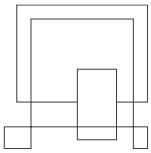


Project Report

Library System

Client-Server

Names of Students:

Jan Vasilcenko – 293098

Karrtiigehyen Veerappa - 293076

Nicolas Popal - 279190

Patrik Horny – 293112

Tomas Soucek – 293103

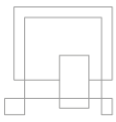
Supervisors:

Henrik Kronborg Pedersen

Joseph Chukwudi Okika

VIA University College

Bring ideas to life
VIA University College



Number of characters: 68454 including spaces

Software Technology Engineering

Semester 2

3 June 2020

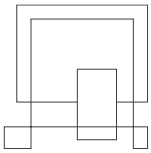
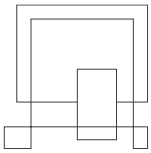
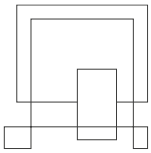


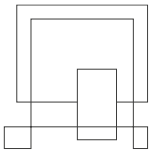
Table of Contents





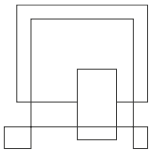
List of Diagrams

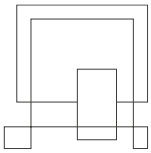
List of Tables



List of Figures

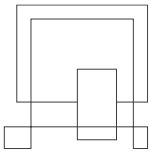
List of Code





Abstract

Libraries are important places where free knowledge can be obtained, and software are made where users can borrow and reserve items with ease. This project is concerning the production of a library system using client-server architecture, where users can borrow and reserve items using the library system. This paper shows how a client-server system is constructed using design patterns and SOLID principles, and how to build a database, which can be connected to the client-server system. Different types of testing are used for functional and non-functional requirements, where the results are discussed on how it can be fixed to improve the quality of the system.



1 Introduction (Nicolas Popal, Karrtiigehyen, Patrik Horny)

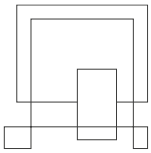
1.1 Background Description (Nicolas Popal, Karrtiigehyen)

Nowadays, while many information can be found on the internet, it is still might not be a reliable source of information compared to those obtained from libraries. This is mostly caused by misinformed information on the internet since anyone with access to internet can publish a website. Also, libraries are very important source of information since many publications are archived and can be accessed for free by people at any time. But manually searching for an item in a library can lead to some confusion and reserving borrowed items can become cumbersome when a lot of people are in a queue for the item.

The availability of knowledge is more important when the library is of a school, for example at VIA. But some students might be discouraged from using the schools' libraries for a myriad of reasons. From an informal interview conducted by the group with a librarian from VIA library on Campus Horsens, it was found that a lot of students do not take advantage of the library as much as the librarians hoped. There are many reasons to this, chief among which is the difficulty of navigating the program where students can borrow or reserve library items. So, a better system is called for.

This project is concerning the development of a system for a library, where customers (users) can borrow, reserve, and check for books. It should be easy for customers to register and log in to the library system, so that the customers would use their CPR number as an account ID, since in Denmark every citizen has their unique CPR number. This makes the CPR perfect for distinguishing between customers.

The library system should be able to store all information about users, but also all information about library items, and all reserved and borrowed items from the library. Customers should have access to all information about items in the library and information about their own account (as a borrowed item, reservations etc.) but they should not be able to access information about other customers.



Importance of library system is to improve reachability of library items for customers. For libraries without a system is hard to find missing borrowed books from customers and this system should help them keep this information updated. It is important also for customers to have information about library items, which they borrowed, so that they can be updated about upcoming deadlines for returning books. Also, the customers can see information about reservations, so that they can check when the library item will be available.

1.2 Definition of Purpose (Patrik Horny)

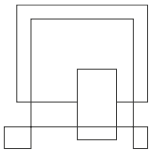
The purpose is to help library's customers to borrow and reserve items from the library so that the customers can easily manage their borrowed items and reservations.

1.3 Delimitations (Karrtiigehyen)

There are some delimitations set to define the scope of the project.

1. There will only be one type of user, which is the customer who will create an account, borrow, and reserve items.
2. Items cannot be added or removed through the system.
3. The contents of items (item's descriptions) cannot be edited through the system.
4. Notifications will not be sent to customers via email, SMS, and etc.
5. The time for keeping a borrowed item cannot be extended by the customer.

The methodology used, time schedule, and risk assessment can be found in the Project Description, which can be found in Appendix A.



2 **Analysis** (Everybody)

In the analysis, the problem domain will be further defined by elaborating upon the contents from the previous section. Firstly, the requirements will be explored in the shape of user stories. Then, use case diagram and descriptions are presented to show the different actors and scenarios involved, from which System Sequence Diagrams can be erected. The conclusion of this section will be with the presentation of the domain model of the system, which will be used as the groundwork for the 3rd section, Design.

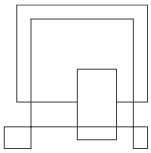
2.1 **Requirements** (Everybody)

SMART principles were used as the framework for the requirements, so that the requirements are testable.

2.1.1 **Functional Requirements** (Everybody)

Critical priority

1. As a customer, I want to borrow item, so that I can use them.
2. As a customer, I want to reserve item, so that I can borrow them later.
3. As a customer, I want to view items, so that I can borrow them.
4. As a visitor, I want to register myself as a customer, so that I can use the library system.
5. As a customer, I want to be able to log in as a customer, so that I can access the library system.
6. As a customer, I want to be able to see the due date for my borrowed items, so that I can return them in time.
7. As a customer, I want to be able to return items which I have borrowed, so that other customers can borrow them.



High priority

8. As a customer, I want to be in a queue for reserved items and check my queue position, so that I can borrow them later.
9. As a customer, I want to be alerted when the due date for borrowed items is near, so that I will remember to return the borrowed items.
10. As a customer, I want to search for specific items by their category (book, movie, or videogame) and title, so that I can view and/or borrow them.
11. As a customer, I want to view the details of items such as genre and year of publishing, so that I know the details of the item I am about to borrow.
12. As a customer, I want borrowed items to be automatically returned when the due date is past, so that other customers who have reserved the item can borrow it.

Low priority

13. As a customer, I want to edit my user info, so that it is updated.
14. As a customer, I want to be alerted when I am entering a queue when I am reserving an already borrowed item, so that I can decide whether to enter the queue or not.
15. As a customer, I want to view my name when I log in, so that I can identify if I have logged into my account.

2.1.2 **Non-Functional Requirements** (Everybody)

1. An intuitive and navigable user interface.
2. The library system should work on separate computer over the same Local Area Network (LAN).
3. Customers must be authenticated by their CPR.
4. Customers cannot modify the information about the items in the library, only view and borrow them.
5. The library should have three types of items: Book, Movie, and Video Game.
6. Software will be connected to the secured database.

7. System should accurately validate users, item reservation and item search.

2.2 **Use Case Diagram** (Everybody)

To get a better understanding of the problem, an use case diagram is developed.

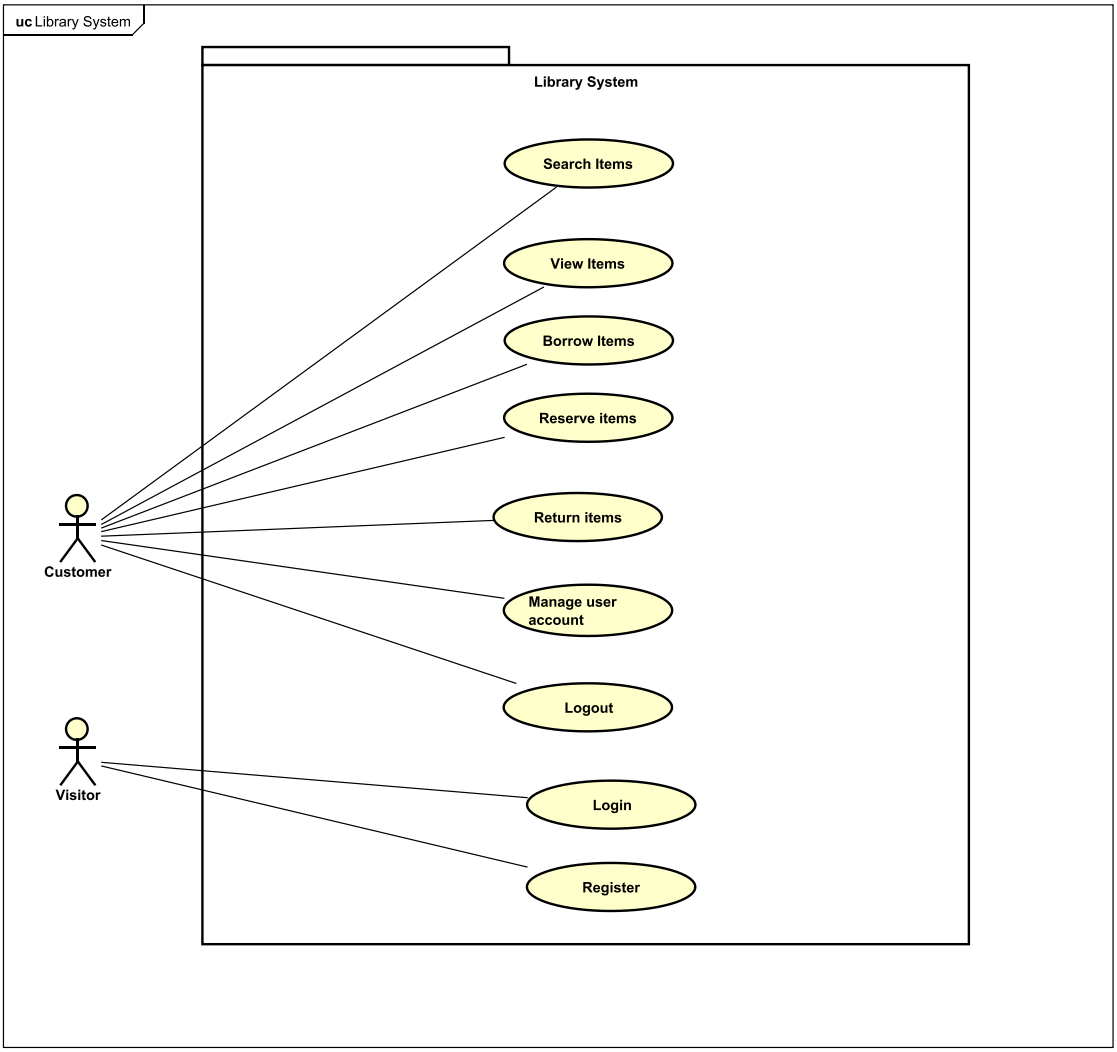
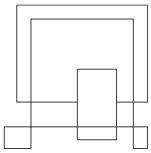


Diagram 1: Use case diagram of the library system

As seen from Diagram 1, there are 2 actors – visitor and customer. For first use all actors are visitors, until they will use the only option, which is available for them – sign up. After registering and logging in, visitor becomes a customer and can use other options as well.



When customer is signed up login option must be used to verify this customer. As a customer other options can be used as view items in the library and search in them, borrow items or if they are not available, then reserve them. Every customer has also option to change its own credentials in manage user account.

2.3 Use Case Descriptions (Everybody)

The use case descriptions show how each scenario will play out. The use case descriptions are based on the use case diagram that was just shown.

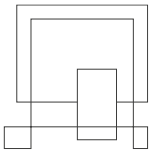
The use case descriptions below are for Borrow items, Reserve items, and Return items. These three cases will serve as the main example throughout this report, and will be referred to frequently. The rest of the use case descriptions can be found in Appendix B.

Borrow items

Use case	Borrow items
Summary	The customer borrows items from the library
Actor	Customer
Precondition	Customer has an account and is logged in. The selected item is not currently borrowed by any other customers
Postcondition	The item will be borrowed by the customer
Main scenarios	1. The customer chooses an item to borrow 2. The system checks the item's availability 3. The item is borrowed by the customer and system has been updated
Alternative scenarios	*a At any time during step 1 Customer cancels 1. The use case ends 2a The item is already reserved 1. System informs customer, that item is already borrowed

Table 1: Use case description for Borrow item

Reserve item



Use case	Reserve items
Summary	The customer reserve items from the library
Actor	Customer
Precondition	Customer has an account and is logged in. The selected item is already borrowed by other customer
Postcondition	The customer was added to the queue for item reserve
Main scenarios	1. The customer chooses book to reserve 2. The system checks the item's availability 3. The item is reserved by the customer and the customer is added to the queue for the item. And the system has been updated
Alternative scenarios	*a At any time during step 1 Customer cancels 1. The use case ends 3a The Customer cancels The use case ends

Table 2: Use case description for Reserve item

Return items

Use case	Return items
Summary	The customer can return borrowed items
Actor	Customer
Precondition	The customer has borrowed items
Postcondition	The due date of the item has been removed and the item is available for reservation
Main scenarios	1. The customer selects the item to return 2. The system checks for due date 3. The item is removed from the customer's borrowed list
Alternative scenarios	*a At any time during step 1 Customer cancels 1. The use case ends 2a The item is past its due date 1. The item is automatically returned

Table 3: Use case description for Return items

2.4 System Sequence Diagram (Nicolas Popal)

The examples shown here are for the System Sequence Diagrams of borrowing and reserving of items, and returning items. All the system sequence diagrams can be found in Appendix C.

System sequence diagram (SSD) is based on use case description. In SSD we are treating system as a black box that means that we are not showing what happens inside of the system. For an example, the system sequence diagrams are used for returning items and reserving/borrowing items from the library system to show the input and output events to the system.

Borrowing and reserving items

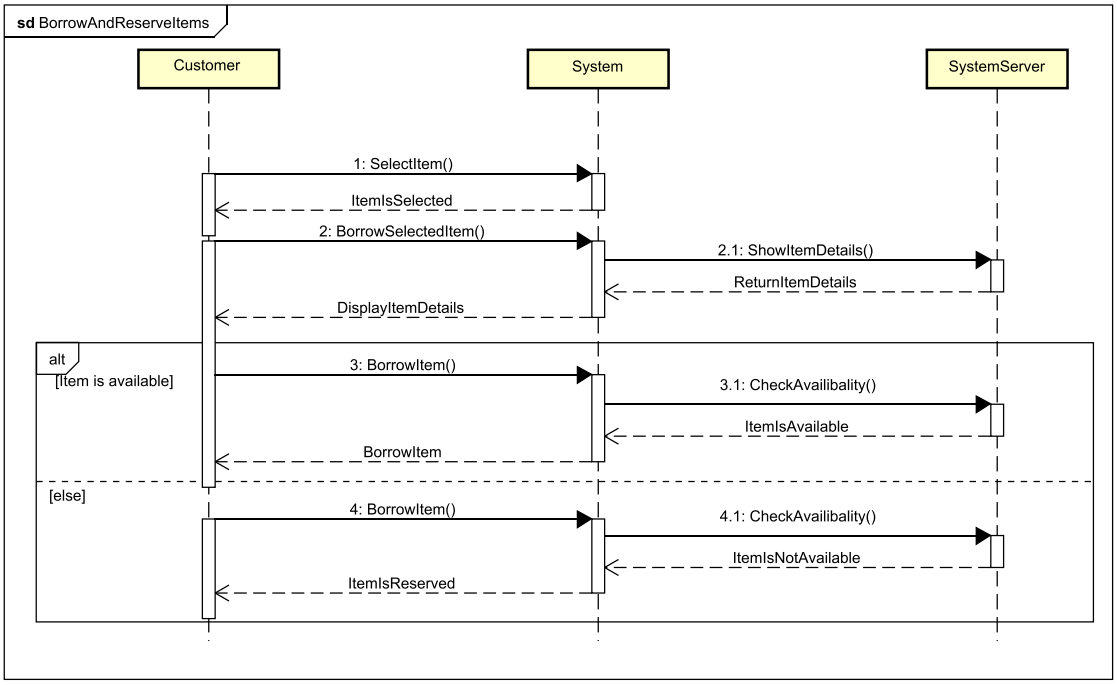
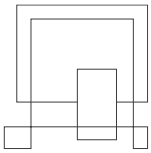


Diagram 2: System sequence diagram for borrowing and reserving items

As you seen in the above diagram, there are 3 actors – customer, system, and system server.



Customer is choosing which item wants to borrow or reserve and then system is contacting system server and returning item details back to the customer.

In this SSD, an if statement with condition of item availability is used. Customer tries to borrow the item; system is contacting system server and system server is checking availability of this item. If the item is available system borrows books for the customer and if not, then customer is assigned to the queue for this item.

Returning items

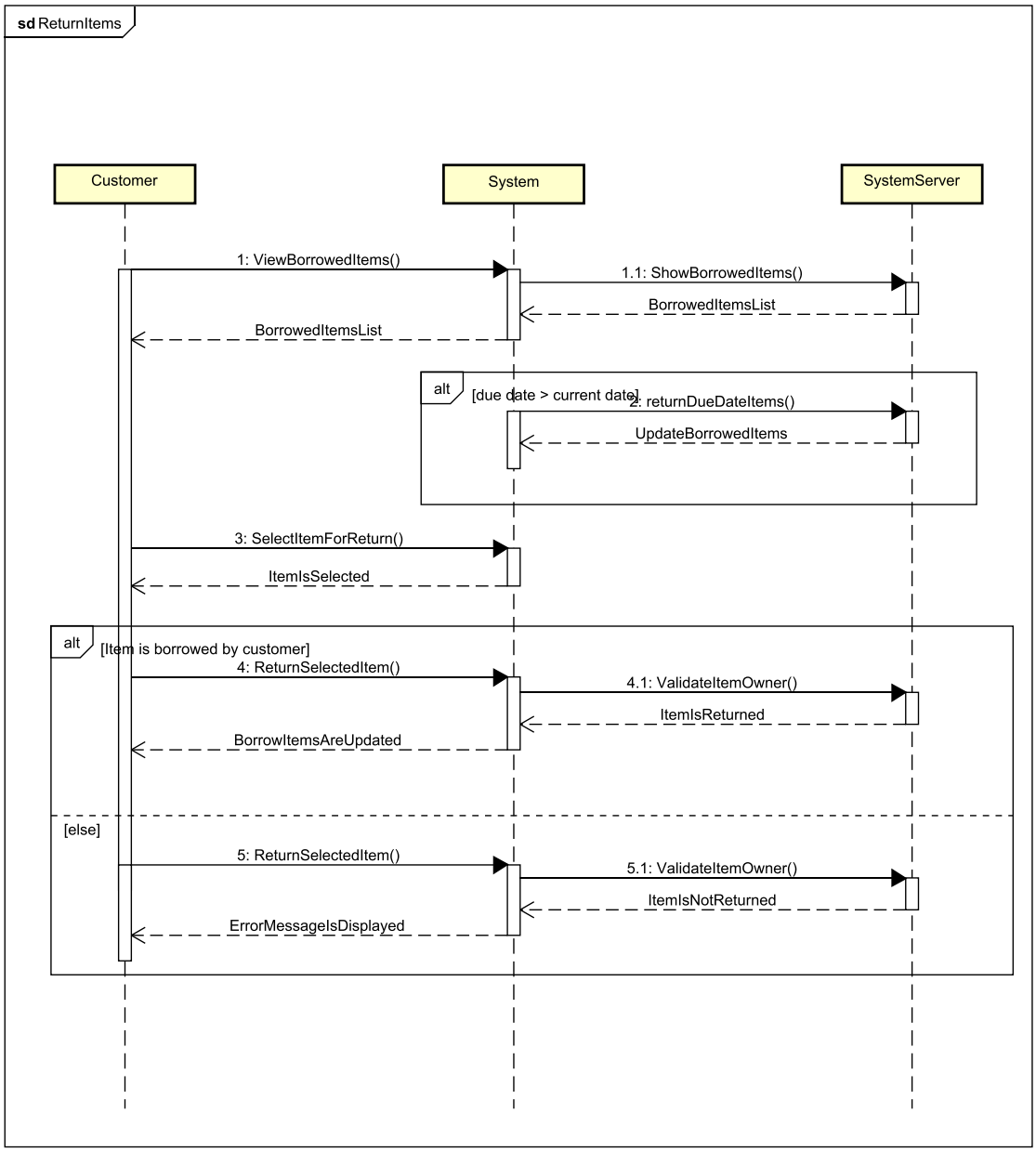
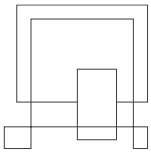


Diagram 3: System sequence diagram for returning items

As you can see on the SSD the first step for the customer is to view list of all borrowed items in the system. For this system needs to contact system server and then list of the items is shown. Then there is an if statement, which is checking if there are any items



which have due date past current date. If there are any, then the item is returned, and list is updated.

The next step for customer is to select item to return in this list. When item is selected, the system checks if this item is borrowed by this customer or not. If yes, then system contacts system server and item is returned. If no, then an error message is displayed.

2.5 **Domain Model** (Karrtiigehyen)

The domain model is used as a visual guide for the problem domain.

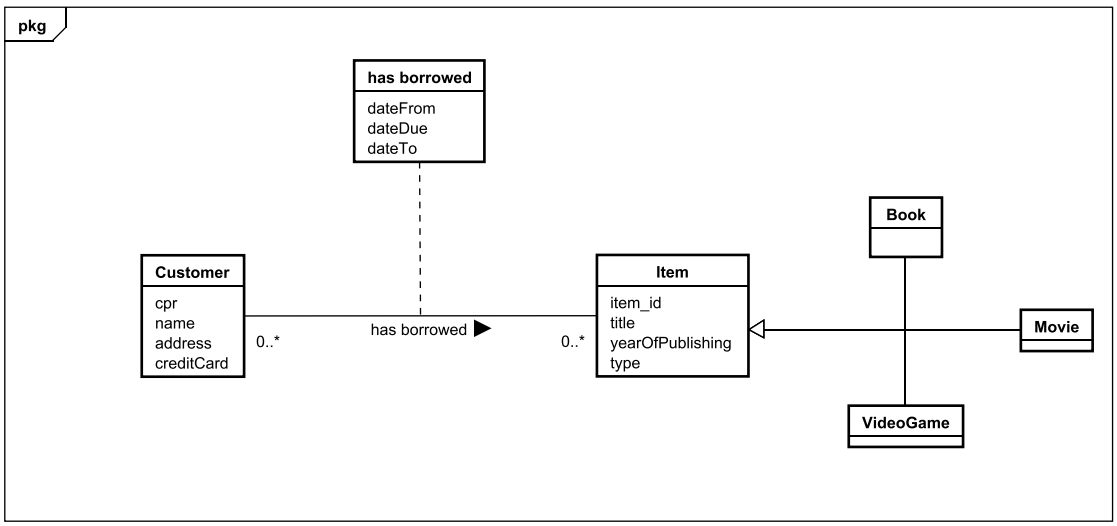
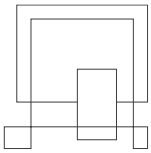


Diagram 4: Domain Model

The concepts (classes) present in the domain model are *Customer*, *Item*, *Book, Movie*, and *VideoGame*. The relationship between *Customer* and *Item* is a many-to-many relationship since many customers can borrow or reserve a certain item, and vice versa. The *Item* class acts as a superclass for the *Book, Movie*, and *VideoGame* classes. The classes *Book, Movie*, and *VideoGame* are chosen from the non-functional requirement which states that there should be these types of items in the library. The *has_borrowed* describes the relationship between *Customer* and *Item*, specifically when a customer borrows or reserves an item.



The development of the domain model was influenced by the user stories from the Requirements section.

3 **Design** (Jan Vasilcenko, Karrtiigehyen, Nicolas Popal)

In this section, the software will be designed more in-depth. The topics that will be discussed are the architecture of the system, design patterns, technologies used, and UI design choices. The outcome of this section will contain the necessary knowledge and diagrams that will be used for the implementation of the system.

3.1 **Architecture** (Karrtiigehyen)

This system is designed by using the logical architecture and layered architectural pattern. The logical architecture is the large-scale organization of the software classes into packages (Larman, 2005). A layer is a group of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system. The layers are organized in way that the 'higher' layers are dependent on the lower services.

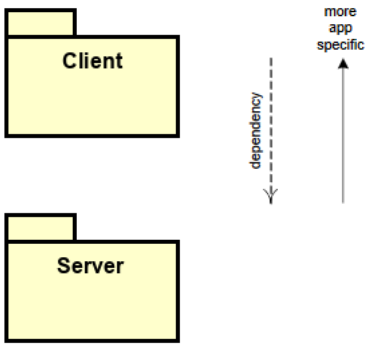
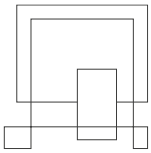


Diagram 5: shows the layered pattern for the library system

In the library system, there are two layers which are client and server as seen in diagram 6. The client contains the user interface and the logic involved with viewing the UI, and the server contains the application logic and domain objects. The server can also interface with the database to extract or manipulate data in from the database. Since the client can be said to be a 'higher' layer compared to the server, it is dependent on the services of the server layer.



The system was designed with layers because of multiple reasons. Firstly, there is separation of concerns. This reduces coupling and dependencies, while improving cohesion and increasing reusability of code. Some layers can also be swapped for new implementations without changing functionalities on other layers. This is mainly applicable to the client side, where change to the UI will be made constantly. While there are many more other benefits, these are the two main reasons for using the layered pattern.

3.2 **Design of Library System** (Jan Vasilcenko, Karrtiigehyen, Nicolas Popal)

In this sub-section, the package diagram will be shown first in order to get a bird's eye view of the system. Then, the package diagram will be discussed further into details with the help of some low detailed class diagrams (class diagrams with no variables and methods), highlighting the design patterns. At the end, some sequence diagrams will be shown and discussed to illustrate the flow of the system.

The SOLID principles were used as a guideline while designing the classes so that dependencies are reduced, meaning that the code should be loosely coupled, highly cohesive, context independent, and easy to divide into separate standalone components. This also makes testing the system much easier.

3.2.1 **Package diagram** (Jan Vasilcenko, Karrtiigehyen)

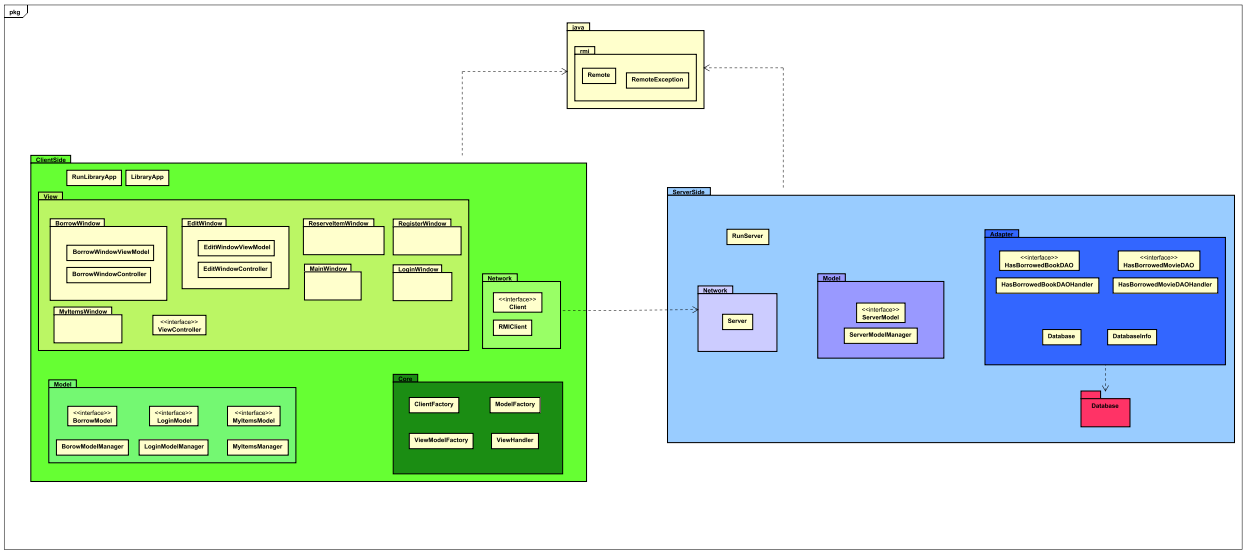


Diagram 6: package diagram of the system

The package diagram is similar to the layer diagram seen in Diagram 6, where there are two packages in the package diagram. The left one (green) is the *ClientSide*, and the right one (blue) is the *ServerSide*. Note that the shared package that is used by both the client and server packages is not shown in Diagram 7 since the package diagram is only meant to give an overview of the system. The shared package will be discussed later though.

The library system's connection is handled via Remote Method Invocation (RMI).

The pink package within the *ServerSide* package is the database package. The database is accessed by the adapter package by using Data Access Objects, which will be discussed later.

The package diagram can be found in Appendix D.

3.2.2 Client-Server Connection (Jan Vasilcenko)

Server creates a registry that client can access. Client is looking up for a registry and when invoking methods, it is using stub as a substitute of remote server object, on server side is skeleton. Both server and client are implementing Remote interface. Every item that the client is sending to server and vice-versa is implementing Serializable, to be able to send and read by client-server system. This system also uses call back to notify each client when changes occur. In case of RMI actual socket layer is hidden from developer so it is easier to deploy. This connection contains multithreaded server.

3.2.3 Client-side (Jan Vasilcenko)

The full low detailed class diagram can be located in Appendix E.

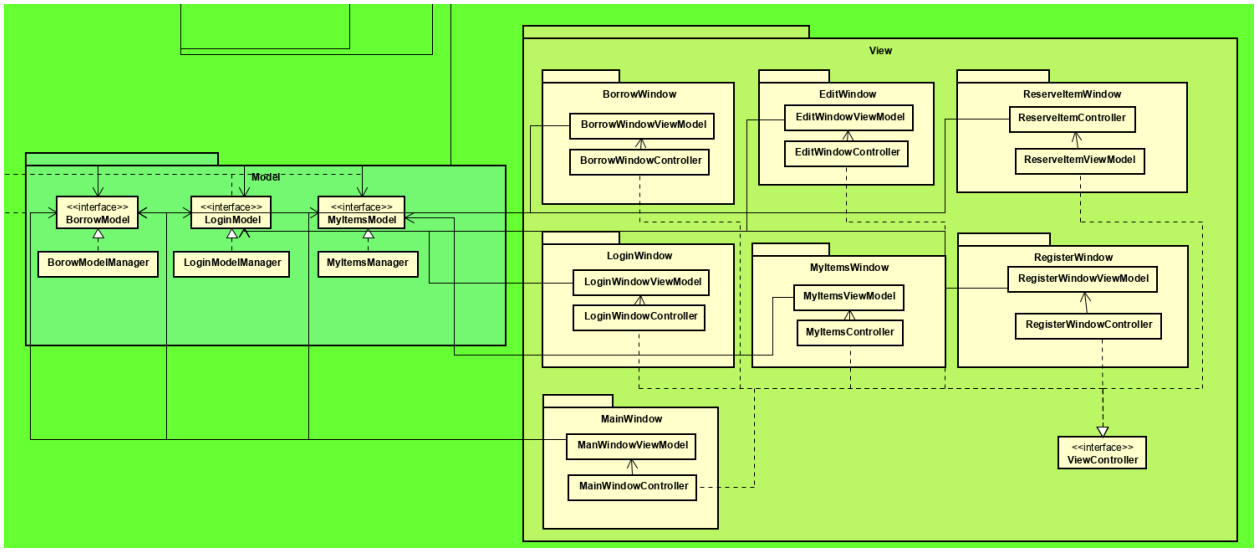


Diagram 7: shows the MVVM in the ClientSide

ClientSide contains many packages such as *View*, *Model*, *Core* and *Network*. **MVVM** design pattern is used in the client side, so that it provides division of responsibilities. One of which are Controllers, which hold the content of UI, and their logic is separated into ViewModels. ViewModels are then using Models to get access to logic in the system. The MVVM structure is useful for this project since multiple people can work on different parts of the library system without waiting for a part of the system to be

finished. For example, one person can work on the *View* package without the completion of the *Model* package.

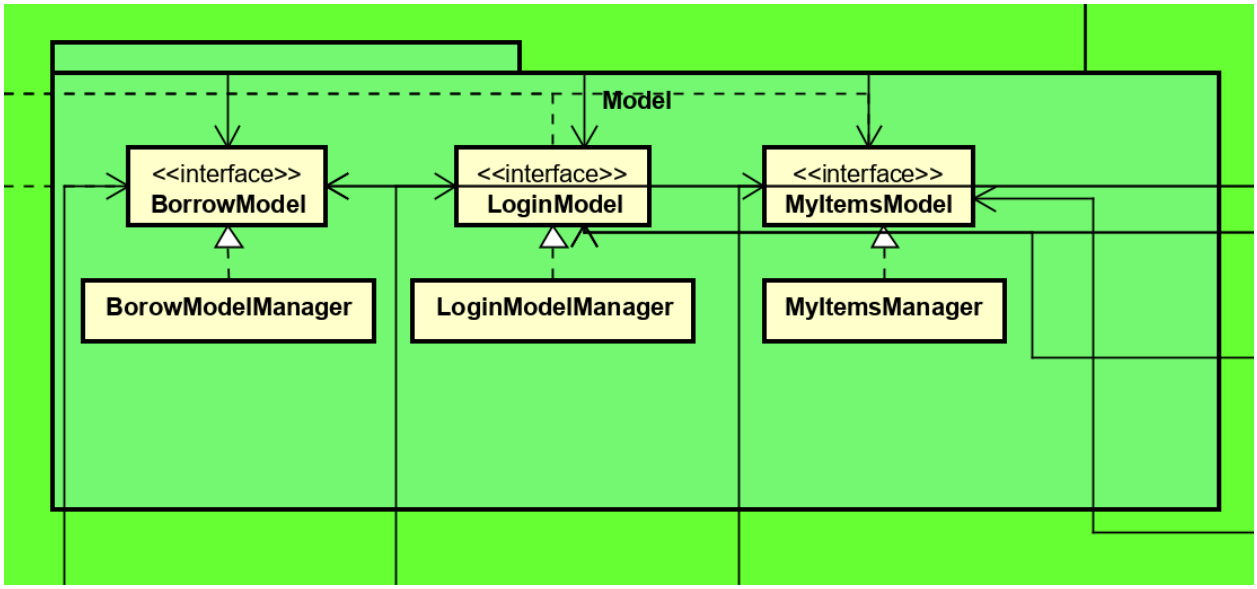


Diagram 8: shows the Model package in the ClientSide package

In the package *Model*, the logic for the client side is located. The model is constructed in a way that it complies with the solid principles, primarily the Interface Segregation Principle, which states that interfaces should not be over bloated, and the interface fits exactly what the client needs. For example, instead of one giant interface in the model, it is split up into several according to what interface the ViewModels are dependent on.

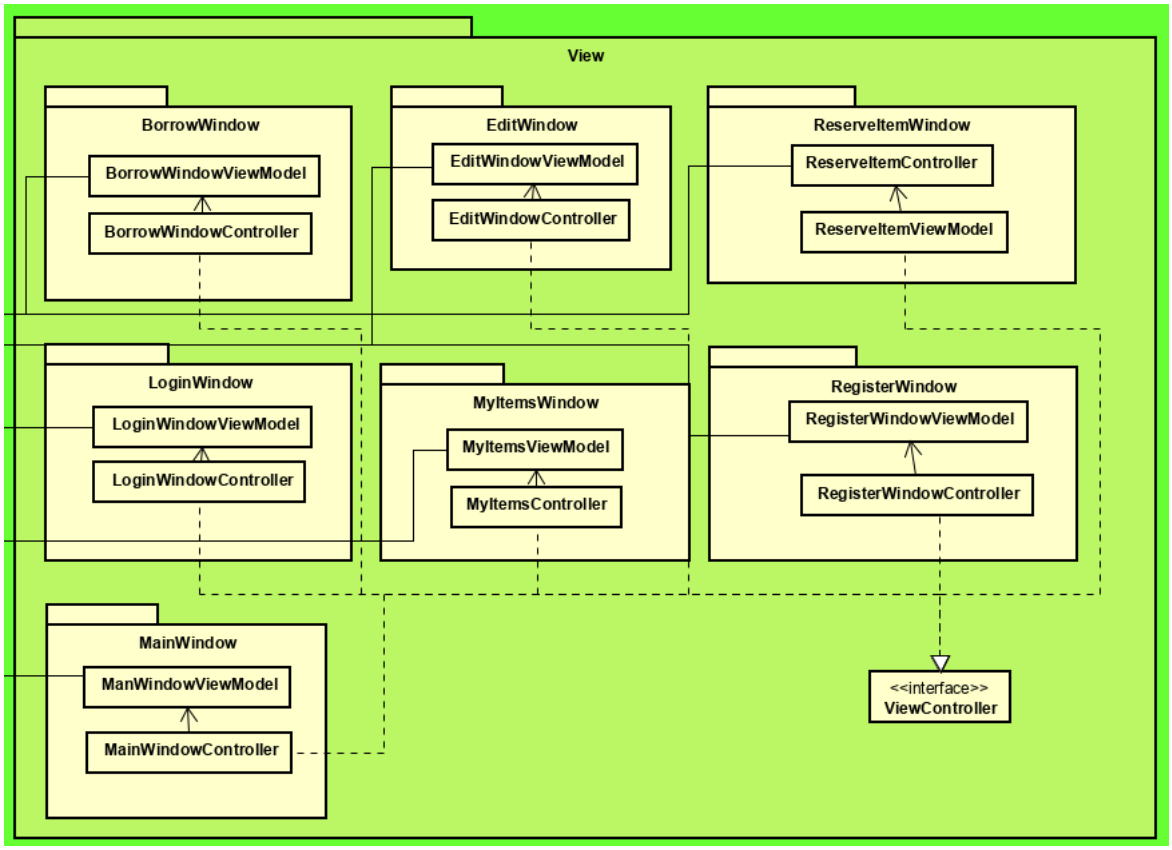


Diagram 9: shows the View package in the ClientSide package

The *View* package contains the classes necessary to display content of the system to the user. Each view is separated into two parts: *Controller* and *ViewModel*. *Controller* holds the FXML component references. *ViewModels* are used to separate application logic from *Controllers*, and it handles the events triggered including functionality with access to the Model.

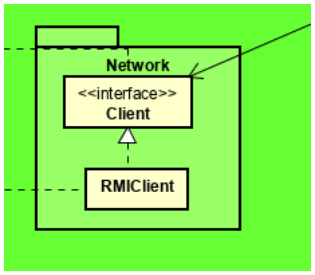


Diagram 10: shows the Network package in the ClientSide package

Network package is where the access to server is located. In this package *Client* is started and connected to the *Server*, registering for server call back, it is supplied with port and server name. It is also location of methods which are remotely called from server.

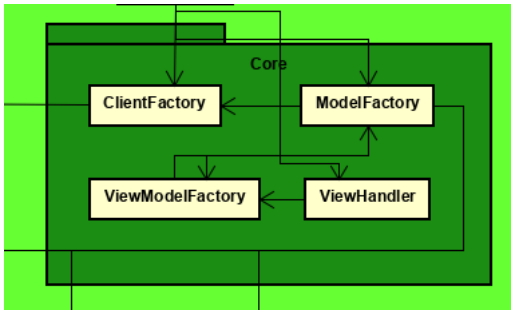


Diagram 11: shows the Core package in the ClientSide package

Core package holds Factory design pattern objects, which are used to create objects when they are needed. This pattern is also modified so that it uses part of Singleton design pattern, so that it holds one instance of that item and returns it when it is needed. Within this package, there are 4 classes, which are *ClientFactory*, *ModelFactory*, *ViewModelFactory* and *ViewHandler*. *ClientFactory*, which has an instance of *Client* interface. *ModelFactory*, that uses multiple *Models* and delegates *Client* to them. *ViewModelFactory* is responsible for creating and holding all *ViewModels* and to provide Models to each one of *ViewModels*. *ViewHandler*, which holds *ViewModelFactory,* is responsible for creating all the *Views* and switching between them.

3.2.4 **Shared** (Jan Vasilcenko)

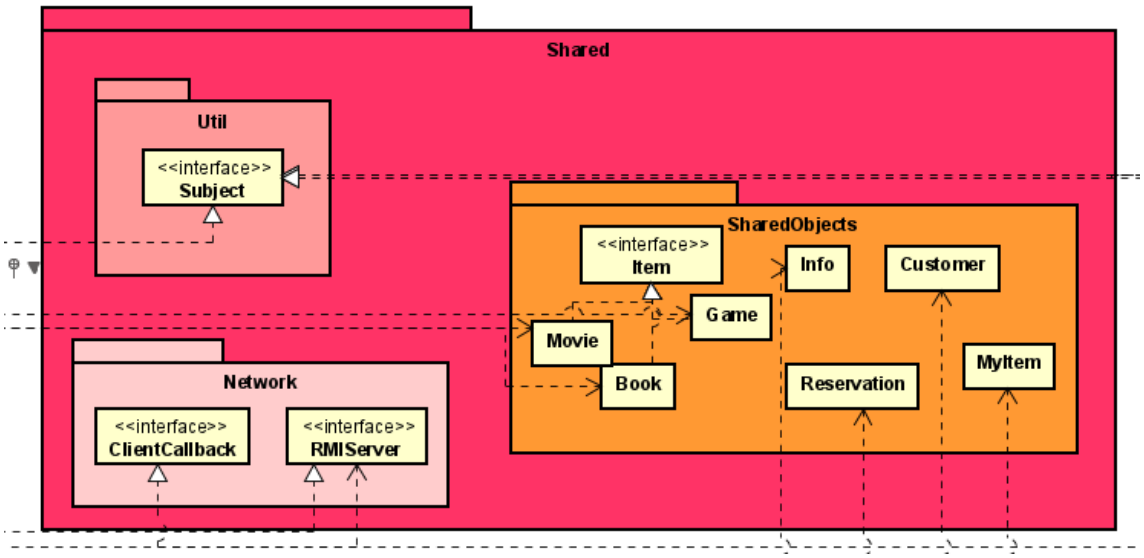


Diagram 12: shows the Shared package

There are multiple objects used by both *ClientSide* and *ServerSide*. For example, utilities for Observer pattern, which is Subject that can add or remove listeners for concrete subjects to implement, used to notify changes to each one of the clients, based on subscription idea. Also, the Objects of the Domain Model can be found in this package, so that client and server can be able to use them.

3.2.5 **Server-side** (Karrtiigehyen)

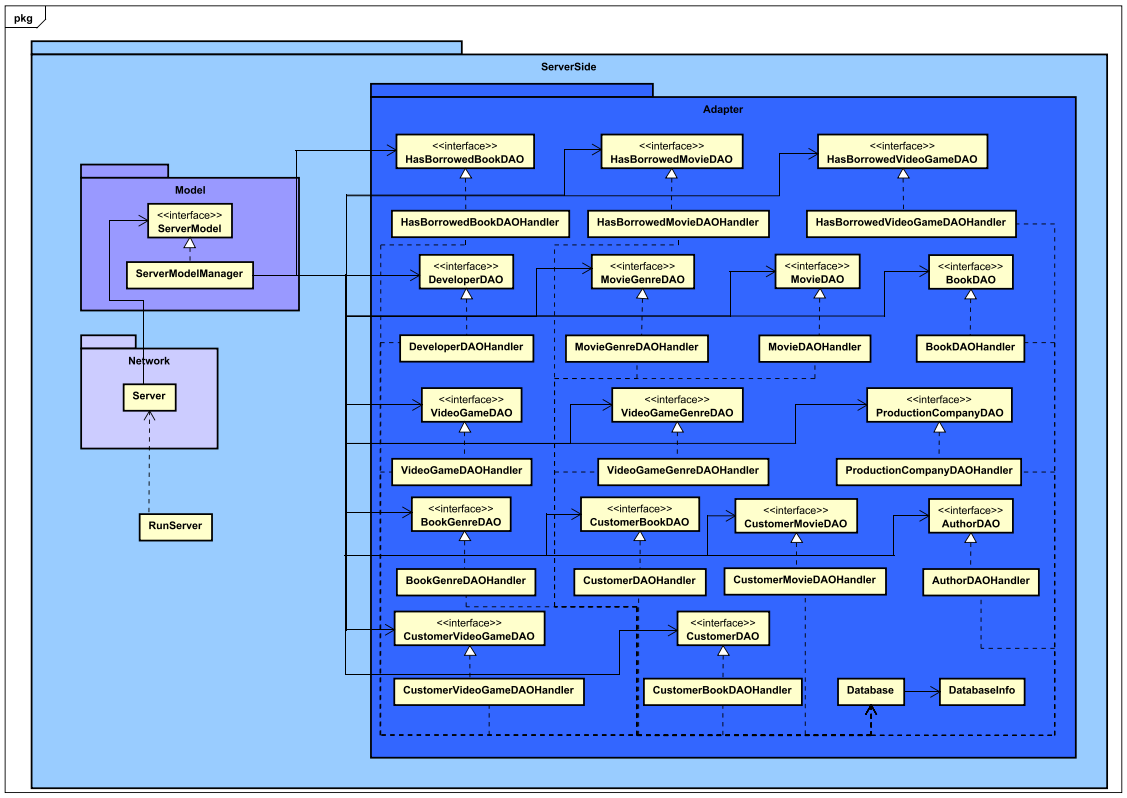
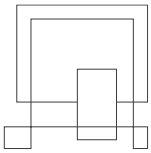


Diagram 13: ServerSide

In the *ServerSide* package, the RMI server, and the Data Access Objects (DAO) can be found.

Firstly, a connection to the database must be made in order to read or manipulate data from the database. This is done by using an API called Java Database Connectivity (JDBC), which will be explained further in the Technologies Used sub-section. **Singleton pattern** is used here so that the DAOs can get an instance of the database without instantiating it each time it is called.

Data Access Object pattern is a design pattern that is used in separating low level data accessing API or operations from high level business services (Tutorialspoint, 2020). DAO interfaces are used to define the methods to be performed on the *ServerModelManager*. DAO concrete classes are classes that implement the DAO



interfaces, where it can get data from the database. The DAOs are mainly designed using CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) as a guideline, but there are some classes where it is not used. For example, some classes will only need to read data from the database. The DAOs are also designed using the SOLID principles, which is why there are many DAO interfaces and concrete classes, to comply with the Single Responsibility Principle. Examples of how the SOLID principles are used in the DAOs can be found in the Implementation section.

An **adapter pattern** is used so that the *ServerModelManager* can call methods from the DAOs. In this case, the *Database* is the Adaptee, the DAO concrete classes are the Adapter, the DAO interfaces are the Target, and *ServerModelManager* is the Client.

There are several problems with the *ServerSide* however. Firstly, the DAOs violate the DRY (**D**on't **R**epeat **Y**ourself) rule. There are several lines of code that will repeat in the DAOs. The **command pattern** could have been used to avoid this problem. The command pattern turns a request into a stand-alone object that contains all information about the request (Refactoring Guru, 2020). The other problem is that the *ServerModel* does not comply with the SOLID principles. The class primarily violates the Dependency Inversion Principle by being a fat interface that the client, *ServerModelManager*, implements. This could have been fixed by having multiple interfaces that suit according to what the clients need, but this was omitted since *ServerModelManager* is the only client that is needed. But an argument could be made that the *ServerModelManager* could have been separated into several other classes to ensure that the Single Responsibility Principle is not violated.

3.2.6 Sequence diagram (Nicolas Popal)

Sequence diagram are based on the methods in the actual code and are drawn so that it can be seen what methods and classes are used sequentially in the system. For example, the system sequence diagrams were used for returning items and reserving/borrowing items from the library system, specifically books.

Borrowing and reserving book

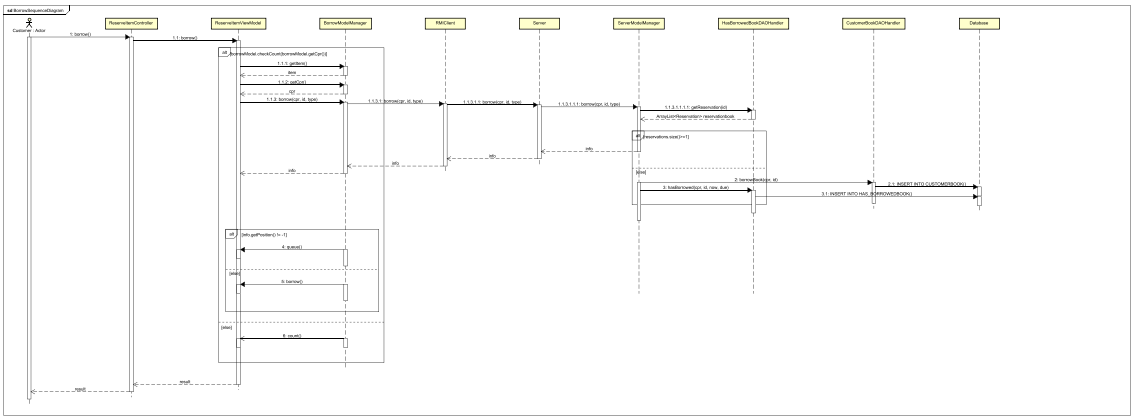


Diagram 14: Sequence diagram for borrowing and reserving items

Book is borrowed or reserved when the confirmation button is clicked. System is distinguishing whether the book is already borrowed or not. This depends on whether the book is reserved or borrowed. When the confirmation button is clicked by the customer the *borrow()* method is activated in the *ReserveItemController*.

As we can see between *ReserveItemViewModel* and *BorrowModelManager* classes are 2 if statements. The first one has a condition, which is checking if user already borrowed or reserved 3 or more items from the library based on their CPR. If customer has 3 or more items, the count of the items is returned. In the second if statement (which is inside of the first) we can see condition, which is checking if book is available or not. This depends whether this book is either reserved or borrowed by the customer.

Returning book

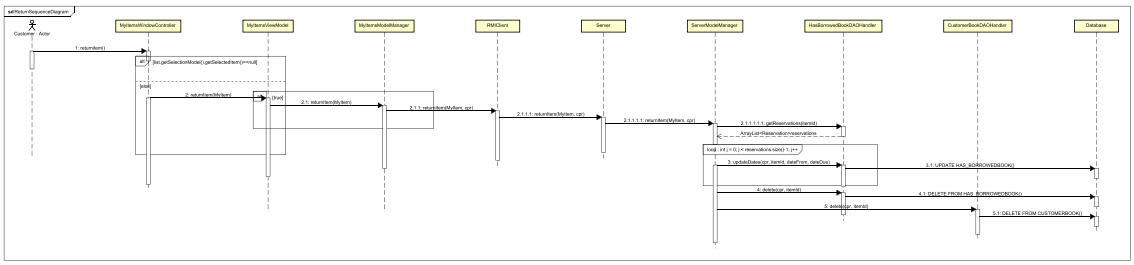
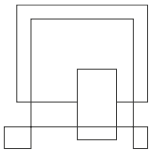


Diagram 15: Sequence diagram for returning books



Books can be returned after highlighting them in the My Items window and clicking on the return button. Also, when user clicks on My Items button in main menu, system checks for borrowed items which are past the return date and will return them automatically. This is also checked in main menu window and when user uses borrow items button in the menu.

In the *ServerModelManager* array list is created and filled with all the reservations of this item. Then in the for loop all the dates of these reservations in this array list are updated and executed to the database. Finally, system deletes this book from borrowed books, so it can be borrowed by other customers who have reserved that, or if it was not reserved it can be borrowed by anyone.

3.3 **Database** (Karrtiigehyen)

3.3.1 **Conceptual Model** (Karrtiigehyen)

The conceptual model is heavily influenced by the domain model. The extra entities that are added to the conceptual model of the database are for holding information about the items that can be retrieved by the server.

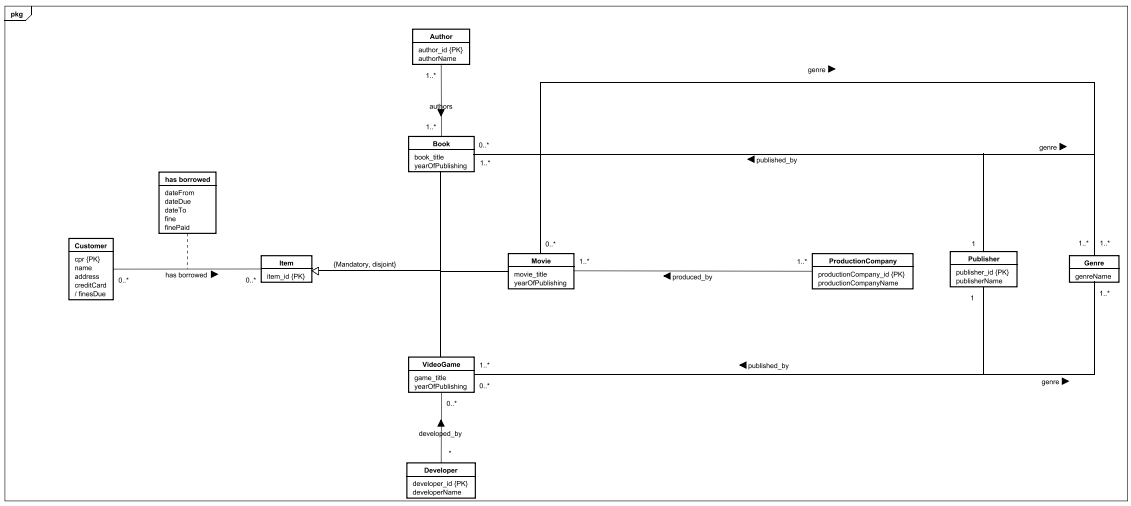


Diagram 16: EE/R diagram of the database

Entity/Relationship modelling models a business situation by describing the relevant entities and their relationships. For example, *VideoGame* and *Developer* are entities, and *developed_by* describes the relationship between the entities.

This conceptual model is in third normal form. Normalization is a technique for producing a set of relations with desirable properties, given the data requirements of an enterprise (Connolly and Begg, 2015). It is in third normal form since it satisfies the first and second normal form, and no non-primary key attribute are transitively dependent on the primary key.

The conceptual model can be separated into five parts: The *Customer* and *Item* part, *Book* part, *Movie* part, *VideoGame* part, and genre part.

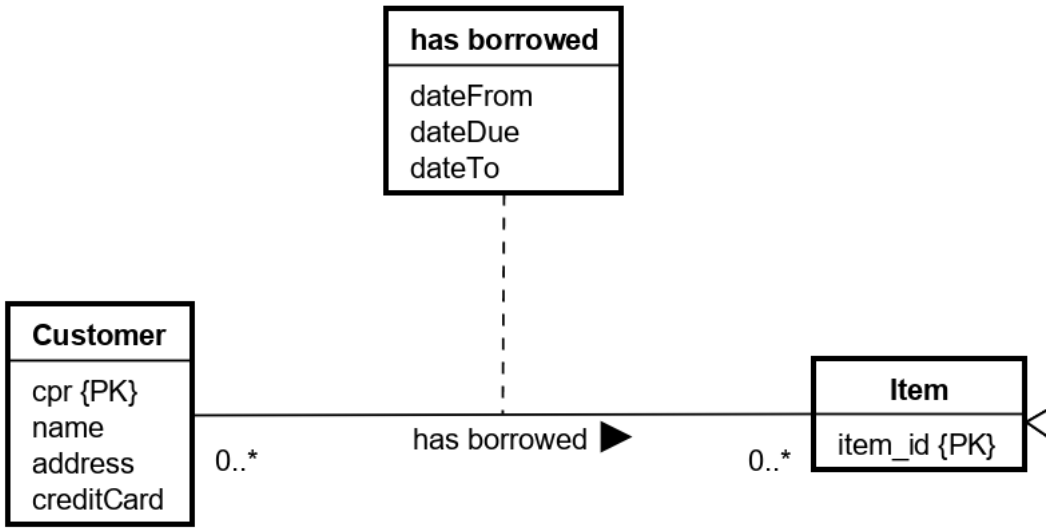


Diagram 17: Customer and Item part

The *Customer* and *Item* part consists of two strong entities, *Customer* and *Item*, and the relationship between the entities. The attributes in *Customer* are chosen from the *Register* use case description, with the cpr being the primary key. *Item* acts as a superclass for the subclasses *Book, Movie* and *VideoGame*, which inherit the primary key *item_id.* The relationship between *Customer* and *Item*, *has_borrowed*, holds the attributes that define the characteristics of borrowing and reserving an item. For

example, the attribute *dateFrom* is the date that the item was borrowed. *dateDue* is the due date for the item to be returned. *dateTo* is the date the item is returned by the customer (since customers can return items late). The *has_borrowed* is a many-to-many relationship since a customer can borrow many items and items can have many customers.

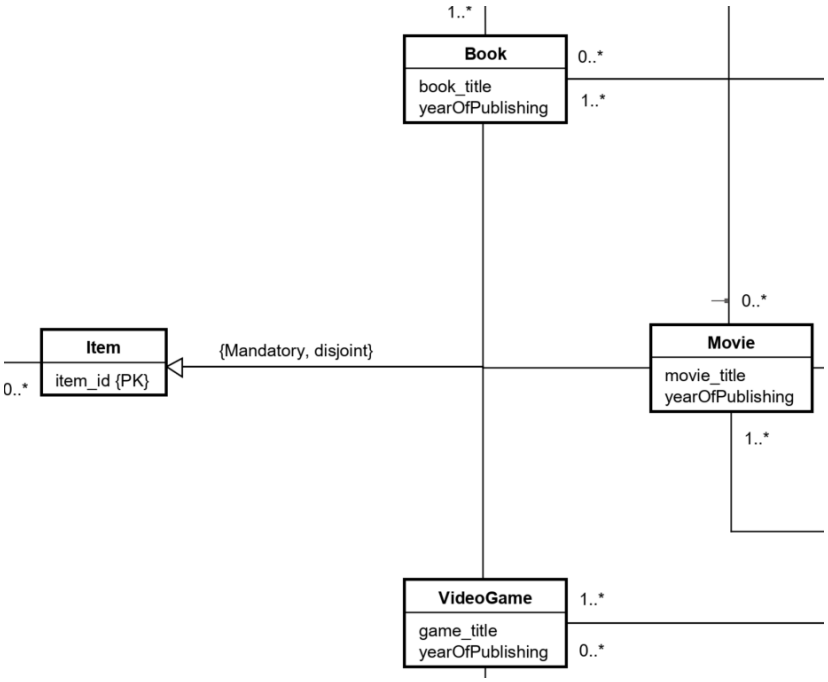


Diagram 18: Relationship between Item, Book, Movie, and VideoGame

Before moving onto the *Book, Movie,* or *VideoGame* parts, the extended entity/relationship modelling between *Item* and *Book, Movie* and *VideoGame* should be discussed. *Item* is the parent entity to the child entities *Book, Movie* and *VideoGame.* The participation constraint is mandatory and it is disjoint, which means there can be many relations, but one relation for each combined superclass/subclass (Connolly and Begg, 2015). Furthermore, the attribute, *item_id* in *Item,* will be inherited by the subclasses and acts as the primary key for the subclasses.

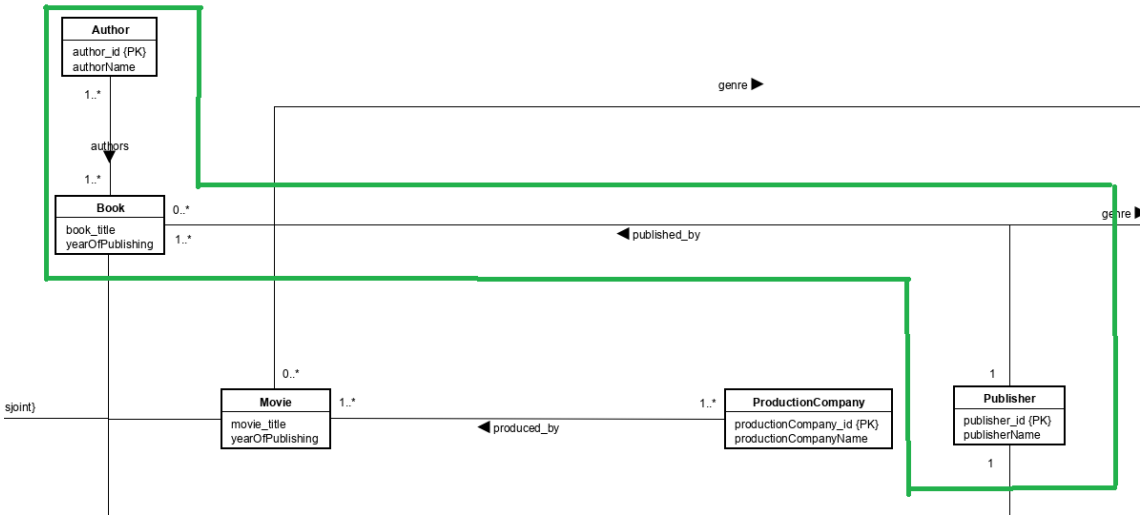


Diagram 19: Book part

The *Book* part consists of three strong entities which are *Book, Author* and *Publisher*. The *Book* entity contains the attributes *book_title* and *yearOfPublishing*. The *Author* entity contains *author_id* and *authorName*, with *author_id* as the primary key. The *author_id* was chosen as the primary key since there can be authors with the same *authorName*. This is the reason an id was chosen, not only for the *Author* entity, but also for the *Developer, Person, Publisher* and *ProductionCompany* entities. The many-to-many relationship, *authors*, describes the relationship between *Book* and *Author.* The *Publisher* entity contains the attributes *publisher_id* (primary key) and *publisherName.* The relationship *published_by*, describes the one-to-many relationship between *Book* and *Publisher* since a book can only have one publisher, but a publisher can publish many books.

Diagram 20: Movie part

The *Movie* part consists of two strong entities, *Movie* and *ProductionCompany*. The entity *Movie* has similar attributes as the *Book* entity, *movie_title* and *yearOfPublishing*, where the *yearOfPublishing* is the year the movie was released. The *ProductionCompany* has an id and the production company's name, and the relationship between *ProductionCompany* and *Movie* is *produced_by*, and it is a many-to-many relationship since a movie can have many production companies, and a production company can produce many movies.

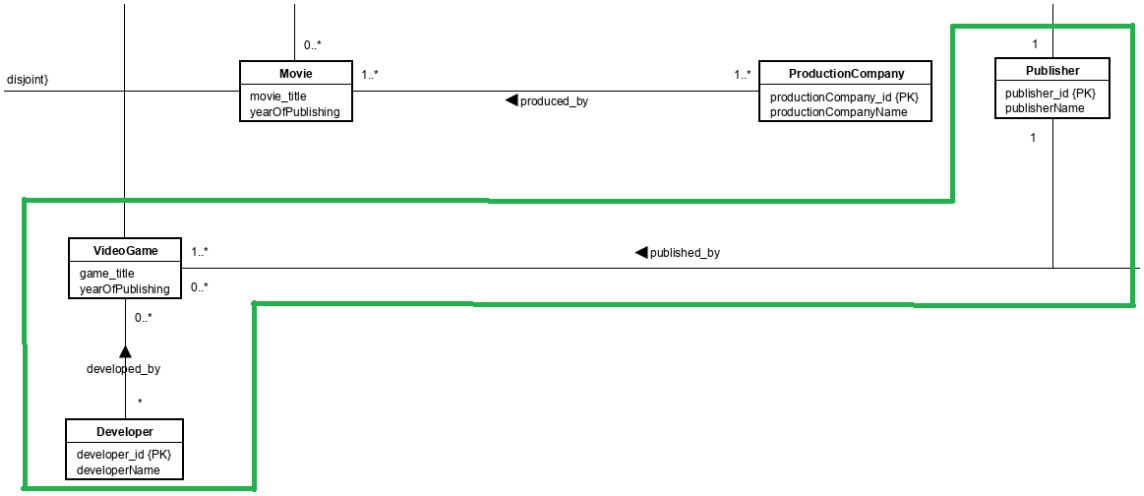


Diagram 21: VideoGame part

The *VideoGame* part has three strong entities which are *VideoGame, Developer* and *Publisher.* The *VideoGame* entity has attributes *game_title* and *yearOfPublishing,* similar to *Book* and *Movie*. The *Developer* entity has an id and a name, and a many-to-many relationship called *developed_by* describes the relationship between *Developer* and *VideoGame. VideoGame* is also related to *Publisher, published_by,* and it is a one-to-many relationship since a video game can only have one publisher, but a publisher can publish many video games.

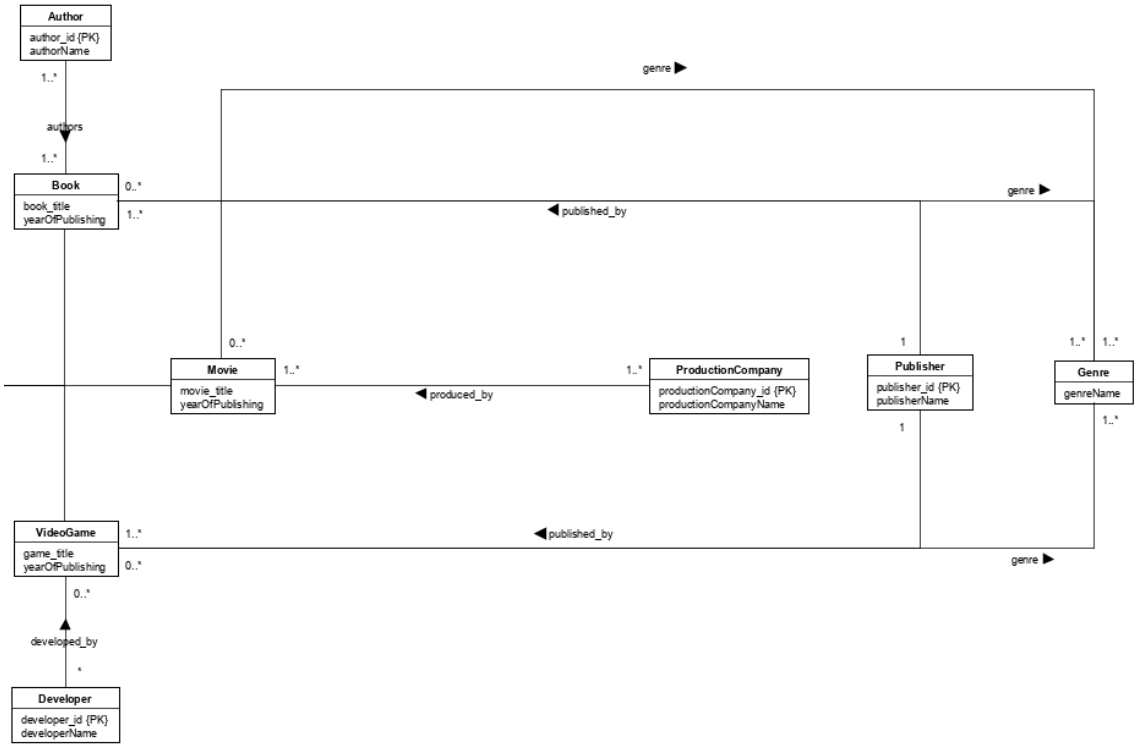


Diagram 22: Genre part

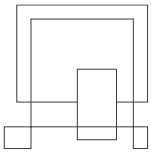
Finally, there is the *Genre* part. *Genre* is a strong entity with only one attribute *genreName.* The relationship between *Genre* and *Book, Movie* and *VideoGame* is called *genre*, and it is a many-to-many relationship since books, movies, and video games can have many genres, and genres can belong to many books, movies and video games.

The conceptual model can also be found in Appendix F.

3.3.2 Logical Model (Karrtiigehyen)

The logical model will be used as the basis for the creation of the physical database. Deriving relations for logical data model was done by following the following steps (Connolly and Begg, 2015):

1. strong entity types
2. weak entity types



3. one-to-many (1:*) binary relationship types
4. one-to-one (1:1) binary relationship types
5. one-to-one (1:1) recursive relationship types
6. superclass/subclass relationship types
7. many-to-many (*:*) binary relationship types
8. complex relationship types
9. multi-valued attributes

Since there were no one-to-one (1:1) binary relationship types, one-to-one (1:1) recursive relationship types, complex relationship types, and multi-valued attributes in the conceptual model, steps 4, 5, 8, and 9 are skipped.

Step 1. Strong entity types

The strong entities are *Customer, Author, Person, Developer, Publisher,* and *ProductionCompany*. Only an example of the logical model of the *Customer* will be shown here since the other strong entities are mapped using the same method.

```
3   Customer(cpr, name, address, creditCard)
4   PRIMARY KEY(cpr)
```

An entity *Customer* has attributes *cpr, name, address* and *creditCard*, with the *cpr* being the primary key.

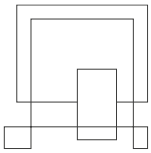
Step 2. Weak entity types

The only weak entity found in the conceptual model is the *Genre* entity.

```
25  Genre(genreName)
26  PRIMARY KEY(genreName)
```

Step 3. One-to-many (1:*) binary relationship types **AND Step 6**. Superclass/subclass relationship types

Step 3 and 6 are combined since the one-to-many binary relationship types in the conceptual model are also superclass/subclass relationship types.



Let us take step 6 first. As an example, the child entity *Book* will be shown.

```
44  Book(item_id, book_title, yearOfPublishing)
45  PRIMARY KEY(item_id)
```

Book inherited the attribute *item_id* from the parent entity *Item,* and is also the primary key.

Step 3 is done next which is adding an extra attribute that describes the relationship between *Book* and *Published*, and adding it as a foreign key as well.

```
32  Book(item_id, book_title, yearOfPublishing, published_by)
33  PRIMARY KEY(item_id)
34  FOREIGN KEY(published_by) REFERENCES Publisher(publisher_id)
```

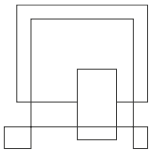
Step 7. Many-to-many (*:*) binary relationship types

Many-to-many binary relationships were the most plentiful in the conceptual model. The relationship between *Customer* and *Item* called *has_borrowed* will be shown as an example.

```
110  Has_borrowedBook(cpr, item_id, dateFrom, dateDue, dateTo, fine, finePaid)
111  PRIMARY KEY(cpr, item_id, dateFrom)
112  FOREIGN KEY(cpr) REFERENCES Customer(cpr)
113  FOREIGN KEY(item_id) REFERENCES Book(item_id)
```

The attributes in *has_borrowed* as well as the primary keys of *Customer* and *Book* are included as the attributes of *has_borrowed.* The primary keys are *cpr, item_id* and *dateFrom* since these will always be unique, and *cpr* and *item_id* act as foreign keys referencing *cpr* in *Customer* and *item_id* in *Item* respectively.

The full logical model is in Appendix G.



3.4 **Technologies Used** (Nicolas Popal)

For the library system, seven technologies were used. Library system is coded in **Java**. Java is used in the client side and in the server side. It is classed-based, object-oriented programming language and computing platform first released by Sun Microsystems (Java, 2020).

For the user interface **JavaFX** is used. JavaFX is used only in the client side of the library system. JavaFX is an open source client application platform (JavaFX, 2020). The applications written using this library can run consistently across multiple platforms (Tutorialspoint, 2020).

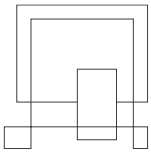
As the requirement was that the system needs to use database, **PostgreSQL** is used. PostgreSQL is an open-source object-relational database system (PostgreSQL, 2020). In the database are stored all the information needed for library system run.

Library system and database must be connected as well. For this connection **PostgreSQL JDBC Driver API** is used. It is an open-source JDBC driver written in pure java (PostgreSQL JDBC, 2020). It was necessary for us to use this library to establish connection between database and library system.

JUnit 5 was used in the testing part as well. JUnit is a testing tool used for testing class methods in order to check the functionality of the code. It is usually used in parts of the code that involve logic to verify code integrity. In our case, we used Junit in combination with Mockito (see below) to check functionality of Adapters and *ServerModelManager* (as these contained most of the logic).

In the testing part, Mockito and JUnit 5 are used. Mockito framework is used because then system is not dependant on the database existence. Mockito can simulate database existence without having to access database or create data in there.

The last used technology is the Maven repository. Maven is an automation tool used in Java to help with managing file structure, tools, and plugins. It was decided to use



Maven during Server testing as one of our testing tools (Mockito) required many dependencies that were difficult to manage without a proper tool.

3.5 **UI design choices** (Jan Vasilcenko)

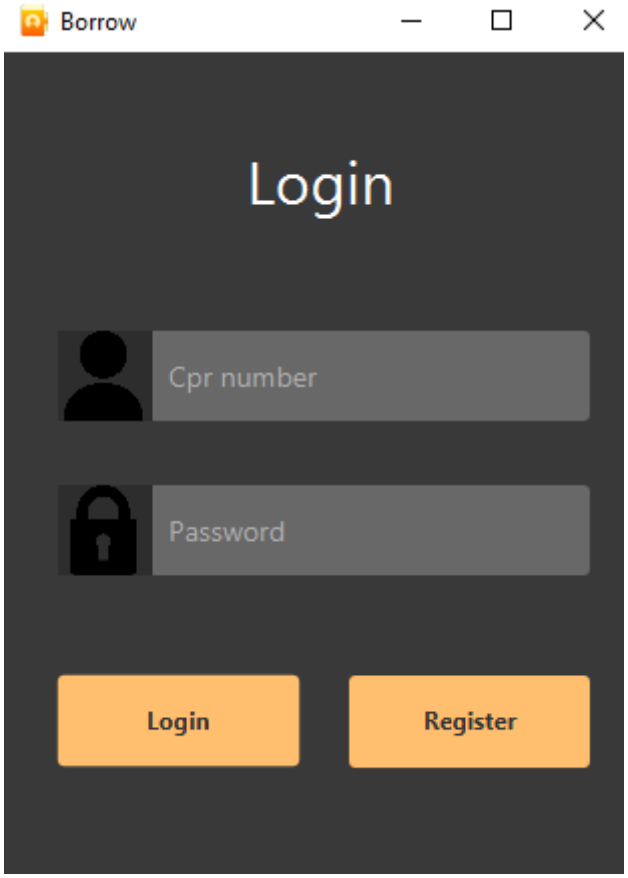


Figure 1: Main window of the library system

Application has dark background to not put pressure on user's eyes. Text is white colour, so that it is readable, and buttons styled in orange colour, which is in psychology described as energetic, uplifting, and bright. Text fields are grey to be easily recognized, when blank it tells user what to type in there. Buttons are big and overall look of UI tries to be as simple as possible. Icons are trying to help user with identifying what to type in text fields. In some windows, there are white headers to help user with navigation.

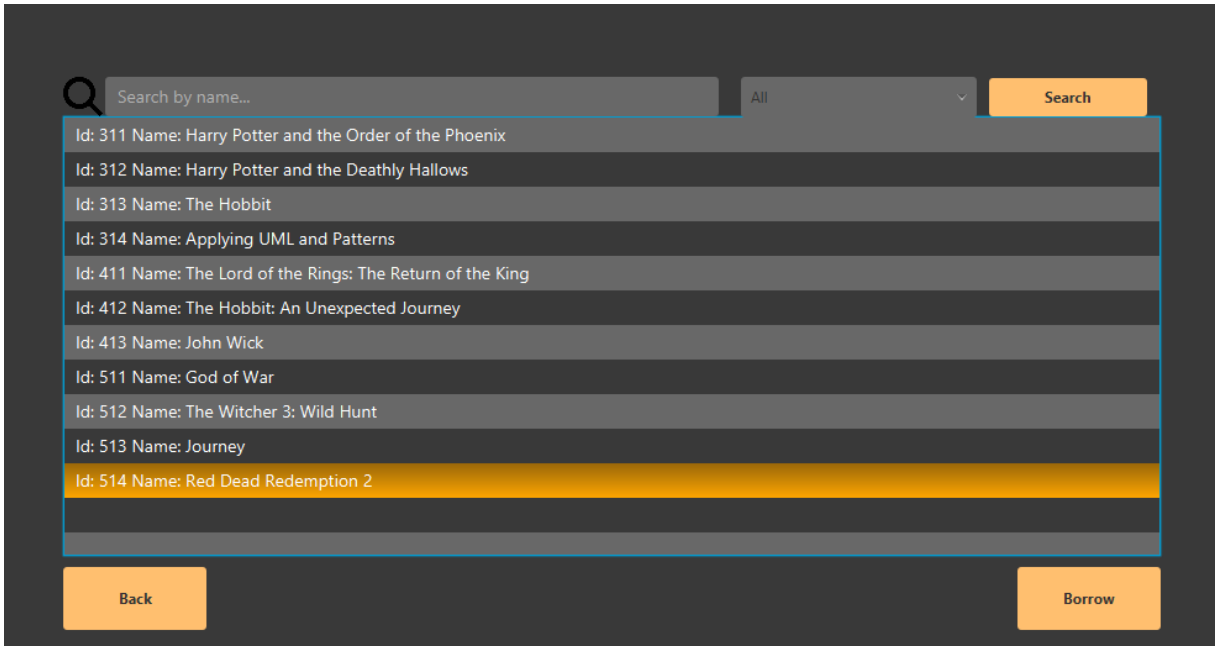


Figure 2: Borrow window of the library system

In windows with lists, they are styled in two colours, lighter gray and darker gray, so that items are easily recognisable. Items displayed in tables have white text colour to be readable. When an item from the list is selected, the selected item will be highlighted in orange so that it is distinguishable from the unselected items. Search field is in lighter gray colour with a search icon to be easily spotted.

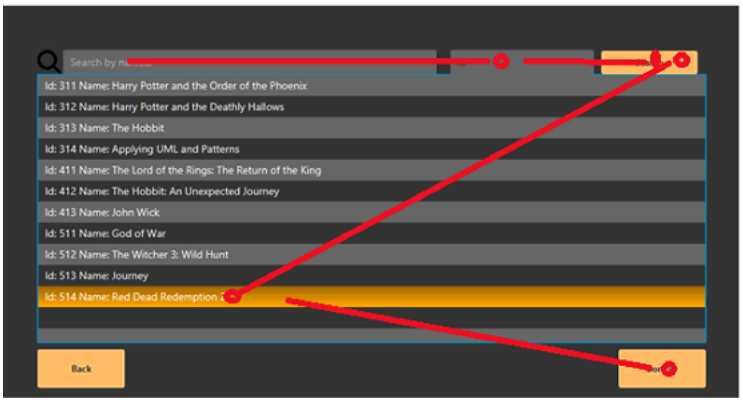
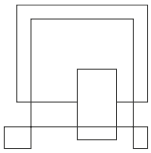


Figure 3: showing the intuitive flow of the borrow window



The search field is placed at the top of the list, and the category drop-down menu is placed right next to it. The search button is put to the top right placement of the window, and the borrow button located at the bottom right of the window. This gives it a nice flow where it will be intuitive for the user to operate.

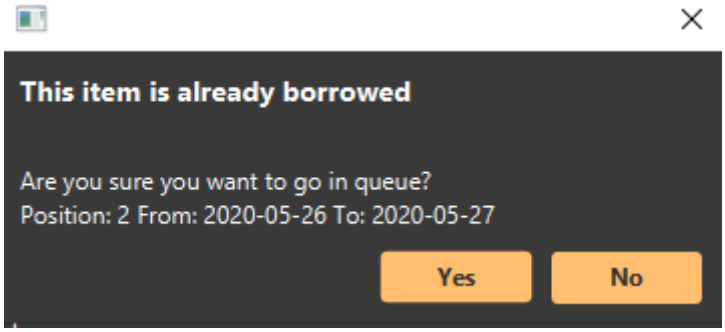
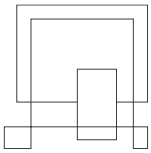


Figure 4: Pop up window alerting cutomer of entering queue for reservation

Alerts apply the same design principles as windows.



4 **Implementation** (Jan Vasilcenko, Karrtiigehyen)

In this section, only interesting and significant parts of the code will be shown and explained. The code for the database is not shown in this section since it is simply the creation of tables according to the logical model shown in the previous section, and populating them with data. GUI is also not discussed here as they are not uniquely done from the standard way of implementing them. Same goes for the Styles, which was done using CSS.

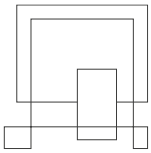
The parts that will be shown are how the RMI connection is set up, call back, an example of the observer pattern, the database connection which implements a singleton pattern, DAOs, how the adapter pattern is used for the DAOs, and the models on the server and client sides. The borrowing, reserving, and returning items will be used as examples to showcase the parts where applicable.

The source code for the database and the library system in Appendix H.

The full class diagrams can be found in Appendix I.

4.1 **RMI** (Jan Vasilcenko)

To establish RMI connection there must be Server and Client constructed and connected, to establish server we need to create an interface which extends Remote and each method must throw RemoteException. Communication is via stub, which is placeholder for remote object on client side.



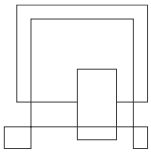
```
public interface RMIServer extends Remote
{
  boolean registrer(Customer customer) throws RemoteException;
boolean login(Customer customer) throws RemoteException;
boolean change(Customer customer) throws RemoteException;
String getname(Customer customer) throws RemoteException;
List<Item> getlist() throws RemoteException;
void remove(Item item) throws RemoteException;
void registerClientDel(ClientCallback clientCallback) throws RemoteException;
void registerClientItems(ClientCallback clientCallback)throws RemoteException;
String getAuthor(int id) throws RemoteException;
Item getItem(int id) throws RemoteException;
String getDeveloper(int id) throws RemoteException;
String getProductionCompany(int id) throws RemoteException;
Info borrow(long cpr,int id,String type) throws RemoteException;
  void borrowconfirm(long cpr,int id,String type, LocalDate date)throws RemoteException;
  List<MyItem> getitems(long cpr)throws RemoteException;
  boolean returnItem(MyItem item,long cpr)throws RemoteException;
  boolean checkCount(long cpr)throws RemoteException;
  List<Item> searchItem(String category,String searchtext)throws RemoteException;
  String getGenre(int id, String type)throws RemoteException;
}
```

Code 1: RMIServer interface

Actual server class must import this interface and create registry on port 1099, because registry uses this port and remote objects also use it and then bind that registry to some name to create a server. The server is calling method *UnicastRemoteObject.exportObject* to export that object and create a stub.

```
public class Server implements RMIServer
{
  private ServerModel serverModel;

  public Server(ServerModel serverModel) throws RemoteException
  {
    UnicastRemoteObject.exportObject( obj: this, port: 0);
    this.serverModel = serverModel;
  }

  public void startServer() throws RemoteException, AlreadyBoundException
  {
    Registry registry = LocateRegistry.createRegistry( port: 1099);
    registry.bind( name: "LibraryServer", obj: this);
  }
```

Code 2: Server class implementing RMIServer interface



To create a client, we need to also have some interface, with methods which should be called on server. Actual client is the class implementing this interface and starting the client so, that it is exporting itself with *UnicastRemoteObject* to create a stub, then getting registry by IP and port 1099 and lastly having instance variable of server Interface to be instantiated with method that finds registry by name, which we entered when constructing server.

```
public class RMIClient implements Client, ClientCallback
{
  private RMIServer server;
  private PropertyChangeSupport support;
  public RMIClient() {support=new PropertyChangeSupport( sourceBean: this);}

  @Override public void startClient()
  {
    try
    {
      UnicastRemoteObject.exportObject( obj: this, port: 0);
      Registry registry=LocateRegistry.getRegistry( host: "localhost", port: 1099);
      server = (RMIServer) registry.lookup( name: "LibraryServer");
      server.registerClientItems( clientCallback: this);
    }
    catch (RemoteException | NotBoundException e)
    {
      e.printStackTrace();
    }
  }
```

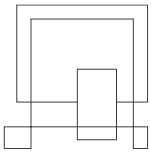
Code 3: startClient() method in RMIClient

4.2 **Call back** (Jan Vasilcenko)

To implement client callback we need interface which will be implemented by Client, also extending Remote and referenced in methods of server interface to register to those callbacks.

```
public interface ClientCallback extends Remote
{
void updateMyItems()throws RemoteException;
}
```

Code 4: ClientCallback interface



Method for registering ClientCallBack in server interface.

```
void registerClientItems(ClientCallback clientCallback)throws RemoteException;
```

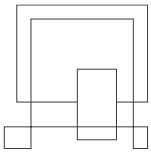
Code 5: registerClientItems() method

In actual implementation, this method try to call method implemented by client and then is adding a listener to listen to.

```
@Override public void registerClientItems(ClientCallback clientCallback)
{
  PropertyChangeListener listener = null;
  PropertyChangeListener finalListener = listener;

  listener = evt-> {
    try
    {
      clientCallback.updateMyItems();
    }
    catch (RemoteException e)
    {
      serverModel.removeListener( eventName: "return", finalListener);
    }
  };
  serverModel.addListener( eventName: "return", listener);
}
```

Code 6: registerClientItems() being implemented

On client side we are implementing ClientCallBack interface and in start client method registering for callback.



```
public class RMIClient implements Client, ClientCallback
{
  private RMIServer server;
  private PropertyChangeSupport support;
  public RMIClient() {support=new PropertyChangeSupport( sourceBean: this);}

  @Override public void startClient()
  {
    try
    {
      UnicastRemoteObject.exportObject( obj: this, port: 0);
      Registry registry=LocateRegistry.getRegistry( host: "localhost", port: 1099);
      server = (RMIServer) registry.lookup( name: "LibraryServer");
      server.registerClientItems( clientCallback: this);
    }
    catch (RemoteException | NotBoundException e)
    {
      e.printStackTrace();
    }
  }
```

Code 7: method for registering call back

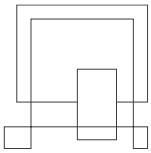
In the end we are implementing callback method, in this case firing property. Which is used when returning borrowed items, so that every client updates its list.

```
@Override public void updateMyItems()
{
  support.firePropertyChange( propertyName: "return", oldValue: null, newValue: null);
}
```

Code 8: firing property

4.3 Observer Pattern (Jan Vasilcenko)

To create observer pattern, there must be interface or abstract class, which has methods to add or remove listeners. In this example it is named event, so the events are defined by String.



```
public interface Subject
{
  void addListener(String eventName, PropertyChangeListener listener);
  void removeListener(String eventName, PropertyChangeListener listener);
}
```

Code 9: Subject interface

This interface must be implemented by concrete class which we want to listen to.

In the LibrarySystem it is extended by MyItemsModel interface and implemented by MyItemsModelManager. For it to work properly, there needs to be an instance variable of PropertyChangeSupport, instantiated in constructor and then implementing those two methods from Subject interface.

```
public class MyItemsModelManager implements MyItemsModel
```

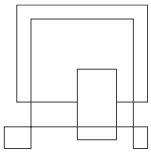
Code 10: MyItemsModelManager implements MyItemModel

```
private PropertyChangeSupport support;

public MyItemsModelManager(Client client)
{
  this.client = client;
  support = new PropertyChangeSupport( sourceBean: this);
```

Code 11: constructor for the MyItemsModelManager

```
 @Override public void addListener(String eventName,
     PropertyChangeListener listener)
 {
   support.addPropertyChangeListener(eventName, listener);
 }

 @Override public void removeListener(String eventName,
     PropertyChangeListener listener)
 {
support.addPropertyChangeListener(eventName, listener);
 }
```

Code 12: implementing the methods from Subject



When it comes to concrete listener there are two ways, either we implement PropertyChangeListener or we can do everything in one method. In constructor we add listener with this as argument and then include name of method to be executed. In this case we are listening to model, which is listening to a client, which is listening for server to fire event, when the event is fired the MyItemsWindowViewModel is notified and runs a method that request actual list of items that client has borrowed.

```
public MyItemsWindowViewModel(MyItemsModel myItemsModel)
{
  this.myItemsModel = myItemsModel;
  this.myitems = new SimpleListProperty<>();
  myItemsModel.addListener( eventName: "return",this::update);
}

private void update(PropertyChangeEvent propertyChangeEvent)
{
  Platform.runLater(()->
      {
        List<MyItem> list = myItemsModel.getitems(myItemsModel.getCpr());
        ObservableList<MyItem> aaa = FXCollections.observableArrayList(list);
        myitems.set(aaa);
      });
}
```

Code 13: shows the update() method

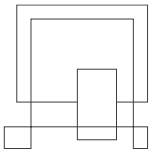
4.4 **Database Connection and Singleton Pattern** (Karrtiigehyen)

In order for the connection to work, a PostgreSQL JDBC Driver had to be downloaded and added as a library to the project.

Then a class called *Database* is created.

```
10  public class Database
11  {
12    private DatabaseInfo databaseInfo = new DatabaseInfo();
13    private String driver = "org.postgresql.Driver";
14    private String url = "jdbc:postgresql://localhost:5432/LibrarySystem?currentSchema=library";
15    private String user = databaseInfo.getUsename();
16    private String password = databaseInfo.getPassword();
```

Code 14: Database class



The driver for the database and the url are stored in String variables *driver* and *url* repectively to be used later. The username and password are stored in String variables as well by using getters to another class for privacy reasons.

```
28      public void connect()
29      {
30        try
31        {
32          Class.forName(driver);
33        }
34        catch (ClassNotFoundException e)
35        {
36          e.printStackTrace();
37        }
38
39        try
40        {
41          connection = DriverManager.getConnection(url, user, password);
42          JOptionPane.showMessageDialog( parentComponent: null,  message: "Connected");
43        } catch(SQLException e)
44        {
45          e.printStackTrace();
46        }
47      }
```

Code 15: shows the connect() method

In the method called *connect()*, the connection to the database is established. If the connection is successful, then a message will show up saying “Connected”. Then a getter for the connection is done so that it can be called in the DAOs.

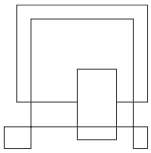
```
49      public Connection getConnection() { return connection; }
```

Code 16: shows the getConnection() method

A singleton is also used in this class.

```
18      private static Database instance;
19      private static Lock lock = new ReentrantLock();
23      private Database() { connect(); }
54      public static Database getInstance()
55      {
56        if (instance == null)
57          synchronized (lock)
58          {
59            if (instance == null)
60              instance = new Database();
61          }
62        return instance;
63      }
```

Code 17: Singleton pattern



The singleton pattern is done by creating a static instance variable that references itself with a private constructor. In the *getInstance()* method, lazy instantiation is done to avoid instantiating every time the method is called, and it is thread safe. The singleton pattern is used so that the DAOs can call the same instance of the Database once it is created.

4.5 **DAOs** (Karrtiigheyen)

The DAOs are resposible for creating, reading, updating and deleting data from the database. So, the DAOs are created to cater the needs of the *ServerModel*. CRUD was used as a framework, but some DAOs only need to read from the database. Examples of borrowing and returning items will be shown to give an overview of how the DAOs are implemented.

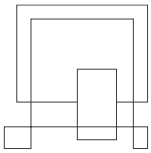
Borrowing items

The following example is for borrowing books.

Firstly, a DAO interface (Target) must be made, so that the DAO concrete class (Adapter) can implement the methods.

```
1   package ServerSide.Adapter;
2
3   import Shared.SharedObjects.MyItem;
4   import Shared.SharedObjects.Reservation;
5
6   import java.time.LocalDate;
7   import java.util.ArrayList;
8
9   public interface HasBorrowedBookDAO
10  {
11    void hasBorrowed(long cpr, int id, LocalDate dateFrom, LocalDate dateDue);
12    ArrayList<Reservation> getReservations(int id);
13    ArrayList<MyItem> getMyBooks(long cpr);
14    void delete(long cpr, int id);
15    void updateDates(long cpr, int id, LocalDate dateFrom, LocalDate dateDue);
16    int checkCount(long cpr);
17    ArrayList<Reservation> getReservations();
18  }
```

Code 18: an example for the DAO interface



In the DAO concrete class, an instance of the Database must be get and the connection as well.

```
12  public class HasBorrowedBookDAOHandler implements HasBorrowedBookDAO
13  {
14    private Connection connection;
15
16    public HasBorrowedBookDAOHandler()
17    {
18      this.connection = Database.getInstance().getConnection();
19    }
```

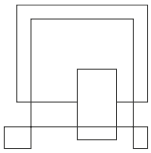
Code 19: Connection to the Database in DAO concrete class

For borrowing a book, the cpr of the Customer, book's id, the date that is borrowed, and the date due must be inserted in the table *has_borrowedBook* in the database as seen in Code 20.

```
21    @Override public void hasBorrowed(long cpr, int id, LocalDate dateFrom,
22        LocalDate dateDue)
23    {
24      try
25      {
26        String insertsql = "INSERT INTO HAS_BORROWEDBOOK VALUES(?, ?, ?, ?, ?, ?, ?)";
27        PreparedStatement insertHasBorrowedBook = connection.prepareStatement(insertsql);
28        insertHasBorrowedBook.setLong( i: 1, cpr);
29        insertHasBorrowedBook.setInt( i: 2, id);
30        insertHasBorrowedBook.setDate( i: 3, Date.valueOf(dateFrom));
31        insertHasBorrowedBook.setDate( i: 4, Date.valueOf(dateDue));
32        insertHasBorrowedBook.setDate( i: 5, Date.valueOf(dateDue));
33        insertHasBorrowedBook.setInt( i: 6,  i1: 0);
34        insertHasBorrowedBook.setInt( i: 7,  i1: 0);
35
36        insertHasBorrowedBook.executeUpdate();
37        insertHasBorrowedBook.close();
38      }
39      catch (SQLException e)
40      {
41        e.printStackTrace();
42      }
43    }
```

Code 20: inserting the customer's and book's information in table in the database

The cpr of the customer and the id of the book must also be inserted in another table called *CustomerBook*, but it is done using the principle of the *hasBorrowed()* method in Code 20.

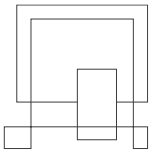


But when customer wants to borrow a book, the system must check whether whether the book is not borrowed or reserved by any other customers. This is done by checking the whether any books with the same id as the book that the customer wants to borrow is present in the *has_borrowedBook* table.

```
71  @Override public ArrayList<Reservation> getReservations(int id)
72  {
73    try
74    {
75      String querysql = "SELECT HAS_BORROWEDBOOK.* FROM HAS_BORROWEDBOOK WHERE ITEM_ID = ? ORDER BY(DATEFROM)";
76      PreparedStatement selectReservations = connection.prepareStatement(querysql);
77      selectReservations.setInt( i: 1, id);
78      ResultSet resultSet = selectReservations.executeQuery();
79
80      ArrayList<Reservation> customerReservation = (ArrayList<Reservation>) convertReservations(resultSet);
81      resultSet.close();
82      selectReservations.close();
83      return customerReservation;
84    }
85    catch (SQLException e)
86    {
87      e.printStackTrace();
88    }
89    return null;
90  }
```

Code 21: checking whether books with the same id are present in the table

If there are any books present, then it will be added to an Arraylist and returned. On line 80 in Code 21, it is calling another method called *convertReservations()* and takes the resultSet as an argument. This method is shown in Code 22.



```
protected List<Reservation> convertReservations(ResultSet resultSet)
{
  ArrayList<Reservation> customerReservation = new ArrayList<>();
  try
  {
    while(resultSet.next())
    {
      long cpr = resultSet.getLong( s: "cpr");
      int item_id = resultSet.getInt( s: "item_id");
      LocalDate datefrom = LocalDate.parse(resultSet.getDate( s: "datefrom").toString());
      LocalDate dateDue = LocalDate.parse(resultSet.getDate( s: "datedue").toString());
      LocalDate dateTo = LocalDate.parse(resultSet.getDate( s: "dateto").toString());
      int fine = resultSet.getInt( s: "fine");
      int finePaid = resultSet.getInt( s: "finepaid");
      Reservation reservation = new Reservation(cpr, item_id, datefrom, dateDue, dateTo, fine, finePaid);
      customerReservation.add(reservation);
    }
    return customerReservation;
  }
  catch (SQLException e)
  {
    e.printStackTrace();
  }
  return Collections.emptyList();
}
```

Code 22: shows the convertReservations() method

This method converts the resultSet into types that Java can read, and add it into an object of Reservation. This is done so that it complies with the Single Responsibility Principle, and it also nullifies the violations of the DRY rule as it can be used again in other methods. Alternatively, the command design pattern could have been used to minimize the replication of code.

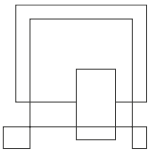
Returning books

The following example is for returning books.

There are two ways of returning books, to return them manually or automatically returning them when the due date is past.

Let’s look at the manual way of returning first.

When a customer wants to return a book, it has to be deleted from the table in the database.



```
@Override public void delete(long cpr, int id)
{
  try
  {
    String querysql = "DELETE FROM HAS_BORROWEDBOOK WHERE CPR = ? AND ITEM_ID = ?";
    PreparedStatement delete = connection.prepareStatement(querysql);
    delete.setLong( i: 1, cpr);
    delete.setInt( i: 2, id);
    delete.executeUpdate();
    delete.close();
  }
  catch (SQLException e)
  {
    e.printStackTrace();
  }
}
```

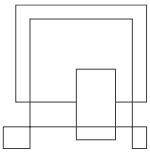
Code 23: Deleting book from the table based on its id

But when returning a book, the dates must be fixed for the customers on reservation for said book. This is done by updating the dates in the table as seen in Code 24.

```
@Override public void updateDates(long cpr, int id, LocalDate dateFrom,
    LocalDate dateDue)
{
  try
  {
    String querysql = "UPDATE HAS_BORROWEDBOOK SET DATEFROM = ?, DATEDUE = ?, DATETO = ? WHERE CPR = ? AND ITEM_ID = ?";
    PreparedStatement update = connection.prepareStatement(querysql);
    update.setDate( i: 1, Date.valueOf(dateFrom));
    update.setDate( i: 2, Date.valueOf(dateDue));
    update.setDate( i: 3, Date.valueOf(dateDue));
    update.setLong( i: 4, cpr);
    update.setInt( i: 5, id);
    update.executeUpdate();
    update.close();
  }
  catch (SQLException e)
  {
    e.printStackTrace();
  }
}
```

Code 24: updates the dates of books

The automatic returning of books should happen when the due date for the books are past the current date. This is done in the *ServerModelManager*, but the DAO must first return all of the borrowed books and the dates so that a method can be done in the *ServerModelManager* that whether the due date is past the current date.



```
214  @Override public ArrayList<Reservation> getReservations()
215  {
216    try
217    {
218      String querysql = "SELECT HAS_BORROWEDBOOK.* FROM HAS_BORROWEDBOOK ORDER BY(DATEFROM)";
219      PreparedStatement selectReservations = connection.prepareStatement(querysql);
220      ResultSet resultSet = selectReservations.executeQuery();
221
222      ArrayList<Reservation> customerReservation = (ArrayList<Reservation>) convertReservations(resultSet);
223      resultSet.close();
224      selectReservations.close();
225      return customerReservation;
226    }
227    catch (SQLException e)
228    {
229      e.printStackTrace();
230    }
231    return null;
232  }
```

Code 25: gets all of the borrowed books and dates

As seen in line 222 in Code 25, the *convertReservations(),* which can be seen in Code 22, is used again to minimize the repetition of code.

After the checking of the dates, the *ServerModelManager* can just call the *delete()* method showed in Code 23 to delete the book from the table in the database.

4.6 **Adapter** (Karrtiigehyen)

The adapter is used so that the *ServerModelManager* can use the DAOs.

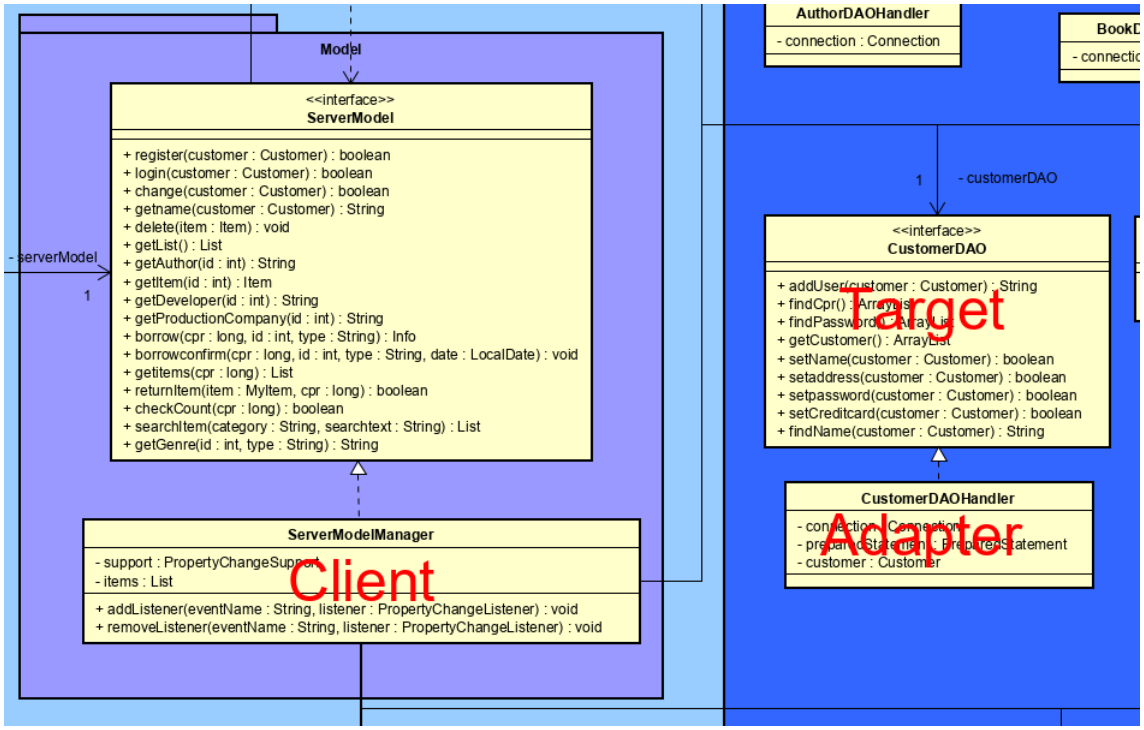
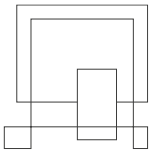


Diagram 23: shows how the adapter pattern is used

In the above diagram, an example of how the adapter pattern is used is shown. The database is not show in the diagram, but it acts as the adaptee.

4.7 **Server Model** (Jan Vasilcenko)

It is an interface which extends Subject to implement observer pattern and holds all methods which are requested by clients.



```
public interface ServerModel extends Subject
{
boolean register(Customer customer);
boolean login(Customer customer);
boolean change(Customer customer);
String getname(Customer customer);
void delete(Item item);
List<Item> getList();
String getAuthor(int id);
Item getItem(int id);
String getDeveloper(int id);
String getProductionCompany(int id);
Info borrow(long cpr,int id,String type);
void borrowconfirm(long cpr,int id,String type, LocalDate date);
  List<MyItem> getitems(long cpr);
  boolean returnItem(MyItem item,long cpr);
  boolean checkCount(long cpr);
  List<Item> searchItem(String category,String searchtext);
  String getGenre(int id, String type);
}
```

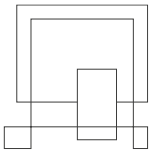
Code 26: ServerModel extending Subjetc

In its implementation it has instance variables of all Data Access Objects which are part of adapter pattern.

```
public class ServerModelManager implements ServerModel
{
  private CustomerDAO customerDAO = new CustomerDAOHandler();
  private BookDAO bookDAO = new BookDAOHandler();
  private PropertyChangeSupport support;
  private List<Item> items = new ArrayList<>();
  private AuthorDAO authorDAO;
  private MovieDAO movieDAO;
  private VideoGameDAO videoGameDAO;
  private DeveloperDAO developerDAO;
  private ProductionCompanyDAO productionCompanyDAO;
  private CustomerBookDAO customerBookDAO;
  private CustomerVideoGameDAO customerVideoGameDAO;
  private CustomerMovieDAO customerMovieDAO;
  private HasBorrowedVideoGameDAO hasBorrowedVideoGameDAO;
  private HasBorrowedMovieDAO hasBorrowedMovieDAO;
  private HasBorrowedBookDAO hasBorrowedBookDAO;
  private BookGenreDAO bookGenreDAO;
  private MovieGenreDAO movieGenreDAO;
  private VideoGameGenreDAO videoGameGenreDAO;
```

Code 27: ServerModelManager

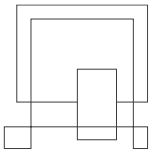
To reduce dependency their implementations are attributes in constructor, where they are instantiated.



```
public ServerModelManager(AuthorDAO authorDAO, MovieDAO movieDAO,
    VideoGameDAO videoGameDAO, DeveloperDAO developerDAO,
    ProductionCompanyDAO productionCompanyDAO,
    CustomerBookDAO customerBookDAO,
    CustomerVideoGameDAO customerVideoGameDAO,
    CustomerMovieDAO customerMovieDAO, HasBorrowedBookDAO hasBorrowedBookDAO,
    HasBorrowedMovieDAO hasBorrowedMovieDAO,
    HasBorrowedVideoGameDAO hasBorrowedVideoGameDAO,
    BookGenreDAO bookGenreDAO, VideoGameGenreDAO videoGameGenreDAO,
    MovieGenreDAO movieGenreDAO)
{
  this.authorDAO = authorDAO;
  this.movieDAO = movieDAO;
  this.videoGameDAO = videoGameDAO;
  this.developerDAO = developerDAO;
  this.customerBookDAO = customerBookDAO;
  this.productionCompanyDAO = productionCompanyDAO;
  this.customerVideoGameDAO = customerVideoGameDAO;
  this.customerMovieDAO = customerMovieDAO;
  this.hasBorrowedVideoGameDAO = hasBorrowedVideoGameDAO;
  this.hasBorrowedMovieDAO = hasBorrowedMovieDAO;
  this.hasBorrowedBookDAO = hasBorrowedBookDAO;
  this.bookGenreDAO = bookGenreDAO;
  this.movieGenreDAO = movieGenreDAO;
  this.videoGameGenreDAO = videoGameGenreDAO;
  this.support = new PropertyChangeSupport( sourceBean: this);
}
```

Code 28: instantiating in constructor

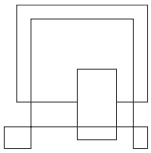
In this class all of the methods are handled and implemented as thread safe by making them synchronized, so only one client at a time has access to these methods. For example method Borrow we need cpr of client, id of item and type of item. Then we are calling DAOs to retrieve every items from database and add them to one ArrayList. If the item is already borrowed we return information about reserving, with particular date and expect users answer. If it is not borrowed already, we are accessing database and borrowing item to client.



```
@Override public synchronized Info borrow(long cpr, int id, String type)
{
  ArrayList<Reservation> reservations = new ArrayList<>();
  ArrayList<Reservation> reservationbook = hasBorrowedBookDAO
      .getReservations(id);
  ArrayList<Reservation> reservationgame = hasBorrowedVideoGameDAO
      .getReservations(id);
  ArrayList<Reservation> reservationmovie = hasBorrowedMovieDAO
      .getReservations(id);
  reservations.addAll(reservationbook);
  reservations.addAll(reservationgame);
  reservations.addAll(reservationmovie);
  if (reservations.size() >= 1)
  {
    LocalDate datedue = reservations.get(reservations.size() - 1).getDatedue()
        .plusDays(1);
    return new Info( position: reservations.size() + 1, datedue);
  }
  else
  {
    LocalDate now = LocalDate.now();
    LocalDate due = LocalDate.now().plusDays(1);
    switch (type)
    {
      case "Book":
        customerBookDAO.borrowBook(cpr, id);
        hasBorrowedBookDAO.hasBorrowed(cpr, id, now, due);
        break;
      case "Movie":
        customerMovieDAO.borrowMovie(cpr, id);
        hasBorrowedMovieDAO.hasBorrowed(cpr, id, now, due);
        break;
      case "Game":
        customerVideoGameDAO.borrowVideoGame(cpr, id);
        hasBorrowedVideoGameDAO.hasBorrowed(cpr, id, now, due);
        break;
    }
  }
```

Code 29: borrow() method

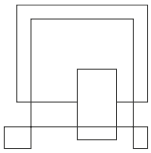
Reservation is handled in another method. There if client agrees to enter queue and reserve item, DAOs are called depending on type of item and inserting reservation into database.



```
@Override public synchronized void borrowconfirm(long cpr, int id, String type,
    LocalDate date)
{

  LocalDate from = date;
  LocalDate due = date.plusDays(1);
  switch (type)
  {
    case "Book":
      customerBookDAO.borrowBook(cpr, id);
      hasBorrowedBookDAO.hasBorrowed(cpr, id, from, due);
      break;
    case "Movie":
      customerMovieDAO.borrowMovie(cpr, id);
      hasBorrowedMovieDAO.hasBorrowed(cpr, id, from, due);
      break;
    case "Game":
      customerVideoGameDAO.borrowVideoGame(cpr, id);
      hasBorrowedVideoGameDAO.hasBorrowed(cpr, id, from, due);
      break;
  }
}
```

Code 30: confirmation of borrow

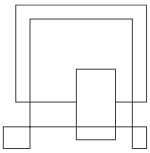
In case of returning items. Firstly, we are checking if that client actually has item borrowed, else the system notifies the user that he does not have that item right now. Then the system deletes borrow information from database and then is rearranging reservations for earlier date. Lastly it fires an event, which updates reservation of every client. This is the case for book, film and videogame.



```
@Override public synchronized boolean returnItem(MyItem item, long cpr)
{
    switch (item.getType())
    {
      case "Book":
        ArrayList<Reservation> reservations = hasBorrowedBookDAO
            .getReservations(item.getId());
        if (cpr != reservations.get(0).getCpr())
        {
          JOptionPane.showMessageDialog( parentComponent: null,
              message: "You do not have this book. You are on queue");
          return false;
        }
        reservations.get(0).setDatedue(LocalDate.now());
        for (int j = 0; j < reservations.size() - 1; j++)
        {
          reservations.get(j + 1)
              .setDatefrom(reservations.get(j).getDatedue().plusDays(1));
          reservations.get(j + 1)
              .setDatedue(reservations.get(j + 1).getDatefrom().plusDays(1));
          hasBorrowedBookDAO.updateDates(reservations.get(j + 1).getCpr(),
              reservations.get(j + 1).getId(),
              reservations.get(j + 1).getDatefrom(),
              reservations.get(j + 1).getDatedue());
        }
        hasBorrowedBookDAO.delete(reservations.get(0).getCpr(),
            reservations.get(0).getId());
        customerBookDAO.delete(reservations.get(0).getCpr(),
            reservations.get(0).getId());
        support.firePropertyChange( propertyName: "return", oldValue: null, newValue: null);
        break;
```

Code 31: returning items

There is also implemented automatic returning of items if they user does not return item till due date. This method is called each time someone is trying to open borrowing window or myitems window. This method simply checks for each section (books,movies and videogames) dates and deletes, those that are outdated, ensuring that they are returned.



```
@Override public void checkDate()
{
  ArrayList<Reservation> bookreservations = hasBorrowedBookDAO.getReservations();
  ArrayList<Reservation> moviereservations = hasBorrowedMovieDAO.getReservations();
  ArrayList<Reservation> videogamereservations = hasBorrowedVideoGameDAO.getReservations();


  for (int i = 0; i < bookreservations.size(); i++)
  {
    if(bookreservations.get(i).getDatedue().isBefore(LocalDate.now()))
    {

      hasBorrowedBookDAO.delete(bookreservations.get(i).getCpr(),bookreservations.get(i).getId());
      customerBookDAO.delete(bookreservations.get(i).getCpr(),bookreservations.get(i).getId());
    }
  }
  for (int i = 0; i < moviereservations.size(); i++)
  {
    if(moviereservations.get(i).getDatedue().isBefore(LocalDate.now()))
    {
      hasBorrowedMovieDAO.delete(moviereservations.get(i).getCpr(),moviereservations.get(i).getId());
      customerMovieDAO.delete(moviereservations.get(i).getCpr(),moviereservations.get(i).getId());
    }
  }
  for (int i = 0; i < videogamereservations.size(); i++)
  {
    if(videogamereservations.get(i).getDatedue().isBefore(LocalDate.now()))
    {
      hasBorrowedVideoGameDAO.delete(videogamereservations.get(i).getCpr(),videogamereservations.get(i).getId());
      customerVideoGameDAO.delete(videogamereservations.get(i).getCpr(),videogamereservations.get(i).getId());
    }
  }
}
```

Code 32: automatic returning of items

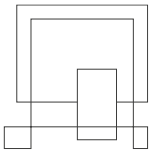
4.8 **Client-Side Models** (Jan Vasilcenko)

Models are classes which hold program logic on client side. In this system they are intersections between ViewModels and Client. Library system contains three client side models, LoginModel,BorrowModel and MyItemsModel each of them handling different operations. They are also passing cpr of logged customer, which could be implemented better by static variable and method.

For example, let us take the most complicated one MyItemModel.

```
public interface MyItemsModel extends Subject
{
  void setCpr(long cpr);
  long getCpr();
  List<MyItem> getitems(long cpr);
  boolean returnItem(MyItem item);
}
```

Code 33: MyItemsModel extending Subject



Its implementation is part of the observer pattern, which is already explained and part of implementation. The MyItemModelManager implements its interface and has two instance variables cpr of client logged in, which is passed from model to model. It also contains client, which is instantiated in its constructor.

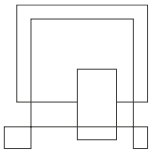
```
public class MyItemsModelManager implements MyItemsModel
{
  private Client client;
  private long cpr;
  private PropertyChangeSupport support;

  public MyItemsModelManager(Client client)
  {
    this.client = client;
    support = new PropertyChangeSupport( sourceBean: this);
    client.addListener( eventName: "return",this::returned);
  }
```

Code 34: MyItemsModelManager

In each method called from ViewModels, it is just calling another method in client.

```
@Override public List<MyItem> getitems(long cpr)
{
  return client.getitems(cpr);
}

@Override public boolean returnItem(MyItem item)
{
 return client.returnItem(item,cpr);
}
```

Code 35: getting items



5 **Test** (Everybody)

5.1 **Black-Box Test** (Everybody)

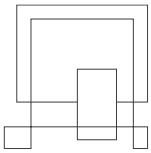
5.1.1 **Acceptance Test** (Jan Vasilcenko, Karrtiigehyen)

Acceptance testing was done to check whether the system meets the functional requirements. it was tested alongside the implementations of the functional requirements. for example, the first functional requirement, which states that the customer should be able to borrow items, was tested right after it was implemented.

The acceptance tests are shown in the table below. The user stories that match the number can be seen in the Functional Requirements section.

Functional requirements number	**Result (Successful or Failed)**
1	Successful
2	Successful
3	Successful
4	Successful
5	Successful
6	Successful
7	Successful
8	Successful
9	Failed
10	Successful
11	Successful
12	Successful
13	Successful
14	Successful
15	Successful

Table 4: The results of Acceptance test



All of the functional requirements were successfully implemented, except for the 9th requirement, which was alerting the customer when the due date for borrowed items was nearing. This will be discussed further in the Results and Discussion section.

5.1.2 Usability Test (Jan Vasilcenko, Karrtiigehyen, Nicolas Popal)

One of the biggest reasons for the production of this library system is to encourage more students to use the library system to borrow and reserve items from the library. Of those, the biggest hurdle is that students cannot easily navigate the user interface. To test whether this library system is intuitive enough to be used, usability testing was conducted.

Usability testing is concerned with the intuitiveness of the system, testing on users who had no prior experience with the system (Nielsen, 1994). The number of users to conduct the test with was chosen to be five, since it is said that the best result come from testing no more than 5 users (Nielsen, 2000).

The usability testing is meant to be iterative, meaning that it should be done multiple times throughout the implementation of the system for constant feedback. In this case, it was not done like that. The feedback acquired from the users can be considered as possible improvements that can be made in the Project Future section, and some of them will be discussed in the Results and Discussion section.

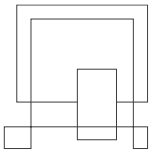
There were series of scenarios made for the users to follow. Then the users are asked to complete the proposed scenario and give any thought and criticisms upon it.

3. Edit user info

You can update your user info such as name, password, address and credit card info.

Figure 5: Scenario of the usability test

The above figure is an example of the scenario presented to the users. More scenarios can be found in Appendix J.



The users were not provided with a user manual prior to the testing. The user manual can be found in Appendix K.

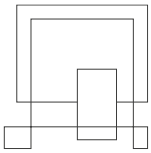
Part of the result of one of the testers is shown below. The full documentation of the usability test is in Appendix J.

Action	Result	Comments
Register	Success	• Registering is fast and nicely done • User doesn't like, that credit card information is mandatory
Login	Success	• User is missing function to log herself with pressing enter button
Edit Use info	Success	• User would like to see all her information and then have option to edit each of the information separately • User doesn't like, that changes are not visible
Search for items	Success	• User would like to search in all items (without changing categories) • Search should be case insensitive
Borrow Items	Success	• User would like to know information about successfully borrowed items (pop up window informing her about it) • Remove commas on the end of the string

Table 5: Result of one of the testers for the usability test

One of the biggest criticisms from many testers was that the testers wanted to have the option to edit their user info separately. Another one is to include some sort of notification to notify the users when they have completed an action such as borrowing an item. The last big criticism found from the usability test was that the searching of books was case sensitive. The testers said that it would be better if the searching was case insensitive.

These problems will be further discussed in the Results and Discussion, and Project Future sections.



5.1.3 Test Case Testing (Patrik Horny, Tomas Soucek)

The test case testing is made according to the use case descriptions. The results can be found in Appendix L.

Test case: Reserve items

Precondition: Customer has an account and is logged in. The selected item is already borrowed by other customer/s

Step	Action	Reaction	Result
1.	Select item to reserve	Confirmation window with details of the item is shown	Success
2.	Click „procceed“	Queue confirmation pops up	Success
3.	Click „yes“	Customer is put into the queue	Success

Figure 6: shows some examples of the test cases

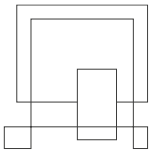
The results of the Test cases turned out positive, with the exception for the Edit User Info. Other than that, the system is fully functioning.

5.2 White-Box Test (Tomas Soucek)

5.2.1 Unit Testing (Tomas Soucek)

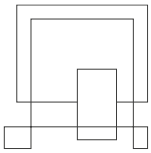
Unit testing’s purpose is to test the logical functionality of the system. The reason why we included unit testing was to ensure the functionality in case of future system’s improvements, enhancements and/or upgrades.

At the beginning of the unit of the test, it was realized, that there were two issues connected with this method of testing. First problem was the dependencies inside of the system itself. Most of the classes have no logic to be tested, since most of the classes serve only as a bridge, passing information from between themselves, so there



was only small number of classes to test, which could be improved. Second problem was the database dependency, as we heavily relied on the database existence, which is not acceptable for the unit testing. For this reason, it was decided to do the unit testing with the use of mocking, to simulate the database existence without the need of having the database access. This consequently led to a decision to separate Server and Client into two completely separate projects, to ensure independency of each of them (because at the beginning of the testing, we found out that Modules in a project are not treated as two completely separate projects, rather than “projects inside of a project”. which can break whole file structure and pathing inside of the project). The broken project structure, combined with the fact that mock testing requires many dependencies in repositories, led to a decision to use Maven Project Structuring tool in Server Project.

As the adapters included most of the system’s logic, the testing consists of only testing the adapters. However, not all methods inside of the adapters could be tested, as they included unknown classes and therefore would need to be tested using Black Box method instead of White Box, therefore these were omitted. The testing showed that there was a necessity of testing multiple same or very similar methods, which resulted in a lot of time spent on testing. The results were all successful, having 100% success rate, meaning that logic inside of the system is functional and correct. The results of the testing can be seen in Appendix Q. The code where the client and server sides are split into separate projects and the unit testing using Mockito is done can be found in Appendix R. This is where the test classes are as well.



6 Results and Discussion (Jan Vasilcenko, Karrtiigehyen)

The testing proved that most of the requirements were met. The usability test also informs that the users find the system to be intuitve and easy to navigate.

All of the non-functional requirements are also fulfilled.

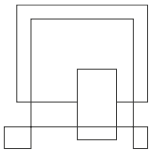
While the library system is functioning and working as intended, there are some flaws highlighted by the testings.

The acceptance tests showed that the system fulfilled most of the functional requirements, except for one. This functional requirement stated that the customer should be notified when the due date for borrowed items are nearing. This requirement failed due to the sparsity of knowledge on databases. A trigger was used to trigger when the due date for a borrowed item is near, but a trigger is triggered only when inserting, updating or deleting on a table in the database. Since it was neither of those, the database has to check periodically for nearing due dates. Nobody could present a solution to this problem, so the focus was redirected to other functional requirements.

In the test case testing, the only the case that failed was the edit user info test case. The problem was that the customer would have to change all of the customer's account information when the customer just wants to change one information. This was also a common critique from the usability test. The solution to this is to set the information from the textfields which are not empty.

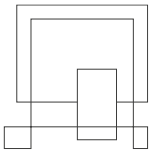
In the usability test, another common feature that confused some user was the searching. The searching for items is case sensitive, which the users are not aware of. Another big critique gained from the usability test was that the users would like a notification where the users perform an action. This could be solved by using alert boxes.

There are some hacks done in the system to get around some problems. For example, the automatic returning of items could not be automated by a trigger or similar stuff in the database. So, the list of borrowed items and their due dates was selected and



stored in an Arraylist, which was later checked in the ServerModelManager for due dates that were past the current date.

When customer logs in with the cpr, it is stored in the client-side model. When a model changes for each window, it passes down that cpr. This is a big flaw, because there are three instances of cpr stored in the system. Solution to this problem is to make one static class holding this information.



7 **Conclusions** (Jan Vasilcenko, Karrtiigehyen)

Overall, the project was a success on fulfilling the purpose of this project, which is to help customers borrow and reserve items, and manage their borrowed and reserved items with ease.

Some limitations were made, such as adding only one type of user which is the customer, to define the scope of the project.

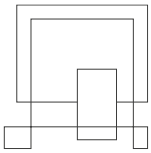
The project was build on the client-server architecture, where the server communicates with a database to store, get, update and delete information about customers and items. RMI was chosen instead of sockets, since not much control is needed for over the communication protocol in this project. For this reason, RMI suits the best for the library system.

Design patterns such as singleton, DAO, adapter and observer patterns are used, to solve common problems. They are also reusable and overall general solutions to some problems.

SOLID principles were used as guidelines for designing the system, so that dependencies are reduced, it is highly cohesive, context independent, and easy to divide into separate standalone components and modify. The DRY rule was also used to avoid code repetition, but some code repetition is still present. An alternative solution for this is to command design pattern.

The database is made to store the data, and normalized into third normal form so that there are no partial and transitive dependencies on the primary keys.

The acceptance test showed that most of the functional requirements were met. The usability test showed that the users found the library system to be easily navigatable and intuitive, which was one of the key problems the VIA's library had. Since this is addressed and solved, it could be said that this project is a success.



8 **Project Future** (Jan Vasilcenko, Karrtiigehyen)

There are multiple improvements that can be added in the future. Some of these improvements are dervied from the acceptance test and usability test. Others are to expand and add some features to improve the functionality of the system.

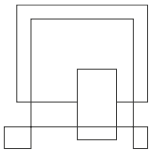
The requirement that failed in the acceptance test, where there should be an alert for the customer when the due date of an item is near, could be fixed.

The editing of the customer's user info was criticized for not being able to change only one info at a time in the usability test. This could be fixed by reworking how the editing user info is done in the system. Another main criticism from the usability test was that the searching for items using the search bar was case sensitive, and that it should search for results regardless of case sensitivity.

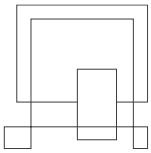
One of the biggest ommision from the system is feature of adding and removing items through the system instead of through the database. To solve this, a second type of user acting as a librarian for the system could be created. The librarian can add, modify and remove items from the system. The librarian can also manage some of the customers' account information.

Another feature that was omitted at the inception of the project was the fine feature. When a customer forgets to return an item that they borrowed, the customer should acquire a fine, which also prevents the customer from borrowing other items until the customers pays the fine. The groundwork for this was laid out such as the functionality of the fine in the database, and the inclusion of credit card info. But this was abandoned because of limited knowledge on databases.

From the usability tests, the users stated that viewing the user info to the customer can be another improvement, so that the customers can refer to their account information at any time. Another improvement that can be made is to cancel reservations. As of now, the customers cannot cancel their reservations. But this is another frequently asked feature that was missing in the usability test.

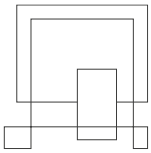


The design of the system could be improved so that it is completely complient on the SOLID principles and the DRY rule. For example, the *ServerModelManager* and the DAOs.



9 Sources of Information

1. Larman, C., 2004. *Applying UML And Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed. Upper Saddle River: Prentice Hall.
2. Connolly, T. and Begg, C., 2015. *Database Systems A Practical Approach to Design, Implementation, and Management*. 6th edition. Essex: Pearson Education Limited.
3. Tutorialspoint, 2020. *Data Access Object Pattern*. [online] Available at: <https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm> [Accessed 27 May 2020].
4. Refactoring Guru, 2020. *Command.* [online] Available at: <https://refactoring.guru/design-patterns/command> [Accessed 27 May 2020]
5. Java, 2020. *What is Java and why do I need it?*. [online] Available at: <https://java.com/en/download/faq/whatis_java.xml> [Accessed 28 May 2020].
6. JavaFX, 2020. *JavaFX*. [online] Available at: <https://openjfx.io/> [Accessed 28 May 2020].
7. Tutorialspoint, 2020. *JavaFX Tutorial - Tutorialspoint*. [online] Available at: <https://www.tutorialspoint.com/javafx/index.htm> [Accessed 28 May 2020].
8. PostgreSQL, 2020. *PostgreSQL: The world's most advanced open source database*. [online] Available at: <https://www.postgresql.org/> [Accessed 28 May 2020].
9. PostgreSQL JDBC, 2020. *PostgreSQL JDBC About* [online] Available at: <https://jdbc.postgresql.org/about/about.html> [Accessed 28 May 2020].
10. Nielsen, J., 1994. *Usability Engineering*. Unknown: Academic Press Inc.
11. Nielsen, J., 2003. *Why You Only Need to Test with 5 Users*. [online] Available at: <https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/> [Accessed 31 May 2020].



10 Appendices

A – Project Description

B – Use Case Descriptions

C – System Sequence Diagrams

D – Package Diagram

E – Low Detailed Class Diagram

F – Conceptual Model of the Database

G – Logical model of the Database

H – Source Code

I – UML Class Diagrams

J – Usability test

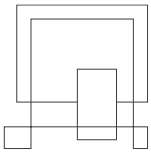
K – User Manual

L – Test Cases

M – Installation Guide

Q – Unit Test Results

R – Source code for Testing



Process Report

Library System

Client-Server

Names of Students:

Jan Vasilcenko – 293098

Karrtiigehyen Veerappa - 293076

Nicolas Popal - 279190

Patrik Horny – 293112

Tomas Soucek – 293103

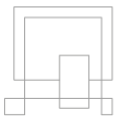
Supervisors:

Henrik Kronborg Pedersen

Joseph Chukwudi Okika

VIA University College

Bring ideas to life
VIA University College



Number of characters: 32582 including spaces

Software Technology Engineering

Semester 2

3 June 2020

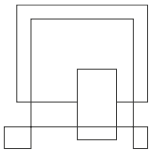
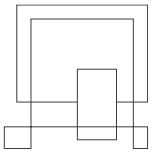
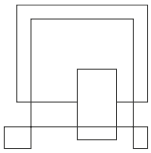


Table of Contents





1 Introduction (Patrik Horny)

We were introduced to the 2nd semester project on 12th of February where we were given lots of information about how the process of making the project is going to look like. We discussed about the examples of client/server systems, along with skills and knowledge required to create a functional and well-structured project.

On 13th of February we handed in our project proposal where we proposed to work on library system and a banking system. Our supervisors gave us a feedback about our project proposal and after some discussion we decided that we will develop a school library reservation system. This seemed like the most reasonable option, since none of us had full grasp of how banking systems work.

During following weeks, we started laying a foundation for our project, where we defined the necessary functionalities and the capabilities, as well as the requirements. We applied our knowledge gained in Software Engineering (SWE) where we learned a lot about how to structure our project and code in a more efficient way. The course Databases software engineering gave us a better insight on how to make a good functioning database and in Software Development with Java (SDJ) course we were taught things about making client/server system.

On the SWE classes, we were told to use a framework called SCRUM (Schwaber & Southerland, 2017) as our main framework to structure our group collaboration. This is done to keep the group more engaged to work on the project. This framework made sure that we did not lose focus on the project and that our time-schedule will be used efficiently.

Since we have been in school for like a month, we were kind of on our own with basically everything. Even if we attended most of the online classes, the online teaching was not the most convenient way of learning, even if teachers made their best effort of explaining things. This made us hesitated if we are doing the databases or SCRUM correctly, but despite the situation, we made our best effort to follow the guidelines and do what is expected of us.

2 Group Description (Tomas Soucek)

Our group consists of 5 people:

Jan Vasilcenko – Czech Republic

Karrtiigehyen Veerappa – Malaysia

Nicolas Popal – Czech Republic

Patrik Horny – Slovakia

Tomas Soucek – Czech Republic

Four members of the group already worked together during semester project one, with the 5th member joining due to disbanding and dissatisfaction with his former group. Our group consists of 3 Czechs, 1 Slovak and 1 Malaysian. Even though the Czech-Slovaks, who are very close due to their historical and cultural background make most part of the group, there we no problems in between the group as we get along very well with each other. Even though Czech and Slovak nationalities are quite close, this is not reflected in the Hofstede's Dimensions (as can be seen in the picture below).

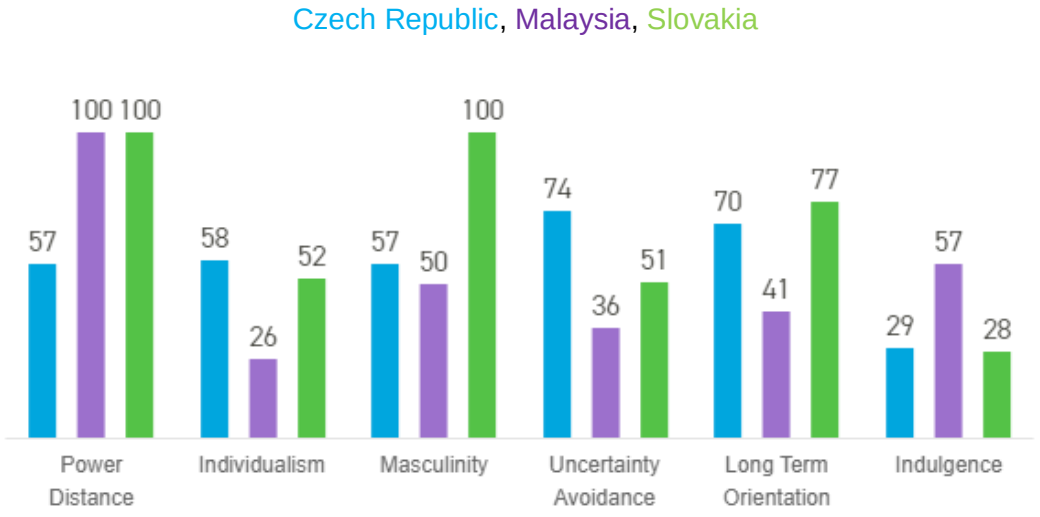
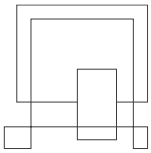


Figure 1: Hofstede's Dimensions



2.1 **Power Distance** (Tomas Soucek)

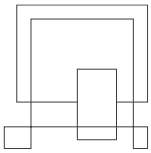
Power Distance measures inequalities between lower and higher ranked members of a certain hierarchy. (family elders vs. children, high management vs. employees) However, power distance is hard to measure inside of the project work as there are no big differences in hierarchy structure since all the members are equal with each other. Most probable scenario would be a person responsible for the project acting arrogant over others, distributing work without much contribution etc. However, nothing similar happened in our case, as we were all satisfied with our roles in our group.

2.2 **Individualism/Collectivism** (Tomas Soucek)

Individualism and Collectivism in project work relates on how people usually work as part of the group. In our case, even though Czechs and Slovaks have mediocre individualism on Hofstede's Scale, this was not our case as we all are quite social and work well as part of the group, tending more towards Collectivism.

2.3 **Masculinity/Femininity and Indulgence/Restraint** (Tomas Soucek)

Masculinity and Femininity represents the values of the group and society. Masculine society strives for success, heroism, and assertiveness, while Feminine strives for cooperation, caring and satisfaction in their work-life balance. Indulgence values enjoying life and having fun, while restraint focuses on social norms and following social standards. In the case of our group, we had a great balance between work and free time, so, we tend to lean more towards Femininity and Indulgence, as we value our time more than a potential success, so the Hofstede's Dimensions are not represented correctly, especially for the member from Slovakia in case of Femininity where Slovakia leans toward Masculinity, and Czechs and Slovaks in case of Indulgency, where both countries lean toward Restraint, as completely opposed to our group's nature. This work-life balance is probably inherited from Denmark's nature as Danes (and generally all Nordics) tend to value their work-life balance more than strive for success, so living in such environment might affect a person to some extent.

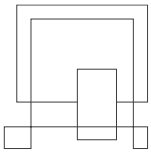


2.4 **Long/Short Term Orientation** (Tomas Soucek)

Long/Short Term orientation displays if the members of the team are focused on smaller tasks rather than long-term ones. In our case, I would say inexperience causes us to be oriented specifically on smaller tasks, than the long-term ones, so at first glance, it might seem that the Hofstede's Dimensions are not represented correctly. However, this might not be the real case.

2.5 **Uncertainty Avoidance** (Tomas Soucek)

Uncertainty Avoidance measures the amount of precautions made to handle unexpected situations, adapting to them, and preventing them. In our case, we did not encounter many unexpected situations, so it is very hard to estimate whether we represent the Hofstede's Dimensions correctly.

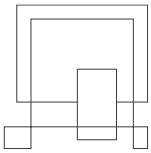


3 **Project Initiation** (Patrik Horny, Tomas Soucek)

When the time had come to choose the project topic, we had trouble with deciding what system we wanted to develop. The supervisors allowed us to choose any topic regarding client/server system, so naturally with that comes lot of options to choose from, unlike in the first semester, where we had only one topic to work on. The amount of possibilities was bigger since we could not correctly guess if the choice would properly fit with the client/server system requirement.

After a long debate, two main proposals were selected, the first one being the “Library System” and the other being “Banking System”. The Banking System was very interesting for all of us; however, we concluded that since we are students, we are accustomed with how the library works much more than how we would implement the Banking System. For the Banking System, we would need to have long debates about how to implement things, which we would not be sure if they were relevant for the system or not.

These conclusions proved to be true after consulting these proposals with our supervisors, who assured us that the Library System is the best option out of the two, and it was consequently selected as our main topic for the second semester project.



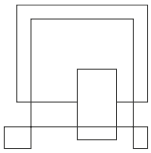
4 Project Description (Tomas Soucek)

4.1 Risk assessments (Tomas Soucek)

After entering initiation phase, we started by making a risk assessment table to pinpoint the riskiest scenarios that might occur during project development and to prevent them and how to react to them. Risk assessment table can be seen below.

Risk	Probability	Impact	Cause	Prevention/Mitigation	Responsible person	Response
Illness (Normal/Corona)	Medium	Medium/ High	Various/Corona epidemic	Healthy lifestyle/Self-prevention	Patrik Horný (SCRUM Master)	Redistributing work
Technical Failure	Low	High	Various	Cloud/Git backups, reliable equipment	Patrik Horný (SCRUM Master)	Replace malfunctioning equipment
Member Sabotage	Low	High	Loss of Motivation, Personal	Team-building, frequent progress checking	Patrik Horný (SCRUM Master)	Redistributing work, Supervisor meeting
Change in User Stories	High	Medium	Inexperience, Inadequate Analysis	Detailed analysis	Tomáš Souček (Product Owner)	Implement changes
Work Overload	Low	Medium	Inexperience	Careful planning, workload distribution acording to skills and needs	Patrik Horný (SCRUM Master)	Redistributing work
Misunderstanding Customer	High	High	Inappropriate communication with the customer	Communication with customer	Tomáš Souček (Product Owner)	Changing the product according to customer's needs

Table 1: Risk Assessment

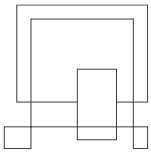


4.2 **Group Contract** (Tomas Soucek)

Next task was setting up the group contract. We have decided to build upon the Group Contract we have made in the first semester, changing only certain parts. Most prominent change was having a focus on every member understanding each part of the project (especially in code itself), so that everyone knows what is it doing, unlike in the first semester project, where some members did not fully understand the code or parts of the process report.

4.3 **Project Description Finalization** (Tomas Soucek)

After finishing both Group Contract and Risk assessment, we went to a Project Description finalization. As we already knew what we wanted to do, what the limitations of our software will be, who is the best at writing and summing these things up, finishing the Project Description was smooth. Although we had to return and change some minor things in the Project Description later in the project, the original version of it is almost indistinguishable from the final version.



5 Project Execution (Jan Vasilcenko, Karrtiigehyen, Patrik Horny)

Due to the given situation about the coronavirus, project execution started rather slowly, because we were not be able to meet each other in person, but when the exam period was about to come, the shape of the project started improving.

5.1 Project Development and Methods (Patrik Horny)

The project was developed during the whole second semester with the most progress made during May and additional weeks in the end the semester reserved for the semester project.

Our “personal’’ meetings were held on a Discord which was a great help, since sharing screen was far more practical than constantly sending files to each other.

As our main framework we have used SCRUM together with a web application called ClickUp, which helped us to be on a track with sprints and project backlog.

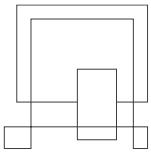
5.2 SCRUM (Patrik Horny)

Scrum proved to be a handy framework during the work on second semester project. At the beginning of each sprint, we held a discussion of what do we want to put into the sprint, what are the main points to put attention to and how much time it is going to take us to do it. In Sprint Retrospective, we realized our mistakes and put into consideration what needs to be fixed and improved.

We formed a group where we set a role for each member of the group with a condition to change roles if somebody does not like his role.

The roles were split as seen below:

- Patrik – Scrum Master – his role was to manage the whole development team and overall making sure that the whole process of making the project is going



smoothly. This includes setting up the meetings, managing project timeline and making sure that the team members have no problem with their given User Story.

- Tomas – Product owner – he was responsible for setting up a Product Backlog and making sure that it was fully understood.
- All members of the group – Development Team.

It was agreed on that the length of the sprint was set to be 3 days with the amount of work done per day set to be 8 hours / member. There was an exception with the last sprint, where we did not want to start a new sprint, but instead we prolonged the sprint.

5.2.1 Product Backlog (Patrik Horny)

The product backlog is like a wish-list from the product owner that had to be written before starting first sprint. This backlog served us as an instruction manual of what needs to be done and in what time.

It was decided that points assigned to each user story were going to be based on approximate time of finishing user story.

It is important to mention, that we decided not to complete all the user stories that were in a backlog. We agreed that we will focus only on the most essential user stories that are necessary to complete and if the time permits, we will try to finish the ones that did not have high or critical priority. The red flag was set for the tasks for the critical priority, yellow flag for the high priority and blue for the low priority.

The product backlog is included in the Appendix N.

Critical priority:

1. As a customer, I want to borrow item, so that I can use them.
2. As a customer, I want to reserve item, so that I can borrow them later.
3. As a customer, I want to view items, so that I can borrow them.
4. As a visitor, I want to register myself as a customer, so that I can use the library system.

Figure 2: An example of the Product Backlog

5.2.2 **Sprint planning** (Patrik Horny)

When the product backlog was finished, we started to plan sprints. We were not exactly sure on which and how many User Stories to put into the sprints, so we decided that we will have fewer tasks to complete which we considered as essentials for the system to work, like database. As the time went on, we learned all the things that we needed for the project, like how to connect Java and Database. With the gained knowledge we were more confident into putting more tasks onto our shoulders and so we put more tasks into the sprints.

Since none of us have experience with SCRUM, planning sprints did not go as expected especially at the beginning. We did not put many tasks into the sprints because of our lack of experience and much more time needed to understand what was explained on the online lessons, especially with working with database which took most of the time. This resulted in tasks being pushed into the next sprint. On the other hand, we thought it was a good thing to focus on less task so they could be done properly.

In general, sprint planning was quite useful. Instead of everyone rushing to do work, spending time on planning resulted in more work done and overall better flow of work.

The planning for each sprint can be found in the Appendix O.

Sprint 3

Sprint planning

Date: 11:30, 20.04.2020

Present: Everyone

Duration: 5 minutes

Recap: Creating a table so user can view items. Add more data to the database. Finish assigning name in the main window for the particular user. Start connecting java and database. Apply some CSS style so the GUI looks more appealing.

Figure 3: An example of Sprint Planning

5.2.3 Sprint backlog (Patrik Horny)

When making a sprint backlog, we did not start working on user stories as they are in order in product backlog, but instead we made a sprint with user stories that share some similarities, like registration and login, reserving and borrowing or making a queue and alerting when being in queue. Then for each user story we would make a list of subtasks which are needed for the story to be completed. When all the subtasks were done, then we would consider the user story as completed. Some user stories have similar subtasks so later we did not include subtasks in all the user stories, because of the repetition. Sprint backlog can be found in Appendix O.

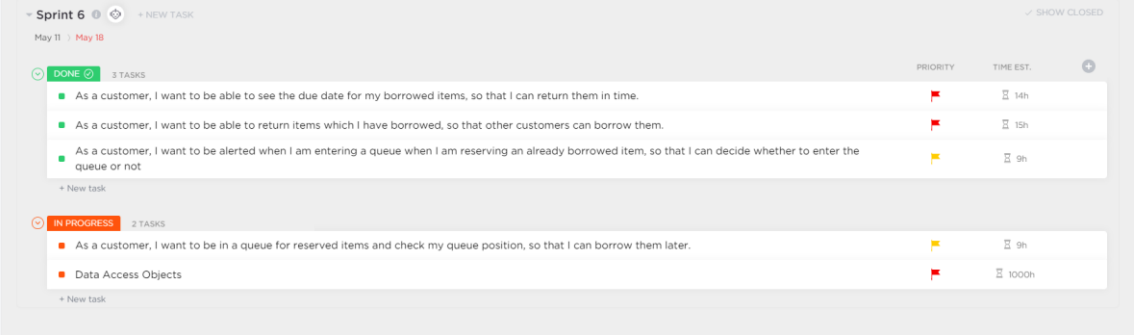


Figure 4: An example of Sprint Backlog

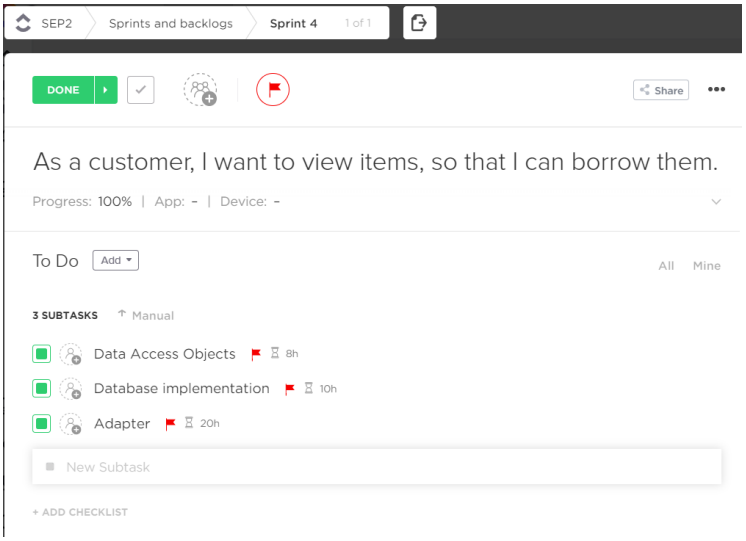


Figure 5: An example of a user story meant to be fulfilled in a sprint

5.2.4 Burndown Chart (Patrik Horny)

After we were done with sprints, we made a burndown chart that tracked our progress. In the chart below you can see that we were behind the schedule, but in the end, we were able to catch up. The burndown chart file can be found in the Appendix P.

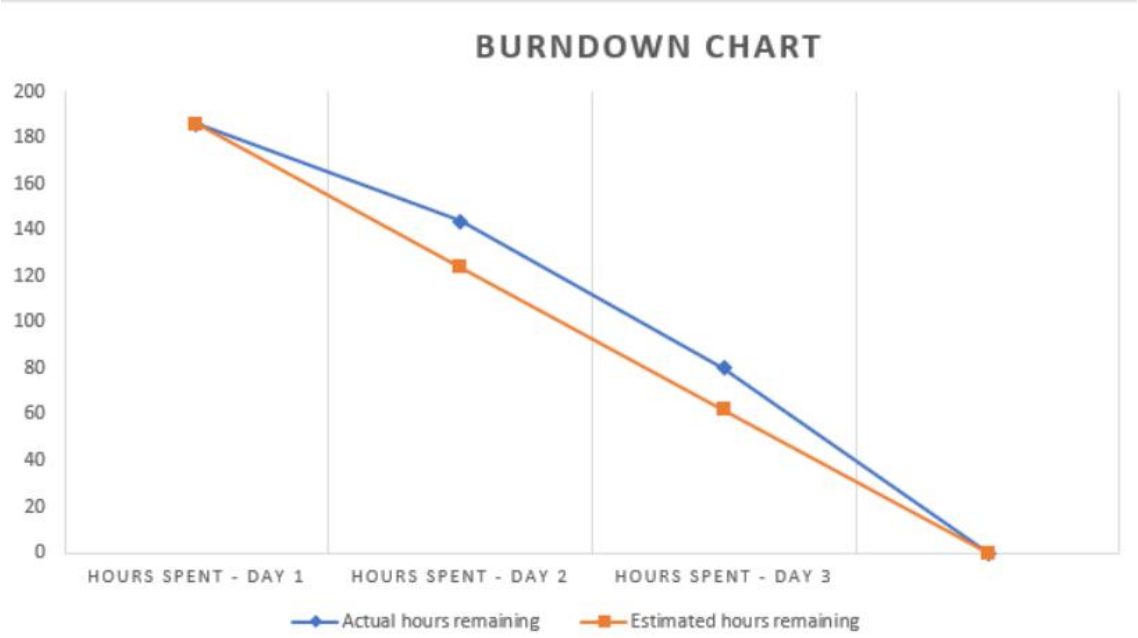
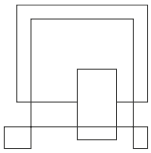


Figure 6: Burndown Chart

5.2.5 Sprint Review and Retrospective (Patrik Horny)

In the sprint reviews we talked about what had been accomplished in the sprint, what is working, if there are any problems and what we will need to focus in the next sprint. Reviews served us as form of a progress checker where we could see how much work was done.



Sprint review

Time: 14:00, 11.05.2020

Present: Everyone

Duration: 5 minutes

Recap:

- Configure GUI elements so they are shown for a particular item
- Create queue
- View borrow bug where user can borrow item twice
- Multiple people can borrow item bug

Figure 7: An example of Sprint Review

In the retrospective we looked at good and bad sides of the sprint, what was well executed as well as what needs improvement. Retrospective helped us to reflect what happened in the sprint, and what needs improving.

Sprint retrospective

Time: 14:00, 11.05.2020

Present: Everyone

Duration: 15 minutes

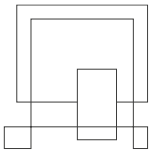
Recap: Managed to make a borrowing work, so nobody can borrow an item if it was borrowed by another customer. A queue was made for the reservation. Encountered a bug where customer can borrow an item two times. Database was needed to be improved so the bug was fixed.

Figure 8: An example of Sprint Retrospective

5.2.6 **Daily SCRUM** (Patrik Horny)

At daily SCRUM we discussed what we will be doing, if there are any immediate problems from the last session.

If we are being honest, these sessions did not really have much depth and lasted just few minutes. Also, we found ourselves often not having a daily scrum, but rather texted a message if there are any problems. Since our group functions much better in personal group meetings, we struggled to put importance to the daily SCRUM.



If we could have meet in person, of course, it would be taken much more seriously.

Daily SCRUM 3

Time: 15:00, 11.04.2020

Present: Everyone

Duration: 8 minutes

Recap: Ran into the problem with Unicast. Work on new GUI. Database updated with data of books, games, movies.

Figure 9: An example of a daily Scrum meeting

The sprint reviews can be found together with retrospective and daily SCRUM in the Appendix O.

5.3 **Unified Process** (Patrik Horny)

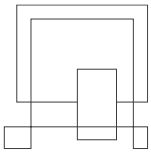
Since SCRUM had no defined development techniques, Unified Process was used. It was used in each sprint, where we would go through the Elaboration, Construction and Transition phases. This gave us a guideline of what should be done in each sprint. Because of the unified process, the functional requirements that were supposed to be done in the sprints were working with no hiccups, and documentation of the implementation of the functional requirements were also done.

5.4 **Critique to the project** (Jan Vasilcenko, Karrtiigehyen, Patrik Horny)

We think that it is important to talk about things that have not gone as well as we hoped for and to talk about the design flaws of the project. This will hopefully make future projects free from these flaws, even though it is sometimes easier said than done.

5.4.1 **General assessment and critique** (Jan Vasilcenko, Karrtiigehyen, Patrik Horny)

We were able to complete all the user stories except for one which was alerting when the due date was coming. This was due to time sparsity.



5.4.2 SOLID Principles (Jan Vasilcenko, Karrtiigehyen, Patrik Horny)

In this semester we were introduced to the SOLID Principles which we were supposed to apply to our project. These principles served as a guideline how to construct the system and make it flexible and reliable.

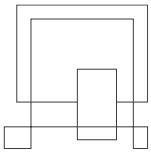
We found ourselves violating these principles at times. A good example of violating the Single Responsibility Principle can be found on the server side. In the server model manager, the class did not have a single responsibility. It took care of all the requests from clients regardless of the type of the request. As a result of this, cohesion of code is reduced. Possible solutions of fixing this issue are to split responsibilities of the server model manager into multiple classes, which should be responsible for a single task.

The next principle we violated was the Interface-Segregation Principle. An example of this would be the server model interface on the server side. Interfaces should fit exactly what the client needs. If the single responsibility principle is followed and the server model manager is split up into multiple classes, then the interface that the classes implement should be split up into multiple interfaces so that it suits what the client requests.

Other than that, other principles were not violated and if they were, they were violated in a less severe manner than Single Responsibility and Interface-Segregation principle.

5.4.3 DRY Rule (Jan Vasilcenko, Karrtiigehyen, Patrik Horny)

Applying DRY rule to the semester project was our goal from the start. But applying the DRY rule in implementation was harder than expected. For example, a lot of the DAO concrete classes have a code repetition. Code repetition was tried to be nullified by making some reusable methods that can be reused in the same classes. But this did not entirely eradicate code repetition. A good solution to this could be to use the command design pattern.



6 Personal Reflections (Everybody)

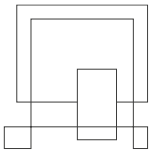
In this section we will discuss our opinions about the whole process of making the second semester project as well as other matters.

6.1 Jan Vasilcenko

Working on this project for me was a big experience, if I just compare it with my project from first semester. I learned how to setup server-client application with Sockets or RMI and to follow SOLID more closely and to include all kinds of design patterns. I also learned how to work with database, which I had the most problems with, but in the end, I managed to operate and add features to it. I did not learn only about programming, but also about designing an application, RMI and database. Still, I have a lot to learn in case of testing, so that the program is more testable and working with database. In making of this project we encountered countless problems, which we solved one by one. We tried to make the system as flexible as possible, so that we can modify and add features. For future of this project there are also a lot of expansions, which can be added as fines and second client window as librarian and server UI. Thanks to my awesome team, communication was never problem and we cooperated on almost everything, everyone bringing up some idea to the project. Idea of SCRUM meetings and sprints was good for me because I like to cooperate and solve problems in team, so that we can see it from different points of view.

6.2 Karrtiigehyen Veerappa

Starting SEP2 was easier compared to starting SEP1 since I have gained some experience on how projects should be conducted. It was also more exciting this time around since we were allowed to choose what system we wanted to do. This freedom was scary at first, but with the guidance of our supervisors this freedom to do what we wanted to do quickly became a fun playground to play in. My group and I had more fun and flexibility in terms of how we implemented the system. The only requirement that had to meet was that the system should be a Client-Server system.



The group that I joined for this project consists of the same members from the previous SEP, since I felt like I could voice out my opinions and felt that everybody was on the same wavelength. There was a new addition to the group as well, making us a 5 people group. This was also an upside since a lot of the work can be split between the 5 of us, meaning more work can be done in less amount of time.

While SCRUM was confusing at first, it became a great way of organizing and motivating me to work on the project. Each sprint was a had a purpose, and the burndown chart made me visualize the amount of work done and that is left. Unified process is also a marked improvement from the Waterfall method we had to use from SEP1. I was constantly going back and making changes, which is a feature not really present in the Waterfall method, but a defining feature in the Unified Process.

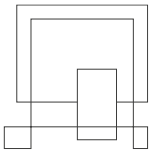
The implementation of the code also became much easier compared to the last semester, since this time we used Separation of Concerns, which made it easier for multiple people to take part in coding. SOLID principles were also a great way of designing the system since it made testing and other matters simpler.

The main problem that I encountered in SEP2 was the closing of the school due to COVID-19. This meant that all group meetings had to be done virtually. It was difficult at first, but I got used to it later. This also meant that the supervisions from our supervisors was virtual, but the supervision was not lacking in any way.

In a nutshell, SEP2 had some problems, but overall was quite satisfactory for me. I learned a lot about software development, in terms of programming and group work.

6.3 Nicolas Popal

On the start of the semester 2nd project was introduced. I was pleasantly surprised, that we can choose theme of the project by ourselves. We thought about 2 themes – bank system and library system. Both were introduced to Henrik and Joseph and they let us work on our favourite – library system.



As time went, we had more and more knowledge and ideas how we going to work on the project, what we will use and so on. Then corona hit Denmark and hard times started. I always struggled to study from home and that's why I always studied in the school premises, but now it was impossible for me as school was closed. I think it affected my work on the project, even though everything was handed in in time.

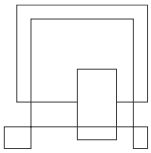
Communication was a key in our group. When VIA was still open, we were regularly meeting there, but after locking down it wasn't possible, so most of the time we were meeting on Discord. On the discord the meeting were okay, we shared our screens so all of us could work on the project and everything went smoothly.

In summary I am glad that this semester is almost behind us and I am looking forward to the 3rd semester when I can use VIA premises fully as before.

6.4 Patrik Horny

I am satisfied with how the project turned out. Even if the whole process took more time than anticipated, we managed to get it done. I think that even if we weren't able to meet in person, we were able to have a good communication with dividing the work on the project. Of course, it did not have the same effect and result as if we meet in person, but we somehow managed. One of the things I learned is how hard it is to manage a group without being with them in personal contact which I think changes a little bit the workflow on the project.

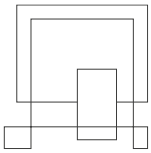
This semester we were introduced to the SCRUM framework to manage our workflow. I think that SCRUM is a good tool for group projects, but we were not able to benefit from it as much as we hoped for. This was due to fact that there were a lot of questions regarding how the whole framework works and if we are utilizing it correctly. Also, documenting everything seemed very tedious and time consuming together with rules to learn which were sometimes not easy to understand. I do not want to say I didn't enjoy SCRUM, because I did, but with only Discord meetings we weren't able to use SCRUM to the full potential.



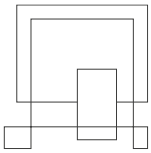
Overall, I think of the second semester project as an interesting experience where I learned a lot of new things.

6.5 Tomas Soucek

For me, this semester was far more difficult than the previous one. The difficulty of some tasks, to my surprise, increased exponentially, especially in Java, which caught me unprepared completely unprepared. Apart from that, personal issues started at the end of December and escalated at the end of January, which subsequently rendered me unable to properly work for more than half of the entire project work. After things started to settle down for me, Corona virus started to spread across the world and soon hit Denmark and many other countries in full strength, leading to a school closure within a blink of an eye. Corona struck us all completely unprepared. Many people started to leave Denmark and before I had a chance to react, I was left isolated from everyone, sticking only to an online communication and online relationships, which I am luckily accustomed to due to long-time gaming experience. Unfortunately, not only due to this situation, my personal issues returned, having overwhelmed me and put me in far worse condition than before. Working remotely seemed impossible for me as physical attendance is the key factor in my motivation and performance, and not meeting the lecturers in person was especially difficult for me, and it was difficult for me to keep up the pace from remote place and online meetings and lessons could not compensate the physical ones fully enough. Although the project was going on far better than in the previous semester, in which our previous group failed, my state of mind made me unable to work properly in group as far as I could. However, due to support and understanding of my colleagues, I was able to overcome the burdens and work at least that much how my state allowed me. Throughout the April and especially May, the project started to shape up well, and although I did not commit as much I would wish to, I think that for myself I can say that we accomplished everything we wanted and I am happy to have such a great groupmates. Also, on the positive note, there were many things that turned out very well. SCRUM framework turned out to be, in my opinion, one of the best tools to projects of any kind and once fully understood, I find it as a necessity for our future projects. Even though we had troubles using its full potential, which I think is the result

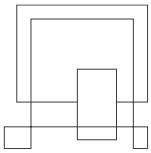


of inexperience, I think we all find it very useful. Also, we started using more complex tools by ourselves to make project development easier, which will be very beneficial for us in the future.



7 **Supervision** (Tomas Soucek)

For this semester, our project supervisors were Henrik Kronborg Pedersen and Joseph Chukwudi Okika. Their supervision was very good. The questions we asked were answered clearly and supervisors were very helpful when solving our problems. The biggest issue in the supervision was the response time, when we sometimes had to wait for number of days to even get a reply. However, due to a coronavirus situation and the fact that some of these questions were asked during the semester project work weeks, when teachers usually get large number of questions and requests from other semester project groups, it is understandable that the response time was slower than we expected.



8 **Conclusions** (Everybody)

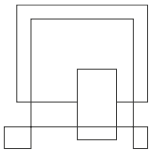
In this semester project, we learned how to construct a client-server system with a connection to the database. From this, we learned how to properly apply various design patterns, and how to structure our work by using the SCRUM framework.

SCRUM was a good tool to keep progress of work and to structure the work left to do. It also allowed us to reflect on how work was done in the Sprint Retrospectives, which showed us how to improve our management of work for the future sprints. The burndown chart was used to properly visualize the work done and left, which led us to better time management. While SCRUM is a good framework to manage the workflow, we could not take full advantage of it. This is because of lack of physical meetings due to the pandemic. All of the meetings and development was done remotely through virtual meetings, which made the SCRUM framework more of a hindrance than a useful tool to structure our work.

Unified process was used as the development technique since SCRUM does not have any defined development techniques. The Unified Process is better compared the Waterfall method used in the previous semester. The Unified Process reflects the real world better since a lot of the phases overlap and previous phases are often revisited and redone.

While developing the software, we constantly kept in mind that the SOLID principles and the DRY rule should be followed, but they were violated in some cases because of our inexperience dealing with these principles.

In a nutshell, the purpose of this semester project was accomplished, with the use of SCRUM and Unified Process despite having difficulty communicating about the project as a result of the pandemic. Virtual meetings were held constantly to fill in the void of not having physical meetings. This taught us the value of meeting in person to discuss problems and cooperate on our project.



9 Sources of Information

1. Schwaber, K. & Southerland, J., 2017. *The Scrum Guide*. [pdf] Unknown: Creative Commons. Available at: < https://www.scrumguides.org/docs/scrumguide/v2017/2017-Scrum-Guide-US.pdf> [Accessed 31 May 2020].