|  |
| --- |
| **MARSE SEP2 S18 – Software development** |

****

**Mihai Draghiciu, 238971  
  
Eric Volmer, 273448  
  
Andrei Mungiu, 273473  
  
Ronalds Andris Kalnins, 260506**

**Supervisor: Troels Mortensen, Bo Brunsgaard Christensen,   
Ib Havn**



**[23 136]**

**Software Engineering**

**2nd semester, Winter 2018**

**19-Dec-2018**

**Contents**

[1. **Introduction**. 3](#_Toc532946917)

[1.1 Problem Statement 5](#_Toc532946918)

[1.2 Delimitations 5](#_Toc532946919)

[2. **Requirements** 5](#_Toc532946920)

[2.1 Functional Requirements 6](#_Toc532946921)

[2.2 Non-Functional Requirements 6](#_Toc532946922)

[3. **Analysis** 7](#_Toc532946923)

[3.1 Use Case Description Diagram 7](#_Toc532946924)

[3.2 Activity Diagram 8](#_Toc532946925)

[3.3 Model diagram class 9](#_Toc532946926)

[4. **Design** 10](#_Toc532946927)

[4.1 MARSE design 10](#_Toc532946928)

[4.2 UML Class Diagram Design 11](#_Toc532946929)

[5. **Implementation** 14](#_Toc532946930)

[6. **Test** 18](#_Toc532946931)

[6.1 Test Specifications 18](#_Toc532946932)

[6.2 Test Plan: 18](#_Toc532946933)

[6.3 Commands tested: 18](#_Toc532946934)

[7. **Results and Discussion** 20](#_Toc532946935)

[8. **Conclusions**  20](#_Toc532946936)

[9. **Project future** 20](#_Toc532946937)

1. **Introduction**.

It is very important that the reader understands the following. The complexity of running a cryptocurrency mining farm has many hardware implications which inevitably change the way the entire operation runs. However, a solution for more stable activities such as inventory management and financial reports is required to allow focus on the more unstable part of the project, which involves constant market research and fast hardware adjustment according to market prices. It is therefore crucial that inventory and finance management have a very solid base for fast data viewing and editing.

In 2009 Bitcoin became the first established cryptocurrency. Although there have been other attempts at creating other cryptocurrencies, those mostly copied Bitcoin. The decentralization of this technology (which provides data integrity, partial anonymization, and data immutability) relies heavily on difficult mathematical algorithms, randomization, and blockchain technology (a way of storing data in a chain link).

In other words, the cryptocurrency is the result of old technologies combined in a new way.

As a service, the cryptocurrency named Bitcoin is based on the idea of having a very wide network of servers from which one is picked every 10 minutes to execute an encryption task that the network requires. This encryption process combined with a randomly picked server from a big network running the software produces a high level of anonymization. Furthermore, the network layer uses other randomly picked servers to check the reliability for each of the previously executed tasks on the network. These are called “confirmations”, the more confirmations a task (known as a transaction) has, the higher its reliability. Theoretically, you can never be 100% sure, practically it is almost impossible to not be sure after 6 confirmations, which accumulate as time passes.

From a hardware perspective, due to the increased collective work required to support the cryptocurrency network, a mining farm requires expensive hardware to make a profit. The profits being the system's algorithm that rewards hardware connected to the network with Bitcoin to their address. To afford such a server and coordinate the actions of contributors connected the crypto network, the current daily task of a cryptocurrency mine is keeping track of each members contribution. This is done using hardware purchase history, including details about who purchased, at what price, from where, when, its delivery status and its current LIVE activity when the hardware is connected to the mining network. The group eventually uses those details to correctly split among members the costs, profits, and the associated risks such as warranty claims and hardware depreciation.

On the financial side, a cryptocurrency mining operation must also be good at keeping track of its financial performance and monitor its worth in real life currency and cryptocurrency in parallel. Consequently, it is important to record exchange rates at the time of crypto transfers, allocated costs, and profits and keep track of these. They also reflect the hardware depreciation and transferred cryptocurrency to each member as well as their contribution to the group costs. These are all task which must be executed flawlessly.

Fast forward to 2018, our customer a mining farm, formed from 17 people ask us to help them. The project group has the name: JMP – Javelin Mining Project.

While the number of people has been relatively small the above-mentioned requirements where easily met. Now, the number of members is starting to grow, and this makes the management become cumbersome, time-consuming and open to errors. The customer is the one that takes care of recording the information about the mined cryptocurrency value and keeps track of the inventory in terms of GPUs, motherboards, locations, date of purchase, etc. and using excel is starting to become more of a bother than it’s worth.

Furthermore, due to the number of increasing members in the group, and not being in the same country as the hardware or even outside Denmark, the group members would like to have a secure chat system for their monthly meeting while viewing the relevant data at the same time.

Therefore, the customer has decided to invest in a software to reorganize all their information into an online database which can make their daily duties easier to manage. Upon our client’s wish, the system requires an online chat that can work at the same time as the data review itself so that better communication can be achieved.

* 1. **Problem Statement**

Communication is currently very difficult when the discussed data is not being displayed to group members that participate in the monthly group meetings. Data visualization is a current major drawback.

* Access and security (user separate access required, data editing access for administrators only)
* Users should have the possibility to access the available financial and inventory information from the project.
* Data visualization should be intuitive and easy to refer to in words a justification of the project  
  1. **Delimitations**

The delimitations are following:

* Use other tools to record LIVE hardware (server) performance and keep this information separate.
* It is not possible to display relevant market updates as that is a subjective matter.
* Do not display graphical representations of data in an attempt to make the data more readable.
* There is not enough time to work on automatic data inputs from external sources.

1. **Requirements**

The requirements of the application itself are as follows:

* Access and security (user separate access required, data editing access for administrators only)
* Users should have the possibility to access the available financial and inventory information from the project.
* Data visualization should be intuitive and easy to refer to in words.
* The admin will be able to manage all the data in the database, which consists of editing, creating and viewing.
* The user will be able to view the available information from the database.
  1. **Functional Requirements**

The users and admin will have access to the application at any given time.

The functional requirements of the application will be the following:

**The admin:**

* Should be able to view the financial.
* Should be able to view the inventory data.
* Should be able to chat with any online actors.
* Should be able to view the system logs regarding user logins
* Should be able to edit the inventory data.
* Should be able to edit the financial data.
* Editing happens in the same window where the data is being viewed.

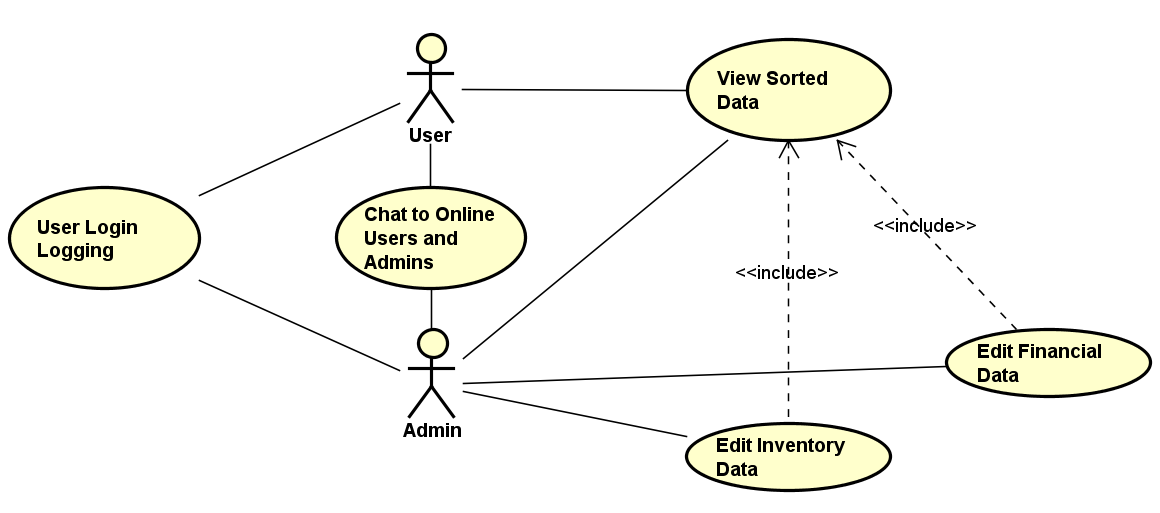
**The user:**

* Should be able to view the financial data
* Should be able to view the inventory data.
* Should be able to chat with any online actors.
* Should be able to view the system logs regarding user login.
  1. **Non-Functional Requirements**

Non-functional requirements will be:

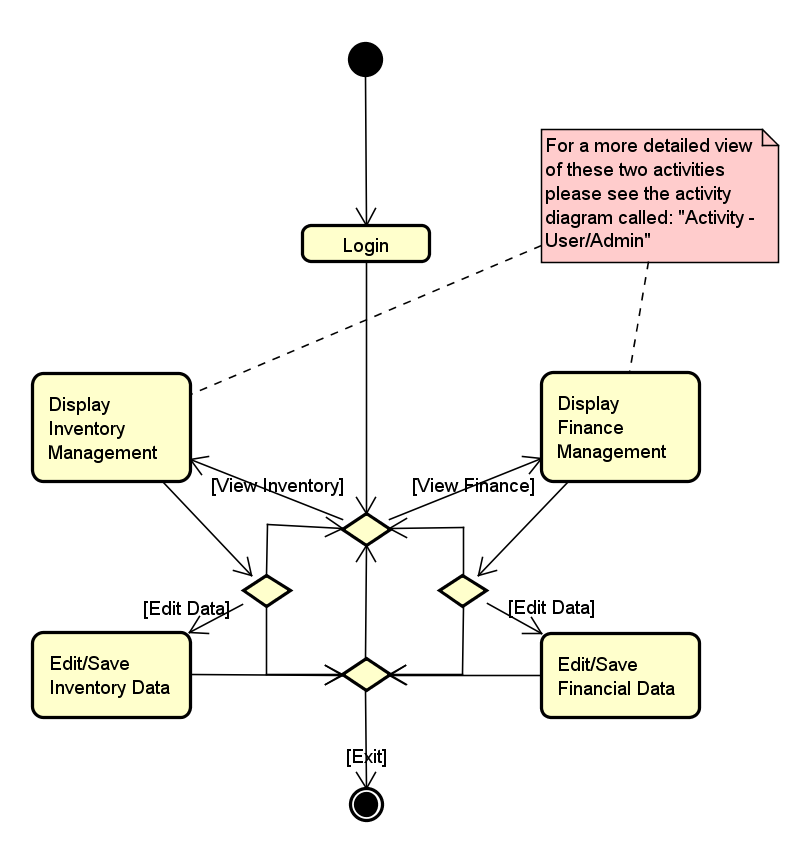
* At any given time, the system response time should be:
  + Viewing data under 10 seconds.
  + Editing data under 1 minute.
* Each method should have a full description of its use case.
* The mean time between failures is not more than 1 day.

1. **Analysis**
   1. **Use Case Description Diagram**



**User Login Logging:** Users log-in will be logged as per customer requirements This should happen automatically.  
**Chat to Online Users and Admins:** Users and Admins can have a communication channel between themselves and all other logged in actors as soon as they authenticate to the application.  
**View Sorted Data:** The user and admin can view the sorted data currently present in the database. This includes the inventory data and financial data.  
**Edit Inventory Data:** The inventory data (gpu, cpu etc) that is currently stored in the database, can be edited by the Admin only.  
**Edit Financial Data:** The financial data (project, electricity etc) that is currently stored in the database, can be edited by the Admin only.

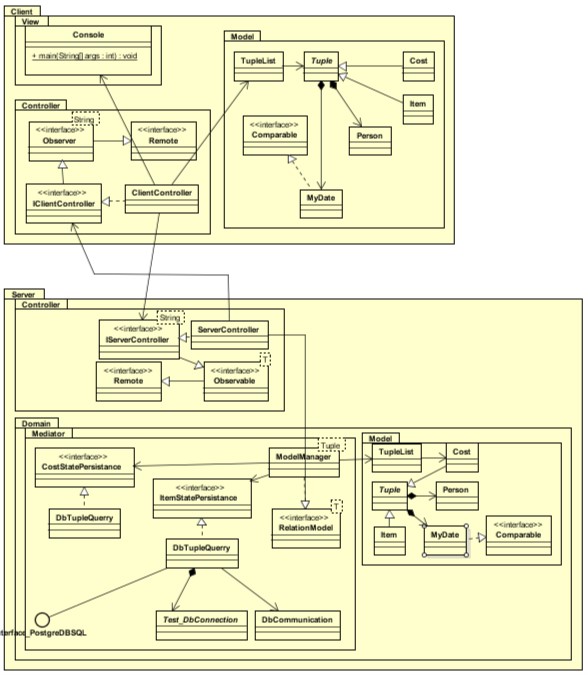
* 1. **Activity Diagram**



This activity diagram walks us through all use case scenarios from a general perspective. It shows how an actor (admin in this case) can interact with all the stored data from the database by using the java application.

The distinguishing possibilities of an admin is to edit and save available financial and inventory data. A user would only be able to display the financial and inventory data but his actions would run through a very similar activity diagram.

* 1. **Model diagram class**

****

This diagram is an early representation of how the connection is predicted to look between the database and the server, as well as the server and the client. The purpose here was to implement the MVC model from the start and build all other design patterns on top of it as the application is built.

1. **Design** 
   1. **MARSE design**

Since the project main purpose is to mainly facilitate easy data read and write access it is imperative that we use design patterns which help solve such standard issues as: remote method invocation, instance uniqueness, database connection, layered and logical design as well as dependency inversion and few others.

Successfully implementing these design patterns allowed our group to avoid errors in the core architecture of the application and saved us a lot of coding time. Consequently, the group was able to focus more on prioritizing application features and proper SCRUM management.

The application is composed from two main parts: The first part is made with the Java (programming language) the second part is made with the PostgreSQL database language.

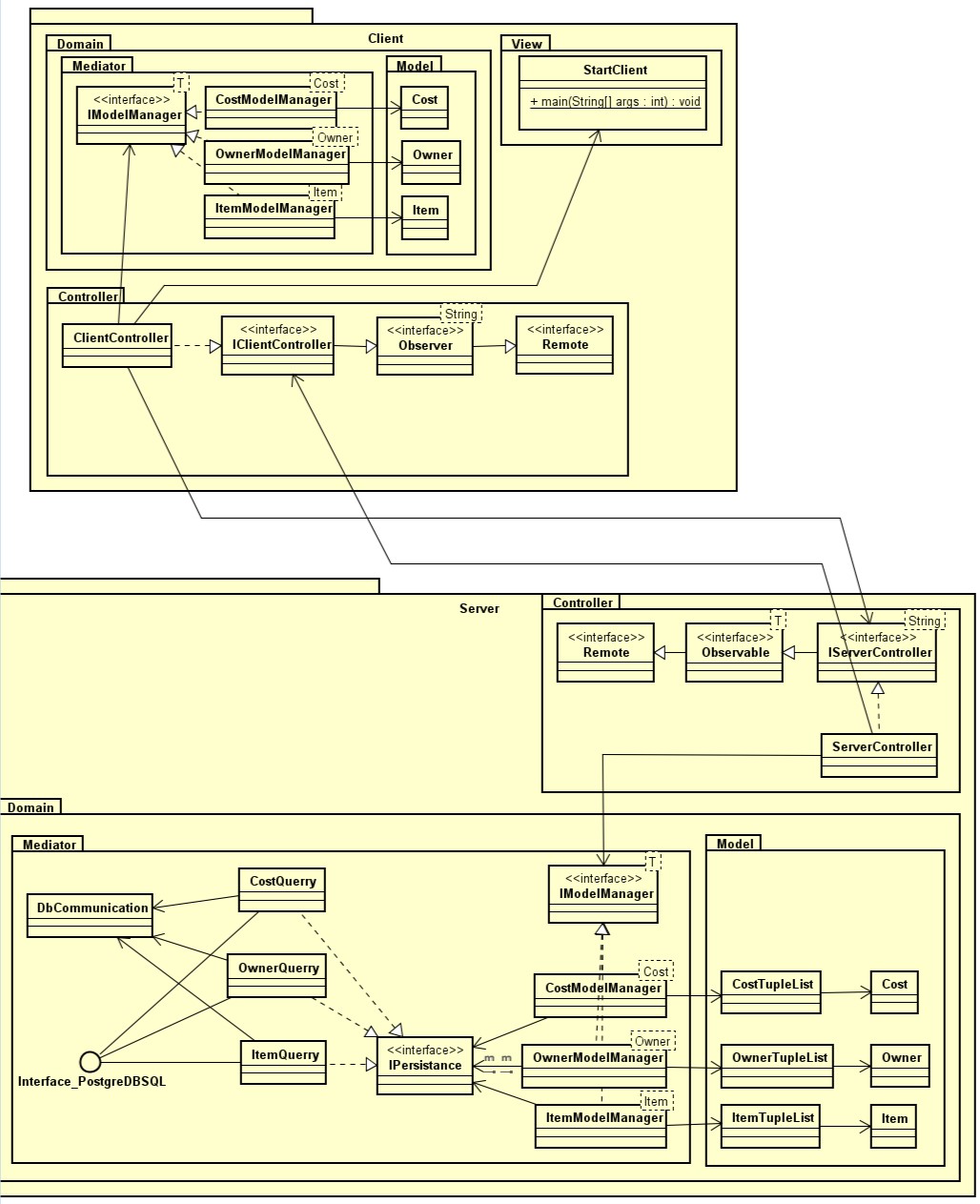
* **Java Packages (MVC Style)** In Java, the application was built with package names that reflect the Module View Controller design pattern. The Domain package is responsible for database access and includes the model and the mediator which have the roles of shaping and querying the data respectively. Inside the “Mediator” package the “Adapter design pattern” is used for converting the ResultSet received from the database into Client friendly data and to facilitate remote data transfer. To fully use the provided functionality the user is provided with basic functionality such as viewing, inserting and updating three types of data: Cost data, Item data and Owner data.
* **UML**

Additionally, Astah was heavily used for planning the entire project starting as early as the inception phase.

* **Version Control** For version control our team has used GitHub, this allowed us to ensure the application transitioned smoothly from one phase to another without damaging the already existing code.
* **Database**  
   Since the customers main requirement was shaping the already available data our team has decided to use a data modeling technique called normalization.

The three normalization steps that we have implemented allowed us to remove duplicate data and redundant data attributes. It also helped us group the available data into fewer but more informative data relations.  
 The improvements on the database side together with properly implemented design patterns on the Java application side made our code very manageable and modular. New features can now be plugged in without affecting any of the code that was written so far. This was especially important since the group knew from the start that the UI and the chat system might be move out towards the later development stages of the project.

* 1. **UML Class Diagram Design**

****

Above is the final version of the Class Diagram which perfectly represents the use of MVC, RMI, Dependency Inversion Principle, Layered Design and Separation of Concerns. The developer team has used several other design patterns which are not visible only by looking at the class diagram. These are: Singleton, Adapter, Proxy.

* **StartServer** 
  + The class with a main method that registers the server Interface into the registry and instantiates the ServerContoller.
* **StartClient**
  + The class with a main method that registers the client Interface into the registry and instantiates the ClientContoller.  
    This class is also waiting for actor input to the console and calls methods based on the input which later forward the request to the server and display the received information from the server.
* **ServerController.**
  + The server controller receives the remote method calls and returns to the client the requested serializable data. It also holds a list of observers which can be notified to take some action.
* **ClientController.**
  + The client controller receives the request from client interface and forwards it to the server controller through remote method invocation (RMI). It is the connection between the Model and the View on the client side.
* **Cost, Owner, Item** 
  + These classes represent objects, both on customer and client side. They are used as containers for the data the client and the server receive. When these objects are filled with data, due to their serializable nature they can be sent to a remotely located machine that is connected to the application.
* **CostModelManager, ItemModelManager, OwnerModelManager (Singleton classes)**
  + These classes act as adaptors between the database and the server controller. They are the single point of merging data into the model. This converts PostgreSQL data into serializable objects (using the model) to be passed on to the client, as well as it converts client requests into PostgreSQL statements (again using the Model) to be passed back to the database for execution.
* **CostTupleList, OwnerTupleList, ItemTupleList (Singleton classes)**
  + Using the singleton design pattern on these classes allows for control over the data uniqueness and reliability. These classes hold a list of respective object types that are used for grouping multiple objects into a single list for facilitating ease of data transfer.
* **IModelManager <<interface>>**
  + Stores methods of generic type that are later being implemented as the specified types of Model Manager. Also satisfies the dependency inversion principle (DIP) and acts as an adapter
* **Observer <<interface>>**
  + Extends Remote(RMI) interface to allow remote method invocation on its core methods. It is used to register clients to the server observer list and is part of the Observer Pattern.
* **Observable <<interface>>**
  + Extends Remote(RMI) interface and is implemented by IServerController responsible for registering observers to observer list and notifying them when required. It is part of Observer pattern.
* **IPersistence <<interface>>**
  + Stores methods, that are later implemented by Query classes. Also fulfills the requirements of the dependency inversion principle (DIP).
* **OwnerQuerry, CostQuerry, ItemQuerry**
  + Using the singleton DbCommunication class, these classes are reading, updating and inserting prepared statements to the database.
* **DbCommunitacion**
  + This class establishes connection to the database and specific schema on instantiation using java.sql.Connection class. It is a singleton to ensure that only one connection is established and used for all database communication.

1. **Implementation**

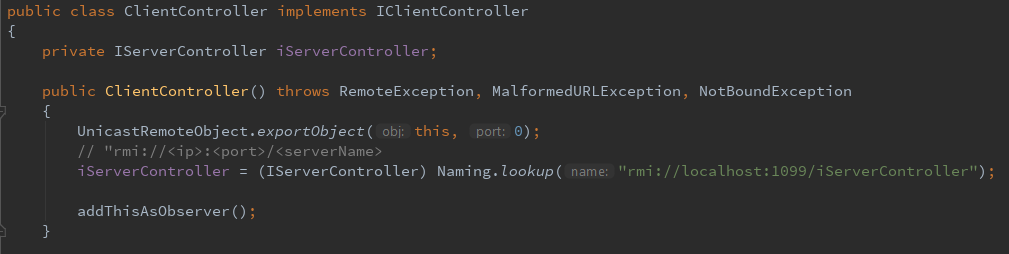
Below we will run the reader through an example of how the user inputs generate a “Cost table”

* **Part 1:** Client uses this class to request for a cost relation  
  **StartClient** is used to request a cost relation from the database. This class also contains the “help” request in order to show the user the list of available commands and what their descriptions.

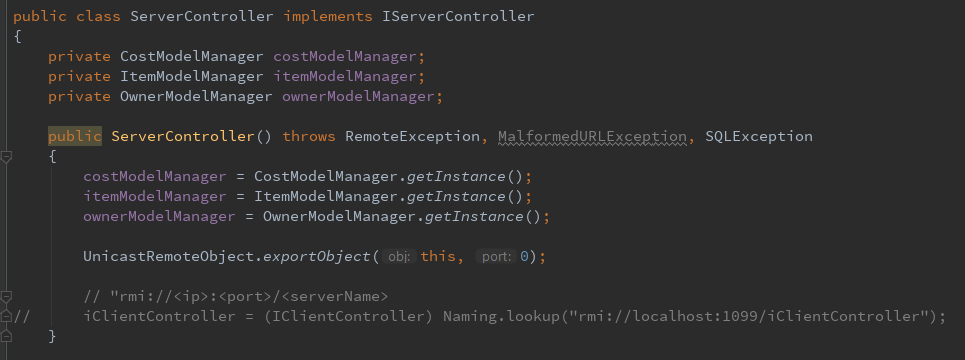


* **Part 2:** The client controller receives the request and forwards it to the Server Controller through RMI.

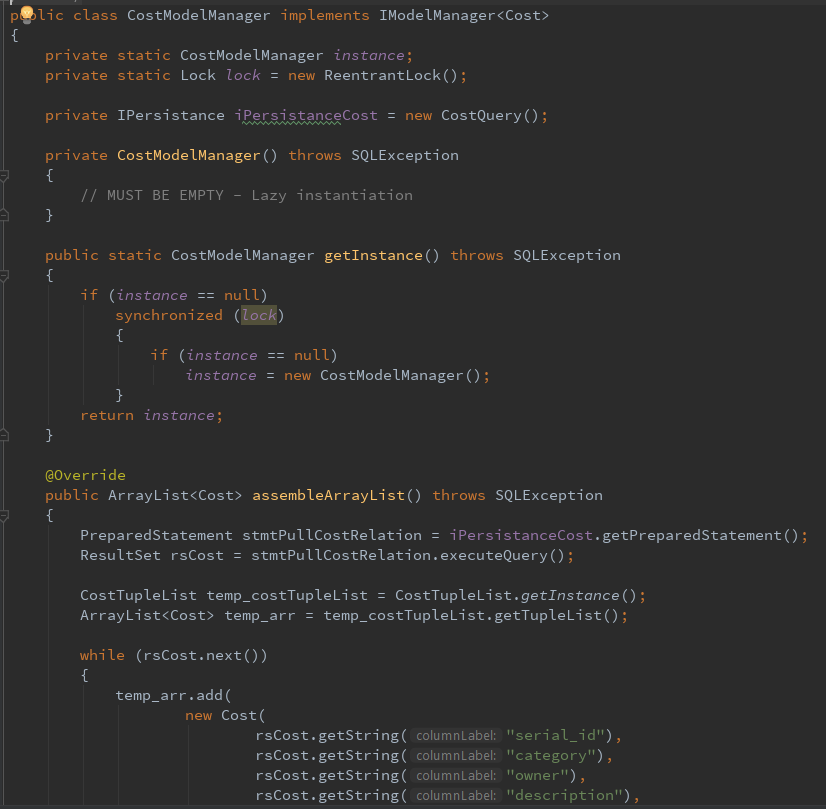
Class **ClientController** receives the request and forwards it to the Server Controller through RMI.



* **Part 3:** The server controller receives the remote method call and executes the command.

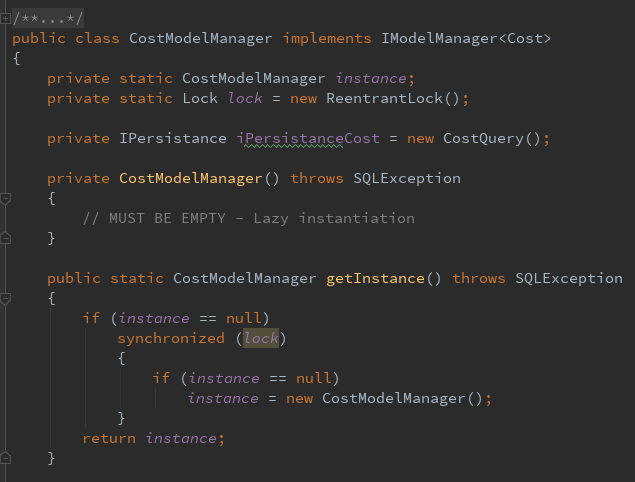
Class **ServerController** receives the remote method call and executes the command.

* **Part 4:** The Cost Model manager receives the request and starts assembling Objects by:
  + Querying the database for information.
  + Instantiating the model.
  + Looping through the ResultsSet received from the database and inserting it into the instantiated model.
  + Returning the instantiated Object with all the data stored in it.

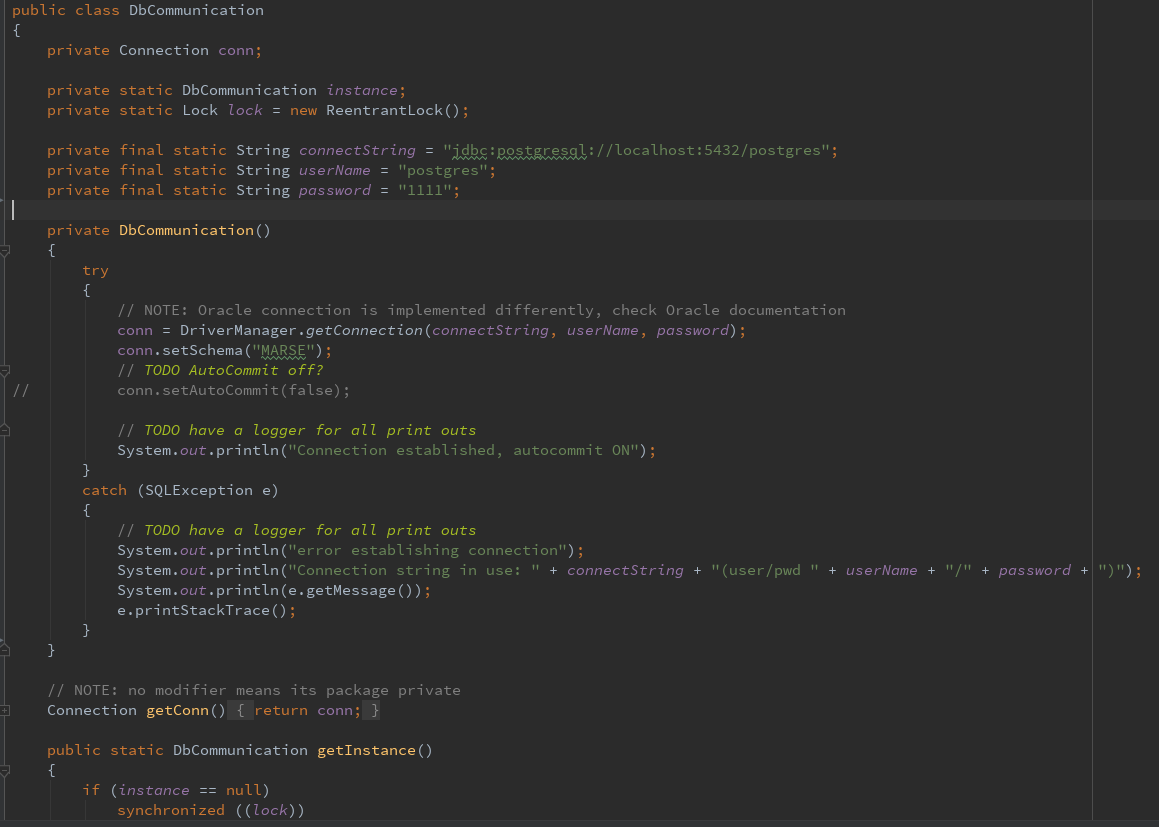


* **Part 5:** The request for data is received and execute according to the class description below.
* **Method Cost Model Manager**

This class acts as an adaptor between the database and the server controller in the single point of merging data into the model. The class was implemented poorly at first and did not fit the functionality it required therefore after adapting a Singletons Pattern for it all the classes that implement the model interfaces have been adapted to Singletons.

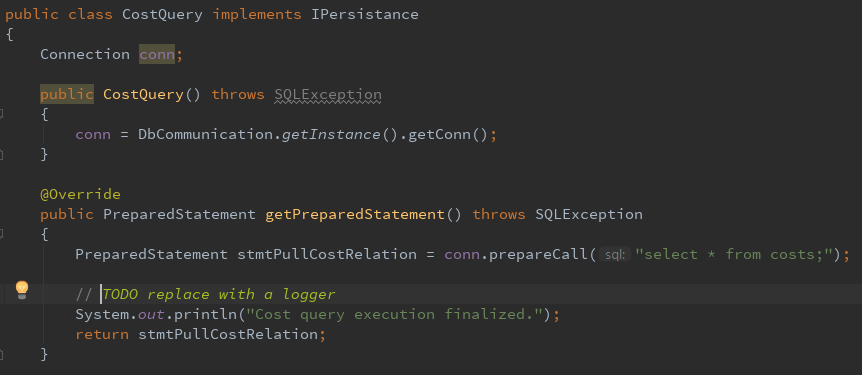


* **Part 6**: DbCommunication instantiates the connection to the database as requested.

Class **DbCommunication** is a singleton and is instantiated in **CostQuerry** class and connects to database and schema using specified statements using connection string, username and password.  
****

* **Part 7:** The Cost Query send and executable PreparedStatement to the Database.

Class **CostQuerry** extends **DbCommunication** class and implements **IPersistance**, this ensures the connection to database and allows to receive, send or even insert prepared SQL statements from/to database.

****

* **Step 8:** The received and assembled data from the database travel all the way back to the client through RMI.

1. **Test**

The testing method that was selected (and for which the Developers had time) was the White-Box testing.

* 1. **Test Specifications**

Goals and Objectives – To streamline as much as possible the process of software

quality assurance by relying on design patterns.

Statement of Scope – The testing must be conducted on all aspects of software that are

responsible for reading, writing and updating information in the User interface or inside

the binary files. It must also be conducted at the end of each Sprint.

Major constraints – The lack of UI which would allow for black box testing.

* 1. **Test Plan:**

The overall testing strategy is to test often and test fully, this goes hand in hand with the SCRUM framework and testing at the end of each Sprint.

* 1. **Commands tested:**

1. Help – lists all commands
2. Costs – lists all costs.
3. Items - lists all items.
4. Owners - lists all owner.
5. Insert Cost - inserts fake hard coded cost.
6. Insert Item - inserts fake hard coded item.
7. Insert Owner - inserts fake hard coded owner.
8. Update Cost - updates a cost with hard coded data.
9. Update Item - updates an item with hard coded data.
10. Update Owner - updates an owner with hard coded data.

The functionality of the above commands was tested and they have fulfilled all functional requirements. In the case of any “command” other than the ones specified is inserted, the program will simply not react, this is intentional. Such kind of behavior allows us to easily plug an interface to the class which contains the trigger commands.

The White Box testing started as soon as the RMI Design Pattern has been implemented.

However, since most of the design pattern implementation happened while looking at the UML the white box testing was mostly applicable for minor events such as testing methods. The classes that used up the majority of the time our team has dedicated to testing where the adapter classes from the Mediator package. Since in those classes (such a -% of the application is involved in building the correct objects from the correct data by executing correct queries to the database.

The group has gathered many valuable insights regarding the importance of dedicating enough time to choosing the appropriate design patterns, since these help solving major problems in a predefined way, thus saving allot of time.

As the group went through various white box testing phases, we realized that as predicted many features will not be implemented due to the lack of time. However, again due to the implementation of design patterns our code proved to be very modular and was structured in such a way that future software upgrades are very easy to connect to the already existing interfaces.

The white box testing phase has also been crucial in helping adjust our burnup and burn down charts since it is only after functionality has been tested that it can be inferred weather it is finalized or requires more work. Thus, we concluded that continuous testing at the end of each Sprint is a very crucial in helping determine the backlog for the next sprint and focusing the available resources and time in the right direction.

The SCRUM methodology allowed us to conduct very few but targeted tests and therefore helped us develop the application in a more controlled and predictable manner.

As a conclusion, our group considers that all currently implemented and user available functionality has been fully tested and working.

1. **Results and Discussion**

The project fulfills part of the requirements but has all the required infrastructure for future software upgrades and unfinished use cases ready. The data transfer from EXCEL to Java/SQL was successful.

The core idea of establishing a communication between clients and server has been completed as well.

The users have the possibility of viewing the data in its entirety. The commands for data viewing are easy and intuitive to use.

Requirements that were not met:

Communication between members during the data viewing/ monthly meetings has not been met developed. This has been predicted as a likely week point from the start and the team took all necessary precaution to develop the application in such manner that not fulfilling these requirements will not hinder the performance of other parts of the application.

Access and security requirement has not been met due to the same reasons as the communication requirement and with the same effect on other application parts.

1. **Conclusions**

1. **Project future**

In terms of project future, the application itself can be adapted to different databases as long as the connection between JAVA and SQL are handled correctly.

For the future, a communication service that wasn’t added during development can be added to the application in order to make meetings between the members go better.

Also, another functionality that was not added during development was access and security as there is no implementation for users and administrators therefore everyone can edit the information, this is an issue that has to be resolved.

1. **Source of information**

[Marr](https://www.forbes.com/sites/bernardmarr/) B. (2017), A Short History of Bitcoin and Crypto Currency Everyone Should Read

Available at: <https://www.forbes.com/sites/bernardmarr/2017/12/06/a-short-history-of-bitcoin-and-crypto-currency-everyone-should-read/#3bab870a3f27> [15 Sep 2018]