Technische Universität Dresden

Fakultät für Elektrotechnik und Informationstechnik

Insitut für Biomedizinische Technik

Manuskript Diplomarbeit

Thema: Entwicklung von Methoden zur Analyse und Aufbereitung biomedizinischer Messdaten

Vorgelegt von: Enrico Grunitz

Betreuer: Dr.-Ing. Sebastian Zaunseder
Eng. Fernando Andreotti

Verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. Hagen Malberg

Tag der Einreichung: XX. MONAT 2012

Selbständigkeitserklärung

Mit meiner Unterschrift versichere ich, dass ich die von mir am heutigen Tag eingereichte Diplomarbeit zum Thema

Entwicklung von Methoden zur Analyse und Aufbereitung biomedizinischer Messdaten

vollkommen selbständig und nur unter zuhilfenahme der angegebenen Quellen und Hilfsmittel erstellt habe. Zitate fremder Quellen sind als solche gekennzeichnet.

Dresden, den 10. Dezember 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abb.	Abbildung
EKG	Elektrokardiogramm
GUI	grafische Benutzeroberfläche *(graphical user interface)*
IBMT	Institut für biomedizinische Technik (der TU Dresden)
LGPL	*GNU Lesser General Public License* [9]
PPG	Fotoplethysmographie *photoplethysmography*
Tab.	Tabelle
UML	*Unified Modeling Language*

1. Einleitung

1.1. Motivation

Ergebnisse automatisierter Biosignalverarbeitungsmethoden werden aus mehreren Gründen oftmals manuell nachbearbeitet. So erfordert die Entwicklung neuer Methoden häufig eine Verifikation der Ergebnisse und eine eventuelle Korrektur der automatisch generierten Ausgabe. Zusätzlich ist eine schnelle visuelle Überprüfung von Ergebnissen, um einen ersten Eindruck über den Effekt einer Änderung an einer Methode zu bekommen, ein Mittel, das in der Entwicklungphase genutzt wird. Daher besteht eine Notwendigkeit eines Werkzeugs, welches die Visualisierung übernimmt und den Entwickler beim Editieren von Messdaten und Ergebnissen der Signalverarbeitung unterstützt.

Ein solches Werkzeug kann durch die Definition und Festlegung von Ein- und Ausgabeformaten zu einer Vereinheitlichung von Datenformaten führen. Durch die Bereitstellung eines solchen Werkzeugs für Dritte kann auch die methodische Grundlage für die Kooperation verschiedener Institutionen geschaffen werden. Um solche Kooperationen zu unterstützen sollte es, aufgrund der unterschiedlichen Voraussetzungen, wenig spezialsierte Anforderungen an seine Umgebung stellen.

1.2. Zielstellung

Das Ziel dieser Arbeit ist ein Programm zu konzipieren und umzusetzen, das unterschiedliche (Bio-) Signale grafisch darstellt und dem Nutzer die Möglichkeit bietet, Zeitpunkte und -intervalle innerhalb des Signalverlaufs zu markieren und mit Kommentaren zu versehen. Hierbei soll insbesonderer die gleichzeitige Darstellung mehrerer Signale unterschiedlicher Natur und Ausprägung unterstützt werden. Die Erstellung und Bearbeitung von Markierungen soll leicht verständlich aus der grafische Benutzeroberfläche (GUI) heraus geschehen. Zudem soll eine Grundlage geschaffen werden, paralell aufgenommene Signale in einem Datensatz zu vereinen.

Zusätzlich soll eine zukünftige Erweiterung der Funktionalität ermöglicht und unterstützt werden. Daher ist eine klare Gliederung der Einzelkomponenten gefordert und die Dokumentation des Quelltextes sowie der einzelnen Programmteile fundamentaler Bestandteil der Aufgabenstellung. Um die Erweiterbarkeit zusätzlich zu verbessern, soll die spätere Einbindung von Methoden der Signalverarbeitung vorbereitet werden. Dafür soll eine einfache Signalverarbeitungsfunktion in das Programm implementiert werden und in

die Benutzeroberfläche integriert werden. Die Arbeit eines zukünftigen Entwicklers wird somit durch die beispielhafte Integration einer zusätzlichen Methode vereinfacht.

Neben der Entwicklerdokumentation soll auch eine seperate Dokumentation für die Benutzer des Programms zur Verfügung gestellt werden. In dieser Nutzerdokumentation soll dem Anwender die Funktionsweise und Bedienung des Programms verständlich gemacht werden.

1.3. Konkretisieren der Aufgabenstellung

Die Entwicklung automatiserter Datenverarbeitungsmethoden am Institut für biomedizinische Technik (IBMT) erfolgt hauptsächlich innerhalb der von Matlab® (MathWorks, Inc., Natick, MA, USA) bereit gestellten Umgebung statt. Das zu erstellende Programm soll aber unabhängig aus lizensrechtlichen Gründen von Matlab® unabhängig sein. Durch die Möglichkeit in Matlab® Java-Code auszuführen, bietet es sich an das Programm mit der Programmiersprache Java zu implementiern. Dadurch werden zwei Ziele erreicht:

- Ausführbarkeit des Programms auf Computersystem durch Nutzung frei erhältlicher Software
- Integration der IBMT-internen Nutzung des Programms in die bereits vorhandenen Entwicklungmethoden und -Werkzeuge

Die Implementierung in Java hat zusätzlich noch den Vorteil, dass die Umsetzung der GUI durch vorhandenen Bibliotheksfunktionen unterstützt und vereinfacht wird. Die Datenspeicherung durch Matlab® erfolgt in einem proprietärem Dateiformat und unterlag in der Vergangenheit häufigen Veränderungen. Daher soll für die Speicherung und den Transport von Daten ein geeignetes, von Matlab® unabhängiges Datensatzformat gewählt werden. Zusätzlich müssen auch Skripte für eventuell anfallenden Import- und Exportfunktionen der Daten bereit gestellt werden. Bei der Erstellung der Konvertierungsskripte soll auf leicht verständliche und einfach Schnittstelle der Fokus gelegt werden. Eine ausführliche Dokumentation der Schnittstelle und der Skripte selbst ist natürlich auch ein Kernpunkt.

Wie schon in Abschnitt 1.2 erwähnt, ist die Erweiterbarkeit des Programmes ein wesentlicher Punkt der Umsetzung. Somit ist die Kapselung der einzelnen Programmkomponenten ein Bestandteil der Aufgabenstellung. Daher ist es notwendig die einzelnen funktionellen Implementierungen voneinander unabhängig zu gestalten. Interaktionen zwischen den einzelnen Komponenten soll über einfach Schnittstellen erfolgen. Zusätzlich ist es notwendig Test zu implementieren, die die korekte Funktionsweise der einzelnen Komponenten verifizieren. Damit kann nach einer geänderten Implementierung die Funktionalität erneut überprüft werden.

Die GUI soll eine intuitive Arbeit mit dem Programm ermöglichen. Da aber Begriffe wie „einfache Bedienbarkeit“ und „leicht verständliche Oberfläche“ von jeder Person anders interpretiert werden und somit nicht objektiv gemessen werden können, muss das

Programm und seine Funktionalität validiert werden. Hierzu sollen typische Arbeitabläufe mithilfe von Beispieldatensätzen ausgeführt werden. Durch die Validierung kann aber nur festgestellt werden, ob die Software den Wünschen und Erwartungen des Nutzers gerecht wird. Die im obigen Absatz erwähnte Verifikation der Funktionalität wird durch die Validierung der Software nicht obsolet.

Um dem Nutzer die Arbeit mit dem Programm zu ermöglichen, soll eine Benutzerdokumentation erstellt werden. In dieser Dokumentation soll dem Nutzer die Möglichkeiten, die ihm das Programm bietet, erläutert werden. Die einzelnen Werkzeuge und die Funktionsweise des Programmes sollen vermittelt werden. Eine weitere Dokumentation soll für die zweite Zielgruppe (zukünftige Entwickler des Programmes) erstellt werden. Im Zusammenhang mit der vorliegenden Arbeit soll innerhalb des zweiten Dokumentes das Wissen vermittelt werden. Genauer werden die einzelnen Bestandteile des Programmes und internen Beziehungen untereinander diskutiert. Die Entwicklerdokumentation besteht aus drei Elementen:

- Einem `pdf`-Dokument, dass den allgemeinen Aufbau und das interne Zusammenspiel der Komponent erklärt.
- Die mit dem Dokumentationswerkzeug Javadoc automatisiert erstellte Dokumentation aller Klassen, deren Membervariablen und Methoden, die die jeweilgen Schnittstellen im Detail beschreibt.
- Die Kommentierung des Quelltextes an notwendigen Stellen, die die Funktionsweise und Implementierung der Algorithmen besser verständlich machen.

Besonders die beispielhaft implementierte Signalverarbeitungsfunktion und die dazugehörige Dokumentation soll eine Weiterentwicklung des Programmes ermöglichen. Wird dieses Ziel erreicht, kann das Programm eine Grundlage für eine Plattform automatiserter Signalverarbeitung bilden.

2. Vorbetrachtungen

2.1. Allgemeiner Softwareentwicklungsprozess

In dieser Arbeit soll der Begriff Softwareentwicklung den Prozess benennen, der alle Aktivitäten und die damit verbundenen (Zwischen-) Ergebnisse bei der Erstellung von Software umfasst. In der Literatur wird auch der Begriff Softwareprozess genutzt [20]. Obwohl verschiedene Vorgehensmodelle für die Erstellung von Software existieren, haben alle Softwareentwicklungsprozesse vier grundlegende Arbeitsaktivitäten gemeinsam [5, 20]:

Softwarespezifikation: Es müssen die konkreten Anforderungen an das zu erstellende Programm ermittelt werden. Dabei wird die Funktion der Software, aber auch die Grenzen der Benutzung definiert.

Softwareentwurf und -implementierung: Nach Analyse der ermittelten Anforderungen kann die Architektur des Softwaresystems entworfen werden. Durch die Zerlegung der Software in mehrere Subsysteme werden die Anforderungen in kleine Teilprobleme getrennt. Diese Teilprobleme sind den verschiedenen Komponenten und Objekten der Software zugeordnet und können seperat gelöst werden. Diese Lösung wird durch den Objektentwurf und die schlußendliche Implementierung des jeweiligen Programmteils erreicht.

Validierung der Software: Die fertige Software muss auf ihre Funktionsfähigkeit überprüft werden. Hierbei ist auch abzusichern, dass die Software neben der gewünschten Funktionalität keine ungewollten Nebeneffekte enthält.

Weiterentwicklung: Ein Programm muss sich im Laufe seiner Lebenszeit weiter entwickeln, um sich ändernden Nutzeranforderungen gerecht zu werden.

Der Punkt Softwarespezifikation ist im Kapitel 3 abgehandelt. Im Kapitel 4 wird neben der Struktur und dem Aufbau des Programms auch auf die Fragen bezüglich der Entwurfsentscheidungen eingegangen. Durch die Validierung der Software muss sicher gestellt werden, dass die Software die Erwartungen des Benutzers erfüllt und den gestellten Bedingungen gerecht wird. Dieser Punkt ist im Abschnitt 3.3 behandelt. Zudem muss das Programm auch schon während der Entwicklung immer wieder auf die korrekte Funktionsweise überprüft werden. Dazu wird auch auf Tests der einzelnen Komponenten in den jeweiliegen Abschnitten im Kapitel 4 eingegangen. Der letzte, der vier oben genannten Aspekte, ist in dieser Arbeit ein nicht zu vernachlässigender Punkt. Weil gerade das Programm für die Nutzung während der Entwicklung von Signalverarbeitungsmethoden

erstellt wird, werden die Anforderungen an das Programm fortlaufend wachsen. Somit soll schon von Beginn an die Erweiterbarkeit dieses Projektes unterstützt und gefördert werden. Deshalb sind auch in den folgenden Anforderungen Punkte enthalten die diesen Aspekt besonders hervorheben und explizit fordern.

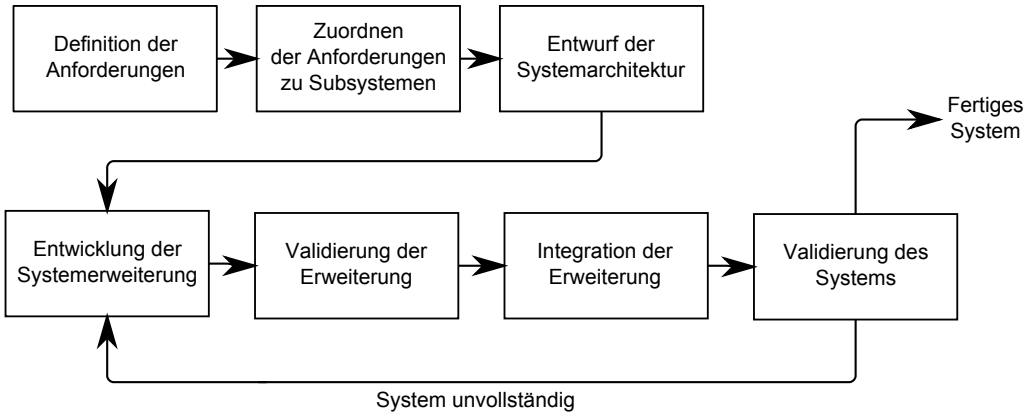

Abbildung 1.: Inkrementeller Softwareentwicklungsansatz nach [20]

Für die in dieser Arbeit zu entwickelnden Sofware wird eine Methode der inkrementellen Entwicklung genutzt. Die Herangehensweise dieser Methode ist in Abb. 1 dargestellt. Nach der anfänglichen Feststellung und Definition der grundlegenden Anforderungen an die Software, wird die allgemeine Struktur des Programms festgelegt. Die Subkomponenten des Programms werden einzeln entworfen und in das bestehende Programm integriert. Da die neuen Subkomponenten immer Teilanforderungen des Programmes erfüllen, steigert sich somit die Funktionalität des Programms. Dieser Ansatz hat den Vorteil, dass die Software schon zeitig im Entwicklungsstadium getestet werden kann und damit auch schon Erfahrungen gesammelt werden können. Zusätzlich können gewünschte Änderungen der Bedienung oder der Funktionalität erkannt und implementiert werden.

2.2. Genutzte Biosignale zur Programmvalidierung

Für die Validierung des Programmes werden typische Arbeitsschritte mit dem Programm auf Beispieldatensätze durchgeführt. In diesem Abschnitt soll kurz die genutzten Daten näher beschrieben werden.

2.2.1. Abdominale Elektrokardiogramme

Für Untersuchungen von EKG-Daten von Feten werden abdominale EKGs von Schwangeren aufgenommen. Die Datenaufzeichnung für das IBMT erfolgt in Zusammenarbeit der Abteilung für Pränatal- und Geburtsmedizin der Universitätsklink Leipzig. Dafür werden die EKGs mit acht Kanälen aufgezeichnet (Abb. 2). Es handelt es sich um sieben abdominale Ableitungen und einem mütterlichen Referenzsignal. Die Aufzeichnungsdauer beträgt rund 20 min und die Daten werden mit 1 kHz abgetastet. Weitere Details über die Daten-

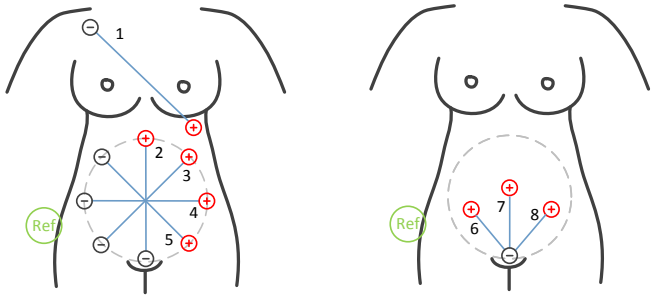

Abbildung 2.: Elektrodenkonfiguration zur Aufnahme des abdominalen EKG-Signals [25]

aufzeichnung und der Datenverarbeitung sind in den Arbeiten von F. Andreotti [1], M. C. Santiago [17] und S. Zaunseder [25] beschrieben. Die Datensätze enthalten typischerweise folgende Elemente:

- Rohdaten (sieben Kanäle plus dem Referenzsignal)
- vorverarbeitete Daten (gefilterte Rohsignale mit einem Bandpass (Durchlassbereich 2 bis 100 Hz [1]) und einem 49 - 51 Hz-Notchfilter)
- Abschätzung der mütterlichen und fetalen EKGs
- Detektionen der mütterlichen und fetalen QRS-Komplexe (als Annotationen)
- mütterliche und fetale RR-Zeitreihen aus der QRS-Detektion abgeleitet

Insgesamt enthält ein jeder Datensatz 29 Kanäle EKG-Daten, 14 Annotationskanäle und 14 RR-Zeiteihen. Eine Kontrolle der automatisierten QRS-Detektion ist in diesem Fall notwendig, da die Detektionsrate zwischen 66,1 und 94,5 % [25] schwankt.

2.2.2. Videogestützte Herzratenermittlung

Sensor	Abtastfrequenz in Hz
Webcam Logitech c170	10
Industriekamera uEye	100
Ohrläppchen Fotoplethysmographie (PPG)	1000
EKG	1000

Tabelle 1.: Abtastfrequenzen der in [26] genutzten Sensoren

Die Herzrate kann auf verschiedene Arten ermittelt werden. Ein Möglichkeit bietet die Analyse von PPG-Signalen. Durch einen kamerabasierten Ansatz kann die Erfassung des PPG-Signals berührungslos erfolgen. In der Arbeit von C. Zhai [26] ist eine solche Herangehensweise ausgearbeitet und untersucht. Um die erreicheten Ergebnisse zu validieren, werden diese mit denen aus bewährten Methoden gewonnenen verglichen. Dazu werden von

Probanden EKG-Daten nach der 1. Ableitung nach Einthoven aufgenommen. Zusätzlich wird ein PPG-Signal am rechten Ohrläppchen mit einen Reflexions-PPG-Sensor aufgenommen und das Gesicht des Probanden mit zwei Kameras gefilmt. Die Abtastfrequenzen der einzelnen Sensoren sind in Tab. 1 dargestellt. Auch hier sei für eine ausführlichere Beschreibung der Signale sowie der Messwertermittlung auf die ursprüngliche Arbeit [26] verwiesen. Die genutzten Datensätze enthalten folgende Daten:

- ein einkanaliges EKG
- Signal des PPG-Sensors
- detektierte R-Zacken aus EKG und Punkte maximalen Anstiegs aus PPG (Annotationen)
- RR-Zeitreihen (abgeleitet aus dem EKG)
- PP-Zeitreihen (abgeleitet aus dem PPG und den beiden Kameradaten)

Zusammengefasst enthält jeder Validierungsdatensatz zwei kontinuierlich abgetastete Signale, zwei Annotationskanäle und vier Zeitreihen (RR- bzw. PP-Zeitreihen). Die manuelle Nachkontrolle ist bei dieser Anwendung für die automatisiert erkannten Annotationen notwedig.

2.3. Begriffe aus der objektorientierten Programmierung in *Java*

In der objektorientierten Programmierung werden die einzelnen funktionalen Bestandteile eines Programms in einzelnen Klassen implementiert. Diese Klassen sind durch ihre Funktionen (*Methoden*) und ihre Variablen (*Member/Eigenschaften*) definiert. Als Objekte werden die einzelnen Instanzen dieser Klassen bezeichnet. Alle Objekte einer Klasse können unterschiedliche Werte in ihren Variablen speichern, somit sind im Allgemeinen alle Methodenaufrufe und Variablen für jedes Objekt lokal gespeichert.

Es können auch Methoden und Eigenschaften für eine Klasse selbst definiert werden — die Klassenmethoden und Klassenvariablen. Klassenvariablen werden für eine Klasse nur genau einmal abgespeichert und sind für alle Objektinstanzen dieser Klasse identisch. Klassenmethoden sind, ebenso wie die Klassenvariablen, für die Klasse definiert und können somit objektübergreifend Funktionalitäten übernehmen, können aber nicht auf die Member der einzelnen Objekte zugreifen.

Durch *Ableitung* von Klassen können Methoden und Eigenschaften von anderen Klassen übernommen werden, ohne dass dabei diese Elemente erneut programmiert werden müssen. Dabei bezeichnet man als *Superklasse* die Klasse von der abgeleitet wurde und die abgeleitete Klasse als *Unter-* bzw. *Subklasse*. Im Allgemeinen kann sobald eine Superklasse genutzt wird auch automatisch jede Unterklasse dieser Superklasse auf gleicher Art und Weise benutzt werden.

Klassen können in *Java* in verschiedenen Paketen (*packages*) angelegt werden. Eine Aufteilung in Pakete ermöglicht es komplexe Probleme sowohl logisch als auch strukturell zu trennen. Zusätzlich wird verhindert, dass es zu Kollisionen von Benennungen kommt.

Es existieren unterschiedliche Modifikatoren die die Sichtbarkeit von Funktionen und Variablen einer Klasse definieren. Die Sichtbarkeit bezeichnet in diesem Zusammenhang die Möglichkeit eine Variable zu lesen bzw. zu verändern oder eine bestimmt Methode aufzurufen. Ist etwas als privat (*private*) deklariert so kann die Funktion oder Methode nur Objekten der implementierenden Klasse selbst auf diese Elemente zugreifen. Das Gegenstück dazu ist die öffentliche (*public*) Deklaration: Alle Klassen besitzen für das entsprechende Element einen Vollzugriff. Elemente die als geschützt (*protected*) markiert sind, erlauben nur Zugriffe durch die deklarierende Klasse und ihre Subklassen.

Ein *Interface* ist die Deklaration einer wohldefinierten Schnittstelle. Durch sie werden Methodennamen und die Parameter definiert, jedoch keine direkte Funktionalität bereit gestellt. Mithilfe von *Interfaces* wird zugesichert, dass jede implementierende Klasse die bezeichneten Methoden zur Verfügung stellt.

Abstrakte Klassen sind Klassen von denen direkt keine Objekte instanziert werden dürfen. Sie besitzen aber dennoch Methoden und Membervariablen und können Funktionalitäten implementieren. Mit ihnen kann abstraktes Verhalten implementiert werden, das für alle Unterklassen gleich bleibt. Spezifische Funktionalität wird erst durch die Unterklasse bereitgestellt. Die Methodik abstrakter Klassen wird insbesondere im `signalprocessing`-Paket genutzt und ist im Abschnitt 4.3 beschrieben.

[3]

3. Spezifikation der Programmfunktionalität

Die Spezifikation der Software ist in dieser Arbeit in zwei Etappen aufgeteilt:

- Aufstellen von Anwendungsszenarien
- Ableitung der notwendigen Anforderungen

Ziel ist es eine Liste mit konkreten Anforderungen an das Programm zu erstellen. Das Aufstellen der Anwendungsszenarien dient einerseits dem Entwickler einen Überblick über die Gesamtproblematik zu erhalten. Zusätzlich dazu wird ersichtlich, welche Arbeitsschritte notwendig sind und wie diese zeitlich zu einander ausgeführt werden. Damit wird die vom Benutzer beabsichtigte Aktion in seine fundamentalen Bestandteile aufgeteilt. Diese Bestandteile sind somit Funktionen die das Programm bereit stellen muss, um die Anforderungen zu erfüllen.

3.1. Anwendungsszenarien

Der erste Schritt stellt das Ausarbeiten von Szenarien dar, die eine Beschreibung eines Merkmals des Programmes aus Sicht des Anwenders ist. Aufgrund dieser informellen Beschreibungen häufiger Arbeitsabläufe und typischer Aufgabenstellungen wird eine Übersicht gewonnen, was die zu erstellende Software leisten soll und der Nutzer erwartet. Mithilfe dieser Erwartungen können im Anschluss die funktionalen Anforderungen an das Programm formuliert und festgelegt werden. Das Ergebnis ist somit die Beschreibung des notwendigen (Software-) Systemumfangs und der zu implementierenden Arbeitsprozesse.

Der Anwender möchte ...

a) einen Datensatzes laden. Dieser Datensatz umfasst mehrere (Bio-) Signale die sowohl mit einer konstanten Abtastrate erfasst wurden als auch Signale die nicht zu äquidistanten Zeitpunkten abgetastet wurden.

b) einen geladenen Datensatz mit allen Änderungen speichern. Hierbei sollen auch Einstellungen gespeichert werden, die die optische Präsentation wiederspiegeln.

c) sich Informationen zu dem geladenen Datensatz und seinen beinhalteten Signalen anzeigen lassen und verändern.

d) bestimmte Signale des Datensatzes auswählen und sich diese in ihrem Verlauf anzeigen lassen (Signalansicht). Hierbei möchte er Bildschirmgröße der einzelnen Ansichten verändern.

e) die Signalansicht bezüglich der Zeit- und der Amplitudenachse vergrößern und verkleinern können (Zoomen). Entlang der Zeitachse möchte er sie verschieben können (Scrollen). Signaleverläufe die parallel aufgenommen wurden, sollen auch zusammen gescrollt werden.

f) in einer Signalansicht mehrere Signale mit denselben Achsen darstellen lassen. Beispielsweise um ein Roh- und ein verarbeitetes Signal miteinander vergleichen zu können.

g) einen Amplitudenbereich eines Signals optisch hervorheben.

h) einzelne Zeitpunkte im Signalverlauf mit einer Markierung versehen und kommentieren. Diese Markierung kann sowohl für ein bestimmtes Signal gelten, aber auch für alle Signale des Datensatzes.

i) einen Zeitabschnitt markieren. Die Markierung der Abschnitte soll analog zur Markierung von Zeitpunkten erfolgen.

j) die Markierungen verändern (zeitlich verschieben, umbennen) oder löschen.

k) Markierungen gemeinsam mit dem Datensatz aber auch unabhängig vom Datensatz abspeichern.

3.2. Anforderungen an das Programm

Mithilfe der oben beschriebenen Anwendungsszenarien kann daraus die konkrete Funktionalität der Software definiert werden. Diese Definition erfolgt durch die Bestimmung konkreter Anforderungen an das Programm. Dabei beschreiben die Anforderungen die konkret umzusetzenden Funktionen und Arbeitswerkzeuge. Zusätzlich wird die im Kapitel 5 beschriebene Validierung der Software die in diesem Abschnitt ausgearbeiteten Definitionen als Grundlage nehmen um das erstellte Programm zu überprüfen und zu bewerten.

Die folgende List von zu erfüllenden Anforderungen ergibt sich aus den oben beschreibenen Anwendungsszenarien. Wenn mehrere Einzelanforderungen in einer Beschreibung enthalten sind, sind diese mit einer Ziffer in Klammern markiert. Das Programm ...

A) muss eine grafische Benutzeroberfläche besitzen.

B) muss ein Datensatzformat unterstützen, das äquidistant (1) und nicht äquidistante (2) abgetastete Signale speichern kann.

C) soll in der Lage sein, Daten aus einem Datensatz zu laden (1). Dem Nutzer muss es ermöglicht werden, diese Signaldaten aus einer Übersicht auszuwählen (2) und in Diagrammen darstellen zu lassen.

D) muss dem Nutzer die Möglichkeit bieten allgemeine Informationen sowohl über den Datensatz (1) als auch über die enthaltenen Daten (2) anzuzeigen.

E) muss in der Lage sein, die Signalverläufe sowohl einzeln (1) in einem Diagramm darzustellen, aber auch mehrere verschieden Signalverläufe (2) in ein und demselben Diagramm zu visualisieren. Diese Signalansichten sollten in ihrer Darstellungsgröße durch den Nutzer veränderbar sein (3).

F) soll dem Benutzer ermöglichen, seine Signalansicht frei „bewegen“ zu können. Es muss eine Vergrößerung und Verkleinerung bezüglich der Abszissen- und der Ordinatenachse unterstützen (1). Zusätzlich ist die Fähigkeit des Verschiebens der Ansicht gefordert (2). Dabei sollen mehrere Diagramme gleichzeitig Verschoben werden können (3).

G) muss in der Lage sein einen Amplitudenbereich ein oder mehrerer Signalansichten optisch hervorzuheben.

H) soll dem Nutzer ein Werkzeug zur Verfügung stellen, das ihm erlaubt Datenpunkte zu annotieren (1). Diese Annotationen sollen optisch in den Signalansichten ersichtlich sein (2) und mit einem Kommentar versehen werden können (3). Ferner ist gefordert, dass vorhandene Annotationen veränderbar sind (4).

I) soll neben der Annotation einzelner Datenpunkte auch die Markierung von Signalbereichen unterstützt werden.

J) muss Änderungen an den Signalen selbst (1) und den Annotationen (2) speichern können. Annotationen müssen unabhängig von Signalen gespeichert werden können (3). Insbesondere dürfen Annotationen sich nicht verändern, wenn sich das Ursprungssignal verändert oder nicht mehr vorhanden ist (4).

K) soll interne Einstellungen abspeichern und von einer Sitzung zur nächsten übernehmen (1). Optionen bezüglich der Darstellung von Signalen sollen in dem Datensatz mit abgespeichert werden können (2).

Die in der Aufgabenstellung geforderte Ausbaufähigkeit der Programms ist nicht durch die Anwendungsszenarien abgedeckt werden. Hierbei handelt es sich um eine nichtfunktionale Anforderung an das Programm. Daher wird die folgenden Anforderung nur auf Basis der Aufgabenstellung formuliert und nicht aufgrund der Erwartungshaltung des Benutzers:

L) Das Programm soll dem Benutzer ermöglichen eine Signalverarbeitungsmethode auf ein gewähltes Biosignal anwenden zu können (1). Dabei muss das Originalsignal unverändert bleiben (2). Der bearbeitete Signalverlauf kann als eigenes Signal im Datensatz abgespeichert werden (3). Die Implementierung dieser Anforderung soll beispielhaft für zukünftige Entwickler erfolgen um die Erweiterbarkeit zu gewährleisten.

3.3. Testszenarien

In diesem Abschnitt sollen Szenarien heraus gearbeitet werden, mit denen die Software am Ende der Implementierung validiert werden kann. Dabei soll die oben geforderte Funktionalität anhand der Behandlung von Biosignalen überprüft werden.

Folgend sind die Testszenarien beschrieben, die zur Validierung der Software durchgeführt werden sollen. Die Ausgangsbedingung ist eine das einfache Starten des Programmes. Somit soll der erste, nicht jeweils explizit genannte Schritt sein, das Programm zu starten. Abgeschlossen wird jedes Testszenario mit dem Speichern der geladenen Daten.

Testszenario 1 Der Benutzer lädt einen Datensatz abdominaler EKG-Daten mit sieben Aufnahmekanälen. Zusätzlich sind noch die approximierten EKG-Signale des Fetus sowie der Mutter im Datensatz gespeichert. Der Benutzer lässt sich alle verfügbaren Informationen zu dem geladenem Datensatz anzeigen. Anschließend lässt er sich einen Kanal sowohl des Rohsignals als auch der beiden abgeleiteten Signale jeweils in einer eigenen Ansicht anzeigen. Er verschafft sich durch eine geringe Zoomstufe einen Überblick über die Signalverläufe. Der Benutzer zoomt auf interessante Bereiche der Aufnahme herein und vergrößert die Ansicht der Einzelsignale. Er schließt die Ansicht des Rohsignals. Der Benutzer markiert in zwei unterschiedlichen, neu zu erstellenden Annotationskanälen die QRS-Komplexe der Mutter, sowie des Fetus (über mindestens fünf Minuten des Signalverlaufs). Zur Überprüfung der Annotationen lässt er sich alle Signale und die gemachten Annotationen in einer Signalansicht darstellen.

Testszenario 2 Der Benutzer lädt den im Testszenario abgespeicherten Datensatz wieder in das Programm. Er überprüft ob die Ansichten und die Einstellungen aus dem ersten Testszenario übernommen wurden. Der Benutzer speichert die Einstellungen der Signalansichten. Er wählt fünf beliebige Annotationen aus versieht diese mit Kommentaren. Der Benutzer löscht jede zweite Annotation des fetalen QRS-Komplexes. Weiterhin soll er mindestens 10 Zeitbereiche markieren, wobei es auch zu Überschneidungen diese Bereiche kommen soll. Er lädt die zuvor gespeicherten Einstellungen und überprüft ob diese richtig geladen wurden.

Testszenario 3 Der Benutzer lädt den Bearbeiteten Datensatz aus dem zweiten Testszenario. Er entfernt die Kanäle der approximierten EKG-Verläufe aus dem Datensatz. Der Benutzer überprüft die gemachten Annotationen mithilfe des Rohsignals.

Testszenario 4 Der Benutzer lädt einen Datensatz mit EKG-und PPG-Signalen. Er wählt PPG und EKG Kanäle und zeigt sie sich in einer Ansicht an. Der Benutzer verschafft sich eine Übersicht durch eine geringe Zoomstufe über die Signalverläufe. Der Benutzer markiert einen Amplitudenbereich im PPG-Signal. Er wendet eine Filterfunktion auf das EKG-Signal an und speichert das Ergebnis im Datensatz ab. Er kontrolliert die Informationen des veränderten Signals. Der Nutzer entfernt das Original-EKG-Signal aus dem Datensatz.

Testszenario 5 Der Benutzer lädt den Datensatz aus dem viertem Testszenario. Er kontrolliert dabei die Einstellungen der Ansichten darauf, ob sie aus dem Szenario 4 übernommen wurden. Er annotiert die QRS-Komplexe des gefiltertem EKG-Signals für mindestens fünf Minuten des Signalverlaufs. Der Benutzer wendet eine weitere Filterfunktion auf das bereits gefilterte Signal an und speichert das Ergebnis erneut im Datensatz ab. Er kontrolliert die Annotationen mit dem erneut gefiltertem Signal.

Tabelle 2.: Abdeckung der Anforderungen durch die Testszenarien (TS)

	Anforderung															
	A	B1	B2	C1	C2	D1	D2	E1	E2	E3	F1	F2	F3	G	H1	H2
TS 1	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
TS 2	✓	✓		✓	✓			✓			✓	✓	✓			✓
TS 3	✓	✓		✓	✓			✓			✓					✓
TS 4	✓	✓	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓		
TS 5	✓	✓	✓	✓	✓			✓			✓	✓	✓	✓		✓

	Anforderung											
	H3	H4	I	J1	J2	J3	J4	K1	K2	L1	L2	L3
TS 1					✓							
TS 2	✓	✓	✓		✓			✓	✓			
TS 3						✓	✓		✓			
TS 4				✓						✓	✓	✓
TS 5				✓	✓	✓	✓	✓	✓	✓	✓	✓

In Tab. 2 ist übersichtlich aufgelistet welche der Anforderungen durch die Testszenarien abgedeckt sind. Es ist erkenntlich, dass alle Anforderungen mindestens durch ein Testszenario überprüft wird. Durch das Bestehen der Testszenarien wird gezeigt, dass das Programm die Erwartungen des Nutzers erfüllt.

Das Testen der Funktionalität mittels bestimmter Biosignale sind aber nur spezielle Einzelfälle. Es kann durch sie nicht die absolute Fehlerfreiheit der Software gezeigt werden. Um entstehende Fehler schon während der Entwicklung abfangen und beheben zu können, wird die Software bzw. die einzelnen Programmteile auch schon einzeln daraufhin getestet, dass sie sich so verhalten, wie es der Programmierer vorgesehen hat. Speziell wird auch das korekte Verhalten im Falle eines Fehlers überprüft. Diese fortlaufenden funktionellen Test werden in den entsprechenden Abschnitten zu den einzelnen Programmkomponenten im Kapitel 4 beschrieben.

4. Programmentwurf und -implementierung

In diesem Kapitel möchte der Autor die Entwicklung des Gesamtprogramms erörtern. Es wird ein Überblick über das umfassende Konzept des internen Aufbaus gegeben um anschließend auf die konkrete Umsetzung der einzelnen Bestandteile einzugehen. Demzufolge sind die kommenden Abschnitte nach den Programmkomponenten gegliedert. In jedem einzelnen Abschnitt wird auf drei Punkte eingegangen:

- Grundlegende Idee und Designkonzept
- Implementierungsdetails
- Tests zur Verifikation der einzelnen Komponente

Es sei darauf hingewiesen, dass die eigentlichen Entwicklung und Implementierung nicht Komponentenweise, sondern nach entsprechenden Funktionalitäten statt findet. Das Einbinden einer bestimmten Funktionalität betrifft oft mehrere Komponenten und ihr Zusammenspiel, wodurch der Ausbau der einzelnen Bestandteile parallel geschieht. Die Gliederung dieses Kapitels nach den Programmteilen dient jedeglich der Übersichtlichkeit.

4.1. [WIP]Überblick über das Gesamtprogramm

– theoretische beschreibung des gesamtkonzeptes – komplette auflisten der subkomponenten in Tab. 3 – graphische Übersicht der Bestandteile siehe Abb. 3

4.2. Behandlung der Datensätze

Aufgrund der Vielfalt der genutzten Datenformate zur Speicherung von Biosignalen und der Nichtexistenz eines einheitlichen Standards [19, 23, 24] muss ein Format gesucht werden, dass zur Behandlung von Datensätzen im Rahmen dieser Arbeit geeignet ist. Die Wahl des Datensatzformates stützt sich auf die vergleichende Untersuchung [19] von Dr. Alois Schlögl der Technischen Universität Graz. Das Unisens-Format besitzt eine Referenzimplementierung mit Programmierschnittstellen sowohl für Matlab als auch für Java und bietet für diese Arbeit optimale Anwendungsvoraussetzungen. Die Organisation über eine menschlich les- und editierbare Headerdatei ist gerade in der Entwicklung interessant. Datensätze können ohne Bearbeitungstools verändert und an die Bedürfnisse angepasst werden. Einer der Kritikpunkte nach [19] ist, dass das Format aus mehreren Dateien besteht. Da die Behandlung von Dateien und Verzeichnissen mit aktuellen Betriebssystemen kaum

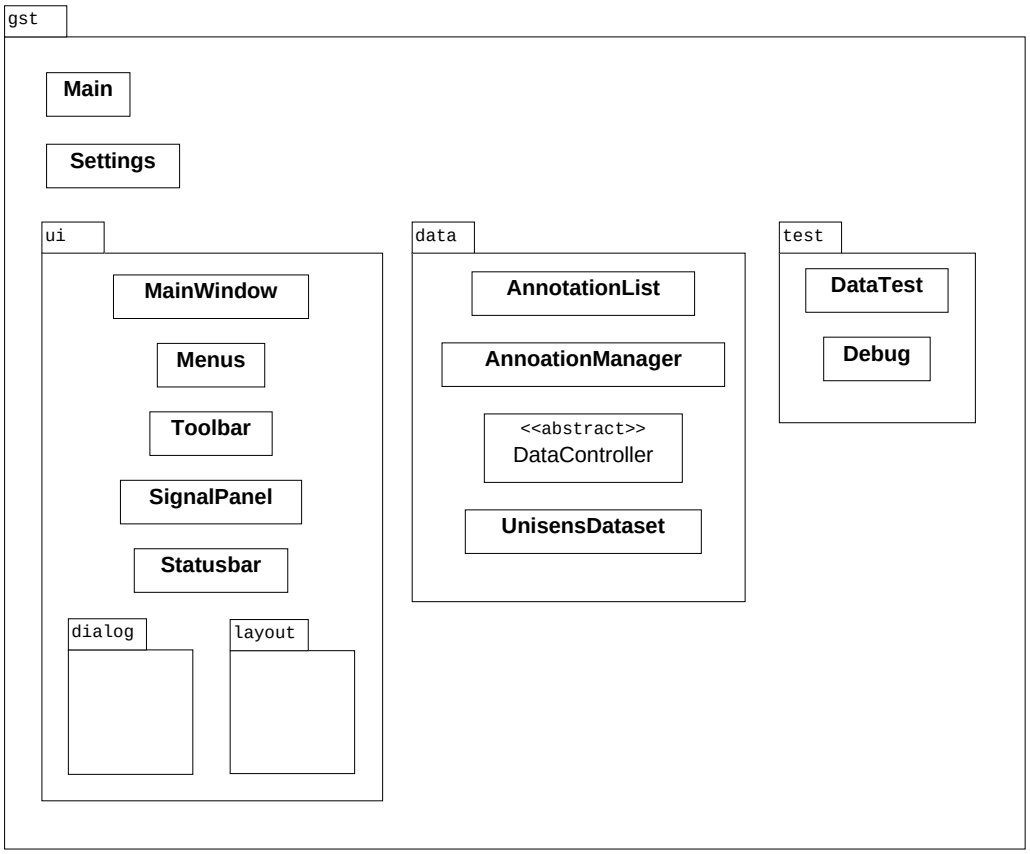

Abbildung 3.: UML-Paket-Übersicht der umgesetzten Software, nur wesentliche Klassen sind dargestellt

Unterschiede für den Anwender darstellt, ist nach Meinung des Autors dieser Punkt nicht von hoher Priorität. Es bietet sogar die Möglichkeit Daten von Sensoren (bei bekannten technischen Parametern wie z.B. Abtastrate und -auflösung) direkt in einen Datensatz zu integrieren. Zusätzlich bietet das Format durch seine Definition eine gute Erweiterbarkeit. Es kann somit an die neue Gegebenheiten und Voraussetzungen angepasst und optimiert werden.

4.2.1. Allgemeiner Aufbau des Unisens Datensatzes

Das vom Forschungszentrum Informatik und Institut für Technik der Informationsverarbeitung der Universität Karlsruhe entwickelte Datenformat Unisens dient der Speicherung und der Dokumentation von Sensordaten [14, 15]. Unisens ist konzipiert, Daten verschiedener Sensoren innerhalb eines Datensatzes zu speichern. Ein Datensatz ist im Dateisystem durch ein eigenes Verzeichnis und eine Headerdatei `unisens.xml` hinterlegt. In der Headerdatei werden alle Informationen über die Bestandteile des Datensatzes, deren Formatierung und ihre semantischen Zusammenhänge gespeichert. Messwerte eines Sensors werden üblicherweise in einer Datendatei innerhalb des Datensatzverzeichnisses abgespeichert. Die in einer solchen Datendatei gespeicherten Daten werden als *Entry* in dem Datensatz bezeich-

Tabelle 3.: Paket- und Klassenübersicht: Paketnamen sind in `True-Type` und Klassennamen in serifenloser Schriftart geschrieben

`gst`	Main		
	Settings		
`gst.`	`data`	AnnotationController	
		AnnotationList	
		AnnotationManager	
		DataController	
		SignalController	
		UnisensDataset	
		ValueController	
`gst.`	`test`	DataTest	
		Debug	
`gst.`	`ui`	AnnotationSelectionDialog	
		DataSelectionDialog	
		MainWindow	
		Menus	
		NamedMouseAdapter	
		Sidebar	
		SignalOverview	
		SignalPanel	
		SignalView	
		SignalViewFactory	
		StatusBar	
		Toolbar	
`gst.`	`ui.`	`dialog`	DatasetSelectionDialog
			EditEventDialog
			EnterFileNameDialog
`gst.`	`ui.`	`layout`	ComponentArrangement
			MultiSplit
			SignalPanelLayoutManager
			VerticalLayoutManager

net. Alle Metainformationen zu den Sensordaten werden in der Headerdatei abgespeichert, so dass die Datendateien selbst immer nur die reinen Messdaten enthalten. Als mögliche Sensordaten werden sowohl kontinuierlich abgetastete Signale als auch ereignisorientierte Daten unterstützt. Unisens unterscheidet zwischen vier Arten von Daten:

Signale *(Signal)*

Signale sind äquidistant abgetastete, numerische Messdaten. Sie zeichnen sich durch eine beliebige aber konstante Abtastrate und Abtastauflösung aus. Zudem können Signale aus mehreren Kanälen bestehen, die alle in ein und derselben Datei abgespeichert werden. Alle Kanäle desselben Signals haben auch dieselbe Abtastrate und -auflösung.

Ereignisse *(Event)*

Ereignisse sind diskrete Zeitpunkte die mit einer textlichen Beschreibung versehen

sind. (z.B. Triggersignale) Sie zeichnen sich durch einen Zeitstempel und einem textuellen Kürzel aus. Optional kann zu jedem Ereignis ein Kommentar hinzugefügt werden.

Einzelwerte *(Value)*

Einzelwerte sind eine Kombination der beiden oben genannten Datenarten. Sie beinhalten numerische Werte die zu bestimmten Zeitpunkten aufgenommen wurden. Mit Einzelwerten ist es möglich Daten zu speichern, die nicht in festen Zeitintervallen gemessen werden.

Benutzerdefinierte Daten *(Custom data)*

Mit dieser Art können anwendungsspezifisch Daten gespeichert werden, die durch die drei oben genannten Arten nicht erfasst werden können. So können beispielsweise schematische Darstellungen des Messaufbaus als Bilddateien oder Patientenakten in Form von Textdateien dem Datensatz hinzugefügt werden. Es ist durch diese vierte Datenart zusätzlich möglich, Daten die in einem proprietären Dateiformat vorliegen, mit im Datensatz zu speichern.

Eine detailliertere Beschreibung des Formates kann der offiziellen Dokumentation [15] entnommen werden.

4.2.2. Details der Unisens Referenzimplementierung

In diesem Abschnitt wird auf einige Details der Umsetzung des Unisens-Formates eingegangen. Für das Unisens-Paket gibt es eine Referenzimplementierung in Java, die unter der GNU Lesser General Public License *[9] (LGPL)* zur Verfügung gestellt wird. Die bereitgestellte Bibliothek ist auf zwei Pakete aufgeteilt: `org.unisens.jar` und `org.unisens.ri.jar`.

Bei der ersten Datei handelt es sich um die Definition des Unisensformates und seiner Bestandteile als Javaklassenstruktur. Diese Definition erfolgt hauptsächlich als *Interface*klassen welche die Schnittstellen zwischen den einzelnen Bestandteilen festlegt. Eine Übersicht der Klassenstruktur und der von außen ersichtlichen Attribute ist in Abb. 4 auf Seite 26 dargestellt. Die vom Unisensformat unterstützten Signalarten sind auch in der Klassenstruktur in Tab. 4 erkennbar.

Signale	`SignalEntry`
Ereignisse	`EventEntry`
Einzelwerte	`ValuesEntry`
Benutzerdefinierte Daten	`CustomEntry`

Tabelle 4.: Signalarten und ihre repräsentierenden Klassen

Aufgrund der Ableitung der Klassen `EventEntry` und `ValuesEntry` von `TimedEntry` ist ersichtlich, dass die Zeitpunkte von Ereignisdaten und Einzelwertdaten über eine virtuelle Abtastrate bestimmt werden. Der Zeitpunkt eines jeden *Event*- oder *Value*-Eintrags ist als ganzzahlige Samplenummer dieser Abtastrate gespeichert. Die Zeit eines Ereignisses,

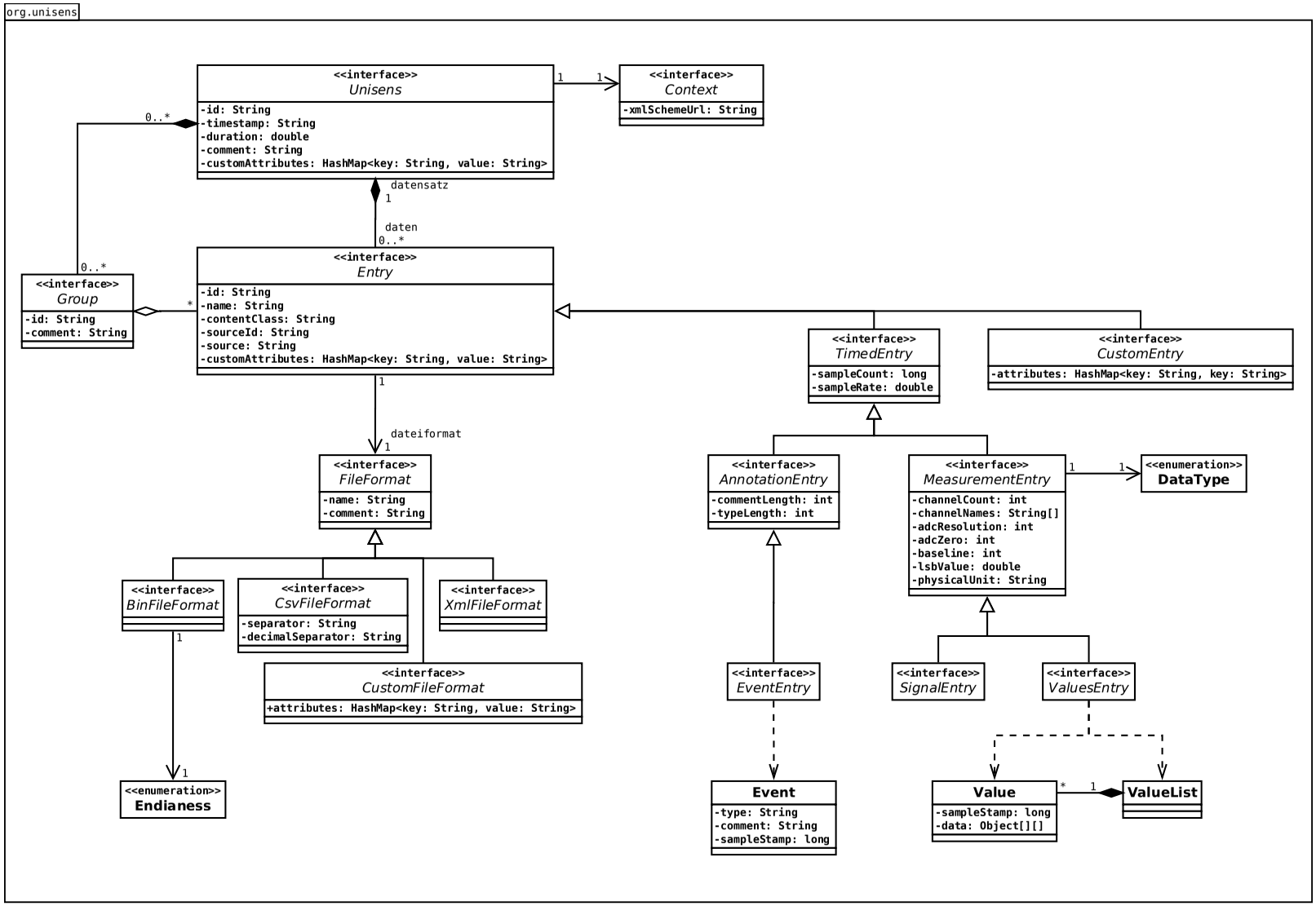

Abbildung 4.: Klassenübersicht der von Unisens definierten Schnittstellen

relativ zum Messbeginn, errechnet sich daher gemäß $Zeitpunkt = \frac{Samplenummer}{Abtastrate}$. Möchte man die Möglichkeit haben, Ereignisse für jeden beliebigen Datenpunkt eines Datensatzes zuordnen zu können, dann muss die virtuelle Abtastrate als das kleinste gemeinsame Vielfache aller vorhandenen Abtastraten gewählt werden.

Die Umsetzung der Funktionalität zur Nutzung des Datensatzformates ist in der zweiten Bibliotheksdatei `org.unisens.ri.jar` abgespeichert. Im Folgenden soll sich der Begriff Referenzimplementierung auf diese funktionelle Umsetzung beziehen. Die Klassen der Referenzimplementierung bestehen aus den Klassennamen der Schnittstellendefinition und dem Suffix „Impl“ (z.B. Objekte die den Datensatz darstellen haben die Klasse `UnisensImpl`). Um schon vorhandene Unisensdatensätze benutzen zu können reicht es aus, die Schnittstellendefinition zu kennen und zu nutzen. Sollen hingegen konkret Objekte erstellt werden, muss auf die Referenzimplementierung zurückgegriffen werden.

4.2.3. Limitationen von Unisens und der Referenzimplementierung

Das Datenformat Unisens hat zwei Einschränkungen die in diesem Abschnitt erwähnt werden sollen. Zum einen weißt die Referenzimplementierung einen Fehler bei der Behandlung von Gruppen auf. Es kann beim Laden eines Datensatzes, in dem Signale in Gruppen zusammengefasst sind, zu einer `NullPointerException` kommen. Insbesondere tritt dieser Fehler auf, wenn innerhalb der Headerdatei der Gruppeneintrag nicht hinter den Dateneinträgen steht.

Des Weiteren bietet die für das Datensatzformat Unisens bereit gestellten Schnittstellen keine direkte Möglichkeit die Datenpunkte eines Dateneintrags zu verändern. Anhängen zusätzlicher Datenpunkte wird unterstützt, aber das Einfügen, Entfernen oder Umsortieren von Datenpunkten ist nicht vorgesehen. Da aber diese Funktionalität gewünscht ist, muss sie zusätzlich implementiert werden. Die Umsetzung in dem entwickeltem Programm der genannten Funktionen ist in den Abschnitten 4.2.4.3 und 4.2.4.4 weiter unten beschrieben.

4.2.4. Programminterne Datenstruktur

Das Paket `gst.data` ist die Schnittstelle des Programms zu dem gewählten Datensatzformat Unisens. Eine Übersicht des Paketes ist im UML-Diagramm in Abb. 5 dargestellt. Dieses Paket erfüllt zwei wesentliche Aufgaben:

- Vereinheitlichung der Behandlung der unterschiedlichen Datenarten
- Abkapselung anderer Programmteile vom gewähltem Datensatzformat Unisens

Die Vereinheitlichung ist notwendig, da Unisens die Daten in vier Arten gliedert: Signale (`SignalEntry`), Werte (`ValueEntry`), Ereignisse (`EventEntry`) und benutzerdefinierte Daten (`CustomEntry`). Für die Bearbeitung und Darstellung ist aber die Unterscheidung von Signalen und Werten innerhalb des Programms nicht notwendig. In beiden Fällen handelt es sich um numerische Werte die zu einem bestimmten Zeitpunkt aufgenommen wurden. Somit sollen diese auch vom Programm gleichartig behandelt werden.

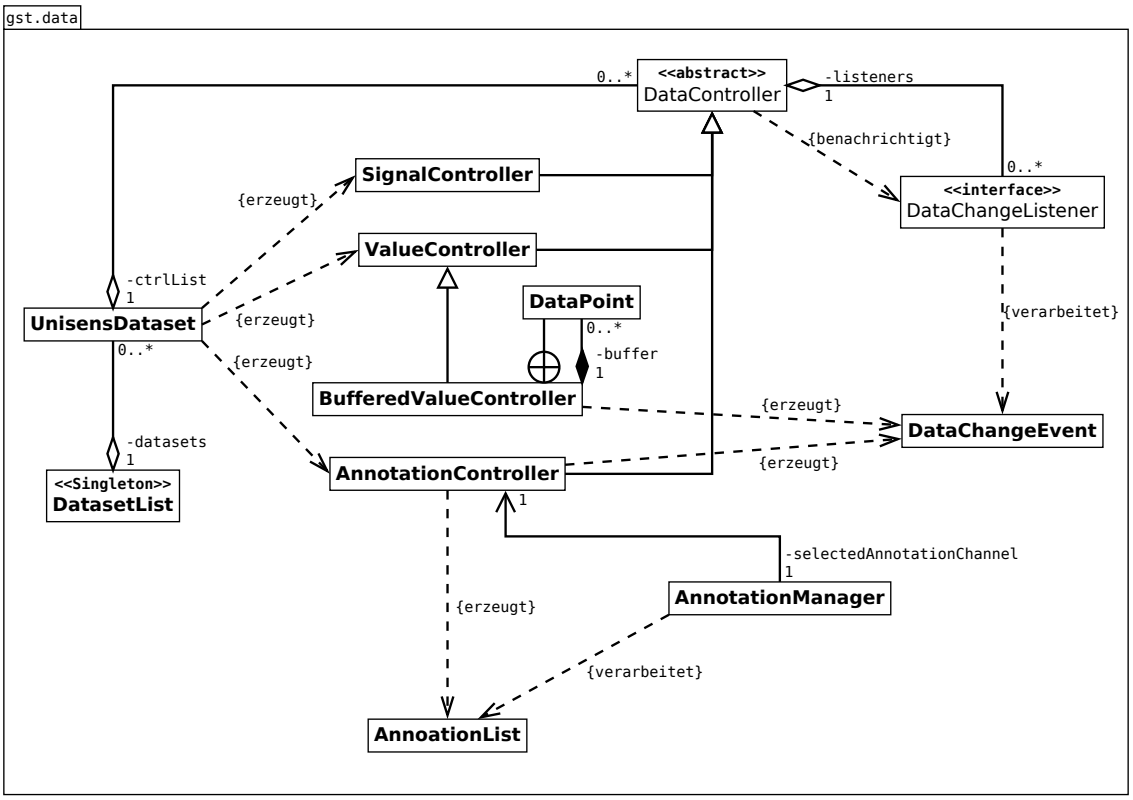

Abbildung 5.: UML-Diagramm des `data`-Paketes inklusive der bereitgestellten Schnittstellen

Daher erhalten alle Datenarten die abstrakte Klasse `DataController` als *Wrapper*. Die jeweilige konkrete Implementierung ist in den drei abgeleiteten Klassen `Signal-`, `Value-` und `AnnotationController` realisiert. Neben der Vereinheitlichung der Schnittstelle zum Datensatz werden auch die Zugriffe auf die Daten vereinfacht. Es werden notwendige Ausnahmebehandlungen (*Exceptions*) durch die Klassen des `data`-Paketes abgefangen und behandelt. Darunter fällt unter anderem die Behandlung von Fehlern, wenn Daten aufgrund von fehlenden Zugriffsrechten nicht gelesen oder geschrieben werden können. Somit ist der Zugriff auf die konkreten Daten des Datensatzes sowohl vereinheitlicht als auch vereinfacht.

Benutzerdefinierte Daten werden durch ihr nicht näher definiertes Format vom Programm in der vorgelegten Version ignoriert. Somit können Daten dieses Typs in einem Datensatz vorhanden sein, schränken aber die Funktionalität des Programms nicht ein. Soll durch zukünftigen Programmversionen beispielsweise die Visualisierung von Bild- oder Videodaten ermöglicht werden, können diese Daten als `CustomEntry` in den Datensatz integriert werden. Das Paket `data` kann bei Kenntnis des konkreten Formates dieser Daten erweitert werden.

Weiterhin wird mit dem `data`-Paket erreicht, dass die restlichen Komponenten des Programms nicht direkt auf das Datensatzformat zugreifen. Sie sind damit von dem gewählten Format abgekapselt und ein Austausch des Formates erfordert nur die Anpassung des

`data`-Paketes. Der gesamte Datensatz ist durch die Wrapper-Klasse `UnisensDataset` im Programm repräsentiert und ist im Abschnitt 4.2.4.1 näher beschrieben. Zugriff auf den Datensatz erfolgt nur über die von der Klasse bereitgestellte Schnittstelle. Weiterhin wird der veränderte Zugriff auf Annotationen durch die Klasse `AnnotationManager` realisiert und die Behandlung von Annotationen selbst über die Klasse `AnnotationList` abgewickelt. Die zwei zu letzt genannten Klassen kapseln die von Unisens bereitgestellten Klassen (`Event` und `EventList`) von den anderen Programmteilen ab.

Neben den zwei oben genannten Aspekten wird auch der Zugriff auf die Daten selbst durch das `data`-Paket verändert. Unisens arbeitet durchgehend mit Samplenummern bei der Indizierung der einzelnen Datenpunkte. Die durch das Paket bereitgestellte Schnittstelle wandelt diese Art des Zugriffs auf eine zeitbasierte Art um. Das bedeutet, anstatt Daten über die Angabe von Samplenummern zu erhalten, bietet die Schnittstelle die Möglichkeit Daten aus einem bestimmten Zeitraum zu erhalten. Die notwendige Umrechnung zwischen Samplenummern und Abtastraten auf entsprechende Zeitpunkte wird im `data`-Paket ausgeführt. Dadurch wird die Existenz verschiedener Abtastraten der unterschiedlichen Signale vor dem restlichen Programm verborgen.

4.2.4.1. Repräsentation des Datensatzes durch die Klassen `UnisensDataset` und `DataController`

Die Klasse `UnsiensDataset` stellt die Schnittstelle des Programms zu den Datensätzen dar. Für jeden geladenen Datensatz wird im Programm ein Objekt der Klasse `UnisensDataset` erzeugt. Diese Objekte bieten die erforderlichen Funktionen um auf die Eigenschaften des Datensatzes, die Eigenschaften der Dateneinträge und die Daten der Einträge selbst zuzugreifen. Beim Laden eines Datensatzes wird für jeden Dateneintrag ein Instanz eines `DataController`s erzeugt und in einer Liste `ctrlList` im `UnisensDataset`-Objekt gespeichert. Je nach Datenart des Eintrags wird zur Laufzeit dynamisch ein Objekt der Klassen `Signal-`, `Value-` oder `AnnotationController` erzeugt. Genauer wird sogar für jeden Kanal ein eigener `DataController` angelegt. Jede Komponente des Programms die auf einen Dateneintrag zugreifen möchte, erhält vom `UnisensDataset` den entsprechenden Controller. Es wird sichergestellt, dass zu jedem Zeitpunkt jedem Dateneintrag genau ein Controllerobjekt zugeordnet ist und keine Duplizierung auftreten kann. Durch spezielle Funktionen `AddNewSignal()`, `AddNewValue()` und `AddNewAnnotation()` können einem Datensatz zusätzliche Dateneinträge hinzugefügt werden. Die entsprechenden `DataController` werden dann in die Liste `ctrlList` aufgenommen und es kann durch diese anschließend auf die Daten zugegriffen werden. Der gesamte Datensatz wird bei einem Aufruf der `save()`-Funktion des `UnisensDataset`-Objektes abgespeichert werden. Diese Speicherfunktion ruft rekursiv für alle `DataController`-Objekte in `crtlList` die entsprechende Funktion `save()` zum Speichern der einzelnen Dateneinträge auf.

4.2.4.2. Das Singleton-Konzept und `DatasetList`

Die geladenen Datensätze werden im Programm in einer Liste zusammengefasst werden. Diese Liste soll im gesamten Programm zugänglich sein. Um das Erzeugen mehrerer `UnisensDataset`-Instanzen für ein und denselben Datensatz zu vermeiden, muss aber sichergestellt werden, dass ein Datensatz nicht mehrfach geladen wird. Diese Aufgabe wird durch die Klasse `DatasetList` übernommen, die als Singleton-Klasse implementiert ist (siehe Abb. 6).

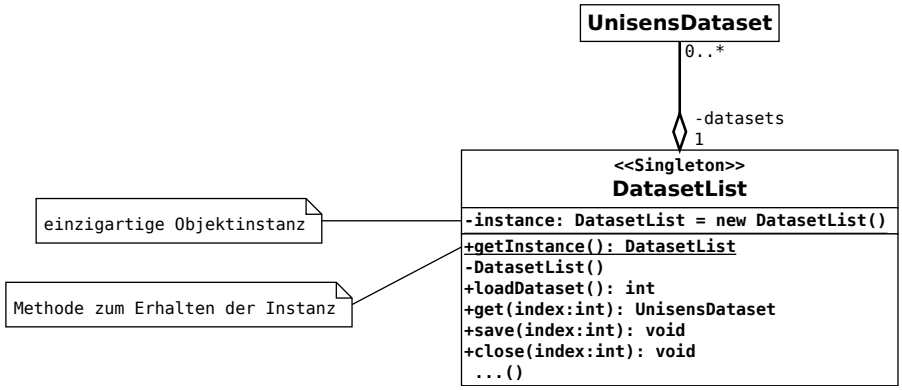

Abbildung 6.: Die Klasse `DatasetList` und das Singleton-Konzept. Aus Gründen der Übersicht sind nicht alle Funktionen aufgelistet (durch `...()` markiert).

Das Entwurfsmuster des Singletons ([10] Seite 127 bis 134) wird angewendet, wenn für eine Klasse nur genau ein Objekt instanziiert werden soll. Es stellt zusätzlich sicher, dass die geschaffene Instanz global im Programm erreichbar ist. Damit die Einzigartigkeit des Objektes dieser Klasse garantiert ist, ist der Konstruktor der Klasse als `private` deklariert. Somit kann keine neue Objektinstanz außerhalb der Klasse `DatasetList` erzeugt werden. Die Instanz selbst wird in der Klassenvariable `instance` gespeichert. Klassenvariablen werden in der Klasse selbst gespeichert und nicht für die einzelnen Objekte dieser Klasse angelegt. Von außen kann auf das konkrete Objekt über die öffentliche (`public`) Klassenfunktion `getInstance()` zugegriffen werden. Klassenfunktionen sind wie Klassenvariablen für die Klassen definiert, aber nicht für die Objekte selbst. Durch diese Singletonimplementierung ist sichergestellt, dass

- die Objektinstanz `instance` der Klasse `DatasetList` innerhalb des Programms einzigartig ist und
- diese Instanz global im Programm über die Funktion `DatasetList.getInstance()` erhältlich ist.

Die Objekte der `UnisensDataset`s werden in einer nicht öffentlichen Liste `datasets` gespeichert. Außerhalb der Klasse kann auf die Datensatzobjekte über einen Index und die `get()`-Funktion zugegriffen werden. Der entsprechende Index wird durch die `load()`-Funktion zurück gegeben. Die `loadDataset()`-Funktion öffnet einen Dialog zum öffnen

einer Datei in dem der Benutzer die Headerdatei `unisens.xml` eines Datensatzes auswählen kann. Dieser Dialog gibt den absoluten Pfad der gewählten Datei zurück, welcher mit denen aller bereits geladenen Datensätzen in der Liste `datasets` verglichen wird. Wurde der vom Nutzer gewählte Datensatz bereits geladen, wird der Index dieses Datensatzes zurück gegeben. Ist der Datensatz jedoch unbekannt, wird er von der Funktion geladen und das `UnisensDataset`-Objekt in der Liste gespeichert. Der Index des neu erzeugten Objektes wird zurückgegeben. Durch diese Behandlung wird sicher gestellt, dass jeder Datensatz nur einmal geladen wird. Mit die Methoden `save()` und `close()`, die auch den Index eines Datensatzes erwarten, werden die Datensätze auf der Festplatte gespeichert bzw. geschlossen. Wird ein Datensatz geschlossen, so wird auch das entsprechende `UnisensDataset`-Objekt aus der `datasets`-Liste gelöscht.

4.2.4.3. Pufferung, Sortieren und Suchen der Klasse `AnnotationController`

Die Referenzimplementierung des Unisensformates sieht für das Lesen und Schreiben von Daten einen ungepufferten Ansatz vor. Das bedeutet sobald Daten vom Programm verändert werden, werden diese Änderungen im Datensatz und somit auf dem Speichermedium ausgeführt. Das hat zur Folge, dass Annotationen in der Reihenfolge in der sie erstellt wurden, in den Datendateien abgespeichert werden. Somit sind die Annotationen zeitlich unsortiert. Wenn nun eine Annotation eines bestimmten Zeitpunktes angezeigt werden soll, muss der gesamte Datensatz nach dieser Annotation durchsucht werden. Dieses Verhalten ist unerwünscht, da somit die Zugriffszeit während der Laufzeit, insbesondere bei großen Datensätzen, stark schwanken können. Dieses Problem wird durch eine gepufferte Implementierung der Klasse `AnnotationController` behoben. Die Annotationen werden beim Laden des Datensatzes in den Arbeitspeicher geladen und mit einem Quick-Sort-Algorithmus nach aufsteigenden Samplenummern sortiert. Haben mehrere Annotationen die gleichen Samplenummern, wird ihre ursprüngliche Reihenfolge beibehalten. Neu erstellte Annotationen werden in dieser Liste an den entsprechenden Stellen einsortiert. Bei der Speicherung der Daten werden die Annotationen in ihrer sortierten Reihenfolge in die Datei geschrieben.

Sowohl bei Lese- als auch Schreibzugriffen wird die Samplenummer der betreffenden Annotation gespeichert. Bei der Suche nach der entsprechenden Position einer gesuchten Annotation wird dann von dieser gespeicherten Samplenummer aus gestartet. Die Suche nach einer Samplenummer erfolgt sowohl vor- als auch rückwärts. Dieser Ansatz hat den Vorteil, dass eine gesuchte Position nach wenigen Suchschritten erreicht wird, da Annotationen durch den Benutzer nur verändert werden können die auch angezeigt werden. Somit befindet sich der Index des letzten Zugriffs immer in der Nähe der gesuchten Position.

Anmerkung zur Quick-Sort-Implementierung: Während Testläufen und Benutzung des Programmes kam es beim Laden von Datensätzen mit ca. 3.000 - 3.600 Annotationen zu Abstürzen des Programmes. Diese Abstürze wurden durch eine `StackOverflowException` verursacht. Der Grund dafür ist, dass die Implementierung des Quick-Sort-Algorithmus

auf Rekursion beruht und die `quicksort()`-Funktion sich selbst aufruft. Jeder Methodenaufruf wird auf dem sogenannten Java *Thread-Stack* abgelegt. Durch die Verschachtelung der Methodenaufrufe reicht der Speicherplatz (typischerweise 512kB) des *Thread-Stacks* nicht aus. Die Größe des *Thread-Stacks* kann beim Programmstart aber durch Parameterübergabe an die *Java-Virtual-Machine* verändert werden. Das Problem konnte mit dem Parameter `-Xss2m` behoben werden, wodurch die maximale Speichergröße der *Thread-Stacks* von 512kB auf 2MB angehoben wird. Der Parameter setzt sich aus dem Präfix `-Xss` und der gewählten Größe zusammen (`512k` steht für 512kB, `2m` für 2MB, usw.). Das Problem wurde für die genannte Datensatzgröße behoben, kann aber bei höherer Annotationsanzahl wieder auftreten. Es kann die Größe des *Thread-Stacks* weiter erhöht werden.

4.2.4.4. Direkter Zugriff auf Einzelwerte (*Values*) mit der Klasse `BufferedValueController`

Eine direkte Änderung einzelner Datenpunkte ist vom Unisensformat nicht vorgesehen. Daher ist für einen direkten Schreibzugriff auf die Daten eine eigene Implementierung notwendig. Für *Values*, d. h. Datenpunkte denen ein Zeitpunkt und ein Wert zugeordnet sind, ist diese Eigenschaft in der Klasse `BufferedValueController` implementiert. In Abb. 5 ist ersichtlich, dass `BufferedValueController` von der Klasse `ValueController` abgeleitet ist. Um die Funktionalität der Schreibzugriffe zu gewährleisten, werden die Daten, analog zur Handhabung in der Klasse `AnnotationController`, in einem Puffer `buffer` zwischengespeichert.

Die einzelnen Wert-Datenpunkte sind durch die interne Klasse `DataPoint` abstrahiert. Diese haben eine Samplenummer `sampleStamp` und einen Wert `data`. Der Datentyp der Werte kann frei gewählt werden, da die Klasse `DataPoint` als Template-Klasse implementiert ist. In der aktuellen Implementierung von `BufferdValueController` ist der Datentyp der Werte als `Double` festgelegt.

Die Datenpunkte sind im Puffer `buffer` nach ihren Samplenummern aufsteigend sortiert. Aufgrund der in Abschnitt 4.2.4.3 beschriebenen Probleme des Quick-Sort-Algorithmus und der Java-*Thread-Stack*-Größe ist die Sortierung auf andere Art und Weise implementiert. Die Daten werden beim anfänglichen Füllen des Puffers nicht erst aus der Datei gelesen und anschließend sortiert, sondern beide Schritte erfolgen gleichzeitig. Hierbei werden die Daten einzeln gelesen, anschließend die zugehörige Position aufgrund ihrer Samplenummer gesucht und letztendlich in den Puffer eingefügt. Die Positionssuche erfolgt, analog zur Implementierung in der Klasse `AnnotationController`, über einen Suchindex. Liegen die Daten schon sortiert in der Datei vor, so wird effektiv nur der Suchindex erhöht und der Datenpunkt in den Puffer eingefügt. Dadurch wird der Zeitverbrauch durch die ständige „Suche“ während des Auffüllens des Puffers verringert. Dieser Ansatz beruht im Gegensatz zur Implementierung in `AnnotationController` nicht auf Rekursion und führt zu keiner tiefen Verschachtelung von Methodenaufrufen.

Neben der Implementierung von Methoden zum Hinzufügen, Löschen und Verändern

von Datenpunkten müssen auch die schon von `ValueController` bereitgestellten Methoden zum Lesen der Datenpunkte und Speichern der Datendatei an den gepufferten Ansatz angepasst werden. Hierfür werden die `getDataPoints()` und die `saveImpl()` Funktionen durch `BufferedValueController` überschrieben. Bei einem Lesezugriff mit `getDataPoints()` werden die Daten aus dem Puffer gelesen und nicht mehr direkt aus der Datendatei. Beim Speichervorgang wird die bereits bestehende Datendatei als *Backup* gespeichert, die neue Datei erzeugt und die Daten in diese neu erzeugt Datei geschrieben. Erst wenn der Schreibvorgang erfolgreich beendet ist, wird die alte Backupdatei gelöscht. Durch diese Herangehensweise ist garantiert, dass durch Fehler beim Schreiben keine Daten verloren gehen.

4.2.4.5. Observer Prinzip zur Reaktion auf Datenänderung

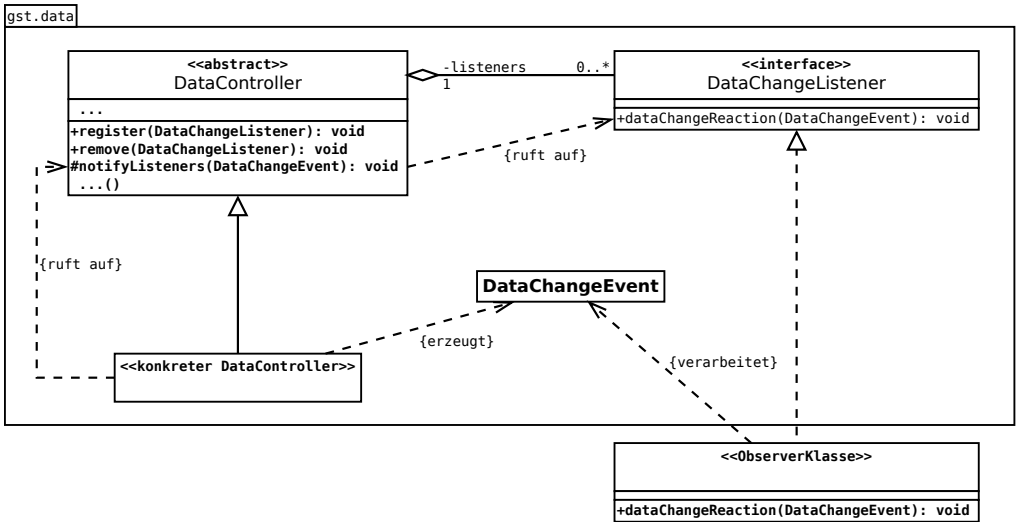

Abbildung 7.: Observer-Prinzip mit durch `DataChangeListener`

An mehreren Stellen im Programm muss auf eine Veränderung von Daten reagiert werden . So muss die Ansicht eines Signalverlaufs angepasst werden oder eine Signalverarbeitungsfunktion ihr Ausgabesignal verändern. Um dieses dynamische Reaktionen zu ermöglichen wird das *Observer*-Entwurfsmuster ([10] Seiten 293 bis 303) angewendet. Eine Klasse die an einer möglichen Veränderung von Daten interessiert ist muss zwei Bedingungen erfüllen:

- Das Interface `DataChangeListener` implementieren.
- Das interessierte Objekt bei dem zu beobachtenden `DataController` registrieren.

Das Interface `DataChangeListener` definiert nur eine Methode: `dataChangeReaction()`. Sie hat keinen Rückgabewert und erhält als Parameter ein `DataChangeEvent`. Das Objekt der Klasse `DataChangeEvent` enthält alle Informationen über die aufgetretene Datenänderung inklusive einer Referenz des entsprechenden `DataControllers`. Somit ist es für

`DataChangeListener`-implementierende Klassen nicht notwendig die `DataController` zu speichern, von denen sie Mitteilungen über Veränderungen erhalten möchten. Die Registrierung eines Objektes bei einem `DataController` erfolgt mit der Methode `register(DataChangeListener)` Die Speicherung der Objekte und die Übergabe der `DataChangeEvents` an alle *Observer* ist komplett in der Klasse `DataController` implementiert.

Die jeweilige `DataController`-Implementierung muss zwei Aufgaben übernehmen:

1. Bei einer Änderung der Daten ein entsprechendes `DataChangeEvent` mit den notwendigen Informationen erzeugen.

2. Das erzeugte Objekt über die Methode `notifyListeners()` der Superklasse `DataController` an die *Observer* weiterreichen.

Diese zwei Schritte werden beispielsweise durch die Klassen `AnnotationController` und `BufferedValueController` bereits implementiert, da beide Klassen die Änderung von Daten ermöglichen. Intern werden bei einem Aufruf von `notifyListeners()` bei allen registrierten *Observern* die Methode `dataChageReaction()` aufgerufen und das `DataChangeEvent` weitergegeben.

4.2.4.6. [WIP]Tests des `data`-Paketes

4.3. Signalverarbeitung im Paket `signalprocessing`

4.3.1. Überblick über das `signalprocessing`-Paket

Die Signalverarbeitung der Software wird durch das Paket `signalprocessing` ermöglicht. Der Schwerpunkt bei der Implementierung dieses Paketes liegt in der Umsetzung einer erweiterbaren Plattform für Signalverarbeitungsfunktionen. Die Bereitstellung und Implementierung konkreter Signalverarbeitungsfunktion spielen eine untergeordnete Rolle. Die Verwendung des Paketes soll an dieser Stelle nur beispielhaft erfolgen. Als Musterbeispiel für die Funktionalität der Signalverarbeitung wird die Berechnung von RR-Zeitreihen herangezogen. Im Detail soll aus den Zeitpunkten von Annotationen eines Kanals eine Zeitreihe der zeitlichen Abstände der gegebenen Annotationen automatisiert erstellt werden. Es wird bewusst eine Signalverarbeitungsfunktion geringer Komplexität implementiert um den Fokus auf das allgemeine Zusammenspiel der beteiligten Klassen und nicht auf den Algorithmus selbst zu legen.

Zur Berechnung der Schlag-zu-Schlag-Intervalle werden folgende Schritte ausgeführt:

1. Auswahl eines Annotationskanals durch den Benutzer

2. Berechnung der Zeitpunkte der Annotationen aus ihren Samplenummern und der (virtuellen) Abtastfrequenz

3. Berechnung der Differenz der Zeitpunkte aufeinander folgender Annotationen

4. Speichern der errechneten Zeitdifferenzen in einem durch den Benutzer benannten Signal

Um die Darstellung der Zeitreihe zu ermöglichen, wird jeder errechneten Differenz ein Zeitpunkt zugewiesen. Die errechneten Datenpunkte werden als `Value` gespeichert. Der Wert V_{value} ist die Differenz der Zeitpunkte $t_{n+1} - t_n$ zweier Annotationen A_n und A_{n+1}. Der Zeitpunkt des `Values` t_{value} wird auf den Zeitpunkt t_{n+1} der zweiten Annotation A_{n+1} festgelegt. Die Implementierung dieses Algorithmus ist im Paket `rrcalc` mit den zwei Klassen `RRCalculator` und `LiveRRCalculator` realisiert.

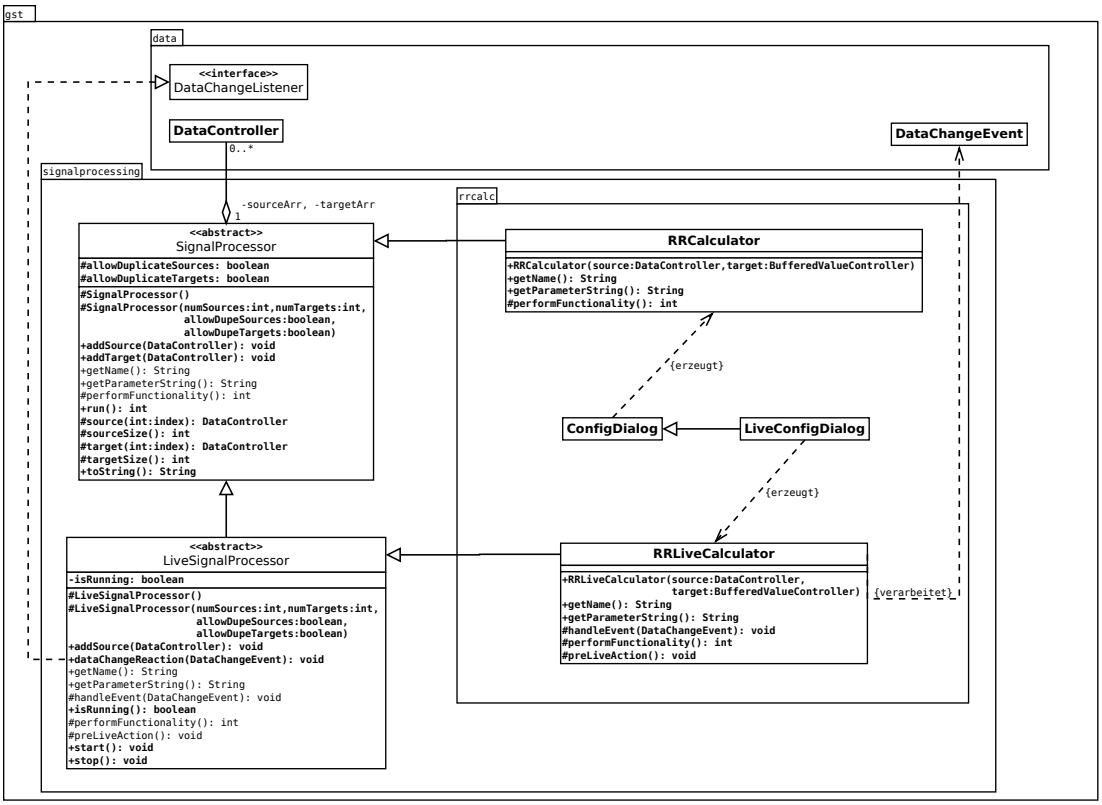

Abbildung 8.: Übersicht über das `signalprocessing`-Paket

In dem `signalprocessing`-Paket wird zwischen zwei Arten der Ausführung von Signalverarbeitungsfunktionen unterschieden (siehe Abb. 8). Die erste Art ist das einmalige Anwenden einer Verarbeitungsfunktion auf ein oder mehrerer Signale und ist im Abschnitt 4.3.2 erörtert. Ein Anwendungsbeispiel dafür sind Filterfunktionen oder die Berechnung von RR-Intervallen aus bereits annotierten EKG-Daten. Die zweite Ausführungsart ist eine kontinuierliche Verarbeitung von Signalen. Hierbei wird auf eine Veränderung des zu verarbeitenden Signals reagiert und die Ausgabe entsprechend angepasst. Das implementierte Beispiel dafür ist die Berechnung und Anzeige der RR-Zeitreihe, während der Benutzer des Programms Annotationen verändert bzw. korrigiert. Diese zweite Verarbeitungsart ist im Abschnitt 4.3.3 weiter unten beschrieben.

4.3.2. Einmalige Signalverarbeitung durch `SignalProcessor`

Die Klasse `SignalProcessor` stellt einen Prototypen einer allgemeinen Signalverarbeitungsfunktion dar. Objekte dieser Klasse besitzen ein oder mehrere Eingangssignale und ebenso ein oder mehrere Ausgangssignale. `SignalProcessor` stellt eine einheitliche Schnittstelle für das Hinzufügen und Entfernen von Ein- und Ausgangssignalen zur Verfügung und speichert die notwendigen Objektreferenzen dieser Signale. Da sie aber nicht die konkrete Funktionalität der Signalverarbeitungsfunktion implementieren kann, ist `SignalProcessor` als abstrakte Klasse implementiert.

`SignalProcessor` übernimmt drei wesentliche Aufgaben implementierter Signalverarbeitungsfunktionen:

1. Bereitstellen einer einheitlichen Schnittstelle mit der alle Signalverarbeitungsfunktionen genutzt werden können
2. Methoden zum Hinzufügen, Entfernen und Speichern der Objektreferenzen der Ein- und Ausgangssignale
3. einheitliche Dokumentation des Signalursprungs für die Ausgabesignale

In den Methoden zum Hinzufügen von Eingangs- und Ausgangssignalen wird sichergestellt, dass ein und dasselbe Signal nicht mehrfach als Ein- bzw. Ausgangssignal verwendet wird. Diese Funktionalität kann optional durch die implementierende Unterklasse verhindert werden. Wird die Funktion `run()` aufgerufen, wird die durch die Unterklasse implementierte Signalverarbeitung durchgeführt. Zusätzlich dazu wird durch `SignalProcessor` die Signalquelle der Ausgangssignale vermerkt. Es werden bei der Ausführung einer Signalverarbeitungsfunktion die Programmversion, der Name der verarbeitenden Funktion und auch die genutzten Parameter in den Ausgangssignalen als Quelle vermerkt. Damit wird erreicht, dass die Entstehungsgeschichte von verarbeiteten Signalen automatisiert aufgezeichnet wird.

In Abb. 8 sind drei abstrakten Methoden ersichtlich, die durch eine konkretisierende Unterklasse implementiert werden müssen. Die Methoden `getName()` und `getParameterString()` dienen der Dokumentation der Verarbeitungsschritte und geben einen `String` mit Funktionsnamen (und eventuell notwendiger Versionsnummer) bzw. die genutzten Parameter der Funktion zurück. Die dritte Methode `performFunctionality()` soll die konkrete Verarbeitung durchführen und die Ausgangssignale entsprechende verändern. Durch diese Umsetzung kann sich ein zukünftiger Entwickler auf die Implementierung des Algorithmus konzentrieren. Alle organisatorischen Nebenabläufe sind durch `SignalProcessor` abgedeckt.

4.3.3. Kontinuierliche Verarbeitung von Signalen durch `LiveSignalProcessor`

Aus Abb. 8 ist ersichtlich, dass die Klasse `LiveSignalProcessor` von `SignalProcessor` abgeleitet ist. Somit wird auch die Funktionalität der einmaligen Signalverarbeitung mit

übernommen. Zusätzlich dazu wird der Funktionsumfang dahingehend erweitert, dass eine kontinuierliche Signalverarbeitung ausgeführt werden kann. Um dieses dynamische Verhalten zu ermöglichen wird durch die Klasse `LiveSignalProcessor` das *Interface* `DataChangeListener` implementiert. Der `LiveSignalProcessor` meldet sich bei allen Eingangssignalen als *Observer* an und erhält dadurch Informationen über veränderte (Eingangs-) Daten. Die gewonnenen Informationen werden, sofern der `LiveSignalProcessor` gestartet ist, an die implementierende Unterklasse weitergegeben, die diese dann verarbeiten kann. Informationen veränderter Daten werden verworfen, wenn die kontinuierliche Signalverarbeitung gestoppt ist. Der Zustand ob die Verarbeitung ein- oder ausgeschaltet ist, wird in der `boolean`-Variable `isRunning` gespeichert. Der Zustand kann über die Methoden `start()` und `stop()` ein- bzw. ausgeschaltet werden.

Eine implementierende Unterklasse muss neben den im Abschnitt 4.3.2 genannten Methoden `getName()`, `getParameterString()` und `performFunctionality()` noch zwei weitere Methoden implementieren: `preLiveAction()` und `handleEvent()`. `handleEvent()` erhält als Parameter die auftretenden `DataChangeEvent`s und soll diese verarbeiten. Die Methode `preLiveAction()` wird durch die Klasse `LiveSignalProcessor` aufgerufen, wenn die kontinuierliche Signalverarbeitung gestartet wird. Mit diesem Zwischenschritt soll ermöglicht werden, dass eventuell notwendige Vorbereitungen ausgeführt werden können bevor das erste `DataChangeEvent` weitergegeben wird. In dem implementiertem Beispiel `RRLiveCalculator` wird dieser Methodenaufruf genutzt, das Ausgabesignal für das entsprechende Eingangssignal zu berechnen.

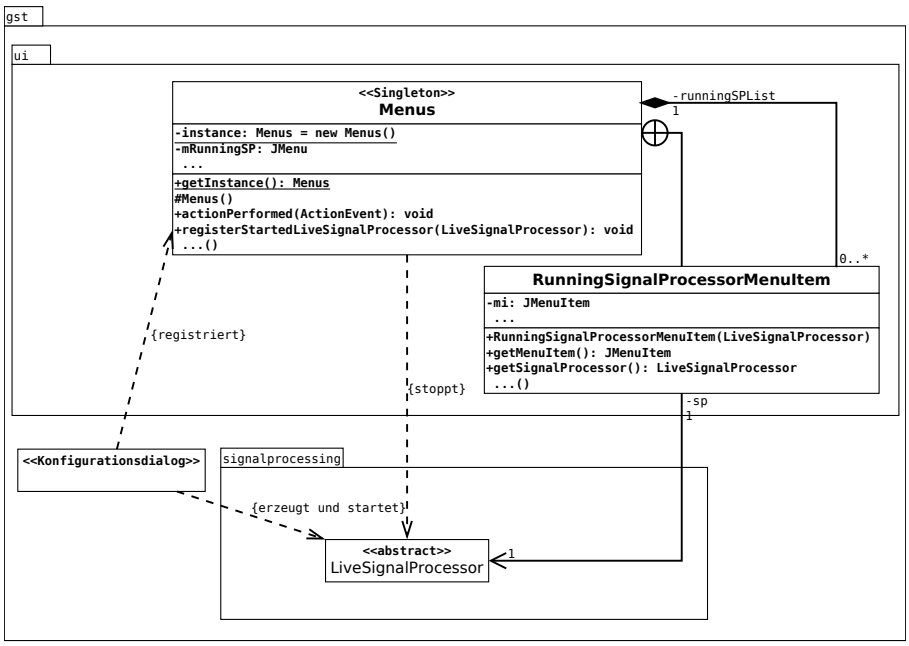

Abbildung 9.: Registrierung eines `LiveSignalProcessor`s im Menü

Damit gestartete kontinuierliche Signalverarbeitungsfunktionen durch den Anwender wieder gestoppt werden können, ist eine automatisierte Menüänderung implementiert (sie-

he Abb. 9). Durch eine Dialogbox, die über das Menü vom Anwender aufgerufen wird, werden die notwendigen Parameter der Signalverarbeitungsfunktion eingestellt. Die Dialogbox erzeugt das `LiveSignalProcessor`-Objekt und übergibt die gewählten Parameter. Nachdem der `LiveSignalProcessor` gestartet wurde, wird er im Menü mit der Methode `registerStartedLiveSignalProcessor()` registriert. Diese Registrierung hat zur Folge:

- Ein Menüeintrag mit dem Namen des gestarteten `LiveSignalProcessor` wird erzeugt und in das Menü eingefügt.
- Die laufende Signalverarbeitungsfunktion und der entsprechende Menüeintrag werden zusammen in einem `RunningSignalProcessorMenuItem`-Objekt gespeichert.

Wird der entsprechende Menüeintrag durch den Nutzer aufgerufen, wird die `stop()`-Methode des `LiveSignalProcessor`s aufgerufen und der Menüeintrag wieder entfernt.

4.3.4. Implementierung weiterer Signalverarbeitungsmethoden

Das Paket `signalprocessing` ist dafür ausgelegt, um um zusätzliche Signalverarbeitungsfunktionen einfach erweitert werden zu können. Die Implementierungen der zwei Hauptklassen `SignalProcessor` und `LiveSignalProcessor` als abstrakte Klassen ermöglicht ein schnelles einbinden neuer Funktionalitäten. Vier Schritte sind durch einen zukünftigen Entwickler auszuführen:

- Auswahl einer Superklasse abhängig davon ob eine einfache oder kontinuierliche Signalverarbeitung stattfinden soll.
- Implementierung einer Unterklasse die die gewünschte Funktionalität bereit stellt.
- Programmierung einer Dialogbox, die dem Benutzer des Programmes Einstellungen der Parameter erlaubt.
- Integration des Konfigurationsdialogs in das Menü des Programms.

Es ist empfohlen alle Klassen in einem eigenem Paket einzubinden um Namenskollisionen zu unterbinden. Das neue Paket kann und sollte als im bestehenden Paket `gst.signalprocessing` angelegt werden, damit die Gliederung der Pakete innerhalb des Programms nach Funktionalität erhalten bleibt. Der Konfigurationsdialog soll nach der Erstellung und Initialisierung des konkreten Signalverarbeitungsobjektes die Verarbeitungsroutine starten (durch den Methodenaufruf von `run()`). Wie schon im Abschnitt 4.3.3 erwähnt, muss die kontinuierliche Verarbeitung durch die Methode `start()` ausgelöst werden. Das Abbrechen der kontinuierlichen Signalverarbeitung durch den Nutzer ist in der vorgelegten Implementierung bereits integriert.

Abbildung 10.: Klassen der grafischen Elemente: I `Menus`, II `Toolbar`, III `SignalPanel`, IV `SignalView`, V `StatusBar`

4.4. [WIP]Benutzerführung

4.4.1. Elemente der grafischen Benutzeroberfläche

In Abb. 10 sind die einzelnen Bestandteile der GUI hervorgehoben. Interaktionen zwischen dem Benutzer und dem Programm erfolgen über fünf Hauptkomponenten:

- Menüzeile durch die Klasse `Menus` repräsentiert
- Werkzeugleiste als Klasse `Toolbar`
- Statuszeile zum Anzeigen von Informationen mithilfe der Klasse `StatusBar`
- einzelne Diagramme als `SignalViews` zur Anzeige der Signalverläufe (im folgendem Signalansichten genannt)
- Zusammenfassung aller Signalansichten auf dem flächengrößtem Bestandteil, dem `SignalPanel`

Während das Menü, die Werkzeugleiste und die Signalansichten Eingaben von dem Benutzer verarbeiten, dient die Statuszeile ausschließlich der Präsentation von Informationen. Die grafische Darstellung der Klasse `SignalPanel` bleibt für den Nutzer größtenteils verborgen, da es sich hierbei um ein organisatorisches Programmelement handelt (siehe dazu Abschnitt 4.4.3).

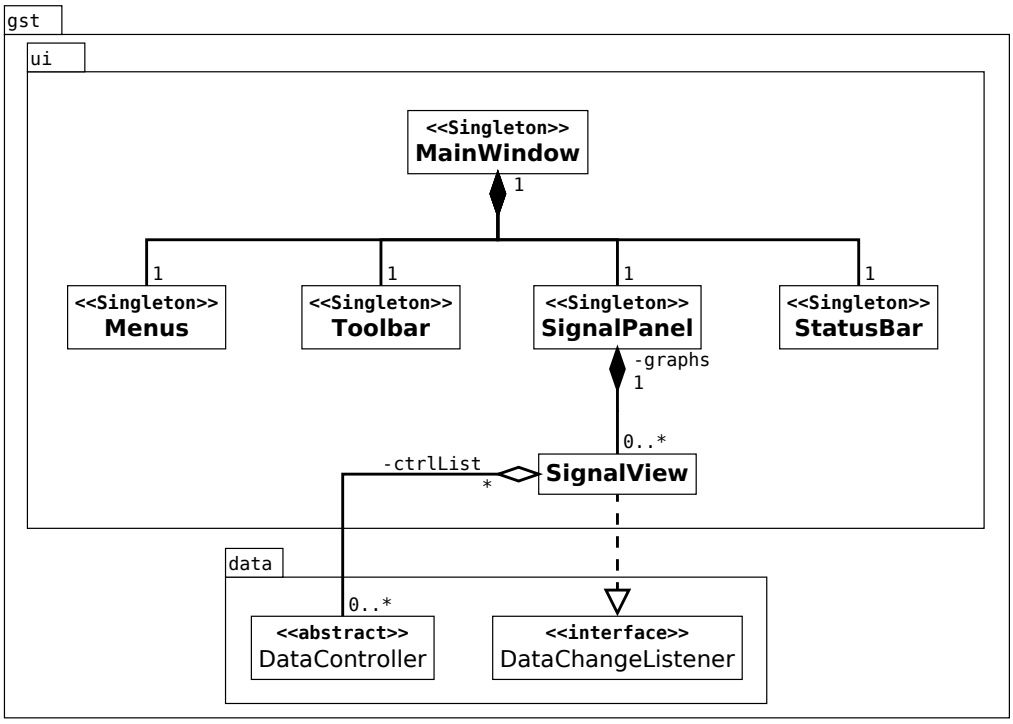

Abbildung 11.: Übersicht über das `ui`-Paket

Ein Übersicht die Klassenhierarchie ist in Abb. 11 dargestellt. Bis auf die `SignalView`-Klasse sind alle grafischen Komponenten nach dem Singleton-Entwurfsmuster implementiert (vgl. Abschnitt 4.2.4.2). Durch diese Entscheidung ist einerseits garantiert, dass die grafischen Elemente nur einmal in der Programminstanz vorkommen und außerdem kann auf sie vereinfacht zugegriffen werden (ähnlich globalen Variablen).

4.4.2. Visualisierung der Signalverläufe

4.4.2.1. Die genutzte *JFreeChart*-Bibliothek

Für die Darstellung der Signalverläufe in Diagrammen wird die *Java*-Bibliothek *JFreeChart* genutzt. *JFreeChart* wird, ebenso wie das *Unisens*-Format, unter der LGPL zur freien Nutzung angeboten. Mithilfe der *JFreeChart*-Bibliothek können Diagramm unterschiedlicher Art zur Visualisierung von Daten dargestellt werden. Neben der Vielzahl der unterstützten Diagrammtypen werden zusätzlich die dynamische veränderbare Graphen unterstützt. Die umfangreichen Einstellmöglichkeiten bei der Anpassung der Diagramme ermöglichen eine nahezu nahtlose Integration der Bibliothekselemente in das bestehende grafischen Gesamtbild. Zudem bietet die *JFreeChart*-Bibliothek, zusätzlich zur grafischen Veranschaulichung von Daten, auch grafische Elemente um Bereiche oder bestimmte Punkte in einem Diagramm gesondert hervorheben zu können.

4.4.2.2. Die Wrapper-Klasse `SignalView`

Für die Integration der Diagramme in die GUI als Signalansichten existiert die Klasse `SignalView`. Objekte der Klasse haben die Aufgabe, die durch die `DataController` bereit gestellten Daten dem Benutzer grafisch zu veranschaulichen. `SignalView` ist eine Unterklasse der Klasse `ChartPanel` aus *JFreeChart* und bildet somit das Verbindungselement zwischen dem Programm und der genutzten Bibliothek. Ein `SignalView`-Objekt ist somit grafisch ein Diagramm auf der Benutzeroberfläche (vgl. mit Abb. 10). Die Darstellung der Daten erfolgt durch zwei Varianten. Darstellung von kontinuierlich abgetasteten Signalen (*Signals*) und Einzelwertezeitreihen (*Values*) erfolgt als Liniendiagramm der Werte über der Zeit. Die grafische Präsentation von Annotationen ist durch senkrechte Striche in den Diagrammen realisiert.

Um die Daten darzustellen muss der entsprechende `DataController` der Signalansicht hinzugefügt werden. Es wird sichergestellt, dass jeder `DataController` nur genau einmal ein und demselben `SignalView`-Objekt hinzugefügt werden kann. Jeder `SignalView` kann beliebig viele `DataController`-Objekte speichern und darstellen. `SignalView` implementiert das Interface `DataChangeListener` des `data`-Paketes und ist daher in der Lage dynamisch auf Änderungen von Daten zu reagieren. Wird z. B. eine Annotation durch den Benutzer dem Datensatz hinzugefügt oder verändert, so wird in jeder Signalansicht diese Änderung auch grafisch dargestellt (sofern der entsprechende Kanal dargestellt wird).

Neben der grafischen Darstellung der Daten bietet die `SignalView`-Klasse weitere optische Rückmeldung für den Nutzer. So wird die aktuelle Mausposition innerhalb eines Diagramms, in Bezug auf die Zeitachse (Abszisse) durch eine gestrichelte senkrechte Linie hervorgehoben. Die Hervorhebung erfolgt nicht nur in dem Diagramm, über dem der Mauszeiger sich gerade befindet, sondern in allen dargestellten Diagrammen. Zudem wird die Signalansicht, über der sich der Mauszeiger befindet, visuell durch Veränderung der Hintergrundfarbe markiert. Damit ist für den Nutzer ersichtlich, welches Diagramm seine Eingaben empfängt. Die Verarbeitung der Maus- und Tastatureingabe des Nutzers durch zwei innere Klassen `SignalViewMouseAdapter` und `SignalViewKeyAdapter`. Details zur Behandlung der Benutzereingabe sind im Abschnitt 4.4.4 weiter unten erörtert.

Das genutzte *JFreeChart*-Paket bietet die Möglichkeit einem Diagramm eine Datenreihe mit mehreren Tausend Datenpunkten zuzuordnen. Die Wahl des darzustellenden Ausschnitts kann durch den Benutzer durchgeführt werden und wird durch die Bibliothek komplett unterstützt. Die Zeit, die die Bibliothek benötigt um ein Diagramm mit all seinen Komponenten darzustellen, sinkt mit der Anzahl der Datenpunkte. Obwohl von *JFreeChart* ein Renderer (das Objekt dass die Datenpunkte in dem Diagramm darstellt) für viele Datenpunkte angeboten wird, sinkt die Darstellungsgeschwindigkeit bei 3000 oder mehr Datenpunkten so stark, dass die Aktualisierung der Anzeige nicht mehr flüssig erscheint. Das Programm benötigt zu viel Zeit um die Diagramme auf dem Bildschirm zu zeichnen, dass die Verarbeitung der Benutzereingabe und die Darstellung merklich verzögert erfolgen. Dieses Problem wird in der eingereichten Implementierung umgangen,

indem den Diagramm-Objekten eine begrenzte Anzahl an Datenpunkten übergeben wird. Die Zahl der dargestellten Datenpunkte ist auf die Breite in Pixeln der Diagramme festgelegt. Damit bleibt die Darstellung flüssig aber es entstehen visuelle Artefakte die durch das Downsampling bedingt sind. Diese Darstellungsartefakte nehmen mit Erhöhung der Zoomstufe ab, da umso mehr Datenpunkte dargestellt werden, je kleiner der Signalausschnitt wird (größere Pixelauflösung pro Zeiteinheit).

4.4.2.3. Dynamische Reaktion auf Veränderung der Daten

Da durch die Implementierung in der Klasse `SignalView` die Darstellung der Daten von den Daten entkoppelt ist, resultiert eine Änderung der Daten nicht direkt in einer Änderung der Anzeige. Es ist aber wünschenswert diese Veränderungen dem Nutzer zu visualisieren. Beispielsweise möchte der Benutzer eine neu hinzugefügte Annotation in allen Signalansichten dargestellt bekommen. Würde dies nicht geschehen, kann beim Nutzer Zweifel aufkommen, ob die Annotation erzeugt wurde oder nicht. Um diese dynamische Aktualisierung der Anzeige zu ermöglichen, wird durch die Klasse `SignalView` das Interface `DataChangeListener` implementiert (siehe Abb. 11). Sofern ein `DataController`-Objekt einer Signalansicht zugeordnet ist, wird das der Ansicht entsprechende `SignalView`-Objekt bei dem `DataController` als `DataChangeListener` registriert. Ändern sich die Daten eines `DataController`s, wird diese Veränderung dem `SignalView`-Objekt mitgeteilt und es wird entsprechend das Diagramm aktualisiert. Damit wird dem Nutzer eine visuelle Rückmeldung bei Änderungen der Daten geboten.

4.4.3. [WIP]Größenbestimmung und Positionierung der Signalansichten durch `SignalPanel`

- organisationseinheit

4.4.4. [WIP]Verarbeitung der Benutzereingabe im Paket `ui`

4.4.4.1. [WIP]Koordiniertes Zoomen und Scrollen durch `SignalPanel`

- Observer-Prinzip - zwei eigene Managerklassen (Unabhängigkeit) `ScrollLockManager`, `ZoomLockManager`

4.4.4.2. [WIP]Verarbeitung der Benutzereingaben zur Steuerung der Ansicht

- Einfügen und Entfernen von Datenkanälen - Steuerung der Ansicht über Zeitfenster - Legende anzeigen - `SignalView` stellt dar - Datenzugriff über abstrakte Definition von `DataController` - Subklassen verarbeiten Maus- und Tastatureingabe

4.4.4.3. [WIP]Verarbeitung und Veränderung von Annotationen

- Einfügen von Annotationen über `AnnotationManager` - Dreiecksbeziehung von `AnnotationController`, `AnnotationList` und `AnnotationManager` erläutern - Suchalgorithmus zum

finden der aktuellen Annotation - Änderung wird durch die Implementierung vom Interface `DataChangeListener` automatisiert aktualisiert

5. Validierung

– NOTIZ: Speicherplatzbedarf bei speicherung in `int16` und `double` – Skripte zur Wandlung Matlab <–> Unisens

5.1. Erfüllung der Anforderungen

5.2. Evaluation der Nutzeroberfläche

5.3. Validierung anhand der Annotation von fetalen EKG-Daten

5.4. Validierung mittels der Annotationüberprüfung von ...

6. Diskussion

6.1. Bewertung der Evaluation

6.2. Ausblick

6.3. Grenzen

Literaturverzeichnis

[1] Andreotti, Fernando: *Extraction of the Fetal ECG from Electrocardiographic Long-term Recordings*, Technische Universität Dresden, Diplomarbeit, August 2011

[2] Barnes, David J. ; Kölling, Michael: *Objects first with Java.* Pearson Education, Inc., 2003

[3] Barnes, David J. ; Kölling, Michael: *Objektorientierte Programmierung mit Java.* Pearson Education Deutschland GmbH, 2003. – Übersetzung der englischen Originalausgabe [2]

[4] Brügge, Bernd ; Dutoit, Allen H.: *Object-Oriented Software Engineering. Using UML, Patterns and Java.* 2. Pearson Education, Inc., 2004

[5] Brügge, Bernd ; Dutoit, Allen H.: *Objektorientierte Softwaretechnik. mit UML, Entwurfsmustern und Java.* 2. Pearson Education, Inc., 2004. – Überstzung der englischen Originalausgabe [4]

[6] Chlebek, Paul: *User Interface-orentierte Softwarearchitektur.* Friedrich Vieweg & Sohn Verlagsgesellschaft mbH, 2006

[7] Cooper, Alan ; Reimann, Robert ; Cronin, David: *About Face 3. The Essentials of Interaction Design.* Wiley Publishing, Inc., 2007

[8] Cooper, Alan ; Reimann, Robert ; Cronin, David: *About Face. Interface und Interaction Design.* Hüthig Jehle Rehm GmbH, 2010. – Übersetzung der amerikanischen Originalausgabe [7]

[9] Free Software Foundation Inc.: *GNU Lesser General Public License.* [Online]. `http://www.gnu.org/licenses/lgpl.html`. Version: Juni 2007. – letzter Aufruf: 28.11.2012

[10] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns. Elements of Reuseable Object-Oriented Software.* Addison-Wesley Publishing Company, 1995

[11] Gilbert, David: *The JFreeChart Class Library Developer Guide.* [Online]. `http://www.scribd.com/doc/82678249/jfreechart-1-0-14-A4`. Version: Februar 2007. – letzter Aufruf: 28.11.2012

[12] Gilbert, David: *JFreeChart Homepage*. [Online]. `http://www.jfree.org/jfreechart/`. Version: November 2012. – letzter Aufruf: 28.11.2012

[13] Jacobson, Ivar ; Booch, Grady ; Rumbaugh, James ; Shanklin, J. C. (Hrsg.): *The Unified Software Development Process*. Addison-Wesley Longman Inc., 1999

[14] Kirst, Malte ; Ottenbacher, Jörg ; Nedkov, Radoslav: UNISENS - Ein universelles Datenformat für Multisensordaten. In: *Workshop - Biosignalverarbeitung* Universität Potsdam, 2008, S. 106 – 108

[15] Ottenbacher, Jörg ; Kirst, Malte: *Unisens Dokumentation*. [Online]. `http://unisens.org/downloads/documentation/Unisens-Dokumentation.pdf`. Version: 2.0, Februar 2010. – letzter Aufruf: 28.11.2012

[16] Rupp, Chris ; Queins, Stefan ; Zengler, Barbara: *UML 2 glasklar*. Carl Hanser Verlag, 2007

[17] Santiago, Marcos C.: *Processing of abdominal recordings by Kalman filters*, Technische Universität Dresden, Diplomarbeit, April 2012

[18] Schäfer, Steffen: *Objektorientierte Entwurfsmethoden. Verfahren zum objektorientierten Softwareentwurf im Überblick*. 1. Addison-Wesley Publishing Company, 1994

[19] Schlögl, A.: An Overview on data formats for biomedical signals. In: Dössel, O. (Hrsg.) ; Schlegel, W.C. (Hrsg.): *IFMBE Proceedings 25/IV*, 2009, S. 1557–1560

[20] Sommerville, Ian: *Software Engineering*. 6. Pearson Education, Inc., 2001. – Übersetzung der englischen Originalausgabe [21]

[21] Sommerville, Ian: *Software Engineering*. 6. Pearson Education, Inc., 2001

[22] Starke, Gernot: *Effektive Software-Architekturen*. Carl Hanser Verlag, 2002

[23] Varri, A. ; Kemp, B. ; Penzel, T. ; Schlogl, A.: Standards for biomedical signal databases. In: *Engineering in Medicine and Biology Magazine, IEEE* 20 (2001), Mai/Juni, Nr. 3, S. 33 –37. – ISSN 0739–5175

[24] Wang, Yibao ; Liu, Yang ; Lu, Xudong ; An, Jiye ; Duan, Huilong: A general-purpose representation of biosignal data based on MFER and CDA. In: *Biomedical Engineering and Informatics (BMEI), 2010 3rd International Conference on* Bd. 2, 2010, S. 689 –693

[25] Zaunseder, Sebastian ; Andreotti, Fernando ; Cruz, Marco ; Stepan, H. ; Schmieder, C. ; Malberg, Hagen ; Jank, A.: Fetal QRS detection by means of Kalman filtering and using the Event Synchronous Canceller. In: *7th International Workshop on Biosignal Interpretation*. Como, Italy, Juli 2012

[26] Zhai, Cui: *Analyse der Eigung verschiedener Messorte für die kamerabasierte Erfassung der Herzrate*, Technische Universität Dresden, Diplomarbeit, 2012

A. UML Dokumentation

B. Daten CD

Inhalt

./Diplomarbeit elektronische Form dieser Diplomarbeit

./Diplomarbeit/src LaTeX-Quelltext dieser Diplomarbeit

./Programm Quellcode des in dieser Arbeit umgesetzten Programms

./Literatur gesammelte Literatur

C. Programm kompilieren mit *Eclipse*

Nachfolgende Schritte sollten ausgeführt werden, damit das erstellte Programm mit der Entwicklungsumgebung *Eclipse* kompiliert werden kann.

1. Importieren der Verzeichnisstruktur in den *Workspace*: Über den Menüpunkt *File*→*Import...* den Import-Dialog aufrufen. Anschließend den Unterpunkt *General*→*File System* auswählen und mit *Next* fortfahren. Dann das auf dem Datenträger enthaltene Eclipse-Verzeichnis auswählen und mit *Finish* bestätigen.

2. Hinzufügen der notwendigen Bibliotheken: Aufrufen des Dialogs zum Ändern der Projekteinstellungen über den Menüpunkt *Project*→*Preferences*. Dabei muss sichergestellt werden, dass das GST-Projekt im *Package Explorer* ausgewählt ist. Unter *Java Build Path*→*Libraries* folgende Einstellungen vornehmen/überprüfen.

    1. Java Runtime Environment System Library: *Add Library...*→*JRE System Library*→*Next*→*Workspace default JRE*→*Finish*

    2. Unisens Bibliotheken: *Add JARs...*→*GST/lib/unisens-2.0.1297/**org.unisens.jar*** sowie *GST/lib/unisens-2.0.1297/**org.unisens.ri.jar***

    3. JCommon und JFreeChart: *Add JARs...*→*GST/lib/JFreeChart-1.0.14/**jcommon-1.0.17.jar*** und *GST/lib/JFreeChart-1.0.14/**jfreechart-1.0.14.jar***