University of Applied Sciences

Traffic Control System
Software Design Document

Agnes Wadee
Coen Stange
Wen Li
Yongshi Liang

Background and Context

This software design document provides details and decisions made for building the "Traffic control system" application. For this a graphical representation UML is used including class diagrams and sequence diagrams.

Definitions and abreviations

UML	Unified Modeling Language
MVVM	Model view viewmodel

Contents

1 System overview

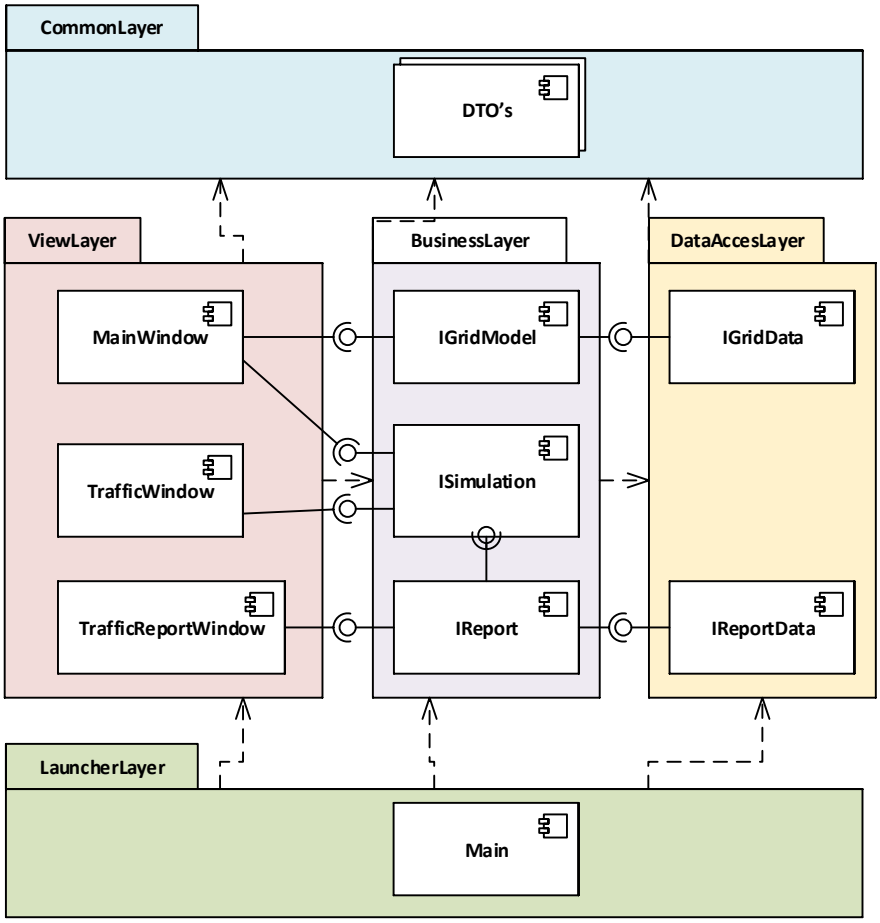


Figure 1: System overview

1.1 Abstraction layers

We design the software according to SOLID principles, which are common rules used for the design of the software. The classes should have only one responsibility, hence our representation logic and business logic are separated. We also use an interface driven design combined with the dependency inversion principle, hence we handle dependencies using dependency injection. By following these rules the system is more testable and abstracted.

1.1.1 Common layer

The `CommonLayer` can be accessed by any layer, and the layer only contains DTO's. Important about the classes of the `CommonLayer` is that they contain no logic at all, hence they only have data. The purpose for this is for easy serialization and a easy way to pass the data through all the layers.

1.2 View layer

Inside of the `ViewLayer` is al the representation logic, this layer handles showing an user interface. For the representation the `MVVM` model is used, this pattern keeps the design and the code of the view seperate.

1.3 Business Layer

This layer contains the business logic of the system, it makes sure that all the business rules are followed. Hence this is the brain of the application which is allowed to make changes to the DTO's from the `CommonLayer`.

1.4 Data Layer

The data layer contains the logic for serializing the objects from the `Common Layer` to another storage. For this application the objects will be serialized to a file.

1.5 Other layers

The application has tests and it also require a launcher layer. Because the application can not be launched from the `ViewLayer` since it has no reference to the `DataAccesLayer`. So in the launcher layer all the dependencies will be handled.

2 Class diagrams

2.1 Common layer

The `CommonLayer` is shown in figure 2. The design of the classes are in mind that it can be shared in between layers and it can be serialized (written to file). Hence the class diagram follow the constraints that it can't use any interfaces or that there is a circular reference (it has to look like a tree). Notice that the `PedestrianEntity` and the `CarEntity` are in an `ObservableCollection` and have observable attributes. When the simulation is running the view will be be notified when attributes are changed.

2.1.1 Entities

The classes of the common layer all end with *Entity. An entity in this program is a serializable object which will be used between all layers. The objects contain no operations, creating such objects can be best done using the object initializer (instead of the constructor). For example take the class point with values X and Y, can be created by `new Point { X = 100, Y = 200 };`

2.2 View layer

In figure 5 the class diagram of the view layer is shown. Inside of the view we do not use the usual `Graphics` used in Forms application to draw on the screen but instead a `Canvas` to draw on the screen. A lot of the drawing is already handled and we have especially bind the right values.

2.2.1 UserControl

An UserControl is a Control shown on the screen hence it also has a View. The UserControl has logic for displaying certain elements on the screen. The UserControl observes the Entities. For example when the value of `CarEntity` has changed then the `CarUserControl` related to that also has to move itself.

2.3 ViewModel

A `ViewModel` is an object which the program can bind to the View of the Window, and then these values are shown on the screen. The values can also be altered for example when put in a TextBox. The responsibility of serializing the values is with WPF, hence we don't need to check if the values we receive are indeed for example an integer. Hence `ViewModel` can also be observed using the `PropertyChanged` event, then the according entities can also be updated.

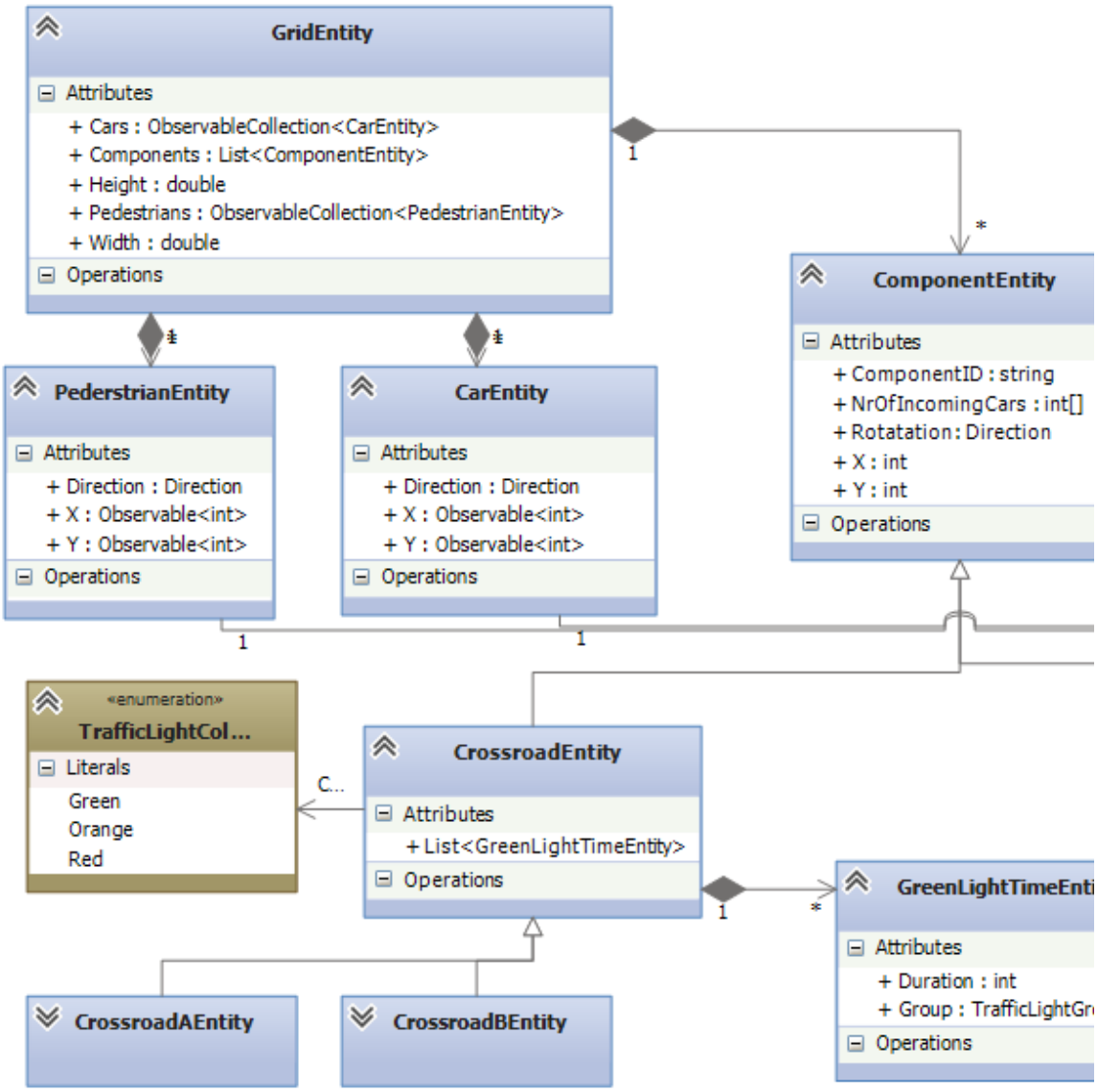
2.4 Business Layer

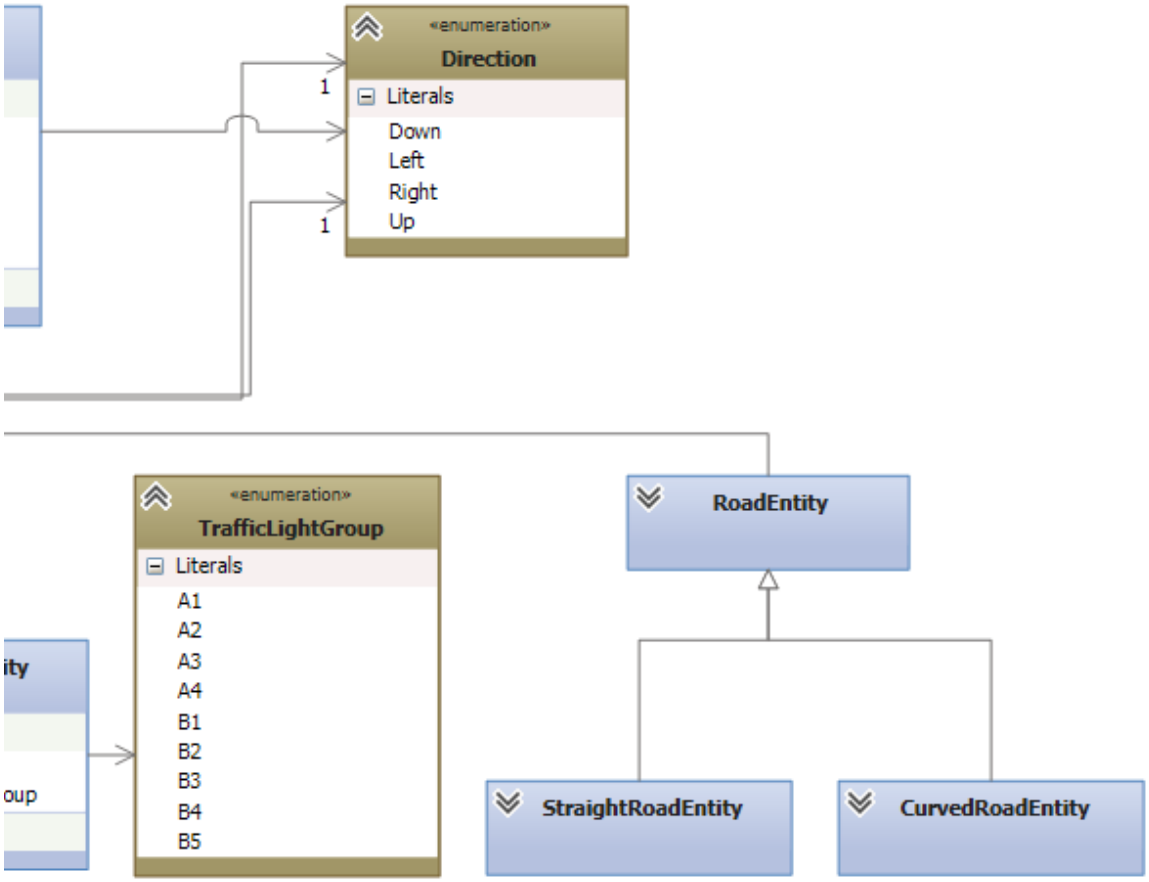
In figure 4 the class diagram of the business layer is shown. We choose to make operations for adding and deleting components instead of using an observable collection since by using operations is more simple to maintain business rules.

2.5 Data Layer

In figure 3 the class diagram of the data layer is shown. The data layer does not show concrete implementation since the classes don't require to have a state.

Figure 2: Class diagram common layer




«enumeration»
Direction
Literals
Down
Left
Right
Up
1
1
«enumeration»
TrafficLightGroup
Literals
A1
A2
A3
A4
B1
B2
B3
B4
B5
RoadEntity
StraightRoadEntity
CurvedRoadEntity

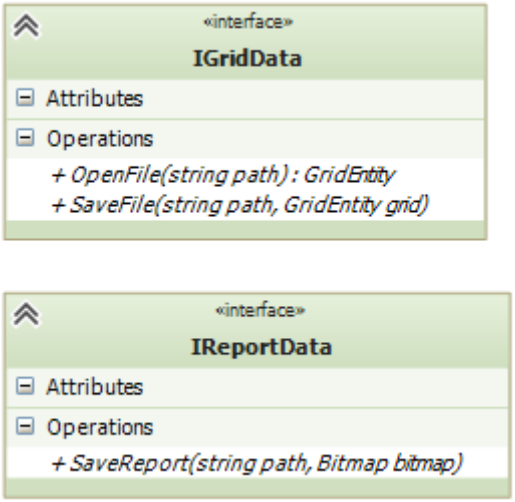


Figure 3: Class diagram data access layer

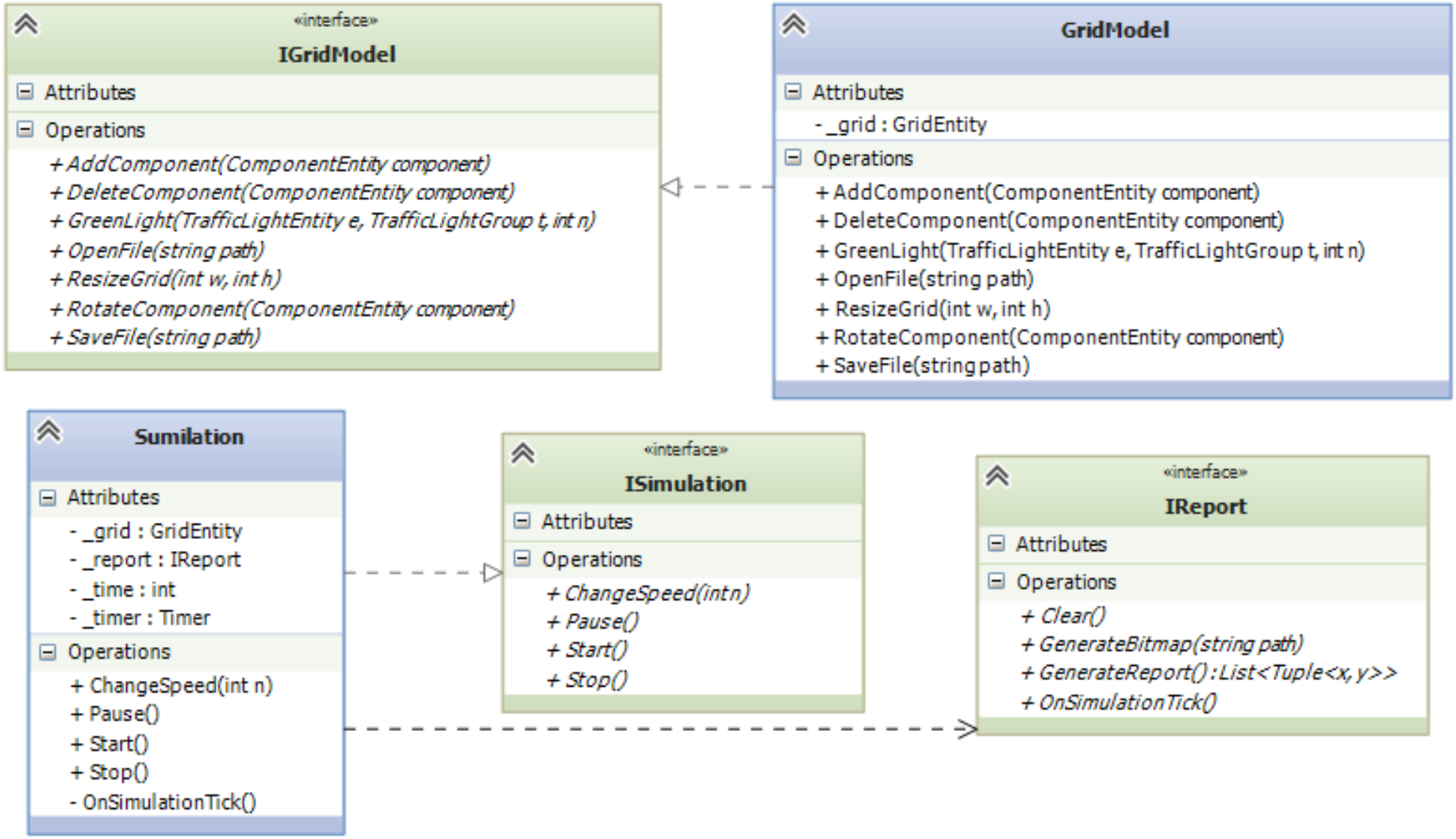


Figure 4: Class diagram business layer

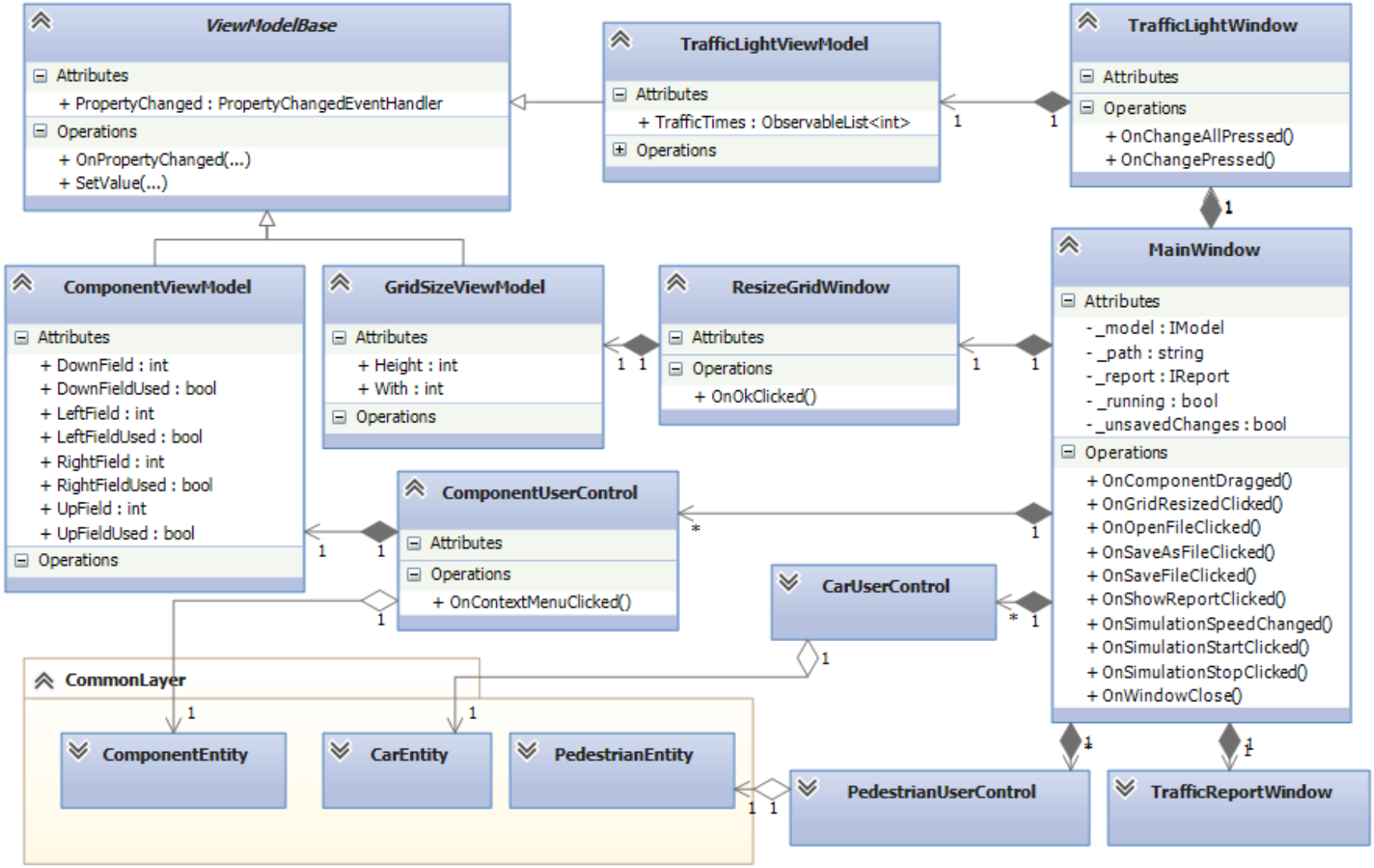


Figure 5: Class diagram view layer

3 Sequence diagrams

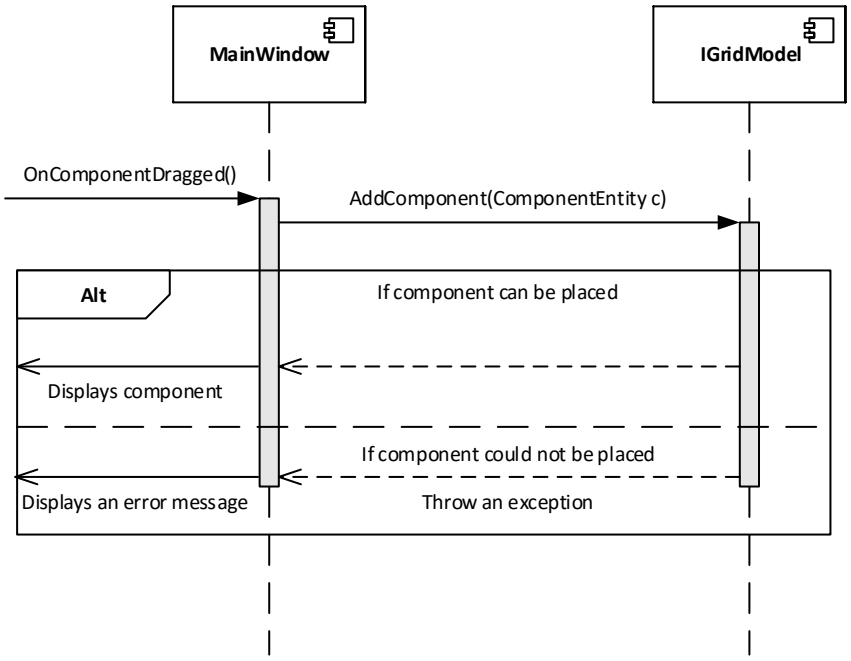


Figure 6: Positioning crossroad.

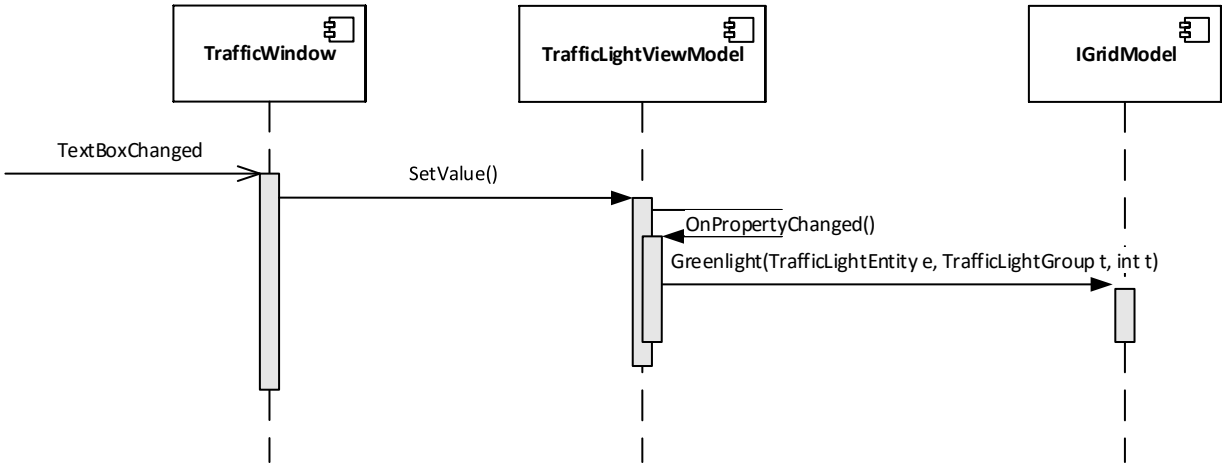


Figure 7: Configuring traffic light times.

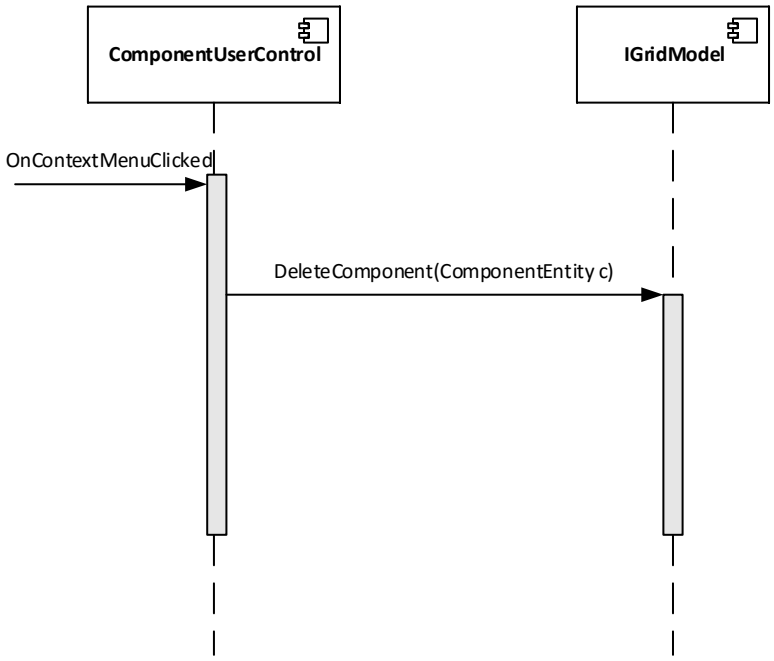


Figure 8: Deleting components.

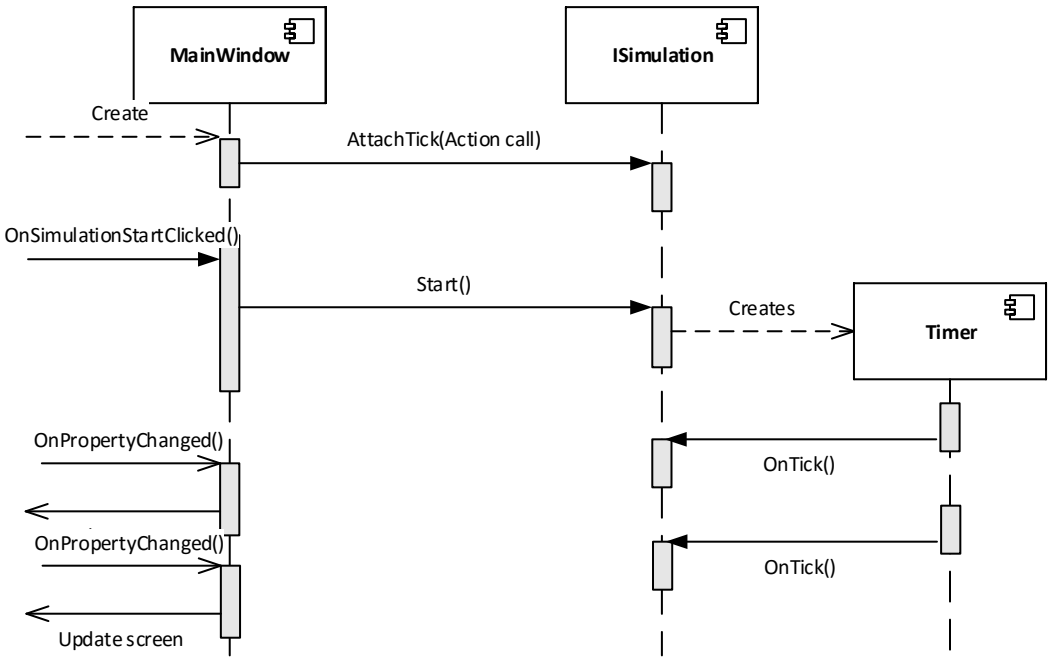


Figure 9: Running simulation.

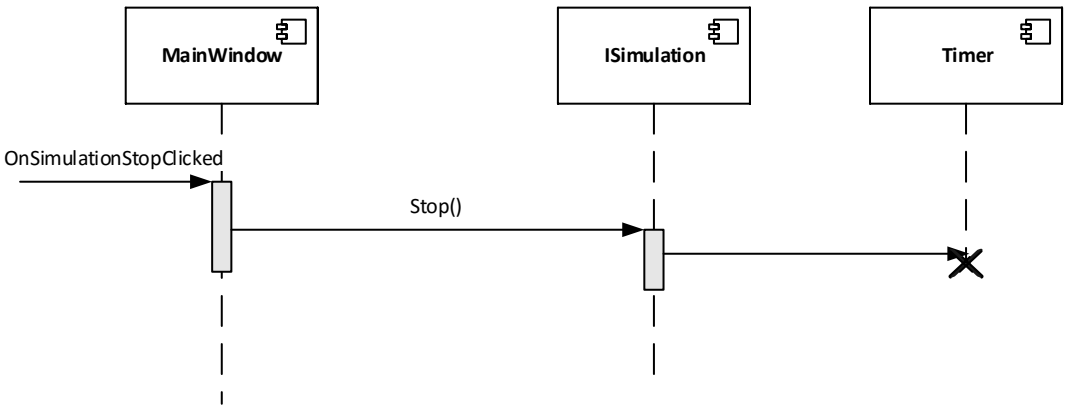


Figure 10: Stopping simulation.

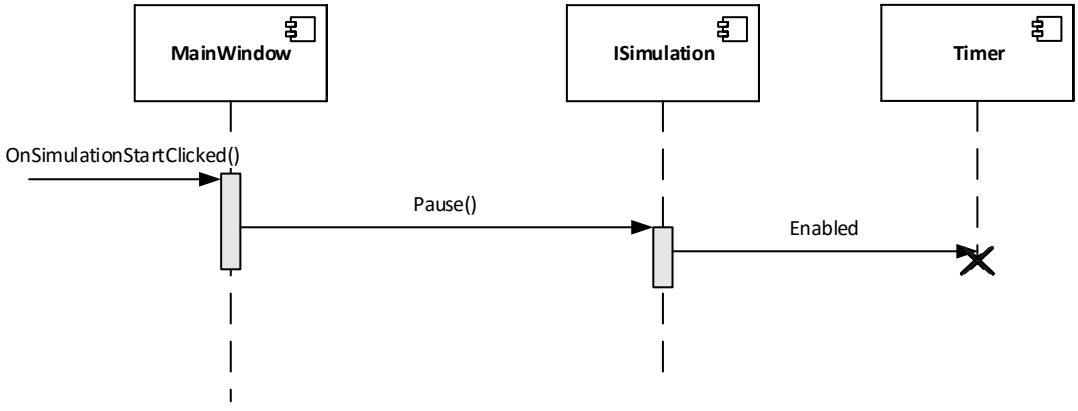


Figure 11: Pausing simulation.

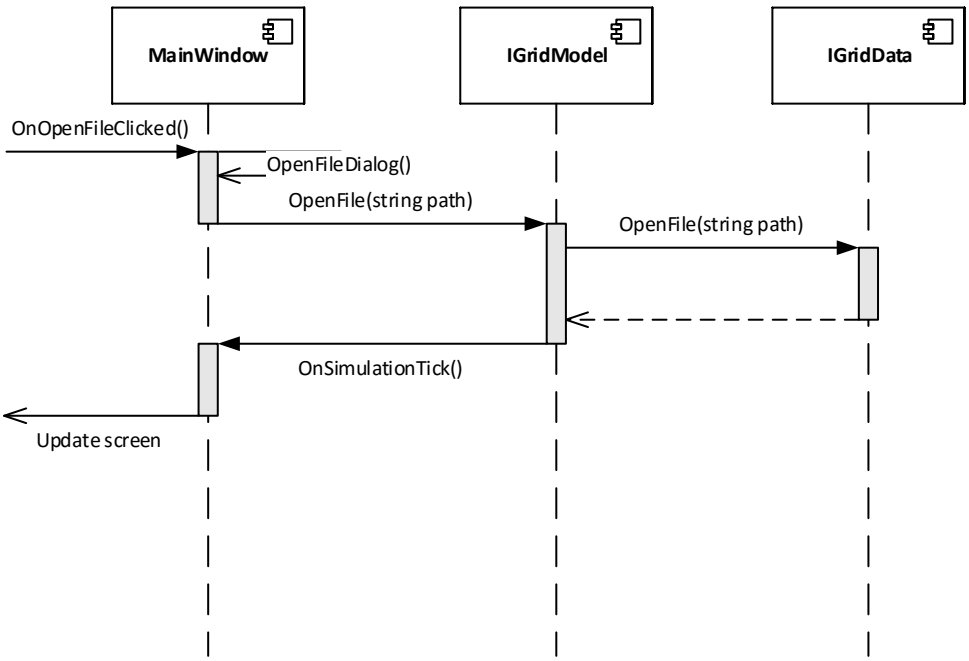


Figure 12: Opening a file.

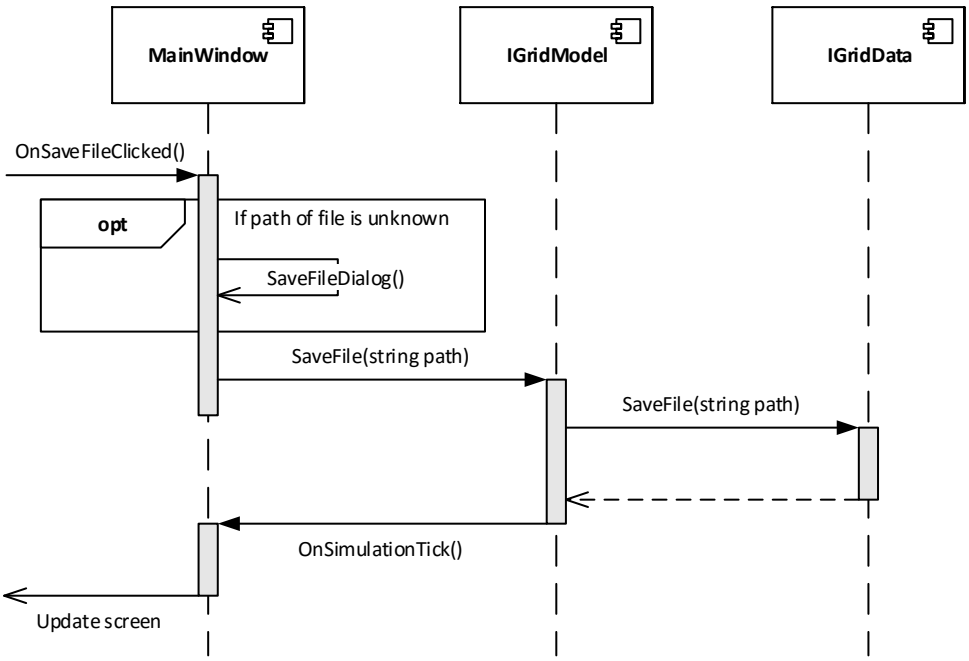


Figure 13: Saving a file.

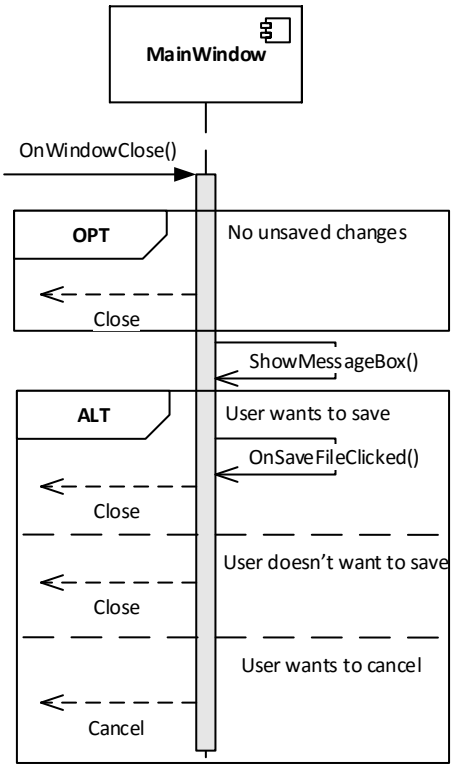


Figure 14: Close window.