2020

Software Requirements Specification

CS 2212B – DELIVERABLE 2
TEAM 1

ALI AL-MUSAWI AALMUSAW@UWO.CA
JINGYI WU JWU583@UWO.CA
JUNHAO WANG JWAN992@UWO.CA

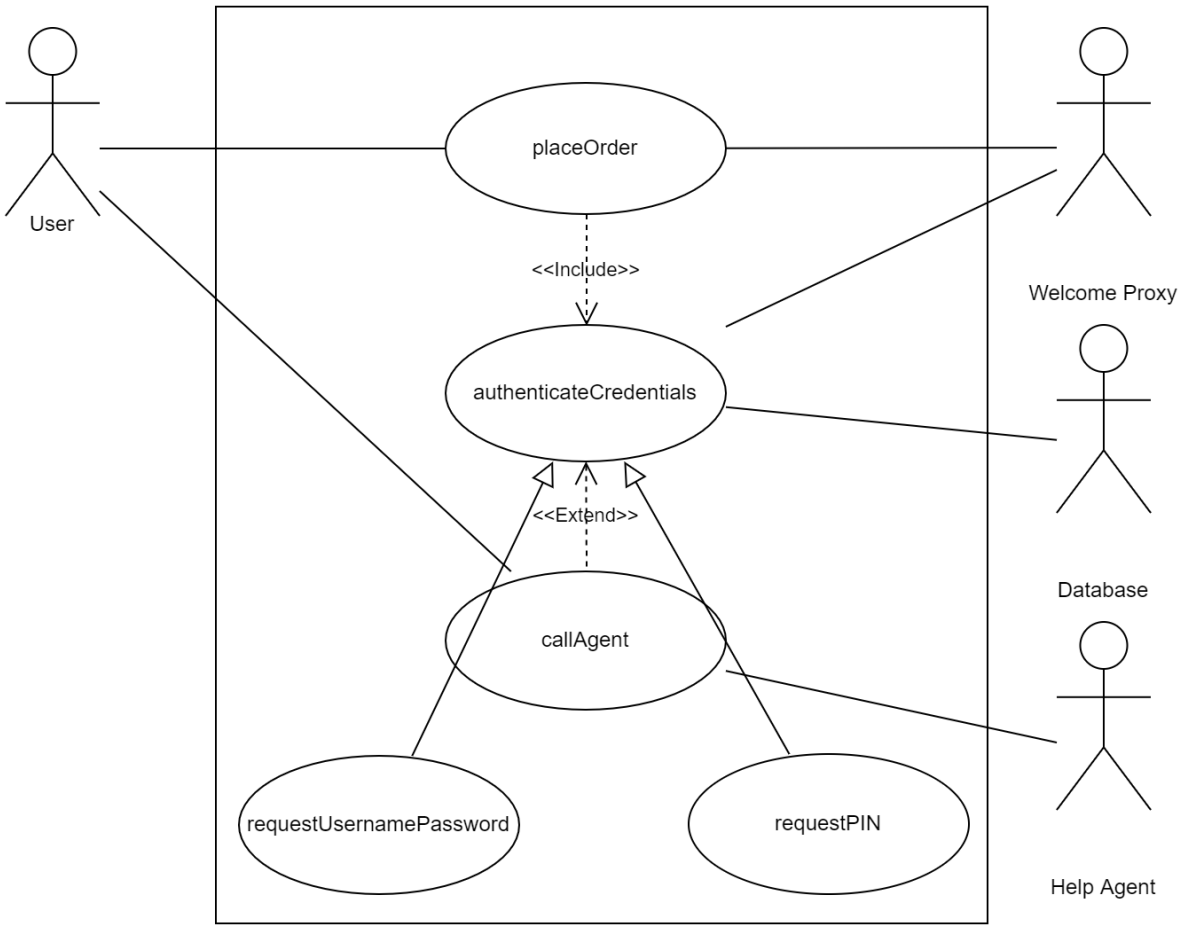
Table of Contents

Introduction

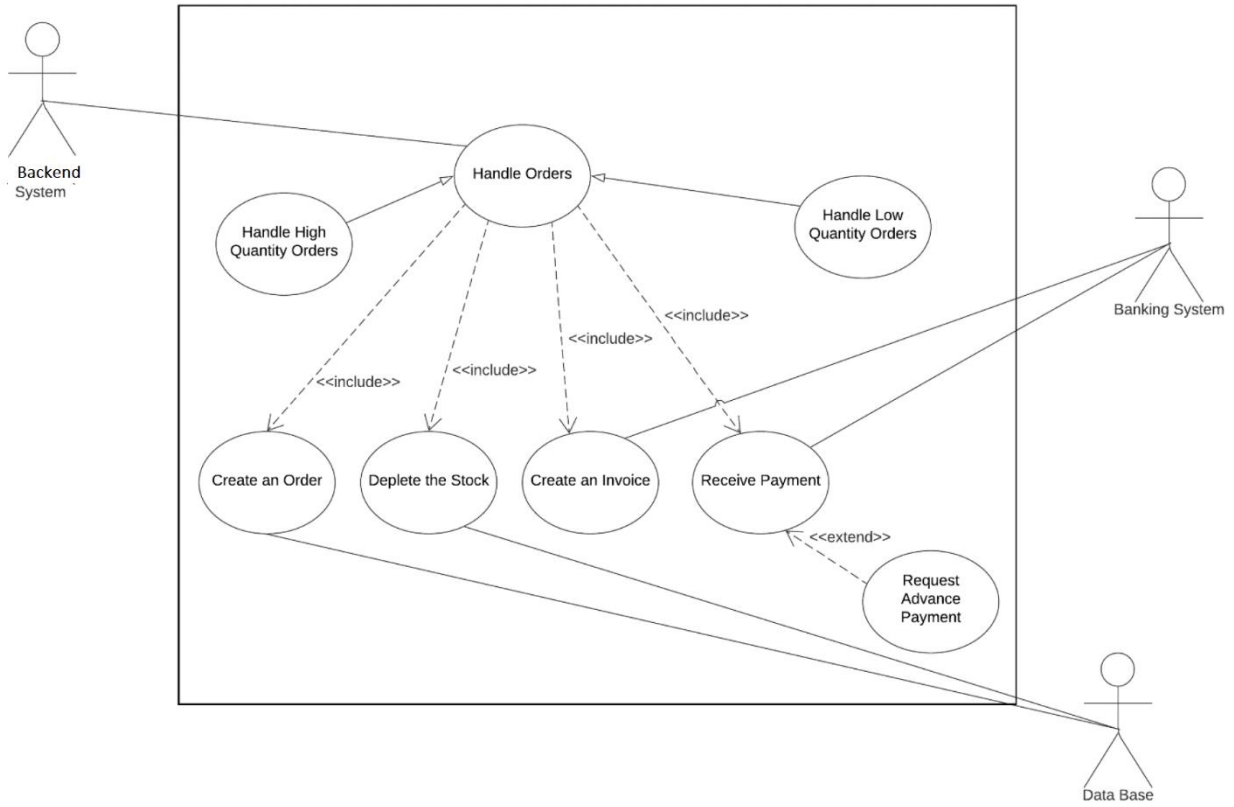
This document describes a set of models that implement the specifications of the software "WAREHOUSE MANAGEMENT SYSTEM". In this document, we show the structural and behavior aspects of the software using Unified Modelling Language 2.0.

Functional Requirements

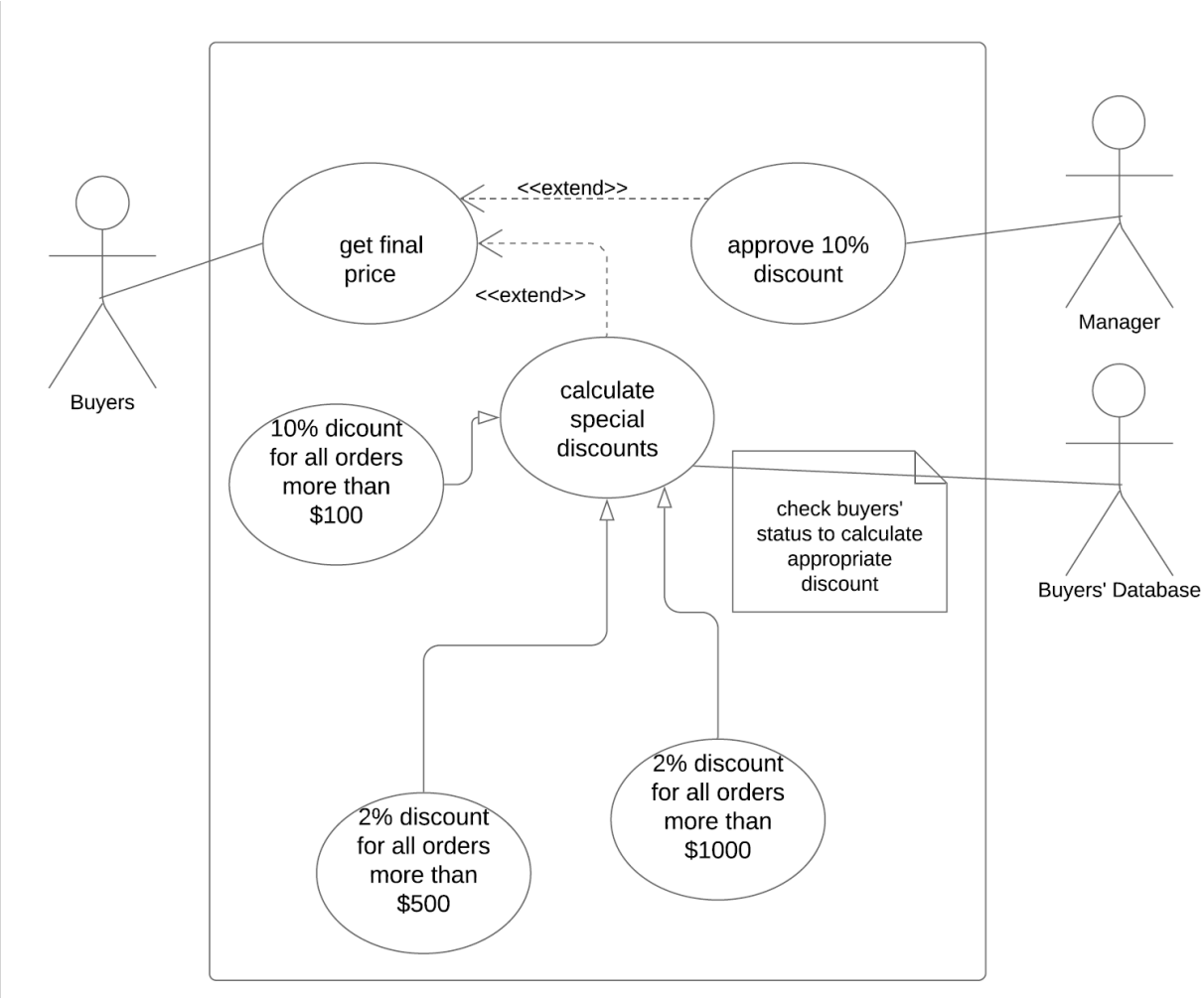
Below, we show the various functional requirements for each use case description we received.



For the first use case, a buyer ('User') may **place** an order through the Welcome Proxy. For the order to proceed, the User's credentials must be **verified** using a Database. There are several ways for verification, either through Username and Password, PIN, or in special cases through a help agent. We use "Extend" here to show that a Help Agent does not usually occur.



For the second use case, we have 3 actors. **Order Handling** is done differently depending on the volume of the quantity ordered, hence the two are different generalizations. For any order to be finalized, **four tasks** need to take place: **Order Creation, Stock Depletion, Invoice Creation,** and **Payment Receipt**. We use <<extend>> to denote that in special cases, payment may be received in advance instead of billing, All billing and invoice generation goes through the banking system. The database is used to save order details.

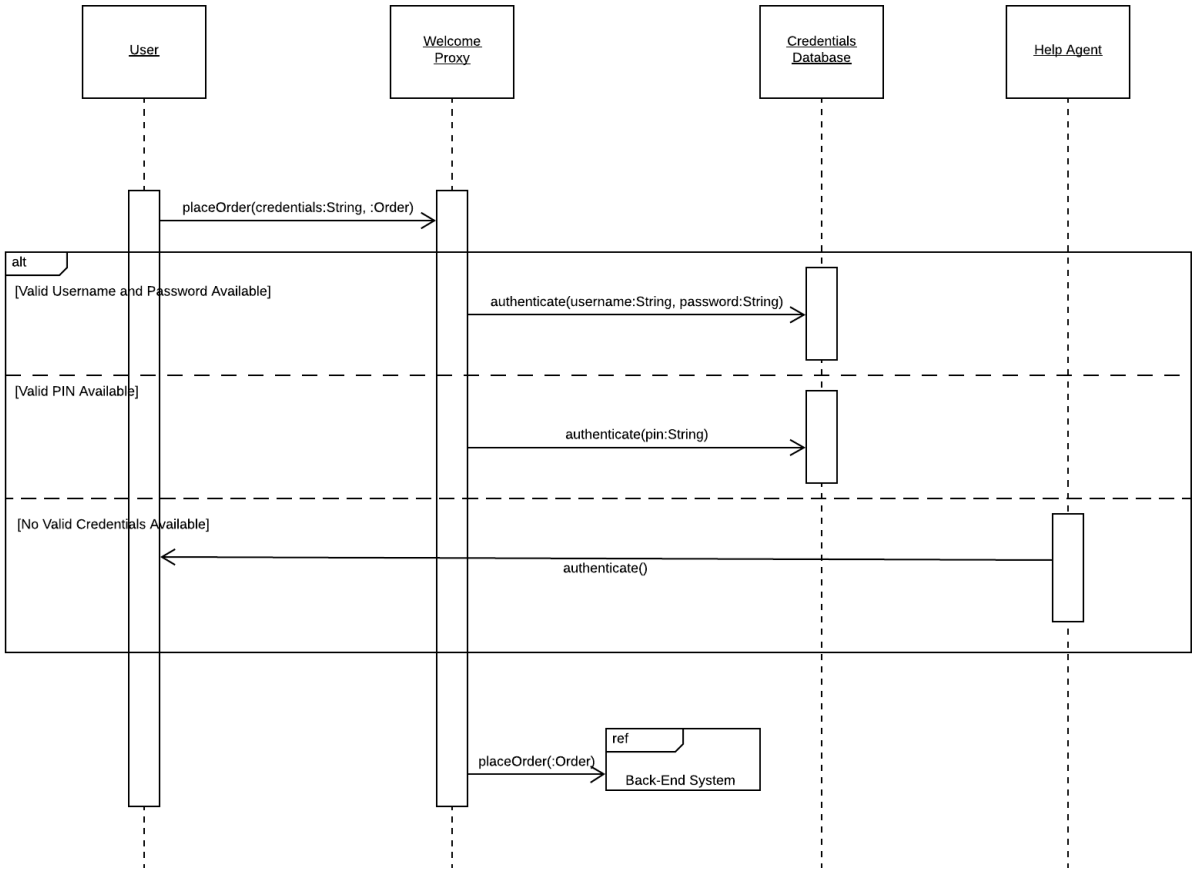


Next, we look at the 3rd Use Case. In this case, we have 3 actors: **buyers, manager,** and a **database.** For the user to get the final price, a discount may or may not be applied. Thus any discount operation has stereotype <<extend>>. There are 3 types of special discounts, depending on the type of the user, which is verified using the database.

Interaction Modelling

Sequence Diagrams

Now, we re-examine the first use case, from a time and behavior perspective.



For the first use case, we have 4 different lifelines associated with the User, Welcome Proxy, Database, and a help agent. The first action to occur is when the User sends a message to the Welcome Proxy to place an order. From here, there are 3 different alternative pathways:

1. If the user supplies valid username and password combination, those are immediately verified by the database.

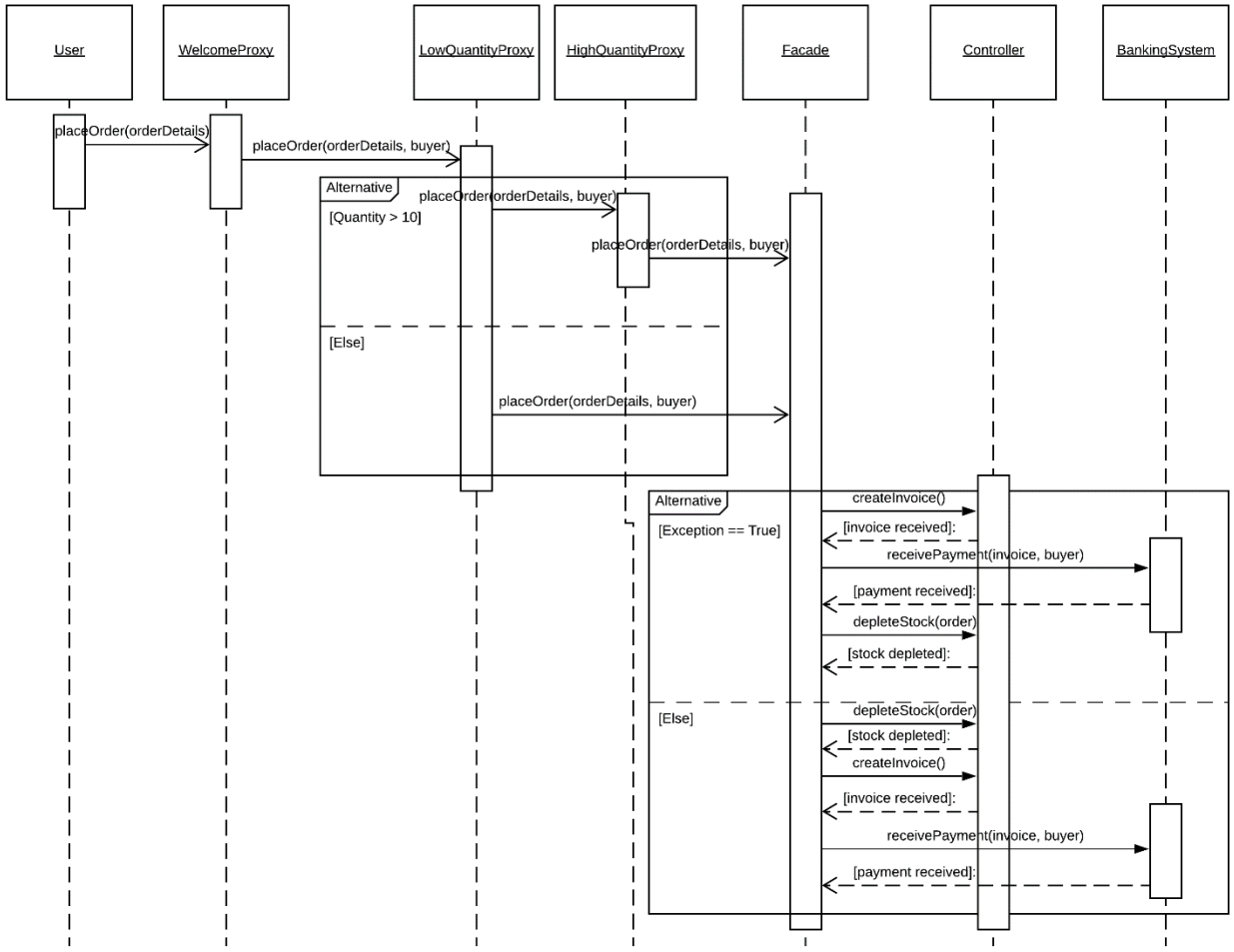
2. Else if the user supplies a valid PIN, it is immediately verified by the database.

3. Else, the help agent helps verify the user.

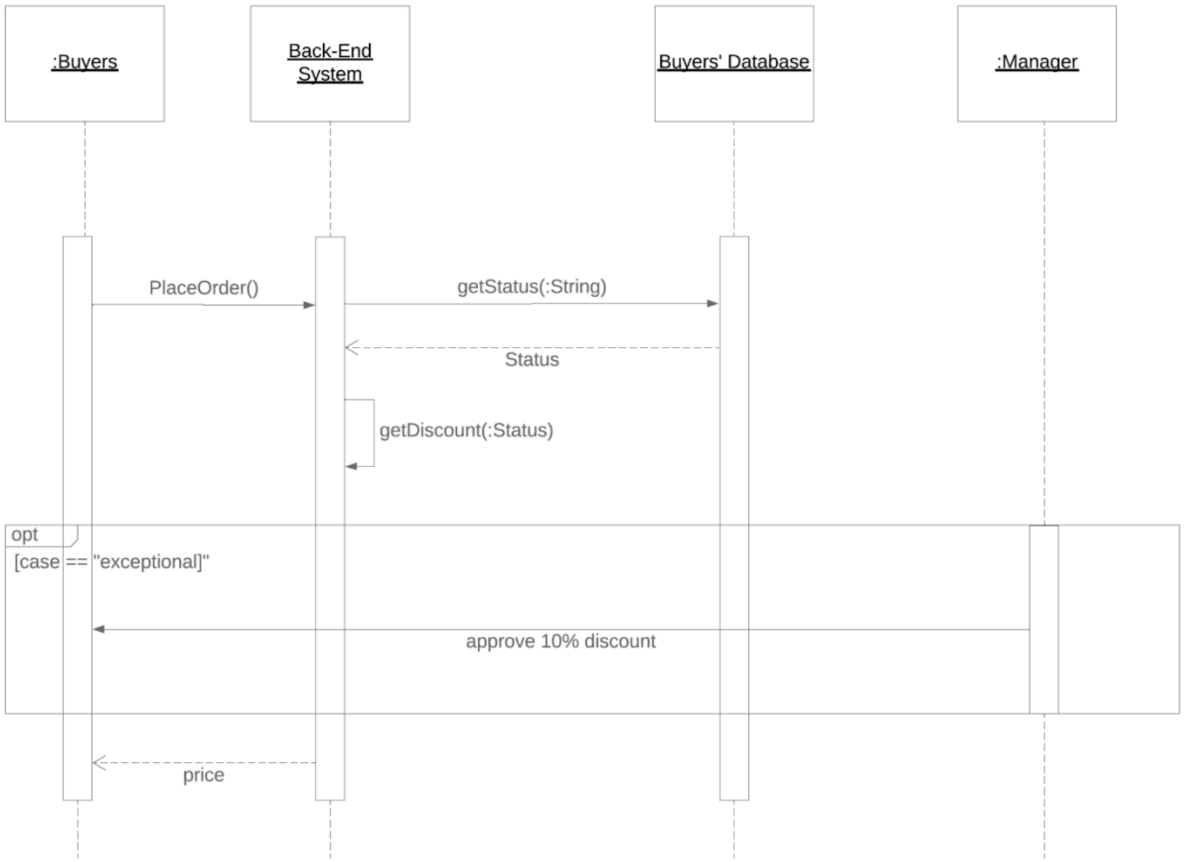
Once verification happens, the Welcome Proxy sends a message to the Back-End System to fulfill the order.

Note the use of asynchronous messaging to allow for further orders.

For the second use case, we start off by the User placing an order through the WelcomeProxy. Next, the order is passed to LowQuantityProxy. Depending on the quantity ordered, the order may be passed to the HighQuantityProxy, hence the use of alt fragment. Eventually, the Façade handles the order. From there, we have another alt fragment: In case the user pays early (before he receives the product), the flow of events changes.

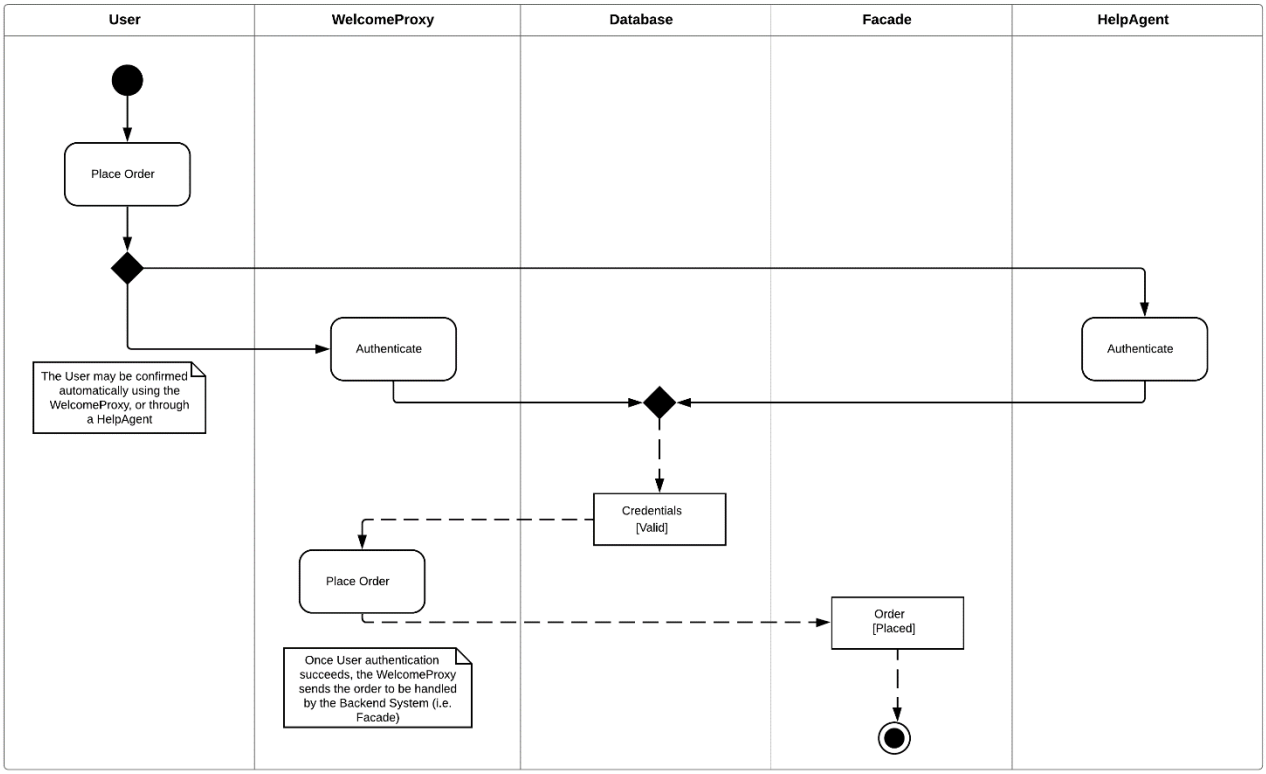


For the 3rd Use Case, we have 4 lifelines. It starts out by the buyer sending a message to place order through the system. For the system to calculate the price, it queries the database (hence a message) to retrieve buyer status to evaluate a proper discount strategy. The self message getDiscount() here implies that a component in the Back-End System calculates the discount. We use the optional fragment to denote the case when the manager may approve a 10% discount. In the end, the system returns the total cost of the order to the Buyer.

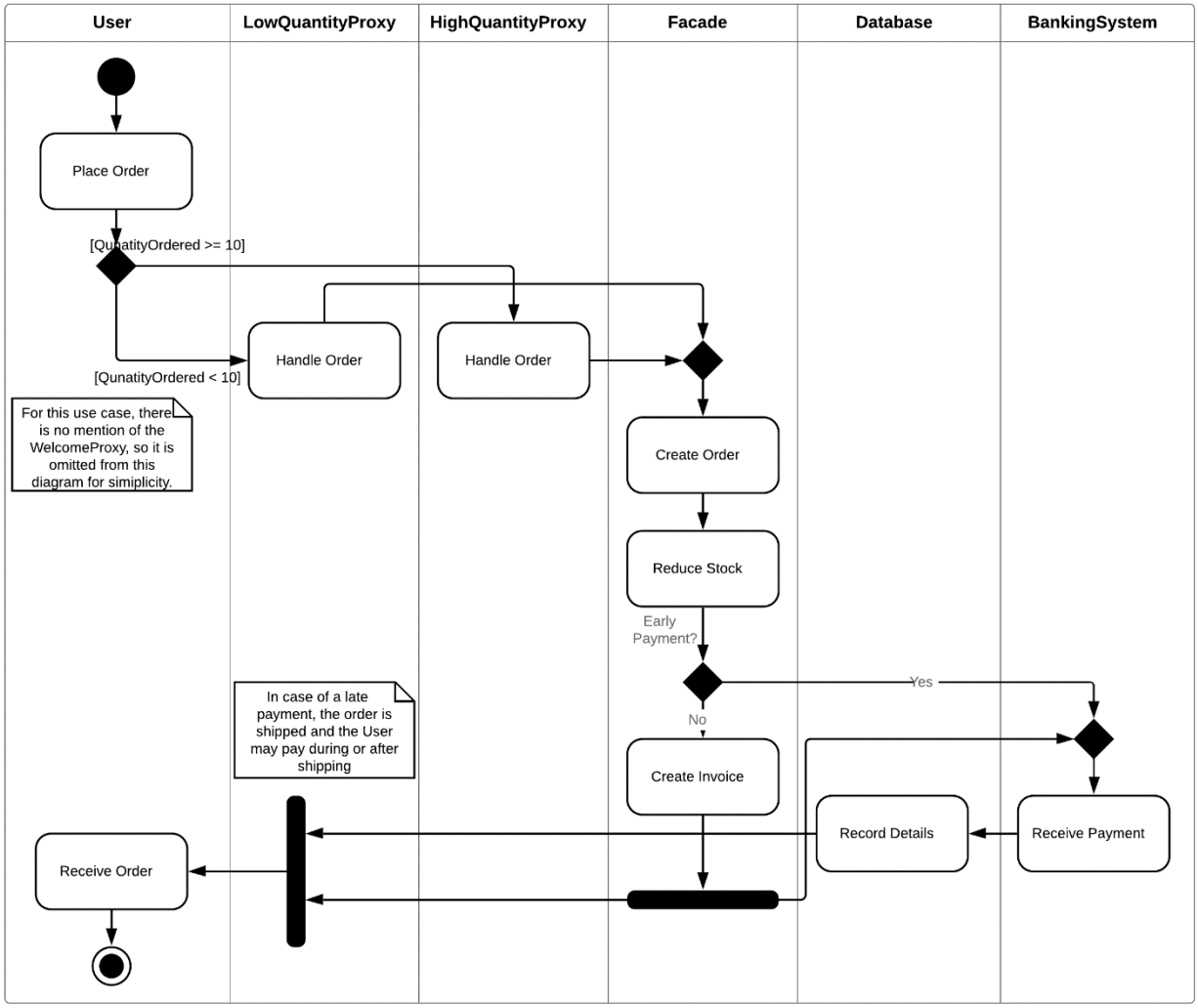


Activity Diagrams

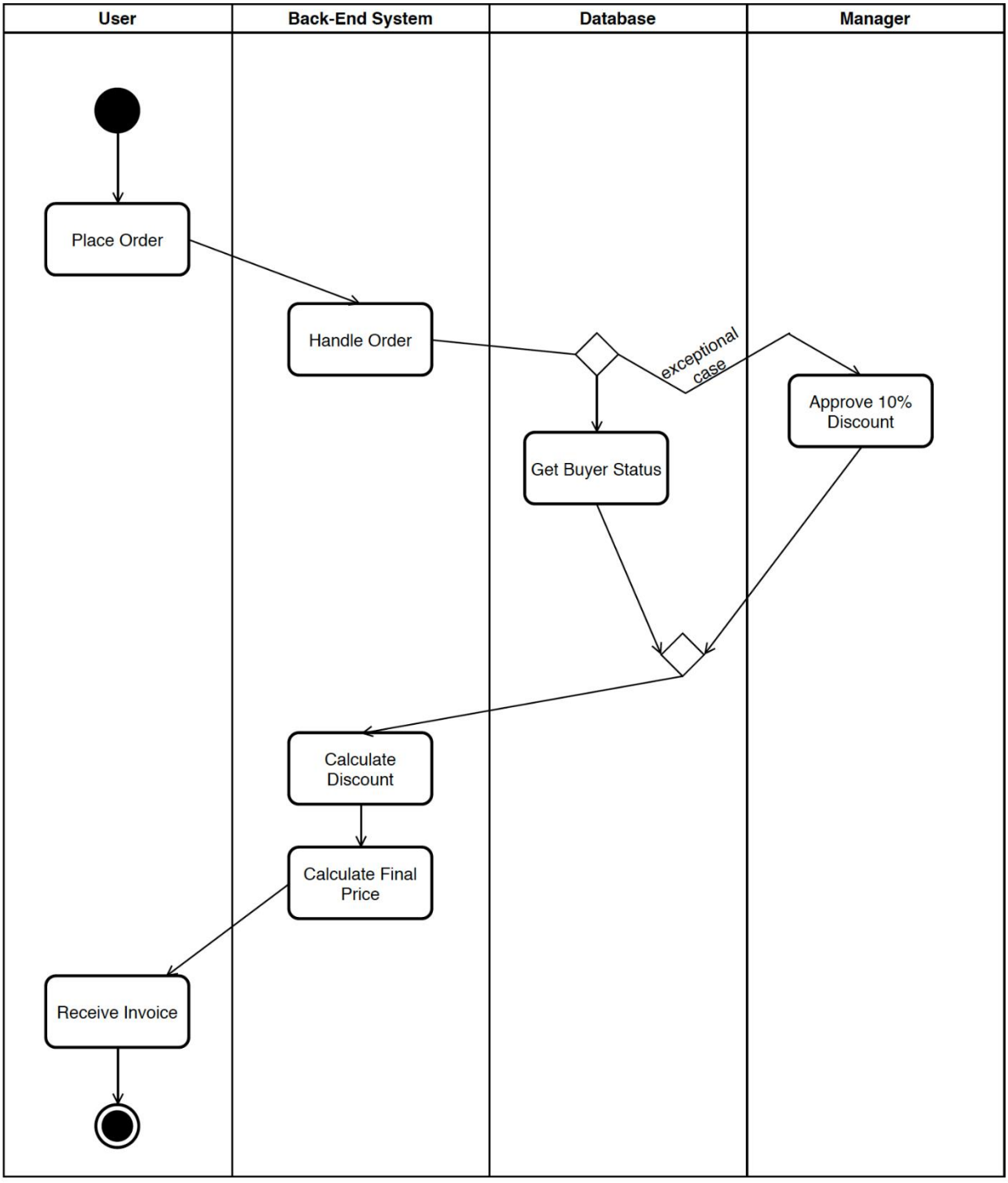
For the first use case, the first process is when the User places an order. From there, we have a diamond (decision) to evaluate: Either authenticating through the Welcome Proxy or a Help Agent. Once authentication is over, we have another diamond, but this time, arrows converge, i.e. all processes end up to the database. We have a rectangle in the Database to show that there is a data moving in this process, namely the valid credentials. Next, the Welcome Proxy places an order on behalf of the user to the Façade, which then terminates the process in this use case.



For the second use case, the process is initiated by the User placing an order. Depending on the quantity ordered, the order may be received by either of LowQuantityProxy or HighQuantityProxy. From there, it sent to the Façade, a main part of the back-end system. From there, the order is created, the stock is reduced, and a decision is made. If the User pays before shipping, no invoice is generated, and payment is immediately received. Else, an invoice is generated, and two concurrent processes happen: Payment Receipt and Order Shipping. Eventually, after payment is received, the order and payment details are recorded in the database, and the process converges to Order Receipt. From there, the process terminates.

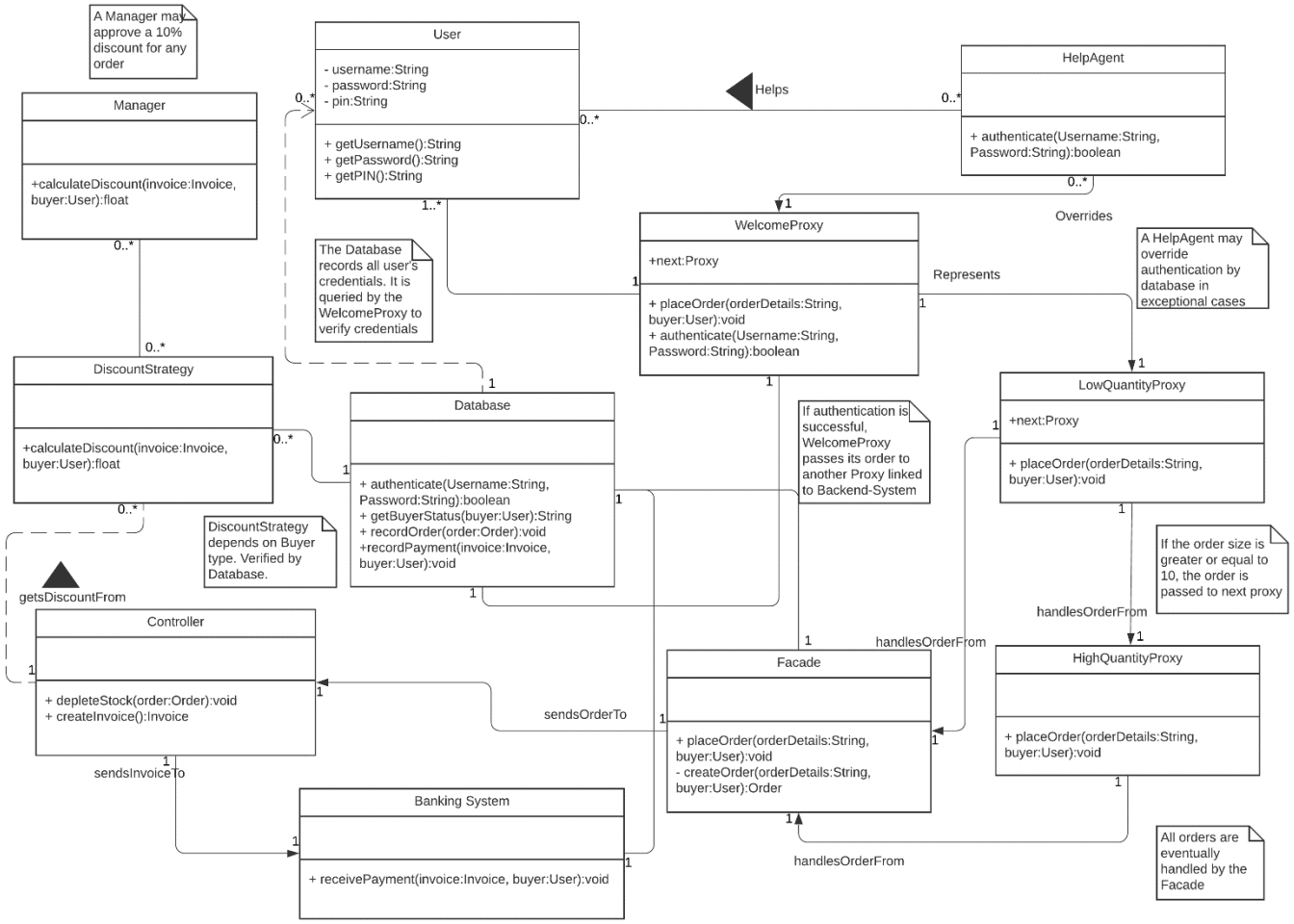


For the third use case, the User initiates the process by placing an order, which is handled by the Back-end System. Next, we have an or-split. The database retrieves the buyer status, which is used to calculate appropriate discount. It is also possible for a Manager to approve a 10% discount. Once the discount is calculated, the system calculates the final price, sends an invoice to the user, and the process terminates.



Domain Model

Now, we turn our attention to a unified domain model of all 3 use cases. Below, we explain the central ideas as well as each class.



The central class in all use cases is a general-purpose database used by many classes for authentication, status verification, order and invoice record-keeping, and other purposes. Some classes are implicit in the use case descriptions, i.e. not mentioned but we have created them to ensure good coupling. Unidirectional arrows denote that some associations are navigated one-way only. For instance, the Welcome Proxy may places an order on behalf of the User through the Low Quantity Proxy only. The Low Quantity Proxy cannot call methods in Welcome Proxy and has no references to it. To read uni-directional associations, we placed a name closer to the class which has to be read first. For instance, HighQuantityProxy -> Façade: Since the name is closer to the Façade, this is read as: "Facade handlesOrderFrom HighQuantityProxy". For some bidirectional associations, we placed a solid triangle to denote how to interpret the association. The base of the triangle denotes which class to read first. For instance, Controller -> Discount Strategy: this is read as "Controller getsDiscountFrom DiscountStrategy". Dependencies are shown using dashed lines. This means any change in the class which has the incoming arrow will cause a change in the dependent class. For instance, in Database -> User, any change in the User information must be coupled with a change in the Database. Some

interfaces are absent to avoid cluttering of the diagram. Note the cardinality of many classes is 1 to 1. This is due to the implementation of the Singleton Design Pattern. There can never be more than one instance, so the cardinality is always 1. Now, we explain each class briefly:

- User: This is any client that uses the system. In the 3 use cases we are dealing with, we only have the buyer to worry about. We used private attributes to hide User credentials. There are public getter methods to allow authentication of the User.
- HelpAgent: This is an extension to the system to override normal routes of authentication in case a legitimate user cannot verify his credentials for some reason. It has an authenticate method of its own.
- WelcomeProxy: This is the class that the User deals with. For the 3 use cases, it allows 2 different methods: authentication and order placing. Note the use of the public attribute next. This attribute ensures that the order placed by the user is passed on to another Proxy in the system that actually handles orders. This is the implementation of the Proxy Design Pattern.
- LowQuantityProxy: This class handles orders depending on the quantity ordered. If the quantity ordered by the user is at least 10, then it passes on the order to next (hence the attribute) Proxy. This is the implementation of the Chain of Command Design Pattern.
- HighQuantityProxy: Extremely similar to the previous class. The only difference is that it does not have a next attribute, and delegates all orders to the Façade, hence implementing the Façade Design Pattern.
- Facade: This is one of the main components of the Back-End System as it deals with various tasks, the most important of which are: creating the actual order. All the proxies do is pass on order details to each other. It is only the Façade that creates the order.
- Controller: Another important class in the Back-End System. It receives the order created by the Façade to reduce stock and generate invoice appropriately.
- DiscountStrategy: This class is a sloppy implementation of the Strategy Design Pattern. For the Controller to calculate the final price, it must receive a discount from this class. A discount may be a 0% discount or any special discount. Any Strategy instance is connected to the Database to check Buyer status. Some Users are associated with a special status that corresponds to a different discount strategy.
- Manager: Similar to the HelpAgent, this class overrides the system if needed to add a 10% discount. Note the minimum cardinality is 0 because it is an optional structure.
- BankingSystem: The purpose of this class is to receive payment from the User. To know the exact price, the Controller class sends a copy of the invoice to this class. Additionally, this class is connected to the Database to bill the User according to the recorded details.
- Database: This is a general-purpose class. It has many different functionalities. Ideally, when this is taken to low-level design, we must have separate databases, each specialized for a purpose. This class has several methods, including authenticate() that the WelcomeProxy may use, getBuyerStatus(), which may be used by the DiscountStrategy, recordOrder() used by the Façade, or recordPayment() used by the BankingSystem.