A Framework for Pain Classification by Automatic Facial Expression Analysis

Project Report

Klaus Schneider

Supervisors:
Michael Siebers,
Prof. Dr. Ute Schmid

Otto-Friedrich University, Bamberg
Faculty Information Systems and Applied Computer Science
Cognitive Systems Group

January 8, 2013

Contents

List of Figures

List of Tables

1. Introduction

Automatic facial expression analysis has gained a lot of interest in the last couple years creating commercial potential in areas such as video conferencing, video telephony, video surveillance and visual speech synthesis [12]. Moreover it can be applied in the context of human-computer interfaces, human emotion analysis and talking heads [7].

An important application area is the automatic recognition and classification of pain using facial expression analysis. Until recently, pain was typically measured by patient self-report in interviews or using visual analog scales (VAS), where the patient indicates the perceived amount of pain on a horizontal scale. These techniques are popular and simple, but are only applicable when the patient can reliably state the level of pain he is feeling. This is not the case when dealing with children, patients with mental disorders and patients with an unsteady state of consciousness [1]. Classic facial pain recognition by skilled observers can be applied in the cases mentioned above, but are expensive, taking the required amount of manual work into account. Different recent studies such as the one by Lucey et. al. [10] deal with the topic of automating pain detection and classification.

In the past, several facial expression analysis frameworks have been proposed which provide the theoretic background for developing a system for pain classification [7]. However, to our best knowledge, a hands-on development framework, written in a popular programming language is still due to be released. In this paper we present a programming framework written in Java that enables researchers to develop an application for pain classification by facial expression analysis. The framework can handle recorded and live video data which will be split into single image frames and processed incrementally in interchangeable subsystems. These subsystems, namely a *Selector*, *Face Detection*, a *Classification System* and an *Aggregator* work together to provide a classified pain value. The clear separation of the subsystems enables researches to work on a single part of the program at a time and provides compatibility using clearly defined interfaces. Our framework is built for simplicity and extensibility using plugins as one of its major design patterns.

In the next section we describe the scientific background of both facial expression analysis and pain classification. Afterwards, we show the conceptual design of our framework and describe the function of each subsystem. In section 4 we describe the Software Engineering process we went through, beginning with the Requirement definition. We describe our High-Level Software Architecture, which will, together with the low-level design, show how the concept in section 3 was realized. Section 4.3 describes the low-level software design and implementation where we provide a description of the deployed third-party frameworks and libraries. Finally, we will give a conclusion of what we learned during this process and an outlook for further improvements of our framework. In the appendix we provide the manual of our framework for both end users and plugin developers.

2. Scientific Background

Facial expression analysis is not a new research topic. In an early work of Ekman and Friesen [6] six primary emotions that dictate a persons facial expression were identified: happiness, sadness, fear, disgust, surprise and anger. These so called *basic emotions* seem to be ubiquitous among human ethnicities and cultures. Later work shows that facial expressions are influenced by different sources, namely mental states, non-verbal communication, physiological activities and verbal communication (Figure 2). In this

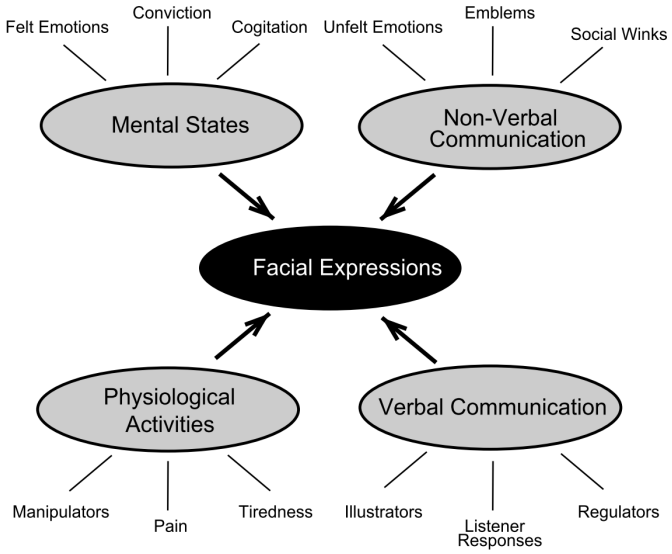

Figure 1: Sources of facial expressions [7]

paper we will concentrate on the classification of *pain*, which is depicted as one of the physiological activities that influence facial expressions. A relatively simple task is to rate the intensity of pain in a controlled environment where interference factors are minimized. However, a major difficulty with facial expression analysis is to differentiate one cause of facial expression, pain in our case, from other causes or anomalies in the video input.

Automatic facial analysis recognition has come a long way since 1978 when Suwa et al. [11] proposed a first investigation. Nowadays the computational power to perform such an analysis is inexpensive and publicly available. However, facial expression analysis is difficult due to the facial differences caused by different age, hair, ethnicity, facial hair, etc. Moreover changes in lightning and movement or rotation of the face can be a disturbing factor [7]. An example of an automatic facial analysis system is pictured in Figure 2. It features a face acquisition stage which includes face detection and face normalization, a stage for facial feature extraction and finally a stage for facial expression classification. These stages are explained in more detail when presenting our framework concept in section 3.

Pain can be classified in many different manners. It is common to differentiate pain according to its duration into acute and chronic pain. By convention, acute pain is defined

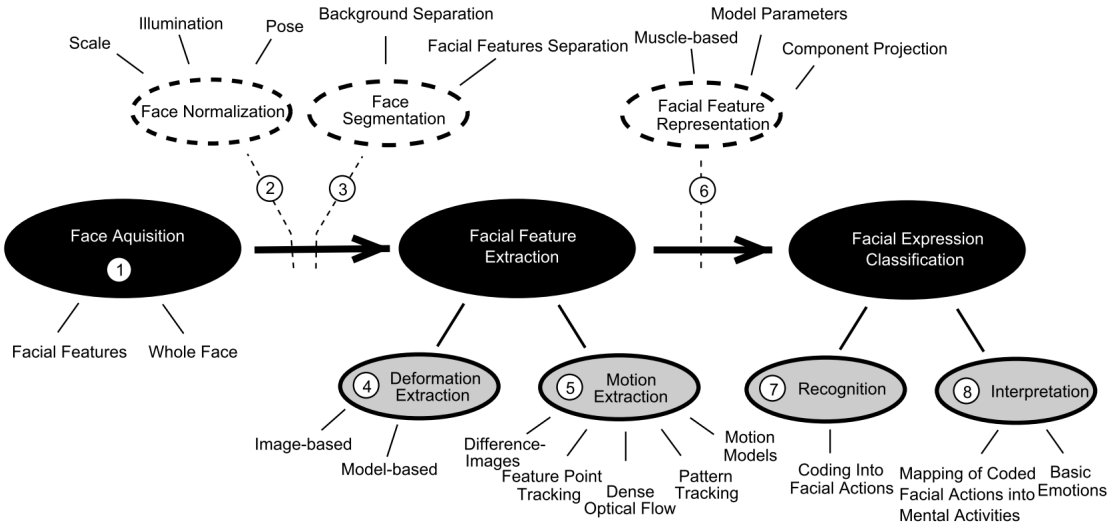

Figure 2: Generic facial analysis framework [7]

to last shorter than 30 days and chronic pain longer than 6 months [2]. Other classification criteria are the body location and the underlying cause of pain. The classification of pain intensity is commonly used, although problematic because the reported intensity is subjective and varies for most patients over the time [2]. It is uncertain how well an automatic classification system can estimate the previous characteristics. However automatically classifying the existence of pain, i.e. pain vs. no-pain, has been conducted successfully with low error rates in the past [1][10].

3. Conceptual Design of our Framework

We designed our framework with regard to the generic framework from Figure 2 and adjusted it to the additional requirements for live or recorded video input und pain classification output.

Due to the fact that facial expression analysis works best when analyzing facial features in motion, the input to our framework, which is depicted in Figure 3, has to be a video stream or a series of pictures. Classification of still pictures is problematic, because they do not reveal subtle changes and dynamics of facial expressions [7]. The first step of the

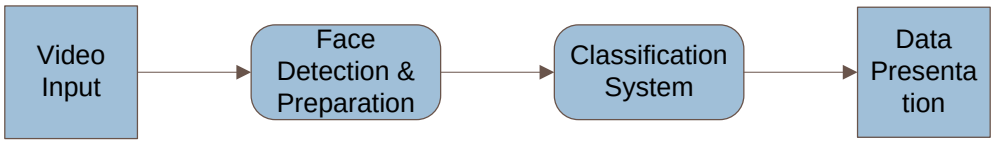

Figure 3: Framework concept

Face Detection & Preparation stage is to perform an automatic face detection. While this is relatively easy relatively easy in controlled settings, it can be problematic in real environments with unequal lighting conditions and crowded backgrounds. After the face is detected it must be prepared in order to filter out deviations that do not correlate to the characteristic that shall be classified, which is pain in our case. The pose, i.e. scale changes or rotations of the face, and illumination variations have been identified as two of these problematic appearance deviations [7].

The *Classification System* comprises the parts *Facial Feature Extraction* and *Facial Expression Classification* from the generic framework above. Various feature extraction methods have been identified and grouped by whether they act locally, i.e. focusing on parts of the face, or holistically processing the face as a whole [7]. Obviously the classification system is different and more specific compared to the generic framework, dealing with rating and categorizing pain instead of human emotions. Different characteristics of pain can be classified such as the stimulus that caused the pain, pain duration, intensity or location. The pain intensity as the probably the most important attribute will be described by a scale from 0 to 10, where 0 indicates no and 10 severe pain [13].

In section 4.3.2 we describe how the concept is realized in our software. The software design which is shown below in Figure 6 on page 12 additionally includes two selector and one aggregation component, which are merely detail optimizations. While they are omitted in the concept model shown above the aggregation component can be categorized inside the *Classification System* and both stages shown in Figure 3 are able to discard or select image data.

4. Software Development of our Framework

4.1. Requirements

Requirements specify the behavior of a software independent of the specific software design. They define *what* the software should do, instead of *how* it should do it.

The most important requirements for our framework are documented below in a format derived from the widely used volere requirement specification [8]. Requirements are grouped together by their type, which is either *Functional*, *Non Functional* or *Constraint*. For each requirement we provide a one sentence description, a rational which describes why the requirement is important and the priority on a scale from 1 to 10. A *fit criterion* is used to make the requirement more specific and measurable. This is used most often along with non-functional requirements. Table 1 shows the predefined language of the requirement specification.

Keyword	Meaning
is/are/has	Description of a given fact
shall/must	Description of a requirement
will/will not	Prediction about the future
shall not/must not	A requirement constraint. The following issue must be prohibited.

Table 1: Key words used in the requirements specification

4.1.1. Functional Requirements

1. **Requirement:** Using the framework it shall be possible to process live, real-time and recorded video data.
   Rationale: In some real settings, for instance in hospitals, live video processing is crucial while high delays or lags will probably be unacceptable. For research purposes recorded data is likely to be more common.
   Priority: 10

2. **Requirement:** The framework must provide an interchangeable component to perform face detection.
   Rationale: In order to perform pain classification based on facial expressions, facial features have to be detected on the current video frame.
   Priority: 10

3. **Requirement:** The framework must provide an interchangeable component to perform pain classification.
   Rationale: Pain classification is a key functionality of the framework.
   Priority: 10

4. **Requirement:** The framework must provide an interchangeable component to aggregate classified pain values.
   Rationale: The classification component from requirement 3 will possibly produce outliers or other problematic values. The *aggregator* component shall counter these phenomena and produce a more reliable result.
   Priority: 10

5. **Requirement:** The framework shall have a selector component for filtering certain frames.
   Rationale: This performance optimization is necessary to enable the framework to work on devices with limited hardware capabilities.
   Priority: 9

6. **Requirement:** The filtering of frames by the selector component from requirement 5 should be possible before and after face detection.
   Rationale: While filtering before the face detection stage will probably be faster, filtering after it might bring better results as more information is available at this point.
   Priority: 8

7. **Requirement:** During image processing, video information, e.g. a single video frame, must not be lost unintentionally.
   Rationale: In many scenarios the consistency between video input and result is important to eliminate indeterminate behavior. Performance problems will eventually occur which are countered with the inclusion of the selector component in requirement 5.
   Priority: 8

4.1.2. Non-Functional Requirements

8. **Requirement:** The framework and its source code shall be well documented.
   Fit criterion: The average developer shall be able to use und extend the framework with a training time of less than an hour.
   Rationale: Plugin developers and end users must be able to acquire a good understanding of the framework to simplify its usage and plugin development.
   Priority: 10

9. **Requirement:** The main functionality of the framework should be easily extensible.
   Fit criterion: Exchanging following components shall be possible without altering the source code: *Selector*, *Face Detector*, *Classificator*, *Aggregator*.
   Rationale: Plugin developers must be able work independently on each plugin. Moreover, they possibly will not have access to the source of the framework.
   Priority: 10

10. **Requirement:** The framework should support arbitrary video input sources.
    Fit criterion: Modification of the subsystem *Video Input Preparation* must be possible with access to the source code.
    Rationale: Changes in this underlying subsystem are expected to be much less frequent than the parts from requirement 9. So the benefits of an interchangeable architecture would not outweigh the implementation costs.
    Priority: 7

4.1.3. Constraints

11. **Requirement:** The framework software and related documentation shall be written in English.
    Rationale: English is the predominant academic language and one of the most widely used languages in the world [9].
    Priority: 7

12. **Requirement:** The framework shall run on different operation systems.
    Fit criterion: As minimal requirement it shall run on Windows and Linux.
    Rationale: Framework users and plugin developers will likely use different platforms and operating systems.
    Priority: 7

4.2. High-Level Software Architecture

The software architecture shows how the large subsystems of the framework work together. It must not be confused with (low-level) component architectures or design patterns. Choosing the right architecture model is crucial to make sure the software is deployable and scalable in its application domain. Changing the software architecture at a later development stage is expensive and it may not pay off to do so.

We choose a layered architecture for our framework, which is modeled after the generic layered architecture depicted in Figure 4 . The layered architecture promotes good structuring, changeability and reusability communicating only through well-defined interfaces between each layer. We decided, that a distributed architecture like the *client-server model* is unnecessary at this time. However, such a system could be implemented and only requires to change the *Data Presentation* subsystem, which is the highest layer of our framework. A layered architecture divides the application into different components with predefined dependency relations. Upper layers can use the functionality of lower layers through clearly defined interfaces which improves reusability and changeability of the software. Two types of the layered architecture can be differentiated. While in the *strict* layered architecture each layer can only use functionality from the layer directly below it, in the *loose* layered architecture layers may be skipped. Our framework architecture, which is shown in Figure 5, uses the latter type, which means that components in the highest layer may directly access the interface of the lowest.

Our lowest layer is the subsystem *Video Input Preparation*. It processes the video source material, which is either a recorded video file or live video data, splits it into single frames and provides a function for the upper layers to retrieve a single image frame. The *VideoFramework* component inside this subsystem can be exchanged and take arbitrary parameters, for instance the frame rate of the output frames, which can be independent of the frame rate of the source material, or the timeout how long the program should wait for the next frame. Moreover, the interface of this subsystem provides methods to

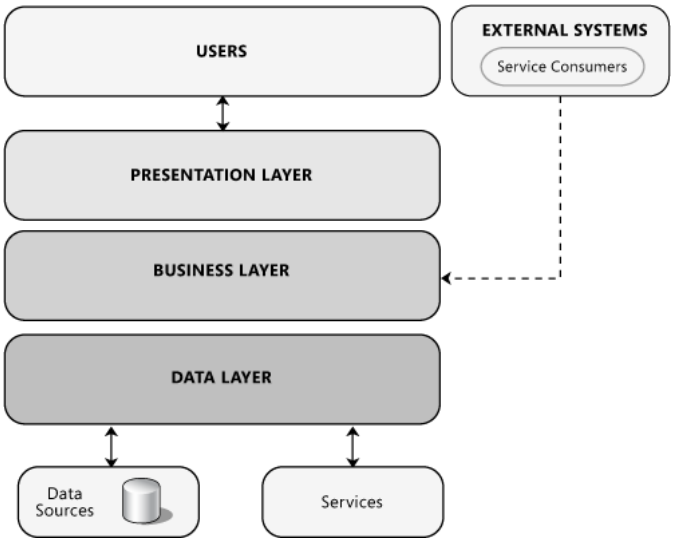

Figure 4: Generic layered software architecture [5]

register, select and retrieve information about a VideoFramework and to start or stop the input processing. We provide a default implementation of the *VideoFramework* to process pre-recorded video files based on Xuggler [4] which is described in detail in section 4.3.1.

The *Data Processing* layer provides the main functionality of our framework. Its interface holds methods to retrieve the final processed image and its pain classification and to register, select and set parameters of plugins. Moreover it allows to gather information about its plugin components and functionality to start and stop image processing. The subsystem is described in more detail below in section 4.3. Although the *Plugins* shown in Figure 5 are developed outside of the framework they are part of the subsystem DataProcessing. The details of the plugin architecture are described in section

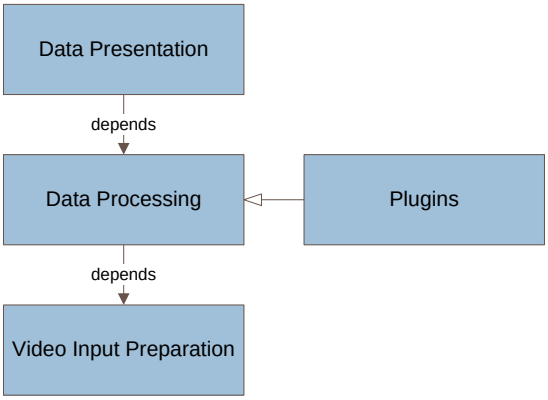

Figure 5: High-level software architecture: subsystems and dependencies

4.3. In section A.2 we provide a guide that shows developers how to write new plugins.

The *Data Presentation* subsystem contains the user interface to control the framework and to display the results in a user-friendly manner. We provide a simple GUI for testing purposes, which can be seen in Figure 9 on page 19.

4.3. Low-Level Design & Implementation

In this section we describe the components and details of the DataProcessing subsystem and provide information about the third party frameworks and software we used to implement our framework.

4.3.1. Video Input Component

As stated before, the *VideoFramework* inside the Video Input Preparation subsystem is interchangeable. We used the Xuggler library (version 5.4) for the default implementation, which allows video processing from files in multiple file formats. Since version 5.2, which was released in April 2012, it contains all native code to perform video decoding in a single jar file. So it can be used independent of the underlying operating system.

Our default *VideoFramework* takes the four parameters *path*, *timeout*, *outputFramerate* and *decodingFramerate*. The path equals the physical location of the video file on the file system and the timeout assigns a time in seconds after which video processing should be stopped. The two different frame rates are a bit difficult to understand. The outputFramerate is the frame rate the video should be converted into single frames. So every 1/outputFramerate second of the playing time of the video a new image frame is captured. The decodingFramerate sets the speed the file is read from the disk and processed. If the decodingFramerate matches the real frame rate of the video it will be processed in about real-time.

4.3.2. Implementation of the Subsystem DataProcessing

The data flow of the DataProcessing subsystem is show in Figure 6. While every plugin

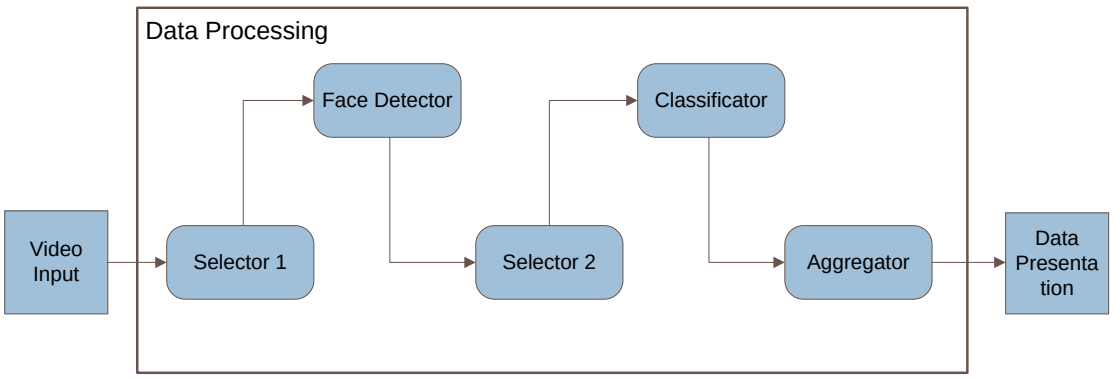

Figure 6: Software design of the subsystem *Data Processing*

has the same generic interface, which is pictured in Figure 7, it has a specific plugin type, which is either *SELECTOR*, *CLASSIFICATOR*, *FACEDETECTOR* or *AGGREGATOR*.

The method *processImage()* of each plugin takes a *ProcessedImage*, which is also shown in Figure 7, as input and gives a modified one as output. Furthermore, there is a method to set an arbitrary amount of generic parameters for a plugin and one to retrieve its description. Each plugin type modifies different attributes of the *ProcessedImage*.

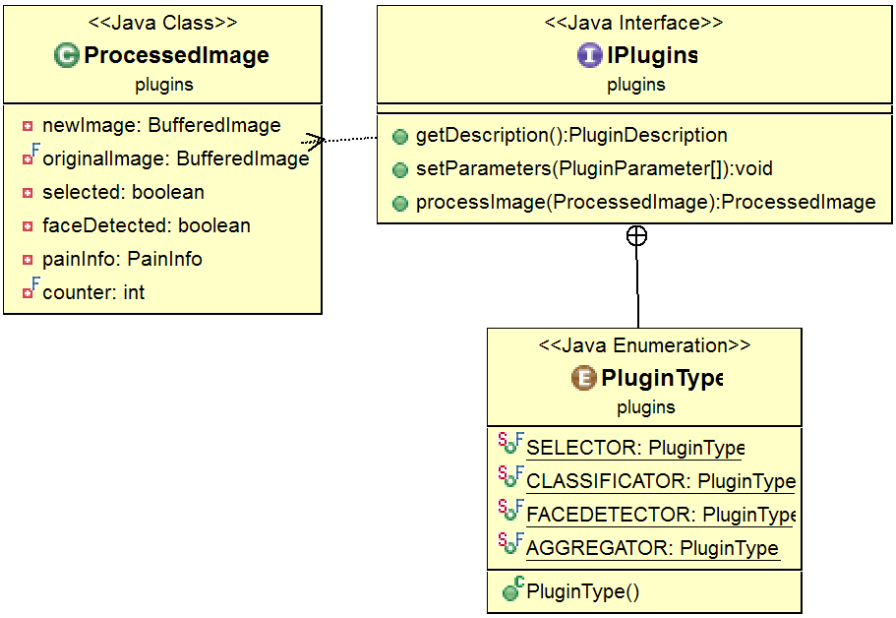

Figure 7: Plugin interface and class diagram of the class *ProcessedImage*

The two selector components can change the attribute `selected` to `true`, `false` or leave it unmodified. The components later in the chain (Face Detector, Classificator and Aggregator) are advised to ignore frames, which are not selected, i.e. have the attribute `selected` set to `false`. To have more flexibility, the end user can choose a different selector implementation for Selector1 and Selector2. We also provide a *DummySelector* which leaves the *ProcessedImage* as it is and a *SimpleSelector* which selects one frame out of n, where n is a given parameter.

The face detector component modifies the attribute `faceDetected` of the *ProcessedImage* and is also able to change the attribute `newImage`. This can be used to eliminate unwanted image parts and through that improve the pain classification results. The original image is preserved, so that it can be displayed in the GUI for reference purposes.

The classificator component changes the content of the *PainInfo* object, which is shown in Figure 8. The pre-defined attributes describe the intensity, probability, type and duration of pain. The pain intensity should be a double value between 0 and 10 as discussed before. So the constant *MAX_PAIN_VALUE* is set to 10 by default. The aggregator component can modify the *PainInfo* attributes to improve the result given by the classificator. For instance, it can eliminate outliers or invalid values.

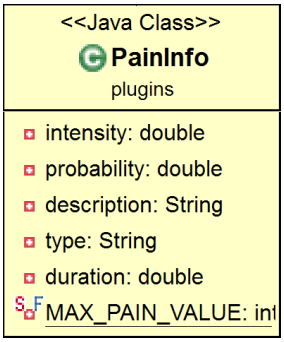

Figure 8: Class diagram of the class *Paininfo*

4.3.3. Plugin Framework

In order to make parts of the framework interchangeable and extensible we used the plugin design pattern. This pattern allows plugins to extend the final framework without the need to recompile it, loading new code at start or even at runtime. In Java multiple third party frameworks exist to implement a plugin pattern, which differ in their functionality and deployment effort. Another possibility would be to write an own plugin framework.

We decided to use the *Java Simple Plugin Framework* (JSPF) [3] in version 1.02, because it provides all required features and is designed for maximum simplicity. Other frameworks that we considered, such as *Open Services Gateway initiative* (OSGI) and *Java Plugin Framework* (JPF), are a lot more heavyweight and better suited for larger applications. JSPF works by implementing clearly defined interfaces and uses Java annotations to mark and describe plugins. It makes writing and deploying plugins very simple as described in section A.2.

5. Conclusion and Outlook

In this paper we have proposed a framework for pain classification by analyzing facial expressions which researchers can use to simply implement and test algorithms to perform tasks such as face detection, face normalization an pain classification. We developed our framework based on a generic framework proposition and used proven methods and principles of software engineering to design and implement it. Moreover, we provide a working prototype with a simple user interface and user manual.

The framework is built for maximum extensibility and presently has only limited functionality. Further work is necessary in the following three parts:

1. **The *VideoFramework* in the subsystem *Video Input Preparation***
   The current *VideoFramework* which is based on Xuggler can only process recorded video data. Although it does that in nearly every video file format, a framework to process live video would be a good addition. This would make our framework utilizable for many interesting real-time applications.

2. **The Plugins**
   The four different plugin components *SELECTOR*, *FACEDETECTOR*, *CLASSIFICATOR* and *AGGREGATOR* provide the main functionality of the framework. Until now only very simple and dummy classes are provided as plugins so the main development should take place in this section. Especially the automatic pain classification is a rather new topic where further research may bring benefits to current systems.

3. **The User Interface**
   Lastly, until now, we have only provided a prototype GUI which provides the necessary functionality for plugin development and testing, but lacks in matters of design and usability. In order to use the framework in a production environment the GUI has to be redesigned. A simple command line user interface is also thinkable which is common with Unix operating systems and would make the framework usable in command line scripts.

Of course this first implementation of our framework is not perfect and only testing in real environments may expose flaws that can be corrected in future versions.

References

[1] Ahmed Bilal Ashraf, Simon Lucey, Jeffrey F. Cohn, Tsuhan Chen, Zara Ambadar, Kenneth M. Prkachin, and Patricia E. Solomon. The painful face - pain expression recognition using active appearance models. *Image and Vision Computing*, 27(12): 1788 – 1796, 2009. ISSN 0262-8856. doi: 10.1016/j.imavis.2009.05.007. URL http://www.sciencedirect.com/science/article/pii/S0262885609000985. Visual and multimodal analysis of human spontaneous behaviour.

[2] B. Eliot Cole, MD, MPA. Pain management: Classifying, understanding, and treating pain. June 2002. URL http://www.turner-white.com/pdf/hp_jun02_pain.pdf.

[3] Ralf Biedert. Java simple plugin framework. accessed January 2013. URL https://code.google.com/p/jspf/.

[4] LLC ConnectSolutions. Xuggler. accessed January 2013. URL http://www.xuggle.com/xuggler/.

[5] Microsoft Corporation. Ethnologue: Languages of the world, sixteenth edition. 2009. URL http://msdn.microsoft.com/en-us/library/ee658109.aspx.

[6] Paul Ekman and Wallace V. Friesen. Constants across cultures in the face and emotion. *Journal of Personality and Social Psychology*, 17(2):124–129, 1971. ISSN 1939-1315(Electronic);0022-3514(Print). doi: 10.1037/h0030377. URL http://psycnet.apa.org/journals/psp/17/2/124/.

[7] B. Fasel and Juergen Luettin. Automatic facial expression analysis: a survey. *Pattern Recognition*, 36(1):259 – 275, 2003. ISSN 0031-3203. doi: 10.1016/S0031-3203(02)00052-3. URL http://www.sciencedirect.com/science/article/pii/S0031320302000523.

[8] The Atlantic Systems Guild. Atomic requirements: where the rubber hits the road. URL http://www.volere.co.uk/pdf%20files/06%20Atomic%20Requirements.pdf.

[9] M. Paul (ed.) Lewis. Ethnologue: Languages of the world, sixteenth edition. *SIL International Publications*, 2009. URL http://www.ethnologue.com/ethno_docs/distribution.asp?by=size.

[10] P. Lucey, J.F. Cohn, I. Matthews, S. Lucey, S. Sridharan, J. Howlett, and K.M. Prkachin. Automatically detecting pain in video through facial action units. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 41(3):664 –674, June 2011. ISSN 1083-4419. doi: 10.1109/TSMCB.2010.2082525.

[11] N. Sugie M. Suwa and K. Fujimora. A preliminary note on pattern recognition of human emotional expression. *International Joint Conference on Pattern Recognition*, pages 408–410, 1978.

[12] Maja Pantic and Leon J. M. Rothkrantz. Machine understanding of facial expression of pain. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.9065&rep=rep1&type=pdf.

[13] National Institutes of Health Warren Grant Magnuson Clinical Center. Pain intensity instruments. July 2003. URL http://painconsortium.nih.gov/pain_scales/NumericRatingScale.pdf.

A. User Manual

A.1. Main Program

This part of the manual provides a description for the end user how to run and use the framework with our preliminary graphical user interface. A screen-shot of the user interface is shown in Figure 9.

The program will be delivered as a single zipped file, which contains everything you need to run it. After unzipping you will see following folder structure:

defaultplugins/ - pre-compiled sample plugins (SimpleClassificator and SimpleSelector)

doc/ - JavaDoc of the whole source code in HTML format

logs/ - The folder for the log entries

jars/ - Jar packages that must be used by your own plugins

plugins/ - The folder for jar packages of your own plugins

videos/ - Video files must (!) be in this folder

main.jar - The main program

After putting your own plugin jar files in the *plugins/* directory and video files in the videos/ directory, you can run the main program from the command line. Change the folder to the directory which contains the file *main.jar* and run the command:

```
java -jar main.jar
```

A.2. Writing a new Plugin

This part of the documentation is aimed at plugin developers. Together with the JavaDoc commentaries in the source code it shows them how to easily write new plugins using the framework.

To write a new Plugin one must take the following steps. To simplify the development the source code of two sample plugins is provided in the folder *defaultplugins/*.

1. Import the libraries `PluginJar.jar`, which contains all relevant classes of the framework and `jspf.core-1.0.3.jar` which contains the classes of the *Java Simple Plugin Framework (JSPF)*.
2. Create your plugin class and implement the methods of the `IPlugins` interface.
3. Set up a description of the plugin with its parameters. The name of the plugin in the description must match the plugins class name.

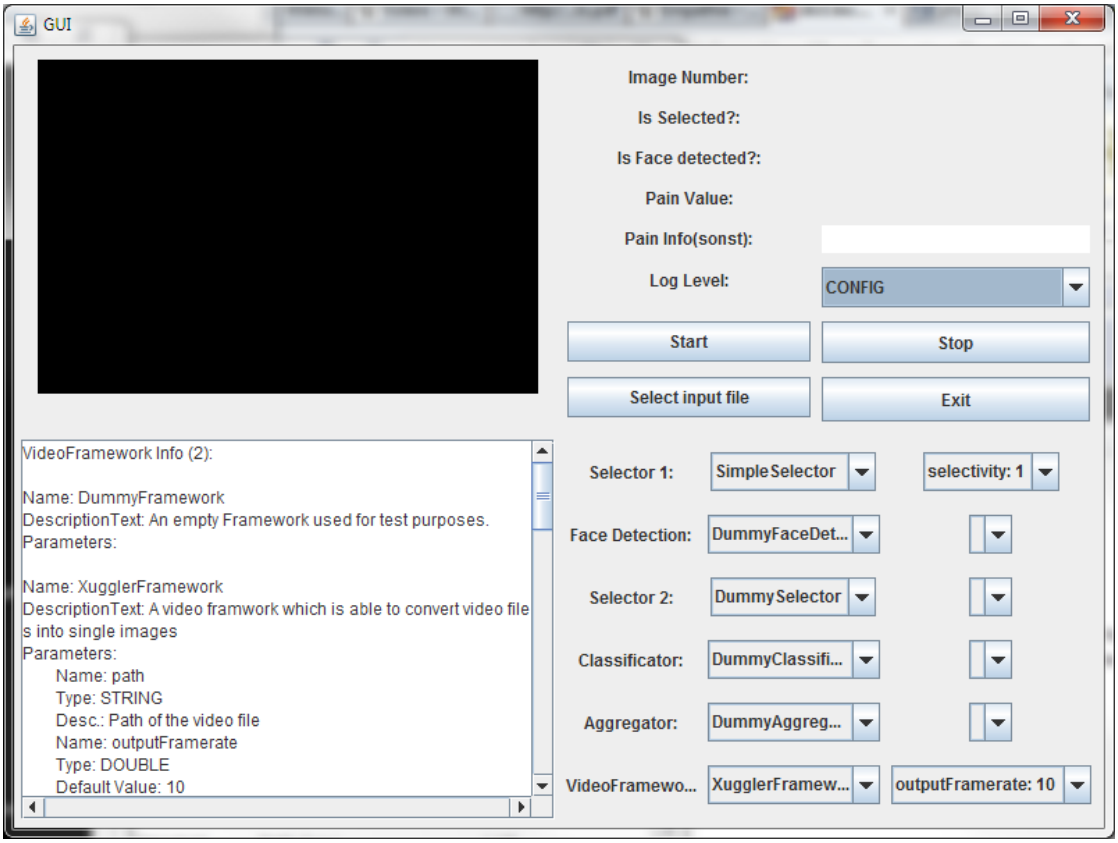

Figure 9: Preliminary graphical user interface

4. Add the annotation *@PluginImplementation* and optionally *@Author* right before the class definition.

5. Export the plugin as a Jar file (not runnable) and copy it the the *plugins/* folder of the main program.

Now the main program should find your plugin and display it in its GUI.