Giulia Bondi Mazzoni - Matr. 260585
Object-oriented programming and modelling - project report

JoggingPal - Project report

1. Software specifications

This project consists in the development of a Windows application aimed at runners and jogging enthusiasts. The application allows registered users to organize and participate in events, that is appointments in which groups of users (e.g. groups of friends or members of a sports team) go running together. After each event participants can compare results. Events can be of two types: in-person-events in which participants meet up and run at the same location, or virtual events in which participants go running at the same time, however each one at a different location chosen independently. Registered users can also create user groups, which can be signed up to events as a whole by the group administrator. Users can join or leave a group at any time. As a future development, the application will allow the creation of multi-level user groups, so that user groups can join other groups.

2. Analysis of the problem

The construction of the application described above requires modelling a number of items of the real world, namely users, events, running locations and event results.
Users need to be able to organize events themselves or to join events created by others, as a consequence the application needs to support usage by multiple users. In order to distinguish which user is currently using the application some form of login is required.
Events can be of either one of two types: in-person or virtual. Both types of events share a number of characteristics, like date and time, a list of participants and an average running speed, which gives an indication of the level of difficulty of the event. In-person-events take place at a specific location, which suggests the existence of a link between an event of this type and a location. Virtual events take place at different locations, which can be chosen by users after signing up to the event. This shows a link between an event participant and a running location. Virtual events also indicate the approximate length of the route, so that users can choose running routes that are as similar as possible.
Running locations indicate the length of the running route and the geographic coordinates for its starting point.
Event results can be uploaded by users after participating in an event and can be compared with other participants. The types of data uploaded to the results can vary from one participant to another. For example a participant may go running using a simple stopwatch which only records the total time required to cover a route, whereas another participant may use a smartphone or a wearable device which records several other parameters. For this reason total

time is considered required information, whereas other data (i.e. maximum speed and average heart rate) are optional.
In order to accommodate some of the functions described above (e.g. selection of a running location, uploading of event results) the application needs to implement a sequence of states for the single participant. These states are:

- Signed up. A participant has signed up to an event, however a running location has not yet been set.
- Location set. A running location for the event has been set, either by the organizer in case of an in-person-event or by the individual participant in case of a virtual event.
- Checked in. A participant confirms that he/she is going to participate in an event. Ideally this state should indicate that the participant has reached the running location at the specified date and time and is about to start running. However this would only be possible with a mobile version of the application.
- Event results uploaded. After participating in the event, the participant has signed up his/her results which can now be viewed by other participants.

The application needs to manage a large amount of information which users can upload, edit and view. For this purpose a database has been implemented, however this first version of the application does not provide persistency of data.

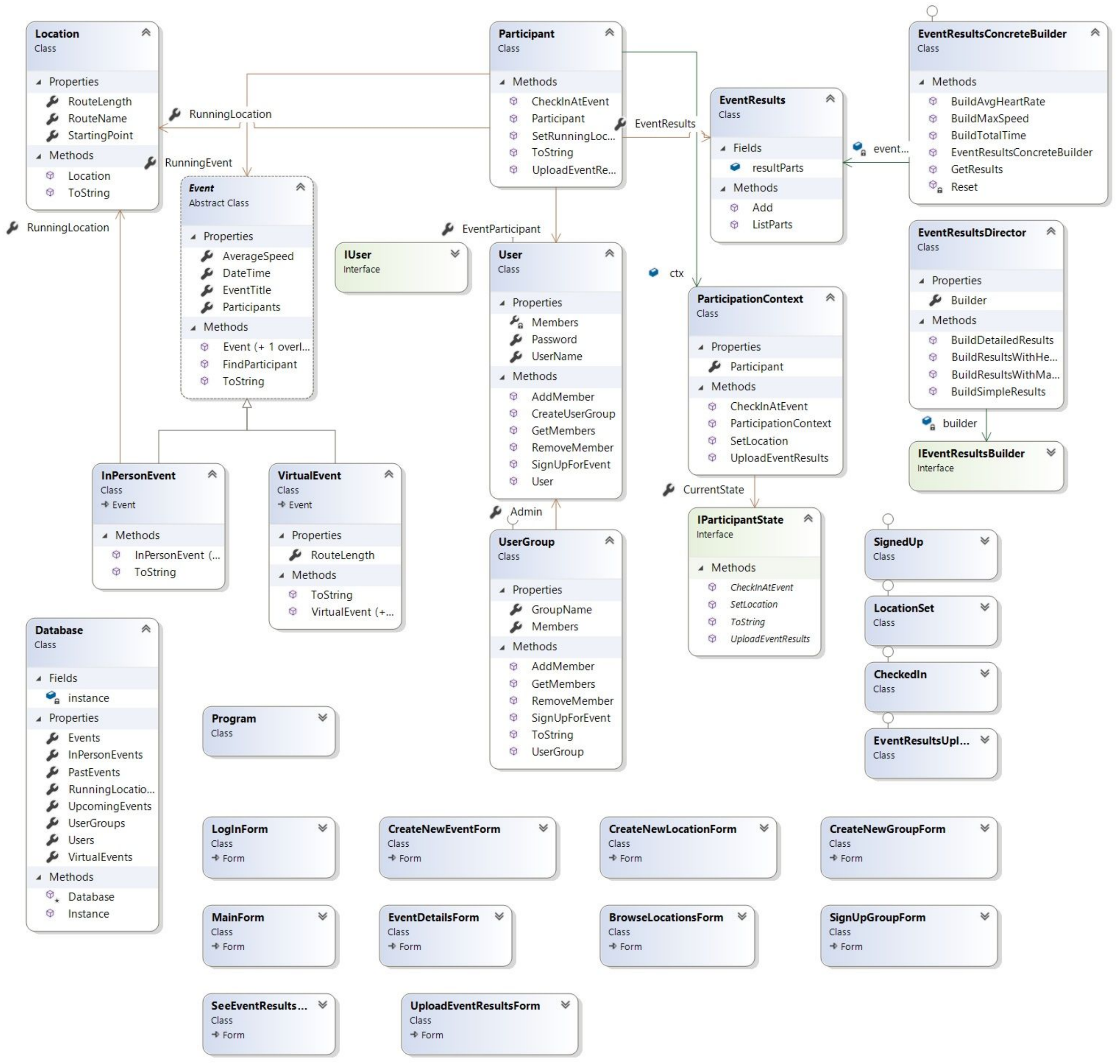
3.1 Description of the software architecture

The project comprises two namespaces: models, containing those classes which represent elements of the real world, and database, containing a database class. The class program and all classes related to the GUI were not included in a namespace.

3.1.2 Models

The namespace models contains the classes Event, Location, Participant, ParticipationContext (for managing the participant states described above), EventResults, User and UserGroup. Class members are implemented as C# properties.
Event was implemented as a class hierarchy consisting of the superclass Event and the two subclasses InPersonEvent and VirtualEvent. Event exposes the properties DateTime, AverageSpeed and EventTitle, which are common to both types of events. InPersonEvent specializes the superclass with the attribute Location, in order to implement the link between an event of this type and the running location. The class VirtualEvent, instead, specializes Event with the attribute RouteLength.
The class Participant contains an object of type User and an object of type Event, in order to implement the many-to-many relationship between the two classes. It also contains an object of type Location. If the Event is of type in-person, the attribute Location will contain the same Location instance as the event. However, if the Event is of type virtual, Location is set with the method SetLocation() of the class Participant. This method implements the possibility of selecting a personal running location after signing up for a virtual event.

The behavior of the remaining classes of the namespace Models will be presented in the “Application of design patterns” section.



3.1.3 Database

The class Database contains an example set of data for users, events, locations, event results and user groups. Instances of the same class are saved in Dictionaries which can be accessed by means of properties. In particular the get property of the Dictionaries InPersonEvents and VirtualEvents loop through the Events dictionary and fill the dictionary with objects of the relevant type. This implementation has the advantage that new objects only need to be saved to the Events dictionary, but the two specialized dictionaries are always up to date whenever a subtype-specific operation is required. The dictionaries UpcomingEvents and PastEvents function based on the same principle, however events are compared to the current date and time and added to the relevant dictionary.

3.1.4 Graphical user interface

The classes related to the GUI extend the class Windows Form. They display the content of Database, validate user input and call the constructors for the classes contained within the namespace Models.

3.2 Application of design patterns

The following paragraphs provide an overview of the design patterns applied to the application implementation.

3.2.1 State pattern - The classes Participant and ParticipationContext

The class Participant is instantiated when a user signs up to an event. However, the operations that the class can perform depend on the sequence of states described in paragraph 2. For example, users can only select a running location after they sign up to a virtual event and not before signing up. Analogously users can only upload their event results after checking in (and then running) at an event. After they have uploaded their results, they cannot upload them again.
In order to implement this state-based behavior, the state design pattern was applied. This pattern is applicable when an object needs to change its behavior at runtime according to its current state.
The implementation consists of the four classes SignedUp, LocationSet, CheckedIn and EventResultsUploaded, one for each of the defined states. All of them implement the IParticipantState interface, which declares the possible operations on states. The state classes define in the method bodies if a state transition is allowed or if an exception should be thrown.

```
class SignedUp : IParticipantState
{
    private static readonly SignedUp instance = new SignedUp();
    private SignedUp() { }

    public static SignedUp Instance { get => instance; }

    public void SetLocation(ParticipationContext ctx)
    {
        ctx.CurrentState = LocationSet.Instance;
    }
```

```
        public void CheckInAtEvent(ParticipationContext ctx) => throw
new InvalidOperationException();

        public void UploadEventResults(ParticipationContext ctx) =>
throw new InvalidOperationException();

        public override string ToString()
        {
            return "Signed up";
        }
    }
```

The class ParticipationContext instantiates an object of type IParticipantState, which stores the current state, namely an instance of one of the four state classes. Finally the class Participant declares a property of type ParticipationContext which can be used from other classes to check the current state of the participant.

3.2.2 Builder pattern - The class EventResults

Participants can upload their results manually after participating in an event. As seen in paragraph 2, total time is a required parameter whereas maximum speed and average heart rate are optional parameters.
In order to implement EventResults as an object composed of different parts, some of which are optional, the builder pattern was applied. This pattern is used to provide a mechanism for creating complex objects made of several parts, which is independent from the processes used to build the single parts.
The class EventResults is implemented as a dictionary of strings, where the key is a label (“Total time”, “Maximum speed”, or “Average heart rate” and the value is the user’s input turned into a string. The class EventResultsConcreteBuilder contains the methods BuildTotalTime(), BuildMaxspeed() and BuildAvgHeartRate(), which have the task of adding the user’s input to the corresponding objects in the dictionary of EventResults. The class EventResultsDirector combines and calls the methods of EventResultsConcreteBuilder according to which parameters the user wants to upload and which are null. For example, the method BuildSimpleResults() calls the method BuildTotalTime() of the concrete builder, whereas the method BuildDetailedResults() calls all three methods. Other methods of EventResultsDirector call the remaining combinations of methods.

```
   class EventResultsDirector
    {
        private IEventResultsBuilder builder;

        public IEventResultsBuilder Builder
        {
            set { builder = value; }
        }

        public void BuildSimpleResults(TimeSpan totalTime)
        {
            builder.BuildTotalTime(totalTime);
        }

        public void BuildResultsWithMaxSpeed(TimeSpan totalTime,
double maxSpeed)
        {
            builder.BuildTotalTime(totalTime);
            builder.BuildMaxSpeed(maxSpeed);
        }

        public void BuildResultsWithHeartRate(TimeSpan totalTime, int
avgHeartRate)
        {
            builder.BuildTotalTime(totalTime);
            builder.BuildAvgHeartRate(avgHeartRate);
        }

        public void BuildDetailedResults(TimeSpan totalTime, double
maxSpeed, int avgHeartRate)
        {
            builder.BuildTotalTime(totalTime);
            builder.BuildMaxSpeed(maxSpeed);
            builder.BuildAvgHeartRate(avgHeartRate);
        }
    }
```

Finally the method UploadEventResults() of the class Participant checks which values the user has uploaded and which were left empty and calls the relevant methods of the director.

3.2.3 Composite pattern - The classes User and UserGroup

As a future development the application will need to allow the creation and usage of multi-level groups, as illustrated in paragraph 1. This entails creating a tree structure, where users are the leaves and user groups are the composite elements, which in turn can contain other groups or users.
In order to implement this class hierarchy, the composite pattern was applied. This pattern allows the creation of both simple and composite objects which can be viewed by the client as the same type.

For this purpose the interface IUser was created, which exposes the methods AddMember(), RemoveMember() and SignUpForEvent(). The class UserGroup contains a list of objects IUser representing the members of the group. The methods AddMember() and RemoveMember() have the task to add and remove elements from this list.

```
    public interface IUser
    {
        void SignUpForEvent(Event selectedEvent);

        void AddMember(IUser user);

        void RemoveMember(IUser user);
    }
```

The feature of composite user groups was not represented graphically in the UI. Only groups with one level, namely user groups containing single users, can be created and managed.

3.2.4 Singleton pattern - The class Database

As mentioned above, the project contains a class called Database providing an example set of data. As other classes need to read and write this set of data, it is necessary that only one instance of Database exists and that it exposes a single point of access. In order to ensure this the singleton pattern was applied.
This way the classes requiring access to the Database class simply need to call the class Instance() method.

```
    class Database
    {
        private static Database instance = null;
…

        public static Database Instance()
        {
            if (instance == null)
                instance = new Database();
            return instance;
        }
```

4. Usage documentation

The application was developed with Visual Studio Community 2019 Version 16.5 and the .NET framework 4.7.2. It can be opened, compiled and run directly from the IDE.

A test user with some sample data has been created with the following log in information: Username: “Tom”, Password: “password”.

5. Use cases

This section describes the most significant use cases. In the case of this application a use case diagram would not guarantee a high enough level of detail. For this reason use cases are listed in text-based form.

Use Case: Log in
ID: UC1
Actor: User
Preconditions: The user has opened the application.
Basic course of events:

The user types username and password and clicks “Ok”.

Postconditions: The user is logged in. The main application window opens and the text “Hello [username]” appears below the tab bar. The user can now use all the functions of the application.
Alternative paths:
If either username or password are incorrect, an error message is shown.

Use Case: Create an in-person-event
ID: UC2
Actor: User
Preconditions: The user has logged in and has selected the tab “Find events”.
Basic course of events:

1. The user clicks “Create new event”.
2. A new window opens. The user selects the tab “In person event”.
3. The user specifies a date, a time, an average speed and a title for the event.
4. The user selects a running location from the list or clicks “Create new location” and follows the subsequent steps.
5. The user clicks “Ok”

Postconditions:
The new event is displayed in the “In-person-events” list. Users can sign up for the event.
Alternative paths:
If the user fails to specify the required information in 2. or 3. relevant error messages are shown.

Use Case: Create a virtual event
ID: UC3

Actor: User
Preconditions: The user has logged in and has selected the tab “Find events”.
Basic course of events:

1. The user clicks “Create new event”.
2. A new window opens. The user selects the tab “virtual event”.
3. The user specifies a date, a time, an average speed and a title for the event.
4. The user specifies a route length.
5. The user clicks “Ok”

Postconditions:
The new event is displayed in the “Virtual events” list. Users can sign up for the event.
Alternative paths:
If the user fails to specify the required information in 2., 3. or 4. relevant error messages are shown.

Use Case: Create a user group
ID: UC4
Actor: User
Preconditions: The user has logged in and has selected the tab “Find groups”.
Basic course of events:

1. The user clicks “Create new group”.
2. A new window opens.
3. The user specifies a group name and clicks “Create group”.

Postconditions:
The group is displayed in the “User groups” list. The creating user is the group administrator and can sign up the whole group to events. Other users can join the group.
Alternative paths:
If the user fails to specify a group name, an error message is displayed.

Use Case: Sign up for an event
ID: UC5
Actor: User
Preconditions: The user has logged in and has selected the tab “Find events”.
Basic course of events:

1. The user selects an event either from the “In-person-events” list or from the “Virtual events” list.
2. The user clicks “Sign up”.

Postconditions:

1. The event appears in the upcoming events list of the tab “User area”.
2. The event state is marked as “Location set” for in-person-events and as “Signed up” for virtual events.

3. The user can check in at the event directly if it is an in-person event. The user can choose a location if it is a virtual event.

Alternative paths:

If the user has already signed up to the event, an error message is shown.

Use Case: Sign up a group for an event
ID: UC6
Actor: User
Preconditions: The user has logged in and has selected the tab “Find events”.
Basic course of events:

1. The user selects an event either from the “In-person-events” list or from the “Virtual events” list.
2. The user clicks “Sign up group”.
3. The “Sign up group” window opens, displaying those groups for which the user is the administrator.
4. The user clicks “Sign up group”.

Postconditions:

1. The event appears in the “Upcoming events” list of the tab “User area”.
2. The event state is marked as “Location set” for in-person-events and as “Signed up” for virtual events.
3. The group members can check in at the event directly if it is an in-person event. Or can choose a location if it is a virtual event.

Alternative paths: None

Use Case: Select a location (only for virtual events)
ID: UC7
Actor: User
Preconditions:

1. The user has logged in and has selected the tab “User area”.
2. The user has already signed for the event.

Basic course of events:

1. The user selects an event from the “Virtual events” list and clicks “Choose location”.
2. The “Browse locations” window opens.
3. The user selects a location from the list and clicks “Ok”. Alternatively the user clicks “Create new location” and follows the subsequent steps.

Postconditions:

1. The event state is marked as “Location set” in the “Upcoming events” list.
2. The user can check in at the event.

Alternative paths:

If the user fails to select an event, an error message is shown.

Use Case: Create a new location
ID: UC8
Actor: User
Preconditions:

1. The user has logged in and has selected the tab “User area”.
2. The user is either in the process of creating a new in-person-event or of choosing a location for a virtual event.

Basic course of events:

1. The user clicks “Create new location”.
2. The “Create new location” window opens.
3. The user specifies a route name, latitude and longitude for the starting point and route length for the running location.
4. The user Clicks “Ok”.

Postconditions: The new running location appears in the “Locations” list.
Alternative paths:
If the user fails to specify the information requested in 3. or these are in the wrong format, specific error messages are shown.

Use Case: Check in at an event
ID: UC9
Actor: User
Preconditions:

1. The user has logged in and has selected the tab “User area”.
2. The user has already signed to the event and a location has been set.

Basic course of events:

1. The user selects an event from the “Upcoming events” list and clicks “Check in at an event”.
2. The event state is marked as “Checked in” in the list.

Postconditions: The user can upload event results as soon as the event has taken place.
Alternative paths:
If the user fails to select an event, an error message is shown.

Use Case: Upload event results
ID: UC10
Actor: User
Preconditions:

1. The user has logged in and has selected the tab “User area”.
2. The event has already taken place and the state is marked as “Checked in”.

Basic course of events:

1. The user clicks “Upload event results”.

2. The “Upload results” window opens.
3. The user specifies the total time (required), the maximum speed (optional) and the average heart rate.
4. The user clicks “Upload results”.

Postconditions:

1. The results can be viewed by clicking “See event results” from the “User area” tab.

Alternative paths:

If the user fails to specify the information requested in 3. or these are in the wrong format, specific error messages are shown.