BCI-Server

Inhaltsverzeichnis:

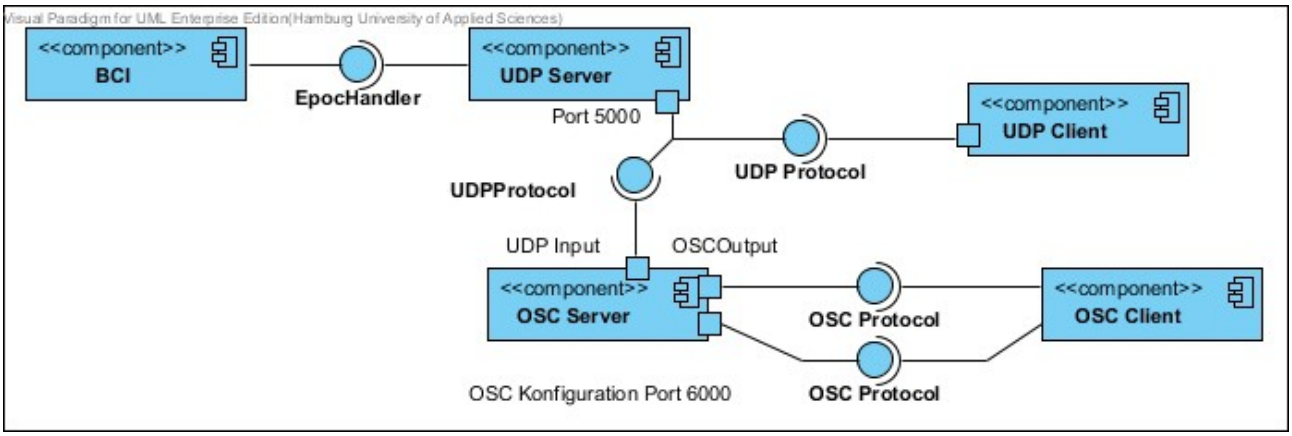
1. Einleitung

1.1 Aufgabe des BCI-Servers

Der BCI Server wurde dafür entwickelt um die Daten des BCI auf andere Prozesse zu verteilen, unabhängig von deren Betriebssystem und Sprache, es werden momentan nur das versenden der EmoStates und der RawDatasamples unterstützt.
Dafür wird OSC als Kommunikationsprotokoll verwende, welches auf UDP aufbaut.

1.2 Aufbau

Das Programm besteht aus zwei Prozessen. Einen grundlegenen UDP Server mit eigenen Protokoll, geschrieben in C++, welcher entfernten Zugriff auf die Daten realisiert und den OSC Server welcher erweiterte Funktionalitäten liefert, geschrieben in Java. Die Kommunikation zwichen den den beiden Prozessen erfolg über ein eigenes UDPProtokoll



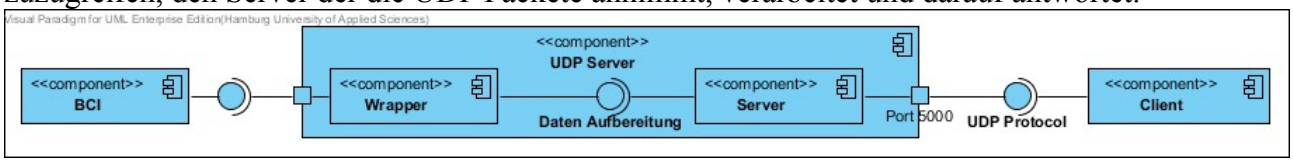
2. UDP Server:

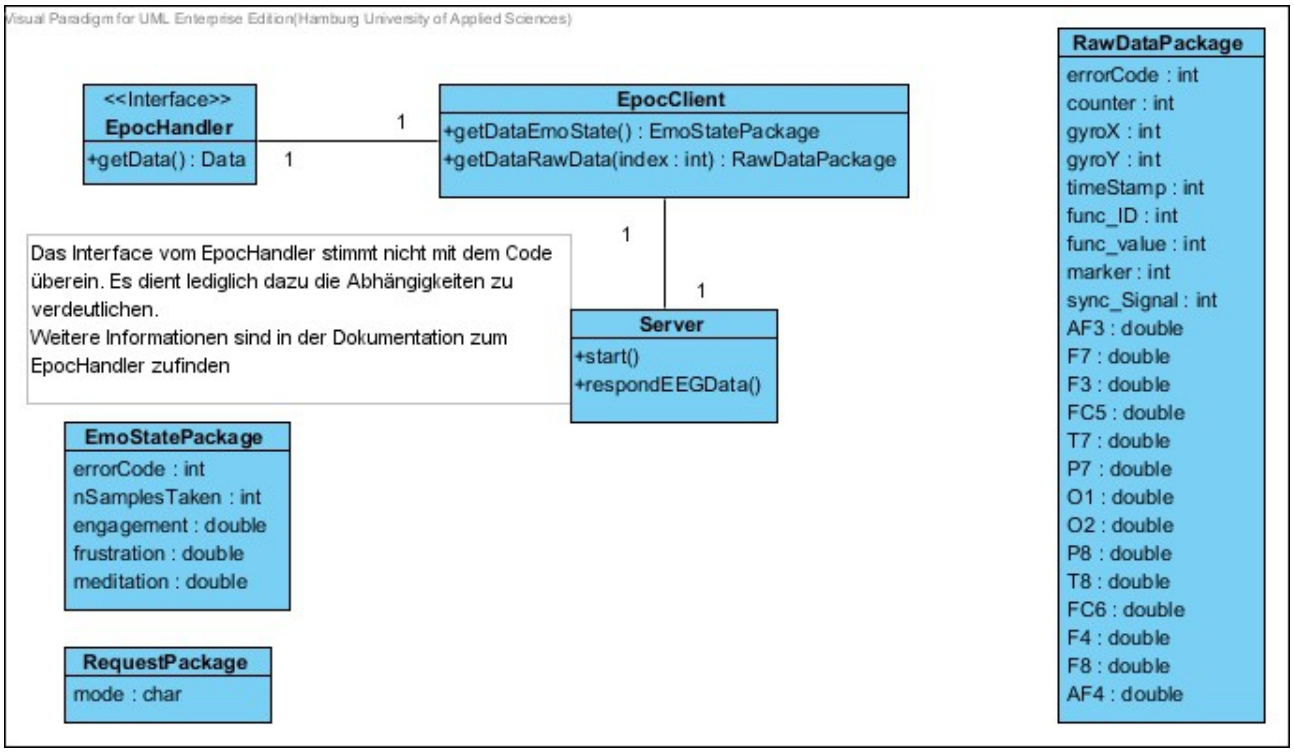
2.1 Einleitung:

Der UDP Server ermöglicht einen Prozess die Daten des BCI zu empfangen mithilfe eines eigenen UDP Protokoll.
Er ist in C++ geschrieben und benutzt zur erstellung der Sockets die Win 32 api.
Als Kommunikationsschnittstelle zum BCI dient der BCI-Wrappe.

2.2 Aufbau:

Der UDP Server unterteilt sich in zweiKomponenten den BCI-Wrapper um auf das BCI zuzugreifen, den Server der die UDP Packete annimmt, verarbeitet und darauf antwortet.





Class EpocHandler:
Schnittstelle zu dem BCI. Weiter informationen sind der entsprechenden Dokumentationen zu entnehemen.

Class EpoCClient:
Wandelt die daten die der EpocHandler liefert in die vom Server benutzten Packages um.

Class Server:
Öffnet den Socket und verarbeitet ankommende Nachrichten.
Es wird dabei kein neuer Thread erstellt.

Pseudocode.
start()
init()
while(true){
package = receivePackage();
if(package->mode == EEGDATA)
	respondEEGData();
}

2.3.1 UDP Protocol:

Die Laborrechner verwenden eine 64 Bit Intel CPU und als Betriebssystem Win 7 64 Bit (Speicherbelegung ist little Endian).

```
struct REQUESTPACKAGE{
	char mode;
}
```
1 Byte

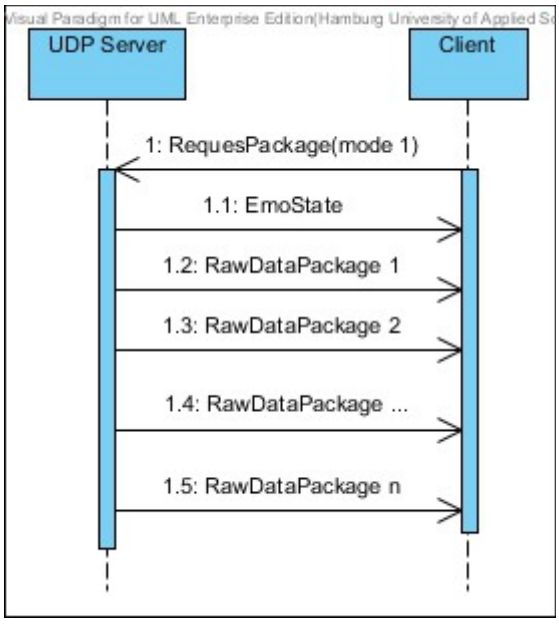
mode definiert welche Daten geschickt werden sollen.
Standard 1 für die EEGDaten.

```
struct EMOSTATEPACKAGE{
	int32_t errorCode;
	int32_t nSamplesTaken;
	double engagement;
	double frustration;
	double meditation;
	double excitement;
}
```
40 Byte

```
struct RAWDATAPACKAGE{
      int32_t errorCode;
      4 Byte data Alignment;
      double counter;
      double gyroX;
      double gyroY;
      double timeStamp;
      double func_ID;
      double func_Value;
      double marker;
      double sync_Signal;
      double AF3;
      double F7;
      double F3;
      double FC5;
      double T7;
      double P7;
      double O1;
      double O2;
      double P8;
      double T8;
      double FC6;
      double F4;
      double F8;
      double AF4;
}
```
152 Byte

2.3.2 Nachrichtenablauf:

Versendet werden immer die Structs aus 2.3.1.
Da für die Berrechnung von den EmoStates mehrere RawData Samples gebraucht werden,
wird mit den anfragen der EEGDaten,
so viele RawDataPackages mitgeschickt wie Samples benötigt wurden.
Die gesamte Kommunikation ist asynchron.



2.4 UDP Server Konstanten

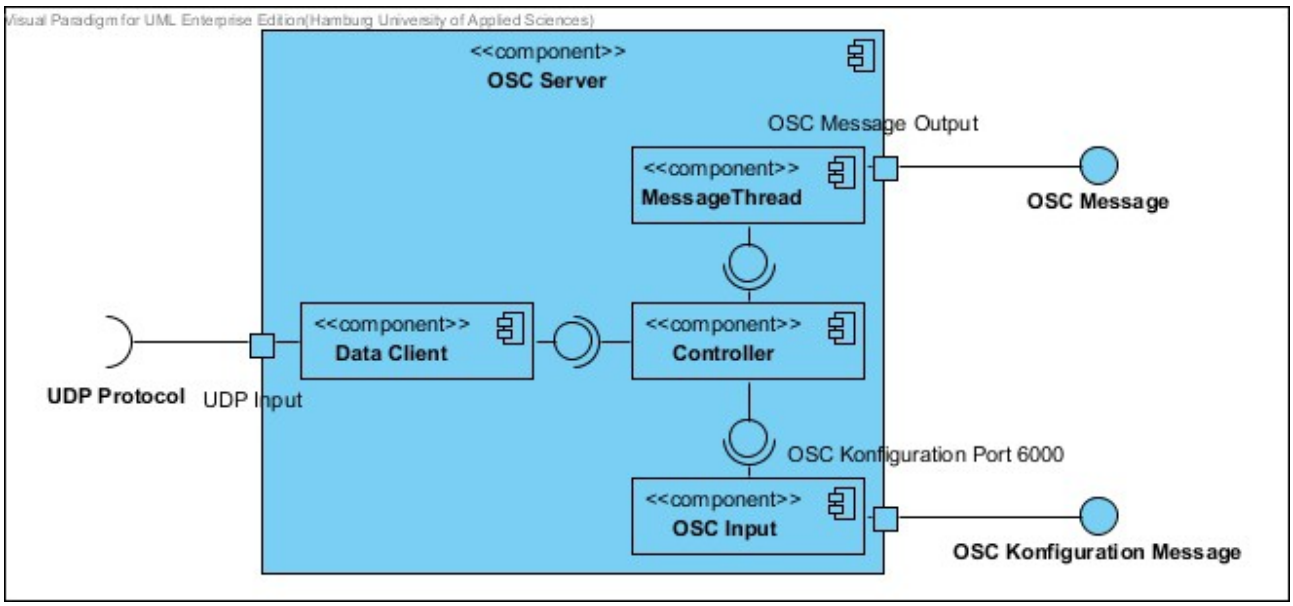
SDTPORT 5000: Der Port auf dem sich der Socket bindet.
EEGDATA 1: Konstante für den Request mode

3. OSC Server:

3.1 Einleitung

Die Grundfunktion des OSC Server ist die Verteilung der EEGDaten in Form von OSC Paketen. Er verfügt aber gleichzeitig auch über eine observerartige Funktion die es einen Erlaubt ein Event zu definieren wenn es eintritt, wird eine vorher festgelegte Nachricht abgeschickt (Kapitel 4).

3.2 Aufbau

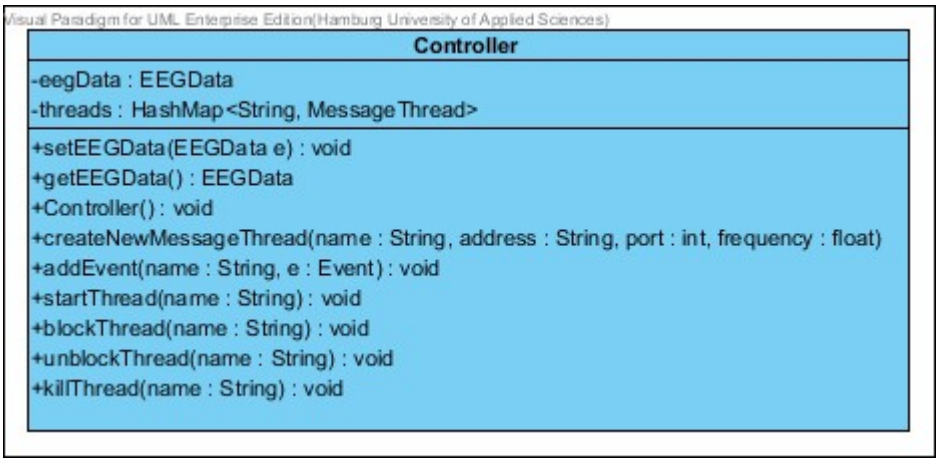


Die Komponente Controller dient als zentrale Verwaltung für die Threads und die EEGDaten.
Data Client kümmert sich um die Kommunikation zum UDP Server und das immer die aktuellen Daten vorliegen.
OSC Input ermöglicht die Konfiguration der MessageThreads und MessageThread überprüft die vorliegenden Nachrichten und sendet dementsprechend OSCMessages.

3.3 Controller

Der Controller dient zur Verwaltng der Message Threads.
Alle Message Threads sind in einer HashMap gespeichert mit ihren Namen als Key.
Der Controller leitet die Methodenaufrufe zu den jeweiligen Thread weiter und verwaltet die Daten des BCI.

Class Controller
package controll



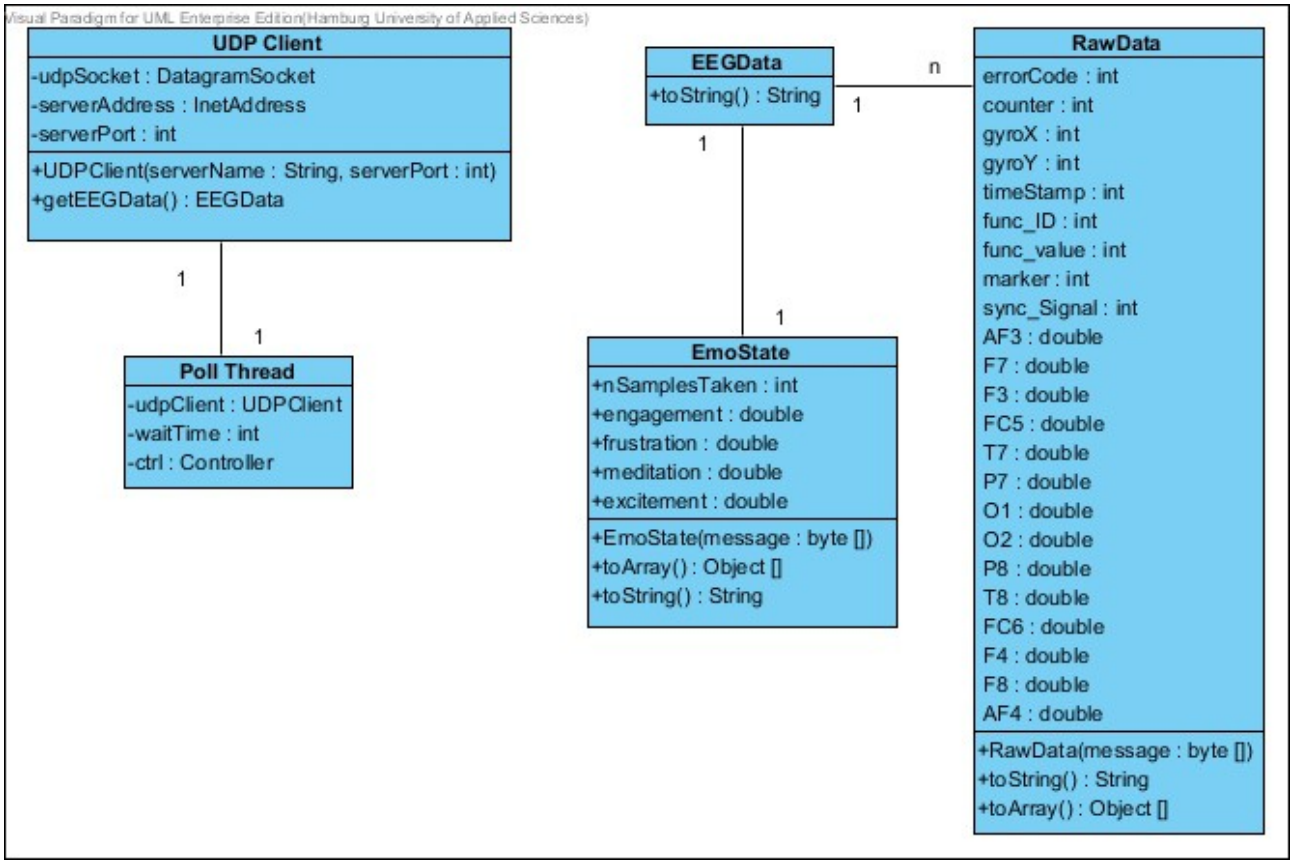
3.4 OSC Input

Beim OSC Input werden OSCNachrichten empfangen um Message Threads
zu Konfigurieren(siehe Kapitel 4)

Class OSCInput
package com.osc

Methoden:
OSCInput(int port, Controller c)
Erstellt einen OSC Inputport und fügt die Listener für die Konfiguration hinzu(siehe Kapitel 4), welche mit dem Controller c interagieren.

3.5 Data Client



3.5.1 Pollthread

Um die Netzwerkbelastung zwischen den UDP Server und den OSC Server möglichst gering zu halten, liegen auf dem Java Server immer die aktullen Nachrichten des BCI vor, mit diesen arbeiten die Message Threads.

Class Pollthread
package controll

Methoden:
Pollthread(UDPClient u, double f, Controller c)
u = Schnittstelle zwichen OSC Server und UDP Server
f = Frequenz wie häufig die Daten vom UDP Server gepollt werden soll
c = Der Controller an den die Daten weitergeleitet werden sollen

3.5.2 UDP Client

Schnittstelle zwischen UDP und OSC Server

Class UDPClient
package com.udp

Schnittstelle zwischen den UDP Server und den OSC Server

UDPClient(String serverName, int port)

serverName = Name oder Textdarstellung der IP des UDP Servers

port = port des C++ Servers

EEGData getEEGData()

Holt die Daten des BCI vom UDP Server und wandelt sie in die EEGData Klasse um.

3.6 Datenpakete

Statt der C++ Strukts werden Klassen verwendet.

Class EmoState
package dataPackages

Variablen:

```
int nSamplesTaken
double engagement
double frustration
double excitement
double meditation
```

Methoden:
EmoState(byte[] data)
Erstellt einen EmoState Klasse aus den Daten einer UDP Nachricht

Object[] toArray()
Wandelt die Variablen in ein Object[] um für die OSC Nachricht
Object[] {nSamplesTaken, engagement, frustration, meditation, excitement}

Class RawData
package dataPackages

Variablen:
```
public final int counter;
public final int gyroX;
public final int gyroY;
public final int timeStamp;
public final int func_ID;
public final int func_Value;
public final int marker;
public final int sync_Signal;
public final double AF3;
public final double F7;
public final double F3;
public final double FC5;
public final double T7;
public final double P7;
public final double O1;
public final double O2;
public final double P8;
public final double T8;
public final double FC6;
public final double F4;
public final double F8;
public final double AF4;
```

Methoden:
RawData(byte[] data)
Erstellt einen RawData Klasse aus den Daten einer UDP Nachricht

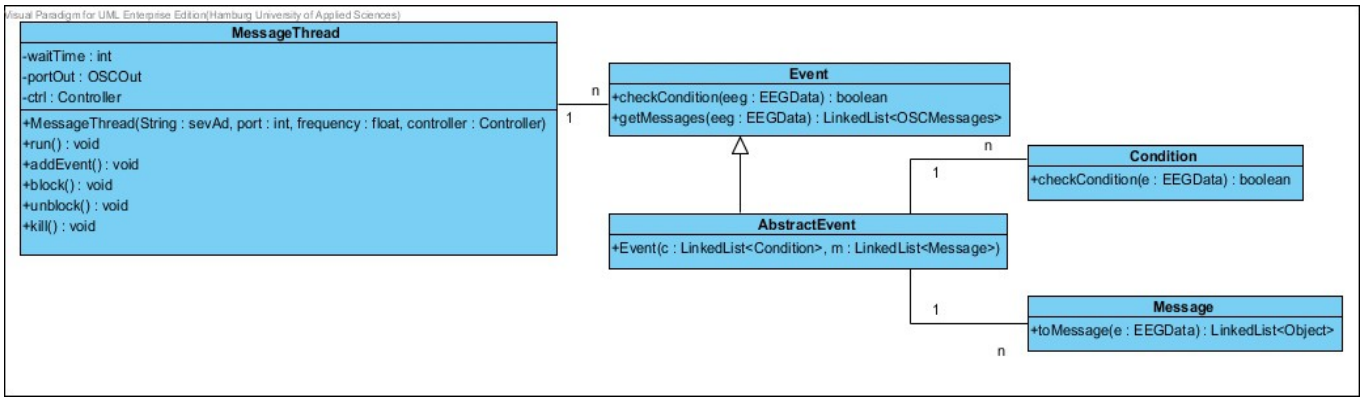
Object[] toArray()
Wandelt die Variablen in ein Object[] um für die OSC Nachricht
Object[] {counter, gyroX, gyroY, timeStamp, func_ID, func_Value, marker, sync_Signal
AF3, F7, F3, FC5, T7, P7, O1, O2, P8, T8, FC6, F4, F8, AF4}

Class EEGData
package dataPackages

EEGData(EmoState e, LinkedList<RawData> r)
Erstellt einen EEG Datensatz aus den EmoState und den zugehörigen RawDatasamples

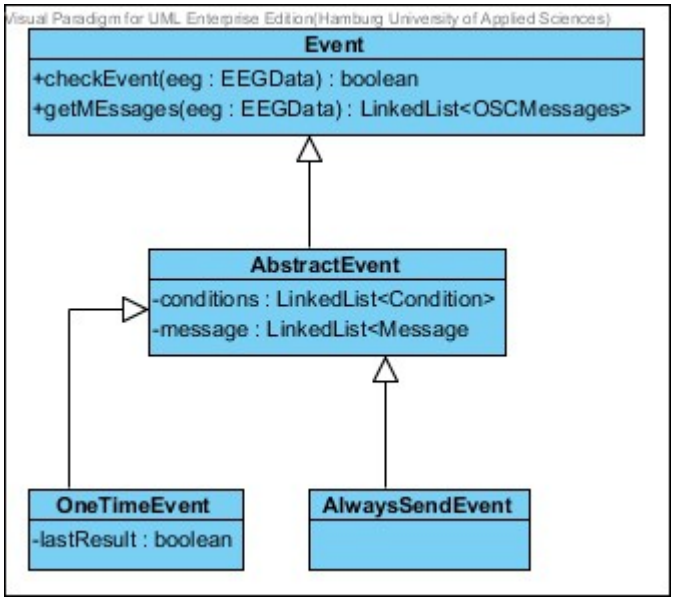
3.7 Message Thread

Überprüft in einer festgelegten Frequenz seine Events mit den Daten des BCI,
sollte ein Event eintreten werden die Nachrichten abgeschickt.




```
while(true){
        if(checkEvent)
                send(getMessages)
}
```

3.7.1 Event



Class Event
package event

Interface welches die Methoden die der Message Thread nutzt einfordert.
boolean checkCondition(EEGData eeg)
prüft ob das Event Eintritt anhand der EEGDaten

to Message(EEGData eeg)
gibt die Messages des Events zurück anhand der EEGDaten

Class AbstactEvent
package event

Das Event baut auf eine Liste mit Condition und eine mit Messages auf und stellt die Methoden für checkEvent und getMessages bereit.
Die Subklassen dieses Events kann man über OSC Konfigurieren

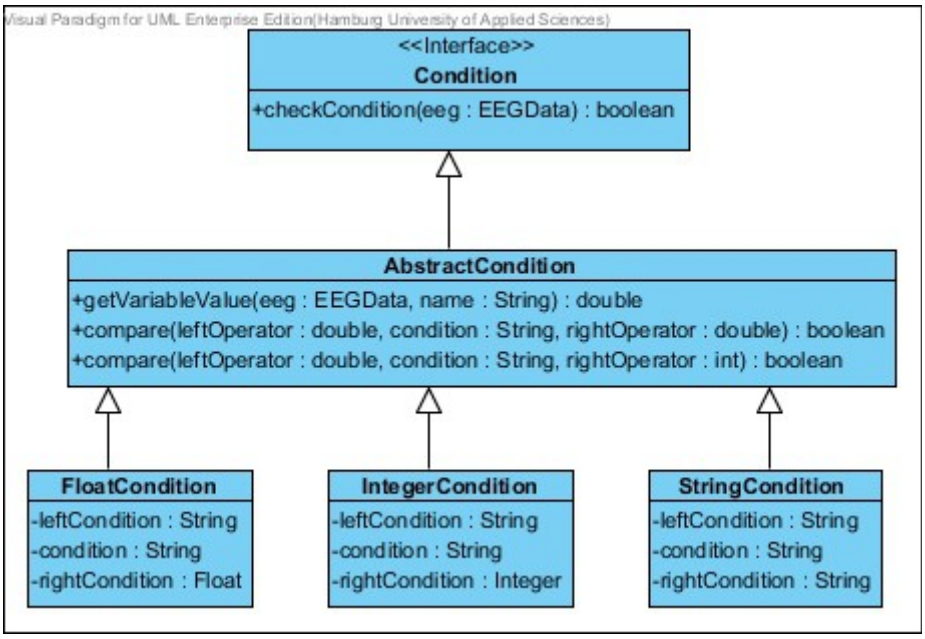
Class AlwaysSendEvent
package event

Gibt super.checkCondition zurück

Class OneTimeEvent
package event

Gibt true zurück wenn ein wechsel von false auf true stattfindet
(super.checkCondition == true && lastResult == false)

3.7.2 Condition



Class Condition
package event.condition

Interface welches die Methoden einforder welche das Event benutzt

public boolean checkCondition(EEGData eeg)

Überprüft anhand der EEGDaten ob die Condition true ist

Class AbstractCondition
package event.condition

getVariableValue gibt dem name entsprehenden Wert aus den EEG Datensatz zurück.
Strings für name:

"emostate.engagement"
"emostate.meditation"
"emostate.excitement"
"emostate.frustration"

compare vergleicht den linken Operator mit den rechten.
Strings für condition:
"=="
"!="
"<"
">"
"<="
">="

Class FloatCondition
package event.condition

Vergleich mit einen Float

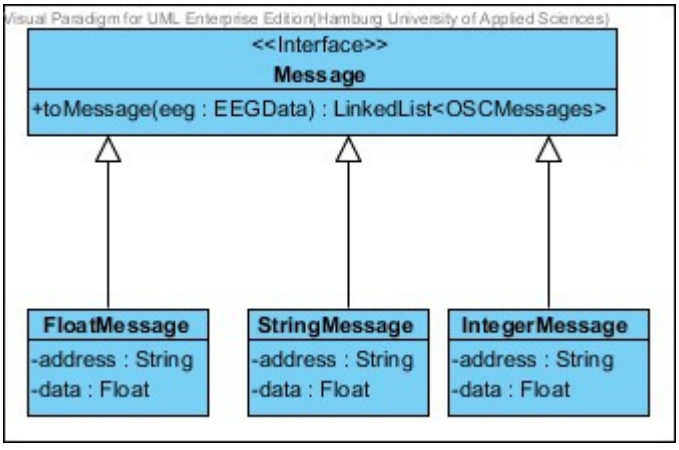
Class StringCondition
package event.condition

Vergleich mit einer Variable

Class IntegerCondition
package event.condition

Vergleich mit einen Int

3.7.3 Message



Message
Interface welches die Methoden einforder welche das Event benutzt

public LinkedList<OSCMessage> toMessage(EEGDaten eeg)
gibt die Message zurück welche anhand der eeg Daten erstellt werden

Float Message
Es soll ein selbstdefinierter Float geschickt werden

Int Message
Es soll ein selbstdefinierter Integer geschickt werden

String Message
Es soll ein selbstdefiniertert String, oder ein Datensatz geschickt

Strings mit besondere Bedeutung zum versenden von Datensätzen
(Groß- und Kleinschreibung ist dabai egal):
"emostate" Vollständiger emostate Datensatz
"emostate.engagement"
"emostate.meditation"
"emostate.excitement"
"emostate.frustration"
":rawdata" Jedes Sample des EEG Datensatz wird als wird als einzelne Nachricht geschickt
"=>rawdata" Alle Samples werden als eine Nachricht geschickt

3.8 Konstanten

Sämtliche Konstanten sind in der Klasse Constants definiert
byte REQUESTEEGDATA = 1
int RESPONSEEMOSTATESIZE = 40
int RESPONSERAWDATASIZE = 184
int SERVERPORTIN = 5000
int OSCPORTIN = 7777
double POLLFREQUENCY = 10

4 Jave Server Konfiguration

4.1 OSCMessages

Eingabe an den Controller über OSCMessages

```
i 32 Bit Integer
f 32 Bit Float
s Ascii String

Abkürzungen:
c Condition steht für sss, ssi oder ssf  leftOperator        Condition    rightOperator
                                         EmoState.Meditation     <             5

Besondere Strings für Condition
EmoState.Meditation
EmoState.Frustration
EmoState.Excitement
EmoState.Engagement

m message steht für ss, si oder sf   address        message
                                     /Ziel/Start     EmoState
```

```
Besondere Strings für Message
EmoState
EmoState.Meditation
EmoState.Frustration
EmoState.Excitement
EmoState.Engagement
:RawData                Die RawData werden in extra OSCMessages
=>RawDatea              Die RawData werden in einer OSCMessage versendet
Man kann String auch aneinanderreihen in dem man sie durch ein ',' trennt


/Server/createThread   ssif name, ip, port, frequency
/Server/blockThread    s    name
/Server/unblockThread  s    name
/Server/startThread    s    name
/Server/killThread     s    name
/Server/loadClass      s    path
/Server/loadClass      ss   path, regex

Es gibt zwei Arten von Events die das Sendeverhalten beeinflussen
AlwaysSend sendet immer wenn die Bedingung erfüllt ist
OneTime sendet immer wenn ein Wechsel von Bedingung nicht erfüllt zu erfüllt
stattfindet

/Server/addEvent/AlwaysSend scm    address condition message
/Server/addEvent/OneTime   scm    address condition message
/Server/addEvent/OneTime   sccm   address condition condition message
/Server/addEvent/OneTime   scmm   address condition message message
/Server/addEvent/OneTime   sccmm  address condition condition message message

Jedes Event kann aus beliebig vielen conditions und messages bestehen dabei werden die
conditions verundet

Hier bei handelt es sich um ein Event ohne condition es kann auch beliebig viele
Messages enthalten
/Server/addPackages sm name address message

EmoStateMessage
iffff nSamplesTaken engagement frustration meditation excitement

RawDataMessage
iiiiiiiiffffffffffffff counter gyroX gyroY timeStamp func_ID func_Value marker
sync_Signal AF3 F7 F3 FC5 T7 P7 O1 O2 P8 T8 FC6 F4 F8 AF4
```

4.2 FileReader

Java Programm das aus einer einer Datei die OSC Befehle ausliest.
Syntax: addresse typOfParameters Parametres

/Server/createThread ssif MyName localhost 7777 5
/Server/startThread s MyName

4.2.1 StringParser

Teilaspekt des FileReaders, welcher auch einzeln benutzbar ist.

4.3 Selbst erstellte Events

In der Main der Klasse start.Start kann man nach der Initialisierung des Programmes eigenen Code hinzufügen und den Controller selbst erstellte Events übergeben.

5 Benutzung des Programmes

5.1 Start Programm

1. Start des UDPServers und des BCIHeadsets
2. 15Sekunden warten
3. Start des OSCServers über die Main in start.Start oder start.StartConfig
   Start nimm die Werte aus der Klasse Constants zum starten.
   StartConfig nimm die Werte aus den Argumenten des aufrufes.

java start.StartConfig ServerIP ServerPort OCPPort Pollfrequency

5.2 Fehlerquellen

-Im HAW-Netz sind die Port blockiert, es msss also ein eigenes Netz aufgebaut werden
-Firewall blockiert das Programm
-Verbindung zum BCI wurde nicht aufgebaut

5.3 Lokale Test

Im Ordner UDPTestServer ist ein UDP Server implementiert der die Pakete mit Dummywerten füllt um die Verbindung zu testen.

6. Code

Im Package test sind Beispiele für die Konfiguration des Servers.
OSC Listen zum starten der Listener und OSCConfig, FileTest und StringTest zum starten des MessageThreads.

7 OSC Implementierungen

Der Code in den Package osc und seinen Subpackages ist eine OpenSource implementierung von Chandrasekhar Ramakrishnan
http://www.illposed.com/software/javaosc.html

8 Glossar:

BCI Brain Computer Interface
OSC Open Sound Controll