FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Framework for configurable video analysis
Student:	Benedek Molnár
Supervisor:	Ing. Jan Hejda, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

The aim of the work is to design a framework for pipeline video processing in order to identify people and their facial expressions of emotions. Design a Python configurable and modular framework for real-time video processing and analysis and storing the results in a database. Using appropriate image processing packages, implement modules for retrieving video from a camera or file, buffering it, recognizing faces in the image, identifying them based on the stored photos, estimating the expression of emotions in faces, exposing the analysis results via HTTP in JSON format, and storing them in a selected database. Implement a web application for configuration and real-time visualization of the analysis process. Document the framework.

Electronically approved by Ing. Michal Valenta, Ph.D. on 4 October 2021 in Prague.

Bachelor's thesis

FRAMEWORK FOR CONFIGURABLE VIDEO ANALYSIS

Benedek Molnár

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jan Hejda, Ph.D.
May 8, 2022

Czech Technical University in Prague
Faculty of Information Technology
© 2022 Benedek Molnár. Citation of this thesis.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Molnár Benedek. *Framework for configurable video analysis.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

Acknowledgments **vi**

Declaration **vii**

Abstract **viii**

List of Acronyms **ix**

1 Introduction **1**

2 Goals **3**

3 Background and requirements **5**
- 3.1 Hydronaut background . . . 5
- 3.2 Requirements . . . 5
- 3.3 The VizEmo package . . . 6

4 Requirements analysis **7**
- 4.1 Modifiable and upgradeable structure . . . 7
- 4.2 Pipeline and buffering . . . 8
- 4.3 Integrating the face and emotion identifying package . . . 8
- 4.4 Storing the information in a database . . . 8
- 4.5 Web application for configuration . . . 8
- 4.6 Exposing the analysis results via HTTP . . . 9
- 4.7 Documentation . . . 9

5 Identifying the components **11**
- 5.1 The source . . . 11
- 5.2 Buffering . . . 11
- 5.3 Buffer manager . . . 12
- 5.4 Plugin . . . 12
- 5.5 Plugin manager . . . 12
- 5.6 Plugin-manager collector . . . 13
- 5.7 Output manager . . . 13

6 Structure **15**
- 6.1 Use case diagram . . . 15
- 6.2 Domain model . . . 16
- 6.3 Class diagram . . . 17

7 Core system **19**
- 7.1 Input sources . . . 19
- 7.2 Managing multiple input sources . . . 20
- 7.3 Buffering the frames . . . 20
- 7.4 Reading and writing JSON files . . . 21

7.5 Managing the buffers . . . 21
7.6 Processing unit . . . 22
7.7 Plugin managers as processing presets . . . 22
7.8 Setting up the pipeline . . . 23
7.9 Communication between modules . . . 23

8 Extensions **25**
8.1 Plugin template . . . 25
8.2 Example extensions . . . 26
8.3 Integration of the face and emotion detection algorithms . . . 26
8.4 Saving the results to the database . . . 27
8.5 Universal database handler . . . 28
8.6 Snapshot viewer . . . 28
8.7 Saving data to CSV . . . 28

9 Web application **29**

10 Testing and documentation **31**
10.1 Testing . . . 31
10.2 Documentation . . . 31

11 Discussion **33**
11.1 Accomplishments . . . 33
11.2 Observations . . . 33

12 Conclusion **35**

A Application screenshots **37**

Contents of the enclosed media **45**

List of Figures

6.1 Use case diagram; Created by the author 15
6.2 Domain model; Created by the author 16
6.3 Class diagram; Created by the author 17

7.1 Single stream pipeline example; Created by the author 19

8.1 Class structure of the plugin template; Created by the author 25

A.1 Web application—Active cameras; Created by the author 37
A.2 Web application—Modifying camera settings; Created by the author 38
A.3 Web application—Active buffers; Created by the author 38
A.4 Web application—Modifying buffer settings; Created by the author 39
A.5 Web application—Active plugin managers; Created by the author 39
A.6 Web application—Active plugins inside a manager; Created by the author 40
A.7 Web application—Modifying the data saver plugin; Created by the author 40
A.8 Web application—Active outputs; Created by the author 41

List of code listings

8.1 Source code example of the gray scale modifier plugin 26

I would like to express my gratitude to my primary supervisor, Ing. Jan Hejda, Ph.D., who guided and helped me throughout this project. I would also like to thank my family who endured this long process with me, always offering support and love.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 8, 2022 . . . .

Abstrakt

Předmětem bakalářské práce je návrh a vytvoření konfigurovatelného frameworku (systému) v programátorském prostředí Python pro analýzu externích dat—videa. Prezentovaný úspěšně navržený a naprogramovaný systém kontinuálně sbírá, zpracovává a exportuje data ve formě „pipeline“. Framework systému je otevřený a umožňuje přidávat další funkce pomoci externích rozšíření—pluginů. Systém má v sobě integrovaný speciální softwarový balíček pro analyzování lidských tváří a emocí, který byl poskytnutý zadavatelem projektu. Tento analyzátor tváří byl řešitelem projektu dále rozšířen o funkci pro ukládání výsledků analyzátoru do externí databáze pro další zpracování. Celkový proces sběru, zpracování, analýzy i exportu surových dat, včetně připojených a použitých rozšíření lze konfigurovat a sledovat pomocí webové aplikace.

Klíčová slova framework, video analýza, rozpoznávání obličeje a emocí, python, opencv, webová aplikace

Abstract

In this project, we designed and successfully created a customizable Python framework for video analysis. The system collects, processes and exports data in the form of a pipeline. The system can be upgraded with plugins—extensions. We integrated a face and emotion analyzer package—provided by the project assigner—and we extended it further, so we would be able to collect the data output into a database. By collecting the output, it allows us to further analyze and process the gathered data. The pipeline and each plugin can be customized through a web application which also includes paths to reach the raw data.

Keywords framework, video analysis, face and emotion recognition, python, opencv, web application, ENGLISH

List of Acronyms

API	Application Programming Interface
CSV	Comma-Separated Values
FPS	Frames Per Second
HTTP	Hypertext Transfer Protocol
ID	Identification
IP	Internet Protocol
JSON	JavaScript Object Notation
REST	Representational state transfer
UML	Unified Modeling Language
URL	Uniform Resource Locator

Chapter 1
Introduction

The idea of this project came from the Hydronaut research group. The Hydronaut project specializes in building underwater research stations which can be used for training, medical, biological and psychological research. Their latest station, the Hydronaut H03 DeepLab, is designed for long term stays for small groups of scuba divers.[1] These experiments require a lot of precision and attention to keep everything smooth and safe, especially in these long term operations. Every experiment requires a whole specialized team to control and keep the life support systems online and to accomplish the goals of the experiment.

In this modern age, we can use the power of computers to automate the majority of the processes. This will allow us to keep everything under control, make precise calculations and predictions and to use our human resources for further development. The goal of this project is to help to improve the experiments, by making the process easier and automated. This will free up human resources who can work on more important parts of the project, while this automated system will still work without intervention in the background.

This framework will allow them to monitor and record human conditions based on their emotions. It is going to be a standalone system based on their face and emotion recognition algorithms which are going to be integrated into our system. Every face picked up by the camera is logged and paired with the corresponding emotion. This data is saved and stored in a database. This will allow other systems to work and analyze this data. More life supporting systems can be built based on this stream of information. From the start, it has to be designed to be easily modifiable and upgradeable. This will happen through an API and a web interface which will be able to change parameters and configuration in the pipeline.

Chapter 2
Goals

The goal of this project is to make a framework for video processing. The two key features of the framework are extensibility and configurability. The video frames need to be processed through a pipeline. This means it has to be modular, where each module will be in a different role in the pipeline. The framework has to work together with the video processing packages provided by Hydronaut. After the analysis, the data must be uploaded and stored in a database. In the case of longer frame processing, we have to buffer the frames to prevent dropping them. It must have an interface which will read data from the database and display it. Besides, it needs to be prepared to retrieve information in JSON format. The configuration will happen through the web application or by manually editing the JSON preset files, which the program reads at initialization.

Chapter 3
Background and requirements

3.1 Hydronaut background

This project serves scientific purposes and it is built to serve specific needs for the research group Hydronaut. The Hydronaut Project is the designer and creator of the DeepLab H03 underwater research station. In 2020, the H03 station's crew on their first mission set a new national record for the Czech Republic, for the longest time spent underwater (over 175 hours). This training and research center is designed for long term stays underwater. As they mention on their website, its usability is exceptionally broad.—*"It includes research and training programs, hyperbaric medicinal and psychological research, development and testing of technology, IRS, special army units or space agencies training."*[1]

During the missions, they use multiple softwares and automated processes to monitor and keep the crew alive. Besides that, they have to accomplish, log and document the goals set by the partners and investors. This framework will help to set up an automated process which will capture, analyze and log the faces and their emotions. It is going to be a superstructure on their existing and working algorithm. This package, edited by my supervisor, Ing. Jan Hejda, Ph.D., was given to me to serve as a basis for face and emotion recognition in my work. The original project was created by Martin Vadlejch, called "Real-time Facial Expression Recognition in the Wild".[2] This upgraded package has to be integrated in to my superstructure to be able to process video sources in real time. We will analyze this and discuss the details in the following sections.

3.2 Requirements

As the first step, we had to get acquainted with the assignment. We made notes on the key functions and elements, which needed to be further clarified. After multiple consultations with the supervisor, he presented us the vision and clarified the key functions. These functions and technologies are going to be analyzed in the next chapter. The key features are:

Modifiable and upgradeable structure As we mentioned in the last section, the goal is to create a framework which is going to be further integrated, used or extended. That means the structure has to be well designed and has to be flexible.

Pipeline and buffering A basic pipeline processes a sequence of tasks. The tasks and processes receive their input generally from the previous stage and transfer their output to the next stage. We need to collect the data into a buffer in case of any of the processes is slower, so we don't lose frames from the real time footage.

Integrate the face and emotion identifying package The VizEmo package given by the supervisor needs to be integrated into the framework. This package contains the working source code, materials, and libraries for face and emotion detection.

Storing information in a database After processing the frames, the information needs to be uploaded into a database, from where it will be available for further processing.

Web application for configuration The configuration will happen through a web application so it would be easily accessible from anywhere and would allow multiple connections in contrast with the offline application.

Exposing the analysis results via HTTP The application needs to have the option to expose the analysis results in a raw JSON format via HTTP so other processes could be connected on this source of data.

Documentation A key element is a good documentation. As we mentioned, this application will be upgraded and integrated. A clear documentation will allow other developers to work quickly and easily with the existing source code.

3.3 The VizEmo package

This package was provided for us by our supervisor and Hydronaut. The package contains a functional and up to date version of the face and emotion identifying application, originally created by Martin Vadlejch. The algorithm is written in Python using the OpenCV library to gather information from the active video source.

The package has two important classes—“FaceIdentifier” and “FERecognizer”. These classes handle the actual face identification and emotion recognition. The processes are run from a central launcher, which allows a few initial settings. Based on our choice and initial configuration, the data is exposed through an HTTP server or can be saved into a CSV file. Due to the high cohesion and loose coupling rule used in this package, we will be able to re-use the main components separately in my framework. This way, we will be able to process the data differently without even touching the original launcher.

Chapter 4
Requirements analysis

In this chapter we are going to discuss, analyze and find possible solutions to the requirements. But first we need to clarify what a framework is.

Framework is a basic software structure which serves as a foundation for software projects with the purpose of making the development easier. It's an abstraction in which it provides generic functionality which can be re-used, modified, and extended. In other words, a framework is a scheme, template that simplifies the process/work. Using frameworks gives us a lot of benefits, like:

- pre-built functions and structures can save us time,
- pre-built and pre-tested parts, this implies a more reliable application,
- easier testing and debugging,
- it can help to prevent code duplicates,
- code is more secure.[3, 4]

4.1 Modifiable and upgradeable structure

The key to this feature is to identify the components correctly. We will refer to these components as managers later in this document. These components are going to handle the assigned processes. The pipeline is going to be the organized line of these managers, where the information will flow through them. By following the high cohesion and loose coupling rule, every manager will fulfill one task and will be configurable on his own.

One of the awesome technologies that we probably use and interact with on a daily basis, but seldom do we realize its existence is the plugin architecture. This architecture consists of two components: a core system and plug-in modules. The idea behind it is to allow adding additional features as plugins to the existing core application, providing extensibility, flexibility, and isolation. The rules and the processes are separate from the core application, allowing us to add, remove and change the plugins at any given point with no effect on the rest of the application.[12]

The core system defines how the system operates and the basic business logic. The plug-ins are independent, stand-alone modules which contain additional features and custom functions to extend the core system.

4.2 Pipeline and buffering

As we mentioned in the previous subsection, an ordered line of managers is going to be the foundation for our pipeline. The pipeline will begin with the camera—source whence we get the information. After getting the information, it has to proceed into the buffer, which will store the frame until it gets popped and processed. When the processing is done, the data can be uploaded and logged into the database while the frame can be displayed or saved.

Based on the given information, the face and emotion identifier can be really slow depending on the used hardware. The buffer is going to help us to store the frames until the previous frame gets processed. This way, we don't lose valuable information from the source. The buffer shouldn't be infinite as our hardware capacity isn't infinite either. Our buffer will have a maximum capacity which will limit the number of stored frames. If the buffer gets filled up, then based on its configuration it will start to drop the newest or the oldest frames to free up space for the new incoming frames.

4.3 Integrating the face and emotion identifying package

Based on the idea of realizing a plug-in architecture mentioned in the section 4.1 the integration can be solved via a plugin. From section 3.3 we know that the algorithm is started from a main launcher file, but all of its components can be re-used separately. Following this idea, we can implement a plugin which is going to replace the role of the launcher. We can initialize the needed classes when we load and initialize the plugin. This way, we will be able to call the plugin to process our data any time during the process.

4.4 Storing the information in a database

At first sight, this idea may seem really easy. By following the thought process and projecting this idea on our plugin architecture, the solution gets more complicated. We have to take into consideration that every plugin will have a different output. All the output data will have a different structure. The data may or may not need to be stored in one database or multiple databases.

Implementing a function right into the face analyzing plugin would allow us to connect and save to the database would solve this problem. This way, we could easily maintain the structure between the output and its table structure in the database. However, this would not solve the problem for any other plugin which would need its own implementation for the same functionality. This would violate the principles of our framework.

The solution to this problem could be a centralized implementation of a database handler in the core system. By having a centralized solution, any of the plugins can create and maintain a connection on their own if it is needed. The benefits are that we can save to multiple different databases, every plugin can define its own data structure and will allow better configuration.

4.5 Web application for configuration

For configuration of the managers, the plan is to make a static web application. By static application we mean that after the request the server will return an HTML page to the user agent which will render it.[6]

The main functionalities are to display and be able to change the configuration of the managers. Regarding Python, we have a lot of technological options to choose from. Here we are going to compare two of them.

Django is a full-stack framework. It is a high-level Python web framework that encourages rapid development and clean, pragmatic design. This framework offers a standard method for fast and effective website development. The primary goal of Django is to create complex database-driven websites. [7, 8]

Flask is a micro framework, which offers basic features of a web application. This framework has no dependencies on other external libraries.[7] It includes a built-in development server, unit testing support, and with Jinja2 it allows us to use page templates. It provides enough flexibility to expand the application quickly and easily.

In conclusion, we are going to use Flask as the key to our core system. With this framework we can create a flexible RESTful API, which can take care of the configuration. In our case, the representation state is going to be HTML except for the functionality where we expose the data in JSON.

4.6 Exposing the analysis results via HTTP

As we mentioned in the last section, we are going to use a REST API which will allow us to communicate with the system via HTTP. This gives us the option to create countless URLs which fulfill our needs. In 4.4 we presented a problem and a concept for the solution. In this topic the same problem arises. Every plugin may or may not have a data output. Each of the outputs could be different.

If we extend the previously mentioned concept, we can use the same logic to also solve the problem with this functionality. If necessary, the core system can ask if the plugin has a data output or not. In case if there is a data output, the data can be requested and processed further.

4.7 Documentation

Through the process, we have to make sure that every part is well documented. This way, the future developers will be aware of all the thoughts and logic behind the modules and elements. Python offers us multiple choices for documentation. We have two basic built-in options, which are the comment and the docstring. A comment is a short string to clarify the purpose of the commented item. It helps the programmers better understand the functionality and intent of the commented element.[9]

A Python docstring is a short description used to document a Python module, class, function or method, so programmers can understand what it does without having to read the documentation.[10] After having our Python code documented with docstring we can use Sphinx to auto generate our documentation. Sphinx is a powerful documentation generator that has many great features for writing technical documentation including for example syntax highlighted code samples and the ability to generate web pages, printable PDFs and more.[11]

Chapter 5
Identifying the components

In this section, we are going to identify and discuss the main components of our core system and pipeline.

5.1 The source

Starting with the source and input. We will refer to this element as "camera", however our input can be a video file, web camera or IP camera. Our structure should be able to manage multiple different sources at once, so each of the inputs will be represented as an object of the camera class. We are going to give an ID for each camera this way we can easily refer to them, and a name so it could be easily identifiable for humans too. A camera needs to hold the CV2 capture unit which will provide us the frames from the source, and we will hold the source path too, so later we could refer to it in the configuration.

Most of the time the sources, for example an IP camera, support different video capturing setups. This is an important configurable element. We should be able to read and change the current configuration. So that we don't always have to reconfigure the cameras, we are going to make default JSON presets. This way, we can easily set up multiple profiles or edit the existing ones. The camera will read a default preset when it gets initialized.

The camera does not always need to be active. We may just created it as a preset or for later sessions. We are going to give it a start and stop function. This will give us the ability to use it only when we are processing the input. The start will initialize the source and create the CV2 reader, while the stop will release this reader.

5.2 Buffering

The buffer is a really important part of the pipeline. This will keep our frames safe and prevent flooding our system. This module will be a collection of frames. We are going to use a queue for this purpose. We will process the oldest frame—first element—while we are filling up the collection from the rear. To not to mix the frames from different cameras, every buffer will be connected to one camera. The buffer is going to be filled with a function running on a separate thread to keep the frame collection outside the main process. This way, we will be able to run multiple buffers with multiple cameras.

The buffer will have three modes to choose from. Infinity mode will ignore the limit of the buffer, while modes "throw-new" and "throw-old" will define which frames to drop. The limit and the mode will be set at initialization from a default configuration file. By using the same

method as at the camera, we will have a default preset which can be changed, or the user can define his own presets.

5.3 Buffer manager

As we discussed in the last section, every buffer is going to have a filler function which will run on its own thread. So we could keep everything under control we are going to create a buffer manager. This will maintain a collection of our existing buffer objects, and we will be able to control them through this manager. The key function of this class is the frame collector function. This function will be sent to a new thread which will collect the frames from the given camera. After collecting the frame, it will be added into the connected buffer. We are going to need a start and a stop function. These functions are going to create, start and join the active streams, threads.

5.4 Plugin

Plugins are going to be the most modifiable and extendable elements. These are going to be the movable and changeable modules in the pipeline. We are going to define a plugin abstract class, which is going to serve as a template for further use. All the active plugins are going to be managed by a plugin manager. Based on the template, we can implement different plugins for different purposes. Some plugins may require an initialization before processing, so we are going to have an initialize function which is going to be called automatically for every plugin at registration. They will share a process method which will define the main function for each plugin. This function is going to be called in the pipeline. Function *process* and *init* are going to be defined as abstract functions, so the subclasses must implement them.

As we discussed in the previous section from some plugins, we may want to get information. The problem is that the all the different plugins may or may not have data output. To solve this problem, we have to create a universal function which will meet the needs of all the different plugins. We can create a function in the template, which is going to be predefined as there is no data output. This return will be handled by the different managers and applications. On the other hand, if the plugin needs to have a data output, it can overwrite the pre-defined method, and return the needed information.

Each plugin can define its own variables and conditions in the initialize function. However, editing these parameters can be really tricky. We have to get the information, edit them, the plugin has to read them and reinitialize itself. The first thought which comes into our mind is to make them as JSON configuration files as we did with the cameras and the buffer. In the case of the buffer and camera, we are trying to create presets from which the module can initialize itself, but they are mostly constant parameters. However, the configuration for the plugins serves a different purpose. They are conditional variables which change the state of the plugin based on the needs and human interaction. We have to reach them quickly and easily. In addition to this, every plugin would require its own configuration file.

The solution for this feature can be presented with a famous quote from Mahatma Gandhi, *"If you don't ask, you don't get it."*. We can ask the plugin, what parameters does it offer for configuration. We can edit those parameters and send them back to the plugin. The plugin will read and parse the information and set up the given values.

5.5 Plugin manager

As it's name says, it will manage the plugins. The purpose of this class is to load and register new plugins and to keep track of the active ones. This way, we will be able to quickly and easily

identify and reach the given plugin. This will be the active module in the pipeline which gets the data, gives the task for the plugins the process the input. After the processing is done, it forwards the data to the next module. A handy feature would be to change the order of the processing between the plugins. By keeping track of the plugins in a list, we could easily modify their order. By modifying the processing order, we can achieve more flexibility and completely different results.

5.6 Plugin-manager collector

By introducing the feature to process multiple inputs, we are forced to have a collector of the different plugin managers. This way, the plugin managers will serve as virtual presets, where the plugins are the "filters" in our data line similar to filters on pictures.

5.7 Output manager

The output manager will be our key launcher for the whole process. It sets up the whole pipeline with the chosen options–camera and plugin manager. This module is the end point of the process. In keywords, the frame and data travels from the source through the buffer, the plugin system which is the main process unit of the pipeline, ending in the output manager which will decide what to do with all the data at the end of the process. We are considering three options, live view, save to file and processing only.

The live view is an option where the processed frame gets displayed in front of the user. It creates a window and displays the incoming frames to the output. During continuous processing, it can serve as a monitoring unit if the source is live, or a playback unit in case of a video source.

The option of saving into a file can be a useful option for later documentation. This option will take a filename and will save the output of the whole process to the video file.

Only processing mode can be useful for projects where the main work is happening inside the plugins. In our case, the processing will happen mostly inside these extensions. This means the face and emotion recognition will save the data into a database. So the source is already processed and the data is already saved before it gets into the last output module. In case there is no need to showcase the frames or to save them into a file this mode will run in the background.

Chapter 6
Structure

In this chapter we will introduce you to the design of the software architecture we are planning to realize. We are already familiar with the requirements and their details, with the key components of our system, so we can start modelling the architecture of our software. We are using UML diagrams to visually represent the design. This way, we will be able to easily track our progress through the implementation, and it will help other developers to see through the structure.

6.1 Use case diagram

We are going to make a use case diagram (6.1) to recapitulate the requirements in keywords. "Use case diagrams are used to graphically depict a subset of the model to simplify communications." [13] The key use cases are when we are manipulating with the components, adding, removing or modifying the setup. Besides that, we have to have implemented other features based on the requirements.

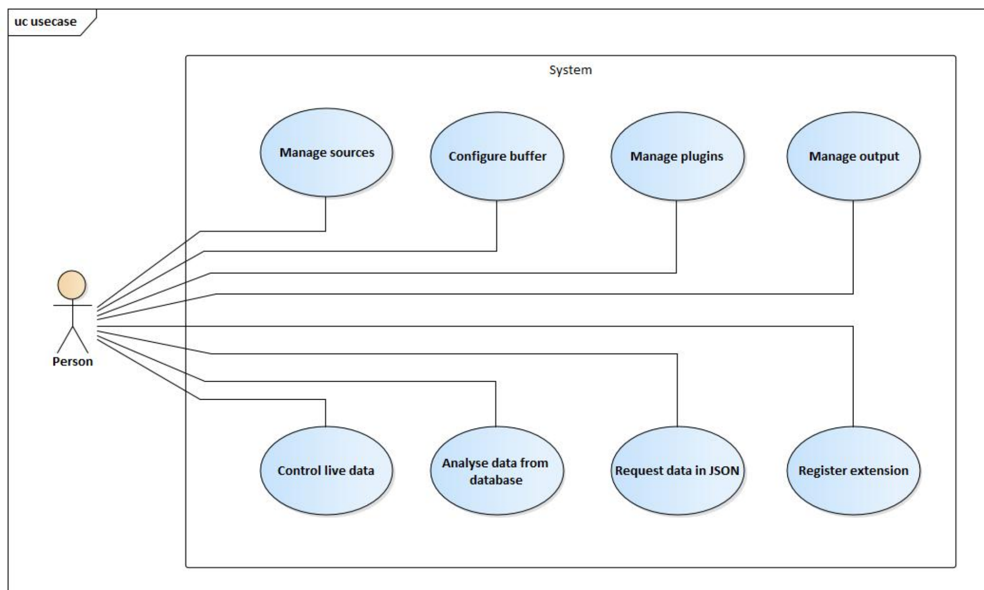


Figure 6.1 Use case diagram; Created by the author

6.2 Domain model

Domain models are a way to describe and model real world entities and the relationships between them. It is a great tool for controlling the complexity of the system under development. Domain models reflect our understanding on the relationships between the entities and responsibilities that cover the problem domain.[14]

In this project the created domain model (6.2) will really help us to identify the relationship between the entities. This way, we can see all the overlapping components, from where to where we need to establish the connections. If we are able to identify the relations correctly, the better we can enhance the extensibility.

To summarize it in a few words, we can see the outline of our pipeline—the frame is coming from the camera through the buffer into the processor which ends with an output. The user can make an implementation on the controller, or he can use our application to modify the given managers.

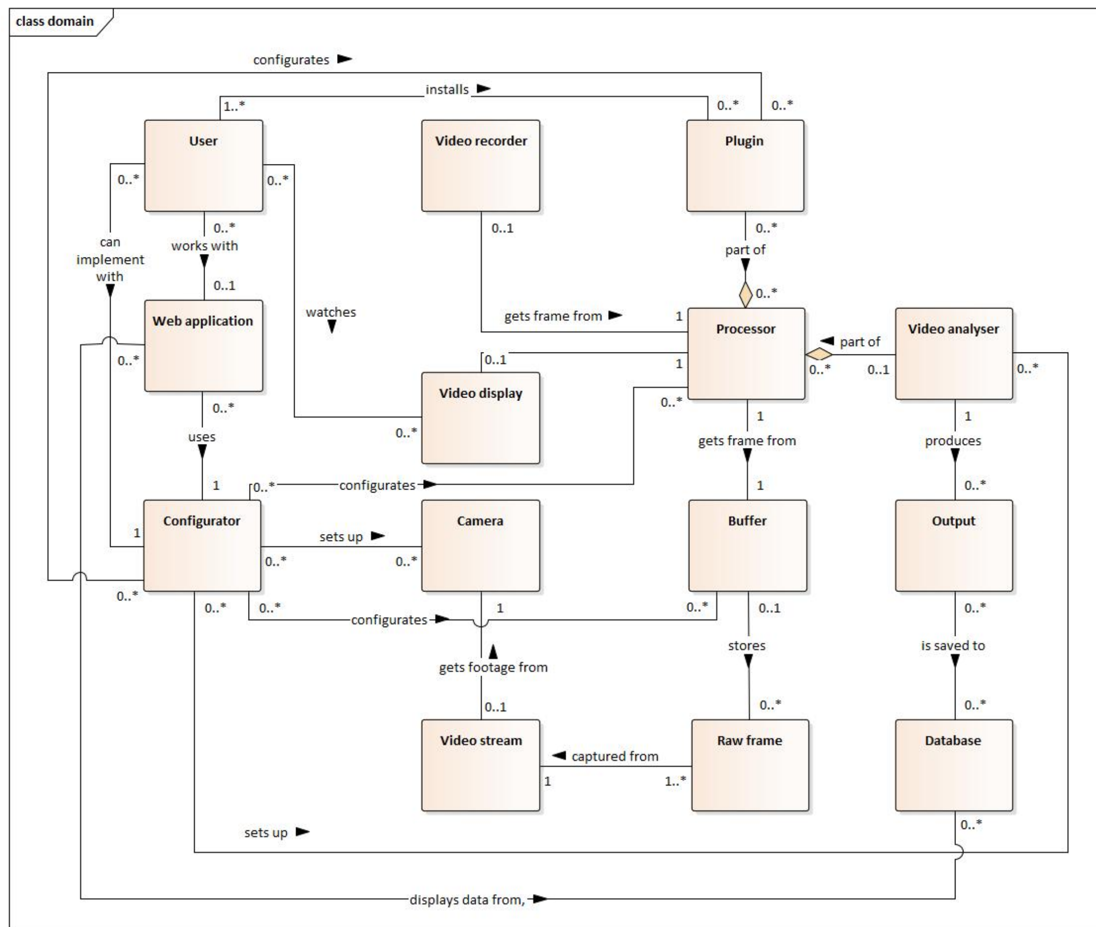
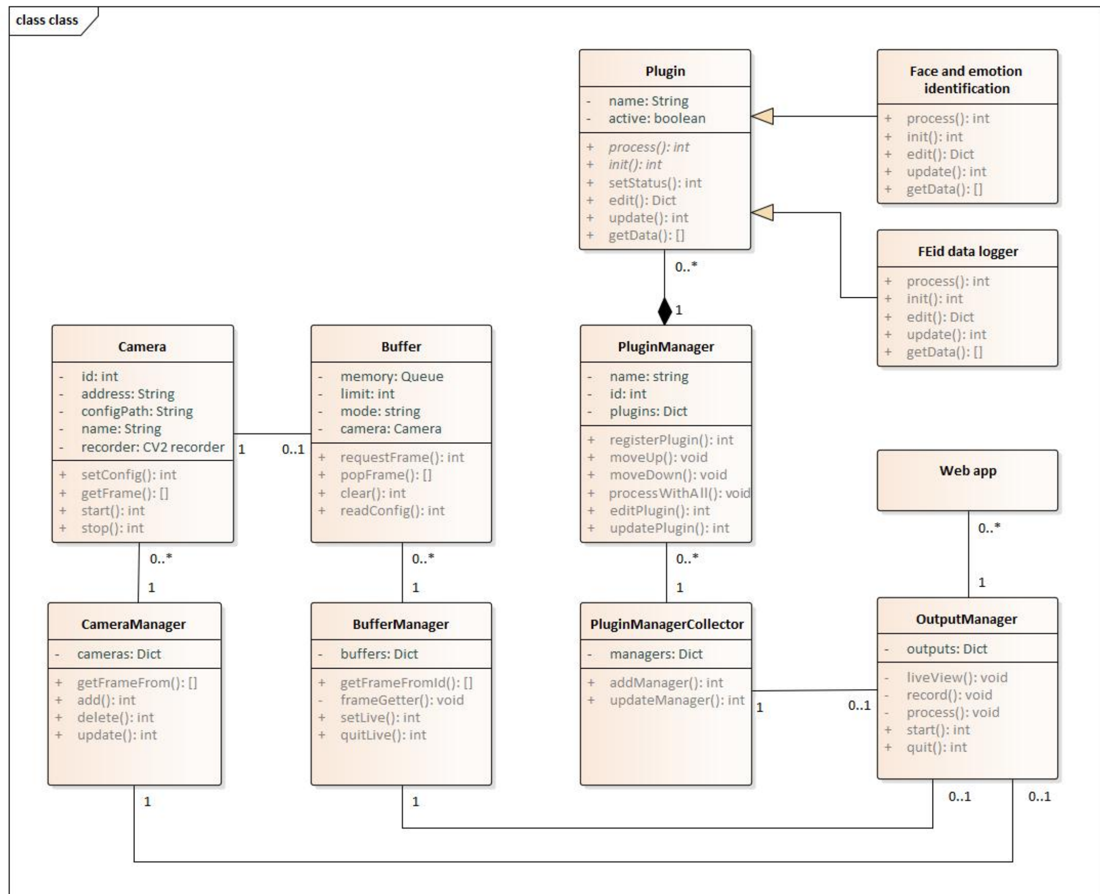


Figure 6.2 Domain model; Created by the author

6.3 Class diagram

Based on our domain model, we are going to create a static class diagram (6.3). A class diagram will give us a static view of the application, visualizing and describing different aspects of the system. The benefits of making a class diagram are that it illustrates the data models, gives a better overview and it servers as a foundation for the implementation.[15]

On the picture we can see and differentiate the core system from the plugin system. We were using the elements, attributes and functions discussed in chapter 5.



■ **Figure 6.3** Class diagram; Created by the author

Chapter 7
Core system

In this chapter we are going to discuss the implementation of the core system. The core system includes modules like the Cameras, CameraManager, Buffer, BufferManager, PluginManager-Collector and the OutputManager. The goal is to create a configurable pipeline which can be extended with the extensions. These extensions will be the processing modules. The reason why we are working with managers instead of single instances is that we want to accomplish a structure where we are able to handle multiple streams at the same time with different configurations.

We created a simple example to symbolize what a single pipeline would look like for one stream. The frame is captured in the camera, transferred to the buffer where it is stored until

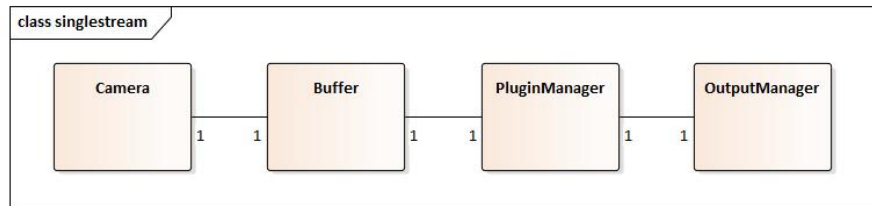


Figure 7.1 Single stream pipeline example; Created by the author

it is not requested by the processor (PluginManager). There the frame is processed by all the active plugins, then it is sent to the OutputManager where the process ends. Here the frame may be displayed or saved into a file.

Our actual pipeline is going to work a little bit differently because of the threading and the multiple streams. Let's look at them one by one.

7.1 Input sources

We are going to take a look at the camera element first. This is the first station in the pipeline. The job of the camera module is to retrieve information from the source. This information is what we call a frame.

We are using the OpenCV library to work with the image and video related elements. OpenCV is an open source library for computer vision and image processing. In this library the frames are represented with three dimensional numpy arrays containing the information pixel by pixel.

ID and name — For each camera we gave an ID and a name. The ID is for internal use, so we could parse and reach the camera more quickly. The name is the name of the camera, so

the user could differentiate the cameras more easily. This name is going to be displayed on the web application.

source — As another attribute we are saving the source. This attribute is going to be the parameter when we are creating the recorder. This can be a built in camera which can be referenced with an integer, IP or file destination which is represented via string.

recorder — Our recorder is an instance of *cv2.VideoCapture* which takes our source as a parameter. This element provides us the functionality, so we would be able to read from the camera.

isInUse — A boolean value to keep track of the current state of our camera. This attribute is true if we have the recorder set up and the camera is ready to provide a frame.

configPath — Contains the path for a camera preset. When a camera is started, it will read this preset and initialize the recorder based on the values read from the preset.

Besides the getters and setters we are implementing a few important functions: *start(), stop(), getFrame(), setConfig()*.

The reason behind creating the *start(), stop()* is that if we set up the cv2.VideoCapture it automatically activates the camera. We don't want the camera to be active all the time. It could cause errors if it is already in use or it will take away the option to use it for any other purpose. With these methods, we only activate the camera when it is needed. This way we prevent blocking any other activities on the device. At starting up the device, we read the preset from the *configPath* and change the settings on the *recorder*. Function *stop()* releases our *recorder* and the camera is no longer blocking.

Our *getFrame()* function is called to get a frame from the source. We set up a repeater to repeat the request until it gets a frame or the repeater reaches the maximum value. This serves as an internal time out system. If we don't get a frame in two seconds (at thirty frames per second) we return a negative answer, which is handled by the caller function where it ends the process.

7.2 Managing multiple input sources

Based on the fact that we can have multiple cameras, we need a manager that keeps track of them. This manager does not have any special assigned tasks, only to keep a list of the available cameras. This module is a middle man between the camera object and the next module in the pipeline. However, this manager will be useful when we are making the user interface for the application. With this module, we are able to list our cameras easily, because we are already tracking them.

source — Our only attribute is a dictionary which is holding the available cameras. In this dictionary the cameras are differentiated by their ID's.

We have implemented multiple getters and setters for basic interactions with the manager and deeper interactions with the cameras. These functions will allow us in the application to interact with the cameras without getting the camera object.

7.3 Buffering the frames

The buffer is going to be the next milestone in our pipeline after getting the input from the source. It's task to keep the frames until processing if our processing unit can't process the earlier frames fast enough.

camera — Each of our buffer object has an assigned camera from whom it gets the frames. This decision to pair a camera to the buffer was born to avoid the amalgamation of frames from different cameras. Thereby each buffer will be able to collect frames only from one camera. That is it can't happen that we get a foreign frame into this stream.

queue — We are using a queue to buffer the frames in the order they got collected—it's FIFO property. The Queue Python library gives us a regular queue option which already implements all the locking semantics for threading which we need.

limit — Limit is a custom integer limit which sets the maximum size for the queue and it is tracked by this class. However, we could set the maximum size of the queue in its constructor, we decided to track it manually, so we would have the option to change the queue size while we are running the application.

amount — This is a counter which saves the number of frames currently in the buffer. This variable helps the functions to work correctly in the current mode.

mode — Mode is a string variable which holds the type of the buffer. Our buffer currently supports three modes. By changing this variable to a different type, we can modify the behavior of our buffer while our application is online.

The buffer is initialized from a default configuration file, from where it sets the limit and the mode. If there would be any error at reading the default file, it is initialized with the in-code setup. We have a function to read a file from the path given in the argument and change the configuration based on the parameters in the file. We have the option to set the mode and limit separately, these functions are used in our web application too.

We have a *requestFrame()* function which requests a frame from the paired camera and handles the frame based on the current mode. In infinity mode, we allow the buffer to go infinitely, while in the throw-old mode, if the buffer is full, we pop the oldest frame and place in the new one. The throw-new mode it skips the frame and leaves the buffer in its current form. On the other hand the function *popFrame()* takes the oldest frame from the queue and returns it.

7.4 Reading and writing JSON files

As we mentioned in the previous sections, we are reading the configurations from JSON files. We created a simple extension which handles our JSON transactions and uses the same return codes as we are using in the whole project. This makes it quicker and easier to read and write the files at any part of the project with a simple line.

7.5 Managing the buffers

To keep the video stream live we have to collect the frames among everything else. The buffer manager is not really a buffer manager, rather it's a thread manager. We are creating threads for the functions which fill up the active buffers. We have one attribute:

liveCameraThreads — This variable is a dictionary where we pair the cameras with the active buffer filling thread and the connected buffer.

We are starting the process with the *setCameraLive()* which activates the camera with its start function. After setting it active, it creates a new buffer and pairs it with the camera. For the new buffer it launches a thread and saves it in to the dictionary. This way we keep all the created elements for later use.

The opposite of this function is *quitLive()*. This function stops the frame collection, deactivates the camera and clears the entry in our dictionary.

7.6 Processing unit

In this section, we are going to discuss the processing unit of our pipeline. We are referring to the plugin manager with this name, because all the image and data processing happens in the plugins. In any other part of the pipeline, we are not manipulating or processing the frames, we are just transferring them. The plugin manager is a set of plugins which can be used as a preset to manipulate with the frames. Its main task is to load and register new plugins, keep track of them and help to make the modifications within the plugins.

To load and register new plugins we are using a Python library called *importlib*. This package provides us the implementation of import, which enables us to load different packages even when our pipeline is already live. This gives us a wide range of options to configure the process.

We need to be aware that every package can be loaded only once. If we would like to run multiple streams with the same plugins, it would load them several times, which would cause an error. To solve this problem, we created a class attribute to save the links referring to the packages. In Python the class attributes are shared between all the classes while the instance attributes belong to only the given instance. If we would like to add a plugin to the manager and it's not registered yet, we will load and register it. In case it is already registered, we are using the link to create a new instance in the new manager.

registeredAll — This is our class attribute. It's a dictionary where we keep all the registered plugins with their name as the identificator.

name — We are keeping a name for the manager to be easily recognizable for humans. We are setting its name when we are creating the manager.

orderNumber — It is an integer counter, so we would be able to give unique IDs for the incoming plugins. We are using this unique ID to order the plugins—which defines the processing order.

activePlugins — A dictionary to hold all the plugins activated in this instance of plugin manager. The plugin is registered in the dictionary with its given ID.

Besides the setters and getters we have the functions *loadPlugin()* and *registerPlugin()* which take care of loading and registering the plugins. For all functions we are using thread blockers until we realize the task. The reason for that is that we are working with the processing units. When the stream and processing are active, any change in the processing order could cause our application to crash. If we would like to edit the processing unit, we have to wait until the current iteration finishes, so we could modify the unit and continue the processing. This way, any modification can be done safely.

We implemented two functions *moveUp()* and *moveDown()* so we would be able to change the order in the dictionary. By changing the order, the processing order changes too. This provides us the option to set up and easily modify the order in which the frame gets processed. By changing the processing order, we can achieve different output results.

However the key function of this class is the *useAll()* function. This is the function which launches the processing on the input. We take all the currently available plugins and send in the input arguments. Each plugin takes the needed data from the arguments, processes it and returns it back. Then the fresh set of arguments is provided for the next plugin. This opens up the possibility for the plugins to communicate with each other.

7.7 Plugin managers as processing presets

By creating a plugin manager, we create a set of plugins which is connected into the pipeline. We may want to create, set up and initialize the set before it is connected into the pipeline or as we

discussed, we may want to run multiple streams. Each stream requires its own plugin manager. To sum up, this gives us two new requirements regarding this part of the project. We need to be able to set up the manager beforehand, and we need to be able to set up multiple managers.

This is where we introduce the module which will collect all our plugin managers. It's task is to keep track of our created plugin managers. It's a simple class with one class attribute which registers the new managers. Besides that, we implemented the needed getters and setters. This manager will hold all of our plugin managers and it takes its position between the buffer and the output.

7.8 Setting up the pipeline

Our last element in the pipeline is our builder element. The outputManager joins all the components together to form a pipeline and to get a video stream up and running. It decides what mode we are going to use and what is going to happen with the frame at the end. It currently supports three modes.

The live view is a mode where the frames are displayed in a window on the host computer. It is a video playback with live data from an active camera. It can be used as a monitoring system just like at surveillance camera systems.

It supports a save to file mode where the frames are exported to a file. This mode takes two extra arguments as input parameters. The first one defines the path and name of the file where it needs to be saved, the second defines the FPS. By modifying the output FPS value, we can speed up or slow down the video. This mode gives the user the option to document and archive the footage for later use.

The third mode it supports is the processing mode. This mode only keeps the process running. It drops the frames at the end of the processing. It is useful when our processing units produce outputs which are saved, and we don't have to see or save the actual output footage.

outputs — We initialize this module with all of our managers. Besides that, we have a dictionary attribute where we are going to save our currently active outputs. An element in this archive contains the pluginManager we are using, the created thread and the mode it is running in.

We start the whole process with the function *start()*. It checks if the given components exist or not, selects and launches the chosen mode in a new thread. By launching it in a new thread, we give the user the option to run multiple streams with multiple different components. The stream can be ended with the function *stop()* which shuts down the components, joins the thread and deletes the entry in our archive.

When the components are launched, the buffer starts to collect frames from the paired camera. It requests a frame from the buffer extends the arguments with the new frame and run the *useAll()* method from the processing unit with the new set of arguments.

7.9 Communication between modules

As we mentioned in the previous sections, we are communicating with a collection of arguments between the modules. We are using a dictionary to save multiple elements with specific keywords. The modules can extend this collection or use parameters from it. Each module can get the data it needs from the supported keyword.

If the dictionary does not have the needed keyword, the module will skip its task and the process will continue with the next module. A missing keyword can indicate, for example, that the processing units are in the wrong order and the consumer is trying to reach data that the producer hasn't produced yet. To solve this problem we implemented functions to change the order of the processing units, discussed in section 7.6. This idea gives us the flexibility to process multiple different elements and provides the option to accept pre-defined arguments.

This technique is used, for example, in the situation where we need to conduct the file name and FPS count from the user to the file saver if the save to file mode is chosen.

Chapter 8
Extensions

In this chapter we are going to discuss the solution and implementation of some key requirements. In the last chapter we talked about the core system, the pipeline and their main components, while in this chapter we are going to look at the plugins which extend the core system. These plugins contain the algorithms to compute, analyze the frames or to operate with the data. These plugins are handled by a plugin manager which we discussed in section 7.6 and the communication happens with the method mentioned in section 7.9.

8.1 Plugin template

This abstract class serves as our base template for further implementation for the plugins. It defines the base attributes and it defines or implements some crucial functions. All the other plugins should extend this class. A not negligible thing to mention is that all the future plugin implementations need to have a class named *Extension*. This is the name we are looking for at extension registration.

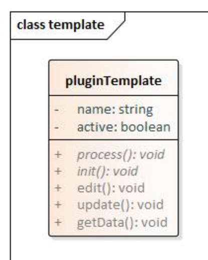


Figure 8.1 Class structure of the plugin template; Created by the author

The class defines a *name* variable which helps us in the identification in a more human form and it defines a Boolean variable called *active*, so we would be able to turn on or off the functionality of the extension.

Besides the getters and setters, we have two abstract functions. The first function is the function *init()*. It allows the developers to implement a personalized initialization function which is called in the constructor of the plugin, or we can use this function to manually re-initialize our plugin. This function should set up the needed environment for the processing function.

The second abstract function is the *process()* function which takes a collection of arguments explained further in section 7.9. This function should contain the frame or data processing logic.

To keep the processing going, it has to return the same or modified set of arguments. This function is called if the plugin is registered in an active pluginManager.

As we mentioned in section 5.4 a huge problem is the personalized configuration of the plugins. Each plugin will have a different process logic with a different environment with different needs. We are going to solve this problem with two methods. The first step is that we are going to request from the plugin what variables does it offer for configuration. This step is realized with the function *edit()* which returns a dictionary with the variable name and its value. It is already implemented to support the variable to change the plugin state. The second step is to send back the edited dictionary for the function *update()* which will then parse the data and update the variables. Both of the functions should be overwritten by the developer of the plugin.

8.2 Example extensions

To show an example, we implemented two simple plugins. They are both OpenCV frame modifiers, converting the frame to a gray scale or inverting the colors. In these examples, we only overwritten the *init()* function to set the correct name and the *process()* function, so we could modify the frame. We kept the original *edit()* and *update()* functions defined in the template. To present the simplicity of creating a plugin for our system, please see the example code below (8.1) (this is the entire file content of the gray scale modifier).

```
from plugins.pluginTemplate import Plugin
import cv2

class Extension(Plugin):

    def process(self, args:dict):
        if 'frame' in args and self.active :
            grayImage = cv2.cvtColor(args['frame'], cv2.COLOR_BGR2GRAY)
            args['frame']=grayImage

        return args

    def init(self):
        self.name = "Translate frame to grayscale"
```

Code listing 8.1 Source code example of the gray scale modifier plugin

8.3 Integration of the face and emotion detection algorithms

As we analyzed the original sources in section 3.3, we came to the conclusion that we can reuse most of the parts only by implementing a new launcher. That's what we realized in the form of a plugin. However, we had to tweak the paths in the package so it would be compatible with ours.

We have overwritten the *init()* function to set up and initialize the emotion recognizer. This part of the package tracks the faces and analyses their emotions. In initial condition the face identifier is turned off due to it's high need of computing power. The face identifiers task is to recognize the faces—listed in the designated folder in the form of JPG files—on the frames.

The analysis returns a data package which is then registered in the arguments collection for further use. The original frame in the arguments is replaced with the new frame including the visualization of the analysis.

The *edit()* and *update()* functions are set up that way, that they enable the user to turn on the face identifier. If this happens, it re-initializes the instance with the new setup.

8.4 Saving the results to the database

As one of our requirements says, we have to save the analysis results into a database. This plugins purpose is to fulfill that requirement. We chose to implement this to support a classic MySQL database, but using the same principles, we can create duplicates of this plugin which can be modified to support different types of databases. To moderate the amount of data coming from the analysis, we are going to make an average on a predetermined number of data. This can be set by the user.

To start from the beginning, we are going to start with the initialization. In this function we set up the needed variables and the environment to work with.

name — As defined in the plugin template we set up the name of the plugin.

login — This variable refers to a path where we have a JSON file containing the login details like host, name, password and the database.

sessionTable — This string variable represents the name of the session table. The session table contains information about the details of the current session. The session is defined as one initialization of this plugin. The structure of the table is defined by the function *tableSession()*.

frameTable — Similar to the sessionTable variable, holds the name of the frameTable. The frameTables structure is defined by the function *tableFrame()*. It logs the details of the saved data connected to the correct session. Every data upload creates a new row in this table.

faceTable — Holds the name of the table handling the faces. As we discussed the frameTable logs the uploads, this table logs the data belonging to the faces connected to the correct frame log. The table structure is defined by the *tableFace()* function.

dataBuffer — As we mentioned, we are going to average some of the data. This list will keep the frames until they get processed and saved.

bufferSize — This number sets the limit when to process the elements kept in the dataBuffer. The default value is set to five, however it can be changed through user configuration.

currentStage — Keeps track of the status of the *dataBuffer*. Used to determine when do we reach the limit.

mode — The value of this Boolean variable defines in what mode will the averaging work. We have two modes. The first mode is the library mode where the averaging happens with identified faces, so we average the data using their identity to differentiate the data packages. The second mode can be used for single person video input, for example, video calls. We choose the closest face to the camera and only average its data. It can happen without identification.

When initializing the plugin, we set up a connection with the database, and we create the tables if they don't exist yet. The overwritten process function only checks what mode we are using and calls the appropriate function. As we mentioned, the *modeLibrary()* averages by identity while the *modeLargest()* only averages the closest face—largest in the frame.

We created a helper class *Person* to help us manipulate with the data. It contains getters, setters and parsing functions. It helps us to quickly manipulate the data or to get them in the correct form.

We implemented a function called *insert()*. The purpose of this function is to make the correct SQL requests to save our data. We are communicating through a universal database handler which we created to simplify the process and make it available for any plugin. We are going to present you that in the next section.

This extension will have an important data output, so we overwrote the *getData()* function defined in the pluginTemplate. This function will request the last log in the database belonging to this section, parse the information using the Person class and return the data in JSON format. This function is used in the application to get the needed information for the visualization and for the function to provide raw JSON data.

8.5 Universal database handler

We created a manager to provide easy accessibility to databases for any plugin. To initialize it, we need a JSON file with the login credentials. It reads the file and creates a connection with the server if the credentials are correct.

To connect to a database we have the function *connectDatabase()* which takes the name of the database as parameter. If the database exists, it sets up the connection to it. If the database does not exist yet it calls the *createDatabase()* function which creates the database and extends the login file with the name of the database. This serves the purpose that next time it can automatically connect to the right database.

We implemented an *execute()* function which handles SQL commands with or without parameters, with or without commit. In case we expect data from the query, the function returns the used cursor, so we are able to fetch the information.

8.6 Snapshot viewer

As an extra feature we added, is a plugin with a useful data line. The plugin enables the user to put it anywhere between the plugins and it will return a JPG snapshot of the current state of the processed frame.

Every time when the processing unit goes through this plugin and calls the process function, it saves the current frame to an instance variable. When the *getData()* function is called, it makes an image response with the stored frame.

It can be useful for further implementation, debugging or quality control between specific plugins. We are going to use it in our web application.

8.7 Saving data to CSV

This extra plugin gives the user the ability to export a larger amount of data straight from the pipeline. This module currently supports our face and emotion analyzer.

In initialization, it opens a default file which can be changed by the user. This file will be the output file containing the data. The processing function gets called in every iteration which saves all the information regarding the analysis paired with a timestamp into the file.

Chapter 9
Web application

We chose to build a Flask application to extend our core system with a user friendly setup. The goal of the web application is to simplify for the user the process of setting up a pipeline and to make the configuration options available. This way, anybody will be able to set up and configure a basic pipeline through a graphical interface without any programming knowledge.

We are going to create Flask routes to set up the required URLs for our application. Each URL will trigger a different function or functionality. Currently, this application is going to be our main launcher, so we set up all the managers here, so we would be able to reach them in the functions fulfilling the URL requests.

Thanks to Jinja2 template engine we can create HTML template files, which we can render in our functions via the *render_template()* function. We created a base template which contains the styles and the header—menu—for our website with an open body which is going to be filled with other templates.

Starting with the index—home—page which presents the application and provides basic information for the user. From here we can use the menu bar to navigate to other parts of the application. The four key pages include *camera, buffer, plugins* and *output*.

On the camera page we are able to add or delete a camera, or we can choose to modify it. All the added cameras are listed including their name and source (see the attachment A.1). Thus, we can easily keep track and modify them. If we hit modify at a camera it renders us a different template, where we are able to change the name and the source of the camera (see the attachment A.2). If something goes wrong, we are using flash messages to display the error message for the user. This method is used on all of our pages.

Under the buffers page we have all the active buffers listed (see the attachment A.3). We can configure our buffers here. We can modify their size limit and their mode (see the attachment A.4). The specified settings become active immediately. However, if we don't have any active streams yet, on this page we will get a reminder message to start a stream.

Under the plugins section we can find our collection of plugin managers. If we don't have any, we can add a new any time. This page lists all the added managers with their collection of plugins for a better overview (see the attachment A.5). By hitting the modify button, we can edit them one by one. In the editor of the managers we can add new plugins, modify the existing ones, re-order them or use their special functions like the data line we created (see the attachment A.6). If the plugin does not support the data function, we are going to get a flash message informing us about it. If it does support it, it will redirect us to the collected data. By clicking on the modify button, we get to a customization page similar to the camera one. Here we have all the options listed that the plugin can offer for customization (see the attachment A.7). We can edit them and similar to the buffer the settings become active immediately.

The output page is where the whole stream gets put together. On this page, similar to the others, we can see all the active streams. To set up a stream we need to have a camera set up and a plugin manager created (see the attachment A.8). Here we can choose which one we want to use of the existing ones, select the proper combination and launch the stream. We are able to choose from the three modes we discussed before—live, save to file and process. We can see an extra field where we can specify extra attributes. These attributes are translated and sent into the stream, however we have to follow the form of a dictionary. We made it pre-filled it with the base setup for the file saver to display an example. After adding the process, it builds the stream of the selected components, and we will be able to see it in the actives list. Here we have the option to stop it or to see live data visualization. However, for data visualization, we have to have set up a functional database connection and it is recommended to have an active data stream from any type of analyzer.

Chapter 10
Testing and documentation

10.1 Testing

To test our implementation and our results, we performed unit, component and interactive tests to test all parts of our application. We can find all of our tests in the project folder under the folder */test*. We were using Pytest to automate all of our tests. It can be launched by entering the test folder with a terminal and executing the *pytest* command. We were trying to test most of the behaviors by writing appropriate tests with correct input, wrong input and edge-cases. By creating these tests, we managed to correct several bugs or malfunctions, and they pointed out some parts of the code which needed extra error handling.

However, for testing the interface we used interactive testing for which we wrote several test cases. These test cases can be found under the folder *test/testcases*. These test cases were performed by the author of this work.

10.2 Documentation

Through the process of implementation, we paid attention to writing detailed comments. By extending them into the form of docstrings we are able to generate a documentation. We discussed in section 4.7 we can use Sphinx to collect the docstrings and string them together into a document format. We used Sphinx to generate an HTML website which ensures better navigation and transparency when we are trying to understand the structure.

The initial setup with the requirements and the installation guide can be found in the included README file. It lists all the items we need, contains an installation guide how to get the application working, how to launch the automated tests or to reach the documentation.

On the index page of the application we made a quick start guide to guide the user through the key elements and the foundations of the app. On this page you can find all the basic information about the project and links to other materials.

Chapter 11
Discussion

11.1 Accomplishments

In this chapter we are going to discuss the accomplishments, requirements and their solutions. Starting with the term framework and the pipeline. We designed and implemented a modular system with multiple individual components. These modules connected together in a line form our pipeline. These modules can be used separately or the pipeline can be extended by connecting more modules in the line. This pipeline is our core system.

One of the modules of the pipeline is a buffering module which stores the frames until they get processed. This way we prevent dropping valuable frames.

To fulfill the extensibility and the integration of the analyzer package we created a processor component based on the idea of the plugin architecture. This module loads and works with all the plugins which are created by inheriting the structure of our base plugin template. To integrate the analyzer package we created a new launcher for it in a form of a plugin which can be loaded into our system. We extended this with an other plugin, which takes the output data from the analyzer and uploads it into a database. Both plugins became optional which allows us multiple combinations, for example we can have one analyzer while we upload the output data into multiple databases.

To achieve the needed configurability at every module we implemented functions to reach most of the key attributes and configurations inside the modules. These are useful at creating superstructures with the framework. The base configurations are saved into JSON configurable files, this way the user can create configuration presets which can be later loaded.

We implemented a web application by using our framework. This way we created an interface which provides an easy setup and configuration platform for everybody, even for the users without any coding knowledge. For visualization the application has a feature which can display the data if there is an active database connection for the selected stream.

We implemented open URL links which return data in raw JSON format to meet the requirements, furthermore it can be immediately used for integration or to connect other applications.

We documented the code via dosctrings, which we used to generate a HTML based documentation. Here the developers can find all the logic and information while it gives them the ability to search between the modules, functions.

11.2 Observations

An important factor, that should be mentioned, is the frame capture rate when reading from a video file source. Our frame getter thread repeatedly requests the frames until the source can

provide information. At a video camera the frame rate is set by the camera itself, however in a video file we already have all the frames waiting to be read. Since our getter repeatedly asks for frames, the source can provide them at the same time, until the file ends. If the processing time is really low, or we have no processing plugins enabled, the frames just flow through the pipeline which causes an accelerated video playback.

Chapter 12
Conclusion

In this project we successfully created a configurable framework for video analysis fulfilling all the requirements. By implementing a plugin architecture, we divided the system into two parts. The elements of the core system line up into a pipeline while the manager based on the plugin architecture provides us the needed extensibility. The core system creates a stream starting from the source from where we get the frames until the end where the frames are saved or displayed. Our pipeline contains a module which takes care about buffering the frames. Therefore, the processing units—plugins—are able to process for a longer time period without losing frames from the stream.

The integration of the face and emotion identification package happens through the plugin system. We set up a new launcher which uses the core modules from the given package. The output data is then shared among the plugins, so we were able to create a separate plugin supporting our analyzer which only deals with saving the data to a database. Due to the advantages of the designed structure, we are able to add any extra plugin for processing.

However, the pipeline can be set up by writing a few lines of code, it's harder to configure, and we are not able to change the settings dynamically. So we implemented a web application where the user can set up and modify multiple different streams. It enables the user to change the settings of the pipeline even when the stream is live. The application makes all parts of the system transparent and easily accessible. Besides that, we implemented a few gates where the application returns raw data thereby it is ready to provide data to different applications for further integration.

We provide a full documentation in an HTML form generated with Sphinx. The documentation describes all the features, the use of the application and all the modules and their logic. The README file provides all the steps to get started, including all the requirements and initial setup.

Appendix A

Application screenshots

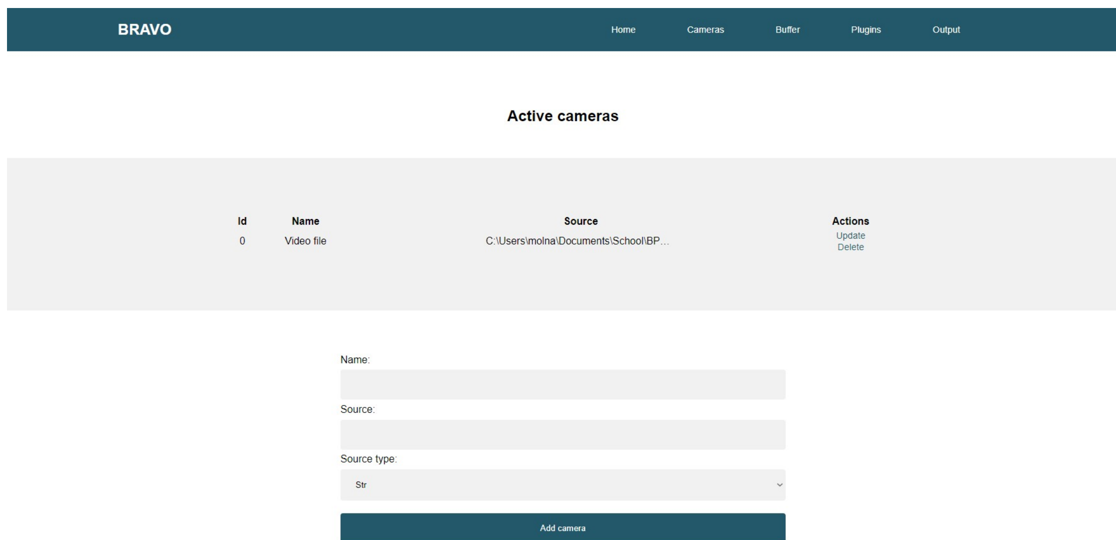
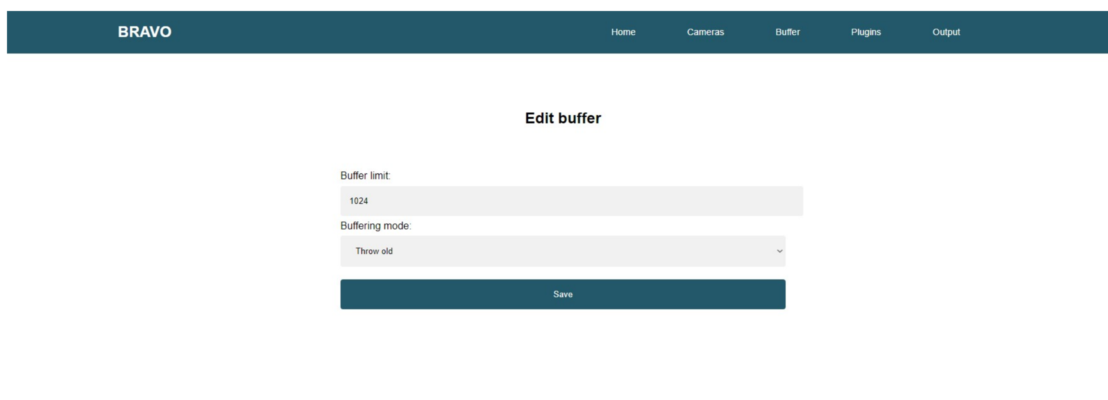


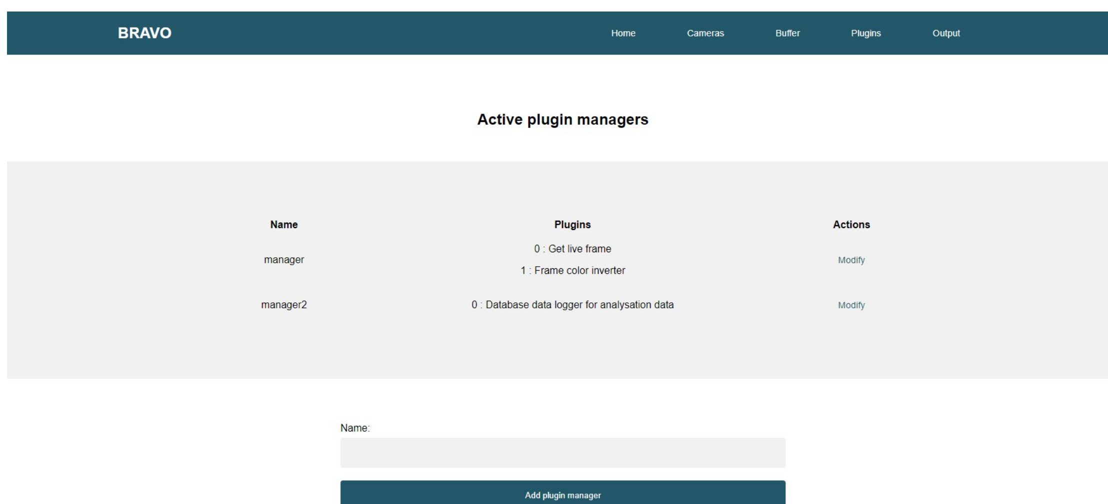
Figure A.1 Web application—Active cameras; Created by the author

■ **Figure A.2** Web application—Modifying camera settings; Created by the author

■ **Figure A.3** Web application—Active buffers; Created by the author



■ **Figure A.4** Web application—Modifying buffer settings; Created by the author



■ **Figure A.5** Web application—Active plugin managers; Created by the author

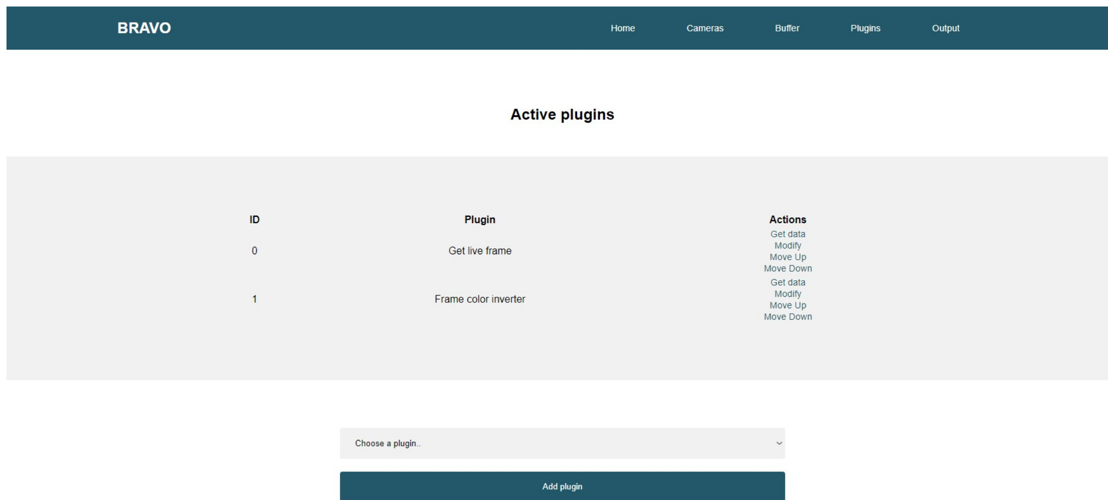


Figure A.6 Web application—Active plugins inside a manager; Created by the author

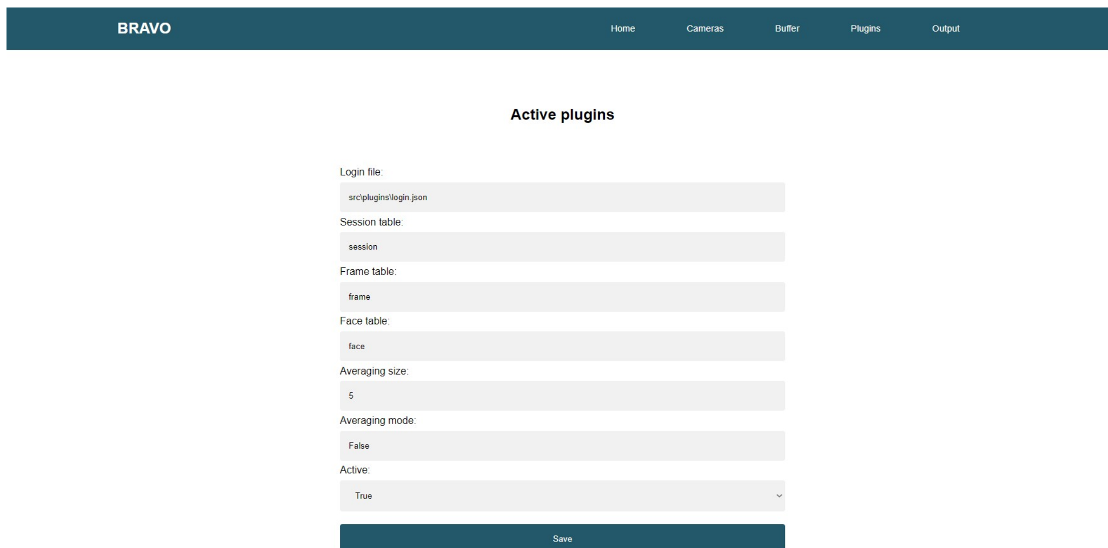


Figure A.7 Web application—Modifying the data saver plugin; Created by the author

BRAVO

Home Cameras Buffer Plugins Output

Active outputs

Source ID	Manager	Mode	Actions
0	Live view	manager	Live Stop

Camera:

Select a camera..

Plugin manager:

Select a plugin manager..

Output mode:

Live view

Arguments:

"filename":"src\\fileOutput\\video.avi","fps":24

Add process

Figure A.8 Web application—Active outputs; Created by the author

Bibliography

[1] Hydronaut Project [online]. [visited 7. 4. 2022]. Available from: https://hydronaut.eu/.

[2] ČVUT DSpace [online]. České vysoké učení technické v Praze, 2016. [visited 9.4.2022]. Available from: https://dspace.cvut.cz/handle/10467/95110.

[3] SINGH, Vijay. What is a Framework? [Definition] Types of Frameworks. In: *hackr.io* [online]. Venture Kite, 2022. [visited 12.4.2022]. Available from: https://hackr.io/blog/what-is-frameworks.

[4] CODEACADEMY, Team. What Is a Framework?. In: *codeacademy.com* [online]. Codecademy, 2021. [visited 12.4.2022]. Available from: https://www.codecademy.com/resources/blog/what-is-a-framework/.

[5] Ginni. What is Pipelining in Computer Architecture?. In: *tutorialspoint.com* [online]. Tutorials Point India Private Limited, 2021. [visited 13.4.2022]. Available from: https://www.tutorialspoint.com/what-is-pipelining-in-computer-architecture.

[6] STATICAPPS, Team. Defining Static Web Apps. In: *staticapps.org* [online]. STATICAPPS.ORG, 2014. [visited 14.4.2022]. Available from: https://www.staticapps.org/articles/defining-static-web-apps/.

[7] CAMPBELL, Steve. Flask vs Django: What's the Difference Between Flask & Django?. In: *guru99.com* [online]. GURU99, 2022. [visited 14.4.2022]. Available from: https://www.guru99.com/flask-vs-django.html.

[8] INTERVIEWBIT, Team. Flask Vs Django: Which Python Framework to Choose?. In: *interviewbit.com* [online]. InterviewBit, 2022. [visited 14.4.2022]. Available from: https://www.interviewbit.com/blog/flask-vs-django/.

[9] PROGRAMIZ, Team. Python Docstrings. In: *programiz.com* [online]. Parewa Labs, 2022. [visited 16.4.2022]. Available from: https://www.programiz.com/python-programming/docstrings.

[10] PANDAS, Team. pandas docstring guide. In: *pandas.pydata.org* [online]. The pandas development team, 2022. [visited 16.4.2022]. Available from: https://pandas.pydata.org/docs/development/contributing_docstring.html.

[11] READ THE DOCS, Team. Getting Started with Sphinx. In: *docs.readthedocs.io* [online]. Read the Docs, 2010. [visited 17.4.2022]. Available from: https://docs.readthedocs.io/en/stable/intro/getting-started-with-sphinx.html.

[12] ELGABRY, Omar. Plug-in Architecture. In: *medium.com* [online]. Medium, 2019. [visited 17.4.2022]. Available from: https://medium.com/omarelgabrys-blog/plug-in-architecture-dec207291800.

[13] OPENUP, Team. Concept: Use-Case Model. In: *utm.mx* [online]. OpenUP. [visited 17.4.2022]. Available from: https://www.utm.mx/ caff/doc/OpenUPWeb/openup/guidances/concepts/use_case_model_CD178AF9.html.

[14] SCALED AGILE, Team. Domain Modeling. In: *scaledagileframework.com* [online]. Scaled Agile, 2021. [visited 18.4.2022]. Available from: https://www.scaledagileframework.com/domain-modeling/.

[15] LUCIDCHART, Team. What is a class diagram in UML?. In: *lucidchart.com* [online]. Lucid Software Inc., 2022. [visited 18.4.2022]. Available from: https://www.lucidchart.com/pages/uml-class-diagram.

Contents of the enclosed media

The external package VizEmo is not included in the source files. In case of interest please contact Ing. Jan Hejda, Ph.D. by email: jan.hejda@cvut.cz.

The most recent source files of this project can be found at: `https://gitlab.fit.cvut.cz/molnaben/video-framework`

```
project ............................................ project source files
├── thesis.zip ........................... source files for this thesis in LaTeX
└── thesis.pdf .................................. this thesis in a form of PDF
```