A document **docs/plugins.pdf** or **docs/plugins.md**, with a description on how levels as plugins were implemented. This should include a discussion of design patterns and frameworks that have been used.

Design Patterns and Frameworks

We used a similar design to the framework used for the eclipse plugins. (https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html#3.)

There are two parts to the program:

1. **The Core** of the program contains interfaces and abstract classes that determine the general execution of the program and how the different parts work together. It provides extension points for the plugins to hook into.
2. **The plugins** hook into the extension points provided by the program by implementing and extending the interfaces and abstract classes in the core. They define the specifics of what the program does.

For this to work the core needs to know two things:

1. What plugins exist.
2. How to get the plugins

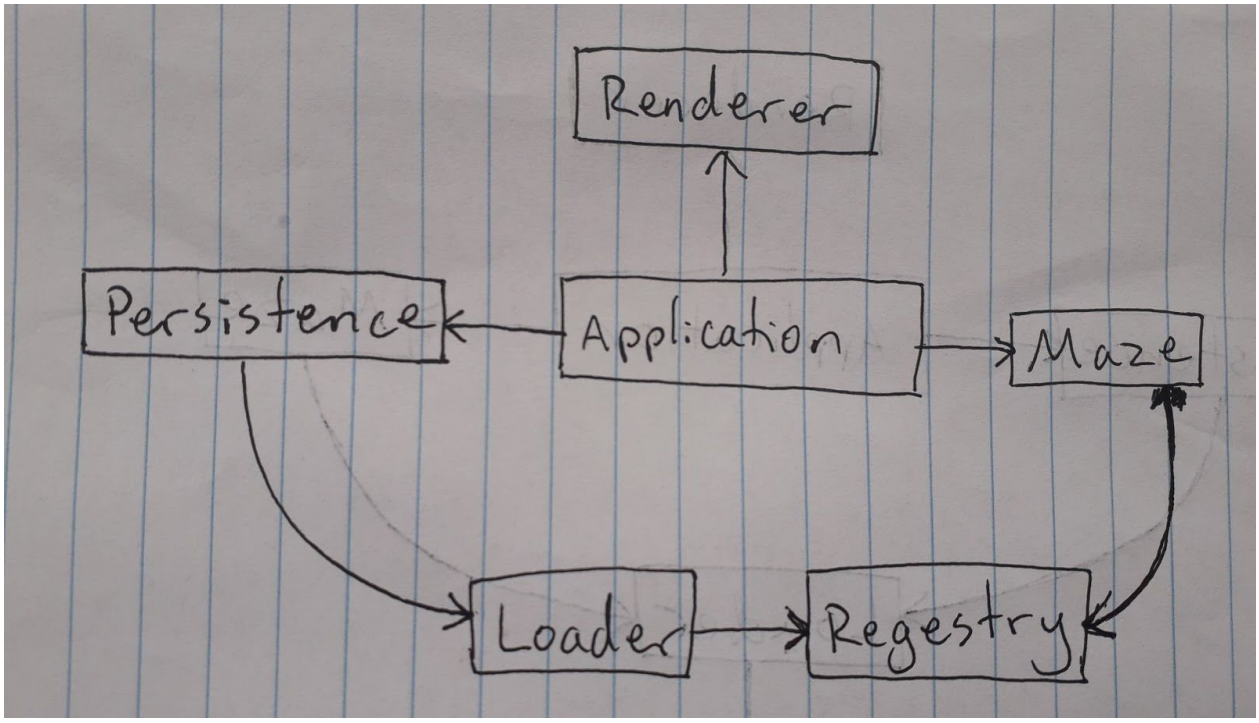
Eclipse does this using a manifest XML file that describes each plugin. The plugins can then be loaded (when they are required) into a registry where they are cached so they don't need to be reloaded every time they are used.

We also used the **Singleton design pattern**:

This pattern means that their can only ever be one instance of the class that implements this pattern. The constructor is made private and the class contains a final field that contains the sentence. It has a static getInstance() method that returns this instance and uses lazy initialisation.

This pattern is useful for classes like the registry where we only want one instance (more than that be redundant). It also means we don't need to pass instances of the registry around the program which would increase complexity and would make classes that don't need the registry dependent on it.

Implementation



The core consists of the 4 main modules from the Project handout (persistence, Application, Maze, Renderer).
The Maze module provides the extension points. These are the Tile abstract class, the Item interface, and the Actor interface (and Enemy abstract class which implements the Actor interface). The core also provides some basic implementations of these interfaces that all levels (plugins) will need. The Wall tile that can not be walked on, the Free tile that can be walked on and can contain an item, the door tiles and key items, and the exit item. It also provides the Chap actor and a basic enemy.

The plugins are level-k folders in the levels folder of the project (We did not have enough time to make them zip files). This folder contains a classes.jar file that contains all the classes of the maze elements (actors/items/tiles) needed for the level, a JSON file that contains a description of the level, and the images of the maze elements (tiles/items/actors). The maze elements hook into (implement/extend) the extension points provided by the core.
In order to be able to ge instances of the classes they must either have a zero argument constructor or a public static build() method that takes either a TileInfo, ItemInfo, or ActorInfo, depending on the type of element the class is.

The Loader (MazeElementLoader) Unlike eclipse we do not use a manifest file Instead it makes assumptions for what it needs to know about the level plugin using the json file that describes the game state for the level and the fact that all the level plugins follow the same structure (tile

classes in a tiles package, item classes in a items package and actor classes in an actor's package).
To load the classes provided by the plugins it uses a URLClassLoader which uses a url to find and load classes that are not on the class path of the program. When loading a level or save file first the url of the URLClassLoader is set to the path of the classes.jar file in the level-k plugin folder by the persistence. Then when loading the JSON file and creating the GameState that describes the level it loads the classes needed for the level as it gets to them in the JSON file.
When the class is loaded it registers the class with the MazeElementRegistry so that the core program knows that it exists and can et instances of it.

The Registry contains the Class objects of all the loaded classes (and the basic classes provided by the program core). It provides a way to get implementations of these classes using the Class objects using reflection. It first attempts to create an instance using the zero argument constructor, then if that does not work (there was no zero argument constructor) it attempts to get an instance using the build() method that takes either ItemInfo, TileInfo, or ActorInfo. If neither of these work it throws an assertion error as it was not able to create an instance (or if assertions are disabled it returns null).
It is also used by ItemInfo/TileInfo/ActorInfo classes (plain old java objects) to make sure the maze elements (tiles/actors/items) they describe are valid and that they contain all the information needed to make the element (tile/actor/item). It does this by getting the list of loaded classes from the registry and using reflection call the getRequiredFields() method (if it exists) to check the fields and name of the element.