Nompumelelo Mtshatsheni – BXMDLL7W9

ITOO311 – Exercise 2

May 27, 2020

Question 1

1.1

- Structure diagrams show the things in the modeled systems, in a more technical term as well as different objects within a system.
  - Class diagram: main building block of any object-oriented solution. It shows the classes in a system, attributes, and operations of each class and the relationship between each class.
  - Component diagram: displays the structural relationship of components of a software system.
  - Deployment diagram: shows the hardware of your system and the software in that hardware
  - Object diagram: sometimes referred to as Instance diagrams are remarkably like a class diagram.

- Whereas, the behavioral diagrams show what should happen within the system, by describing how the objects interact with each other to create a functioning system
  - Use case diagram give a graphic overview of the actors involved in a system, different functions needed by those actors and how these different functions interact.
  - Activity diagram represent workflows in a graphical way. They can be used to describe the business workflow or the operational workflow of any component in a system.
  - State machine diagram are like activity diagrams, although notations and usage change a bit
  - Sequence diagram: in UML show how objects interact with each other and the order those interactions occur.

1.2

- Step 1 – Define states: capturing the behavior of the subject-object through modeling these various states and transitions between them.

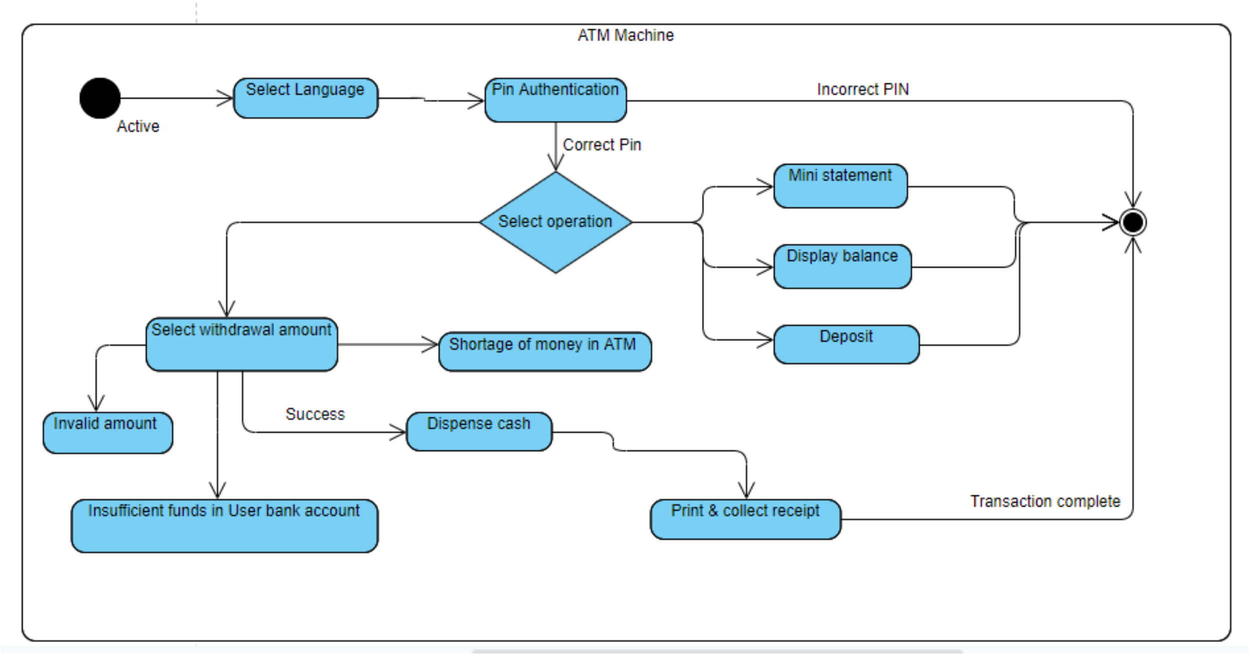
- Step 2 – Describe states: drawing states in the diagram, you probably want to elaborate on what the states are about for others to understand a little better. To do that, we can choose to add documentation to those states.

- Step 3 – Draw transitions: once finished drawing the states, let us turn our attention to describing the relationships between them. To depict a transition between two states, we draw a directed line from the source to the target state

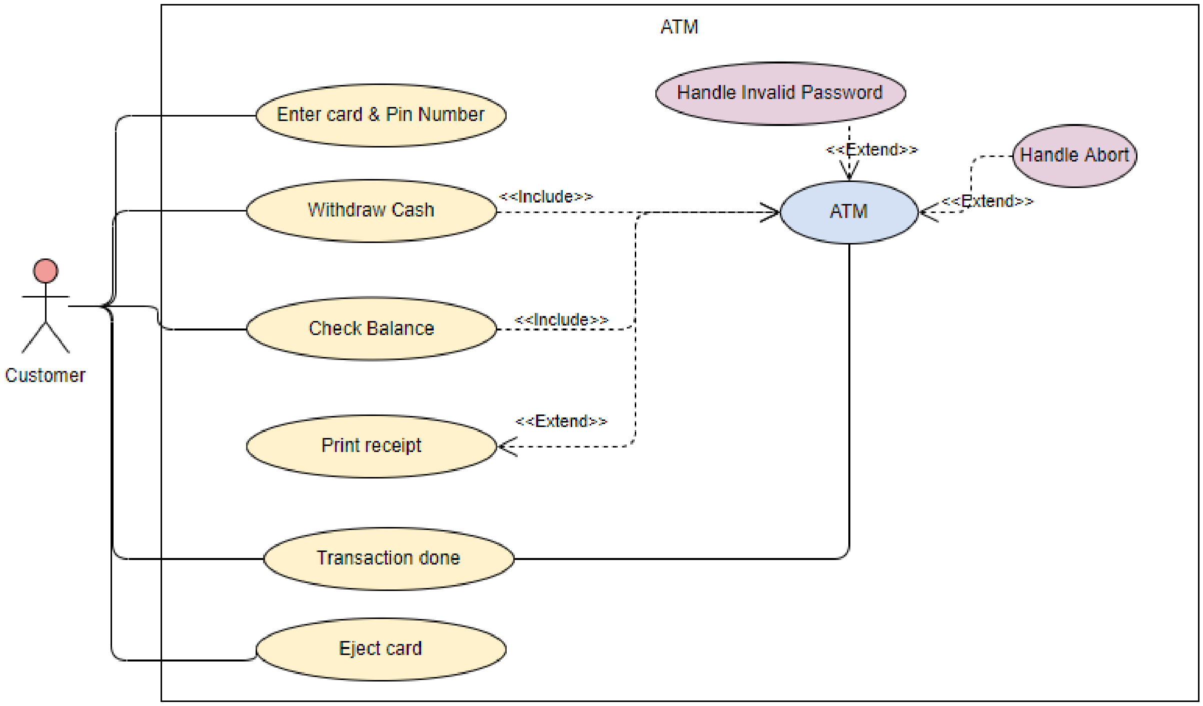
- Step 4 – Define transition triggers: A transition from one state to another takes place when the designated trigger event fires. A trigger event can be an event from the external world or simply a user's interaction.

- Step 5 – Define guard conditions: a transition would not be appropriate, although the same trigger event fires. e.g, let us just say that someone can withdraw funds only when there are sufficient funds available in the account. So, it would be good for us to impose a constraint to check against that before allowing the transition to happen. In UML, this constraint is called a guard condition

1.3



1.4



Question 2

2.1

The Waterfall Model was the first process model to be ever introduced. It is also mentioned to be a linear-sequential life cycle model. It is remarkably simple to comprehend and make use of. Each phase must be completed fully before the next phase could commence. This kind of software development model is fundamentally used for the project which is small and there are no uncertain requirements.

The advantages of a waterfall model are as follows; it is straightforward and comfortable to comprehend and utilize. It is easy-going to operate expected to the inflexibility of the model – each phase has certain deliverables and a reassessment procedure. It works well for less significant projects where obligations are undoubtedly characterized and very well understood.

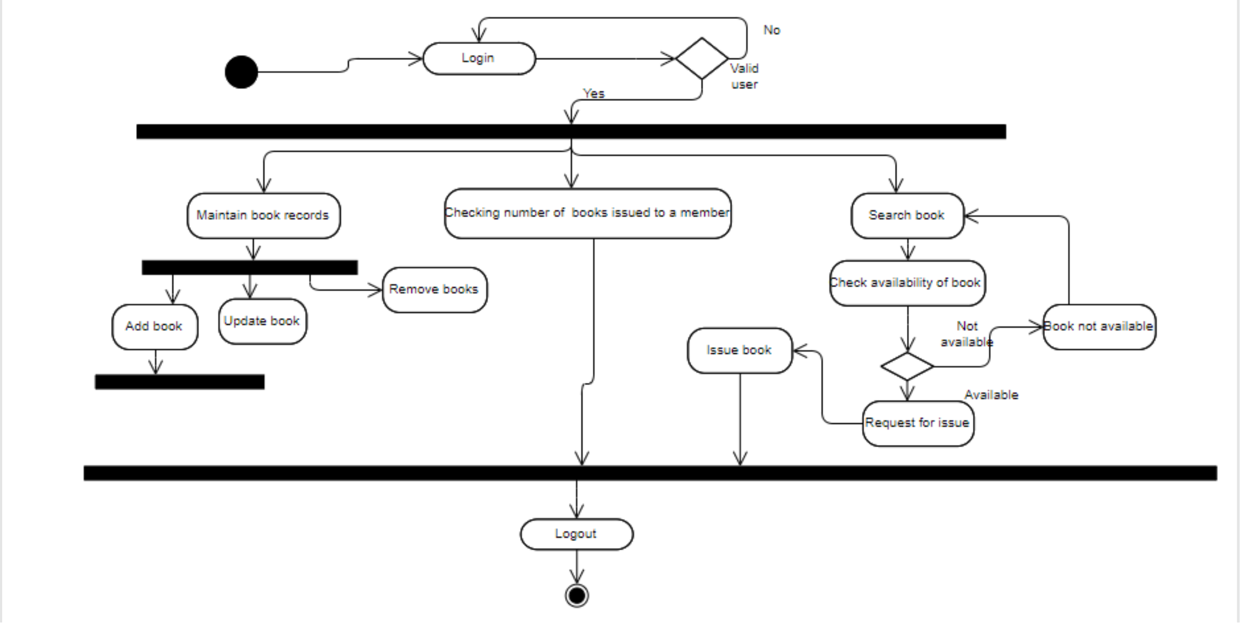
Whereas the disadvantages of the waterfall overlook at the application are in the testing stage, it is exceedingly difficult to go back and change something that was not well-thought-out in the concept stage. The extreme sums of risk and ambiguity. It is not a good model to be used for complex projects. It is highly unsuitable for projects that have a high level of requirements with a moderate risk of changing.

In short, the model is having a smaller amount of customer interaction is participating during the enhancement of the product. When the result is ready then only it can be displayed to the end-users. After the creation is established and if any failure occurs then the rate of fixing such issues is extremely high, because we need to update everything from the document till the logic.

Hence, the prototype model is an alternative of suspending the obligations before a layout or coding can progress, a disposable prototype is constructed to comprehend the constraints. A prototype is established centered on the presently well-known requirements. It is a software development model. When using a prototype, the user can get off the system, ever since the collaborations with the prototype can allow the user to better comprehend the requirements of the required system. Prototyping is an appealing concept for complex and huge systems for which there is no handbook procedure or accessible system to assist verifying the requirements.

Although the advantages of a prototype model; the clients are aggressively engaged in the development. Most errors will be detected in an earlier stage. It has quicker user feedback when available most importantly in giving better solutions. Whereas, its disadvantage indicates in employing and refurbishing ways of a developing system. This model will increase the complexity of the systems as the scope will be beyond the essential plans. The unfinished product may trigger the application to not be utilized as the full system was intended. It tends to have an incomplete analysis procedure.

2.2



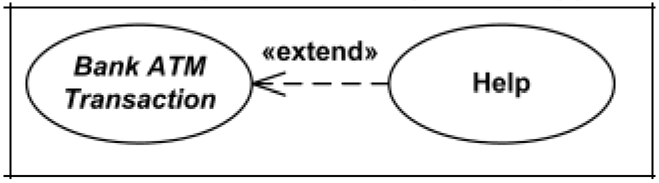
Question 3

3.1

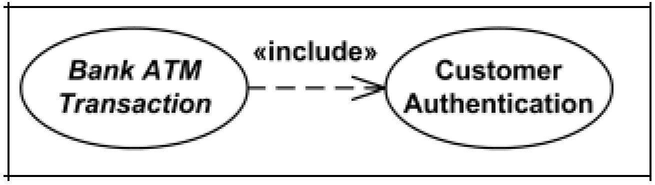
- It is reliable for customers
- It can accommodate more than 10000 people for booking a ticket
- Able to deliver on their services

3.2

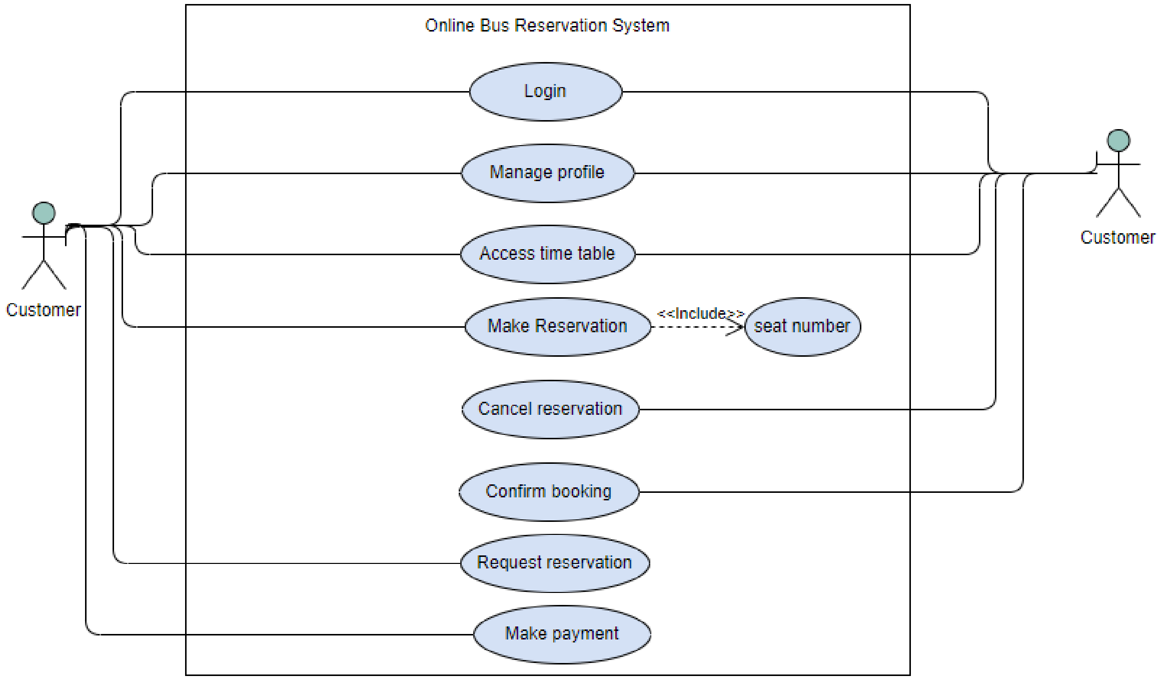
- Extend
  - it has at least one explicit extension location
  - could have an optional extension condition
  - the extending use case is optional, supplementary



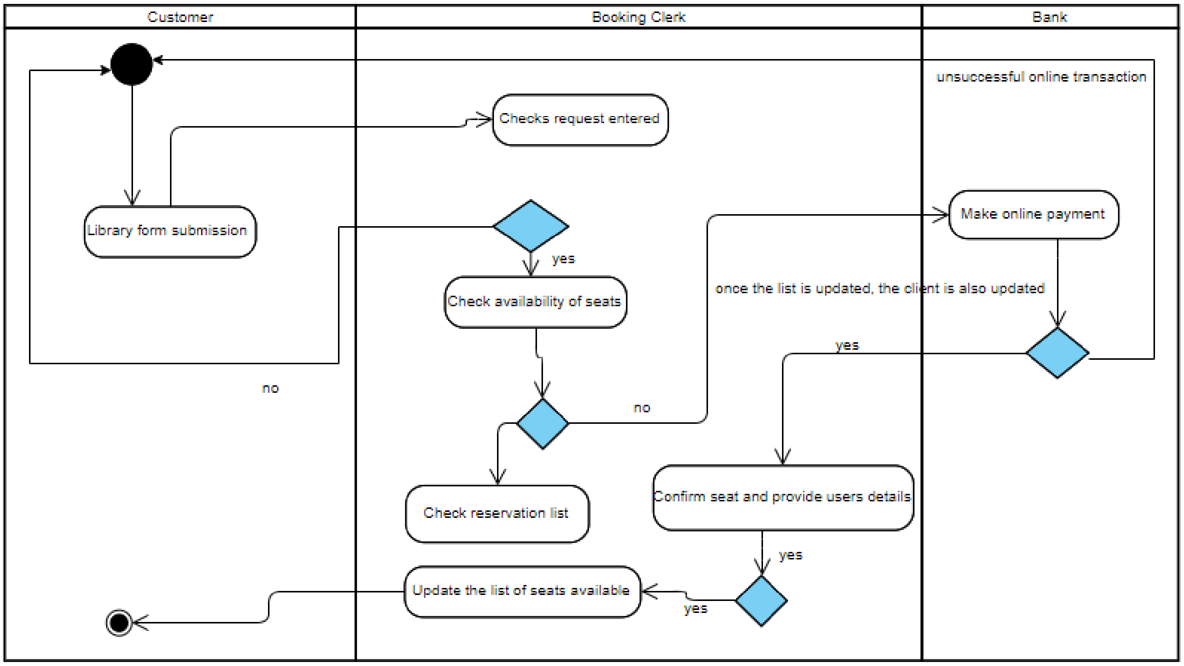
- Include
  - it includes use case required, not optional.
  - there is no explicit inclusion location but is included at some location
  - no explicit inclusion conditions



3.3


Online Bus Reservation System
Login
Manage profile
Access time table
Make Reservation
<<Include>>
seat number
Cancel reservation
Confirm booking
Request reservation
Make payment
Customer
Customer

3.4


Customer
Booking Clerk
Bank
unsucessful online transaction
Checks request entered
Library form submission
yes
Check availability of seats
Make online payment
once the list is updated, the client is also updated
yes
no
no
Check reservation list
Confirm seat and provide users details
yes
Update the list of seats available
yes

Question 4

4.1

- **Single -responsibility principle**
  - S.R.P for short. A class should have one and only one reason to change, meaning that a class should have only one job.

- **Open-closed principle**
  - Objects or entities should be open for extension, but closed for modification
  - If you have a general understanding of OOP, you probably already know about polymorphism. We can make sure that our code is compliant with the open/closed principle by utilizing inheritance and/or implementing interfaces that enable classes to polymorphically substitute for each other

- **Liskov substitution principle**
  - Is probably the hardest one to wrap your head around when being introduced for the first time
  - In programming, the Liskov substitution principle states that if S is a subtype of T, then objects of type T may be replaced (or substituted) with objects of type S.
    The formula mathematically as:
  - Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T
  - generally, it states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program

- **Interface segregation principle**
  - Hence, this principle is easy to comprehend. In fact, if you are used to using interfaces, chances are that you are already applying this principle.
  - When programming, the interface segregation principle states that no client should be forced to depend on methods it does not use.
  - Put more simply: Do not add additional functionality to an existing interface by adding new methods.
  - Instead, create a new interface and let your class implement multiple interfaces if needed.
  - A client should never be forced to implement an interface that it does not use, or clients should not be forced to depend on methods they do not use

- **Dependency inversion principle**
  - When programming, the dependency inversion principle is a way to decouple software modules.

- Entities must depend on abstractions not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions
- Using this principle, we need to use a design known as a dependency inversion pattern, most often solved by using dependency injection. The dependency injection is a huge topic and can be as complicated or simple as one might see the need for, is used simply by ‘injecting’ any dependencies of a class through the class’ constructor as an input parameter.

4.2

Cohesion	**Coupling**
The indication of the relationship within the module.	It is the indication of the relationships between modules.
It shows the modules relative functional strength	It shows the relative independence among the module
While the construction cohesion strives to be higher, a component in the cohesive module focus on a single task with the interaction with other modules of the system	When designing you should strive for low coupling namely the dependency between modules should be less
Has a natural extension of data hiding	It makes private fields, having private methods and non-public classes provide loose coupling