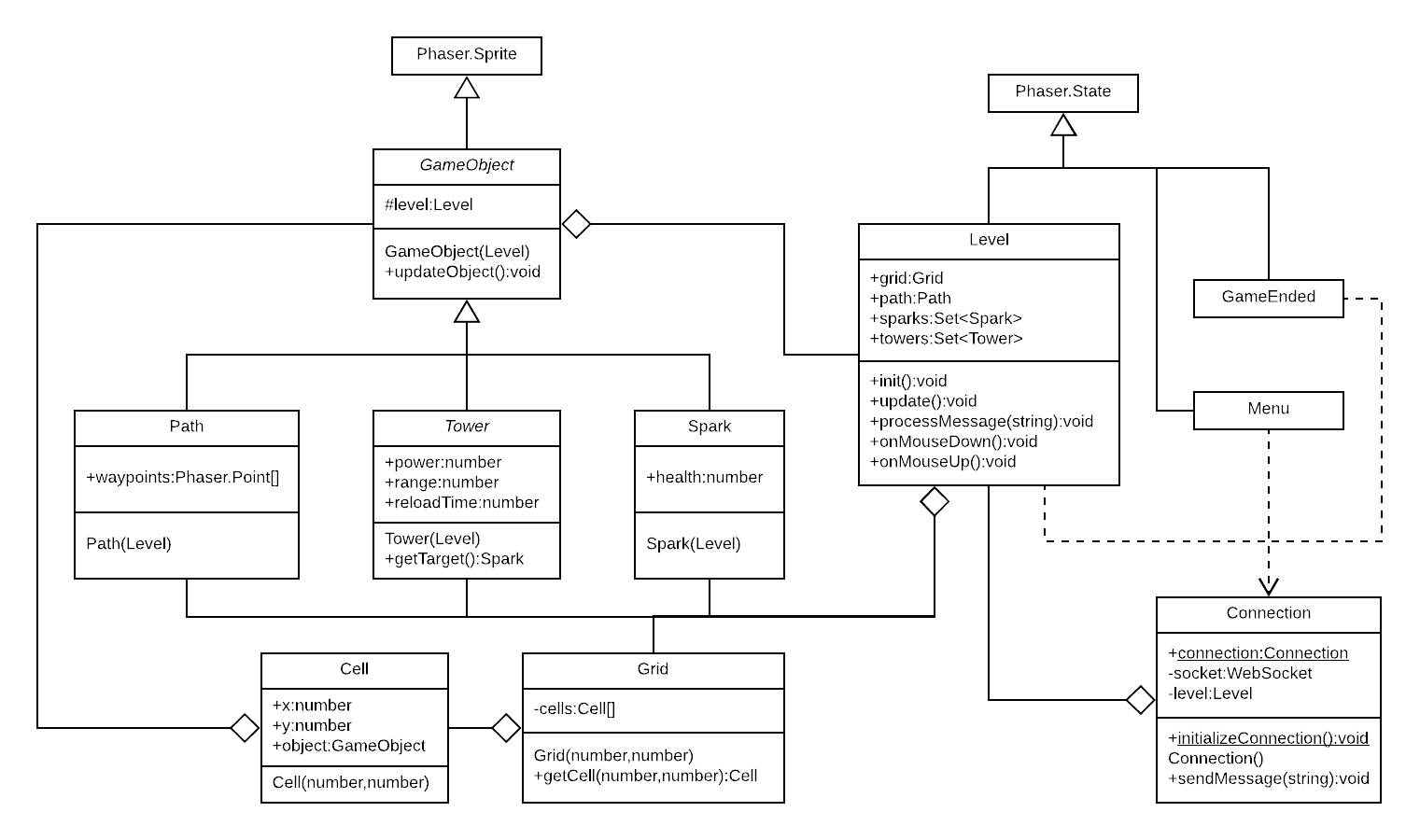
Front end main architecture

The front end is what the users will run in their browser. So this is the code of the game itself, including all sprites and animations. This will be done using a framework. The decision has been made to use Phaser as a framework and to use TypeScript to actually code the game. The game is grid-based, so the towers and the path of the sparks need to be aligned with the grid.

# Alternatives

## The main idea

The first and main idea for an architecture is the following. This architecture is more of a guideline than an actual solution.



The framework Phaser works with states. Each *Phaser.State* instance represents a screen, such as the menu screen and the game over screen. The *Level* state represents the main game screen. So the *Level* state contains all the *GameObjects*, such as the *Path*, *Sparks* and *Towers*.

When a player is in the *Menu* state, a connection should be made. This way, the connection can be verified before the player starts playing. During the gameplay, the connection will be used to verify the formal specifications of the player. Since states can not communicate with each other, this is done by making the *Connection* object a singleton with a public static instance. The *Menu* state initializes the *Connection* object. The *Level* state gives the *Connection* object a pointer to the *Level* state object, such that the *Connection* object can call its *processMessage* method whenever a message is received from the *GameServer*. The *Connection* object also has a *sendMessage* method with which a message can be sent to the *GameServer*.

As stated earlier, the *Level* state represents the main game screen, and hence contains all the *GameObjects*. The *GameObject* class abstracts objects in the game with a sprite. Behind the scenes, the *GameObject* class extends the built-in *Phaser.Sprite* class. Due to the large amount of dependencies between *GameObjects*, the simplest way to let them communicate with each other is giving every *GameObject* a pointer to the *Level*. Some examples of *GameObject* are *Path*, *Tower* and *Spark*.

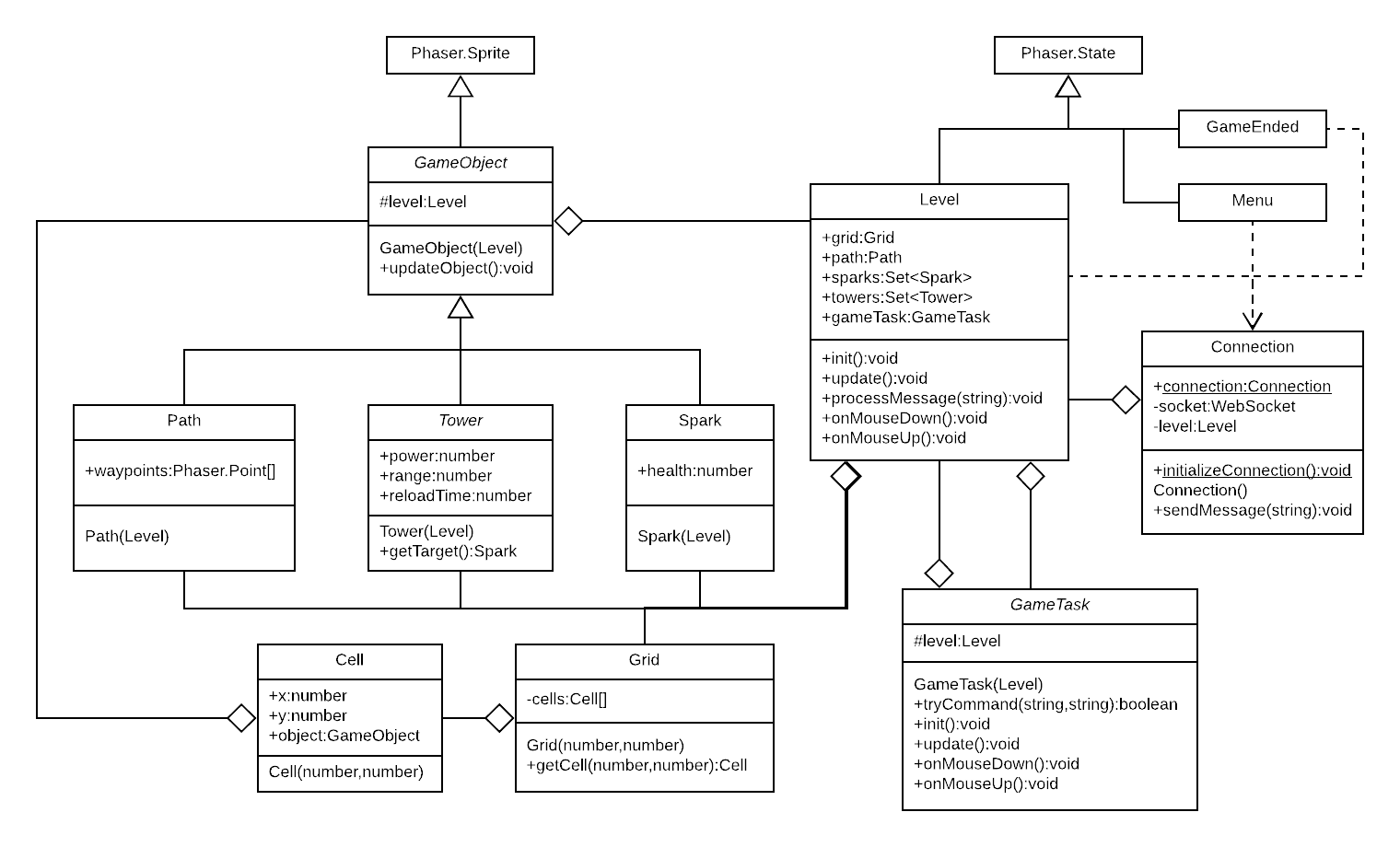
A *Path* is just a collection of waypoints, which are stored as *Phaser.Point* objects. A *Tower* is abstract, so every *Tower* instance at least has a certain power, range and reload time. The *Sparks* are the entities walking along the path, which at least need to have a certain health. These *GameObjects* do not need to define a position, as this is already implied by the *Phaser.Sprite* inheritance.

As earlier mentioned, the game will be grid-based, so the *Level* needs to keep track of the grid. This is, for example, to prevent towers to be placed on the path, or on each other. The *Grid* object just keeps track of the *Cell* objects. Each *Cell* keeps track of the *GameObject* that is currently occupying the *Cell*. If there is no such *GameObject*, this will be null. This is, for example, to be able to quickly verify whether a tower can be placed on a certain cell.

## An improvement

There is one big problem. The *Level* class has too many responsibilities. A lot of logic, for example towers shooting, can be done inside specific *GameObject* instances. But more general logic, for example towers being bought, will be placed inside the *Level* class. So this architecture unfortunately does not scale very well with the amount of different commands, mouse clicks and updates that will be defined.

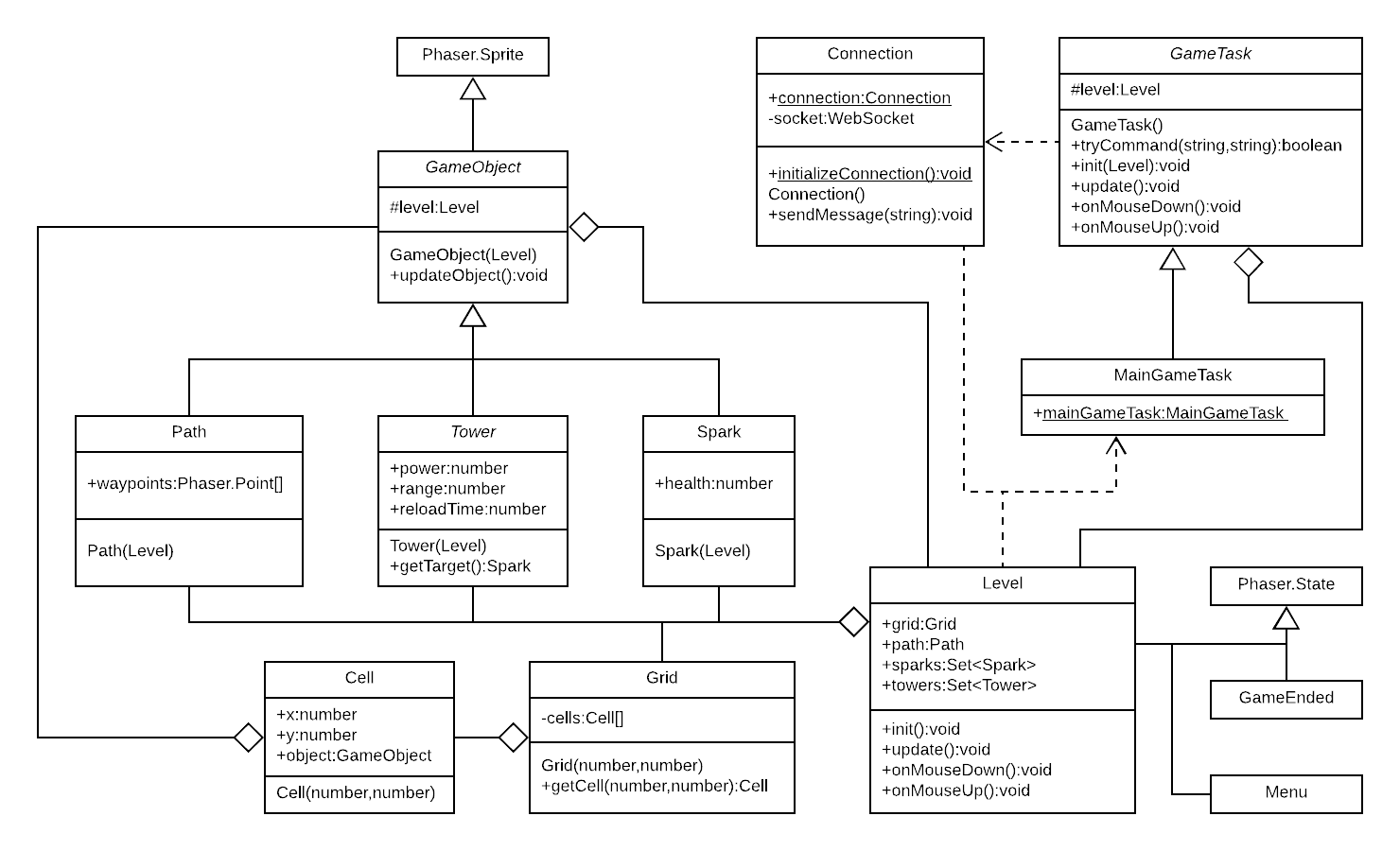
One solution is the following architecture.



The only difference in the diagram is that a *GameTask* class has been added, and the *Level* class has a *GameTask* as one of its fields. The idea is the same as in the game server architecture, except *GameTasks* here handle multiple events and not only commands. These events are an update method, which is called every tick, an *onMouseDown* method, which is called whenever the mouse button of the user is clicked, and an *onMouseUp* method, which is called whenever the mouse button of the user is released. This way every *GameTask* can determine what needs to happen in any of these scenarios. There is also an *init* method, which is called when the game is being initialized. Just as in the *GameServer*, these *GameTasks* may take the form of a tree, where there is a *MainGameTask* distributing the events over other *GameTasks*.

## An alternative

An alternative solution is the following.



The main difference from the previous diagram is that *Level* does not have an instance of *GameTask* anymore. Instead of the *Level* class having a *MainGameTask* instance, *MainGameTask* is now a public singleton. Unlike the *GameServer*, the *Client* is only responsible for one client, which is the reason this is possible in the first place.

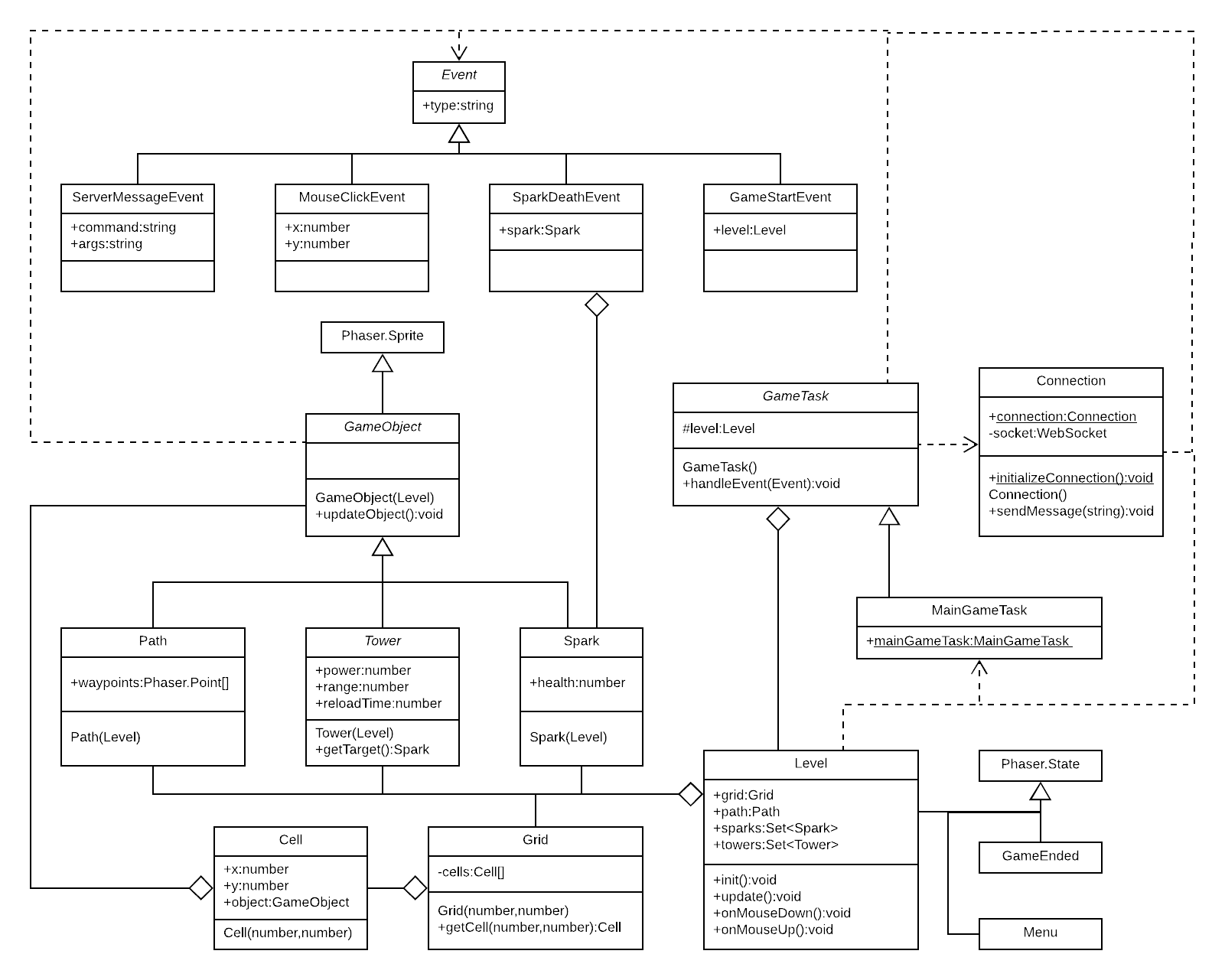
One benefit this has over the previous architecture is that the *MainGameTask* can already be constructed when the player is in the menu. The *MainGameTask* has to be public and static for this, because *Menu* and *Level* can not communicate directly. In the constructor of a *GameTask*, the *Level* object can not be a parameter anymore, so the *Level* object is now a parameter of the init method, which should be called when the player starts the game and the game needs to be initialized.

The reason this might be useful is because there will already be communication with the *GameServer* when the player is in the menu. This way, this can also be done with a *GameTask*. For example, the path of the level the player is going to play can now be sent while the player is in the menu, such that a *PathGameTask* can store the path. When the player starts the level, the path is already known and can immediately be displayed.

The logic handling messages from the *GameServer* while the player is still in the menu does not need to be in the *Menu* class, as the *Communication* class can access the *MainGameTask* itself and call the *processMessage* method. This also means that the *Level* class does not need a *processMessage* method anymore.

## Another alternative

Another alternative solution is the following.



The main difference from the previous diagram is that the *GameTask* class only has one method. The methods that were in *GameTask* got abstracted to the *handleEvent* method. The argument of this method is an *Event* object, which is very general. An *Event* can be anything. Examples of *Event* instances are *ServerMessageEvent*, *MouseCickEvent*, *SparkDeathEvent* and *GameStartEvent*.

Inside an *Event* class, arguments for the *Event* can be stored. For example, *ServerMessageEvent* stores the command and arguments of the message received from the server, and *SparkDeathEvent* stores the *Spark* that has died. Every event has a type string, which should be a unique identifier of the type of *Event*. This way, *GameTask* instances can simply use a switch statement to consider all *Event* types where the *GameTask* needs to do something.

The reason this architecture is worth considering is because new *Event* types can easily be defined and used. When there is a certain *Event* where a lot of things need to happen, instead of calling a lot of methods at the place of the *Event*, the *MainGameTask* can be called with the *Event* as parameter. The *Level* class, the *Connection* class, any *GameObject* class and any *GameTask* class need to be able to call *MainGameTask* with an *Event*. As *GameObjects* can call *MainGameTask*, they do not need to have a pointer to the *Level* object anymore.

# The decision

In general, the way the solutions are listed here is such that every next architecture solves a problem the previous one had, but it becomes more complicated in the process. Hence, the last architecture is potentially the most expandable, but it is the hardest one to understand. New features are most easily added to the last architecture. Sometimes, adding an event handler is all that needs to be done.

However, the last architecture is also the most complicated architecture that is listed here. In such a big project where multiple people write code, it is important that everybody is on the same page when they write code. Otherwise, the references and dependencies can become quite a mess and hard to understand. This can cause obscure bugs to appear. Thus, as the last architecture is the most complicated, even though it has the least technical problems, it might not be the best decision.

Very much worth mentioning is that early on in development, the architecture under the tab ‘An improvement’ was implemented. This means that implementing one of the later architectures is a major refactor. But this architecture has a major problem with communication between the *Menu*, *Level* and *Connection* classes which is fixed in the next architecture. The problem that the last architecture solves is not as important. The code becomes cleaner if the problem is solved, but it is not a necessary change.

Considering all these arguments, the decision has been made to implement the second to last architecture.