Building an Application Framework for EMG Data Acquisition

Martin Macheiner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Mobile Computing

in Hagenberg

im Juni 2018

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 25, 2018

Martin Macheiner

Contents

Acknowledgements

Before starting with the thesis, I want to take some moments and thank and appreciate the people, who helped me writing this thesis.

I want to thank my family and my friends. I want to especially thank my mother, who helped me crafting and sewing the different textile prototypes throughout the thesis.

Furthermore I want to thank Markus Altenhuber, who participated as a project partner of the master project. He helped me a lot to gather new inputs for my master thesis.

I want to especially thank Anita Vogl and Michael Haller for their support of the Research & Development department located in Hagenberg. Their support of materials and knowledge helped me a lot to overcome a stalemate during my concept finding phase.

Moreover I want to thank FH-Prof. DI. Stephan Selinger. He supported me through the last three semesters and had always time for me, when my focus drifted away from the actual concept.

Thank you very much.

Abstract

The domain of sensing muscular activity steadily increasing since the 1960's. The popularity of electromyography (EMG), the way to sense muscular activity either invasive or non-invasive, has increased tremendously due to the reason of the ability of electromyographic controlled prostheses. However, data acquisition in this domain is restricted to few competitors. The thesis examines the important software qualities and software architecture of an EMG data acquisition system, in order to compete with commercial software products. It comprises an implementation of a reference system, which exhibits all required software qualities, namely portability, maintainability, modifiability, extensibility and performance.

The system is implemented as a framework, as this type of software incorporates a well structured and modifiable foundation. The name of the framework is entitled simply *EmgFramework*. *EmgFramework* is a data acquisition system exactly tailored to the needs of scientists and developers in the domain of electromyography (EMG) sensing. It provides a rich toolbox in order to sense a variety of use cases. The evaluation is primary based on evaluating system qualities, such as named as follows: Portability, Maintainability & Modifiability and Performance. In addition a concrete use case was evaluated to examine the functionality of the whole framework: The measurement of an isotonic and isometric exercise utilizing a set components of the framework.

The results proved a solid software architecture as the foundation of the framework. The system is highly portable (around 20% need reimplementation), has a stable and maintainable software core and a low memory footprint, either for the desktop application and the Android app. Drawbacks can be summarized with the rather rudimentary ecosystem, both for plugins of additional functionality and for producer devices, as the framework tries to utilize an open source approach for the hardware producer devices as well. The core system can perform only a rather restricted set of tasks, which could be augmented in a further point in time, as the system is built upon a modifiable and maintainable software core.

Kurzfassung

Das Forschungsgebiet der Muskelaktivitätenmessung wächst seit den 1960er-Jahren stetig. Der Aufschwung von Elektromyographie (EMG), sprich die Möglichkeit Muskelaktivität entweder invasiv oder nichtinvasiv zu messen, folgte in den letzten Jahren unter anderem durch die Fortschritte der Forschung im Bereich der elektromyographisch gesteuerten Prothesen. Jedoch beschränkt sich die Auswahl der dazugehörigen Datenerfassungs-Software auf ein ausgewähltes Feld von Herstellern. Die Masterarbeit befasst sich mit der Frage, welche Software-Architektur und welche Software-Qualitäten für ein kompetitives Datenerfassungssystem – im Vergleich zu kommerziellen Produkten – vonnöten sind. Die Arbeit beinhaltet eine Implementierung des Refenzsystems, welches die erforderlichen Software-Qualitäten Portierbarkeit, Wartbarkeit, Modifizierbarkeit, Erweiterbarkeit und Leistungsfähigkeit aufweist.

Das System ist als Framework ausgelegt, da dieser Softwaretyp ein strukturiertes und erweiterbares Grundgerüst ermöglicht. Das System wird unter dem Namen *EmgFramework* geführt. *EmgFramework* ist ein Datenerfassungssystem, welches exakt auf die Bedürfnisse von Forschern und Entwicklern der Elektromyographieforschung zugeschnitten ist. Es umfasst eine Toolbox, welche verschiedene Anwendungsfälle abdeckt. Die Evaluierung fußt primär auf der Auswertung der Systemqualitäten durch zugehörige Metriken. Zusätzlich wird die Funktionalität anhand einer konkreten Fallstudie evaluiert: Die Durchführung einer isotonischen und isometrischen unter der Zuhilfenahme von Systemkomponenten.

TheDieResultate verweisen auf eine solide Software-Architektur als Grundgerüst des Frameworks. Das System ist höchst portabel (nur 20% der Software erfordern eine erneute Implementierung), hat einen stabilen und erweiterbaren Kern und weist eine geringe Speicherauslastung (jeweils Desktop und Android) auf. Zu den Nachteilen zählen das rudimentäre Ökosystem für funktionale Plugins und für *Producer*-Geräte, da das Framework den Open-Source-Ansatz auch auf die Hardware-Ebene anwendet. Das Kernsystem verfügt nur über einen begrenzten Satz von Funktionen, welcher aber über Plugins zu einem späteren Punkt erweitert werden kann.

Chapter 1

Introduction

1.1 Motivation

The world is a connected one. Everything is nowadays connected. Be it the smart thermostat, or even the smart body scale. At IBM they call it already the age of data [31]. IOT, namely the internet of things is one of the trends in recent years. Statistics estimate the number of IOT-enabled devices to 75 billion devices in 2025 (compared to 15 billion in 2015) [68]. This trend enhances the opportunities to nearly every topic and field of research thinkable. The possibility to connect devices via the internet turns every device into a data source or a data sink. Big companies like Google, Microsoft and ARM try to lower the entry barrier by providing operating systems perfectly adjusted for IOT use cases [25]. In the simplest form an IOT device is summarized as a single sensor – or sensing platform with optional actors – with the capability to talk to a remote server. Remote data acquisition systems basically operate on the same principle, but are often powered by proprietary software. Most new projects have to reinvent the wheel and most likely have to implement a custom remote data acquisition system which is best suited to their needs.

One concrete use case of such a system is EMG (Electromyography) measurement. Electromyography can be simplified expressed as the measurement of muscular activity. Muscular activity data evaluation is an active research topic. Pattern recognition for mechanical prostheses or muscle fatigue detection are just two use cases of what is actually possible with EMG data. EMG data recording under laboratory environment can be achieved easily. Noise reduction can be applied to an already known environment. Data can be transmitted via cables or specialized devices and data evaluation can take place on a desktop computer with tailored analysis applications or well-known frameworks (Matlab, LABVIEW). In the domain of research this is a perfectly fine approach. But such systems require a predefined environment to work in. They are inflexible to transfer medium changes and depend on the vendor specific remote sensing devices. They cannot easily adapt to a new environment. The reference implementation *EmgFramework* covers all of the mentioned shortcomings of a remote data acquisition system. It combines the power of IOT-enabled recording devices with the ability to be fully customizable for each EMG data related use case.

1.2 Problem Statement

As data is ubiquitous these days, data acquisition frameworks should be as well. Unfortunately data acquisition systems can have a broad range of different requirements and a "one-fits-all" approach is likely neither trivial nor even feasible. Therefore data acquisition systems focus on one topic (e.g. EMG) in particular. In the case of EMG measurement, those systems are built to measure muscular activity with a domain specific sampling frequency. Despite the fact, that there are already some well established data acquisition systems available, a flexible and customizable open-source solution is not available. Most comprehensive systems utilize their own hardware, which can only operate with the corresponding controlling software. Due to this fact, professional EMG data acquisition and evaluation is only possible at rather high costs. One of the leader in EMG data acquisition software is Delsys with the software solution *EMGworks* [20]. The closed ecosystem provides some undeniable benefits, but it is not possible to build a low cost measurement system, because of the licensing costs. On the other hand a customer receives a system where sensor devices and host devices work seamlessly together. Delsys provides a rich software platform for their devices. Generally speaking a highly extensible, flexible and customizable system is preferred. A system, which can be tailored to the expected needs. Therefore the goal is to implement an open-source data acquisition framework for EMG data measurement. An open-source solution would eradicate such shortcomings. Additionally an open-source software solution can incorporate open standards to communicate to any kind of sensing device. This would circumvent vendor lock-ins, as a sensing device does not necessarily need to communicate with a endpoint from the same vendor.

1.3 Outline

EMG data acquisition is a domain specific problem. Therefore the opening chapter starts with an introduction to EMG measurement, which describes the features of EMG data and the crucial points of sensing EMG data. The first chapter also lists the use cases for EMG data acquisition, to briefly summarize why a solid and clean EMG measurement system matters. The outline of some well-known problems should indicate which problems will affect the built measurement software and which also affect all the other EMG measurement systems. The chapter Related Work does not solely cover acquisition systems which work with EMG data. More generic approaches and some IOT-enabled approaches are covered as well. This knowledge funnels into the problems and requirements of an EMG data acquisition system. The chapter describes the shortcomings of related projects in terms of software quality and defines the conceptual and architectural foundation of the data acquisition system. The driving software qualities are outlined here explicitly. The concept and the prototype is explained in chapter 5. The final evaluation is followed by a case study examining the applicability of the system. The conclusion chapter will revisit the made decisions, the utilized architecture and identify bottlenecks with annotations for further improvements. In the end there is a discussion about benefits and drawbacks of the built data acquisition system.

Chapter 2

Introduction to EMG Measurement

What is electromyography? Carlo J. De Luca was an expert in this field of research and described it with the following words: "Electromyography is the discipline that deals with the detection, analysis, and the use of the electrical signal that emanates from contracting muscles [15]." Simply spoken, contracting muscles emanate a measurable voltage. De Luca suggests to rather use the term *myoelectric (ME)* over *Electromyography (EMG)*, although EMG is still the dominant term in nowadays literature. De Lucas statement concludes that an EMG signal is a continuous, analog signal.

2.1 Introduction to Electromyography

The human body is a complex organism. How EMG signals are composed inside the muscle is only explained on a rather abstract level. EMG measurement and analysis is domain specific. Therefore a basic knowledge about the topic is essential. This chapter only scratches the surface of the physiology of a human muscle, but facilitates an understanding how a human body emanates such a measurable metric.

2.1.1 Electromyography

Electromyography can be measured, when a muscle contraction increases and therefore more muscle fibers are activated, which leads to an increase of the signal amplitude [15]. The signal is the manifestation of neuromuscular activation of the contracting muscle and represents the ionic flow across the membrane of the muscle fibers. Muscle fibers themselves are activated in a group called motor unit. On muscle activation they generate the so called *Motor Unit Action Potential*, which is the fundamental part of an EMG signal. The signal itself is composed by a varying number of the mentioned *Motor Unit Action Potentials*, abbreviated *MUAPs* [15, 41]. A detailed view of a human muscle is outlined in figure 2.1.

It is assumed that the Greeks were the first who used electric shocks from electric eels as a kind of therapy. Francesco Redi was the first who refined this approach in 1666 and over a century later in 1791 Luigi Galvani proved the work of Redi. In 1849 Reymond DuBois proved that it is possible to detect EMG signals from the human muscle. Until the 1960s EMG measurement didn't really perceive much interest. During the 1960s the introduction of wire electrodes for intramuscular EMG and the possibility of controlling

externally powered prostheses, EMG gained attraction and the research accelerated [15]. There are two ways how an EMG signal can be measured, which will be explained in

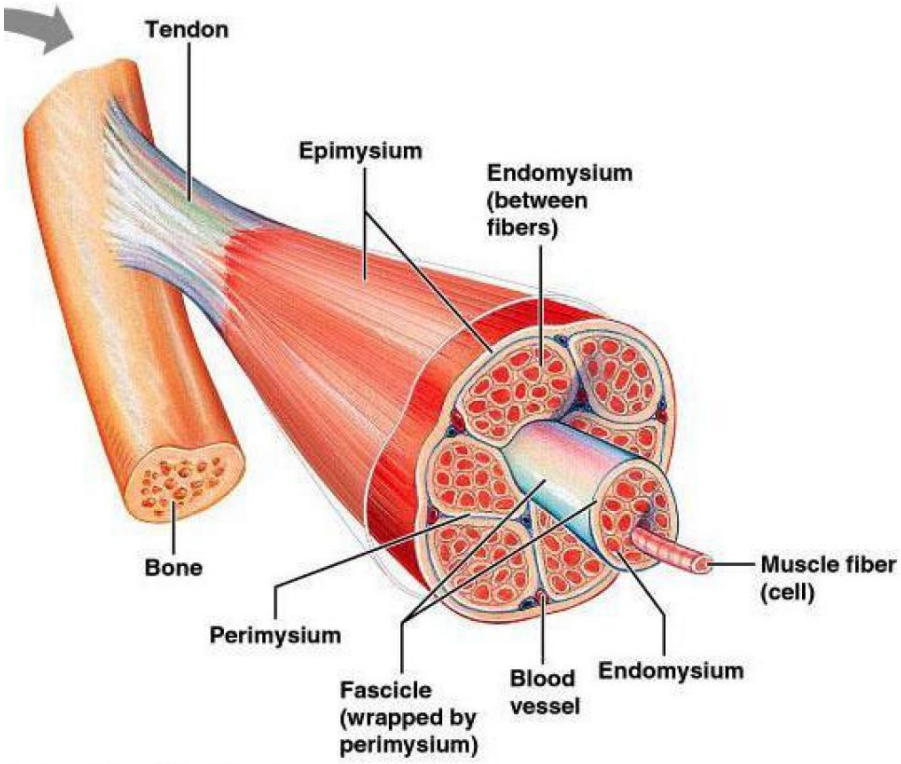


Figure 2.1: Human muscle explanation.
Source: www.anatomybodysystem.com/anatomy-of-skeletal-muscle-fiber

detail in section 2.1.2. Invasive measurement is achieved by introducing wires or needles directly into the muscle. Noninvasive measurement is possible by mounting electrodes on the skin [22]. Usually surface EMG measurement (sEMG) is preferred, because of its non-invasiveness. It should be mentioned that both approaches provide different conditions for the measurement. For sEMG an increase of the amplitude is noticeable, but it is in a range between 0 - 10 mV (peak-to-peak) or 0 - 1.5 mV (rms). The usable energy of the signal is available in a spectrum between 0 - 500 Hz, with the most dominant energy lies within the range of 50 - 150 Hz. In comparison intramuscular EMG can sense up to 2000 Hz [16]. According to the Nyquist theorem it is assumed that electromyographic data is sampled with at least 1000 Hz.

An EMG measurement system is susceptible for two types of concern. The signal itself (while creation inside the muscle) is susceptible to physiological, anatomical and biochemical factors, which can affect the EMG signal [15]. For measurements the signal-to-noise ratio (SNR) is crucial, because of the small measuring amplitudes. A bad SNR can render the whole measurement invalid, because the interesting data is covered in noise. Signal distortion is the other type of concern. There are different kinds of noise and possibilities of distortion, but most of them can be easily eliminated in a laboratory

environment [16].

2.1.2 Surface EMG vs. Intramuscular EMG

It was already mentioned int 2.1.1 that there are two ways to sense the muscular activity. Intramuscular EMG measurement with wire electrodes was first introduced in the 1960s [15]. An alternative to the wire electrodes are the so-called *needle electrodes*. Both approaches are invasive. This means that the electrodes is mounted directly on the muscle of interest. Intramuscular electrodes have the advantage to exactly measure a specific muscle even under low contraction and it's easy to reposition the needle after insertion. Measuring EMG data with a needle or wire electrode must be performed under constant surveillance, because electrode must not displace itself [15]. A laboratory environment can provide the prerequisites fur such a measurement.

Surface EMG (sEMG) uses adhesive skin electrodes instead. It provides a noninvasive and therefore more flexible way of sensing a muscle. Due to the fact that the electrodes are mounted on the skin, it is not possible to granularly sense each individual muscle. Other muscles can interfere the signal. This problem is called *cross-talk* [21]. Despite the drawbacks, sEMG is popular, because skin electrodes can be easily applied within a few steps. Subjects are advised to abrade and shave their skin on the desired placement, in order to remove dead tissue [15].

The different approaches can be categorized for the following use cases:

Surface electrodes Kinesiological, neurophysiological, psychophysiological studies, interfacing a with external prosthesis.

Needle electrode MUAP characteristic and motor unit control property detection.

Wire electrode Kinesiological and neurophysical studies of deep muscles [15].

Due to the fact that surface EMG support a variety of use cases and because of its non-invasiveness, sEMG provides a reliable source of muscular activity measurement.

2.1.3 Surface Electrode Geometry

Surface EMG offers a fast and reliable way of measuring. But the design (or geometry) of the utilized electrode is important. The electrode dictates the requirements of the recording software. Therefore the importance of electrodes should not be underestimated. One distinguishes between two types of electrodes: Passive and active electrodes. Passive electrodes only consist of a conductive, metallic detection surface. They directly sense the current on the skin. Active electrodes contain a high input impedance amplifier directly in the housing of the electrode. They are sometimes called "dry" electrode, because they do not necessarily need a conductive medium, but are prone to noise. Active electrodes are promising, but they are not fully established in the research field [15].

Electrodes are a critical part of the whole measurement system, because once noise

is introduced, it is almost impossible to improve data quality after the signal detection of the electrode [16]. A differential amplifier can remove relative nearby power line noise from the signal[1]. The amplifier subtracts two signals – where it is assumed that both are exposed to the same source of noise – and amplifies the difference. Common signals, such as noise will be removed [16]. The schematic of an active electrode is pictured in

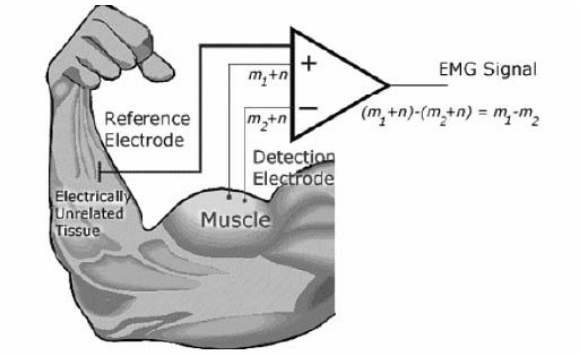


Figure 2.2: Active electrode - differential amplification.
Source: www.openprt.in/raw-emg-acquisition

figure 2.2. The reference electrode should be mounted on electrically unrelated tissue. As a rule of thumb it should be placed as far away as possible in the ideal case over an outstanding bone. The reference electrode must provide excellent contact with the skin and has a dimension of 2x2 cm. A good placement can decrease power line noise [16]. Depending on the muscle of interest, some places for reference electrodes can be: tail bone, pelvic bone, knee or elbow. If bony surface areas are too far away, it is acceptable to place it on a non-contracting muscle tissue [16].

The geometry of an electrode can highly influence the signal-to-noise ratio (SNR). The company Delsys [16] suggested to use a pair of pure silver electrodes with a dimension of 10x1 mm, placed 10 mm apart of each other. Noise reduction is performed due to the large surface areas.

Intelligent electrode design also impacts the bandwidth. Bandwidth is affected by the inter-detection surface spacing, the distance between the two electrodes. Usually the bandwidth lies between 0 - 500 Hz.

Every surface electrode is susceptible to cross-talk. Bigger dimensions of the electrode and bigger inter-detection spacing make the measurement more prone to cross-talk. Therefore, larger electrodes do not provide a better sensing performance, as they might introduce additional noise from cross-talking muscles [16].

2.2 Feature Extraction

Before algorithms can perform calculations based on the data, the raw signal must be converted into a usable format for algorithms to work with. This process is called *feature extraction*. Two types of features are known: Time and frequency domain features. Time-

[1]The different forms of noise will be explained in section 2.4.1

Frequency features can be seen as a third form of feature type, but as the name supposes it combines both domains.

2.2.1 Time Domain Features

Time domain features can be extracted directly from the captured EMG signal. Although they are easy to calculate, research proves that frequency domain features are considered more powerful than time domain features alone [71]. The most common fea-

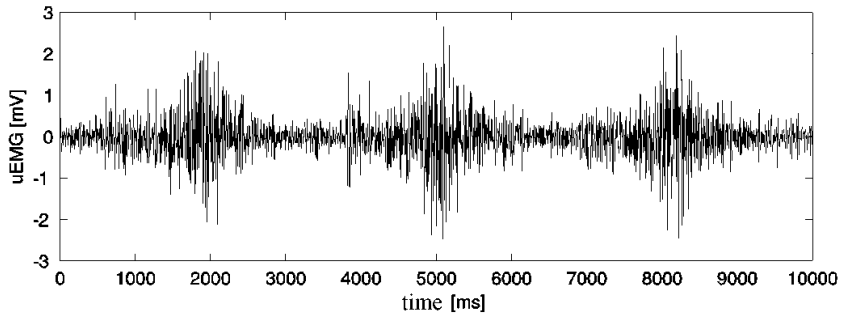


Figure 2.3: EMG signal over time.
Source: www.forum.allaboutcircuits.com/threads/emg-signal-to-wav.117337

tures, which can be extracted from such a signal as in figure 2.3 are Integrated EMG (IEMG), Mean Absolute Value (MAV) and the Root Mean Square (RMS). IEMG was the preferred and commonly accepted way for processing EMG signals. More powerful devices paved the way for a convenient calculation of the root mean square. RMS has a clear physical meaning (the power of the signal) and therefore should be preferred over the IEMG feature [16].

Integrated EMG - IEMG

Integrated EMG is the simplest approach, by just summing up the absolute amplitude values. Albeit the simplicity, the metric can be used as an index to detect muscle activity [71]. It is formulated as

$$IEMG = \sum_{i=1}^{N} |x_i|. \tag{2.1}$$

Mean Absolute Value - MAV

MAV calculates the average of the absolute values of a time series with length i. It can be seen as a moving average calculation. With MAV it is possible to detect muscle contraction [71]. It is formulated as

$$MAV = \frac{1}{N} \sum_{i=1}^{N} |x_i|. \tag{2.2}$$

Root Mean Square - RMS

RMS is related to a constant force and non-fatiguing contraction [71]. It is formulated as

$$RMS = \sqrt{\frac{1}{N}\sum_{i=1}^{N} {x_i}^2}. \tag{2.3}$$

Although De Luca et al. stated that RMS is the preferred choice for time-domain features [16]. There are experiments conducted by Clancy et al., which proved that the MAV feature processing is equal or better in theory and experiment, than RMS feature processing as the maximum likelihood estimator of the EMG amplitude [10, 71].

Zero Crossing Rate - ZCR

ZCR is unlike the other time-domain features not directly derived from the amplitude. It is the number of times the signal crosses the y-axis during a given interval of time. The feature provides an estimation of frequency-domain properties [71].

2.2.2 Frequency Domain Features

The Fast Fourier Transform (FFT) is one way to conveniently view the captured data in the frequency domain. The frequency domain allows different approaches and calculations and does not depend on the amplitude of the signal, whereas IEMG, MAV and RMS are dependent on the amplitude.

Autoregressive Coefficients - AR

An autoregressive model uses autoregressive coefficients to calculate a sample as a linear combination of previous samples and a white noise error. Autoregressive coefficients are useful for pattern recognition tasks [71]. It is formulated as

$$x_n = -\sum_{i=1}^{p} a_i x_{n-i} + w_n. \tag{2.4}$$

The current sample is denoted as x_n, while p indicates the order of the model. w_n is the white noise error factor. Paiss et al. states that AR can be used to investigate muscular fatigue [64].

Mean & Median Frequency - MNF & MDF

Mean and Median Frequency are computed based on the power spectrum (P) of the EMG signal. P_j denotes the value of the power spectrum at frequency bin j, whereas f_j denotes the frequency of the spectrum at bin j [71]. Hence the formula is stated as

$$MNF = \frac{\sum_{j=1}^{M} f_j P_j}{\sum_{j=1}^{M} P_j}. \tag{2.5}$$

The mean frequency is the average frequency in the power spectrum. The median frequency is the frequency where the power spectrum is divided into two parts with the same amount of energy [71] and formulated as

$$MDF = \frac{1}{2}\sum_{j=1}^{M} P_j. \tag{2.6}$$

Modified Mean & Modified Median Frequency - MMNF & MMDF

The modified variant of mean and median frequency replaces the power spectrum P_j with the amplitude spectrum A_j. Otherwise the calculation is identical and formulated as

$$MMNF = \frac{\sum_{j=1}^{M} f_j A_j}{\sum_{j=1}^{M} A_j}, \tag{2.7}$$

respectively

$$MMDF = \frac{1}{2}\sum_{j=1}^{M} A_j. \tag{2.8}$$

Although the envelope of the amplitude spectrum and the power spectrum is similar, the amplitude spectrum provides unlike bigger amplitude values than the power spectrum, while having less variation than the power spectrum. These properties of the modified computation lead to more robust features [71].

The publication of Veer et al. [71] investigated Modified Mean and Modified Median frequencies as a robust and superior EMG signal feature. The results are discussed in section 3.7.

2.2.3 Joint Analysis of EMG Spectrum and Amplitude - JASA

The JASA approach combines both domains and includes the amplitude and the spectrum in the calculation. JASA considers only the use case of muscle fatigue (section 2.3.2). The feature distinguishes between muscle fatigue-induced or force-related changes in the EMG signal [9]. JASA distinguishes between four scenarios:

1. Muscle force increase - Amplitude increases, spectrum shifts right,
2. Muscle force decrease - Amplitude decreases, spectrum shifts left,
3. Muscle fatigue - Amplitude increases, spectrum shifts left,
4. Recovery from fatigue - Amplitude decreases, spectrum shifts right [9].

JASA uses either the *Mean Frequency* or the *Median Frequency* as the frequency-domain feature. For the time domain it uses either the *Root Mean Square* or *Electrical Activity* feature. Electrical Activity is derived from the rectified and low-pass filtered raw EMG signal[2] Both domain features are calculated for a short period of time. Luttmann et al. suggested a frame length between five to ten seconds [48].

[2]Some sensors already do on-board filtering and rectification. Devices like the MyoWare Muscle Sensor (http://www.advancertechnologies.com/p/myoware.html) provide an extra output for raw EMG data.

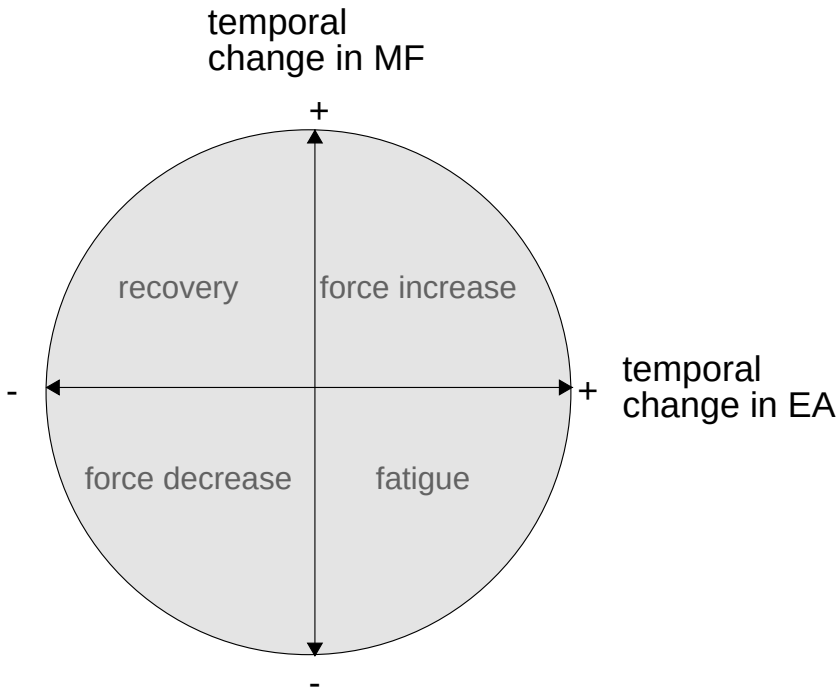


Figure 2.4: Joint Analysis of EMG Spectrum and Amplitude explanation.
Source: Schematic from the publication *Electromyographical indication of muscular fatigue in occupational field studies* [48].

JASA pseudo code

Listing 2.1 shows a pseudo code implementation of the JASA algorithm. A computation buffer of EMG data is necessary. The computation buffer for time and frequency domain features depends on the sampling frequency and on the sensing window width. The slope of regression lines is used as a quantitative indicator of temporal change. The buffer for regression values is limited to the required amount to map one minute. Regression lines are calculated using the least squared estimate. After receiving the slope value after a time span of one minute, the algorithm computes the temporal change to the last value. The pseudo code contains the possibility to plot computed temporal changes in time and frequency domain to an user interface as the last step.

Algorithm 2.1: JASA implementation.

```
Require: fs: Double, timeWindow: Double, data: DataSource
  tBuf ← zeros(fs × timeWindow × 2)
  fBuf ← zeros(fs × timeWindow)
  regressionTimeBuffer ← zeros((60/timeWindow) × 2)
  regressionFreqBuffer ← zeros(60/timeWindow)
  jasaBuffer ← List()
  tCounter ← fCounter ← regCounter ← 0
  while data.hasData do
    tBuf[tCounter] ← fBuf[fCounter] ← data.readValue()
    fCounter++; tCounter++
    if fCounter >= fBuf.size then
      f ← mdf(fBuf)                    ▷ Median Frequency in frequency domain
      regressionFreqBuffer.add(f)
      fBuf.clear()
      fCounter ← 0
    end if
    if tCounter >= tBuf.size then
      t ← rms(buffer)                  ▷ Root Mean Square in time domain
      regressionTimeBuffer.add(t)
      regCounter++
      if regCounter >= (60 / timeWindow) then
        tSlope ← Regression(regressionTimeBuffer).slope
        fSlope ← Regression(regressionFreqBuffer).slope
        jasaPoint ← computeTemporalChange(tSlope, fSlope, jasaBuffer)
        plotValues(jasaPoint)
        jasaBuffer.add(jasaPoint)
        regressionTimeBuffer ← regressionFreqBuffer ← regCounter ← 0
      end if
      tCounter ← 0
    end if
  end while
```

2.3 Use Cases

The question may arise why research in the domain of electromyography is even desirable. Applications and research fields span across neurophysiology, kinesiology, rehabilitation medicine and biomedical engineering [15]. Due to the abstractness of the fields, an array of use cases stress out the importance of electromyography in research.

2.3.1 EMG-controlled Prostheses

A human being controls its action with the mind. The activation of muscles is performed by a neurophysiological signal originating in the brain. The muscles act accordingly to the information they received within the activation signal. Therefore muscles are the

actors of the human mind. Amputees who lost a limb through an accident still have the mental ability to move the disembodied limb. The disembodied limb could be replaced by Research shows that it is possible to discriminate up to ten different forearm motions based on a two channel EMG signal [59]. Such approaches evolved over time, introducing machine learning to improve the discrimination rate. Chan et al. used several types of features, including amplitude, zero crossing rate, an autoregressive model and further frequency characteristics to train an artificial neural network, for the sake of a more fine grained prosthesis control [8].

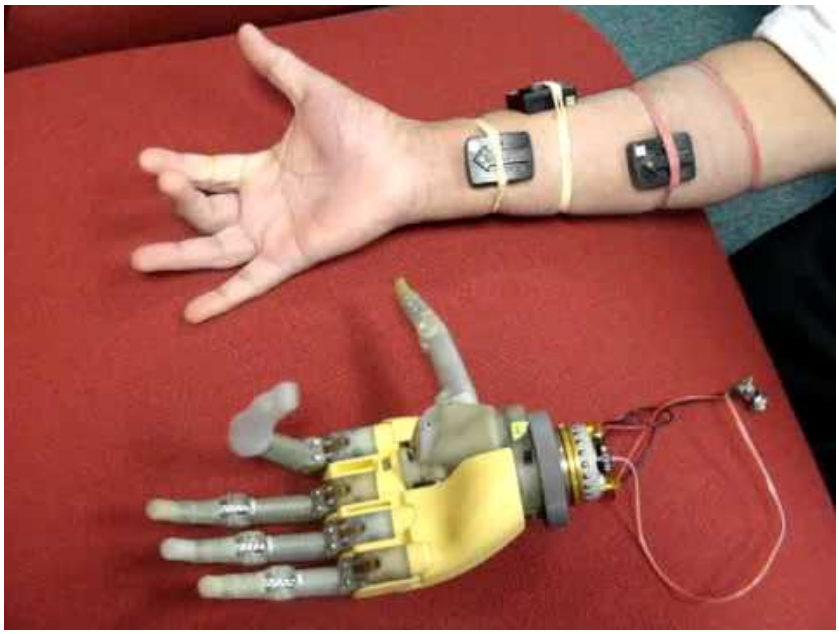


Figure 2.5: EMG-enabled prosthetic hand prototype.
Source: www.youtube.com/watch?v=j3RU1bHwLQc

2.3.2 Muscle Fatigue Detection

Sports and injuries typically go hand in hand. Muscular fatigue can lead to injuries, and therefore an indication of muscle fatigue could prevent injuries. A way to estimate muscle fatigue is to take a blood sample from the subject and determine the lactate threshold [9]. Clearly this approach can't be performed in real-time. Local muscle fatigue is reflected in the properties of myoelectric signals. sEMG measuring on the other hand can be performed in real-time [9]. Felici [22] conducted an experiment to observe gait deviations based on muscular fatigue and concluded that damaged muscles can be detected in the recorded EMG data. As a side notice he stated that sEMG data is in most cases insufficient as a single source of truth. Luttmann et al. [48] uses the JASA approach to determine muscle fatigue based on the EMG amplitude and its spectral properties.

2.3.3 Gait Analysis

In the past few decades an assessment of gait disabilities and functional gait disorder was a very complex measurement, including several types of sensors, and experiments were hardly repeatable [38]. Researching how a human actually walks led to the introduction of KEMG - kinesiological EMG. KEMG provides scientists an in-depth insight into the muscular contribution of functional gait disorders.

2.4 Well-known Problems

There are reoccurring problems when it comes to the measurement of EMG signals. A data acquisition framework requires high quality data, whereby data quality starts with the quality of the electrode and the recording environment. All of these problems can be addressed, when the data recording environment is set up carefully.

2.4.1 Noise

Noise can be the effect of various causes, may it be because of the skin-electrode interface, the environment or artificial sources like motion (movement) artifacts. A thoughtful test environment and well-considered filter can minimize the impact of noise [17]. Four categories of noise are present in an EMG signal.

Ambient noise A wide range of electromagnetic devices can cause ambient noise. Depending on the power line it affects whether the 50 Hz or the 60 Hz frequency. Filtering is advised, as the magnitude of the noise can be a multiple of the original EMG signal.

Electronic components Electronic components have a inherent noise, which is tightly coupled with its production quality. The noise ranges from 0 - several kHz and cannot be fully neutralized. High quality measurement products suppress inherent noise.

Motion artifacts A proper setup can reduce the impact of motion artifacts. Motion artifacts are caused by the detection surface or the wired connection between the electrode and the differential amplifier. Motion artifacts can highly impact data analysis, but it is rather simple to filter, as the most energy lies in the band between 0 - 20 Hz [22].

Inherent instable signal Due to physiological reasons, the EMG signal is in particular unstable between 0 - 20 Hz. It is advised to consider it as unwanted noise [16, 17].

2.4.2 Aliasing and Filtering

The Nyquist theorem states that the sampling frequency for any signal should be at least twice the highest frequency component of that signal. The highest frequency component for EMG signals depend on the used electrode technology. Wire electrodes have a range from 20 - 2000 Hz. Needle electrodes have even a broader range and can sense between 1000 - 10000 Hz. Surface electrodes used for sEMG measurement have a comparatively narrow range of 20 - 500 Hz [16].

Aliasing is the effect which occurs, if the Nyquist theorem is violated. Frequency components which have a higher frequency as the sampling frequency are "folded" back in the spectrum. Take a signal with a range of 0 - 50 Hz and a sampling frequency of only 90 Hz. The signal component of 50 Hz will be folded back to 40 Hz and falsify the frequency spectrum.

Sampling EMG signals with at least 1000 Hz is highly recommended [18] with an additional bandpass between 10 - 500 Hz [42]. De Luca et al. suggested even a higher boundary of 20 Hz [17]. If there are cases where the sampling frequency is smaller than 500 Hz, an aliasing filter is advised.

2.4.3 Cross-Talk

Cross-talk describes the problem of sensing muscular activity of adjacent muscles in addition to the actual muscle of interest [15]. Figure 2.6 gives a visual representation about the issue. Adjacent muscles are sensed and pollute the actual EMG signal. Cross-

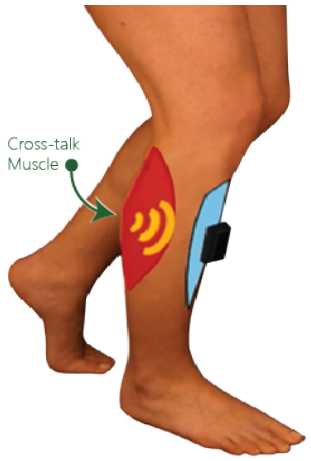


Figure 2.6: Cross talking muscle.
Source: www.delsys.com/products/software/emgworks/sqm/factors

talk only affects sEMG due to its large sensing surface, compared to intramuscular EMG. Usually it is not a big concern, as the electrode is ideally sensitive enough to only sense the nearby muscle [15].

In order to avoid or minimize cross-talk, intelligent electrode placement is necessary. The electrode should be placed between a motor point and a tendon with the detection surfaces in a right angle to the muscle fibers. This implies, that the electrode is mounted parallel to the muscle fibers. The electrode must not be placed on a motor point. The motor point is the point in the muscle with the greatest neural density. According to Luca et al. [16] this is the worst location, as minor displacements can cause unpredictable signal distortion.

Chapter 3

Related Work

The related solutions, applications and approaches do not exclusively handle EMG data acquisition systems. The span ranges from traditional EMG data acquisition systems like EMGworks, but also covers general-purpose data acquisition systems towards IOT-ready solutions.

3.1 EMGworks by Delsys

The commercial company Delsys is inevitable coupled with the personality of Carlo J. De Luca. De Luca was the founder of Delsys and president until his death in 2016. He was a coryphaeus in EMG signal research, with over 100 publications and more than 30000 citations [26, 32]. Unlike Motion Lab Systems (section 3.2), Delsys regularly publishes scientific research papers, offer access to their knowledge center and directly link papers on the webpage.

The company has solutions for both sEMG and dEMG, although dEMG is a technology on top of sEMG. De Luca and Kline introduced the term *dEMG* for decomposition EMG [41]. dEMG decomposes the sEMG signal into its individual firing instances of each motor unit [41]. The technology should help to investigate the connection between neural activity and the muscle control. It is also possible to measure muscle fatigue on a motor unit level [13].

Delsys provides a rich set of hardware platforms. It distinguishes between wireless modules and desktop modules. Wireless sensing devices can work up to 40 meters and provide an analog value between -5V to 5V. Software systems can sense up to 16 sensing devices simultaneously. Some sensors utilize an accelerometer output in addition to the analog EMG signal. The so-called *desktop system* uses sEMG sensors with a wired connection. Contact dimension and contact spacing adheres to Delsys' published sensor geometry specification [16]. Wired electrodes provide a better performance in terms of noise level and power consumption. Wireless sensing devices consume less than 6 Watt comparing to typical 20 mW power consumption of the wired equivalent. Noise is estimated with 1.2 µV RMS compared to 0.5 mV RMS of the wireless component [19].

Delsys provides their own software solution called *EMGworks*. Nevertheless it is possible to integrate the hardware into custom data acquisition systems. The process depends if an analog or digital system integration is targeted. Analog systems can directly

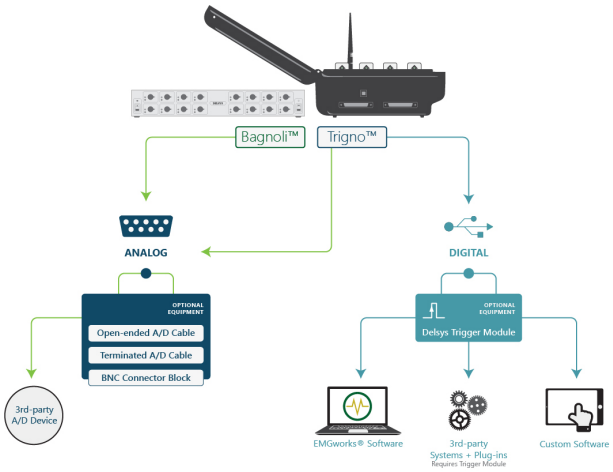


Figure 3.1: Delsys system integration.
Source: delsys.com/wp-content/uploads/2016/07/systems-integration_workflow.jpg

measure a voltage value from the wired electrodes, while wireless and digital signals can be processed via the software EMGworks or via custom software. Due to a Software Development Kit (SDK) it is possible to incorporate Delsys hardware products into custom data acquisition solutions with language bindings for MATLAB, LABVIEW, Java and C#. The SDK allows developers to pair wireless sensors, monitor data and configure devices. TCP/IP is used as the connection medium [19]. The commercial product EMG-

Figure 3.2: EMGworks by Delsys.
Source: delsys.com/wp-content/uploads/2016/01/emgworks-acq-01.png

works provides an out-of-the-box solution for high-quality EMG measurement systems. EMGworks has integrated EMG databases, anatomical maps of the human body, a visual trajectory editor, signal exploration features, multi-point calibration functionality and experiment workflows.

The signal monitor helps to identify and improve data quality, while streaming data in real-time with optional filtering. The anatomical map guides the sensor placement and allows adjustment of sensor parameters. Experiment workflows introduce custom data collection protocols, which can be assembled with a user-friendly drag-and-drop interface. The software ships with a vast array of comprehensive analysis methods [20].

Delsys provides one of the richest EMG measurement ecosystems. The SDK offers the capability for developers to integrate the sensor hardware components into a custom data acquisition system. Unfortunately the aspect of data analysis is neglected in this use case. Data analysis features are included in EMGworks. EMGworks provides a seamless acquisition and analysis experience for the licensing costs of 500$. Experiment workflows enable inexperienced users to build a tailored data acquisition task workflow by an intuitive user interface.

3.2 Motion Lab Systems

The company Motion Lab Systems, Inc. [69] located in the USA, offers commercial products, but unlike Delsys they do not publish scientific papers and do not actively contribute to research. It was founded back in 1987 and provides powerful EMG systems, which include software and hardware. The devices are used by scientists all over the world [69]. Motion Lab provides a palette of EMG measurement hardware. The hardware uses a bandwidth of 10 - 1000 Hz, even up to 2000 Hz. The devices can sense 12 - 28 channels simultaneously with a maximum signal delay of 2 ms at full bandwidth. They offer a dedicated preamplifier, which is basically an active electrode.

From the software perspective Motion Lab Systems lag behind the hardware branch. They offer software solutions, but they are limited to Windows only. They are available as a free evaluation version. In general the applications are mainly utility applications, which help to convert or manipulate domain specific file formats. The analysis software and the graphing software are divided into two applications, whereas the utilized analysis features are a superset of the graphing features. Next to the two main applications there is a dedicated real-time EMG data acquisition application. The applications are capable of processing C3D files, raw EMG data files and standardized ASCII files.

EMG Graphing Software The basic graphing software is capable of reading data, applying filters, reducing noise and storing data in the supported file formats. It offers the capability to access databases for EMG activity comparisons, whereby the database distinguishes between adults and children.

EMG Analysis Software The applications allows frequency domain inspection with FFT and power spectrum and time domain feature extraction with moving average, RMS calculation, Zero Crossing Rate and integrated EMG. EMG analysis includes all basic features of EMG Graphing.

Windaq - Real-time data acquisition Real-time sensing needs dedicated hard and software, but is capable of capturing 32 channels and store data directly to the disk. The software promises real-time data acquisition on every Windows host, while executed in a multitasking environment. Windaq is either available as a PRO or LITE version, whereby the LITE version can only acquire 16 channels in parallel and can only sample with 240 Hz. Next to these restrictions the LITE version is only capable of displaying data but cannot record it, which renders the use case as a data acquisition obsolete.

The software is quite powerful, but it requires dedicated hardware. The applications are divided, but real-time support could be bundled with the standard applications in order to provide a single data acquisition software solution. In general, information next to the website are rare [69]. The software is proprietary, which means it is not extensible and a license is necessary in order to fully use all the features.

3.3 EMS - Extensible Measurement System

Nogiec et al. outlined in 2001 a Java-based extensible measurement system (EMS), which aimed to replace the broad ecosystem of data acquisition systems. Nogiec et al. aimed to define one expandable base system, which was capable of conducting an array of domain-specific tests and furthermore be able to provide various algorithms and analysis methods. The system required to be portable, run-time reconfigurable and the admissibility of parallel signal processing. The architecture should allow a distributed usage later on [61].

The framework itself was component-based, whereas components can be of general purpose or domain specific behavior. General purpose components can be shared between different use cases. The framework should have enabled developers to built platform-independent applications on top of it. Communication between components was established using a software bus with a dedicated routing service. The router component provided a multitude of communication options. Well-known concepts like unicast, multicast and broadcast were supported. The router stored the addresses in an internal routing table for the address-based communication, with the alternative of content-based communication (publish-subscribe pattern). The core itself exposed a collection of core components and was mainly responsible for routing and messaging.

The components were basically just Java Beans[1] whereby so-called adapter objects connected the component to the software bus. The adapter object is thereby an interface, which provides the functionality for communication. Depending on the implementation, components could be either *producer* or *consumer*. Data was sent between components via the adapter to the router. The router component took care of the message delivery.

Components themselves were stateful, but only to a minimum degree. The defined states were unkown, init, running, stopped. The noteworthy detail here is that the state transitions were triggerable by the framework and from the internal state. A graphical user interface was optional for each component.

The system was configurable via XML-based configuration files. Configuration files could have been updated during run-time and then being persisted as an XML file. The whole system could be described only with configuration files. It was proposed that no programming was necessary.

EMS should have provided an alternative to vendor-specific implementations. Its architecture and shallow hierarchy enabled even inexperienced developers to add new components while keeping development costs low. The framework comprised a solid architecture, although omitted some essential system characteristics. There was no hint about the supported sampling frequencies and a maximum transmission rate. The system also lacks the capability of storing acquired data and did not mention supported

[1] http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

file types. Moreover the framework highlights its portability, which is basically justified by the use of Java as a platform independent programming language. While the paper was published in 2001, a platform independent approach using Java is a valid point, but nowadays with the utilization of mobile phones and the internet of things, the usage of the Java Swing framework for graphical user interfaces drastically imposes the portability of EMS. It was also stated that the system is potentially capable of being a distributed system. This implies that the framework did not offer the possibility to talk to remote hosts.

Due to this shortcomings Nogiec et al. extended and improved EMS by incorporating data stream processing [60]. The base system already provided a dynamic reconfiguration and a routing service, which was enhanced to a self-adjusting system. A Pipes & Filters pattern and a layered architecture were incorporated. The layered architecture allowed the separation between processing, presentation and persistency functionality.

The system relied on the same component principle as EMS, but with a more distinct allocation of roles. Data acquisition components were referred as a data event source, which collected data produced by other components. Dedicated data visualization or persistence components decoupled the system of its operating environment. Due to its Pipes & Filters architecture it was possible to seamlessly operate on the data stream [60].

3.4 COLDEX - New Data Acquisition Framework

Grech wrote a paper [28] about the renewal of the data acquisition framework at CERN during his 2 month summer studentship period. The project aimed to engage a new data acquisition architecture, as the old architecture inhibited maintainability and scalability. The old system mainly relied on a serial communication interface. The retrieval and storage of information for different experiments was not possible through other communication mediums. Grech's proposed architecture supported a serial communication, but used TCP/IP connections as the preferred communication medium. The framework was responsible for accessing PLC (programmable logic controller). The architecture must have considered an additional layer of CERN-specific middleware. The open-source library *libnodave*[2] was utilized as a novel networking packaged data exchange protocol for the Siemens PLCs [28]. Some PLCs required active polling in order to retrieve data.

LabVIEW was chosen as the development platform, as it was already used at CERN. The challenge of the new framework was the incorporation of a wide range of data acquisition modules and the corresponding driver libraries. The previous architecture did not follow any structured approach. The communication was tightly coupled via the serial connection interface. The new communication layout introduced a National Instruments PXIe-1082 Data Acquisition System as the *System Controller*, the main instance for acquiring data from an orchestra of connected devices. It is noticeable that the cheapest version of the data System Controller costs nowadays around 4000€. The main communication tool was RADE (Rapid Application Development Environment), which was also based on LabVIEW and originates in CERN.

CERN utilizes a vast array of accelerator controls. A middleware layer called *CERN*

[2] https://github.com/netdata/libnodave

controls Middleware (CMW) introduced a common communication abstraction layer. Clients can be accessed with the Middleware Remote Device Access (RDA). The system-wide sampling frequency was 66Hz. COLDEX was inter-operable with Java due to the Java API for Parameter Control. COLDEX highlighted parameters as the key concept, as every control value can be seen as a parameter [28]. Serial communication could not be abandoned, because of the legacy devices, which required a serial interface. Due to the short period of the project it was not possible to integrate it, but the architecture allowed a utilization of serial devices at any point in time.

COLDEX provided a mechanism for storing acquisition configurations. Therefore it was possible to store configuration data into an Excel file. LabVIEW provided its own implementation of the *Queued Message Handler* design pattern, on which the framework is built upon. The design pattern extends the producer-consumer pattern, where producer independently provide data and consumer process the received data at their own pace [28]. The data acquisition console utilized four parallel loops, each with a dedicated role. Communication between loops was possible via events and messages. The dedicated roles were namely:

Display The Display loop received messages from the DAQ (data acquisition) loop and transformed it into a visualizable representation. Usually the representation contained the timestamp and the value, which made it a good candidate to plot.

Master The Master loop was the processing loop and the receiver of user input. Queues were the preferred way to send commands.

Logger The loop ran asynchronously in the background and received basic commands from the main loop. Data was received from the DAQ.

DAQ The acquisition module subscribe to the actual data sources outlined in the configuration table. The acquisition loop sent the raw data to the logging loop and a filtered set to the display loop.

The final prototype was able to sense 35 sources simultaneously through an array of interfaces [28]. Due to the short time only a rough architecture was implemented, whereas the ability to drive analog and serial devices was missed out. But keeping the fact in mind that this project was implemented only during 2 months, the outcome was remarkable. The established architecture and the consumer-producer pattern formed together a well-thought data acquisition framework.

3.5 Remote Data Acquisition Using Bluetooth

Loker et al. proposed in 2005 a remote data acquisition system, which was capable to transfer data between two Bluetooth modules [46]. The system was based on LabVIEW. The architecture was fairly simple. A master and a slave computer formed a data acquisition system. The computers were connected via a RS-232 port to a Bluetooth module. Software and hardware were optimized on both sides, and software determined which

node acted as the master and which was the slave. The master was referred as the local computer, while the slave was associated as the remote computer. The local computer was capable to transfer control information to the remote host. Those information included number of samples, sampling frequency and the acquisition mode. After a successful data acquisition under the given parameters, the remote host sent the captured data back to the master node. The project was developed during a senior course at the Berendt College. The goal of the project was to build an automated data acquisition system, using Bluetooth as the data transfer technology and a LabVIEW-based software system. Data was captured via a DAQ board [46].

The local computer could specify the acquisition settings, the so-called operational modes. Single acquisition mode exactly sampled one data set with the given parameters. The continuous mode indefinitely sampled single data sets and sent it to the local computer. The last mode identified the system to stop the transmission. The dataset was defined by its number of samples and the sampling frequency, whereas both parameters are controlled by the operator. The continuous mode additionally supported a time span between dataset acquisition. The system utilized a maximum data rate of 460 kb/s, as this was the maximum transmission rate via UART. After a successful boot and the setup of the Bluetooth radio, the master sent an inquiry command. The inquiry command was a discovery packet. The slave answered with a packet containing its address. A test packet was sent after the initialization, which should ensure the integrity of the channel. After the data transfer a disconnection message terminated the link. Usually the master automatically created the communication link with the slave. The remote host kept being idle, until the master sent an inquiry message. After the connection was established the slave received a packet with the acquisition parameters. After acquisition the data packet was sent to the master.

It is noticeable that the described system was packet-based. It is also worth mentioning that the data is only transferred after it was fully captured. This infers that the system was not capable of real-time data acquisition, but on the other hand the system was capable of sensing data with a high sampling frequency, because usually the transfer medium is prone to act as a bottleneck. In this example the maximum transfer rate was dictated by the transfer rate of the serial interface. However, the system was tested under the continuous data acquisition mode to capture 28 samples with a sampling frequency of 10 kHz and a delay of 250 ms between the data sets. The DAQ board delivered a sinusoidal waveform which was sensed by the slave node and periodically transferred to the master node. The LabVIEW application visualized the captured waveform [46].

The students were given the possibility to evaluate the project. Interestingly two of four students suggested to improve the data transfer protocol, by periodically sending data, instead of upload data once the data set is fully captured. Although it was only a student's project, it proposed a solid and structured approach of implementing a remote data acquisition system, based on a wireless data transfer technology.

3.6 Development of a Custom Data Acquisition System Based on the Internet of Things

Internet of things (IOT) was one of the buzzwords in recent years. Despite the hype of the last years the internet of things is not a novelty. Mark Weiser defined ubiquitous systems in 1991 [74]. His requirement for the computer of the 21st century was that computers vanish in the background, but still assist human in the everyday life. His definition can be seen as the blueprint for IOT. Due to the success of the small and inexpensive development boards of Arduino and the spread of the Raspberry Pi micro computer, the development of IOT applications accelerated. Cheap, internet-capable,

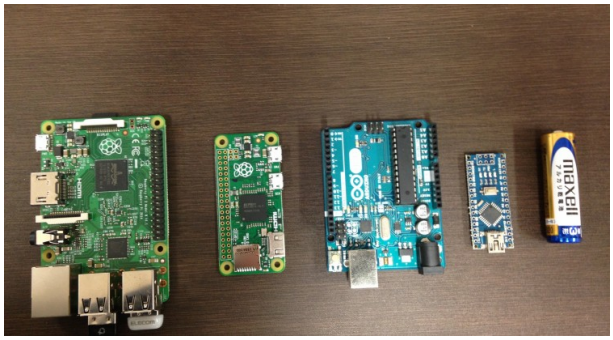


Figure 3.3: Internet of things capable devices.
Source: www.bisland.xyz

single-purpose devices with a hardware abstraction programming layer lowered the barrier for developers for creating a variety of ubiquitous applications. Developers do not have to code in C anymore and write the TCP network stack by themselves. Devices like the Raspberry Pi runs its own operating system and can be programmed among others with Java or Python [55]. The network stack is handled by the operating system.

The power of this technology lies in its scalability and matches perfectly with the requirements of modern data acquisition systems. Vujovic described the collection and analysis of large dataset as one of the main challenges in engineering and natural science [72]. He promoted the development of a modifiable, low cost, scalable system for research methodologies in a university environment. Commercial data acquisition systems for education were often general-purpose, multi-functional systems [72]. The system contained a host controller and multiple sensor nodes [57]. The sensor nodes communicated with the host controller. The sensor nodes could easily be replaced with IOT-enabled devices as seen in figure 3.3. Vujovic stated three advantages of using a Raspberry Pi as a custom sensor node:

- Dedicated Linux-based operating system,
- Various forms of connectivity,
- General input output pins (GPIO).

These capabilities dramatically improved the productivity for embedded device programming, because of the utilization of high-level programming languages and provi-

sioning of operating system functions.

The approach described in the paper used four different kinds of sensors. The data acquisition node was capable of sensing temperature, gas levels, light and ambient light intensity. Data was published via a RESTful web service using an Apache Tomcat server and retrieved via the HTTP protocol in a JSON encoded format [72]. The usage of a RESTful web service was proposed in an earlier paper [73]. Vujović proposed an archi-

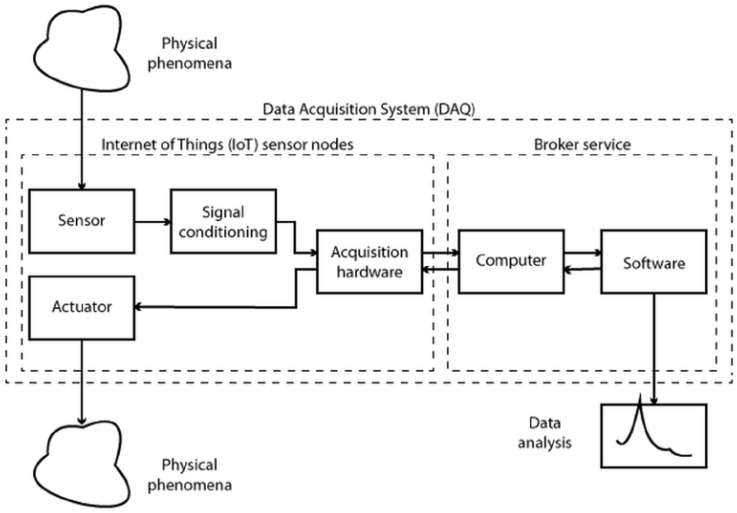


Figure 3.4: Architecture of an IOT-based data acquisition system.
Source: Architecture proposed by Vujović [72].

tecture depicted in figure 3.4, where a single broker service could orchestrate a multitude of sensor nodes. The nodes fully implemented the workflow of analog sensing, digital converting, multiplexing and amplification (if needed) of the signal. The broker service collected data and stored it in a database. The service was also responsible for visualizing and exposing data.

The depicted solution was inexpensive and scalable. The usage of HTTP and JSON made it versatile and easy to apply. It was an excellent example how IOT could be applied for educational purposes, but the architecture inhibited especially performance. The solution worked well as a multi-purpose data acquisition system as it was specified, but was infeasible for a single-purpose data acquisition system for several reasons. JSON might be a handy format for data transmission, but introduces a memory and performance overhead, as parsing messages costs CPU time. This overhead may be neglected by sampling durations bigger than 1 second, but high performance data acquisition systems need a higher sampling frequency than 1 Hz. In these cases JSON imposes a performance penalty. Binary formats like ProtoBuf are considered for better performance. The broker service orchestrates the sensor nodes. This component is necessary in all data acquisition systems. The problem hereby lies in the data flow direction. The broker has to pull to the data from the nodes, instead of the nodes push the data to the broker. The approach of Vujović requires every sensor node to run its own RESTful web service. As IOT devices are usually battery-powered, the power consumption should be

kept low. The broker service also has to know the location and address of each and every sensor node, in order to pull the data.

Vujović drew a solid IOT-based data acquisition system for educational purposes, but with several drawbacks regarding performance and architecture.

3.7 A Novel Feature Extraction for Robust EMG Pattern Recognition

Data acquisition systems do not merely acquire data, they also provide ways to analyze the captured data. Feature extraction was already explained in section 2.2 based on the paper of Veer et al. named *A Novel Feature Extraction for Robust EMG Pattern Recognition* [71]. The paper mainly described the different features of an EMG signal, but covered the general aspect of data analysis as well. As analysis is an integral part of data acquisition systems, the paper was related to the processing unit of a data acquisition system.

Noise is an inherent part of an EMG signal and some sources can hardly be eliminated, already described in section 2.4.1. Veer et al. suggested to use features, which were capable to perform well, even under noisy conditions [71]. Noise per se was not the concern, because the different causes could be addressed with filters. Random noise in the frequency range of the energy of the EMG signal was hard to remove without losing information. White Gaussian Noise (WGN) as a proxy of random noise was a valid approach to model noise in an EMG signal. Veer et al. highlighted the importance of the selection of robust and WGN tolerant features for the sake of pattern recognition quality.

The hypothesis of the paper was built around the question, if modified mean and median frequencies – which use the amplitude spectrum instead of the power spectrum – performed better, compared to other time and frequency domain features, in pattern recognition, depending on different signal-to-noise ratios, calculated as

$$SNR = 10 \log \frac{P_{clean}}{P_{noise}}. \tag{3.1}$$

The experiment was conducted using two pairs of silver-based electrodes, separated by 2 cm. A bandpass between 10 - 500 Hz filtered the signal. The signal was sampled with 1000 Hz with an ADC wit a resolution of 16 bit[3]. The signal was sensed on the right forearm with a sample size of 256 ms and a real-time-constraint of 300 ms. A second data set was recorded in a noisy environment.

13 time domain features and five frequency domain features were computed. Due to the simplicity, time domain features could be calculated in real-time. Time domain features are valuable for detection muscle contraction and muscular activity, while frequency domain features expose information about the muscle fatigue [71].

The calculation of the percentage error (PE) was used for each feature as the comparable metric. The percentage error was defined as

$$PE = |\frac{feature_{clean} - feature_{noise}}{feature_{clean}}| \times 100\%. \tag{3.2}$$

[3]The proposed acquisition parameters by Delsys [18].

The feature vectors were derived from the different data sets. The smaller the PE value, the better the tolerance towards noise. Performance evaluation was actually done with the *Myoelectric Control Development Toolbox* [7]. The software was declared as a pattern recognition library written in MATLAB and utilized an array of machine learning techniques to classify EMG data.

The results showed, that RMS is the most robust time domain feature. In the frequency domain the modified mean and modified median outperformed the other features. It also proves that because of the single feature per channel setup, it is possible to combine different features to a more powerful feature vector. Single robust features are namely MMNF, MAV and RMS, among others. It is noticeable that the modified mean frequency (MMNF) outperforms the other features. In fact, MMNF only misclassifies 6% of the samples in a good EMG signal and only 10% classification error in a weak EMG signal with a SNR of 0 dB. The error shrinks down to 0.4% as the SNR equals 20 dB. Therefore the modified mean and modified median frequency are a qualified choice for either single or multi feature vectors for pattern recognition [71].

3.8 Conclusion Related Work

Each of the depicted papers describe one or more scenarios, which influence the shape of the proposed system. The knowledge and the shortcomings related work accumulate into chapter 4, defining the boundaries of an extensible data acquisition system for electromyography.

EMGworks is the state of the art EMG data acquisition system. It is superior to Motion Lab Systems as it provides a full documentation and a broader hardware support. Mechanisms like work flows bear much potential. Motion Lab Systems seem a little outdated, but nevertheless they provide functionality at high level, but restricted to Windows machines.

Nogiec pleads for an extensible architecture in the general domain of measurement systems. Extensibility and portability are deeply integrated into the system. He also introduces the distinction between producer and consumer nodes.
The article of Grech about COLDEX outlines the importance of communication abstraction. The orchestration of an array of devices required an accurate communication interface, independent of the actual device.

The work of Loker and his students proposed a rather simplistic, but effective protocol of communication between a producer and consumer host. The paper highlights the strengths of the producer-consumer pattern on different hosts.

Vujović takes internet of things into consideration. His approach uses small, IOT-enabled devices as data nodes, which can be queried by for data. This would be a cheap and scalable solution for an orchestra of sensing devices.

The paper of Veer et al. does not focus on software, but rather the theoretical aspect of data extraction in EMG signals and the technical requirements. The knowledge should be considered, when building an own sensing device.

Chapter 4

Problems and Requirements of EMG Data Acquisition Systems

Software is rewritten and newly implemented, because of the simple fact that it lacks essential software qualities, which are critical for the use case of the target application [30].

The broad range of related works indicates the quality of already published and released data acquisition software. Depending on the actual scenario, different commercial products like *EMGworks* will definitely suit the needs of the corresponding use case. However, special use cases and distinct features will certainly require a set of utilized qualities. If these qualities are not met with existing software, a new system incorporating these qualities has to be created.

This chapter outlines the quality attributes a modern and new data acquisition system has to impose. Therefore the outlined qualities will create the specification of the data acquisition system *EmgFramework*. The newly implemented system is compared regarding the specified qualities with the related commercial products.

4.1 Software Quality Shortcoming of Related Work

The related work focused on data acquisition systems in general and especially for EMG. Software is not rewritten because of missing functionality, but moreover because it is simply not scalable, not testable, too slow or not secure anymore [5, p. 63]. Usually there is no need to rewrite existing software, if the software utilizes functionality for the required use case. The concept of functionality can be further abstracted of the concept of qualities and is defined in the ISO standard group 25000 (known as SQuaRE: System and Software Quality Requirements and Evaluation) [34]. A detailed outline of software product quality is pictured in figure 4.1.

Each of the related projects suffer from the miss of at least one essential software quality. These qualities are also called non-functional requirements (or sometimes referred as extra-functional properties) [11, p. 23], because they rather shapes the system in terms of structure and behavior. Therefore the set of utilized qualities have a direct impact on the software architecture of the system. Software architecture is a rather abstract term. Clements et al. describes the architecture as the blueprint of the system.

The architecture impacts the development process as well, because it dictates the assignment of implementation and designing tasks. The architecture is the primary driver of the utilized system qualities [11, p. 13]. A modern framework for EMG data ac-

Figure 4.1: The definition of software product quality.
Source: www.iso25000.com/index.php/en/iso-25000-standards/iso-25010

quisition should exhibit an architecture which incorporates the qualities of portability, maintainability, performance and extensibility.

4.1.1 Portability

According to the ISO standard *Portability* is defined as the degree of efficiency to which a system or software component can be transferred from one execution environment into another execution environment [34]. Portability is the broader term for the subqualities *Adaptability*, *Installability* and *Replaceability*.

In this concrete scenario Portability is seen as the degree to which the software can run on mobile platforms. Usually EMG measurement is conducted in a scientific environment. The case for the usage of EMG devices in the everyday life is hard to test. Therefore most projects suffer from the lack of a portable software. Only EMGworks offers the capability to run certain products on mobile devices (Android only). Next to it EMGworks provides an API to acquire data with their sensors. But the software only enables developers to access their hardware, evaluation software is proprietary [20]. Portability concerns also the part of sensing devices. Usually they are dedicated microcontroller operated devices. Software is specially written for these devices. Portability is hard to achieve, due to the low hardware abstraction of these devices. Increasing portability for sensing devices may impose an performance overhead, but would effectively reduce development time for new sensor devices. Except of EMGworks, all other related projects do not utilize this attribute.

4.1.2 Maintainability

Maintainability is described as the degree to which a system can be modified, improved, corrected or adapted to new requirements [34]. The quality comprises *Modularity*, *Reusability*, *Modifiability* and *Testability*. Reusability is outlined as an extra quality, although it is seen quite natural for object-oriented programming programming languages. Modularity describes the characteristic of a decoupled architecture, where

discrete components are prone to changes in a different component. The terms Modifiability and Testability are outlined separately, because of its reoccurring usage. Those two terms are used regularly to describe a software architecture, although defined in the group of Maintainability. Modifiability describes the degree of which a system is capable of altering the software, without the need of massive refactoring. Testability describes the factor to which a system is able to perform tests [34]

Granja-Alvarez et al. stated 20 years ago that nearly 60 percent of the resources are spent for maintenance [27]. This metric is clearly outdated, but it proves that maintainability has an important role in software engineering since decades. It is tightly linked with Extensibility (depicted in detail in section 4.1.4). Maintainability is a software quality which no modern application should abolish. In general all related projects are either closed-source or a proof-of-concept implementation. Therefore a comparison to related projects is infeasible.

4.1.3 Performance

The component regarding *Performance* is comprised by *Time Behavior*, *Resource Utilization* and *Capacity*, although the the term performance is mostly associated with resource utilization [34]. Time behavior is quite important, as it imposes timing constraints for processing times. Time behavior should not be mistaken with a real-time system. Real-time systems are defined in ISO 2382.

EMG acquisition applications are not necessarily time critical, but data acquisition systems can have real-time capability as a requirement. Generally speaking the data should be acquired with a low latency time. Performance matters, due to the demand of a portable system, the system must feature a small memory and CPU footprint in order to run on mobile devices. Vujovic et al. proved the feasibility of an IOT-enabled data acquisition system [72], although the architecture manifests some unnecessary performance overhead. He suggested that every sensing node should run its own web server for publishing data. The broker service – the counter part of the sensing node – had to pull data in a regular basis. Running its own web server is cost intensive. IOT devices should rather keep a low memory and energy footprint, due to their mobile nature. The data was encoded with JSON, which hinders time behavior and latency, because of message parsing and redundant data transfer.

4.1.4 Extensibility

The term *Extensibility* is not part of the ISO 25010 definition, but commonly used as a software quality attribute, sometimes used synonymously for *Modifiability*. Johansson describes Extensibility as the ability of a system to incorporate new functionality, without effects on internal structure and data flow. Whereby the definition identifies Modifiability, Maintainability and Scalability as the three most important quality attributes, which directly impact Extensibility [36]. Anyway, these terms and definitions are highly mingled. For the sake of a clear definition, this is the only term, which is not directly cited from the ISO standard.

However, Extensibility is a quality attribute which nearly applies to all software solutions and is most likely a good cause for software rewriting. Grech redesigned the data acquisition system for CERN, due to the lack of extensibility [28]. A modern EMG

acquisition system should be able to incorporate new functionality, for example new tools or new sensing devices. Nearly all related work lack a flexible driver interface. Related projects support a single medium for data transfer between the sensing device (client) and the acquisition device (server). Grech took an integration of serial communication into consideration, but didn't implement it [28]. EMGworks and Motion Lab Systems use both their proprietary transport layer with proprietary sensing devices and a proprietary data evaluation software [20, 69].

Pyacq[1] is a open-source project written in Python, which has an extensible architecture, but only utilizes communication via TCP/IP. The fact that it is written in Python hinders another quality attribute: Portability.

4.2 Cost Factors

The project management has to decide when it is advisable to build a software module or tool on its own, or buy the product from a commercial vendor. Clements et al. outlines the convenience of software bought from the commercial market, which ships with a predefined set of functionality, whereby the remaining requirements in functionality must be integrated into the bought modules [11].

Baker described the problem of Build vs. Buy with an example of a feature flagging platform [4]. Feature flagging became quite popular over the last years, whereas developers can ship functionality and enable it remotely via a flag. Unlike other publications, Baker focused only on the engineering part, leaving out the business aspect of such a decision. The decision making was formulated as a list of questions. Homegrown software must exhibit certain software qualities, such as Maintainability, Performance and Scalability. Therefore the question was if a company or project team has enough resources and time to build an additional software component, which will meet the quality requirements. Though, bought software components still need a system integration and they must be maintained over time as well [4] (visualized in figure 4.2).

Fowler took a more comprehensive approach to the topic, as he included the business aspect next to the technical aspects [23]. He terms of COTS (commercial-off-the-shelf) components and COGS (cost of goods sold). COTS were commercial components, whether hardware or software. They were usually more flexible and designed for a multipurpose usage. They tend to be shipped with more features than custom components. COGS described the cost of manufacturing each unit and played an important role on cost calculation. Fowler highlighted that the decision making is market-dependent. Further Fowler outlined the six considerations regarding the decision:

Quantity Thinking of a hardware product, the amount of units plays a crucial role. Large and more complicated systems tend to have a higher profit margin. From a business viewpoint this means that large systems will earlier reach the breakeven point. Smaller products will usually have to sell thousands of them, in order to reach the breakeven point. Still, the quantity consideration applies to software as well.

[1] https://github.com/pyacq/pyacq

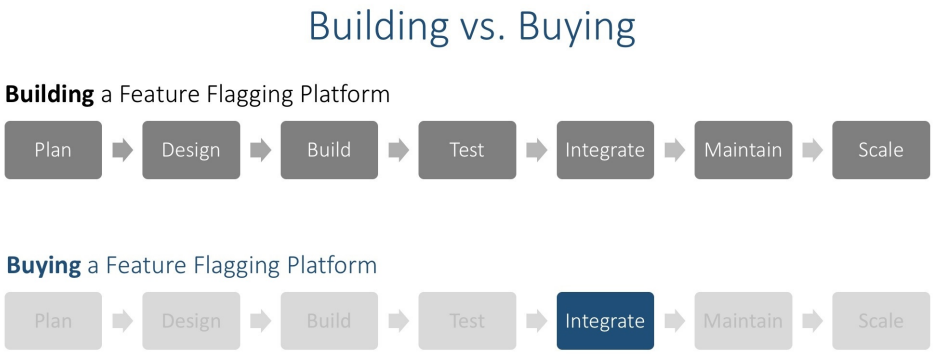


Figure 4.2: Difference between Build vs. Buy workflows.
Source: blog.launchdarkly.com/buying-vs-building-a-feature-flagging-system/

Cost Fowler gave prominence to the cost factor of COTS. Building a custom design could be more expensive than using COTS, whereas COTS are more expensive, as they need integration into the system.

Time Buying components should reduce the time and resources of development (mentioned by Baker [4]). From the software perspective this required an architecture, which provided an interface to bought software components. Refactoring the architecture to comply to bought components could be time-intensive.

Specifications Wide or narrow specifications are an indicator to build or buy components. Fowler suggested to build custom components when the specifications are very right or very loose [23].

Resources A project team or company must exhibit expertise in the topic and time, in order to build custom components. If the expertise is missing it is advisable to trust vendor and its components.

Technical Support The decision about technical support will most likely affect enterprise applications or hardware manufacturer. The aspects of support from the vendor and provided support for customers should be kept in mind.

Fowler concluded that a thoughtful compromise of building and buying seems the best solution for some scenarios [23].

Cortellessa et al. presented an optimization framework for build or buy decisions regarding the software architecture [14], which was based on a mathematical model. The novelty was the introduction of cost and quality attribute variables for decisions. Whereas related work depicted the most suitable COTS components, but additionally comprised a decision for a custom implementation. The model was capable to even define the amount of component testing. The framework was built for embedding into a

Cost Benefit Analysis Method, which should support the decision process of Software Architects [14].

4.3 EmgFramework - an Open-Source Framework for EMG Data Acquisition

Due to the various shortcomings or disadvantages of the related projects, the application framework *EmgFramework* was created in order to tackle those shortcomings. One integral drawback of existing projects was the absence of a flexible and exchangeable connection. Most of related work utilized only a single medium for communication, albeit it be an ISM radio band, bluetooth or ethernet. The proprietary of sensing nodes disallowed an open standard for data exchange, although there were at least SDKs to incorporate such sensor nodes into an existing application, although this implicates a vendor lock-in. *EmgFramework* was built with the vision of removing any possible vendor lock-in on the sensor node or client side.

The openness of the system is one of the core features, and therefore the fundamental architecture must incorporate quality attributes like portability and extensibility. These two qualities implicitly require maintainability as well. A maintainable and well-structure architecture can reduce maintainability costs separated modules and components and stable interfaces between them [50, p. 140]. If the code base is not maintainable, it is likely that new features and tools (extensibility) are hard to implement and to port to new platforms. The last critical quality is performance. Delsys suggested a sampling rate of EMG signals with at least 1000 Hz [33]. Data should be displayed in nearly real-time. Data must be processed, filtered and stored, both on the desktop and on mobile applications. Especially on mobile devices performance is crucial.

4.3.1 Qualities

The reference implementation focuses on the qualities of portability, extensibility, maintainability and performance, as each of them is a key driver of the architecture.

Portability

The system is divided into a data producer node (or nodes) and a consumer or server node. Data producer send data to the server and the server evaluates and processes the data. Portability is the key for both, the producer and the consumer. The consumer should run at least on desktop devices and for the sake of mobility and field tests on mobile devices as well. Producer nodes should also run on different hardware to avoid vendor lock-ins. Producer should not be limited to a certain hardware platform or a certain programming language.

Extensibility

One of open source biggest assets is the ability of extending or forking existing software. To fully exhibit this asset the architecture must be designed for extensibility. A user can

perform a variety of tasks and computations on EMG data. Therefore the architecture must provide a mechanism to comfortably extend the functionality.

Maintainability

Maintainability is partially covered by extensibility. In addition the implemented system must be modular and reusable, next to being modifiable. As the framework should run on mobile platforms and on desktop platforms, maintainability of existing code is crucial, because at least two user interfaces depend on the stability of defined interfaces. Breaking interface contracts by bad architecture and hardly maintainable code induce costs of fixing broken contract definitions in the user interface layer.

Performance

A data acquisition systems is responsible to capture an arbitrary amount of data with an arbitrary sampling frequency and stores it on the file system, while optionally displaying the data to the user. Diverse bottlenecks can hinder basic functionality of the system, and bottlenecks can cause data drop in the core acquisition system. Runtime behavior is an important fact. Power and memory consumption as well, due to the mobile context.

4.3.2 Utilized Technologies

Taking the described quality attributes into consideration, decisions regarding technologies must be made to enforce these qualities. While maintainability and extensibility are widely affected by the imposed structure of the architecture, whereas utilized technology can – to a certain degree – affect portability and performance.

Kotlin

Kotlin is a modern object-oriented programming language by JetBrains introduced in 2011[2]. It provides a rich set of functional aspects, but is to 100% Java inter-operable. Therefore Kotlin code compiles into Java byte code and runs in a Java Virtual Machine (JVM). Due to its interoperability with Java it is executable on Android as well. Furthermore Kotlins well-thought architecture supports transpilation into JavaScript. Business logic written in Kotlin is reusable for desktop applications, Android apps and Web apps. JetBrains is currently working on an expiremental LLVM-backend for Kotlin, called Kotlin/Native[3]. Kotlin/Native should run on native platforms, where no JVM is available. Such platforms are iOS, Raspberry Pi (without Java) or on micro controller like Arduino. To this point (May 2018) Kotlin/Native is still in a beta phase. A working version would have astonishing impacts on portability and maintainability. Source code for IOT-enabled devices and micro-controller can interact with functionality of existing modules, without rewriting code in platform specific programming languages.

EmgFramework is built upon Kotlin due to the promise of Kotlin/Native. Source code for client devices and server devices are managed in a single repository, whereas

[2] https://kotlinlang.org/

[3] https://github.com/JetBrains/kotlin-native

client side code can access shared modules such as messaging, which improves maintainability to the fact, that the messaging module is the single source of truth. Changes in messaging will directly affect both parties. Portability is massively improved as Kotlin code is eligible to run on a variety of platforms.

Reactive Programming Paradigm

Reactive programming inverts the data direction from a pull mechanism to a push mechanism. Software components in reactive programming are no longer polling data from depending components, instead they are registering themselves by so called observables for data updates. ReactiveX[4] is a collection of open source projects, all with the goal to introduce reactive-functional programming to a variety of programming languages. Reactive programming also provides a stable and rich set of concurrency functionality, where especially on Android this is vital, as work on the main thread should be minimized. This approach is going to be incorporated, because performance should be improved by the avoidance of polling. The pipes and filters architectural pattern generally uses references between filter to communicate. EmgFramework uses a more lightweight approach, where receivers must subscribe to data updates of the previous component. This enables data publishing components to have $1 : n$ relationships to subscribing components.

Communication Technologies

Communication medium independence is one of the key features of the framework. Therefore a driver represents an abstraction of the communication layer. Drivers per se are data-agnostic, which means drivers do not validate data or impose a certain exchange format. This increases portability, because rewriting drivers is relative cheap, in comparison to the whole framework. Drivers, which are only present for one platform can be exchanged by another medium on another platform. EmgFramework is going to utilize network (UDP), simulation (playback evaluation) and MQTT driver out of the box. A serial interface is additionally supported for desktop systems, while a Bluetooth link is separately available for Mobile and desktop. Next to portability it also impacts performance and productivity. Improvements in the communication protocol and implementation of new drivers can be done independently of the core system.

Communication Protocols

EmgFramework neglects communication protocol to a certain degree. Bluetooth or serial communication do not depend on a communication protocol and the network driver uses plain UDP as a transmission protocol. There is a MQTT (Message Queue Telemetry Transport) driver implementation, because MQTT is in comparison to HTTP data-centric (instead of document-centric) [66]. MQTT is built for the internet of things, as a lightweight, yet reliable (it is based on TCP/IP) protocol [65]. MQTT needs a broker service / server; a clear disadvantage in comparison to the plain UDP socket.
Other communication protocol would be Weave, CoAP or AMQP. Weave is developed

[4] http://reactivex.io/

by Google and is an IOT communication protocol. In comparison to MQTT it supports different transportation layers like Zigbee, Bluetooth Low Energy and Wifi. It uses schemes to describe data and devices. Security is deeply integrated in the system. In general Weave is used for machine to machine (M2M) communication [6].
The Constrained Application Protocol (CoAP) is a HTTP-like protocol, based on UDP. AMQP (Advanced Message Queuing Protocol) is aimed for enterprise applications. With a smallest packet size of 60 bytes it exceeds the MQTT smallest packet size by the factor 30, but provides in general richer messaging patterns than MQTT [65].

MQTT outperforms other protocols by its simplicity and its lightweight implementation. Nevertheless, further mentioned protocols can be implemented into the system at a later stage.

4.3.3 Supported Producer Hardware

The portability quality attribute ensures, that data producer run a variety of hardware platforms. As a reference implementation a simple Arduino producer with serial connectivity and a more sophisticated Bluetooth-connected producer based on Android Things are implemented.

Android Things

Google announced with Android Things an IoT platform based on Android. Android Things application are programmed with Java or Kotlin with already available tools. An important feature: There is basically no difference in programming for an Android phone or for Android Things. IoT-devices connected to a display can even show the same user interface as on phones. The most important feature is, that applications run in a managed environment within an operating system and applications do not directly depend on hardware related code. Developers do not have to write an own TCP/IP stack, or integrate an own Bluetooth/Wifi module or utilize a simplistic file system on the SD-card. The developer can work on a full, yet IoT-optimized operating system. Kotlin, as the programming language, is natively supported in Android Things.

Arduino

Due to its simplicity an Arduino producer with a serial interface is implemented. Arduinos usually have a rather small size in memory and computation power. The implementation should outline architectural decisions for low-end devices and its performance is going to be evaluated.

4.4 Usability & Reusability

The quality attributes of usability and reusability are outlined separately. The framework follows an approach of *Usability by Reusability*, which means, that the usability for the user is high, achieved by the reusability of software components.

EMGworks by Delsys provides a rich user interface for creating so-called "workflows". Workflows are the concatenation of components, whereas each block has its own functionality and responsibility, pictured in figure 4.3. Users can rearrange and connect

components for their exact use case. The system is usable even for users with limited knowledge about the domain, but EMGworks requires full knowledge about the utilized hardware and sensors of Delsys. EmgFramework should incorporate such a usability,

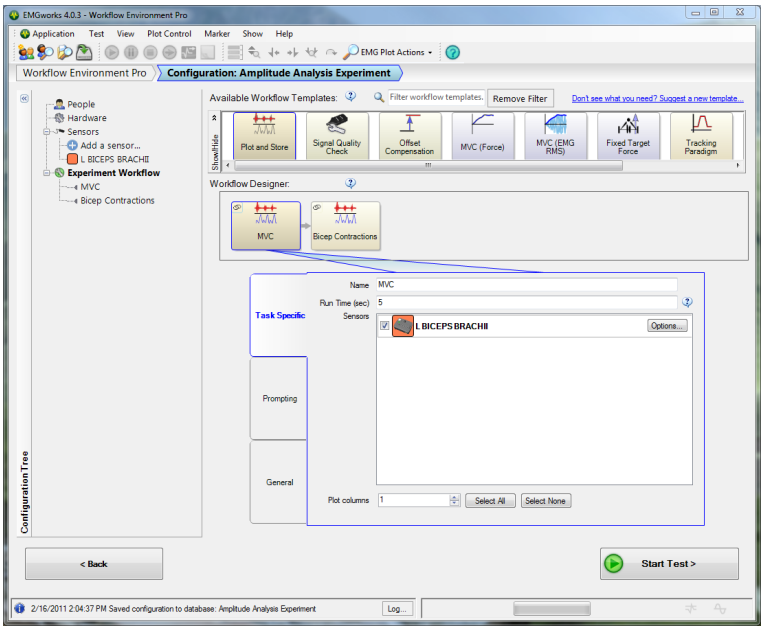


Figure 4.3: EMGworks workflow environment.
Source: www.delsys.com/KnowledgeCenter/NetHelp/default.htm

that creating such workflows is possible with a drag and drop user interface. Users and developers should use familiar components of the core system, when building custom acquisition cases. The feature *Acquisition Case Designer* should provide exactly this functionality. Developers are able to implement software components and mark these components as reusable in a flexible and lightweight manner. Components are flexible itself. A fine-grained level of configuration allows developers to define the degree to which users can access functionality of the components. Some components are configurable to a certain degree. The framework provides an interface to alter these configuration on a component level.

Usability by reusability should engage developers to create modular software, which is reusable by the system. Whereas users can implement new components into their custom acquisition cases.

Chapter 5

Concept & Prototyping

The complexity of the application and the scope of the required software qualities led to the decision to implement a framework. Due to the fact, that producer and consumer are deployed on different hosts, rendered the implementation as a simple library insufficient. Both sites need a specific implementation and utilize platform specific code for several use cases. A framework otherwise provides a common and reusable skeleton architecture, whereby developers can extend it to their own needs, which improves reusability at a broader level. Johnson et al. describes frameworks as semi-complete applications [37]. *EmgFramework* provides such a skeleton architecture, where developers can extend it to their needs.

The chapter describes the underlying software architecture as well as implementation details for so-called data producer devices (the devices which perform the actual data sensing) and data consumer applications (the applications which provide data evaluation and storage).

5.1 Software Architecture

Kruchtens *4+1 View Model of Architecture* is used to describe the software architecture. The model consists of four different views, while the fifth view is a manifestation of concrete scenarios or use cases based on these views [44]. The view model solves the problem of separating concerns regarding the architecture into single views. The views are namely the *logical view*, the *development view*, the *process view* and the *physical view*.

The logical view – or conceptual view – describes the object model or domain model of the system (in object oriented design methods) [43]. Clements et al. describes it in a more generic way: The view contains major design elements and their relationships [11]. The process view shows concurrency and synchronization, while the physical view describes the mapping of software artifacts onto hardware, similar to the UML deployment diagram [44]. The development view focuses on the static organization of the development environment. It describes the source code in terms of packages and module dependencies. The development view indicates reusability, portability, and even development cost evaluation and planning [43, 44]. Kruchten utilizes the Booch notation for the views, but ultimately leaves notation open to the software architect [44].

In order to achieve view consistency, each view has a corresponding legend and view definition.

5.1.1 Logical/Conceptual View

The logical or conceptual view can be visualized with a rough class diagram, where so-called major design elements are the important classes of the framework. The central design tasks of the conceptual view include the indication of *conceptual components, connectors and configurations* [5, 11].

Elements	Classes, Interfaces, Annotations
Relationships	Composition, aggregation, realization, dependency
Properties	Name, stereotypes
Selection criteria	Major design elements of the framework

Table 5.1: View definition - Conceptual View.

The conceptual view is assembled to a high degree of *abstract classes* or *interfaces*. The left part of the conceptual view in figure 5.1 is concerned about the communication logic, whereas the center and the right part focuses on utilizing actual use cases. The `EmgInteractor` delegates the incoming entity data to the use case implementations. In general these use cases are implemented with the abstract concepts of `Filter`, `FrequencyAnalysis` or `Tool`, whereas tool is a mighty class, which has a wide access to all other classes in the *core* module.

The UI part is fully decoupled by the `PlatformView` interface, where platforms provide different views for UI-related use cases.

The principle of data storage is abstracted as well. It depends on the core entity `EmgData` and implementations and provider of different kind of storage will transform the entity `EmgData` to a writable representation.

The classes in the bottom of figure 5.1 are directly related with the concept of the *Acquisition Case Designer*, the concept of *workflows*. This concept is further outlined in section 5.5. The conceptual view shows no actual usage of the annotation `EmgComponent`, because the policies of the framework forbid declaring abstract classes or interfaces as `EmgComponent`. Only concrete implementations can be declared as an `EmgComponent`, which can be used by a `Workflow` instance. Annotations are used instead of interfaces, as annotations expose a more flexible way to define workflow-compatible components.

EmgFramework is designed with a high reusability in mind. Therefore all elements are working to a high degree independent from other elements. Due to this fact, a realization of the `Tool` interface is one of the most powerful elements, as it can incorporate other independent elements.

It is noteworthy, that the framework tries to eliminate tight coupling with abstract classes and expedite the use of interfaces instead.

5.1.2 Development View

The view on development is parted into the *Code View*, which provides insight into the source code structure, and the *Build View* (or module view), which provides a visual

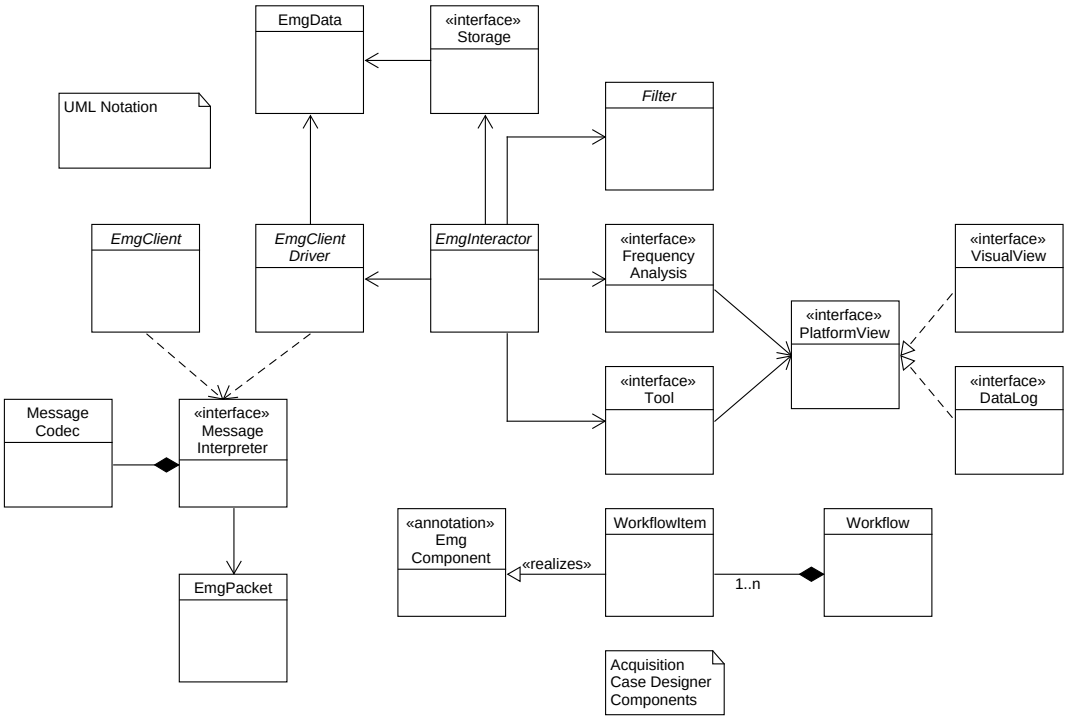


Figure 5.1: Conceptual View of EmgFramework.

representation of the different modules of the framework with their dependencies at build time. The *Code View* provides minimal information at the framework abstraction level and is therefore omitted.

The code is structured into five modules depicted in figure 5.2, which form the foundation modules. The modules *clientdriver*, *client* and *messaging* are responsible for the chain of data acquisition, starting at the producer device while sending data to and receiving data on the consumer site. These modules are rather small and provide the skeleton implementation, while the messaging module is a concrete implementation, where all other modules adhere to this implementation. The *core* provides the skeleton implementations for a vast array of use cases, next to concrete implementations of them and basic drivers as well. Package names solely relate to functionality of different modules. The *casedesigner* module allows other modules to enrich components with the ability to be `EmgComponent`-compatible. The *casedesigner* module utilizes a set of annotations, with which other modules can annotate plain objects as an `EmgComponent`. Other modules are provided with the possibility – but are not required – to incorporate the casedesigner module. The incorporation ensures, that components expose themselves to the case designer drag and drop user interface. The user interface connects compatible `EmgComponents`, thus creating a custom workflow.

Build View

The Build View or Module View describes the dependencies between the declared modules, ignoring third-party or system library dependencies. Each module is annotated with the number of calls by other modules, indicating the dependency usage.

Elements	Modules
Relationships	Dependencies
Properties	Name, number of module calls (optional)
Selection criteria	Dependencies between top level modules

Table 5.2: View definition - Build View.

In general most of the modules utilize *clientdriver* and *messaging* as it is the base layer of communication. *core* is utilized by the UI modules, whereas no other module is dependent on concrete UI or producer modules.

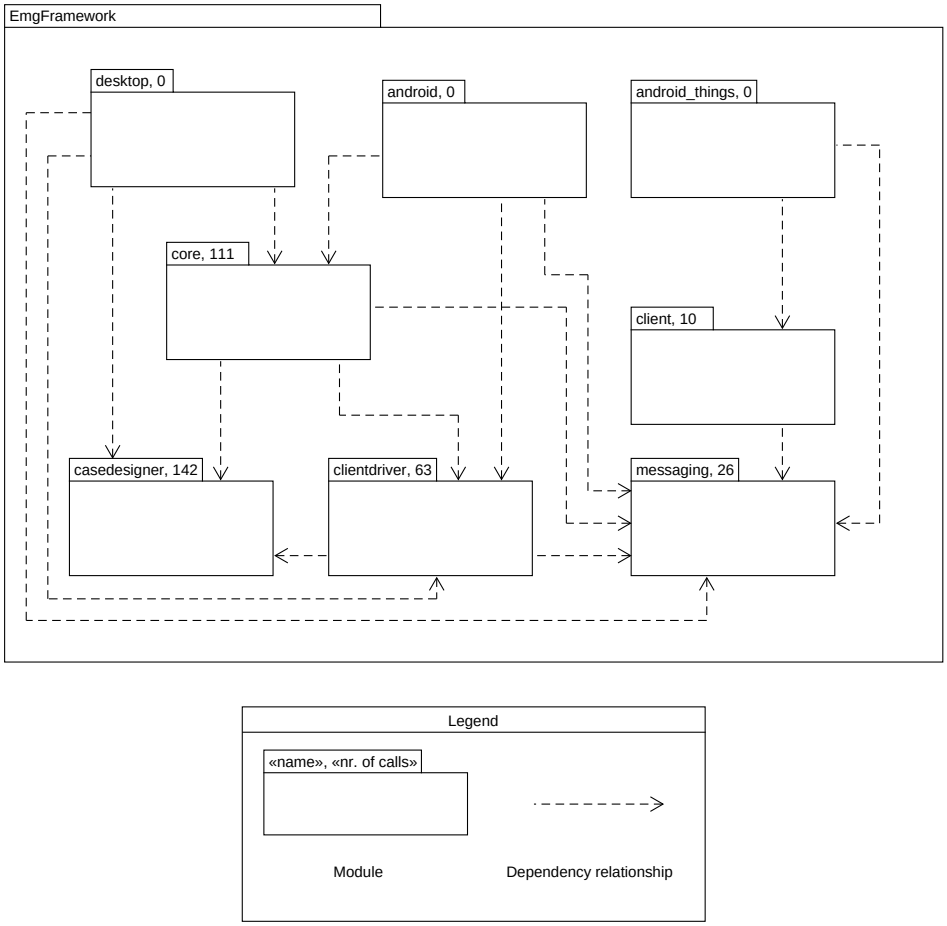


Figure 5.2: Build View at module level.

5.1.3 Component View

Kruchten suggests a *Process View*, which depicts concurrency and synchronization [43]. However, a component and connector view in UML standard is a more informative view, as concurrency can be described in the physical view as well. The component diagram offers a more detailed way to analyze dependencies between the components.

Elements	Components, ports, connectors
Relationships	provides, requires
Properties	Name
Selection criteria	Top level modules as components with external interfaces (no abstract classes)

Table 5.3: View definition - Component View.

In the diagram 5.3 components are first level modules in the source code. These modules expose an array of interfaces for internal and external usage. The diagram considers only external interfaces and moreover only interfaces. Abstract classes are not considered. The diagram depicts a more fine grained dependency analysis, but neglects abstract class dependencies. Therefore the component diagram shows not the exact dependencies as the *Build View*, as the build view considers abstract classes. The interfaces belong to a certain task, which a component can require or provide. In combination with the *Build View* the *Component view*, provides a detailed dependency analysis between the modules.

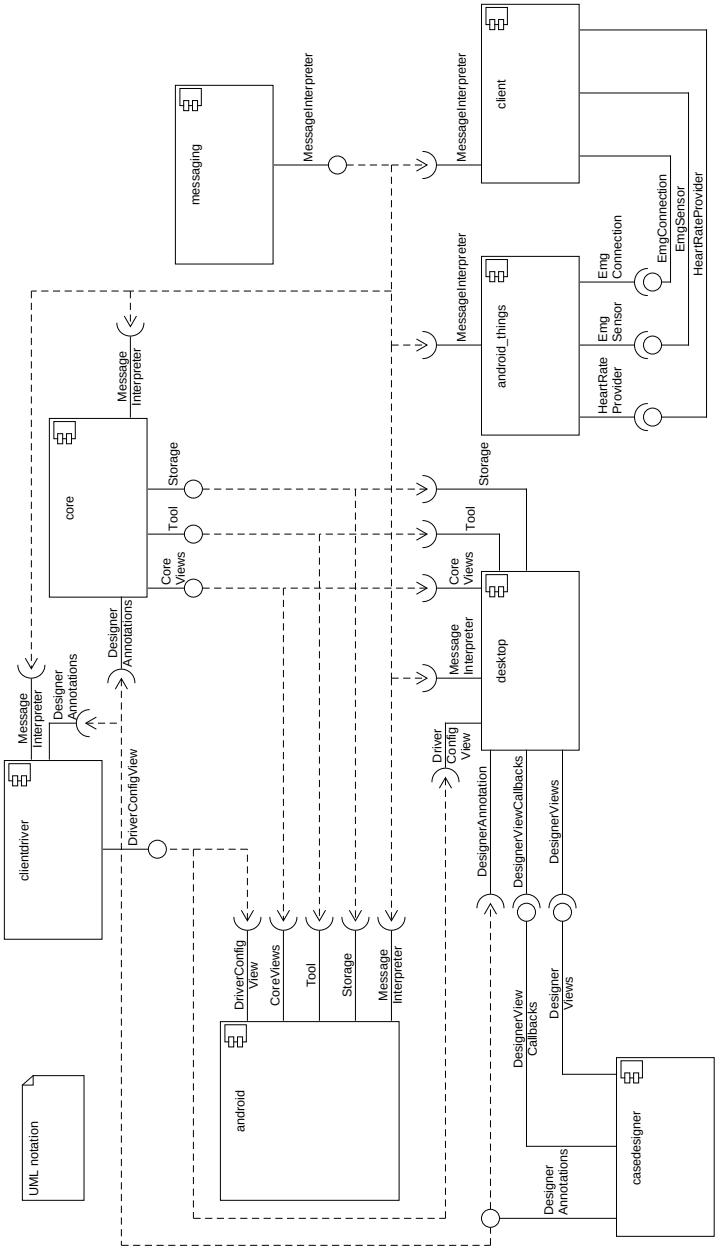


Figure 5.3: Component View at module level.

5.1.4 Execution View

Am UML Deployment diagram is used as the representation of Kruchtens so-called Execution View. The view basically describes the mapping between software artifacts to physical hardware. Elements, which are recommended to be mapped are networks, processes, tasks and objects [43]. The execution view is depicted as an UML deployment

Elements	Devices, environments, components, artifacts
Relationships	hosts, communicates
Properties	Name, stereotype (optional)
Selection criteria	Top level components and execution environment

Table 5.4: View definition - Execution View.

diagram (figure 5.4), with the benefit to use a standardized diagram for such a task.

The consumer device is hosting its software inside a Java Virtual Machine (JVM). Execution of the consumer system requires some sort of user interaction, whether it is running on mobile devices, on desktop applications or a command line interface. Therefore the component *UI* depicts the scenario of user interaction. The components *clientdriver* and *messaging* are responsible for communication purposes. The producer device ideally requires no execution environment like a JVM. However, the Android Things implementation uses an execution environment, thus it is optionally. The *client* component hosts the related source code artifacts for data acquisition. The framework itself combines the *client* with the *messaging* component. Nevertheless, the *messaging component* must be deployed on either sides. Communication is established over a *Medium*. *EmgFramework* is not restricted to one single communication medium. The modularized driver approach allows the integration of different mediums at any time.

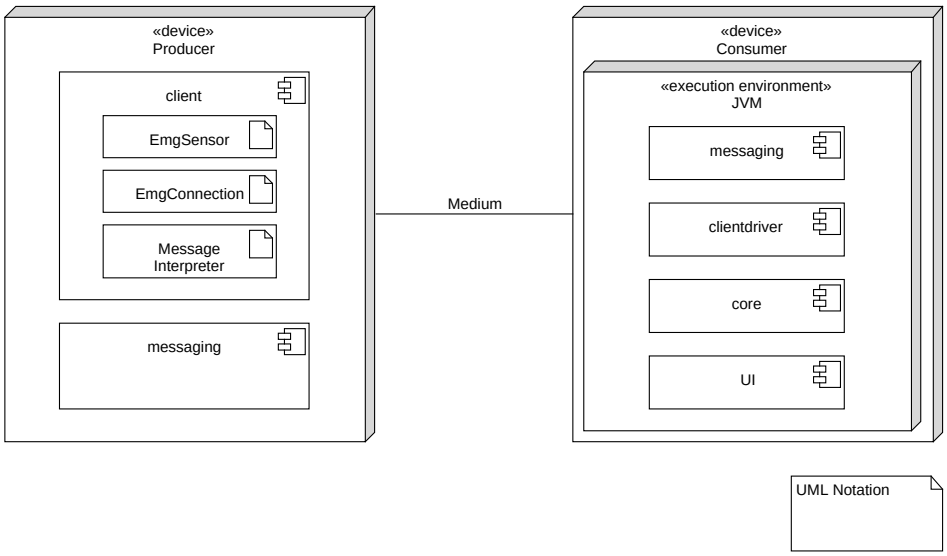


Figure 5.4: Execution View as Deployment Diagram of an EmgFramework application.

5.2 Architectural Styles and Patterns

The architecture consists of basically three different styles or patterns. Communication is based on a producer-consumer pattern, whereas the producer is the remote device, which renders this pattern to a relative of client-server communication. Platform independent user interfaces are realized by the use of the Model-View-Presenter pattern. The concept of workflows is built upon the Pipes & Filter pattern.

5.2.1 Producer-Consumer Pattern

In particular the system utilizes a mixture between the producer-consumer and the client-server pattern. Therefore both nomenclatures are respectively used in the documentation. Per definition the producer-consumer pattern addresses the issue, that a producer adds data to a shared data structure like a queue. The consumer removes and processes the data from the shared data structure. Both consumer and producer work independently of each other and can access the shared data with different frequencies. The actors must not be synchronized [70].

However, in the case of *EmgFramework* the producer typically resides on another host, therefore can be seen as a client device. Data is shared over a common medium and stored in the data pool of the corresponding driver. Data in this pool is consumed by subscribed listeners on the consumer site.

5.2.2 Model-View-Presenter pattern

The Model-View-Presenter (MVP) pattern is an advancement of the Model-View-Controller (MVC) pattern. The MVC pattern allows the separation of data, state and business logic. Views are by definition loosely coupled to the model and therefore easy to exchange, but still, the view has access to the model, as pictured in figure 5.5. However, due to the tight coupling of the controller the MVC pattern inhibit testability, because the controller can't be unit tested. It also inhibits maintenance and modularity, because of the tight coupling between controller and views. For mobile platforms like Android, this coupling leads to difficult unit testing and extensions [53]. MVP addresses this con-

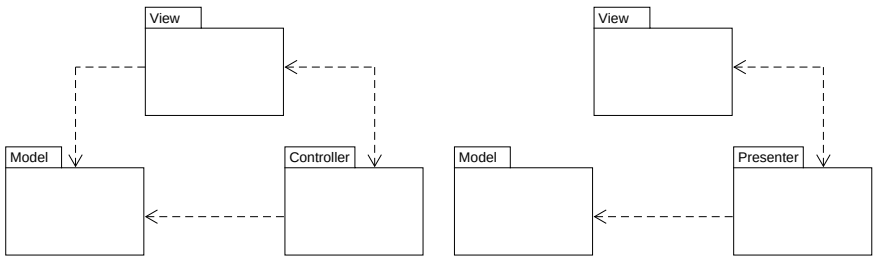


Figure 5.5: Difference between MVC and MVP architecture.
Source: www.slideshare.net/JavierdePedroLpez/mobgen-android-session-mvp-rx

cern by fully decoupling the view with the newly introduced presenter. The view is no longer a concrete implementation, but rather an interface. Therefore presenter do not

lack the capability of testability. *EmgFramework* utilizes thoughtful presenter design, which removes platform dependent code from the presenter, hence increasing testability, portability and even reusability by crossing platform boundaries [53].

5.2.3 Pipes & Filters

The pipes and filters pattern is utilized to realize workflows in the *Acquisition Case Designer* application. The use case of the Acquisition Case Designer is further outlined in section 5.5. The major design elements mentioned in the conceptual view 5.1 are mapped to filter components, each connected to another filter component by a pipe. Filter components receive input data from the input pipe, performs operations and computations and sends the processed data via an output pipe to the next filter. Next to filter and pipes there are source and sink components. Source components initiate the data flow, as they are the first component, which produces the data. The sink component receives the final computed data [63]. *EmgFramework* uses an adaption of this pattern.

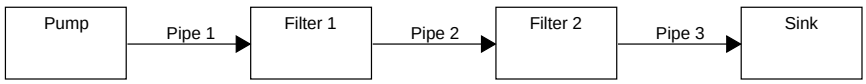


Figure 5.6: Pipes and Filters illustration.
Source: www.rantdriven.com/post/2009/09/16/Simple-Pipe-and-Filters-Implementation-in-C-with-Fluent-Interface-Behavior.aspx

Filter are defined by annotations and the components define an input method and use an reactive approach to push data to the next subscribed filter. Pipes are used to ensure compatibility of two filter by transforming data between them. Pipes extract certain fields from the received data or adjust the data type. They do not alter the input data in any form. Driver classes are mapped to data sources and tools or platform views can incorporate data sink behavior.

5.3 Framework Implementation

The section covers the basic and key concepts of the overall framework implementation. Module structure and content is listed, communication flow is explained and integral concepts of the framework are outlined.

The implementation details for the reference producer implementation is depicted in detail in section 5.4.

5.3.1 EmgFramework Modules

The *EmgFramework* consists of eight separate modules, whereas the modules *desktop* and *android* depict usable, concrete platform specific implementations of the framework. The module *android_things* represents a functional implementation of the *client* module for the Android Things platform.

client The client module is designed to run exclusively on the client device platform. This module only contains abstract classes or interface definitions, due to the fact, that those tasks require device specific code. Still, the scaffold is the same for each and every device. Each target platform requires platform specific code for establishing the communication, sense hardware sensors or connect to heart rate providers.

clientdriver The counter part of the client module. While the client module is deployed on the producer platform, this module is essential in order to communicate with the connected client. The utilized `MessageInterpreter` version is crucial, as the protocol versions are only backwards compatible. A V3 client driver cannot parse V1 message, whereas a V3 client can send data, which a V1 driver can interpret. Driver must be implemented according to their target platform. Platform independent drivers are included in the core module.

core The implementation is platform independent, in order to assure it is safe to run it on any target platform. The platform independence is realized by decoupling UI and storage specific code. Simulation, network and MQTT devices are directly supported. The core module offers frequency analysis methods, filters, tools and storage capabilities. Tools and frequency analysis methods logic is implemented, without the corresponding UI. Views are implemented in the concrete platform specific modules.

desktop The desktop module lacks the capability of a mobile experience, but provides a more potent hardware environment. Usually *EmgFramework* 's policy restricts platform specific modules to contain business logic, except of file storage and platform specific driver. It mostly contains UI related code, due to the fact, that the architectural concept of a Model-View-Presenter pattern is implemented. The desktop module comes with an own driver for serial and Bluetooth communication.

casedesigner *EmgFramework* allows non-experts in the domain of EMG data acquisition to use the application via a simple drag and drop UI. The module casedesigner offers the technical capabilities to mark concrete classes as a so-called `EmgComponent`, which exposes required functionality for the drag and drop UI. As the core module, the module requires platform specific modules to implement the corresponding UI.

messaging The messaging module is the base layer of communication. It contains the interface for `MessageInterpreter`, which defines the interpretation of messages between client and server. Because of its generic interface characteristic, it is theoretically possible to change the message data format and its message interpreter class at runtime. Due to evaluation purposes the module provides a custom `EmgMessageInterpreter`, `ProtoBufMessageInterpreter` and `JsonMessageInterpreter`, whereas all utilize the `EmgPacket` class as the data container, which is transferred between both parties. `EmgPacket` is capable of transmitting a list of EMG sensor values, a timestamp and the heart rate of the test subject. Regardless of the data format, all concrete implementations of MessageInterpreter must adhere to the protocol versioning.

android The concrete mobile implementation of the consumer site. The mobile application nearly contains the same set of functionality, except the adapted implementation of the case designer, due to the small display constraint. The core module already provides mechanisms to run on mobile devices by utilizing UI buffering and background threading. The Android implementation provides a dedicated Bluetooth driver.

android_things The *EmgFramework* supports Android Things as the primary producer platform. The module contains a Bluetooth, network and mqtt connection. Heart rate is provided by a Bluetooth Low Energy device. The actual EMG data is sensed via a *I2C* connected EMG sensor.

5.3.2 Communication Protocol

Each consumer driver and each producer must adhere to the same communication protocol. The communication is bidirectional. Producers do not only send data, they also receive configuration data from the consumer. The whole communication flow is depicted in figure 5.7. The protocol does not define welcome/init messages, nor relies on authentication or encryption. As soon as devices are connected the consumer sends an

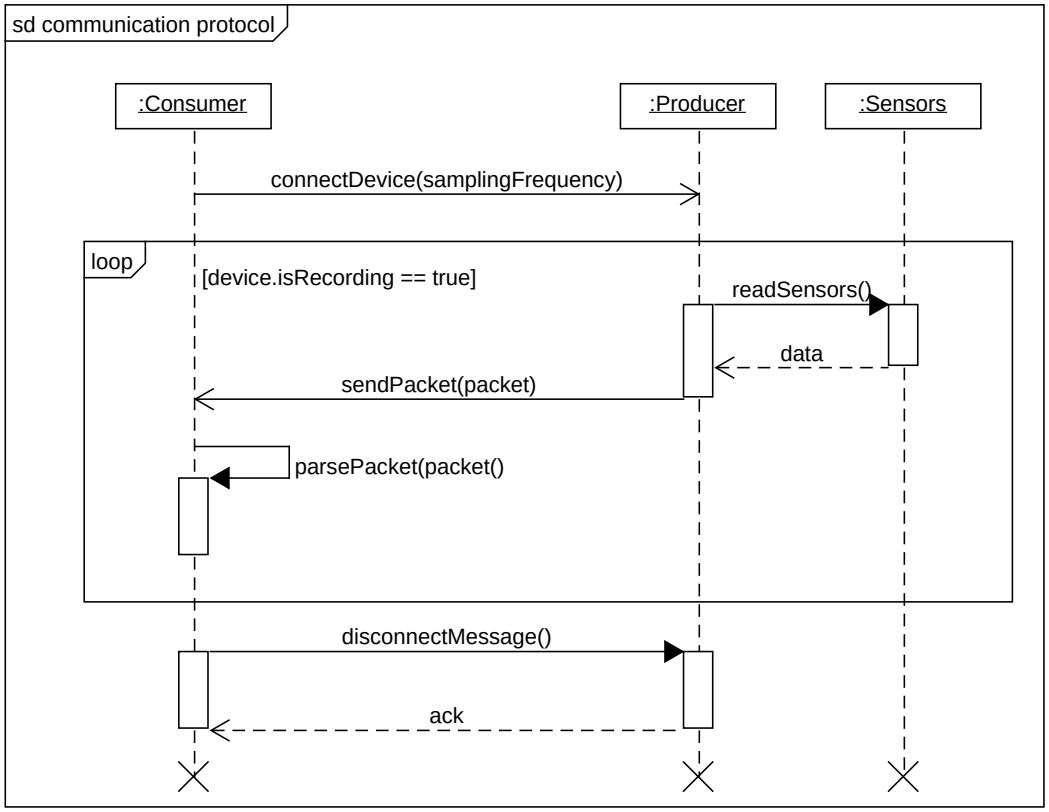


Figure 5.7: Sequence diagram of communication protocol.

initial frequency message and keeps receiving data. The producer awaits the connection of the consumer and starts sending data based upon the received sampling frequency. After acquisition the consumer sends a disconnect message to ensure a safe disconnect. The producer stops sending data upon disconnect.

Data Exchange Format

Communication is established via a driver. The driver provides access to a medium and functionality to send and receive data through it. As drivers are data-agnostic, `MessageInterpreter` components are introduced to process the data. The interpreter consumes data received through the driver medium interface and converts data into a usable object format. The instance is also responsible for converting the object on the client side into a transferable format. This approach provides the flexibility to change the structure of the payload.

For evaluation purposes three flavors of data exchange formats are going to be implemented. Starting with a self defined format, with three different versions, where the first version only sends an arbitrary amount of emg data channels. Version two introduces a timestamp, whereas version three provides a field for heart rate data, as this is additionally supported in *EmgFramework* . The primitive format uses colons and commas to separate data.

```
# v1 - Only EMG data. Multiple channels are separated by ","
1,2,3,4
# v2 - EMG data and timestamp, parameters are separated by ":"
1518436800:1,2,3,4
# v3 - EMG data, timestamp and heart rate
1518436800:1,2,3,4:60
```

Listing 5.1: EmgFramework data exchange format versioning scheme.

Driver

Communication is decoupled from other tasks and processes. The `EmgClientDriver` acts as the interface between the client or producer and the rest of the application. Low level communication details are encapsulated within the driver class. The framework supports clients for *Simulation*, *Network* and *MQTT* out of the box on any platforms, whereas there are specific implementations for the *serial* interface on desktop devices and *Bluetooth*, both for Android and desktop. The usual policy allows only one active driver at the time. Each driver has a corresponding `MessageInterpreter`, which is responsible for message parsing and crafting. Producer and consumer site must adhere to the same message version (shown in figure 5.1) and to the same message interpretation.

The architecture assumes that the consumer application knows the coordinates of the producer device(s). Each medium and corresponding driver needs a set of parameter in order to connect to the producer. The network client needs for example the port and IP address of the producer. Therefore drivers are configurable with a platform independent settings view.

Message Interpreter

The abstraction of interpreting received data in form of the `MessageInterpreter` allows punctual improvements of the data format by replacing its syntax. Though, the versioning scheme of figure 5.1 is retained for every interpreter implementation.

The simplistic data format with colons and commas and a custom data parsing routine is implemented in the `EmgMessageInterpreter` class and shown in listing 5.2. An improved data exchange format is implemented based on the binary serialization format *Protocol Buffers* by Google[1], within the class `ProtoBufMessageInterpreter`. Protocol Buffers (or short ProtoBuf) are platform and language independent. The Protocol Buffers implementation renders custom data parsing code obsolete. Due to the binary nature of Protocol Buffers, this interpreter implementations use in addition a `MessageCodec`, which translates the binary format into a string encoded entity. For comparison reasons `JsonMessageInterpreter` implements a JSON-based data exchange format.

```
class EmgMessageInterpreter: MessageInterpreter<EmgPacket> {

    private fun parseV1(msg: String): EmgPacket? {
        return if (msg.contains(",")) { // > than 1 channel
            val values = msg.split(",")
                    .dropLastWhile { it.isBlank() }
                    .map { it.toDouble() }
            // Do not process damaged packages
            return if (values.isNotEmpty()) EmgPacket(values) else null
        } else {
            // Ensure that single channel has a valid format
            val singleChannel = msg.toDoubleOrNull()
            if (singleChannel != null) {
                EmgPacket(listOf(singleChannel))
            } else null
        }
    }
    private fun parseV2(params: List<String>): EmgPacket? {
        return if (params.size >= 2) {
            val timestamp = params[0].toLongOrNull() ?: System.currentTimeMillis()
            parseV1(params[1])?.setTimestamp(timestamp)
        } else null
    }
    private fun parseV3(params: List<String>): EmgPacket? {
        val hr =  if (params.size >= 3) { // Try get heart rate
            params[2].toIntOrNull() ?: -1
        } else -1
        // Fallback anyway to V2 message parsing
        return parseV2(params)?.setHeartRate(hr)
    }
}
```

Listing 5.2: Custom message parsing routine of the default `EmgMessageInterpreter`.

[1] https://developers.google.com/protocol-buffers/

5.3.3 Key Concepts

The system supports an array of key concepts, which supports different scenarios and use cases. Ultimately these features work independent of each other, but can be grouped and strung together supporting a single use case. Key concepts reduce the complexity of implementing new use cases by providing a reusable scaffold.

Filter

Filtering is an essential part of EMG data acquisition. Therefore the framework must support a suitable way to integrate a core principle in a reusable and easy to use manner. *EmgFramework* supports following list of filters:

- No filter (raw data),
- 50 Hz Butterworth band stop,
- 10 Hz Chebyshev low pass,
- Running average filter,
- Savitzky-Golay filter,
- Threshold filter.

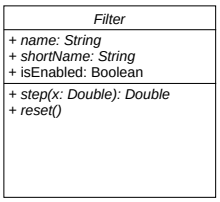


Figure 5.8: Filter class.

No filter returns the unfiltered data. This filter is used, when a component must adhere to the filter concept, but also utilizes unfiltered data processing. `VisualView` is one recipient of unfiltered and filtered data processing. Other filters address a particular scenario. The abstract filter class as depicted in figure 5.8 only containing framework relevant attributes. The class exposes the `step(x: Double): Double` function, which is responsible for the actual filtering.

Section 2.4.1 refers to the problem of ambient noise. Ambient devices may pollute the signal. *EmgFramework* provides a 50 Hz band stop filter for ambient noise filtering. A 60 Hz filter for international usage must be considered.

According to section 2.4.2, the signal should be filtered with a lower boundary of at least 10 Hz. Therefore the framework provides a 10 Hz low pass filter. Referring to section 2.4.2 a 20 Hz low pass filter should be considered as well.

A Running average filter provides a convenient way to filter a volatile signal and therefore it is part of the framework. The filter window size is adjustable by the user.

In order to preserve peaks in the signal, but still apply filtering, the Savitzky-Golay filter is part of the system. The filter width is changeable during execution.

The threshold filter is a general purpose filter to limit signals. The actual threshold is a property of the filter, which can be adjusted at run-time.

The *Running Average Filter*, *Savitzky-Golay Filter* and *Threshold Filter* are parameterized filters, therefore their properties may change over time. The filter concept fits ideally with the *Acquisition Case Designer* application, which offers a drag and drop UI. This UI also provides an easy way to change these properties. The usual desktop application requires extra customization dialogs to change them.

Frequency Analysis

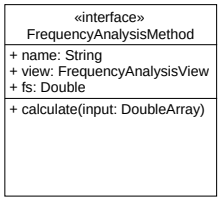


Figure 5.9: Frequency analysis interface.

Feature extraction only for time domain features is for some applications infeasible (described in section 2.2). Time domain feature extraction is most times straightforward and can be applied directly on the captured data. However, frequency features as depicted in section 2.2.2 are most likely based on Fourier transform data. Therefore the framework utilizes a small array of those predefined methods, summarized under the term `FrequencyAnalysisMethod`. *EmgFramework* utilizes following forms of frequency analysis:

- Fast Fourier transform (FFT),
- Power spectrum,
- Mean and Median frequency.

Basically all three can methods may hold a reference to a view of type `FrequencyAnalysisView`. Most applications won't probably require a view much more than for debugging purposes, therefore it is optionally. A concrete implementation of a frequency domain feature in section 2.2.2 is the computation of the mean- and median frequency. This computation uses internally the power spectrum computation, which internally uses the FFT computation. In general all functionality is exposed to developers and not kept private, in order to not force the usage of third party components, while functionality is already implemented by the framework. Mean- and median computation and the power spectrum are sampling frequency aware. These components require the sampling frequency as a parameter. The Fast Fourier Transform implementation provides a parameter to exclude the offset. The muscle fatigue tool uses the mean- and median frequency computation for the implementation of the JASA algorithm (see section 2.2.3). All three methods are implemented using the JTransforms FFT library[2].

Tools

Tools are the most powerful concept, as they encapsulate use cases. While filters and frequency analysis methods can be used independently to analyze generic data sets, a tool implementation can utilize all implemented functionality of other components. Because of this nature tools can exactly serve one use case. The framework implements already a `ConconiTool`, which allows subjects to participate in an automated *Conconi Test*, in order to determine maximum anaerobic and aerobic thresholds [12]. Peak detection in a set of data points is implemented via the `PeakDetectorTool`. In order to determine muscular fatigue, the JASA algorithm is implemented in the `MuscleFatigueTool` (which internally is constructed of a set of core components) and to identify the deflection point of a data set (e.g. in heart

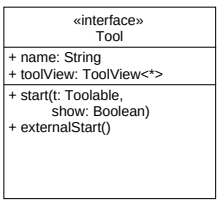


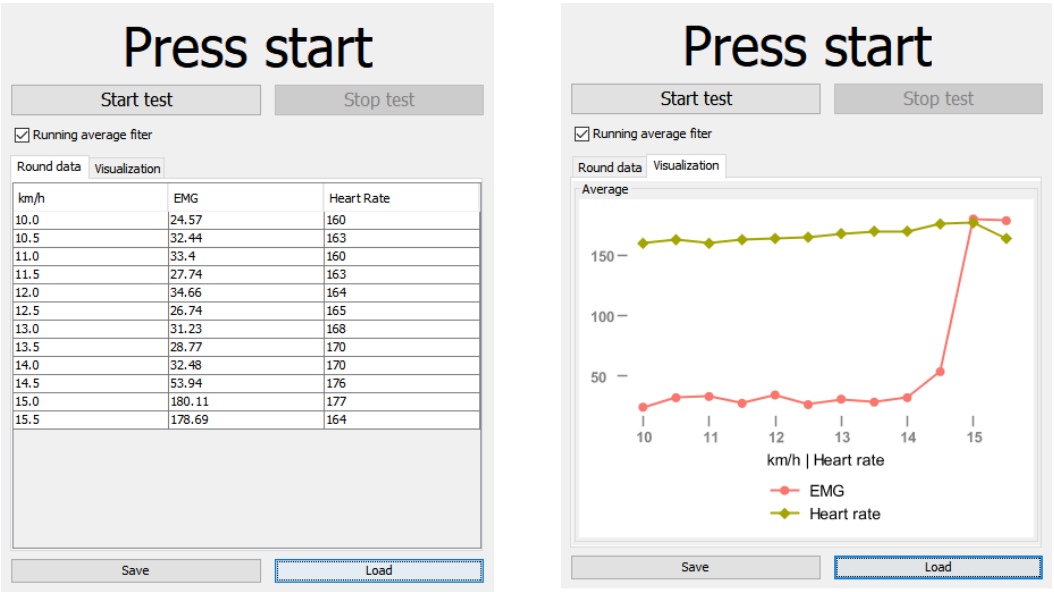
Figure 5.10: Tool interface.

[2] https://github.com/wendykierp/JTransforms.

rate data), the `DeflectionPointTool` implements the respective methods.

Conconi Tool

The Conconi test investigates the change of heart rate over time, in order to determine the anaerobic threshold [12]. The test is ideally conducted on a treadmill, where every 200 meters the tempo is increased by 0.5 km/h. *EmgFramework* incorporates this test for further investigations of the correlation between electromyographic data and heart rate data. Therefore the system also offers native support of heart rate sensing. Figure 5.11 shows the user interface of the Conconi Tool on desktop devices.



(a) Conconi tabular view.

(b) Conconi graphical view.

Figure 5.11: The user interface of the Conconi Tool. Figure 5.11a shows the tabular data representation, while figure 5.11b shows the data visualization.

The tool is built in order to make Conconi tests without the urgent need of a supervisor. However, it is still advised to use a second person as a supervisor. After starting the test a countdown of five seconds will be played. The application automatically connects with the device and starts the data acquisition. After each step the countdown starts again, indicating the subject / supervisor to increase the tempo. Data is visualized either in a tabular form or as a graph. An entry in the tabular view contains the current tempo of the subject, the last heart rate value and the last EMG value of each step. The summary of each steps can be stored and loaded at any point in time for further investigations. Additionally one can choose to apply a running average filtering of the signal. This is optionally, because the component may be used as part of a workflow, where the signal is already filtered.

Muscle Fatigue Tool

Muscle fatigue detection is based on the JASA algorithm [48] described in section 2.2.3. It considers changes in time and frequency domain simultaneously. Basically the algorithm performs periodical computations in time and frequency domain. Luttmann et al. suggests a time window between 5 - 10 seconds [48]. These values are used for time window sizes of *Root Mean Square* and *Median Frequency* computation. After one minute the calculated values for time and frequency domain will be used to perform a regression analysis for each domain respectively. The slope of both regression analyses is taken as the quantitative indicators of temporal change. Thus, temporal change has to be calculated between the last two observation points.

The JASA algorithm was implemented, because it has a reasonable size of computational complexity, therefore making it eligible to run on mobile devices. A downside of the algorithm is, that it is mainly used to investigate isometric, static muscular contractions [29].

The window sizes of the computations cannot be altered by any user interface. These window widths can only be altered at compile time. Exposing these variables to users should be considered in the future.

Peak Detector Tool

For some time series a peak detection might be of interest. *EmgFramework* provides therefore a tool for peak detection and visualization. Usually the user can manually select the time window on which the peaks should be detected. The tool exposes a UI to alter a set of variables, namely the window width between two adjacent peaks, the minimum threshold value of a peak, the decay rate how quickly old peaks are forgotten and if peaks should be set in relation to the local average. The user interface is depicted in figure 5.12.

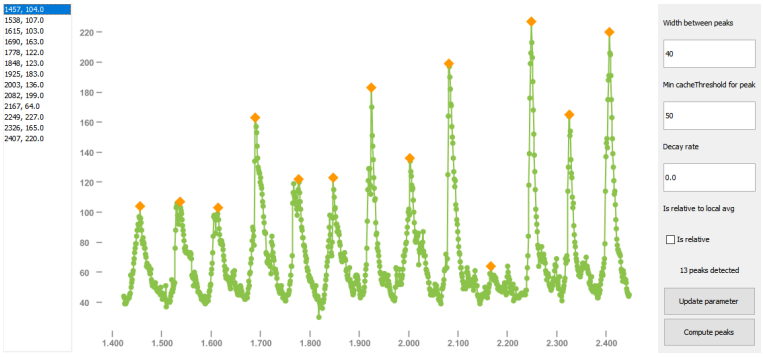


Figure 5.12: Visualization of the peak detection tool.

If peaks are detected the tool publishes the indices of found peaks to possibly subscribed listeners. The UI highlights the peaks in a time series graph and also provides a textual representation in a list view.

Peak Detection Algorithm

The peak detection algorithm is based on the implementation of Simon Dixon and his Beatroot project[3]. It provides fast and reliable results. It is parameterized and can be tuned by the user. The algorithm parts the time series in windows of the width w. The width also denotes the distance between two adjacent peaks. The algorithm searches for the maximum value in between this window width. In the next step the algorithm checks if the maximum value correlates with the window width. As a last step the algorithm performs a threshold check. If the value exceeds the threshold value t and no relative average ra is taken into account, the value is considered as a peak. The algorithm can be summarized with the following steps:

1. Iterate over every value of the time series.
2. Examine the decay rate d for probable peak.
3. Examine lower and upper limit of the window.
4. Search for the maximum value inside this peak.
5. Check if the maximum value lies in the middle of the window.
6. If it's positioned in the middle, perform threshold check.
7. If value passes threshold check, it is a peak.
8. Go back to 1. until array is fully traversed.

The decay rate d, window width w, relative average ra and threshold t are variables, which can be set by the user through the UI.

Deflection Point Tool

Some observations can only be made in a visual manner. For example, the point of deflection in the Conconi heart rate can be detected with the bare, but the outcome is not understandable for machines. This tool provides the ability to compute the deflection point of an arbitrary time series. The component is compatible with the *Acquisition Case Designer* and can compute a deflection point and can forward the result to subscribed listeners.

The utilized algorithm is explained in detail in the next section, while the pseudo code of the algorithm is further outlined in listing 5.1.

Deflection Point Algorithm

The algorithm is a variety of the work of Leitner et al. [45]. His algorithm suggests applying two regression lines in the data set, with minimal standard deviation of the straight lines. The point where both regression lines intersect each other is most likely the deflection point.

The implementation of the algorithm takes the other way of taking a predefined discrete point in the data set and assume this is the deflection point. Two regression lines are drawn through the parted data set and the *Root Mean Square Error* is computed. The point with the least error is the deflection point.

A clear disadvantage is visible at first sight. The deflection point must be a discrete

[3] https://code.soundsoftware.ac.uk/projects/beatroot

value of the dataset and cannot lie between two data points. On the other hand the discrete values indicate a finite amount of possibilities to check as the deflection point. Kara et al. proposes another way for even better deflection point approximations: A third order curvilinear regression method, which is called *Dmax*. Dmax calculates a non-linear regression line over the whole data set. As a second step it draws a straight line between the start and the end point. The point, which is the most far away from the line in a right angle is considered as the point of deflection [39]. The *Dmax* method should be taken into consideration for future development of the tool.

But the algorithm uses for now the simple approach mentioned above. An approach of iterating and dividing the time series into two parts. Each part is approximated by a simple regression line. The point where both lines have the smallest error is most likely to be the deflection point. A visualization of the basic algorithm is pictured in figure 5.13.

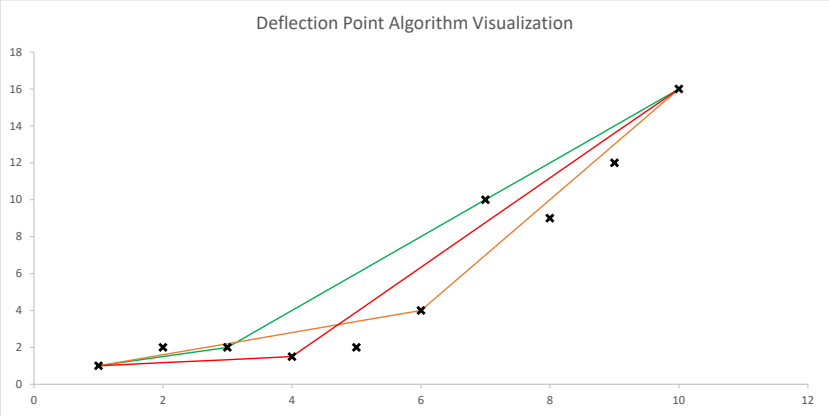


Figure 5.13: Visualization of the Deflection Point algorithm.

However, this algorithm has several pitfalls and edge cases, which must be taken into consideration. The overall time series must at least contain four data points with different x-values. Although the algorithm itself is agnostic of x-values, the differentiation is necessary for the underlying computation library[4]. The lower limit of four values is due to the fact, that regression analysis needs at least a pair of points, in order to reliably compute the slope. After calculating the *Root Mean Square Error* for both approximations, the algorithm calculates the confidence, which is a scalar value. It uses both error values and a so-called `ConfidenceGauge` for calculation. The gauge receives both error values and can calculate an arbitrary value as confidence. The framework uses a weighted error gauge, weighting the error of the second line with a factor of 0.6, while weighting the error of the first line with 0.4. This penalizes errors of the right approximated line more, as the algorithm assumes, that this line is more important, and therefore the error should be smaller anyway.

In addition the algorithm computes the angle between both lines. The user can define certain thresholds for the angle and the confidence. If these criteria are met, the

[4]Apache Commons Math library.

tool will post the found value to the view for rendering.

However, these additional properties are still under active development and disabled by default. The algorithm will always return the best approximation it can compute. Experimental developers can work on a more refined property solution. The tool works with a *Always return a result* strategy, which is sufficient in most cases.

Algorithm 5.1: Deflection point detection implementation.

Require: points: List<Double>, gauge: ConfidenceGauge
Ensure: points.size >= 4 ▷ Only perform calculation if more than 4 data points are available
function CALCULATEDEFLECTIONPOINT(points: List<Double>)
$rmseMap \leftarrow Map()$
for i in 1 until point.size - 1 **do**
$(leftRegression, rightRegression) \leftarrow approxDeflectionLines(i, points.size)$
$rmseMap[i] \leftarrow (leftRegression.rmse, rightRegression.rmse)$
end for
$(bestFitIdx, confidence) \leftarrow calculateBestFitConfidence(rmseMap, gauge)$
$angle \leftarrow calculateAngleBetweenLines(bestFitIdx, points.size)$
return $(bestFitIdx, angle, confidence)$
end function

Buffered computations

Due to the periodic calculations the framework introduced an abstract helper class called `BufferedComputation`, depicted in figure 5.14. It is basically a buffer of a generic type `I`, which computes a value of type `O` of the buffered values, as soon as the buffer is full. Subscribers of such a buffer are informed via an asynchronous, reactive mechanism. Due to its asynchrony computation can be shifted to another thread. The framework incorporates following computations:

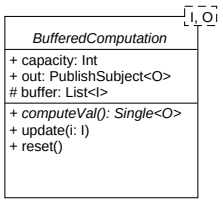


Figure 5.14: BufferedComputation class.

- Integrated EMG (IEMG) computation,
- Mean Absolute Value (MAV) computation,
- Regression Analysis computation,
- Mean-Median Frequency computation,
- Root Mean Square computation.

IEMG and MAV are two time domain features described in section 2.2.1. The feature is a scalar computation of a series of values. This series of values is implemented in form of the buffer. The computation of the features is automatically triggered when the buffer reaches its defined capacity.

Regression Analysis follows the same principle. In the actual implementation it takes a set of values and computes the regression slope. The JASA algorithm (section 2.2.3) uses internally the slope of time and frequency domain regression features, whereas the length of the regression data is known beforehand. Regression Analysis is performed

using the Apache Commons Math library[5].

Root Mean Square and Mean-Median Frequency computations are also used in the JASA algorithm (section 2.2.3).

The main reason of `BufferedComputation` subclass implementations are the fact, that most computations require a series of data to perform on. This key concept combines the data buffer with asynchronous, non-blocking computation.

Workflows

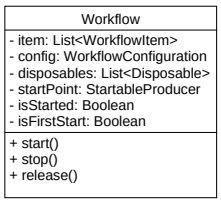


Figure 5.15: Workflow class.

The concept of workflows is part of the *Acquisition Case Designer*, an application which allows the reusability of already implemented components by an easy-to-use drag-and-drop user interface. This application is further discussed in section 5.5. Workflows depend immensely on reflection, in order to create a workflow of custom components. The workflow itself exposes methods to start and stop the workflow and release its resources. The `Workflow` class depicted in figure 5.15, utilizes a builder subclass, which actually builds the list of `WorkflowItem`s. A workflow must match certain constraints in order to be built. These constraints and a detailed documentation are listed in detail in section 5.5.

EmgFramework uses annotations to equip classes with compatibility to the workflow component. Available annotations are listed in section 5.5. Usually the workflow connects components with input and output ports. Components can either provide an input or an output port, or both. Annotations can augment casual classes with the ability to work inside workflows. The following section outlines the steps, which are necessary to transform an annotated class to a so-called `WorkflowItem`.

Workflow-Annotated Class processing

In order to use classes inside a `Workflow`, two independent steps must take place. Assuming the class `PeriodicRmsComponent` shown in figure 5.16 is eligible as a workflow item, as it incorporates all required annotations. At startup-time the framework will scan the classpath for annotated classes. Any annotated class exposes a component type. For example, the `PeriodicRmsComponent` will be of the type `EmgComponentType.RELAY`, which indicates, that the component has an input and an output port. During startup-time, the system extracts basic information of the component and creates a corresponding component object, in this case an `EmgRelayComponent`. Any of these components extend from the base class `EmgBaseComponent`. These components are mainly used for drawing objects, but they hold all required information for the final transformation step. After the user creates a workflow with the *Acquisition Case Designer* application, the workflow is verified and built.

At this point the system verifies if the workflow matches all its constraints and components can be connected. With the use of reflection each component is transformed into

[5] http://commons.apache.org/proper/commons-math/

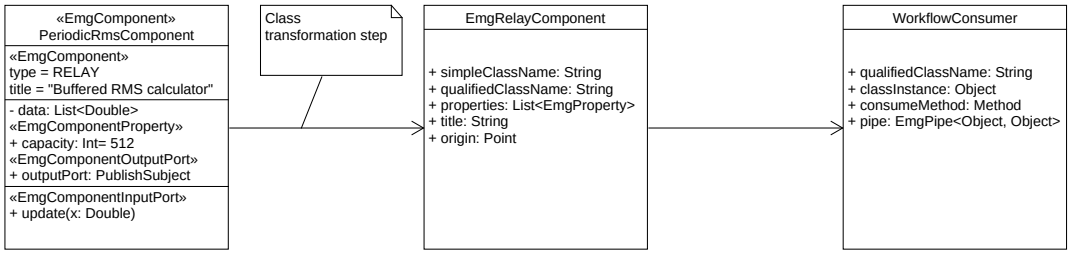


Figure 5.16: Workflow component class transformation.

either a `WorkflowConsumer` or `WorkflowProducer`. In this particular case it is assumed, that the `PeriodicRmsComponent` consumes data from the other connected component, therefore it is transformed into a `WorkflowConsumer`. In general a `WorkflowItem` holds one producer and several connected consumers.

While building the workflow, the components get connected. They are connected with a reflective approach outlined in listing 5.3. The framework verifies, that each producer field contains an output port of type `io.reactivex.PublishSubject`. This port pushes data to each subscribed consumer. Each consumer transforms the data with the corresponding pipe. After data transformation, each consumer invokes the corresponding update method with the transformed data.

```
val item: WorkflowItem
(item.producer.field.get(item.producer.instance) as PublishSubject<*>)
  .subscribe { data ->
    item.consumer.forEach { consumer ->
        val transformed = consumer?.pipe?.pipe(data)
        consumer?.method?.invoke(consumer.instance, transformed)
      }
  }
```

Listing 5.3: Connecting a consumer and producer via reflection.

This workflow mechanism pushes data as it is processed. The whole workflow works independently of each other. As soon as one component can publish data, it updates all connected subscribers with new data. The same principle applies to further connected components. These subscriptions will be automatically canceled, after the system is stopped by the user.

5.4 Producer Implementation

To not rely on Kotlin/Native in a beta version, the producer is implemented using *Android Things*, running on a Raspberry Pi 3. To fully describe the system, the section is parted into a hardware and software section.

5.4.1 Hardware

The utilized hardware is pictured in figure 5.17. The standard Raspberry Pi 3 does not contain an Analog-to-Digital converter (ADC), therefore the Explorer Shield 700

provides an external 8-Bit ADC. The Explorer Shield 700 is directly plugged onto the Raspberry Pi 3, which connects the ADC via the I2C bus. The actual EMG data sensing is performed by a MyoWare Muscle Sensor (AT-04-001), which is connected to a analog input pin of the shield and is powered by this very shield. However, the sampling frequency of the sensor is not provided in any data sheet. The heart rate provider is realized by an Bluetooth Low Energy (BLE) capable device. The Wahoo Tickr X allows the producer to connect via BLE to the heart rate sensor.

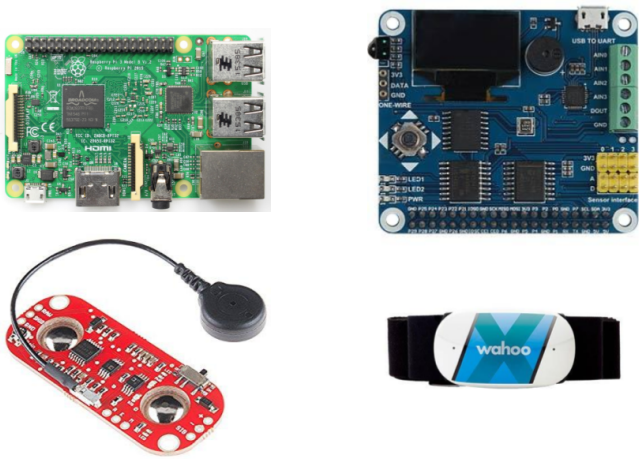


Figure 5.17: Utilized producer hardware.

Android Things eases the obstacle of an embedded device debugging, as the whole system is based on Android and therefore the log and a user interface is accessible via HDMI. Debugging can be done via ethernet. The Raspberry Pi 3 in addition with Android Things also allows data transfer via Wifi or Bluetooth.

5.4.2 Software

The behavior of the producer is already implemented in the abstract class `EmgClient`, which is part of the *client* module. The module *android_things* is a fully functional implementation of the *client* module. The module subclasses `EmgClient` within the class `AndroidThingsClient`. The module must provide suitable implementations for `EmgSensor`, to actually sense the EMG data, `EmgConnection`, which is an interface for communicating with the consumer site and `HeartRateProvider`, which is responsible for heart rate sensing. A class diagram for the core functionality of the producer is depicted in figure 5.18. Kotlin itself allows to implement abstract parent properties as constructor parameters. Therefore all three mentioned classes are injected via the constructor. The Android platform is only responsible for injecting dependencies into the `EmgClient` and start respectively stop the client on the corresponding lifecycle states.

The `EmgConnection` is implemented with Bluetooth and Wifi functionality. The de-

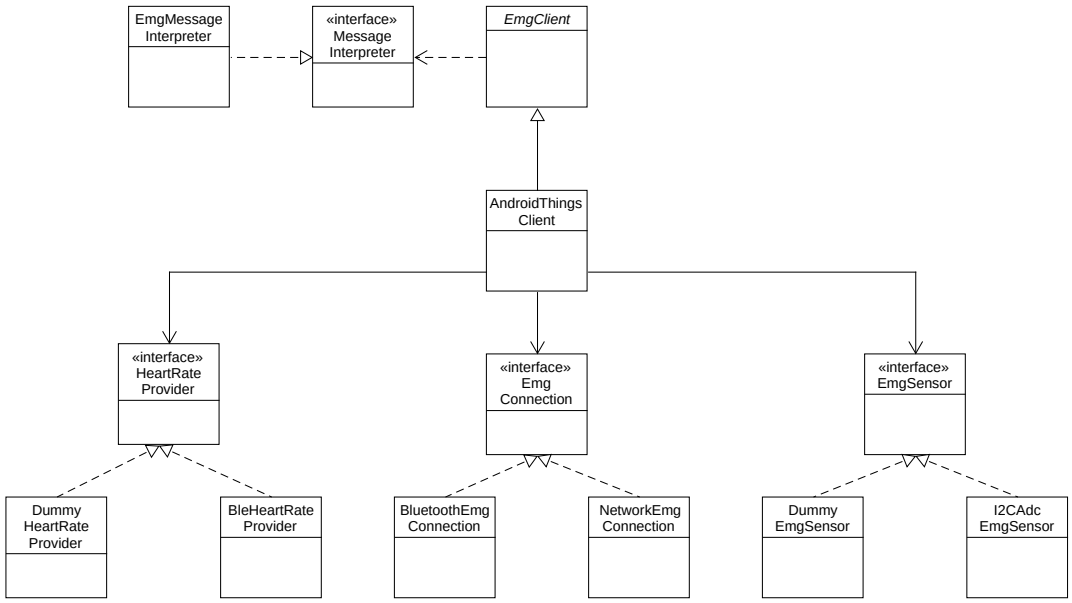


Figure 5.18: Producer class diagram.

pendency injection framework (in this case Dagger[6]) decides which connection is used. The same principle applies for `EmgSensor` and `HeartRateProvider`. The implementations `I2cAdcEmgSensor` and `BleHeartRateProvider` are used for production code. In order to test the system the framework provides dummy implementations of the heart rate provider and the EMG sensor. The code for Android Things can also run on handheld devices. In the *android_things* module this is realized by introducing two different build types: *things* and *phone*. The build artifact *phone* contains no dependencies to Android Things and uses the dummy implementations, but the behavior and the connection can be tested with a real device. The BLE connection to the heart rate provider is established using the reactive RxAndroidBle library[7], which senses the heart rate within an interval of one second. The EMG sensor is accessed via the Android Things driver library[8]. The AD conversion is in synchronization with the connections send frequency.

The `AndroidThingsClient` is started in the the `onStart()` respective stopped in the `onStop()` methods of the Android Things platform.

5.5 Acquisition Case Designer

In order to minimize the entrance barrier for users with a background others than programming, a usable and easy-to-learn UI should leverage the usage of *EmgFramework* applications. The user can work with blocks of functionality and can connect these

[6]https://google.github.io/dagger/
[7]https://github.com/Polidea/RxAndroidBle
[8]https://github.com/davemckelvie/things-drivers

blocks together, to build a customized workflow. To do so, the framework comprises a module called *casedesigner*. This module has two main functionalities. First, it provides the definition of the application's UI and business logic. And second, the tools and mechanisms to integrate plain source code artifacts (classes) within the UI. All this functionality leads to the application *Acquisition Case Designer*, which comes bundled within the desktop application.

The intention of the application is to offer all users – despite of their background – a blueprint, which enables the users to design workflows tailored to their custom use cases, while reusing already existing code artifacts. The user interface, as part of the desktop application is depicted in figure 5.19.

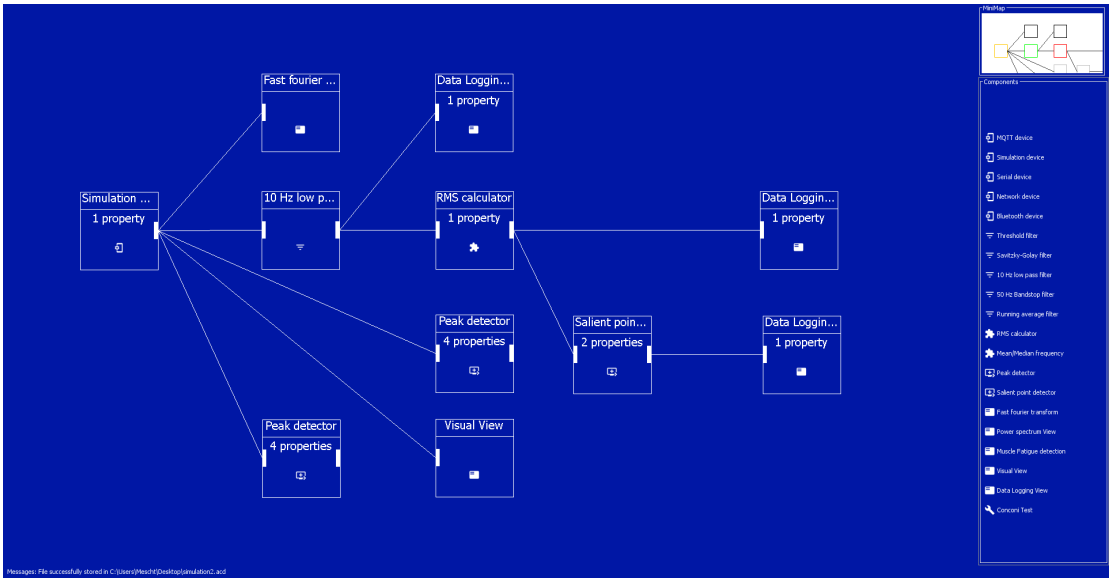


Figure 5.19: Acquisition Case Designer user interface prototype.

The system exposes five unique component types, namely: Filter, Sink, Relay Sink, Tool, Device and Relay. Sinks only consume data. Filter and Tools map the functionality of their corresponding classes. Typically views are of the type sink, but tools may also behave as sinks. The *Visual View* represents the sink in form of a graph, while the *Data Logging View* display data in form of a console. Relay Sinks indicate the user, that this component processes data, may show partial results in an independent UI, but is capable of sending results to connected components. Every computation can be implemented as Relay Sinks without UI. Components like the *RMS calculator* are actually of the type Relay Sink. Internally such computations use the concept of `BufferedComputation`s (see 5.3.3). In the actual implementation it is not transparent to the user which component exposes an independent UI.

The *Acquisition Case Designer* uses a so-called `Workflow` as the internal structure. Data is passed along blocks, which are connected with pipes. Due to the behavior a *Pipes and Filters* architectural pattern fits the actual use case.

The application should on the one hand leverage the usage of *EmgFramework* and increase reusability of the framework on the other hand. The user has the ability to

create complex workflows by reusing existing components. But also developers can integrate existing components into new components. The framework supports a reflective and plugin-based dependency injection. New modules are loaded at startup-time. In order to assure that a class is treated as an `EmgComponent`, it must annotate its class as an `@EmgComponent`. Annotations provide a more flexible way to make classes EmgComponent compliant. Basically all components are built at run-time, composed out of the corresponding annotations via **reflection**.

In theory the application can be deployed as a standalone application, but in order to incorporate already existing and compatible components, it must hold a dependency to at least the *core* and *clientdriver* modules, where these modules already integrate and adhere to the `EmgComponent` annotation of module *casedesigner*.

Annotations vs Interfaces

One architectural decision is to use annotations instead of interfaces as the primary mechanism to enforce compliance to the *Acquisition Case Designer* application. This decision shifts the policy enforcement and sanity checks to run-time, while interfaces could enforce the policies of the application at compile time. But interfaces come with the huge drawback, as they are less flexible for this particular use case, graphically explained in figure 5.20.

In general every component must declare itself as EmgComponent-compliant, by providing the utilized component type and the displayable name. The component type is essential for meta model (detailed described in 5.5.1) to enforce its policies and constraints. Some components may expose attributes to the user, which can be manipulated at run-time. Other components encapsulate a driver or comprise an additional UI or need to be started at first place.

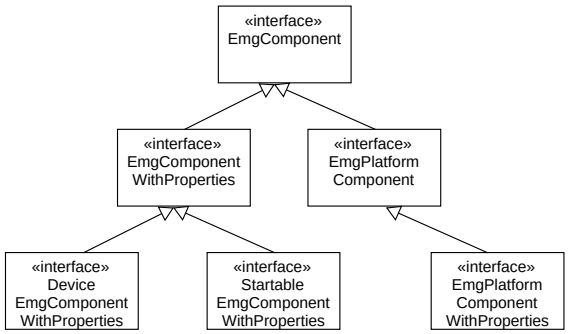


Figure 5.20: Interface structure providing the equivalent functionality as annotations.

Taking all these requirements into consideration, one will come up with an inheritance hierarchy shown in figure 5.20. But clearly, this inheritance hierarchy provides a bad flexibility and new features would let the hierarchy grow even more. Sheldon et al. concludes that deep inheritance hierarchies are good for reusability, but bad for maintenance. He suggests if maintenance is a key quality, one should sacrifice reusability by deep hierarchies for the sake of maintainability and replace it with shallow inheritance

hierarchies [67].

Annotations though can decorate the class and force compliance by not altering the implementation and therefore do not force developers to refactor existing source code, in order to be compliant. The weakness of interfaces is shown especially when a component wants to expose mutable properties to the user. This can be done either using callbacks mechanisms, which is bad, as it requires refactoring, or by using reflection, which would violate compile time policy enforcement.

Annotations provide a lightweight, yet fine-grained control of the components behavior, with the trade-off, that policies and the correctness of the component are enforced at run-time using **reflection**. Customs annotations can provide compliance by minimizing refactoring and without loosing flexibility or the dependency on a rather complex interface inheritance hierarchy.

Provided Annotations

EmgFramework provides a set of *casedesigner* annotations, which allow fine grained control of the component behavior. In order to define an arbitrary class to be compatible, it needs the `@EmgComponent` annotation. This annotation marks the class visible for the UI. This annotation is mandatory for all components. Other annotations are specific to the specified functionality. The annotation `@EmgComponentProperty` exposes adjustable component properties (String, Integer, Double) to the UI. An example would be the port and IP address of a network capable device.

```
@EmgComponent(val type: EmgComponentType, val displayTitle: String)

@EmgComponentProperty(val defaultValue: String, val displayName: String)

@EmgComponentInputPort(val consumes: KClass<*>)

@EmgComponentOutputPort(val produces: KClass<*>)

@EmgComponentPlatformView(val viewType: ComponentViewType, val viewWidth: Int)

@EmgComponentStartablePoint(val viewProperty: String, val viewClass: KClass<*>)

@EmgComponentEntryPoint

@EmgComponentExitPoint
```

Listing 5.4: Annotations provided by *EmgFramework*.

`@EmgComponentInputPort` marks the function, which consumes the output data of the previous component. The annotation must provide the consumable data type, otherwise the `Workflow.Builder` cannot verify if two components are eligible to be connected. The annotation `@EmgComponentOutputPort` marks the counterpart of the input port annotation. This annotation must mark a class field of type `io.reactivex.PublishSubject`, as this class implements an observable behavior.

`@EmgComponentPlatformView` is used for components, which require a platform specific UI. The workflow will allocate display space for the potion of UI. The policy enforces the annotated field to be either of type `javax.swing.JComponent` for desktop applications and `android.view.View` on the Android platform. Especially `Tools` require the

`@EmgComponentStartablePoint` annotation, in order to trigger the component's start method by the workflow.

`EmgClientDriver` has two distinct annotations: `@EmgComponentEntryPoint` and `@EmgComponentExitPoint`. Both annotations mark usually the corresponding `connect()` and `disconnect()` method of the driver. The workflow must know which methods to invoke, when reacting to a start or stop command.

The sample below in listing 5.5 illustrates the usage of annotations to register an independent class to be marked as an `EmgComponent` in the *Acquisition Case Designer* UI. The class receives a double value as input and stores it internally in a buffer. Once the buffer is full the class computes the *Root Mean Square* value of all values. The `PeriodicRmsComponent` publishes the asynchronous computation originally via a `PublishSubject`.

```
@EmgComponent(type = EmgComponentType.RELAY, displayTitle = "RMS calculator")
class PeriodicRmsComponent {

    @JvmField
    @EmgComponentProperty("512", "Values for calculation")
    var capacity: Int = 512

    @JvmField
    @EmgComponentOutputPort(Double::class)
    var outputPort: PublishSubject<Double> = PublishSubject.create()
        ...
    @EmgComponentInputPort(Double::class)
    fun update(x: Double) {
        ...
    }
        ...
}
```

Listing 5.5: EmgComponent extension for class `PeriodicRmsComponent`.

Instead of forcing developers to refactor their code, the solution with annotations does not alter the logical flow, yet registers and exposes the class to the *Acquisition Case Designer* application.

5.5.1 Meta Model

The meta model denotes the logical actors and constraints between each actor of a single workflow within the *Acquisition Case Designer*. It provides a conceptual view on the workflow, while imposing constraints, which are enforced at run-time during the workflow instantiation.

The meta model in UML notation, enriched with OCL, is shown in figure 5.21 and reflects a single, but prominent distinction between an interface based approach in comparison to the applied annotation & reflection based approach: The `EmgBaseComponent` is omnipotent. It may hold references to input and output ports, platform views and device driving logic as well. Though, its correct behavior is strictly enforced by constraints. Each subtype of `EmgBaseComponent` enforces its own set of constraints. Device drivers for example require mutual exclusivity between `EmgComponentInputPort` and `EmgComponentEntryPoint`, whereas the latter one indicates the driver component. A

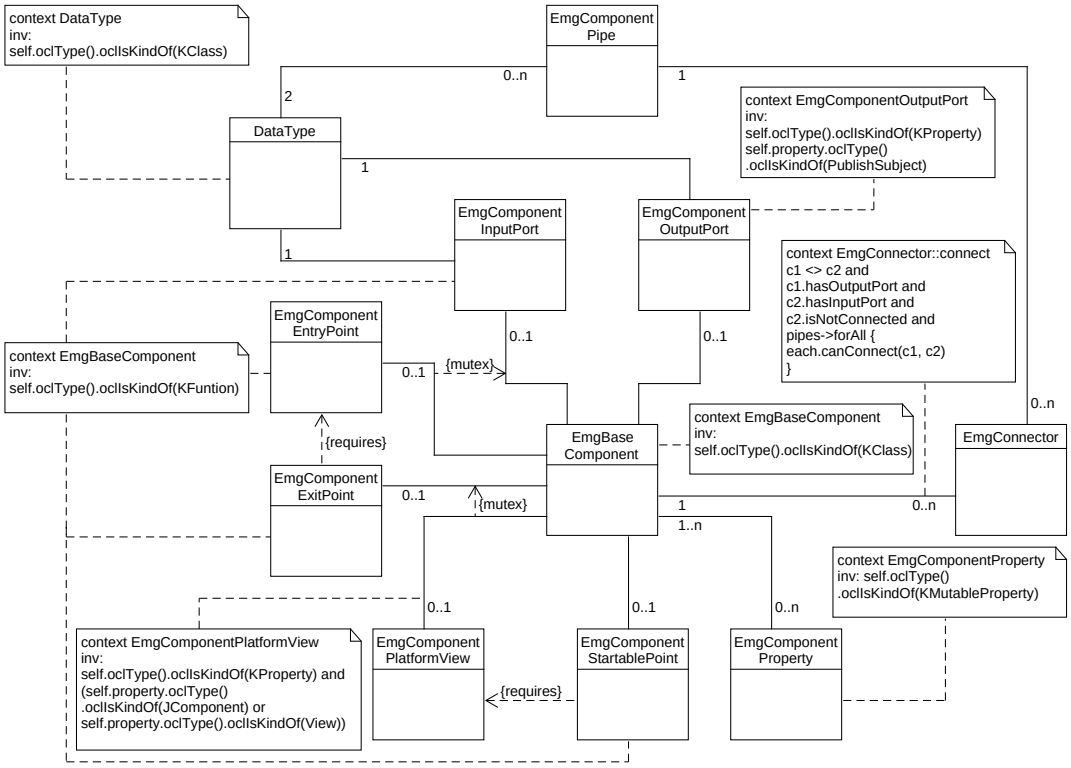


Figure 5.21: Acquistion Case Designer meta model.

driver component must not define an input port. Whereas UI-based components, which are using `EmgComponentPlatformView`, are mutually exclusive with device driver annotations[9]. Further, the meta model enforces strict type information. Annotated output ports must be properties of the type `io.reactivex.PublishSubject`, while platform views require to be either `javax.swing.JComponent` or `android.view.View`, depending on the target platform. In order to establish a connection between two components, the context of `EmgConnector::connect` must be evaluated to true, which takes the geometry of both components and the available pipes into consideration. The enforcement policy, that annotated types and data types must be of classes like `KClass`, `KFunction`, etc. are partially true, as Kotlin provides pure Java interoperability. Hence Java equivalents are supported as well.

[9]Device driver explicitly expose mechanisms for accessing a configuration UI.

Chapter 6

Evaluation & Results

In order to assure that the described architecture in chapter 5 fulfills its software quality requirements, measurements for the key software qualities are performed. These qualities are namely performance, portability, maintainability & modifiability and reusability. The measurements and the computed metrics build the foundation of the evaluation and the result part. The following evaluation is performed either on a source code level or at run-time.

The main purpose of the evaluation process is providing comparable metrics to describe the framework. Where data of *EMGworks by Delsys* is available, metrics will be compared. The evaluation data is compared internally as well. Just because the framework supports different protocols and different transmission technologies, this does not mean they are all equally performing well. The flexibility of the communication layer exposes superb evaluation cases for the `EmgClientDriver` and the `MessageInterpreter`. Although the system does not describe itself as a real-time system, timing is still a critical part of the user experience.

6.1 Performance

Although the system does not have any real-time constraints, performance is critical due to the fact, that the software should support mobile platforms as well. Mobile devices are equipped with a more potent hardware after each release cycle, but in general computation and battery are still the key factors a developer must address. But also for desktop applications the memory footprint should be kept low.

The performance evaluation is split into two parts. The first part is concerned about the performance of the driver interface. A throughput test is used, in order to verify the drivers performance of packet handling. The second part addresses the memory and CPU footprint of both supported platforms. The desktop application metrics are also compared with the *EMGworks* system requirements.

6.1.1 Driver Latency

The driver latency evaluation tests the driver implementation with different data formats and through different mediums. The framework provides driver for the serial interface, Bluetooth, simulation, network (UDP), and MQTT. The framework also supports three

different types of `MessageInterpreter`s. Each driver is tested with each type of interpreter.

The exchange of the `MessageInterpreter` is a simple way to tune performance, as the communication link implementation does not change, while the message protocol format changes.

Test Setup

The test setup follows a suggestions of Martin [51, p. 317]. The producer device sends data for 10 seconds with 1000 Hz sampling frequency. This equals to 10.000 data packets. Each packet provides 1 data channel of EMG data and supports the protocol version v3, including heart rate. The test is performed for all five `EmgClientDriver` and for all three `MessageInterpreter`. Each driver uses an individual setup.

Serial An Arduino Mega ADK with 16 MHz clock speed is used as the producer device. The serial port is configured with 8 data bits and 1 stop bit. Baud rate is set to 230400.

Bluetooth An Android test app implementation assures a separated producer and consumer device. The test app runs on a Nexus 5 with Android 6.0.

Simulation This client has a special position inside the framework, as it is a playback client, with no dedicated producer host device. Therefore the client is running local and in-memory.

Network Uses the same test app as Bluetooth, but uses a UDP library for communication (Nexus 5, Android 6.0).

MQTT Same as the network test setup (Nexus 5, Android 6.0). The quality of service (QoS) for MQTT forces an exact once delivery (QoS = 2).

	Serial	Bluetooth	Simulation	Network	MQTT
JSON protocol	52.561s	47.484s	10.000s	10.187s	118.772s
Self-defined protocol	53.094s	50.085s	10.004s	10.035s	115.175s
Protocol Buffers	-	59.700s	10.002s	10.018s	122.912s

Table 6.1: Desktop Driver latency comparison.

Due to the overall bad performance of the serial interface, Protocol Buffers for serial communication were actually not implemented.

6.1.2 Resource Utilization

Performance evaluation contains RAM and CPU footprint as well. RAM and CPU footprints are measured with VisualVM[1] for the desktop application, and with the developer tools of Android Studio[2] for the Android app. Footprints are captured for idle state (minimum user interaction) and operating state. Data is recorded and averaged over a period of 10 minutes. The operational state comprises of data playback with the simulation driver, applying three filters with additional visualization. The test environments for both supported platforms are as follows:
Desktop: Asus Zenbook UX32VD, Windows 10 64-bit, 10 GB RAM, Intel Core i5-3317U @ 1.7 GHz.
Android: LG Nexus 5, Android 6.0 32-bit, 2 GB RAM, Snapdragon S800 @ 2.3 GHz.

	Desktop application	Android app
Executable file size	22.9 MB	13.7 MB
RAM (idle)	50 MB	85 MB
RAM (operating)	120 MB	150 MB
CPU (idle)	1%	0%
CPU (operating)	25%	30%

Table 6.2: Performance comparison of application.

CPU usage is for both platforms in between 0 - 30% of the utilized processor. The memory footprint of the desktop application exceeds the footprint of the mobile app. The measured values for the desktop application are compared with the minimum system requirements of *EMGworks*[3], in order to put the metrics into relation.

	EmgFramework desktop	EMGworks
Operating system	Windows, Linux, MacOS	Windows
System memory	64 MB	2 GB
RAM	512 MB	N/A
CPU	1.5 GHz	2.0 GHz
Graphics memory	N/A	128 MB

Table 6.3: Resource comparison of EmgFramework desktop and EMGworks.

Based on the measurements of table 6.2 for the desktop application, the recommended system capabilities are stated in table 6.3 and are directly compared to those of *EMGworks*. The peaks of maximum load reached 250 MB of RAM and 70% of the 1.7 GHz CPU. Hence, the recommended values consider an adequate margin. No data was available for the RAM requirements of *EMGworks* and the minimum graphics memory of the *EmgFramework* desktop application.

[1] https://visualvm.github.io/
[2] https://developer.android.com/studio/
[3] https://www.delsys.com/products/software/emgworks

6.2 Portability

Albeit *Portability* is part of the ISO 25010 family, there is no clear metric on how it can be measured. In general it is defined as the degree of efficiency to which a system or software component can be transferred from one execution environment into another execution environment [34]. However, James Mooney defined a formula for calculating the portability of a software module [56]. The formula relates the costs of porting with the costs of redeveloping the whole system and is defined as

$$DP = 1 - \frac{c_{port}}{c_{redevelop}}. \tag{6.1}$$

If the value is bigger than 0, it indicates, that porting the software is more cost-effective than redeveloping. A value of 1 indicates perfect portability, while a value smaller than 0 suggests redevelopment over software porting.

Costs are most likely defined as money or time. Hence usual cost calculation is not feasible for the project. Based on the formula the calculation is altered, that the costs are the concrete files respectively the lines of code, which need to be rewritten for a new platform. Table 6.4 lists and maps the file structure for the concrete implemented code.

	Desktop	Android	Android Things
Views	23	23	2
Driver	3	1	-
Misc	10	9	13
Classes	36	33	15
LOC / Class	93	77	56

Table 6.4: Class listing and mapping for desktop & Android application and Android Things producer device software.

Table 6.4 clearly outlines the fact, that most platform specific code belongs to the view implementations. Around 66% of the implemented code is concerning about the views. In average 35 classes need to be ported (or newly implemented) to ensure consumer applications to run on the new platform. The Model-View-Presenter pattern in combination with Kotlin as the main programming language allows a strong multi-platform support and reusability of the *core* module.

However, the drawback of these metrics is, that they cannot be put into relation within the whole framework implementation. Based on the equation 6.1 defined by Money, an adapted formula is applied, in order to compute the percentage of platform independent source code. The value PR (portability rate) is defined as

$$PR = \frac{f_{developed}}{f_{developed} + f_{avgported}}. \tag{6.2}$$

It takes the altogether developed files as the numerator and divides it by the developed files plus the average amount of ported files. The average is necessary, because multiple platform implementations would falsify the result, as the metric defines the percentage

for a single new platform port.

A more fine grained metric replaces the ported files with the ported lines of code, formulated as

$$PR_{LOC} = \frac{LOC_{developed}}{LOC_{developed} + LOC_{avgported}}. \tag{6.3}$$

Classes can vary in the implementation complexity. Less, but complex classes would falsify the formula 6.2 in relation to the actual porting costs.

The portability rate of the system $PR = 83\%$, while the LOC weighted portability rate $PR_{LOC} = 78\%$.

6.3 Maintainability and Modifiability

For the sake of active development and improvements a system must exhibit the software qualities of maintainability and modifiability. A set of metrics is utilized, in order to evaluate these qualities. The *Martin metrics* check the overall maintainability of the modules. This set of metrics evaluates if the project structure and module dependencies fulfill the requirements for a maintainable system. The *Average LOC metric* for extensible classes offer a prediction of the effort to extend and modify the system. Lastly the *MOOD metrics* set check the overall system design in terms of object oriented design.

Martin metrics

Martin describes the metrics as part of the stable abstractions principle [50, pp. 122-131] and calls it the *Martin metrics* [49]. The metrics operate on module level and compute the instability and the abstractness of each module. Instability is defined as

$$I = \frac{fan_{out}}{fan_{in} + fan_{out}}, \tag{6.4}$$

whereas fan_{out} describes the count of outgoing dependencies (import statements in other modules) and fan_{in} as the incoming dependencies, where other modules import the functionality of the current module. A value of $I = 0$ indicates a maximum of stability [50, p. 122].

The abstractness of each module is defined as

$$A = \frac{f_{abstract}}{f_{count}}, \tag{6.5}$$

where the abstractness is the quotient of the abstract files in a module divided by all files in the module [50, p. 127]. These two values should be balanced for every component. However, the dependency injection mechanism of *EmgFramework* does not directly reference artifacts of the *core* and *clientdriver* package directly. Referencing is accomplished by the usage of reflection at run-time.

Next to that, Martin defined the *Main Sequence*. Modules should whether be fully stable and fully abstract (coordinates [0, 1]) or fully instable and fully concrete (coordinates [1, 0]). The *Main Sequence* is the line between those two points. Balanced modules

should lie as close as possible to the main sequence [50, p. 131]. The distance to the main sequence is defined as

$$D = |A + I - 1|. \tag{6.6}$$

The values (I, A, D) of each module are listed in table 6.5. The computed metrics are

	Instability	Abstractness	Distance
android_things	1	0.083	0.083
android	1	0.273	0.273
casedesigner	0	0.426	0.574
client	0.154	0.667	0.179
clientdriver	0.045	0.667	0.283
core	0.526	0.356	0.118
desktop	1	0.028	0.028
messaging	0	0.25	0.75
SD	0.473	0.238	0.252

Table 6.5: Source code abstractness and instability metrics.

feasible of statistical analysis. Therefore the standard deviation is calculated for all three metrics. The metrics are visualized for all modules in figure 6.1. The grey lines indicate the standard deviation of the distance metric D. Most modules should preferably lie inside the standard deviation corridor.

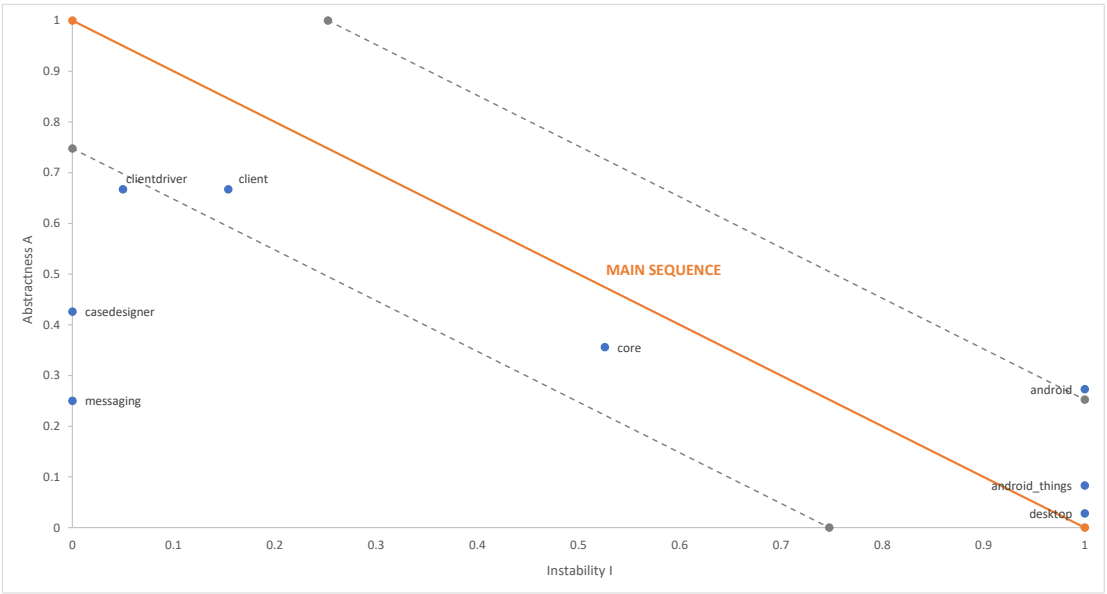


Figure 6.1: Martin code metrics visualization.

The modules *casedesigner* and *messaging* value of distance D of 0.574 respectively 0.75 require further consideration.

Average Lines of Code of Extensible Classes

A simple prediction metric is based on the average lines of code of the intended extensible classes. Modifiable classes are parted in three groups: `Filter` and `Tools` are use case oriented, while `EmgClientDriver` and `MessageInterpreter` are message oriented and `EmgBaseComponent` and `EmgComponentPipe` are case designer oriented classes.

	Implementations	Average lines of code
Filter	5	29
Tool	4	112
EmgClientDriver	6	97
MessageInterpreter	3	56
EmgBaseComponent	7	13
EmgComponentPipe	9	9

Table 6.6: Modifiability metrics regarding average lines of code for extending key features of *EmgFramework*.

This metric can help to predict future work. For example a company wants to extend *EmgFramework* with a custom client driver of existing hardware, which utilizes a different form of message interpretation. The data of this client should be evaluated using a custom tool. The company can now predict the rough amount of LOC needed for their use case. The prediction is based on the already implemented source code. The formula of this particular example is defined as

$$LOC_{custom} = LOC_{EmgClientDriver} + LOC_{Tool} + LOC_{MessageInterpreter} = 265. \quad (6.7)$$

MOOD Metrics

In 1995 Fernando Brito e Abreu introduced the metrics of object-oriented design - MOOD [1]. This suite of six metrics describe the quality of the object-oriented architecture in a comparable manner. Object-oriented languages are based on three key mechanisms: Encapsulation, Inheritance and Polymorphism [50].

The *Method Hiding Factor (MHF)* and the *Attribute Hiding Factor (AHF)* metrics measure the leverage of the encapsulation and information hiding feature of object-oriented languages [35]. It measures the visibility of fields and methods from one class to the others. The MHF should not be kept too low, as an increase in the MHF value indicates a decrease of bug density and therefore increases the software quality [54].

The metrics *Method Inheritance Factor (MIF)* and *Attribute Inheritance Factor (AIF)* offer a way to inspect the ratio of inherited methods and all declared methods in all classes. These metrics can be too high, if too much inheritance is used and the inheritance trees grow, whereas interfaces reduce the MIF value [2].

Polymorphism and coupling is covered by the *Coupling Factor (CF)* and the *Polymorphism Factor (POF)* metric. The polymorphism factor analyzes the inheritance tree and describes the amount of situations, where a method can be invoked by several descendants. The coupling factor is a metric, which describes the knowledge of a particular class of the existence of other classes. Inheritance is the tightest form of coupling. The

child class is tightly coupled to the parent class. Ideally the coupling factor is quite low, this increases reusability, because the class is not dependent on other classes. Loose coupling is reached via interface definitions [2, 35]. Data for already analyzed projects like

	MFC	GNU	Motif	**EmgFramework**	Optimal range
MHF	24.6%	13.3%	39.2%	**12.06%**	12.7 - 21.8%
AHF	68.4%	84.1%	100%	**87.25%**	75.2 - 100%
MIF	83.2%	63.1%	64.3%	**43.88%**	66.4 - 78.5%
AIF	59.6%	62.6%	50.3%	**61.22%**	52.7 - 66.3%
CF	9.0%	2.8%	7.6%	**5.24%**	0.0 - 11.2%
PF	2.7%	3.5%	9.8%	**-**	2.7 - 9.6%

Table 6.7: MOOD metrics with comparison.

the Microsoft Foundation (MFC) or Motif are provided by Aivosto [2] and is outlined with the metrics of *EmgFramework* in figure 6.7. Al-Ja'afer et al. suggests a percentage range for each metric, based on previous work [35].

Due to the shallow inheritance hierarchy[4] the polymorphism factor was not computed. Nearly all metrics lie in the suggested range. MHF still lies in the suggested tolerance zone. The Method Inheritance factor is the only metric, which lies far outside the suggested range. This is due to the fact, that EmgFramework makes comprehensive use of interfaces, whereas interfaces lower the MIF metric.

6.4 Acquisition Case Designer

Testing the *Acquisition Case Designer* in terms of usability is neither trivial nor would there be a comparable data set for other applications. The focus lies on the increase of reusability of existing software components. The *casedesigner* module provides additional functionality to augment the original supported functionality. It opts for minimum impact of the core system, yet providing rich enhancements and increasing reusability. The small impact on the core system is going to be evaluated. The impact is measured in terms of introduced code overhead to the foundation modules.

In general all business logic of the *Acquisition Case Designer* is bundled in the *casedesigner* module. External written plugins may opt-in to depend on the module, in order to render their components compliant. The core system utilizes an array of 20 reusable `EmgComponents` and 14 `EmgConnectionPipes` from the start. Pipes are a necessary drawback of the architecture, as two components must be connected with a suitable pipe, which transforms one output data type to the next consumable input type. Hence it's possible that some `EmgComponent`s are rendered useless for production, if no suitable pipe is provided at run-time.

The evaluation considers the additional amount of work in terms of averages lines of code per component family. A component family describes the key features, which are extensible, but must not explicitly fall into this pattern. *Filter*, *Driver* and *Tool* are familiar concepts of *EmgFramework*. Though, *Pipe*, *View* and *Misc* are newly in-

[4] max. inheritance depth = 1

troduced component families. The evaluation takes the average amount of additional lines of code, which are necessary to make the artifact EmgComponent-compliant. The compliance is achieved by adding `@EmgComponent*` annotations.

Component Family	Average additional lines of code
Pipes	14
Driver	5
Filter	6
Tool	5
Analysis	2
View	4
Misc	5
All	**4.65**

Table 6.8: Average additional lines of code per each component family.

The results vary between 2 - 6 additional lines of code for components, but with the fact, that no additional code refactoring is necessary. The average additional lines of code per components is **4.65**. Each component needs not more than 5 additional lines of code to comply to the component definition.

Whereas an `EmgComponentPipe`, which transform one data type into another consumable type, requires 14 additional lines of code. Components define exactly one data type for consumable data. The framework must ensure, that there are suitable pipes for new components in order to connect them. If plugin developers introduce new data types, they must provide suitable pipes in the plugin as well. Each new type conversion requires new class instances of `EmgComponentPipe`, which indicates a clearly not scalable and maintainable behavior.

Chapter 7

Case Study: Evaluating EMG Data During an Isotonic and Isometric Exercise with a Network Connected Device

The actual use case is split into two parts to examine the strengths and weaknesses of the framework in detail. One test addresses an isometric or static exercise, where the subject sits on the chair with stretched legs and a 5kg weight on the end of the legs and tries to hold this position as long as possible. Isotonic exercises refer to exercises with a dynamic nature, like weightlifting or running [52]. The second test exercise comprises a Conconi test [12].
The two tests should prove the applicability of the system. The setup of the test requires a supervisor, who takes care of the measurement during the physical activity of the test subject. He coordinates data acquisition and a proper test environment during the exercises. The application is built with the *Acquisition Case Designer* tool, in order to outline the strengths of the system in terms of extensibility.

The isometric test setup schematic is shown in figure 7.1. The schematic refers to the notation used in the software. Note: The data source is a simulation device, which means play back data. This is used to ensure replicable results.

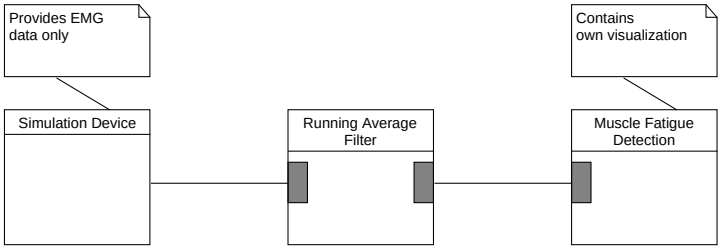


Figure 7.1: Schematic of the corresponding *Acquisition Case Designer* application for the isometric test setup.

The test setup contains a device, which sends data filtered to a muscle fatigue detection tool (which internally uses the JASA algorithm – see section 2.2.3).

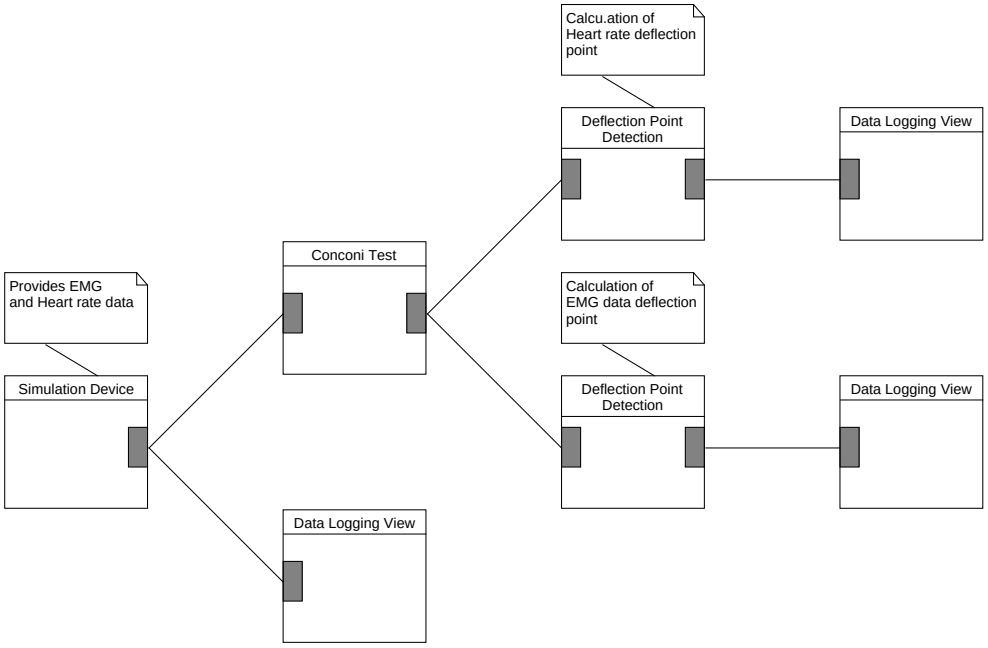


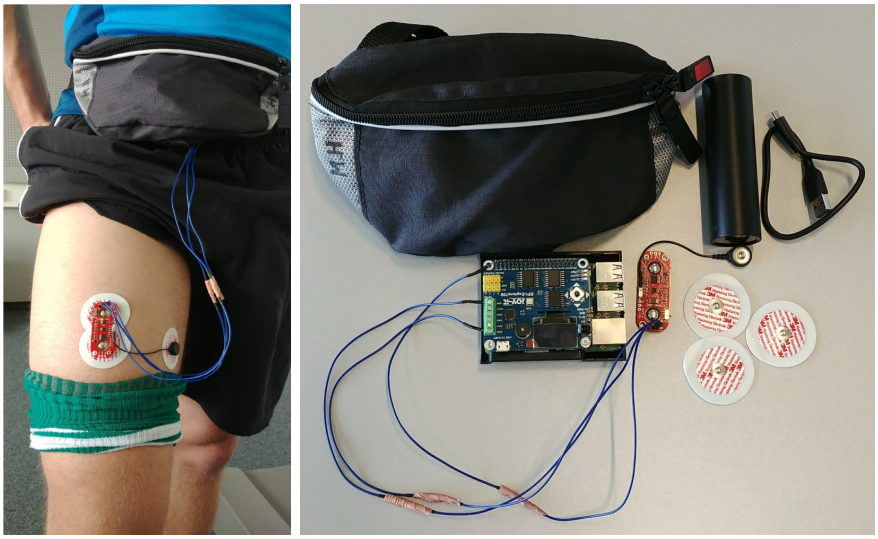
Figure 7.2: Schematic of the corresponding *Acquisition Case Designer* application for the isotonic test setup.

The isotonic system under test is depicted in figure 7.2. The application conducts a Conconi test and sends data further to deflection point detection tools, which examine the deflection point of heart rate and EMG signal in respective. *Data Logging Views* are added for convenience for the test supervisor.

7.1 Setup

The Conconi test is a lactate step test and is conducted on a treadmill. The test subject is equipped with a belt bag, which contains a Raspberry Pi with the responsibility of data acquisition. The embedded system runs the reference implementation of the *android_things* module targeting the Android Things platform. The device is connected to the consumer application via the network interface. Sampling frequency of the device is set to 1000 Hz. An electrode is placed on the upper leg of the test subject and connected via cables to the Raspberry Pi.

The isometric exercise should provoke muscular fatigue in a short period of time. The isotonic test comprises a Conconi test, which investigates the trend of the heart rate over time. After each step the last heart rate value and the last EMG value is taken as the representative of the step. The EMG signal is filtered with a running average filter over a time window of five seconds. The Raspberry Pi provides an interface for accessing Bluetooth Low Energy paired heart rate provider devices. The test subject wears a *Wahoo Tickr X* chest belt. The utilized components of the test setup is further outlined in figure 7.3.



(a) Electrode wiring.

(b) Utilized EMG components.

Figure 7.3: Test subject setup with electrode placement and wiring 7.3a and the set of utilized EMG measurement components 7.3b (heart rate components excluded).

7.2 Procedure

The introduction already mentioned a supervisor, which assist the test subject while conducting the test. For the isometric test, the supervisor places the weight onto the stretched feet and starts and stops the data acquisition. For the Conconi test the supervisor switches the tempo and observes the acquired data.

The test procedure for the isometric test emphasize high muscular activity at a short period of time. Usually subjects will hold the position wit stretched legs for two to four minutes. Therefore the JASA algorithm is adapted. Luttmann et al. suggests a window of 5 seconds for Median frequency computation and a 10 seconds window for Root Mean Square computation. This accumulates in 12 frequency feature per minute and 6 time feature values per minute. Two linear regressions are built based on these feature vectors. The quantitative muscle fatigue indicator is the pair of slopes of both regression lines [48]. However, due to the small time frame of maximum four minutes this is not applicable. Therefore the algorithm uses a 2 seconds time window for frequency domain features and a 5 seconds time window for time domain features. The algorithm also decreases the regression time from 60 seconds to 20 seconds, in order to retrieve more quantitative features.

The procedure for the Conconi test follows its protocol: The test subject runs on a treadmill and every 200 meters the test supervisor increases the tempo by 0.5 km/h. The test starts at 10 km/h and is continued until the test subject reaches the point of total exhaustion. Muscular fatigue measurement is not applicable, due to the isotonic exercise behavior.

Both tests were conducted by a group of six test subjects within the age of 21 to

25. Each subject was familiar with running and cardiovascular exercises. No untrained subjects were examined. A detailed overview of the subjects is seen in table 7.1.

	Age	Height (cm)	Weight (kg)	Gender
Subject 1	22	176	77	M
Subject 2	21	184	70	M
Subject 3	22	183	85	M
Subject 4	22	185	76	M
Subject 5	25	174	69	M
Subject 6	25	182	82	M

Table 7.1: Tabular overview of the test subjects.

7.3 Components Under Test

The *Acquisition Case Designer* builds a new application, every time a workflow is spawned. Each workflow utilizes an arbitrary set of components, exposed through the *casedesigner* module. The case study applications utilize following components:

- Simulation Device,
- Data Logging View,
- Conconi Test,
- Deflection Point Detection,
- Muscle Fatigue Detection.

The simulation device and data logging view are irrelevant for the test scenario. The other three components will be explained in detail.

Figure 7.4 offers a dynamic view on the system and how the components interact with each other. For overall convenience it combines both test setups into one view. During the test, the muscle fatigue object does not run at the same time as the Conconi tool or the deflection point tool. The reader should retrieve insights into the rather complex use case when all three components would interact with each other.

The whole application is built upon a reactive approach. This means, that the Main lifeline tells the device to start sensing. The device asynchronously updates the Main object if new data is available.

After data is received on Main, it is pushed to the connected components via their exposed `update()` method. Note that the components `RunningAverageFilter` and `DataLogginView` are not outlined as separate lifelines for the sake of a more readable diagram. Both components perform trivial tasks, which do not need further explanation.

The Conconi and the MuscleFatigue objects are update via the Main object. The Conconi object only updates the DeflectionPoint object if a step is finished. Both MuscleFatigue and DeflectionPoint objects shift the time-consuming computation of their work off to a worker thread.

The user can terminate the application, which will cause the Main object to disconnect with the device. Thus the loop terminates and the lifelines of the objects end.

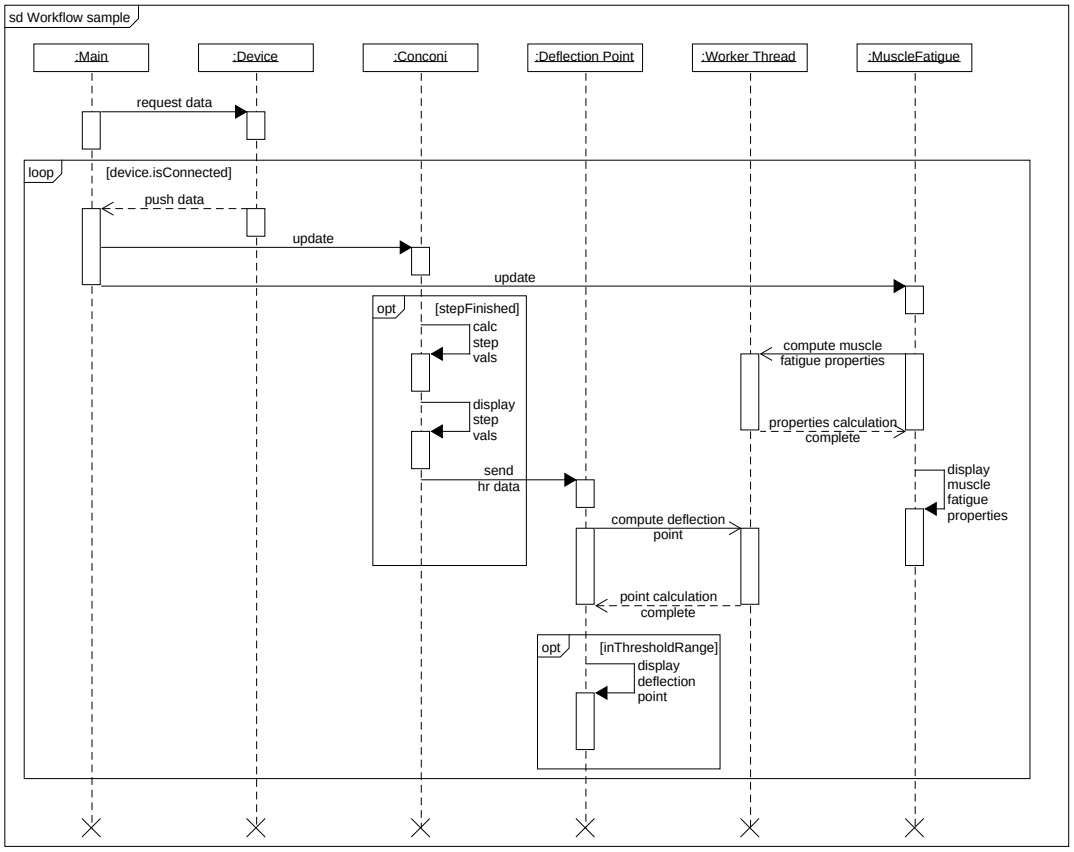


Figure 7.4: UML Sequence diagram of the combined application of both test cases.

The Conconi Tool

The Conconi Tool per se cannot be evaluated, as there is no measurable data available. In the particular use case of applying the isotonic workout, the verifiable data is provided by the deflection point detector. The Conconi Tool itself is limited to the functionality of partitioning the incoming data into steps of varying length. For the sake of reusability, deflection point detection is performed in a reusable tool implementation. Hence the lack of measurable and verifiable data.

The Muscle Fatigue Tool

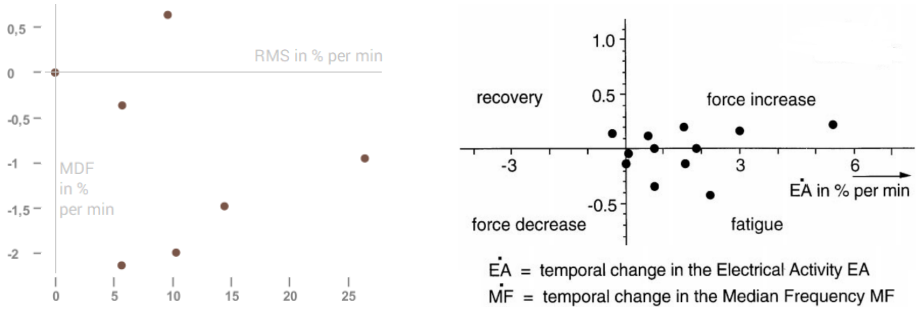
The isometric test setup tries to force the muscle to fatigue in a time frame of two to four minutes and measure this fatigue with the implementation of the JASA algorithm. For the sake of clarity the last captured data point of muscular fatigue indication is shown in table 7.2 with the corresponding muscle state. One should recall the meaning of the four quadrants in the two dimensional space of muscular fatigue indicators in the JASA algorithm (figure 2.4).

Four of the six subjects show signs muscular fatigue during the data recording. One reflects the increase of force, while another shows signs of the decrease of force.

	Last RMS value	Last MDF value	Muscle state
Subject 1	27.12	1.77	Force Increase
Subject 2	127.06	-0.79	Fatigue
Subject 3	-36.04	-1.99	Force Decrease
Subject 4	48.91	-0.58	Fatigue
Subject 5	10.33	-1.99	Fatigue
Subject 6	61.99	-2.83	Fatigue

Table 7.2: Settling point of JASA algorithm values per subject with corresponding meaning.

An ideal final muscle fatigue detection by the JASA algorithm is shown in figure 7.5b. The temporal change of the time domain is plotted on the x-axis, while the temporal change of the frequency domain is plotted on the y-axis. The muscular fatigue detection over time of subject 5 is shown in figure 7.5a. Figure 7.5a also indicates the shift of the point cloud into the lower right quadrant. The JASA algorithm outlines this as the quadrant of muscular fatigue.



(a) JASA interface of *EmgFramework*.

(b) JASA example by Luttmann [48].

Figure 7.5: Custom implementation (7.5a) compared to a sample of Luttmann (7.5b).

The framework is able to detect four times out of six muscular fatigue. The subjects had to perform the isometric exercise as long as possible. Subjects may give up earlier, but nevertheless, a trend towards muscular fatigue should be visible. Data sets were analyzed as a whole, meaning that the last point is not an outlier of the data set. The last point follows the trend of the previous data points. For the sake of clearness only the last data point is depicted, as it is a representative of the data set.

The Deflection Point Detection Tool

The tool exposes the capability to find the deflection point of an arbitrary series of values. The tool neglects units and time information, only the discrete series of y-values is taken into consideration. The user interface of the tool is depicted in figure 7.6, showing the deflection point of a time series of EMG values respectively heart rate values, as an outcome of the Conconi Test.

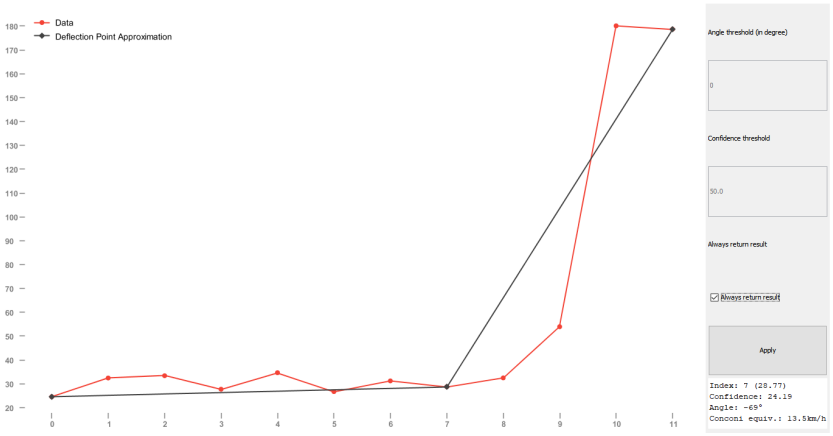


Figure 7.6: Deflection point detection of EMG values on the y-axis and discrete time values on the x-axis.

The results for the six participants is listed in table 7.3. The columns denoted with the prefix *Dmax* indicate the reference value of the Dmax algorithm. The reference values using the Dmax algorithm are computed by a software suite provided by Newell et al. [58][1]. The differences between the Dmax algortihm and the framework's implementation are outlined in detail in section 5.3.3.

	HR Dmax	HR Tool	Diff	EMG Dmax	EMG Tool	Diff
Subject 1	12.28	15.00	2.72	13.42	14.50	1.08
Subject 2	12.12	13.50	1.38	12.76	13.00	0.24
Subject 3	11.43	11.50	0.07	12.91	12.50	0.41
Subject 4	12.36	12.50	0.14	14.55	14.00	0.55
Subject 5	13.83	14.00	0.17	13.76	13.50	0.26
Subject 6	11.32	11.50	0.18	12.86	11.50	1.36
SD	0.90	1.41	**1.07**	0.69	1.08	**0.46**
AVG	-	-	**0.78**	-	-	**0.65**

Table 7.3: Comparison between Dmax calculated deflection points and the deflection points calculated by the Deflection Point Tool.

The left columns list the results of the heart rate deflection points, while the right columns show deflection points of the EMG data. The table also provides a dedicated column for the absolute difference between the observed and the reference value. The value itself represents the tempo of the Conconi test. Thus, 12.28 indicates the deflection point is during the tempo of 12.28 km/h. Indices of the vector would be less descriptive, therefore the table represents the tempo of the test steps. However, there is no actual index of 12.28, because steps have a discrete size of 0.5 km/h. Hence the deflection point lies between the steps with the corresponding tempo of 12 km/h and 12.5 km/h.

[1] R-scripts for Dmax computation are available at http://www.nuigalway.ie/maths/jn/Lactate/html/lactate-r.html.

The evaluation of the deflection point detection tool aims for the minimization of two metrics. First, the difference between the measured and the reference value should be minimal. The metric targets accuracy. Due to the limitations of the implemented framework's algorithm, only discrete x-values can be marked as the deflection point. Therefore all values are either integers or half's. Because of this shortcoming an exact approximation of the algorithm is unlikely.

The second metric exactly covers this shortcoming. It computes the standard deviation of all absolute differences, highlighted in the last row. The assumption of this metric is, that due to the discrete x-values, an exact approximation cannot be made. But if all approximations have a similar distance to the reference data, the framework provides at least precise results.

The results lie in an acceptable range. Test subject 1 is an outlier compared to the other participants. The average absolute difference values are **0.78** for heart rate deflection point errors and **0.65** for EMG deflection point errors. Keeping in mind, that an error of 0.5 means that the algorithm classified the adjacent data point as the deflection point. This results in an index error of 1. Therefore the accuracy lies in the range of 1-2 misclassified x-values. Precision is higher within the EMG data set, leading to a standard deviation of the absolute difference of **0.46**. The EMG data set has less fatal outliers than the heart rate data set with the standard deviation of the absolute difference of **1.07**.

7.4 Results

The results of this case study are parted into two sections. The first part addresses the results of the data evaluation, the second part focuses on the evaluation of the sensing environment as a whole.

The Conconi Tool did not provide any data to evaluate. The simple functionality of partitioning data was not further investigated. The muscle fatigue detection tool showed acceptable results, with a muscle fatigue detection rate of four out of six participants. The test setup of the isometric test may be improved by using an exercise which lasts over a longer period of time. Luttmann et al. investigated muscular fatigue of surgeons during operations within a time span of 18 - 83 minutes [48]. An improved setup may lead to more significant results.

The deflection point tool detected the deflection point within the range of one to two data points. The results are compared to the reference algorithmic implementation of a Dmax deflection point algorithm by Newell et al. [58]. The deflection point algorithm of *EmgFramework* should consider using the Dmax algorithm instead of the simple two line regression approximation with a predefined deflection point. The selection of the deflection point as a single value in the data set must be discussed critically, as it imposes constraints to accurate deflection point calculation.

The sensing environment exposes critical flaws on the producer site. Using Android Things as the primary platform for producer devices must be carefully reconsidered. Not only that it is infeasible to reliably connect to an external heart rate sensor, it is also not possible to maintain a stable sampling frequency of the data, which leads to

sporadic data transfer. This problems may be caused from an insufficient performant implementation, rather than incapabilities of the system. However, both outlined points are crucial for the overall system. It is noteworthy that sampling frequency was sporadically established at the required level, but during other sessions dropped to a level of 200 - 400 Hz. Heart rate data acquisition was only applicable using external devices, although the producer implementation would be capable of connecting with Bluetooth LE devices.

At least one should mention the overall positive experience with the *Acquisition Case Designer*, which provides proper functionality for building complex workflows within a short period of time. A whole sensing environment can be set up and configured in an interactive and flexible way.

Chapter 8

Conclusion

It's noteworthy to mention, that the system has not been gone through an architectural transformation, therefore the conclusion is mainly based on the evaluated result data. The framework has never been evaluated by the means of a suitable software architecture by a third party. In general there are a set of scenario-based evaluation methods, such as *Scenario-based Analysis Method (SAAM)*, *Architecture Tradeoff Analysis Method (ATAM)* or *Active Reviews for Intermediate Design (ARID)*. Babar et al. [3] introduced a framework for choosing the right evaluation method in order to apply a architectural transformation based on the outcome of the suiting method.

8.1 Implementation Scope

The rather small system provides the capabilities of structured extensibility. The set of utilized filters is of reasonable size, while a fine grained filter generator approach would be more suitable. Nevertheless, *EmgFramework* is equipped with a solid toolbox of tools. The *Conconi Tool* and *Muscle Fatigue Detection Tool* encapsulate concrete use cases. Other tools for peak detection and deflection point finding are more likely to be used by other tools. Communication can be established using five different driver interfaces with one dedicated playback driver. MQTT and the serial drivers are discouraged. Evaluation concludes the usage of the network or the Bluetooth driver. In addition to the desktop application, the framework's architecture offers a fully functional Android implementation. Compared to its competitors *EMGworks* and *Motion Lab Systems*, *EmgFramework* exposes a much smaller set of functionality. On the other hand the competitors do not provide a fully featured mobile application and are restricted to certain platforms. Overall they force users to use components of their ecosystem. Both do not provide a plugin mechanism at application level.

The *Acquisition Case Designer* adopts one concept of *EMGworks*, namely the workflows. These workflows provide full reusability of existing components. Therefore the implementation policy enforces reusability at component level. This may include driver, views, filter, tools and any arbitrary implementation adhering to the workflow protocol. The narrow scope of the core system is compensated through its rich extensibility features.

8.2 Case Study results

The results of chapter 7 show significant shortcomings of the framework's sensing environment. Serious flaws are revealed in the producer device implementation. This must be addressed, in order to provide a stable measurement system. The flaw of an inconsistent sampling frequency is not acceptable. This leads to further investigation and evaluation as Android Things as the primary target platform for producer devices. However, the utilization of other sensor platforms of renowned manufacturers is still possible and should be considered as well. But also the native heart rate sensing distinguished itself with malfunction. Although the framework's main purpose is to sense EMG data it is still not tolerable that such an important feature is not working well.

The software itself proved worth using and the implemented tools work in an acceptable manner.

8.3 Software Qualities

Chapter 6 is focused primary on quality characteristic evaluation. Data and metrics were collected mainly for portability, maintainability and performance.

Portability

The portability metrics between files and lines of code differ not more than 5%. The overall portability factor of 80% indicates, that 80% of the system is reused across platforms, while only 20% need a reimplementation for the target platform. These 20% are comprised by views or platform-specific capabilities, such as driver or storage. Due to the utilization of Kotlin (and Java), all desktop operating systems use the same view abstraction. This allows the distinction between *Android* and *Desktop*, while the latter supports the main operating systems out of the box. The desktop version also has an improved installability, as it does not require an installation on the system. The 23 MB executable can operate even from a USB stick, only depending on installed serial driver libraries – which usage is actually discouraged for production applications. Kotlin was utilized because of Kotlin/Native in the first place, which is considered for later development towards portability.

Maintainability & Modifiability

The set of of maintainability and modifiability metrics conclude a solid foundation for further development. The conclusion is backed by the comprehensive suite of metrics. The MOOD metric set evaluates the adherence to object oriented design. The metrics for *EmgFramework* lie between the suggested boundaries, concluding a structured and object oriented code concept. Extensibility cannot be verified with quantitative data. One only may conclude that extensibility comes at a rather low cost, due to the modular system and small, maintainable classes with an average of less than 100 lines of code. The modules *messaging* and *casedesigner* require further inspection. Refactoring should make *messaging* more abstract and therefore robust. However, the module has no dependencies (stable) and contains concrete implementations. This is because both *client* and *clientdriver* module depend on it as the single source of truth for messaging.

Nevertheless, the system must be refactored in order to provide an even higher degree of extensibility and modifiability. The system provides capabilities of a plugin system. The best way to provide a lightweight extensible core would require a refactoring of the *core* module. The problem can be solved by splitting the *core* module into a *core* and *corex* module, whereas the latter contains the extensible logic (filter, tool, analysis method).

Performance

Performance is a critical software quality, although the system does not enforce real-time constraints. Nevertheless, the latency test reveals dramatical results for MQTT and the serial interface. Although Bluetooth also showed drastic results, this results should be manageable with tuning and a more reliable software implementation. High data sampling rates demand a more elaborate communication protocol. Network and simulation driver are the preferred drivers for further usage.

The system has a reasonable memory footprint. Although the visualization of the data in form of the so-called `VisualView` implementations hindered performance. However, views are just a detail in the framework and therefore they can be seamlessly replaced with new implementations.

8.4 Architectural Decisions

Considering the effect of the decisions made in chapter 4, the framework did well specifying and converting decisions into actions. The enhanced consumer-producer pattern is necessary to handle data transfer between both parties. Kotlin and the Model-View-Presenter pattern keep the system portable with the utilization of an Android app, running on the software core. The Pipes & Filters pattern proved worth using it for the workflow implementation for the *Acquisition Case Designer*.

Although the system has a well-thought architecture, it may miss some important design concepts. The system violates the dependency rule of Martin [50], indicating that dependencies of artifacts point inward towards higher level policies. A layered architecture as Fowler suggests [24] could align the flow of dependencies. Such architectural decisions would highly benefit of an architectural evaluation from an independent expert, which has not taken place yet.

8.5 System Flaws

Throughout the evaluation and test phase two components exposed evident system flaws: communication flow and visualization. The flaws of the producer device implementation are already addressed in section 8.2.

Communication & Driver

Communication between producer and consumer offers a list of flaws. The protocol does not specify any welcome message, while an exit message is optional. The consumer can

only alter the sampling frequency and no other parameters of the producer device. Also the data transfer itself may require reworking.

Assuming an acquisition scenario of a test subject with two attached EMG sensors and a heart rate chest belt in addition. Sampling frequency is set to $fs = 1000\ Hz$. If data is transferred using the `EmgPacket` packeting approach for only 1 second, this accumulates in

$$S_p = 1000 \cdot (2 \cdot 64 + 32 + 64) = 27.3\ kB \tag{8.1}$$

of sent data. The packeting approach `EmgBurst` replaces the EMG data type of double with float, heart rate data type of integer with short and bundles one second of data with a single timestamp and heart rate value. Hence, `EmgBurst`s transferred size per second is computed as

$$S_b = (1000 \cdot (2 \cdot 32)) + 16 + 64 = 7.82\ kB. \tag{8.2}$$

Moreover, an `EmgBurst` is transferred only once a second, therefore the communication medium is idle the time between, which saves battery, reduces latency and renders buffering logic on the consumer site obsolete.

Still, EMGworks outperforms the system. Nor et al. [62] describes a test setup for a system of 16 EMG and 48 accelerometer analog channels for integration with motion capture and other 3rd party data acquisition systems in his paper *EMG Signals Analysis of BF and RF Muscles In Autism Spectrum Disorder (ASD) During Walking. EmgFramework* cannot spawn such a comprehensive sensing system by now.

Visualization

During evaluation the visualization of acquired data with sampling rates higher than 100 Hz offered some severe drawbacks of performance. This is mainly because of the non-performant implementation for the desktop and Android. However, the view is a detail of the whole application and neatly decoupled by the rest of the application. The visualization can be easily replaced by a more performant implementation. The graph library SciChart can be used for Android, as it offers high-performance real-time graphs for medical and engineering applications [47]. Unfortunately it is not supported for Java/Kotlin on the desktop (WPF only), and therefore the desktop application would require a manual lightweight Swing or AWT implementation instead.

Chapter 9

Discussion

The chapter will focus on the advantages and disadvantages the system brings to the table, compared to other systems or viewed in isolation. System usage and entrance barrier for users and scientists are discussed as well. The chapter will be closed with an outlook of the further points of development.

9.1 Advantages

The main advantage of *EmgFramework* over its competitors is its availability as open source software. Competitors as *EMGworks* or *Motion Lab Systems* do not provide such a format of extensibility to their applications. The system provides a portable, maintainable and extensible base, which is backed by an array of metrics. It does not enforce a vendor lock-in, meaning that the consumer site will only require a producer device which adheres to the communication protocol. Delsys provides an API for their Trigno platform (depicted in figure 3.1). With this API and the capabilities of the framework, it is possible to establish a connection to their sensors via the driver interface. This demonstrates the power of the framework, as it can easily integrate existing hardware. But it can also extend its software, by a simple plugin system. Tools, filters and drivers are designed to be extensible and the UI is conceived to handle an arbitrary size of these key elements. Plugins are loaded at runtime from a folder which contains the plugins bundled as a JAR file. The small size of its executable makes it easy to transfer and highly portable, at least for the desktop application. The application can be started across different desktop operating systems without installation[1]. The abstraction of the device driver make the system useful for any kind of EMG data acquisition. New devices and connection mediums can be integrated with around 100 lines of code. Developers of EMG related projects with requirements regarding a custom hardware can easily adapt and extend the framework for their needs, without loosing too much time for implementing the overall foundation for their projects.

[1] It neglects the fact, that the serial driver requires pre-installed system libraries. However, serial clients are discouraged in production code.

9.2 Drawbacks

EmgFramework is not considered as a commercial product nor as an all-in-one solution for scientists and developers who want to start capturing EMG data. It only provides a software foundation, where future developers can extend it to their needs and requirements. The core functionality is relatively small, compared to competitors like *EMGworks*. Also the support for producer devices is limited to a single reference implementation for Android Things, which exposes inconsistencies in establishing a coherent sampling frequency and heart rate sensing. The system provides capabilities for extending and implementing additional features. Casual users won't have the time nor the money to implement a fully functional custom device. The way how *EmgFramework* tries to incorporate functionality is with plugin developers, who publish their custom implementations and make it available for the public. However, this plugin community is not existing at the moment, concluding that new functionality is only published by the core development team. The communication protocol needs improvements as well. Connect and disconnect events are not proper handled by both parties and data transfer can be optimized to preserve battery of the producer device. The producer device and the consumer applications must form a more coherent system. The drawbacks can be summarized with the rather rudimentary ecosystem of the whole framework.

9.3 Usage / Entrance Barrier

A fact that should not be neglected is the fact of the usage of the system and its entrance barrier for people with a non-technical background. The core system itself has a minimalistic user interface, but requires a technical background in order to use it. The *Acquisition Case Designer* on the other hand tries to minimize this entrance barrier for all kind of users. The main purpose of the *Acquisition Case Designer* is to force reusability of custom software component and to lower the entrance barrier for non-technical scientists or users in general. Developers are given a concise API for their custom components, helping the developers to provide a readable and understandable interface to the users. The case designer application incorporates an intuitive drag and drop user interface as part of the desktop application. A side bar exposes all the available components and a mini-map. Users can connect two components with the right mouse button and alter exposed component properties with the middle mouse button. This enables users to design a custom workflow designed for their needs, with components and their required properties. Properties are changeable properties of a class. The UI is limited to the desktop application, as the small UI of the phones would hinder efficient workflow designing. However, the utilization of already created ACD files[2] can be brought to the smartphone application later on.

A factor which is ignored at the moment is the utilization of standard file formats, such as *.arff* for Machine Learning and *.c3d* for data acquisition. The lack of these supported file formats will decrease the acceptance of the application, but due to flexibility of the framework, the formats can be handed in a later point in time.

[2] Acquisition Case Designer files

9.4 Outlook

The system still needs some selective but crucial improvements. The *Acquisition Case Designer* still needs some reworking in order to provide a convenient usage of the application. Other improvements concern portability, performance or functionality.

The main challenge of the pipes & filters approach is the requirement of a suitable pipe, which converts the produced data type of one component into a consumable data type of the other component. This comes with the complication, that the code base must utilize a suitable pipe in order to combine two components at runtime. Unfortunately this can't be checked at compile time. Filters should be replaced with a reflection-based approach, which allows the user to select the output property via a UI. This would also eliminate the drawback of not destructuring complex data structures. By now, *Acquisition Case Designer* cannot pass a single property of a complex data type to the consuming component. A reflection-based connection approach would eliminate both problems.

Another concern regarding the *Acquisition Case Designer* takes the instances of properties into account. Property instances are now shallow copied across multiple instances of a component. This hinders multiple components of the same type of having different properties. This should be replaced with a deep copy and is considered as an implementation detail.

The problem of the communication protocol is already mentioned in section 8.5 with a calculation how to reduce transferred data. The new communication protocol can drastically reduce the overhead, preserve battery of the producer device and streamline the bidirectional communication flow for better connect and disconnect handling.

The *Acquisition Case Designer* is partly incomplete. It works for demonstration purposes, but as a workflow cannot utilize a storage component, it renders the use case of data acquisition obsolete, as data cannot be stored. The system will incorporate an `EmgStorageComponent` annotation, which turns storage components into reusable case designer components.

Performance issues of the `VisualViews` are mentioned in section 8.5. The implementation does not satisfy the quality of visualization components, as they drastically hinder the perceived quality of the applications, as it slows down the user interface on Android and on the desktop application. A new and performant `VisualView` for Android and desktop is necessary.

The *client* module needs refactoring. More code and logic should be put into the module, away from concrete implementations. The portability metric of the `EmgThingsClient` showed that the concrete producer implementation requires to many files to implement. More code should be moved into the *client* module in order to make producer ports more cost efficient.

The utilization of Kotlin/Native would greatly benefit from the refactoring of the client module, as new platforms could be utilized with even less code from a single code base. Ports to Arduino and other embedded systems rather than Android Things is possible at a rather low cost, as soon as Kotlin/Native is part of the development process. Kotlin/Native allows a mobile iOS app as well as a variety of target platforms for producer devices with a fraction of the port costs without Kotlin/Native.

Machine Learning is heavily used in the EMG research community. Karlik summarizes techniques of Machine Learning algorithms for a EMG prosthesis connection [40]. The techniques utilize a set of already implemented components, such as time series analysis and frequency analysis. *EmgFramework* already provides this set of functionality, which can be enriched with Machine Learning capabilities. The architect must decide to either fully integrate Machine Learning via frameworks as Weka[3] or loosely provide functionality by providing data storage (and loading) for Machine Learning data sets. The tight coupling and integrated support would be preferable, but imposes a penalty of maintenance and extensibility. Support for data sets and data modification would be less cost intensive.

[3] https://www.cs.waikato.ac.nz/ml/weka/

References

Literature

[1] F Brito e Abreu. "The MOOD metrics set". In: *proc. ECOOP*. Vol. 95. 1995, p. 267 (cit. on p. 71).

[2] Aivosto. *MOOD and MOOD2 metrics*. Online. 2010 (cit. on pp. 71, 72).

[3] Muhammad Ali Babar, Liming Zhu, and Ross Jeffery. "A framework for classifying and comparing software architecture evaluation methods". In: *Software Engineering Conference, 2004. Proceedings. 2004 Australian*. IEEE. 2004, pp. 309–318 (cit. on p. 83).

[4] Justin Baker. *Buying vs. Building a Feature Flagging Platform*. Online. Feb. 2016. URL: http://blog.launchdarkly.com/buying-vs-building-a-feature-flagging-system/ (cit. on pp. 29, 30).

[5] Len Bass. *Software architecture in practice*. Pearson Education India, 2012 (cit. on pp. 26, 37).

[6] Bruce Beare. *Brillo / Weave Part 1: High Level Introduction*. Online. Apr. 2016. URL: http://events17.linuxfoundation.org/sites/events/files/slides/Brillo%20and%20Weave%20-%20Introduction_v3_1.pdf (cit. on p. 34).

[7] Adrain DC Chan and Geoffrey C Green. "Myoelectric control development toolbox". *CMBES Proceedings* 30.1 (2017) (cit. on p. 25).

[8] Francis HY Chan et al. "Fuzzy EMG classification for prosthesis control". *IEEE transactions on rehabilitation engineering* 8.3 (2000), pp. 305–311 (cit. on p. 12).

[9] Mario Cifrek et al. "Surface EMG based muscle fatigue evaluation in biomechanics". *Clinical Biomechanics* 24.4 (2009), pp. 327–340 (cit. on pp. 9, 12).

[10] EA Clancy and Neville Hogan. "Theoretic and experimental comparison of root-mean-square and mean-absolute-value electromyogram amplitude detectors". In: *Engineering in Medicine and Biology Society, 1997. Proceedings of the 19th Annual International Conference of the IEEE*. Vol. 3. IEEE. 1997, pp. 1267–1270 (cit. on p. 8).

[11] Paul Clements et al. "Documenting software architectures: views and beyond". In: *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society. 2003, pp. 740–741 (cit. on pp. 26, 27, 29, 36, 37).

[12] Ferrari Conconi et al. “Determination of the anaerobic threshold by a noninvasive field test in runners”. *Journal of Applied Physiology* 52.4 (1982), pp. 869–873 (cit. on pp. 50, 51, 74).

[13] Paola Contessa, Carlo J De Luca, and Joshua C Kline. “The compensatory interaction between motor unit firing behavior and muscle force during fatigue”. *Journal of neurophysiology* 116.4 (2016), pp. 1579–1585 (cit. on p. 15).

[14] Vittorio Cortellessa, Fabrizio Marinelli, and Pasqualina Potena. “An optimization framework for “build-or-buy” decisions in software architecture”. *Computers & Operations Research* 35.10 (2008), pp. 3090–3106 (cit. on pp. 30, 31).

[15] Carlo De Luca. “Electromyography”. *Encyclopedia of Medical Devices and Instrumentation* (2006) (cit. on pp. 3–5, 11, 14).

[16] Carlo J De Luca. “Surface electromyography: Detection and recording”. *DelSys Incorporated* 10 (2002), p. 2011 (cit. on pp. 4–8, 13–15).

[17] Carlo J De Luca et al. “Filtering the surface EMG signal: Movement artifact and baseline noise contamination”. *Journal of biomechanics* 43.8 (2010), pp. 1573–1579 (cit. on pp. 13, 14).

[18] Gianluca De Luca. “Fundamental concepts in EMG signal acquisition”. *Copyright Delsys Inc* (2003) (cit. on pp. 14, 24).

[19] Inc. Delsys. *Delsys Homepage*. Online. URL: https://www.delsys.com/ (cit. on pp. 15, 16).

[20] Inc. Delsys. *EMGworks*. Online. URL: https://www.delsys.com/product/emgworks-software/ (cit. on pp. 2, 16, 27, 29).

[21] NA Dimitrova, GV Dimitrov, and OA Nikitin. “Neither high-pass filtering nor mathematical differentiation of the EMG signals can considerably reduce crosstalk”. *Journal of Electromyography and Kinesiology* 12.4 (2002), pp. 235–246 (cit. on p. 5).

[22] F Felici. “Applications in exercise physiology”. *Electromyography: Physiology, engineering, and noninvasive applications* (2004), 365e379 (cit. on pp. 4, 12, 13).

[23] Kim Fowler. “Build versus buy”. *IEEE Instrumentation & Measurement Magazine* 7.3 (2004), pp. 67–73 (cit. on pp. 29, 30).

[24] Martin Fowler. *Presentation Domain Data Layering*. Online. Aug. 2015. URL: https://martinfowler.com/bliki/PresentationDomainDataLayering.html (cit. on p. 85).

[25] Andrew Froehlich. *8 IoT Operating Systems Powering The Future*. Online. 2016. URL: https://www.informationweek.com/iot/8-iot-operating-systems-powering-the-future/d/d-id/1324464?image_number=6 (cit. on p. 1).

[26] Google-Scholar. *Carlo J. De Luca publication statistic*. URL: https://scholar.google.at/citations?user=Djx4HhoAAAAJ&hl=de&oi=ao (cit. on p. 15).

[27] Juan Carlos Granja-Alvarez and Manuel Jose Barranco-Garcia. “A method for estimating maintenance cost in a software project: a case study”. *Journal of Software Maintenance: Research and Practice* 9.3 (1997), pp. 161–175 (cit. on p. 28).

[28] Christian Grech. *COLDEX New Data Acquisition Framework*. Tech. rep. 2015 (cit. on pp. 19, 20, 28, 29).

[29] Göran M Hägg, Alwin Luttmann, and Matthias Jäger. "Methodologies for evaluating electromyographic field data in ergonomics". *Journal of electromyography and kinesiology* 10.5 (2000), pp. 301–312 (cit. on p. 52).

[30] Javier Gonzalez Huerta. *Lecture notes in Software Architecture and Qualities*. Lecture. Dec. 2017 (cit. on p. 26).

[31] IBM. *The age of data and opportunities*. Online. Dec. 2014. URL: http://www.ibmbigdatahub.com/blog/age-data-and-opportunities (cit. on p. 1).

[32] Delsys Inc. *De Luca obituary*. 2016. URL: https://www.delsys.com/obituary-deluca_carlo/ (cit. on p. 15).

[33] Delsys Inc. *What factors affect EMG Signal Quality?* Online. 2015. URL: https://www.delsys.com/products/software/emgworks/sqm/factors/ (cit. on p. 31).

[34] ISO. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE)*. ISO 25010. Geneva, Switzerland: International Organization for Standardization, 2010 (cit. on pp. 26–28, 68).

[35] J Al-Ja'Afer and K Sabri. "Metrics for object oriented design (MOOD) to assess Java programs". *King Abdullah II school for information technology, University of Jordan, Jordan* (2004) (cit. on pp. 71, 72).

[36] Niklas Johansson and Anton Löfgren. "Designing for Extensibility: An action research study of maximizing extensibility by means of design principles". B.S. thesis. 2009 (cit. on p. 28).

[37] Ralph E Johnson and Brian Foote. "Designing reusable classes". *Journal of object-oriented programming* 1.2 (1988), pp. 22–35 (cit. on p. 36).

[38] MP Kadaba et al. "Repeatability of kinematic, kinetic, and electromyographic data in normal adult gait". *Journal of Orthopaedic Research* 7.6 (1989), pp. 849–860 (cit. on p. 13).

[39] MEHMET Kara et al. "Determination of the heart rate deflection point by the Dmax method". *Journal of sports medicine and physical fitness* 36.1 (1996), pp. 31–34 (cit. on p. 54).

[40] Bekir Karlik. "Machine learning algorithms for characterization of EMG signals". *International Journal of Information and Electronics Engineering* 4.3 (2014), p. 189 (cit. on p. 90).

[41] Joshua C Kline and Carlo J De Luca. "Error reduction in EMG signal decomposition". *Journal of neurophysiology* 112.11 (2014), pp. 2718–2728 (cit. on pp. 3, 15).

[42] Peter Konrad. "The abc of emg". *A practical introduction to kinesiological electromyography* 1 (2005), pp. 30–35 (cit. on p. 14).

[43] Philippe Kruchten. "Architectural blueprints—The "4+ 1" view model of software architecture". *Tutorial Proceedings of Tri-Ada* 95 (1995), pp. 540–555 (cit. on pp. 36, 40, 42).

[44] Philippe B Kruchten. "The 4+ 1 view model of architecture". *IEEE software* 12.6 (1995), pp. 42–50 (cit. on p. 36).

[45] H Leitner, P Hofmann, and G Gaisl. "A method for the microcomputer aided determination of the anaerobic threshold by means of heart rate curve analysis". *Graz Biomed Eng Aust* 88 (1988), pp. 136–141 (cit. on p. 53).

[46] David R Loker et al. "Remote data acquisition using bluetooth". *Proceedings of the American Soc. For Eng. Education (ASEE)* (2005) (cit. on pp. 20, 21).

[47] SciChart Ltd. *SciChart Scientific, Medical & Engineering Charting*. Online. 2018. URL: www.scichart.com/why-scichart-perfect-for-scientific-medical-engineering-applications/ (cit. on p. 86).

[48] Alwin Luttmann, Matthias Jäger, and Wolfgang Laurig. "Electromyographical indication of muscular fatigue in occupational field studies". *International Journal of Industrial Ergonomics* 25.6 (2000), pp. 645–660 (cit. on pp. 9, 10, 12, 52, 76, 79, 81).

[49] Robert Martin. "OO design quality metrics". *An analysis of dependencies* 12 (1994), pp. 151–170 (cit. on p. 69).

[50] Robert C Martin. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017 (cit. on pp. 31, 69–71, 85).

[51] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009 (cit. on p. 66).

[52] Michelle Matte. *Isotonic Vs. Isometric Muscle Exercises*. Online. Sept. 2017. URL: https://www.livestrong.com/article/449913-isotonic-vs-isometric-muscle-exercises/ (cit. on p. 74).

[53] Eric Maxwell. *MVC vs. MVP vs. MVVM on Android*. Online. Jan. 2017. URL: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/ (cit. on pp. 43, 44).

[54] Subhas C Misra and Virendra C Bhavsar. "Relationships between selected software measures and latent bug-density: Guidelines for improving quality". In: *International Conference on Computational Science and Its Applications*. Springer. 2003, pp. 724–732 (cit. on p. 71).

[55] Simon Monk. *Programming the Raspberry Pi: getting started with Python*. TAB Electronics, 2015 (cit. on p. 22).

[56] James D Mooney. "Bringing portability to the software process". *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV* (1997) (cit. on p. 68).

[57] BOSTJAN Murovec and SLAVKO Kocijancic. "A USB-based data acquisition system designed for educational purposes". *International Journal of Engineering Education* 20.1 (2004), pp. 24–30 (cit. on p. 22).

[58] John Newell et al. "Software for calculating blood lactate endurance markers". *Journal of sports sciences* 25.12 (2007), pp. 1403–1409 (cit. on pp. 80, 81).

[59] Daisuke Nishikawa et al. “EMG prosthetic hand controller using real-time learning method”. In: *Systems, Man, and Cybernetics, 1999. IEEE SMC’99 Conference Proceedings. 1999 IEEE International Conference on*. Vol. 1. IEEE. 1999, pp. 153–158 (cit. on p. 12).

[60] JM Nogiec and K Trombly-Freytag. *A dynamically reconfigurable data stream processing system*. Tech. rep. Fermi National Accelerator Laboratory (FNAL), Batavia, IL, 2004 (cit. on p. 19).

[61] JM Nogiec et al. “EMS: a framework for data acquisition and analysis”. In: *AIP Conference Proceedings*. Vol. 583. 1. AIP. 2001, pp. 255–257 (cit. on p. 18).

[62] Mohd Nor et al. “EMG signals analysis of BF and RF muscles in autism spectrum disorder (ASD) during walking”. *International Journal on Advanced Science, Engineering and Information Technology* 6.5 (2016), pp. 793–798 (cit. on p. 86).

[63] Jorge L Ortega-Arjona. “The Pipes and Filters Pattern. A Functional Parallelism Architectural Pattern for Parallel Programming.” In: *EuroPLoP*. 2005, pp. 637–650 (cit. on p. 44).

[64] Omry Paiss and Gideon F Inbar. “Autoregressive modeling of surface EMG and its spectrum with application to fatigue”. *IEEE transactions on biomedical engineering* 10 (1987), pp. 761–770 (cit. on p. 8).

[65] Paolo Patierno. *MQTT & IoT protocols comparison*. Online. Feb. 2014. URL: https://de.slideshare.net/paolopat/mqtt-iot-protocols-comparison (cit. on pp. 33, 34).

[66] Marina Serozhenko. *MQTT vs. HTTP: which one is the best for IoT?* Online. Mar. 2017. URL: https://medium.com/mqtt-buddy/mqtt-vs-http-which-one-is-the-best-for-iot-c868169b3105 (cit. on p. 33).

[67] Frederick T Sheldon, Kshamta Jerath, and Hong Chung. “Metrics for maintainability of class inheritance hierarchies”. *Journal of Software: Evolution and Process* 14.3 (2002), pp. 147–160 (cit. on p. 62).

[68] Statista.com. *IOT device trend*. Online. Nov. 2016. URL: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/ (cit. on p. 1).

[69] Motion Lab Systems. *Motion Lab Systems*. Online. URL: https://www.motion-labs.com/ (cit. on pp. 17, 18, 29).

[70] Cornell University. *Concurrency - Producer/Consumer Pattern and Thread Pools*. Online. URL: https://www.cs.cornell.edu/courses/cs3110/2010fa/lectures/lec18.html (cit. on p. 43).

[71] Karan Veer and Tanu Sharma. “A novel feature extraction for robust EMG pattern recognition”. *Journal of medical engineering & technology* 40.4 (2016), pp. 149–154 (cit. on pp. 7–9, 24, 25).

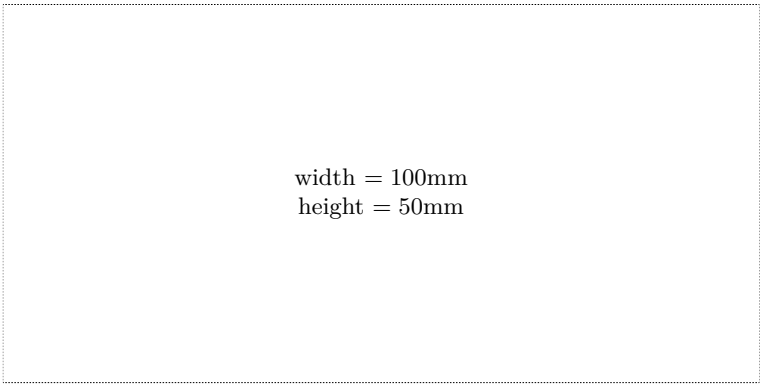
[72] Vladimir Vujović. “Development of a custom Data Acquisition System based on Internet of Things”. *UNITECH* 15 (2015), pp. 20–21 (cit. on pp. 22, 23, 28).

[73] Vladimir Vujović and Mirjana Maksimović. “Raspberry Pi as a Sensor Web node for home automation”. *Computers & Electrical Engineering* 44 (2015), pp. 153–171 (cit. on p. 23).

[74] Mark Weiser. “The computer for the 21st century.” *Mobile Computing and Communications Review* 3.3 (1999), pp. 3–11 (cit. on p. 22).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —