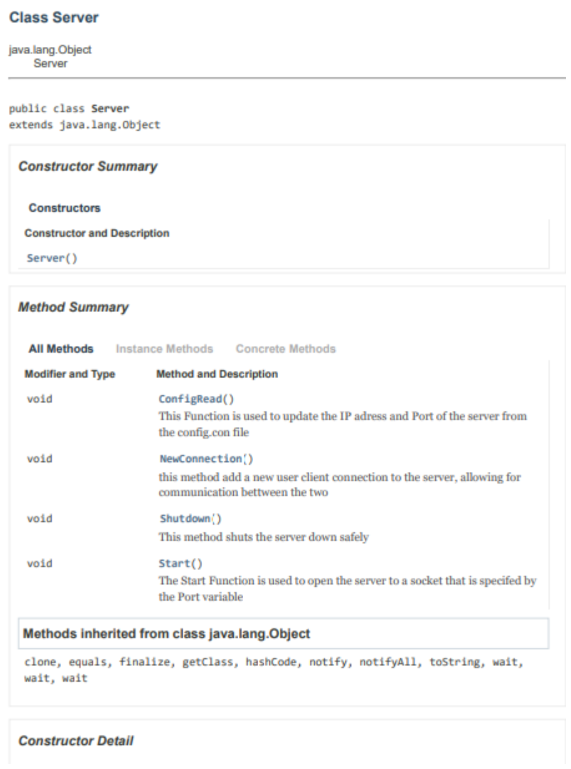
`password` - , password for corresponding username

Terminate

```
public void Terminate()
```

deletes the User class's information

Server Classes



Class Server

java.lang.Object
    Server

```
public class Server
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor and Description
`Server()`

Method Summary

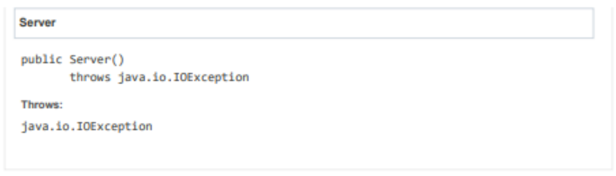
All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
`void`	`ConfigRead()` This Function is used to update the IP adress and Port of the server from the config.con file
`void`	`NewConnection()` this method add a new user client connection to the server, allowing for communication betwween the two
`void`	`Shutdown()` This method shuts the server down safely
`void`	`Start()` The Start Function is used to open the server to a socket that is specifed by the Port variable

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait,
wait, wait
```

Constructor Detail

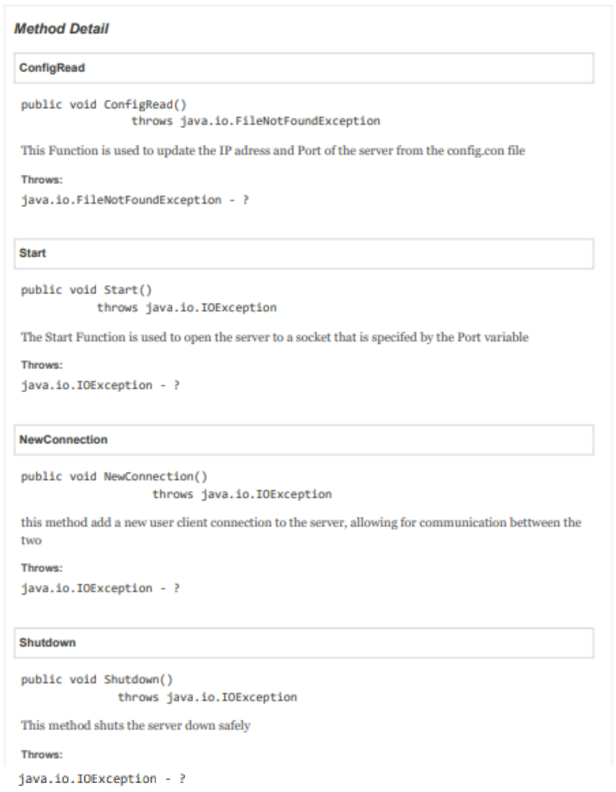


Server

```
public Server()
       throws java.io.IOException
```

Throws:

`java.io.IOException`



Method Detail

ConfigRead

```
public void ConfigRead()
                throws java.io.FileNotFoundException
```

This Function is used to update the IP adress and Port of the server from the config.con file

Throws:

`java.io.FileNotFoundException - ?`

Start

```
public void Start()
           throws java.io.IOException
```

The Start Function is used to open the server to a socket that is specifed by the Port variable

Throws:

`java.io.IOException - ?`

NewConnection

```
public void NewConnection()
                   throws java.io.IOException
```

this method add a new user client connection to the server, allowing for communication betwween the two

Throws:

`java.io.IOException - ?`

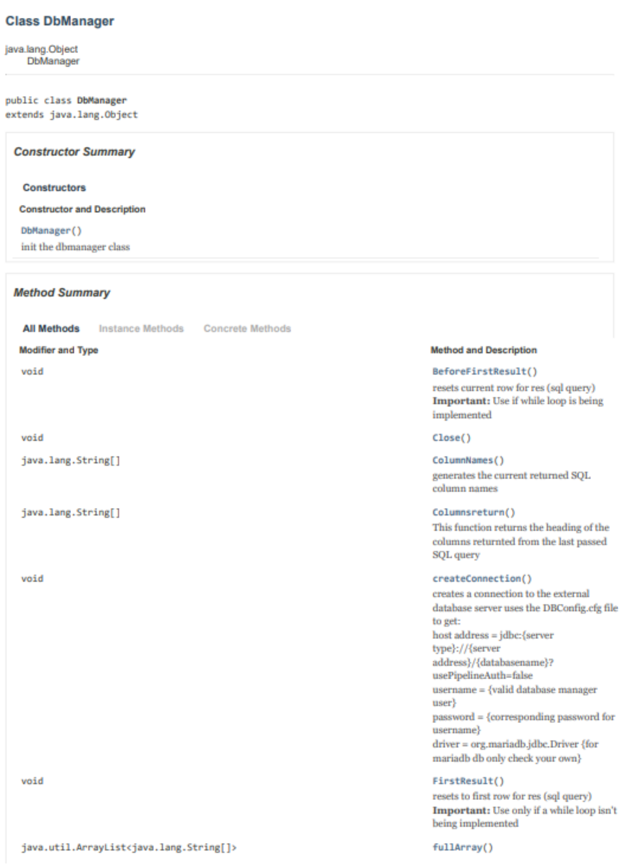
Shutdown

```
public void Shutdown()
              throws java.io.IOException
```

This method shuts the server down safely

Throws:

`java.io.IOException - ?`



Class DbManager

java.lang.Object
    DbManager

```
public class DbManager
extends java.lang.Object
```

Constructor Summary

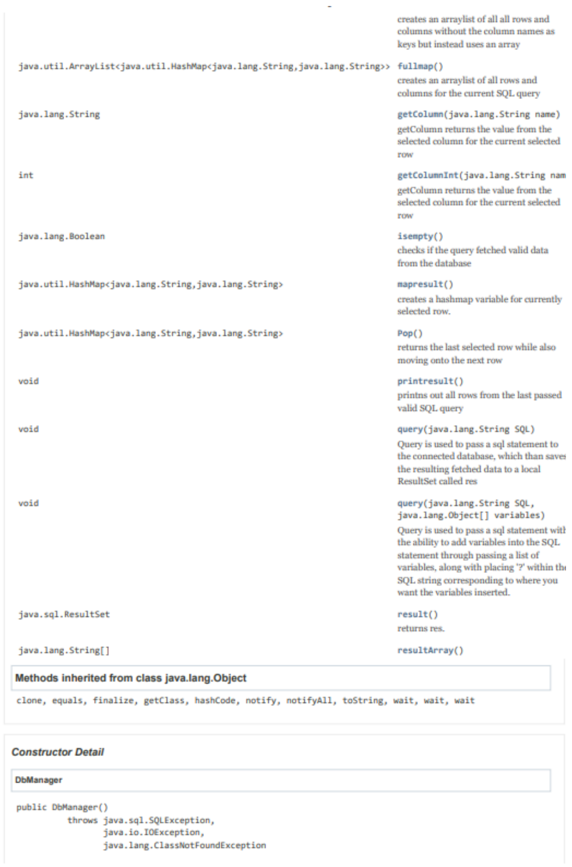
Constructors

Constructor and Description
`DbManager()` init the dbmanager class

Method Summary

All Methods Instance Methods Concrete Methods

Modifier and Type	Method and Description
`void`	`BeforeFirstResult()` resets current row for res (sql query) **Important:** Use if while loop is being implemented
`void`	`Close()`
`java.lang.String[]`	`ColumnNames()` generates the current returned SQL column names
`java.lang.String[]`	`Columnsreturn()` This function returns the heading of the columns returnted from the last passed SQL query
`void`	`createConnection()` creates a connection to the external database server uses the DBConfig.cfg file to get: host address = jdbc:{server type}://{server address}/{databasename}?usePipelineAuth=false username = {valid database manager user} password = {corresponding password for username} driver = org.mariadb.jdbc.Driver {for mariadb db only check your own}
`void`	`FirstResult()` resets to first row for res (sql query) **Important:** Use only if a while loop isn't being implemented
`java.util.ArrayList<java.lang.String[]>`	`fullArray()`



	creates an arraylist of all all rows and columns without the column names as keys but instead uses an array
`java.util.ArrayList<java.util.HashMap<java.lang.String,java.lang.String>>`	`fullmap()` creates an arraylist of all rows and columns for the current SQL query
`java.lang.String`	`getColumn(java.lang.String name)` getColumn returns the value from the selected column for the current selected row
`int`	`getColumnInt(java.lang.String nam` getColumn returns the value from the selected column for the current selected row
`java.lang.Boolean`	`isempty()` checks if the query fetched valid data from the database
`java.util.HashMap<java.lang.String,java.lang.String>`	`mapresult()` creates a hashmap variable for currently selected row.
`java.util.HashMap<java.lang.String,java.lang.String>`	`Pop()` returns the last selected row while also moving onto the next row
`void`	`printresult()` printns out all rows from the last passed valid SQL query
`void`	`query(java.lang.String SQL)` Query is used to pass a sql statement to the connected database, which than saves the resulting fetched data to a local ResultSet called res
`void`	`query(java.lang.String SQL, java.lang.Object[] variables)` Query is used to pass a sql statement wit the ability to add variables into the SQL statement through passing a list of variables, along with placing '?' within th SQL string corresponding to where you want the variables inserted.
`java.sql.ResultSet`	`result()` returns res.
`java.lang.String[]`	`resultArray()`

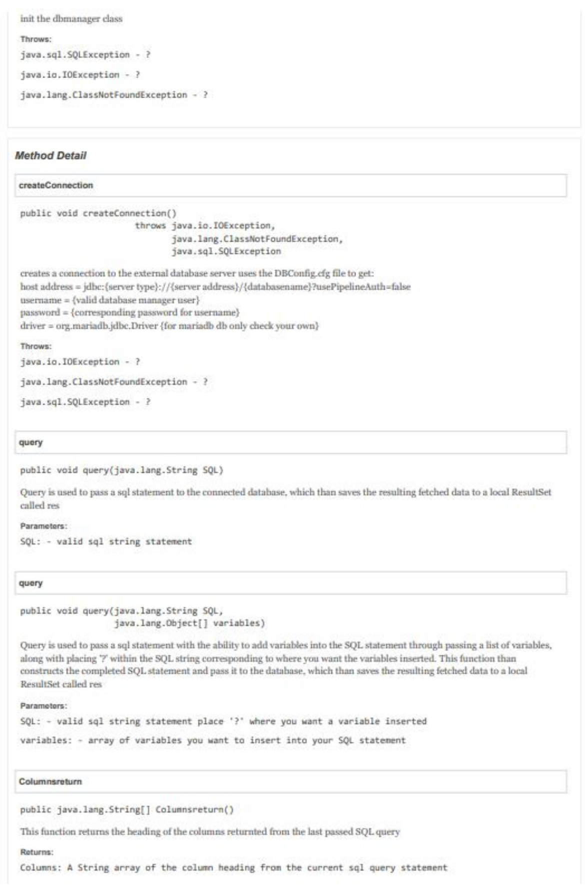
Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

DbManager

```
public DbManager()
          throws java.sql.SQLException,
                 java.io.IOException,
                 java.lang.ClassNotFoundException
```



init the dbmanager class

Throws:

`java.sql.SQLException - ?`

`java.io.IOException - ?`

`java.lang.ClassNotFoundException - ?`

Method Detail

createConnection

```
public void createConnection()
                      throws java.io.IOException,
                             java.lang.ClassNotFoundException,
                             java.sql.SQLException
```

creates a connection to the external database server uses the DBConfig.cfg file to get:
host address = jdbc:{server type}://{server address}/{databasename}?usePipelineAuth=false
username = {valid database manager user}
password = {corresponding password for username}
driver = org.mariadb.jdbc.Driver {for mariadb db only check your own}

Throws:

`java.io.IOException - ?`

`java.lang.ClassNotFoundException - ?`

`java.sql.SQLException - ?`

query

```
public void query(java.lang.String SQL)
```

Query is used to pass a sql statement to the connected database, which than saves the resulting fetched data to a local ResultSet called res

Parameters:

`SQL: - valid sql string statement`

query

```
public void query(java.lang.String SQL,
                  java.lang.Object[] variables)
```

Query is used to pass a sql statement with the ability to add variables into the SQL statement through passing a list of variables, along with placing '?' within the SQL string corresponding to where you want the variables inserted. This function than constructs the completed SQL statement and pass it to the database, which than saves the resulting fetched data to a local ResultSet called res

Parameters:

`SQL: - valid sql string statement place '?' where you want a variable inserted`

`variables: - array of variables you want to insert into your SQL statement`

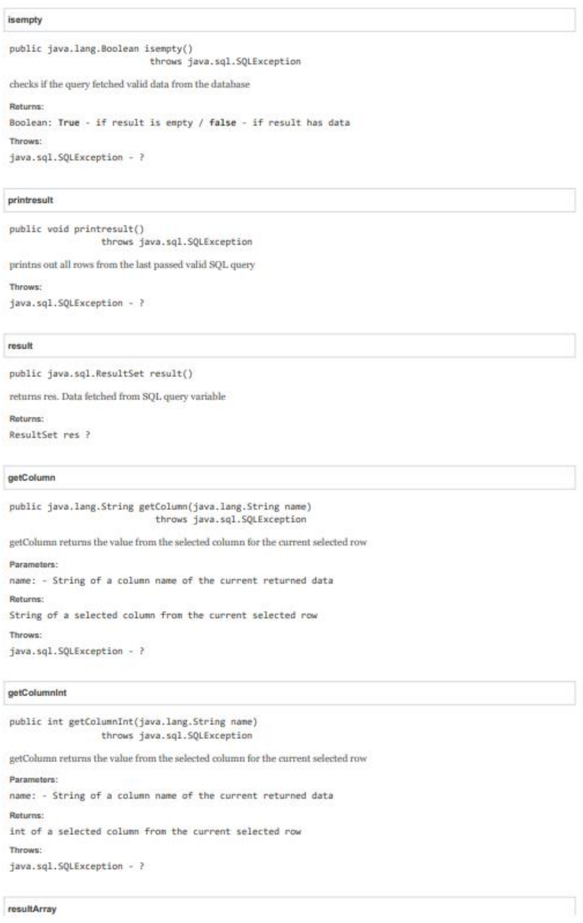
Columnsreturn

```
public java.lang.String[] Columnsreturn()
```

This function returns the heading of the columns returnted from the last passed SQL query

Returns:

`Columns: A String array of the column heading from the current sql query statement`



isempty

```
public java.lang.Boolean isempty()
                          throws java.sql.SQLException
```

checks if the query fetched valid data from the database

Returns:

`Boolean: True - if result is empty / false - if result has data`

Throws:

`java.sql.SQLException - ?`

printresult

```
public void printresult()
                 throws java.sql.SQLException
```

printns out all rows from the last passed valid SQL query

Throws:

`java.sql.SQLException - ?`

result

```
public java.sql.ResultSet result()
```

returns res. Data fetched from SQL query variable

Returns:

`ResultSet res ?`

getColumn

```
public java.lang.String getColumn(java.lang.String name)
                           throws java.sql.SQLException
```

getColumn returns the value from the selected column for the current selected row

Parameters:

`name: - String of a column name of the current returned data`

Returns:

`String of a selected column from the current selected row`

Throws:

`java.sql.SQLException - ?`

getColumnInt

```
public int getColumnInt(java.lang.String name)
                 throws java.sql.SQLException
```

getColumn returns the value from the selected column for the current selected row

Parameters:

`name: - String of a column name of the current returned data`

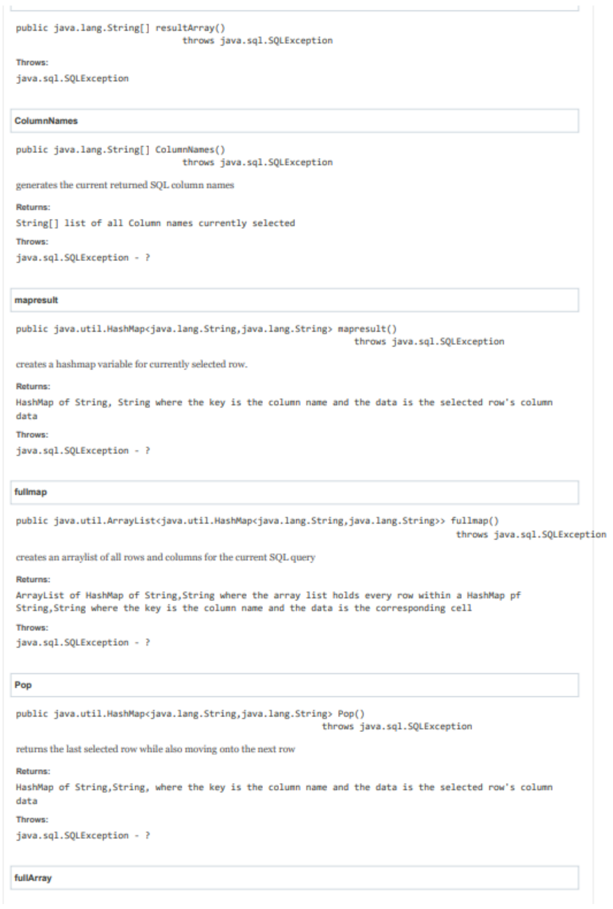
Returns:

`int of a selected column from the current selected row`

Throws:

`java.sql.SQLException - ?`

resultArray



```
public java.lang.String[] resultArray()
                          throws java.sql.SQLException
```

Throws:

`java.sql.SQLException`

ColumnNames

```
public java.lang.String[] ColumnNames()
                          throws java.sql.SQLException
```

generates the current returned SQL column names

Returns:

`String[] list of all Column names currently selected`

Throws:

`java.sql.SQLException - ?`

mapresult

```
public java.util.HashMap<java.lang.String,java.lang.String> mapresult()
                                                            throws java.sql.SQLException
```

creates a hashmap variable for currently selected row.

Returns:

`HashMap of String, String where the key is the column name and the data is the selected row's column data`

Throws:

`java.sql.SQLException - ?`

fullmap

```
public java.util.ArrayList<java.util.HashMap<java.lang.String,java.lang.String>> fullmap()
                                                                                  throws java.sql.SQLException
```

creates an arraylist of all rows and columns for the current SQL query

Returns:

`ArrayList of HashMap of String,String where the array list holds every row within a HashMap pf String,String where the key is the column name and the data is the corresponding cell`

Throws:

`java.sql.SQLException - ?`

Pop

```
public java.util.HashMap<java.lang.String,java.lang.String> Pop()
                                                      throws java.sql.SQLException
```

returns the last selected row while also moving onto the next row

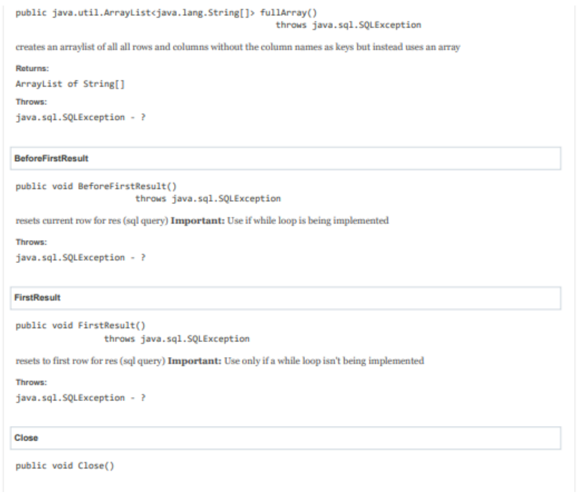
Returns:

`HashMap of String,String, where the key is the column name and the data is the selected row's column data`

Throws:

`java.sql.SQLException - ?`

fullArray



```
public java.util.ArrayList<java.lang.String[]> fullArray()
                                               throws java.sql.SQLException
```

creates an arraylist of all all rows and columns without the column names as keys but instead uses an array

Returns:

`ArrayList of String[]`

Throws:

`java.sql.SQLException - ?`

BeforeFirstResult

```
public void BeforeFirstResult()
                       throws java.sql.SQLException
```

resets current row for res (sql query) **Important:** Use if while loop is being implemented

Throws:

`java.sql.SQLException - ?`

FirstResult

```
public void FirstResult()
                 throws java.sql.SQLException
```

resets to first row for res (sql query) **Important:** Use only if a while loop isn't being implemented

Throws:

`java.sql.SQLException - ?`

Close

```
public void Close()
```

GUI Designs

The GUI design mock-ups have been split into 3 categories, to help represent the function and flow of the various screens.

GUI Flow between user's home screens

The user's home screens shows the GUI that will greet the user when they boot up the program. With the user being taken to the login screen on boot up. With a correct password and user name input sending them to either the generic user or admin home page depending on their role. Tow other screens are also show, these two screens have been include here as they both are pop ups that help the user change personal information or view organisation information that do not directly impact the market place.

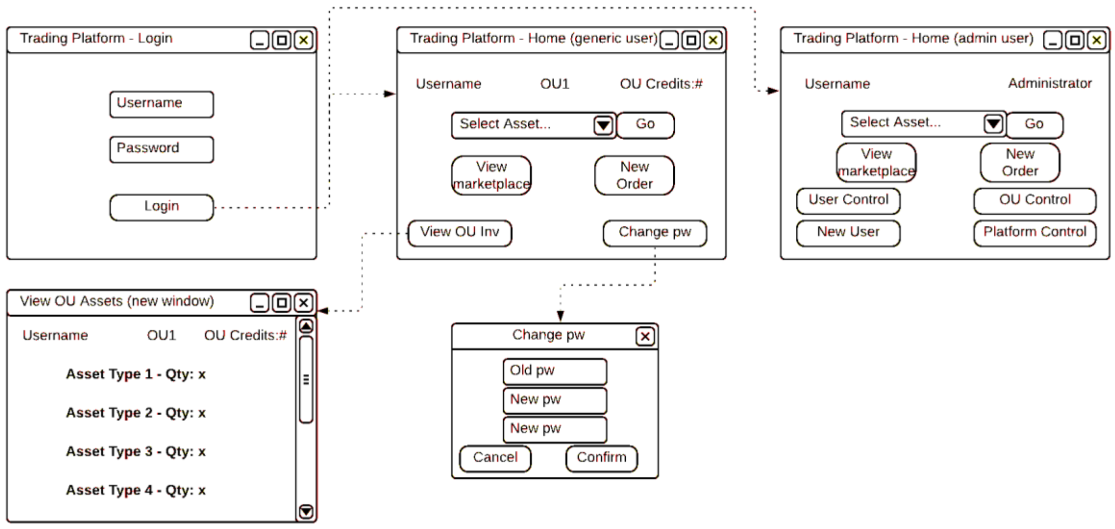
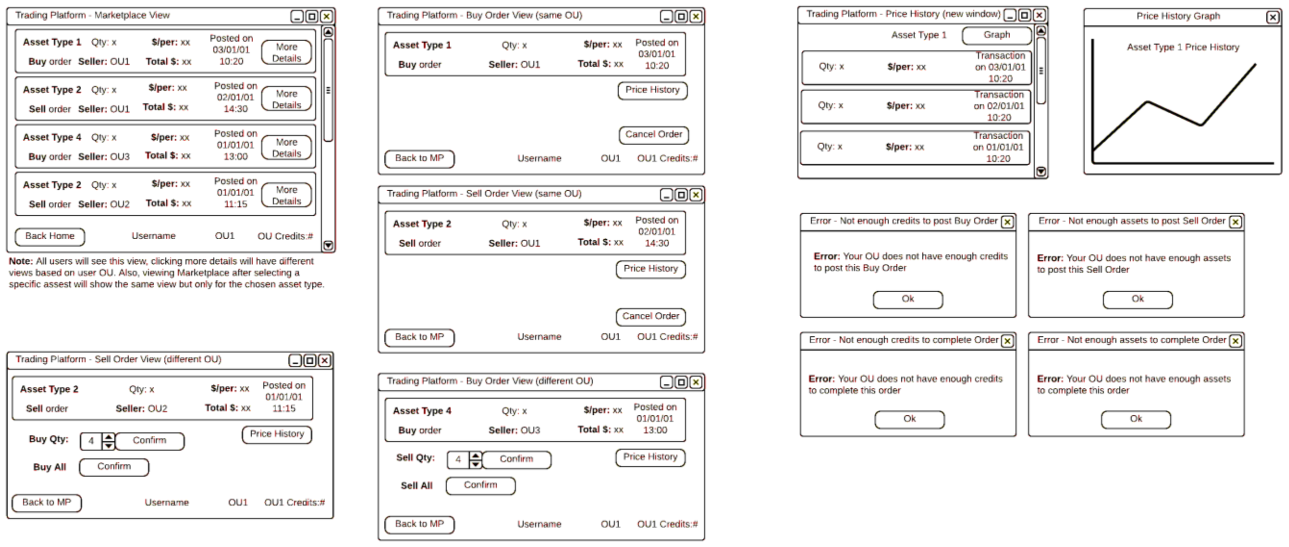


Figure 3 - trading Platform user interface GUI screens

Trading Platform GUI

The trading platform GUI screens represent the main functionality of the GUI. Allowing the user to search and view all current trades, while also allowing them to place new buy or sell request.


Trading Platform - Marketplace View
Asset Type 1 Qty: x $/per: xx Posted on 03/01/01 10:20 More Details
Buy order Seller: OU1 Total $: xx
Asset Type 2 Qty: x $/per: xx Posted on 02/01/01 14:30 More Details
Sell order Seller: OU1 Total $: xx
Asset Type 4 Qty: x $/per: xx Posted on 01/01/01 13:00 More Details
Buy order Seller: OU3 Total $: xx
Asset Type 2 Qty: x $/per: xx Posted on 01/01/01 11:15 More Details
Sell order Seller: OU2 Total $: xx
Back Home Username OU1 OU Credits:#
Trading Platform - Sell Order View (different OU)
Asset Type 2 Qty: x $/per: xx Posted on 01/01/01 11:15
Sell order Seller: OU2 Total $: xx
Buy Qty: 4 Confirm
Buy All Confirm
Price History
Back to MP Username OU1 OU1 Credits:#
Trading Platform - Buy Order View (same OU)
Asset Type 1 Qty: x $/per: xx Posted on 03/01/01 10:20
Buy order Seller: OU1 Total $: xx
Price History
Cancel Order
Back to MP Username OU1 OU1 Credits:#
Trading Platform - Sell Order View (same OU)
Asset Type 2 Qty: x $/per: xx Posted on 02/01/01 14:30
Sell order Seller: OU1 Total $: xx
Price History
Cancel Order
Back to MP Username OU1 OU1 Credits:#
Trading Platform - Buy Order View (different OU)
Asset Type 4 Qty: x $/per: xx Posted on 01/01/01 13:00
Buy order Seller: OU3 Total $: xx
Sell Qty: 4 Confirm
Sell All Confirm
Price History
Back to MP Username OU1 OU1 Credits:#
Trading Platform - Price History (new window)
Asset Type 1 Graph
Qty: x $/per: xx Transaction on 03/01/01 10:20
Qty: x $/per: xx Transaction on 02/01/01 10:20
Qty: x $/per: xx Transaction on 01/01/01 10:20
Price History Graph
Asset Type 1 Price History
Error - Not enough credits to post Buy Order
Ok
Error - Not enough assets to post Sell Order
Ok
Error - Not enough credits to complete Order
Ok
Error - Not enough assets to complete Order
Ok

Note: All users will see this view, clicking more details will have different views based on user OU. Also, viewing Marketplace after selecting a specific assest will show the same view but only for the chosen asset type.

Error: Your OU does not have enough credits to post this Buy Order

Error: Your OU does not have enough assets to post this Sell Order

Error: Your OU does not have enough credits to complete this order

Error: Your OU does not have enough assets to complete this order

Figure 4 - Trading platform marketplace GUI screens

Trading Platform Administration GUI

The administrative GUI is a backend interface that is only accessible to users with an admin role. These screens allow admins to add new assets, users and organisation. While also allowing them to delete them or change the credits or assets owned by an organisation.

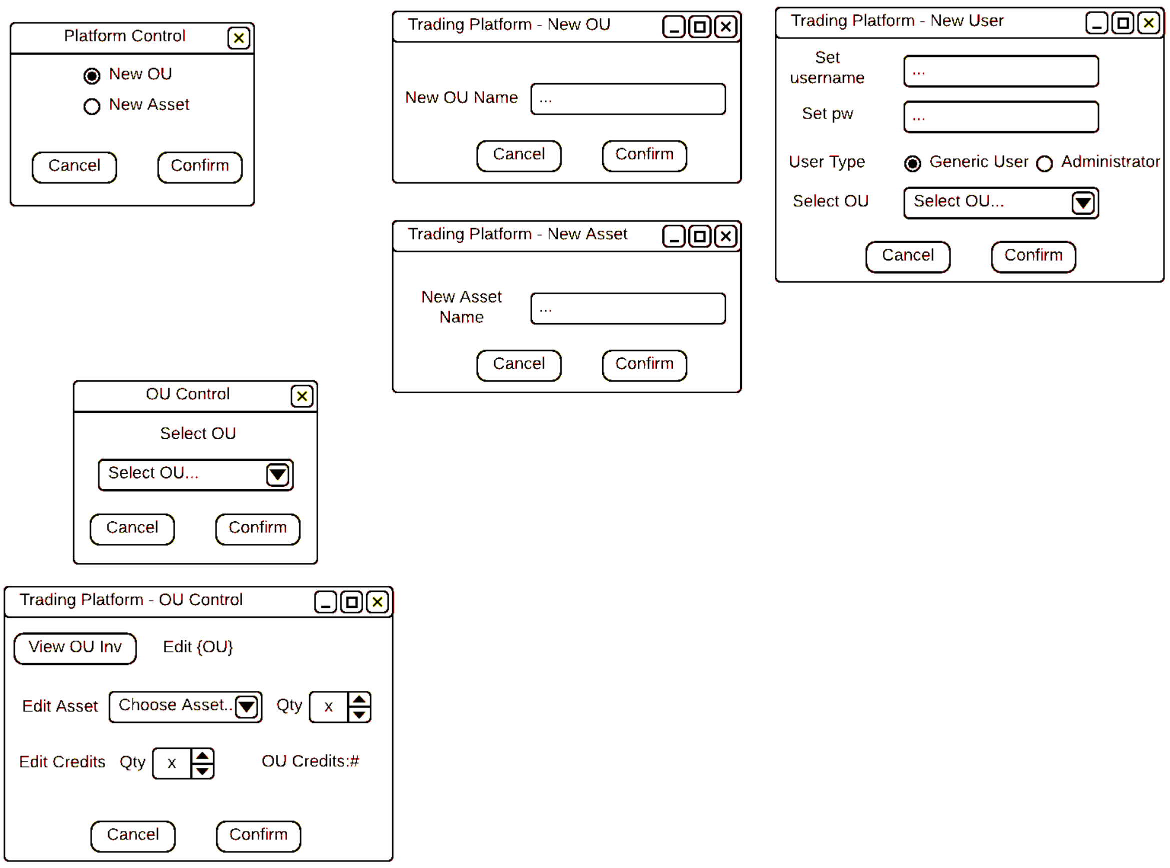


Figure 5 - Trading platform admin GUI screens

Database Schema

The database has been broken down into 7 tables, Users, Organisations, Trade, Asset, OrgainsationAssets, TradeHistoryLined, TradeHistory.

Assumptions:

- Users can only be part of a single organisation.
- An organisation can have many assets.
- An organisation can have many trades.
- A single asset type will be listed in each trade.
- The trade table will contain both sell and buy requests.
- A trade sell request only becomes history once all asset units are sold.
- A trade sell can have many buy requests.

The **Users** table contains all the information pertaining to a unique user such as names, password, and organisation. It also contains fields that can be used at a latter date if the client ever wants to expend the functionality of the trading platform such as an email address and logged in, in case email functionality or interacting with other users become a feature.

The **Organisation** table contains information pertaining to the various unique organisations, such as name and credits.

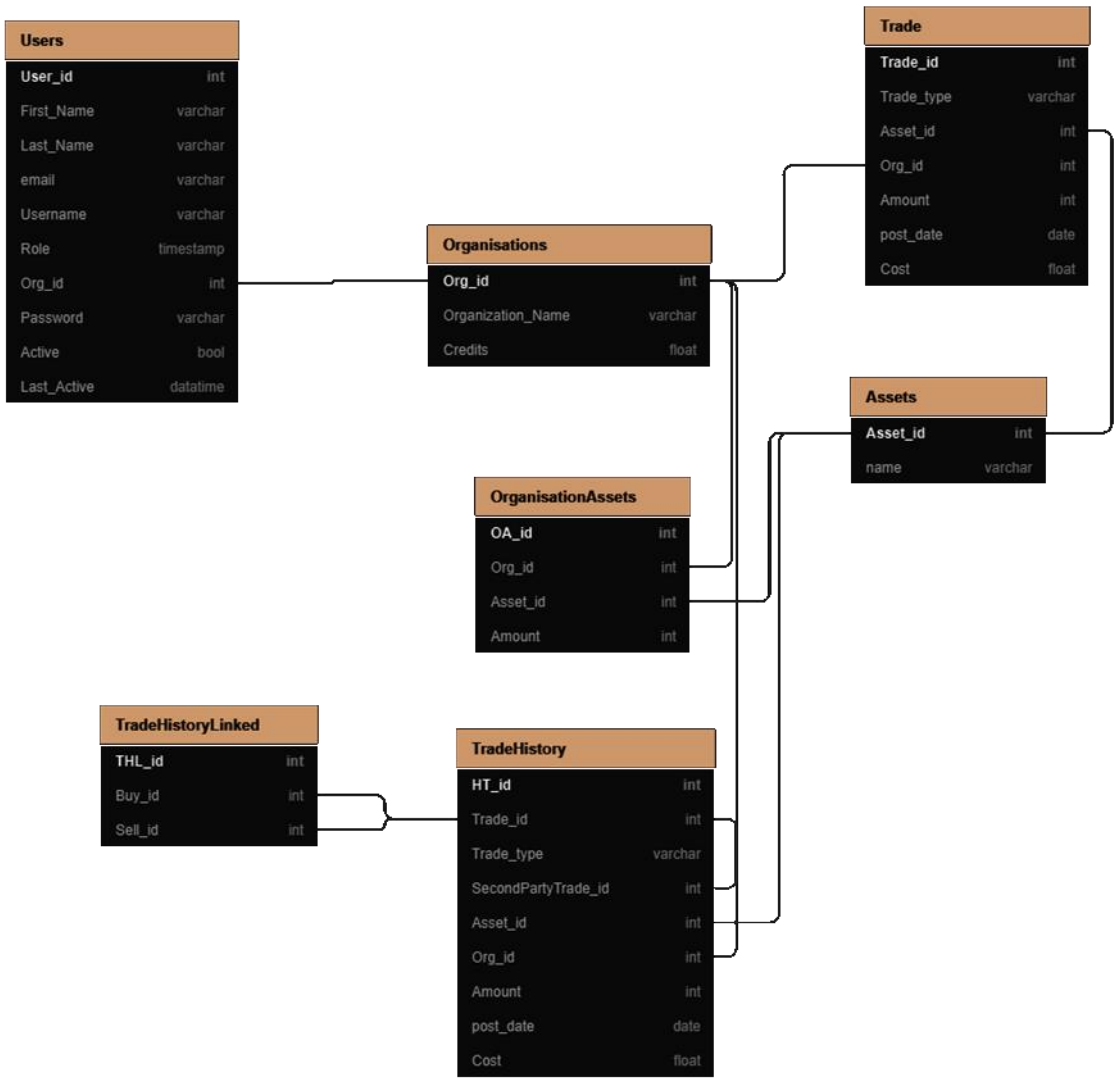
The **Trade** table contains the information relating to buy and sell request currently active on the marketplace, including the trade type, asset, amount and requested price.

The **Asset** table includes the information about each asset type, name and description.

The **OrganisationAsset** table Contains the information contains the information on what asset each organisation owns, linking the Asset_id and Organisation_id, along with adding the quantity of each asset an organisation owns.

The **TradeHistory** table, contains the information about past trades on the marketplace, including the post data, asset amount, sell price and the trade id.

The **TradeHistoryLinked** table contains the information about what buy and sell orders relate to each other in the TradeHistory table, with it having both a buy and sell id, which takes the id from the corresponding TradeHistory_id from the TradeHistory table.



Users	
User_id	int
First_Name	varchar
Last_Name	varchar
email	varchar
Username	varchar
Role	timestamp
Org_id	int
Password	varchar
Active	bool
Last_Active	datatime

Organisations	
Org_id	int
Organization_Name	varchar
Credits	float

Trade	
Trade_id	int
Trade_type	varchar
Asset_id	int
Org_id	int
Amount	int
post_date	date
Cost	float

Assets	
Asset_id	int
name	varchar

OrganisationAssets	
OA_id	int
Org_id	int
Asset_id	int
Amount	int

TradeHistoryLinked	
THL_id	int
Buy_id	int
Sell_id	int

TradeHistory	
HT_id	int
Trade_id	int
Trade_type	varchar
SecondPartyTrade_id	int
Asset_id	int
Org_id	int
Amount	int
post_date	date
Cost	float

Figure 6 - Database diagram representing the various tables and the connecting keys

Network protocol

To enable communication between the server and the client a thread on the server will constantly listen for new request from clients. When a new request is received the given information from the client will then be transferred over to the correct function, running on another parallel thread. In theory there should be around 5 threads running on the server, A connection thread, A buy request thread, sell request thread, Administrative thread, and an update thread. These 5 threads will encompass all major functions that will need to be preformed by the server, making sure that all requests are completed at a reasonable rate. The connection, buy and sell threads could also have an increased thread count if the hardware permits it, helping the server to handle an increased workload, with each of the main functions having 2+ threads each.

The Server thread will split out the client's request through interpreting the String sent from the client-side program. This will be done through splitting up the String at the character '|' (need to make '|' a reserved character meaning no user can add '|' to a username, password, or asset name). Once the string has been split up into an array, the first element will be an integer representing the task requested by the user, with the corresponding list being in table 1. The next elements in the array will vary depending on the task requested by the user with all other elements being inputs for the function requested. To send information back to the client the port opened to the server will be saved into hash map on the server, with the key being the connection IP, while the data is the socket. The key will then be passed along with the inputs to the function requested, so that any data can be returned to the correct port.

The codes for all the function have been split up by function type, so that the function codes look like 'ix' where 'i' is the function class and 'x' is the unique identifier for it within that class. For example any function pertaining the operation of the market place as an i of 2, while the sell function as an $'x'$ of 1 given it the server code 21, while a buy request has a server code of 22.

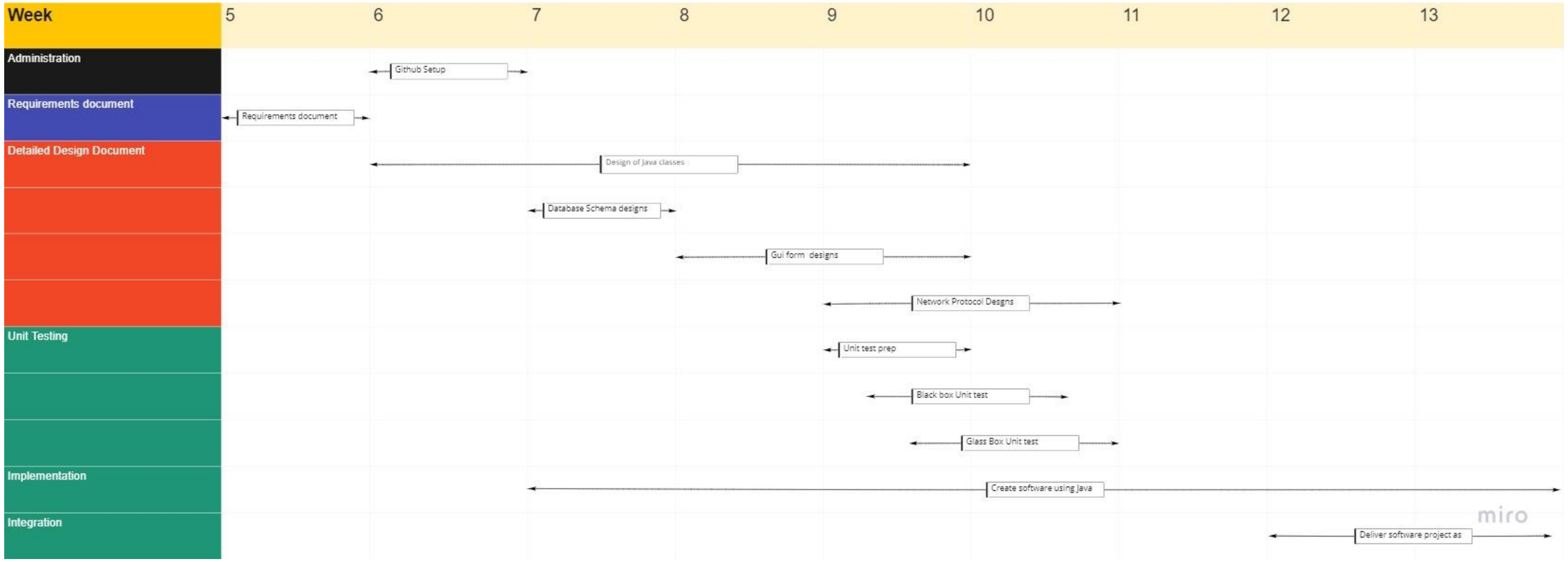
When a client has sent a request to the server the client will wait for a response to be given, with it being returned in the same way it was sent. However on the client side no task id will be sent back as the client already knows how to interpret the data since it has been waiting for the response from the server.

Server code	Function	Sent inputs
User Functions		
10	Login	(Username, hashed password)
11	Change password	(Username, hashed password, hashed new password)
12	View inventory	(OU_id)
Market Place Functions		
20	View marketplace	▨
21	Sell Request	(OU_id, asset, amount, price)
22	Buy Request	(OU_id, asset, amount, price)
23	View asset	(asset_id)
24	View asset listing	(trade_id)
Administrative Functions		
30	Add user	(user_id, hashed_password, OU_id, user type)
31	Add OU	(OU_name)
32	Add asset	(asset name, asset description)
32	Add asset to OU	(OU_id, asset_id)
33	Add credits to OU	(OU_id, amount)
34	Remove user	(user_id)
35	Remove OU	(OU_id)
36	Remove asset	(asset_id)
Helper Functions		
40	Fetch credits	(OU_id)
41	Fetch assets	▨
42	Fetch Organization assets	(OU_id)
43	Close Connection	▨

Table 1 - Task ID and arguments passed to the server from the client

Appendix

Sprint Planning for 3/05 – 16/05


Week
Administration
Github Setup
Requirements document
Requirements document
Detailed Design Document
Design of Java classes
Database Schema designs
Gui form designs
Network Protocol Desgns
Unit Testing
Unit test prep
Black box Unit test
Glass Box Unit test
Implementation
Create software using Java
Integration
Deliver software project as
miro