**Perfume Shop**

Student: Vulsan Bianca Maria

Coordinating teacher: Iordan Anca

Table of contents

|  |  |  |
| --- | --- | --- |
| 1 | Problem’s context................................................................................ | 3 |
| 2 | Used tools............................................................................................ | 3 |
|  | 2.1 Java................................................................................................. | 3 |
|  | 2.2 JavaFx............................................................................................. | 4 |
|  | 2.3 JUnit............................................................................................... | 4 |
| 3 | Programming language choice motivation...........................................  Domain Driven Design………………………………………………………………………  SOA…………………………………………………………………………………………………. | 5 |
| 4 | Diagrams.............................................................................................. | 6 |
| 5 | Application description........................................................................ | 7 |
|  | 5.1 Client Server.................................................................................. | 7 |
|  | 5.2 Database communication.............................................................. | 8 |
|  | 5.3 Database provider.......................................................................... | 9 |
|  | 5.4 Available operations....................................................................... | 9 |
| ­­ | 5.5 Security........................................................................................... | 10 |
|  | 5.6 Observer Pattern………………………………………………………………………… | 11 |
|  | 5.7 Singleton Pattern………………………………………………………………………..  5.8 Mediator Pattern………………………………………………………………………..  5.9 Specification Pattern…………………………………………………………………..  5.10 Bridge Pattern………………………………………………………………………….. |  |
| 6 | Conclusion............................................................................................ | 13 |

1. Problem’s context

The objective of this topic is to familiarize with the Model-View-Controller architectural pattern, but also with the Observer behavioral design pattern.

Develop an application that can be used in a chain of perfume stores. The application will have 3 types of users: an employee of a perfume store, a manager of the chain of perfume stores, and an administrator.

After authentication, employee users can perform the following operations:

* Filtering perfumes by the following criteria: manufacturer, availability, price;
* Viewing the list of all the products from the shop where he works;
* Searching for a product by name;
* CRUD operations regarding the persistence of perfumes in the store where that employee works;
* Save the list of products from the shop in different formats: csv, json, xml, txt.

After authentication, manager users of the chain of perfume stores can perform the following operations:

* Viewing the list of all perfumes in a selected store sorted by the following criteria: name and price;
* Filtering perfumes by the following criteria: shop, manufacturer, availability, price;
* Searching for a perfume by name;
* Save the list of products in different formats: csv, json, xml, txt;
* Visualize statistics about the products in the chain.

After authentication, administrator users can perform the following operations:

* CRUD operations for user-related information;
* Viewing the list of all users;
* Filter the list of user by the role column.

The graphical interface of the application should be available in 2 different international languages.

1. Used tools

The proposed solution was developed using the Java programming language. The Java framework that was used for the graphical interface is JavaFx. Testing of the database operations was performed using Junit.

2.1. Java

Java is a popular object-oriented programming language, widely used for its capabilities in developing a range of application types, from desktop and web applications to mobile and even embedded applications.

One of the main features of Java is its platform independence. Java code can be compiled into bytecode that can run on any platform that has a Java Virtual Machine (JVM). This means that a Java program can run on different operating systems, such as Windows, macOS, and Linux, without needing to recompile the code for each platform.

Java also has a large and active community of developers who contribute to its libraries and frameworks. This makes it easy to find and use third-party libraries for common tasks, such as working with databases or creating web applications. Some popular Java libraries and frameworks include Spring, Hibernate and so on.

Another strength of Java is its security model. Java provides a sandbox environment that can prevent malicious code from accessing system resources or harming the user's system. Java also provides support for encryption, digital signatures, and other security features, which make it a popular choice for developing secure applications.

In addition to these features, Java also has a strong focus on performance and scalability. Java programs can use multiple threads to take advantage of multi-core processors and handle large amounts of data. Java also provides tools for profiling and optimizing code to improve its performance.

Overall, Java is a versatile and popular programming language with many strengths, including platform independence, a large community, a strong security model, and performance and scalability features. Its popularity and versatility have made it a popular choice for developing a wide range of applications, from desktop and web applications to mobile and embedded systems.

2.2. JavaFx

JavaFx is a popular framework for developing desktop applications with a user interface (UI) and provides a set of tools and libraries to make it easier to create and manage UI components. Some of the features that JavaFx offers include support for multimedia, web technologies, 2D and 3D graphics, and animations.

By using JavaFx, we can create a desktop application that can run on different operating systems, including Windows, macOS, and Linux. JavaFx also provides a wide range of UI controls, such as buttons, text fields, combo boxes, tables, and charts, which you can use to create an intuitive and interactive user interface. This way, we can create different views and UI components to implement these functionalities and provide a seamless user experience.

2.3. JUnit

JUnit is a popular testing framework for Java, which provides a set of tools and libraries to make it easier to write and run automated tests for Java code. JUnit is widely used by Java developers to ensure the correctness and reliability of their code.

JUnit follows the principle of "test-driven development" (TDD), which involves writing tests before writing the actual code. With JUnit, you can write test cases that define the expected behavior of your code, and then run those tests to verify that the code behaves as expected. This helps to catch errors and bugs early in the development process and can save time and effort in the long run.

JUnit provides a wide range of tools and features to make it easier to write and run tests. Some of the key features of JUnit include:

* Annotations: JUnit uses annotations to mark methods as test cases, setup methods, and teardown methods.
* Assert statements: JUnit provides a set of assert statements that you can use to check the expected behavior of your code.
* Test suites: JUnit allows you to group test cases into test suites, which can be run together.
* Test runners: JUnit provides several test runners that can run your tests and report the results.

In addition to these features, JUnit also supports integration with other testing tools, such as Mockito for mocking dependencies and Selenium for testing web applications.

1. Programming language choice motivation

The choice of using Java as the programming language for this project was motivated by several factors, among them having the following: the Object-Oriented nature of the Java programming language, allowing the division of complex tasks into small classes with clearly defined purposes, over already established patterns. The cross-platform compatibility was also a factor to be taken into consideration for further development of the system. Some other significant attributes of Java consisted of the security provided by the language, but also the wide range of tools and frameworks available for any given goal.

For our specific purpose, developing the system for a chain of stores selling perfumes came naturally, taking into consideration all the previous factors, each of them contributing to one of the sub-goals the system aims to achieve.



1. Domain Driven Design

Domain-Driven Design (DDD) is an approach to software development that focuses on understanding and modeling the business domain in the software design process. It emphasizes creating software systems that closely align with the business domain, making the software more effective in solving real-world problems.

Here are some key concepts and principles of Domain-Driven Design:

Ubiquitous Language: DDD promotes the use of a common language shared by all team members, including business stakeholders, domain experts, and developers. This shared language helps ensure a clear and consistent understanding of the domain concepts and enables effective communication.

Bounded Context: DDD suggests dividing a large system into smaller Bounded Contexts, each representing a specific domain area with its own models, rules, and terminology. Bounded Contexts help in managing complexity and maintaining clear boundaries between different parts of the system.

Entities: Entities represent domain objects with unique identities and have a lifecycle. They encapsulate behavior and state, enforcing domain-specific invariants and ensuring consistency within the domain.

Value Objects: Value Objects are objects that have no unique identity but are defined by their attributes or values. They are immutable and can be shared and compared by their values. Value Objects are often used to represent concepts like dates, addresses, or measurements.

Aggregates: Aggregates are clusters of domain objects treated as a single unit. An Aggregate has an Aggregate Root, which is responsible for maintaining the consistency and integrity of the Aggregate. All interactions with objects within an Aggregate should go through its root.

Domain Services: Domain Services encapsulate domain logic that does not naturally fit into an entity or value object. They provide operations and behaviors that involve multiple entities or cross-aggregate operations.

Domain Events: Domain Events represent something that has happened in the domain and is of interest to other parts of the system. They are used to communicate and decouple different parts of the system, enabling eventual consistency and asynchronous processing.

Strategic Design: DDD also includes strategic design aspects that deal with the overall architecture and organization of the software system. It involves identifying bounded contexts, defining context maps to manage relationships between contexts, and establishing clear and explicit boundaries.

DDD emphasizes the collaboration between domain experts and software developers to gain a deep understanding of the business domain and create a software system that closely aligns with the business requirements. By applying DDD principles and patterns, developers can build more maintainable, scalable, and flexible systems that better reflect the needs of the business

1. SOA

SOA, or Service-Oriented Architecture, is an architectural style that promotes the design and development of software systems as a collection of loosely coupled and interoperable services. In an SOA, services are the fundamental building blocks that encapsulate discrete functionality and can be accessed and used by other services or client applications.

Here are some key characteristics and concepts associated with SOA:

Services: Services are self-contained units of functionality that provide well-defined interfaces for interaction. They can be granular and focused on specific tasks or business processes. Services are designed to be modular, reusable, and independent of the underlying implementation technologies.

Loose coupling: SOA emphasizes loose coupling between services, meaning that each service operates independently and communicates with others through well-defined interfaces. This loose coupling allows for flexibility, scalability, and easier maintenance and updates of individual services.

Service contract: Services in an SOA adhere to a service contract, which defines the interface, behavior, and communication protocols. The contract specifies how the service can be accessed, what data formats are used, and any required security or quality of service attributes.

Service discovery: In an SOA, services need to be discoverable, meaning that other services or client applications can easily find and utilize them. Service discovery mechanisms include service registries, directories, or service brokers that maintain information about available services and their capabilities.

Service composition: SOA enables the composition of services to create larger and more complex applications or business processes. Services can be combined and orchestrated to achieve specific goals or workflows. This composition can be dynamic and flexible, allowing for the adaptation and reconfiguration of services at runtime.

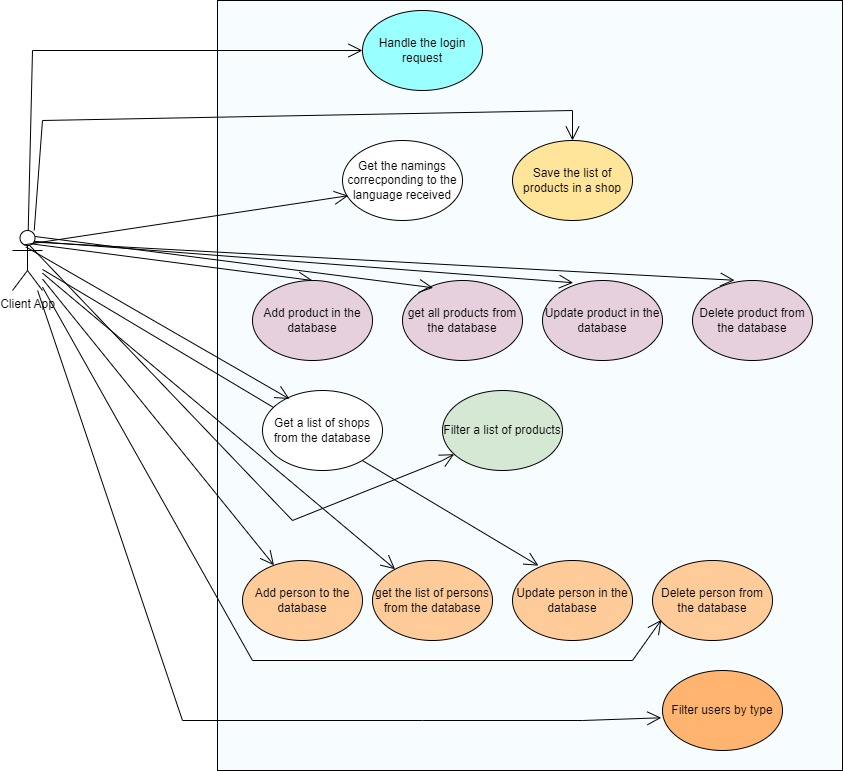
Interoperability: SOA promotes interoperability between different systems and technologies. Services can be implemented using various programming languages, platforms, or frameworks, as long as they adhere to the defined service contract. This allows for the integration of heterogeneous systems and promotes system integration and collaboration.

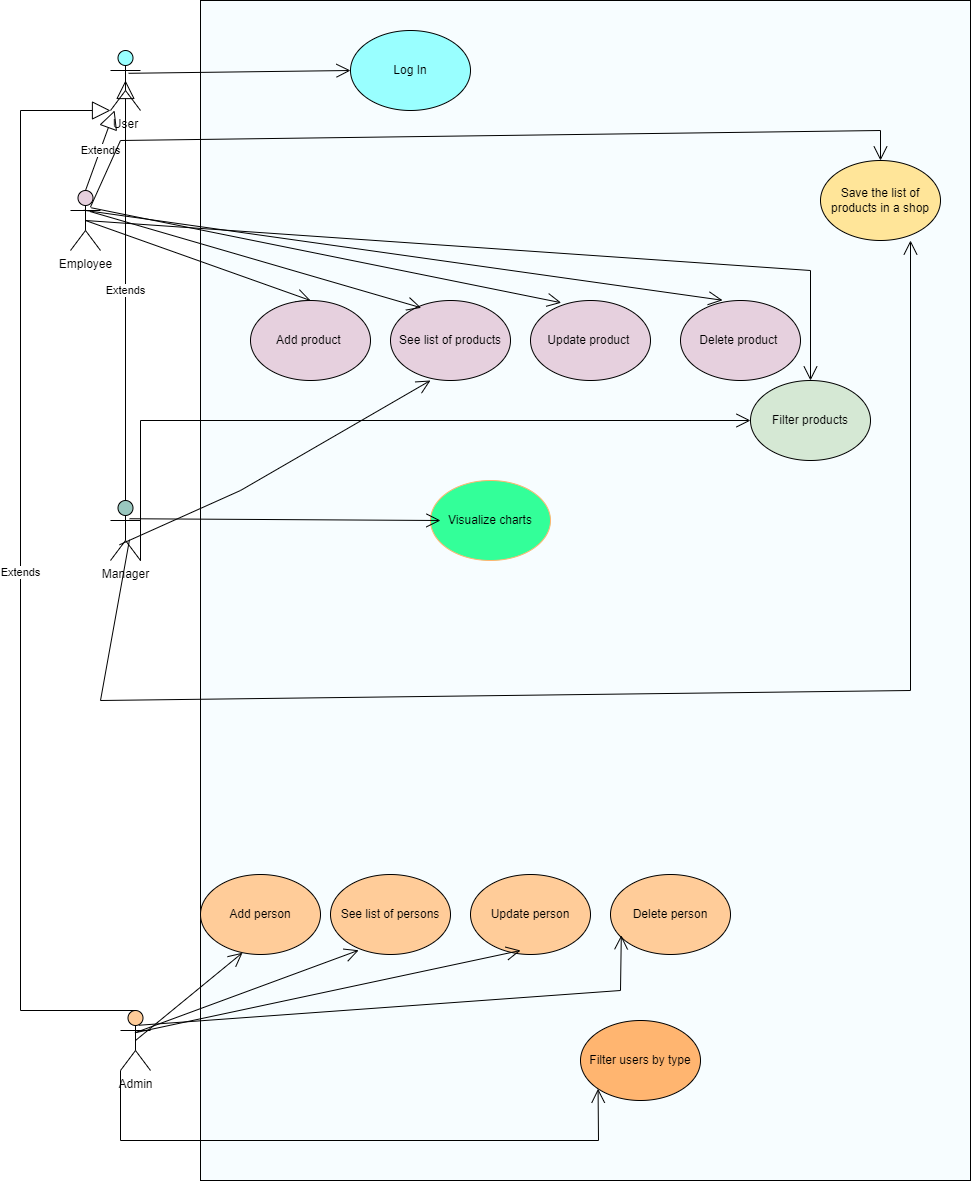
Governance: SOA typically involves governance mechanisms to manage the lifecycle of services, ensure adherence to standards and policies, and monitor service performance and quality. Governance includes aspects such as service design, development, deployment, versioning, security, and monitoring.

SOA has been widely adopted as an architectural approach for building complex and distributed systems, especially in enterprise and cloud computing environments. It provides benefits such as modularity, scalability, reusability, and the ability to integrate diverse systems and technologies. However, successful implementation of SOA requires careful design, planning, and consideration of factors such as service granularity, service identification, and governance processes.

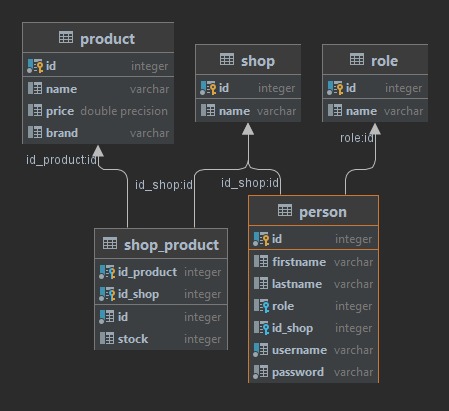
1. Diagrams

4.1. Use case diagram



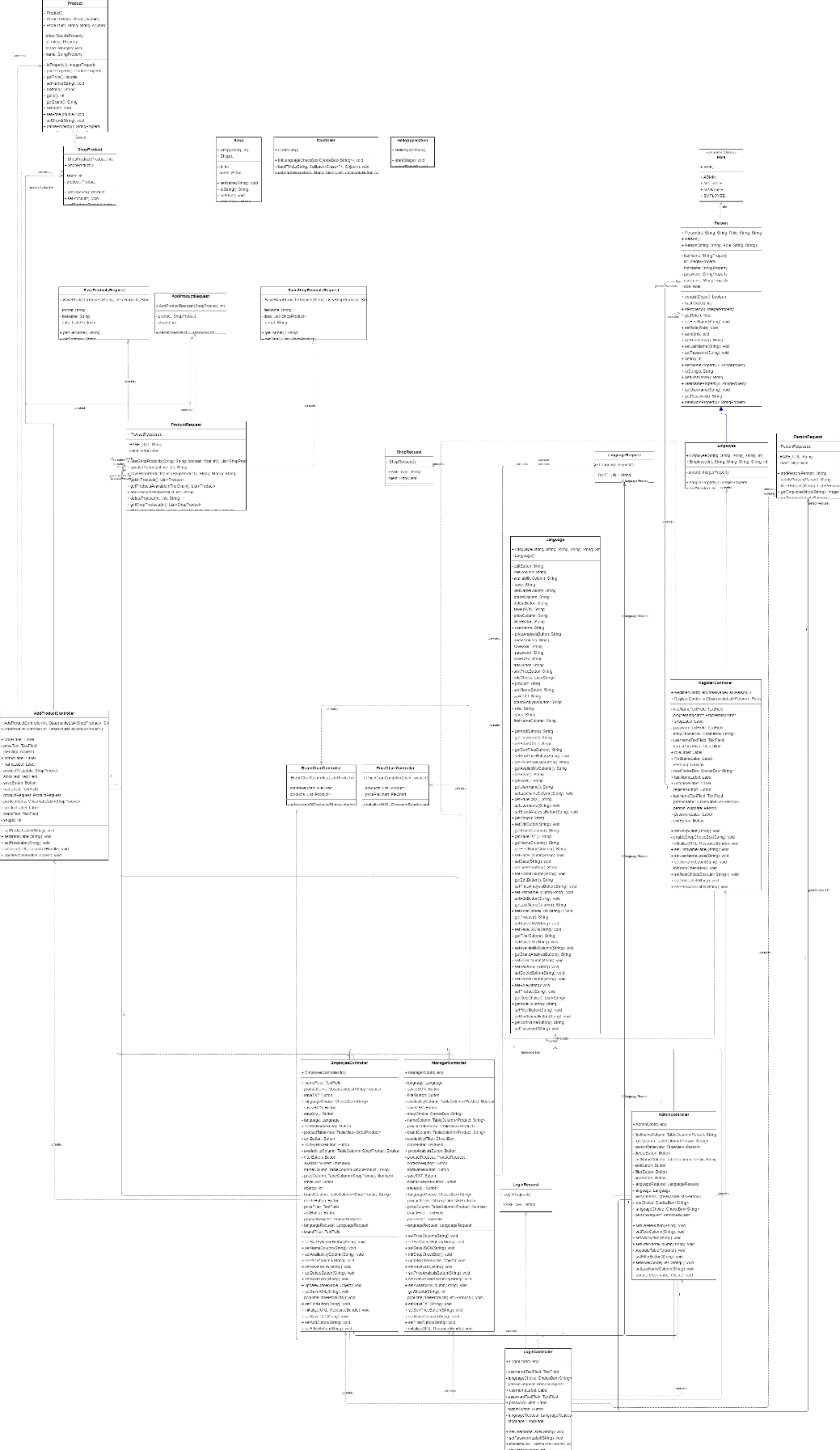


4.2. Database diagram

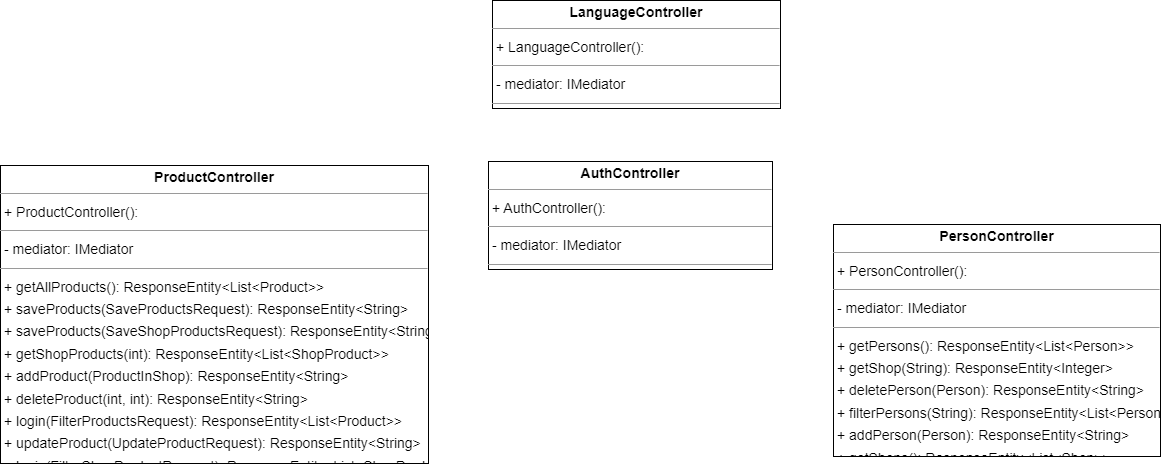


4.3. Class diagram

4.3.1. Client



4.3.2. Part of server

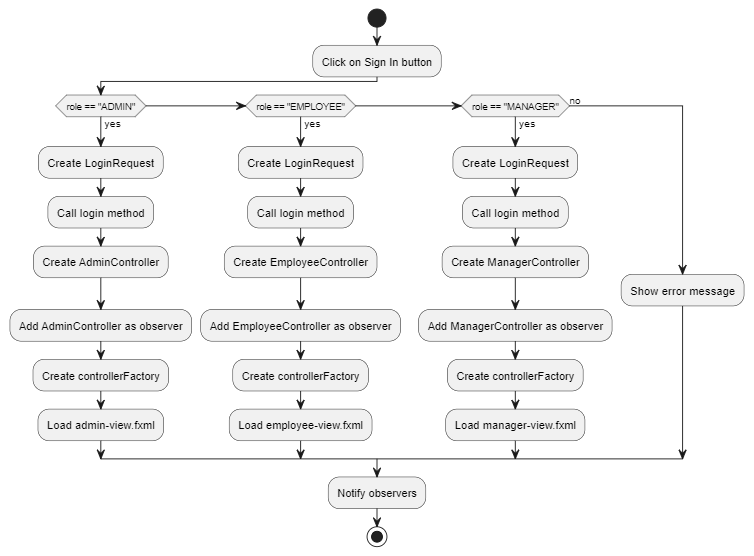




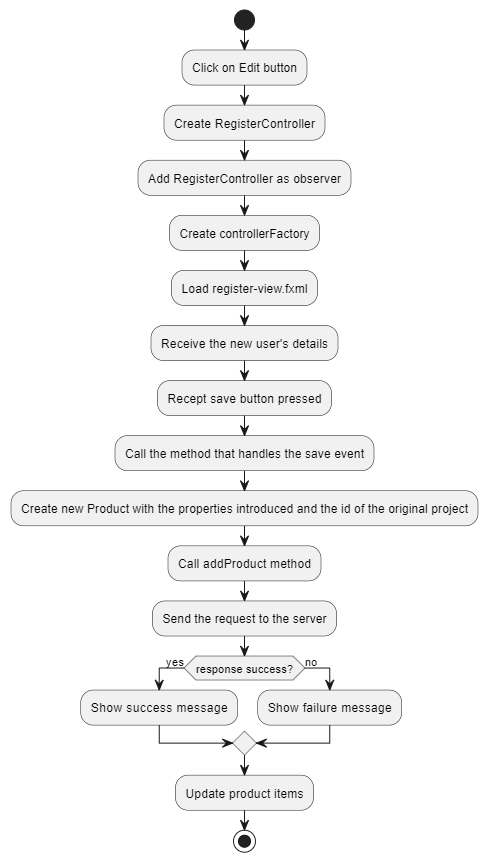
4.4. Activity Diagrams

4.4.1. On the Client Side

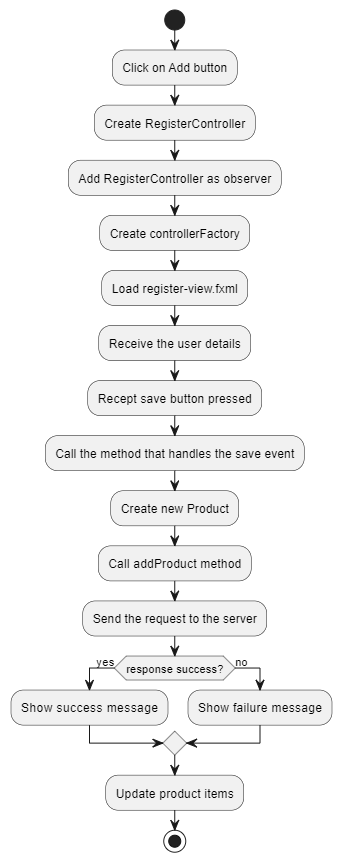
Log in



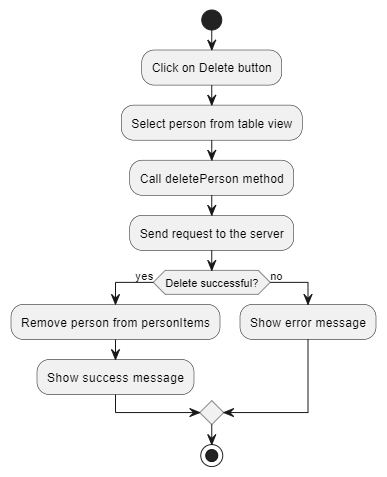
Admin updates person



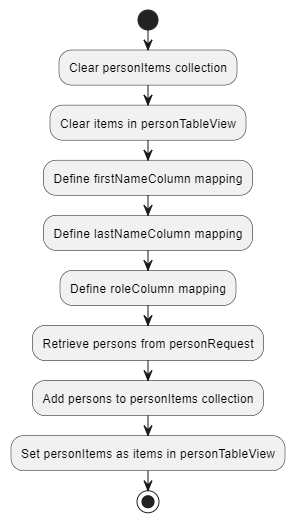
Admin adds person in the system



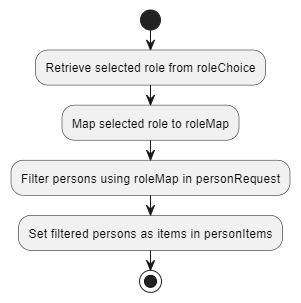
Admin deletes person from the system



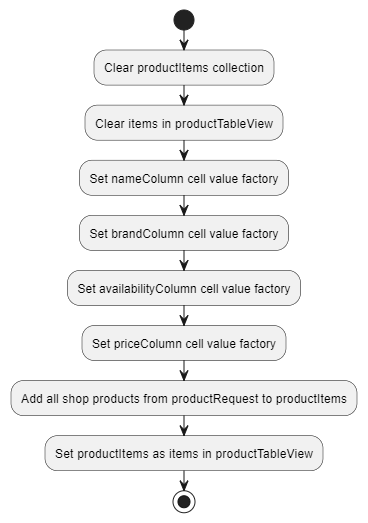
Admin sees the list of persons



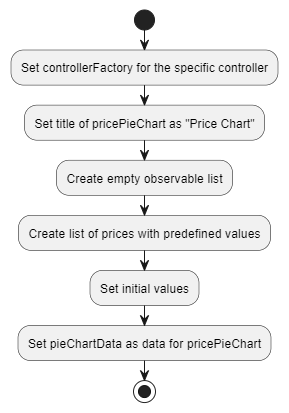
Admin filters the list of persons



See the list of products



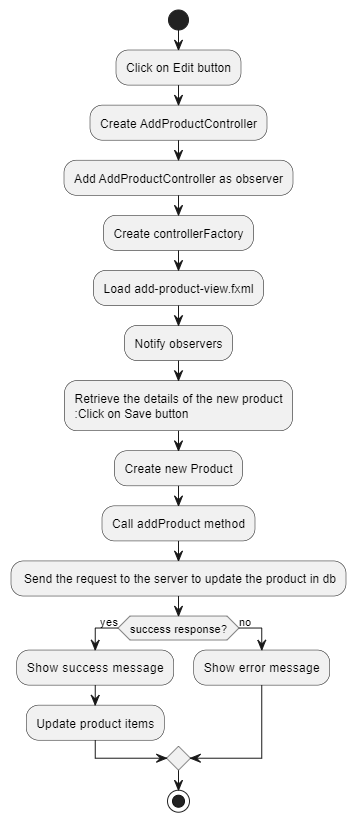
Manager visualizes the statistical analysis



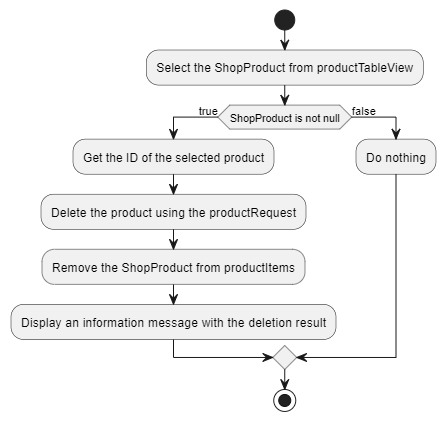
Add product



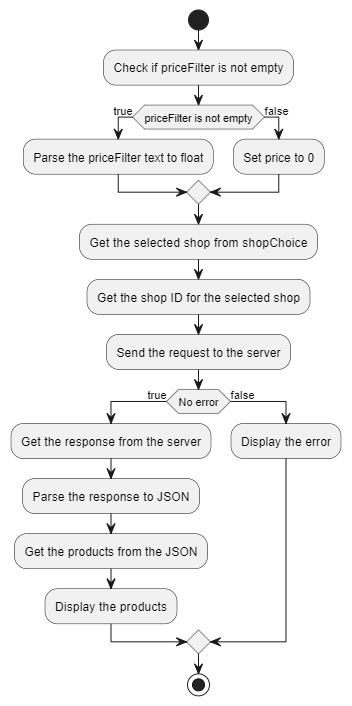
Update product



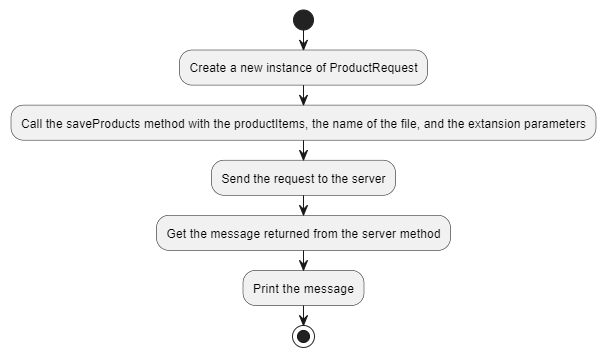
Remove product



Filter Products

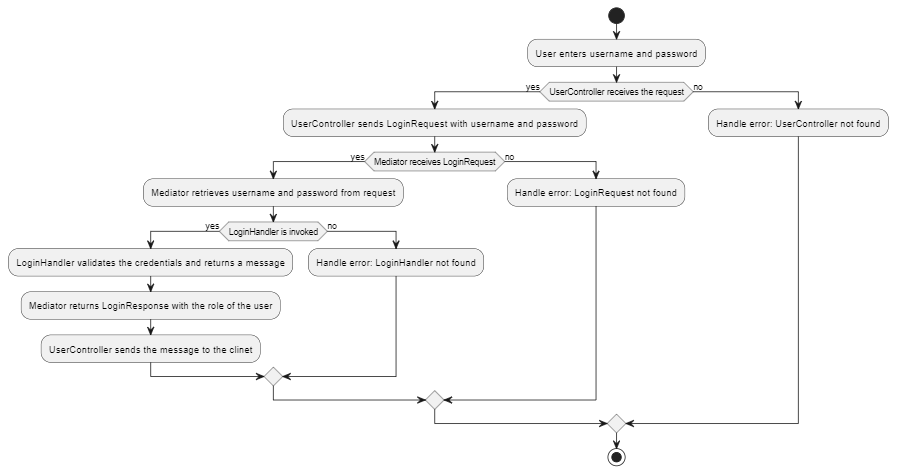


Save the list of products

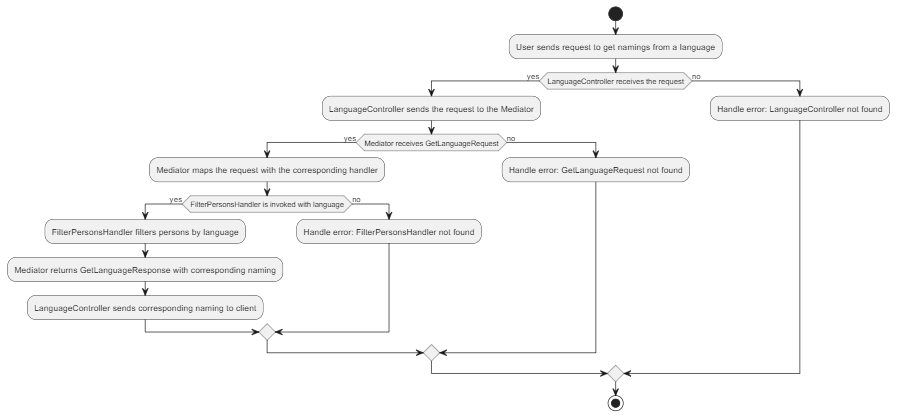


4.4.2. On the Server Side

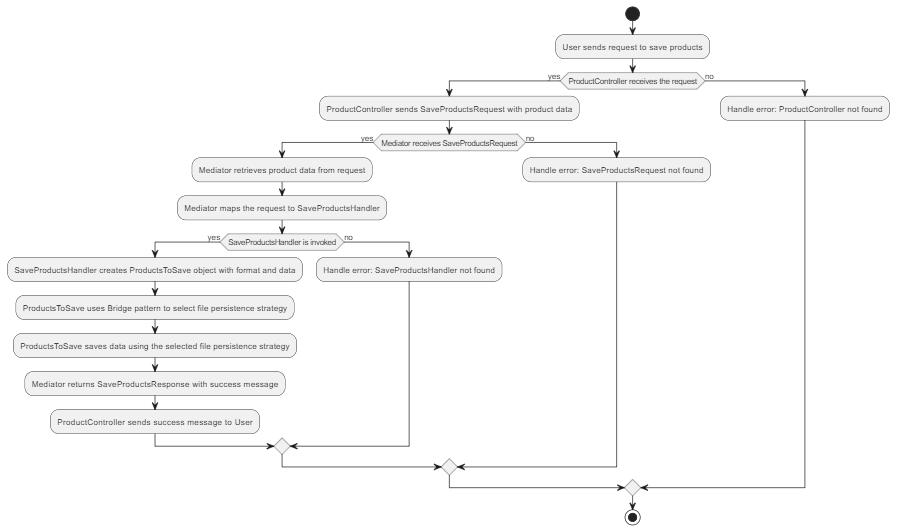
Handle login request



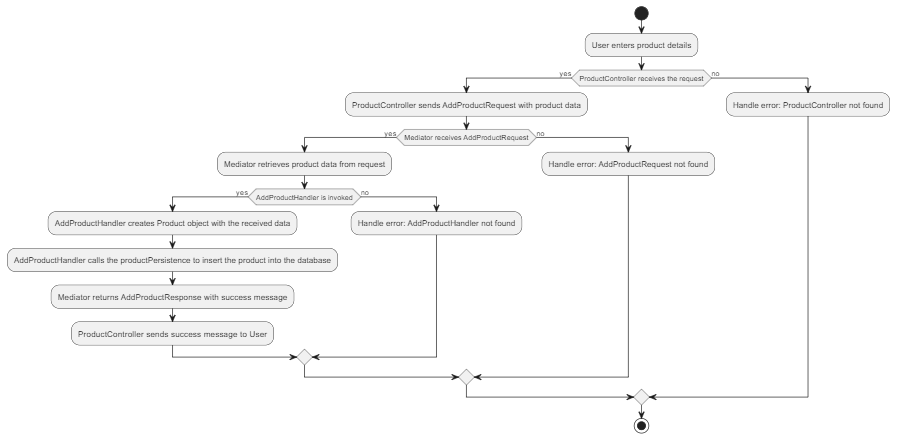
Get the naming corresponding to the language received



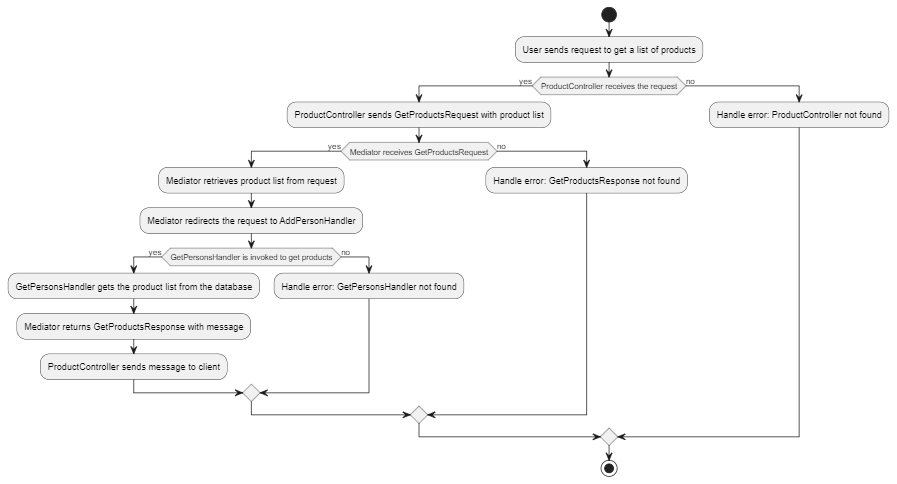
Save the list of products in a shop



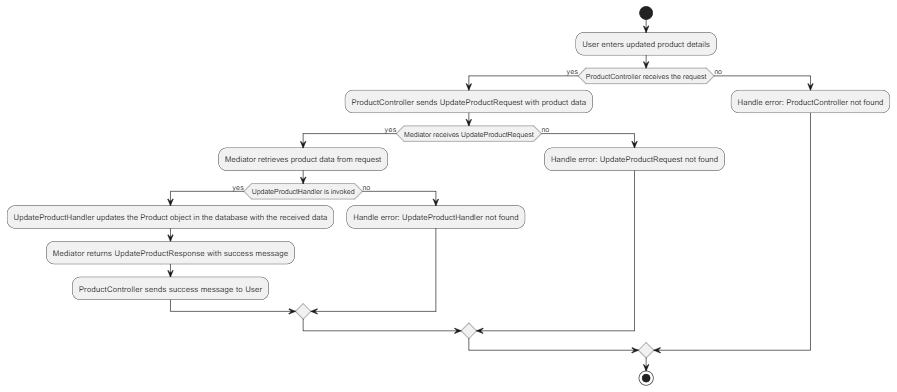
Add product in the database



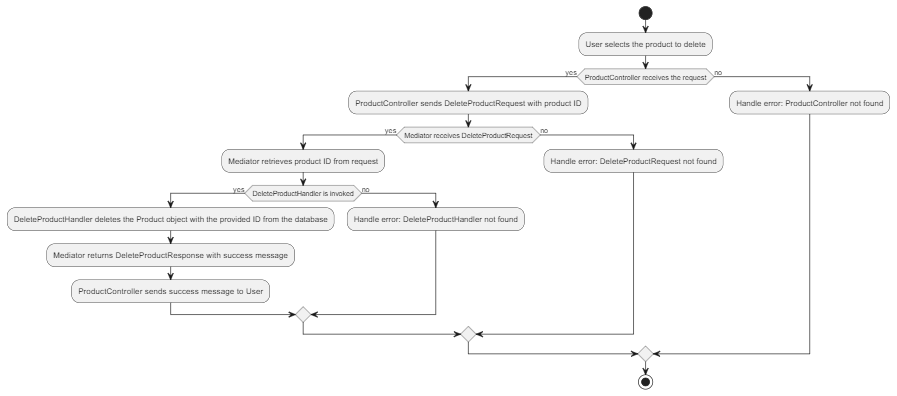
Get all products from database



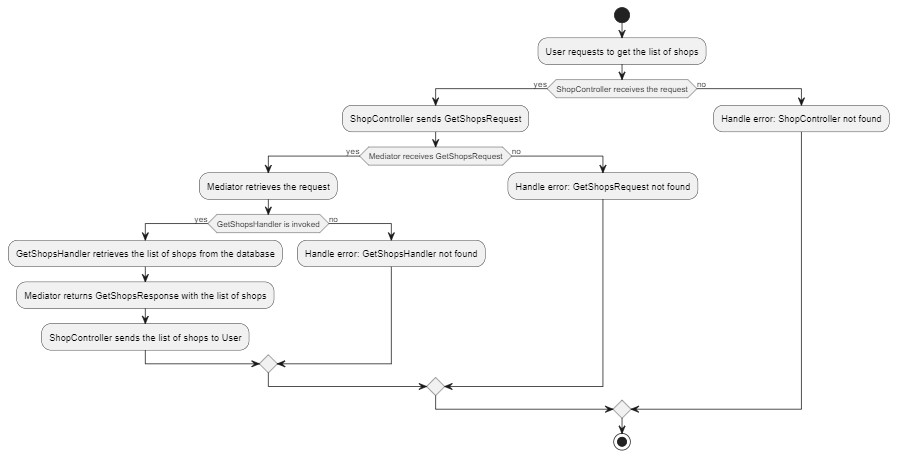
Update product in the database



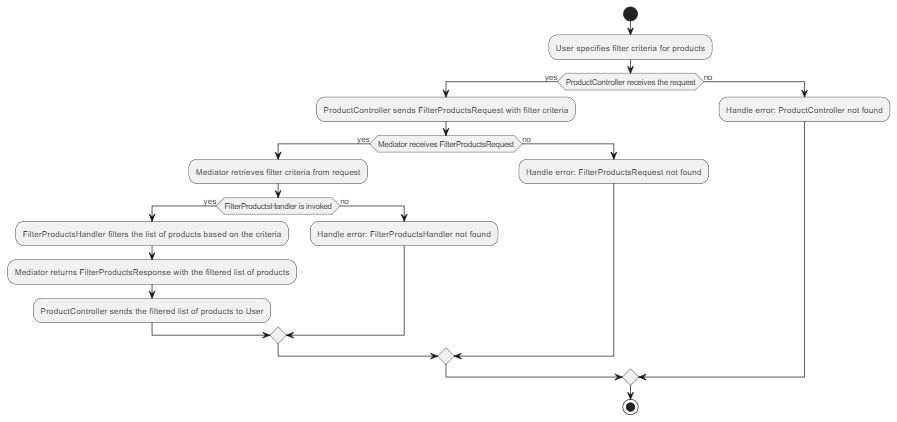
Delete product from the database



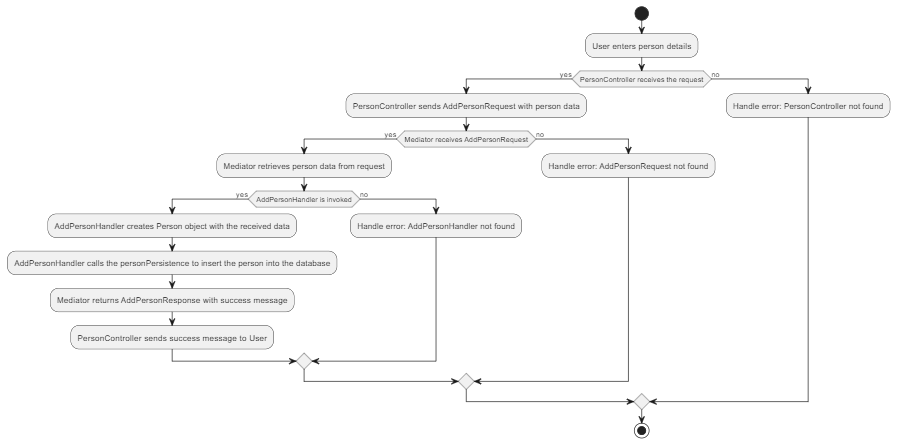
Get a list of shops from the database



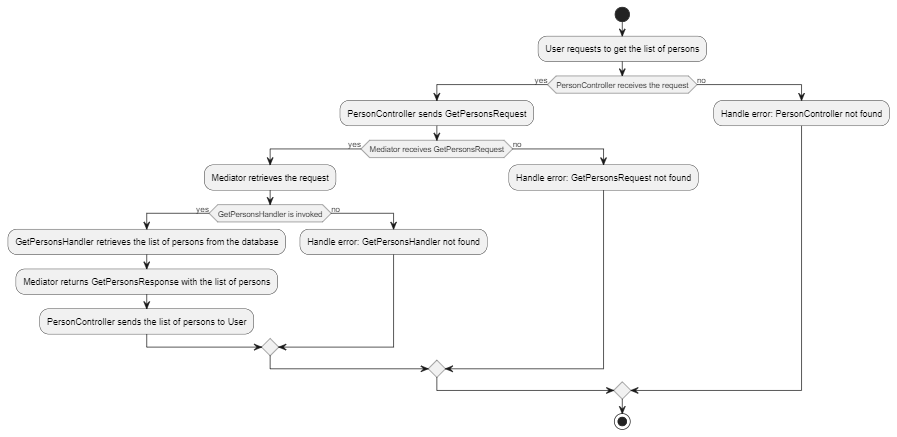
Filter a list of products



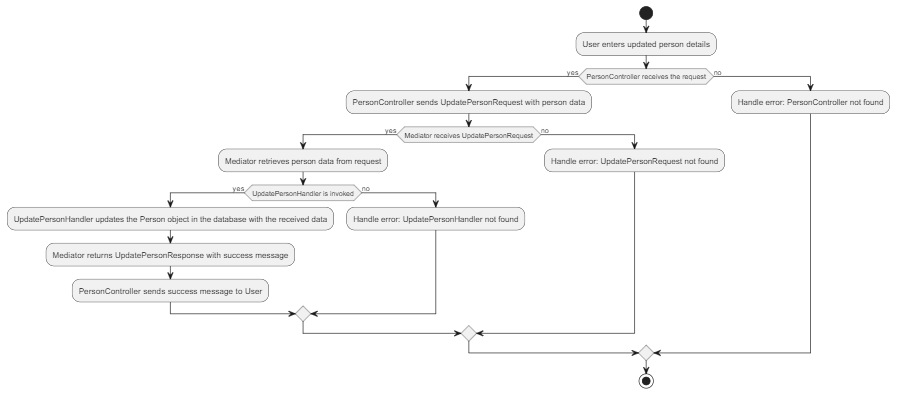
Add person to the database



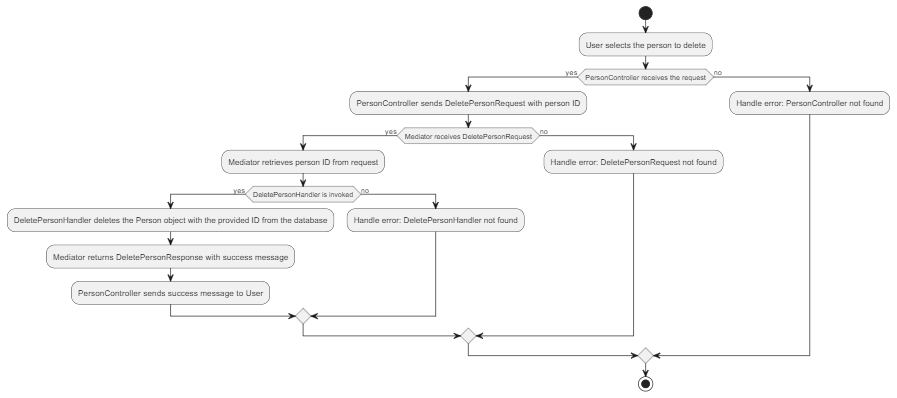
Get the list of persons from the database



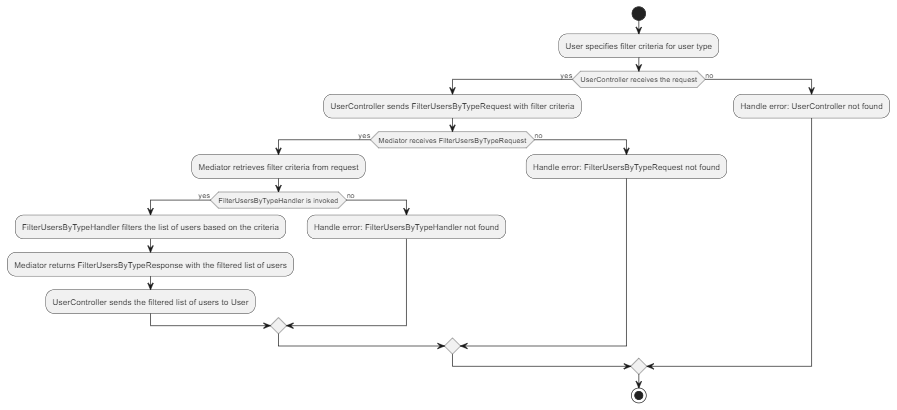
Update person in the database



Delete person from the database



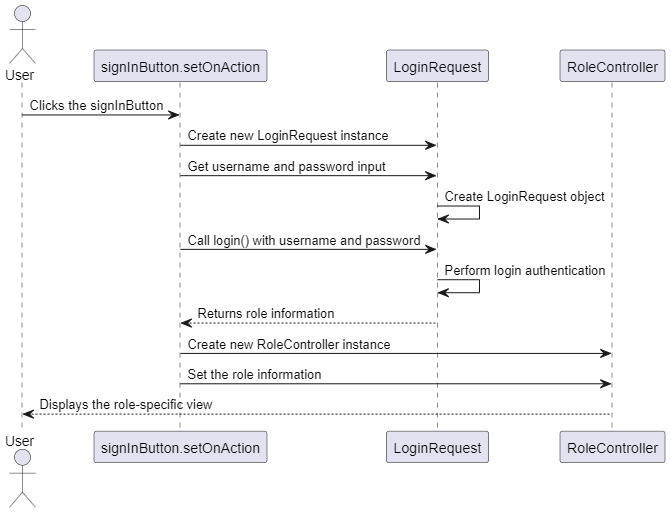
Filter users by type



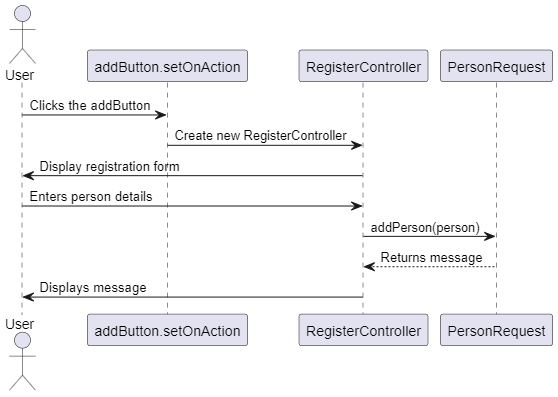
4.5. Sequence diagrams

4.5.1. On the Client Side

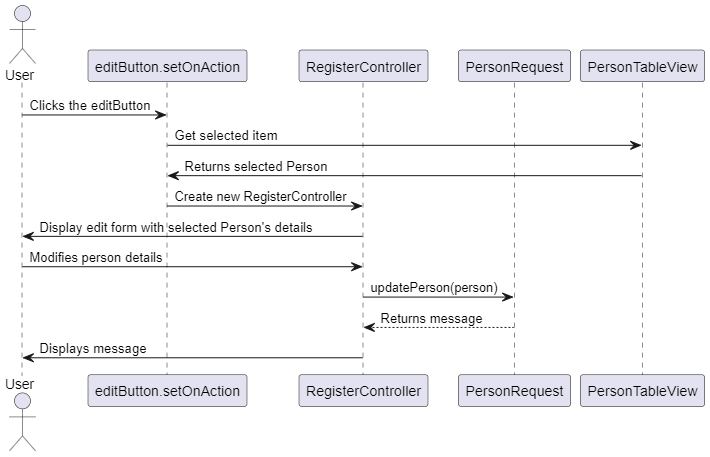
Log In



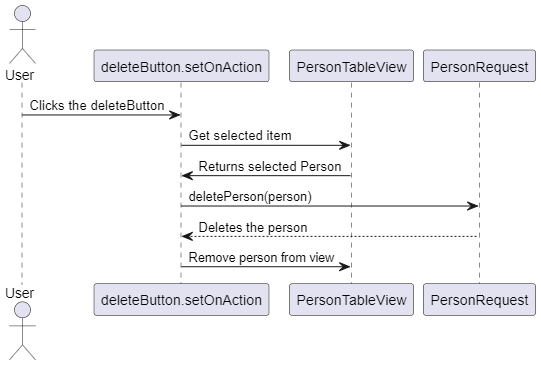
Add person



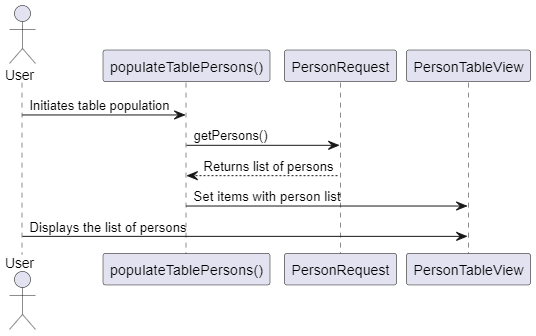
Edit person



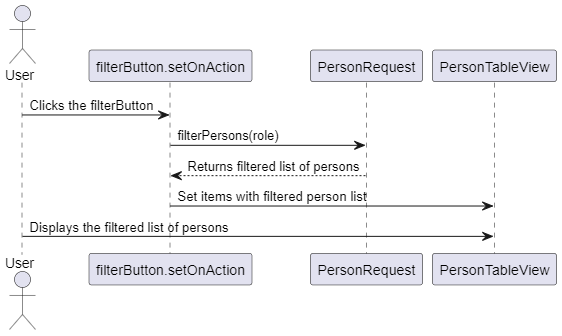
Remove person



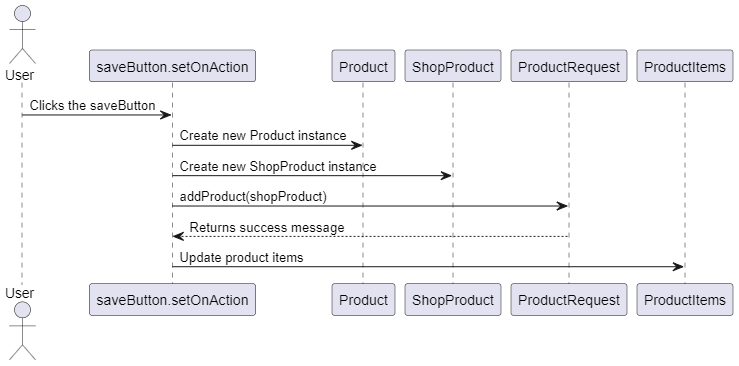
See list of persons



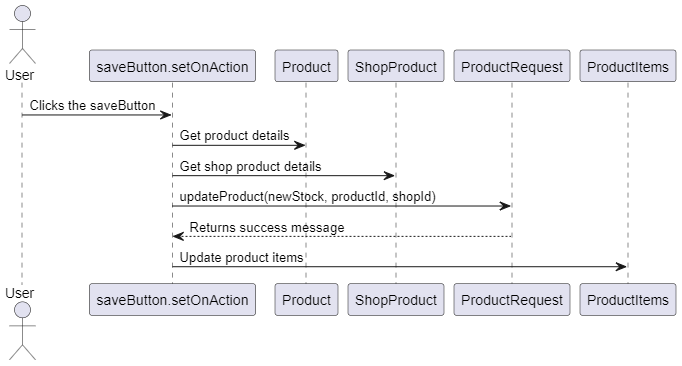
Filter persons by role



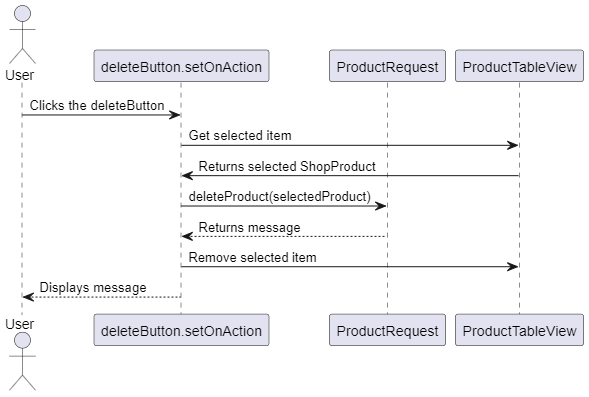
Add product



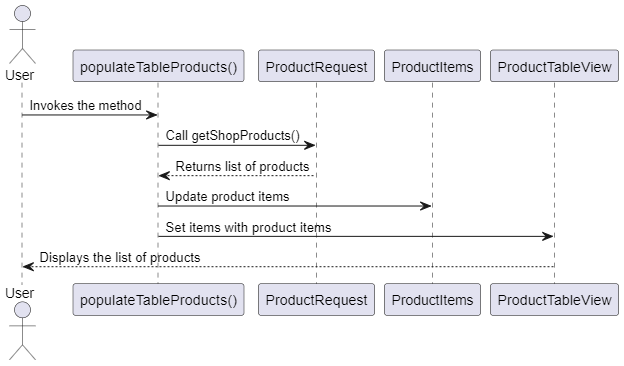
Edit product



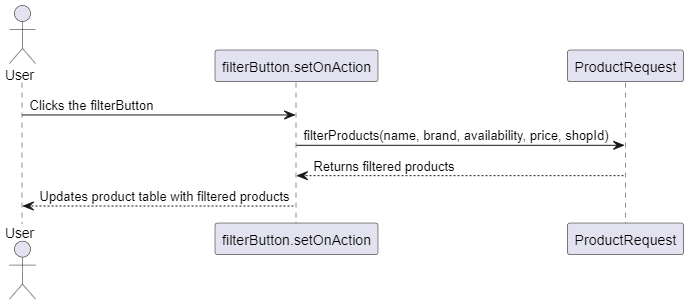
Delete product



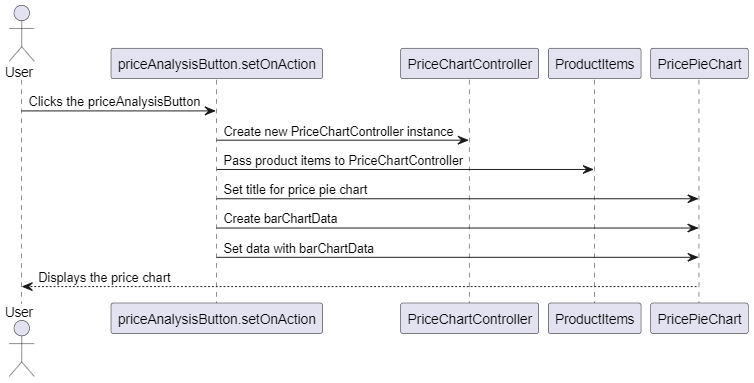
See list of products



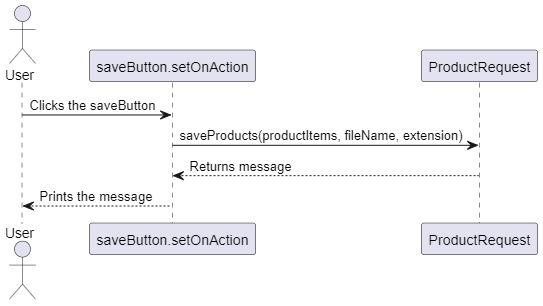
Filter products



Visualize chart

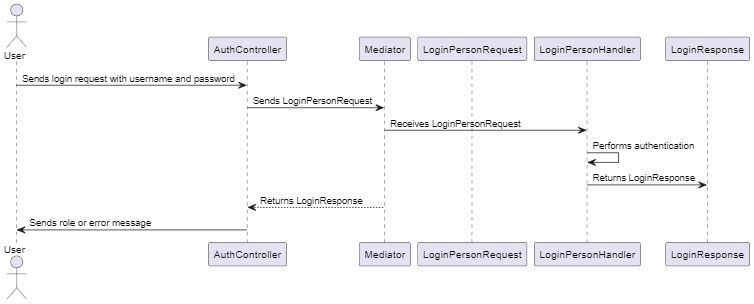


Save the list of products

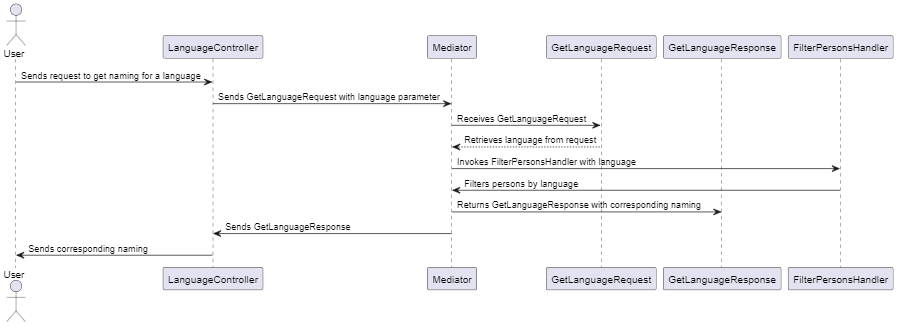


4.5.2. On the Server Side

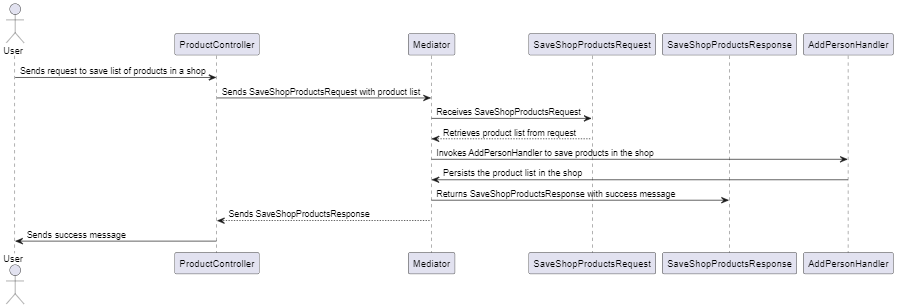
Handle login request



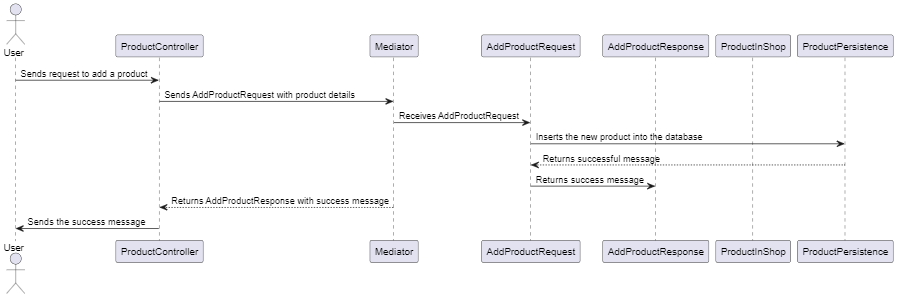
Get the naming corresponding to the language received



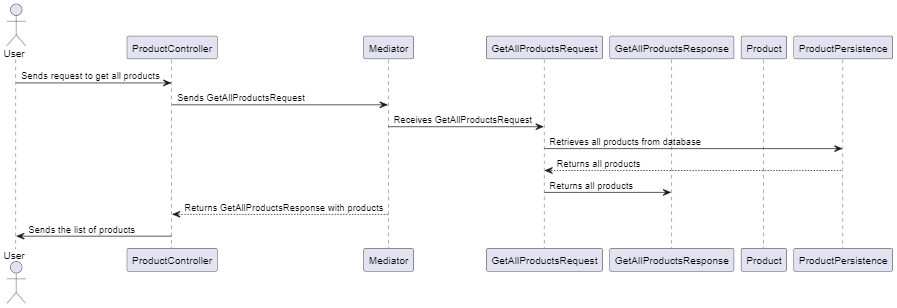
Save the list of products in a shop



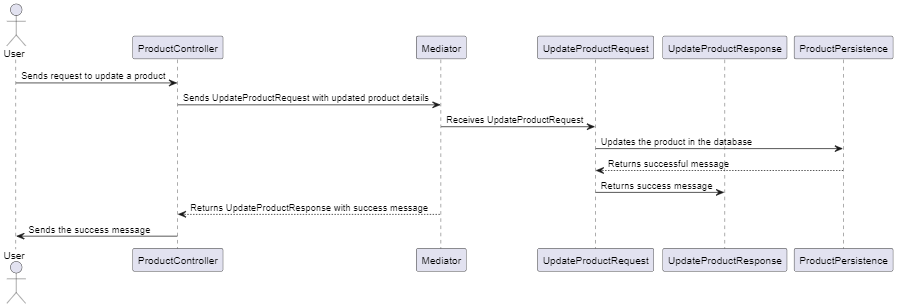
Add product in the database



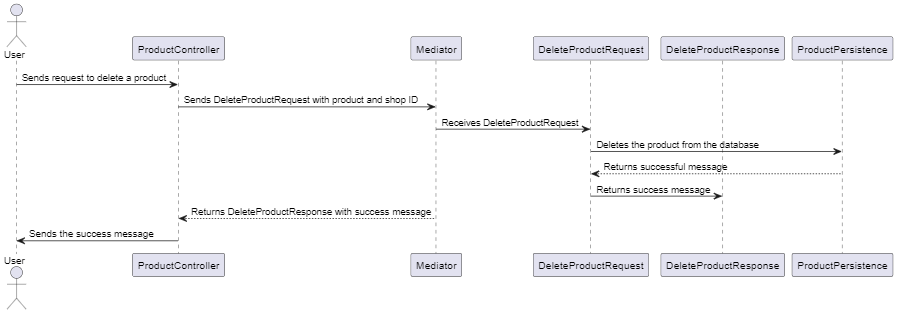
Get all products from database



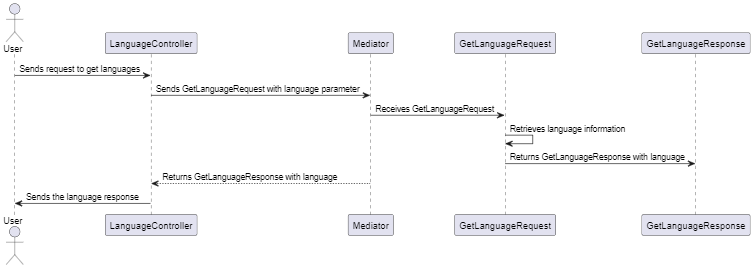
Update product in the database



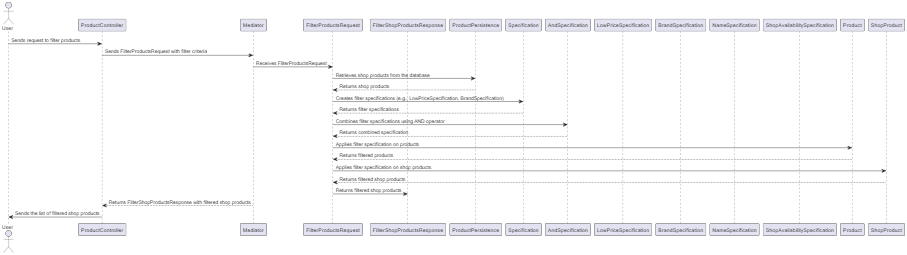
Delete product from the database



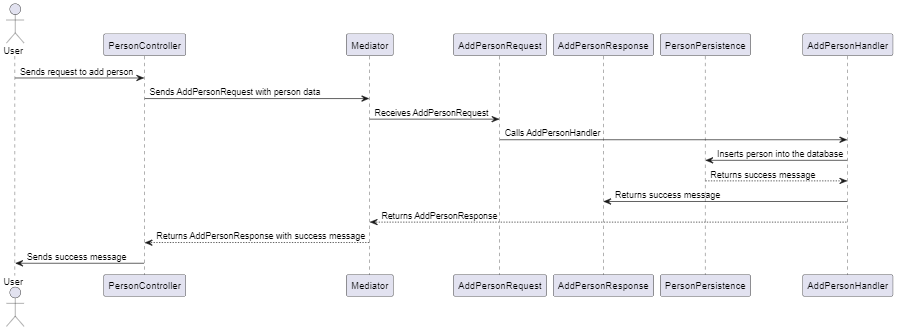
Get a list of shops from the database



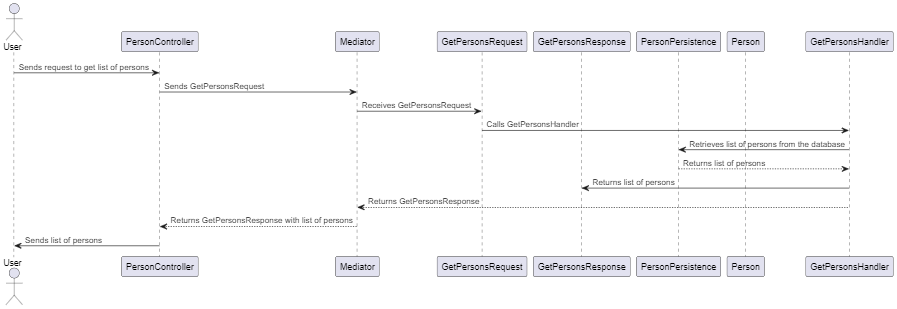
Filter a list of products



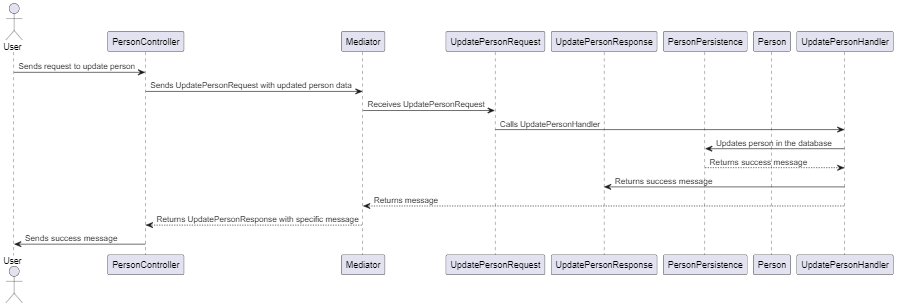
Add person to the database



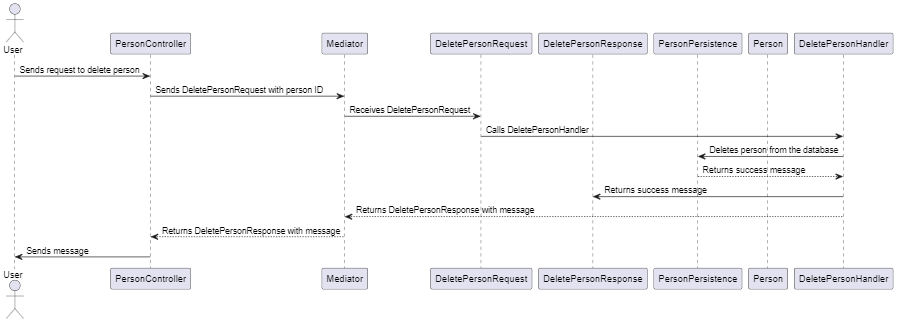
get the list of persons from the database



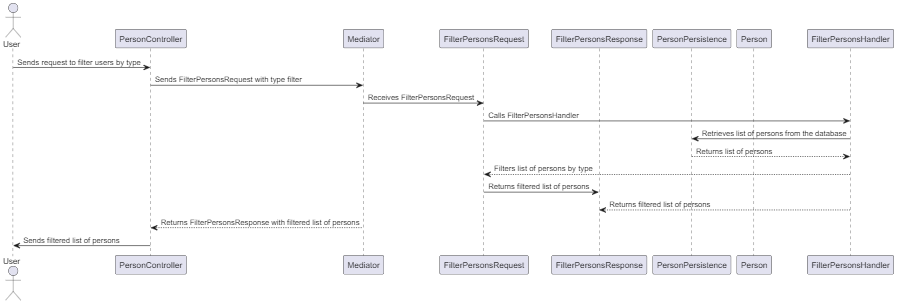
Update person in the database



Delete person from the database



Filter users by type



1. Application description

As it was already described in the previous chapters of the documentation, the goal of the provided solution is to allow different roles (employee, manager, administrator) of a perfume shop chain to perform specific tasks for their assigned role. For this matter, the pattern that was used for developing the application was MVC: Model View Controller.

5.1. Client Server

5.2. Database communication

For the database communication, we used the persistence model. The persistence pattern is a design pattern used to separate the storage and fetching logic from the rest of the application. The main goal of this pattern is to provide an abstract interface to the application for interacting with the database, which reduces coupling between the application and the database, and makes it easier to switch between different database systems.

In this pattern, an implementation is defined for each entity in the application that needs to be persisted and it provides the logic for interacting with the database. It defines the methods for interacting with the database, such as creating, reading, updating, and deleting records, together with the logic for actually performing these operations.

In our perfume shop chain program, you used the persistence pattern to communicate with the database, separating the persistence logic from the rest of the application. We defined an implementation for each entity in the application that needed to be persisted, such as the Product and Person entities and we implemented them to provide the logic for interacting with the database.

For example, the ProductPersistence interface defined the methods for creating, reading, updating, and deleting perfume (product) records in the database and the logic for performing these operations. Similarly, the PersonPersistence class defined the methods for interacting with the User (general role for each person in the system) records in the database.

Using the persistence pattern in our application made it more modular and easier to maintain, as the persistence logic was separated from the rest of the application, and each persistence implementation could be developed and tested independently. Additionally, by providing an abstract interface to the application for interacting with the database, we made it easier to switch between different database systems if necessary in the future.

5.3. Database provider

As for our database provider, we used PostgresSQL. An advantage of Postgres is its scalability. It is designed to handle large volumes of data and can be scaled horizontally across multiple servers to accommodate growing data needs. This makes it a good choice for businesses that anticipate significant growth in the future.

Postgres also has a robust set of features that can be used to build complex applications, such as the ability to create custom data types, stored procedures, and triggers. This flexibility makes it a popular choice for developers who need to build sophisticated applications that require advanced functionality.

In addition, Postgres has a strong focus on data integrity and reliability. It supports a variety of features to ensure data consistency, including transaction management, multi-version concurrency control, and referential integrity constraints. This makes it a good choice for applications where data accuracy is critical, such as in the perfume shop chain system where accurate inventory management is crucial.

5.4. Available operations

The perfume shop chain application features a multi-user system with three types of users: employees, managers, and administrators. After authentication, each user type has access to specific operations and functionalities.

Employee users are authorized to perform filtering operations on perfumes based on manufacturer, availability, and price. They are also authorized to perform CRUD (Create, Read, Update, Delete) operations on perfume data specific to the store where they work. These operations allow employees to manage the inventory of perfumes and ensure that customers have access to the products they need.

Manager users, on the other hand, have access to a broader set of operations. They can view a list of all perfumes in a selected store, sorted by name or price. They are also authorized to filter perfumes by manufacturer, availability, and price, allowing them to manage inventory across multiple stores. In addition, managers can search for a specific perfume by name, facilitating quick access to important information.

Administrator users have the highest level of access in the system. They are authorized to perform CRUD operations on user-related information and can view a list of all users in the system. This functionality allows administrators to manage user accounts, grant and revoke access as needed, and maintain the security and integrity of the system.

5.5. Security

Login-based authentication and authorization is a common method used to secure applications by verifying the identity of users and controlling their access to resources. This method typically involves requiring users to provide a username and password to gain access to the application, which is then verified against a database of user credentials.

Authentication is the process of verifying a user's identity, while authorization determines what actions or resources a user is allowed to access. These processes are crucial for the security of the application, as they prevent unauthorized access to sensitive information and functionalities.

In the perfume shop chain system, login-based authentication and authorization is particularly important due to the sensitive nature of the data involved. For example, employees should only be able to view and modify data related to the specific store they work in, while managers and administrators should have broader access to view and modify data across multiple stores.

By requiring users to log in and verifying their credentials, the system can ensure that only authorized users are able to access sensitive information and functionalities. This helps to prevent unauthorized access, data breaches, and other security threats.

Additionally, login-based authentication and authorization can provide an audit trail of user activity, allowing administrators to track who accessed what data and when. This can be useful for identifying security breaches and investigating suspicious activity.

5.6. Observer Pattern

Observer design pattern is a behavioral design pattern that defines a one-to-many relationship between objects, so that when one object changes its state, all its dependents are notified and updated automatically. This pattern is also known as the publish-subscribe pattern.

The key components of the Observer pattern are:

**Subject**: This is the object that is being observed. It maintains a list of its dependents (observers) and notifies them when its state changes.

**Observer**: This is the object that is interested in the state of the subject and wants to be notified when the subject's state changes.

**Concrete Subject**: This is a subclass of the Subject class that implements the specific behavior of the subject.

**Concrete Observer**: This is a subclass of the Observer class that implements the specific behavior of the observer.

The Observer pattern, as presented in *Figure 2*, is useful in situations where you have one or more objects that need to be notified when another object's state changes. It promotes loose coupling between objects, since the subject doesn't need to know anything about the observers, and the observers don't need to know anything about each other.

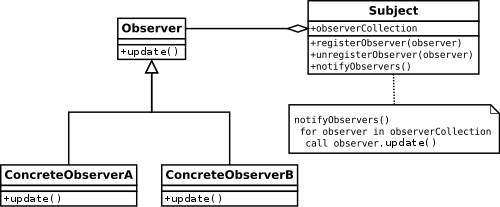


Figure 2

In this application, the observer patters in implemented for selecting the language of the GUI. The LoginController is the observable, while the LanguageController is the observer. When the language is changed, the observable notifies the observer to update the interface.

5.7 Singleton Pattern

Singleton is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. It is often used when there should be only one instance of a class throughout the entire application, and that instance needs to be easily accessible.

Here are the key characteristics and considerations of the Singleton pattern:

Single Instance: The Singleton pattern guarantees that only one instance of the class is created and exists during the lifetime of the application.

Global Access: The Singleton instance is typically globally accessible, meaning that any part of the application can access it without needing to pass references or create new instances.

Private Constructor: To prevent direct instantiation of multiple instances, the Singleton class typically has a private constructor, which restricts external code from creating new instances.

Static Instance Method: The Singleton class provides a static method that allows other parts of the code to access the single instance. This method is commonly named getInstance() or similar.

Lazy Initialization (optional): The Singleton instance can be lazily initialized, meaning that it is created only when it is first requested. This approach can help optimize resource usage and improve performance.

Thread Safety (if required): If the Singleton needs to be accessed concurrently by multiple threads, thread safety measures should be taken to ensure that the instance is created and accessed correctly in a multi-threaded environment.

Potential Downsides: The Singleton pattern, when overused, can lead to tight coupling and make testing more difficult. It should be used judiciously and only when there is a genuine need for a single instance throughout the application.

5.8 Mediator Pattern

Mediator is a behavioral design pattern that promotes loose coupling between components by encapsulating their interactions within a mediator object. It enables communication between components without them needing to have direct references to each other.

Here are the key components and concepts of the Mediator pattern:

Mediator: The Mediator is an interface or an abstract class that defines the communication protocol and contracts between the components. It typically declares methods for communication and coordination between the components.

Concrete Mediator: The Concrete Mediator implements the Mediator interface and provides the actual implementation of the communication protocol. It coordinates the interactions between the components and maintains references to them.

Colleague Components: Colleague components are the individual components that need to communicate with each other. They are aware of the Mediator and use it to send and receive messages to/from other components.

Loose Coupling: The Mediator pattern promotes loose coupling between components by abstracting the communication logic into a central Mediator. Components don't need to know about each other; they only communicate through the Mediator.

Simplified Communication: Instead of components directly communicating with each other, they send messages to the Mediator, which handles the routing and coordination. This simplifies the communication process and reduces dependencies between components.

Centralized Control: The Mediator pattern centralizes control and coordination logic in a single object. It can enforce certain rules or policies for communication, and it can also perform additional tasks such as logging, error handling, or transformation of messages.

The Mediator pattern is useful in scenarios where there are many interacting components, and managing direct communication between them becomes complex and error-prone. It promotes better maintainability, extensibility, and testability by decoupling components and reducing dependencies.

It's important to note that the Mediator pattern should be used judiciously. While it can simplify communication, it can also introduce additional complexity if not designed properly. It's important to consider the trade-offs and carefully design the communication protocol and responsibilities of the Mediator and the components involved.

5.9 Specification Pattern

The Specification pattern is a design pattern that allows you to define complex business rules or conditions as separate objects and combine them using logical operators to create more complex rules. It provides a way to encapsulate criteria and query objects in a reusable and composable manner.

Here are the key components and concepts of the Specification pattern:

Specification: The Specification is an interface or an abstract class that defines the contract for a specific business rule or condition. It typically declares a method like isSatisfiedBy or isSatisfied that takes an object and evaluates whether the rule is satisfied or not.

Concrete Specifications: Concrete specifications are the implementation classes that define specific business rules or conditions. They implement the Specification interface and provide the logic to evaluate the rule.

Composable Specifications: Specifications can be combined using logical operators such as AND, OR, and NOT to create more complex rules. Composable specifications allow you to build up complex criteria by combining simpler specifications.

Client: The client is responsible for creating and using specifications to evaluate objects against the defined rules. It can create new specifications, combine them, and pass them to the appropriate components for evaluation.

The Specification pattern helps in decoupling the business rules from the client code and promotes reusability and maintainability. It allows you to define rules as separate objects, making it easier to add, modify, or remove rules without affecting the client code.

Here's an example to illustrate the Specification pattern:

Let's say you have a list of products and you want to filter them based on certain criteria, such as price and category. You can create specifications like PriceSpecification and CategorySpecification that evaluate the price and category of a product, respectively. Then, you can combine these specifications using logical operators to create more complex rules. For example, you can combine them using an AND operator to find products that satisfy both the price and category criteria.

The client code can then use these specifications to filter the list of products. By utilizing the Specification pattern, you can easily change or extend the filtering criteria without modifying the client code, and you can reuse the specifications in different contexts.

Overall, the Specification pattern provides a flexible and reusable way to define and combine business rules or conditions, allowing for dynamic and configurable behavior in your application.

5.10 Bridge Pattern

The Bridge pattern is a design pattern that decouples an abstraction from its implementation so that they can vary independently. It allows the abstraction and the implementation to be modified and extended independently without affecting each other.

Here are the key components and concepts of the Bridge pattern:

Abstraction: The Abstraction is an interface or an abstract class that defines the high-level interface for the clients. It typically contains higher-level operations or behaviors that rely on the implementation.

Refined Abstraction: Refined Abstraction is a subclass of the Abstraction that further extends or customizes the high-level interface provided by the Abstraction.

Implementation: The Implementation is an interface or an abstract class that defines the low-level interface for implementing the Abstraction. It provides the primitive operations that the Abstraction relies on.

Concrete Implementation: Concrete Implementation is a subclass of the Implementation that provides the actual implementation of the low-level operations.

The Bridge pattern allows the Abstraction and the Implementation to vary independently. This means you can have multiple implementations of the same Abstraction or introduce new Abstractions without affecting each other. The bridge between the Abstraction and the Implementation is established through composition, where the Abstraction contains a reference to the Implementation.

The Bridge pattern is useful in scenarios where you want to decouple an abstraction from its implementation. It is particularly beneficial when you have multiple hierarchies that need to be combined in a flexible way. It promotes flexibility, extensibility, and maintainability by separating the interface from the implementation.

Here's an example to illustrate the Bridge pattern:

Let's say you have a shape hierarchy with different types of shapes like Circle, Square, and Triangle. Additionally, you have a rendering hierarchy with different rendering types like SVG, OpenGL, and Canvas. Instead of creating separate subclasses for each combination of shape and rendering type, you can apply the Bridge pattern.

You can define an Abstraction interface for the shapes, which declares high-level methods like draw or resize. Then, you can define an Implementation interface for the rendering types, which declares low-level methods like renderCircle or renderSquare. Each shape subclass can have a reference to an Implementation object.

This way, you can easily create new shapes or rendering types independently and combine them at runtime. The Abstraction and Implementation can evolve and extend without affecting each other. For example, you can add new shapes or new rendering types without modifying existing classes.

Overall, the Bridge pattern provides a way to separate the high-level abstraction from its implementation details, enabling flexibility, extensibility, and easier maintenance of the codebase.

1. Conclusions

In conclusion, developing this perfume shop chain application using Java and JavaFx has been a great learning experience for me. I have gained a better understanding of the MVC pattern, observer design pattern, and JUnit testing. Additionally, I have improved my skills in database management and login-based authentication and authorization.

During the development process, I faced some challenges, such as implementing the persistence pattern and integrating the Postgres database with the application. However, I was able to overcome these challenges through research and experimentation.

While the current version of the application meets the specified requirements, there is room for improvement. For example, adding more search criteria for perfumes, implementing a shopping cart feature, and integrating a payment gateway for online purchases could enhance the user experience.

Looking to the future, I would like to explore the possibility of making the application cloud-based to enable easier access and scalability. Additionally, I plan to continue learning about new technologies and frameworks to enhance my development skills further.

Overall, this project has been an excellent opportunity for me to apply my Java programming knowledge to a real-world scenario, and I am proud of the outcome.