SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGY

GYM BRO

Gym companion

Martin Szabo

2023

Table of Contents

Introduction

GymBro is an innovative and user-friendly application designed to simplify the process of finding the perfect gym for your fitness routine. GymBro offers a range of features and functionalities that cater to the needs of gym enthusiasts looking for a gym that fits their requirements.

1 Overview

GymBro is an innovative and user-friendly desktop companion application that has been designed to help gym enthusiasts who are on the lookout for the best possible gym that suits their workout needs within their city. This application aims to make the gym-hunting process as smooth as possible by providing users with an interactive map that showcases all the gyms located in the area.

Clicking on a gym on the map, users can explore the facilities offered, check out the gym's website, and get an idea of what the gym’s offering. The app provides information on each gym, including its. contact details, and hours of operation. Users are also able to save their favorite gyms.

With GymBro, users can rest assured that they are making an informed choice when it comes to selecting a gym that meets their preference.

2 Features

GymBro offers variety of features such as:

1. Dynamic map of gyms – Map of gyms where user can hover over a gym and click on it. After clicking on it the user will be presented with information about the gym he clicked on.
2. Showing gym information – Once user chooses a gym. He'll be presented with information about the gym. (Name, Location, …)
3. Filtering out favorite gym's – User can add selected gym to his favorites list. Then from the main menu he can view a map where only his favorite gyms will be presented.
4. Saving favorite gym's – Additionally the user can save his favorites list locally. When he reopens GymBro his favorites list will remain the same.

Figure 1 Main menu

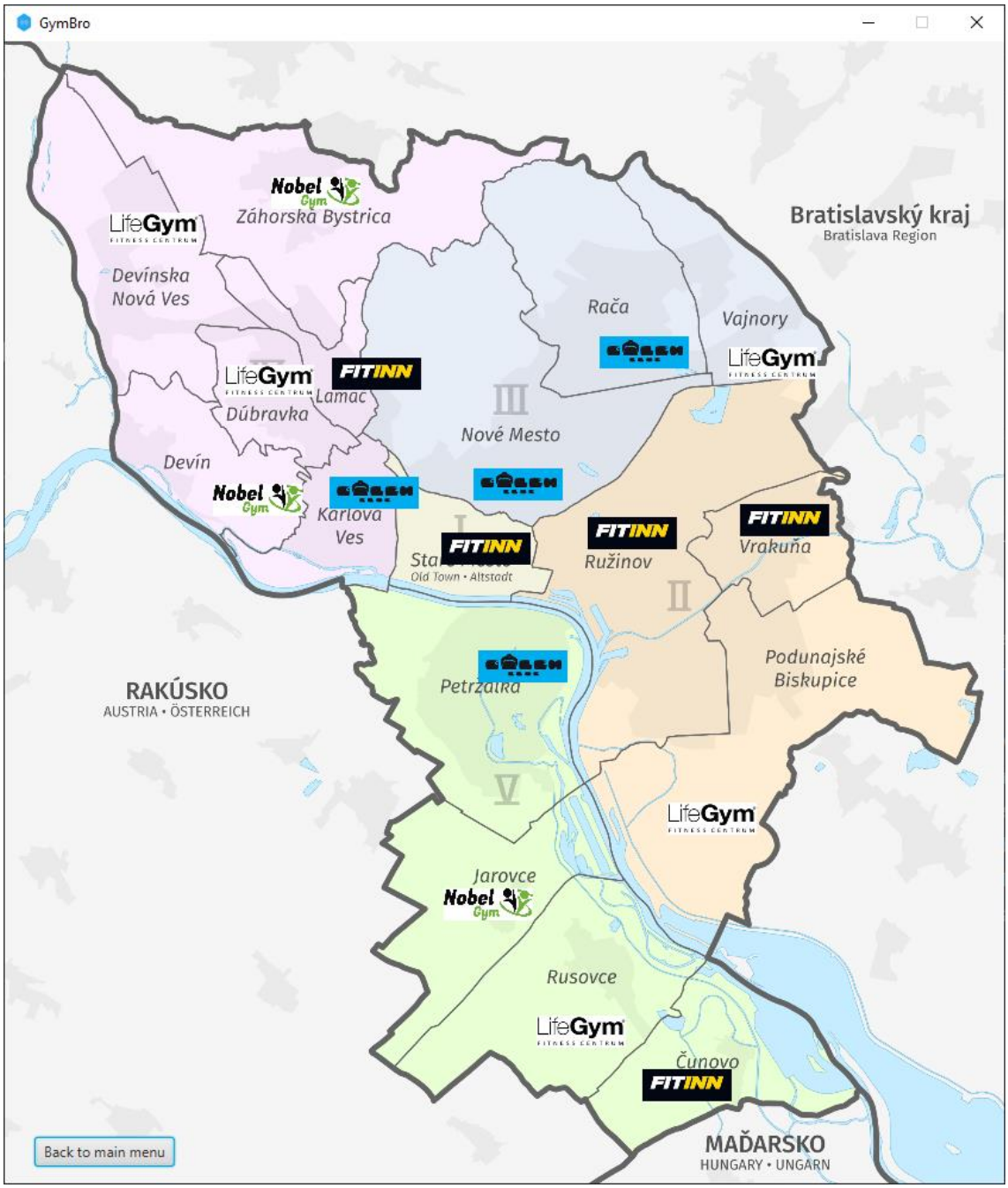


Figure 2 Map of gyms

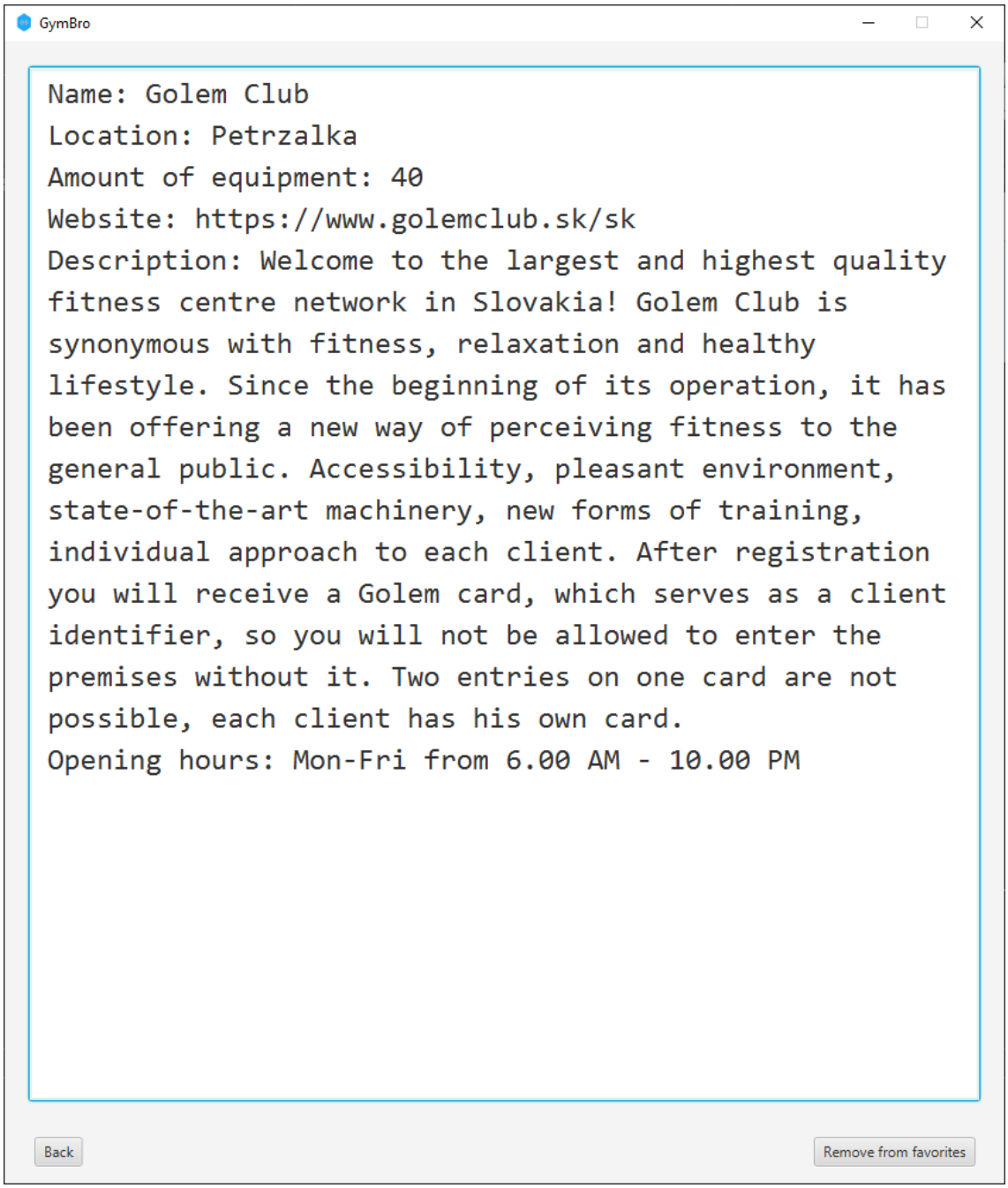


Figure 3 Gym overview

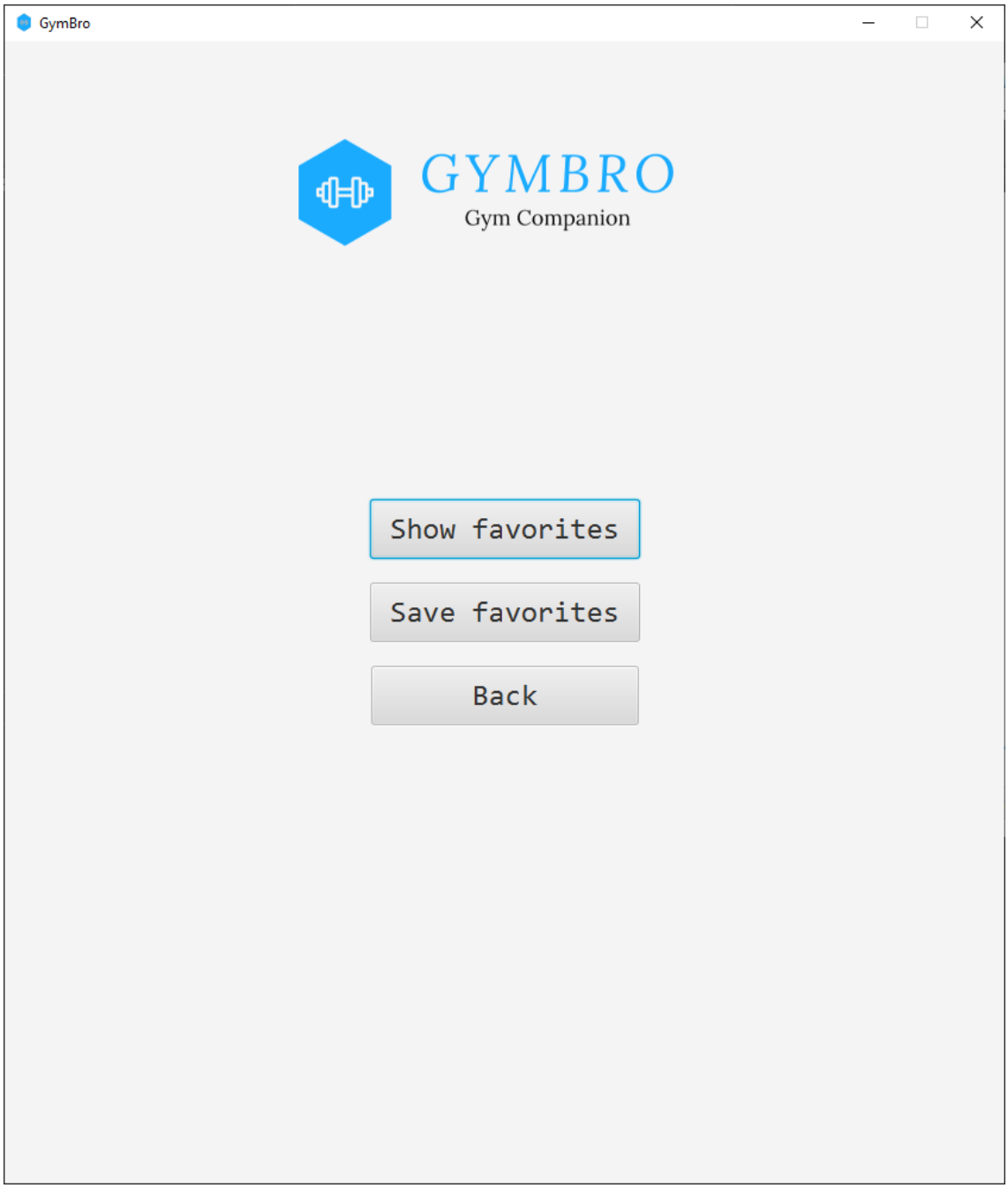


Figure 4 Favorites submenu

Figure 5 Favorites map

3 Requirements

Requirements for running GymBro are following:

- OS: Windows 10/11 or Debian-based Linux distro.
- Java JRE 11+.
- Monitor with 1920x1080 resolution.

4 Implementation

GymBro is implemented in the Java programming language using JavaFX and Gradle build system. The app focuses on Object-Oriented Programming (OOP) principles and design patterns.

4.1 Design patterns

Design patterns are solutions to common programming problems that have been developed and refined over time. The app's use of design patterns has helped to improve its functionality.

4.1.1 Builder pattern

The Builder pattern is a creational pattern that allows for the construction of objects. In GymBro, the Builder pattern is used to create the gym object. The GymBuilder class is responsible for setting the various attributes of the gym, such as the gym's name, hours of operation and more, and then returning the constructed gym object. This allows for the creation of complex gym objects without requiring a long constructor.

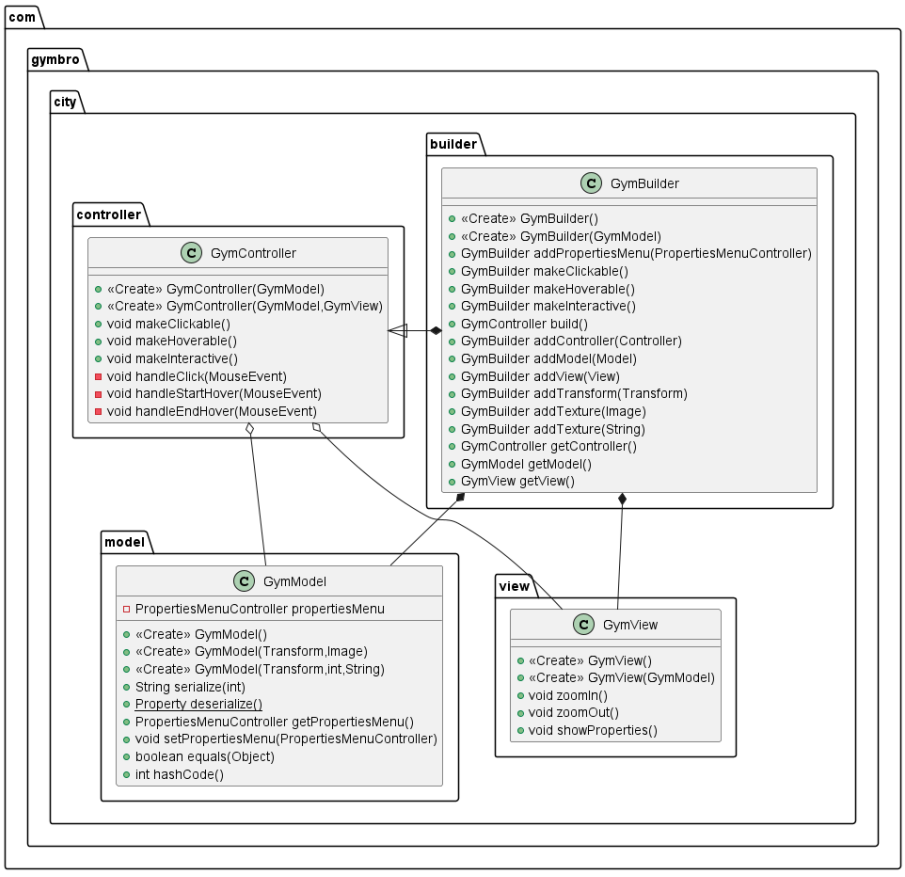


Figure 6 GymBuilder diagram

4.1.2 Factory pattern

The Factory pattern is another creational pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. In GymBro, the Factory pattern is used to create the different types of gyms. The GymFactory class is responsible for creating the appropriate type of gym based on the user's preferences. This makes it easy to add new types of gyms in the future without having to modify the existing code.

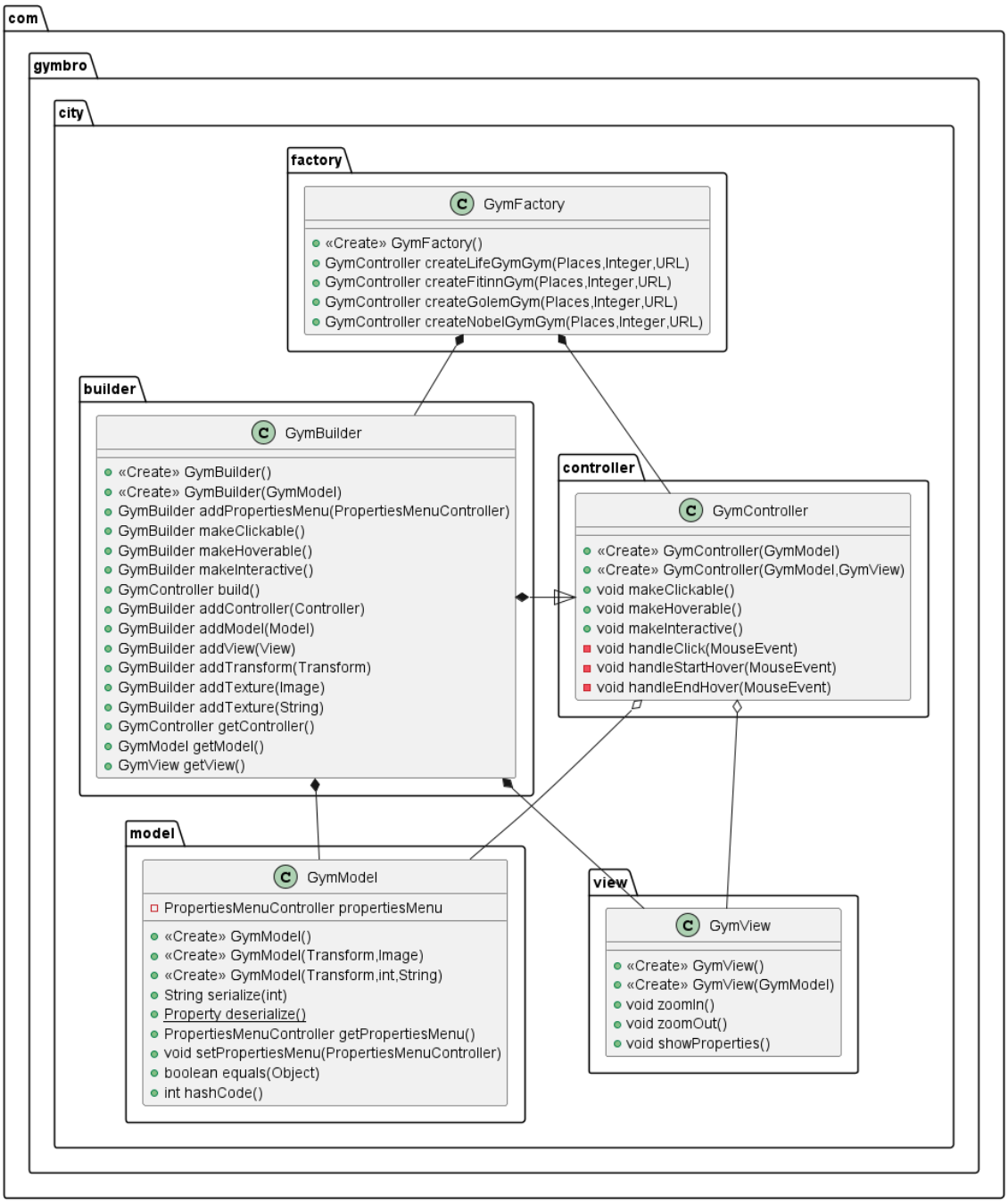


Figure 7 Factory design pattern diagram

4.1.3 Model-View-Controller Pattern

The Model-View-Controller (MVC) pattern is a widely used architectural pattern that separates the concerns of an application into three distinct components: the model, the view, and the controller. In GymBro, the MVC pattern is used to separate the application's logic (the model), the graphical user interface (the view), and the user input handling (the controller). This makes it easy to modify the application's behavior or interface without affecting the other components. To simplify the creation of an MVC project a builder pattern is used to achieve the best of both worlds. The code clarity of MVC and the easy creation of instances via builder.

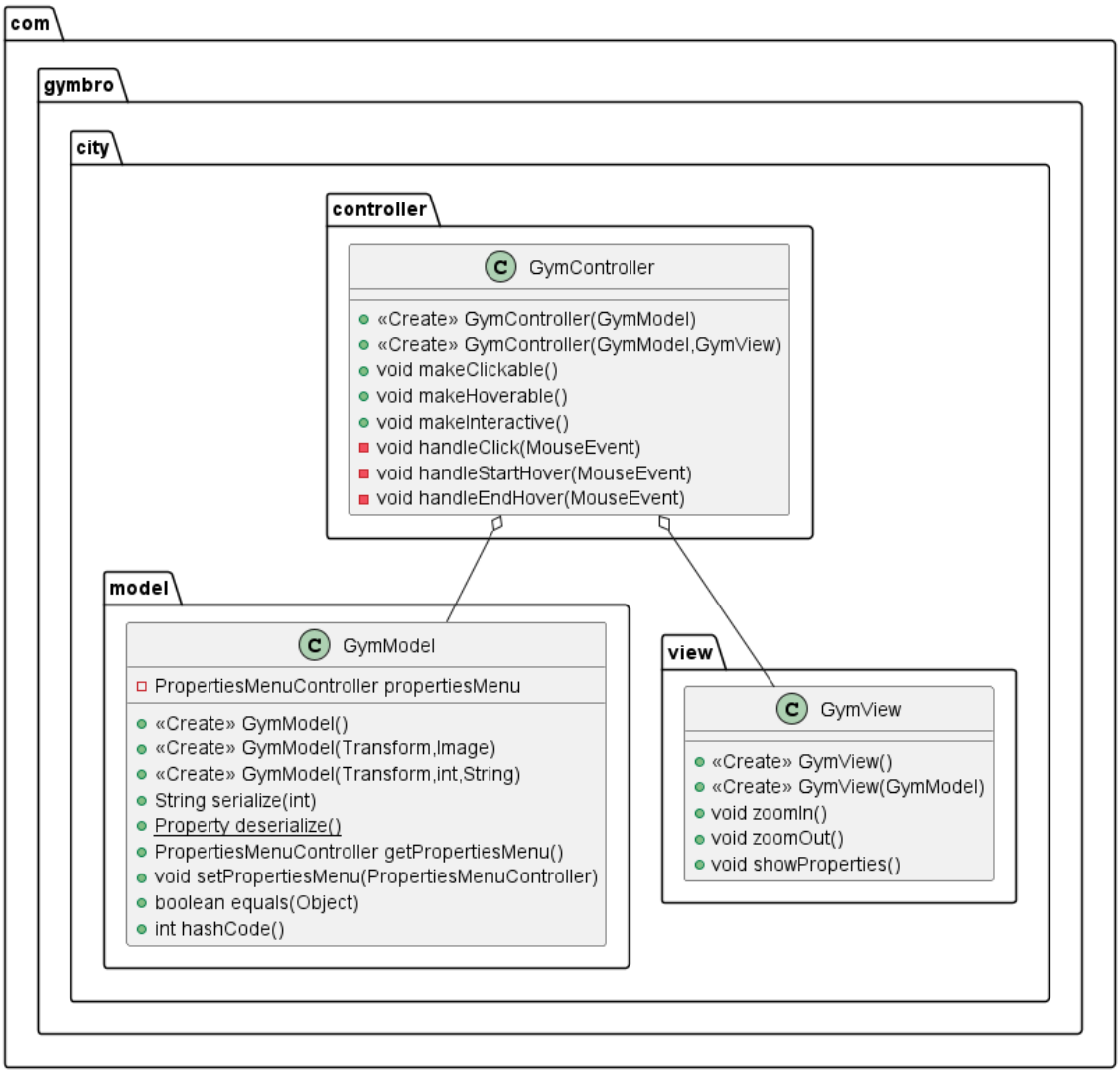


Figure 8 Model View Controller architectural design pattern

4.1.4 Singleton Pattern

The Singleton pattern is a creational pattern that ensures that a class has only one instance and provides a global point of access to that instance. In GymBro, the Singleton pattern is used for the Window class. The Window class is responsible for viewing all the gyms in the application. By using the Singleton pattern, we ensure that only one instance of the window exists, which ensures consistency throughout the application.

```
public class Window{

    3 usages
    private static Window instance = null;

    // ...
```

Figure 9 Declaring a private instance of class Window within the class.

```
public static Window getInstance(){
    if(Window.instance == null) Window.instance = new Window();
    return Window.instance;
}
```

Figure 10 Singleton getter function for private instance

4.2 OOP Principles

4.2.1 Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a new class to be based on an existing class, inheriting all its properties and behaviors. In GymBro, inheritance is used to create specialized classes that have additional functionality beyond what is provided by the base class.

For example, the MapController class inherits from the Controller class, which provides basic functionality for handling user input and updating the application's state. The MapController class adds specific functionality for handling map-related actions, such as displaying the gym locations on the map and allowing users to search for gyms in their area.

Inheritance allows for code reuse and promotes the creation of well-organized and modular code. By creating specialized classes that inherit from a base class, developers can avoid duplication of code and create a more maintainable and scalable application. There are 9 inheritance hierarchies in the implementation as well as 1 interface hierarchy (Serialization interface). See the 3rd section of this documentation to get a full overview of inheritance hierarchies. Here is GymBro's inheritance hierarchy structure:

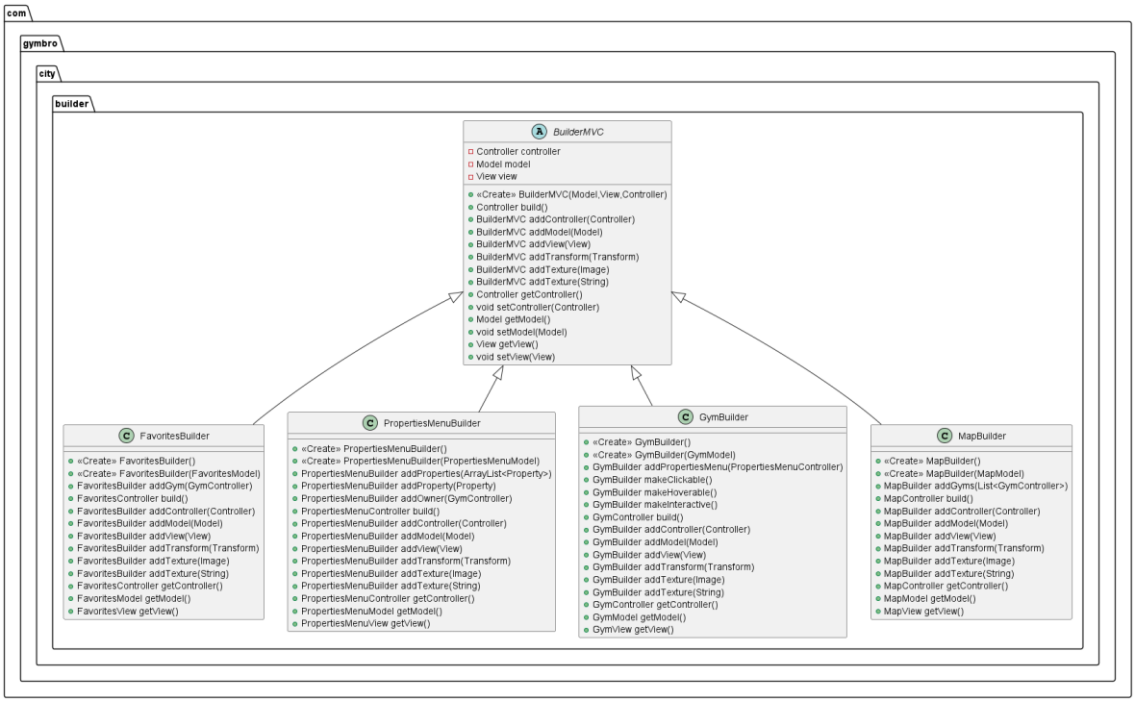


Figure 11 Inheritance hierarchy #1

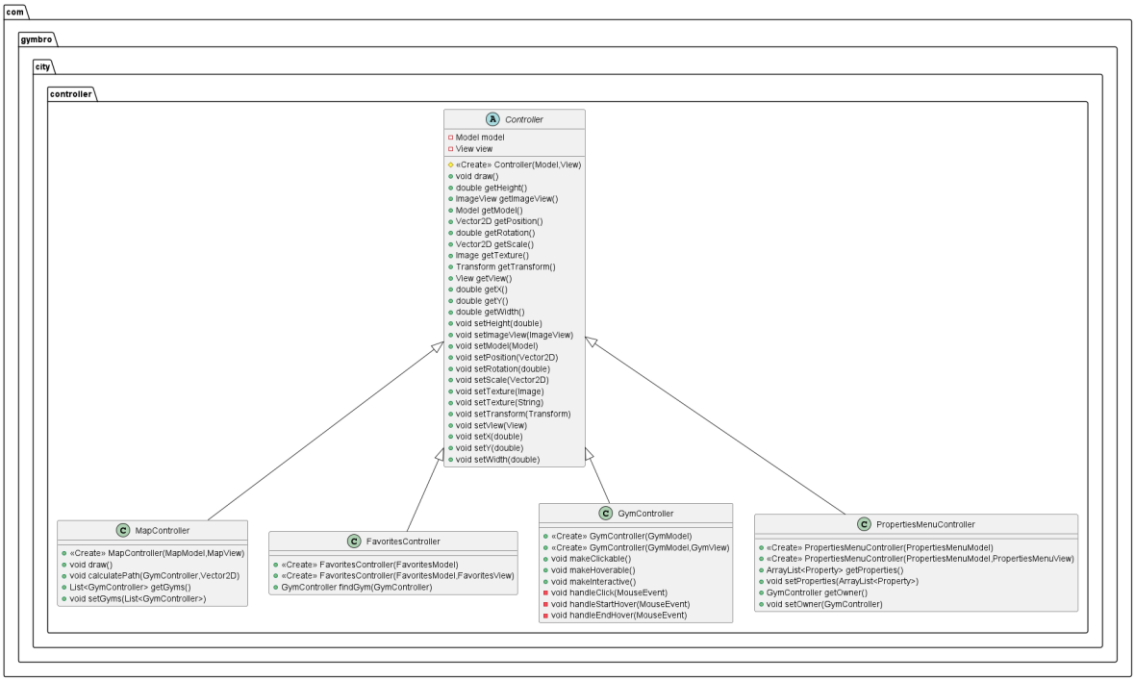


Figure 12 Inheritance hierarchy #2

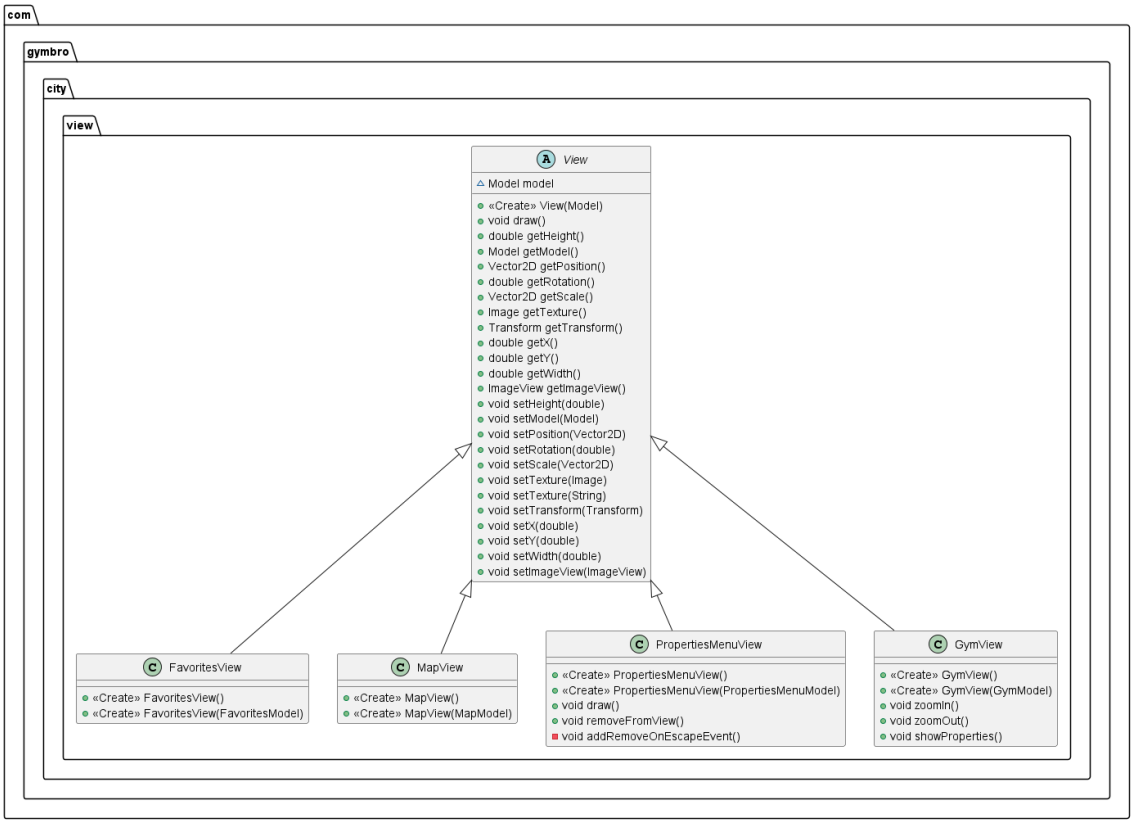


Figure 13 Inheritance hierarchy #3

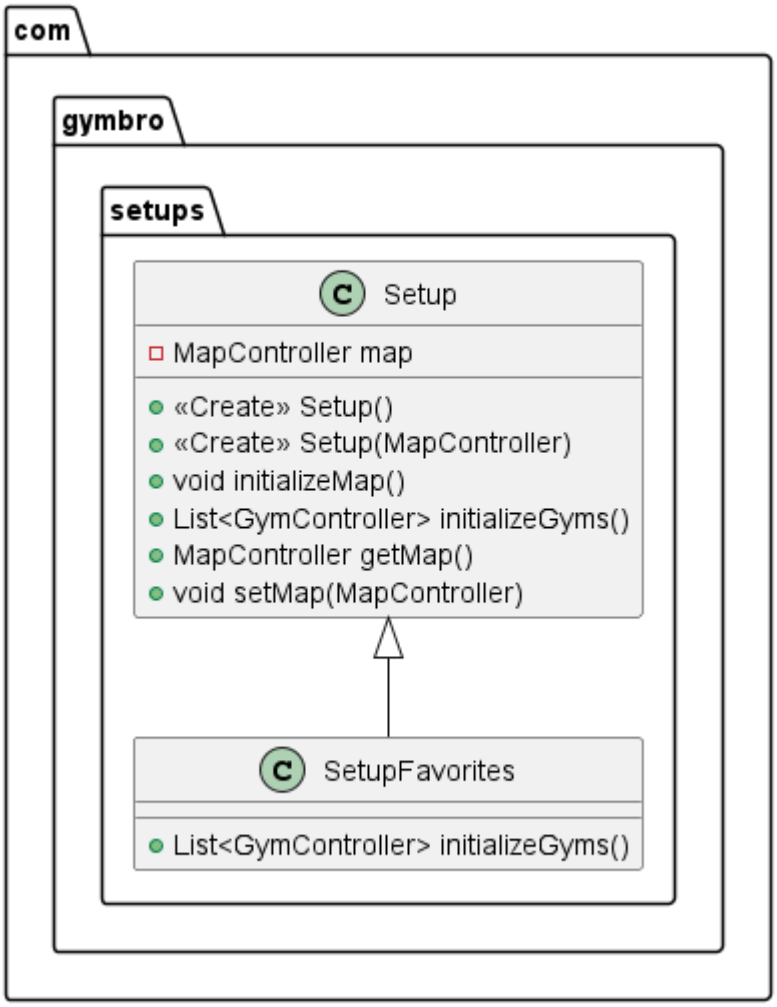


Figure 14 Inheritance hierarchy #4

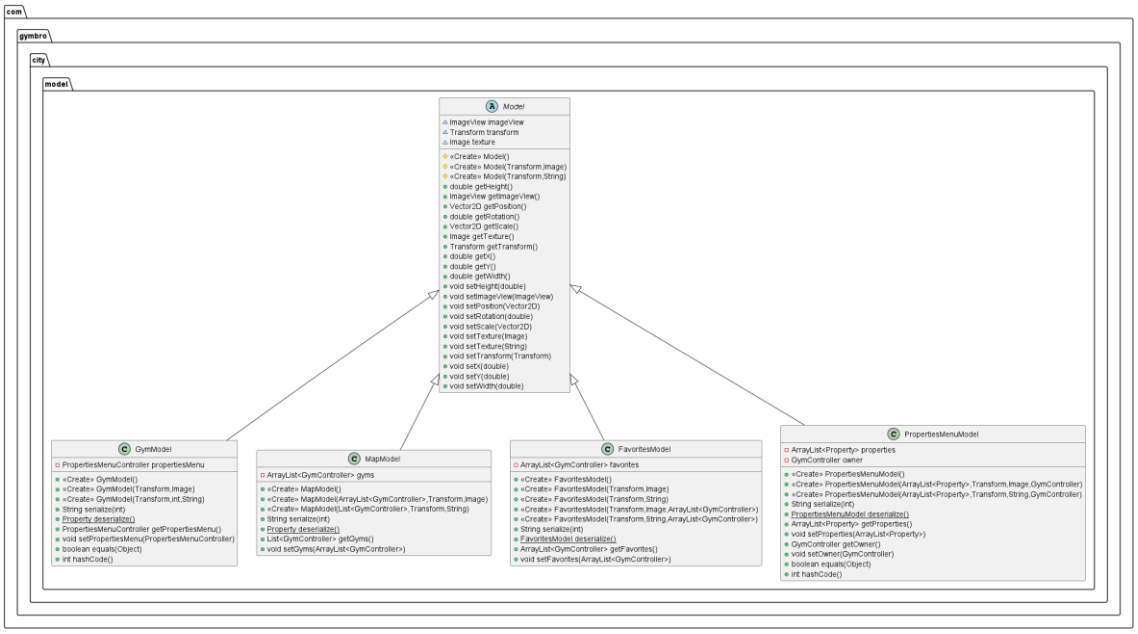


Figure 15 Inheritance hierarchy #5

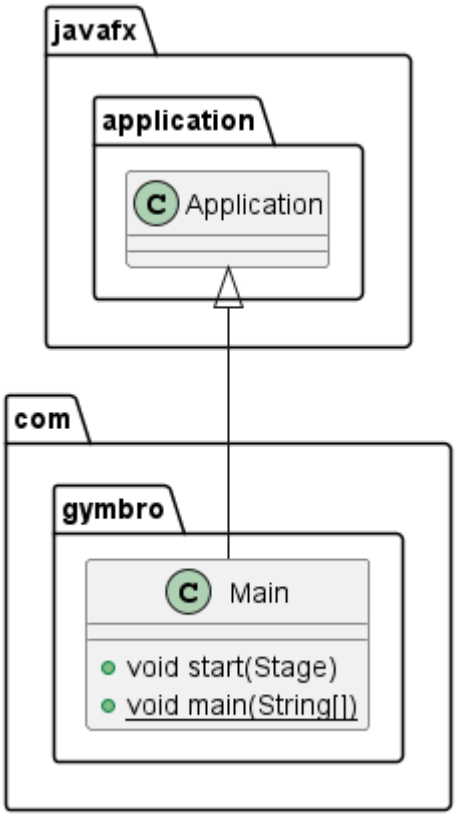


Figure 16 Inheritance hierarchy #6

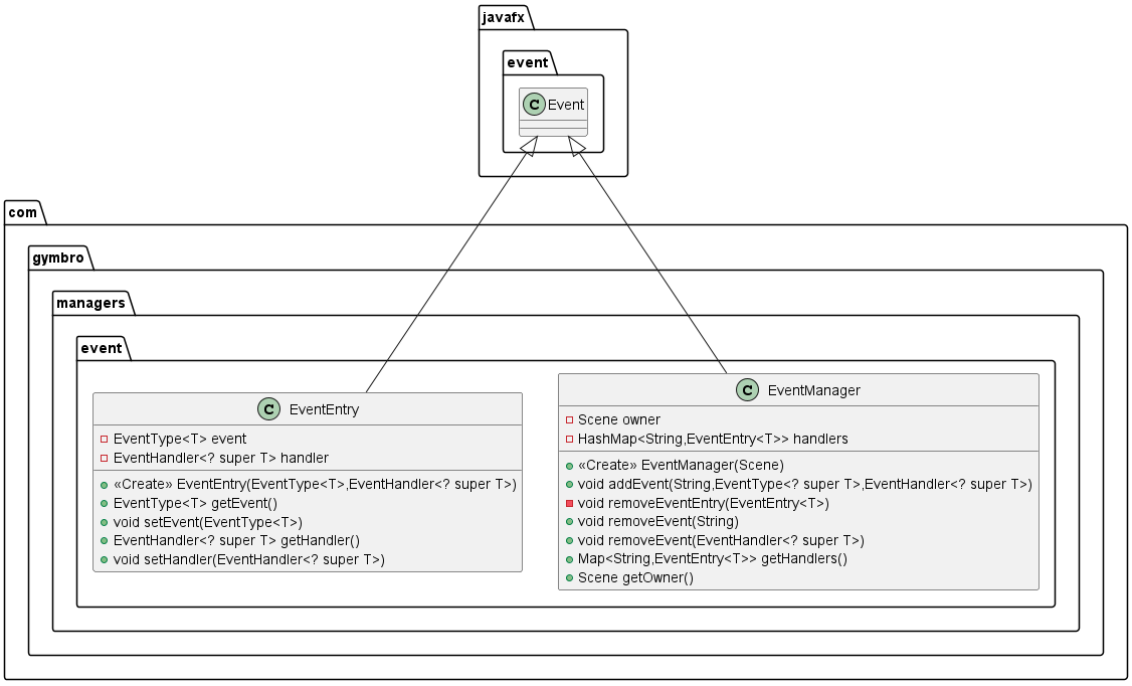


Figure 17 Inheritance hierarchy #7

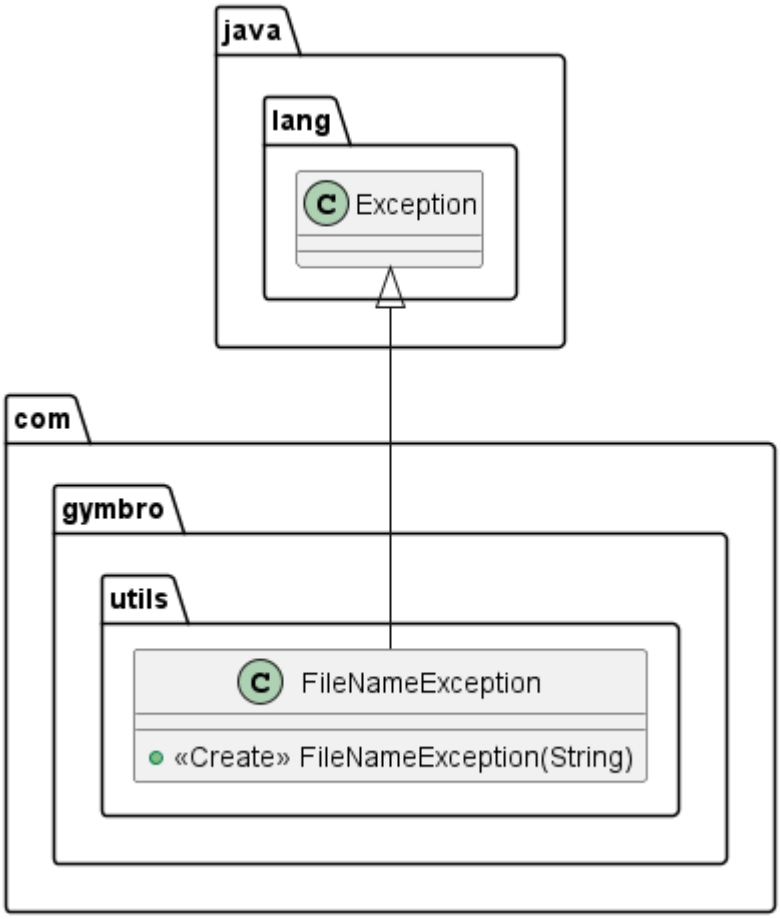


Figure 18 Inheritance hierarchy #8

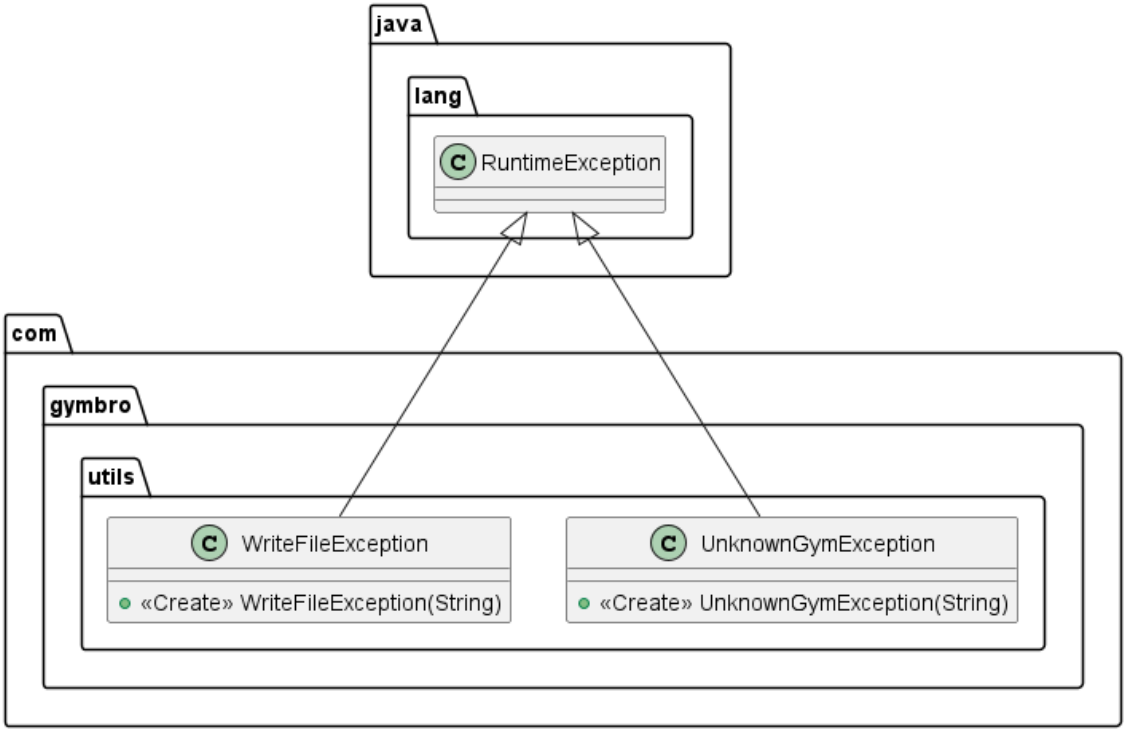


Figure 19 Inheritance hierarchy #9

Figure 20 Interface hierarchy #1

4.2.2 Encapsulation

Encapsulation is an important concept in object-oriented programming that refers to the practice of hiding the internal details of an object and exposing only the necessary information to the outside world. This helps to improve the security, maintainability, and reliability of an application.

In GymBro, encapsulation is used extensively to protect the data and behavior of the application's classes. For example, the Gym class encapsulates all the data associated with a gym, such as its name, address, and hours of operation. This data is accessed only through the public methods of the class, which provide a controlled and secure way to modify the data. 44 out of 47 variables in the implementation use encapsulation. Only constants don’t as they are not prone to any change.

4.2.3 Aggregation

Aggregation is a relationship between two objects where one object contains a reference to another object, but the two objects are not dependent on each other. In other

words, one object can exist without the other. Aggregation is represented in UML diagrams using a diamond-shaped arrow pointing from the containing class to the contained class. In GymBro, an example of aggregation is the relationship between the MapModel and the Gym class. The MapModel class contains a collection of Gym objects, but the Gym objects do not depend on the MapModel. GymBro's implementation has 28 cases of aggregation inside the code. See the 3rd section of this documentation to get a full overview of aggregation hierarchy in GymBro.

4.2.4 Composition

Composition is a stronger form of aggregation where one object is composed of one or more other objects, and the composed objects cannot exist without the parent object. In other words, the parent object is responsible for the creation and destruction of the composed objects. Composition is represented in UML diagrams using a filled diamond-shaped arrow pointing from the containing class to the contained class. In GymBro, an example of composition is the relationship between the GymView and the MapView class. The GymView is composed of a MapView, which is responsible for rendering the map. Overall GymBro's implementation consists of 28 compositions. See the 3rd section of this documentation to get a full overview of composition hierarchy in GymBro.

4.2.5 Polymorphism

Polymorphism is a concept in object-oriented programming that allows objects of different classes to be treated as if they were objects of the same class. This is achieved through inheritance and interfaces. Inheritance allows a subclass to inherit properties and methods from a superclass, while interfaces define a set of methods that must be implemented by a class. Polymorphism allows for more flexible and modular code, as objects of different types can be treated in the same way. In GymBro, an example of polymorphism is the relationship between the GymController and the MapController. Both classes inherit from the Controller class and can be treated as objects of the same type, even though they have different implementations of the methods defined in the Controller class. In total there are 7 classes that contain polymorphism. They consist of 12 object instances. Here are a few examples from the code.

1. Class FavoritesModel: The "model" object implements polymorphism.
2. Class PropertiesMenuModel: The "property" object implements polymorphism, and the "model" object implements polymorphism.

3. Class MainMenuFxmlController: The "controller" object implements polymorphism two times.
4. Class Parser: The "model" object implements polymorphism three times.
5. Class PropertiesMenuView: The "property" object implements polymorphism.

4.2.6 Overriding

Overriding is an OOP principle where a class inherited from another class rewrites the implementation of an inherited method. Another case of overriding is if an interface declares methods but leaves the implementation up to the class that is implementing said interface. The class that implements this interface must override all methods that are declared in the interface. It is primarily used in MVC hierarchy as well as Serialization interface. Example from Serialization interface:

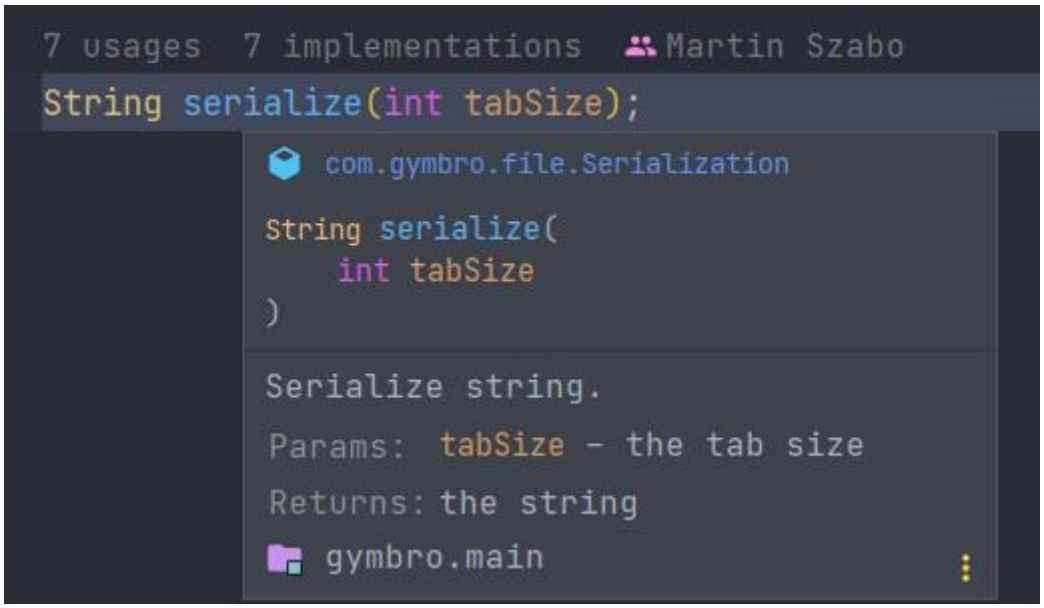


Figure 21 Serialize method declaration in Serialization interface.

```
@Override
public String serialize(int tabSize){
    StringBuilder builder = new StringBuilder();

    builder.append(beginObjectProperty( name: "GymModel", tabSize));

    builder.append(this.getTransform().serialize( tabSize: tabSize + 1));
    builder.append(addEnding( newline: true, comma: true));

    builder.append(addStringProperty( name: "Texture", this.getTexture().getUrl(), tabSize: tabSize + 1, newline: true, comma: true));

    if(this.getPropertiesMenu() != null){
        builder.append(this.getPropertiesMenu().getModel().serialize( tabSize: tabSize + 1));
    }

    builder.append(addEnding( newline: true, comma: false));
    builder.append(endObjectProperty(tabSize));
    return builder.toString();
}
```

Figure 22 GymModel's implementation (override) of serialize method.

4.2.7 Overloading

Overloading is another OOP principle where a class has two methods of same name but different arguments (types and count of arguments). Overloaded methods are great when you need to handle multiple cases of input arguments differently.

```java
/**
 * Set texture.
 *
 * @param texture the texture
 */
Martin Szabo
public void setTexture(Image texture){
    this.texture = texture;
    this.setImageView(new ImageViewHandler().initialize(
            this.getTexture(),
            this.getTransform()
    ));
}

/**
 * Set texture.
 *
 * @param resource the texture path
 */
Martin Szabo *
public void setTexture(String resource){
    try{
        String path = Objects.requireNonNull(Main.class.getResource(resource)).toExternalForm();
        this.texture = new Image(path);


        this.setImageView(new ImageViewHandler().initialize(
            this.getTexture(),
            this.getTransform()
        ));
    }
    catch(Exception ignored){} Either remove or fill this block of code.
}
```

4.3 Generic classes

In the provided code, there is a generic class called EventManager. It is used to manage event handling in a JavaFX scene. The class has a type parameter T that extends Event, which means it can be used with any subtype of the Event class.

The EventManager class has three fields: owner, handlers, and event. The owner field represents the JavaFX scene that the event manager is associated with. The handler's field is a HashMap that stores the event handlers associated with the event manager. The event field represents the type of event that is being handled.

The addEvent method is used to add a new event to the event manager. It takes an id parameter that represents a unique identifier for the event, an eventType parameter that represents the type of event being added, and a handler parameter that represents the event handler associated with the event. The method tries to cast the eventType to the EventType<T> type and adds the event handler to the owner scene and the handler's map.

The removeEvent method is used to remove an event from the event manager. It takes an id parameter that represents the unique identifier for the event to be removed or a handler parameter that represents the event handler to be removed. The method uses the getHandlers method to retrieve the event entry associated with the given id or handler. It then removes the event entry from the owner scene and the handler's map.

The getHandlers method returns the HashMap of event handlers associated with the event manager. The getOwner method returns the owner JavaFX scene associated with the event manager.

The EventEntry class is also a generic class used to represent an entry in the EventManager handler's map. It has two fields: event represents the type of event being handled, and handler represents the event handler associated with the event. The getEvent method returns the type of event being handled, and the getHandler method returns the event handler associated with the event.

```
public class EventManager<T extends Event>{
    2 usages
    private final Scene owner;
    2 usages
    private final HashMap<String, EventEntry<T>> handlers;

    // ...
```

Figure 23 Event Manager generic class declaration

```
public class EventEntry<T extends Event>{

    3 usages
    private EventType<T> event;
    3 usages
    private EventHandler<? super T> handler;

    // ...
```

Figure 24 Event Entry generic class declaration

4.4 Serialization interface and default methods

The purpose of this interface is to define a contract that any implementing classes must follow to be serializable, meaning they can be transformed into a byte stream for storage or transmission.

The default keyword before the method signature means that a default implementation of the method is provided, which is used if the implementing class does not override it.

To make a class serializable, it must implement the Serializable interface and provide its own implementation of the serialize() method. By doing so, the class agrees to follow the contract defined by the Serializable interface and can be safely serialized.

After implementing the interface, the class can use provided default methods to serialize its attributes with ease.

```
public interface Serialization{

    /**
     * Add string property string.
     *
     * @param name    the name
     * @param value   the value
     * @param tabSize the tab size
     * @param newline the newline
     * @param comma   the comma
     * @return the string
     */
    3 usages   Martin Szabo
    default String addStringProperty(String name, String value, int tabSize, boolean newline, boolean comma) {
        return addTabs(tabSize) + "\"" + name + "\": \"" + value + "\"" + addEnding(newline, comma);
    }

    // ...
```

Figure 25 Serialization interface and one of the default methods

4.5 CRUD Principle

The GymBro application includes a simple favorites system for gyms. This system implements the CRUD (Create, Read, Update, Delete) principle, allowing users to add new gyms to their list of favorites, view their favorite gyms, update the information associated with a favorite gym, or remove a gym from their list of favorites. This system provides users with a simple way to manage their list of favorite gyms.

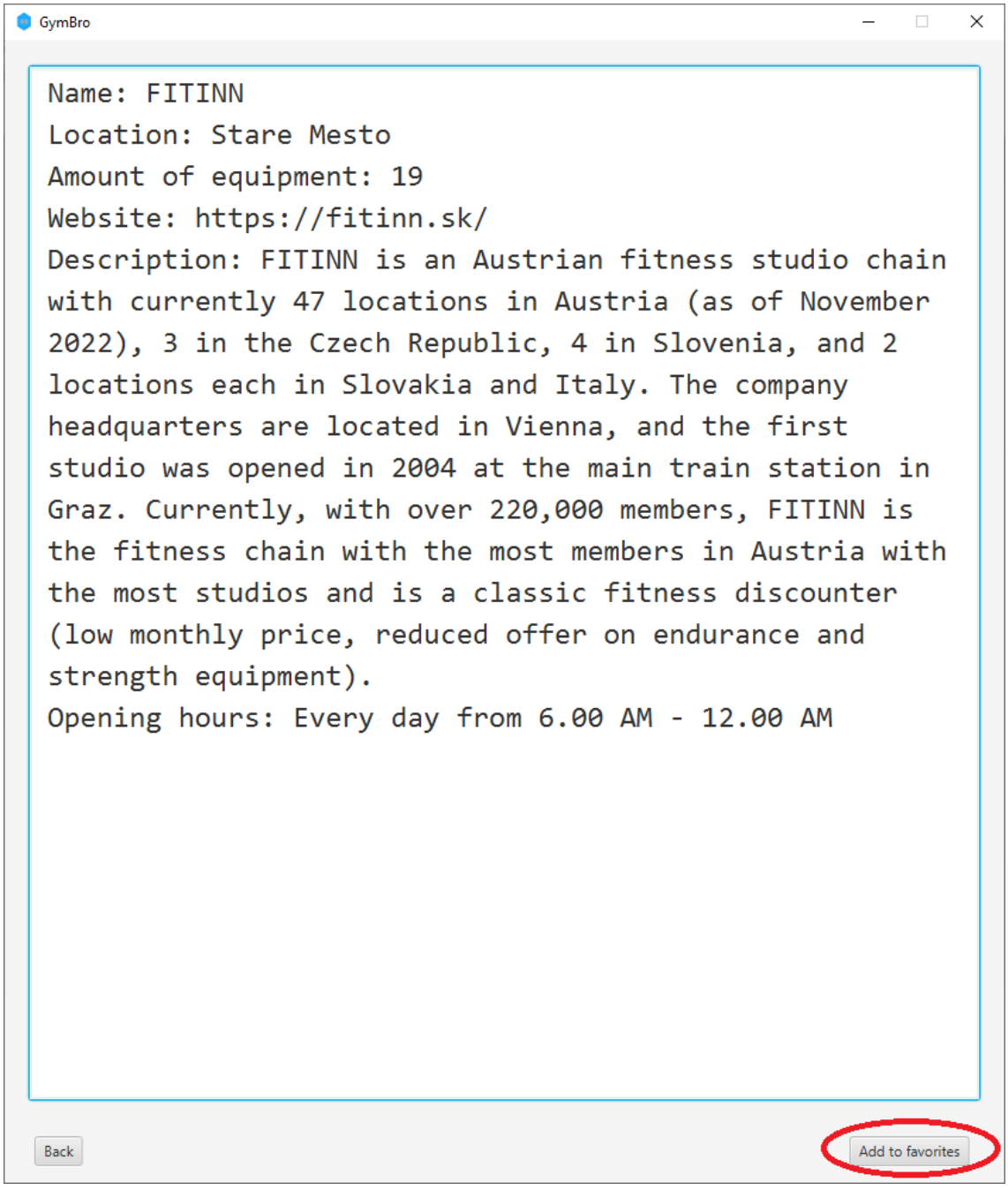


Figure 26 Add to favorites button.

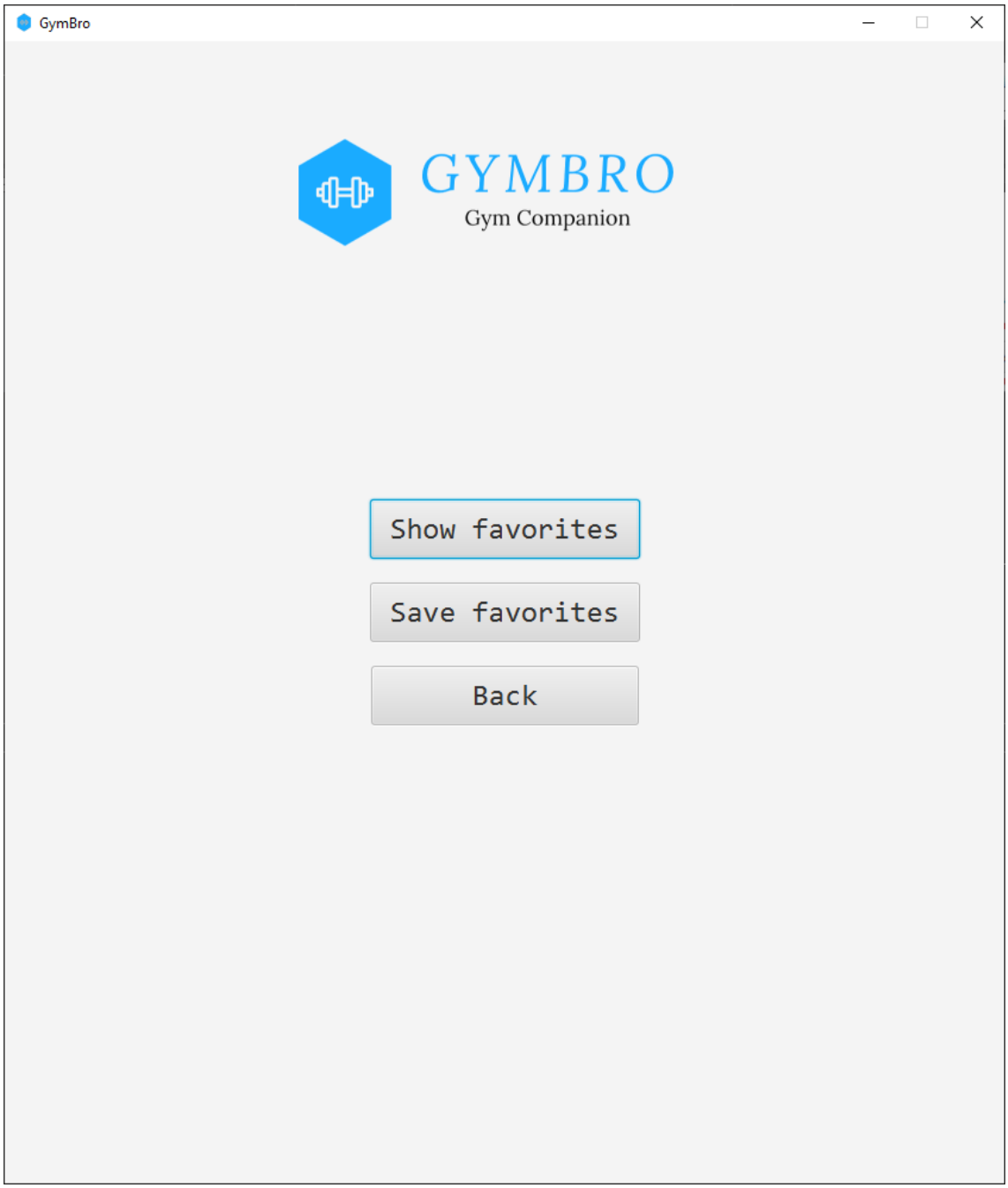


Figure 27 Show favorites button.

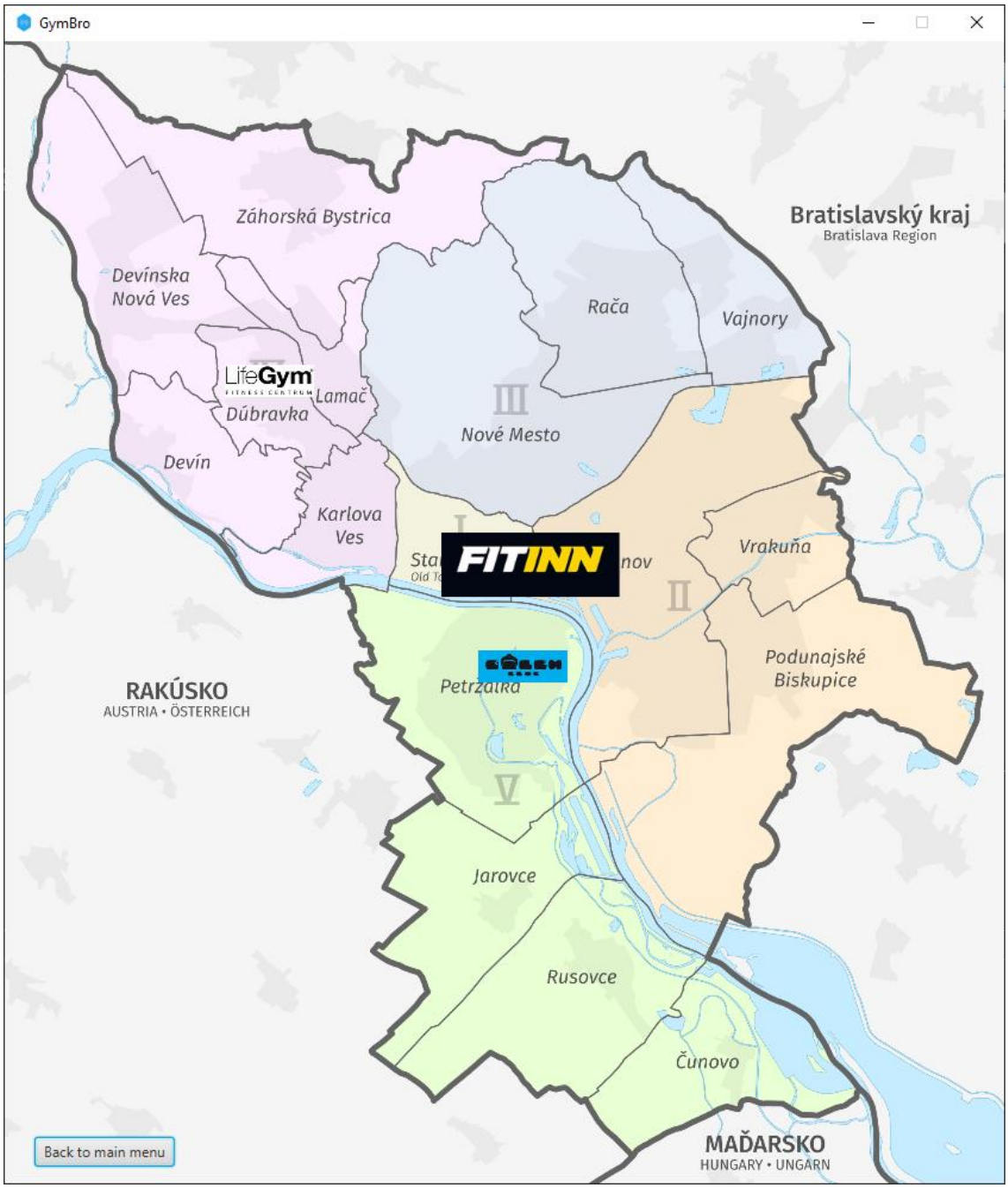


Figure 28 Hovering over the FITINN gym that has been added to the to the favorites.

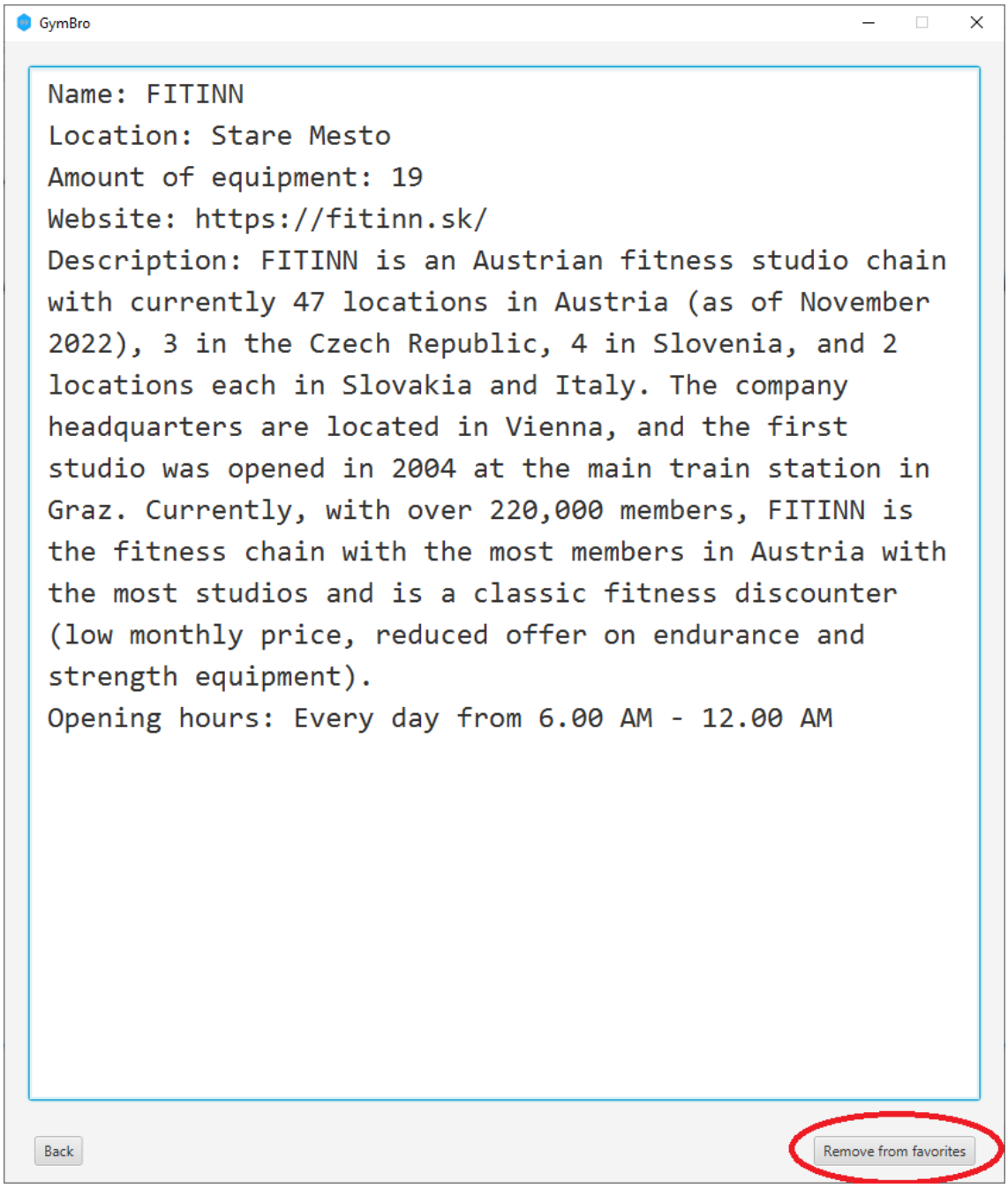


Figure 29 Gym that has been added to the favorites.

4.6 Lambda

The GymBro application uses lambda expressions in a few places to simplify code and improve readability. For example, when handling button events in the GUI, lambda expressions are used to define the actions that should be taken when a button is clicked. Lambda expressions are also used in conjunction with Java Streams to simplify filtering and sorting of data.

```
EventHandler<KeyEvent> backToMainMenu = keyEvent → {
    if(
        keyEvent.getCode().equals(KeyCode.ESCAPE) &&
        this.getSceneManager().getActiveName().equals(Constants.ID_MAP_PANE)
    ){
        this.handleBackToMainMenu();
    }
};

this.getEventManager().getKeys().addEvent(
    Constants.ID_BACK_TO_MAIN_MENU_EVENT,
    KeyEvent.KEY_PRESSED,
    backToMainMenu
);
```

Figure 30 Lambda expression / function

4.7 Multithreading

Multithreading is used in the GymBro application to ensure that the GUI remains responsive even when performing potentially long-running tasks, such as loading data. Background threads are used to perform these tasks, allowing the user to continue using the application without interruption.

```
mainThread = new Thread(() → Main.main(args), name: "mainThread");
mainThread.start();
```

Figure 31 Multithreading example

4.8 Custom exceptions

Your solution has implemented three custom exceptions: FileNameException, UnknownGymException, and WriteFileException. These exceptions are thrown in specific cases to handle errors. For example, FileNameException is thrown when the specified file cannot be opened for parsing in Parser.java. UnknownGymException is thrown in GymsHandler class when an unexpected value of gym type is encountered. WriteFileException is thrown when the file cannot be written to in FavoritesHandler class.

4.8.1 RuntimeException example (UnknownGymException)

```
public class UnknownGymException extends RuntimeException{
    1 usage   Martin Szabo
    public UnknownGymException(String message){
        super(message);
    }
}
```

Figure 32 UnknownGymException implementation

```
            // ...

        case "NobelGym":
            return gymFactory.createNobelGymGym(location, amountOfEquipment, website);
        default:
            throw new UnknownGymException("Unexpected value of gym type: " + type);
```

Figure 33 Throwing UnknownGymException

4.8.2 Normal exception example (FileNameException)

```
public class FileNameException extends Exception{
    2 usages   Martin Szabo
    public FileNameException(String message){
        super(message);
    }
}
```

Figure 34 FileNameException declaration

```
public static void openFile(String fileName) throws FileNameException{

    // ...
```

Figure 35 Declaring that openFile throws FileNameException.

```
        // ...

        catch(IOException e){
            e.printStackTrace();
            throw new FileNameException("Could not open file for parsing!");
        }
```

Figure 36 Throwing the FileNameException from openFile method.

4.9 Application logic

Application logic handles manipulation of data through various classes. User interacts with GUI. GUI then calls handlers and various other logic classes to manipulate data in the application. Logic classes then provide GUI with the correct data to display.

4.10 Graphical user interface

The GymBro application includes a graphical user interface that allows users to interact with the application in a visual way. The GUI includes a list of favorite gyms, a form for adding new gyms, and buttons for managing favorites. The GUI was designed with simplicity and ease-of-use in mind, providing users with an intuitive interface for managing their list of favorite gyms.

4.11 Handlers and managers

The GymBro application includes event handlers and managers to handle user interactions with the GUI. These handlers are responsible for responding to button clicks, form submissions, and other user actions. By using event handlers, the application can provide feedback to the user in real-time and ensure that the user's actions are performed correctly.

```
public class SceneEventManager{
    3 usages
    private EventManager<? extends KeyEvent> keyEventManager; Rep
    3 usages
    private EventManager<? extends MouseEvent> mouseEventManager;
    3 usages
    private Scene owner;
```

Figure 37 SceneEventManager manager example

4.12 Package structure

The project has a well-organized package structure that separates the components of the system into logical groupings. The main package is com.gymbro, which contains sub-packages for the various modules of the system, including city, setups, and utils.

The city package contains the Model-View-Controller (MVC) architecture components of the system. The model sub-package contains the classes for the different models of the system, including GymModel, MapModel, FavoritesModel, and PropertiesMenuModel. The view sub-package contains the classes for the different views of the system, including GymView, MapView, FavoritesView, and PropertiesMenuView. The controller sub-package contains the classes for the different controllers of the system, including GymController, MapController, FavoritesController, and PropertiesMenuController. Additionally, the builder sub-package contains the classes for the different builders of the system, including GymBuilder, MapBuilder, FavoritesBuilder, and PropertiesMenuBuilder.

The setups package contains the classes related to setting up the system, including the Setup class and the SetupFavorites class.

The utils package contains various utility classes, including the Vector2D class, which represents a two-dimensional vector, the Property class, which represents a key-value pair of properties, and the Exception class, which contains the exceptions of the system, including FileNameException, UnknownGymException, and WriteFileException.

Overall, the package structure is organized, modular, and follows the best practices of software design, making it easy to understand and maintain.

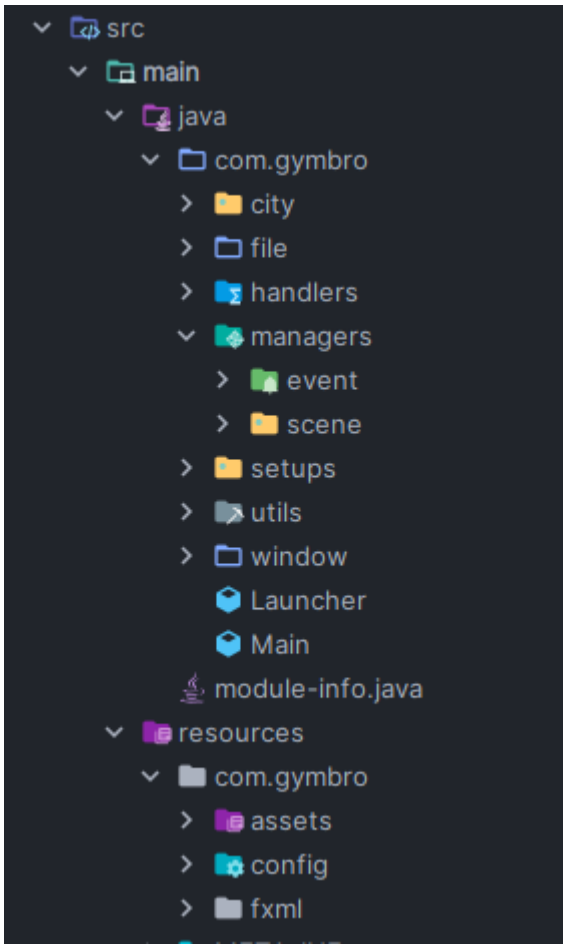
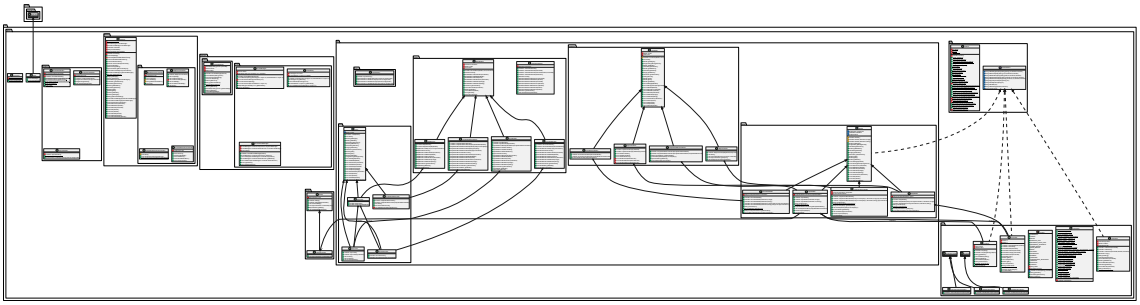


Figure 38 Package structure of GymBro

5 Class diagram

The class diagram of GymBro provides a visual representation of the various classes, interfaces, and their relationships in the system. It helps to understand the overall architecture of the system and how different components are related to each other. For better readability a .SVG diagram file is included with this documentation so it can be zoomed in without quality loss.



6 Conclusion

GymBro is an innovative and intuitive desktop application that simplifies the process of finding the perfect gym to fit your workout needs. The use of JavaFX, Gradle, and Object-Oriented Programming principles have resulted in a feature-rich application that is user-friendly and efficient. The implementation details of design patterns such as the Builder, Factory, Model-View-Controller, and Singleton patterns have significantly improved the functionality and user experience of the application. Furthermore, the implementation of OOP principles such as inheritance, encapsulation, and polymorphism has resulted in a well-organized and modular codebase that is easy to maintain and scale. Overall, GymBro is a must-have application for anyone who takes their fitness seriously and wants to achieve their fitness goals with ease and convenience.