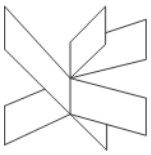


PROJECT REPORT

Looking for Devs – Freelancing Platform

Client/Server System

Students

Stefan Harabagiu – 266116

Andrei Cioanca – 266105

Christian Sørensen – 267142

ICT Engineering 2nd Semester

Supervisors

Mona Wendel Andersen

Joseph Chukwudi Okika

Ib Havn

Knud Erik Rasmussen

Students

Andrei Cioanca – 266105

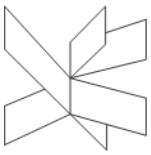
I hereby declare that my project group and I prepared this project report and that all sources of information have been duly acknowledged.

Stefan Harabagiu – 266116

I hereby declare that my project group and I prepared this project report and that all sources of information have been duly acknowledged.

Christian Sørensen – 267142

I hereby declare that my project group and I prepared this project report and that all sources of information have been duly acknowledged.



Acknowledgements

The team wishes to formally acknowledge VIA University College's active implication in the evolution and completion of this project work. All documents contained or referred to from this project are based on VIA University College's templates and designs. All information present in this document, while original, could not have been made possible without VIA's designs and templates.

Version history

Current: Version 1.8.1

Version 1.0.0 – 06.04.2018

Version 1.1.0 – 13.04.2018

Version 1.2.0 – 20.04.2018

Version 1.3.0 – 27.04.2018

Version 1.3.2 – 04.05.2018

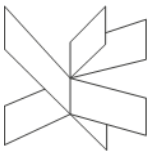
Version 1.5.0 – 11.05.2018

Version 1.6.0 – 18.05.2018

Version 1.7.0 – 25.05.2018

Version 1.8.0 – 01.06.2018

Version 1.8.1 – 08.06.2018



Abstract

In today's IT industry there is an increased appeal for freelancing. People see it as an opportunity to work in a flexible environment where they can choose their own working hours and the field in which they will work. Moreover, the usual payment rates for freelance workers exceed the ones of full-time employees.

Our product, "Looking for Devs", abbreviated, "LFD" is looking to fill the niche that is dedicated platforms for IT-based freelancers on PC.

The system has been developed in Java and incorporates a robust and secure database. It fulfills most of the Product Owner's requirements and stands as a robust and easily manageable system that can hold the user's data.

Based on the Product Owner's desires the system can keep track of two types of user data, one type for the regular user and one type for the companies. It can keep track of announcements for both users and companies, it can keep track of what users applied to which announcement and can easily save the data to the database it is linked to in order to not lose any critical information.

The system also incorporates a GUI, made simple and clean in order for the user to navigate it easily. It facilitates the use of the system by providing an intuitive interface for interaction.

In the end the Product Owner received a robust and malleable program coupled with a secure database in order to further his business goals.

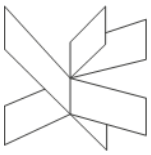


Table of Figures

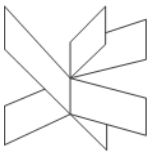
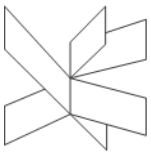


Table of Contents



I. Introduction

In today's IT industry there is an increased appeal for freelancing. People see it as an opportunity to work in a flexible environment where they can choose their own working hours and the field in which they will work. Moreover, the usual payment rates for freelance workers exceed the ones of full-time employees.

As with any other type of employment, freelance work is susceptible to a supply and demand (Source) within a specific market which is a lot more volatile than full-time employment. Freelancers don't benefit from a stable work environment, as they experience frequent change between employment and unemployment status.

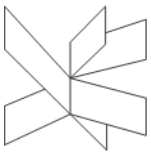
However, in the US there is an increase (Source) in freelance workers and it is expected that they will become a workforce majority in the near future. In the world of freelancing IT jobs (Source) are the highest paid. This may be because of changing trends within society or an increasing demand from IT companies. From a stakeholder's point of view, hiring a freelance worker is a lot cheaper (Source) in the long run than full-time employees as they do not require health insurance, paid leave, company benefits, holiday allowance, etc.

Having mentioned these, at the moment freelancers have a hard time finding work and making connections within the industry in order for their employment to become more stable. Large companies are not the only ones benefiting from freelancers, a good portion of those benefiters are small start-up companies that do not have the gravitas to attract and keep full-time employees for long periods of time.

A small company does not benefit from a HR department so they have to actively search on different platforms for an employee. Apart from this example, there are also people that have small projects in development that may want to hire freelancers, but have no easy way of keeping an organized working environment between themselves and the freelancers.

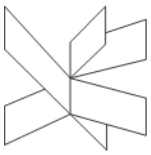
In Denmark the CVR and CPR numbers represent a form of identification for companies and individuals respectively. Freelancers face the danger of coming in contact with fake contractors and risk not getting paid. Furthermore, there is also the risk of facing identity theft from other freelancers, especially when they are not "protected" by a large company. In Denmark, however, the use of a CPR number through NemID ensures that people can legally and easily identify themselves online. (Source)

Denmark also makes use of part time workers as an important part of its economy. There are many recruitment websites that offer a secure way for people to find temporary jobs(link-Vikar). A lot of those websites use some kind of database to hold the user's credentials, this makes most of them vulnerable to credentials and user data loss.



While “vikar-type” recruitment agencies may focus on manual labor which is thoroughly satisfied throughout Denmark, many IT-based freelancers lack a proper platform to search for employment which in turn is basically a more specialized version of the former. (Source)

Payment methods for freelancers vary greatly from platform to platform. Some of them, like Upwork and Freelancer have a variety of methods the user can choose from, Milestone Payments, Invoices and Transfer Funds. (link) The payment comes directly from the employer. On Worksome, however, a Danish freelancing website, the payment is issued by the website itself and they use Invoice as a method of payment. (Source)



II. Requirements

The requirements below overall represent what the Product Owner would like the system to do overall.

II.1 Functional requirements

1. A user should be able to apply to an announcement.
2. A user should be able to browse a list of announcements.
3. A company should be able to publish announcements.
4. A company should be able to see a list of people interested in their announcement.
5. A company should be able to hire someone from the list.
6. The system should be able to save all current data to a database.
7. The system should be able to read all current data from a database.
8. The system should be able to maintain a constant connection to the database.

II.2 Non-functional requirements

1. The system has to be implemented in Java.
2. The system should use a database for primary storage.

The team feels these requirements are suitable for what the Product Owner envisions the program to look like in the end. These make the system complex in nature while retaining a robust and clean structure with good maintainability. These requirements are better explained and detailed in the next section of the project report, the analysis, where the group presents the system itself through diagrams that would aid the uninitiated in understanding the team's design.

III. System Analysis

The outline of the system is as follows, it contains two types of clients, users and companies. The two clients interact with each other through the two main activities of the program, signing up for and creating jobs respectively. The companies can add or remove announcements and hire a specific user from the list of those that applied to that respective announcement. The users in turn can permanently access a list of all the available announcements and are able to sign up to any number of them with the information they fill in during the registration process.

The figures below show the basic actions that each type of client can do.

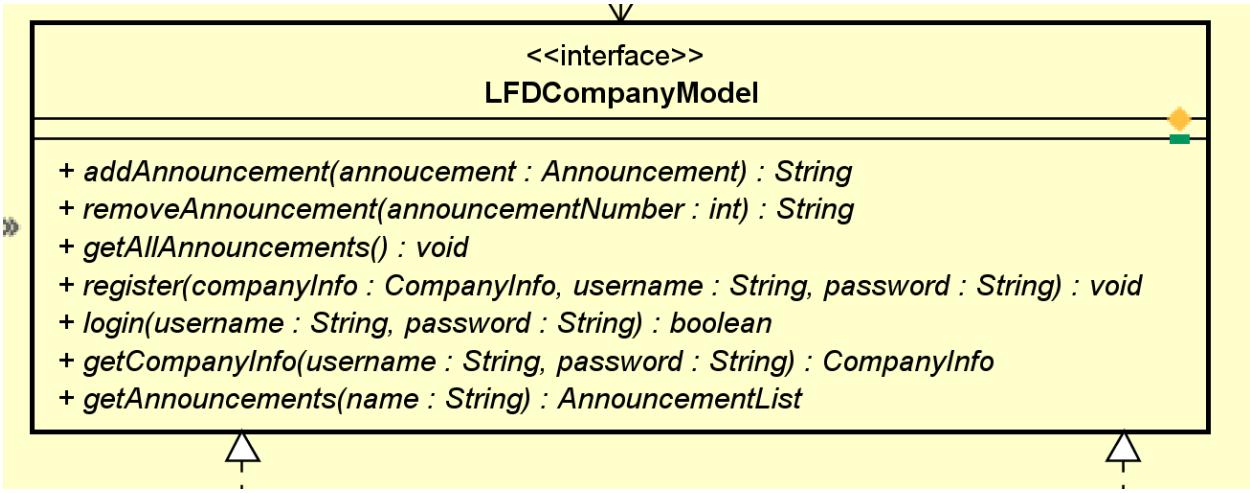


Figure 1 - Company Client Model Interface

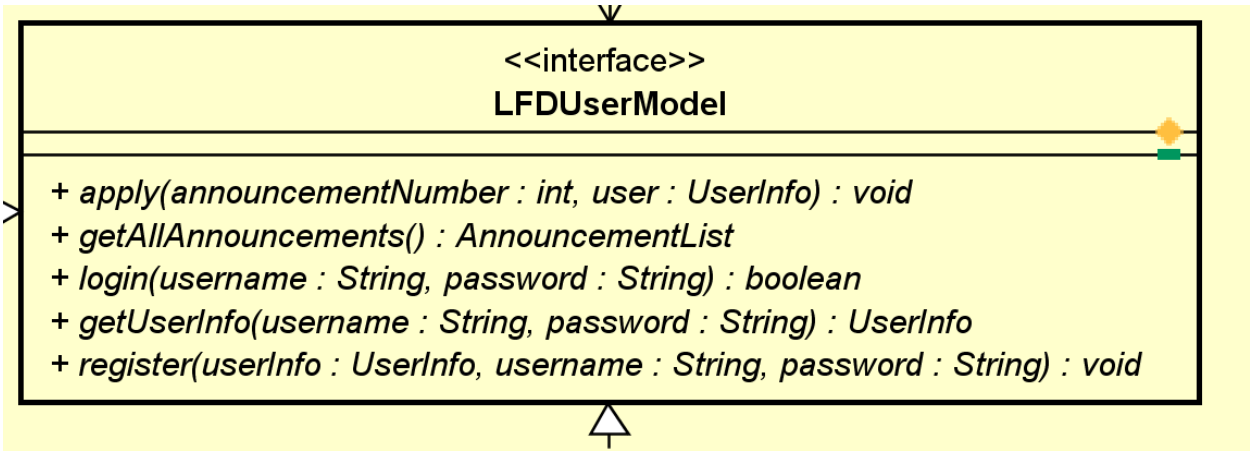


Figure 2 - User Client Model Interface

To accompany them are included detailed descriptions for two specific use cases in the system, listing the available announcements from the perspective of a user and creating a new announcement from the perspective of a company.

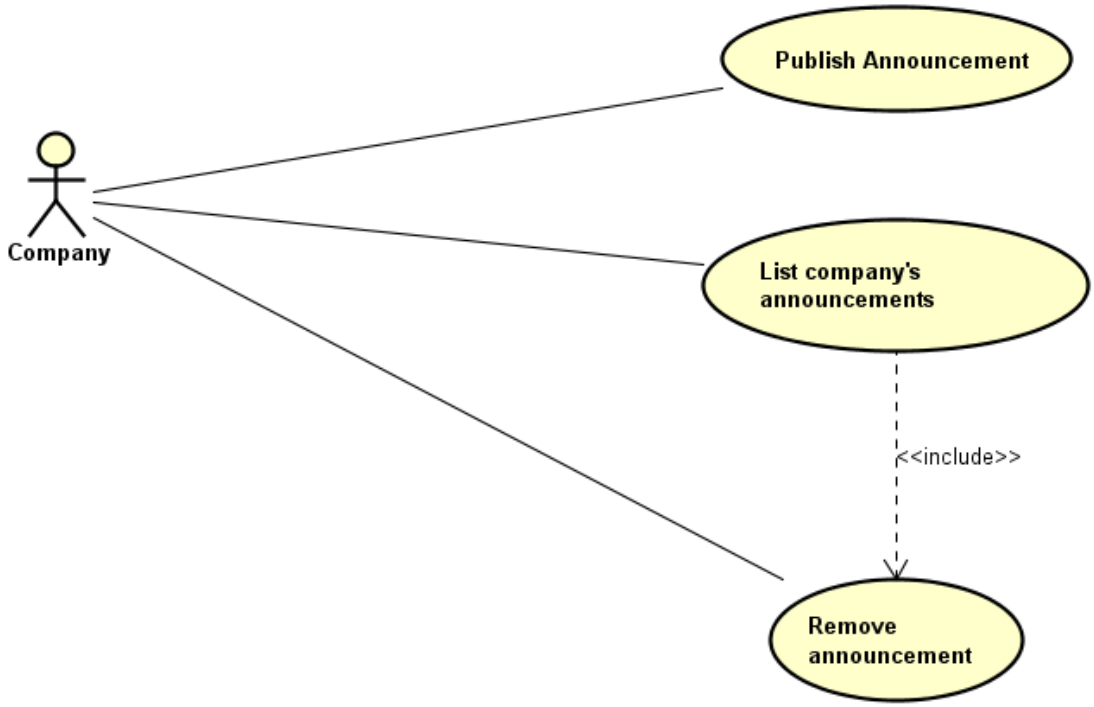


Figure 3 - Company Use Case Diagram

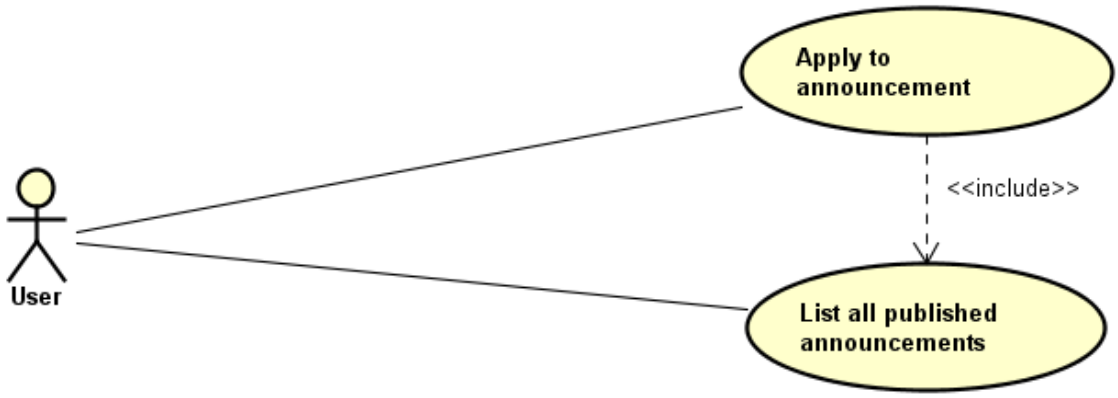


Figure 4 - User Use Case Diagram

In order to put those use case diagrams into perspective, here is an activity diagram for one of the more complex use case, in order to paint a more comprehensive picture of how the system works.

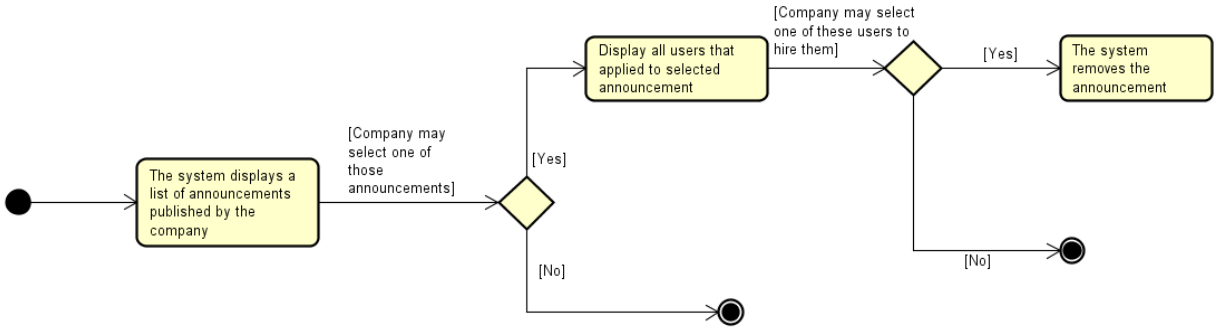


Figure 5 - Activity Diagram

Moreover, a System Sequence Diagram was made after this specific UseCase in order to have a clearer view of the system during this process.

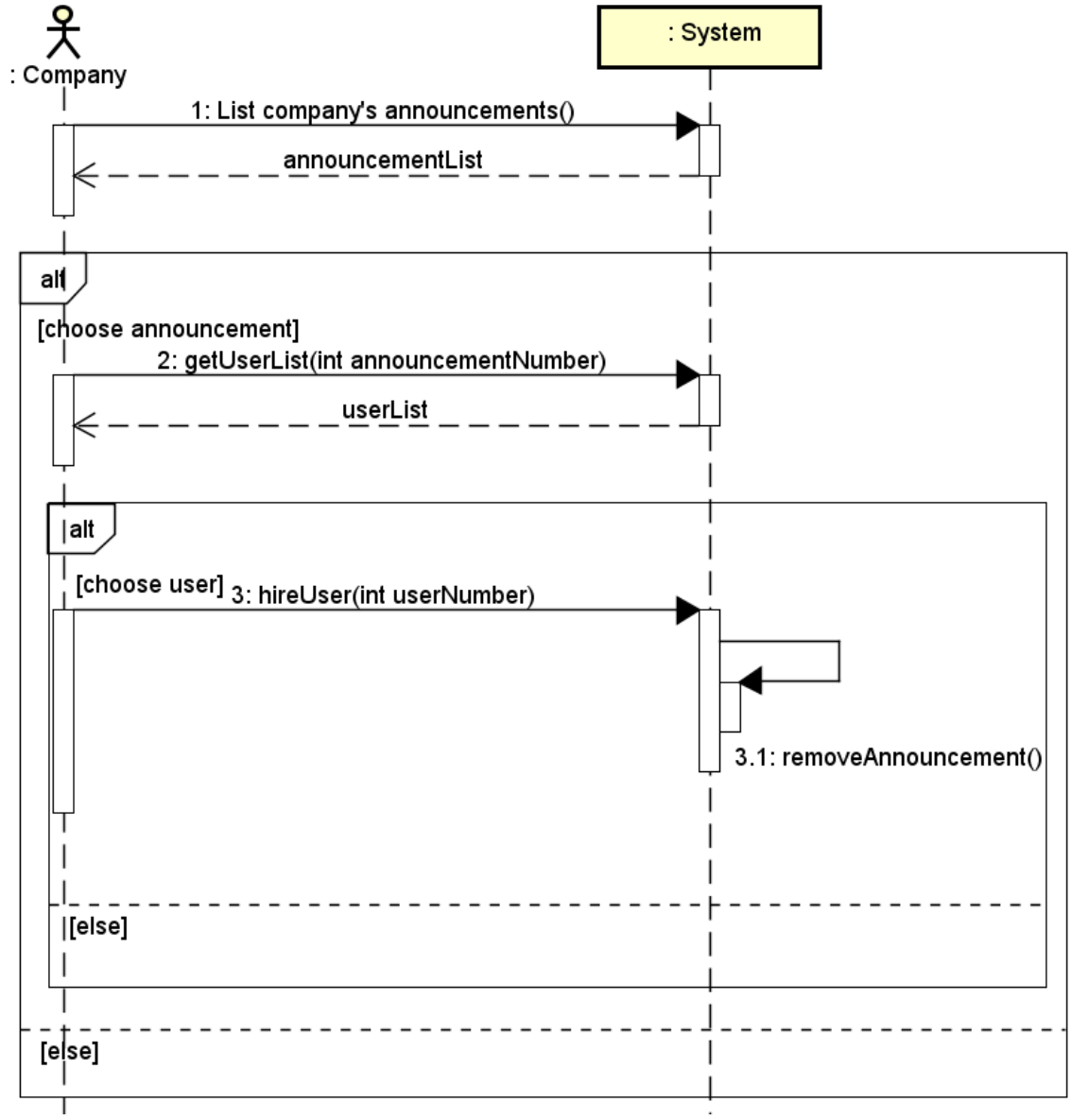


Figure 6 - System Sequence Diagram

As seen from these figures the only complexity that may arise in the system is from the actual communication between the two clients.

IV. System Design

From the initial diagrams the team gradually began work on the more substantial blueprint prior to the actual system implementation, the UML Diagram.

The developers initially came up with this simple diagram to encompass what the system is all about.

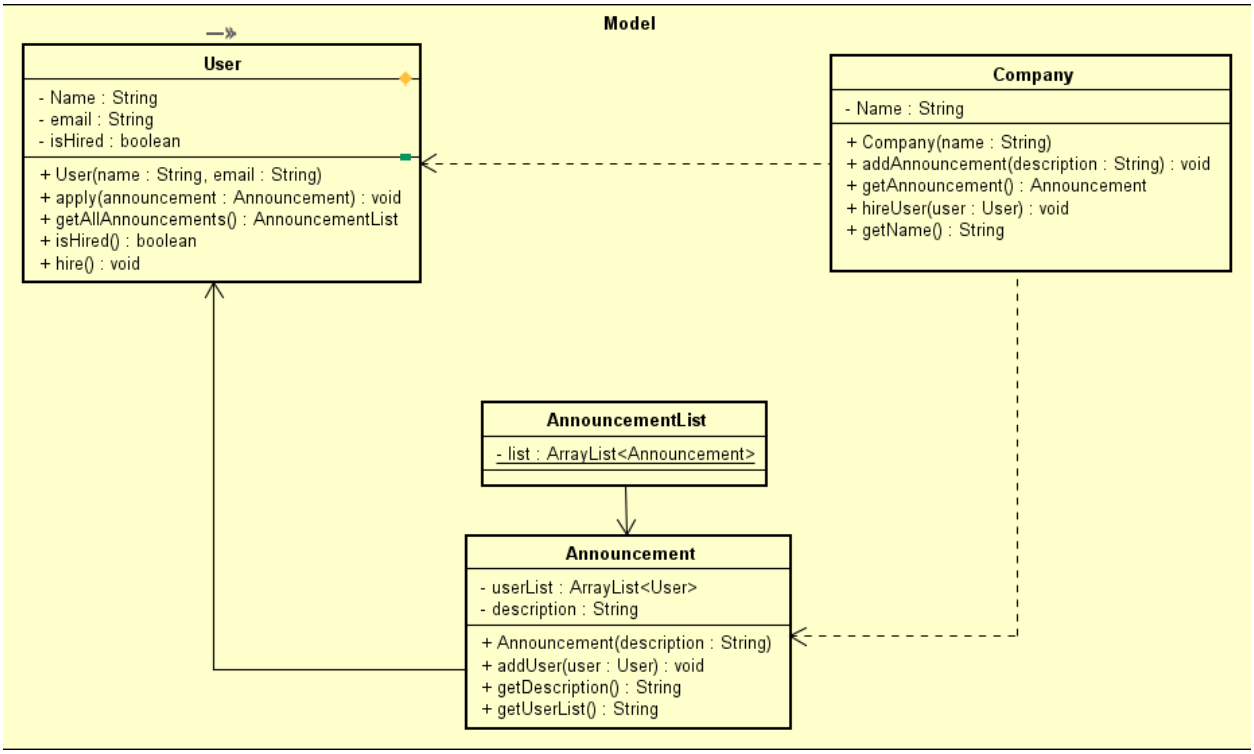


Figure 7 - Initial Model

The system at this stage revolves around four classes, a user, a company, an announcement and a list holding the numerous announcements.

Naturally when time had come to deepen the design of the model, the team gave the original diagram more thought and when it came to the critical decision to keep, modify or discard the model they chose to keep most of it while modifying it slightly to fit the needs of a more complex program.

Once talks about implementing the model above in a client-server system started, the developers knew they would have to have three different versions of their program, each with a separate client. In other words, the user has its own model, the company has its own model and the server also has its own model, for maintenance and testing purposes.

Here are the UML Diagrams of all of those models.

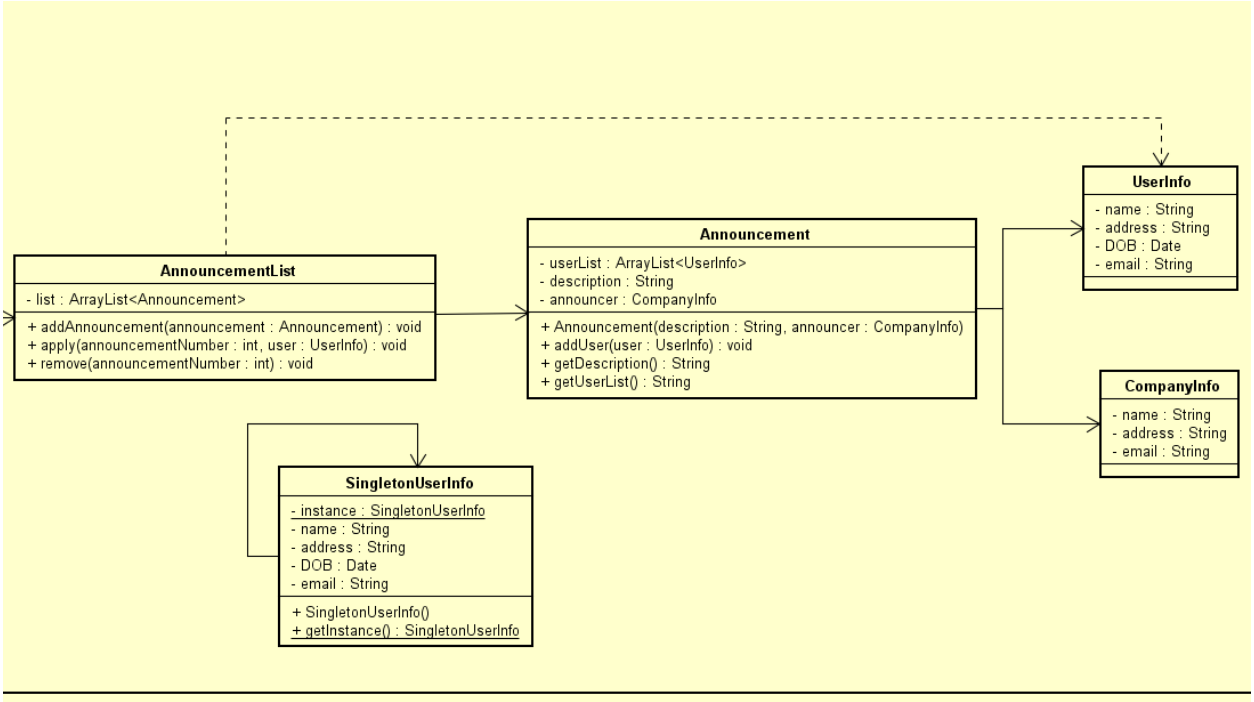


Figure 8 - User Model

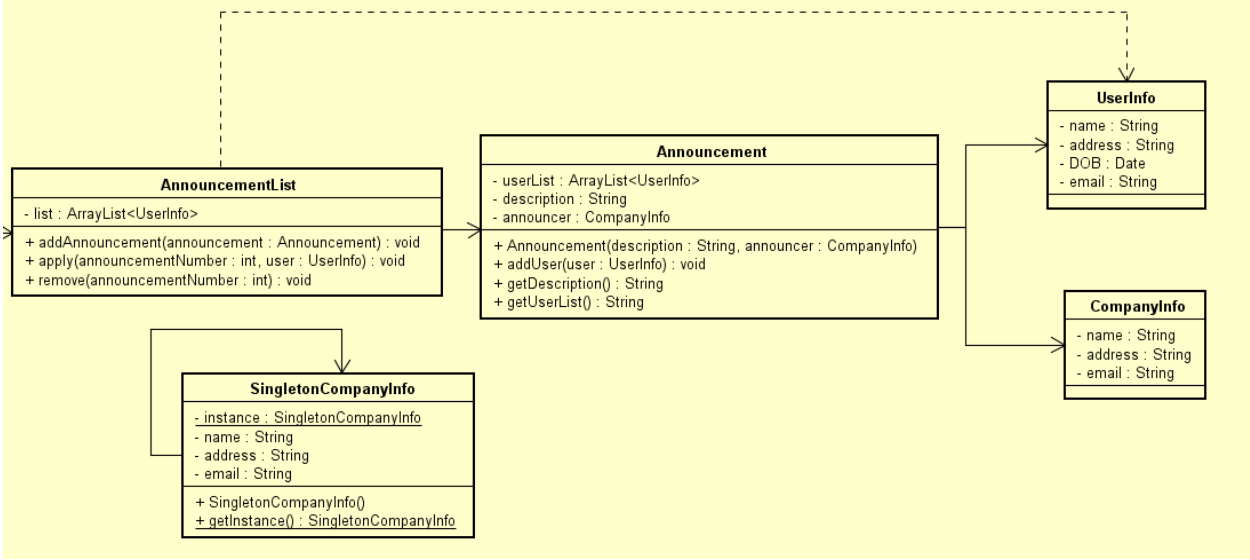


Figure 9 - Company Model

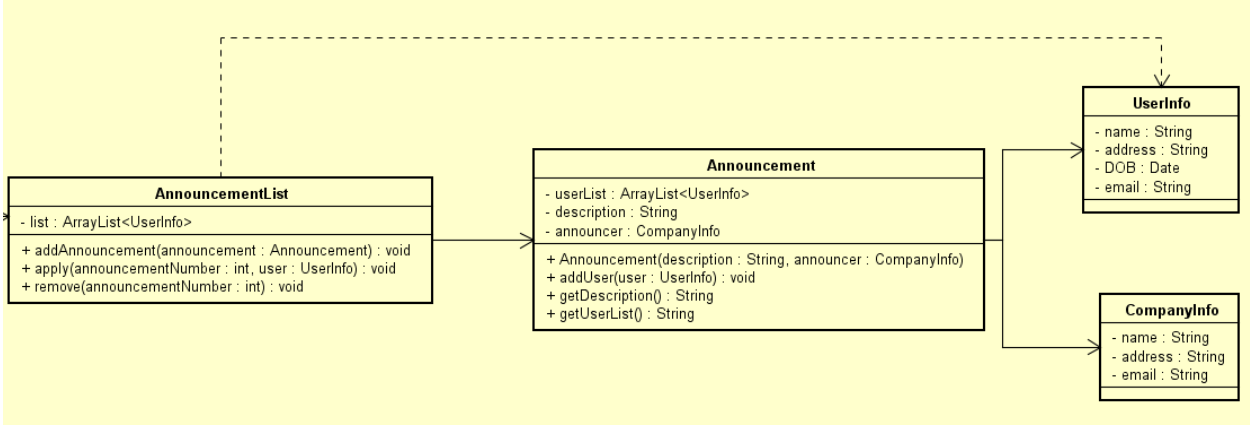


Figure 10 - Server Model

Once the team settled on how the actual model would look like in the end for each separate client then they started talking about the design of the server.

After a few design meetings they settled on adopting a socket-based server rather than RMI due to the fact that it is much more customizable than the latter and it felt more comfortable.

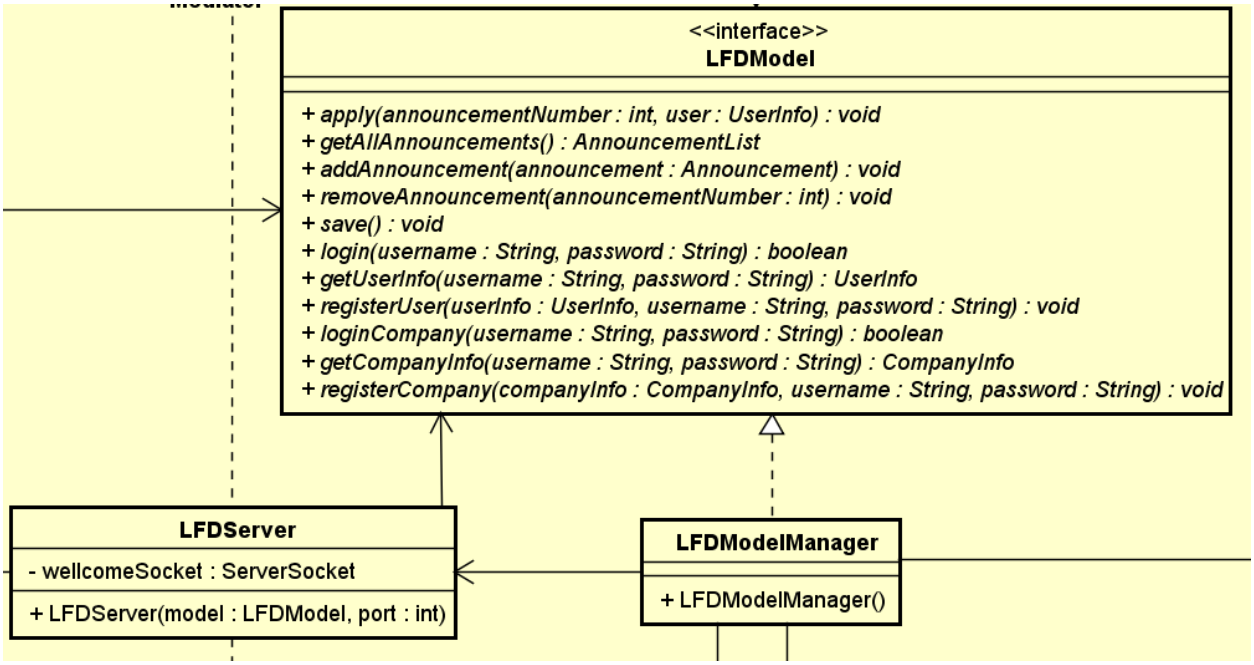


Figure 11 - Server and Model Manager

As you can see in the diagram above, the server is pretty straight forward, the LFDServer class contains the welcoming socket for all the clients connecting to it while the LFDModelManager contains all the methods that the server is able to do and communicate them to the connected clients.

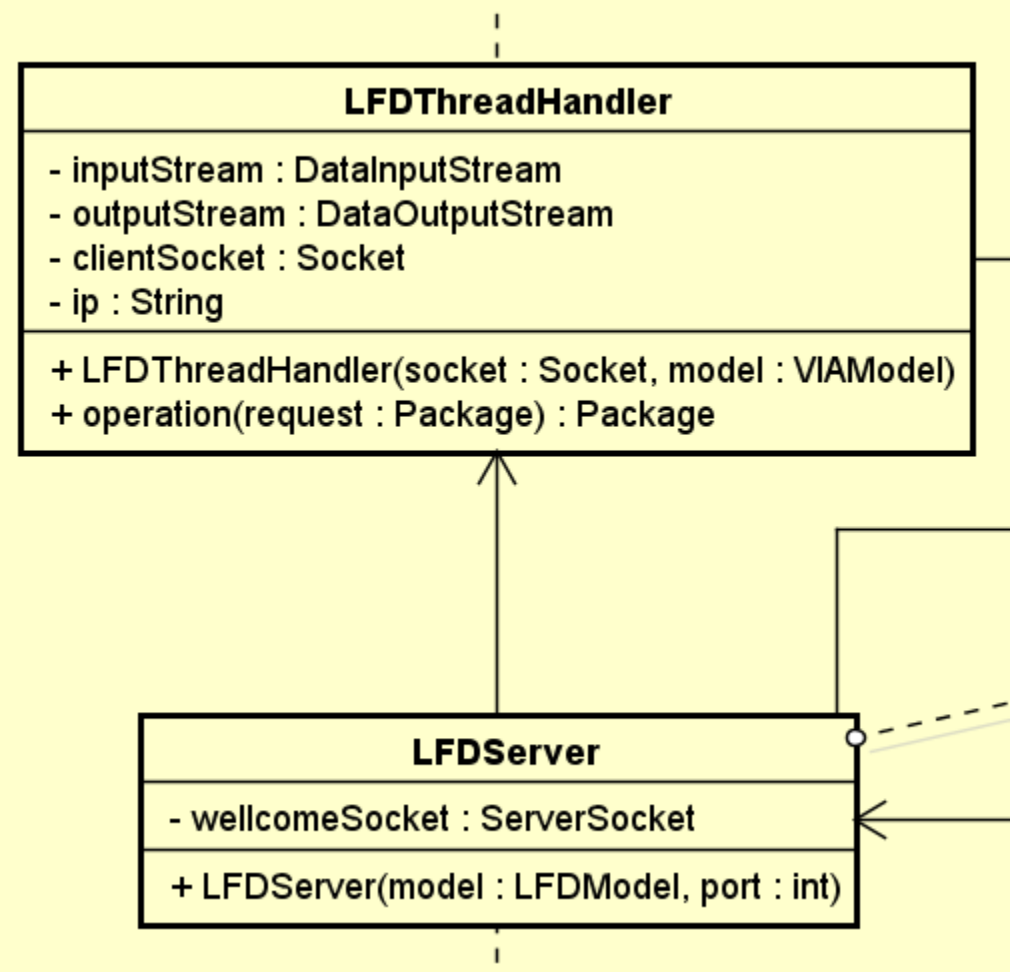


Figure 12 – Thread Handler

Every time a client connects to it, the server opens a new thread for it using the thread handler class.

The server and respective client communicate using custom-made packages. The client registers a command, packs it up using the instructions in the package class, sends it to the server, then the server reads it and responds accordingly, by sending a response back to the user. The user, company and server, each have the same package class that contains all the different types of requests they can have. For example, the user can send packages like “GET” or “APPLY” which

tell the server that it needs to give the list of announcements to the requestee or apply to a specific announcement, given by the requestee in the package.

Last but not least, every ounce of data that the system needs such as keeping track of user and company information or of all announcements and applications of users to them is stored in a SQL Database. Below you can see how the system links up and "communicates" with the database.

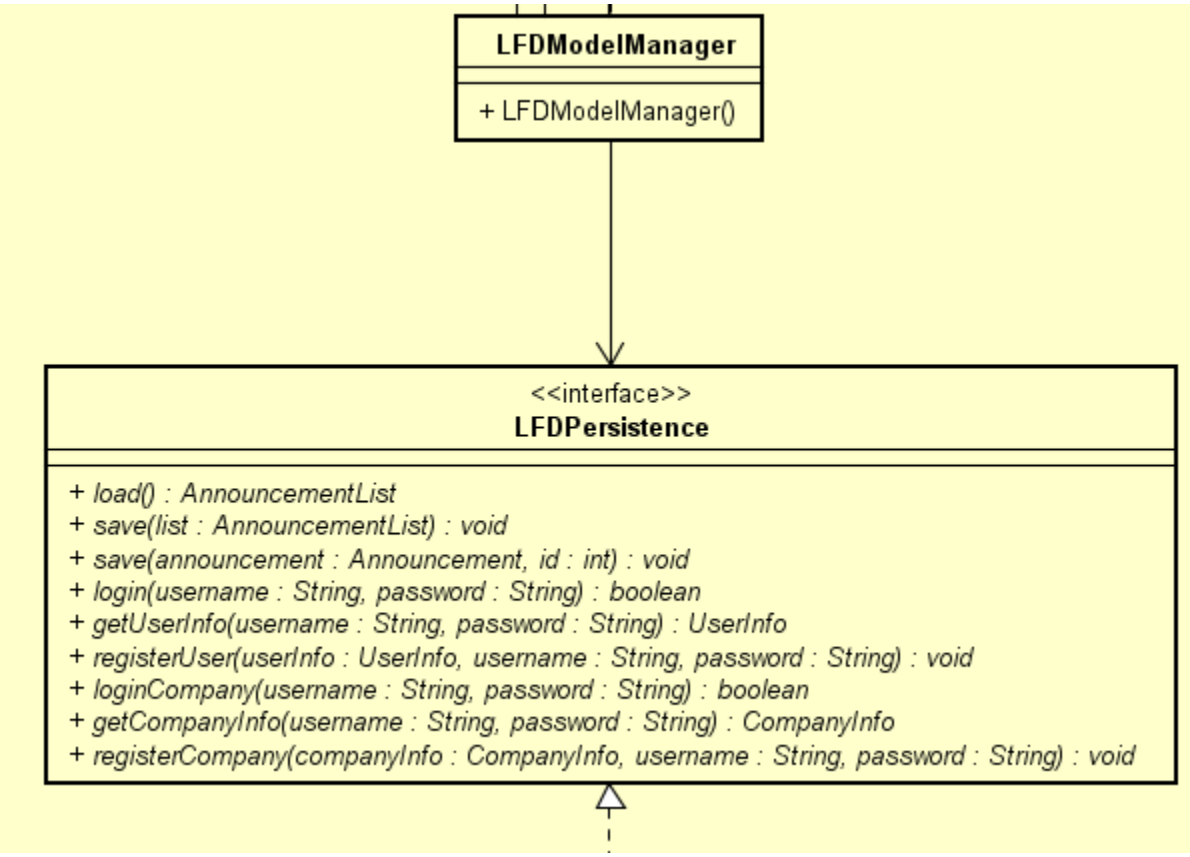


Figure 13 - Database Connection

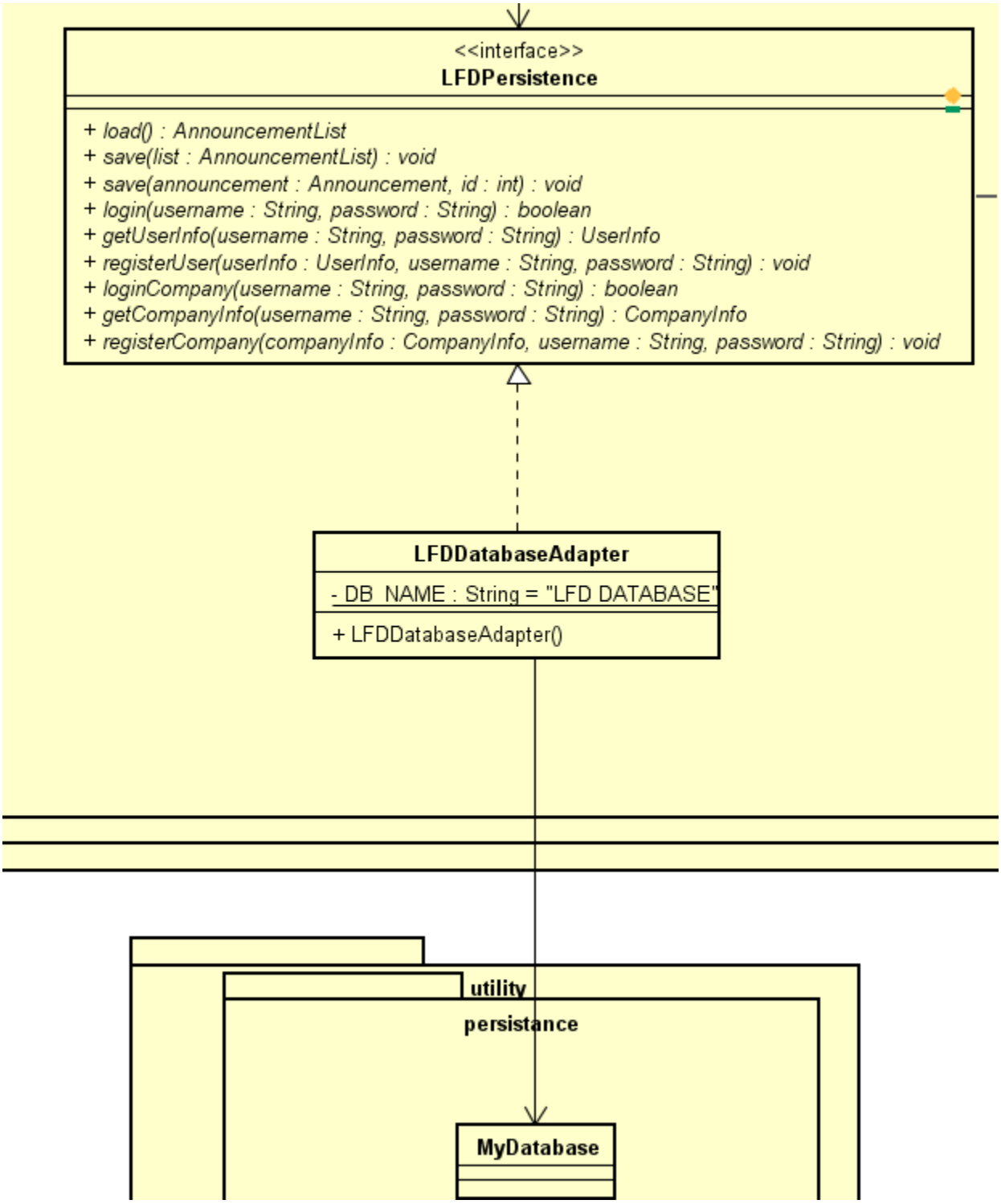
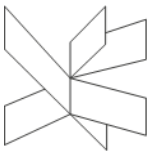


Figure 14 - Database Connection



Every time the program is opened the system loads the data from the server, and every time it closes, the server takes the locally stored data (more information about this in the next part of our document), wipes the database clean and stores the new information in the database thus ensuring constant updates and giving accurate information to both companies and users.

The following section will cover the actual implementation of the aforementioned system.

V. System Implementation

Even with this tiny project the actual code implementation can get out of hand pretty quickly especially when the final goal is a proper client-server system that also incorporates a database.

The team used a few design patterns while developing the system, the biggest one being the MVC (Model View Controller). As the name suggests, its purpose is to clearly split the responsibility in the system and to keep everything nice and tidy inside their own packages. Each class has a clear purpose and a single responsibility, independent of the rest of the system in terms of functionality.

Since this design pattern is present in both client and server, the project report will only detail how it works for the server and mention the differences when it comes to the client side.

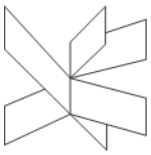
Server

View

The view acts as the forefront of what the user behind the computer will see, it contains an interface with methods for showing text to the screen and getting text from the keyboard. The console implements this interface. The same console also automatically starts the next part of the MVC, the controller. It is also kept in a while loop, always displaying to the user the desired things such as a list of all the commands available while also waiting for input from him.

The interface also acts as an observer in the system, alongside the observable class ModelManager.

Note: The main purpose for the view in the server is to test and control certain things in the system, it closely resembles the views in the user client and the company.



Controller

The controller acts as the bridge between the view and the actual domain of the system. It executes, using the method with the same name, a certain case depending on the input from the while loop inside the view.

The specific executed method is part of an interface in the mediator, it handles all data coming from and to the system.

Model

The model contains all business logic related to the system. It functions using two primary classes, Announcement and AnnouncementList. The announcement class holds all information regarding a specific announcement in the system, it also contains a list of all instances of the class UserInfo (the class that holds the information for a user) that have applied to that announcement. The "announcer" field also holds an instance of the class CompanyInfo (the class that holds the information for a company) which specifies which company actually made the announcement. Finally, the AnnouncementList holds an array list filled with Announcement objects. The same list is also sent to the ModelManager in the Mediator package so that the data can be transferred to other parts of the system.

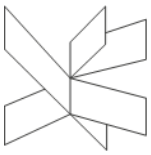
The server model also contains a class called ID, this class automatically generates a unique identifier for each object that the system uses such as an announcement, a user or a company. It also keeps track of the latest object thus ensuring different ids with each added object. These ids are crucial to the database since they also act as primary keys.

Mediator

The mediator acts as a façade for the whole model, it handles the connection between the controller and the model, it handles all data coming to and from the system and is easily the biggest part of the system.

It contains a model interface with all the methods that the server actually has access to, basically everything that someone can do with the server itself such as getting a list of announcements, removing and announcement or adding one, etc.

The model manager implements this interface. It also starts the server which makes use of the thread handler to continuously wait for a connection from any client and then put them on a separate thread. This thread contains vital information about the connected client such as IP and also gives them an input and output stream to be able to communi cate with the server. The server communicates with its clients through the package class. Whenever a client requests information from the server it does so using a custom-made package such as "GET" or "APPLY".



After creating such a package, the client sends it to the server where it is processed and then gets a reply package back with the necessary information.

The model manager serves as an Adapter for the model to the server, it delegates all work coming in from the methods in the interface to the actual server that handles the execution. It is also an observable class, part of the observer design pattern implemented throughout the system.

The model manager also does the integration of the database in the system. The way the team set it up is that the server holds all of the system's information locally, when the server closes naturally through the console it truncates all the data in the database and then saves all the data stored locally into the database. The next time the server is fired up, it loads all the information from the database and re-creates the local data thus ensuring that it is up to date.

The interface implemented by the Database Adapter contains custom made methods such as "Load" or "Save". These methods contain sql strings that are sent to the database in order to retrieve or save information, from or to the database.

The database itself was originally intended to only contain three tables, a table for users, a table for companies and a table for announcements, all of them using the id generated by the server for each one as their primary keys.

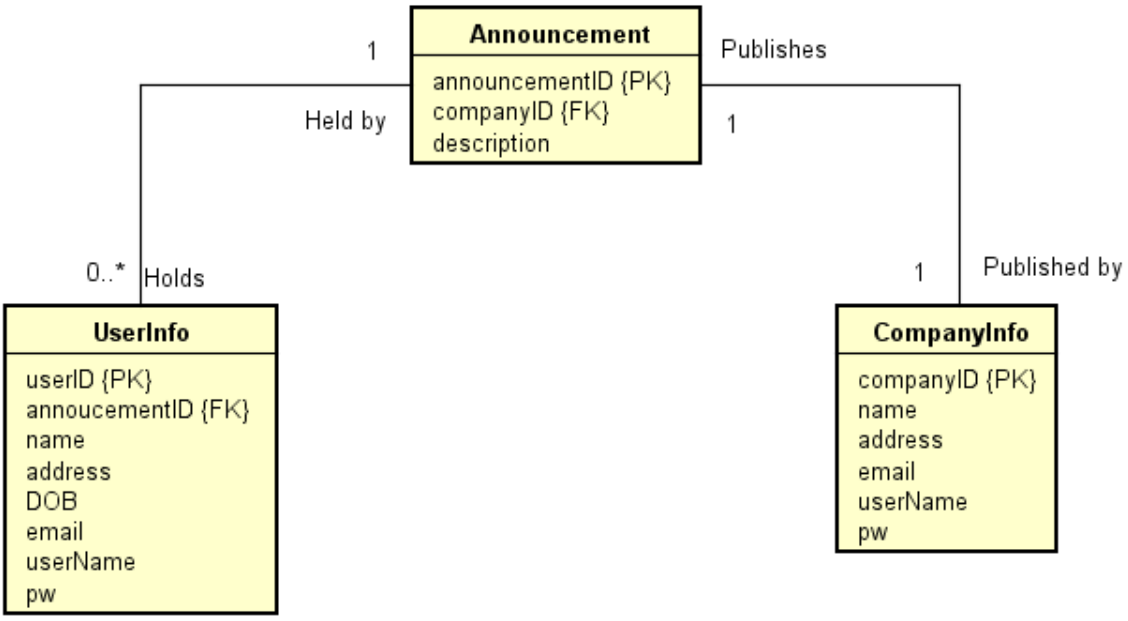


Figure 15 - ER Diagram Old

Over time the team found that there was a problem with this diagram, since the UserInfo table had to keep track of all announcements it applied to there was no way for it to hold more than one seeing as it uses announcementID as a foreign key, so a new diagram was needed, one with an extra table called “applications” that handled the link between users and announcements.

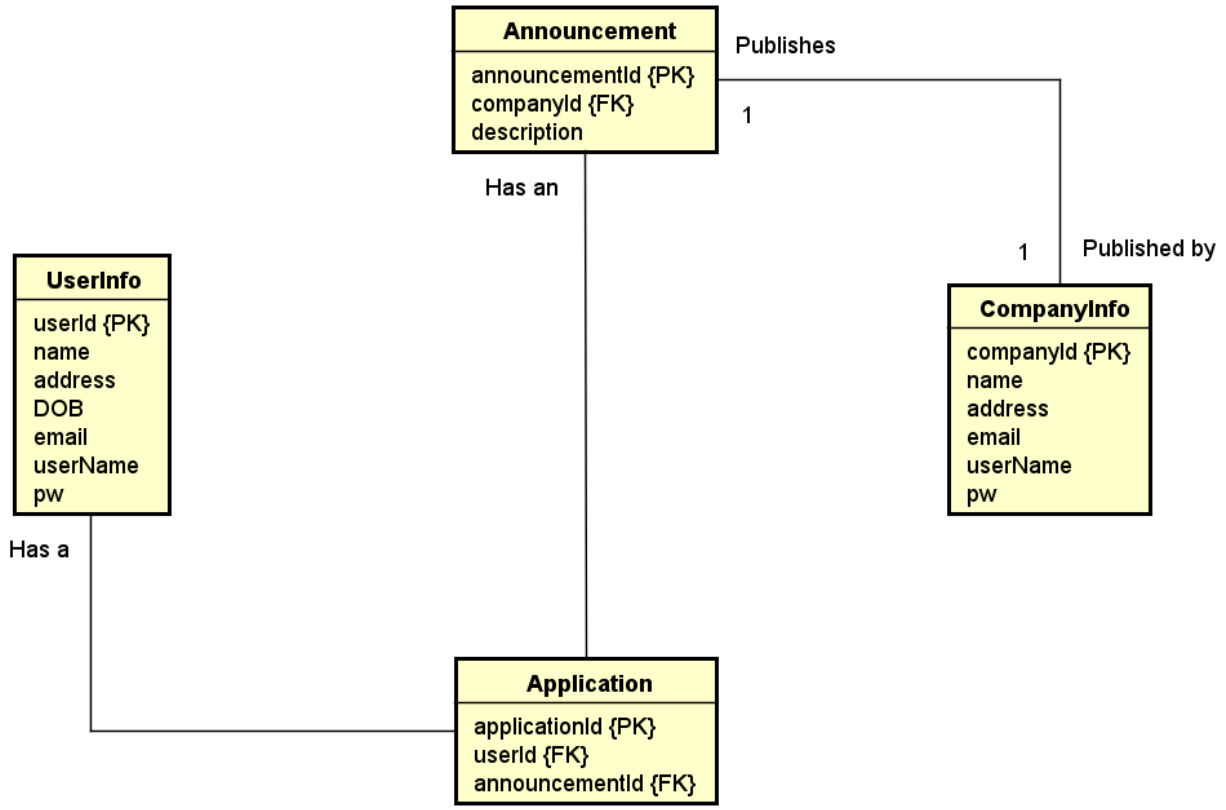
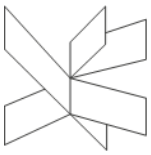


Figure 16 - ER Diagram Current

Database business rules

The design of the database is specific to the needs of the system so the data stored in the domain model can be uploaded to the database and then retrieved without any loss. The database is comprised of 4 different tables: UserInfo, Announcement, CompanyInfo and Applications.

1. UserInfo Table
   - Responsible for storing the data of a UserInfo type object from the domain model (name, address, date of birth, email)
   - Stores the login credentials of every user (username and password)
2. CompanyInfo Table
   - Responsible for storing the data of CompanyInfo type object from the domain model (name, address, email)
   - Stores the login credentials of every company (username and password)



3. Announcement and Appplications Tables
    - Responsible for storing the data of an Announcement type object from the domain model (description, announcer, user list)

Note: In order to keep track of which users were stored in every announcement's user list an Applications table was designed. An application consists of a user's and a company's ID, this way there is a reference to which users have applied to a specific announcement.

Client

The client side also implements the MVC design pattern. The view and controller for both users and companies are essentially the same as in the server. The only dramatic changes are seen in the mediator package.

In the Model Manager, alongside the absence of a database, the observer pattern is no longer used. Instead of having all the work from the model interface delegated to a server, it is given to a client which specializes in establishing a connection to the specified ip address and port.

The client side also requires a special Receiver class. Since it is in constant communication with the server, it always receives replies in Json format, the receiver simply converts these into a readable package format. The same Package class as in the server is used to read those packages.

As you can see the client also misses a thread handler class since it does not have the same responsibilities as the server.

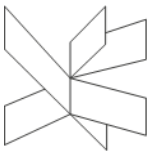
VI. Testing

Unit Testing

The unit testing in Junit was done in order to test specific parts from the model of the system. The purpose of it is to see if everything in the model is working as expected.

Before the testing

The unit test will instantiate each object with the following data before each test. The unit test class has an attribute for each shown instance.



```
@Before
public void setUp() throws Exception
{
   dateSut = new Date(10, 10, 2018);
   userInfo = new UserInfo("Poul", "StreetStreet", this.dateSut, "Email");
   companyInfo = new CompanyInfo("IT-service", "StreetStreet", "Email");
   announcement = new Announcement("Hallo", companyInfo);
   announcementList = new AnnouncementList();
}
```

Figure 17 - setUp

Test for getters

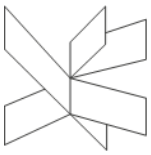
All the getters from both the UserInfo and CompanyInfo class were tested as seen below. In order to verify that both the constructors and get methods work properly a number of tests were run for every class in the model.

```
// Test for UserInfo
@Test
public void TestUserGetInfo()
{
   boolean assertion = (userInfo.getName().equals("Poul")
         && userInfo.getAddress().equals("StreetStreet")
         && userInfo.getDate().toString().equals(dateSut.toString())
         && userInfo.getEmail().equals("Email"));
   assertTrue(assertion);
}
```

Figure 18 - Get Test

Test of singleton.

The singletons from the model of both the User and the Company had to be tested in order to make sure that they work as expected. The following test will ensure that there can only be one instance of the object by calling the getInstance method and storing what is returned in two different variables.



```
// Test for SingletonUser
@Test
public void TestIfSingletonIsWorking()
{

   SingletonUserInfo singleOne = SingletonUserInfo.getInstance("Ib",
         "StreetStreet", "email.com", dateSut);
   SingletonUserInfo singleTwo = SingletonUserInfo.getInstance("ole",
         "StreetName", "email.dk", dateSut);

   assertTrue(singleOne.equals(singleTwo));

}
```

Figure 19 - Singleton Test

Testing of AnnouncementList

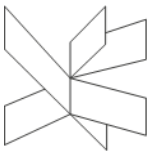
Because the AnnouncementList class contains an ArrayList the methods responsible for adding data to it were tested to verify if they work properly.

```
@Test
public void testOfAnnouncementListNotEmpty()
{

   announcementList.addAnnouncements(announcement);
   announcementList.addAnnouncements(announcement);
   assertTrue(!(isEmpty(announcementList.getAnnouncement(0))));

}
```

Figure 20 - Announcement List Test



The following test assumes that the ArrayList is empty and tests wether it was instantiated properly or not.

```
@Test
public void testOfAnnouncementListEmpty()
{
   try
   {
      assertTrue(isEmpty(announcementList.getAnnouncement(0)));
   }
   catch (IndexOutOfBoundsException e)
   {

   }
}
```

Figure 21 - Announcement List Test Empty

In order to further testing on this class the size of the list was verified after adding a couple of elements.

```
@Test
public void testSizeOfArrayList()
{
   announcementList.addAnnouncements(announcement);
   announcementList.addAnnouncements(announcement);
   assertTrue(announcementList.size() == 2);

}
```

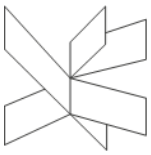
Figure 22 - Announcement List Test Size

The apply method in the announcement list is an important part of the system; this being one of the main ways the client interacts with the system so the following test was made in order to check if it works as intended.

```
@Test
public void testApplyToAnnouncement()
{
   announcementList.addAnnouncements(announcement);
   announcementList.apply(0, userInfo);
   assertTrue(
         !(announcementList.getAnnouncement(0).getUserList().equals("")));

}
```

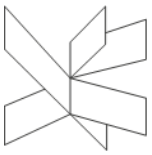
Figure 23 - Apply To Announcement Test



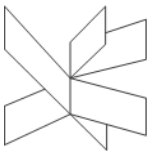
Test Cases

In order to test the system in a broader manner a set of test cases were devised. Each test case takes place in a unique scenario within the system and after going through those scenarios and comparing the end result with the expected one, the functionality the main features of the system were tested.

Test Case	Description	Actor	Precondition	Expected Result	Steps	Result
1	Apply to an announcement	User	Logged in and there are announcements in the system	User's information is added to the announcements list of applied users.	1. The user enters 2 to choose "Apply to announcement" 2. The user chooses an announcement to apply to	User's information is added to the announcements list of applied users.
2	List all announcements	User	Logged in and there are announcements in the system	The system displays all announcements stored.	The user enters 1 to choose "List all announcements"	The system displays all announcements stored.
3	Adding a new announcement	Company	Logged in	An announcement that has a description, the company's information and an empty user list is created.	1. The company presses 1 to choose "Add announcement" 2. The company enters a description for the announcement	An announcement that has a description, the company's information and an empty user list is created.
4	Removing an announcement	Company	Logged in and there are announcements in the system	The system removes the announcement from the list.	1. The company presses 2 to select "Remove announcement"	The system removes the announcement from the list.



					2. The company enters the number of the announcement they wish to remove	
5	List the company's announcements	Company	Logged in and there are announcements in the system	A list with the announcements published by the company is displayed.	1. The company presses 3 to choose "List my announcements"	A list with the announcements published by the company is displayed.
6	List an announcement's list of applied users	Company	Logged in, there are announcements in the system and the company chose "List my announcements" and did not press 0 afterwards	A list with the information of all users that applied to that announcement is displayed.	1. The company enters the number of an announcement	A list with the information of all users that applied to that announcement is displayed.
7	Hire a user that applied to an announcement	Company	Logged in, there are announcements in the system, the company chose "List my announcements", then chose an announcement and did not press 0 afterwards	The chosen announcement is removed from the system.	1. The company enters the number of a user	The chosen announcement is removed from the system.
8	List all announcements but there are no announcements stored	User	Logged in and there are no announcements published	An empty list is displayed and the program returns to the main menu.	1. The user enters 1 to choose "List all announcements"	An empty list is displayed.



9	Apply to an announcement but there are no announcements stored	User	Logged in and there are no announcements published	The program returns to the main menu.	1. The user presses 2 to choose “Apply to announcement ” 2. The user enters an announcement’s number	A package is sent to the server but there is no reply. The client keeps waiting for a reply.
10	List the company’s announcements but there are no announcements stored	Company	Logged in and there are no announcements published by this company	An empty list is displayed and the program returns to the main menu.	1. The company presses 3 to choose “List my announcements ”	An empty list is displayed and the program returns to the main menu.
11	List the user list of an inexistent announcement	Company	Logged in, the company chose “List my announcements ” and did not press 0 afterwards	A message is displayed to the company and the company return to the main menu.	1. The company enters a number	A message is displayed to the company and the company return to the main menu.
12	Remove an inexistent announcement	Company	Logged in	The program returns to the main menu.	1. The company presses 2 to select “Remove announcement ” 2. The company enters a number for an inexistent announcement	A package is sent to the server but there is no reply. The client keeps waiting for a reply.

Sequence diagram

In order to test that the data flows through the system properly a sequence diagram that goes through the MVC design pattern and the connection to the server was made.

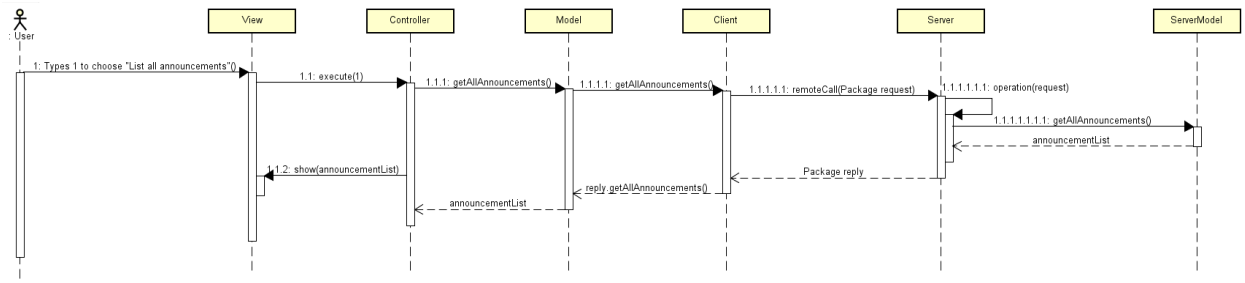


Figure 24 - Sequence Diagram

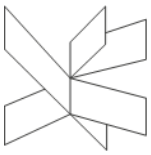
VII. Results and Discussion

The developers' feel the results are appropriate for what they first set out to create. The project has been satisfactory towards the Product Owner even if most of the quality of life parts of the system were not implemented. The system functions at a base level with no added "flavor", with the amount of work put in from all developers the team can safely say that they expected to have to cut out parts of the planned implementation that seemed, at least at the start, doable, in order to finish the system in the required timeframe.

VIII. Conclusion

All functional requirements and non-functional requirements have been met and as it stands the system is working flawlessly and achieves everything the Product Owner needs it to do. Everything that has been implemented has been done in such a manner as to ensure maximum maintainability throughout the system's life.

Overall the project has yielded powerful results for both product owner and developer. The stakeholder received a stable and robust system accompanied by a thorough database while the developer received tremendous experience in how to tackle a client-server system and especially how much work there is even during the smallest projects.



IX. Project Future

From a technical viewpoint there are quite a few improvements that the developers could work on.

First of all, better security when it comes to saving the data to the database. Having it being saved only when the server is closed naturally through the console is a very big red flag when it comes to stability as any power outage or a simple click on the red x is going to mess up the whole system for both users and companies.

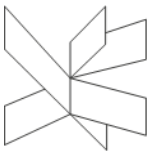
Second, there are a lot of unimplemented parts of the system, they aren't part of the actual requirements since the system can work without them but they offer incredible quality of life improvements such as being able to edit your own data without having to register a second time as a user or a company. Being able to see the profile of the company that put up the announcement or even seeing the profile of the user that applied to a specific announcement.

Lastly, when the developers first started out they thought about implementing a chat system between the clients of the server but as time went by it proved too difficult and was scrapped. A chat system would definitely be a planned feature for the future.

X. List of Appendices

Note: Clicky, clicky…

- Project Description
- User Guide
- Source Code
- Analysis Diagrams
- Design Diagrams
- Scrum



XI. Sources of Information

- Ben Matthews, 2017. Freelance Statistics: The Freelance Economy in Numbers. *Freelance* [e-journal] 1(14), pp.1-1. Available through: Ben Matthews blog <https://benrmatthews.com/freelance-statistics/> [Accessed 7 March 2018]
- Upwork, 2017. *Freelancers predicted to become the U.S. workforce majority within a decade, with nearly 50% of millennial workers already freelancing, annual "Freelancing in America" study finds.* [online] Available at: <https://www.upwork.com/press/2017/10/17/freelancing-in-america-2017/> [Accessed 7 March 2018]
- Upwork, 2018. *Upwork releases Q4 2017 Skills Index, ranking the 20 fastest-growing skills for freelancers.* [online] Available at: < https://www.upwork.com/press/2018/02/07/q4-2017-skills-index/> [Accessed 5 March 2018]
- Lahle Wolfe, 2017. 1 in 3 Americans are now independently employed. *Women in Business* [e-journal] 1(10), pp.1-1. Available through: The Balance website <https://www.thebalance.com/freelance-worker-statistics-3514732> [Accessed 27 February 2018]
- The Danish Ministry of Culture, 2014. *Consolidated Act on Copyright 2014.* [pdf] Available at: https://kum.dk/fileadmin/KUM/Documents/English%20website/Copyright/Act_on_Copyright_2014_Lovbekendtgoerelse_nr._1144_ophavsretsloven_2014_engelsk.pdf [Accessed 2 March 2018]
- Admin, 2015. The Importance of User Experience (UX) Design in Software Development. *Whitepapers* [e-journal] 1(6), pp.1-1. Available through: Codium website <https://www.codium.com.au/the-importance-of-user-experience-ux-design-in-software-development/> [Accessed 20 February 2018]
- Connoly, T. M. and Begg, C. (2009) *Database systems: A Practical Approach to Design, Implementation, and Management.* 5th ed., USA: Addison-Wesley Publishing Company
- Larman, C. (2004) *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.* 3rd ed., Upper Saddle River, NJ, USA: Prentice Hall PTR