Manuel Hanifl, Brian Schneider

[E-Mail-Adresse]

Tour Planner

FH Technikum Wien • SS2023

Exposee

Form a team of two students to develop an application based on the GUI frameworks C# / WPF or  
Java / JavaFX. The user creates (bike-, hike-, running- or vacation-) tours in advance and manages  
the logs and statistical data of accomplished tours

**Table of content**

[Protocol about the technical steps and decisions you made (designs, failures and selected solutions) 2](#_Toc139565975)

[Desing pattern 2](#_Toc139565976)

[Failures 3](#_Toc139565977)

[Document your application features using an UML use case diagram 3](#_Toc139565978)

[Document your UI-flow using wireframes 4](#_Toc139565979)

[Document the application architecture using UML: 5](#_Toc139565980)

[class diagram or sequence diagram for full-text search 5](#_Toc139565981)

[Explain why these unit tests are chosen and why the tested code is critical 6](#_Toc139565982)

[Tracked time spent with the project 7](#_Toc139565983)

[GitRepo-Link 8](#_Toc139565984)

# Protocol about the technical steps and decisions you made (designs, failures and selected solutions)

## Desing pattern

**Model-View-ViewModel**

The MVVM pattern is a software architectural design pattern that separates the user interface (View) from the business logic (Model) using a mediator called ViewModel. We embraced this pattern to achieve better code organization and maintainability. The ViewModel served as an intermediary between the View and Model, allowing for easier data binding and facilitating unit testing of the business logic.

**Singleton Pattern**

The Singleton pattern was utilized to ensure that a class has only one instance and provides a global point of access to it. This pattern was applied in scenarios where we needed to control the creation and access to a single object, such as a configuration manager or a database connection. By employing the Singleton pattern, we achieved efficient resource management and centralized control over critical system components.

**Factory Pattern**

The Factory pattern was employed to encapsulate the object creation process and provide a flexible way to create instances of related classes. By utilizing this pattern, we were able to abstract the creation logic from the client code and ensure that the appropriate object was created based on specific conditions or configurations. The Factory pattern greatly enhanced the extensibility and maintainability of our application, allowing us to add new object types with minimal changes to the existing codebase.

**Observable Pattern**

To facilitate communication between different components of our application, we implemented the Observable pattern using an Observable List. This pattern enables objects to notify their observers automatically when their state changes. By applying the Observable pattern, we established a loosely coupled system where observers could dynamically register or unregister themselves to receive updates. This pattern was particularly beneficial in scenarios where real-time data synchronization or event-driven behavior was required.

**Client Server Pattern via Rest API**

The Client-Server pattern, implemented through a REST API, was employed to separate the concerns of client-side and server-side functionalities. This pattern enabled us to establish a clear division of responsibilities, where the client application focused on user interactions and the server application handled data processing and storage. By utilizing a REST API, we ensured a standardized and scalable communication interface between the client and server components, promoting interoperability and ease of integration with other systems.

## Failures

**Data Management**

By far the most problems were caused by the decision to put all the data into the models. If I could redo the project that would be the biggest and most important change I would definitely make, as this decision made the code a lot more complicated and prone to bugs. Because this way no change to the data could be made directly but first needed to be passed to the Model which then acted as a State (or Store, however you want to call it) which in turn made the changes a lot more complicated.

**Fulltext search**

Inadequate indexing and search algorithms resulted in poor search performance and inaccurate results. We realized the significance of optimizing indexing techniques and choosing suitable search algorithms to improve search functionality and enhance the user experience.

**Tourlog creation**

Incomplete or inconsistent data entries in tour logs were observed due to a lack of proper handling of interruptions or errors during the logging process. To overcome this, we implemented transactional mechanisms and improved error handling routines to ensure data consistency and completeness.

# Document your application features using an UML use case diagram

TourPlanner



Customer

# Document your UI-flow using wireframes

Ein Bild, das Screenshot, Reihe, Multimedia-Software, Software enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Screenshot, Software, Zahl enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Software, Multimedia-Software, Reihe enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

**TourMap**

**TourInfo**

**TourLog**

**EditTourLog**

**EditTourInfo**

# Document the application architecture using UML:

# class diagram or sequence diagram for full-text search

Ein Bild, das Text, Screenshot, Diagramm, Reihe enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Screenshot, Diagramm, Schrift enthält.

Automatisch generierte Beschreibung

# Unique feature

Our project features a unique addition: the implementation of Swagger. This open-source platform enhances our application by providing a browser-based interface for showcasing and documenting API endpoints. With Swagger, developers and clients can easily explore the available endpoints, view parameters, and understand return values. By opening the Swagger interface in the browser, users can customize parameters and execute test calls, promoting efficient utilization of our API.

# Explain why these unit tests are chosen and why the tested code is critical

1. UnitTest
2. UnitTest
3. UnitTest
4. UnitTest
5. UnitTest
6. UnitTest
7. UnitTest
8. UnitTest
9. UnitTest
10. UnitTest
11. UnitTest
12. UnitTest
13. UnitTest
14. UnitTest
15. UnitTest
16. UnitTest
17. UnitTest
18. UnitTest
19. UnitTest
20. UnitTest

# Tracked time spent with the project

|  |  |  |
| --- | --- | --- |
| Id | Commit | date |
| 5a2f4a7b8ea94f7fba3d84de5e1e30936f0c076f | Initial commit | Tue Jun 13 14:01:50 2023 |
| df92dbf11ca1926fea296ce260b684b1a0c78a8d | update gitignore | Tue Jun 13 14:10:20 2023 |
| 7f01bd6a62ab9c54bf6ff860b3cc2d0436111563 | getting started with di | Thu Jun 29 03:10:11 |
| 85deb6ae8c60dc1acf0df01c1ec2b631175f3f75 | DI is done, Working on UI | Sat Jul 1 21:18:19 2023 |
| f9777c2aa76d231faab09172b0fdd4c9f42d3814 | add some nice icons | Sat Jul 1 21:54:15 2023 |
| 1b65a6911eb2e63556f83159b5efdf10dd7ce25a | fix tourLogCreation bugs | Sun Jul 2 20:57:58 |
| 2f07e9429615673901cc3004bc7a347746e230fc | further bugFixes and MapQuest is done | Tue Jul 4 18:41:57 |
| 11857e654f8c50c7cf97752228d5596d03efb0f5 | started with fullTextSearch | Wed Jul 5 15:49:31 2023 |
| b93fc3a27ef72388bd6edc9d944332d8c73a2804 | implement search and further bugfixes | Thu Jul 6 15:07:16 |
| 27bd1dfa92193bb85a172f254e02df21d07ce374 | finally delete the TourPlanner dir | Thu Jul 6 15:09:21 |
| 72527ca99371d8742d270d0f4437013967b9eed1 | add input validation | Thu Jul 6 15:37:54 2023 |

Spent time about 80 hours.

# GitRepo-Link

https://github.com/Dovahkiin02/FH\_swen02\_TourPlanner