**Final Project in Software Engineering**

**Application Design Document**

Omer Presler, Amit Grumet, Rotem Cohen

**Chapter 1 – Use Cases**

**1.1 User Profiles — The Actors**

1) Admin: system administrator is a user with permission to alter the database and the system’s businesses and add new users to the system.

2) Business: a business in the kibbutz. Is managed by the business’s workers (kibbutz members, or outside workers).

3) Kibbutz member: any kibbutz member, who can use the services of the kibbutz’s businesses.

4)  Accounting: accounting users can get business expenses and profit reports.

**1.2 Use-cases**

1. Add a new user:

Actor: Admin

Pre-condition: None

Post-condition: A user is added to the database.

Main-flow: An admin enters a user’s details: name, budget number (if has any, the user may not be a kibbutz member, but only a business worker), phone number, and email address, then clicks on “Add”, then a new user is added to the database.

Alternative-flow: When one of the details is incorrect; a phone number with more or less than 10 digits or one of the mandatory details is missing, the user will not be added to the database, and feedback for failed action displays to the admin’s screen.

Positive test: After adding a new user, check if he exists in the database.

Negative test: After failed adding action, check the user wasn’t added to the users’ database.

1. New request created:

Actor: Kibbutz member

Pre-condition: There is a business.

Post-condition: A request has been created.

Main-flow: A member opens a business’s page, browses, adds what he wants, and requests it (requests a cake for example). After that, a request has been created.

Alternative-flow: If the request’s text is empty, it won’t appear to the business’s workers.

Positive test:  After creating a new request a new request will appear.

Negative test: No new request will appear after creating an invalid new request.

1. On-Site Purchase – member side:

Actor: Kibbutz member

Pre-condition: A member with a budget number is an on-site store and is willing to purchase a product.

Post-condition: The purchase is saved in the databases.

Main-flow: A member buys a product from a business, and the cashier adds the purchase to the database through the system. The member will be able to see his purchase on his orders page in the system.

Alternative-flow: None

Positive test:  After creating a new purchase, it will appear in the database, and the member will be able to see it.

Negative test: None

1. On-Site Purchase – business side:

Actor: Business

Pre-condition: A member with a budget number is an on-site store and is willing to purchase a product.

Post-condition: The purchase is saved in the database.

Main-flow: A member buys a product from a business, and the cashier adds the purchase to the database through the system and thus concludes the transaction.

Alternative-flow: None

Positive test:  After creating a new purchase, the purchase will appear in the database. The business will be able to see that purchase.

Negative test: None

1. Add a product to a business’s website:

Actor: Business

Pre-condition: None

Post-condition: The product has been added to the store.

Main-flow: The actor enters the business, clicks on “Add new product”, add a new product to the business adds photos.

Alternative-flow: When the product’s details are incorrect, the product won’t be added to the business, and an error message will display on the actor’s screen.

Positive test:  The product will be saved in the database after creating a new product.

Negative test: After creating an invalid product, the product will not be added to the database and an error message will appear.

1. Add a service to a business:

Actor: Business

Pre-condition: The actor has the permission to add a service.

Post-condition: The service is added to the business.

Main-flow: The actor enters the business, clicks on “Add new service”, add a new service to the business.

Alternative-flow: When the service’s details are incorrect, the product won’t be added to the business, and an error message will display on the actor’s screen.

Positive test:  After creating a new service the product will be saved on the database.

Negative test: After creating an invalid service the product will not be added, and an error message will appear.

1. Remove a service from a business:

Actor:  Business

Pre-condition: There is a service in the business.

Post-condition: The service was deleted from the business.

Main-flow: The actor enters the business, enters the services list, picks the service he would like to remove, clicks on “Remove”, and the service is removed from the database.

Alternative-flow: None

Positive test:  After removing an existing service the service will not be on the database.

Negative test: After removing a non existing service the service will not be removed, and an error message will appear.

1. Remove a user:

Actor: Admin

Pre-condition: The user whom the admin wants to remove exists in the database.

Post-condition: The user appears as non-active.

Main-flow: An admin enters the list of all users, picks a user whom he would like to remove, clicks on “Remove”, then the user is a non-active user.

Alternative-flow: None

Positive test: After removing a user, check if the database is updated.

Negative test: None

1. Restore a user:

Actor: Admin

Pre-condition: The user, whom the admin wants to restore, exists in the database.

Post-condition: The user appears as active.

Main-flow: An admin enters the non-active users’ list, picks a user whom he would like to restore, clicks on “Restore”, then this user appears as active.

Alternative-flow: None

Positive test: After restoring a user, check if the databases is updated.

Negative test: None

1. Add new business:

Actor: Admin

Pre-condition: The new business’s manager is a user in the system.

Post-condition: A business is added to the system.

Main-flow: An admin clicks on “Add new business”, adds the business’s details: name, business manager (from users’ list), clicks on “Add”, then the new business is added to the businesses’ database.

Alternative-flow: If one of the details is incorrect, the business will not be added to the database, and feedback for failed action displays to the admin’s screen.

Positive test: After adding, the database contains the business.

Negative test: After failed adding action, check that the database doesn’t contain the business.

1. Remove business:

Actor: Admin

Pre-condition: The business which the admin wants to remove exists in the system and appears as an active business.

Post-condition: The business appears as non-active.

Main-flow: An admin enters the businesses’ list, picks a business which he would like to remove, clicks on “Remove”, then the business appears as non-active.

Alternative-flow: None

Positive test: After removing a business, check if the database is updated.

Negative test: None

1. Restore business:

Actor: Admin

Pre-condition:  The business which the admin wants to restore exists as a non-active business.

Post-condition: The business appears as active.

Main-flow: An admin enters the non-active businesses’ list, picks a business which he would like to restore, clicks on “Restore”, then the business is active.

Alternative-flow: None

Positive test: After restoring, check if the database is updated.

Negative test: None

1. Add new admin:

Actor: Admin

Pre-condition: The user, who the admin would like to make admin, is in the system and active.

Post-condition: The chosen user has admin permissions.

Main-flow: An admin enters the users’ list, chooses a user, clicks on “Make admin”, then the user has admin permissions.

Alternative-flow: None

Positive test: After adding admin, check if the user has admin permissions.

Negative test: None

1. Remove a product from a business’s website:

Actor: Business

Pre-condition: There is a product in the business.

Post-condition: The product was removed from the business.

Main-flow: The actor enters the business, enters the products’ list, picks the product he would like to remove, clicks on “Remove”, and the product is removed from the database.

Alternative-flow: None

Positive test:  After removing an existing product the product will not be on the database.

Negative test: None.

1. Change request status:

Actor: Business

Pre-condition: There is a request in the business.

Post-condition: The request’s status has been changed, and a notification is sent to the requesting member.

Main flow: The actor enters the request, clicks on “change status”, changes it, and a notification is sent to the requesting member.

Alternative-flow: the request is made by a non-active user.

Positive test: After changing the status, the change has been made in the worker’s site and in the requesting member’s site.

Negative test: If the member is non-active, a notification won’t be sent.

1. Close a request:

Actor: Business

Pre-condition: There is a request in the business.

Post-condition: The request appears as closed.

Main-flow: The actor enters the business, picks the request, changes its status to “Closed”, and then the request moves from the open requests list to the closed requests list.

Alternative-flow: None

Positive test: After closing a request it will be moved to the closed requests list and be removed from the open requests list.

Negative test: None

1. Reject a request:

Actor: Business

Pre-condition: There is a request to reject/accept.

Post-condition: The request appears as rejected.

Main-flow: The actor enters the business, picks the request, changes its status to “rejected” and explains the way, and then the request moves from the open requests list to the rejected requests list.

Alternative-flow: None

Positive test: After rejecting a request it will be moved to the rejected requests list and be removed from the open requests list.

Negative test: None

1. display report:

Actor: Accounting

Pre-condition: There is a business in the system.

Post-condition: The accounting worker receives a table of a business’s expenses.

Main-flow: The actor enters the business, clicks on “get report”, and chooses the time frame.

Alternative-flow: The actor chose a future time frame, and the report won’t be sent.

Positive test: After selecting a business and a time frame, the report displays.

Negative test: None.

1. Display order history by parameter:

Actor: Business

Pre-condition: There is a business in the system.

Post-condition: The worker receives the history of orders of the business by the requested parameter.

Main-flow: The actor enters order history, filters the result by the requested parameter (budget number, house number, time frame, etc), then the result displays.

Alternative-flow: None.

Positive test: After filtering, the appropriate data displays.

Negative test: None.

1. Display a house location on google maps:

Actor: Member

Pre-condition: There is location data for houses.

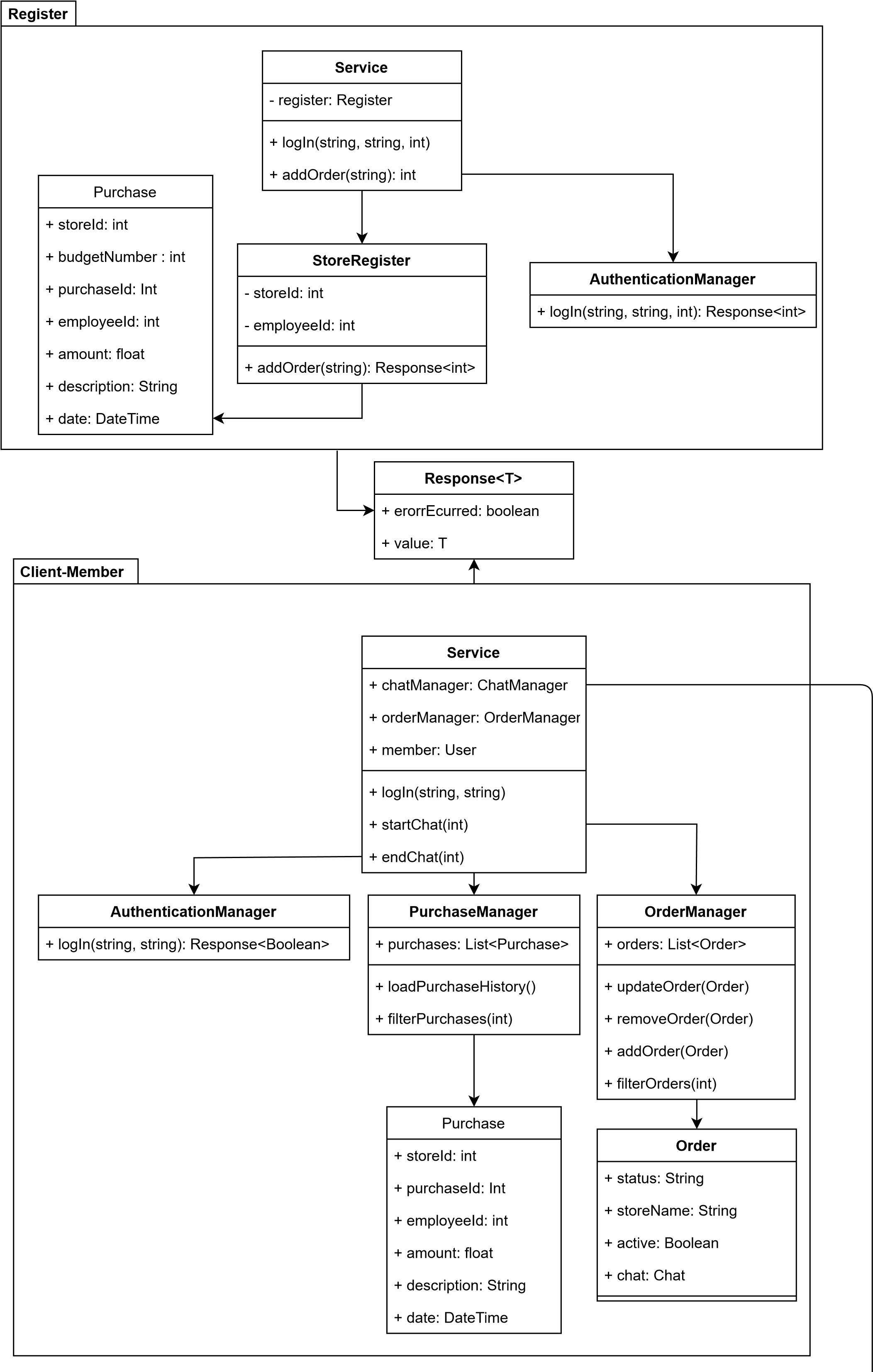
Post-condition: The actor receives the location of the house on google maps.

Main-flow: The actor enters other member’s contacts, clicks on house’ and then the google maps location displays.

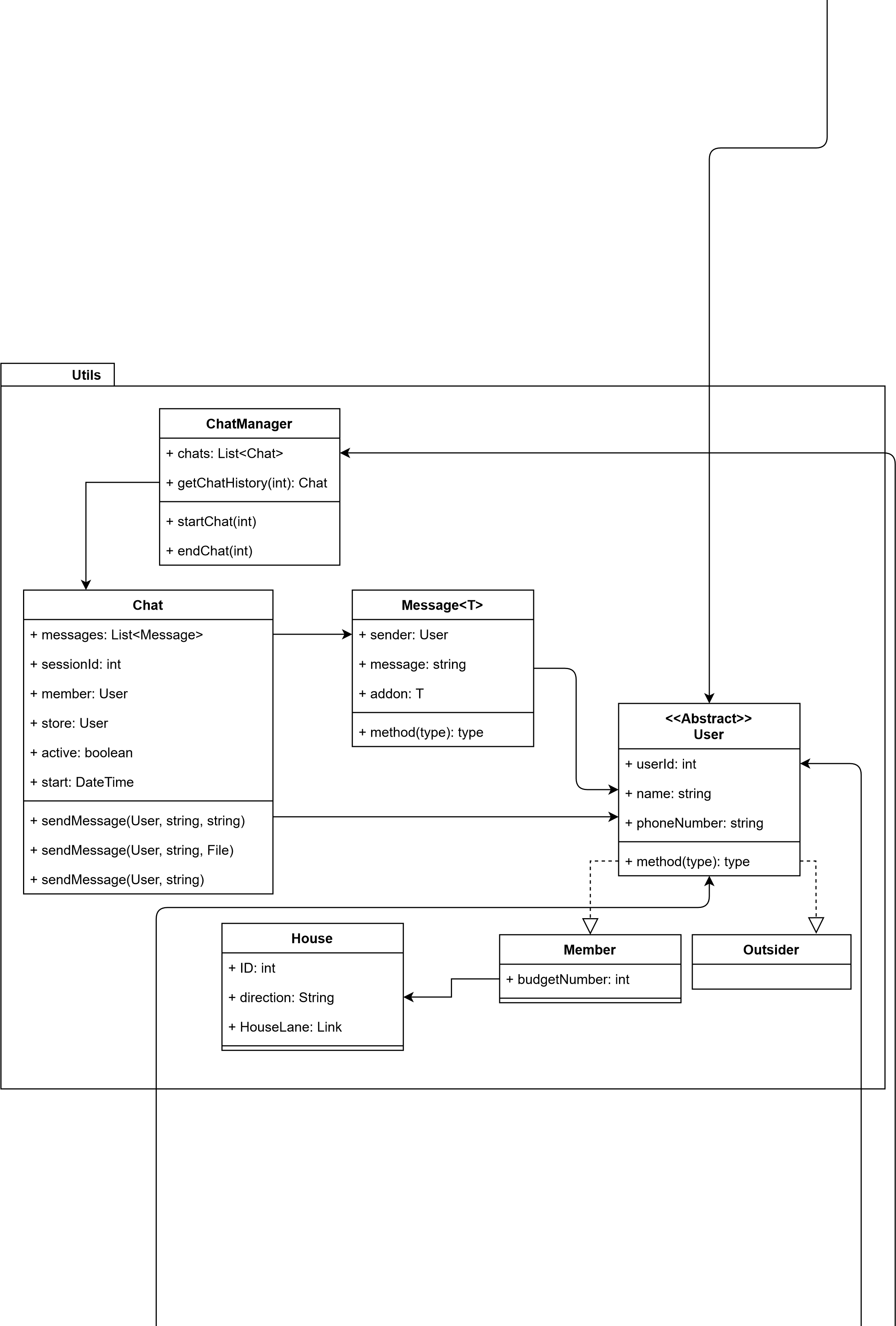
Alternative-flow: If the location data is not set in the database, an error message displays.

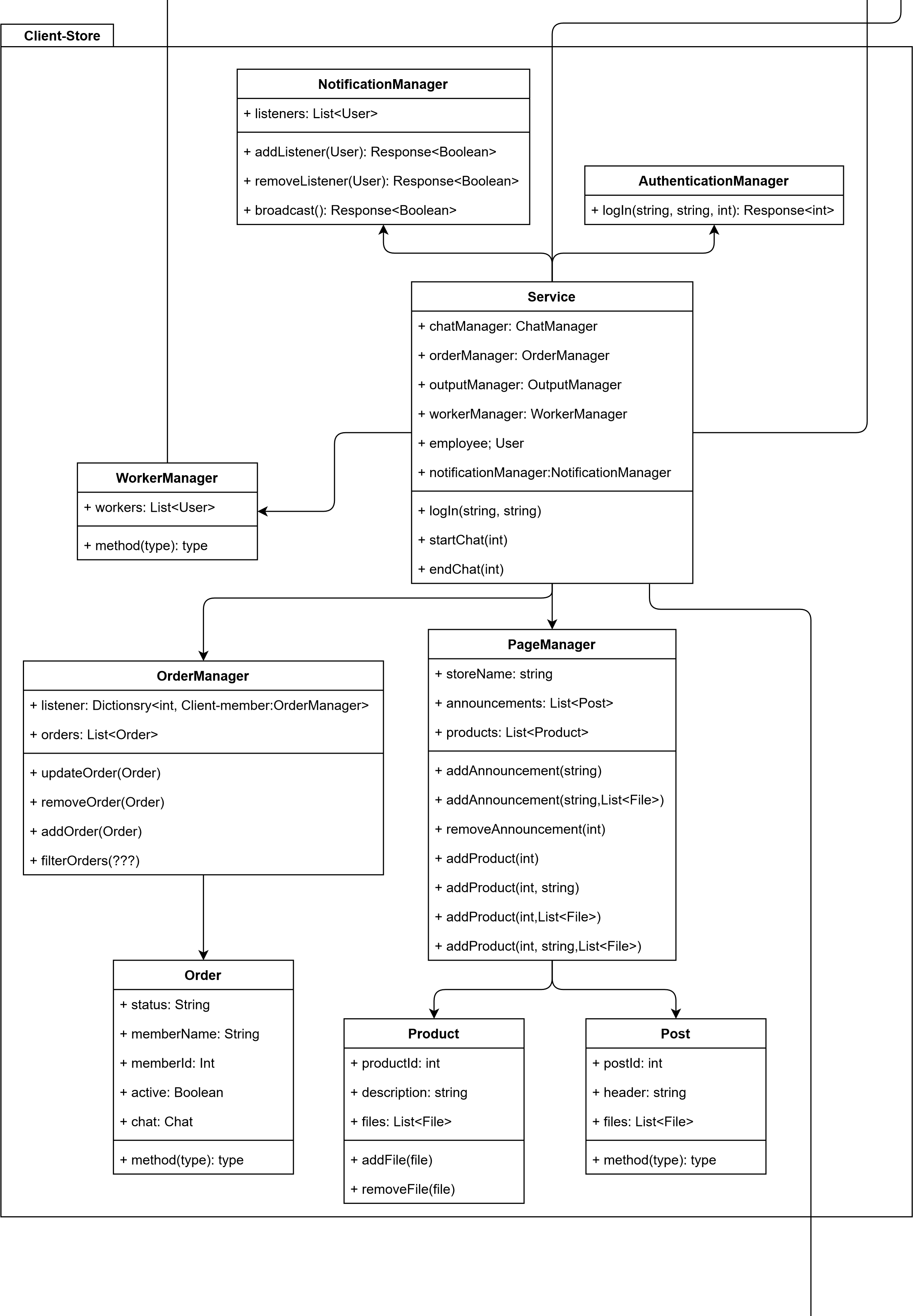
Positive test: After entering the house’s location, google maps open with the location.

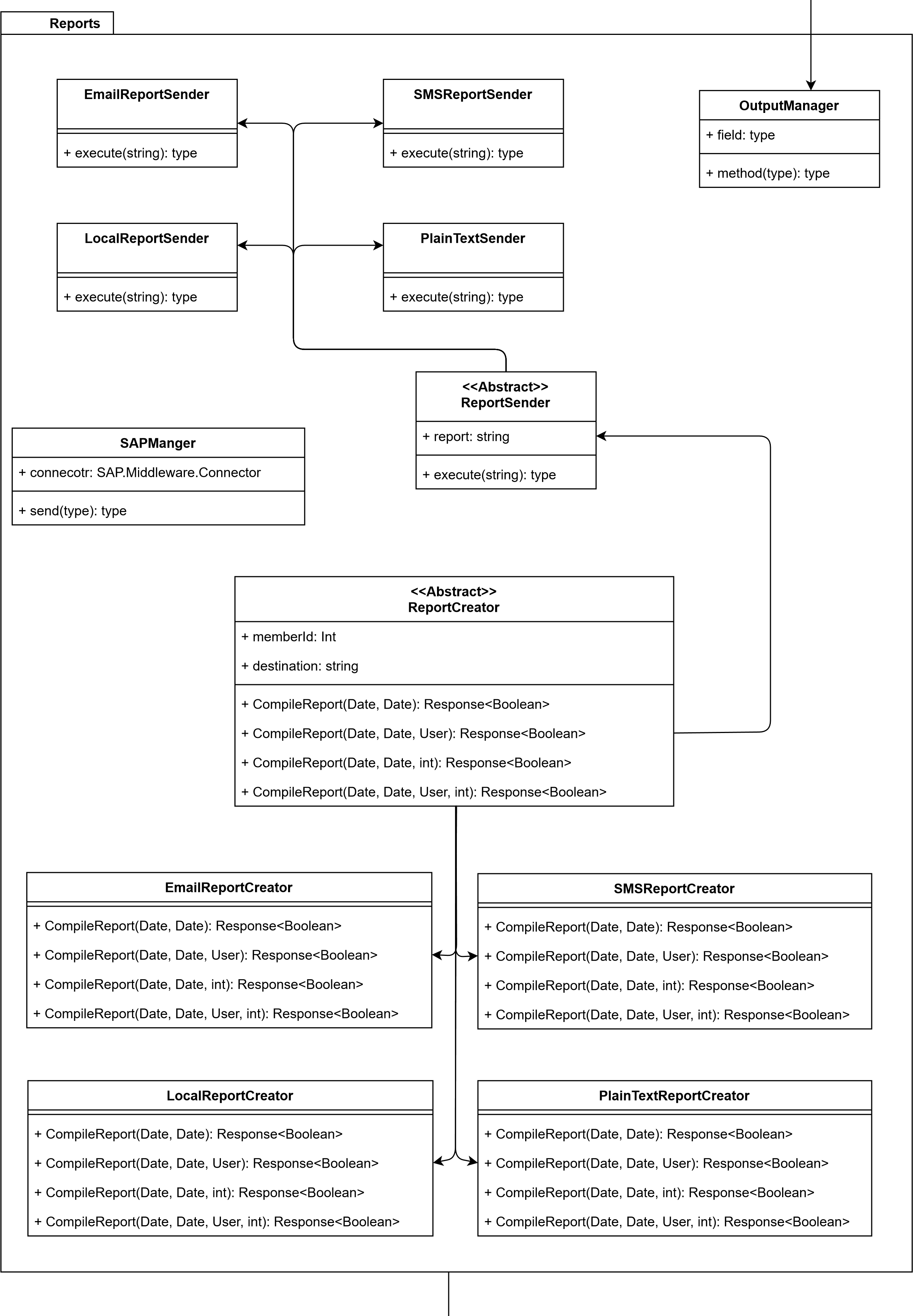
Negative test: An error message displays after entering a missing house’s location.

**e**

**Chapter 2 – System Architecture**







ExternalServices

|  |  |
| --- | --- |
| **Exceptions** |  |
| **Interface**  **>>**  **<<**  **IException**  +  message: String  **ClientException**  **StoreException**  **RegisterException** | |

**Chapter 3 – Data Model**

**3.1 Description of Data Objects**

Will be using the NHibernate ORM library.

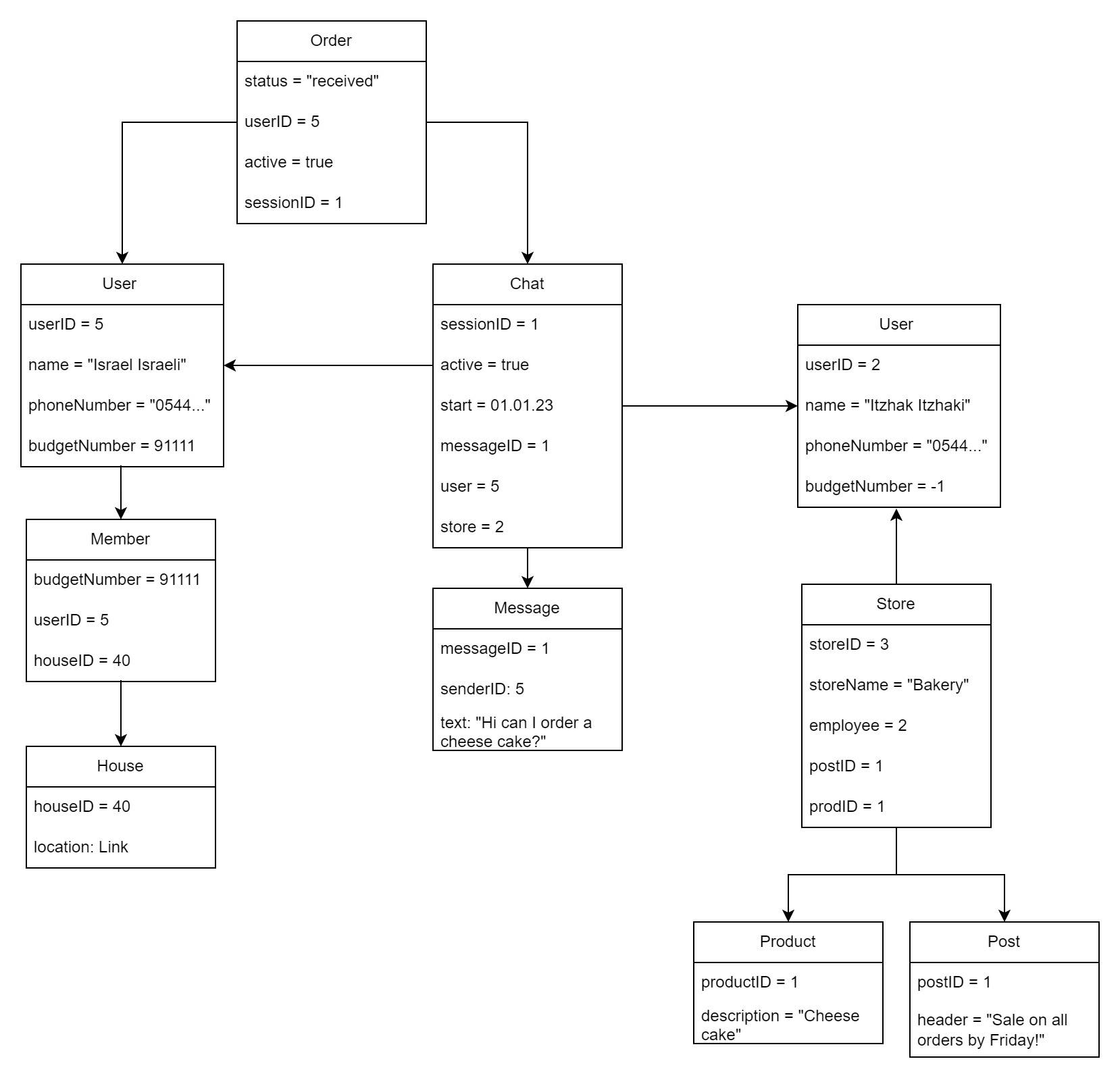
NHibernate is an object-relational mapping (ORM) tool for .NET languages such as C#. It allows developers to work with a database using object-oriented concepts, rather than having to write SQL queries or use other database-specific APIs. To use NHibernate in our system, we will need to create mappings between our C# classes and the database tables we want to work with. Then use the NHibernate API to perform database operations such as querying, inserting, updating, and deleting data. NHibernate provides a flexible and powerful way to interact with a database from a C# application.

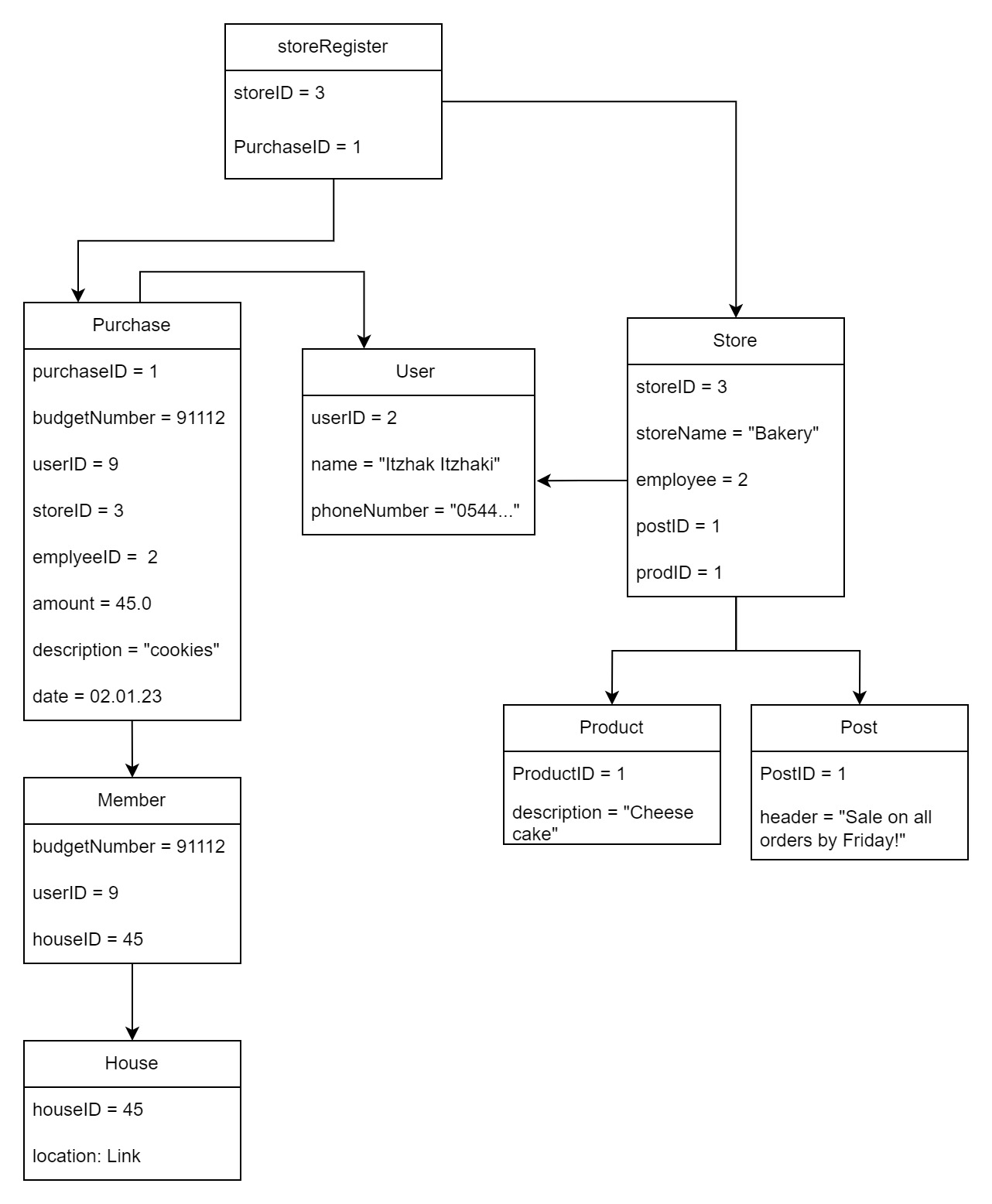
The tables we will save in the database:

* Chat: the chat between two users.
* Purchase: the purchase in the store register in the different businesses.
* Massage: the massage in the chat.
* House: an object that represents a house in the kibbutz.
* User: s user in the system can be either a kibbutz member or non kibbutz member.
* Member: a kibbutz member
* Store: a store consists of posts and products.
* Post: a post with photos text links etc.
* Product: a product a store is offering to consumers.
* Order: an order made on the system.

**3.2 Entity Relations Diagrams (ERD)**

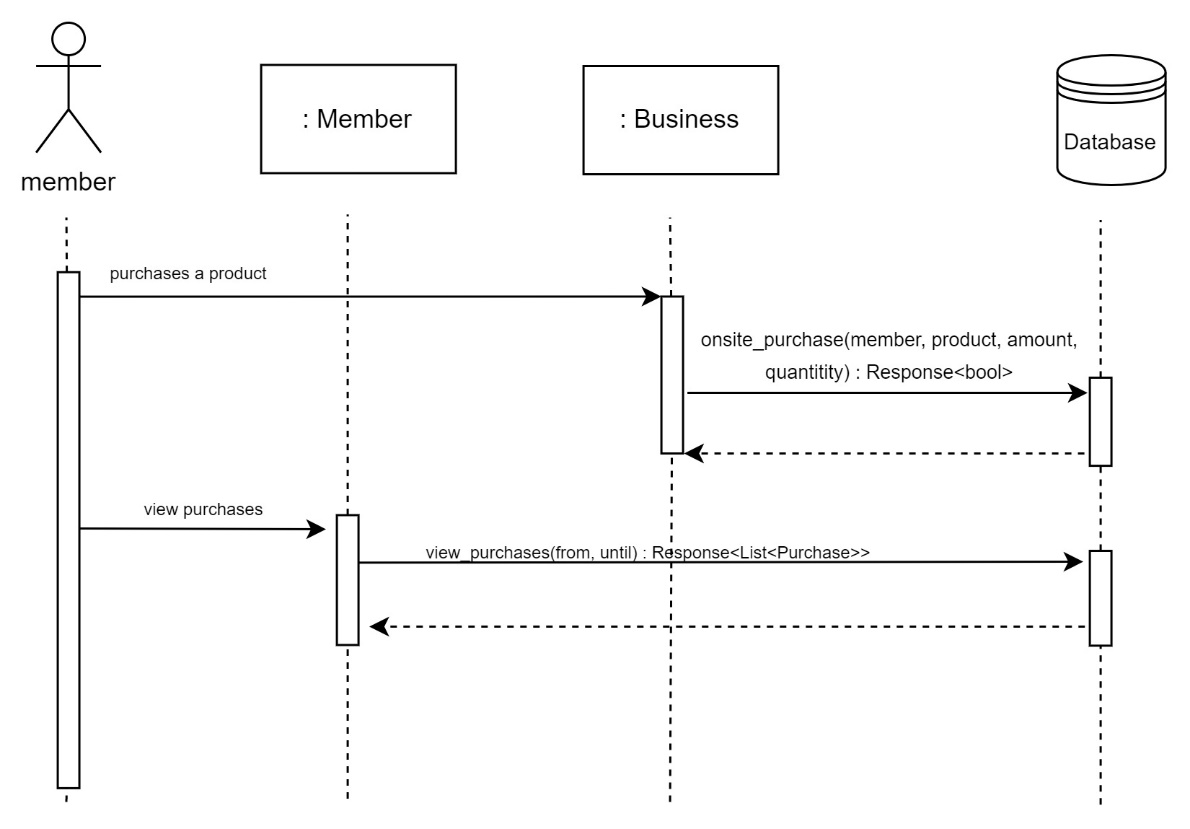
* A description of an order in the system:

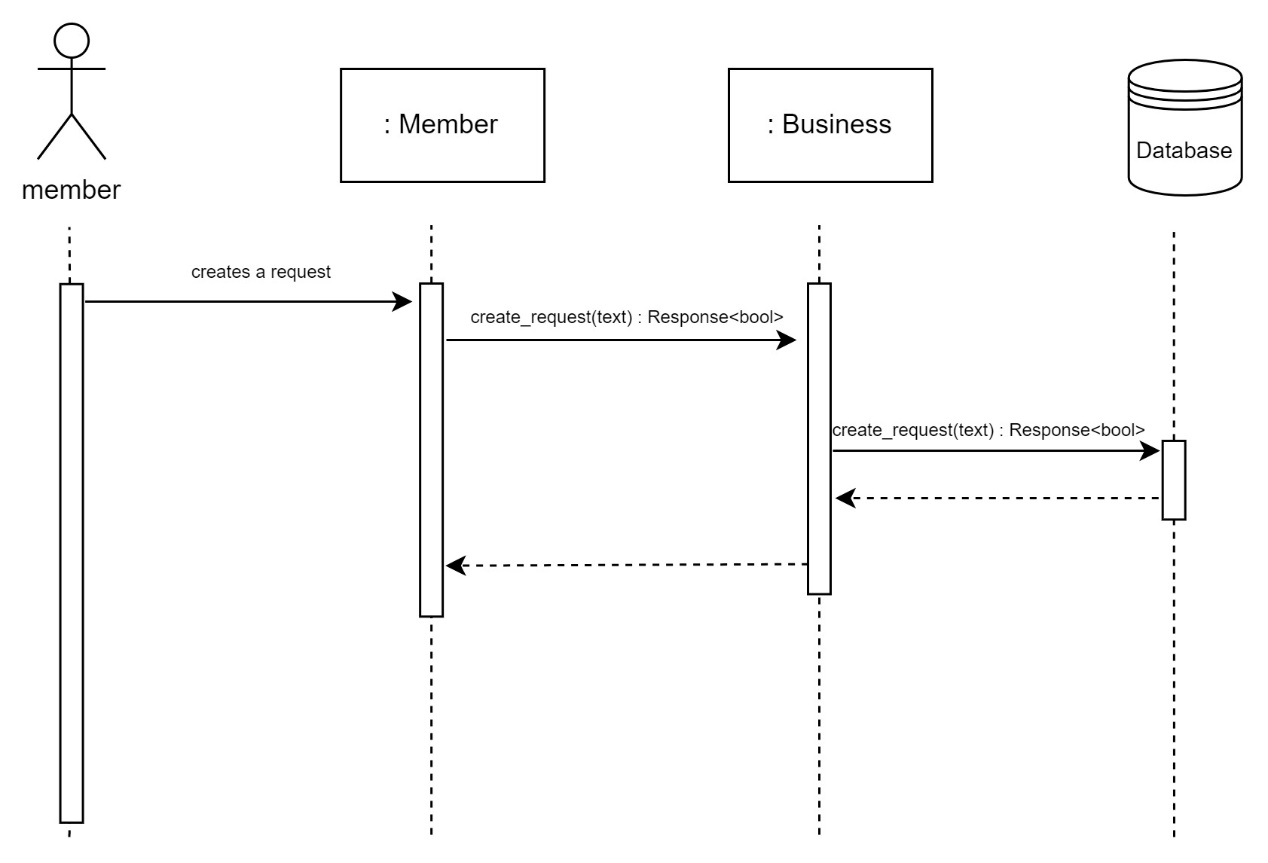
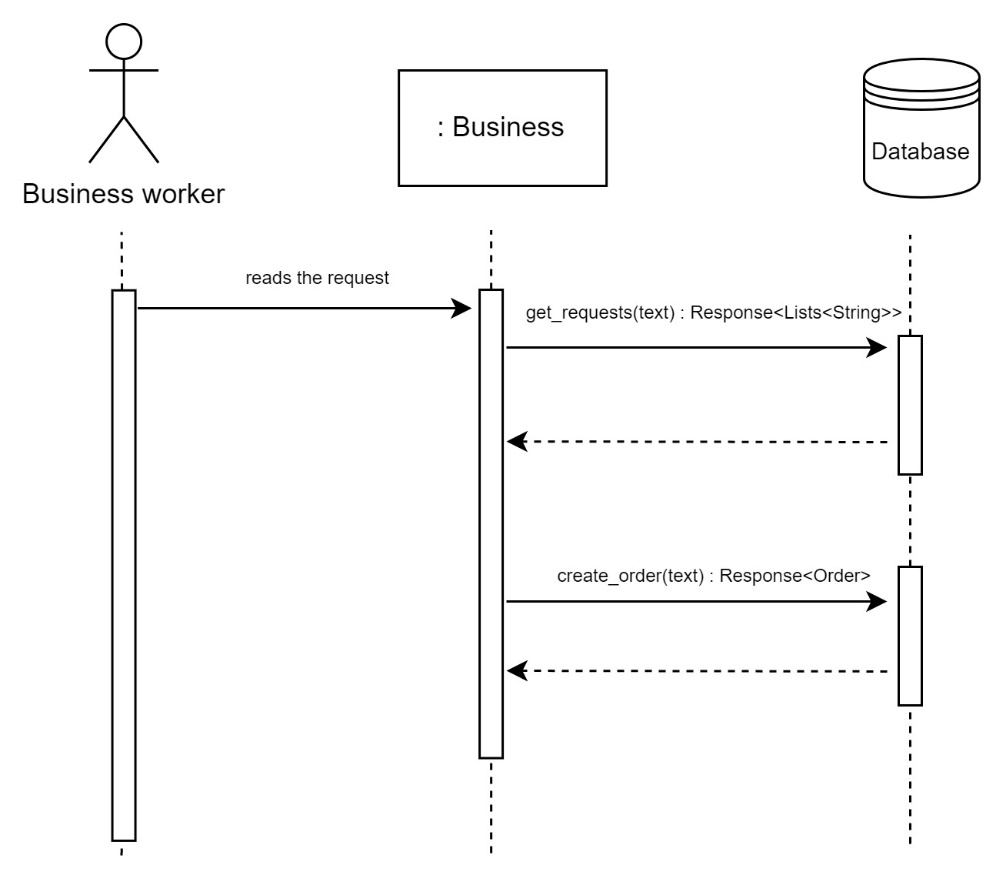
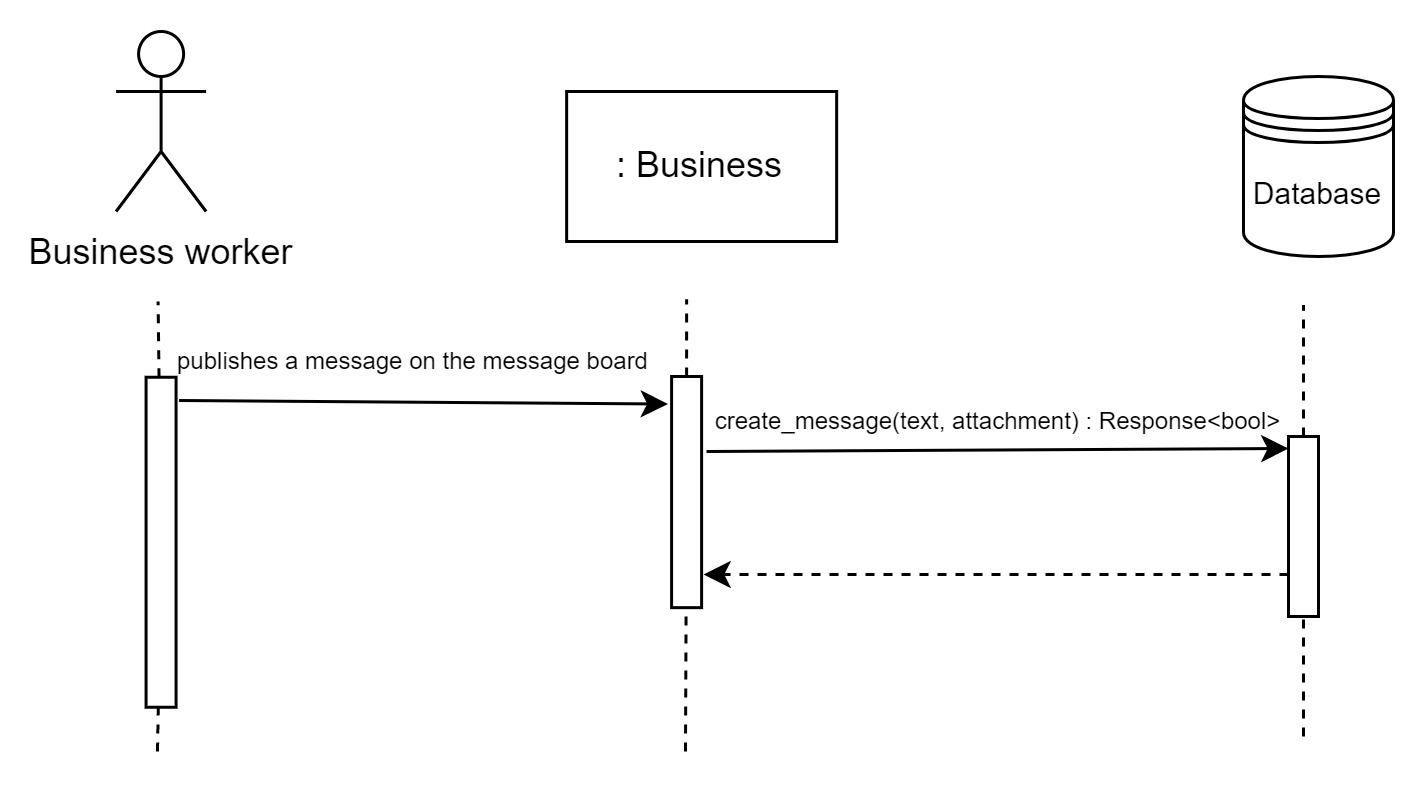


* A description of an order on the store (on-site):

**Chapter 4 – Behavioral Analysis**

**4.1 Sequence Diagrams**

* On-site purchase – member side:  
  
* On-site purchase – business side:  
  Diagram

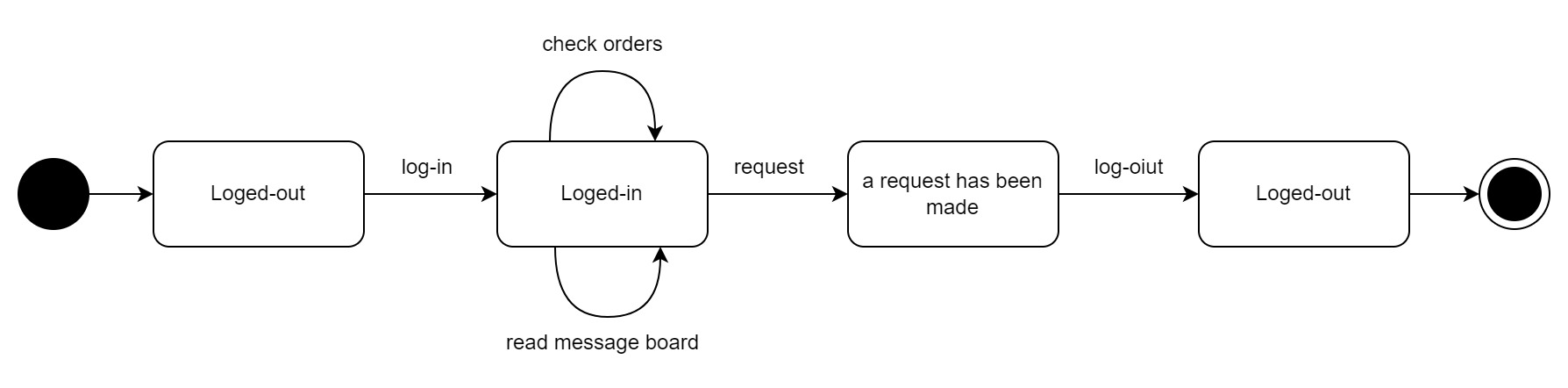
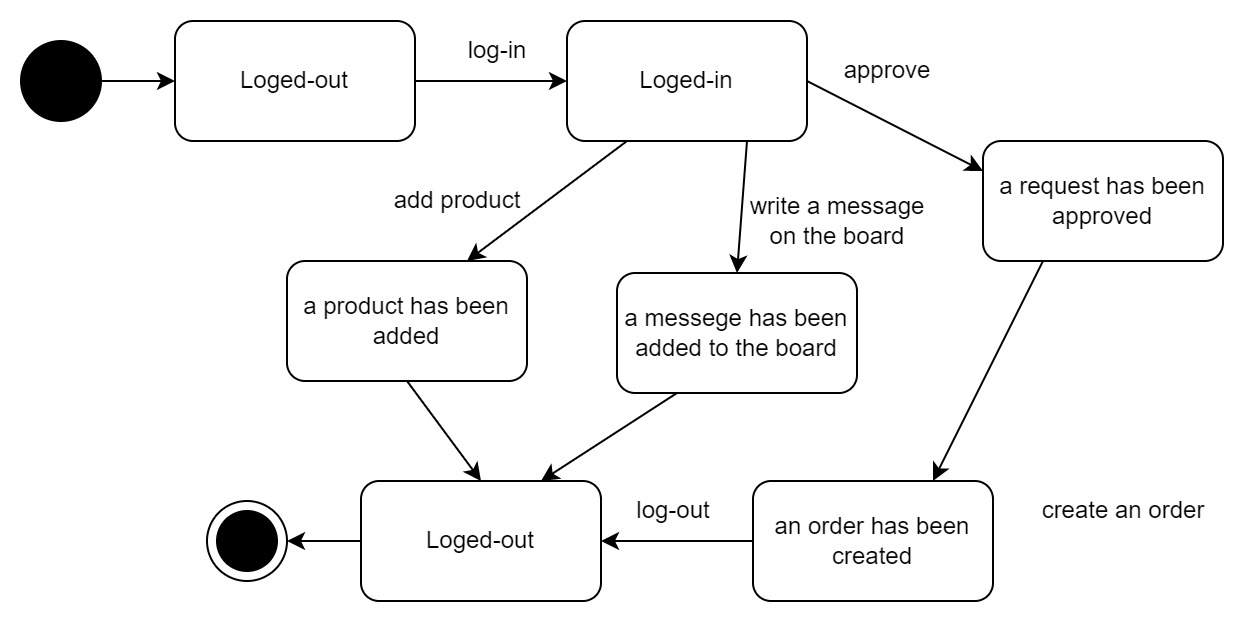
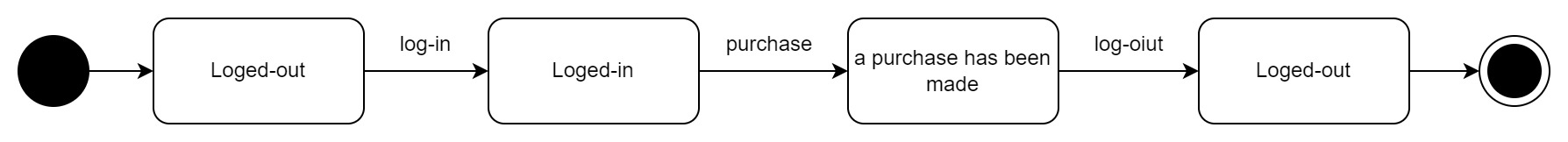
  Description automatically generated
* Create a request:  
  
* Create an order:  
  
* Create a message on a business’s message board:  
  
* Get report:  
  Diagram

  Description automatically generated

**4.2 Events**

* When starting the system, there will be connections to external systems, such as SAP. The system should integrate with currently used finance systems so that the transition to the new system will be in continuously to what had happened until now.
* When receiving a shutdown signal the program should write a message to the system logger, save all open buffers, close all open files, and terminate.
* When a kibbutz member buys a product in the on-site store, the store worker will enter that purchase in the store register interface, that will be added to the database. A message will be written to the system logger.
* When a business worker will publish a message to the message board, a notification will be sent as a broadcast to all the subscribed members.
* When a kibbutz member sends a request to a business, the business workers will receive a notification about the new request.
* When a business worker creates an order or updates its details, the member whom it is his order, will receive a notification.

**4.3 States**

* A state chart that demonstrates a kibbutz member’s state:
* A state chart that demonstrates a business worker’s states:
* A state chart that demonstrates a store register’s states:

**Chapter 5 – Object Oriented Analysis**

**5.1 Design Patterns**

In the system we are creating we will use several design patterns:

* Observer: This is a behavioral design pattern that notifies subscribers about events that happen to the object there are observing. We will use it in a couple of cases. The first case is when a business receives a request from a kibbutz member, the business now creates an order with the details of the member’s request. For example, a member asks the maintenance business to fix a pipe that exploded in his backyard. The business worker reads that request and creates an order (with date, status, description, and price if the member must add money for the repair). The member will be notified that an order of his request was created. Another case is when a business advertises a message on its message board, the members will be notified about it.
* Singleton: This is a creational design pattern that assures a class has only one instance while providing a global access point to this instance. We will use it when authenticating a user’s logging in (any user) and managing the database. The use of a singleton is important because we would like to maintain a single object that is responsible for logging in and authentication, so there won’t be any duplicates, and therefore mismatches when handling creating data from the database.
* Factory: This is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. We will use it when sending reports to accounting. We don’t want to restrict accounting to a format of reports. There will be an interface named report- creator, and the classes that implement it are different types of reports. Using this design pattern will make changing the format of the reports easy.

**Chapter 6 – User Interface Draft**

Chart, bubble chart

Description automatically generatedGraphical user interface, application

Description automatically generated

A picture containing rectangle

Description automatically generatedGraphical user interface

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generatedA picture containing graphical user interface

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generated

Graphical user interface, text, application, chat or text message

Description automatically generated

Table

Description automatically generated

**Chapter 7 – Testing**

We will use a regular unit integration system. In addition, we will test with CI/CD scheme. CI/CD stands for Continuous Integration/Continuous Deployment. It is a software engineering practice that aims to minimize the time between writing code and releasing it to users by automatically building, testing, and deploying code changes.

The goal of CI/CD is to enable developers to make small, frequent changes to their codebase, and have those changes automatically built, tested, and deployed to production, without the need for manual intervention. This helps to speed up the development process and reduce the risk of errors and bugs.

To implement CI/CD, developers use tools such as Jenkins, Travis CI, and CircleCI, which automatically build and test code changes, and then deploy them to production if they pass the tests. This allows developers to focus on writing code, rather than worrying about the deployment process.