Architecture Design

Team Free Pizza

Name	*NetID*	*Studentnumber*
Tim Yue	tyue	4371925
Bart van Oort	bvanoort	4343255
Steven Meijer	stevenmeijer	4368061
Thomas Kolenbrander	tjkolenbrander	4353137
Etta Tabe Takang Kajikaw	eettatabe	4373758

TABLE OF CONTENTS

1. INTRODUCTION

In this document, we will be presenting a rough sketch of the application/software which we will be working on in the project multimedia systems. We will make use of high level components to explain how the architecture of our software or application will look like, therefore, making it easily understandable and visualizable to users. The explained high level components are then split into subsystems which are described in section 2.

1.1 Design goals

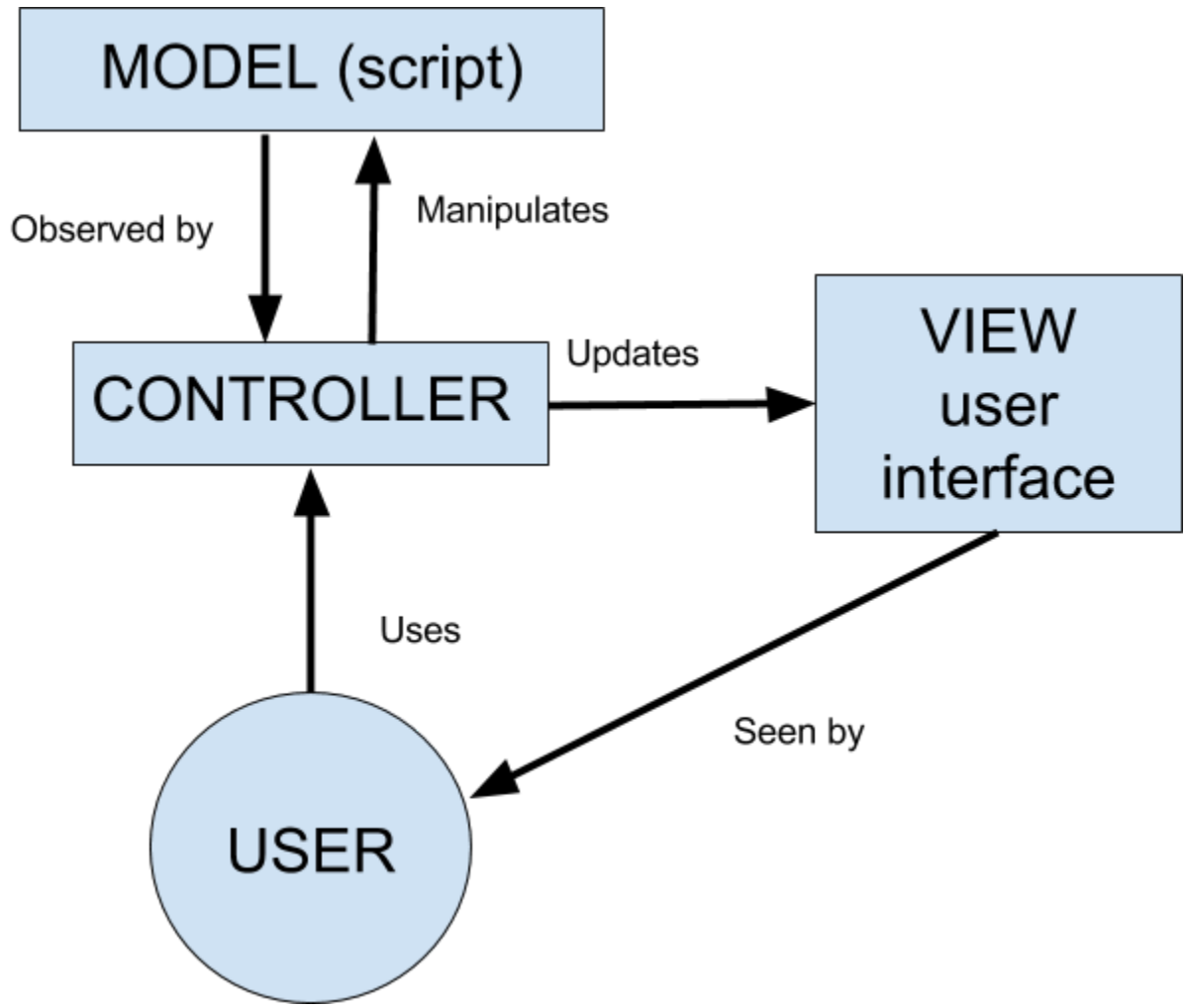
We want to maintain a proper code base, that has code of good quality, so it is not just easy to understand, but also easy to maintain. Good quality code can be achieved by using important design principles and patterns where needed, and by making use of code quality tools. For this project, we will use CheckStyle, PMD and FindBugs for static analysis, the JUnit test framework and EclEmma for code coverage, and Travis CI with Maven as the automated building tool. During the design of our product, we will focus on the following design goals, aside from the goals specified in the product vision.

- *Usability:*
  We want our application to be easy to use, so a new user can start using our product with minimal explanation. To achieve this, we need a simple and easy to understand user interface, taking into account what interfaces the users are already used to.
- *Extensibility:*
  We also want our application to be easily extendible. When other programmers decide they want to add features of their own to our application, they should be able to understand the code easily and be able to perform this addition easily. This can be achieved by using abstract classes and interfaces and keeping documentation on all classes, methods and interfaces.
- *Performance :*
  We also want our application to be performant in terms of responsiveness and stability under a certain workload. In this case, the program should not become slow, crash or unstable when multiple users try to make use of it or access it at the same time. This can be achieved by carrying out performance testing with multiple users, which can also serve to investigate, measure and validate other quality attributes of the program such as scalability and reliability, resulting in a better working software.
- *Robustness:*
  In general, building an application that is totally robust is a very difficult task, because of the large amount of inputs or input combinations which may exist[1]. We want to have an application that is able to cope with errors during execution, and can deal with erroneous inputs in an appropriate manner such as display of warning messages. This will be achieved by carrying out fault injection where the coverage of a test can be improved by introducing faults to test code paths[2]. In this way, our system may not be totally robust, but responsive to user inputs.

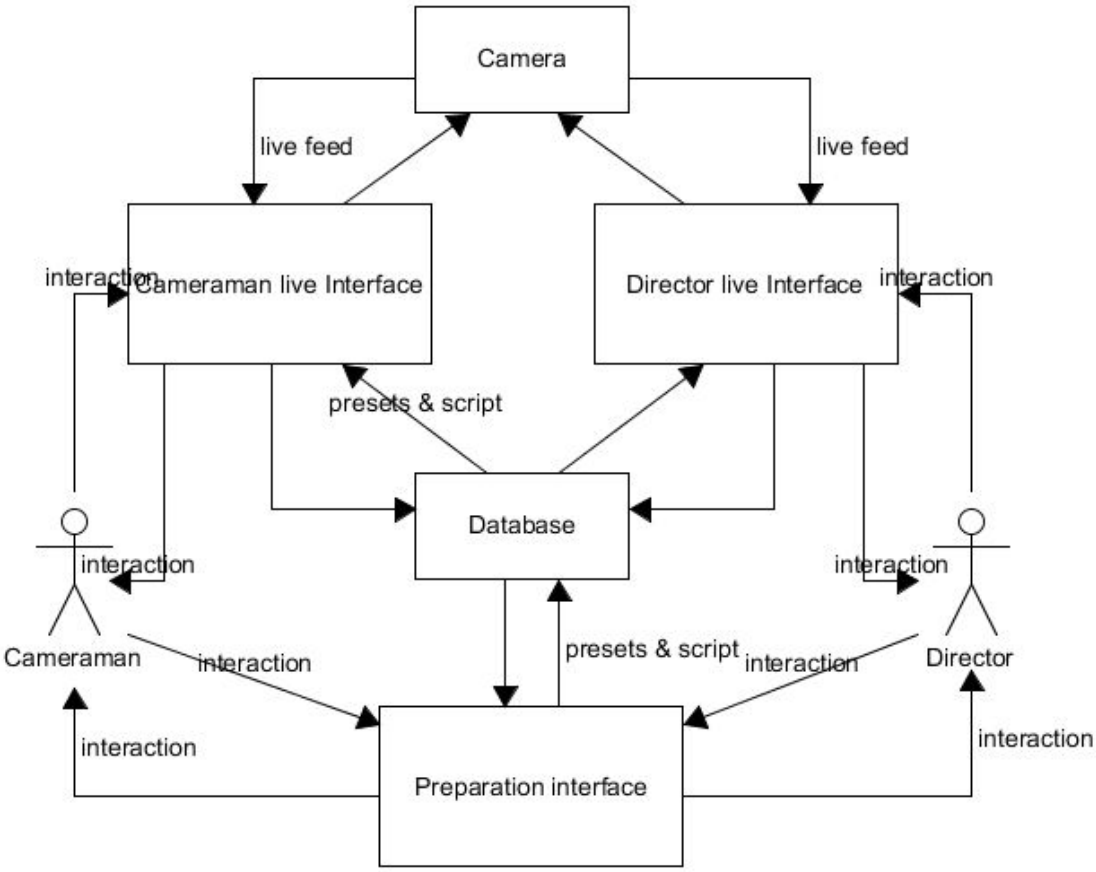
2. SOFTWARE ARCHITECTURE VIEWS

2.1 Subsystem decomposition

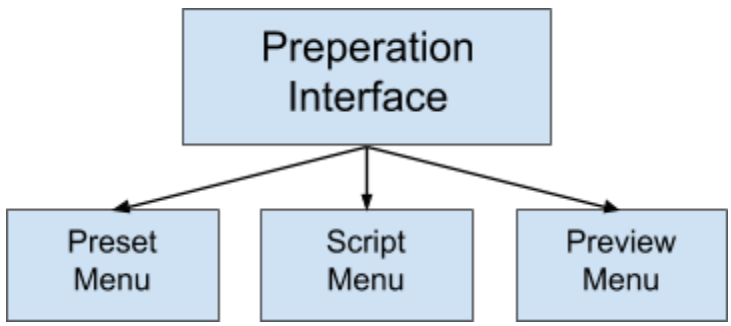
First of all we will implement the Model-View-Controller design pattern. This means our system will consist of three main components: the model, the view and the controller. The view will of course be the user interface of our program. The user interface will show the script, which is implemented in the model. The user is able to change the model (which basically is the script) with the user interface. This is currently implemented in the program.



The model, view and controller will also be divided into subsystems. The view will have three different interfaces. One used before going into live recording, the other two are used during live recording. The director and the cameramen will both have their own, yet similar, view of the program, as displayed below. In the schema below the database can be seen as the model and the interfaces have implemented a controller component. Our code currently features a separate view for the cameraman and the director.



The model will consist of the implementation of the digital scripts. Since this is the main component, we will probably implement this like some sort of database. The controller is mostly used in the script creation process. The director can add shots and presets with the controller, which will update the script. Before going in production the director can edit the script, add presets and preview the script with the preparation interface. The preparation interface consists of these three menus. As displayed on the right. These menus are currently all implemented in our code.



2.2 Hardware/software mapping

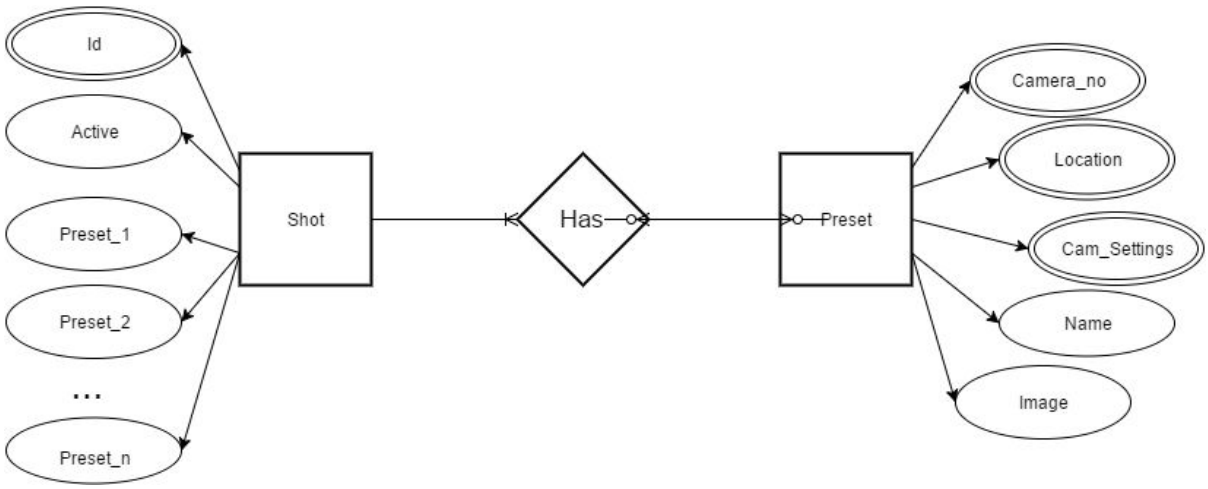
In order to communicate with the remote cameras we will use IP commands. The software sends IP commands to the cameras over the available local network, which are then interpreted by the camera's API to finally be executed by the camera itself.

Currently, there is an abstract CameraConnection class that handles communications with a camera. A camera then has one such object to represent the connection of that virtual camera to its physical counterpart. Two classes currently implement this CameraConnection class, namely MockedCameraConnection and LiveCameraConnection. A LiveCameraConnection uses HTTP messages to connect to a Panasonic AW-HE130 camera and receive and interpret responses. A MockedCameraConnection is a simple, local implementation of a CameraConnection to provide some simple functionalities without actually connecting to a camera. The CameraConnection class can be extended to create drivers like LiveCameraConnection to connect to other cameras.

As explained above, the cameramen and the director will have a different view of the program, to allow everyone to quickly see what is most important to them. Since the director is the main person who decides when the application goes to the next shot, the main instance of the program will run on his computer. All other instances of the program, such as the view for the cameramen, will connect to this main program to get its data, and, along with the current view of the camera itself, set up their own view of the program. Having a main application like his allows for easy controlling of all instances. This still needs to be implemented.

2.3 Persistent data management

In most systems, there is the presence of a database to store the data which that system utilizes. In the same manner, we would have one central script object that will be able to store the data which we will need during the entire process. We want to store and reuse shots and the presets of the individual cameras during those shots in order to create and play out a script.



For shots we will store the active camera and the presets per camera, up to an n number of cameras. The shots will be given an Id as key. The presets will be stored by camera number,

location the camera is looking towards, and its camera settings such as color settings. Furthermore, a name and image is stored to help the user identify which preset this is. This has already been implemented in the back-end.

2.4 Concurrency

A lot, if not all, desktop and mobile devices nowadays have multiple processor cores, meaning they can run multiple processes at the same time. We want to be able to exploit this by making use of multi-threading. To do so, we will have a number of threads, running a number of independent tasks. For example, the main user interface could run on one thread, while the current camera view window runs on another and the back-end model of the application runs on yet another thread. This does however bring the problem of shared resources. One thread should not be able to access shared data while another thread is writing to the data, because this causes inconsistencies in the data. In computer science, this is commonly referred to as the shared data problem [3]. In order to omit this problem, we will make use of mutexes, which enforce that only a single process may read or write the data at any time.

During live production we have two different interfaces, the director and the cameramen view. They will both be able to interact with the interface, so this might give concurrency issues. We plan to solve this issues by always giving the director priority, since there is only one director at a time this will solve all collisions between a cameraman and a director. When two cameraman want to edit the same value, we need to have another solution. We will use a first-come first-serve policy. When an edit is in progress and another party wants to edit the same property, the current edit will be performed and the third party will get a message that his changes could not be implemented.

Currently we only have one thread so there are no concurrency problems yet. However, the classes ‘SaveScript’ and ‘LoadScript’, for saving and loading scripts to and from the filesystem, do use mutexes to achieve thread safety, in a possible future case that saving and loading may be handled by a separate thread.

3. GLOSSARY

CheckStyle

Tool used for checking the style of Java code. It gives warnings about use of indentation, method length, comments explaining the code, variable and class names, etc.

EclEmma

EclEmma is a free plugin that calculates the percentage of code accessed by tests. It can be used to identify which parts of your Java program are lacking test coverage.

FindBugs

Static analysis tool that finds bugs in your code. It looks for bug patterns. Difficult language features. Misunderstood API methods. Misunderstood invariants when code is modified during maintenance. Garden variety mistakes: typos, use of the wrong boolean operator

Git

A version control system which takes snapshots of your project to which you can go back to.

GitHub

A web-based Git repository which allows users to collaborate on Git projects.

JUnit

JUnit is a framework to write repeatable tests. This is standard for Java.

Maven

A tool that automates building of the project. It describes how software is built, and describes its dependencies.

Mutex

A mutex, derived from 'Mutual Exclusion', is a dummy object that other objects in Java can lock onto, by means of the keyword 'synchronized'. This keyword then ensures that only one thread at a time can have access to this mutex, thus enforcing mutual exclusion of everything that happens within the synchronized block. Writing to and reading from shared values should be done with the use of mutexes to ensure mutual exclusion.

PMD

Tool used for analysing source code. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, duplicated code, overcomplicated expressions, suboptimal code, and so forth.

Travis CI

A tool to build and test projects hosted at GitHub. Travis CI automatically detects when a commit has been made and pushed to a GitHub repository that is using Travis CI, and each time this happens, it will try to build the project and run tests.

REFERENCES

1. http://groups.csail.mit.edu/mac/users/gjs/6.945/readings/robust-systems.pdf
2. J. Voas, "Fault Injection for the Masses," Computer, vol. 30, pp. 129–130, 1997.
3. Kamal, R. (2008). *Embedded systems* (2nd ed., pp. 326-327). New Delhi: Tata McGraw-Hill.