VILNIUS UNIVERSITY
MATHEMATICS AND INFORMATICS FACULTY
SOFTWARE ENGINEERING

Locals to Locals

Fourth Laboratory Work

Software Engineering II

Submitted by:
Arnas Rimkus,
Deividas Kučinskas,
Matas Lazdauskas,
Jaroslav Kochanovskis,
Gytis Bečalis

Supervisor: Assist. Dr. Vytautas Valaitis

Vilnius 2021

Summary

As we continue to learn software engineering principles and apply them in development of an actual application, in this paper we will define the architecture of the system, use adequate viewpoints and perspectives that will help to analyze a change request in the system and implement automated tests. The tasks for this part are as follows:

1. Define the architecture of the system and the change considering architectural styles and patterns, non-functional requirements.
2. Adequate viewpoints and perspectives should be selected, explaining the decisions and the architecture should be described according to them.
3. Implement at least some of the planned changes (agreed with the lecturer).
4. Implement an automated integration or system test, integrate it with the CI/CD processes.

Table of Contents

Contents

1. Requirements

To clarify the exact changes to be made, we have created a formalized list of requirements. To note, requirements are usually classified as functional and non-functional, however in accordance with recent research we have chosen the name quality requirements instead of non-functional as to not understate the importance of these types of requirements.

1.1 Functional Requirements

FR 1. Create a swipe card page for buyers to find vendors' services

- FR 1.1. Page must contain a swipe card with:
    - FR 1.1.1. Name of the service
    - FR 1.1.2. Image of the service
    - FR 1.1.3. Type of the service
    - FR 1.1.4. Name of the vendor
    - FR 1.1.5. Distance to the service
    - FR 1.1.6. Service average review score
    - FR 1.1.7. Short description
    - FR 1.1.8. Button to select service, which redirects to vendors service details page
    - FR 1.1.9. Button to skip service
    - FR 1.1.10. Button to report service
    - FR 1.1.11. The card border color is to be determined by the card's score.
- FR 1.2. Swipe cards to be arranged in order of distance from the buyer
- FR 1.3. If all services are looked at, start again from the beginning of the list

FR 2. Track statistics:

- FR 2.1. The system should track the number of times the service was viewed, selected, and skipped.
- FR 2.2. Display statistics to the vendor.
- FR 2.3. Add visual trophies to the vendor's service to display popularity.

1.2 Quality Requirements

QR 1. Usability requirements

- QR 1.1. All updates must be in English and Lithuanian
- QR 1.2. All updates to be designed with mobile UI in mind first

QR 2. Capacity requirements

QR 2.1. New features should not impact the general performance of the website (average load times to remain in the same 10% timeframe)

QR 3. Availability requirements

QR 3.1. New features must work just as well as the whole service

QR 4. Interoperability requirements

QR 4.1. Usage of PostgreSQL

QR 5. Reliability requirements

QR 5.1. New features should be relatively bug free (no unhandled crashes)

QR 6. Regulatory requirements

QR 6.1. Comply with GDPR requirements

2. System Architecture

Due to the nature of the system, being born as a student assignment project, originally it was created without a specific software architecture in mind. However, after analyzing the system, we have found the system to follow some of the well-known architectural styles and architectural patterns. As our project develops, we have created this architecture document to better understand the system and to develop it more efficiently and coherently.

2.1 Architectural Styles

While there are sources that state that architectural styles and architectural patterns are synonyms, in our work we differentiate the two by considering architectural styles as being concerned with the organization of the code, while architectural patterns as a way to solve recurring architectural problems.

2.1.1 Client-Server

The structural style of our system is in the three-tier client-server architecture - a very common architectural style for web applications. The system has three tiers - the presentation tier, the logic tier, and the data tier. These tiers can be directly mapped to usual fields of software development - front end, back end, and database management. While our system is currently only hosted locally, this style choice allows us to deploy our system to be hosted online with minimal adjustments. Also, we believe it separates development concerns neatly for our web application-based system. For our team, this architectural style fits well, as we are used to working in this arrangement.

2.1.2 Database-Centric

The shared memory aspect of our architecture was mostly created in accordance to a database-centric style. Again, this is a very common architectural choice, with many web applications having a central relational database to store all the system data. Our system is currently set up to store data in a locally hosted PostgreSQL database. We think this choice fits our system well, because our system is not complex or large enough to warrant a more complex data sharing architecture, like a rule-based system or blackboard system.

2.1.3 Representational State Transfer

Our system at least partially implements the REST style. Our web application follows several principles of REST: it has a client-server architecture as stated previously, is stateless as no session state is stored in the server and cacheable, although currently caching is not implemented. The system is also using uniform interfaces and a layered system, all of which was achieved with the use of ASP.NET framework, which automatically allows the setup of a RESTful application. The only principle that is not being followed is code on demand, as our system has no such functionality. We believe that the REST style fits our system needs, especially the benefits of increased simplicity and reliability.

2.2 Architectural Patterns

To better understand the system's architecture, we analyzed what kind of patterns were applied in the development. It is important to understand whether they provide the system with the desired functionality and quality attributes, afterwards we can use the information about these patterns when designing the implementation of the newly requested change. To find the architectural patterns used in our system, we have examined what problems did the original developers face while developing the system and analyzed the code.

2.2.1 MVC

The architectural pattern that most obviously fit our system was the MVC (model-view-controller) pattern. This architectural pattern was used for developing user interfaces in the system. We believe that this architectural pattern was a good choice for the system, as it is a popular solution for designing web applications, allowing for easier development of the system, especially for learning developers. Also, by dividing the related application logic into three interconnected elements, it allowed to separate internal representations of information from the ways information is presented to and accepted from the user. This allows us to implement our new Swipe Card system more easily, as with little change to the information already held in the system, we can create whole new user interfaces.

2.2.2 CRUD

To support frontend logic and provide data CRUD pattern is applied. This architecture pattern says that you can cover all the functionality for an entity by providing these operations (Create, read, update, delete). It makes the system design easy. This pattern suits the application since for most entities all that is required is either receiving POST requests from the user and storing data in the database or loading pages with data of entities from the database. The only problem is that it limits business logic to only storing and providing data to the user.

2.3 Architecture Perspectives

To analyze the system furthermore, we tried to measure it from different perspectives, identify problems and threats to provide feasible solutions.

2.3.1 Security

To ensure security of sensitive data, ability to reliably control and recover system from security breaches we analyzed security perspective.

2.3.2 Sensitive User Data

As for every service that collects and uses user data it is important to protect it. The most sensitive user data is e-mail addresses and password. Since this data is stored in the Database, one of the simplest and most dangerous threats are SQL injections. Usage of Entity Core Framework in all the code related to CRUD operations serves as protection against this threat. User account passwords are encrypted upon registration/password change and only then stored in the database. This makes data unusable during unexpected leaks.

2.3.3 Unauthorized Access

Another important aspect is that users should not access the data they are not supposed to access. (e.g., Administrator reports panel.) To stop this from happening, different user roles have been implemented. This has been done by using a role management system that comes with the ASP.NET Core framework and is more reliable than unknown third-party frameworks.

Currently, the logging in system does not limit the number of attempted logins. This leaves the breach, allowing us to use “brute force” methodology in attempt to access other users accounts. To solve this threat a simple mechanism such as requiring email confirmation after a certain number of unsuccessful logins could be implemented.

2.3.4 Malicious Attacks

The system currently has some flaws: there is currently no protection against distributed denial-of-service (DDoS) attacks or other types of malicious intent that influence the availability of our web service. For example, denial-of-service (DoS) attacks could be achieved simply by one person repeatedly trying to login to the system with incorrect data, as we have no type of protection implemented against such actions. This could increase unplanned downtime, which is already an issue.

2.3.5 Downtime

Unplanned downtime might already be a problem due to handling unexpected severe bug reports. Yet another cause of this downtime is the fact that whenever planned updates or bug fixes will inevitably be rolled out, the service will be shut down for maintenance, as we have no systems for updating the system while it is running, simultaneously.

2.3.6 Performance and Scalability

To ensure best user experience as well as ability to handle increased processing volumes we analyzed this perspective to get better insights.

2.3.7 Performance Issues

The performance of this web application is quite problematic. For example, threading (a possible performance optimization) is only used once, and on a performance-wise insignificant part. This limits the possible response time potential vastly, as for every action and new page you must wait for all the required context to load as well, which results in quite a bit of waiting for a user. The performance issues would only be exasperated when a large user base is formed, as server stress would add to the already existing problem. However, this lack of performance formed via current code structure has at least one benefit: it makes the system undeniably more predictable and maintainable for developers.

2.3.8 Scalability Limits

A lot of the system mechanisms read the whole collection of data from the database and filters it locally. As this does not pose a threat when a small amount of data is collected/stored, it strongly limits the scalability of the system and hurts overall user experience because of long loading times. This could be solved by changing parts of the code that get data from Database to filter data with SQL queries rather than pulling all related data.

2.3.9 Internationalization

As our system solves a problem that is relevant not only for a specific location, there is potential to adopt it globally. The only problem with the current situation is that the system only supports two languages: Lithuanian and English. To make it more accessible to global communities the system would have to provide support for at least ten most spoken languages around the world. As the beginning of the language support mechanism is already implemented, this eases the solution of this problem because only additional translation of text is required. Another possible improvement would be the addition of different service categories that align with specific locations/cultures.

2.3.10 Requested Change and Perspectives

The requested change does not influence security and internationalization perspectives in any way or uses them as they are already implemented. The only field where attention should be paid is performance and scalability. As the cards can be generated for all existing services it should be done incrementally and not all at once and should follow guidelines that are pointed out in the “Performance and Scalability perspective section.

3. Viewpoints

There are many different architecture viewpoints to consider while analyzing a software system, however not all of them are relevant to each system. We chose to analyze the viewpoints that we believe are the most appropriate and relevant to our project right now.

3.1 Context Viewpoint

Context viewpoint describes the relationships, dependencies and interactions between the system and its environment. It will help to present what and where our changes have been implemented in the system.

3.1.1 System Scope and Responsibilities

Our system helps to connect local businesses with a larger clientele base. Main benefit for users is that customers can find local goods that are mostly ecofriendly and by purchasing them, support local vendors. The system helps to establish contact between customers and vendors, so that both sides benefit from it. The service is free to use. It is important to note that our system is to connect two groups of people and provide information but does not provide selling/buying services. Requested change will provide enhanced experience for customers to ease the process of finding nearest vendor services.

3.1.2 Identity of External Entities, Services and Data Used

System uses an external service, the OpenStreetMap API that allows a map with service points to be displayed. OpenStreetMap API also helps to select addresses in creating service for vendors, so vendors do not have to type it manually. This API also provides coordinates of the vendor's entered address. Location data from this API allows us to determine service distance from the user's real time location so we can provide nearest services and sort them by distance.

3.1.3 UML Component Diagram

This section will illustrate the implementation and components of the system. Each Swipe card is constructed by a controller using data of service from the database. Since statistical data as well as reports of inappropriate content can be collected from the swipe cards, this data is stored in the database. This is illustrated in the Figure 1.

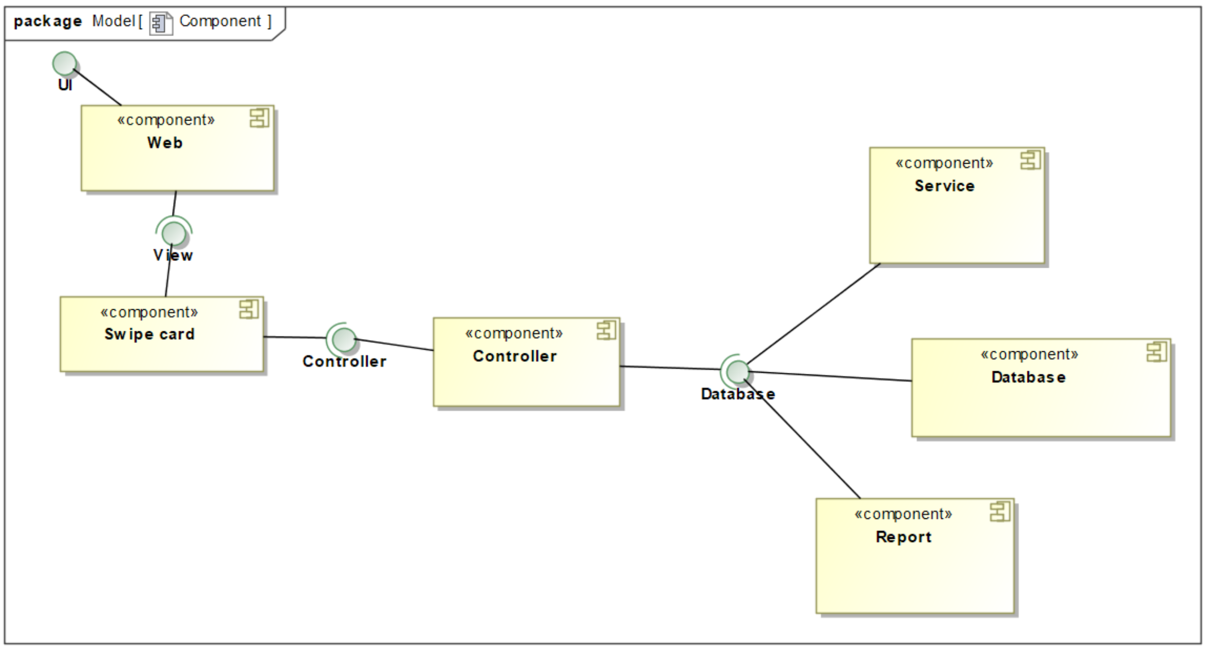

Figure 1 Component Diagram

3.1.4 UML Use Case Diagram

The requested change expands the ability of finding services for the users. As illustrated in Figure 2 newly added Swipe cards allow users to either select or skip (directly or by reporting) services.

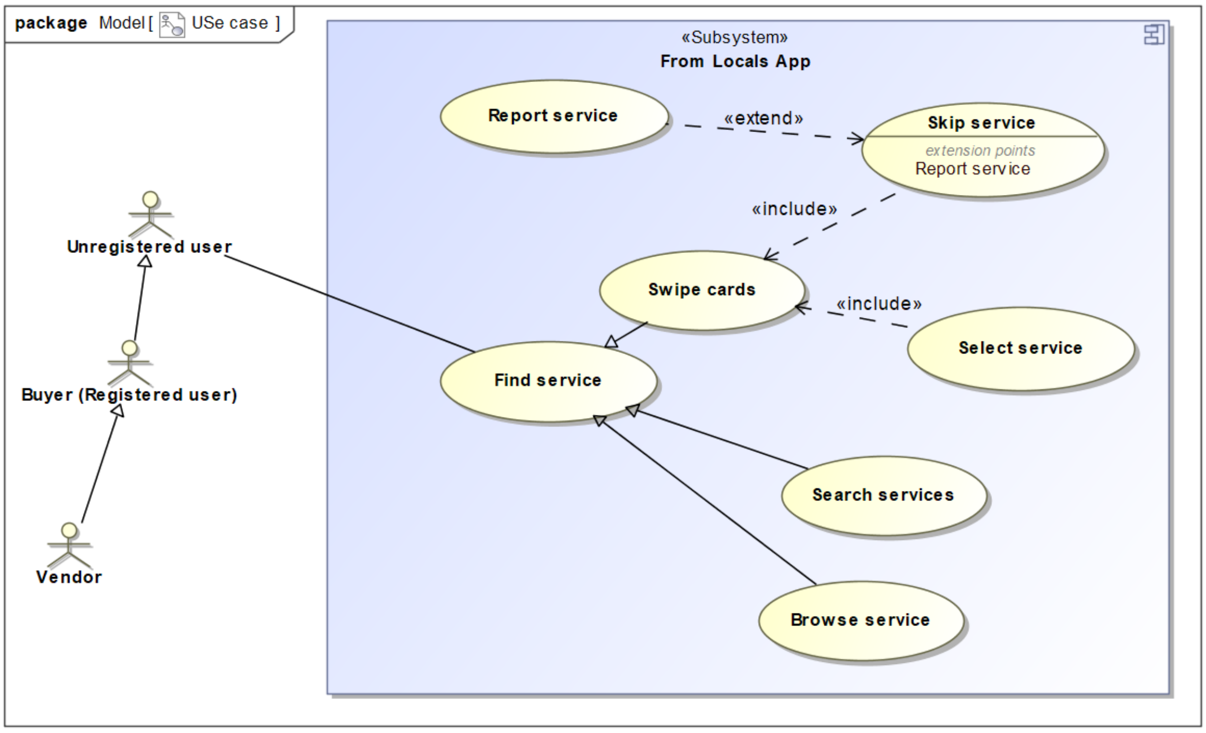

Figure 2 Use Case Diagram

3.1.5 Data Flows

Data flow displays how information between components flows. Swipe card get information from the controller that receives data from the database. User via the web component can see swipe card and decide what to do next. Web component sends commands to controller component what the user wants to do next.

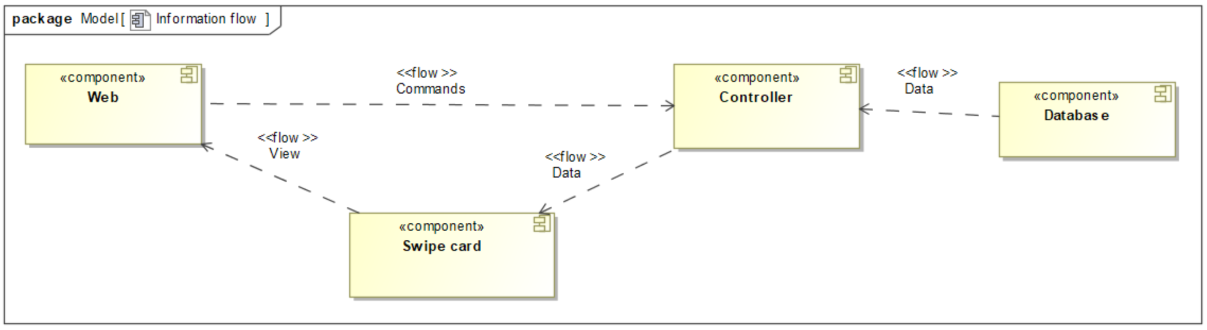

Figure 3 Data Flow Diagram

3.1.6 Interaction Scenarios

On displayed Swipe card person can choose to skip, select, or report a given service. If a person is not interested in service, then he skips swipe card. If a person notices something inappropriate, then he reports it. If a person is interested in service, then he will be redirected to the service page.

3.1.7 Security Perspective

The system is only accessible for the end users through the Web UI. It allows the potential threat of DDoS and DoS attacks that are discussed in more detail in section 2.3.4. In the same principle brute force could be used to access accounts by using "brute-force". Most of these problems could be solved by choosing hosting services that provide protection against mentioned problems.

3.1.8 Performance and Scalability Perspective

Every component on the Web UI should be responsive and loading times for all new features should follow QR 2.1. Loading performance for the external interfaces are strongly dependent on the internal interfaces. By itself, every user interface should load without waiting for data processing to enhance user experience.

3.2 Functional Viewpoint

A functional viewpoint describes the system's functional elements, their responsibilities, primary interactions and demonstrates how the system will perform the functional requirements of stakeholder needs. We chose this viewpoint to clearly showcase that the stakeholders wanted the new addition to complement and work well with the current architecture.

3.2.1 Current functionality

The system is set up in such a way that the database is the cornerstone of our functionality. We collect user information they provide via the UI and store it in the database. The system's functionality of what it achieves lies in gathering and choosing what data to show to which users, and ways to manage it, for the former's example, the main pages are all essentially related to gathering data or displaying it, which is filtered via the code, for an example of the latter, removing false or malicious information via an admin page is a way to moderate. The result for users is a simple system of reliable information sharing in a convenient way. As for the logic which achieves this functionality, the system is easily identifiable as having an MVC architecture: models, views and controllers do the heavy-lifting and simultaneously have the most responsibilities all while heavily depending on one another.

3.2.2 UML Composite Structure Diagram

This diagram demonstrates how the "Swipe card" functional requirement will fit into the structure. As can be perceived, it will be contained in the "Vendor" class because the relationship they have is one of composition.

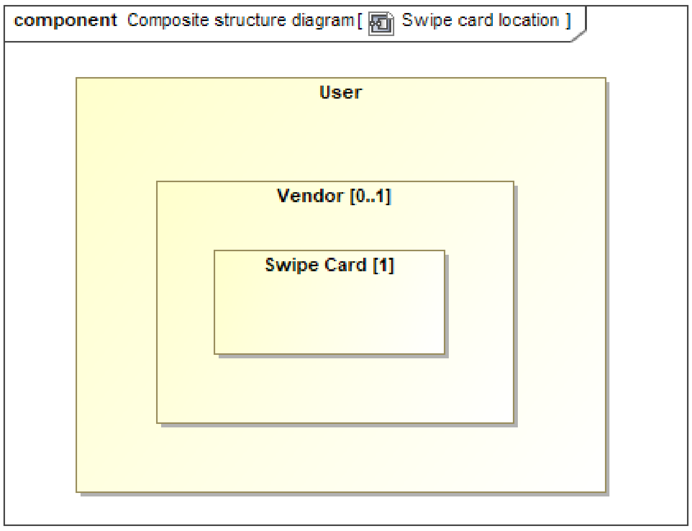
component Composite structure diagram [Swipe card location]
User
Vendor [0..1]
Swipe Card [1]

Figure 4 Composite Structure Diagram

3.2.3 Required Change Implementation

As can be seen in this class diagram, and further expanding on the composite structure diagram from before, the implementation of the swipe card (new functional requirements desired by stakeholders) can be easily achieved following an existing MVC architecture pattern, adding to an already existing class, and therefore, without changing the architecture, only adding some additional functionality.

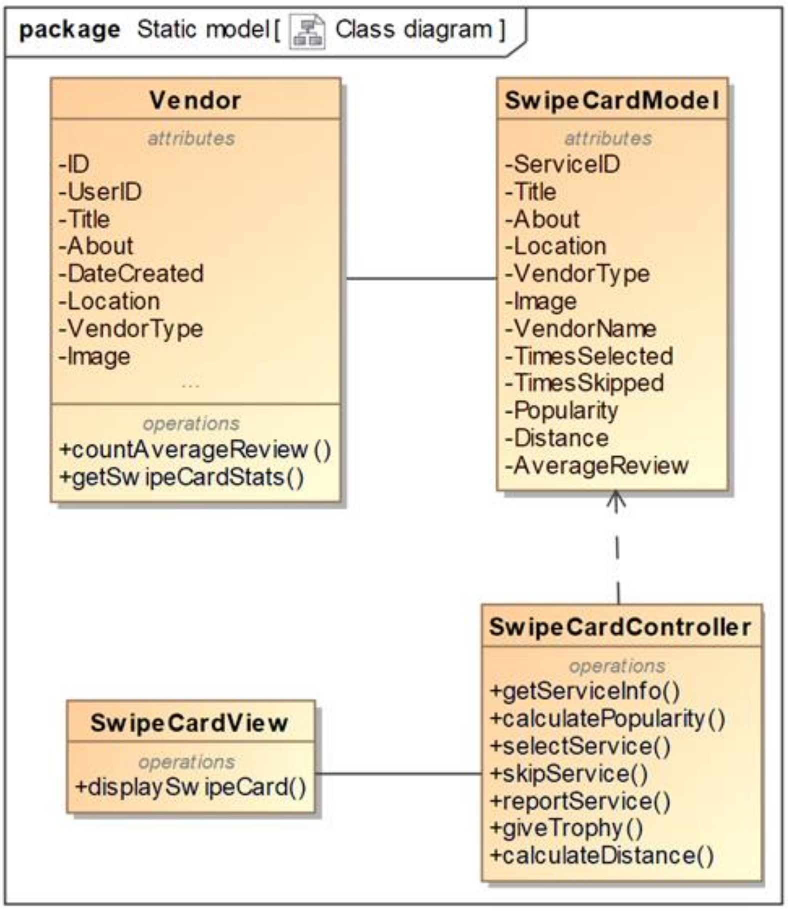

Figure 5 Class diagram

3.2.4 UML Component Diagram

It is visible in Figure 1 seen before, that the system completes its objectives in a database-centric architecture pattern. The functionality is based on simply working with the database, storing, and displaying user information via views and controllers, and so the architecture remains unchanged after implementing swipe cards.

3.2.5 Security Perspective

The most sensitive component is the database since all sensitive data is stored here. Safeness of database is discussed in section 2.3.2. Another important component is a Swipe card because false data could be generated, that affects vendors, if a swipe card of the same service would be accessible more than once in card deck rotation (e.g., skipping same service multiple times). Other components related to the change are only to process and display data received from the database to the user.

3.2.6 Performance and Scalability Perspective

To increase performance and loading time all database tables should be in Fourth normal form and Boyce-Codd Normal Form. As this will allow to decrease duplicate data and avoid anomalies, data will be processed faster.

3.3 Development Viewpoint

Development viewpoint describes the architecture that supports the software development process. This viewpoint helps us to organize and standardize the whole project.

3.3.1 UML Package Diagram

As already mentioned, the MVC (model-view-controller) pattern fits our system well. We can clearly see that in the package diagram on Figure 6. It is also clear that to realize required changes to the system we can continue the development process using the MVC pattern. As we see in the diagram swipe cards are modeled the same way as other models in our system.

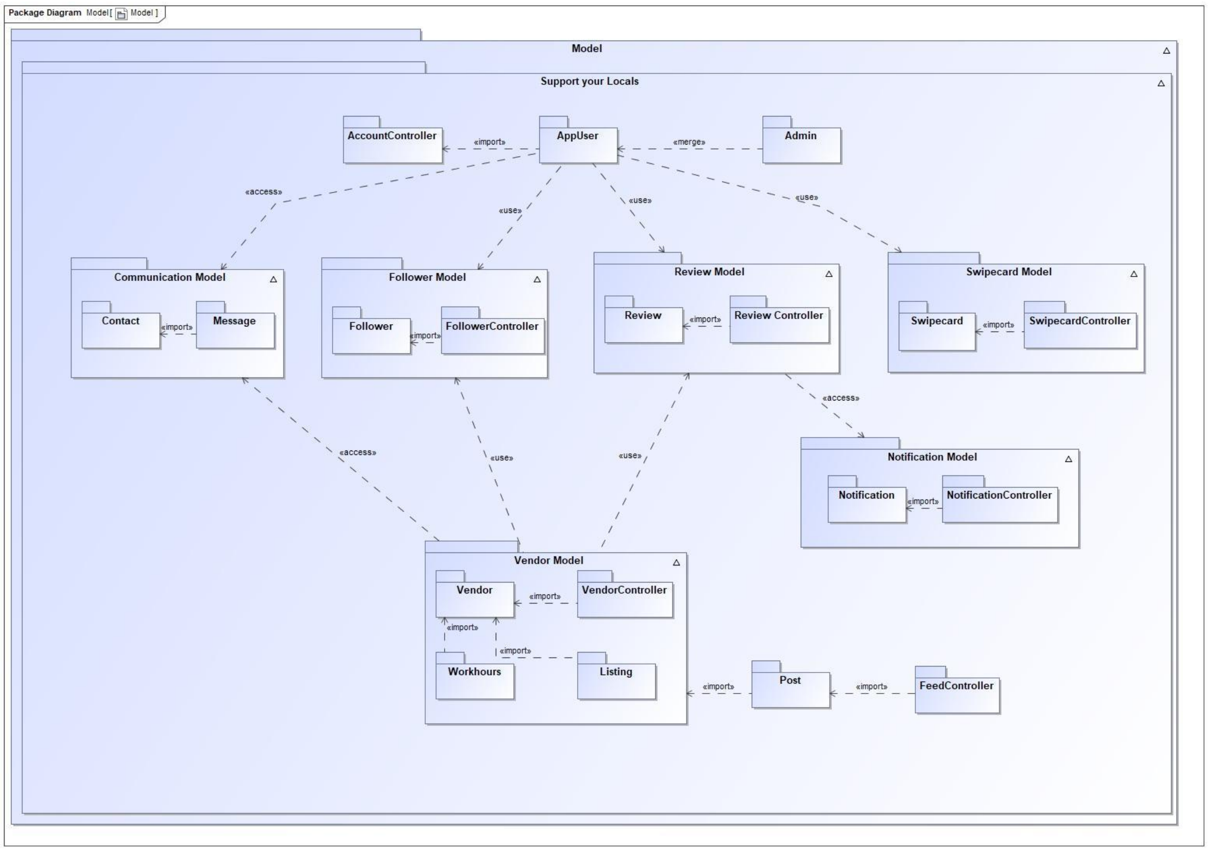

Figure 6 Package Diagram

3.3.2 UML Activity Diagram

Our realization of the required change fits the system client - server architectural style as well. During customer service selection activity, the whole process happens when the customer (client) communicates with the system (server) and it is illustrated in Figure 7. Service swiping blends into our system well, as we see in the diagram, with realized change, customers can choose desired service not only via search and browsing, but also via swipe cards.

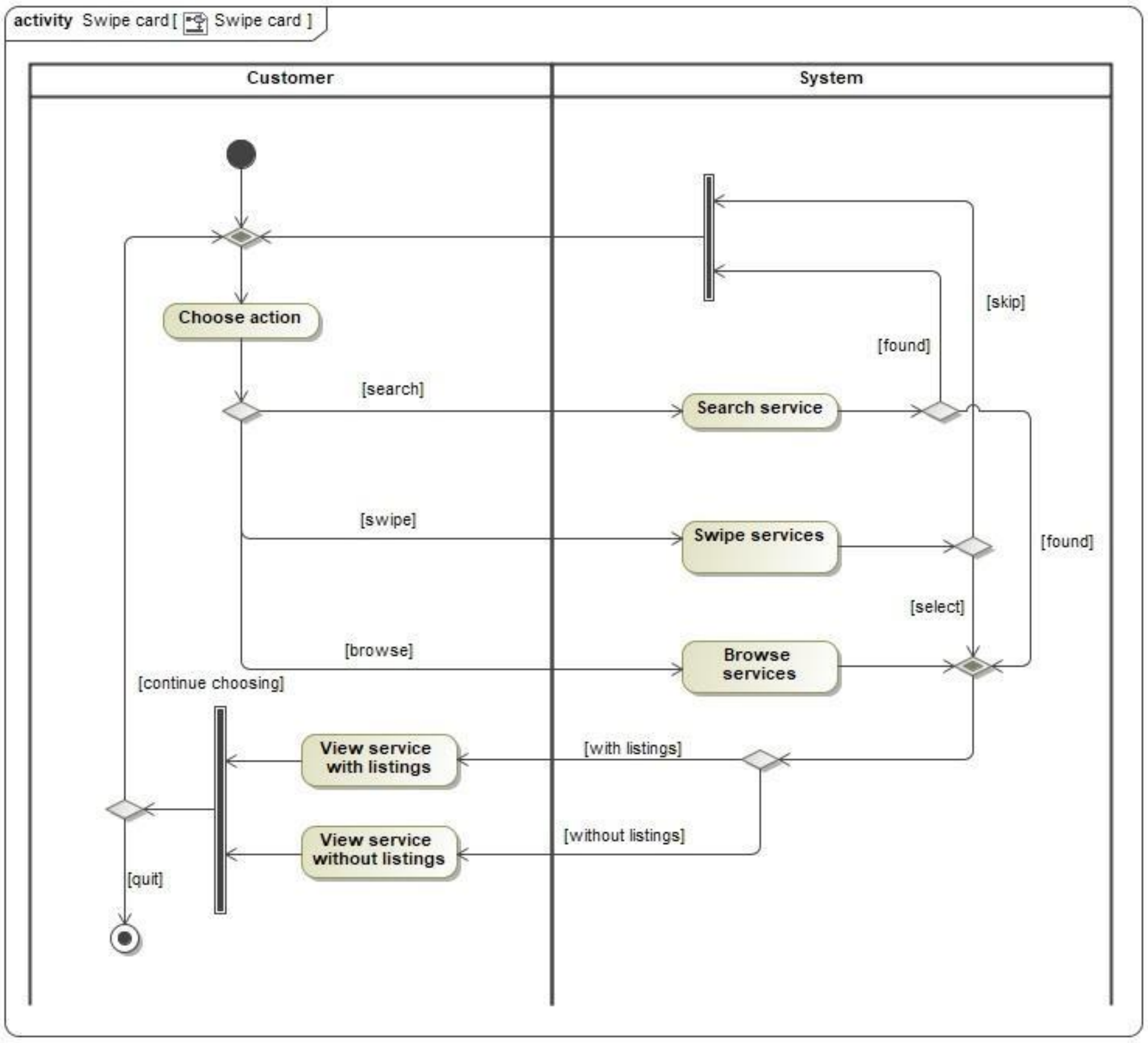

Figure 7 Activity diagram

3.3.3 Code Organization

Version control is ensured using the distributed version control system Git. We use GitHub cloud-based hosting service to manage Git repositories. While developing required system changes continuous integration philosophy has been used. Configuration management was also ensured using the Git system.

3.3.4 Common Processing

The system tries to handle all possible exceptions. System exceptions that occur are logged as well. Logging helps to trace back problems with the system and provides better understanding to developers what does not work right. All logs are with UTC timestamps to keep a unified time system. To ensure better user experience and protect information about the system, when an unhandled exception happens, exception text is logged and not provided to the user. User instead is redirected to the error page.

3.3.5 Performance and Scalability Perspective

To increase the performance related to newly requested features, a sorting algorithm should be used that works well with a large amount of randomly sorted data, where we need to sort services by distance from the user. Another important thing is to note that it is necessary to avoid creating any “God class” by creating new or modifying an existing class during feature implementation. It is required to keep the single-responsibility principle and to not complicate the testing process.

4. Testing

To ensure that our system works as expected and to improve its general quality, we have implemented automated testing by adding unit and integration tests. Automated tests allow our team to immediately notice at least some of errors that we did not catch during development, while also saving us time we would otherwise have to spend on manual testing.

4.1 Automated Testing Tools

To implement unit and integration testing, we used several external packages in our project, as we wanted a more convenient and developed solution than the vanilla .NET provides. To add the testing project and to run unit tests, we use the xUnit NuGet package, for mocking of objects for integration testing, we use the MoQ library and for more convenient and readable tests we use the Fluent Assertions package.

4.2 Automated Test Builds

From our previous work, we had already set up an automated software workflow, using GitHub Actions to run code quality and build tests on the systems source code. We have set up our pipeline to automatically run these tests on every ‘push’ and pull request, however we only run the code style tests on the parts of the code that changed to decrease testing times. After we wrote our tests using the tools mentioned previously, we only needed to adjust minor details in our pipeline to automatically run our tests on every build.

4.3 Implemented Tests

Our priority when developing the tests was to examine which parts of the system are the most crucial to test in the immediate timeframe, because we did not have the resources to fully implement testing in the whole system. We determined the changes we implemented during this work should have the priority, afterwards the testing efforts should go to our previously implemented features, and finally to the original system features from before our involvement in the project. Currently we have implemented some unit and integration tests for the swipe-card system, for example to ensure that the distance and review score calculations are correct.

5. Traceability Matrices

We created a viewpoint to the functional requirements traceability matrix, shown in Table 1, to ensure that all requirements are covered in our requirements elaboration. We skipped requirements FR 1.1.1 – FR 1.1.11 since they strongly relate to FR1.1 and no model/diagram addresses them separately. We also covered some architecture perspectives related to certain viewpoints separately from overall coverage of the system. This coverage can be seen in Table 2.

Table 1. Viewpoint to Functional Requirements Traceability Matrix

		FR	FR 1.1.	FR 1.2.	FR 1.3.	FR 2.1	FR 2.2	FR 2.3
Viewpoint	Diagram							
Context	Component		X					
	Use Case		X	X	X			
	Data flow		X			X		
Functional	Composite Structure		X					
	Class		X	X		X	X	X
	Component		X					
Developmen t	Package		X					
	Activity		X					

Table 2. Viewpoint to Architecture Perspective Traceability Matrix

	Viewpoint	Context	Functional	Development
Perspective				
Security		X	X	
Performance and Scalability		X	X	X

6. Results

In this laboratory work, we have achieved the following results:

1. Performed system architecture's analysis, identifying architecture styles and patterns their benefits and current situation as well as discussed different architecture perspectives.
2. Chosen adequate viewpoints and described architecture according to them.
3. Analyzed how requested change fits in the established system architecture.
4. Planned and described the test-case and its scenario.
5. Implemented a part of the requested change and started implementing tests to cover the system.

7. Conclusions

From the results of this laboratory work, we can draw the following conclusions:

1. The current system already follows the architecture styles and patterns that are suitable for its development.
2. The proposed change fits the existing architecture well.