FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Softwaredesign zur Programmierung eines Elektrostimulators mit integriertem Compliance Management

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Florian Guggenberger
Matrikelnummer 0226774

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Ao. Univ.-Prof. DI Dr. Dr. h.c. Winfried Mayr
Projekt-Ass. DI Matthias Krenn
Am Zentrum für Medizinische Physik und Biomedizinische Technik, Medizinische Universität Wien

Wien, 28.04.2011

(Unterschrift Verfasser) (Unterschrift)

Erklärung

FLORIAN GUGGENBERGER
Sankt Lorenzen 23
9654 St. Lorenzen im Lesachtal

"Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit einschließlich Tabellen, Karten und Abbildungen, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, am 28. April 2011 .....
Unterschrift

Danksagung

Der Dank gilt an dieser Stelle allen Menschen die mich während des gesamten Studiums begleitet und unterstützt haben.
Meiner Familie die trotzt langer Durststrecken den Glauben an mich nie verloren hat und mir auch in schwierigen Zeiten stets den Rücken stärkte.
Meinen Freunden für die aufbauenden Worte, tollen Gespräche, langen Nächte und Durchhalteparolen. Meinen Studienkollegen für die fachliche Beratung während der Fertigstellung dieser Arbeit.
Ganz besonderrs danken möchte ich aber meiner Freundin Margret, die mir mit ihrer einfühlsamen Art, zum wichtigsten Menschen in meinem Leben geworden ist. Danke für die großartige moralische Unterstützung und das entgegengebrachte Verständnis!
Bedanken möchte ich mich auch bei meinem Betreuer DI. Matthias Krenn für den vielseitigen Input und die tolle Begleitung der Arbeit.

Kurzfassung

In dieser Diplomarbeit wird eine Softwarearchitektur vorgestellt, um einen neuromuskulären Stimulator mit vielseitigen Stimulationsmustern zu programmieren. Die entwickelte Software erlaubt durch grafische Manipulation eine flexible Zusammenstellung der Stimulationsparameter. Das Softwaredesign bietet die Möglichkeit zur Festlegung von Messpunkten zur Auswertung von Messsignalen. Auch die Aufzeichnung von Compliance-Daten wird durch das vorgestellte Konzept unterstützt.
Es werden die geeigneten Datenstrukturen dargestellt um mit einem neuromuskulären Stimulator zu kommunizieren und Daten auszutauschen. Ein Hauptaugenmerk wird auf das Management der der Compliance-Daten gelegt, um Aussagen über die Art und zeitliche Intensität der Trainingseinheiten treffen zu können. Es wird dabei sowohl auf die Datenstruktur als auch auf deren Visualisierung eingegangen.

Die entwickelte Softwarelösung beschreibt eine Strategie zur Umsetzung der Softwareentwicklung mit Hilfe des .NET-Framworks und der Windows Presentation Foundation unter Verwendung des Model-View-ViewModel-Entwurfmusters. Durch ein interaktives Bedienkonzept mit der Möglichkeit zur grafischen Manipulation und einem modernen Erscheinungsbild bietet die Softwareanwendung ein einfach zu bedienendes Werkzeug um komplexe Stimulationsdaten für Muskeltrainings mit funktioneller Elektrostimulation zu generieren und damit ein Stimulationsgerät zu steuern. Durch die Visualisierung der Trainingsdaten liefert die Applikation einen raschen Überblick über den Trainingsverlauf und kann auch zum Vergleich der Ergebnisse von Stimulationen herangezogen werden.

Abstract

This thesis introduces a software application to programm an electrical stimulation device with different stimulation patterns. The easy to use software, which was developed within this thesis, supports flexible manipulation of several parameters in a intuitive matter. Also the setting for recording measurement signals can be adjusted with this application. Compliance of training with the stimulation device is recorded and can be managed within the software.
The data structures used for realisation were introduces and the concept for dealing with data exchange between the software and stimulation device would be described. An important part of the project deals with the management of compliance to collect data related to duration and intesity of sessions with electical stimulation.

In this work a strategy for compliance management with functional electrical stimulation is pronounced. It shows the technology used to implement the real world problem with Windows Presentation Foundation based on the .NET-Framwork. The meaning of the Model-View-ViewModel design pattern in context with WPF is described and the thesis would show the utilisation of the pattern within this work. As a result the software developed, allows an easy to use creation of various stimulation patterns with the possibilty to monitor measurements taken throughout stimulation trainings. Comparison of measurement signals could also be done with the application. And as a key feature the software allows recording of compliance data to analyse the training sessions for each patient.

Abkürzungen

.NET Microsoft Dot-Net-Framework. 14, 15

ADC Analog Digital Converter. 51

AP Aktionspotential. 4

BAML Binary Application Markup Language. Binäre Representation von XAML.. 24, 25

BCD Binary Coded Decimal. Binär kodierter Zahlenwert. Im Zusammenhang mit dieser Arbeit wird BCD immer in Big-Endian Reihenfolge erwähnt.. 56

CIL Common Intermediate Language. 15

CLR Common Language Runtime. 15

EMG Elektromyographie. 12, 13

Exception dt. Ausnahme. 21

FES Funktionelle Elektrostimulation. 3, 7, 9

FT fast twitch = schnellzuckend. 7

GUID Global Unique Identifyer. 16-bit lange, eindeutige Nummer zur Identifikation von Datensätzen.. 55, 56

NMES Neuromuskuläre Elektrostimulation. 3, 9–11

ST slow twitch = langsamezuckend. 7

USB Universal Serial Bus. 1

XAML Extensible Application Markup Language. Auf Xml-basierendes Format zur deklarativen Erstellung von Benutzeroberflächen in WPF.. 24, 25

Zoll Im englischen auch Inch genannt. 1 Zoll entspricht 25.4mm. 18

Abbildungsverzeichnis

Tabellenverzeichnis

Inhaltsverzeichnis

KAPITEL 1

Einführung

Muskeltraining kann den Abbau der Muskulatur und die daraus folgende Imobilität vermindern. Insbesondere die ältere Bevölkerung ist durch zunehmende körperliche Inaktivität vom Muskelabbau betroffen. Die funktionelle Elektrostimulation ist bei Paraplegikern eine anerkannte Trainingsmethode der Muskulatur. Durch die Ausdehnung der Erkenntnisse der Elektrostimulation auf den Bereich der altersbedingten Muskelatrophie können Schlüsse für den Einsatz der Stimulation der älteren Bevölkerung gewonnen werden.

Zahlreiche Studien haben sich bereits mit der Wirkung der Elektrostimulation beschäftigt und es gibt eine Vielzahl von Stimulationsgeräten am Markt. Der Großteil der existierenden Systeme bietet multiple Einstellungsmöglichkeiten der Stimulationsparameter an, um verschiedene physiologische Aspekte der Elektrostimulation zu beleuchten. Allerdings wird bei keinem der mir bekanntem Elektrostimulatoren, weder die Art und Dauer der Anwendung mitprotokolliert, um qualitative Aussagen über den Erfolg der Trainingseinheiten mit dem Stimulationsgerät zu treffen, noch werden Messdaten aufgezeichnet, die Rückschlüsse über den Fortschritt des Erfolges zulassen.

Es ist also von Vorteil, ein System zur Verfügung zu haben, wodurch zum einen eine möglichst variable Justierung der Parameter möglich ist, und zum anderen eine Aussage über Dauer, Art und Effekt der Anwendung getroffen werden kann. Unsere Lösung für ein solches System sieht hierfür eine Hardware vor, die in der Lage ist, elektrische Stimulationsimpulse zu generieren die transkutan über Elektroden appliziert werden. Um die universellen Einstellungsmöglichkeiten für eine Stimulation zu gewährleisten, wird die Hardware über USB an einen Computer angebunden. Über die Software am Computer wird es möglich, den Stimulator mit mannigfaltigen Stimulationsprotokollen zu programmieren. Des weiteren werden mit der Stimulator-Einheit Messungen getätigt, die wiederum über die Software ausgelesen werden können. Durch diese Messungen sollen nach der Auswertung, Rückschlüsse über den Verlauf des Stimulationstrainings

möglich sein. Durch die Software werden alle Messergebnisse visualisiert und archiviert. Für weitere Analysen der Ergebnisse bietet die Softwareapplikation eine Exportfunktionalität der Daten an. Die Software stellt somit ein Schlüsselkonzept des Projekts dar. Diese Diplomarbeit beschreibt nun in weiterer Folge insbesondere die Umsetztung der Softwareseite dieses Projekts.

KAPITEL 2

Hintergrund

2.1 Neuromuskuläre Elektrostimulation

Als neuromuskuläre Elektrostimulation (NMES) versteht man das Auslösen von Muskelkontraktionen in einzelnen Muskeln oder Muskelgruppen durch elektrische Impulse. Werden die Kontraktionen koordiniert ausgelöst, sodass eine Funktion, zum Beispiel das Greifen eines Glases, unterstützt oder übernommen wird, spricht man von der so genannten funktionellen Elektrostimulation (FES) [1].
Die Geschichte der Elektrostimulation hat lange Tradition und muss um das Jahr 46 n.Chr. begonnen haben. Der römische Arzt Scribonus Largus beschreibt erstmals die Behandlung von Kopfschmerzen und Gicht mit Hilfe der elektrischen Entladungen des Zitterrochens. Ein weiterer Meilenstein in der Anwendung der Elektrostimulation gelang 1790 Luigi Galvani mit seinem Froschschenkel-Experiment. 1952 schaffte es Paul M. Zoll die Aufrechterhaltung des Herzschlages durch externe Elektroden für 20 Minuten zu gewährleisten. Acht Jahre später (1960) wurde von Schardack der erste batteriebetriebene Herzschrittmacher eingesetzt [2].

2.1.1 Elektrophysiologische Grundlagen

Die Voraussetzungen für die Erregbarkeit von Nerven und Muskelzellen durch einen elektrischen Reiz sind durch die spezifischen Eigenschaften der Zellmembran mit Ausbildung einer Ladungsseparation zwischen dem Zellinneren und dem Zelläußeren gegeben. Dieser Ladungsunterschied zwischen extrazellulären Raum und Cytoplasma bedingt eine semipermeable Membran mit selektiven Ionenkanälen. (Abb. 2.1) Im Ruhezustand der Zelle existiert intrazellulär eine höhere Konzentration von Anionen gegenüber dem Äußeren der Zelle und führt so zu einer negativen Potentialdifferenz. Diese Potentialdifferenz wird als Ruhepotential bezeichnet und liegt je nach Zelltyp zwischen $-60mV$ und $-90mV$.

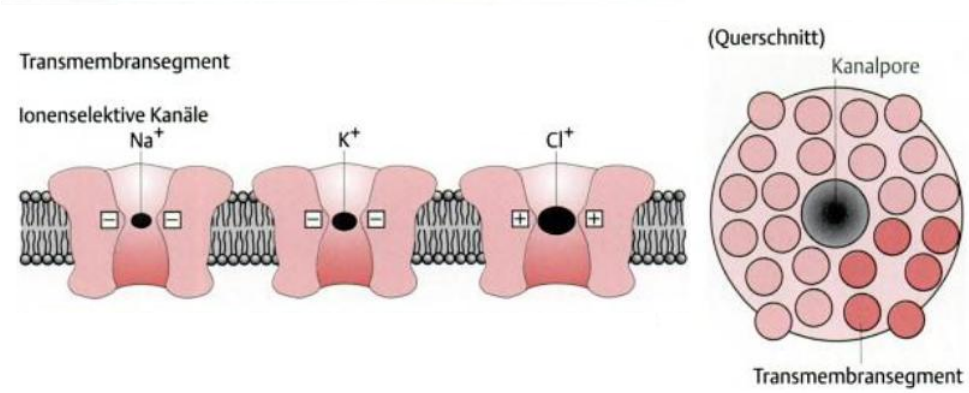


Abbildung 2.1: *Membran mit selektiven Ionenkanälen. Rechts ein Transmembransegment im Querschnitt. (Quelle: Reichert (2000) [3], Abbildung modifiziert.)*

Der Potenzialunterschied kann über die Goldmann-Hodgkin-Katz-Gleichung

$$U_M = \frac{RT}{zF} \cdot \ln \frac{P_{Na} \cdot [Na^+]_a \, P_K \cdot [K^+]_a \, P_{Cl} \cdot [Cl^-]_i}{P_{Na} \cdot [Na^+]_i \, P_K \cdot [K^+]_i \, P_{Cl} \cdot [Cl^-]_a}$$

unter Berücksichtigung der Ionenkonzentrationen aus Tabelle 2.1 berechnen werden.

Ionen	Intrazellulär (i)	Extrazellulär (a)
Na^+	10 mmol/l	145 mmol/l
K^+	124 mmol/l	2 mmol/l
Cl^-	2 mmol/l	77 mmol/l

Tabelle 2.1: *Typische Ionenkonzentrationen in den intrazellulären und extrazellulären Flüssigkeiten. Reichert (2000) [3]*

Ausgelöst durch einen Reiz, kommt es zum Einstrom von positiv geladenen Na^+ Ionen vom extrazellulären Raum in das Innere der Zelle und zum Anstieg des Membranpotenzials. Dieser Vorgang wird als Depolarisation bezeichnet. Überschreitet das Membranpotenzials einen gewissen Schwellenwert (bei Muskel- und Nervenzellen etwa -60mV [4]), öffnen die spannungsabhängigen Na^+-Kanäle und es kommt zum Auslösen eines Aktionspotentials (AP). Dabei ist nur von Bedeutung, ob der Schwellwert überschritten wird, und es ist unerheblich wie stark der auslösende Reiz war. In weiterer Folge wird die selektive Leitfähigkeit der Membran für Na^+-Ionen erhöht und es kommt zu einem immer rascheren Einstrom von Na^+ Ionen(Abb. 2.2).

Dies bewirkt einen ebenso schnellen Anstieg des Membranpotenzials und dauert an, bis die Na^+-Kanäle vollständig geöffnet sind ($\sim$1ms). Kurze Zeit nach dem Öffnen der Na^+-Kanäle kommt es ebenfalls spannungsbedingt, zu einer gesteigerten Permeabilität der K^+-Kanäle und dadurch zu einem Ausstrom der K^+-Ionen. Dieser Vorgang läuft allerdings wesentlich langsamer und auch länger ab, weshalb das Membranpotenzial wieder

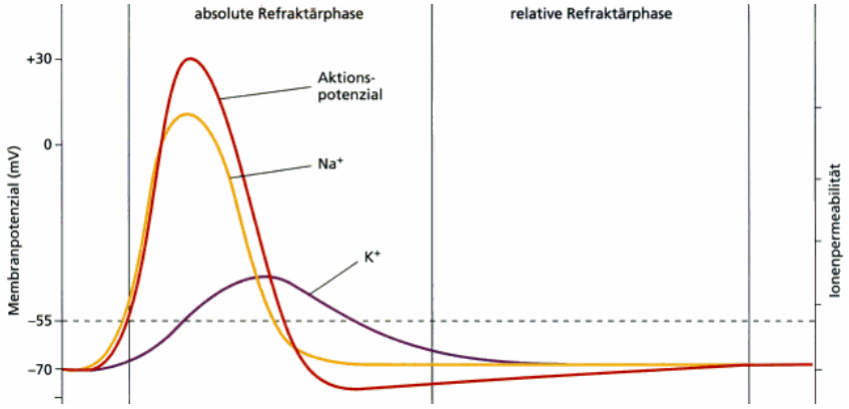


Abbildung 2.2: *Aktionspotential mit Leitfähigkeiten der Ionenkanäle (Quelle: Silverthorn (2009) [5])*

repolarisiert und gegen das Ruhepotential strebt und dieses sogar unterschreitet. Diese Phase wird auch als Hyperpolarisation bezeichnet.
In der Zeit vom Auslösen des Aktionspotentials bis zur Wiederherstellung des Ruhepotentials kann kein weiteres Aktionspotential ausgelöst werden. Während diesen Zeitraums befindet sich die Zelle im so genannten Refraktärzustand. Dabei wird zwischen absoluter und relativer Refraktärzeit unterschieden. Während innerhalb der absoluten Refraktärzeit gar kein Aktionspotential ausgelöst werden kann, erreicht das Aktionspotential in der relativen Refraktärzeit nicht mehr die maximale Amplitude (Abb. 2.3).

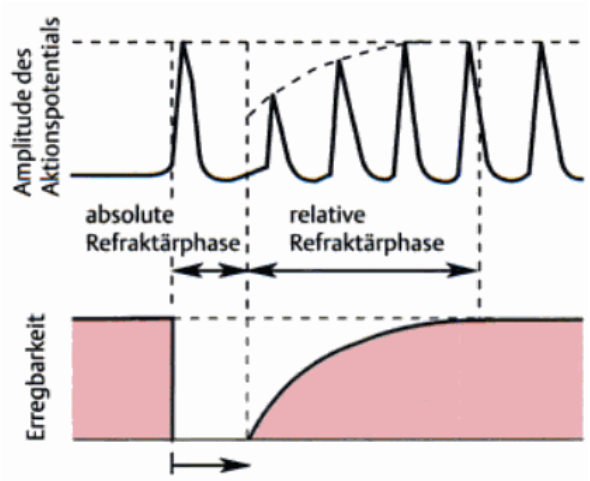


Abbildung 2.3: *Refraktärzeiten nach einem Aktionspotential (Quelle: Reichert (2000) [3])*

Muskel und Muskelfasertypen

Ein skeletaler Muskel setzt sich aus einer großen Anzahl von Faserbündel zusammen, die ihrerseits mehrere Muskelfasern beinhalten. Eine Muskelfaser besteht wiederum aus mehreren Myofibrillen (Abb. 2.4). Diese Myofibrillen bestehen aus Aktinfillamenten mit dazwischen liegenden Myosinfilamenten. Die beiden Proteine Myosin und Actin sind in definierten Einheiten angeordnet und werden als Sarkomer bezeichnet [6].

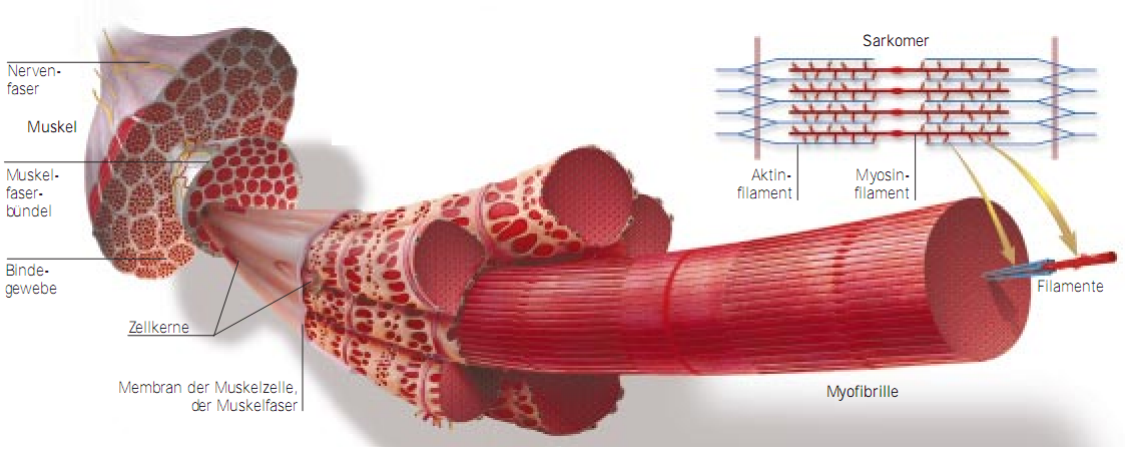


Abbildung 2.4: *Architektur eines skeletalen Muskels (Quelle: Kasnot (2005) aus [7])*

Die Myosinfilamente mit ihren Myosinköpfchen sind zusammen mit dem Aktinfillamenten die Hauptakteure bei der Muskelkontraktion. Wird die Muskelfaser elektrochemisch erregt, hängen sich die Myosinköpfchen am Actin ein und knicken um. Dadurch kommt es zu einer Verschiebung des Aktins und somit zu einer resultierenden Kontraktion der Muskelfaser. Eine Muskelfaser wird von einer motorischen Nervenfaser (α-Motoneuron) angesteuert, wobei ein α-Motorneuron über Axomterminale mehrere Muskelzellen aktiviert.. Eine Gruppe von Muskelfasern, die vom selben Motorneuron angesteuert wird, nennt man auch motorische Einheit.
Ein einzelner Stimulationsimpuls führt zu einer schwachen Kontraktion der Muskelfaser und entwickelt in etwa nur ein Drittel der Kraft, die der Muskel imstande ist zu produzieren [6]. Eine solche Kontraktion wird auch Twitch (engl. = Zuckung) genannt. Durch ein Wiederholen von solchen Einzelreizen, können sich diese zu einer tetanischen Reizung überlagern und so die maximale Muskelkraft erreichen. (Abb. 2.7). Die erzeugte Kraft ist dabei abhängig von der Frequenz der Stimulation und von der Art des Muskeltyps. Einige Muskeln kontrahieren und entspannen sehr schnell(schnelle Muskeln), während andere(langsame Muskeln) mehr Zeit dafür benötigen. In Abb. 2.5 ist zu erkennen, dass schnelle Fasern erst bei höheren Frequenzen die maximale Kraft entwickeln und dass langsame Muskeln bereits bei ca. 20Hz tetanisch kontrahieren. Dies zeigt dass die entwickelte Kraft eines speziellen Muskels sowohl von der Stimulationsfrequenz, als auch aus den bestehenden Muskelfasertypen abhängt.

Muskelfasern lassen sich, wie schon erwähnt, auf Grund von unterschiedlichen Kotrak-

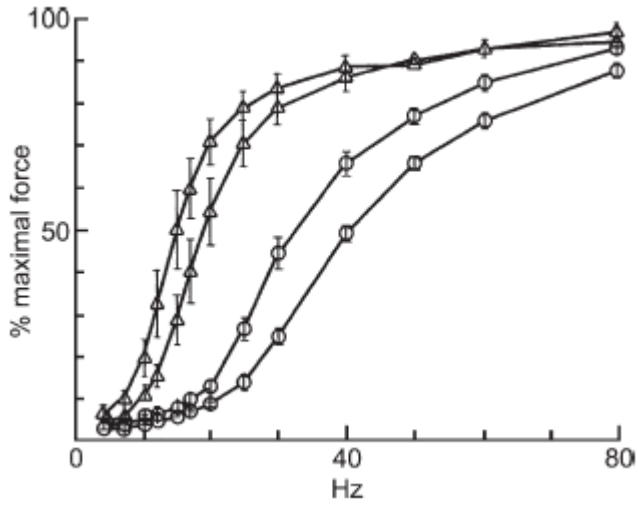


Abbildung 2.5: *Zeigt die maximale tentanische Kraft von jeweils zwei langsamen (Δ) und schnellen (Φ) Motoreinheiten über der Stimulationsfrequenz. Quelle: Vrbová (2008) [6]*

tionsgeschwindigkeiten und ihrer Ermüdungsresistenz, stark vereinfacht in zwei Typen kategorisieren.

Typ I Faser: Die langsame Faser oder auch ST-Faser (engl. slow twitch = langsam kontrahierende Faser), hat den kleinsten Querschnitt und kontrahiert mit einer Kontraktionsdauer von 100-200ms bei geringer Kraftentwicklung. Fasern vom Typ I ermüden langsam und erholen sich rasch.

Typ II Faser: Fasern vom Typ II werden auch FT-Faser (engl. fast twitch = schnellzuckende Faser) bezeichnet. Motoreinheiten, dieses Fasertyps, ermüden relativ schnell und werden vor allem bei hohen Kraftaufwand beansprucht. Dieser Fasertyp kontrahiert und entspannt sehr schnell.

Neben der Frequenz der auslösenden Aktionspotentiale, kann die Kraft des Muskels auch durch die Anzahl der aktivierten Motoreinheiten gesteuert werden. Dabei werden bei geringem Kraftaufwand zunächst die kleineren Strukturen und somit Fasern vom Typ I aktiviert. Wird mehr Kraft benötigt, werden nach und nach die Motoreinheiten mit höheren Empfindlichkeitsschwellen, also Typ II-Motoreinheiten, rekrutiert. Wichtig ist dabei zu bemerken, dass bei FES die Rekrutierung genau umgekehrt erfolgt und dies als inverses Rekrutment bezeichnet wird. Alpha Motoneurone, welche Type II Muskelzellen ansteuern, haben einen größeren Querschnitt und sind somit sensitiver auf externe elektrische Felder. Dies kommt zustande, weil bei den größeren Strukturen der Abstand zwischen den Ranvier-Knoten[1] größer ist, und dadurch bei gleicher Stromstärke eine

höhere transmembrane Spannungsdifferenz induziert werden kann.

Muskel- und Nervfasern besitzen beide eine elektrisch aktive Membran, wobei die Nervenzellen einen wesentlich geringeren Aktivierungsschwellenwert besitzen (Abb. 2.6). [8].

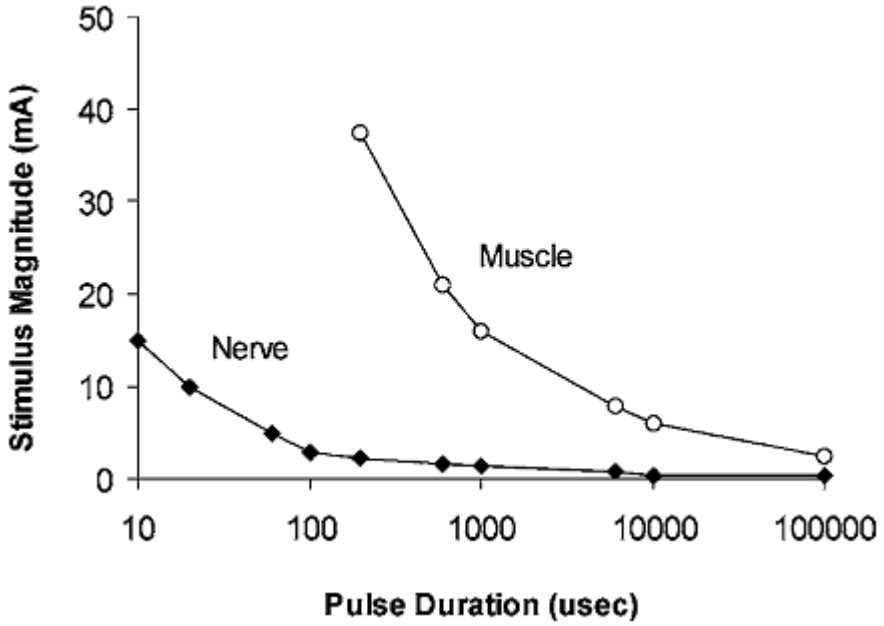


Abbildung 2.6: *Nerven können mit kürzeren Impulsdauern und Stimulationsamplituden erregt werden als dies bei der direkten Erregung von Muskelfasern der Fall ist.(Quelle: P.Hunter Peckham (2005) [8]*

Bei der neuromuskulären Stimulation ist das Motorneuron vom Vorderhorn des Rückenmarks bis zur neuromuskulären Verbindung im stimulierten Muskel intakt. Ein Defekt des Motorneurons auf dieser Strecke reduziert die Anwendung von funktioneller Elektrostimulation. Dies ist beispielsweise bei Poliomyelitis (kurz Polio) und peripheren Nervverletzungen der Fall. Zusätzlich muss die Funktionalität der neuromuskulären Verbindung und des Muskelgewebes gegeben sein, um eine Stimulation erfolgreich durchzuführen zu können. Daraus ist zu schließen, dass die neuromuskulären Elektrostimulation bei erregbaren Motorneuronen, intakten neuromuskulären Verbindungen und gesunden Muskel auch eine Anwendung finden kann.

Die neuromuskuläre Elektrostimulation wird mit mindestens zwei Elektroden durchgeführt, wobei zwischen monopolarer und bipolarer Konfiguration unterschieden wird. Bei beiden Konfigurationen wird die aktive Elektrode in der Nähe des zu stimulierenden peripheren Nervs platziert. Bei der monopolaren Elektroden-Konfiguration wird die indifferente Elektrode in einiger Entfernung, zum Beispiel im Bereich der Sehne angebracht,

[1] Myelinfreier Axonabschnitt. Die Myelinscheide wirkt isolierend und erhöht den Widerstand der Axonmembran. Fördert eine schnelle Fortleitung von Aktionspotentialen.

und besitzt häufig eine größere Oberfläche als die Aktivelektrode. In der bipolaren Konfiguration wird die Referenzelektrode direkt neben der aktiven Elektrode angebracht. Es gilt zu erwähnen, dass die bipolare Stimulation eine höhere Selektivität aufweist, weil ein so platziertes Elektrodenpaar ein lokal stärkeres elektrisches Feld erzeugt [9].

Weiters wird im Rahmen der Elektrostimulation zwischen spannungs- und stromgeregelten Geräten unterschieden. Spannungsgeregelte Stimulatoren halten die Ausgangsspannung konstant. Da durch die Beziehung von Spannung und Widerstand, bei steigender Elektrodenimpedanz, z.B. bei Abziehen der Elektrode von der Haut, die Stromstärke sinkt, können dadurch Verbrennungen der Haut vermieden werden. Spannungsgesteuerte Stimulatoren werden deshalb für die transkutane Elektrostimulation eingesetzt. Dem gegenüber kommen Geräte mit Konstantstromquellen eher bei Implantaten zum Einsatz, weil bei ihnen die abgegebene Leistung, unabhängig von der Elektrodenimpedanz, immer die selbe ist.

Für die Vergleichbarkeit von Studien sind einige Parameter bei FES bzw. NMES von besonderer Bedeutung.

Impulsform: Die Stromimpulse für die Elektrostimulation können entweder eine monophasiche oder biphasiche Form aufweisen. Monophasische Kurvenformen bestehen aus sich wiederholenden, einseitigen für gewöhnlich negativen (kathodischen) Impulsen. Biphasische Impulsformen setzten sich aus einem negativen und positiven Anteil zusammen. Der erste negative Impulsanteil führt zum Auslösen eines Aktionspotentials, während der zweite Impulsanteil dem Ausgleich der Ladungsverschiebung dient. Dies ist besonders wichtig, da es bei der Anwendung von Gleichstrom an der Kontaktstelle von Elektrode(metallisch) und Gewebe(Elektrolyt) zur Elektrolyse kommt. Dies hätte die Auflösung der Anode und deren Eintrag ins Gewebe zur Folge.

Stimulationsfrequenz: Ganz allgemein lässt sich sagen, dass für die Stimulation der menschlichen Muskulatur, Frequenzen zwischen 1Hz bis 100Hz als typisch gelten. Generell wird der Frequenzbereich von 10Hz - 25Hz für die Aktivierung von langsamen Muskelfasern und der Bereich von 45Hz - 75Hz für schnelle Fasern eingesetzt.

Pulsweite: Durch die Pulsweite lässt sich die abgegebene Leistung also die Intensität der Stimulation regulieren. Die Pulsweite wird typischerweise in μs angegeben und liegt im Bereich von $80\mu s$ bis $500\mu s$.

Amplitude Ein weiterer Stimulationsparameter für die Regelung der Intensität ist die Stimulationsamplitude. Abhängig davon ob mit konstanten Strom- oder Spannungsquellen stimuliert wird, wird die Amplitude in Volt[V] oder Milliampere[mA] angegeben und liegt für Spannungsstimulationen zwischen 0 - 100 Volt.

Ruhezeit: Die Ruhe- oder Pausezeit ist die Zeitspanne in der kein Stimulationsstrom fließt. In der Literatur ist in diesem Zusammenhang auch oft von ON-OFF Relation die Rede. Dies bezeichnet das Verhältnis von Stimulationszeit zur nachfolgenden Ruhezeit.

Muskeltraining

Unter Training wird die Adaption des Körpers auf zusätzliche Aktivität bezeichnet. Ein Muskeltraining kann für therapeutische Zwecke gezielt eingesetzt werden um einerseits eine verlorengegagene Funktionalität, z.B. bei einer reversiblen Muskelschwäche aufgrund einer Atrophie wieder herzustellen (Restitution), oder eine körperliche Beeinträchtigung z.B. bei Instabilität, Amputation oder Gelenkversteifung auszugleichen(Kompensation). Beim Training unterscheidet man im allgemeinen zwei Arten von Mukelkontraktionen:

(a) Isometrische Muskelkontraktion: Bei der isometrischen (statischen) Kontraktion treten intramuskuläre Spannungsänderungen auf, ohne dass es zu einer Längenänderung der Muskeln kommt. Diese Kontraktionsform tritt hauptsächlich bei der Stabilisierung des Stützapparates auf.

(b) Dynamische Kontraktion: Dabei ändert sich die intramuskuläre Spannung und es kommt zu einer Verkürzung (bei konzentrischen Kontraktionen) oder einer Verlängerung (bei exzentrischen Kontraktionen) des Muskels.

Durch das Muskeltraining mittels Elektrostimulation kann, wie beim herkömmlichen Krafttraining, die Ausdauer des Muskels als auch der Kraftzuwachs gesteigert werden [10]. Durch das Training mit Elektrostimulation kommt es zum Umbau des Muskels. Relative gesehen nehmen Type I Muskelfaseren zu. Im Gegensatz zum konventionellen Training ist jedoch beim Muskeltraining mit NMES, eine Umwandlung der Fasertypen nicht nur von Typ II-Fasern in Typ I-Fasern möglich, sondern auch in die umgekehrte Richtung [11].

Gemäß den Gesetzen der Elektrophysiologie (vgl. 2.1.1, Seite 6) kann ein Muskeltraining mit neuromuskulärer Elektrostimulation bei tetanischer Kontraktion erfolgen. Die Frequenz bei der einzelne Impulse zu einer tetanischen Kontraktion verschmelzen liegt bei etwa 30Hz. Die Kontraktionsamplitude steigt oberhalb dieser Frequenz nur mehr sehr langsam an und erreicht ihr Maximum bei einer Frequenz von 60 - 70Hz. Wegen der Tatsache, dass die Ermüdbarkeit des Muskels beim Training ab einer Frequenz von 50Hz sehr rasch zunimmt, macht ein Muskeltraining laut Schönle (2004) [10] mit höheren Frequenzen keinen Sinn.

Wie auch das konventionelle Muskeltraining hat auch das Training durch Elektrostimulation seine Gültigkeit [13]. Sowohl beim Muskelaufbau für den Kraftzuwachs, als auch in der Rehabilitation. Während beim Training mit NMES nur der motorische Nerv direkt stimuliert wird, werden beim herkömmlichen Training höhere Ebenen des zentralen Nervensystems involviert. Auch der beim Trainieren bekannte “Cross-Education-Effekt”, wobei es beim Belasten einer Muskelgruppe, zum Beispiel des rechten Quadriceps, auch zu einem Muskelzuwachs im linken Bein kommt, kann durch das Training durch externe elektrische Stimulation beobachtet werden [14].

In der Vergleichsstudie von Bax et al. [15] werden unter anderem aber auch die Grenzen des Muskeltrainings mit NMES aufgezeigt. Zusammenfassend lässt sich sagen,

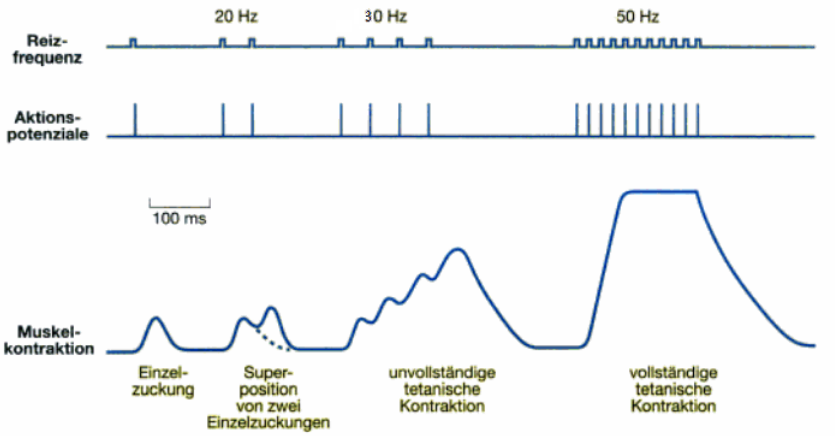


Abbildung 2.7: *Bei einer Frequenz von etwa 20Hz, kommt es zur Überlagerung von Einzelzuckungen. Ab 30Hz beginnen die Einzelzuckungen zu verschmelzen um bei etwa 50Hz in eine tetanische Kontraktion überzugehen, die für das Training des Muskels entscheidend ist. (Quelle: Speckmann (2008) [12] (Abbildung modifiziert)*

dass ein Muskeltraining mit Elektrostimulation einen besseren Trainingserfolg liefert als wenn gar nicht trainiert wird. Ein herkömmliches Muskelaufbautraining kann durch die neuromuskuläre Elektrostimulation also nur unterstützt, keinesfalls aber ersetzt werden. Die Abbildung 2.8 illustriert die Grenzen der Anwendbarkeit der Elektrostimulation im Muskeltraining.

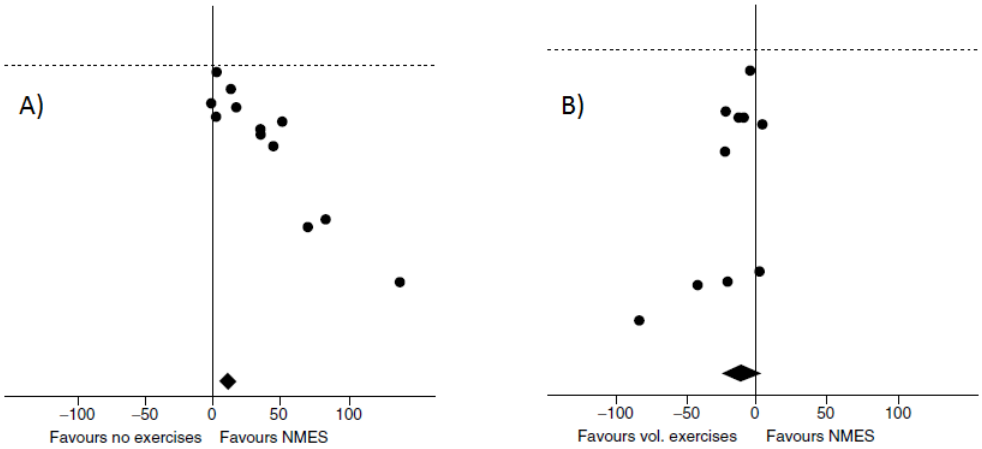


Abbildung 2.8: *Beide Graphen zeigen die Ergebnisse verschiedener Studien zum Thema Elektrostimulation . In (A) wird bei Probanden die im Gegensatz zur Kontrollgruppe (kein Training) mit NMES trainierten, eine klare Bevorzugung von Muskeltrainings mit NMES gezeigt. Während in (B) beim Vergleich von konventionellen Muskeltraining zum NMES-Training, die Tendenz klar zum physikalischen Training tendiert. (Quelle: Bax et al. (2005) [15])*

Das neuromuskuläre System im Alter

Die Muskelgröße und die damit verbundene neuromuskuläre Funktion ändern sich im Laufe eines Lebens dramatisch. Während in den ersten Lebensjahren ein sehr schnelles und stetes Wachstum vorherrscht, kommt es bei älteren Personen zu einem dominierenden Rückgang der Muskelgröße. Ist der Zuwachs an Muskelmasse vor der Geburt vor allem durch eine Zunahme an Muskelfasern gekennzeichnet, wird in der Kindheit eine Vergrößerung vorwiegend durch die Größenzunahme der postmitotischen Muskelzellen erreicht. Im Alter von etwa 30 Jahren wird die maximale Muskelfunktion erreicht. Ab diesem Zeitpunkt kommt es zu einer Abnahme der körperlichen Leistungsfähigkeit die durch eine geringere Adaptionsfähigkeit der Muskulatur bedingt wird. Ab dem vierzigsten Lebensjahr verliert der Mensch durchschnittlich 1-2% seiner Muskelmasse pro Jahr. Zwischen dem 50. und 60. Lebensjahr etwa 1.5% und im höheren Alter sogar um die 3% pro Jahr. Beeinträchtigt die Abnahme von Muskelmasse und Muskelkraft im fortgeschrittenen alter auch die körperliche Funktionalität, spricht man vom Krankheitsbild der Sarkopenie [16, 17]. Der Begriff Sarkopenie leitet sich vom Griechischen ab und bedeutet soviel wie "Fleischmangel" (sarxs = Fleisch, penia = Mangel) und wurde 1988 von Rosenberg geprägt. Relativ betrachtet ist die Abnahme der Muskelmasse größer als die der Muskelkraft und die verschiedenen Fasertypen und Muskelgruppen sind unterschiedlich betroffen [18].

2.1.2 Neuromuskuläres Monitoring

Mit quantitativen Assessment-Methoden soll das Muskeltraining mit Elektrostimulations überwacht werden. Die Anzahl der Methoden zur Messung der Muskelfunktion ist sehr vielfältig und reicht von invasiven Methoden (z.B Diopsie) bis zu bildgebenden Verfahren wie zum Beispiel Ultraschall. Ich möchte hier keine vollständige Aufzählung und Beschreibung aller zur Verfügung stehender Methoden geben sondern mich lediglich auf die zwei Arten der Funktionsmessung beschränken, die auch in unserem Projekt Anwendung finden.

Elektromyographie

Die Elektromyographie (EMG) ist eine Methode um die elektrischen Aktivität von Muskeln messtechnisch zu erfassen. Im wesentlichen stehen zwei Methoden zur Erfassung eines EMG-Signals zur Verfügung. *a*) Die minimal invasive Erfassung mit Nadelelektroden erlaubt eine etwas höhere räumliche Auflösung und bietet einen etwas günstigeren Signal-Rauschabstand. Bei dieser Methode werden dünne Elektroden direkt in den Muskel gestochen [19] um das EMG-Signal abzuleiten. *b*) Die Ableitung des myoelektrischen Signals über Oberflächen-Elektroden auf der Haut.

Das EMG-Signal ist eine Repräsentation des elektrischen Potentialfeldes, welches durch die Depolarisierung der äußeren Muskelfasermembran (dem Sarkolemma) hervorgerufen wird. Das Signal entsteht durch die elektrische Aktivität der Muskelfaser

während einer Kontraktion. Deshalb kann die Signalquelle an den depolarisierten Zonen der Muskelfaser lokalisiert werden [20].

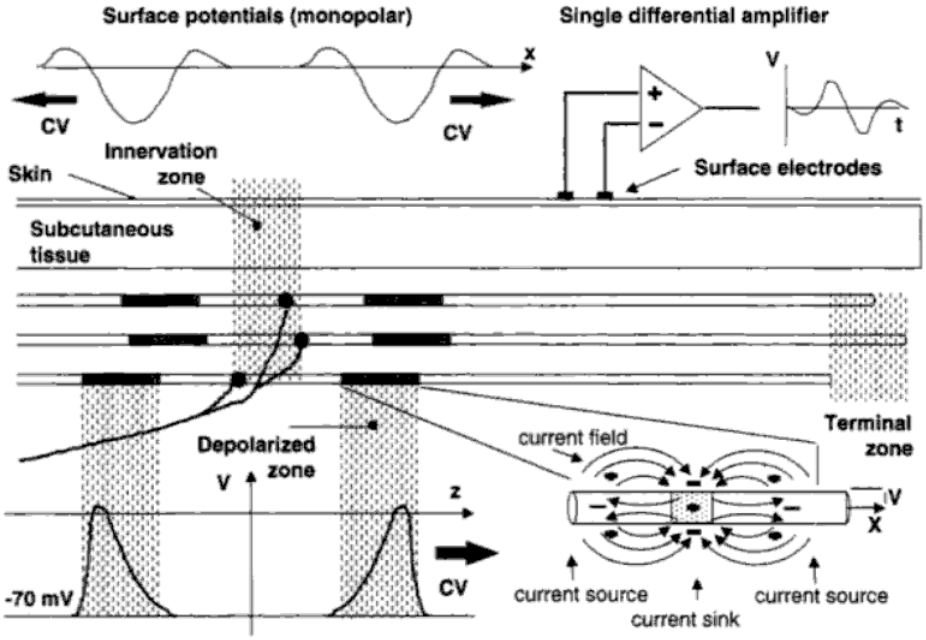


Abbildung 2.9: *Zeigt wie sich ein Aktionspotential vom Inervationspunkt entlang der Muskelfasern in beiden Richtungen ausbreitet. Die Quelle eines EMG-Signal kann als "Current Tripol" (*2 *Quellen,* 1 *Senke) aufgefasst werden. (Quelle: Merletti (2004) [20])*

In der Regel wird durch die Elektromyographie immer die elektrische Aktivität vieler, an einer Kontraktion beteiligten motorischen Einheiten gemessen. Das EMG-Signal muss also als Interferenzmuster sich überlagernder Einzelsignale aufgefasst werden. Dabei spielt sowohl der Abstand der Elektroden zueinander als auch der Abstand der Elektrode zur Signalquelle eine entsprechende Rolle beim Zustandekommen des komplexen EMG-Signals. Die elektrische Antwort eines Muskels auf die Stimulation seines Muskelnervs wird auch als M-wave bezeichnet und repräsentiert wie schon erwähnt, die synchronisierte Aktivität aller motorischer Einheiten eines Muskels die durch die externe Stimulation angeregt wurden [21].

Das myokinematische Signal

Eine weitere Methode um die Muskelaktivität messbar zu machen beschreibt die Akzelerographie. Es ist ein myographisches Verfahren bei dem die Beschleunigung des stimulierten Muskels gemessen wird. Die Messmethode bedient sich des zweiten Newtonschen Gesetzes, wonach gilt: Kraft = Masse x Beschleunigung (F=m*a). Eine bekannte, konstante und freibewegliche Masse wirkt im Falle ihrer Beschleunigung eine Kraft aus. Durch diese Kraftentwicklung kann auf die Beschleunigung des Systems geschlossen werden. Als

Sensoren werden heute vor allem sogenannte micro-elektro-mechanische(MEMS) Systeme auf Siliziumbasis eingesetzt. Das sind Feder-Masse-Systeme bei denen sowohl die Masse selbst, als auch die Federn aus nur wenigen μm breiten Silizium-Stegen gefertigt sind. (Vgl. Abb. 2.10)

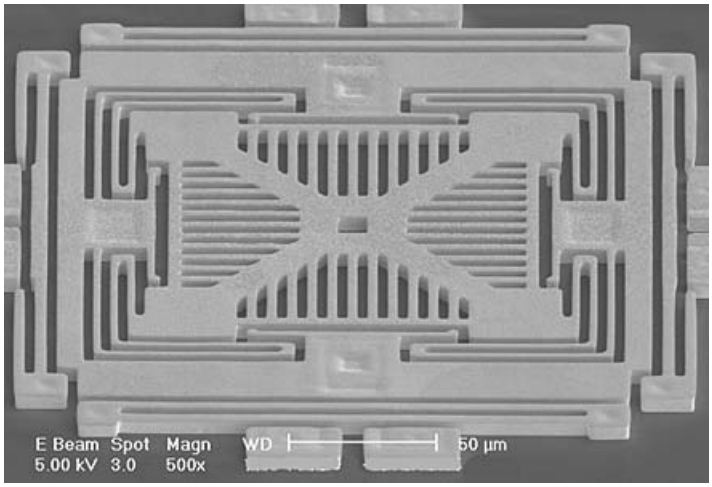


Abbildung 2.10: *3–Achsen Beschleunigungssensors. Die Struktur zeigt auf Federn aufgehängte Masse, gefertigt aus Silizium. Um die MEMS-Struktur vor Staub und Feuchtigkeit zu schützen wird sie im Laufe der Fertigung in ein Chip-Gehäuse aus Plastik gepackt. (Quelle: Guillou (2011) [22])*

Durch Auslenkung der Masse bei Beschleunigung, verändert sich zwischen dem beweglichen Teilen und den fixierten Bezugselektroden die elektrische Kapazität. Durch die gemessene Kapazitätsänderung kann auf die Beschleunigung geschlossen werden. Solche Bauelemente sind durch die Massenfertigung sehr billig geworden [23] und erlauben durch geeignete Schnittstellen, wie zum Beispiel der in unserem Projekt zur Anwendung kommende Sensor (LIS3LV02DL, ST Microelectronics), welcher über eine SPI- und I^2C-Schnittstelle verfügt [24], einen sehr komfortablen Einsatz.

2.2 Informationstechnische Hintergründe

Die im Rahmen dieser Diplomarbeit erstellte Software wurde für das Betriebssystem Windows (ab Version Windows XP) entwickelt. Die Tatsache, dass die Software auf eine Betriebssystem-Familie begrenzt werden konnte erleichterte die Auswahl der verwendeten Technologie. Als erster wichtiger Designschritt, im Rahmen der Umsetzung des Softwareprojekts, wurde daher als Plattform für die Entwicklung der Software auf das .NET-Framework gesetzt.

2.2.1 Das .NET Framework

Das .NET Framwork ist eine Entwicklungsplattform die im vollem Umfang nur für das Windows-Betriebsystem zur Verfügung steht. Aber es existieren auch Implementierungen für andere Betriebssysteme die jedoch nur beschränkte Funktionalität bieten. Hierbei ist vor allem das Mono-Projekt zu erwähnen welches das .NET-Framework als "Open-Source" für mehrere Betriebssysteme umsetzt [25].

Die Entwicklung des .NET Framework wurde in den späten 1990 Jahren unter dem Namen Next Generation Windows Services (NGWS) begonnen. Seit der Veröffentlichung der Version 1.0 im Jahre 2002 entwickelte sich das Framework kontinuierlich weiter und existiert heute in seiner vierten Version (Stand: März 2011). Seit der Version 3.0 ist auch die Windows Presentation Foundation(WPF) [Vgl. 2.2.2], durch die in unserem Projekt die Visualisierung umgesetzt wurde, Teil des .NET-Frameworks. (Vgl. Abb. 2.11)

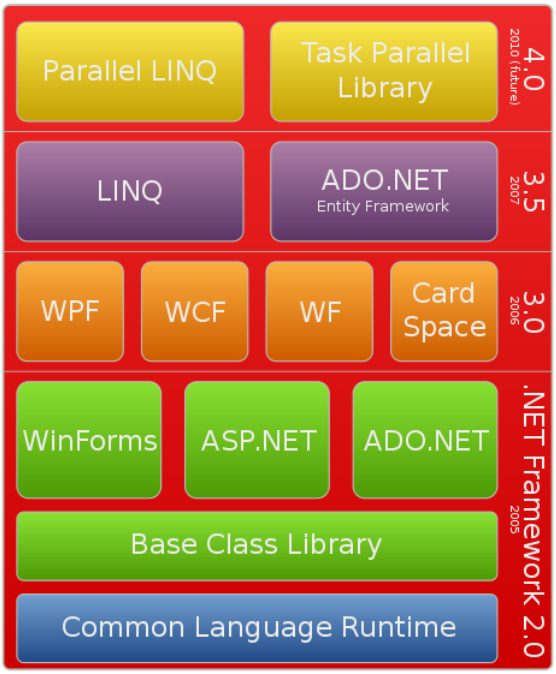


Abbildung 2.11: *Stack des .NET Frameworks (Bildquelle: Wikipedia (2011) [26])*

Das .NET Framework besteht im wesentlichen aus zwei Hauptkomponenten. Der Common Language Runtime (CLR) und der .NET-Framework Klassenbibliothek. Die CLR ist quasi das Herzstück der .NET-Architektur und verwaltet die Codeausführung zur Laufzeit, stellt wichtige Dienste wie etwa das Speichermanagement oder die Thread-Verwaltung zur Verfügung [27]. Die Laufzeitumgebung des .NET-Frameworks (CLR)ist vom Konzept her ähnlich der Java-Virtual-Machine (JVM), mit der Ausnahme dass die JVM den Bytecode interpretiert, während beim .NET-Konzept der Programmcode zunächst in die sogenannte Common Intermediate Language (CIL) übersetzt wird und dieser dann für die (CLR) kompiliert wird. Die Common-Language-Runtime dient somit

allen Sprachen als Laufzeitumgebung, die sich durch die Common Intermediate Language repräsentieren lassen [28].

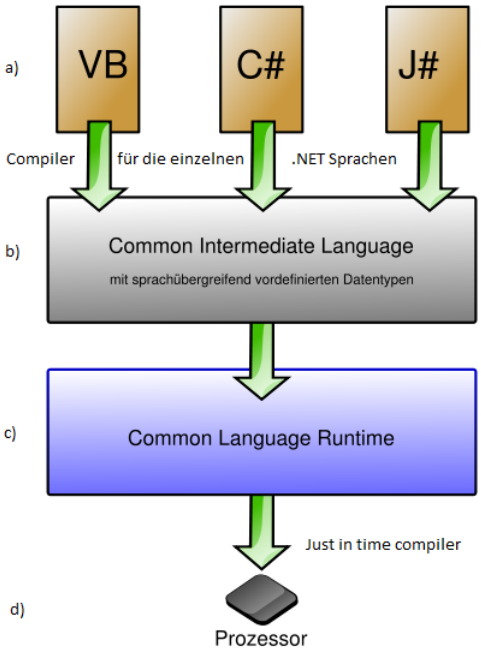


Abbildung 2.12: *a) Programmcode kann in mehreren .NET Sprachen geschrieben werden. Einzige Bedingung sie müssen sich durch die Common Intermediate Language(CIL) repräsentieren lassen. b) Durch jeweils eigene Compiler wird der Programmcode in einen Bytecode übersetzt. c) Die CLR dient dem CLI-Code als Laufzeitumgebung. Durch einen JIT (Just in time) Compiler wird der Intermediate Code in nativen Maschinencode (unmanaged code) übersetzt, der vom Betriebssystem ausgeführt wird. d) (Bildquelle: Wikipedia (2011) [26])*

Microsoft unterstützt durch die Entwicklungsumgebung Visual-Studio 2010 schon von Haus aus 5 sogenannte .NET-Sprachen.

(1) C++/CLI: Damit können C++ - Entwickler wie gewohnt Applikationen für das .NET-Framework erstellen.

(2) *C#*: Eine im Zuge der .NET-Framwork-Entwicklung kreierte Programmiersprache mit java-ähnlicher Syntax.

(3) Visual Basic.NET: Eine proprietäre von Microsoft entwickelte, auf Basic aufbauende Sprache. Nach meiner Meinung und der Meinung eines professionellen Programmierers[1] ist die Sprache Basic, durch die mögliche Verwendung des .NET-Frameworks, nun auch für den professionellen Einsatz in der Softwareproduktion einsetzbar.

(4) *J#*: Eine für das .NET-Framework exportierte Implementierung von Java.

(5) *F#*: Eine funktionale Programmiersprache für die Verwendung in Zusammenhang mit dem .NET - Framework.

Von Drittanbietern [29,30] werden auch noch andere Sprachen für das .NET-Framework zur Verfügung gestellt. Das Besondere an der Mehrsprachigkeit des .NET-Frameworks ist jedoch nicht die Tatsache dass mit mehreren Sprachen auf dem Framework entwickelt werden kann, sondern dass ein und das selbe Programm, gleich aus mehreren Sprachen programmiert werden kann. Deshalb wurde die CLS (Common Language Specification) definiert, die einen Basissatz von Sprachfeatures bereitstellt und Regeln für die Verwendung dieser Features definiert [31]. Den vollen Umfang des .NET-Framworks zu beschreiben würde den Rahmen dieser Einführung ganz leicht sprengen. Ich möchte deshalb, vor allem in Bezug auf die Umsetzung des Projektes, noch auch eine sehr wichtige Komponente eingehen.

2.2.2 Die Windows Presentation Foundation

Die Windows Presentation Foundation (WPF) beschreibt ein neues grafisches Repräsentationssystem für Windows Applikationen. WPF ist auflösungsunabhängig und nutzt, via DircetX, direkt die Kapazität moderner Grafikhardware. In Zeiten vor WPF nutzten Entwickler über Jahre hinweg zwei wesentliche Module des Windows-Betriebssystems für grafische Benutzerschnittstellen:

- User32 stellt das für Windowsapplikationen typische "Look and Feel" von Buttons, Textfelder, etc. zur Verfügung
- GDI/GDI+ dient dem Rendern von Text und Bildern

Neuere Softwarepakete, wie zum Beispiel *Windows Forms* vereinfachten zwar den Umgang mit User32 und GDI, konnten aber die Limitationen der zugrundeliegenden Systemkomponenten nicht verbessern, da sie immer noch auf diesen aufbauten.Mit der Einführung von WPF wird nun nicht mehr GDI/GDI+ als Technologie verwendet, sondern auf DirectX aufgebaut. Grafische Ausgaben die mit WPF umgesetzt werden, ob 3D-Animationen oder nur das Rendern von Text, werden über die DirectX-Pipline verarbeitet. Damit ist es auch möglich, herkömmliche Anwenderprogramme mit komplexen grafische Ausgaben, wie etwa Transparenz oder Antialiasing auszustatten [32].

Die Windows Presentation Foundation ersetzt aber nicht nur ein altes System sonder integriert auch zahlreiche Neuerungen. Im folgendem Abschnitt werden die Eigenschaften von WPF näher beschrieben.

Auflösungsunabhängigkeit

Traditionelle Windowsprogramme wurden unter der Annahme einer bestimmten Monitorauflösung entwickelt und für diese Auflösung optimiert. Normalerweise sind

[1] Roland Ludwig, langjähriger Arbeitskollege und Chef-Programmierer bei Grabner-Instruments, 1220 Wien.

Programme die unter solchen Vorrausetzungen entwickelt wurden nicht skalierbar. Ein daraus resultierender Effekt ist, dass bei einer höheren Bildschirmauflösung das Programmfenster kleiner dargestellt wird und nur mehr erschwert lesbar ist. WPF umgeht dies indem es die DPI-Einstellungen des Systems und die sogenannten geräteunabhängigen Einheiten verwendet, um die Größe der grafischen Elemente zu berechnen. Eine geräteunabhängige Einheit ist definiert als 1/96 Zoll. Angenommen man erzeugt ein sehr kleines Element mit WPF, dessen Größe 96x96 Einheiten beträgt. Unter Verwendung der Windows- Standardeinstellungen (96 dpi) würde jede geräteunabhängige Einheit genau einem Pixel entsprechen, da WPF folgende Gleichung verwendet um die physikalische Größe eines visuellen Elements zu berechnen.

$$\begin{aligned} \textit{[physikalische Größeneinheit]} &= \textit{[Geräteunabhängige Größeneinheit]} \times \textit{[System DPI]} \\ &= 1/96\ \textit{Zoll} \times 96\ \textit{dpi} \\ &= 1\ \textit{pixel} \end{aligned}$$

WPF nimmt wegen der Systemeinstellungen (96 dpi) also an, dass 96 Pixel 1 Zoll entsprechen. In Wirklichkeit hängt dies aber vom verwendeten Monitor ab. Bei einem 20-Zoll Monitor mit einer maximalen Auflösung von 1600x1200 Pixel ergibt sich nach Pythagoras eine Bildpunktdichte von:

$$\textit{Bildpunktdichte[DPI]} = \frac{\sqrt{1600^2 + 1200^2}\ \textit{pixel}}{20\ \textit{zoll}} = 100\ \textit{dpi}$$

In diesem Fall wäre die Bildpunktdichte etwas höher als vom Betriebsystem prognostiziert und das Element würde etwas kleiner als 1 Zoll dargestellt. Im Vergleich würde ein 15-Zoll Monitor mit einer Auflösung von 1024x768 eine Bildpunktdichte von 85 dpi ergeben, wodurch das Element etwas größer als 1 Zoll dargestellt werden würde. In diesen Fällen verhält sich die Darstellung von WPF Elementen exakt wie traditionelle Windows-Oberflächen. Worin sich nun WPF aber unterscheidet ist das Verhalten bei Veränderung der DPI-Systemeinstellung. Ändert man die Einstellungen z.B auf 120 dpi, nimmt das System an, dass nun 120 pixels benötigt werden um 1 zoll auszufüllen. So ergibt sich:

$$\begin{aligned} \textit{[physikalische Größeneinheit]} &= \textit{[Geräteunabhängige Größeneinheit]} \times \textit{[System DPI]} \\ &= 1/96\ \textit{Zoll} \times 120\ \textit{dpi} \\ &= 1.25\ \textit{pixel} \end{aligned}$$

Das heißt die ”WPF-Rendering-Engine” assoziert eine geräteunabhängige Einheit mit 1.25 pixel. Im Falle des 96x96 großen Elements, würde dieses nun in einer Größe von 120x120(da 96x1.25 = 120) pixel angezeigt werden. Und dies ist genau

das Verhalten das man erwartet. Ein Element mit einer Größe von 1 Zoll auf einen Standardmonitor, sollte auch auf einem Monitor mit größerer Bildpunktdichte 1 Zoll groß sein.

Trennung von Design und Verhalten

Bei der Entwicklung von WPF-Anwendungen hat der Entwickler die Möglichkeit, das Programm sowohl mit herkömmlichen Source-Code zu realisieren, als auch auf eine sogenannte Markup-Sprache zurückzugreifen. In der Realität wird sowohl die Markup-Sprache als auch Programmcode Verwendung finden. In WPF findet XAML (Extensible Application Markup Language) als Marukupsprache Verwendung (Siehe Abschnitt XAML auf Seite 24). XAML ist eine auf XML basierte Sprache die verwendet wird, um die grafischen Elemente einer Softwaranwendung deklarativ zu implementieren [33]. Dies ermöglicht Designern mit keiner oder wenig Programmiererfahrung die Oberfläche einer Anwendung sehr einfach zu gestallten. Der Vollständigkeit wegen, muss hier erwähnt werden, dass der deklarative Ansatzt zur Erzeugung der Benutzeroberfläche weder neu noch einzigartig für WPF ist. Auch für andere Programmiersprachen und Entwicklungsplattformen existieren Lösungen, die auf die der deklarative Erzeugung von grafischen Elementen aufbauen [35].

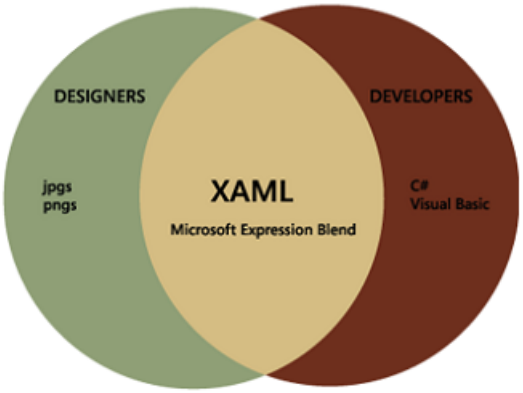


Abbildung 2.13: *XAML bildet als deklarative Markup-Sprache eine Schnittstelle zwischen Designer und Entwickler. (Quelle: Agrusa (2009) [34])*

Architektur von WPF

Die Windows Presentation Foundation wird vorwiegend als “managed code”, implementiert in C# bereitgestellt. Lediglich eine Komponente aus der Multilayer-Architektur von WPF, nämlich der “Media Integration Layer” (milcore) ist als nativer Code zur Verfügung. Diese Schicht zeichnet sich für die Umsetzung von .NET-Objekten in Direct3D-Texturen verantwortlich und wickelt die Kommunikation mit DirectX ab [32]. Abbildung 2.14 zeigt die WPF Architektur und deren Integration in das System.

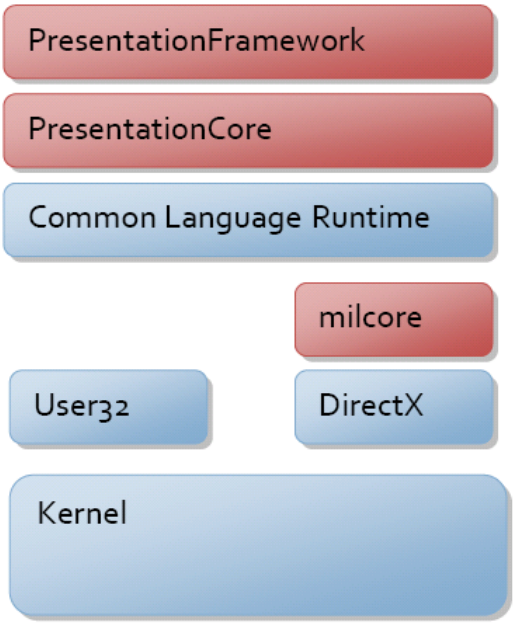


Abbildung 2.14: *In rot die Schichten der WPF, die Services der anderen Schichten (in blau) nutzen. (Quelle: Microsoft Developer Network (2011) [36])*

Presentation Framework In der PresentationFramework-dll sind alle Top-Level-WPF-Elemente enthalten wie beispielsweise Fenster, Panels und andere Steuerelemente. Hier sind auch alle höheren Abstraktionen wie *Styles* oder *Templates* inkludiert.

Presentation Core Diese *Assembly* stellt die Datentypen `UIElement` und `Visual` bereit. Davon erben alle *Shapes* und Steuerelemente wie etwa Schaltflächen oder Textfelder.

milcore Wie schon erwähnt ist dies das Herzstück des Rendering-Systems und als einzige Komponente der WPF in nativen Code implementiert. Das Betriebssystem Windows Vista verwendet die milcore.dll um den Desktop zu rendern.

User32 Wird von WPF zwar noch verwendet um die Rechte einer Anwendung zu bestimmen, ist aber im Renderingprozess nicht mehr beteiligt.

DirectX Low-Level-API durch welche alle grafischen Elemente in WPF wiedergegeben werden.

Kernel Low-Level-APIs um mit dem Betriebssystem zu kommunizieren. Beispielsweise um das Entfernen eines USB-Gerätes zu erkennen.

Die WPF Klassenhirachie

Im Folgenden werden einige fundamentale Klassen der Windows Presentation Foundation und deren Hierarchie innerhalb der WPF erläutert. Außerdem wird auf das Threading-Model der WPF eingegangen um einen Überblick über die Arbeitsweise von WPF zu erhalten. Die meisten Klassen innerhalb der WPF-Klassenhirarchie sind von der `DispatcherObject`-Klasse abgeleitet. Abbildung 2.15 zeigt die wichtigsten WPF Klassen und ihre Beziehung zueinander.

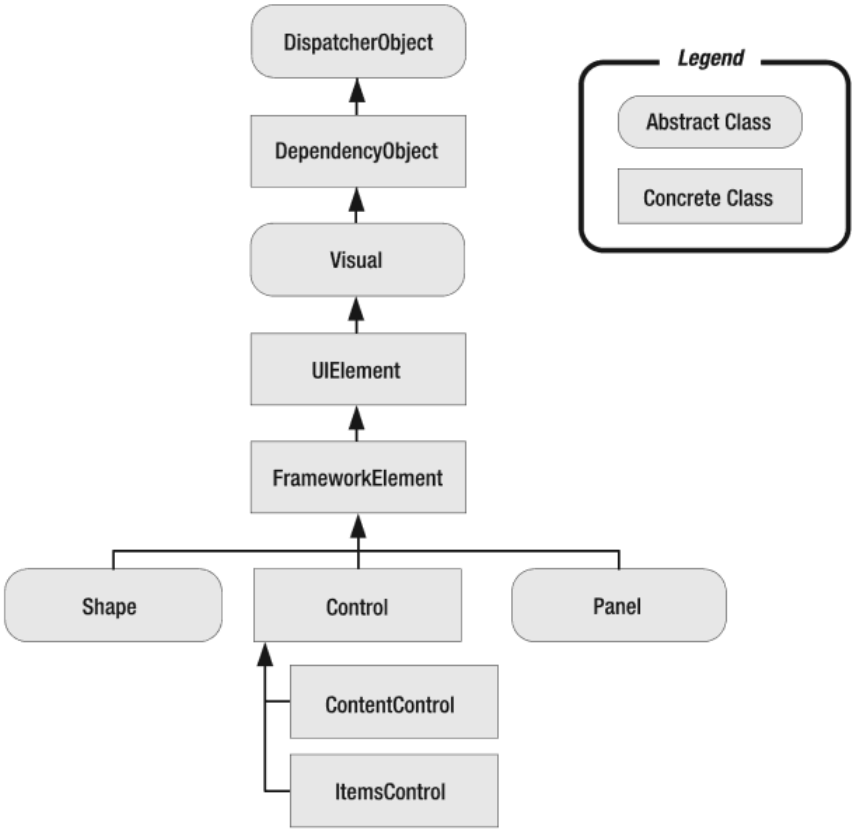


Abbildung 2.15: *Fundamentale Klassen der Windows Presentation Foundation. (Quelle: MacDonald (2008) [32])*

• **DispatcherObject**
Ein `DispatcherObject` stellt die wesentliche Funktionalität im Umgang mit Parallelität und Threading zur Verfügung. WPF basiert auf dem STA(Single Thread Affinity)-Model. Dies bedeutet, dass ein WPF-Objekt nur von einem Thread, in welchem es erzeugt wurde, verwendet werden kann. Ein Zugriff auf das Objekt durch einen anderen Thread, löst zur Laufzeit eine *Exception* aus. Die Kommunikation zwischen mehreren Threads zu ermöglichen, ist die Aufgabe des sogennanten `Dispatchers`. Dieser kann durch Priorisierung der ihm zugeteilten Aufgaben sowohl deren Zugriff als auch die

zur Verfügung stehende Bearbeitungszeit der Aufgaben steuern. Beim Erzeugen eines von `DispatcherObject` abgeleiteten Objekts, erhält dieses Objekt einen Zeiger auf den `Dispatcher`. Das `DispatcherObject` regelt nun den Zugriff auf den aktuellen `Dispatcher` und stellt Methoden für das Prüfen der Zugriffsberechtigung zu Verfügung [32, 36, 37].

• DependencyObject
Ein zentrales Prinzip in WPF ist die Interaktion von grafischen Elementen über Eingenschaftsattribute. Auch bekannt als `DependencyProperties`. Dabei wird eine Eigenschaft, also ein *Property* einer Klasse überwacht und im Falle einer Änderung dieser Eigenschaft, wird das System über die Zustandsänderung benachrichtigt. Die `DependencyObject`-Klasse stellt nun Methoden zur Verfügung, die eine Registrierung solcher "Abhängigkeitseigenschaften" ermöglichen. Die beiden Klassen `DependencyProperty` und `DependencyObject` spielen vorallem bei der Datenbindung einen entscheidende Rolle.

• Visual
Jedes grafische Objekt in WPF ist im Grunde ein Objekt der `Visual`-Klasse. Diese Objekte beinhalten zusätzliche Informationen bezüglich der Darstellung des Elements(z.B. Transparenz od. Transformationen wie Rotation und Skalierung). `Visual` ist auch die Schnittstelle zwischen den verwalteten API Code-Teilen und dem nichtverwalteten `milcore`.

• UIElement
Dieses Modul kümmert sich um die Aktionen bei Eingabe von Maus oder Keyboard. Hier werden die Eingaben in konfortablere Events wie etwa `MouseOver` übersetzt. Auch die Kommandos, an die etwa eine Schaltfläche binden kann um einen Befehl abzusetzten, werden in diesem Modul verarbeitet. Die wohl wichtigste Aufgabe von `UIElement` ist aber die Layoutbehandlung. In früheren grafischen Bibliotheken (z.B. User32) wird nur die absolute Positionierung von Elementen unterstützt, HTML kennt dagegen 3 Layout-Modi (flow, absolut und table). In WPF wirken zwei Mechanismen beim Erstellen der Positionierung von grafischen Elementen, Measure und Arrange. *Measure* erlaubt einem Element zu bestimmen wie viel Platz es einnehmen wird. Mit *Arrange* kann ein übergeordnetes Element alle seine ihm untergeordneten Elemente positionieren und deren Größe bestimmen. Prinzipiell ist jedes Element fähig sich an die natürliche Größe seines Inhalts anzupassen.

• FrameworkElement
Das `FrameworkElement` baut auf dem `UIElement` auf und implementiert Funktionalität die dort lediglich deklariert wurde. Des weiteren wird auch hier das Layout-Konzept fortgeführt und stellt Eigenschaften wie z.B *HorizontalAlignment, VerticalAlignment, etc.* zur Verfügung. Die Kernpunkte des `FrameworkElement` sind jedoch die Datenbindung und die `styles` (Stile) mit denen Beispielsweise einer Schaltfläche ein ganz eigenes Aussehen gegeben werden kann.

• **Shape**
Alle grundlegende geometrische Formen wie etwa Rechtecke, Polygone, Ellipsen od. Linien erben von der abstrakten `Shape`-Klasse.

• **Control**
Ein Steuerelement oder `control` bietet die Möglichkeit mit dem Benutzer zu interagieren. Von dieser Klasse abgeleitet werden beispielsweise Textfelder, Schaltflächen oder Listboxen. In der `control`-Klasse sind auch Eigenschaften zum setzten der Hintergrundfarbe oder des Fonts implementiert. Das wichtigste an der `control`-Klasse ist aber die `Template`-Unterstützung. `Control Templates` sind Vorlagen durch die Steuerelementen ein eigenes Aussehen verpasst werden kann.

• **ContentControl**
Die `ContentControl`-Klasse ist die Basisklasse für alle Steuerelemente die einen Inhalt aufnehmen. Diese Elemente reichen von ganzen WPF-Fenstern bis hin zu einfachen Labels die nur einen String enthalten.

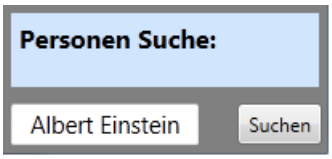
• **ItemsControl**
`ItemControl` bildet die Basisklasse für alle Steuerelemente die eine Sammlung an Elementen verwalten wie zum Beispiel die `ListBox` oder `TreeView`. Solche ListenElemente sind überaus flexibel. Sie können ganz einfache Datentypen wie Strings beinhalten oder andere Steuerelemente bis hin zu eigens kreierten Steuerelementen (`UserControls`).

• **Panel**
Dies ist die Mutterklasse aller Kontainer-Steuerelemente die ein odere mehrere Subelemente enthalten können. Beispiele dafür wären das `Stackpanel` oder `Canvas`.

XAML

Ursprünglich ist XAML ausschließlich für WPF entwickelt worden und trug den Namen "extended Avalon Markup Language". Avalon war der initiale Arbeitstitel für WPF. Heute steht XAML für eXtensible Application Markup Language und kann als eine eigenständige .NET-Sprache aufgefasst werden. XAML basiert auf XML und dient zur deklarativen Erstellung von .NET-Objekten [38]. In Verbindung mit der Windows Presentation Foundation bietet XAML eine komfortable Schnittstelle zwischen dem Design der Oberfläche und der dahinter liegenden Programmlogik (Vgl. 2.13 auf Seite 19). Obwohl nicht zwingend erforderlich, wird bei der Entwicklung von WPF-Anwendungen meist die Auszeichnungssprache XAML für das Erstellen der Benutzerschnittstelle herangezogen.

```
1 <Grid>
2   <Label Content="Personen Suche:"/>
3     <StackPanel>
4        <TextBox />
5        <Button Content="Suchen" />
6     </StackPanel>
7 </Grid >
```



Quellcode 2.1: *Beispiel für simple GUI in XAML*

In Quellcode 2.1 ist eine sehr einfache WPF Benutzerschnittstelle in XAML dargestellt. Durch die XML-ähnliche Syntax ist die Struktur der Oberfläche auch ohne große Kenntnisse der Softwareentwicklung leicht nachvollziehbar. Dies hilft beispielsweise Designern, die nur mit XML vertraut sein müssen, komplexe Oberflächen zu gestallten. Dieser Aspekt ist vor allem in Hinsicht auf konkrete Arbeitsteilung bei einem Softwareprojekt, an dem vielleicht Designer und Programmierer gemeinsam arbeiten, von entscheidender Bedeutung.

Nun sind XAML-Dateien wie oben beschrieben zwar leicht lesbar, bringen aber den Nachteil mit sich, dass die Interpretation bei Ausführung entsprechend länger dauert als bei kompiliertem Code. In WPF wird dieser Umstand dadurch umgangen, dass alle XAML-Files beim Kompilieren als binäre Repräsentation im BAML (für Binary Application Markup)-Format gespeichert werden. Diese Binärdateien werden dann im resultierenden EXE-File, als Resource eingebettet. BAML-Dateien sind in der Größe nicht nur kleiner als ihr XAML-Pendant, sondern durch Compiler-Optimierungen auch wesentlich schneller zur Laufzeit. [39]. Gleichwohl können XAML-Dateien aber auch ohne Übersetzung in BAML-Konstrukte verwendet werden. Dies ist wieder für den Design-Prozess von Bedeutung. Mann will beispielsweise bei Änderungen an direkten Datenquellen, wie xml-Daten, eine sofortige Veränderung der grafischen Anzeige erhalten. In diesem Zusammenhang gilt zu erwähnen dass eine WPF- Anwendung auf 4 unterschiedliche Art und Weisen erzeugt werden kann:

- **Code-Only** Dies ist der traditionelle Ansatz für die Erstellung von WindowsForms. Dabei wird die grafische Oberfläche ausschließlich aus Code-Anweisungen aufge-

baut.

```
Grid grid = new Grid();
Label label = new Label();
  label.Content = "Personen Suche:";
grid.Children.Add(label);
StackPanel stackpanel = new StackPanel();
TextBox textbox = new TextBox();
Button button = new Button();
  button.Content = "Suchen";
  stackpanel.Children.Add(textbox);
  stackpanel.Children.Add(button);
grid.Children.Add(stackpanel);
```

Quellcode 2.2: *Code-only Variante des auf Seite 24 vorgestellten XAML Code aus Bsp. 2.1.*

Durch den in Listing 2.3 gezeigten Programm-Code kann das einfache Beispiel, welches in Code-Teil 2.1 gezeigt wurde als reines C# Programm dargestellt werden ohne XAML zu benötigen.

- **Code und nicht kompiliertes XAML** Dieser sehr interessante Aspekt von WPF und XAML bietet die Möglichkeit, Oberflächen zur Laufzeit dynamisch einzubinden. Dabei wird eine Oberflächendefinition als XAML-Datei vom kompiliertem Programm geladen und dargestellt. Dies ermöglicht einen sehr dynamischen Aufbau der Oberfläche [32].

```
//Lade XAML Inhalt von externer Datei
FileStream s = new FileStream ("Parts.xaml", FileMode.Open);
DependencyObject rootElement = XamlReader.Load(s);
this.Content = rootElement;
```

Quellcode 2.3: *Zeigt wie Teile einer Benutzeroberfläche vom Programmcode geladen werden können.*

Ein entscheidender Nachteil bei dieser Methode ist aber die Tatsache dass man die Ereignisse der grafischen Elemente (*Events*) den entsprechenden Ereignisbearbeitungsmethoden (*Event Handler*) im Code von Hand zuweisen muss.

- **Code und kompiliertes XAML** Dies ist der gewöhnliche und am meisten verwendete Ansatz um Software-Anwendungen mit WPF und XAML zu erzeugen. Dabei werden die in XAML definierten Oberflächen in BAML übersetzt und zusammen mit dem kompilierten Code in eine ausführbare Datei eingebettet. Zur Laufzeit werden die BAML- Dateien extrahiert um daraus die Benutzeroberfläche zu generieren.

- **XAML Only** Es ist auch möglich lauffähige Anwendungen ganz ohne Programm-Code zu erstellen. Die Bezeichnung von solchen XAML-Dateien ist *loose XAML*

[39]. Solche Dateien können direkt mit dem Internet Explorer geöffnet werden. Aufgrund einiger Einschränkungen, die diese Methode mit sich bringt (keine Event-Handler möglich) und die Verfügbarkeit einer moderneren Technologie (Silverlight) um browserfähige Applikationen zu entwickeln, sei diese Variante hier nur Vollständigkeit halber erwähnt.

Die XAML Syntax
Wird in der Markup-Sprache ein Element deklariert, z.B. `<Button Content="Suche"/>`, so entspricht dies nicht nur einer logischen Repräsentation des grafischen Elements, sondern kommt vielmehr einer Instantiierung des korrespondierenden WPF- Objekts gleich. Das heißt dass zur Laufzeit der entsprechende Standardkonstruktor des UIElements aufgerufen wird. Wie zu erkennen ist wird ein Objektelement in XAML innerhalb spitzer Klammern (< , >) deklariert. Um Objektelemente mit Eigenschaften zu versehen existieren in XAML verschiedene Möglichkeiten [40]

Attributsyntax: Die Eigenschaft (oder Attribut) eines Objektelements kann dabei mit dem Zuweisungsoperator(=) und einer Zeichenkette in Anführungszeichen festgelegt werden. So deklariert beispielsweise der Ausdruck:
`<Canvas Visibility="Visible"/>` ein sichtbares Canvas-Element.

Eigenschaftenelementsyntax: Nicht alle Eigenschaften eines Objektelements lassen sich über die Attributsytax festlegen. In manchen Fällen kann ein Attribut aus mehreren Typen bestehen. In einem solchen Fall muss über die Deklaration eines zusätzlichen XAML-Knotens die Eigenschaft beschrieben werden. Die Syntax für den StartTag des Eigenschaftselement lautet: *<Typname.Eigenschaftsname>*. Der Inhalt ist ein Objektelement vom Typ den das Attribut erwartet. Der Tag muss mit *</Typname.Eigenschaftsname>* wieder geschlossen werden.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="30"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
 ...
</Grid>
```

Quellcode 2.4: *XAML Eigenschaftenelementsyntax*

Syntax zum setzten der Inhaltseigenschaft: Klassen haben die Möglichkeit exakt eines ihrer Attribute als Inhaltseigenschaft (*ContentPropertyAttribut*) auszuweisen.

```
[ContentProperty("Name")]
public class Person
{
   public Person()
    {
    }
```

```
7
8     public string Name
9     {
10        get { return _name; }
11        set { _name = value; }
12    }
13
14    private string _name;
15 }
```

Quellcode 2.5: *Zeigt wie eine Klasse (Person) eine ihrer Eigenschaften (Name) als Inhaltseigenschaft-Attribut ausweisen kann.*

Diese Eigenschaft muss dann nicht explizit gesetzt werden, sondern wird durch das Eigenschaftselement gesetzt, welches zwischen dem öffnenden und und schließenden Tag deklariert wurde.

```
1 <Border>
2   <Label Content="Name:"/>
3 </Border>
4 <!-- explizites setzten vom Child-Attribut -->
5 <Border>
6   <Border.Child>
7     <TextBox Content="Name"/>
8   </Border.Child>
9 </Border>
```

Quellcode 2.6: *Zuweisung des ContentProperties durch die Inhaltssyntax und explizites setzten des Attributs.*

Das Element `Border` hat als seine Inhaltseigenschaft das Attribut `Child` festgelegt. Im ersten Fall wird nun das Objektelement `Label` dieser Inhaltseigenschaft automatisch zugewiesen. Der äquivalente zweite Fall weist das Element `TextBox`, direkt dem Inhaltsattribut (`Child`) zu. Ist eine Inhaltseigenschaft vom Type *ICollection*, entspricht also einer Auflistung, so können dieser Eigenschaft durch das Inhalts-Model von XAML mehrere Objektelemente zugewiesen werden.

Namensräume in XAML

Wird in XAML ein Objektelement deklariert, zb. `<Button/>` so weis der XAML-Prozessor zunächst nicht, welche Instanz von Button referenziert werden sollte. Zum Beispiel gibt es auch Klassendefinitionen von Button die von Drittanbietern stammen. Es muss dem Parser also mitgeteilt werden aus welchem Namensbereich referenziert werden soll. Dazu dienen in XAML die Attribute `xmlns` und `xmlns:x`.

```
1 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
2 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Quellcode 2.7: *XAML - Deklaration der Namensräume*

Die erste Deklaration adressiert den WPF- Namensraum und enthält die Verweise auf alle WPF Klassen und Steuerelemente. Die zweite Anweisungen deklariert den Namensraum oder eigentlich eine Sammlung von Namensräume für Elemente die von der Darstellung unabhängig sind. Dies wären Elemente wie Arrays, statische und dynamische Ressourcen und Bindings. Dieser Namensraum ist auch für das Zusammenspiel von Code-Behind Dateien mit XAML Deklarationen über partielle Klassen verantwortlich und wird per Konvention dem Präfix x: zugeordnet.

Häufig verwendete Konstrukte mit x:-Präfix [40]

x:Key Stellt einen eindeutigen Schlüssel für Ressourcen bereit. Ressourcen werden in Wörterbüchern *ResourceDictionary* abgelegt und erlauben durch `x:Key` eine eindeutige Identifizierung.

x:Class Um die XAML Deklarationen zusammen mit einem Hintergrund-Code (`Code-Behind`) verwenden zu können, muss über den Ausdruck `x:Class` der CLR-Klassenname angegeben werden. Dadurch ist der Hintergrund-Programmcode als partielle Klasse mit der XAML Deklaration verbunden.

x:Name Durch dieses Attribut lassen sich Laufzeitnamen für Objektelemente vergeben. Dadurch ist die Instanz nicht mehr anonym und kann vom Laufzeitcode angesprochen werden.

```
<Canvas x:Name="m_chartArea" Width="100" Height="125">
  ...
</Canvas>
```

Quellcode 2.8: *XAML - Deanonymisierung von Objektelementen*

Das heißt es wird möglich, im Code-Behind-File das `Canvas`-Objekt anzusprechen und seine Eigenschaften zu verändern.

```
private void SetChartSize(double height, double width)
{
  m_chartArea.Height = height;
  m_chartArea.Width = width;
}
```

Quellcode 2.9: *Das Canvas Element aus 2.9 kann im Programmcode angesprochen und verändert werden.*

x:Null Legt den *null*-Wert in XAML fest.

Will man im XAML-Code auf eigene Datentypen verweisen oder benötigt man die Einbindung von `Assemblys` die nicht in WPF (`PresentationCore`, `Presentation-Framwork` oder `WindowsBase`) enthalten sind, müssen diese ebenfalls durch das `xmlns`-Attribut eingebunden werden und können mit einem eigenen Präfix versehen werden.

```
1  <UserControl
2     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4     xmlns:local="clr-namespace:Step"
5     xmlns:burst="clr-namespace:BurstViewControls;assembly=BurstControlLibrary"
6     x:Class="Step.MyUserControl"
7     >
8   <Canvas Name="m_chartArea">
9     <local:BurstChart Name="m_burstChart" />
10    <burst:BurstView Name="m_burstView" Background="Green" />
11    ....
12  </Canvas>
13 </UserControl>
```

Quellcode 2.10: *Zeigt die Vorgehensweise zum Einbinden von benutzerdefinierten Datentypen*

In Zeile 4 von Quellcode: 2.2.2 wird der lokale Namensraum definiert und dem Präfix `local` zugewiesen. In Zeile 5 wird dem Präfix `burst` die Assembly `BurstControl-Library` zugewiesen. Diese Assembly enthält im Namensraum `BurstViewControls` den Type BurstView, der in Zeile 10 deklariert wird. Zeile 9 zeigt die Deklaration einer Instanz der Klasse `BurstChart` aus dem lokalen Namespace `Step`.

Spracherweiterungen – Markup Extensions

Ein XAML-Parser übernimmt den Wert einer Eigenschaft als Zeichenfolge und konvertiert diese in eine Primitive oder übersetzt den Wert mittels `TypConverter`. Dies ist für die meisten Attributzuweisungen völlig ausreichend. In einigen Fällen ist es aber wünschenswert, Eigenschaftswerte entsprechend eines schon existierenden Objekts zu setzten oder mittels Datenbindung den Attributwert dynamisch zu verändern. Für diese Szenarien stehen die Spracherweiterungen oder *Markup Extensions* zur Verfügung. Eine *Markup Extension* ist eine in geschwungenen Klammern ({ und }) angegebene Attributzuweisung und verwendet folgende Syntax: `{ Erweiterungsklasse Argument }`. Alle Markup-Erweiterungen werden durch Klassen implementiert die von System.Window.Markup.MarkupExtension abgeleitet sind.

XAML definierte Markupextension

x:Static In diesem Fall ist die Markup-Erweiterung die `StaticExtension`-Klasse. Es ist Konvention, das Wort Extension, bei Angabe von Erweiterungsklassen wegzulassen.

```
1  <UserControl>
2    <Rectangle Fill="{x:Static SystemColors.ActiveCaptionBrush}"
         Stroke="Black" Width="100"/>
3  </UserControl>
```

Quellcode 2.11: *Verwendung der XAML Markuperweiterung mit der StaticExtension-Klasse*

Stößt der XAML-Parser nun auf die Anweisung in geschwungenen Klammern (Zeile 2) so wird eine Instanz der Klasse `StaticExtension` erzeugt, und dem Konstruktor die Zeichenfolge "SystemColors.ActiveCaptionBrush" als Argument übergeben. Durch den Aufruf der Methode *ProvideValue()* (definiert in der Basisklasse) wird der Wert der statischen Eigenschaft SystemColors.ActiveCaptionBrush zurückgeben und somit die `Fill`-Eigenschaft von `Rectangle` gesetzt.

x:Type Damit kann ein Verweis auf einen Type festgelegt werden. Der Typ wird dabei mit als Typ-Namen angegeben. Dieses Sprachkonstrukt kommt vorwiegend bei der Definition von Styles in Einsatz. Dort wird etwa die Eigenschaft `Style.TargetType` mittels x:Type angegeben, um festzulegen für welchen Type der Style Verwendung findet.

WPF-spezifische Markuperweiterungen

In WPF werden Markup-Erweiterungen vor allem für die Bereitstellung von Ressourcen (`StaticResource`, `DynamicResource`) und zur DatenBindung (`Binding`) unterstützt [41].

StaticResource Stellt einen Wert für eine Eigenschaft zur Verfügung der nur einmal, nämlich beim Laden von XAML, referenziert wird. Dies bedeutet das eine nachträgliche Änderung der Ressource zur Laufzeit, nicht zu einer Änderung der Eigenschaft führt.

```
1 <UserControl>
2   <UserControl.Resources>
3     <SolidColorBrush x:Key="BgColor" Color="#FF4D4DCA"/>
4   </UserControl.Resources>
5   <Rectangle Fill="{StaticResource=BgColor}" Stroke="Black" Width
        ="100"/>
6 </UserControl>
```

Quellcode 2.12: *Verwendung der WPF Markuperweiterung für statische Resourcen*

Diese Erweiterung wird verwendet um eine Eigenschaft festzulegen deren Wert schon vorher definiert wurde. Dies bringt die Möglichkeit mit sich, dass mehrere Elemente die selbe Ressource nutzten können, und bringt Vorteile für die Wartbarkeit.

DynamicResource Im Gegensatz zur `StaticResource`-Erweiterung stellt diese Extension einen Wert bereit der eine Laufzeitreferenz zu der angegebenen Resource hat. Der Wert wird bei jedem Aufruf neu referenziert und kann dadurch zur Laufzeit eine Änderung erfahren.

Binding Liefert einen datengebundenen Wert für eine Eigenschaft. Als Quelle für die Datenbindung wird der Datenkontext des übergeordneten Objekts verwendet. Mehr zum Thema Datenbindung im Abschnitt 2.2.2 auf Seite 32.

Code-Behind

Unter dem Ausdruck `Code-Behind` wird Programmcode verstanden, der beim Kompilieren von XAML, mit den im Markup-Code definierten Objekten, verknüpft wird. Er dient dazu die deklarativ definierte Benutzeroberfläche mit Funktionalität zu erweitern. Während also im Markup-Code die Struktur der grafischen Elemente festgelegt wird, ist auf der Code-Behind Seite das Verhalten der GUI implementiert. Dort wird auf Benutzerereignisse reagiert und es können zur Laufzeit dynamisch grafische Elemente nachgeladen werden.
Technisch gesehen erfolgt die Verknüpfung von `Markup` und `Code-Behind` durch partielle Klassen.

```
1  <!-- File: BurstView.xaml -->
2  <UserControl x:Class="StepApplication.BurstView">
3    <Button x:Name="SearchButton" Content="Suche" Click="OnSearchClick"
       />
4  </UserControl>
```

```
1  // File: BurstView.xaml.cs
2  namespace StepApplication
3  {
4    public partial class BurstView : UserControl
5    {
6      public BurstView()
7      {
8        InitializeComponent();
9      }
10     private void OnSearchClick(object sender, RoutedEventArgs e)
11     {
12       // do searching ...
13     }
14   }
15 }
```

Quellcode 2.13: *Das Zusammenspiel von Markup und Code-Behind durch partielle Klassen*

Durch die Angabe der `x:Class`-Direktive wird dem XAML-Parser mitgeteilt, dass die partielle Klasse `BurstView` mit dem Markup-Code asoziiert werden soll. Moderne Entwicklungsumgebungen wie etwa das VisualStudio von Microsoft legen schon automatisch das Programmgerüst der partiellen Klasse beim Erzeugen einer XAML-Datei an. Will man dies aber per Hand machen muss darauf geachtet werden, dass die partielle Klasse vom Typ erbt, der in der Markupdefinition das Stammelement bildet. Im Beispiel 2.13 wäre dies `UserControl`.
Einen zentralen Aspekt im Zusammenwirken von Markup und Code-Behind bilden Ereignisse die sogenannten *Events*. Anzeigeelemente können Ereignisse absetzten die durch geeignete Methoden, die als *EventHandler* bezeichnet werden, behandelt werden können. *EventHandler* werden dabei als Instanzmethoden in der partiellen Code-Behind-Klasse implementiert. (Zeile 10 im Quellcode 2.13)

Im Zusammenhang mit Code-Behind muss auch erwähnt werden das Code für die Programmlogik in XAML auch "inline" deklariert werden könnte. Dafür existiert die Direktive `x:Code` und wird in der Syntax der Form `<x:Code><![CDATA[ ... ] ]> </x:Code>` angegeben. Allerdings erschließt sich mir der Sinn von Inline-Code in XAML nicht. Denn ich sehe einen klaren Vorteil in der sauberen Trennung von Aussehen der Oberfläche und dahinter liegendem Programmcode. Genau dieser Vorteil wird aber durch das Vermischen von Programmcode und XAML-Deklaration zunichte gemacht. Ich würde daher vermuten, dass die Möglichkeit Inline-Code im deklarativen Teil einzubetten, eher historische Gründe haben mag. Ich habe jedenfalls während der gesamten Entwicklung der Software auf eine strikte Trennung von Design und Logik geachtet.

Angehängte Eigenschaften

Das Konzept der *AttachedProperties* also der angehängten Eigenschaften wird in WPF dazu verwendet um Eigenschaften elementübergreifend zugänglich zu machen. Ein *AttachedProperty* stellt dabei eine Eigenschaft dar, die von einem anderen Typ definiert wird, als von dem der sie verwendet. Die Syntax zur Angabe von angehängten Eigenschaften ist immer folgender Form: `DefinierenderTyp.Eigenschaftsname` Um diese Methodik zu verdeutlichen kann folgendes Beispiel herangezogen werden:

```
1 <DockPanel>
2   <TextBox DockPanel.Dock="Top" Width="120" Height ="20" .../>
3 </DockPanel>
```

Quellcode 2.14: *Das TextBox Element kann durch Verwendung des AttachedProperty (DockPanel.Dock) innerhalb eines DockPanels korrekt positioniert werden.*

Jedes Steuerelement hat interne Eigenschaften, im Fall eines Textfeldes wären dies etwa Farbe, Größe und Schriftart, etc ... Positioniert man ein Textfeld in einem Container (`Camvas`, `DockPanel`, etc.) so werden zusätzliche Eigenschaften zugänglich. Wird das Textfeld beispielsweise einem `DockPanel` hinzugefügt, kann das Element auf die Eigenschaft `Dock` verweisen, die in der Klasse `DockPanel` deklariert ist und so innerhalb des `DockPanels` korrekt positioniert werden. Die Eigenschaft DockPanel.Dock würde sich vom Textfeld auch aufrufen lassen wenn dies nicht in einem `DockPanel` beinhaltet wäre, allerdings würde dies keinerlei Wirkung erzielen. Insbesondere zeigt dieser Fall, dass das Element welches eine angehängte Eigenschaft nutzt, kein Wissen über das Vorhandensein des Objekts benötigt welches die Eigenschaft deklariert.

Datenbindung

Eine grafische Benutzerschnittstelle hat die Aufgabe Daten zu visualisieren. Damit die Visualisierung mit den zugrundeliegenden Daten synchron läuft, steht in WPF der Mechanismus der Datenbindung zur Verfügung. Unter Datenbindung kann also der Prozess verstanden werden, der die Daten der Business-Logik mit der Visualisierung verknüpft. Ist ein grafisches Element also an Daten der Geschäftslogik gebunden, und erfahren diese

eine Veränderung, so wird die Datenbindung dafür sorgen, dass das grafische Element die Änderung wiederspiegelt.

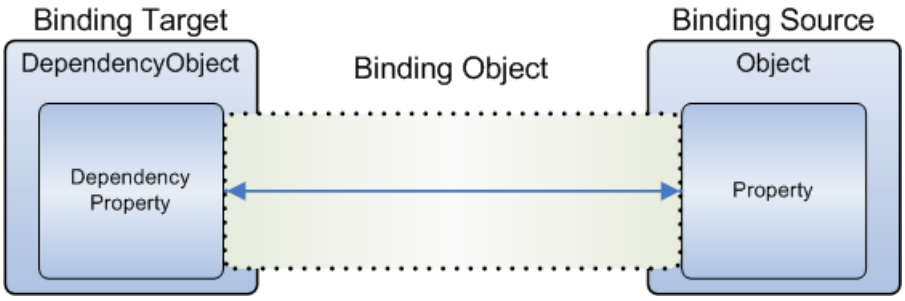


Abbildung 2.16: *Prinzipielles Konzept der Datenbindung (Quelle: MSDN (2011) [42])*

Ganz allgemein wird von Datenbindung gesprochen wenn Daten eins `Quellobjekts` verwendet werden, um damit (eine) Eigenschaft(en) in einem Zielobjekt automatisch zu verändern. Während die Zieleigenschaft immer ein `DependencyProperty` sein muss, ist es nicht zwingen erforderlich, dass die Datenbindung zwischen Business-Objekten und UIElementen stattfindet. Wenngleich diese Konstellation einen weiteren wichtigen Baustein, zu einer sauberen Architektur, mit Trennung von Daten und deren Darstellung bildet.
Die Kopplung von Quell- und Zielobjekt bei der Datenbindung erfolgt über eine Instanz der Klasse `Binding`. Diesem Objekt wird neben Quell- und Zielobjekt auch der Bindungspfad übergeben. Der Bindungspfad gibt an, an welche Eigenschaft(Property) des Quellobjekts gebunden werden soll. Zusätzlich kann auch die Richtung des Datenflusses mit Hilfe des `BindingObjects` gesteuert werden.

Richtung des Datenflusses
Der Nutzen des Datenflusses von der Datenquelle zum Bindungsziel ist offensichtlich. Ändern sich die Daten, so soll auch deren Anzeige aktualisiert werden. In manchen Fällen ist es aber auch notwendig den Datenfluss der Bindung umzukehren. Bei Benutzereingaben beispielsweise, ist es gewünscht, die Daten bei Änderung der Eingabe auf aktuellen Stand zu haben. Dafür gibt es beim Mechanismus der Datenbindung die Möglichkeit die Richtung des Datenflusses auf vier Arten zu steuern.

- `OneWay`: Reflektiert Änderungen im Quellobjekt automatisch im Zielobjekt. Allerdings werden Änderungen im Zielobjekt nicht im Quellobjekt wiedergespiegelt. Diese Bindungsart wird dann verwendet, wenn etwa das Zielobjekt keine Möglichkeiten bietet sein Werte zu verändern und es daher nicht notwendig ist die Werte des Zielobjekts zu überwachen.

- `OneWayToSource`: Dies ist die Implementierung des `OneWayBindings` in die Gegenrichtung. Dient vor allem bei der Übernahme von Eingabewerten aus dem UI.

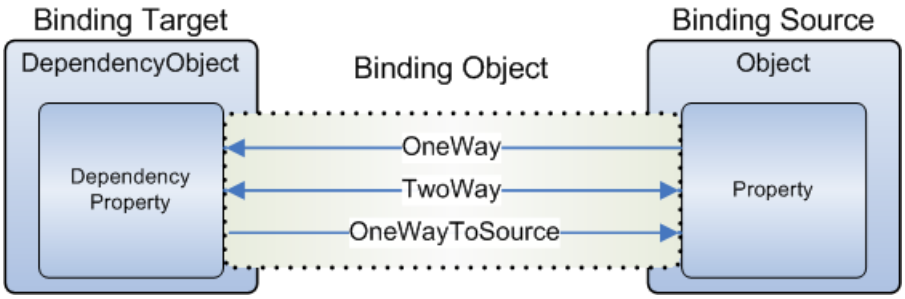


Abbildung 2.17: *Illustration der Datenflussrichtungen (Quelle: MSDN (2011) [42])*

- `TwoWay`: Bei diesem Bindungsmodus führen Änderungen im Quell- oder Zielobjekt zur Aktualisierung der jeweiligen Gegenseite. Durch diese Möglichkeit werden vollständig, interaktive Anwendungen umgesetzt.

- `OneTime`: Eine weitere, nicht illustrierte Möglichkeit ist das "einmalige" Binden an eine Datenquelle. Dabei erfolgt die Aktualisierung des Zielobjekts nur einmal und zwar zum Zeitpunkt der Initialisierung der Datenbindung.

Natürlich könnte immer der Zweiwege-Modus verwendet werden. Es ist aber zu beachten, dass eine Datenbindung in nur eine Richtung wesentlich performanter arbeitet, da nur die Änderungen an einer Seite beobachtet werden müssen.

Datenbindungen werden über die in Abschnitt 2.2.2 (Seite 29) beschriebene `Binding-Extension` angegeben.

```
1 <UserControl>
2   <UserControl.DataContext ="{StaticResource Burst}" />
3       ...
4   <TextBox Text="{Binding Repeats}" .../>
5 </UserControl>
```

Quellcode 2.15: *Deklaration einer Datenbindung in XAML durch Verwendung der Binding-MarkupExtension*

Eine Datenbindung, wenn nicht anders angegeben, bezieht sich immer auf den Datenkontext. Der Datenkontext (`DataContext`) ist eine Abhängigkeitseigenschaften (`DependencyProperty`) und zeigt auf das Datenobjekt an welches gebunden werden soll. Der Datenkontext wird vererbt. Wie im Quellcodebeispiel 2.16 wird dem Stammelement `UserC ontrol` ein `DataContext` ein Datenobjekt vom Typ `Burst` zugewiesen. Das Untergeordnete Element `TextBox` erbt diesen Datenkontext und kann dadurch auf eine Eigenschaft (`Repeats`) des Datenkontextes binden.

Damit eine Überwachung von gebundenen Eigenschaften zu einer Aktualisierung im Zielobjekt führt, muss die Quelle einen geeigneten Benachrichtigungs-Mechanismus im-

plementieren. Dies geschieht durch das Einbinden der `INotifyPropertyChanged` - Schnittstelle. Die Schnittstelle verlangt das Implementieren des `PropertyChanged` – `Events` vom Typ `PropertyChangedEventHandler`, an welches sich die Datenbindung subskribiert.

```
1  public class Burst : INotifyPropertyChanged
2  {
3    ...
4    public int Repeats
5    {
6      get { return m_repeats; }
7      set {
8        m_repeats = value;
9        OnPropertyChanged("Repeats");
10       }
11   }
12   ...
13   public event PropertyChangedEventHandler PropertyChanged;
14   private void OnPropertyChanged(String info)
15   {
16     if (PropertyChanged != null)
17     {
18       PropertyChanged(this, new PropertyChangedEventArgs(info));
19     }
20   }
21 }
```

Quellcode 2.16: *Zeigt die Implementierung des Benachrichtigungs-Mechanismus bei Datenbindung durch die INotifyPropertyChanged Schnittstelle*

KAPITEL 3

Projektbeschreibung

Wie schon in der Einführung unter Abschnitt 2.1.1 (Seite 12) beschrieben, kommt es altersbedingt sowohl zu einer Abnahme der Muskelkraft als auch zur Reduktion der Muskelmasse. Die Erkenntnisse aus der Behandlung von Paraplegikern durch funktionelle Elektrostimulation, soll im Rahmen des internationalen Projekts *MOBIL*[1] (Mobilität im Alter) auf den Bereich der Sarkopenie ausgedehnt werden.
Dazu sollte ein tragbarer, transkutaner elektronischer Stimulator, zur Anwendung der funktionellen Elektrostimulation bei geriatrischen Patienten entwickelt werden. Durch eine geeignete Anbindung an einen Computer wird eine flexible Einstellung von Stimulationsparametern ermöglicht. Dabei spielt die im Rahmen dieser Diplomarbeit entwickelte Software die entscheidende Rolle. Die Applikation wird auch dafür eingesetzt die Compliance-Daten, also die Daten über den Gebrauch und die Verwendung des Stimulators, aus dem Gerät auszulesen, zu visualisieren und in geeigneter Form zu archivieren.

3.1 Anforderungen

Zielsetzung ist es, das geriatrische Patienten in Eigeninitiative, Trainings mit funktioneller Elektrostimulation durchführen. Um Aussagen über den Erfolg der Trainingseinheiten treffen zu können ist es notwendig sowohl die Art als auch die Dauer der Anwendung zu protokollieren. Im Folgenden sind die wesentlichen Anforderungen an Hard- und Software für eine erfolgreiche Umsetzung der Zielsetzungen angeführt.

Anwendbarkeit Da der Stimulator von geriatrischen Patienten vorwiegend in Eigeninitiative angewendet werden soll, muss das Gerät leicht bedienbar sein und daher auf ein Minimum an Benutzerinteraktion beschränkt sein.

Sicherheit Der Stimulator muss gewährleisten, dass auch im Falle einer fehlerhaften oder falschen Anwendung keine körperlichen Schäden auftreten dürfen.

[1] Projekthomepage: http://www.physmed-vienna.at/index.php?index=3&url=mobil-at

Erhebung der Compliance Der Stimulator muss in der Lage sein, Daten zu speichern, die Aufschluss über die Art und Dauer seiner Verwendung, während der Trainingseinheiten geben.

Messdatenerfassung Zusätzlich zur Kontrolle der Anwendung sollen Messwerte zur Bestimmung der Elektrodenimpedanz (Strom und Spannung), des evozierten myoelektrischen Signals (EMG bzw. M-Wave) sowie die Daten eines Beschleunigungssensors erfasst werden[2].

Schnittstelle zur Software Die Hardware muss eine geeignete Anbindung(USB) an den Rechner zur Verfügung stellen.

Einstellung der Stimulationsparameter Durch die Software wird eine flexible Adjustierung sämtlicher Stimulationsparameter ermöglicht. Dabei kann die Veränderung der Parameter wahlweise durch graphische-interaktive oder textuelle Eingaben erfolgen. Die Einstellungen werden dann auf das Stimulationsgerät geladen, damit dieses autark damit arbeiten kann.

Universelle Veränderung der Stimulationsmuster Die Zusammenstellung verschiedener Stimulationsmuster soll durch die Softwareanwendung gewährleistet sein. Dazu gehören das Speichern und Laden von vorhandenen Stimulationsmustern genauso wie die interaktive Arrangierung via “Drag and Drop”

Visualisierung Alle aufgezeichneten Daten (Compliance- und Messdaten) müssen durch die Software visualisiert werden. Dabei muss nach Aspekten des User-Interface-Designs eine Übersichtlichkeit und sinnvolle Zuordnung der grafischen Ausgabe erfolgen. Insbesondere ist die Schlüsselanforderung, nämlich die Darstellung der Compliance, von hoher Priorität und geeignet umzusetzen.

Patientenverwaltung Um den Gebrauch der Softwareapplikation nicht nur auf dieses Projekt zu beschränken, sollte eine möglichst flexible Patientenverwaltung implementiert werden.

Datenexport Für weitere Analysen der Messwerte oder der Compliance-Daten muss die Software über geeignete Exportmechanismen und - Formate verfügen.

3.2 Übersicht

Der Stimulator wird mit Akkumulatoren betrieben damit die Portabilität gewährleistet ist. Prinzipiell ist das Stimulationsgerät autonom einsatzfähig. Das heißt es kann ohne jegliche Verbindung zum Rechner und damit zur Software operieren. Die Datenspeicherung im “Stand-Alone” -Betrieb erfolgt auf eine SD (Secure Digital) - Speicherkarte. Der Stimulator arbeitet microkontroller-gesteuert und gibt die Stimulationsimpulse über

[2]Diese Messdaten werden unter anderem dafür eingesetzt um das Sicherheitsprotokoll zu erfüllen

Oberflächenelektroden ab.
Die Messwerterfassung zeichnet sowohl Spannung als auch Strom auf, um einen Rückschluss über die Elektrodenimpedanz zu ermöglichen. Außerdem wird das evoziierte myoelektrische Signal erfasst um diagnostisch, therapeutische Aussagen zu begründen. Auch werden durch einen Akzelerometer die Muskelzuckungen protokolliert um gegebenenfalls ein falsches Anlegen der Elektroden zu überprüfen.

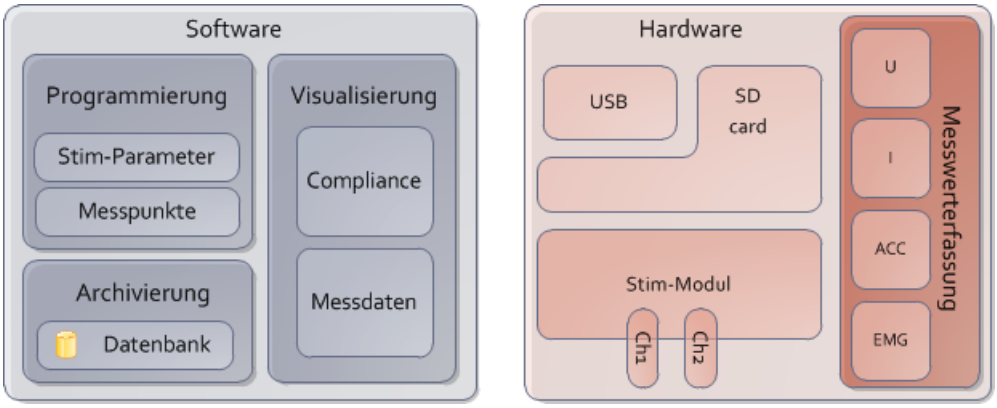


Abbildung 3.1: *Zeigt eine Übersicht der wesentlichen Elemente von Hard- und Software.*

Die Software kümmert sich grob gesagt, um die flexible Einsatzmöglichkeit der Stimulations-Hardware. Beim Verbinden des Stimulators mit einem Rechner auf welchem die Software installiert ist, wird es möglich, unterschiedlichste Stimulationsmuster auf den Stimulator zu laden, und die Messergebnisse vom selben zu akquirieren. Die Erkennung eines verbundenen Stimulationsgerätes erfolgt dabei automatisiert.
Da der Stimulator nahezu keine Bedienelemente besitzt wird die Programmierung und Parametereinstellung gänzlich durch die Software durchgeführt. Weiters kümmert sich die Applikation auch um die Visualisierung sowohl der erfassten Messwerte als auch der Daten zur Anwendungskontrolle. Die Software speichert die Daten von mehreren Stimulatoren und Trainingseinheiten und bietet sowohl eine Synchronisation von verteilten Datensätzen als auch entsprechende Exportfunktionen für weitere Analysen und den Austauch von Stimulationsprotokollen.
Die Software muss für alle neueren Windows-Betriebssysteme ab WindowsXp lauffähig sein. Andere oder ältere Betriebssysteme aus der Windows- Familie müssen nicht unterstützt werden. Die Software sollte ein Erweiterungskonzept implementieren, durch welches es ermöglicht wird auch andere Schnittstellen als USB recht einfach zu integrieren.

3.3 Ablauf und Zusammenwirken von Soft- und Hardware

Um einen Eindruck der Funktionalität und Umfangs des Projektsbereiches aus Softwareanwendung und Stimulationsgerät zu erhalten, sei im Folgenden ein schematischer Ablauf illustriert. Dieser zeigt die wesentlichen Funktionsblöcke der Soft- und Hardware. Der Datenaustausch zwischen Software und Stimulator erfolgt durch Dateien die über eine USB-Verbindung transferiert werden. Initiiert wird ein Datentransfer dabei immer von der Software. Der Stimulator stellt die Daten, verpackt in Dateien lediglich über seine SD-Speicherkarte als Massenspeicher zur Verfügung.

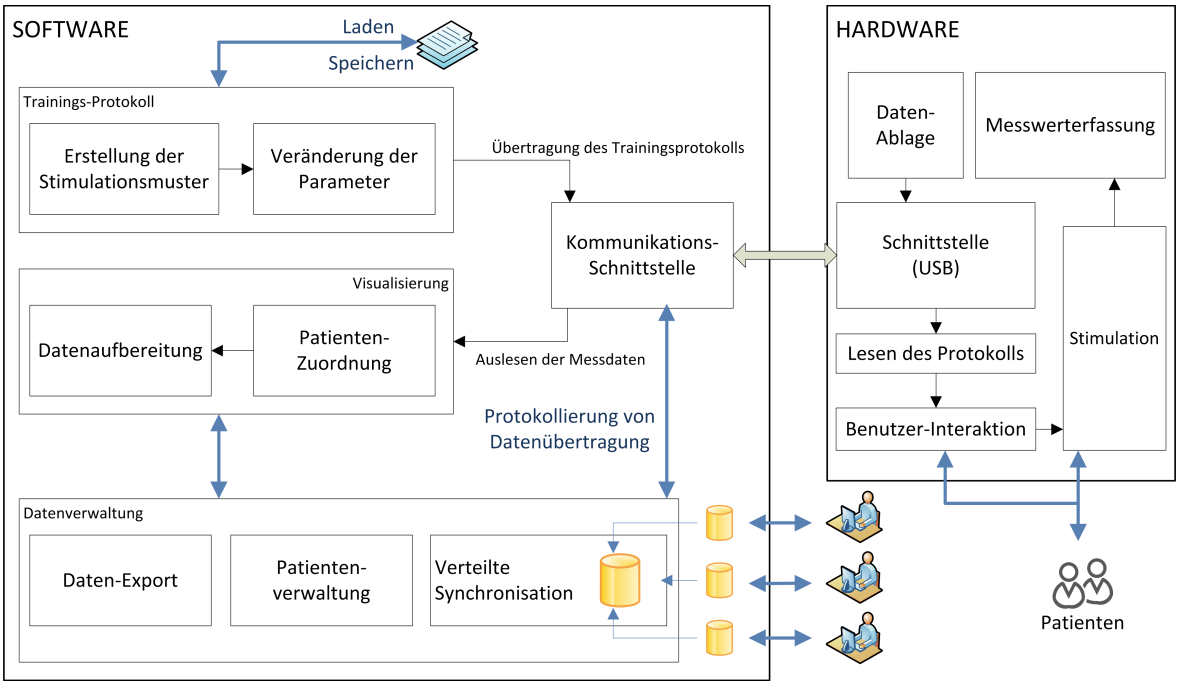


Abbildung 3.2: *Aufgaben und Prozesse der Soft- und Hardware des Stimulators.*

3.4 Prozessbeschreibung und Ablauf am Stimulator

Am Stimulator werden die Informationen in Dateien auf der SD-Karte hinterlegt. Beim Starten einer Trainingseinheit werden die Dateien mit den Stimulationsparametern nach und nach eingelesen. Aus den darin gespeicherten Daten lässt sich auf Grund der Definition eines Stimulationsmusters (siehe Abbschnitt 4.5.2) eine Impulsfolge generieren. Aufgrund der Informationen der Messwerterfassung entscheidet der Stimulator ob für den gerade generierten Impulse Messungen aufgezeichnet werden sollen. Die Ergebnisse dieser Messungen werden wieder in Dateien auf der SD-Karte gespeichert.

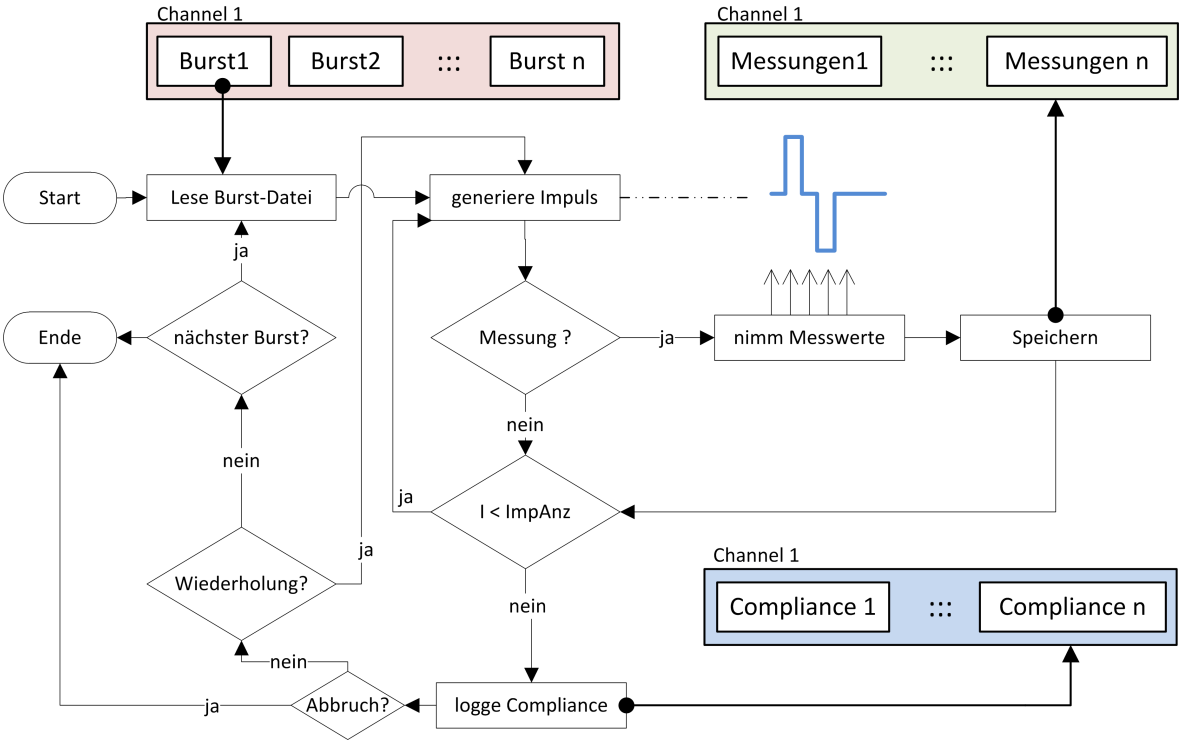


Abbildung 3.3: *Schematischer Ablauf einer Stimulation am Stimulator. Die farblich hinterlegten Felder zeigen die Zugriffe auf das Dateisystem und somit die Schnittstelle zur Software.*

Nach jedem Durchlaufen eines Stimulationszykluses wird dies vom Stimulator protokolliert und ebenfalls in eigene Files auf die SD-Karte geschrieben. Abbildung 3.3 zeigt den Ablauf eines Stimulationstrainings am Stimulator.

KAPITEL 4

Aufbau und Datenstruktur

In diesem Kapitel wird zunächst etwas näher auf die Daten eingegangen. Das heißt es wird die Natur der zu behandelnden Daten der beiden großen Datenkategorien (Stimulations- u. Compliancedaten) erläutert. Bevor die konkreten Aspekte der Softwareentwicklung wie Klassendesign und detailliertere software- relevante Informationen bearbeitet werden, sollen in diesem Kapitel auch die Protokolle zum Datenaustausch zwischen Softwareanwendung und Stimulationsgerät beleuchtet werden.
Auch die Wertebereiche sämtlicher Daten, angefangen von Stimulationsparametern bis hin zu gemessenen EMG-Daten werden hier angegeben. Weiters wird die Strategie für die Ermöglichung eines autonomen Betriebs des Stimulators und die damit verbundenen Konfigurationsdaten näher beschrieben. Der Leser wird in diesem Kapitel also mit den verwendeten Daten und deren Wertebereiche bekannt gemacht und es werden Strategien gezeigt wie der Datenaustausch erfolgen kann.

4.1 Stimulations Parameter

Als Stimulationsdaten werden die Informationen bezeichnet, die dem Stimulator mitgeteilt werden, sodass dieser damit, autonom die gewünschten Stimulationsmuster erzeugen kann. In diesen Daten kodiert sind sowohl alle Stimulationsparameter, als auch Angaben über die Häufigkeit und den Zeitpunkt der zu akquirierenden Messproben.
Um Details der Stimulationsdaten zu verstehen, soll zunächst auf das vom Stimulator zu erzeugende Stimulationssignal, welches auf diesen Daten beruht, näher eingegangen werden. Das vom Stimulationsgerät erzeugte Signal ist eine Folge von Stimulationsimpulsen die in unserem Fall biphasische Rechteckimpulse[1] darstellen.

[1] Biphasische Rechteckimpulse gewähren Gleichstromfreiheit, verhindern damit Elektrolyse an den Elektroden und vermeiden so einen Materietransport von Elektroden-Material in den Körper. (Vgl. Abbschn. 2.1.1)

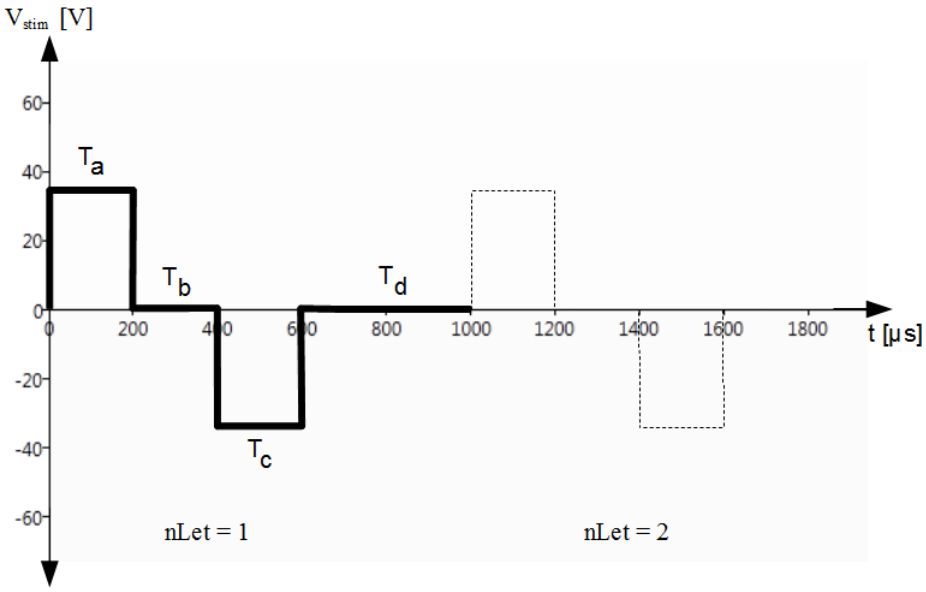


Abbildung 4.1: *Rechteckiger, biphasischer Stimulationsimpuls mit Ladungsausgleich.*

Ein Stimulationsimpuls (Abb. 4.1) kann durch sechs Größen beschrieben werden. Die Amplitude, gleichbedeutend mit der Intensität der Stimulation, den vier Zeiten T_a bis T_d und der Anzahl(nLet) der intermediären Impulse.
Die Zeiten geben die Pulsdauer für positiven (T_a) und negativen (T_c) Impulsteil an, sowie die Nullzeit (T_b) zwischen den alternierenden Impulsanteilen, als auch die Intermediärzeit (T_d) die als die Dauer zwischen zwei Einzelimpulsen aufgefasst werden kann.

Nlet ist ein ganzzahliger Wert und gibt an wie viele Intermediär-Impulse hintereinander folgen. Eine Folge von intermediären Impulsen wird aber immer noch als Einzelimpuls angesehen. Unsere Hardware unterstützt in der derzeitigen Ausbauphase zwar nur einen Zwischenimpuls, also Nlet=1, die Software ist aber schon jetzt dafür ausgelegt, spätere Hardwaremodelle mit mehreren Zwischenimpulsen zu unterstützen.

4.1.1 Stimulationselement

Durch die oben erwähnte Beschreibung des Stimulationsimpulses, ist es uns nun möglich einen Schritt weiter zu gehen und den Begriff des `Elements` einzuführen. Ein Stimulations-Element beschreibt im wesentlichen die Abfolge mehrerer Einzelimpulse. Neben den Grössen, die einen Impuls beschreiben, beinhaltet ein Element auch alle relevanten Stimulations-Parameter wie Frequenz, Amplitudenverlauf und Dauer der Einzelimpulsfolge.
Ein Element ist aus Sicht der Software das kleinste zur Verfügung stehende Teilstück. Alle Vorgänge die ein Stimulationselement betreffen sind daher atomarer Natur. Dies

Grösse	Min	Max	Typisch
Amplitude [V]	0	60	30
$T_a[\mu s]$	80	1000	500
$T_b[\mu s]$	0	2000	0
$T_c[\mu s]$	80	1000	500
$T_d[\mu s]$	0	9999	0
$Nlet$	1	9	1

Tabelle 4.1: *Wertebereich für typischen Stimulationsimpuls und seine beschreibenden Parameter*

bedeutet, dass ein Element als vollständig betrachtet werden muss. Es ist daher beispielsweise nicht möglich einzelne Stimulationsimpulse aus dem Element zu entfernen. Das Element kann nur als ganze (atomare)Einheit hinzugefügt oder entfernt werden.

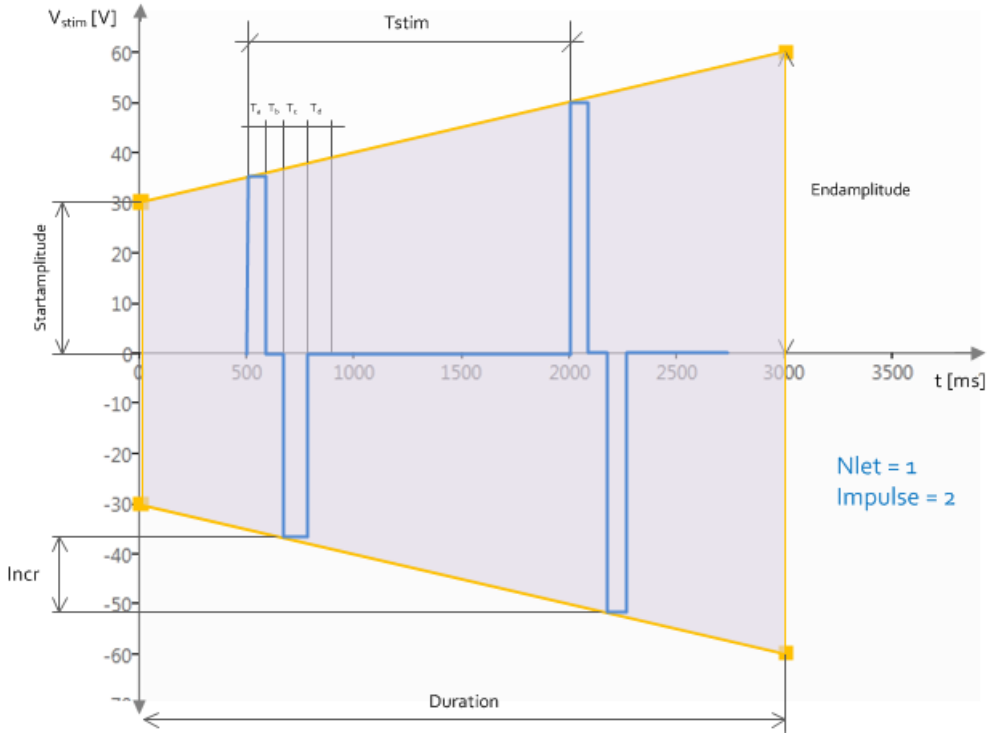


Abbildung 4.2: *Aufbau eines Burstelements: Biphasische Stimulationsimpulse (blau) und definierende Hüllkurve(gelb) mit Startamplitude und Inkrement. Der farbig markierte Bereich stellt ein Stimulationselement dar.*

Für eine kompakte Beschreibung eines Elements wurde wie in Abbildung 4.2 er-

sichtlich, die Hüllkurve des zu erzeugenden Signals gewählt. Die Hüllkurve lässt sich definieren durch die Startamplitude, Startzeitpunkt[2], Endamplitude und Endzeitpunkt, also die Dauer des Elements.

Um die programmiertechnische Interpretation des Signals in Hinblick auf die Verarbeitung mittels Mikrocontroller schon so effizient wie möglich zu machen, wird aber bei der Kodierung der Hüllkurve anstatt des Endpunktes (Endamplitude, Endzeitpunkt) ein Inkrement angegeben. Dies zeigt die Amplitudenänderung, ausgehend von der Startamplitude, pro Einzelimpuls an. Diese Darstellung erlaubt es dem Mikroprozessor, ohne großen Rechenaufwand, die Impulse sequentiell abzuarbeiten. Dafür wird neben dem Inkrement auch die Impulsanzahl bereitgestellt.
Die Impulsanzahl ergibt sich indirekt durch die Stimulationsfrequenz. Diese ist durch die Periodendauer der Stimulationsimpulse T_Stim gekennzeichnet. Wieder Aufgrund der atomaren Natur eines Elements hängen sowohl Impulsanzahl, Elementdauer und Frequenz direkt voneinander ab und beeinflussen sich gegenseitig. Die Impulsanzahl ist immer ganzzahlig und bewirkt daher bei Änderung der Frequenz, eine andere Elementdauer. Wie auch umgekehrt eine längere oder kürzere Elementdauer bei gleichbleibender Frequenz eine unterschiedliche Impulsanzahl bewirkt.

Grösse	Min	Max	Typisch
Start-Amplitude [V]	0	60	0
Increment [V/Imp]	-30	$+30$	1
Impulsanzahl	1	500000	12000
Frequenz [Hz]	0.5	500	20
$Dauer[sec]$	$1 \cdot 10^{-}5$	420	300
$Endamplitude[V]$	0	60	30

Tabelle 4.2: *Wertebereiche für Stimulationselement. Die kursiv angeführten Grössen sind indirekt abgeleitete Parameter die mit einem oder mehreren Werten zusammen hängen*

Wie wir später noch sehen, werden die Impulse innerhalb eines Elements, beginnend bei 1, durchnummeriert. Dieser Vorgang dient auch der Verarbeitung der Daten am Mikrocontroller. Dieser zählt nämlich alle Impulse die er generiert und kann durch die Addition des Inkrement und der Information der Impulsanzahl ohne Zwischenberechnung den Amplitudenverlauf generieren.
Auch bei der Angabe der Messpunkte kommt die Nummerierung der Impulse zu tragen. Dort wird über die Impulsnummer festgelegt, bei welchen Impulsen eine Messwertaufnahme zu erfolgen hat.

[2]Aufgrund der atomaren Beschaffenheit eines Elements, ist der Startzeitpunkt eines Elements immer 0s. Wenngleich dieser Punkt im Verlauf einer Stimulationsmuster-Zusammenstellung einen beliebigen Wert annehmen kann.

Wir wissen nun also das ein Stimulationssignal aus einer Folge von Einzelimpulsen besteht. Die Einzelimpulse ihrerseits können mehrere Zwischenimpulse (Nlets) enthalten. Ein Stimulationselements bildet eine Folge von Einzelimpulsen ab, wobei die Frequenz innerhalb eines Elements, als konstant anzusehen ist. Will man nun komplexe Stimulationsmuster erzeugen, so muss ein weiterer Begriff erläutert werden.

4.1.2 Stimulations Burst

Ein `Burst` wird in dieser Arbeit (fälschlicherweise[3]) als Komposition mehrerer Stimulationselemente bezeichnet. Ein Burst erlaubt also eine Abfolge von Stimulationsimpulsen mit variabler Frequenz. Da wie schon erwähnt, innerhalb eines Stimulationselements ja konstante Frequenz gilt, bietet die Zusammenfassung mehrerer Elemente zu einem Burst, eine variable Abfolge der Stimulationsfrequenz. Ein Burst ist somit als das kleinste Teil eines komplexeren Stimulationsmusters zu betrachten.

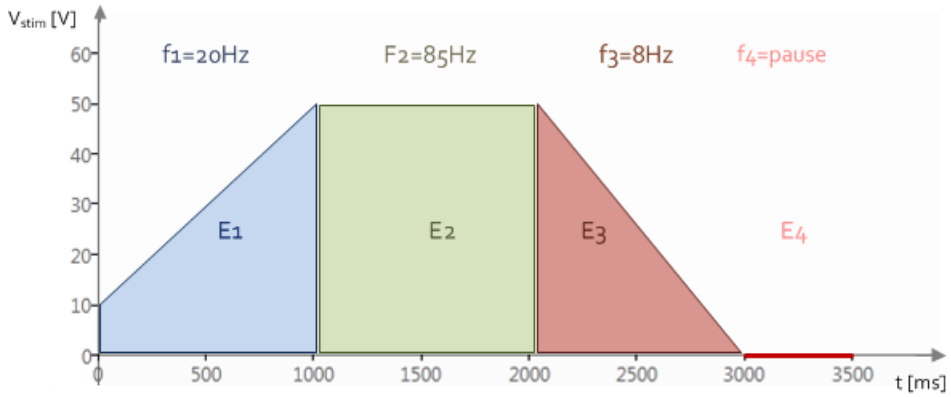


Abbildung 4.3: *Burst mit vier verschiedenen Elementen und unterschiedlichen Frequenzen.*

Da Stimulationsimpulse biphasisch und daher symmetrisch um die x-Achse sind, wird im Folgenden die Hüllkurve eines Elements nur mehr für den positiven Impulsteil angegeben (Vgl. Abb. 4.3). Auch die Modellierung im Programm, erfolgt über den positiven Anteil der Hüllkurve.
Hat ein Element eine Startamplitude von 0 Volt und ein Inkrement von 0 Volt/Impuls, beginnt und endet also ein Element mit 0 Volt, so wird dieses Element als Stimulationspause ausgewiesen (E_4 in Abb. 4.3). Für eine Stimulationspause gelten eigene Regeln als für "normale" Elemente. Eine Pause kann theoretisch beliebig lang sein und produziert keine Stimulationsimpulse. Darum besitzt eine Pause auch keine Frequenz und keine Impulsanzahl.

[3]Der Wortbedeutung nach sollte ein Burst eine Folge von Zwischenimpulsen beschreiben. In unserem Fall den Begriff des Einzelimpulses. Aufgrund der historischen Entwicklung des Softwareprojektes wurde aber zunächst auf eine vorhandene Codebasis zurückgegriffen, wo ebenso entwicklungsbedingt, der Begriff des Bursts für eine Folge von Einzelimpulsen im falschen Zusammenhang verwendet wurde.

Eine weitere Besonderheit des Bursts ist, dass dieser als kleinste Einheit eines komplexeren Stimulationsmusters, bei sequentieller Abarbeitung der Impulsfolgen, wiederholt werden kann. Es ist also möglich durch die Angabe einer Anzahl an Wiederholungen die Elemente innerhalb eines Bursts vermehrt hintereinander ausführen zu lassen. Dies führt direkt zur nächsten übergeordneten Struktur, dem Stimulationskanal.

4.1.3 Kanal

In Anlehnung an die Hardware, die die Stimulationsimpulse auf Kanälen an die Oberflächenelektroden ausgibt, wird auch der Begriff des Kanals in der Software verwendet, um die Ausgabe an Impulsfolgen für jeweils ein Elektroden-Paar zu beschreiben. Aus Sicht der Software beinhaltet ein Kanal keinen, einen oder mehrere Bursts. Enthält ein Kanal keinen Burst, so wird für diesen Kanal auch keine Information an das Gerät weitergegeben, und es kommt in weiterer Folge auch zu keinem generierten Stimulationsimpuls an den Elektrodenausgänen dieses Kanals.

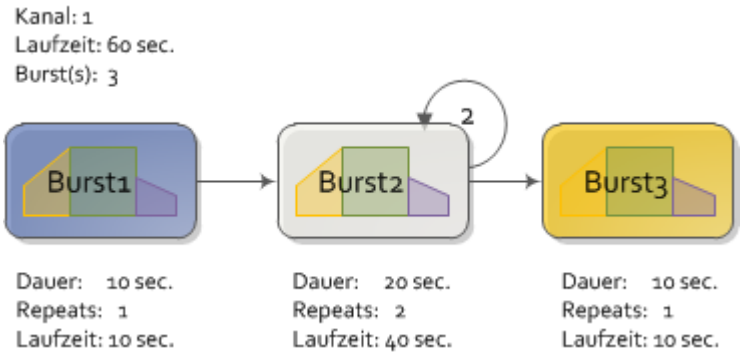


Abbildung 4.4: *Schematische Darstellung eines Stimulationskanals. Beinhaltet 3 Bursts von denen Burst2 2x wiederholt wird.*

Ein Burst eine in sich geschlossene Einheit. Daher ist die Reihenfolge der Bursts innerhalb des Kanals beliebig und kann variiert werden. Von der Bedienung her wird dies in der Software mittels Drag n' Drop umgesetzt. Ein Kanal wir immer von links nach rechts abgearbeitet. Wiederholungen von Bursts erfolgen immer direkt im Anschluss an den ersten Durchlauf des jeweiligen Bursts.
Neben der Nummer des Kanals, die direkt mit der hardwaremäßigen Kanalnummer assoziiert wird, ist auch die Gesamtlaufzeit der Impulsfolge eine Größe zur Beschreibung des Kanals. Die totale Dauer einer Stimulation setzt sich demnach zusammen aus der Summe aller Burst-Laufzeiten. Die Burst-Laufzeit ergibt sich dabei aus der Dauer des jeweiligen Bursts $\times$ seinen Wiederholungen.

$$LZ_{Ch} = \sum_{i=1}^{n} Bd_i * Bw_i$$

Auch die Information für welchen Muskel das zusammengestellte Stimulationsmuster angewendet werden soll, wird im Kanal kodiert. Diese Information ist allerdings nur innerhalb der Software relevant, da die Hardware kein Wissen über die Anatomie benötigt. Die Software benötigt diese Information aber sowohl für die Zuordnung der Messungen als auch für die Visualisierung. Daher wird bei der Übertragung der Kanalinformation auch der Zielmuskel übertragen, auch wenn die Information von der Hardware nicht verwendet wird.

4.2 Messdaten

Während einer Stimulation können pro Stimulationsimpuls vier verschiedene Signale aufgezeichnet werden. Strom, Spannung, EMG und das Signal eines Beschleunigungssensors. Der Zeitpunkt der Messwertaufnahme wird dabei durch die Software festgelegt. Dabei ermöglicht die Applikation für jedes Messsignal, die Abtastzeitpunkte separat zu wählen um dadurch die Datenmengen begrenzen zu können.
Neben den Daten der Messwerte, die durch das Stimulationsgerät erzeugt werden, gibt es also auch Daten die dem Stimulator mitteilen, welche Signale wann aufgezeichnet werden sollen. Wir unterscheiden also Daten die zum Mikrocontroller gehen und solche die vom Stimulator erzeugt, und von der Software ausgelesen werden. Diese "Retour"-Daten werden unten weiter gesondert behandelt. Zunächst zu den Daten welche die Messwertaufnahme beschreiben.

Daten zur Messwertaufnahme

Die Daten zur Messwertaufnahme werden für jeden `Burst` gespeichert. Jeder Burst besitzt also Informationen darüber, wann welche Signale während seiner Abarbeitung durch den Stimulator aufgezeichnet werden sollen. Die Messwertaufnahme gilt dabei auch für jede weitere Wiederholung des Burst.
Die Messung selber erfolgt jeweils für einen Stimulationsimpuls. Der Stimulator entscheidet auf Grund der Information die ihm durch die Software zur Verfügung gestellt wurde, ob und welche Signale, für den jeweils aktuell generierten Stimulationsimpuls, gemessen werden sollten. (Vgl. Abb. 3.3)
Die Angabe der Messpunkte erfolgt zeitbezogen. Das heißt es wird ein Zeitpunkt beziehungsweise eine Zeitspanne festgelegt, während dieser, alle auftretenden Impulse eine Messwertaufnahme erfahren.

Da der Stimulator nicht mit Zeitpunkten, sondern intern mit Impulsnummern arbeitet, werden die Zeitpunkte der Messwertaufnahme in ein Format übergeführt, indem die Zeiten durch Impulsnummer kodiert sind. Abbildung 4.6 zeigt die Nummerierung der Stimulationsimpulse innerhalb eines `Bursts`.

Um einen Stimulationsimpuls unabhängig vom Zeitpunkt seines Auftretens zu beschreiben, muss neben seiner Nummer auch stets die Nummer des Elements und die

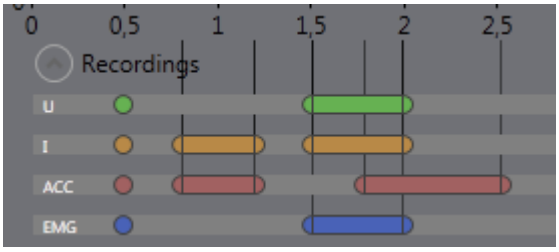


Abbildung 4.5: *Beispiel für die Festlegung der Messwertaufnahme durch die Software. Die vier unterschiedlichen Signale können getrennt voneinander aufgezeichnet werden.*

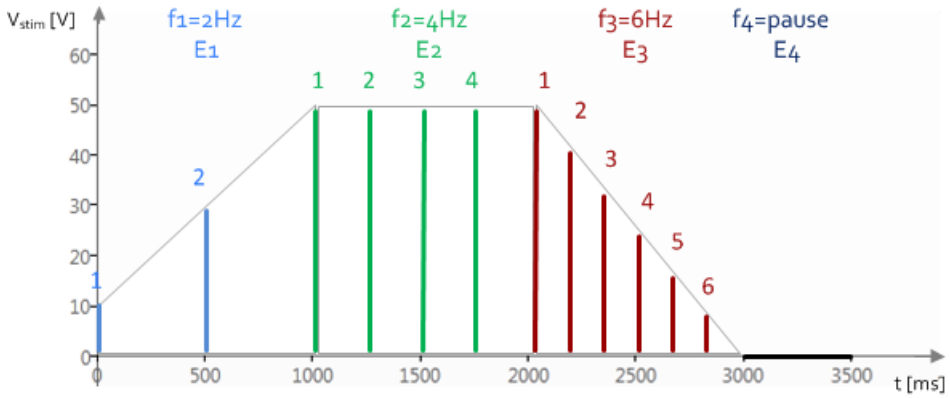


Abbildung 4.6: *Zeigt einen Burst mit vier Elementen (E_1 bis E_4) und eingezeichneten Impulsen. Die Nummerierung der Stimulationsimpulse wird in jedem Element neu begonnen.*

Nummer der aktuellen Wiederholung angegeben werden. Ein Stimulationsimpuls wird also programmintern wie folgt kodiert.

KanalNr	BurstNr	Wiederhohlung	ElementNr	ImpulsNr
1	1	1	2	3

Tabelle 4.3: *Programminterne Kodierung eines Stimulationsimpulses ohne die Angabe des Zeitpunktes. Diese Kodierung würde nach dem Beispiel in Abbildung 4.6 einem Zeitpunkt von* 1.5 *sec. entsprechen.*

Messsignaldaten

Sollten für einen Stimulationsimpuls Messung durchgeführt werden, so veranlasst der Mikrocontroller im Stimulator die Abtastung des jeweiligen Signals. Die Abtastrate für die jeweiligen Messungen können dabei ebenfalls über die Software festgelegt werden. Dazu werden dem Stimulationsgerät beim Übermitteln der Stimulationsdaten auch Konfigurationsinformationen übergeben, in denen die verschiedenen Abtastwerte der einzelnen

Signale hinterlegt sind. Nähere Informationen zu den Konfigurationsdaten unter Abschnitt 4.5.1.
Um einen Eindruck vom Signalverlauf und dessen Größenordnungen der einzelnen Messungen zu erhalten sind hier experimentelle Messwertaufzeichnungen protokolliert. Die Messwerte sind im Labor mit Hilfe eines Oszilloskops aufgenommen worden. Abbildung 4.7 zeigt den Strom- und Spannungsverlauf eines spannungsgesteuerten Stimulationsimpulses.

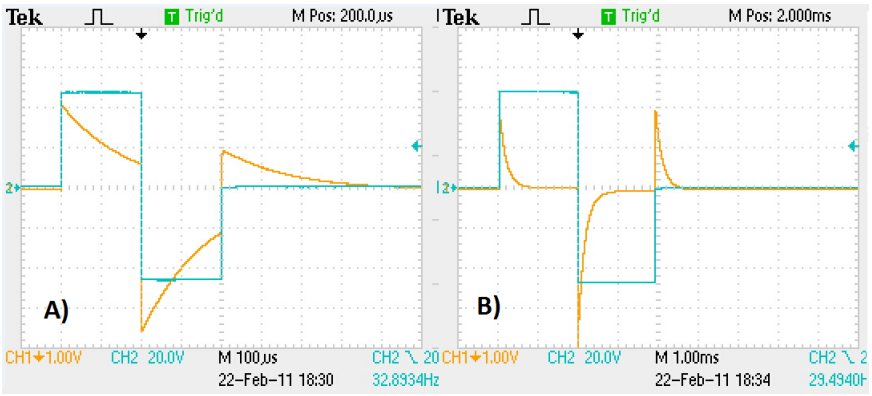


Abbildung 4.7: *Experimentelle Messungen: In A) u. B) ist jeweils der Spannungsverlauf (blau) und der Stormverlauf(gelb) zu sehen. In A) bei einer Impulsbreite von* $200\mu s$ *und in B) bei* $2ms$.

In Abbildung 4.7 sind die typischen Signalverläufe für Strom und Spannung ersichtlich. Auffallend ist die kapazitive Wirkung des Haut-Elektrodenüberganges, der speziell bei kurzen Impulsen Einflüsse hat. Abbildung 4.8 zeigt einen Verlauf eines evozierten Muskelpotenzials. Hier ist am Beginn der Messwertaufzeichnung der Artefakt des Stimulationsimpulses zu erkennen.

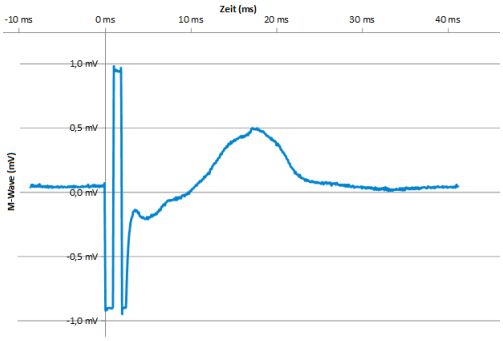


Abbildung 4.8: *Zeigt einen typischen Verlauf der M-Wave. In den ersten* $4ms$ *ist der Stimulationsartefakt zu erkennen.*

Alle Messwerte werden vom Stimulator durch seine Analog-Digital-Konverter (ADC) ermittelt. Um ausreichend viele Messpunkte aufzeichnen zu können, ist die Datendich-

te entsprechend hoch. Damit der Mikrokontroller nicht unnötig Rechenzeit dafür verschwendet, die ADC-Werte in die passenden physikalischen Größen umzurechnen, werden alle Messdaten als direkte ADC-Werte gespeichert und erst durch die Software am PC umgerechnet.
Damit hierbei eine möglichst große Kompatibilität zu nachfolgenden Versionen der Hardware gewährleistet wird, legt der Stimulator die für die korrekte Umrechnung benötigten Informationen ebenfalls in seinen Konfigurationsdaten ab. (Siehe Abschn. 4.5.1)

4.3 Compliance Daten

Die Daten zur Überwachung der Anwendung müssen so beschaffen sein, dass Rückschlüsse über das Trainingsverhalten des Patienten getroffen werden könnnen. Dies betrifft zum einen die Zeitdauer des Trainings. Es muss also nachträglich möglich sein zu erkennen, ob das von der Software eingestellte Trainingsprogramm vollständig angewendet wurde oder ob ein vorzeitiger Abbruch durch den Patienten erfolgte.
Ein weiterer Punkt in Bezug auf die Anwendungsüberwachung ist auch das Monitoring der Stimulationsintensität. Der Anwender des Stimulators hat die Möglichkeit die Intensität in gewissen Rahmen selber zu beeinflussen. Er kann also das vorgegebene Trainingsprogramm in seiner Intensität abschwächen oder verstärken. Es muss daher möglich sein, die Änderungen der Trainingsintensität aufzuzeichnen und in geeigneter Form der Software zur Verfügung zu stellen.

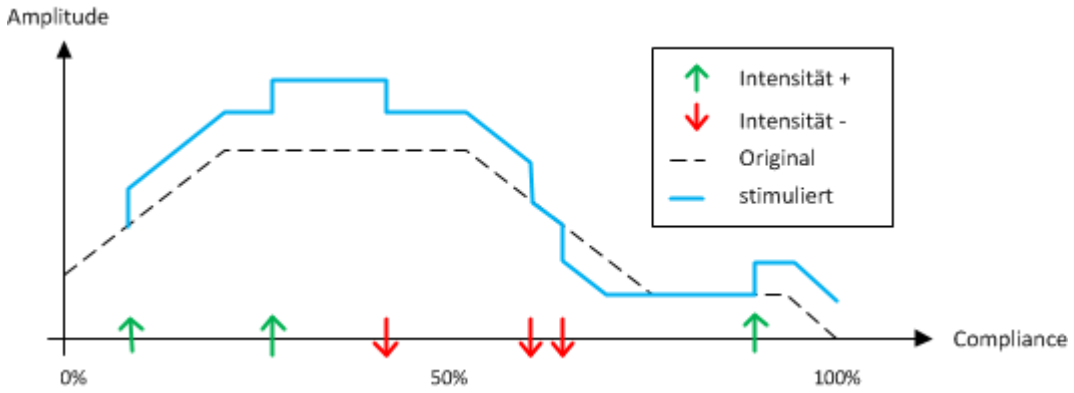


Abbildung 4.9: *Schematische Darstellung der Compliance-Daten. Zeigt den originalen Amplitudenverlauf einer Trainingseinheit (strichliert) und den tatsächlichen Verlauf (blau) durch die Intensitätssteuerung durch den Benutzter (rote und grüne Pfeile). 100% entsprechen ein vollständig absolviertes Trainingsprogramm.*

Wie auch Abbildung 4.9 zeigt beschreiben die Compliance-Daten zwei Teilaspekte. Die Änderung des Amplitudenverlaufs sowie die tatsächliche Trainingszeit in Bezug auf die geplante Stimulationszeit.
Wie schon bei der Angabe der Messpunkte (Abbschnitt 4.2) wird auch die Compliance mit Hilfe der Impulsnummern gespeichert. Das heißt, nicht der Zeitpunkt einer Amplitu-

denänderung wird festgehalten, sondern die Nummer des Impulses bei dem die Änderung aufgetreten ist.

4.4 Datenaustausch

Da der Stimulator autonom, also ohne Verbindung zum Rechner und damit zur Software operieren soll, müssen alle Informationen persistent am Stimulator vorliegen. Das Stimulationsgerät benutzt als Speichermedium deine SD-Karte um darauf die Daten als Dateien zu speichern. Es liegt daher nahe, auch den Datentransfer zwischen Soft- und Hardware über Dateien abzuwickeln.
Der Stimulator präsentiert sich über seine USB-Schnittstelle als Massenspeichermedium und wird beim Verbinden mit dem PC vom Betriebsystem als Wechseldatenträger erkannt. Wie die automatische Erkennung eines Stimulationsgerätes im Detail erfolgt, wird im Abschnitt 5.2.2 auf Seite 79 näher beschrieben.

4.4.1 Ordner- und Dateienstruktur

Um am Mikrokontroller nicht vermehrt Ressourcen in Anspruch nehmen zu müssen, und dennoch eine korrekte Datenzuordnung zu gewährleisten, sollte die Zuteilung der Daten ohne Metadaten erfolgen. Zu diesem Grund wird eine Ordner- und Dateienstruktur festgelegt, an die sich sowohl Stimulator als auch Software zu halten haben.
Diese Struktur sieht für jeden Kanal einen Ordner vor, indem jeweils ein Ordner für Stimultionsdaten, Messwertaufzeichnung und Messergebnisse enthalten ist. Im Ordner für Stimulationsdaten befinden sich die Informationen welche Stimulationsimpulse mit welchen Parametern ausgegeben werden sollten. Dabei wird für jeden Burst genau eine Datei angelegt, welche die im Burst enthaltenen Elemente beschreibt.

Aus Abbildung 4.10 geht hervor, dass jeder Kanal eine eigene Konfigurationsdatei besitzt, die sich von der übergeordneten Konfigurationsdatei im Wurzelverzeichnis des Stimulators unterscheidet. In der Konfigurationsdatei für den jeweiligen Kanal wird das Ziel der Stimulation, also der Muskel für den die Trainingseinheit angedacht ist, vermerkt. Obwohl diese Information für den Stimulator eingentlich nutzlos ist, wird sie am Gerät hinterlegt, damit gewährleistet wird, das die Software die spätere Zuordnung der Messungen erledigen kann.
Nachdem nun bekannt ist wie die Daten am Stimulator abzulegen sind beziehungsweise in welcher Struktur die gemessenen Daten vorliegen, widmen wir uns nun der eigentlichen Datenstruktur, also dem Aufbau der Dateien die zum Datenaustauch herangezogen werden.

4.5 Definition der Dateiformate

Alle Dateien werden in binärer Form gespeichert. Dies ist vorallem auf der Mikrocontroller-Seite von Bedeutung, da dadurch eine schnellere und einfachere Bearbeitung der Daten

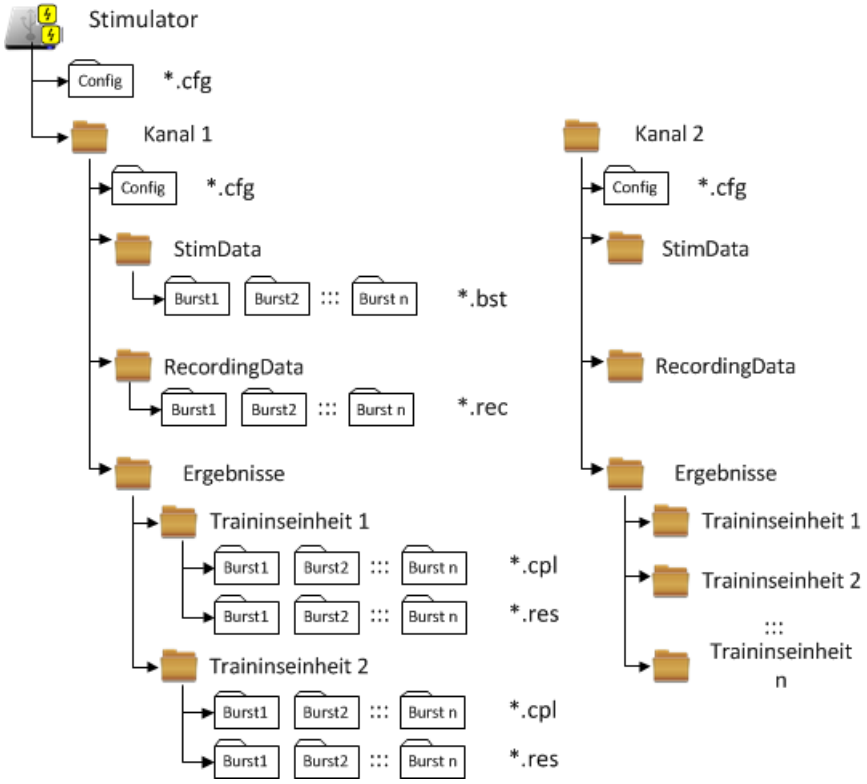


Abbildung 4.10: *Ordnerstruktur auf der SD Karte des Stimulators. Für jeden Kanal existiert ein eigener Ordner indem alle Informationen zu dem jeweiligen Kanal enthalten sind. In einer übergeordneten Konfigurationsdatei werden die Abtastraten für die Messsignale als auch Informationen für die Umrechnung der ADC-Werte gespeichert.*

erfolgen kann. Prinzipiell werden fünf verschiedene Dateitypen für den Datenaustauch zwischen Softwareapplikation und Stimulationshardware verwendet.

(1) Dateien die eine Impulsfolge beschreiben, (2) Konfigurationsdateien, (3) Dateien die Informationen zur Messwertaufnahme beinhalten, und zwei Dateitypen die vom Stimulator im Falle einer Messung angelegt werden. (4) Messdateien, diese beinhalten die Daten zu den Messsignalen und (5) Compliance-Dateien die Auskunft über den Verlauf der angewendeten Trainingseinheiten liefern.

4.5.1 Konfigurationsdateien (*.cfg)

Das System kennt zwei verschiedene Konfigurationsdateien. Zum einen existiert pro Kanal eine Konfigurationsdatei die angibt für welchen Muskel die Stimulation gelten soll im Verzeichniss des jeweiligen Kanal liegt. Zum anderen gibt es eine globale Datei die unter anderem Informationen zur Umrechnung der Messwerte beinhaltet. Während die Kanal-Konfigurationsdateien unterschiedlich groß sein können, hat die globale Datei eine definierte Länge von 80 Bytes.

Kanal-Konfiguration

Diese Datei wird durch die Software bei jeder Übertragung von Stimulationsdaten an den Stimulator neu geschrieben. Der Stimulator selbst greift weder lesend noch schreibend auf diese Datei zu. Sie beinhaltet die Region, beziehungsweise den Muskel für den die Stimulation gelten soll. Diese Informationn dient der Software zur Zuordnung und Visualisierung der Messungen. Die Datei ist sehr einfach aufgebaut und hat folgendes Format.

	Länge	**Target-Information**			
	26	A	B	...	Z
Kodierung	0x1A	0x65	0x66	...	0x90
Byte	0	1	2	...	n

Tabelle 4.4: *Format der Konfigurationsdatei für die Kanäle*

Das Längen-Byte gibt an wieviele Zeichen, kodiert im BCD-Code folgen. Die *Target-Information* hat dabei die Form: `Muskelname_Lage_Seite`.
Zum Beispiel: `Hamstrings_posterior_sinister`. Der Name identifiziert dabei den Muskel oder die Muskelgruppe, die Lage (anterior, posterior) dient der farblichen Darstellung bei der Visualisierung und durch die Seitenangabe kann zwischen links (sinister) und rechts (dexter) unterschieden werden.

Globale Konfiguration

Die Datei für die globale Konfigurationsinformation befindet sich im Stammverzeichnis des Speichermediums im Stimulator und trägt den Namen *config.cfg*. Hier wird neben den Informationen zur Umrechnung der ADC-Werte aus den Messungen auch die Abtastraten für die einzelnen Messsignale gespeichert. Ausßerdem befinden sich in dieser Konfigurationsdatei auch eine 16 Bit lange Nummer (GUID) zur Identifikation der Datenübertragung und ein weiterer 16 Bit langer Schlüssel zur Kodierung der Personen- respektive Patienteninformation.
Diese Datei wird ebenso wie die Konfigurationsdateien für die einzelnen Kanäle, bei jeder neuen Datenübertragung von der Sofware zum Stimulator neu geschrieben beziehungsweise auf den aktuellsten Stand gebracht. Im Gegensatz zur Kanalkonfiguration hat diese Datei immer das selbe Format und damit eine feste Länge von 80 Byte.

DownloadID Die DownloadID, bestehend aus 16 Byte, dient zur Identifikation des Datentransfers. Jedes mal wenn Stimulationsdaten von der Software zum Stimulator geschickt werden, wird in der Datenbank der Software ein Datensatz mit dieser Identifikationsnummer angelegt. Beim Auslesen des Stimulators wird diese Nummer herangezogen um Daten entsprechend zuordnen zu können. Dies ist insbesondere dann von Bedeutung, wenn die Programmierung, also das Aufspielen der Trainingsvorgaben auf den Stimulator, und das Auslesen an unterschiedlichen

DownloadID				**PersonID**				$\mathbf{Ts_u}$		$\mathbf{Ts_i}$		$\mathbf{Ts_{acc}}$		$\mathbf{Ts_{emg}}$
0	1		15	16				32		36		40		44

$\mathbf{K_u}$		$\mathbf{D_u}$		$\mathbf{K_i}$		$\mathbf{D_i}$		$\mathbf{K_{acc}}$		$\mathbf{D_{acc}}$		$\mathbf{K_{emg}}$		$\mathbf{D_{emg}}$	
48		52		56		60		64		68		72		76	

Tabelle 4.5: *Definition der globalen Konfigurationsdatei. Enthält Informationen zum Datentransfer, Patientenschlüssel, Abtatsraten sowie zur Umrechnung der ADC-Werte.*

Orten passiert. Die DownloadID erlaubt also Datensätzte aus unterschiedlichen Datenbanken zu synchronisieren.

PersonID Ebenfalls eine 16 Byte lange GUID und erfüllt den gleichen Zweck wie die DownloadID und wurde zwecks konfortablerer Handhabung beim persistenten Ablegen in die Datenbank eingeführt.

$\mathbf{Ts_u \ldots Ts_{emg}}$ Entsprechen den Abtastintervallen für die Messsignale. Ts_u entspricht dabei dem Intervall für die Abtastung der Spannung Ts_i dem Intervall für die Strommessung usw. Die Angabe der Intervalle erfolgt in Mikrosekunden [μs] und wird im BCD-Format[4] angegeben.

$\mathbf{K_u, D_u \ldots K_{emg}, D_{emg}}$ Die Paare (K_x, D_x) beschreiben die Umrechnungsfaktoren von den ADC-Werten zu den physikalischen Größen. Die Umrechnung erfolgt dabei nach der Geradengleichung $y = kx+d$. Es wird damit also der Multiplikationsfaktor (K_x) und eine mögliche Ordinatenverschiebung (D_x) der Messwerte, die durch den Analog-Digital-Konverter des Stimulators erfasst werden, angegeben.
Die Tatsächliche Umrechnung erfolgt dann durch die Software. Diese Strategie fördert auch die Kompatibilität zu unterschiedlicher Hardware.

4.5.2 Die Burstdatei (*.bst)

Pro zu stimulierenden Burst wird im Ordner `StimData` (Vgl. Abb. 4.10) des jeweiligen Kanals, eine Burstdatei angelegt. Dies passiert durch die Software und geschieht während des Transfers der Stimulationsdaten. Burstdateien haben die Endung (*.bst) und beinhalten im Dateinamen eine Nummer die angibt an welcher Stelle sich der Burst im Kanal befindet.
Die Burstdatei beinhaltet die Anzahl der Elemente die im Burst beinhaltet sind und gibt an wie oft der Burst wiederholt werden soll. Auch die jeweiligen Stimulationsparameter

[4] In dieser Arbeit wird im Zusammenhang mit dem BCD-Format, immer die vom Mikrocontroller verwendete *Big-Endian* Reihenfolge angenommen.

der Stimulationselemente werden als sogenannten Element-Packete in dieser Datei abgelegt. Eine Burstdatei bildet somit alle nötigen Informationen ab, die für die Ausgabe einer gewünschten Impulsfolge vorhanden sein müssen.

Header		**Element Data**								**END**
#E	#R	Element Packet 0				Element Packet n				EOF
		38-Byte				38-Byte				
0	1	2		39	...	2+n·38			n-1	n

Tabelle 4.6: *Struktur einer Burstdatei mit Anzahl der Elementen und Wiederhohlungen sowie den Element-Packeten.*

Element Packet

										Nlet Settings									
E	**StartAmp**		**Incr**		**N**		**T**		**Nl**	**Ta**		**Tb**		**Tc**		**Td**			
	4-Byte		8-Byte		4-Byte		4-Byte			4-Byte		4-Byte		4-Byte		4-Byte			
0	1		5		13		17		21	22		26		30		34			

Tabelle 4.7: *Struktur eines Elementpackets. Die Größe eines Packetes ist immer konstant und beträgt 38 Byte.*

Die Burstdatei beginnt mit einem Header bestehend aus zwei Bytes der angibt wieviele Elemente im Burst enthalten sind und wieoft der Burst wiederholt werden soll. Danach folgen die Element-Informationen in Blöcken zu je 38-Byte. Dabei entspricht der erste Block dem ersten Element im Burst, der zweite Block dem zweiten Element und so fort. Diese Reihenfolge dient der Kodierung der Elementenfolge. Abgeschlossen wird eine Burstdatei mit einem *End of File*(EOF)- Byte.

Druch die Angabe der Startamplitude, der Hüllkurvensteigung und der Impulsanzahl lässt sich die Endamplitude eines Elementes berechnen. Dies nutzt insbesondere die Software bei der Rekonstruktion der ausgelesenen Daten. Der Stimulator benötigt die Angabe der Endamplitude nicht, da er jeden Puls schrittweise abarbeitet. So wird durch Addition des Inkrements und ausgehend von der Startamplitude der Wert der Endamplitude automatisch erreicht.
Ebenso verhält es sich mit der Elementdauer, die auch nicht explizit angegeben und gespeichert wird. Dazu dient sowohl die Anzahl der Impulse als auch die Periodendauer derselben. Der Mikrocontroller am Stimulator generiert entsprechend der Periodendauer die Stimulationsimpulse und produziert so die entsprechende Stimulationsfrequenz.

4.5.3 Dateien mit relevanten Informationen zu den Messungen

Der Stimulator nimmt im Rahmen einer Stimulation normalerweise keine Messungen auf. Mann muss ihm deshalb mitteilen wann, welche Messungen getätigt werden sollen. Die Information besteht darin die Nummer des Stimulationsimpulses, bei dem eine Messung

Symbol	Beschreibung	Datentyp	Wertebereich
#E	Anzahl der Elemente	byte	$1 \ldots 255$
#R	Anzahl der Wiederholungen	byte	$1 \ldots 255$
Element Packet	Elementdaten	byte[38]	$1 \ldots 255$
Element-Daten			
E	Nummer des Elements im Burst	byte	$1 \ldots 255$
StartAmp	Wert der Startamplitude[μV]	uint	$0 \ldots 2^{32}$
Incr	Steigung der Einhüllenden[μV/Imp]	long	$-2^{32} \ldots 2^{32}$
N	Impulsanzahl	uint	$0 \ldots 2^{32}$
T	Periodendauer des Stim-Impuls[μs]	uint	$0 \ldots 2^{32}$
Nl	Anzahl der Nlets	byte	$1 \ldots 255$
T_a	Dauer des positiven Impulses	uint	$0 \ldots 2^{32}$
T_b	Nullzeit der Stimulation	uint	$0 \ldots 2^{32}$
T_c	Negative Impulsdauer	uint	$0 \ldots 2^{32}$
T_d	Zeit zum nächsten intermediären Imp.	uint	$0 \ldots 2^{32}$

Tabelle 4.8: *Erklärung der Symbole zur Beschreibung der Datenstruktur eines Bursts.*

stattfinden soll, anzugeben und mit dem Schlüssel für die gewünschen Signalaufzeichnungen zu versehen.
Die eigentlichen Messdaten werden getrennt von der Information zur Aufnahme der Messungen gespeichert. Dieser Weg wurde gewählt um die Implementierung auf der Stimulatorseite zu erleichtern. Es existieren daher zwei Dateitypen die mit dem Komplex der Messwerterfassung assoziiert werden.

Datei zur Messwertaufzeichnung (*.rec)

Der Anwender legt mit Hilfe des Programms die Zeitpunkte innerhalb eines Bursts fest, an denen Messsignale aufgezeichnet werden sollen. (Vgl. Abb. 4.5). Die Software rechnet die angegebenen Zeiten in entsprechende Impulsnummern um legt bei einem Datentransfer für jeden Burst, für den Messsignale aufgezeichnet werden sollen, eine Datei im Ordner *RecordData* an. Die Datei enthält dabei im Namen die Nummer des Bursts und trägt die Endung *.rec. Wird für einen Burst keine Signalaufnahme festgelegt, so wird auch keine *.rec-Datei dafür geschrieben.
Diese Dateien haben unterschiedliche Größen und können je nach Anzahl der zu vermessenden Stimulationsimpulse entsprechend groß werden. Jeder Impulse bei dem ein Signal entnommen werden sollte ist in dieser Datei nur einmal vermerkt. Das heißt, sollt für einen Impulse sowohl das Signal des Beschleunigungssensors als auch die Signale für Strom und Spannung aufgezeichnet werden, kommt die Nummer des entsprechenden Impulses doch nur einmal in der Datei vor. Dazu wird zu jedem Impuls ein Schlüssel hinterlegt, der angibt welche Signale aufgezeichnet werden sollen.

		Element$_1$					*Element*$_n$				
		impulse dataA		*impulse dataB*				*impulse dataA*		*impulse dataX*	
#E	**LEN_E$_1$**	**C_E$_1$N$_i$**	**N$_i$**	**C_E$_1$N$_j$**	**N$_j$**	...	**LEN_E$_n$**	**C_E$_n$N$_x$**	**N$_x$**	**C_E$_n$N$_y$**	**N$_y$**
1	4-Byte	1	4-Byte	1		...					
0	1	5	6	10		...					n

Tabelle 4.9: *Struktur des Datenfiles zur Messwerterfassung.*

Codebyte : **C_E$_*$N$_*$**

	Status				**Signale**			
	Reserved				EMG	Acc	I	U
Bits	7	6	5	4	3	2	1	0

Tabelle 4.10: *Kodierungsbyte. Gibt an welche Signale gemessen werden sollen.*

Um die Daten in der Datei eindeutig zuordenbar zu machen muss bekannt gemacht werden, wieviele Elemente im Burst enthalten sind. Rein theoretisch stände die Information auch in den Burstdateien (Siehe 4.5.2) zur Verfügung. Für eine einfachere Verarbeitung der Dateien wird diese Informaion hier redundant geführt.
In der Datei befindet sich für jedes, der n Elemente im Burst, ein Längen-Block (LEN_E_n) der angibt, wieviele Bytes an Impulsdaten folgen. Impulsdaten bestehen dabei immer aus fünf Bytes wobei das erste Byte angibt welche Signale gemessen werden sollen und die restlichen vier Bytes spezifizieren die Impulsnummer des jeweiligen Elements. Zur genaueren Studie sei folgendes exemplarisches Beispiel (4.11) angegeben.

#E	**LEN_E$_1$**				**C_E$_1$N$_i$**	**N$_i$**				**C_E$_1$N$_j$**	**N$_j$**			
0x01	0	0	0	0x0A	0x03	0	0	0	0x01	0x0F	0	0	0	0x02
1	10				3	1				15	2			

Tabelle 4.11: *Exemplarische Struktur einer Datei zur Erfassung der Messsignale. Gibt an das der Burst ein Element enthält (#E) und davon die ersten beiden Impulse (*$\mathbf{N_i} = 1$,$\mathbf{N_j} = 2$*) gemessen werden sollen. Und zwar für den ersten Impulse die Signale U und I (*$\mathbf{N_i} = 3$*) und für den zweiten Impuls alle Signale (*$\mathbf{N_j} = 15$*)*

Zur Übersicht seien hier die Symbole zur Beschreibung der Datei für die Messwertaufnahme noch einmal gelistet.

Wie oben beschrieben wird die Nummer eines Impulses nur einmal in der Datei vermerkt, auch wenn für diesen Impulse mehrere Messungen durchgeführt werden sollten. Wird ein Impuls eines Elements keiner Messung unterzogen, so wird er in der Datei auch nicht vermerk. Allerdings werden für einen Burst, sobald auch nur ein einziger Impuls gemessen werden soll, alle Elemente dieses Bursts zumindest im Längenblock der Datei angeführt, auch wenn in einem Element keine Impulse gemessen werden. Hat ein Burst

Symbol	Beschreibung	Datentyp	Wertebereich
#E	Anzahl der Elemente	byte	$1 \ldots 255$
LEN_E_x	Datenbytes für Element x	uint	$0 \ldots 2^{32}$
$C\_E_xN_y$	Code der zu messenden Signale f. Impulsenummer N_y	byte	$1 \ldots 15$
N_y	Impulsnummer	uint	$0 \ldots 2^{32}$

Tabelle 4.12: *Symbole zur Beschreibung der Struktur des Messdatenaufzeichnungsfiles.*

z.B. vier Elemente und wird nur ein Impuls daraus zur Messung angegeben, so hat die Datei eine Mindestgröße von 22 Bytes[5]. Als Konsequenz der Definition der beschriebenen Datenstruktur gilt auch, dass die Dateilänge in Bytes immer ein Vielfaches von fünf sein muss, wenn sie gültige Daten enthalten soll.

Dateien mit Messdaten (*.res)

Der zweite Dateityp der mit Messungen in Verbindung zu bringen ist trägt die Endung *.res. In diesen Dateien werden die eigentlichen Messergebnisse gespeichert. Diese Dateien werden während einer Stimulation, bei der Stimulationsimpulse zur Messung ausgewiesen wurden, vom Stimulator geschrieben. Zu jeder Trainingseinheit wird im Ordner *Ergebnisse* ein weiterer Ordner mit Datum und Uhrzeit des Trainingsstarts angelegt. In diesem Ordner wird dann pro Burst eine Datei abgelegt, welche die Daten der Messsignale beinhaltet.
Für jeden gemessenen Impuls wird ein Block mit Informationen zu allen vier Messsignalen geschrieben. Wird für den jeweiligen Impuls ein Signal nicht gemessen so wird dessen Längenangabe trotzdem geschrieben und mit NULL gefüllt. Durch die Längenangabe bleibt die Struktur flexibel und es ist möglich pro Trainingssitzung eine unterschiedliche Anzahl an Abtastpunkten heranzuziehen.

Block Id			**U Messung**			**I Messung**			**ACC Mess.**		**EMG Mess.**		
nE	nR	nImp	LEN_U	Wert U_1	Wert U_n	LEN_I	Wert I_1	...	LEN_{acc}	...	LEN_{emg}	...	...
0	1	2	6	10	+4	...							n

Tabelle 4.13: *Struktur eines Messblocks der pro zu messenden Impuls in eine Messdatei gespeichert wird.*

Durch die Angabe der 6 Byte großen Blockidentifikation, bestehend aus Nummer des aktuellen Elements, Nummer der aktuellen Wiederholung und die Nummer des aktuellen Impulses kann die Messung später eindeutig zugeordnet werden. Ein solcher Block wird

[5] 1 Byte für die Anzahl der Elemente, 4 × 4 Byte für die Längenangabe der vier Elemente und mindestens 1 × 5 Byte für einen zu messenden Impuls.

für jeden Impuls geschrieben, auch wenn nur eine Messung dafür vorliegt. Ist dies der Fall so würden die Längenangabe der restlichen Messungen den Wert null enthalten und es würden keine Messwerte angegeben sein.
Die vier Längenangaben, pro Messsignal bestehend aus vier Bytes gibt an, vieviele Bytes an Messwerten bis zur nächste Längenangabe folgen. Die Messwerte selber entsprechen den durch den Mikrocontroller abgetasteten ADC-Werten und sind jeweils 4 Byte lang. Eine Umrechnung in die tatsächlichen physikalischen Einheiten wird durch die Software mit Hilfe der Konfigurationsdateien durchgeführt. Alle Angaben erfolgen wie gewohnt im BCD- Format.

Symbol	**Beschreibung**	**Datentyp**	**Wertebereich**
nE	Nummer des aktuellen Elements	byte	$1 \ldots 255$
nR	Nummer der aktuellen Wiederhohlung	byte	$1 \ldots 255$
nImp	Akutelle Impulsnummer	uint	$1 \ldots 2^{32}$
LEN_x	Länge der Messdaten f. Messung x	uint	$0 \ldots 2^{32}$
$Wertx_n$	n-ter ADC-Wert der Messung x	uint	$0 \ldots 2^{32}$

4.5.4 Compliancedatei (*.cmp)

In der Compliancedatei laufen die Informationen zur Anwendungsüberwachung zusammen. Es wird sowohl Zeitpunkt und Zeitdauer, als auch die Änderung der Stimulationsintensität einer Trainingseinheit protokolliert. Wieder wird für jeden Burst eine Datei angelegt. Die Datei enthält im Namen die Nummer des Bursts und trägt die Endung *.cmp.
Die Compliancedatei kann als “Log-File” verstanden werden, in der nach jedem komplett durchlaufenen Element, die Daten zu diesem stimuliertem Element in die Datei geschrieben werden. Zusätzlich werden in der Datei alle Amplitudenänderungen vermerkt, die durch Benutzereingaben am Stimulator hervorgegangen sind.

Modified		**AmplitudeVector$_1$**				...	**AmplitudeVector$_n$**			
nE	nR	nr_1	ne_1	nImp_1	ΔA_1	...	nr_n	ne_n	nImp_n	ΔA_n
		10 Byte				...	10 Byte			
0	1					...				

Tabelle 4.14: *Struktur der Compliancedatei.*

Für jeden Burst von dem zumindest das erste Element vollständig abgearbeitet wurde, wird eine Compliancedatei erstellt die mindestens aus den ersten beiden Bytes besteht. Diese beiden Bytes geben an welches Element (nE) bei der wievielten Wiederholung (nR) gerade abgearbeitet wurde. Nach jedem vollständigen Durchlauf eines Elements werden zumindest diese beiden Bytes vom Stimulator neu geschrieben. Diese

Information kann somit als Vortschritt der Stimulation aufgefasst werden.
Tritt nun eine durch den Benutzer initiierte Intensitätsänderung auf, so wird ein 10 Byte großer Datenvektor in der Datei hinterlegt. Dabei wird neben dem aktuellen Element und der aktuellen Wiederholung auch die Impulsnummer und der Wert der Amplitudenänderung vermerkt.

Symbol	**Beschreibung**	**Datentyp**	**Wertebereich**
nE	Nummer des aktuellen Elements	byte	$1 \ldots 255$
nR	Nummer der aktuellen Wiederhohlung	byte	$1 \ldots 255$
nr_n	Wiederhohlung der n-ten Amplitudenänderung	byte	$1 \ldots 255$
ne_n	Elementnummer der n-ten Amplitudenänderung	byte	$1 \ldots 255$
nImp_n	Impulsnummer der Amplitudenänderung	uint	$1 \ldots 2^{32}$
ΔA_n	Wert der n-ten Amplitudenänderung	int	$-2^{16} \ldots 2^{16}$

Die Amplitudenänderung wird vom Stimulator als Relativwert angegeben. Die Angabe erfolgt dabei aber nicht in physikalischen Einheiten sondern durch die Ausgabe eines Zahlenwertes der am Stimulator einem bestimmten Amplitudenwert entspricht. Dieser Wert wird erst durch die Software in die korrekte Einheit [V], unter zur Hilfenahme der Konfigurationsinformationen, überführt.

KAPITEL 5

Implementierung

Im Folgenden Kapitel werden die software-spezifische Themen behandelt. Es wird die zugrundeliegende Softwarearchitektur beschrieben und gezeigt, wie einzelne Problemstellungen in der Software gelöst wurden. Dieses sehr praxisnahe Kapitel erklärt die verwendete Strategie zur persistenten Archivierung der Daten anhand der zum Einsatz kommenden Datenbank. Auch auf die Anbindung der Datenbank über *Object-Relational-Mapping*, wird dabei eingegangen.
Ein Hauptaugenmerk wird auf das Implementierungsmuster *Model-View-ViewModel* und dessen Umsetzung im Rahmen dieses Softwareprojekts gelegt. Es werden auch technische Einzelheiten kurz erläutert die dem Leser helfen sollen die Software und deren Aufbau zu verstehen.
Auf gegangene Irrwege wird bewußt verzichtet und es wird nur auf endgültige Lösungen eingegangen. Auch soll dieses Kapitel keine Softwaredokumentation in dem Sinne sein, als dass ein Programmierer damit eine "Bedienungsanleitung" für die weitere Implementierung und Erweiterung der Software zur Verfügung hat. Vielmehr sollte dieses Kapitel dem interessierten Leser dabei helfen die Software und deren Ab- beziehungsweise Zusammenhänge zu verstehen und nachvollziehen zu können.

5.1 Übersicht

Da die Entwicklung der Softwarapplikation auf die Plattform der Windows-Betriebsysteme beschränkt werden konnte, wurde von vorne herein voll auf das .NET-Framwork gesetzt. Zur Gestalltung der Oberfläche wurde unter diesen Vorraussetzungen die Windows Presentation Foundation (WPF) gewählt. Diese Kombination bietet somit ein Arbeitsumfeld am aktuellen Stand der Technik. Konkret wird das .NET-Framwork in der aktuellsten Version 4.0 (Stand: März 2011) eingesetzt.
Als Programmiersprache kommt C#, ebenfalls in aktueller Version 4.0 zum Einsatz. Es werden aber keine Features der Sprachversion gegenüber der Version 3.0 benutzt. Die

Software wäre daher auch kompatibel zur Version *C#3.0*.
Somit wird die für die Windowsprogrammierung derzeit am häufigsten verwendete Kombination (.NET, C#, WPF) eingesetzt.

5.1.1 Softwarearchitektur

Die Projektstruktur besteht im groben Überblick aus vier Teilbereichen. Den bedeutendsten Teil bildet die Hauptapplikation, die ihrerseits für die graphische Benutzerschnittstelle, die Anbindung an die Hardware und die komplette Programmlogik verantwortlich ist. Die weiteren drei Teilbereiche werden jeweils in einer dynamischen Bibliothek (*.dll) zusammengefasst und von der Hauptapplikation verwendet. Die hier vorliegende Aufteilung ist in Abbildung 5.1 skizziert und zeigt die Abhängigkeiten der Teilbereiche zueinander.

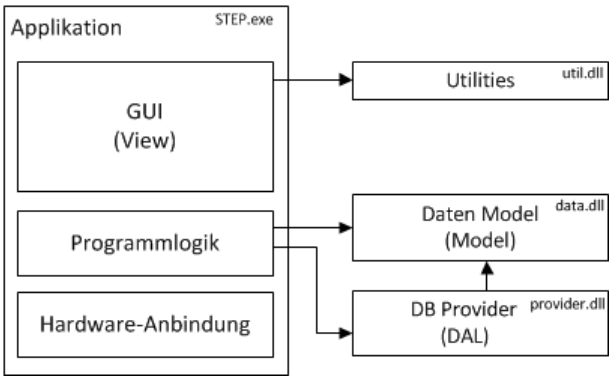


Abbildung 5.1: *Blockdiagramm der Projektstruktur*

Das Datenmodell besitzt kaum Intelligenz und dient vorwiegend der geeigneten Repräsentation der Daten. Hier sind alle Datentypen deklariert die für die Software von Bedeutung sind.
Der Teilbereich des Datenbank Providers dient als *Data Access Layer* (DAL). Über diese Ebene werden vorallem persistente Daten zur Verfügung gestellt. Der Provider implementiert auch die Anbindung an die Datenbank und beinhaltet die objekt-relationale Zuordnung (`object relational mapping`) der Objekte zu den Datenbanktabellen.
Der Block der `util.dll` wird ausschließlich von der grafischen Oberfläche benötigt um bei Drag and Drop Operationen unterstützend auszuhelfen.
Im Verlauf der Entwicklung wurde großer Wert darauf gelegt, eine saubere Trennung von Benutzeroberfläche, Programmlogik und Daten zu erreichen. Zu diesem Zweck wurde das Entwurfsmuster `Model-View-ViewModel` näher studiert und in der Implementierung umgesetzt.

Das MVVM Entwurfsmuster

Das `Model-View-ViewModel` Entwurfsmuster soll helfen in der Softwarearchitektur eine saubere Trennung von Darstellung, Verhalten und den Daten zu erzielen. Das Entwurfsmodel wurde 2005 von John Grossman, einem Softwarearchitekten der auch WPF mitentwickelte, veröffentlicht [43].
Obwohl das `Model` für alle Technologien die Datenbindung unterstützen angewendet werden kann, steht es doch sehr im Zusammenhang mit WPF und nutzt dessen Vorteile aus um eine wartbarere und testbarere Softwarearchitektur umzusetzten. Insbesondere das Konzept der "gesichtslosen" Steuerelemente, also Steuerelemente ohne Aussehen (Vgl. ContentControl in Abschn. 2.2.2) von WPF, unterstützt die klare Trennung von Benutzerschnittstelle und des darunterliegenden Verhaltens.

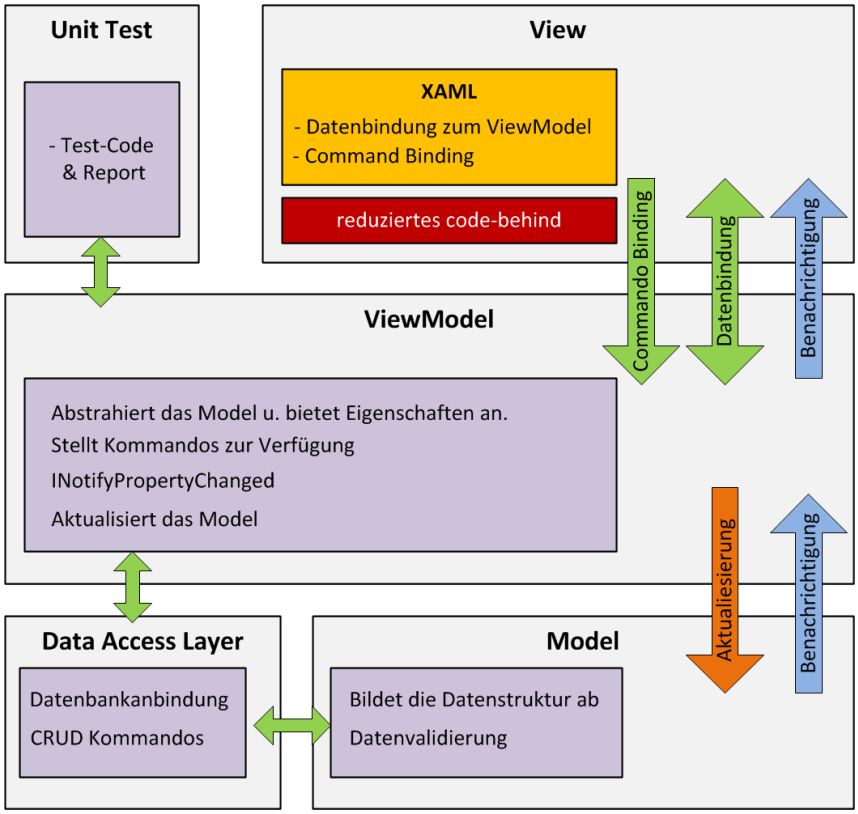


Abbildung 5.2: *Schichten des MVVM Entwurfsmusters*

Die View hat Wissen über das `ViewModel` und das `ViewModel` weiß über die Existenz des `Models`, nicht aber umgekehrt.

View Die View repräsentiert alles was der Benutzer zu Gesicht bekommt und beinhaltet visuelle Steuerelemente, Animationen und interaktive Schaltflächen. Die `View` ist neben der Repräsentation der Daten auch für die Ereignissbehandlung und de-

ren Weiterleitung an das `ViewModel` verantwortlich. Die lose Kopplung der `View` an das `ViewModel` erfolgt durch Datenbindung. Dabei wird dem DataContext der `View` das entsprechende `ViewModel` als Quelle für die Datenbindung zugewiesen. Für jede unterschiedliche Visualisierung existiert eine eigene `View`, auch wenn dabei auf die gleichen Daten und somit auf das gleiche `ViewModel` zugegriffen wird. Die Angabe des DataContextes bedingt das Wissen der `View` über das `ViewModel`. Umgekehrt hat das `ViewModel` keine Kenntniss über die `View`.

ViewModel Das `ViewModel` spielt die Schlüsselrolle beim Loslösen der grafischen Schnittstelle, beziehungsweise beim Trennen von Anzeige und Daten. Es dient quasi als Vermittler zwischen den Daten und deren Anzeige. Um das Verhalten des `ViewModel` etwas näher zu erläutern sei folgendes Beispiel angeführt. Die Anzeige einer Uhrzeit könnte durch eine analoge oder digitale Uhr dargestellt werden. Dies würde zwei unterschiedlichen `Views` entsprechen. Die Daten liegen aber als Zahlenwert vor. Zum Beispiel als Anzahl der Sekunden die seit dem 1.1.1970 (UnixTime) vergangen sind. Das `ViewModel` als vermittler zwischen den Beiden Schichten hält nun einfach die formatierten Daten mit Angaben zu Stunden, Minuten, Sekunden etc.
Werden Daten durch Benutzereingaben manipuliert, so sorgt das ViewModel für die korrekte Umrechnung und Übermittlung der Änderungen an das `Model`.

Model Das `DatenModel` ist verantwortlich für die Zuverfügungstellung der Daten und deren Verwendung in WPF. Alle Zugriffspunkte des Models weden vom UI-Thread aus aufgerufen. Um über Änderungen zu informieren muss das Model das Interface `INotifyPropertyChanged` implementieren. Zeitintensive Berechnungen werden vom Model ausgelagert um die Benutzterschnittstelle nicht zu blockieren. Das Model beinhaltet die Daten zur Laufzeit und kann dabei Daten aus mehreren Quellen kombinieren. Die Model-Klassen sind prinzipiell sehr einfache Klassen und erlauben ein einfaches Testen mittels Unit-Tests.

Der Ablauf und Kontrollfluss im MVVM-Entwurfsmuster passiert generell immer nach folgendem Schema:

(1) Der Benutzer interagiert mit der Oberfläche.

(2) Das `ViewModel` reagiert auf die weitergeleiteten Ereignisse.

(3) Das `ViewModel` ruft die entsprechenden Methoden im `Model` auf.

(4) Daraufhin werden vom `Model` geeignete Aktionen im DAL ausgelöst.

(5) Die DAL Schicht sendet die geeigneten Kommandos an die Datenbank.

(6) Der Zustand der Datenbank wird geändert und die entsprechenden Daten werden retourniert.

(7) Weiterreichen und Änderung der Daten durch das `Model` an das `ViewModel`.

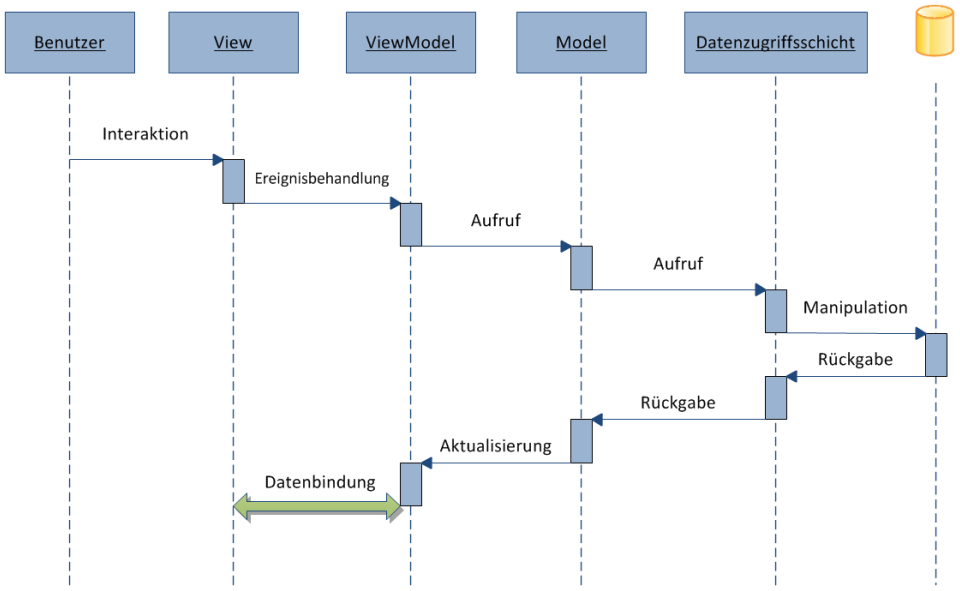


Abbildung 5.3: *Ablauf im MMVM-Entwurfsmuster*

(8) Aktualiesierung der `View` durch die Datenbindung.

(9) Warten auf weitere Benutzereingaben die den Zyklus erneut in Gang setzen.

Vorteile des MVVM-Entwurfsmusters

- Weniger Code im Code-Behind-File der View. Dadurch einfachere und übersichtlichere Handhabung bei Änderungen an der Benutzerschnittstelle.
- Die grafische Benutzerschnittstelle kann zur Gänze in XAML implementiert werden. Dies vereinfacht die Zusammenarbeit von Programmierern und Designern.
- Der Mechanismus der Datenbindung hat eine lose Kopplung von Oberfläche und Programmlogik zur Folge. Dadurch wird viel Programmcode eingespart und der Programmierer braucht sich nicht um die Aktualiesierung der Benutzeroberfläche zu kümmern.

Nachteile und Kritik am MVVM

- In sehr grossen Projekten kann die Erstellung von geeigneten ViewModels sehr aufwändig werden, da das finden eines generalisierten ViewModels unter solchen Voraussetzungen viel Aufwand bedeuten kann.
- Bei aller Mächtigkeit des Mechanismus der Datenbindung, ist diese auf Grund ihrer deklarativen Implementierung in XAML schwer zu debuggen. Dies erschwert die Fehlerfindung bei Implementierungsfehlern die die Datenbindung betreffen.

5.1.2 Modellierung der Daten

Die Softwarapplikation muss im wesentlichen drei Funktionsblöcke abbilden. *a*) Die Einstellung und Zusammensetzung der Stimulationsparameter *b*) Das Programmieren des Stimulators *c*) Auslesen des Stimulationsgerätes, und Anzeigen der Ergebnisse sowie *d*) die Daten- und Patientenverwaltung.

In Abbildung 5.4 ist das Klassendiagramm für die Datenstruktur der Stimulationsinformation zu sehen. Ein Kanal wird durch die Klasse `Channel` abgebildet und enthält eine Liste an `Bursts`. In der Burst-Klasse werden sowohl die Daten über die Messwertaufnahme verwaltet (`Recording`) sowie die Elementinformationen. Dafür verwaltet die Burst-Klasse eine Liste an Elementen.
Die Element-Klasse beinhaltet die eigenlichen Parameter die einen Stimulationsimpuls beschreiben. Die Position der Elemente innerhalb eines `Bursts` wird dabei von der Burst-Klasse verwaltet und gesetzt.

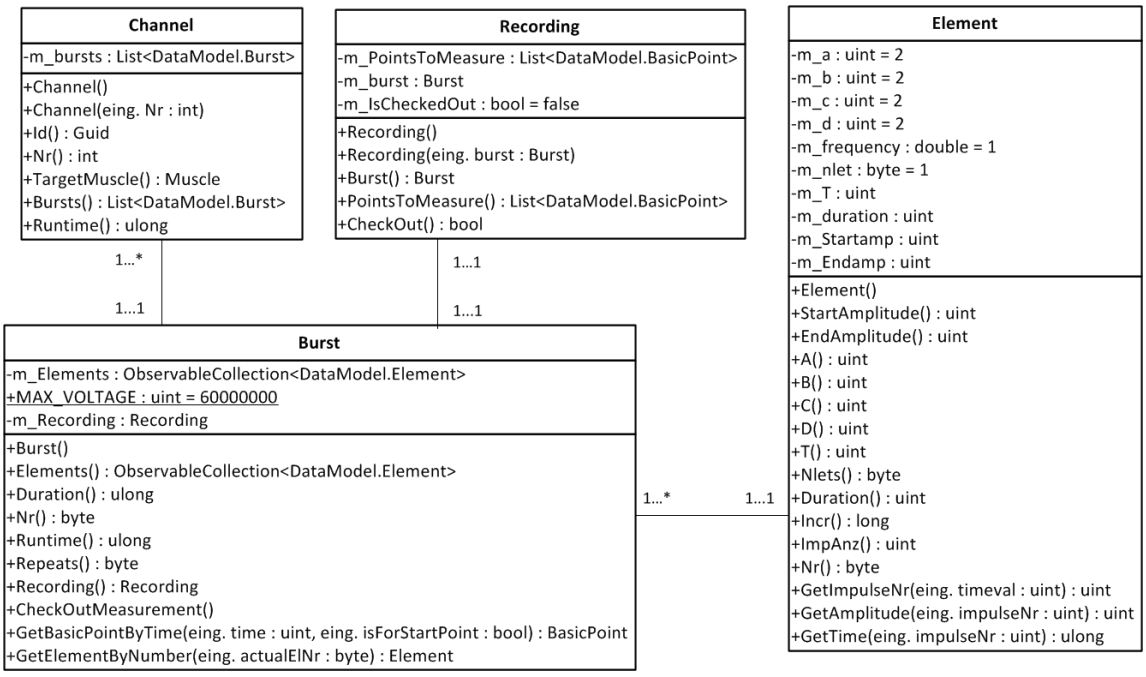


Abbildung 5.4: *Ausschnitt aus dem Klassendiagramm, zeigt die Datenstrukturen und deren Komposition zur Abbildung der Stimulationsdaten*

Auf dieser Grundstruktur, die die wesentliche Struktur der wichtigsten Daten abbildet, baut die gesamte Software auf. Die in Abbildung 5.4 dargestellten Klassen stellen im wesentlichen nur die Datenstruktur zur Verfügung und beinhalten bis auf die Recording-Klasse kaum Logik. Die Recording-Klasse dient dazu die Messpunktaufzeichnung zu verwalten. Sie speichert also alle vom Benutzer angegebenen Messpunkte und rechnet diese mit Hilfe der Burstinformation in ein geeignetes Ausgabeformat um.
Wesentlich interessanter sind die ViewModel-Klassen die schon eine erste Abstraktion

der Datenklassen bilden und auch schon Zustände der Benutzeroberfläche mit einbinden. Prinzipiell besitzt jede `View` ein `ViewModel`. Ein `ViewModel` kann aber für mehrere `Views` als Grundlage dienen.

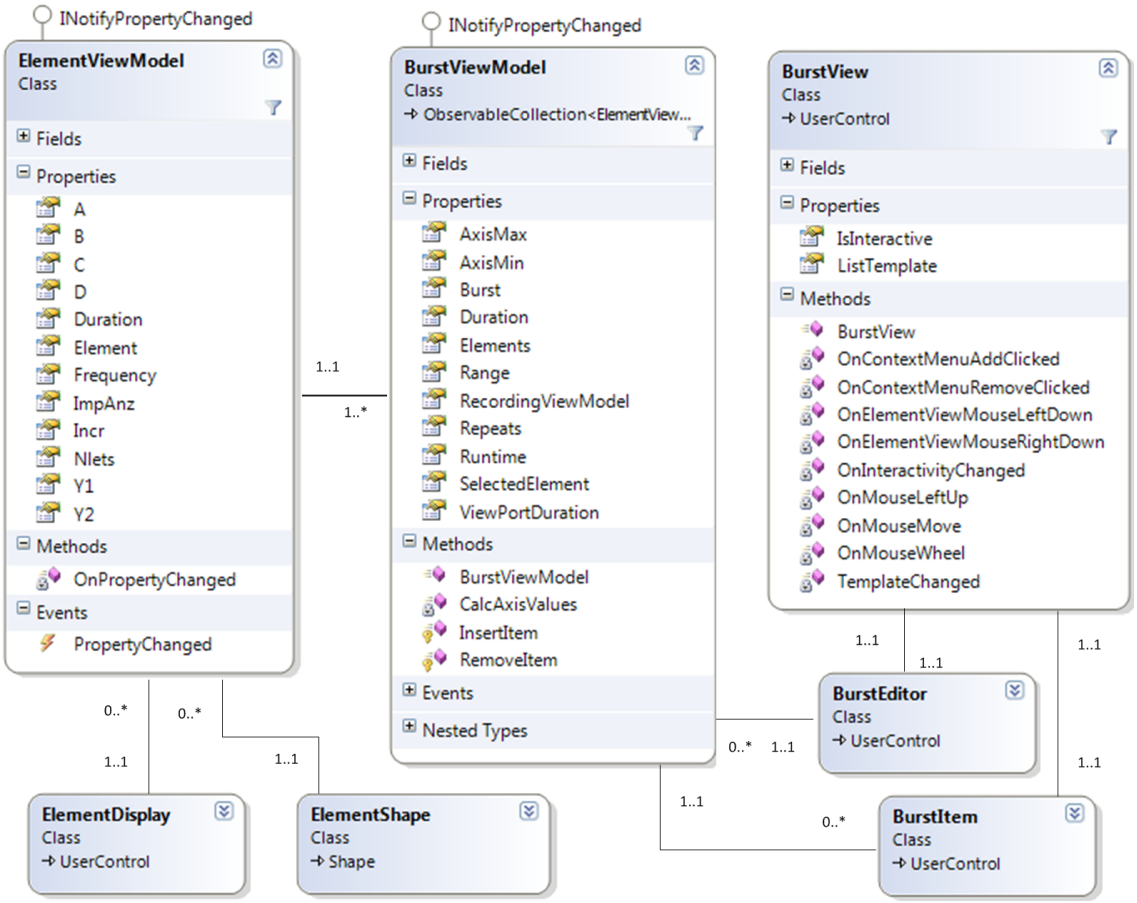


Abbildung 5.5: *Zeigt die ViewModels als Abstraktion für die Element- und für die Burst-Klasse. Auch die Zusammenhänge mit den Views die diese beiden ViewModels nutzten sind skizziert. (Unvollständiges Klassendiagramm)*

Man erkennt aus Abbildung 5.5 das beispielsweise das `BurstViewModel` für zwei verschiedene `Views` (BurstEditor und BurstItem) als `ViewModel`-Klasse dient. In unserem Fall dient die BurstEditor-Klasse zur grafischen und interaktiven Manipulation eines Bursts. Die BurstItem-Klasse stellt nur die Hüllkurve eines Bursts dar und erlaubt keine interaktive Veränderung derselben.
Beide Klassen verwenden zur Darstellung der Hüllkurve die BurstView-Klasse. Diese Klasse bietet die Möglichkeit einen Burst wahlweise interaktiv darzustellen um ihn grafisch zu manipulieren, oder ressourcenschonend ohne Interaktivität anzuzeigen wenn es sich um die reine Visualisierung eines Burst handelt. Durch den vererbten Datenkontext wird die BurstViewModel-Klasse in beiden Fällen an die `BurstView` weitergereicht.

Zur näheren Erläuterung des Konzepts der ViewModel-Klassen, möchte ich anhand der ElementViewModel-Klasse eine detailiertere Ausführung daraus geben. Wenn man die Datenstruktur eines Elements noch einmal betrachtet (Vgl. Abb. 4.2 in Abschn. 4.1.1, Seite 44), so wird ersichtlich, dass die eigentlichen Daten in der Element-Klasse (Abb. 5.4) keine Eigenschaft für die Frequenz besitzt, sondern dies Information über die Periodendauer kodiert wird. Da für die grafische Darstellung die Frequenz von Bedeutung ist, übersetzt die ElementViewModel-Klasse diese Information in eine eigene Eigenschaft an die in weiterer Folge die Oberfläche zu Zwecken der Anzeige bindet. Die Eigenschaften der Elementdaten bezüglich der Start- und Endamplitude werden durch das ViewModel durch die Eigenschaften (Y1, Y2) weitergereicht. Durch die Datenbindung mittels Converter werden die beiden Eigenschaften in Pixelwerte umgerechnet, die für die Anzeige der Geometrie am Bildschirm benötigt wird.
Im Zusammenhang mit den Amplitudenwerten gilt zu erwähnen, dass die Validierung der Daten ebenfalls vom ViewModel übernommen wird. Dies entspricht zwar nicht zu 100% der Konvention des MVVM-Musters, wonach die Validierung der Daten eher in das Model verlagert werden würde. Diese Entscheidung wurde aber zu Gunsten der Schlichtheit der Model-Klassen getroffen.
Ein weiteres wichtiges Merkmal der ViewModel Klassen ist die implementierte Schnittstelle INotifyPropertyChanged. Dadurch wird bei der Datenbindung die View über Änderungen der Eigenschaften im ViewModel benachrichtigt und automatisch aktualisiert. Quasi alle Klassen aus der ViewModel-Ebene implementieren diese Benachrichtigungsschnittstelle. Im Falle der Beziehung zwischen BurstViewModel und ElementViewModel, wird diese Schnittstelle aber nicht nur von der Datenbindung genutzt um die Benutzerschnittstelle zu benachrichtigen, sondern auch die darüberliegende Ebene in der selben Schicht. Konkret verwendet die BurstViewModel-Klasse den Benachrichtigungs-Mechanismus um bei Änderungen der Elementdauer die eigene Laufzeit anzupassen.
Anhand der BurstViewModel-Klasse ist auch ersichtlich wie das Verhalten der Oberfläche an die ViewModel-Klasse ausgelagert ist. Zum Beispiel wird das aktuell selektierte Element nicht direkt durch die Benutzerschnittstelle markiert, sonder indirekt durch Datenbindung auf Grund der entsprechenden Veränderung der Eigenschaft SelectedElement im ViewModel.
Genauso wie eine View aus mehreren Views (z.B. BurstEditor) aufgebaut sein kann, können auch ViewModel-Klassen mehrere ViewModels enthalten. So besteht die BurstViewModel-Klasse aus einer Liste von ElementViewModels. Auch das dem BurstViewModel übergeordnete ChannelViewModel besteht aus einer Liste von BurstViewModels.
Das ViewModel des Bursts verwaltet auch schon wirklich wichtige Elemente die zur Darstellung benötigt werden. Die Eigenschaften AxisMax und AxisMin geben beispielsweise den Wert an den die Achsen in der Anzeige erhalten. Genauer bedeutet dies, nicht die Oberfläche bestimmt die Werte ihrer angezeigten Achsen, sondern das ViewModel berechnet die Werte und benachrichtigt über die Schnittstelle den Datenbindungsmechanismus der View die gegebenenfalls die Anzeige korrigiert.

Das zentrale ViewModel - der DataStore

Die Architektur des Projekts wurde so gewählt, dass an einer zentralen Stelle alle Fäden zusammenlaufen. Dafür wurde die `DataStore`-Klasse eingeführt. Dies ist eine *Singletone*- Klasse von der zur gesammten Laufzeit des Programms nur eine Instanz existiert. Dies garantiert dass alle anderen Klassen, die ein Objekt der DataStore-Klasse halten, das selbe Objekt referenzieren und so die Konsistenz der Daten leichter gesichert werden kann.
Das `DataStore`-Objekt ist also quasi das Haupt-ViewModel von dem aus zu allen anderen ViewModels verzweigt wird. Es ist nicht mit einem BasisViewModel zu vergleichen, von dem in der Literatur immer wieder die Rede ist. Dies wäre eine Basisklasse von der alle ViewModels erben und die schon die Hauptfunktionalitäten eines ViewModels implementiert. Auf dieses mitunter sehr übersichtliche Design wurde auf Grund des späten Umstiegs auf das MVVM-Entwurfsmuster zu einem recht späten Zeitpunkt der Entwicklung verzichtet.

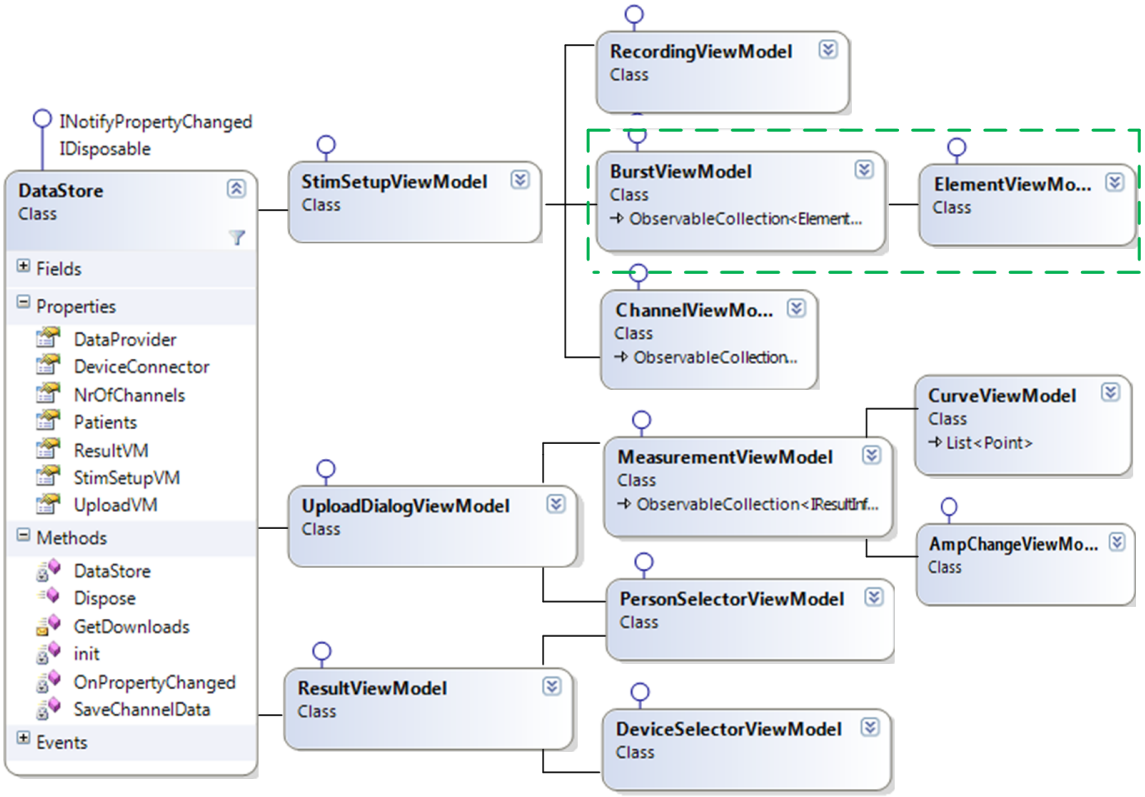


Abbildung 5.6: *Das DataStore-Objekt als zentrales ViewModel-Objekt. Von hier aus verzweigen sämtliche ViewModel-Klassen. Der grün markierte Bereich zeigt den in Abbildung 5.5 beschriebenen Teilbereich.*

Das `DataStore`-Objekt ist das erste Objekt das beim Starten der Applikation er-

zeugt wird. Während der Laufzeit existiert durch das Erzeugen als Singletone-Objekt immer nur eine Instanz dieser Klasse. Sie verwaltet eigentlich alle weiteren ViewModels und kümmert sich um die Einbindung des `DataAccessLayers (DAL)` und somit um die Verbindung zur Datenbank.
Auch die Anbindung der Hardware, also das Erkennen der angeschlossenen Stimulatoren wird in dieser Klasse bewerkstelligt. Dazu dient eine Instanz des `DeviceConnectors`. Diese Instanz wird auch anderen ViewModels zur Verfügung gestellt falls diese Informationen über verbundene Geräte beziehen müssen. Dies ist insbesondere die Klasse `DeviceSelectorViewModel`.
In Abbildung 5.6 auch zu erkennen sind die drei großen Teilbereiche und ihre Pendants als ViewModel-Klassen.

A) Der Teilbereich zum Erstellen der Stimulationsinformationen, abgebildet durch die `StimSetupViewModel`-Klasse. Hier werden alle Aktionen zur Manipulation und Zusammenstellung der Stimulationsparameter vereinigt. Auch das Laden und Speichern der Stimulationsinformation wird in diesem Teilbereich realisiert.

B) Das Auslesen des Stimulationsgerätes wird über die `UploadDialogViewModel`-Klasse abgewickelt. Insbesondere die Reassemblierung der ausgelesenen Daten ist Aufgabe dieses Teilbereichs. Und

C) der Teilbereich der Ergebnisbehandlung repräsentiert durch das `ResultViewModel`. Hier findet vorallem die Visualisierung der Stimulationsergebnisse statt. Auch das Sichten aller patientenbezogener Daten und schon persistent gespeicherten Resultate wird von diesem Teilbereich übernommen.

5.1.3 Datenbankaufbau

Als Datenbank wird eine “Sql Server Compact” Datenbank in Version 3.5 verwendet. Das Produkt ist eine von Microsoft entwickelte relationale Datenbank die frei erhältlich ist und gratis mit der Applikation veröffentlicht und verteilt werden kann [44]. Die Wahl zu Gunsten dieser Datenbank viel auf Grund der Unterstützung von `LINQ` und dem `Entity Framework`. Auch das komfortable Management der Datenbank mit Hilfe der Entwicklungsumgebung vereinfachte die Entscheidung diese Datenbank zu verwenden.
Eine Eigenheit der Datenbank gegenüber anderen Produkten ist die Tatsache, dass die Datenbank im selben Prozess läuft wie die Anwendung die sie verwendet. Dabei verwenden alle Instanzen der Datenbank auch den selben Speicherbereich.

Da mit Hilfe des `Entity-Frameworks` aus der Datenbank ein “objekt-relationales-mapping” durchgeführt werden kann, also die Überführung von den Tabellen der Datenbank in Objekte die von der Programmlogik verwendet werden können, ließen sich diese Objekte auch direkt als Datenschicht verwenden. In der Praxis wird dies auch tatsächlich recht häufig eingesetzt. Hier aber kommt eine “Zwischenschicht” zum Datenzugriff (DAL) zum Einsatz.
Um die Unterscheidung von Objekten der Model-Klassen zu erleichtern, wird für die Datenbanktabellen das Präfix `Tb_`(für Table) vergeben. Durch das Datenmodell stehen

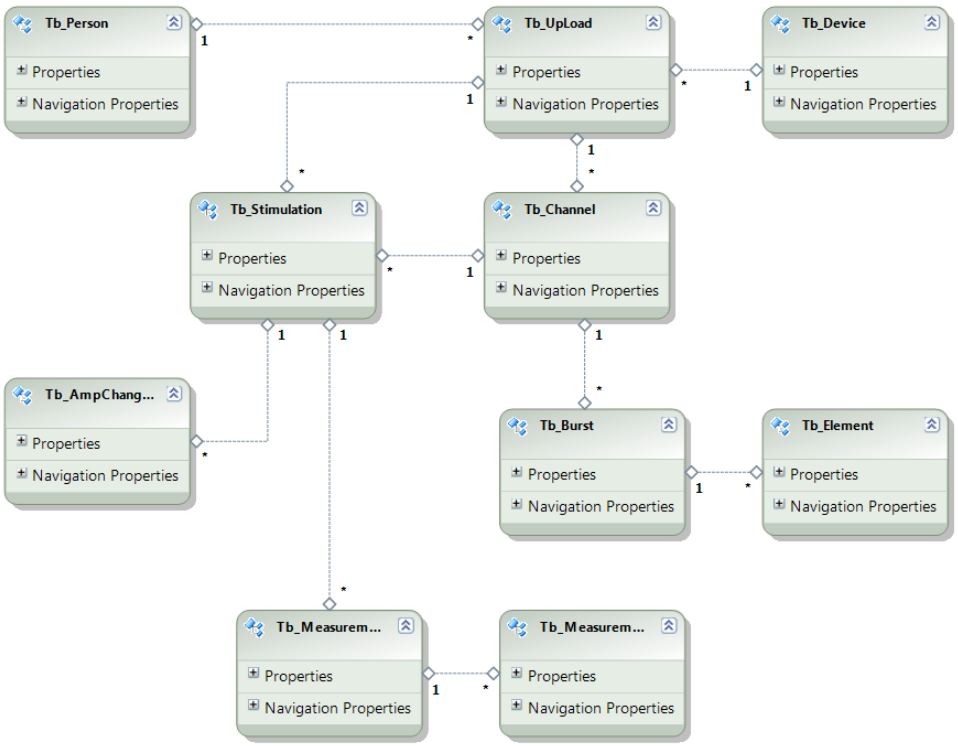


Abbildung 5.7: *Das aus der Datenbank mit Hilfe des Entity-Frameworks generierte Datenmodell und die Abhängigkeiten der Entitäten. Der Aufbau bzw. die Struktur ähnelt der im Abschnitt 5.1.2 vorgestellten Datenstruktur.*

nun Objekte zur Verfügung die sich wie normale Klassen verwenden lassen.
Alle Zugriffe zur Datenbank laufen im Paket `DataProvider` ab, welches als dynamische Bibliothek ins Projekt eingebunden ist. Die Klasse `DatabaseDataProvider` in dieser Bibliothek implementiert das Interface `IDBProvider`, welches alle nötigen Methoden für Datenbankzugriffe bereitstellt. Dieses Prinzip leitet sich vom *Repository*-Entwurfsmuster ab. Die ViewModels die Zugriff auf die Datenzugriffsschicht benötigen, halten eine Instanz vom Typ `IDBProvider`. Ein weiterer Vorteil neben einer sauberen Trennung der verschiedenen Schichten ist, dass durch die Implementierung des Interfaces auch andere Datenbanken oder dahinterliegende Speicherarten angebunden werden könnten.
Die Datenbank selbst lieg als einzelne Datei (*StepAppDB.sdf*) im Programmverzeichnis der Applikation. Durch die Klasse `StepAppDBEntities` wird durch die objektrelationale-Zuordnung mit Hilfe des *Entiy-Frameworks* die Datenbank repräsentiert.
Die Klasse `DatabaseDataProvider` implementiert die Schnittstelle `IDBProvider` und die Logik der Abfragen gegen das Datenmodell. Dabei wird vorallem *LINQ to Entity* als Anfragemechanismus verwendet.

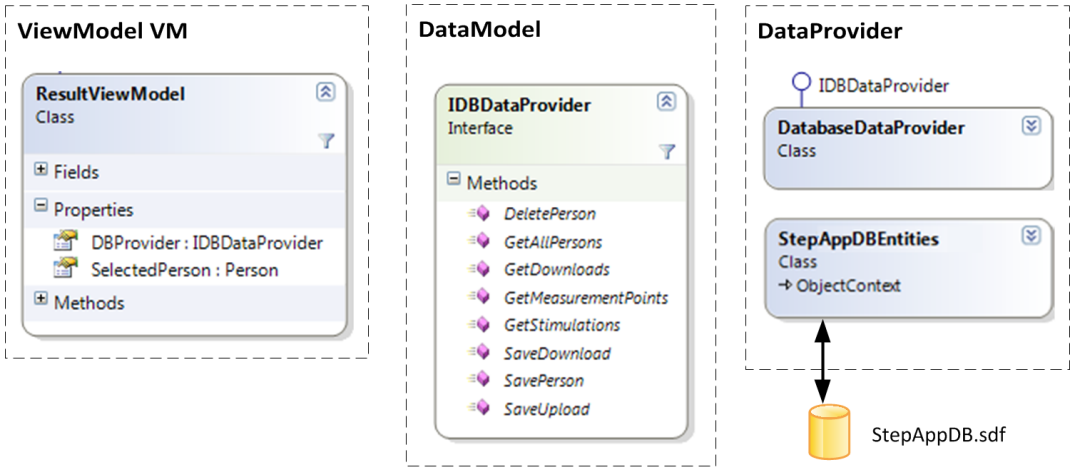


Abbildung 5.8: *Zugriff auf die Datenbank mit Hilfe des Entity-Frameworks. Die Implementierung der Schnittstelle IDBDataProvider realisiert das Repository-Pattern. Alle ViewModels die Zugriff auf die Datenbank benötigen halten eine Instanz der Repository-Schnittstelle.*

5.1.4 Verwendete Werkzeuge

Die Entwicklung der Softwareapplikation fand abwechselnd auf zwei Maschinen statt. Zum einen am Laptop, auf dem auch alle Tests im Labor und am Institut stattfanden und zum anderen auf einem Desktop PC der vorallem für die Entwicklung der Oberfläche und bei Heimarbeiten verwendet wurde. Der Datenaustausch beziehungsweise die Synchronisation der Änderungen wurde wie auch die Versionskontrolle über einen lokalen SVN-Server am Desktop-Rechner abgewickelt. Für den Datenaustausch mit den anderen Projektmittgliedern kam ein webbasiertes Filesharing-Programm zum Einsatz.

Zweck	**Werkzeug**	**Beschreibung**
Hardware	Laptop	Intel Core2 Duo 2×2.5 GHz, 3.0 GB RAM, Win7 32-Bit
Hardware	PC	AMD QUAD Core Athlon II X4 4×2.9 GHz, 4.0 GB RAM, Win7 32-Bit
Entwicklungsumbegung	Visual Studio 2010	Version 10.0.30319.1 RTMRel
Designumgebung	Expression Blend 4	Edition Ultimate Version 4
Versionskontrolle	Subversion	TortoiseSVN 1.6.7 und Server 1.6.9

Tabelle 5.1: *Auflistung der verwendeten Werkzeuge*

Auf beiden Rechnern wurde mit einer durch die MSDNAA[1] der TU-Wien lizensierten Version des Betriebssystems Windows 7 (Professioanl Edition) gearbeitet.
Als Entwicklungsumgebung kamen zwei weitere Werkzeuge der MSDNAA zum Einsatz.

Das VisualStudio 2010 (Ultimate Edition) und Expression Blend 4. Mit Hilfe des VisualStudios 2010 wurde die komplette Programmlogik implementiert und alle automatisierten Unit-Tests geschrieben. Auch beim Design und der Manipulation der Datenbank sowie natürlich beim Debuggen war diese Entwicklungsumgebung eine große Hilfe.
Für das Design der Oberfläche, dem Erstellen von grafischen Elementen und zur Realisierung der Datenbindung wurde hauptsächlich das Programm Expression Blend in Version 4 verwendet.

5.2 Implementierungsdetails

Um einen etwas tieferen Einblick in die technischen Details der Programmierung der Anwendung zu erhalten, sollen an dieser Stelle einige ausgewählte Strategien der Implementierung näher betrachtet werden. Vorwiegend werden in diesem Abschnitt WPF- bezogene Implementierungsdetails wie das Konzept der `DependencyProperties`, `Templating` oder die Kommandobindung behandelt. Aber auch Anfragen an die Betriebssystemebene mit Hilfe der `Windows Management Instrumentation` (WMI) werden an Hand von Beispielen näher beleuchtet.

5.2.1 Grafische Manipulation der Stimulationselemente

Ein sehr bedeutendes Problem bestand darin die grafische Manipulation der Stimulationsparameter durch die Softwareanwendung zu realisieren. Es sollte dem Benutzer durch Maus-Interaktion ermöglicht werden sowohl Anfangs- und Endamplitude, als auch Stimulationszeit grafisch zu verändern. Die Schwierigkeit liegt darin, dass eine Änderung der Elementedauer auch Auswirkungen auf die Burstdauer und in weiterer Folge auf die Kanallaufzeit hat.

Wird also die Zeit eines Elements verändert, nimmt die grafische Darstellung des Elements einen größeren Bereich ein und verschiebt dabei alle nachfolgenden Elemente entlang der Zeitachse. Da aber ein Element durch die Datenspezifikation (Vgl. Abschnitt 4.1.1) als atomar angesehen wird und deshalb in keinem Zusammenhang mit den benachbarten Elementen steht, ist die Umsetzung nicht trivial.
Zur Lösung des Problems wird die grafische Darstellung eines Elements als `UserControl` ausgeführt das seine Größe am Bildschirm selbstständig anpassen kann. Dazu wird die Breite (`Width`) des `UserControls` mit Hilfe multipler Datenbindung und über einen `Converter` an die Eigenschaft `Duration` des ElementViewModel sowie die Eigenschaften `ViewPortDuration` und `ActualWidth` gebunden. Die Mehrfachbindung wird deshalb eingesetzt, da die Position des Steuerelements am Bildschirm von mehreren Faktoren abhängt. Nämlich

(a) von der tatsächlichen Dauer der Stimulationszeit des Elements (b) der Größe des übergeordneten Elements (BurstView). Diese Tatsache kommt vorallem bei Größenänderungen zum Beispiel durch minimieren des Fensters zum Tragen. Je weniger Pixel für

[1] MSDNAA = Microsoft Developer Network Academic Alliance. Dabei können Microsoft-Produkte über die Hochschule zur nicht kommerziellen Nutzung erstanden werden.

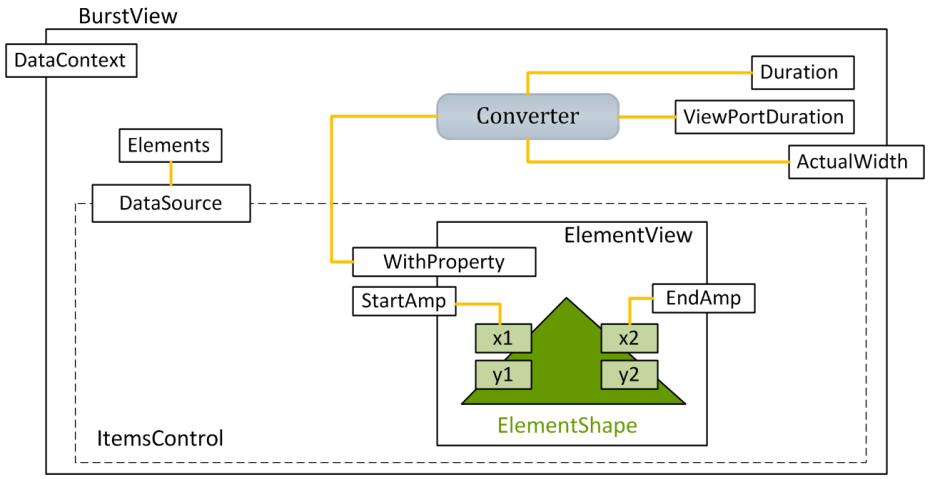


Abbildung 5.9: *Schematischer Aufbau der BurstView mit skizzierter Mehrfachbindung und deren Abhängigkeiten für die Größenanpassung der ElementView.*

die Darstellung zur Verfügung stehen desto komprimierter die Darstellung des Elements. Und (c) von der Dimension des Ausschnitts welcher angezeigt werden sollte. Dieser Ausschnitt wird über die Eigenschaft `ViewPortDuration` im ViewModel angegeben. Diese beschreibt das Zeitfenster welches am Bildschirm sichtbar ist.

Die Mehrfachbindung hat den Vorteil dass bei Änderung einer abhängigen Eigenschaft die Breite des Steuerelements automatisch neu berechnet wird und sich dadurch die Größe selbstständig anpasst.
Dadurch dass für die Darstellung der Elemente ein ItemsControl verwendet wird und die `ElementViews` als Vorlagen für die Anzeige dienen, wird durch die Größenanpassung der einzelnen Elemente auch die Verschiebung entlang der Zeitachse ohne zusätzlichen Rechenaufwand erreicht. Die Elemente im ItemsControl ordnen sich nämlich nacheinander an, wobei der Platzbedarf der jeweiligen Stuerelemente durch das ItemsControl selbst bestimmt wird.

```
1  <DataTemplate x:Key="AcitveElementViewDataTemplate">
2   <local:ElementView MouseLeftButtonDown="OnElementViewMouseLeftDown" MouseRightButtonDown="
        OnElementViewMouseRightDown" >
3    ....
4     <local:ElementView.Width>
5      <MultiBinding Converter="{StaticResource DurationToWidthConverter}" Mode="TwoWay">
6       <Binding Path="Duration" Mode="TwoWay"/>
7       <Binding RelativeSource="{RelativeSource FindAncestor, AncestorType={x:Type UserControl}}"
            Path="DataContext.ViewPortDuration" Mode="OneWay"/>
8       <Binding RelativeSource="{RelativeSource FindAncestor, AncestorType={x:Type UserControl}}"
            Path="ActualWidth" Mode="OneWay" />
9      </MultiBinding>
10    </local:ElementView.Width>
11  </local:ElementView>
```

Quellcode 5.1: *Deklaratives MultiBinding der Eigenschaft Width vom ElementView Steuerelement zur Größenanpassung am Bildschirm*

Wie Abbildung 5.10 zeigt, besteht ein `ElementView`-Steuerelement aus einer `ElementShape` und zwei `MarkerShapes`, alles von `Shape` abgeleitete Klassen. Die `ElementShape` bildet dabei die Fläche unter der Hüllkurve ab und positioniert dazu ein `Path`-Element am Bildschirm. Dazu implementiert die `ElementShape`-Klasse die Dependency-Properties x_1, y_1 bzw. x_2, y_2 die im übergeordneten ElementView-Steuerelement an dessen, ebenfalls als Abhängigkeitseigenschaften ausgewiesenen Eigenschaften (*Startamplitude* bzw. *Endamplitude*) gebunden werden. Die Expositionierung von vier Abhängigkeitseigenschaften der `ElementShape`-Klasse ermöglicht die individuelle Reaktion auf Änderungen der verschiedenen Endpunkte(Markers). Dadurch müssen auch nicht immer alle Linien-Segmente (A,B,C in Abb. 5.10) bei jeder Änderung neu gezeichnet werden und hilft bei einer flüssigen grafischen Darstellung auch bei schnellen Mausbewegungen. Die Verwendung der `ElementView`-Klasse als Datenvorlage (DataTemplate) für ein Element des `ItemsControl`-Steuerelements in der BurstView-Klasse verlangt die Mehrfachdatenbindung wie in Quellcode 5.1 gezeigt.

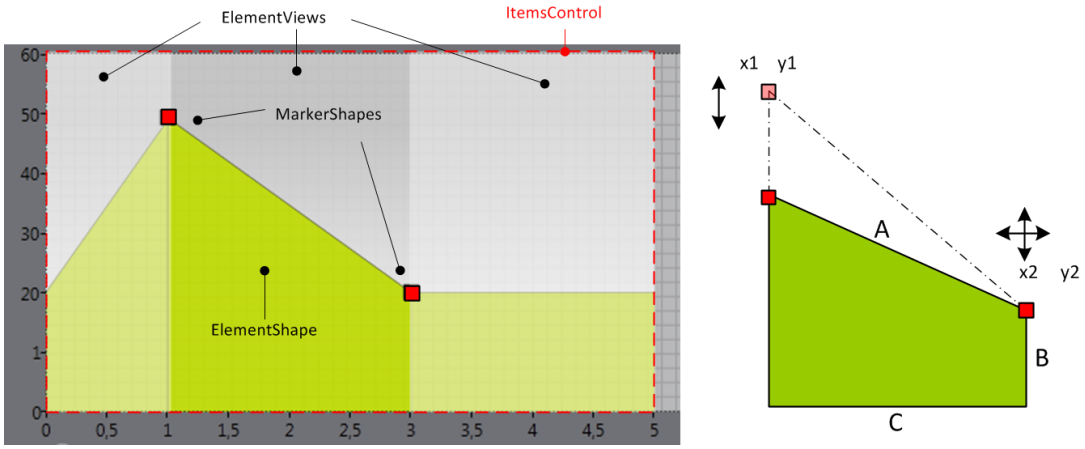


Abbildung 5.10: *Grafische Darstellung eines Burst(BurstView) mit ElementViews als Vorlagen für die Elemente des ItemsControls. Rechts die schematische Änderung eines einzelnen Stimulationselements und dessen Aufbau aus Linien-Segmenten.*

Die eigentliche Benutzerinteraktion erfolgt fast vollständig auf der Ebene der `BurstView`. Lediglich zur Lokalisierung beziehungsweise Bestimmung der zu ändernden Parameter werden die `MarkerShapes` auf Ebene der `ElementView` herangezogen. Durch die Aktivierung eines Markers kann auf die zu ändernde Eigenschaft geschlossen werden. Ist ein entsprechender Marker aktiv, werden alle Mauseingaben durch den selben Konverter rückgerechnet mit dem die Datenbindung stattfindet. Dadurch werden die Mauskoordinaten wieder in die physikalischen Werte übersetzt und so die entsprechenden Eigenschaften im ViewModel gesetzt. Zu betonen ist aber, dass nie ein Marker direkt durch Mauseingaben manipuliert wird sondern dies geschieht immer indirekt durch die Datenbindung, welche im Fall des Setztens der Eigenschaft eine Benachrichtigung auslöst, die wiederum den Marker entsprechend am Bildschirm positioniert.
Das Besondere bei Änderungen des Stimulationselements im Allgemeinen und bei der

grafischen Manipulation im Speziellen ist, dass eine Änderung der Stimulationszeit nur quantisiert erfolgen kann. Da die Impulsanzahl im Element ganzzahlig sein muss ist die Quantisierung abhängig von der Impulsperiode respektive der Stimulationsfrequenz. Die Dauer mus mindestens eine Impulsperiode lang sein. Die Validierung dieser Zustände wird durch das entsprechende ViewModel (`ElementViewModel`) durchgeführt.

```
1  public uint Duration
2  {
3   get { return m_element.Duration; }
4   set
5   {
6    uint nPulses = (uint)Math.Round(value / m_element.T));
7    uint newDuration = nPulses * m_element.T;
8    value = Math.Max(m_element.T, newDuration);
9
10   if (Element.Duration != value)
11   {
12    Element.Duration = value;
13    OnPropertyChanged("Duration");
14   }
15  }
16 }
```

Quellcode 5.2: *Codeausschnitt aus den ElementViewModel zur quantisierten Änderung der Elementedauer*

Damit die Änderung der Elementdauer quantisiert abläuft, wird zunächst aus der Frequenz bzw. Periodendauer und der aktuellen Elementdauer die Anzahl der Stimulationsimpulse berechnet (Zeile 6 Code 5.2) und auf einen ganzzahligen Wert gerundet. Dadurch ist nun bekannt wieviele Stimulationsimpulse, bei gleichbleibender Frequenz, im Element enthalten sein müssen. Um die exakte Elementedauer zu erhalten wird nun die errechnete Impulsanzahl mit der tatsächlichen Periodendauer multipliziert. Dies entspricht der Quantisierung (Zeile 7). In Zeile 8 wird die Bedingung geprüft ob die Dauer des Elements mindestens eine Impulsperiode lang ist. Nur wenn sich die neu berechnete Elementdaten von der ursprünglichen unterscheidet wird dem Datenobjekt (`Element`) die Eigenschaft neu zugewiesen und die Benachrichtigung für den Bindungsmechanismus abgesetzt. Diese Performancemaßnahme äußert sich vorallem bei großen Periodendauern, wo sich der quantisierte Wert, bezogen auf die Bildschirmpunkte, nur bei etwa allen 100 Bildpunkten ändert. So werden bei den Mausbewegungen des Benutzters nicht unnötige Benachrichtigungen abgesetzt, die zu einem stockenden Ablauf der Benutzerinteraktion führen kann.
Ein weiteres Implementierungsdetail im Zusammenhang mit der Darstellung eines Bursts ist die Verwendung von `DataTemplates` oder Datenvorlagen. Diese erlauben das flexible Wechseln der optischen Erscheinung von Datenelementen und deren Verhalten. In der BurstView kommen Datenvorlagen zum Einsatzt um die Interaktivität aus- oder einzublenden. Bei der reinen Anzeige eines Bursts wie sie etwa in den `BurstItems` zur Kanal-Darstellung vorkommen, wird zu Gunsten der Performance auf eine interaktive Veränderung verzichtet. Die BurstView definiert zu diesem Zweck zwei verschiedene `DataTemplates` im deklarativen XAML - Code(Ausschnitt in Quellcode 5.1). Des weiteren exponiert die BurstView-Klasse eine Abhängigkeitseigenschaft vom Type

`DataTemplate` und dem Namen `ListTemplate`. Das ItemsControl der BurstView bindet nun seine Eigenschaft des `ItemsTemplate` an das `ListTemplate` der BurstView. Durch Setzten des Interaktivitäts-Status der BurstView setzt die BurstView intern das entsprechende DataTemplate für diese Eigenschaft. Damit lässt sich auch zur Laufzeit die Interaktivität zu oder weg schalten.

5.2.2 Hardware-Erkennung

Unsere Hardware, der Stimulator wird über USB an den Rechner angebunden um Daten auszutauschen. Dabei verhält sich der Stimulator wie ein USB - Massenspeichermedium. Jedes USB-Gerät lässt sich durch seine USB-VendorID (VID) und ProductID (PID) identifizieren. Da die Softwarapplikation nicht wahllos alle, beim Betriebssystem angemeldeten Wechseldatenträger als Stimulatoren erkennt, erfolgt die Hardware-Identifikation in zwei Stufen. Erstens wird das VendorID / ProductID Paar des Stimulators mit allen gültigen VID/PID Paaren aus der Datenbank verglichen. Wird ein identisches Schlüsselpaar gefunden, wird im nächsten Schritt die Dateistruktur (Vgl. Abschn. 4.4.1[Ordnerstruktur] auf Seite 53) am Gerät überprüft. Findet die Software eine gültiges Dateisystem mit korrekter Struktur, wird das Gerät als Stimulator erkannt und ausgewiesen.

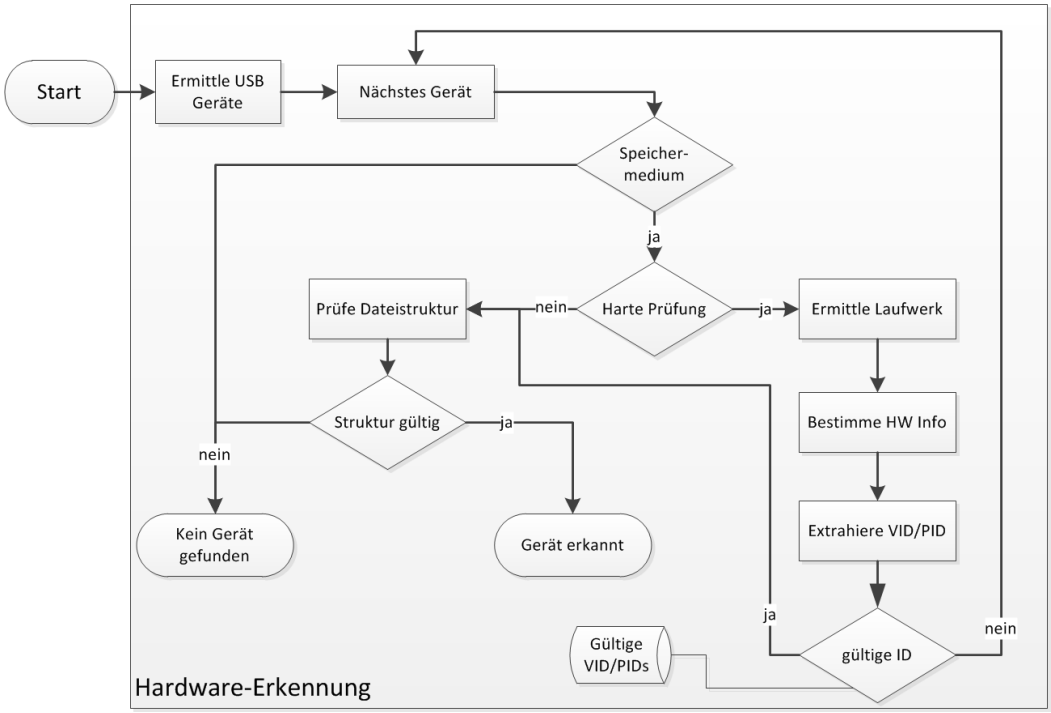


Abbildung 5.11: *Ablauf der Hardwareerkennung.*

Vor allem der Bereich der Bestimmung von den Hardwareinformationen ist technologisch interessant und im Projekt von besonderer Bedeutung und wird daher im Folgenden näher beschrieben. Zur Aquirierung der Hardwareinformation wird in unserem

Fall die Windows-Verwaltungsinstrumentation (Windows Management Instrumentation, WMI) verwendet. Dies ist ein Teil des Betriebssystems der den Zugriff auf Verwaltungsinformationen aus der Softwareapplikation heraus ermöglicht [45]. Durch die WMI - .NET-Technologie lassen sich die auf der ursprünglichen WMI aufbauende Klassen aus der gewohnten .NET -Programmierumgebung ansprechen.
Bei der Erkennung des Stimulators kommt die WMI gleich auf zwei Ebenen zum Einsatz. Als erstes wird ein sogennanter `ManagementEventWatcher` eingesetzt um die Hardwarekonfiguration zu beobachten (Quellcode 5.3).

```
1  public DeviceSelectorViewModel()
2  {
3   m_watcher = new ManagementEventWatcher("SELECT * FROM Win32_VolumeChangeEvent");
4   m_watcher.EventArrived += this.OnDeviceChanged;
5   m_watcher.Start();
6  }
7   ...
8  public void OnDeviceChanged(object sender, EventArrivedEventArgs args)
9  {
10  int eventtype = int.Parse(args.NewEvent.GetPropertyValue("EventType").ToString());
11  if (eventtype == 3 || eventtype == 2) //3...detached 2...arrived
12    // react on the event here...
13 }
```

Quellcode 5.3: *Zeigt die Initialisierung eines ManagementEvenWatchers um bei Änderungen der Speichermedien benachrichtigt zu werden und darauf zu reagieren*

Konkret wird eine Benachrichtigung angefordert, falls sich in der Systemkonfiguration die Speichermedien ändern. Dies geschieht im Konstruktor des `DeviceSelectorViewModel`. Auffallend ist die SQl-ähnliche Syntax der Zeichenkette die dem Konstruktor des Event-Watchers übergeben wird. So lassen sich durch WMI.NET relativ einfach sehr komplexe Szenarien abbilden. Durch die Implementierung des `EventHandlers` *OnDeviceChanged* (ab Zeile: 8 im Quellcode 5.3) kann so auf das Hinzufügen oder Entfernen von Wechseldatenträger im Speziellen oder ganz allgemein auf Änderungen der Speichermedien im System reagiert werden. Als erste Aufgabe übernimmt WMI in unserem Projekt also das Auslösen der eigentlichen Hardwareerkennung. Dort kommmt WMI zum zweiten mal auf einer etwas anderen Ebene zum Einsatz.

```
1  private string ExtractHwID(USBStimDevice device)
2  {
3   string drivename = device.Root.TrimEnd('\\'); //zb. G:\
4   try
5   {
6    ManagementObjectSearcher searcher =
7       new ManagementObjectSearcher("SELECT * FROM Win32_LogicalDisk WHERE (Name = '" + drivename +
            "')");
8
9    foreach (ManagementObject logicalDisk in searcher.Get())
10   {
11    foreach (ManagementObject diskPartition in logicalDisk.GetRelated("Win32_DiskPartition"))
12    {
13     foreach (ManagementObject drive in diskPartition.GetRelated("Win32_DiskDrive"))
14     {
15      foreach (ManagementObject entity in drive.GetRelated("Win32_PnPEntity"))
16      {
17       foreach (ManagementObject controller in entity.GetRelated("Win32_USBController"))
```

```
18          {
19            foreach (ManagementObject device in controller.GetRelated("Win32_PnPEntity"))
20            {
21              if (test["Service"].ToString() == "USBSTOR")
22                return device["DeviceID"].ToString();
23              ...
```

Quellcode 5.4: *Programmcode zum Auslesen der Vendor- und ProductID eines USB-Wechseldatenträger bei gegebenen Laufwerksbuchstaben mit Hilfe der WMI*

Quellcode 5.4 zeigt wie durch gezielte WMI-Anfragen mit Hilfe des `ManagementObjectSearcher` und den Zusammenhängen zwischen den einzelnen WMI-Klassen letztlich die Vendor- und ProductID eines Wechseldatenträgers ausgelesen werden kann. Durch geeignete Iterationen über die richtigen WMI-Klassen kann diese Information von der Eigenschaft `DeviceID` in der WMI-Klasse `Win32_PnPEntity` bezogen werden. Man beachte aber die Zeilen 15 bis 22 des obigen Quellcodes. Man würde erwarten das schon unmittelbar nach Zeile 15 die Information zur Verfügung stehen würde, es ist aber so, dass hier lediglich die Information bezüglich des Laufwerks sichtbar ist. Die Abfrage der `DeviceID` ergibt an diesem Punkt die PnP-Identifikationsnummer der Form

`USBSTOR\DISK&VEN_USB&PROD_DISK...`

Hier ist also lediglich zu erkennen dass es sich um einen USB-Speicher (USBSTOR) handelt, nicht aber welche HardwareID dieser hat. Dazu muss man noch eine Ebene tiefer gehen um über die Klasse des `Win32_USBController` wieder die entsprechende Win32_PnPEntität zu erhalten bei der die GeräteID dann bezüglich des USB-Controllers und in folgender Form vorliegt.

`USB\VID_xxxx&PID_yyyy\1234567890`

In der Zeichenkette die die Abfrage der `DeviceID` nun zurückliefert ist neben der Vendor- und ProductID auch die Seriennummer des USB-Gerätes vermerkt. Diese wird im weiteren Programmablauf mit den gültigen Vendor- und ProductID Paaren die in der Datenbank hinterlegt sind verglichen.

5.2.3 Aspekte der Visualisierung von Messsignalen

Die Visualisierung der Messsignalkurven wird prinzipiell mit dem WPF-Toolkit[2] realisiert. Die Bibiliothek bietet unter anderem Unterstützung zu sämtlichen Diagrammen. Ein wesentlicher Vorteil bei der Erstellung von Diagrammen mit Hilfe des Toolkits ist, dass optionale Achsenangaben und deren Werte automatisch an den Wertebereich der Kurven angepasst werden können.

Obwohl die Bibliothek grundlegnd die Datenbindung unterstützt musste aber im Rahmen dieser Antwort einige Erweiterungen und Anpassungen durchgeführt werden um

[2]Das WPF-Toolkit ist frei verfügbar unter http://wpf.codeplex.com/releases/view/40535 und wurde im Release vom Februar 2010 verwendet.

eine dem MVVM-Pattern gerecht werdende Datenbindung zu realisieren. Auch für die interaktive Aus- und Einblendung einzelener Kurven bietet das Toolkit keine eingebaute Unterstützung. Für genau diesen beiden Aspekte wurde eine Erweiterungs-Klasse erstellt die diese Aufgaben übernimmt.
Zur Erstellung von Diagrammen wird die Klasse `Chart` des WPF-Toolkits herangezogen. Diese Klasse kann mehrere `ChartSeries` Objekte aufnehem, die die eigentlichen Daten anzeigen. Diese Objekte können durch ihre Abhängigkeitseigenschaft `ItemsSource` an eine Liste mit Datenpunkten binden. In unserem Fall ist kann aber die Anzahl der anzuzeigenden Kurven zur Laufzeit variieren. Dadurch kann nicht von vorne herein ein `ChartSeries` Objekt, welches die Datenbindung gemäß dem MVVM-Entwurfsmuster umsetzt, deklariert werden. Die `Chart`-Klasse wiederum bietet keine Möglichkeiten an mehrere Datensätzte zu Binden. Durch die Implementierung der ChartHelper-Klasse wird unter anderem dies ermöglicht. Dazu wird ein `DependencyProperty` vom Typ `IEnumerable` im ChartHelper angelegt (Code 5.5).

```
1 public class ChartHelper
2 {
3  public static readonly DependencyProperty SeriesSourceProperty =
4      DependencyProperty.RegisterAttached("SeriesSource",
5                                  typeof(IEnumerable),
6                                  typeof(ChartHelper),
7                                  new PropertyMetadata(SeriesSourceChanged));
```

Quellcode 5.5: *Ausschnitt aus der ChartHelper-Klasse; Definition der Abhängigkeitseigenschaft an die mehrere Datensätze gebunden werden können*

Dadurch kann deklarativ im XAML-Code an die Kollektion der Kurven die zur Anzeige gebracht werden sollen, gebunden werden (Code 5.6). In der Klasse ChartHelper wird im `CallBackHandler` (SeriesSourceChanged) der Abhängigkeitseigenschaft `SeriesSourceProperty` bei jeder Änderung der zugrundeliegenden Daten, für jeden Eintrag eine neue Chart-Serie erstellt.

```
1 <chartingToolkit:Chart Title="U Curves"
2  helper:ChartHelper.DependentValueBinding="Y"
3  helper:ChartHelper.IndependentValueBinding="X"
4  helper:ChartHelper.SeriesSource="{Binding UCurves}"
5  Style="{DynamicResource OverviewChartStyle}"
6  Palette="{DynamicResource ColorPalette}"
7  />
```

Quellcode 5.6: *Die ChartHelper-Erweiterung ermöglicht das Binden an eine Kollektion von Kurvendaten in XAML*

Im zugrundeliegenden ViewModel (MeasurementViewModel) liegen die anzuzeigenden Daten als Liste von `CurveViewModels` vor. Ein `CurveViewModel` ist im wesentlichen eine Liste aus Datenpunkten die sowohl als Basis für die Visualisierung als auch für den Export der Kurvendaten verwendet wird. Die vorgestellte ChartHelper-Klasse erlaubt wie im Quellcodeausschnitt 5.6 gezeigt, die Liste der `CurveViewModels` an ein `Chart`-Objekt zu binden.

KAPITEL 6

Zusammenfassung

Um die Erfolge und Auswirkungen des Trainings mit funktioneller Elektrostimulation bei älteren Patienten beurteilen zu können ist es notwendig zu wissen, wie lange die Patienten das Training anwenden. Da die Anwendung des in dieser Arbeit vorgestellten Systems in Eigenregie und zu Hause stattfinden kann, muss also eine Überwachung der Trainingsdauer durch das Stimulationsgerät stattfinden. Durch verschiedene Stimulationsmuster und Impulsparameter kann eine optimale Kombination für die Anwendung der Elektrostimulation bei älteren Menschen gefunden werden. Das System muss daher in der Lage sein, sehr rasch und einfach verschiedene Stimulationsmuster zu generieren und deren Parameter möglichst einfach zu verändern. Durch die Möglichkeit den Strom und die Spannung der Stimulationsimpulse zu messen kann die Elektrodenimpedanz berechnet werden. Dies lässt Rückschlüsse über die korrekte Positionierung der angelegten Stimulationselektroden zu. Dieser Mechanismus kann auch dafür verwendet werden um bei falsch angelegten Elektroden das Training aus Sicherheitsgründen abzubrechen und stellt so einen wesentlichen Beitrag zur sicheren Anwendung des Stimulationssystems dar.

In dieser Diplomarbeit wird das Design und die Umsetztung einer Softwareanwendung dargestellt, die die oben genannten Aspekte erfüllt. Dafür werden zunächst die theoretischen Hintergründe der Elektrostimulation beleuchtet. Es wird auf die elektropysiologischen Grundlagen eingegegangen und es wird erklärt, wie und warum das Auslösen von Muskelkontraktionen mit elektrischen Reizen überhaupt möglich ist. Um einen Einblickk in die Funktionsweise der Muskulatur zu erhalten, wird gezeigt wie ein Muskel aufgebaut ist und aus welchen Strukturen er besteht. Der Unterschied zwischen direkter Stimulation eines Muskels und der Reizung seines Muskelnervs wird herausgearbeitet und es wird beschrieben warum die Nervenfaser einfacher zu erregen ist, als der Muskel selbst. Die wichtigsten Parameter und deren Auswirkung für die Anwendung der Elektrostimulation werden vorgestellt. Diese Parameter können letztlich durch die Softwareapplikation verändert werden um das Stimulationsgerät mit unterschiedlichsten

Stimulationsmustern zu programmieren.
Ein weiterer Punkt behandelt das Thema des Muskeltrainings und erklärt die Veränderung des Muskelaufbaus in Folge des Trainingseffekts. Insbesondere der Zusammenhang des Muskeltrainings mit der funktionellen Elektrostimulation wird aufgeführt und der Nutzen der Elektrostimulation zur Behandlung von muskulären Dysfunktionen wird diskutiert. Die Patientenzielgruppe für welche das hier vorgestellte System vorwiegend zum Einsatz kommen wird, sind Menschen im fortgeschrittenem Lebensalter. Daher wird kurz auf die hauptsächlich im Alter auftretende Sarkopenie eingegangen und anhand von Studien wird die Berechtigung der neuromuskulären Stimulation für diese Patientengruppe untermauert.

Die im Rahmen dieses Projekts verwendeten Strategien des neuromuskulären Monitorings werden kurz erleutert um die Problemstellung in der Software verstehen und lösen zu können. Die zur Umsetzung der Softwarelösung verwendeten Technologie wie das .NET-Framework und die Windows Presentation Foundation werden eingehend studiert. Besonderer Fokus wird hierbei auf die Markup-Sprache XAML gelegt, die es erlaubt graphische Benutzeroberflächen unabhängig von der Programmlogik zu designen. Dabei werden die wichtigsten Spracherweiterungen vorgestellt und die Konzepte der Datenbindung behandelt. Das Software-Entwurfsmuster Model-View-ViewModel wir kurz erklärt und es wird gezeigt wie mit Hilfe dieser Softwarearchitektur eine saubere Umsetzung und Implementierung des Programmcodes stattfinden kann. Des weiteren wird der Objekt- relationale Mapper des .NET Framworks vorgestellt mit dessen Hilfe die Anbindung an eine MS-SQLCE-Datenbank erfolgt. Die wesentlichen Strukturen der Daten im Zusammenhang mit dem Datentransfer vom und zum Stimulationsgerät werden eingehend erklärt. Es wird der Ablauf des Datenaustauschs zwischen Softwareanwendung und Stimulationshardware genau beschrieben und durch die vorgestellten Datenaustauschprotokolle wird gezeigt wie eine funktionierende Umsetzung und Problemlösung aussehen kann. Auch die Visualisierung der Stimulationsergebnisse wird kurz behandelt und es wird darauf eingegangen wie mit Hilfe der Windows Presentation Foundation, XAML und der Programmiersprache C#, eine optisch ansprechende Benutzerschnittstelle geschaffen werden kann. Das interaktive, graphische Konzept zur Manipulation der Stimulationsparameter wird ebenfalls kurz erleutert. Abschließend werden die wichtigsten Funktionalitäten der Softwareanwendung in einem Benutzerhandbuch zusammengefasst und deren Schritte zur Realisierung durch die Anwendung erklärt.

ANHANG A

Benutzerhandbuch

Dieses Kapitel sollte einem Leihen als Hilfestellung im Umgang mit der Softwareapplikation dienen. Es wird gezeigt wie die Applikation anzuwenden ist um die wichtigsten Funktionalitäten durchzuführen. Es wird darauf eingegangen wo der Benutzer welche Informationen findet und wie die Abläufe beziehnungsweise Arbeitsschritte aussehen.
Der aufmerksame Leser sollte nach diesem Kapitel in der Lage sein die Anwendung effizient zu nützen und alle relevanten Aufgaben die mit der Software zu lösen sind, umzusetzten.
Folgende Themen werden behandelt:

Tabelle A.1: *Inhaltsverzeichnis des Benutzerhandbuches*

A.1 Übersicht

Um einen Überblick über die Benutzeroberfläche zu erhalten, werden an dieser Stelle die vier großen Teilbereiche der Software etwas näher beleuchtet.

(1) Der Startbildschirm

(2) Die Oberfläche zum Management der Compliance-Daten

(3) Die Schnittstelle für das Auslesen des Stimulators und

(4) Die Oberfläche zur Personenverwaltung

A.1.1 Initialarchitektur der graphischen Oberfäche

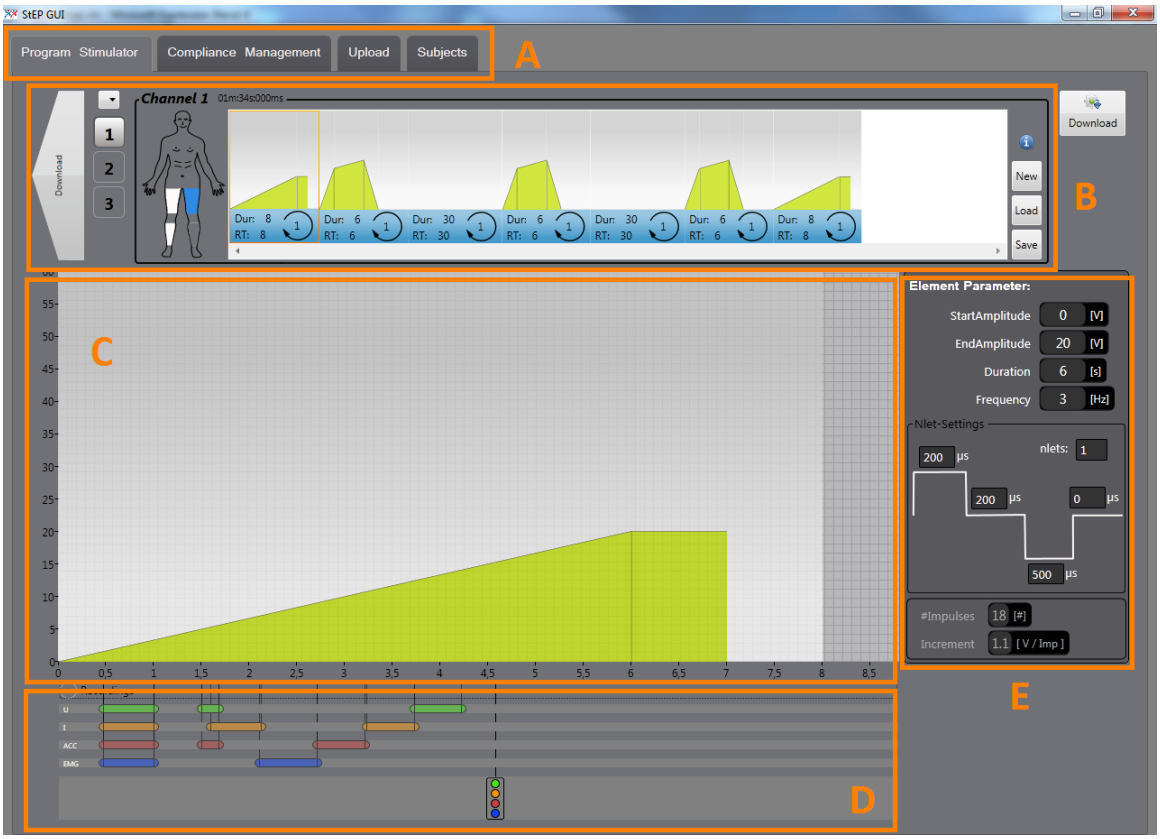


Abbildung A.1: *Hauptbildschirm der Applikation.*

A Im Navigationsbereich wird zwischen den funktional getrennten Teilbereichen hin und her gewechselt. Die Navigation ist in der Anwendung immer sichtbar, links oben positioniert.

B Dieser Bereich zeigt die aktuelle Kanalinformation an. Ganz links kann das aktuelle Stimulationsmuster von allen Kanälen auf den Stimulator übertragen werden(Siehe A.2.2). Wahlweise kann aber auch eine zuvor übertragene Informationn zur weiteren Bearbeitung geladen werden. Des weiteren kann die Anzahl der Kanäle in diesem Bereich ausgewählt werden(A.2.1).

C Dieser Bereich dient der grafischen Manipulation der Stimulationsparameter(A.2.1).

D Um die Punkte der Messwerterfassung festzulegen können in diesem Bereich grafisch Markierungen festgelegt werden(A.2.1)

E Hier kann die Hüllkurve des Stimulations-Bursts manuell geändert werden. Auch die Stimulationsfrequenz und Impulszeiten werden hier manipuliert.

A.1.2 Verwaltung der Anwendungsdaten

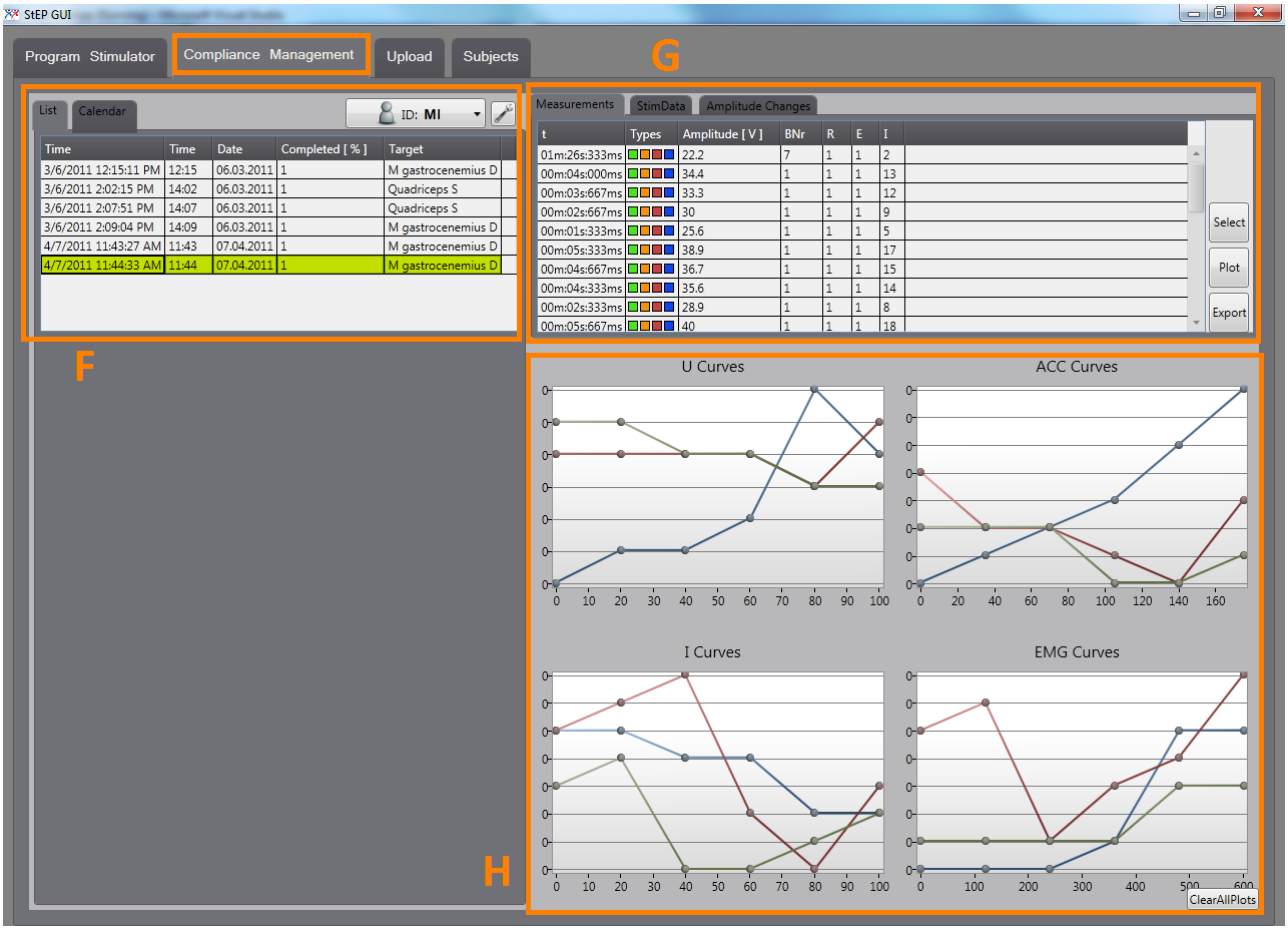


Abbildung A.2: *Übersich der Compliance Verwaltung*

F Dieser Bereich zeigt alle eingelesenen Stimulationen zu jeweils einem ausgewählten Patienten. Die Ansicht kann wahlweise in Listenform oder als zeitliche Übersicht in einer Kalenderansicht betrachtet werden. Für nähere Informationen siehe Abschnitt A.2.4.

G Dieser Teilbereich erlaubt die Anzeige der getätigten Messungen zu einer ausgewählten Stimulation (A.2.4). Hier kann auch die Amplitudenänderung zur Anzeige gebracht werden (Siehe A.2.4) oder es kan das verwendete Stimulationsprotokoll betrachtet werden(Mehr dazu im Abschnitt A.2.4).

H Hier werden die Messsignalkurven angezeigt und können im Detail betrachtet werden.

A.1.3 Upload und Patientenverwaltung

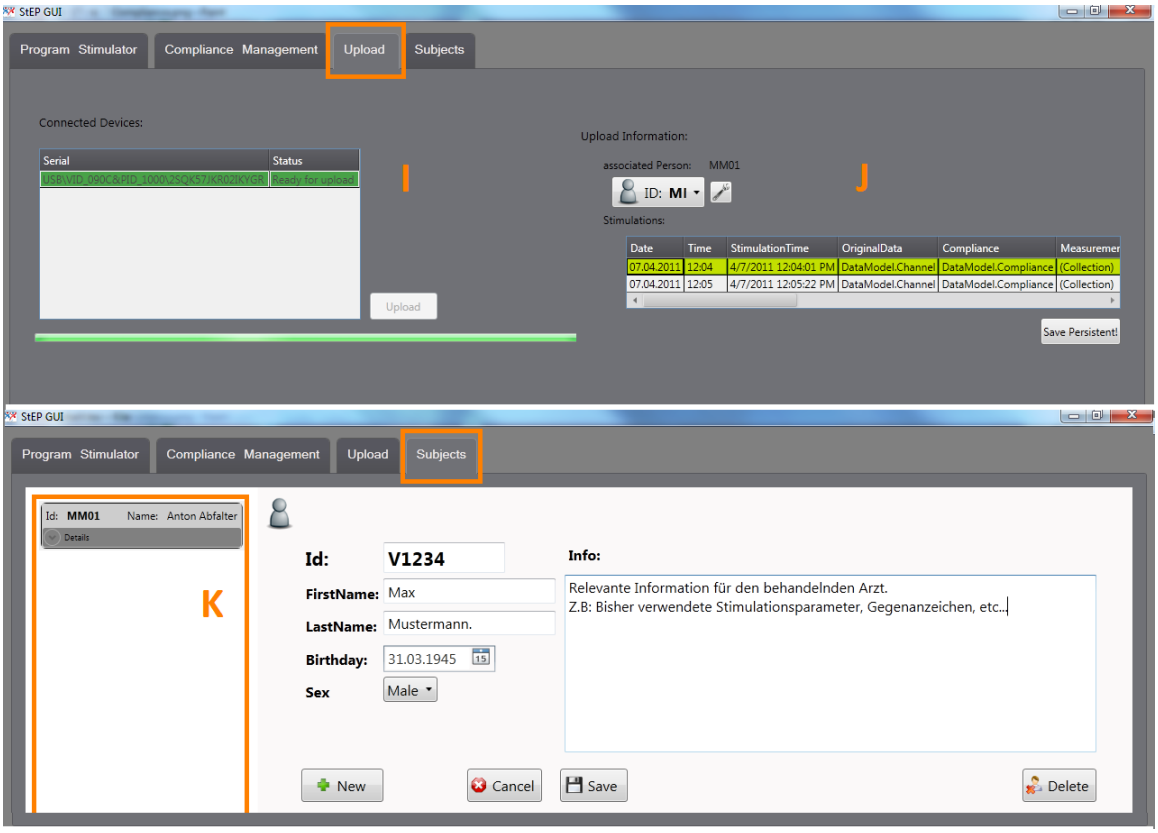


Abbildung A.3: *Zeigt zwei Teilbereiche der Applikation im oberen Bereich ist die Oberfläche zu sehen durch die das Auslesen eines Stimulators erfolgt, und im unteren Bereich ist die Benutzerschnittstelle zur Patientenverwaltung ersichtlich.*

I Hier werden alle angeschlossenen Stimulatoren gelistet. Die farbliche Hinterlegung zeigt an in welchem Zustand sich ein Stimulationsgerät befindet. Nach erfolgter Datenübertragung vom Stimulator werden in Ansicht J die vom Patient erledigten Stimulationen angezeigt.

J Nach erfolgtem Upload vom Stimulator sind hier alle am Stimulator gespeicherten Stimulations-Sitzungen sichtbar. Diese Ansicht erlaubt auch die Änderung der Zuordnung zu einem anderen Patienten als am Stimulator vermerkt. Weiters können überflüssige Ergebnisse gelöscht werden um nicht in die Datenbank aufgenommen zu werden. Schließlich wird hier eine persistente Sicherung der Stimulationen veranlasst. Dies veranlasst die Software zur Speicherung der Daten in die Datenbank und löscht die Daten am Stimulator für dessen weitere Verwendung.

K Stellt die Patientenliste in der Oberfläche der Patientenverwaltung dar. Daraus wird ein Subjekt ausgewählt welches verändert werden soll. In dieser Oberfläche können auch neue Subjekte erstellt oder gelöscht werden.

A.2 Arbeitsabläufe

A.2.1 Zusammenstellung der Stimulationsmuster

Die Applikation startet immer mit dem zuletzt bearbeiteten Stimulationsmuster. Beim ersten Start ist noch kein Stimulationsmuster zur Verfügung und daher sind die Kanalinformationen leer. Der erste Schritt besteht nun darin die Anzahl der Kanäle, für die Stimulationsparameter vorliegen sollen, einzustellen.

Auswahl der Kanalanzahl

Standardmäßig stehen zwei Kanäle für die Einstellung der Stimulationsparameter zur Verfügung. Will man die Die Anzahl der Kanäle ändern, steht dafür ein "DropDown-Menü" (Vgl. (1) in Abb. A.4) am Startbildschirm zur Verfügung.

Dadurch kann die Anzahl der Kanäle auf maximal 9 und minimal 1 Kanal verändert werden. Sind für einen Kanal Stimulationsparameter eingestellt, werden diese gespeichert und auch nach der Umstellung der Kanalanzahl wieder angezeigt. Wird ein Kanal nach der Veränderung der Kanalanzahl nicht mehr angezeigt, z.B wenn von drei Kanälen auf zwei reduziert wird, würde Kanal 3 aus der Liste verschwinden. So werden in diesem Fall ebenfalls die Informationen zu diesem Kanal gespeichert, und bei der nächsten Umstellung nach der der Kanal wieder angezeigt wird werden auch diese Informationen wieder geladen. So wird garantiert, dass keine eingestellten Informationen verloren gehen außer wenn sie aktiv überschrieben werden. Steht die Anzahl der Kanäle einmal fest, kann noch der aktive Kanal ausgewählt werden. Dafür wird einfach die Nummer des entsprechenden Kanals in der Kanalliste (Vgl. (2) Abb. A.4) markiert. Nur für den aktiven Kanal gilt die angezeigte Kanalinformation.

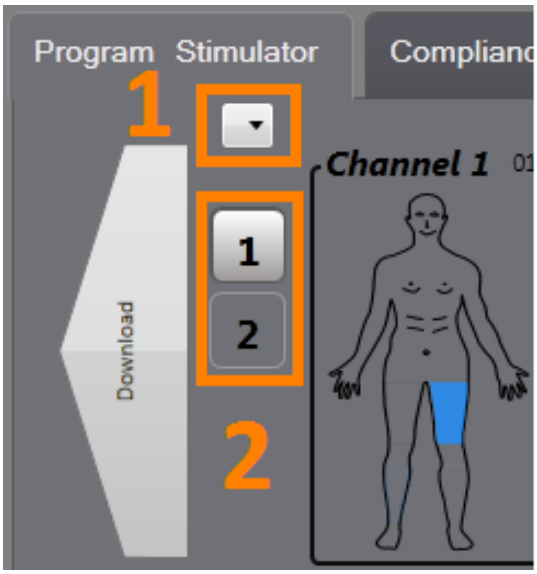


Abbildung A.4: *Einstellung der Kanalanzahl und Auswahl des aktiven Kanals*

Speichern und Laden von Stimulationsdaten

Um einmal zurechtgelegte Stimulationsmuster nicht jedesmal wieder mühsam zusammen stellen zu müssen, bietet die Software die Möglichkeit die angelegten Stimulationsmuster zu speichern. Man kann dabei wählen ob die gesammte Kanalinformation oder der aktuell markierte Burst gespeichert werden soll. Genauso verhält es sich beim Laden der Stimulationsinformation. Zu beachten ist, dass beim Laden der kompletten Kanalinformtion, die aktuelle Information überschrieben wird, während beim Laden eines Bursts dieser als letzter zu den bestehenden Bursts im Kanal angereiht wird. Auch beim erstellen eines neuen Bursts, wird ein Standard-Burst hinten angefügt. Das Laden bzw. Speichern erfolgt mit Hilfer der Burst-Dateien (*.burst) für einen einzelnen Burst und den Kanal-Dateien (*.channel) für die Kanalinformation. Dabei handelt es sich technisch gesehen um XML-Dateien. Diese können auch verwendet werden um Stimulationsdaten unter mehreren Anwendern auszutauschen.

Burstreihenfolge

Die Reihenfolge der Bursts innerhalb eines Kanals lässt sich recht einfach mit Drag and Drop verändern. Dafür wird ein Burst markiert und mit gedrücktem linken Mausknopf an die gewünschte Stelle verschoben. Soll ein Burst gelöscht werden so wird dieser zuerst markiert und per Tastatureingabe <Entf> gelöscht. Man bemerke dass das Löschen nicht rückgängig gemacht werden kann.

Verändern der Burstinformation

Um einen Burst verändern zu können muss dieser in der Ansicht der Kanalinformation (Vgl. (B) Abb. A.1) markiert werden. Der markierte Burst wird dann im BurstEditor sichtbar (Vgl. (C) Abb. A.1). Dort können nun die im Burst enthaltenen Elemente markiert und entweder grafisch oder manuell bearbeitet werden.

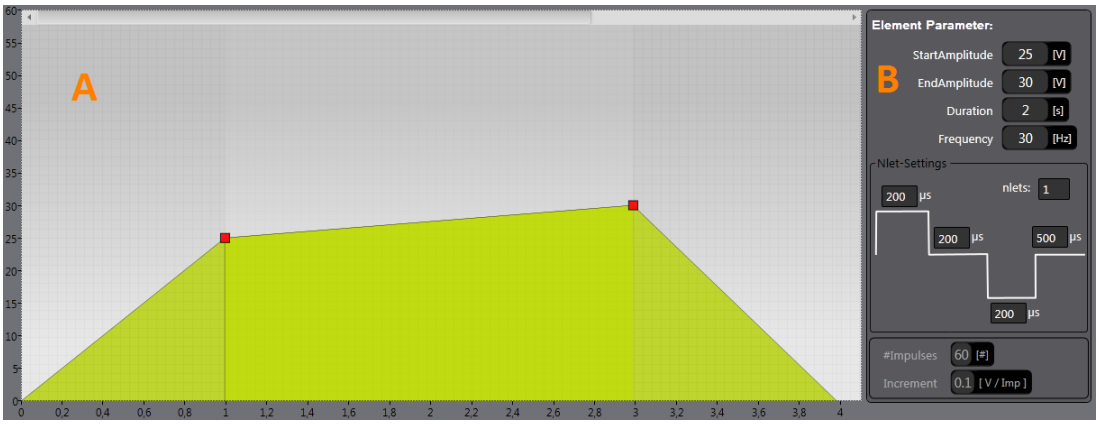


Abbildung A.5: *BurstEditor zur grafischen Änderung der Hüllkurve A) und zur manuellen Manipulation B) der Burstdaten und der Stimulationsparameter*

Die Daten des im Bursteditor markierten Stimulationselement werden rechts (Vgl. B) in Abb A.5) angezeigt. Standardmäßig ist immer das jeweils erste Element des aktuellen Burst markiert. Ein Element kann durch klicken im BurstEditor markiert werden. Ein markiertes Element zeigt seine beiden Endpunkte als rote Rechtecke an. Durch verschieben der Rechtecke kann bei beiden Endpunkten die Amplitude verändert werden. Zusätzlich kann beim rechten Endpunkt auch die Zeit respektive die Dauer des Elements verändert werden.
Soll ein Element aus dem Burst gelöscht oder eingefügt werden, wird das Kontextmenü des Bursteditors mit der rechten Maustaste aufgerufen.

Festlegen der Messpunkte

Zum Festlegen der Zeitpunkte an denen die Messsignale aufgenommen werden sollen, steht das Messwertaufnahme-Fenster (Vgl. (B) in Abb. A.6) zur Verfügung. Dabei kann für jeden markierten Burst separat festgelegt werden wann welches Signal erfasst werden soll.

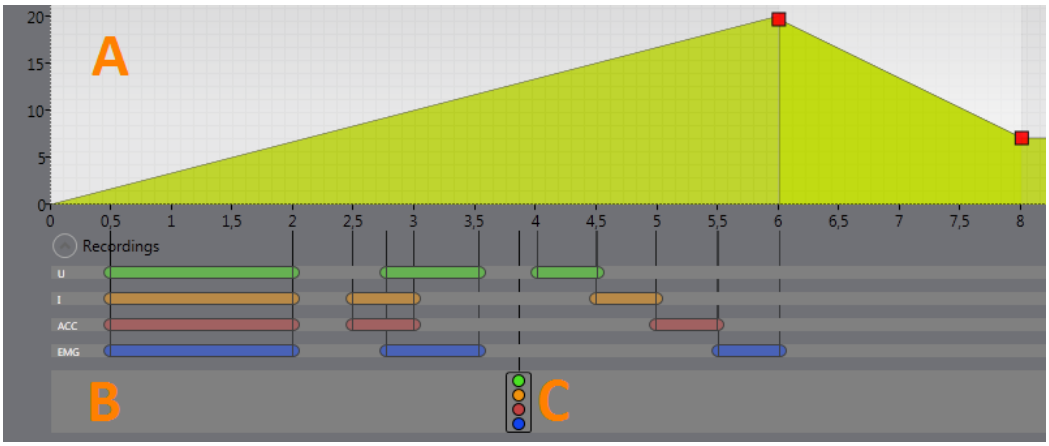


Abbildung A.6: *A) Bursteditor B) Festlegen der Messwertaufnahme C) Signalauswahlschieber*

Ein Punkt zur Messwertaufnahme kann dabei auf zwei verschiedene Arten gesetzt werden. Einmal durch ziehen der Maus (mit gedrückter linker Maustaste) direkt im Slot des Signals, oder durch den Signal-Auswahl-Schieber (Vgl. (C) in Abb. A.6). Durch den Signal-Auswahl-Schieber lassen sich die exakten Zeitpunkte für mehrere Signale auf einmal festlegen. Dabei funktioniert das Setzten der Messpunkte ebenso mit gedrückter linker Maustaste. Für die Auswahl, für welches Signal ein Messpunkt gesetzt werden sollte, kann im Signal-Auswahl-Schieber durch drücken der rechten Maustaste auf die entsprechende Signalfarbe, die Markierung zu oder abgeschalten werden.

A.2.2 Übertragung der Stimulationsdaten auf den Stimulator

Nachdem die Stimulationsdaten wie gewünscht eingestellt und angepasst wurden, können diese an einen Stimulator übermittelt werden. Dazu wird der `Download-Button` in der

Ansicht der Kanalinformation (Vgl. Abb. A.4) verwendet. Dieser öffnet ein Dialogfenster wie in Abbildung A.7 ersichtlich.

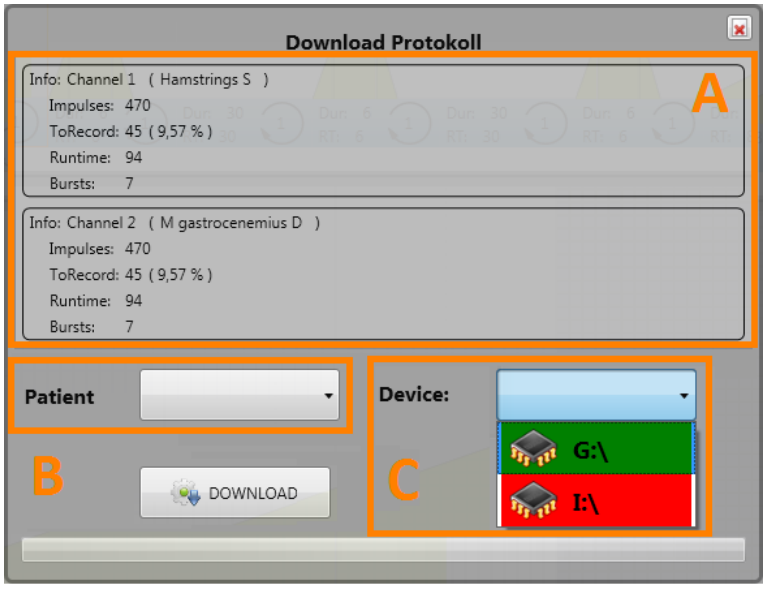


Abbildung A.7: *A) Übersicht der Stimulationsdaten B) Auswahl des Patienten C) Auswahl der Hardware*

Dort findet sich noch einmal eine Übersicht (Vgl. (A) in Abb. A.7) der Stimulationsdaten die übertragen werden sollen. Daraus ist zu erkennen (a) Für wieviele Kanäle Stimulationsdaten übertragen werden sollen (b) Die Anzahl der Stimulationsimpulse sowie der zu stimulierende Muskel pro Kanal (c) Wieviele Impulse gemessen werden sollen, mit der Angabe der prozentuellen Angabe bezogen auf die gesamte Impulsanzahl (d) Die Laufzeit für jeden Kanal in Sekunden, sowie (e) Die Anzahl der Bursts pro Kanal.

In diesem Dialogfenster wird auch die Zuordnung zum Patienten getätigt (Vgl. (B) in Abb. A.7). Sollten bei der Patientenzuordnung Fehler passiert sein, können diese später beim Auslesen der Ergebnisse aus dem Stimulator immer noch korregiert werden. (Mehr dazu in Abbschn. A.2.2 auf Seite 91).
Nun muss noch ein Stimulator ausgewählt werden an den die Daten übermittelt werden sollen. Im Dialogfenster steht dabei eine Liste aller mit dem Rechner verbundenen Stimulatoren zur Verfügung (Vgl. (C) in Abb. A.7). Die farbliche Hinterlegung zeigt an ob ein Stimulationsgerät bereit ist, Stimulationsdaten zu übernehmen (grün), oder ob sich auf dem Gerät schon Protokolle befinden(orange). Ist das Gerät rot hinterlegt, bedeutet dies dass sich am Stimulator noch nicht gespeicherte Resultate befinden. In diesem Falle müssen zunächst die Resultate vom Stimulator geladen werden, um diesen wieder für neue Stimulationsparameter freizugeben.

A.2.3 Auslesen des Stimulators

Zum Auslesen eines Stimulators muss zur Upload-Ansicht gewechselt werden (Vgl. (A) in Abb. A.1). Hier wird zunächst eine Liste mit allen verbundenen Stimulatoren gezeigt. Wieder sind die Geräte farblich hinterlegt. Rot bedeutet in diesem Zusammenhang dass keine Resultate am Stimulator existieren und grün werden jene Geräte gekennzeichnet, die Ergebnisse von Stimulationssitzungen besitzten. Man muss zunächst ein Gerät aus dieser Liste markieren (Vgl. (A) in Abb. A.8) um den Upload-Button freizugeben.

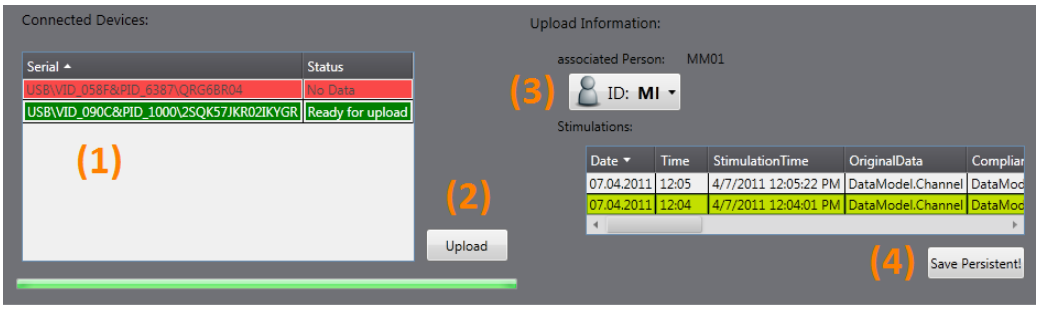


Abbildung A.8: *Vorgehensweise beim Auslesen eines Stimulators*

Durch das Drücken des Upload-Knopfes werden die Resultate vom Gerät gelesen und rechts in der Ergebnissliste angezeigt. Der nächste Schritt besteht nun darin die Zuordnung zum Patienten zu kontrollieren. Wurde diese beim Übertragen auf den Stimulator versehentlich falsch eingegeben, kann dies nun korrigiert werden. Durch das persistente Speichern werden die Daten vom Stimulator gelöscht und in die Datenbank geschrieben. Der Stimulator ist nun wieder bereit neue Stimulationsprotokolle aufzunehmen.

A.2.4 Verwaltung der Compliance-Daten

Um die Compliance-Daten zu verwalten, wird in das Compliance-Management-Menü gewechselt (Vgl. (A) in Abb. A.1). In dieser Ansicht können personenbezogene Ergebnisse von Stimulationen gesichtet und verglichen werden. Auch der Export der Daten wird aus dieser Ansicht heraus getätigt.

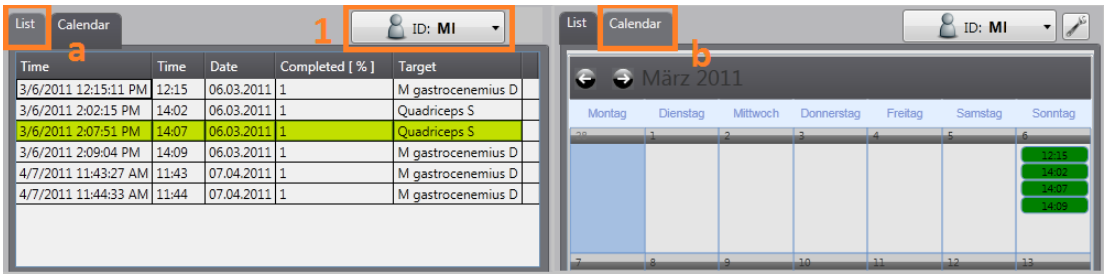


Abbildung A.9: *Ansicht der personenbezogenen Stimulationen. Wählweise als Liste (links) oder rechts als Kalenderansicht.*

Auswahl der Stimulationen

Man wählt zunächst einen Patienten aus der Patienten Liste (Vgl. (1) in Abb. A.9). Daraufhin werden alle jemals von diesem Patienten getätigten und gespeicherten Stimulationen angezeigt. Wahlweise können die Stimulationen in Listenform (Vgl. (a) in Abb. A.9) oder als zeitliche Übersicht in einer Kalenderansicht (Vgl. (b) in Abb. A.9) dargestellt werden. In der Kalenderansicht werden die Stimulationen mit weniger Informationen angegeben. Hier sind lediglich Datum und Uhrzeit der Stimulation ersichtlich. In der Listenansicht hingegen stehen alle Informationen zu den getätigten Stimulationen zur Verfügung. Allerdings dind in der Kalenderansicht die Stimulationen farblich hinterlegt. Dies zeigt an, ob eine Stimulation vollständig, also laut Trainingsprotokoll beendet wurde (grüne Hinterlegung), oder nicht (rote Hinterlegung). In beiden Ansichten lassen sich einzelne Stimulationen markieren um mehr Details zu einer Stimulation zu erfahren.

Details zu den Stimulationen

In der Detailansicht, ersichtlich in Abbildung A.10, können drei unterschiedliche Ebenen der Stimulationsergebnisse betrachtet werden. (a) Die Ergebnisse der Messwertaufnhame (b) das Stimulationsprotokoll und (c) die Änderungen der Amplitude durch den Benutzer während einer Stimulation.

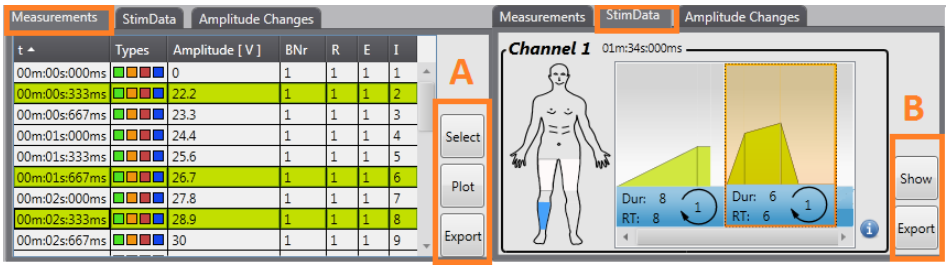


Abbildung A.10: *Links die Übersich über alle Messpunkte und rechts das verwendete Stimulationsprotokoll.*

Anzeigen der Messkurven

In den Details zur Messung werden alle vermessenen Impulse gelistet (Abb. A.10). Jeder Impuls wird dabei mit dem Zeitpunkt seines Auftretens ab Beginn der Stimulation, den enthaltenen Messkurventypen, seiner tatsächlichen Amplitude und der eindeutigen Identifikationsangabe (BurstNr, Wiederholung, ElementNr u. ImpulsNr) angegeben.
Über die Stuerleiste auf der rechten Seite der Detailansicht (Vgl. (A) in Abb. A.10) lassen sich alle Messungen markieren (<Select>), die selektieren Messungen als Kurven anzeigen <Plot> oder die Daten zu den Messungen in Dateien für die weitere Analyse exportieren (<Export>). Durch <Strg + Mausklick> lassen sich einzelne Impulse selektieren.

Ansicht des verwendeten Protokolls

Öffnet man die Karteikarte `StimData`, so kann das bei der aktuell selektierten Stimulation verwendete Stimulationsprotokoll betrachtet werden. Daraus ist das Stimulationsmuster, der zu stimulierender Muskel, die Kanalnummer und die Stimulationsdauer der Trainingseinheit ersichtlich. In dieser Ansicht kann das Stimulationsprotokoll allerdings nicht verändert werden. Dafür steht auf der rechten Seite der Ansicht eine Steuerleiste zur Verfügung CompDetail (Vgl. (B) in Abb. A.10). Durch Drücken von <Show> wird das Protokoll im `BurstEditor` angezeigt. Um die Bearbeitung zu starten muss im Navigationsmenü auf die Karteikarte `ProgramStimulator` zurückgewechselt werden.
Der Export <Export> des Stimulationsprotokoll veranlasst die Software ein “Komma separaed value file” (CSV) anzulegen .

Quellcodeverzeichnis

Literaturverzeichnis

[1] T. Berger, *Brain-Computer Interfaces, An international assessment of research and development trends.* 978-90-481-7960-2, Springer, 1st edition. softcover version of original hardcover edition 2008 ed., 2008.

[2] B. Lüderitz, "milestones und details zur geschichte der kardialen elektrophysiologie," in *Rationale Arrhythmiebehandlung* (T. Lewalter, ed.), pp. 1–10, Steinkopff, 2006.

[3] H. Reichert, *Neurobiologie.* Thieme, 2000.

[4] J. Bille and W. Schlegel, *Medizinische Physik 1: Grundlagen.* Springer, 1999.

[5] D. Silverthorn, *Physiologie.* Pearson Studium, Pearson Studium, 2009.

[6] G. Vrbová, O. Hudlicka, K. S. Centofanti, and G. Vrbová, "Plasticity of the mammalian motor unit," in *Application of Muscle/Nerve Stimulation in Health and Disease* (G. Stienen, ed.), vol. 4 of *Advances in Muscle Research*, pp. 1–22, Springer Netherlands, 2008.

[7] A. C. Company, *Systems & structures: the world's best anatomical charts.* The World's Best Anatomical Chart Series, Anatomical Chart Co., 2005.

[8] P. H. Peckham, "Functional electrical stimulation for neuromuscular applications," *Biomedical Engineering*, vol. 7, pp. 327–360, 2005.

[9] P. A. Grandjean and J. T. Mortimer, "Recruitment properties of monopolar and bipolar epimysial electrodes.," *Annals of Biomedical Engineering*, vol. 14, no. 1, pp. 53–66, 1986.

[10] C. Schönle, *Rehabilitation.* Praxiswissen Halte- und Bewegungsorgane, Thieme, 2004.

[11] H. Marèes, *Sportphysiologie.* Sport & Buch Strauß, Köln, 2002.

[12] E. Speckmann, *Physiologie.* Urban & Fischer bei Elsev, 2008.

[13] T. Paillard, "Combined application of neuromuscular electrical stimulation and voluntary muscular contractions," *Sports Medicine*, vol. 38, pp. 161–177(17), 2008.

[14] O. A. Zhou S., "Effects of unilateral voluntary and electromyostimulation training on muscular strength on the contralateral limb," *Hong Kong Journal of Sports Medicine and Sports Science*, vol. XIV, pp. 1–11, 2002.

[15] L. Bax, F. Staes, and A. Verhagen, *Does Neuromuscular Electrical Stimulation Strengthen the Quadriceps Femoris?: A Systematic Review of Randomised Controlled Trials*, vol. 35. 2005.

[16] T. Münzer, "Sarkopenie im alter," *Swiss Medical Forum*, vol. 10, no. 1, pp. 188–190, 2010.

[17] Wikipedia, "Sarkopenie — Wikipedia." http://de.wikipedia.org/wiki/Sarkopenie, März 2011. [Online; letzter Besuch: 18-April-2011].

[18] J. M. Bauer, R. Wirth, D. Volkert, H. Werner, C. C. Sieber, and für die Teilnehmer des BANSS-Symposiums 2006, "Malnutrition, sarkopenie und kachexie im alter - von der pathophysiologie zur therapie," *Dtsch med Wochenschr*, vol. 133, no. 07, pp. 305,310, 2008. 305.

[19] S. Michel, "Externe Elektromyostimulation und lokale Muskelerm
üdung," 2003.

[20] R. Merletti and P. Parker, *Electromyography: physiology, engineering, and noninvasive applications*. IEEE Press series in biomedical engineering, IEEE/John Wiley & Sons, 2004.

[21] D. Stashuk, "Emg signal decomposition: how can it be accomplished and used?," *Journal of Electromyography and Kinesiology*, vol. 11, no. 3, pp. 151 – 173, 2001.

[22] D. F. Guillou, "Packaging mems: New manufacturing methodology substantially reduces smart mems costs." http://archives.sensorsmag.com/articles/1203/20/main.shtml. [Online; letzter Besuch: 28-März-2011].

[23] Wikipedia, "Beschleunigungssensor — Wikipedia." http://de.wikipedia.org/wiki/Beschleunigungssensor. [Online; letzter Besuch: 14-März-2011].

[24] S. Microelectronics, "Datenblatt — lis3lv02dl." http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/CD00091417.pdf. [Online; letzter Besuch: 14-März-2011].

[25] "Compatibility – mono." http://mono-project.com/Compatibility. [Online; letzter Besuch: 26-März-2011].

[26] ".net – wikipedia." http://de.wikipedia.org/wiki/.NET. [Online; letzter Besuch: 14-April-2011].

[27] Microsoft, ".net framework conceptual overview." http://msdn.microsoft.com/en-us/library/zw4w595w.aspx, 2011. [Online; letzter Besuch: 1-April-2011].

[28] T. Thai and H. Lam, *.NET framework essentials*. Essentials Series, O'Reilly, 2003.

[29] "Embarcadero." http://www.embarcadero.com/. [Online; letzter Besuch: 16-März-2011].

[30] "Dyalog." http://www.dyalog.com/. [Online; letzter Besuch: 16-März-2011].

[31] "Cls (common language specification)." http://msdn.microsoft.com/de-de/library/12a7a7h3%28VS.90%29.aspx. [Online; letzter Besuch: 18-März-2011].

[32] M. MacDonald, *WPF in C# 2008: Windows Presentation Foundation With .NET 3.5*. Apress Series, Apress, 2008.

[33] "Einführung in wpf." http://msdn.microsoft.com/de-de/library/aa970268.aspx, 2011. [Online; letzter Besuch: 22-März-2011].

[34] R. Agrusa, V. Mazza, and R. Penso, "Advanced 3d visualization for manufacturing and facility controls," in *Human System Interactions, 2009. HSI '09. 2nd Conference on*, pp. 456 –462, may 2009.

[35] P. Louridas, "Declarative gui programming in microsoft windows," *Software, IEEE*, vol. 24, pp. 16 –19, july-aug. 2007.

[36] Microsoft, "Wpf architecture." http://msdn.microsoft.com/en-us/library/ms750441.aspx, 2011. [Online; letzter Besuch: 25-März-2011].

[37] S. Wildermuth, "Build more responsive apps with the dispatcher," *MSDN Magazine*, vol. 10, Oktober 2007.

[38] C. Sells and I. Griffiths, *Programming Windows presentation foundation*. O'Reilly Series, O'Reilly, 2005.

[39] M. MacDonald and M. MacDonald, *XAML*. Apress, 2010.

[40] "Übersicht über xaml(wpf)." http://msdn.microsoft.com/de-de/library/ms752059.aspx, 2011. [Online; letzter Besuch: 27-März-2011].

[41] "Markup extensions and wpf xaml." http://msdn.microsoft.com/en-us/library/ms747254.aspx, 2011. [Online; letzter Besuch: 27-März-2011].

[42] "Data binding overview." http://msdn.microsoft.com/de-en/library/ms752347%28v=VS.85%29.aspx, 2011. [Online; letzter Besuch: 28-März-2011].

[43] J. Smith, "Wpf apps with the model-view-viewmodel design pattern," *MSDN Magazine*, vol. 2, pp. 1–19, 2009.

[44] "Microsoft sql server 2008 r2 compact edition." http://www.microsoft.com/sqlserver/en/us/editions/compact.aspx, 2011. [Online; letzter Besuch: 04-April-2011].

[45] MSDN, "Zusammenfassung zur wmi .net-technologie." http://msdn.microsoft.com/de-de/library/ms257353%28v=VS.80%29.aspx, 2011. [Online; letzter Besuch: 08-April-2011].