LATVIJAS UNIVERSITĀTE

DATORIKAS FAKULTĀTE

SENSORU DATU APSTRĀDES UN FIZISKĀS AKTIVITĀTES ANALĪZES SISTĒMA FIZIKĀLĀS MEDICĪNAS UN REHABILITĀCIJAS ĀRSTIEM

KVALIFIKĀCIJAS DARBS

Autore: **Jekaterina Jevtejeva**

Studenta apliecības Nr.: jj19021

Darba vadītāja: Dr.dat. Vineta Arnicāne

RĪGA 2022

ANOTĀCIJA

Šī kvalifikācijas darba mērķis ir aprakstīt programmatūru sensoru datu apstrādei un pacientu fiziskās aktivitātes analīzei. Tā ir sistēma, kas saņem datus no elpošanas un kustību sensoriem, apstrādā tos atbilstoši sistēmas lietotāja pārvaldītajā konfigurācijas datnē norādītiem treniņa soļiem un sūta treniņu soļu vizuālas, skaņas un tekstuālas instrukcijas kopā ar apstrādātiem sensoru datiem uz paplašinātās realitātes brillēm.

Šī sistēma ir domāta fizikālās medicīnas un rehabilitācijas ārstiem, kas izstrādā treniņu programmu saviem pacientiem un vēlas integrēt sensoru tehnoloģijas savā ārstēšanas praksē zinātniskai izpētei, efektīvākai, viegli pieejamai objektīvu datu analīzei un uzlabotai pacientu pieredzei ar reāllaika atgriezenisko saiti par viņu fizisko aktivitāti paplašinātās realitātes brillēs.

Atslēgvārdi: sensors, medicīna, rehabilitācija, Java, Spring, Docker

// ABSTRACT

“Sensor data processing and physical activity analysis system for physical medicine and rehabilitation physicians”

The purpose of this document is to describe sensor data processing and physical activity analysis software. It is a system that receives data from breathing and motion sensors, processes the data according to workout steps specified in a user-managed configuration file, and sends visual, audible, and textual workout instructions along with the processed sensor data to an augmented reality headset.

This system is intended for physical medicine and rehabilitation physicians who develop a training program for their patients and want to integrate sensor technologies into their treatment practices for scientific research, a more efficient, easily accessible objective data analysis and a better patient experience with real-time feedback about their physical activity in the augmented reality headset.

Keywords: sensor, medicine, rehabilitation, Java, Spring, Docker

SATURS

IEVADS 5

APZĪMĒJUMU SARAKSTS 8

1. VISPĀRĒJAIS APRAKSTS 10

1.1. Esošā stāvokļa apraksts 10

1.2. Pasūtītājs 10

1.3. Produkta perspektīva 10

1.4. Darījumprasības 11

1.5. Lietotāja raksturiezīmes 11

1.6. Vispārējie ierobežojumi 12

1.7. Pieņēmumi un atkarības 12

2. PROGRAMMATŪRAS PRASĪBU SPECIFIKĀCIJA 13

2.1. Funkcionālās prasības 13

2.1.1. Pirmais sprints 13

2.1.2. Otrais sprints 15

2.1.3. Trešais sprints 16

2.1.4. Ceturtais sprints 18

2.2. Nefunkcionālās prasības 19

2.2.1. Uzturamība 19

3. PROGRAMMATŪRAS PROJEKTĒJUMA APRAKSTS 20

3.1. Dekompozīcijas apraksts 20

3.2. Atkarību un resursu apraksts 22

3.3. Saskarņu apraksts 22

3.4. Apakšsistēma “Java-Application” 23

3.4.1. Dekompozīcijas apraksts 23

3.4.2. Atkarību apraksts 24

3.4.3. Detalizētais moduļu apraksts 26

3.4.3.1. Treniņa konfigurācijas ģenerēšana: modulis “Workout” 26

3.4.3.2. Sensoru datu iegūšana no sensoriem: modulis “Adapter” 31

3.4.3.3. Datu apstrāde un pārraide: modulis “Multicast” 38

3.4.3.4. Lietotnes konfigurācijas: modulis “Config” 51

3.4.3.5. Tīmekļa lappuse: modulis “Web” 57

3.5. Treniņa konfigurācijas uzglabāšana: apakšsistēma “Etcd” 62

3.6. Treniņa konfigurācijas pieprasījumu apstrāde: apakšsistēma “Etcd-Proxy” 62

3.7. Lokālu video un audio datņu izmantošana treniņa konfigurācijā: apakšsistēma "Media-Proxy" ... 64
3.8. Plux sensora modelis: programma "PluxMock" ... 65
3.9. Lietotāja saskarnes projektējums ... 68
3.10. Sistēmas palaišana ... 70
4. TESTĒŠANAS DOKUMENTĀCIJA ... 73
4.1. Testēšanas metodes ... 73
4.2. Automatizētā apakšsistēmas "Java-Application" klašu vienībtestēšana ... 73
4.3. Manuālā sistēmas prasību testēšana ... 74
5. PROJEKTA ORGANIZĀCIJA ... 82
5.1. Konfigurāciju pārvaldība ... 83
5.2. Kvalitātes nodrošināšana ... 83
5.3. Darbietilpības novērtējums ... 84
REZULTĀTI UN DISKUSIJA ... 85
SECINĀJUMI ... 86
IZMANTOTĀ LITERATŪRA UN AVOTI ... 87
PIELIKUMI ... 88

IEVADS

Mūsdienās jaunas tehnoloģijas strauji ienāk mūsu ikdienā un ir iekļuvušas jebkurā cilvēka darbības jomā, ieskaitot medicīnu. Rehabilitācijas un fizikālās medicīnas nozarē, datu analīze un efektīvs rehabilitācijas process ir divi būtiski aspekti, kuru uzlabošana kļuva par pamatu projektam "ARehab", pie kura darba autore strādāja savas programmēšanas prakses ietvaros uzņēmumā "Fanout SIA".

Projekta mērķis ir veikt zinātnisku izpēti par valkājamo sensoru un paplašinātās realitātes tehnoloģiju izmantošanu rehabilitācijas pacientu motivācijas, progresa un datu apstrādes uzlabošanai ar nolūku nākotnē integrēt šīs tehnoloģijas ar medicīnas sensoriem rehabilitācijas un fizikālās medicīnas ārstu praksēs, veicinot rehabilitācijas pacientu motivācijas un pašapziņas uzlabošanu, objektīvu datu analīzi, viegli pieejamu progresa vērošanu un ilgtermiņa rehabilitācijas pārraudzību [1].

Šī kvalifikācijas darba mērķis ir aprakstīt sistēmu ar nosaukumu "ARehab Portable", kas, pirmkārt, kalpo par sava veida "raidītāju" – tā saņem datus par pacienta elpošanu un kustībām no valkājamajiem elpošanas un kustību sensoriem, apstrādā tos un pārsūta tālāk uz paplašinātās realitātes brillēm, kas ir uzliktas pacientam. Otrkārt, sistēma no lietotāja pārvaldītas konfigurācijas datnes ģenerē treniņu pacientam, kas sastāv no vairākiem treniņa soļiem, katrs no kuriem, savukārt, satur dažāda veida - tekstuālas, vizuālas, skaņas – instrukcijas, kas pacientam izskaidro, kā jāizpilda vingrinājums. Sistēma apstrādā sensoru datus un, apvienojot tos ar informāciju par tekošo treniņa soli, atver datu plūsmu uz paplašinātās realitātes brillēm. Sistēmas darbības rezultātā pacients sevis priekšā var redzēt vingrinājumu instrukcijas un reāllaika atgriezenisko saiti par to, cik veiksmīgi viņš izpilda vingrinājumu.

Sensoru datu apstrādes un fiziskās aktivitātes analīzes sistēmas "ARehab Portable" izstrāde notika saskaņā ar spējās izstrādes metodoloģiju. Paplašinātas realitātes lietotnes izstrāde notika paralēli projekta ietvaros un šajā kvalifikācijas darbā netiek aprakstīta.

Nolūks

Kvalifikācijas darbs "Sensoru datu apstrādes un fiziskās aktivitātes analīzes sistēma fizikālās medicīnas un rehabilitācijas ārstiem" ir dokumentācija, kas apraksta izstrādātā programmatūras produkta "ARehab Portable" prasības, izstrādi un testēšanu. Šī dokumentācija ietver programmatūras prasību specifikāciju, programmatūras projektējuma aprakstu, lietotāja saskarnes projektējumu, programmatūras testēšanas dokumentāciju, kā arī informāciju par projekta organizāciju, kvalitātes nodrošināšanu, konfigurāciju pārvaldību un darbietilpības novērtējumu.

Darbības sfēra

Aprakstītās sistēmas mērķis ir palīdzēt rehabilitācijas un fizikālās medicīnas ārstiem, kas izstrādā treniņu programmu ar dziļās muskulatūras vingrojumiem un elpošanas tehnikām saviem pacientiem un vēlas pielietot sensoru un AR tehnoloģijas zinātniskai izpētei par šo tehnoloģiju izmantošanu pacientu ar apakšējo ekstremitāšu amputācijām motivācijas, progresa uzlabošanai un savā ārstēšanas praksē rehabilitācijas procesa paātrināšanai un pacientu dzīves kvalitātes uzlabošanai.

Izstrādājamā sistēma darbojas ar Plux elpošanas un kustību sensoru sistēmām [2]. Sistēma ļauj ārstam aprakstīt treniņa programmu pacientiem, izmantojot tekstuālas, vizuālas un skaņas instrukcijas, ļauj pārlūkprogrammā labot sensoru iestatījumus un palaist treniņu, kas automātiski pārslēgs soļus un sūtīs apstrādātus sensoru datus (skaitļus no 0 līdz 1 atkarībā no tā, cik veiksmīgi tiek izpildīts vingrinājums) tālāk uz AR brillēm, kur notiek apstrādāto sensoru datu dinamiska vizualizācija. Rezultātā pacients savās AR brillēs redz katra treniņa soļa instrukcijas un saņem reāllaika atgriezenisko saiti par savu fizisko aktivitāti.

Projekta ietvaros, "ARehab Portable" nav pirmā produkta versija. Sistēmas "ARehab Portable" nolūks ir optimālākā, stabilākā risinājuma izstrāde, kas darbojas ar lietojumprogrammatūru "OpenSignals" sensoru datu iegūšanai.

Saistība ar citiem dokumentiem

Programmatūras prasību specifikācijas izstrādē tika ievērotas *LVS 68:1996 „Programmatūras prasību specifikāciju ceļvedis"* [3] standarta prasības.

Programmatūras projektējuma apraksta izstrādē tika ievērotas *LVS 72:1996 "Ieteicamā prakse programmatūras projektējuma aprakstīšanai"* [4] standarta rekomendācijas.

Dokumenta pārskats

Dokuments sastāv no 5 daļām:

1. Vispārējais apraksts – esošā stāvokļa apraksts, informācija par pasūtītāju, produkta perspektīvu, darījumprasībām, lietotāja raksturiezīmēm, ierobežojumiem, pieņēmumiem un atkarībām;
2. Programmatūras prasību specifikācija:
   2.1. Funkcionālās prasības – detalizēti aprakstītas programmatūras produkta veicamās funkcijas, sagrupētas pa sprintiem un noformētas lietotājstāstos un sprinta uzdevumos;

2.2. Nefunkcionālās prasības – programmatūras produkta prasības, kas apraksta, kā sistēmai jādarbojas;

3. Programmatūras projektējuma apraksts:
   3.1. Dekompozīcijas apraksts – informācija par sistēmas sadalīšanu projektējuma komponentos;
   3.2. Atkarību un resursu apraksts – attiecību starp projektējuma komponentiem un to pieprasītiem resursiem attēlojums;
   3.3. Saskarņu apraksts – informācija par sistēmas komponentu ārējām un iekšējām saskarnēm;
   3.4. Apakšsistēma “Java-Application” – apakšsistēmas dekompozīcijas, atkarību, saskarņu apraksti un tās moduļu detalizētais apraksts;
   3.5. Treniņa konfigurācijas uzglabāšana: apakšsistēma “Etcd” – apakšsistēmas detalizētais apraksts;
   3.6. Treniņa konfigurācijas pieprasījumu apstrāde: apakšsistēma “Etcd-Proxy” - apakšsistēmas detalizētais apraksts;
   3.7. Lokālu video un audio datņu izmantošana treniņa konfigurācijā: apakšsistēma “Media-Proxy” – apakšsistēmas detalizētais apraksts;
   3.8. Plux sensora modelis: programma “PluxMock” – programmas detalizētais apraksts;
   3.9. Lietotāja saskarnes projektējums – realizētās sistēmas lietotāja saskarnes projektējuma apraksts;
   3.10. Sistēmas palaišana – informācija par to, kādas procedūras ir nepieciešamas sistēmas lietotājam sistēmas palaišanai un izmantošanai;
4. Testēšanas dokumentācija:
   4.1. Testēšanas metodes – sistēmas testēšanai izmantoto metožu apraksts;
   4.2. Automatizētā apakšsistēmas “Java-Application” klašu vienībtestēšana – testēšanas apraksts;
   4.3. Manuālā sistēmas prasību testēšana – testēšanas apraksts;
5. Projekta organizācija:
   5.1. Konfigurāciju pārvaldība – informācija par realizētās sistēmas konfigurāciju pārvaldībai izmantotajām metodēm un rīkiem;
   5.2. Kvalitātes nodrošināšana – metodes, kas tika izmantotas izstrādātās sistēmas kvalitātes nodrošināšanai;
   5.3. Darbietilpības novērtējums – darba izstrādei pavadītā laika novērtējums.

APZĪMĒJUMU SARAKSTS

AR (*Augmented Reality*) – paplašinātā realitāte

Bean – Spring satvara kontekstā: objekts, kas veido lietojumprogrammas pamatu un kuru pārvalda Spring IoC konteineris

BITalino – kompānija, kas izstrādā BITalino Plux biosignālu iegūšanas sensoru sistēmas

Bluetooth – bezvadu datortīklu standarts ar mazu darbības rādiusu

CLOC (*Count Lines of Code*) – rīks koda rindiņu automātiskai skaitīšanai

DI (*Dependency Injection*) - paņēmiens, kurā objekts saņem citus objektus, no kuriem tas ir atkarīgs, ko sauc par atkarībām

Docker – programmatūras platforma, kas izmanto operētājsistēmas līmeņa vizualizāciju programmatūras piegādei Docker konteineros

Docker attēls (*Docker image*) – lasāma veidne, kas satur instrukciju kopu konteinera, kas var darboties Docker platformā, izveidei

Docker komponēšana (*Docker compose*) – rīks dažādu Docker konteineru izvietošanas un vienlaicīgas palaišanas centralizētai pārvaldei

Docker konteineris (*Docker container*) – izolētās, izpildāmās programmatūras pakotnēs, kas sevī iekļauj visu nepieciešamu lietotnes palaišanai

Docker sējums (*Docker volume*) – Docker konteineru datņu sistēmas konteineru datu glabāšanai

Dockerfile – datne ar kodu, kas automātiski izveido Docker konteinerus

DTO (*Data Transfer Object*) – viens no projektēšanas modeļiem, ko izmanto datu pārsūtīšanai starp procesiem vai apakšsistēmām

Etcd – viegla atslēgu-vērtību datu glabātava, kas nodrošina uzticamu veidu, kā uzglabāt datus, kuriem ir jāpiekļūst dalītajai sistēmai vai mašīnu klasterim

Gradle – lietotņu uzbūvēšanas automatizācijas rīks vairāku programmēšanas valodu programmatūras izstrādei

IP (*Internet Protocol*) – Interneta protokols

Java – objektorientēta programmēšanas valoda

JSON (*JavaScript Object Notation)* – datu apmaiņas formāts strukturētu datu pārraidei tīklā

JUnit – Java satvars automatizētai vienībtestēšanai

Kontaktligzda (*Socket*) – programmatūras objekts, kas darbojas kā beigu punkts un veido divvirzienu tīkla sakaru saiti starp servera un klienta puses programmām

Lua – programmēšanas valoda

MAC (*Media Access Control*) adrese – multivides piekļuves kontroles adrese

Nginx – viegls tīmekļa serveris / starpniekserveris

OpenSignals – programmatūra reāllaika biosignālu vizualizācijai, kas spēj tieši mijiedarboties ar visām Plux ierīcēm

Plux – kompānija, kas izstrādā biosignālu iegūšanas sensoru sistēmas

PPA – programmatūras projektējuma apraksts

PPS – programmatūras prasību specifikācija

Spring – Java satvars, bibliotēku kopums, kas ļauj izmantot gatavas struktūras programmatūras koda komponentu pārvaldei

Spring IoC (*Inversion of Control*) konteineris – Spring satvara kodols, kas veido objektus, konfigurē un apkopo to atkarības, pārvalda to dzīves ciklu

STOMP (*Simple Text Oriented Messaging Protocol*) – vienkāršs sadarbspējīgs protokols, kas paredzēts asinhronai ziņojumu pārsūtīšanai starp klientiem

TCP (*Transmission Control Protocol*) – pārraides vadības protokols

Unity - starpplatformu spēļu dzinējs

URL (*Uniform Resourse Locator*) – vienotais resursu vietrādis

WebSocket – datorsakaru protokols, kas nodrošina pilndupleksus sakaru kanālus, izmantojot vienu TCP savienojumu

YAML (*YAML Ain't Markup Language*) – datu serializācijas valoda

1. VISPĀRĒJAIS APRAKSTS

1.1. Esošā stāvokļa apraksts

Sensoru iekārtu izmantošana rehabilitācijas un fizikālās medicīnas nozarē pēdējos gados ir strauji paplašinājies, pateicoties sensoru attīstībai, lētākām pārvietojamām ierīcēm un savienojamības tehnoloģiju attīstībai. Līdz ar sensoru tehnoloģiju pielietošanu rehabilitācijā, pieaug tās procesā iegūto datu apjoms, un viena no nozares problēmām ir esošās ierobežotās iespējas šādu datu apjomu apstrādāt, interpretēt un saprotamā veidā izmantot rehabilitācijas pacientu labā efektīvākam rehabilitācijas procesam [1].

Projektā, kura ietvaros ir izstrādāta sistēma "ARehab Portable", tika realizēta ideja apvienot sensoru tehnoloģijas ar paplašināto realitāti rehabilitācijas nolūkiem. Projekta, kuram ir izstrādāta sistēma "ARehab Portable", mērķis ir dziļi izprast lietotāju pieredzi un novērtēt nozares gatavību šādu tehnoloģiju integrēšanai rehabilitācijā, veicot pētījumus un pielāgojot risinājumus atbilstoši pacientu un viņu ārstu vajadzībām.

Projekta ietvaros izstrādātās iepriekšējās produkta versijas paredzēja treniņa augšupielādi, palaišanu un pārvaldi tikai pacienta AR brillēs, kas ievērojami ierobežoja ārsta iespējas kontrolēt treniņa gaitu. Kā arī, AR brilles sazinājās ar sensoriem pa tiešo caur Bluetooth savienojumu, kas apgrūtināja pieslēguma problēmu risināšanu. Izstrādājamā sistēma "ARehab Portable" piedāvā stabilāku risinājumu, kas sazinās ar Plux sensoriem caur to dzimto lietojumprogrammu "OpenSignals" pa TCP savienojamu vieglākai pieslēguma problēmu risināšanai un ļauj ārstam jeb sistēmas lietotājam pilnībā pārvaldīt treniņa veidošanu, palaišanu, treniņa gaitu un iegūto datu apstrādi atkarībā no katra pacienta sagatavotības un rehabilitācijas mērķiem.

1.2. Pasūtītājs

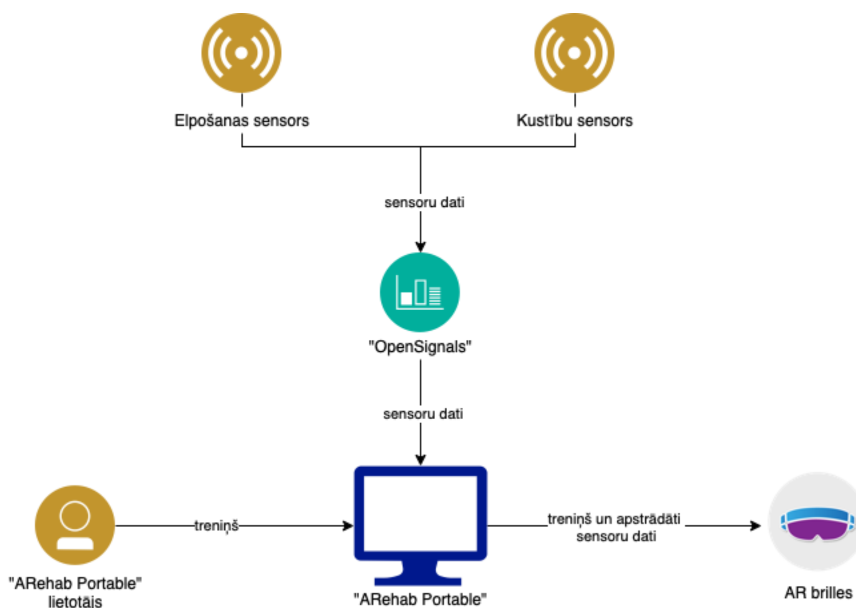
Sistēma ir izstrādāta uzņēmumā "Fanout SIA". Sistēmas pasūtītājs ir fizioterapeite, pētniece, risinājuma koncepta izstrādātāja kompleksai ķermeņa rehabilitācijas īstenošanai cilvēkiem ar apakšējo ekstremitāšu amputācijām, izmantojot paplašināto realitāti un valkājamos sensorus [1].

1.3. Produkta perspektīva

"ARehab Portable" ir daļa no lielāka projekta, tāpēc tā ir atkarīga sistēma, kas ir saistīta ar citām projekta sastāvdaļām - Plux sensoru datu iegūšanas lietojumprogrammatūru

"OpenSignals", no kuras "ARehab Portable" saņem sensoru datu plūsmu, un ar AR brillēm, uz kuriem "ARehab Portable" sūta apstrādāto datu plūsmu.

Attēlā zemāk ir shematiski parādītas visas projekta sastāvdaļas un sistēmas "ARehab Portable" ārējās saskarnes (sk. *Att.1.1.*).



Att.1.1. **Projekta sastāvdaļas un "ARehab Portable" ārējās saskarnes**

1.4. Darījumprasības

Izstrādājamajā sistēmā tiks nodrošināta sekojoša funkcionalitāte:

- No lietotāja pārvaldītās treniņa konfigurācijas datnes ģenerēt treniņa soļus ar tekstuālām, vizuālām, skaņas instrukcijām, kas tiek sūtītas uz AR brillēm;
- Ļaut pārlūkprogrammā uzstādīt sensoru un datu analīzes iestatījumus, palaist treniņu un pārslēgt treniņa soļus, kuriem reāllaikā jāpārslēdzas arī pacienta AR brillēs;
- Saņemt reāllaika sensoru datus no Plux sensoru lietojumprogrammatūras "OpenSignals" un apstrādāt tos atbilstoši tekošām treniņa solim;
- Pārsūtīt apstrādātus sensoru datus uz AR brillēm reāllaikā.

1.5. Lietotāja raksturiezīmes

Sistēma ir paredzēta rehabilitācijas un fizikālās medicīnas ārstiem, kas pacientiem izstrādā treniņu programmu ar elpošanas un kustību vingrojumiem. Sistēmas lietotājam ir nepieciešamas dziļās zināšanas un pieredze darbā ar Plux elpošanas un kustību sensoriem. Kā arī, lietotājam ir jāprot darboties ar "OpenSignals" lietojumprogrammatūru Plux sensoru datu iegūšanai un apstrādei, jābūt pazīstamam ar Plux sensoru datu formātu, jāprot iegūt un analizēt

datus. Sistēmas lietotājam ir jāprot darboties ar teksta, video, audio datnēm un pārlūkprogrammām.

1.6. Vispārējie ierobežojumi

Sistēmas lietošanai ir nepieciešams stacionārs vai portatīvais dators. Sistēmai ir jādarbojas ar Plux kustību un elpošanas sensoriem caur “OpenSignals” lietojumprogrammatūru. Sistēmai ir jānodrošina saskarnes ar “OpenSignals” un ar AR lietotni, kas saņems datu plūsmu no “ARehab Portable”. Sistēmai ir jānodrošina reāllaika sensoru datu iegūšana no “OpenSignals” un reāllaika apstrādātu sensoru datu pārsūtīšana uz AR brillēm.

1.7. Pieņēmumi un atkarības

Tiek pieņemts, ka datorā, kuram tiek izstrādāta programmatūra, būs pieejama Windows, Linux vai MacOS operētājsistēma, būs uzinstalēta “OpenSignals” lietojumprogrammatūra un kāda no populārām pārlūkprogrammām, piemēram, Google Chrome, Mozilla Firefox, Microsoft Edge, Safari. Tiek pieņemts, ka AR brilles ar palaisto AR lietotni atradīsies tajā pašā privātā tīklā kā dators, uz kura tiks palaista sistēma “ARehab Portable”.

2. PROGRAMMATŪRAS PRASĪBU SPECIFIKĀCIJA

2.1. Funkcionālās prasības

Funkcionālās prasības programmatūras produktam “ARehab Portable” ir sadalītas pa sprintiem. Tās ir noformulētas lietotājstāstos un sprinta uzdevumos.

Lietotājstāsta un sprinta uzdevuma identifikators sastāv no:

- Sprinta numura formātā ***SN***, kur *N* ir sprinta numurs pēc kārtas;
- Lietotājstāsta vai sprinta uzdevuma atslēgas vārdiem angļu valodā.

Katrā sprintā ir iekļauti lietotājstāsti un sprinta uzdevumi, kas tika realizēti sprinta ietvaros.

2.1.1. Pirmais sprints

Pirmajā sprintā ir iekļauti lietotājstāsti un sprinta uzdevumi, kas saistās ar reāllaika sensoru datu iegūšanu un ar sensora modeli, kas daļēji atdarina “OpenSignals” lietojumprogrammatūras ārējo saskarni (sk. *Tab.2.1.*, *Tab.2.2.*).

Tab.2.1. Lietotājstāsts: **Reāllaika sensoru datu iegūšana no “OpenSignals”**

Lietotājstāsta identifikators
S1_SENSOR_DATA_ACQUISITION
Lietotājstāsts
Kā sistēmas lietotājs **Vēlos**, lai sistēma “ARehab Portable” pieslēdzas sensoru datu plūsmai no lietojumprogrammas “OpenSignals” un saņem reāllaika datus no Plux sensoriem.
Akceptēšanas kritēriji
• Palaižot lietotni, sistēma mēģina pieslēgties sensoru datu plūsmai. • Ja savienojums nav uzstādīts, sistēma izdod paziņojumu *“Plux connection failed”* un mēģina atkārtoti pieslēgties pēc 1 sekundes. • Kad ir uzstādīts savienojums ar sensoriem, sistēma izdod paziņojumu *“Plux connection established”* un uzsāk datu iegūšanu.

Tab.2.2. Sprinta uzdevums: **Plux sensora modelis-aizbāznis**

Sprinta uzdevuma identifikators
S1_PLUX_MOCK
Apraksts
Prasības mērķis ir nodrošināt izstrādātājam (gadījumā, ja izstrādātājam nav iespējas strādāt ar reāliem Plux sensoriem) iespēju strādāt ar lietojumprogrammas saskarni, kas daļēji atdarina “OpenSignals” lietojumprogrammatūras saskarni, kad tā mijiedarbojas ar ieslēgtiem

Plux sensoriem. Šīm nolūkam ir jāizstrādā atsevišķa izpildāma programma, kas ir paredzēta izstrādātājiem un testētājiem un nav daļa no gala lietotājam domātas sistēmas.
Ievade

Programma sagaida kādu no sekojošām komandām:
`start`
`stop`
`config,{MAC|DEVICE ID},samplingfreq`
`config,{MAC|DEVICE ID},activechannels`
`config,{MAC|DEVICE ID},samplingfreq,{VALUE}` - `VALUE` formātam ir jāatbilst OpenSignals *"TCP/IP Module Guide"* [5] dokumentācijā norādītām;
`config,{MAC|DEVICE ID},activechannels,{VALUE}` - `VALUE` formātam ir jāatbilst OpenSignals *"TCP/IP Module Guide"* [5] dokumentācijā norādītām.

Apstrāde
Tiek veikta pārbaude, vai komanda sākas ar `start`, `stop` vai `config`. • Ja nē, tad tiek pārbaudīts, vai komanda ir pieminēta OpenSignals *"TCP/IP Module Guide"* [5] dokumentācijā. Ja ir, tad tiek parādīts paziņojums *Nr.1*. Ja nav, tad tiek parādīts paziņojums *Nr.4*. • Ja komanda sākas ar `config`, tiek pārbaudīts, vai trešais parametrs atbilst kādai no augstāk pieminētajām `config` komandām. Ja nē, tad tiek parādīts paziņojums *Nr.3*. • Ja komanda sākas ar `config`, tiek pārbaudīts, vai parametru skaits ir korekts. Ja nav, tad tiek parādīts paziņojums *Nr.2*.
Izvade
• Pēc komandas `start` ir nepieciešams uzsākt nejaušu sensoru datu ģenerēšanu atbilstoši OpenSignals *"TCP/IP Module Guide"* [5] dokumentācijā norādītajam formātam. • Pēc komandas `stop` ir nepieciešams pabeigt sensoru datu ģenerēšanu, ja tā tika iepriekš uzsākta. • Pēc komandas `config,{MAC|DEVICE ID},samplingfreq` ir nepieciešams parādīt šī parametra vērtību pieprasītai ierīcei atbilstoši dokumentācijā norādītajam formātam. • Pēc komandas `config,{MAC|DEVICE ID},activechannels` ir nepieciešams parādīt šī parametra vērtību pieprasītai ierīcei atbilstoši dokumentācijā norādītajam formātam. • Pēc komandas `config,{MAC|DEVICE ID},samplingfreq,{VALUE}` ir nepieciešams nomainīt šī parametra vērtību uz doto pieprasītai ierīcei atbilstoši dokumentācijā norādītajam formātam. • Pēc komandas `config,{MAC|DEVICE ID},activechannels,{VALUE}` ir nepieciešams nomainīt šī parametra vērtību uz doto pieprasītai ierīcei atbilstoši dokumentācijā norādītajam formātam.
Paziņojumi
Nr.1. *"Not implemented"*. Nr.2. *"Wrong param count"*. Nr.3. *"Unknown command"*. Nr.4. *"Undefined command"*.

2.1.2. Otrais sprints

Otrajā sprintā tika iekļauti lietotājstāsti un sprinta uzdevumi, kas sasitās ar treniņa konfigurācijas datni un treniņa konfigurācijas redzamību AR lietotnei (sk. *Tab.2.3.*, *Tab.2.4.*, *Tab.2.5.*, *Tab.2.6.*).

Tab.2.3.. Lietotājstāsts: **Treniņa izveide, pārlūkošana un labošana treniņa konfigurācijas datnē**

Lietotājstāsta identifikators
S2_WORKOUT_CONFIG
Lietotājstāsts
Kā sistēmas lietotājs **Vēlos** teksta datnē glabāt, aplūkot un labot informāciju par treniņa vingrojumiem un to soļiem, **Lai** spētu pilnībā pārvaldīt treniņa struktūru.
Akceptēšanas kritēriji
• Par katru treniņa vingrojumu treniņa konfigurācijas datnē ir iespējams glabāt, apskatīt un labot sekojošu informāciju: o Nosaukums (obligāts): vingrojuma nosaukums. • Par katru vingrojuma soli treniņa konfigurācijas datnē ir iespējams glabāt, apskatīt un labot sekojošu informāciju: o Nosaukums (obligāts, var būt tukša simbolu virkne): tekstuāla instrukcija o Video (neobligāts): ceļš līdz lokālai video datnei, kas satur video-instrukcijas; o Audio (neobligāts): ceļš līdz lokālai audio datnei, kas satur audio-instrukcijas; o Ilgums (neobligāts): soļa ilgums sekundēs; ja nav norādīts, tad solim ir manuāla pāreja (jāgaida lietotāja ievade, lai to pārslēgtu uz nākamo); o Piepūles veids (neobligāts, formātā “breathe” vai “muscle”): norāda, kuram piepūles veidam (un attiecīgi kuram sensoram – elpošanas vai kustību) atbilst solis, lai ņemtu datus priekš šī soļa no pareizā sensora; o Diagramma (neobligāts, *true* vai *false,* ja nenorāda, tad pēc noklusējuma *false*): informatīva vērtība priekš AR lietotnes, kas pasaka, vai šajā solī AR brillēs ir jāveic datu vizualizācija.

Tab.2.4. Sprinta uzdevums: **Treniņa soļu šabloni un grupas treniņa konfigurācijas datnē**

Sprinta uzdevuma identifikators
S2_STEPS_TEMPLATES_GROUPS
Apraksts
Vietas ietaupīšanai treniņa konfigurācijas datnē jānodrošina iespēja apkopot treniņa soļus grupās un veidot treniņa soļu šablonus.
Ievade
Nav.
Apstrāde

Sistēmas palaišanas brīdī, ģenerējot treniņa konfigurāciju no treniņa konfigurācijas datnes, soļu grupas ir jāpārveido par atsevišķiem soļiem un soļu šablonus - par īstiem soļiem.
Izvade
Nav.
Paziņojumi
Nav.

Tab.2.5. Lietotājstāsts: **Treniņa konfigurācijas redzamība AR lietotnei**

Lietotājstāsta identifikators
S2_WORKOUT_CONFIG_AR
Lietotājstāsts
Kā sistēmas lietotājs **Vēlos,** lai, palaižot "ARehab Portable", uzģenerētā treniņa konfigurācija kļūst pieejama AR lietotnei, **Lai** AR lietotne varētu nolasīt treniņa soļus un attēlot to tekstu, video un skaņas datnes pacienta AR brillēs.
Akceptēšanas kritēriji
• Palaižot sistēmu, tā mēģina saglabāt konfigurāciju un izdod paziņojumu, ka notiek konfigurācijas saglabāšana. • Kad konfigurācija tika saglabāta, sistēma izdod paziņojumu *"Config saved"*.

Tab.2.6. Lietotājstāsts: ***Sensoru iestatījumu glabāšana konfigurācijas datnē***

Lietotājstāsta identifikators
S2_DEVICES_CONFIG
Lietotājstāsts
Kā sistēmas lietotājs **Vēlos** teksta datnē glabāt informāciju par sensoriem, **Lai** spētu pārvaldīt, kurš sensors atbilst noteiktam piepūles veidam.
Akceptēšanas kritēriji

• Par katru sensoru teksta datnē var glabāt sekojošu informāciju:
  ○ Sensora MAC adrese;
  ○ 1 vai vairāki piepūles veidi, kurus sensors atbalsta: "*breathe*" vai "*muscle*"; katru piepūles veidu var atbalstīt tikai viens sensors.

2.1.3. Trešais sprints

Trešajā sprintā ir iekļauti lietotājstāsti, kas saistās ar tīmekļa lappusi, treniņa palaišanu un sistēmas lokalizāciju (sk. *Tab.2.7.*, *Tab.2.8.*, *Tab.2.9.*).

Tab.2.7. Lietotājstāsts: **Treniņa palaišana tīmekļa lietotnē**

Lietotājstāsta identifikators
S3_ WEB_WORKOUT
Lietotājstāsts

Kā sistēmas lietotājs **Vēlos** pārlūkprogrammā pēc lietotnes palaišanas redzēt visus treniņa soļus, sagrupētus pa vingrojumiem, un palaist to automātisku pārslēgšanu, noklikšķinot uz pogas *"Start!"*, **Lai** redzētu, kāds solis ir ieslēgts un kā soļi automātiski mainās atbilstoši to ilgumiem.
Akceptēšanas kritēriji
• Palaižot lietotni, pēc treniņa konfigurācijas saglabāšanas, pārlūkprogrammā var redzēt visus treniņa soļus pēc kārtas, sagrupētus pa treniņa vingrojumiem. • Par katru vingrojumu var redzēt tā nosaukumu. • Par katru soli var redzēt tā tekstu un ilgumu (vai *MANUAL*) un piepūles veidu (ja norādīts) formā "*BREATHE*" vai "*MUSCLE*". • Noklikšķinot uz pogas *"Start!"*, treniņa soļi sāk automātiski pārslēgties pēc kārtas atbilstoši soļos noradītajiem ilgumiem; ja solī nav norādīts ilgums, tas nozīmē, ka solim ir manuāla pāreja, lietotājam pašam ir jāpārslēdz solis - sistēma gaida ievadi no lietotāja. • Lietotājs var pārslēgt soļus abos virzienos manuāli neatkarīgi no tā, vai solim ir norādīts ilgums vai ir manuāla pāreja. • Kārtējam ieslēgtam solim var redzēt tā dinamisku progresa indikatoru procentos.

Tab.2.8. Lietotājstāsts: **Vērtību robežu un sensoru kanālu iestatījumi pārlūkprogrammā**

Lietotājstāsta identifikators
S3_SENSOR_CONFIG_WEB
Lietotājstāsts
Kā sistēmas lietotājs **Vēlos** pārlūkprogrammā katram piepūles veidam (formātā "*Breathe*" vai "*Muscle*") norādīt vērtību robežas – minimālo un maksimālo vērtību, kā arī katram piepūles veidam atbilstošo sensora kanāla numuru, **Lai** sistēma izmantotu šīs robežas, lai aprēķinātu relatīvo vērtību (tas ir, cik tuvu maksimāli labam mērījumam ir pacienta kārtējais mērījums), un lai sistēma izmantotu pareizu sensora kanāla numuru sensoru datu iegūšanai katram piepūles veidam.
Akceptēšanas kritēriji
• Palaižot lietotni, pārlūkprogrammā katram piepūles veidam formātā "*Breathe*" vai "*Muscle*" ir iespējams norādīt skaitli, kas atbilst sensora kanālam, no kura jālasa dati. • Katram piepūles veidam formātā "*Breathe*" vai "*Muscle*" ir iespējams norādīt divus skaitļus - minimālo un maksimālo vērtību. • Pēc datu ievades ir iespējams uzklikšķināt uz pogas, kas saglabā šos iestatījumus. • Sistēma izmanto norādītos kanāla numurus atkarībā no kārtējā soļa piepūles veida. • Sistēma, apstrādājot sensoru datus, izmanto norādītās robežvērtības, lai izskaitļotu relatīvo vērtību, kas ir skaitlis no 0 līdz 1, divi cipari aiz komata. • Robežvērtības ir iespējams mainīt ar palaistu treniņu bez nepieciešamības atkārtoti palaist lietotni.

Tab.2.9. Lietotājstāsts: **Treniņa lokalizācija**

Lietotājstāsta identifikators
S3_WORKOUT_LOCALIZATION
Lietotājstāsts
Kā sistēmas lietotājs **Vēlos** teksta datnē glabāt informāciju par to, kurā valodā es vēlos palaist treniņu, **Lai** spētu pārslēgt valodu, kurā sistēmai ir jāņem treniņa soļu instrukcijas no treniņa konfigurācijas datnes.
Akceptēšanas kritēriji
• Konfigurācijas datnē ir iespējams norādīt, kurā valodā nākamā sistēmas palaišanas reizē jāpalaiž treniņu. • Treniņa konfigurācijas datnē katram treniņa vingrojuma solim iespējams norādīt tā tekstu, ceļus līdz audio un video datnēm vairākās valodās. • Kad palaiž sistēmu, no treniņa konfigurācijas datnes tiek ņemtas soļu tekstuālas, video un audio instrukcijas pieprasītajā valodā.

2.1.4. Ceturtais sprints

Ceturtajā sprintā ir iekļauti lietotājstāsti un sprinta uzdevumi, kas saistās ar sensoru datu apstrādi un pārraidi (sk. *Tab.2.10.*, *Tab.2.11.*, *Tab.2.12.*).

Tab.2.10. Lietotājstāsts: **Sensoru datu apstrāde**

Lietotājstāsta identifikators
S4_PATIENT_EFFORT_ACCUMULATOR
Lietotājstāsts
Kā sistēmas lietotājs **Vēlos,** lai, atkarībā no treniņa solī norādītā piepūles veida (“*breathe*” vai “*muscle*”), reāllaika sensoru dati šī treniņa soļa ietvaros tika apstrādāti sekojoši: ○ Ja piepūles veids ir “*breathe*”, tad izmantojot “*Moving Average”* metodi, aprēķinot vidējo no katram *100* vērtībām (sensora mērījumiem); ○ Ja piepūles veids ir “*breathe*”, tad izmantojot “*Root Mean Square”* metodi ar sekojošu formulu: $$RMS = \sqrt{\frac{1}{n}\sum_i x_i^2}$$ kur n (mērījumu skaits) ir *500* un x_i ir kārtējā vērtība (sensora mērījums), **Lai** sistēma spētu veikt korektu reāllaika sensoru datu apstrādi un analīzi.
Akceptēšanas kritēriji
• Sistēma apstrādā sensoru datus atbilstoši formulām.

Tab.2.11. Lietotājstāsts: **Datu pārraide**

Lietotājstāsta identifikators
S4_MULTICAST
Lietotājstāsts

Kā sistēmas lietotājs **Vēlos**, lai, pārlūkprogrammā palaižot treniņu, palaižas apstrādāto sensoru datu un informācijas par treniņa gaitu pārraide, **Lai** AR lietotne varētu saņemt un attēlot treniņa datus pacientam.
Akceptēšanas kritēriji
• Noklikšķinot uz pogas "*Start!*", sistēma sāk apstrādāto sensoru datu pārraidi atbilstoši treniņa soļiem. • Noklikšķinot uz "*Start!*", informācija par treniņa gaitu nonāk līdz AR brillēm, lai AR lietotne pacientam attēlotu treniņa vingrinājumu soļus atbilstoši palaistā treniņa gaitai pārlūkprogrammā – kad kārtējais solis ieslēdzās, AR brilles uzreiz attēlo to pacientam. • Sistēmas lietotājs var redzēt, kādi dati tiek sūtīti.

Tab.2.12. Sprinta uzdevums: **Datu pārraides ziņojuma formāts**

Sprinta uzdevuma identifikators
S4_MULTICAST_MESSAGE
Apraksts
Jārealizē sekojošs datu pārraides ziņojumu formāts: `[stepNr, sensorValue, prevSensorValue, maxSensorValue]`, kur: • `stepNr` ir tekošā treniņa soļa numurs, • `sensorValue` ir korekti apstrādāta kārtējā sensora vērtība, • `prevSensorValue` ir pēdējā sensora vērtība – šī vērtība būs rezervēta nākamajām sistēmas versijām; šajā versijā pēc AR lietotnes izstrādātāja pieprasījuma tai vienmēr jābūt vienādai ar 1, • `maxSensorValue` ir maksimālā vērtība - šī vērtība būs rezervēta nākamajām sistēmas versijām; šajā versijā pēc AR lietotnes izstrādātāja pieprasījuma tai vienmēr jābūt vienādai ar 1.
Ievade
Nav.
Apstrāde
Nav.
Izvade
Nav.

2.2. Nefunkcionālās prasības

2.2.1. Uzturamība

Lai garantētu programmatūras uzturamību, ir jāievēro lietojumprogrammatūras "OpenSignals" TCP ārējās saskarnes dokumentācijas [5] prasības sensoru datu iegūšanai.

3. PROGRAMMATŪRAS PROJEKTĒJUMA APRAKSTS

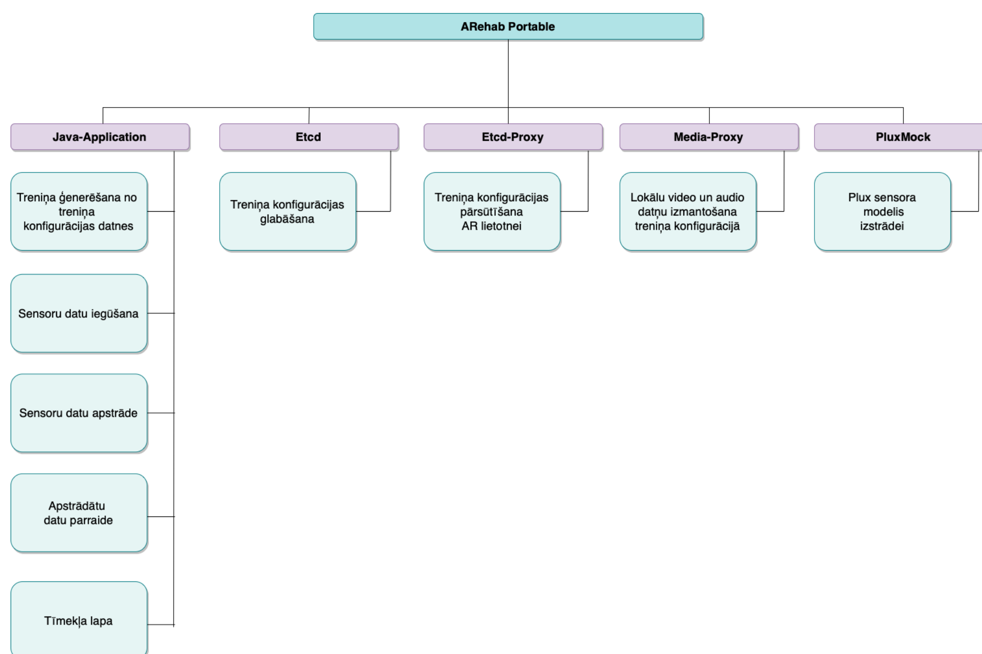
3.1. Dekompozīcijas apraksts

Sistēma "ARehab Portable" ir sadalīta piecos sistēmas komponentos – apakšsistēmas "Java-Application", "Etcd", "Etcd-Proxy", "Media-Proxy" un programma "PluxMock". Četras apakšsistēmas kopā veido lietotni, kas ir domāta sistēmas lietotājam, savukārt programma "PluxMock" ir paredzēta sistēmas izstrādātājiem un testētājiem.

Lai realizētu sistēmas vieglāku pārnesamību un uzturēšanu, tika nolemts izmantot Docker rīku sistēmas komponentu palaišanai izolētos konteineros, kas nav atkarīgi no ārējās vides, jo Docker izmanto operētājsistēmas līmeņa vizualizāciju. Katrs no sistēmas komponentiem "Java-Application", "Etcd", "Etcd-Proxy" un "Media-Proxy" ir atsevišķs Docker konteineris. Konteineru vienlaicīgai palaišanai tiek nolemts izmantot Docker komponēšanu – visi konteineri ir aprakstīti datnē "*docker-compose.yml*", kas palaiž visus konteinerus vienlaicīgi. Vienīga papildus procedūra, kuru nepieciešams veikt, lai palaistu sistēmu citā vidē, ir uzinstalēt lietojumprogrammu "Docker Desktop", ja vides operētājsistēma ir Windows vai MacOS (vai "Docker" Linux operētājsistēmai), ja iepriekš tas nebija izdarīts.

Programma "PluxMock" nav atsevišķs Docker konteineris, jo to nav nepieciešams piegādāt gala lietotājam – tā ir paredzēta izstrādes un testēšanas nolūkiem, tāpēc "PluxMock" ir izpildāma programmas datne.

Diagrammā zemāk ir attēlots sistēmas "ARehab Portable" sadalījums komponentos, katram komponentam ir apkopotas tā galvenās augsta līmeņa funkcijas (sk. *Att.3.1.*).



Att.3.1. Diagramma: **Sistēmas "ARehab Portable" komponenti un to galvenās funkcijas**

Apakšsistēma “Java-Application” ir vissvarīgākā sistēmas sastāvdaļa, kas nodrošina galveno sistēmas loģiku. Tā atbild par sensoru datu iegūšanu, apstrādi, treniņa ģenerēšanu no lietotāja pārvaldītās konfigurācijas datnes, treniņa palaišanu, treniņa un sensoru datu pārraidi, tīmekļa lapas darbību. Apakšsistēma ir sadalīta vairākos moduļos. Tā ir realizēta programmēšanas valodā Java, izmantojot Gradle rīku moduļu uzbūvēšanai un Spring satvaru apakšsistēmas komponentu pārvaldei un palaišanai.

Apakšsistēma “Etcd” ir sistēmas komponents, kas atbild par treniņa konfigurācijas glabāšanu tādā veidā, kas nodrošina treniņa konfigurācijas pieejamību AR brillēm. Komponents tiek palaists kā atsevišķs Docker konteineris, tomēr, atšķirībā no pārējiem Docker konteineriem, šim nav atsevišķā Dockerfile, jo tiek izmantots jau gatavs publiski pieejams Docker attēls “*bitnami/etcd:latest*” [6].

Apakšsistēma “Etcd-Proxy” atbild par AR lietotnes pieprasījumu apstrādi un treniņa konfigurācijas pārsūtīšanu AR lietotnei pēc pieprasījuma. Komponents darbojas kā starpniekserveris starp “Etcd” un AR lietotni.

Apakšsistēma “Media-Proxy” ir sistēmas komponents, kas pārķer visus ar treniņa konfigurācijā izmantotām audio un video datnēm saistītos AR lietotnes pieprasījumus un atgriež tai pieprasītās datnes no sistēmas lietotāja datora. Komponents darbojas kā starpniekserveris starp datņu serveri jeb sistēmas lietotāja datoru un AR lietotni.

Programma “PluxMock” nav daļa no gala lietotājam paredzētās sistēmas. Tās realizācijai tika izmantota programmēšanas valoda Java.

Tabulā zemāk ir apkopota informācija par to, kuras sistēmas prasības vai to daļas realizē katrs sistēmas komponents (sk. *Tab.3.1.*).

Tab.3.1. Sistēmas komponenti un to realizētās sistēmas prasības

Sistēmas komponents	**Funkcionālās prasības, kuras komponents realizē**
Apakšsistēma **“Java-Application”**	*S1_SENSOR_DATA_ACQUISITION* *S2_WORKOUT_CONFIG* *S2_ WORKOUT_CONFIG_AR* *S2_STEPS_TEMPLATES_GROUPS* *S2_DEVICES_CONFIG* *S3_ WEB_WORKOUT* *S3_SENSOR_CONFIG_WEB* *S3_WORKOUT_LOCALIZATION* *S4_PATIENT_EFFORT_ACCUMULATOR* *S4_MULTICAST* *S4_MULTICAST_MESSAGE*
Apakšsistēma **“Etcd”**	*S2_ WORKOUT_CONFIG_AR*
Apakšsistēma **“Etcd-Proxy”**	*S2_ WORKOUT_CONFIG_AR*
Apakšsistēma **“Media-Proxy”**	*S2_ WORKOUT_CONFIG_AR*
Izpildāmā programma **“PluxMock”**	*S1_PLUX_MOCK*

3.2. Atkarību un resursu apraksts

Apakšsistēma “Java-Application” ir atkarīga no komponenta-apakšsistēmas “Etcd”. Tās pieprasītais resurss ir lietotāja pārvaldītā treniņa un sistēmas konfigurācijas datne “*application.yml*”. Apakšsistēma saņem sensoru datu plūsmu no lietojumprogrammatūras “OpenSignals” vai nejauši uzģenerētos datus tādā pašā formātā no izstrādei un testēšanai paredzētās programmas “PluxMock”. Apakšprogramma sūta treniņa konfigurāciju uz “Etcd”. Kā arī, “Java-Application” atver reāllaika datu plūsmu uz AR brillēm, kas, atbilstoši tekošām treniņa solim (ja treniņš ir palaists), satur apstrādātus reāllaika sensoru datus un tekošā treniņa soļa numuru.

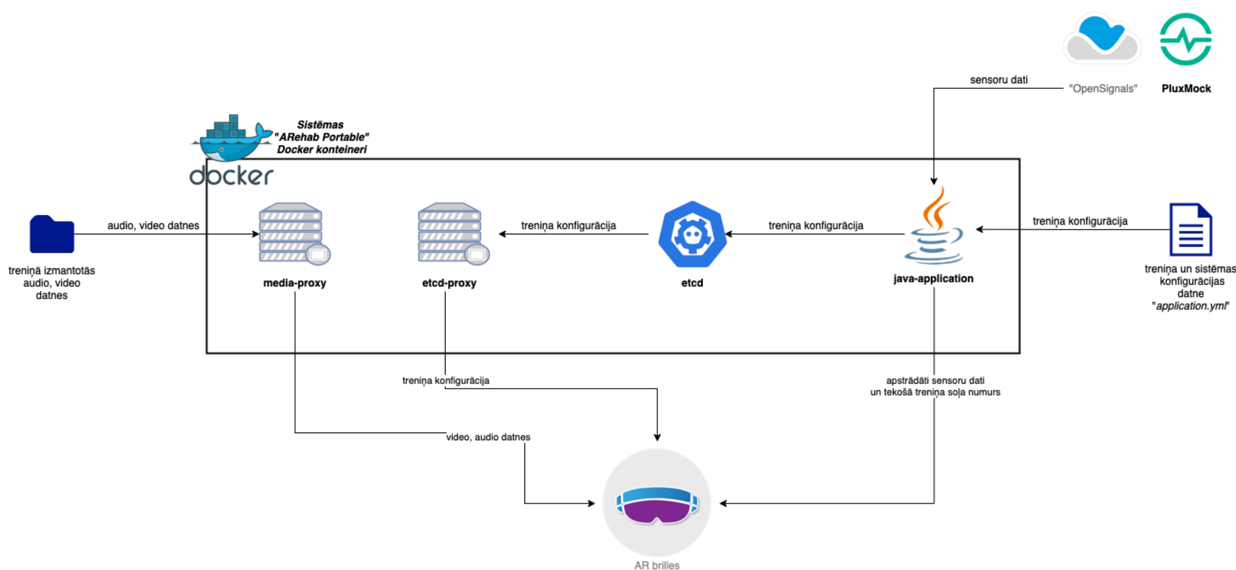
Apakšsistēma “Etcd” saņem pieprasījumus uz treniņa konfigurācijas saglabāšanu no “Java-Application” un pieprasījumus uz treniņa konfigurācijas lasīšanu no “Etcd-Proxy”.

Apakšsistēma “Etcd-Proxy” ir atkarīga no komponenta-apakšsistēmas “Etcd”. Tā saņem pieprasījumus uz treniņa konfigurācijas iegūšanu no AR brillēm, sūta pieprasījumu uz treniņa konfigurācijas lasīšanu uz “Etcd” un, saņemot konfigurāciju, sūta to uz AR brillēm.

Apakšsistēmas “Media-Proxy” pieprasītais resurss ir sistēmas lietotāja datorā glabājamās audio, video datnes, kuras tiek izmantotas treniņa konfigurācijā. “Media-Proxy” saņem pieprasījumus uz datņu lasīšanu no AR brillēm un sūta attiecīgās datnes.

3.3. Saskarņu apraksts

Diagrammā zemāk ir shematiski parādīts, kā sistēmas komponenti mijiedarbojas viens ar otru (sk. *Att.3.2*). Sistēmas komponenti, izņemot “PluxMock”, ir attēloti kā Docker konteineri.



Att.3.2. Diagramma: **“ARehab Portable” sistēmas komponentu mijiedarbība**

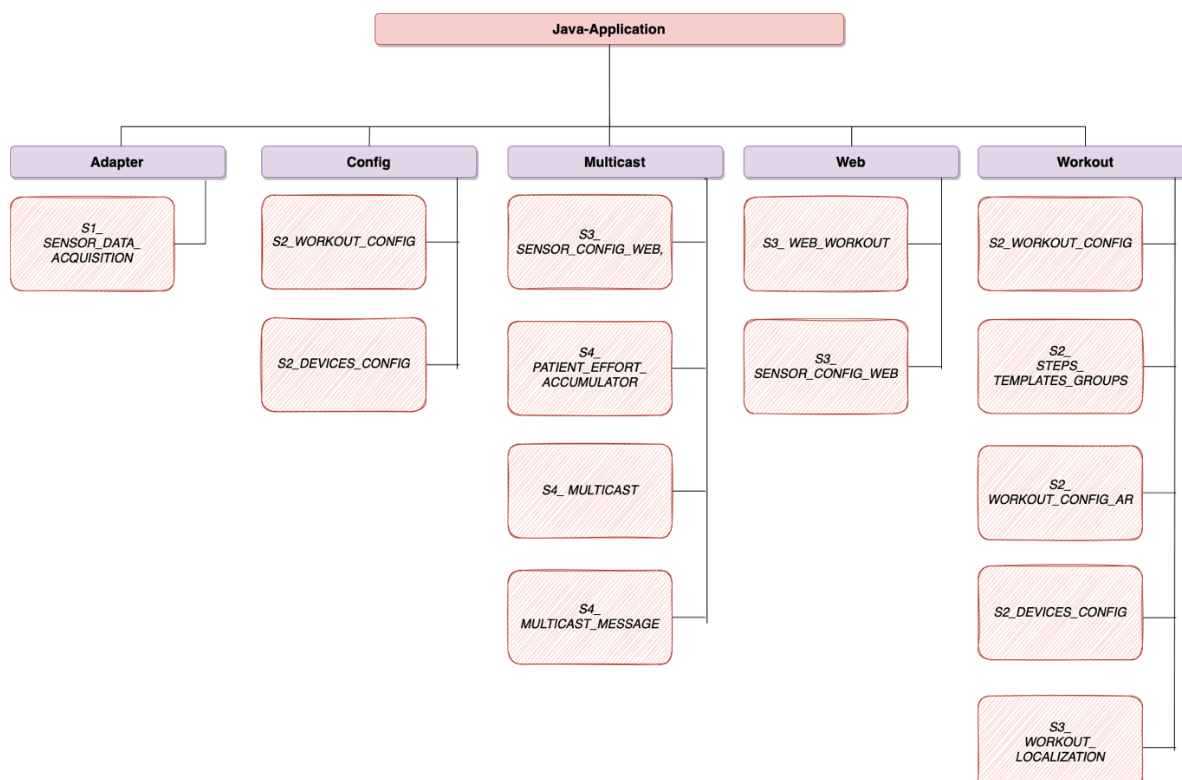
Lai palaistu sistēmu, treniņa un sistēmas konfigurācijas datne "*application.yml*" Docker komponēšanas datnē "*docker-compose.yml*" jānorāda kā Docker sējums, lai komponenta "Java-Application" Spring lietotne spētu to identificēt un izmantot kā lietotnes resursu. Lai "Java-Application" varētu saņemt sensoru datus no "OpenSignals" vai "PluxMock", konfigurācijas datnē ir jānorada vieta, uz kuru nāks sensoru datu plūsma - datnē tie ir lauki `pluxHostname` un `pluxPort`. Konfigurācijas datnē arī ir nepieciešams norādīt lauku `streamingHost` un `multicastPort`, kuri pasaka, uz kuru adresi "jānovirza" AR lietotni, lai tā varētu saņemt datu plūsmu (parastā gadījumā tā ir sistēmas lietotāja datora IP adrese). Šo adresi ir nepieciešams iekļaut treniņa konfigurācijā, kas tiek pārsūtīta no "Java-Application" uz "Etcd", lai AR lietotne caur "Etcd-Proxy" saņemtu adresi, uz kuru tai jāiet, lai saņemtu datu plūsmu.

Lai nodrošinātu treniņu konfigurācijas pārsūtīšanu starp konteineriem "Java-Application" un "Etcd", tos ir jāapvieno vienā Docker tīklā. Lai nodrošinātu treniņu konfigurācijas pārsūtīšanu starp konteineriem "Etcd" un "Etcd-Proxy", tos arī ir jāapvieno vienā Docker tīklā.

Lai "Media-Proxy" spētu izmantot lietotāja datora datnes, direktorija ar datnēm Docker komponēšanas datnē "*docker-compose.yml*" jānorāda kā Docker sējums.

3.4. Apakšsistēma "Java-Application"

3.4.1. Dekompozīcijas apraksts



Att.3.3. Diagramma: **Apakšsistēmas "Java-Application" sadalījums pa moduļiem**

Diagrammā augstāk (sk. *Att.3.3.*) ir shematiski attēlots apakšsistēmas sadalījums pa moduļiem, norādot katra moduļa realizējamās sistēmas prasības vai to daļas.

Modulis "Adapter" ir atbildīgs par sensoru datu iegūšanu no "OpenSignals". "Config" pārvalda treniņa konfigurācijas datni, kā arī satur svarīgas lietotnes konfigurācijas un pārvalda apakšsistēmas komponentus. Moduļa "Multicast" nolūks ir datu pārraide sistēmas prasībās pieprasītajā formātā. Moduļa "Web" uzdevums ir pārvaldīt tīmekļa lapu. Modulis "Workout" ģenerē un pārvalda treniņa konfigurācijas, kā arī atbild par treniņa lokalizāciju.

3.4.2. Atkarību apraksts

Līdz ar to, ka "Java-Application" ir Java Spring lietotne, tās komponentus un to atkarības pārvalda Spring IoC konteineris. Lietotnes galvenie komponenti un konfigurācijas ir apzīmēti ar Spring komponentu anotācijām *@SpringBootApplication*, *@Service*, *@Controller*, *@Configuration,* savukārt Spring *Bean* komponenti un to atkarības tiek definēti moduļa "Config" klasē "BeansConfig".

SpringBoot lietotnes klase "Application"

Moduļa klase "Application" satur izpildāmo metodi `main(String args[])` un ir apzīmēta ar SpringBoot anotāciju *@SpringBootApplication.* Šī metode palaiž visu lietotni. Klase pieprasa moduļa klases "ARehabPortable" instances klātbūtni un savā metodē "ARehabPortable" instancei izsauc metodi `start()`.

Klase-serviss "ARehabPortable"

Moduļa klase "ARehabPortable" ir apzīmēta ar Spring anotāciju *@Service*. Tās pieprasītie komponenti ir moduļa "Workout" klašu "WorkoutManager", "ClientUnityWorkoutService" objekti, moduļa "Multicast" klašu "PatientActivityPluxConnector", "PatientActivityAccumulator", "PatientActivityTcpMulticast" objekti, moduļa "Web" klases "WebSocketBroker" objekts. To klātbūtni sistēmā vienā eksemplārā nodrošina moduļa "Config" klasē "BeansConfig".

Klases metodē klases "ClientUnityWorkoutService" objektam tiek izsaukta tā klases metode `sendConfig()`, kas sūta treniņa konfigurāciju uz sistēmas komponentu "Etcd". Moduļa "Multicast" klašu objektiem tiek izsauktas to klašu metodes `start()`, klases "WorkoutManager" objektam tiek izsaukta tā klases metode `startManager()`, savukārt klases "WebSocketBroker" objektam tiek izsaukta tā klases metode `listen()`.

Klase-kontrolleris "ARehabController"

Klase ir apzīmēta ar Spring anotāciju *@Controller.*

Klase-kontrolleris "WebSocketBroker"

Klase ir apzīmēta ar Spring anotāciju *@Controller.*

Klase-konfigurācija “ARehabConfig”

Klase ir apzīmēta ar Spring anotāciju *@Configuration.*

Klase-konfigurācija “WebSocketConfig”

Klase ir apzīmēta ar Spring anotāciju *@Configuration.*

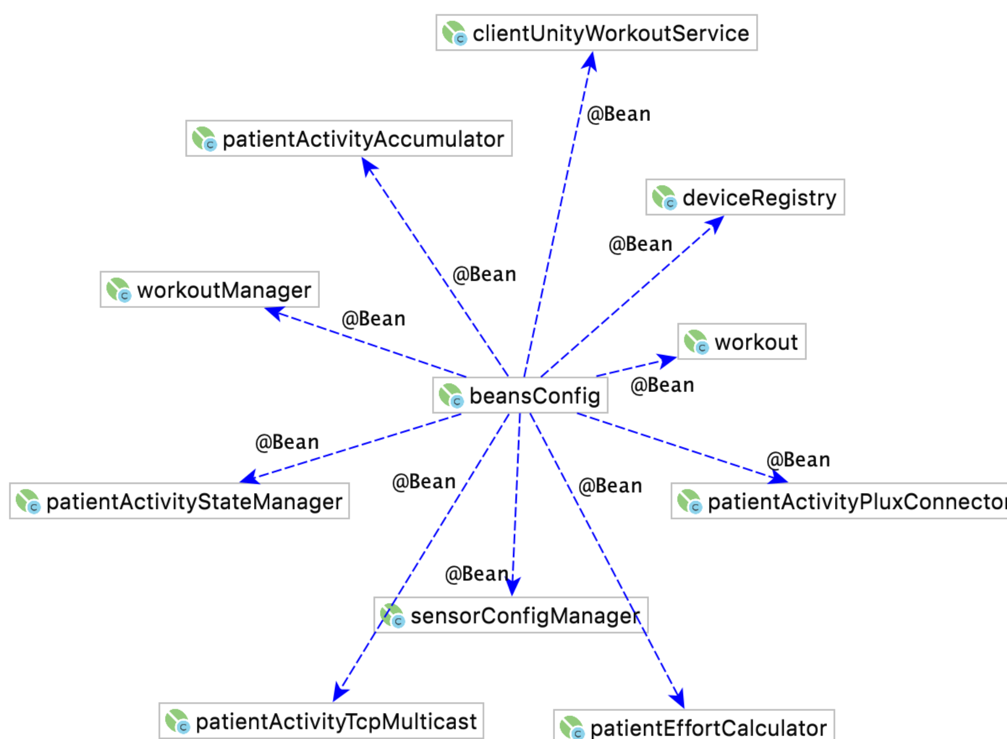
Klase-konfigurācija “WorkoutConfig”

Klase ir apzīmēta ar Spring anotāciju *@Configuration.*

Klase-konfigurācija “BeansConfig”

Klase ir apzīmēta ar Spring anotāciju *@Configuration.* Tā definē sistēmas *Bean* komponentus un to atkarības. Klase pieprasa klašu “WorkoutConfig”, “ARehabConfig” objektu klātbūtni, ko Spring nodrošina automātiski. Klasē “BeansConfig” tiek veidoti šādu klašu objekti un to atkarības - moduļa “Workout” klases “ClientUnityWorkoutService”, “WorkoutManager”, “Workout”; moduļa “Multicast” klases “PatientActivityStateManager”, “PatientActivityPluxConnector”, “PatientActivityAccumulator”, ‘PatientActivityTcpMulticast”, “PatientEffortCalculator”, “PluxDeviceRegistry”, “SensorConfigManager”.

Diagrammā zemāk (sk. *Att.3.4.*) ir parādītas klašu instances, ko veido “BeansConfig”.



Att.3.4. **Lietotnes “Java-Application” Spring *Bean* komponenti**

3.4.3. Detalizētais moduļu apraksts

3.4.3.1. Treniņa konfigurācijas ģenerēšana: modulis "Workout"

"Workout" ir sistēmas "ARehab Portable" apakšsistēmas "Java-Application" modulis, kas atbild par treniņa konfigurācijas ģenerēšanu, glabāšanu un pārsūtīšanu. Līdz ar to, ka sistēmai nav domāta datu bāze, visi dati tiek glabāti Java programmas atmiņā.

Moduļa nolūks

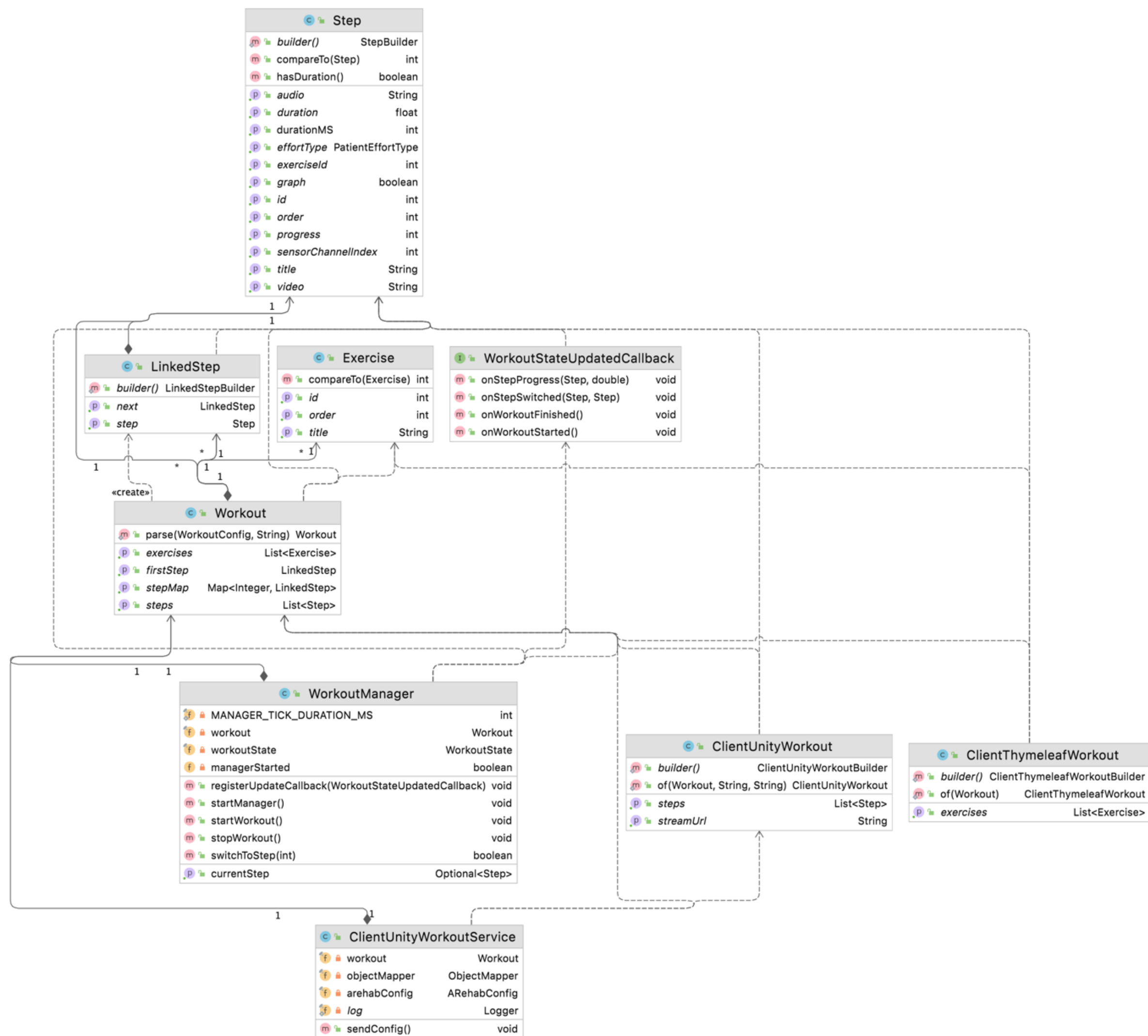
Pēc sistēmas funkcionālajām prasībām, treniņa ģenerēšanai jānotiek no sistēmas lietotāja pārvaldītās treniņa konfigurācijas datnes (lietotājstāsts *S2_WORKOUT_CONFIG*). Modulis "Workout" neizmanto treniņa konfigurācijas datni (datnes struktūra ir aprakstīta pie moduļa "Config" apraksta) kā resursu tiešā veidā – ar to nodarbojas moduļa "Config" klase "WorkoutConfig", kas apraksta treniņa konfigurācijas datni ar Java datu struktūrām.

Viens no moduļa "Workout" nolūkiem ir pārvērst treniņa konfigurāciju par atsevišķiem treniņa soļiem un pārveidot to citiem sistēmas komponentiem nepieciešamajos veidos. Piemēram, treniņa konfigurācijas datnē daži soļi tiek apvienoti soļu grupās vai apkopoti soļu šablonos vietas taupīšanai (sprinta uzdevums *S2_STEPS_TEMPLATES_GROUPS*), bet tāds treniņa konfigurācijas formāts nav piemērots tīmekļa lappusei un AR lietotnei. Moduļa "Workout" uzdevums ir pārvērst soļu grupas un šablonus par atsevišķiem soļiem, ģenerējot treniņa konfigurāciju ar soļu sarakstu tādos veidos, kuros tā vēlāk tiks izmantota tīmekļa lappusē un AR lietotnē.

Modulis "Workout" pēc sistēmas palaišanas saņem moduļa "Config" klases "WorkoutConfig" instanci un interpretē tās saturu, pārveidojot to par atsevišķiem treniņa soļiem un saglabājot tos atmiņā. Kopā ar "WorkoutConfig" instanci, modulis "Workout" saņem arī informāciju par lietotāja izvēlēto sistēmas valodu, lai zinātu, kuras valodas vērtības ir jāņem no soļu atribūtiem - šādā veidā tiek apmierināta prasība par sistēmas lokalizāciju (lietotājstāsts *S3_WORKOUT_LOCALIZATION*).

Cits moduļa "Workout" nolūks ir uzturēt un pārsūtīt treniņa konfigurāciju konkrētam sistēmas komponentam nepieciešamajā veidā. Divi sistēmas komponenti, kuriem modulis "Workout" sagatavo un sūta treniņa konfigurāciju noteiktā formātā, ir apakšsistēmas "Java-Application" modulis "Web" un apakšsistēma "Etcd".

Klašu diagrammā zemāk (sk. *Att.3.5.*) ir parādītas visas moduļa "Workout" klases, saskarne un to atkarības. Dažas klases tika apvienotas pakotnē, kas diagrammā netiek attēlots.



Att.3.5. Klašu diagramma: **Moduļa “Workout” klases, saskarne un to atkarības**

Pakotne “workout.dto”

Treniņa konfigurācijas saglabāšana Java programmas atmiņā notiek, izmantojot DTO objektus, kas pārsūta datus starp procesiem. Moduļa “Workout” DTO modeļi ir aprakstīti moduļa pakotnes “dto” klasēs: “Step”, “LinkedStep”, “Exercise”, “Workout”, “ClientThymeleafWorkout”, “ClientUnityWorkout”.

Klase “Step”

Klase apraksta treniņa soļa struktūru. Līdz ar to, ka treniņa soļiem jābūt noteiktā kārtībā, lai veidotos treniņš, klasei ir jāimplementē Java saskarni *Comparable<Step>*. Klases lauki apraksta visus treniņa soļa atribūtus no treniņa konfigurācijas datnes (datnes struktūra ir aprakstīta pie moduļa “Config” apraksta). Viens no tiem ir lauks `PatientEffortType effortType`, kas ir klases no moduļa “Multicast” instance.

Klasei ir lauks `exerciseId`, kas savieno doto soli ar vingrojumu, kuram tas pieder. Šī attiecība reprezentē attiecību *viens-pret-daudziem* – vienam vingrojumam pieder daudzi soļi.

Klase implementē saskarnes *Comparable* metodi veidā `compareTo(Step step)` un atgriež starpību soļu kārtas numuros (lauks `order`).

Klasei ir arī sekojošas metodes:

- `hasDuration()` – atgriež patieso vērtību, ja solim ir norādīts ilgums;
- `getDurationMS()` – atgriež soļa ilgumu milisekundēs.

Klase "LinkedStep"

Klase apraksta struktūru, kas apvieno konkrētu soli ar norādi uz nākamo soli. Tas ir nepieciešams, jo treniņa soli ir jāapvieno vingrojumos, un vingrojumus – treniņā, saglabājot soļu kārtību.

Klase "Exercise"

Klase apraksta treniņa vingrojumu. Vingrojumam ir identifikators, kārtas numurs un nosaukums. Līdz ar to, ka vingrojumiem jābūt noteiktā kārtībā, lai veidotos treniņš, klasei ir jāimplementē Java saskarne *Comparable<Step>* un tās metode veidā `compareTo(Exercise exercise)`, atgriežot starpību vingrojumu kārtas numuros.

Klase "Workout"

Klases lauki apraksta treniņa struktūru, kas sastāv no vingrojumu un soļu sarakstiem.

Klases statiska metode `parse(WorkoutConfig workoutConfig, String userLang)` tiek izsaukta no konfigurācijas klases "BeansConfig" (modulis "Config") pēc sistēmas palaišanas. Metode atgriež klases "Workout" instanci. Metodes realizācija paredz klases "Workout" iekšējās (statiskas) klases "WorkoutParser" izmantošanu. Klases "WorkoutParser" metodes ir sekojošas:

- `parse(String userLang)` – tiek izsaukta no klases "Workout" metodes `parse(WorkoutConfig, String)`, katram solim izsauc metodi `parseStep(Step, int, String)` un ģenerē "Workout" klases instanci, kuru pieprasa "BeansConfig";
- `parseStep(WorkoutConfig.Step step, int exerciseId, String lang)` – apraksta soļu interpretēšanas loģiku, soļu grupām izsauc metodi `applyGroup(String,int)`, soļu šabloniem izsauc metodi `getTemplate(String)`;

- `applyGroup(String groupName, int exerciseId, String lang)` – no soļa grupas veido sarakstu ar soļiem;
- `getTemplate(String templateName)` – no soļa šablona veido soli.

Klase "ClientThymeleafWorkout"

Klase apraksta treniņa konfigurācijas formātu, kas ir vajadzīgs modulim "Web", un attiecīgi iekļauj treniņa konfigurācijā šim modulim nepieciešamus laukus. Klases statiska metode `of(Workout workout)` ģenerē klases "ClientThymeleafWorkout" instanci, kas reprezentē treniņa konfigurāciju "Web" modulim nepieciešamajā veidā. Metode tiek izsaukta no moduļa "Web" klases-kontrollera "ARehabController".

Klase "ClientUnityWorkout"

Klase apraksta treniņa konfigurācijas formātu, kas ir vajadzīgs apakšsistēmai "Etcd" treniņa konfigurācijas saglabāšanai priekš AR lietotnes, un attiecīgi treniņa konfigurācijā iekļauj apakšsistēmai nepieciešamus laukus. Klases statiska metode `of(Workout workout, String streamingHost, String mediaHost)` ģenerē klases "ClientUnityWorkout" instanci. Metode tiek izsaukta no moduļa "Workout" klases "ClientUnityWorkoutService", kas, savukārt, griežas tieši pie "Etcd" apakšsistēmas, lai saglabātu treniņa konfigurāciju priekš AR lietotnes "Etcd" datu glabātavā.

Zemāk ir norādīts treniņa konfigurācijas formāts – klases "ClientUnityWorkout" instance JSON formātā, kas tiek saglabāta Etcd datu glabātavā priekš AR lietotnes (sk. *Att.3.6.*).

```
{
    "streamUrl": "localhost:8888",
    "steps": [
        {
            "id": 1,
            "description": "Example step 1",
            "videoUrl": "http://localhost:8082/media/example.mp4",
            "audioUrl": null,
            "showGraph": false
        },
        {
            "id": 2,
            "description": "Example step 2",
            "videoUrl": null,
            "audioUrl": "http://localhost:8082/media/example.mp3",
            "showGraph": false
        }
    ]
}
```

Att.3.6. **Treniņa konfigurācijas testa fragments JSON formātā**

Citas moduļa "Workout" klases un saskarne, kas nav DTO modeļi, ir klases "WorkoutManager", "ClientUnityWorkoutService" un saskarne "WorkoutStateUpdatedCallback".

Saskarne "WorkoutStateUpdatedCallback"

Saskarne piedāvā sarakstu ar metodēm, kas dod informāciju par treniņa un soļu progresiem: `onWorkoutStarted()`, `onWorkoutFinished()`, `onStepSwitched(Step currentStep, Step previousStep)`, `onStepProgress(Step step, double progress)`. Šīs metodes tiek izmantotas moduļa "Workout" klases "WorkoutManager" iekšējā (statiskā) klasē "WorkoutState", lai mainītu treniņa un tekošā soļa stāvokļus, un apakšsistēmas "Java-Application" moduļa "Web" klasē "WebSocketBroker", lai sūtītu atbilstošus ziņojumus par treniņa un soļu progresiem saziņā pa *WebSocket* protokolu reāllaika treniņa un soļu progresa attēlošanai tīmekļa lappusē.

Klase "WorkoutManager"

Klase satur iekšēju (statisku) klasi "WorkoutState", kas reprezentē un atbild par aktuālu palaistā treniņa stāvokli. Klases "WorkoutManager" metode `startManager()` veido pavedienu, kas ilgstoši izsauc klases "WorkoutState" metodi `tick()`, kas, savukārt, gaida, kamēr netiks palaists treniņš. Kad treniņš ir palaists, klase "WorkoutState" sāk atbildēt par tekošo soli, tā reāllaika progresu, soļu pārslēgšanu (lauka `currentStep` atjaunošanu) un izsauc attiecīgos saskarnes "WorkoutStateUpdatedCallback" piedāvātās metodes.

Klases "WorkoutManager" instance tiek veidota moduļa "Config" klasē "BeansConfig" un tiek izmantota kā Spring komponents klasē "PatientActivityPluxConnector". Kā arī, klases "WorkoutManager" metode `startManager()` tiek izsaukta klasē "ARehabPortable", kas norāda uz to, ka "WorkoutManager" palaišana notiek uzreiz pēc sistēmas palaišanas. Pārējās klases "WorkoutManager" metodes – `startWorkout()`, `switchToStep(int stepId)`, `stopWorkout()`, `getCurrentStep()` – izsauc atbilstošās iekšējas klases "WorkoutState" metodes un tiek izmantotas apakšsistēmas "Java-Application" moduļa "Web" klasē "WebSocketBroker", kas atbild par treniņa palaišanu un pārvaldi no tīmekļa lapas.

Klase "ClientUnityWorkoutService"

Klases instance tiek veidota moduļa "Config" klasē "BeansConfig". Klase satur tikai vienu metodi – `sendConfig()`, kas tiek izsaukta klasē "ARehabPortable", kas palaiž sistēmu. Šī moduļa klase ir atbildīga par treniņa konfigurācijas saglabāšanu "Etcd" datu glabātavā, lai AR lietotne varētu to nolasīt un izmantot (lietotājstāsts *S2_WORKOUT_CONFIG_AR*). Metode ģenerē klases "ClientUnityWorkout" instanci no uzģenerētā un Java atmiņā pastāvošā treniņa (klases "Workout" instances) un tad sazinās tieši ar apakšsistēmu "Etcd". Lai sazinātos

ar apakšsistēmas “Etcd” Docker konteineri, metode izmanto apakšsistēmas “Java-Application” moduļa “Config” klases-konfigurācijas “ARehabConfig” lauku `etcdHost`. Lauki, kas ir nepieciešami “ClientUnityWorkout” klases instances izveidošanai, izmantojot “ClientUnityWorkout” klases metodi `of(Workout workout, String streamingHost, String mediaHost)`, arī tiek ņemti no klases-konfigurācijas “ARehabConfig”.

3.4.3.2. Sensoru datu iegūšana no sensoriem: modulis “Adapter”

Modulis “Adapter” ir apakšsistēmas “Java-Application” modulis, kura nolūks ir sazināties ar lietojumprogrammatūru “OpenSignals” ar mērķi iegūt sensoru datus no Plux elpošanas un kustību sensoriem.

Moduļa nolūks

Moduļa “Adapter” mērķis ir realizēt sistēmas funkcionālās prasības par sensoru datu iegūšanu (lietotājstāsts *S1_SENSOR_DATA_ACQUISITION*). Modulis nodrošina sistēmai iespēju pieslēgties lietojumprogrammatūras “OpenSignals” sensoru datu plūsmai, izmantojot tās saskarni, un iegūt datus no Plux elpošanas un kustību sensoriem. “OpenSignals” TCP saskarne ir detalizēti aprakstīta tās oficiālajā dokumentācijā [5].

Realizējot sensoru datu iegūšanu, tika ņemta vērā iespēja, ka nākotnē sistēmas pasūtītājs pieprasīs, lai sistēma spētu strādāt arī ar cita veida sensoriem, kas var nebūt BITalino Plux sensori. Līdz ar to, realizējot adapterus sensoru datu plūsmai, tika veidotas vairākas saskarnes, kuru individuāla realizācija varēs nākotnē atšķirties atkarībā no sensora ražotāja.

Modulis iekļauj pakotnes “bitalino.callback”, “bitalino.plux” un atsevišķas saskarnes.

Pakotne “bitalino.callback”

Pakotnē tiek aprakstītas saskarnes, kas reprezentē uzvedību, kuru jārealizē, implementējot “callback” metodes BITalino sensoriem jeb tādas metodes, kas tiek izsauktas, kad iestājas kāds notikums saistībā ar BITalino sensoru datu plūsmu no lietojumprogrammatūras “OpenSignals” (sk. *Att.3.7.*).

BitalinoConnectionEstablishedCallback	
onEstablished()	void

BitalinoConnectionFailedCallback	
onFail()	void

BitalinoOutputCallback	
onCharReceived(char)	void

BitalinoSensorDataCallback<T>	
onSensorData(T)	void

Att.3.7. **Moduļa "Adapter" pakotnes "dto" saskarnes**

Saskarne "BitalinoConnectionEstablishedCallback"

Saskarne definē metodi, kuru jārealizē un jāizsauc, kad ir izveidots savienojums ar sensoru datu plūsmu – metode `onEstablished()`.

Saskarne "BitalinoConnectionFailedCallback"

Saskarne definē metodi, kuru jārealizē un jāizsauc, kad nav izdevies izveidot savienojumu ar sensoru datu plūsmu– metode `onFail()`.

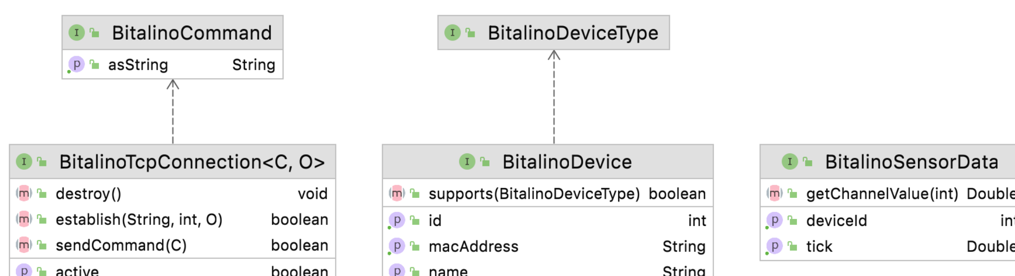
Saskarne "BitalinoOutputCallback"

Saskarne definē metodi, kuru jārealizē un jāizsauc, kad no datu plūsmas ir izdevies iegūt kaut kādus izvades datus – metode `onCharReceived(char)`. Lietojumprogrammas "OpenSignals" TCP saskarnei ir noteikts formatējums sensoru datu izvadei, kuru sistēmai ir jāpārzina un jāmāk apstrādāt.

Saskarne "BitalinoSensorDataCallback"

Saskarne definē metodi, kuru jārealizē un jāizsauc, kad no sensoru datu plūsmas ir izdevies iegūt sensoru mērījumus – metode `onSensorData()`. Sistēmai, apstrādājot no datu plūsmas iegūtos simbolu virknes, jāatšķir, kurā brīdī beidzas formatējums un sākas reāli sensoru dati.

Pārējās šī moduļa saskarnes, kas neatrodas pakotnē "callback" un kuru realizācijas ir atrodamas pakotnē "bitalino.plux", ir saskarnes "BitalinoCommand", "BitalinoDevice", "BitalinoStepSensorData" un "BitalinoTcpConnection" (sk. *Att.3.8.*).



Att.3.8. **Moduļa "Adapter" saskarnes**

Saskarne "BitalinoCommand"

Saskarne definē metodi, kuru jārealizē, implementējot šo saskarni konkrētam sensoram – metode `getAsString()`. Komandas ir daļa no "OpenSignals" lietojumprogrammatūras dokumentācijas definētās saskarnes [5].

Saskarne "BitalinoDevice"

Saskarne definē metodes, kurus jārealizē, implementējot šo saskarni konkrētam sensoram – metodes `getId()`, `getName()`, `supports(BitalinoDeviceType)`, `getMacAddress()`. Tā ir informācija, kuru sistēmai jāzina par katru sensoru – sistēmas lietotājs ieliek šo informāciju konfigurācijas datnes *"application.yml"* pirmajā daļā, par ko atbild moduļa "Config" klase-konfigurācija "ARehabConfig".

Saskarne "BitalinoDeviceType"

Saskarne apraksta piepūles veidu, kuram konkrētais sensors tiek izmantots – piemēram, elpošanu vai kustības. Saskarne nepiedāvā nekādas metodes, bet pastāv šajā modulī, lai, realizējot konkrētā sensora klases, izstrādātājs ņemtu vērā, ka sensoram jāmāk norādīt, kuram piepūles veidam to izmanto.

Saskarne "BitalinoSensorData"

Saskarne definē metodes, kurus jārealizē, implementējot šo saskarni konkrētam sensoram – metodes `getDeviceId()`, `getTick()`, `getChannelValue(int)`. Saņemot sensoru datus no "OpenSignals", sistēmai katrā brīdī jāsaprot, no kura sensora tie nāk un no kura sensora kanāla jāņem dati tālākai apstrādei.

Saskarne "BitalinoTcpConnection"

Saskarne ir ģenēriskā, tai ir divi tipu parametri – `<C extends BitalinoCommand>` un `<O extends BitalinoOutputCallback>`. Saskarne definē metodes, kurus jārealizē, implementējot saskarni konkrētam sensoram – `establish(String, int, O)`, `isActive()`, `sendCommand(C)`, `destroy()`. Saskarne apraksta klasi, kuras nolūks ir izveidot TCP savienojumu ar "OpenSignals" un pārvaldītu to.

Pakotne "bitalino.plux"

Pakotnē atrodas klases un saskarnes, kas realizē vai manto no iepriekšminētajām saskarnēm. Šīs klases un saskarnes ir domātas tieši Plux sensoriem.

Pakotne "bitalino.plux.callback"

Pakotnē tiek aprakstītas saskarnes, kas reprezentē uzvedību, kuru jārealizē, implementējot metodes Plux sensoriem, kuras tiek izsauktas, kad iestājas kāds notikums saistībā ar Plux sensoru datu plūsmu no "OpenSignals". Šīs pakotnes saskarnes manto no attiecīgām saskarnēm no pakotnes "bitalino.callback" un pati par sevi

nepiedāvā nekādas jaunas metodes. No saskarnes “BitalinoOutputCallback” nemanto neviena cita saskarne, taču to realizē šīs pakotnes klase “PluxOutputCallback”.

Saskarne “PluxConnectionEstablishedCallback”

Saskarne “PluxConnectionFailedCallback”

Saskarne “PluxSensorDataCallback”

Klase “PluxOutputCallback”

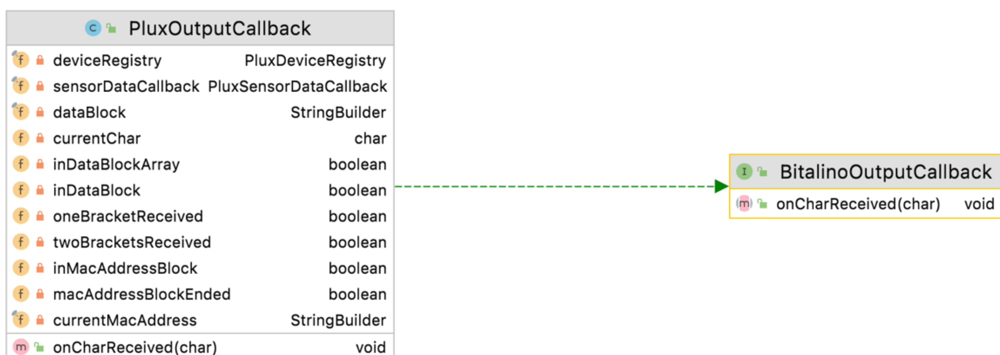
Klase realizē saskarni “BitalinoOutputCallback”. Tās metode `onCharReceived(char)` apraksta algoritmu, pēc kura sistēmai jādarbojas, no “OpenSignals” lietojumprogrammas sensoru datu plūsmas saņemot kaut kādus izvades datus. Sistēmai ir jāatšķir formatējumu no reāliem datiem, un šī klase atbild par formatējumu.

Zemāk ir norādīts “OpenSignals” lietojumprogrammas izvades sensoru datu (saīsināts labākai uzskatāmībai) fragments; ar sarkano un zaļo krāsām ir apzīmēti fragmenta daļas, kas sistēmu interesē – sensora MAC adrese, no kura tika iegūti dati, un paši sensora dati; viss pārējais ir formatējums (sk. *Att.3.9.*).

```
{"returnCode": 0, "returnData": {"00:00:00:00:00:3F": [[3150.0, 0.0, -
1.48728, 1.48805, -1.48723, -0.40745], [3151.0, 0.0, -0.72334, 0.75849,
-0.75922, 1.39836]], "11:11:11:11:11:3D": [[3172.0, 0.0, 1.50224, -
0.0764, -0.0932, -0.1676], [3173.0, 0.0, 1.50261, -0.077, -0.0906, -
0.169]]}}
```

Att.3.9. **“OpenSignals” lietojumprogrammas izvades sensoru saīsināts datu fragments**

Klases un tās metodes `onCharReceived(char)` mērķis ir “atmest” formatējumu, bet sistēmai svarīgas daļas uzkrāt un pārsūtīt uz vajadzīgām vietām. Attēlā zemāk ir parādīta klases struktūra (sk. *Att. 3.10.*).



Att.3.10. **Moduļa “Adapter” klases “PluxOutputCallback” struktūra**

Kārtējā sensora MAC adrese (datu fragmentā apzīmēta ar sarkano krāsu) tiek "uzkrāta" pa simboliem datu struktūras `StringBuilder` mainīgajā ar vārdu `currentMacAddress`, bet kārtējā sensora dati tiek uzkrāti pa simboliem datu struktūrā *StringBuilder* mainīgajā ar vārdu `dataBlock`.

Kad metode identificē kārtējā sensora datu bloka beigas, tiek inicializēts mainīgais `device` ar tipu "BitalinoDevice". Lai to inicializētu, metode izsauc moduļa "Multicast" klases "PluxDeviceRegistry" metodi `findByMacAddress(String)` – šī metode pārbauda, vai sensors, no kura tika iegūti dati, ir pieminēts kā sensors sistēmas lietotāja pārvaldītā konfigurācijas datnē *"application.yml"*; ja ir, tad metode atgriež objektu ar tipu "PluxDevice". Pēc tam klases "PluxSensorDataCallback" instancei tiek izsaukta metode `onSensorData(T)`, kā parametru padodot klases "PluxSensorData" statiskās metodes `ofString(int, String)` (kā parametru pieņem ierīces jeb sensora identifikatoru un sensora datus) atgriežamo vērtību. Tad mainīgajam `dataBlock` tiek anulēts virknes garums, un sākas datu iegūšana no nākamā sensora.

Pārējās pakotnes "bitalino.plux" klases un saskarnes, kas neatrodas pakotnē "callback", ir saskarne "PluxCommand" un klases "PluxAdapter", "PluxSensorData" un "PluxTcpConnection".

Saskarne "PluxCommand"

Saskarne manto no saskarnes "BitalinoCommand" un nepiedāvā nekādas papildus metodes. Tās realizācija notiek ar anonīmās klases palīdzību klasē "PluxAdapter".

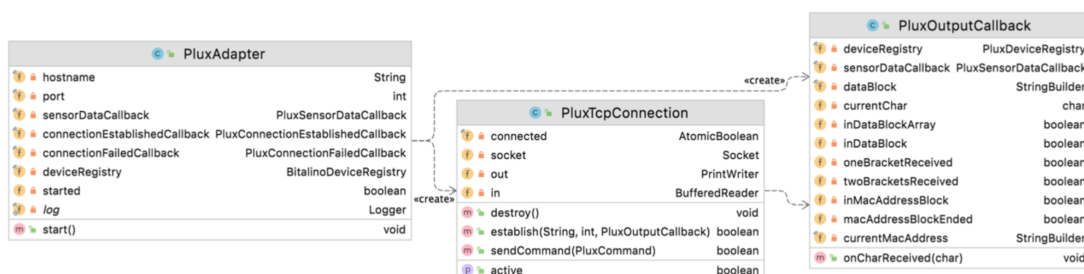
Klase "PluxAdapter"

Klases nolūks ir izveidot savienojumu ar lietojumprogrammas "OpenSignals" sensoru datu plūsmu un uzsākt sensoru datu iegūšanu. Ja savienojums ir izveidots, tiek parādīts paziņojums *"Plux connection established"*, ja nav – tiek parādīts paziņojums *"Plux connection failed. Retrying in 1s…"*, un tiek mēģināts atkārtoti izveidot savienojumu pēc 1 sekundes (lietotājstāsts *S1_SENSOR_DATA_ACQUISITION*).

Klasei ir viena metode `start()`, kas izveido pavedienu (*thread*). Pavedienā vispirms tiek izveidoti klašu "PluxTcpConnection" un "PluxOutputCallback" instances, klases "PluxOutputCallback konstruktoram padodot parametrus ar tipiem

"PluxDeviceRegistry" un "PluxSensorDataCallback". Tad tiek izveidota mūžīga cilpa, kurā tiek mēģināts izveidot savienojumu ar sensoru datu plūsmu, izsaucot klases "PluxTcpConnection" metodi `establish(String, int, PluxOutputCallback)`. Ja tas izdodas, tiek izvadīts augstākminētais paziņojums, tiek izsaukta klases "PluxConnectionEstablishedCallback" metode `onEstablish()` un tiek izsaukta klases "PluxTcpConnection" metode `sendCommand(PluxCommand)`, realizējot saskarni "PluxCommand" ar anonīmās klases palīdzību, kā komandu padodot virkni *"start"* (šī komanda, pēc "OpenSignals" dokumentācijas, uzsāk sensoru datu izvadi pieslēgtam TCP klientam [5]). Ja savienojumu izveidot neizdevās, tiek izvadīts augstākminētais paziņojums un tiek izsaukta klases "PluxConnectionFailedCallback" metode `onFail()`. Pavediens taisa sekundes ilgu pauzi un tad atkārto mēģinājumu.

Klašu diagrammā zemāk tiek parādīts klases "PluxAdapter" atkarības (sk. *Att. 3.11.*).



Att.3.11. **Moduļa "Adapter" klases "PluxAdapter" struktūra un atkarības**

Klase "PluxSensorData"

Klase realizē saskarni "BitalinoSensorData" un reprezentē vienu sensoru datu "vienību", kas sastāv no sensora identifikatora un saraksta ar šī sensora datiem.

Metode `getChannelValue(int channelIndex)` atgriež sensoru datu saraksta vērtību ar indeksu (`channelIndex + 2`), tāpēc, ka sensora datu blokā pirmās divas vērtības ir speciālās vērtības, pirmā apzīmē datu vienības kārtas numuru (*tick*) un otrā ir Plux rezervētā vērtība, kas vienmēr ir nulle. Tikai pēc šīm divām pozīcijām nāk katra sensora kanāla mērījums.

Zemāk ir norādīts 4-kanālu sensora datu bloka (saīsināts labākai uzskatāmībai) piemērs, ar zaļo krāsu tiek apzīmēti reāli sensora dati, bet ar sarkano – pirmās divas pozīcijas, kas nav sensora dati; katra zaļā vērtība ir kārtējā sensora kanāla mērījums (sk. *Att.3.12.*).

```
{"returnCode": 0, "returnData": {"00:00:00:00:00:3F": [[3150.0, 0.0, -
1.48728, 1.48805, -1.48723, -0.40745], [3151.0, 0.0, -0.72334, 0.75849,
-0.75922, 1.39836]]}}
```

Att.3.12. **4-kanālu Plux sensora datu bloka saīsināts piemērs**

Klases metode `ofString(int, String)` saņem sensora identifikatoru ar tipu *int* un simbolu virkni ar datiem un veido jaunu "PluxSensorData" instanci, pārvēršot virkni par vērtību ar tipu *Double* sarakstu.

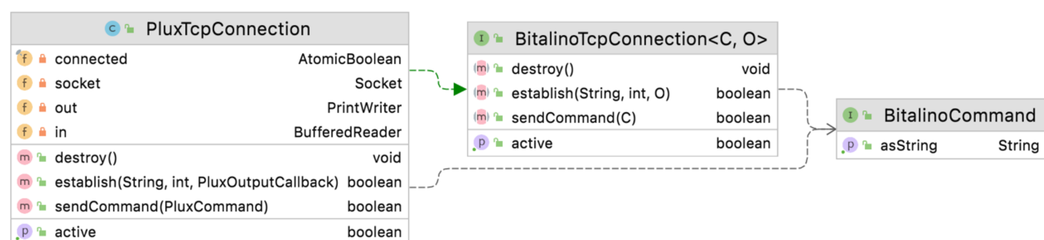
Klase "PluxTcpConnection"

Klase realizē ģenērisko saskarni "BitalinoTcpConnection" ar tipiem `<PluxCommand, PluxOutputCallback>`. Tās nolūks ir izveidot un pārvaldīt savienojumu ar "OpenSignals" programmatūras sensoru datu plūsmu. Tas tiek panākts ar *Socket* savienojuma palīdzību un tās ievades un izvades plūsmām. Ievades plūsma tiek izmantota, lai lasītu "OpenSignals" sūtītos baitus, savukārt izvades plūsma – lai varētu sūtīt komandu *"start"* [5] sensoru datu iegūšanai.

Klases galvenā metode `establish(String, int, PluxOutputCallback)` kā pirmos divus parametrus saņem klases-konfigurācijas "ARehabConfig" parametrus `pluxHostname` un `pluxPort`. Ar šiem parametriem tiek mēģināts izveidot jaunu Java klases *Socket* instanci `socket`, Java klases *PrintWriter* instanci `out` (kā parametru padodot `socket.getOutputStream()`), Java klases *BufferedReader* instanci `in` (kā parametru padodot `new InputStreamReader(socket.getInputStream())`). Tad tiek izveidots jauns pavediens, kurā tiek mēģināts ievadā lasīt simbolus pēc kārtas, pārveidot kārtējo simbolu par tipa *char* vērtību, un izsaukt klases "PluxOutputCallback" metodi `onCharReceived(char)`.

Klases metode `sendCommand(PluxCommand)` izvades plūsmai `out` izsauc metodi `println(String)`, lai izdrukātu komandu izvades plūsmā.

Diagrammā zemāk ir parādītas klases "PluxTcpConnection" realizējamā saskarne un atkarība (sk. *Att. 3.13.*).



Att.3.13. **Klases "PluxTcpConnection" realizējamā saskarne un atkarība**

3.4.3.3. Datu apstrāde un pārraide: modulis “Multicast”

Pēc sistēmas funkcionālajām prasībām, sistēmai ir jāveic iegūto sensoru datu apstrāde un pārraide (lietotājstāsti *S4_PATIENT_EFFORT_ACCUMULATOR*, *S4_ MULTICAST,* sprinta uzdevums *S4_MULTICAST_MESSAGE*). Par šo prasību realizāciju atbild apakšsistēmas “Java-Application” modulis “Multicast”.

Moduļa nolūks

Moduļa “Multicast” mērķis ir realizēt iegūto skaitlisku sensoru datu par pacientu fizisku piepūli apstrādi pēc funkcionālajās prasībās norādītajām formulām (lietotājstāsts *S4_PATIENT_EFFORT_ACCUMULATOR*) un pārraidi pieprasītajā formātā kopā ar informāciju par tekošo treniņa soli (lietotājstāsti *S4_PATIENT_EFFORT_ACCUMULATOR*, *S4_MULTICAST,* sprinta uzdevums *S4_MULTICAST_MESSAGE*).

Modulis sastāv no vairākām pakotnēm un atsevišķām klasēm un saskarnēm. Septiņas šī moduļa klases ir apzīmētas klasē-konfigurācijā “BeansConfig” ar Spring anotāciju *@Bean* un tiek veidotas sistēmas palaišanas brīdī kā sistēmas svarīgi komponenti.

Lai nodrošinātu datu pārraidi (multiraidi), modulis atver Java servera ligzdu *(ServerSocket)* uz porta, kas ir norādīts sistēmas lietotāja pārvaldītajā konfigurācijas datnē “*application.yml”* un aprakstīts klases-konfigurācijas “ARehabConfig” attiecīgajā laukā `multicastPort`. Servera ligzda apstrādā servera puses komunikāciju - tā gaida, kad tīklā tiks saņemti pieprasījumi. Modulī realizētais ziņojumu starpnieks pārvalda klientu savienojumus ar servera ligzdu un publicē ziņojumus sistēmas prasību definētajā formātā pieslēgtiem klientiem.

Pakotne “patientactivity”

Saskarne “PatientActivityConsumer”

Saskarne manto no ģenēriskās saskarnes “MulticastMessageConsumer” ar tipu *<PatientActivityMessage>*. Nekādas papildus metodes saskarne nepiedāvā.

Saskarne “PatientActivityMessage”

Saskarne manto no saskarnes “MulticastMessage”. Tās pievienotās metodes ir metodes `getStepId()`, `getEffort()`, `getPrevValue()`, `getMaxValue()`. Šīs metodes reprezentē īpašības, kurām jāpiemīt visiem datu pārraides ziņojumiem (sprinta uzdevums *S4_MULTICAST_MESSAGE*).

Klase “PatientActivityConsumerSocket”

Klase realizē saskarni “PatientActivityConsumer”. Klases lauki ir `id` ar datu tipu *int* un `out` ar datu tipu *PrintWriter*. Metode `getId()` atgriež attiecīgo lauku, savukārt metode `onMessage(PatientActivityMessage)` laukam `out` izsauc metodi

`println(String)`, kā parametru padodot metodes saņemtā ziņojuma simbolu virkni. Klase reprezentē datu pārraides klientu, kas saņems datu pārraides ziņojumus.

Klase "PatientActivityMessageBroker"

Klase manto no ģenēriskās klases "SimpleMulticastMessageBroker" ar parametru tipiem `<PatientActivityMessage, PatientActivityConsumer>`. Klase reprezentē ziņojumu starpnieku ziņojumu par pacienta fizisko aktivitāti pārraidei. Klases objekts tiek inicializēts moduļa "Config" klasē-konfigurācijā "BeansConfig" un padots kā parametrs citam sistēmas komponentam – klases "PatientActivityTcpMulticast" instancei.

Klase "PatientActivityMessageImpl"

Klase realizē saskarni "PatientActivityMessage" un reprezentē konkrētu sistēmas prasībās pieprasīto pārraides ziņojuma formātu (sprinta uzdevums *S4_MULTICAST_MESSAGE*). Klasei ir divi lauki – `stepId` ar tipu *int*, kas reprezentē konkrētajā brīdī ieslēgto treniņa soli, un `effort` ar tipu *double*, kas ir pacienta piepūle ieslēgtā soļa ietvaros – skaitlis, kas iepriekš jau tika apstrādāts ar vajadzīgām formulām.

Klasei ir četras *get* metodes. Metodes `getStepId()` un `getEffort()` atgriež attiecīgo lauku vērtības, savukārt metodes `getPrevValue()` un `getMaxValue()` atgriež skaitli 1. Pēdējās divas vērtības ir rezervētas nākotnes prasībām (sprinta uzdevums *S4_MULTICAST_MESSAGE*).

Klases metode `getAsString()` formatē un atgriež ziņojumu kā simbolu virkni, kas atbilst pieprasītajām ziņojuma formātam.

Pakotne "patientactivity.effort"

Saskarne "PluxPatientEffort"

Saskarne apraksta metodes, kuras jāimplementē, realizējot klasi, kas reprezentē pacienta piepūli. Šīs metodes ir `getStepId()`, `getType()` (atgriežamais datu tips – "PatientEffortType") un `getSensorValue()` (atgriežamais datu tips – *double*).

Klase "PluxPatientEffortImpl"

Klase realizē saskarni "PluxPatientEffort". Klasei ir saskarnes metodēm atbilstošie lauki.

Saskarne "AccumulatedPatientEffort"

Saskarne apraksta, kādas metodes jārealizē klasei, kas implementēs jau "uzkrāto" pacienta piepūli – tas ir, sensora mērījumi, kuriem jau tika piemērota

kāda no saraksta samazināšanas jeb datu uzkrāšanas stratēģijām. Saskarnes definētās metodes ir `getStepId()`, `getType()` un `getAccumulatedValue()`.

Klase "AccumulatedPatientEffortImpl"

Klase realizē saskarni "AccumulatedPatientEffort". Klasei ir saskarnes metodēm atbilstošie lauki.

Klase "NullPatientEffort"

Klase realizē saskarnes "PluxPatientEffort" un "AccumulatedPatientEffort". Tā reprezentē pacienta piepūles realizāciju ar neitrālu uzvedību – metode `getStepId()` atgriež skaitli -1, metodes `getSensorValue()` un `getAccumulatedValue()` atgriež skaitli 0.

Klase "PatientEffortType"

Klase realizē saskarni "BitalinoDeviceType" (modulis "Adapter"), Plux sensoru kontekstā idejiski saistot sensora tipu ar pacienta piepūles veidu. Klase definē divus piepūles veidus jeb sensoru tipus – *BREATHE* un *MUSCLE*. Klasei ir metode `getByName(String)`, kas saņem simbolu virkni un atgriež atbilstošu pacienta piepūles veidu. Šī metode tiek izsaukta no klases "Workout" (modulis "Workout"), ģenerējot treniņu, lai atvieglotu treniņa konfigurācijā norādītās simbolu virknes sasaistīšanu ar konkrētu sistēmā glabājamo piepūles veidu.

Klase "PatientEffortCalculator"

Klase tiek apzīmēta ar Spring anotāciju *@Bean* klasē-konfigurācijā "BeansConfig" un tās instance tiek veidota sistēmas palaišanas brīdī, kā parametru konstruktoram padodot klases "SensorConfigManager" izveidoto objektu.

Klasei ir viena metode `calculate(Step, PluxSensorData)`. Metode kalpo par dažu moduļu "satikšanas" vietu – metode kā parametrus pieņem klašu "Step" (modulis "Workout") un "PluxSensorData" (modulis "Adapter") objektus. Metode nosaka soļa piepūles veidu, tad iegūst attiecīgā sensora kanāla relatīvu vērtību, izsaucot klases "SensorConfigManager" metodi `inboundByType(PatientEffortType, PluxSensorData)` (kas aprēķina pareizā sensora kanāla relatīvu vērtību, izmantojot sensoru konfigurācijā norādītās minimālo un maksimālo vērtības). Beigās metode atgriež jaunu klases "PluxPatientEffortImpl" objektu.

Pakotne "patientactivity.effort.accumulator"

Saskarne "EffortAccumulator"

Saskarne apraksta metodes, kuras jārealizē, implementējot sensoru datu uzkrājēju – metodes `accumulate(double)` un `reset()`.

Saskarne "EffortAccumulatorFactory"

Saskarne definē metodi, kuru jārealizē, rakstot implementāciju klasei, kas atbild par datu uzkrājēju veidošanu katram piepūles veidam – metode `create(PatientEffortType)`.

Klase "EffortAccumulatorFactoryImpl"

Klase implementē saskarni "EffortAccumulatorFactory". Klasei ir divi lauki ar tipu *int* – `muscleAccumulatorFrameLength` un `breatheAccumulatorFrameLength`. Šo lauku vērtības tiek aprēķinātas moduļa "Config" klasē-konfigurācijā "ARehabConfig" (vērtības reprezentē mērījumu skaitu sistēmas prasībās norādītajās formulās (lietotājstāsts *S4_PATIENT_EFFORT_ACCUMULATOR*) un nonāk līdz klasei, kad tiek izveidota tās instance klasē "PatientActivityAccumulator", kuras objekts, savukārt, tiek veidots sistēmas palaišanas brīdī kā sistēmas komponents caur klasi-konfigurāciju "BeansConfig" un kas paņem šīs vērtības no konfigurācijas "ARehabConfig").

Klases metode `create(PatientEffortType)` atgriež jaunas klašu "BreatheEffortAccumulator" vai "MuscleEffortAccumulator" instances atkarībā no saņemtā piepūles tipa, kā konstruktora parametru padodot attiecīgi `breatheAccumulatorFrameLength` vai `muscleAccumulatorFrameLength`.

Klase "BreatheEffortAccumulator"

Klase realizē saskarni "EffortAccumulator" , reprezentē sensoru datu uzkrājēju elpošanas treniņa soļiem un realizē sistēmas funkcionālo prasību par elpošanas sensora datu apstrādi (lietotājstāsts *S4_PATIENT_EFFORT_ACCUMULATOR*).

Klasei ir trīs lauki:

- `frameLength` ar tipu *int*, kas reprezentē kārtējā sensoru mērījumu "ietvara" garumu,
- `reduceStrategy` ar tipu "ListReduceStrategy", kas uzreiz tiek inicializēts, veidojot jaunu "ListAverageStrategyImpl"

klases objektu, un kas reprezentē saraksta "samazināšanas" stratēģiju jeb formulu, pēc kuras sistēmai jāapstrādā dati,

- `frame` ar tipu *List<Double>*, kas reprezentē kārtēju sensoru mērījumu "ietvaru" un uzreiz tiek inicializēts.

Klases metode `accumulate(double)` saņem kārtējo pacienta piepūles skaitlisku vērtību no sensora, pievieno to `frame` sarakstam un tad izsauc klases "ListAverageStrategyImpl" metodi `accumulate(Collection<Double>)`, kā parametru padodot `frame`. Ja `frame` izmērs ir sasniedzis `frameLength`, no `frame` tiek noņemts elements saraksta 0.pozīcijā, tādejādi nodrošinot *first-in-last-out* mehānismu saraksta elementu pārvaldei. Metode atgriež skaitli ar tipu *double*, kas ir ar stratēģijas formulu apstrādāts `frame`.

Klase "MuscleEffortAccumulator"

Klase realizē saskarni "EffortAccumulator", reprezentē sensoru datu uzkrājēju kustību treniņa soļiem un realizē sistēmas funkcionālās prasības par kustību sensora datu apstrādi (lietotājstāsts *S4_PATIENT_EFFORT_ACCUMULATOR*).

Klasei ir tādi paši lauki, kā klasei "BreatheEffortAccumulator", bet, inicializējot saraksta samazināšanas stratēģiju, tiek veidota jauna klases "RMSReduceStrategy" instance. Klases metodes `accumulate(double)` atbilst klases "BreatheEffortAccumulator" metodes realizācijai.

Pakotne "multicast.patientactivity.effort.accumulator.step"

Saskarne "StepAggregatedAccumulator"

Saskarne apraksta metodi, kuru jārealizē, implementējot šādu datu uzkrājēju – metode `accumulate(PluxPatientEffort)`.

Klase "StepAggregatedAccumulatorImpl"

Klase implementē saskarni "StepAggregatedAccumulator" un reprezentē augstākā līmeņa sensoru datu uzkrājēju, kas saņem pacienta piepūli klases "PluxPatientEffort" objekta veidā un novirza tās vērtību uz pareizu zemāka līmeņa datu uzkrājēju "BreatheEffortAccumulator" vai "MuscleEffortAccumulator".

Klasei ir divi lauki – atslēgu-vērtību saraksts `accumulators` ar tipu *Map<PatientEffortType, EffortAccumulator>*, kas glabā piepūles veidam atbilstošus datu uzkrājējus, un `lastStepId` ar tipu *int*, kura sākotnēja vērtība ir -1.

Klases konstruktors kā parametru sagaida klases “EffortAccumulatorFactory” objektu. Konstruktorā, sarakstā `accumulators` kā atslēgas tiek ielikti piepūles tipi *BREATHE* un *MUSCLE*. Tām atbilstošās vērtības jeb datu uzkrājēji tiek veidoti, izsaucot klases “EffortAccumulatorFactory” objektam metodi `create(PatientEffortType)`.

Klases metodē `accumulate(PluxPatientEffort)` vispirms no `accumulators` saraksta tiek paņemts atbilstošs datu uzkrājējs. Tiek pārbaudīts, vai tekošais solis ir vienāds ar `lastStepId`; ja nav, tad datu uzkrājējam tiek izsaukta metode `reset()`. Lauka `lastStepId` vērtība tiek nomainīta uz tekošā soļa identifikatoru. Datu uzkrājējam tiek izsaukta metode `accumulate(double)`, kā parametru padodot `currentEffort.getSensorValue()`, kur `currentEffort` ir metodes parametrā saņemtais “PluxPatientEffort” objekts. Beigās metode atgriež jaunu klases “AccumulatedPatientEffortImpl” objektu, kā konstruktora parametrus padodot `lastStepId`, `accumulated` (datu uzkrājēja metodes atgrieztā vērtība), `currentEffort.getType()`.

Pakotne “patientactivity.tcp”

Pakotnes struktūra ir parādīta klašu diagrammā zemāk (sk. *Att.3.14.*).

Klase “PatientActivityStateManager”

Klase tiek apzīmēta ar Spring anotāciju *@Bean* klasē-konfigurācijā “BeansConfig” un tās instance tiek veidota sistēmas palaišanas brīdī. Klases nolūks ir pārvaldīt pacienta fiziskās aktivitātes stāvokļus. Klases lauki ir `lastPluxEffort` ar tipu “PluxPatientEffort” un `lastAccumulatedEffort` ar tipu “AccumulatedPatientEffort”. Abi lauki tiek uzreiz inicializēti, veidojot

jaunu klases “NullPatientEffort” instanci. Pārējie divi lauki, kas arī uzreiz tiek inicializēti, ir ar tipu *Object* – `pluxEffortLock` un `accumulatedEffortLock`.

Metode `getLastPluxEffort()` satur koda bloku ar Java atslēgas vārdu *synchronized*, kas nodrošina, ka tikai viens pavediens var izpildīties iekš koda bloka, kas sinhronizēts ar monitora objektu `pluxEffortLock`. Metode atgriež `lastPluxEffort`.

Att.3.14. **Moduļa “Multicast” pakotnes “patientactivity.tcp” struktūra**

Sinhronizēta (ar atslēgas vārdu *synchronized*) metode `setLastPluxEffort(PluxPatientEffort)` satur koda bloku ar atslēgas vārdu *synchronized* ar monitora objektu `pluxEffortLock`. Metode piešķir laukam `lastPluxEffort` parametrā saņemto piepūles vērtību.

Metode `getLastAccumulatedEffort()` satur koda bloku ar atslēgas vārdu *synchronized* ar monitora objektu `accumulatedEffortLock`. Metode atgriež `lastAccumulatedEffort`.

`setLastAccumulaedEffort(AccumulatedPatientEffort)` satur koda bloku ar atslēgas vārdu *synchronized* ar monitora objektu `accumulatedEffortLock`. Metode piešķir laukam `lastAccumulatedEffort` parametrā saņemto apstrādātu piepūles vērtību.

Klase “PatientActivityAccumulator”

Klases nolūks ir uzkrāt sensoru datus un izsaukt augstākā līmeņa datu uzkrājēju. Tā tiek apzīmēta ar Spring anotāciju *@Bean* moduļa “Config” klasē-konfigurācijā “BeansConfig” un tās instance tiek veidota sistēmas palaišanas brīdī.

Klases konstruktors kā parametrus saņem klases "PatientActivityStateManager" instanci, datu tipa *int* vērtības (tiek saņemtas no klases-konfigurācijas "ARehabConfig") `accumulationTickDurationMS`, `muscleAccumulatorFrameLength`, `breahteAccumulatorFrameLength`. Konstruktorā tiek veidota klases "EffortAccumulatorFactoryImpl" instanci, kā parametrus padodot `muscleAccumulatorFrameLength` un `breahteAccumulatorFrameLength`. Tiek veidota arī jauns klases "StepAggregatedAccumulatorImpl" objekts, kā konstruktora parametru padodot izveidoto "EffortAccumulatorFactoryImpl" instanci.

Klases metode `start()` tiek izsaukta no klases "ARehabPortable". Metodē tiek veidots pavediens, kas `activityStateManager.getLastPluxEffort()` vērtību padod kā parametru klases "StepAggregatedAccumulatorImpl" metodei `accumulate(PluxPatientEffort)`, kuras atgriezto vērtību, savukārt, piešķir vērtībai `activityStateManager.setLastAccumulatedEffort(AccumulatedPatientEffort)`. Pavediens taisa pauzi, kuras ilgums ir vienāds ar `accumulationTickDurationMS` vērtību.

Klase "PatientActivityPluxConnector"

Klase tiek apzīmēta ar Spring anotāciju *@Bean* klasē-konfigurācijā "BeansConfig" un tās instance tiek veidota sistēmas palaišanas brīdī. Klases nolūks ir darboties ar klases "WorkoutManager" (modulis "Workout") objektu un ieslēgt Plux adapteri, lai uzsāktu pieslēgšanu sensoru datu plūsmai un datu iegūšanu. Klase kalpo par svarīgu vietu sistēmā, kurā "saiet" kopā vairāki moduļi – "Multicast", "Workout" un "Adapter".

Klases metode `start()` tiek izsaukta no klases "ARehabPortable". Uzdevumi, kurus pilda metode, ir:

- Ar anonīmu klasi realizē saskarni "PluxSensorDataCallback" (modulis "Adapter") – realizācijā tās metode `onSensorData(PluxSensorData)` vispirms klases "WorkoutManager" objektam pieprasa tekošā soļa numuru. Tad, ja solim ir norādīts piepūles veids, tiek izsaukta metode `patientEffortCalculator.calculate(Step, PluxSensorData)`, kā parametrus padod soli un parametrā saņemto klases "PluxSensorData" objektu. Tad pacienta aktivitātes pārvaldnieka `activityStateManager`

laukam `lastPluxEffort` tiek piešķirta ar metodi `calculate(Step, PluxSensorData)` izrēķinātā sensora kanāla relatīva vērtība.

- Ar anonīmu klasi realizē saskarni "PluxConnectionFailedCallback" (modulis "Adapter") - realizācijā tās metodē `onFail()` pacienta aktivitātes pārvaldniekam `activityStateManager` laukam `lastPluxEffort` piešķir jaunu klases "NullPatientEffort" objekta vērtību.
- Izveido klases "PluxAdapter" (modulis "Adapter") instanci un izsauc tai metodi `start()`, lai sistēma pieslēgtos sensoru datu plūsmai.

Klase "PatientActivityTcpMulticast"

Klase tiek apzīmēta ar Spring anotāciju *@Bean* klasē-konfigurācijā "BeansConfig" un tās instance tiek veidota sistēmas palaišanas brīdī. Klases mērķis ir uzsākt un pārvaldīt apstrādāto sensoru datu un informācijas par tekošo treniņa soli pārraidi (lietotājstāsts *S4_MULTICAST*).

Klases metode `start()` tiek izsaukta no klases "ARehabPortable". Šī metode atver Java servera ligzdu *(ServerSocket)* `multicastSocket` uz porta, kas ir norādīts sistēmas lietotāja pārvaldītajā konfigurācijas datnē laukā `multicastPort` (datnes struktūra ir aprakstīta pie moduļa "Config" apraksta).

Tad metode veido un palaiž jaunu pavedienu, kurā sazinās ar lauku `activityStateManager` (klases "PatientActivityStateManager" objektu), lai nolasītu tā lauka `lastAccumulatedEffort` vērtību. Tad pavediens veido jaunu ziņojumu jeb klases "PatientActivityMessageImpl" instanci, kā parametrus padodot `lastAccumulatedEffort.getStepId()`, `lastAccumulatedEffort.getAccumulatedValue()`. Tad laukam `messageBroker` (klases "PatientActivityMessageBroker" objekts) tiek izsaukta metode `publish(M)` ar tikko izveidoto ziņojumu kā parametru. Pavediens taisa pauzi, kuras ilgums ir vienāds ar konfigurācijas klases "ARehabPortable" izrēķināto vērtību `multicastTickDurationMS`.

Metode veido un palaiž vēl vienu pavedienu, kas izveido jaunas klases "PatientActivityTcpClientHandler" instances ar konstruktora parametriem `multicastSocket` un `messageBrocker` metodē `run()`.

Klase "PatientActivityTcpClientHandler"

Klase ir atbildīga par jaunu datu pārraides klientu fizisku pieslēgšanu. Klases pavediena metodē `run()` ir cilpa, kurā tiek veidots kārtējā klienta identifikators *int* `clientId`. Tiek inicializēti mainīgie `socket` ar tipu *Socket*,

`out` ar tipu *PrintWriter* un `in` ar tipu *BufferedReader*. Tiek mēģināts mainīgajam `socket` piešķirt vērtību `multicastSocket.accept()` un piešķirt vērtības mainīgajiem `out` un `in`, izmantojot `socket.getOutputStream()` un `new InputStreamReader(socket.getInputStream())` attiecīgi.

Tad klases laukam `messageBroker` tiek izsaukta metode `registerConsumer(C)`, kā parametru padodot jaunu klases "PatientActivityConsumerSocket" objektu ar konstruktora parametriem `clientId` un `out`. Tiek izveidots jauns pavediens, kas izpilda iekšējās (statiskas) klases "OnCloseConnectionHandler" pavediena metodi `run()`.

Iekšējai klasei "OnCloseConnectionHandler" ir metode `disconnect()`, kas izsauc `messageBroker.unregisterConsumer(clientId)`. Metodē `run()` tiek mēģināts nolasīt kaut ko no `in` un, saņemot ievades-izvades kļūdu *IOException*, tiek izsaukta metode `disconnect()`, jo klients ir atslēdzies.

Pakotne "sensor"

Pakotne satur saskarni un klases, kuras ir saistītas ar sensoru konfigurāciju jeb sensoru iestatījumiem, kurus sistēmas lietotājs norāda tīmekļa lappusē – katra piepūles veida sensora kanāla numurs, minimālā un maksimālā vērtība mērījumiem (lietotājstāsts *S3_SENSOR_CONFIG_WEB*).

Saskarne "SensorConfig"

Saskarne apraksta uzvedību, kuru jārealizē sensora konfigurācijas klasei, un definē metodes, kuras jārealizē, implementējot šo saskarni – metodes `getChannelIndex(PatientEffortType)` un `getBounds(PatientEffortType)`.

Klase "Bounds"

Klase reprezentē katra piepūles veida sensora minimālās un maksimālās vērtību pāri. Klasei ir divi lauki - `lower` ar tipu *Double* un `upper` ar tipu *Double*.

Klase "DefaultConfigImpl"

Klase realizē saskarni "SensorConfig" un piedāvā noklusēto saskarnes realizāciju. Metode `getChannelIndex(PatientEffortType)` atgriež vērtību *null*, savukārt metode `getBounds(PatientEffortType)` atgriež jaunu klases "Bounds" instanci ar *null* vērtībām abos tās laukos. Tādejādi, klase apraksta saskarnes "SensorConfig" *null*-objektu realizāciju ar neitrālu uzvedību.

Klase "PluxDevice"

Klase realizē moduļa "Adapter" saskarni "BitalinoDevice", piedāvājot saskarnes realizāciju tieši Plux sensoriem.

Klase "PluxDeviceRegistry"

Klase realizē ģenērisko saskarni "BitalinoDeviceRegistry" un reprezentē Plux sensoru reģistru, kas veidojas, balstoties uz lietotāja pārvaldīto konfigurācijas datni *"application.yml"* (lauks `devices` klasē-konfigurācijā "ARehabConfig"). Klase tiek apzīmēta ar Spring anotāciju *@Bean* klasē-konfigurācijā "BeansConfig. Klasei ir viens lauks `devices` ar tipu *Map*<*Integer*, `PluxDevice`>. Metode `get(int)` atgriež `devices` elementu pēc sensora identifikatora; metode `getAll()` atgriež visas `devices` vērtības. Metode `findByMacAddress(String)` paņem no `devices` vērtībām tādu, kuras MAC adrese ir vienāda ar parametrā norādīto.

Klase "SensorConfigImpl"

Klase realizē saskarni "SensorConfig". Klasei ir divi lauki – `channels` ar tipu *Map*<`PatientEffortType`, *Integer*> un `bounds` ar tipu <`PatientEffortType`, `Bounds`>. Saskarnes metodes `getChannelIndex(PatientEffortType)` un `getBounds(PatientEffortType)` atgriež attiecīgā saraksta vērtību pēc `PatientEffortType`. Klases pievienotā statiska metode `of(IncomingSensorConfigUpdateMessage)` saņem kā parametru DTO ziņojumu par sensoru konfigurācijas atjaunošanu. Metode atjauno abus sarakstus, ievietojot jaunas vērtības pēc piepūles veidiem.

Klase "SensorConfigManager"

Klases nolūks ir pārvaldīt sistēmā pastāvošo sensoru konfigurāciju un izmantot sensoru konfigurācijā norādītās vērtības, lai no sensora datiem (kārtējās klases "PluxSensorData" instances) paņemtu datus no konfigurācijā norādītā sensora kanāla un izmantotu minimālo un maksimālo vērtības relatīvās vērtības aprēķināšanai (lietotājstāsts *S3_SENSOR_CONFIG_WEB*). Šī klase ir svarīgs sistēmas komponents un ir apzīmēta ar Spring anotāciju *@Bean* klasē-konfigurācijā "BeansConfig".

Spring inicializē klases objektu, un tās laukam `config`, kas ir saskarnes "SensorConfig" instance, tiek uzreiz piesaistīta jauna klases "DefaultSensorConfig" instance. Klases metodes `getConfig()` un `setConfig(SensorConfig)` attiecīgi atgriež vai atjauno šī lauka vērtību.

Klases metodes `inboundByType(PatientEffortType, PluxSensorData)` nolūks ir no klases "PluxSensorData" objekta paņemt mērījumu no pareizā sensora kanāla (pēc konfigurācijā norādītā) un izrēķināt mērījuma relatīvo vērtību, izmantojot konfigurācijā norādītās minimālo un maksimālo vērtības. Lai iegūtu sensora kanāla

vērtību, tiek izsaukta klases "PluxSensorData" metode getChannelValue(int) ar pareizu kanāla numuru.

Pakotne "utils"

Šajā pakotnē atrodas apkalpojoša veida saskarne un klases, kas piedāvā stratēģijas kāda uzdevuma risināšanai.

Pakotne "utils.reduce"

Pakotnē atrodas saskarne un klases, kuru nolūks ir apstrādāt sensoru datus pēc sistēmas prasībās definētajām formulām. Abas formulas ir saistītas ar saraksta vērtību uzkrāšanu un saraksta "samazināšanu" līdz vienam skaitlim.

Saskarne "ListReduceStrategy"

Saskarne aprasta metodi, kuru jārealizē, implementējot jebkuru saraksta samazināšanas stratēģiju – metode accumulate(Collection<Double>). Metodei kaut kādā veidā "jāuzkrāj" skaitļu sarakstu un jāatgriež skaitli ar datu tipu *double*.

Klase "ListAverageStrategyImpl"

Klase realizē saskarni "ListReduceStrategy". Metode accumulate(Collection<Double>) atgriež saraksta vidējo vērtību. Šo klasi izmanto klase *"BreatheEffortAccumulator"*, tādējādi realizējot sistēmas funkcionālās prasības par elpošanas sensora datu apstrādi (lietotājstāsts *S4_PATIENT_EFFORT_ACCUMULATOR*).

Klase "RMSReduceStrategy"

Klase realizē saskarni "ListReduceStrategy". Metode accumulate(Collection<Double>) aprēķina saraksta elementu kvadrātu summu un atgriež kvadrātsakni no tās dalījuma ar saraksta garumu, šādā veidā realizējot RMS formulu. Šo klasi izmanto klase *"MuscleEffortAccumulator"*, tādējādi realizējot sistēmas funkcionālās prasības par kustību sensora datu apstrādi (lietotājstāsts *S4_PATIENT_EFFORT_ACCUMULATOR*).

Saskarne "BitalinoDeviceRegistry"

Saskarne piedāvā metodes, kuras jārealizē, implementējot sensoru reģistru kādam konkrētam sensoru veidam. Šīs metodes ir T get(int deviceId), Collection<T> getAll() un Optional<T> findByMacAddress(String macAddress).

Saskarne "MulticastMessage"

Saskarne apraksta metodi, kuru jārealizē jebkurai datu pārraides ziņojuma realizācijai – metode `getAsString()`.

Saskarne “MulticastMessageBroker”

Saskarne ir ģenēriskā, tai ir divi tipu parametri – `<M extends MulticastMessage>` un `<C extends MulticastMessageConsumer<M>>`. Saskarne apraksta uzvedību jeb metodes, kuras jārealizē jebkurā ziņojumu starpnieka implementācijā – tās ir metodes `registerConsumer(C)`, `unregisterConsumer(int)` (kā parametrs tiek padots klienta identifikators) un `publish(M)`.

Saskarne “MulticastMessageConsumer”

Saskarne ir ģenēriskā ar vienu tipa parametru `M`. Saskarne apraksta metodes, kuras jārealizē datu pārraides ziņojumu klienta implementācijā – metodes `getId()` un `onMessage(M)`.

Klase “SimpleMulticastMessageBroker”

Šī abstrakta ģenēriskā klase ar tipu parametriem `<M extends MulticastMessage>` un `<C extends MulticastMessageConsumer<M>>` realizē ģenērisko saskarni “MulticastMessageBroker”`<M, C>`.

Klases lauks `messageRelay` ar datu tipu *ConcurrentHashMap* ir atslēgu-vērtību saraksts, kur atslēgas ir *Integer* tipa klienta identifikators un vērtība ir bloķējošā rinda ar datu tipu `BlockingQueue<M>`.

Metodē `registerConsumer(C)` vispirms tiek izveidota jauna rinda `messageQueue` ar datu tipu *LinkedBlockingQueue*. Šī rinda tiek piesaistīta klienta identifikatoram, un šāds pāris tiek ielikts kopīgajā atslēgu-vērtību sarakstā `messageRelay`. Tiek izveidots jauns pavediens, kurš tiek apturēts, ja `messageRelay` vairs nesatur klienta identifikatoru kā vienu no atslēgām. Pavedienā tiek mēģināts nolasīt ziņojumu no rindas `messageQueue` ar Java klases *LinkedBlockingQueue* metodes `take()` palīdzību, kas iegūst un noņem šīs rindas galveni. Tad klientam tiek izsaukta metode `onMessage(M)`.

Metodē `unregisterConsumer(int)` no kopīgā atslēgu-vērtību saraksta `messageRelay` tiek izsaukta metode `remove(Object)`, kā parametru padodot klienta identifikatoru.

Metodē `publish(M)` katrai kopīgā atslēgu-vērtību saraksta `messageRelay` vērtībai – tai ir, katrai rindai ar datu tipu *BlockingQueue<M>* - tiek izsaukta Java klases *BlockingQueue* metode `offer(E)`, kā parametru padodot ziņojumu, ko metode saņēma. Metode `offer(E)` ievieto norādīto ziņojumu šajā rindā.

3.4.3.4. Lietotnes konfigurācijas: modulis "Config"

"Config" ir apakšsistēmas "Java-Application" modulis, kas pārvalda Java lietotnes konfigurācijas. Modulis piedāvā klases un realizē metodes, kas strādā ar galvenajiem Java lietotnes komponentiem. Šī moduļa pienākumos ietilpst arī darbības ar treniņa konfigurācijas datni *"application.yml"*.

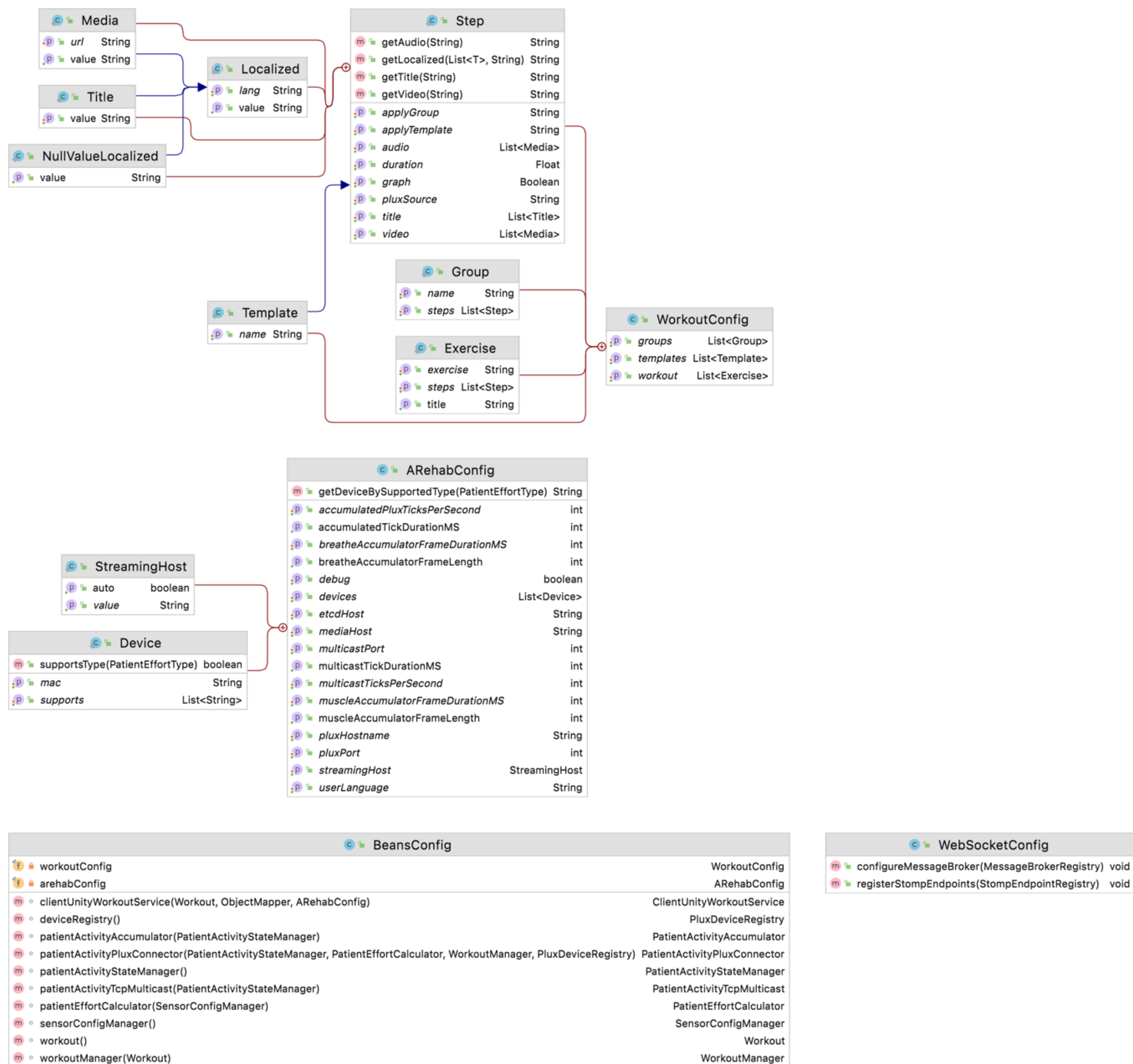
Moduļa nolūks

Moduļa mērķis ir uzbūvēt Java lietotnes "mugurkaulu" – visus komponentus, kuri ir nepieciešami, lai, palaižot sistēmu, tā varētu pildīt savas funkcijas.

Visas šī moduļa klases ir apzīmētas ar Spring anotāciju *@Configuration*. Divas šī moduļa klases ir apzīmēti ar SpringBoot anotāciju *@ConfigurationProperties*, jo tās strādā ar YAML konfigurācijas datni *"application.yml"*. Šī datne kalpo par konfigurācijas datni, kuru lietotājs var pārvaldīt pats un no kuras arī tiek ģenerēts treniņš (lietotājstāsts *S2_WORKOUT_CONFIG*).

Konfigurācijas datne ir sadalīta divās daļās. Pirmā konfigurācijas daļa atbild par sistēmas darbam nepieciešamajiem parametriem un sistēmas palaišanas iestatījumiem. Šo daļu apraksta moduļa "Config" klase "ARehabConfig". Otro konfigurācijas daļu, kas ir treniņa konfigurācija, apraksta klase "WorkoutConfig".

Klašu diagrammā zemāk ir parādīta moduļa struktūra (sk. *Att.3.15.*).



Att.3.15. **Moduļa "Config" struktūra**

Klase "BeansConfig"

Spring DI mehānisms atbild par Spring lietotnes komponenšu izveidi un iespraušanu citos komponentos, un klase "BeansConfig" ir apakšsistēmas "Java Application" galvenais *Bean* komponenšu avots un pārvaldnieks.

Klases "BeansConfig" metodes apraksta, kuru klašu instances ir nepieciešamas apakšsistēmas "Java-Application" darbam un kurās vietās tās ir vajadzīgas. Visas šīs klases metodes ir apzīmēti ar Spring anotāciju @*Bean*.

Klasei ir divi lauki – `workoutConfig` un `arehabConfig`, kuri ir atbilstošo klašu-konfigurāciju instances.

Visas klases "BeansConfig" metodes ir apkopotas tabulā zemāk (sk. *Tab.3.2.*).

Tab.3.2.: Klases "BeanConfig" *@Bean* metodes

***@Bean* metode**	**Apstrāde un atgriežamā vērtība**
`Workout workout()`	Izsauc klases "Workout" metodi `parse`, kā parametru padodot `workoutConfig`, un atgriež metodes atgriežamo vērtību – klases instanci
`WorkoutManager workoutManager(Workout workout)`	Atgriež klases instanci
`ClientUnityWorkoutService clientUnityWorkoutService(Workout workout, ObjectMapper objectMapper, ARehabConfig aRehabConfig) )`	Atgriež klases instanci
`PatientActivityStateManager patientActivityStateManager()`	Atgriež klases instanci
`PatientActivityPluxConnector patientActivityPluxConnector(PatientActivityStateManager activityStateManager, PatientEffortCalculator patientEffortCalculator, WorkoutManager workoutManager,PluxDeviceRegistry deviceRegistry)`	Atgriež klases instanci, kā vienu no parametriem klases konstruktoram padod arī `arehabConfig` laukus `pluxHostname` un `pluxPort`
`PatientActivityAccumulator patientActivityAccumulator(PatientActivityStateManager activityStateManager)`	Atgriež klases instanci, kā parametrus konstruktoram vēl padod `arehabConfig` laukus `accumulatedTickDurationMS`, `muscleAccumulatorFrameLength`, `breatheAccumulatorFrameLength`
`PatientActivityTcpMulticast patientActivityTcpMulticast(PatientActivityStateManager activityStateManager)`	Atgriež klases instanci, kā parametrus konstruktoram vēl padod `arehabConfig` laukus `multicastPort`, `multicastTickDurationMS`
`PatientEffortCalculator patientEffortCalculator(SensorConfigManager sensorConfigManager)`	Atgriež klases instanci
`PluxDeviceRegistry deviceRegistry()`	No `arehabConfig` paņem sarakstu ar ierīcēm (izsaucot klases "ArehabConfig" metodi `getDeviceSupportedType(PatientEffortType)`), ieliek tos sarakstā pēc to piepūles veida (*breathe* vai *muscle*), padod to kā parametru konstruktoram un atgriež klases instanci
`SensorConfigManager sensorConfigManager()`	Atgriež klases instanci

Klase "ARehabConfig"

Klase "ARehabConfig" atbild par konfigurācijas datnes *"application.yml"* pirmo daļu, kas satur sistēmas palaišanas iestatījumus. Klases lauki pilnībā apraksta visus konfigurācijas

datnes iestatījumu laukus, kas YAML datnē seko pēc uzraksta “*arehab*” – YAML datnes pirmās daļas fragmenta piemērs sekos zemāk (sk. *Att.3.16.*).

```yaml
arehab:
  # plux access point
  pluxHostname: host.docker.internal
  pluxPort: 5555
  # ARehab port for multicast
  multicastPort: 8888
  # how many ticks ARehab will produce every second
  multicastTicksPerSecond: 500
  # how many Plux ticks per second will be accumulated
  accumulatedPluxTicksPerSecond: 500
  # how long is accumulation period of  the muscle exercises
  muscleAccumulatorFrameDurationMS: 1000
  # how long is accumulation period of  the breathe exercises
  breatheAccumulatorFrameDurationMS: 200
  # ARehab streaming host
  streamingHost: 192.168.1.1
  # etcd server
  etcdHost: "http://etcd:2379"
  # ARehab display language
  userLanguage: en
  # ARehab media server
  mediaHost: "http://localhost:8082"
  # debug mode
  debug: false
  # devices used for acquisition
  devices:
    - mac: '00:00:00:00:00:00'
      supports:
        - breathe
    - mac: '11:11:11:11:11:11'
      supports:
        - muscle
```

Att.3.16. **YAML konfigurācijas datnes “application.yml” pirmās daļas fragmenta piemērs**

Visi konfigurācijas datnes pirmās daļas parametri (un attiecīgi klases “ARehabConfig” lauki) ir apkopoti tabulā zemāk (sk. *Tab.3.3.*).

Tab.3.3. Konfigurācijas datnes “*application.yml*” pirmās daļas parametru un klases-konfigurācijas “ARehabConfig” lauku apraksti

Lauks	**Apraksts**
`pluxHostname`	Saimniekdatora, uz kura tiks palaista lietojumprogramma “OpenSignals” sensoru datu iegūšanai, adrese (“Java-Application” tiks palaista Docker konteinerī, tāpēc ir jāņem vērā, ka adresei jābūt redzamai no Docker konteinera – piemērā norādītā adrese `host.docker.internal` informē konteineri, ka jāvēršas pie saimniekdatora, uz kura ir palaists Docker)
`pluxPort`	Saimniekdatora, uz kura tiks palaista lietojumprogramma “OpenSignals”, ports sensoru datu iegūšanai (pēc oficiālas

	dokumentācijas "TCP/IP Module Guide" [5] tas ir ports 5555)
multicastPort	Ports, kuram jāpieslēdzas AR lietotnei, lai pieslēgtos apakšsistēmas "Java-Application" datu plūsmai
multicastTicksPerSecond	Cik datu vienumu sistēmai jāģenerē vienā sekundē
accumulatedPluxTicksPerSecond	Cik sensoru mērījumu / datu vienumu sistēmai jāsaņem vienā sekundē
muscleAccumulatorFrameDurationMS	Ilgums milisekundēs, kas parāda, cik ilgi jāveic viens mērījums kustību treniņa soļos no kustību sensoriem ("ARehabConfig" klases metode `getMuscleAccumulatorFrameLength()`)
breatheAccumulatorFrameDurationMS	Ilgums milisekundēs, kas parāda, cik ilgi jāveic viens mērījums elpošanas treniņa soļos no elpošanas sensoriem ("ARehabConfig" klases metode `getBreatheAccumulatorFrameLength()`)
streamingHost	IP adrese, uz kuru jāiet AR lietotnei, lai pieslēgtos apakšsistēmas "Java-Application" datu plūsmai
etcdHost	Etcd datu glabātuves saimniekdatora adrese ("Java-Application" un "Etcd" apakšsistēmu konteineri ir apvienoti vienā Docker tīklā, tāpēc pie konteinera var vērsties pēc tā nosaukuma)
userLanguage	Valoda, kurā jāpalaiž treniņš
mediaHost	Video, audio un attēlu datņu saimniekdatora adrese un ports; ģenerējot treniņa konfigurāciju AR lietotnei, šī adrese tiek apvienota ar treniņā norādītiem ceļiem veidā "*/media/filename*" līdz datnēm.
debug	Atkļūdošanas parametrs izstrādātājiem
devices	Sensoru (ierīču) saraksts (lietotājstāsts *S2_DEVICES_CONFIG*): katras ierīces MAC adrese un piepūles veids, kuram ierīce atbilst (*breathe* vai *muscle* – pasaka, vai tas ir elpošanas vai kustību sensors).

Klase "WorkoutConfig"

Moduļa "Config" klase "WorkoutConfig" atbild par treniņa konfigurāciju – konfigurācijas datnes otro daļu, kurā tiek aprakstīts treniņš ar treniņa soļiem, kas satur dažāda veida instrukcijas.

Pēc sistēmas funkcionālajām prasībām, treniņa soļus jāspēj apvienot grupās un taisīt soļu šablonus (sprinta uzdevums *S2_STEPS_TEMPLATES_GROUPS*). Realizētās treniņa konfigurācijas, kas atbilst šai prasībai, datnes fragments sekos zemāk (sk. *Att.3.17.*).

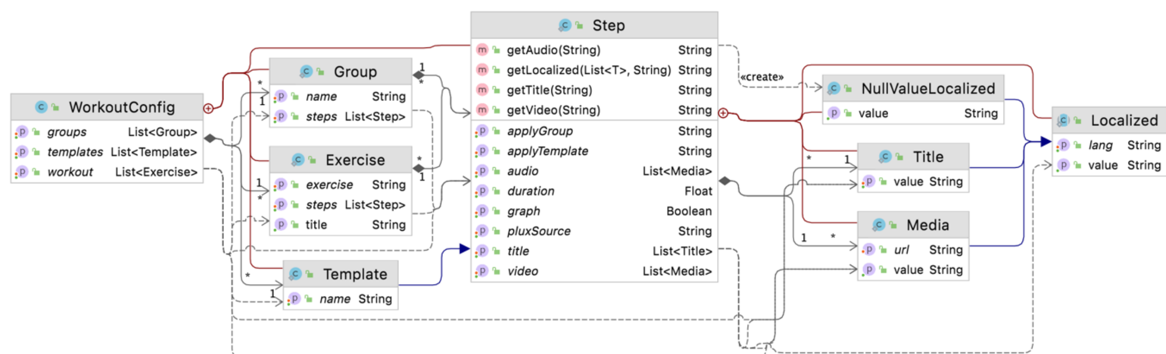
```yaml
arehab-workout:
  templates:
    - name: countdown-tick-breathe
      audio:
        - lang: 'en'
          url: /media/countdown-tick.mp3
      duration: 1.5
      graph: true
      pluxSource: breathe
   groups:
    - name: countdown-321-breathe
      steps:
        - title:
            - lang: 'en'
              value: '3'
          applyTemplate: countdown-tick-breathe
        - title:
            - lang: 'en'
              value: '2'
          applyTemplate: countdown-tick-breathe
        - title:
            - lang: 'en'
              value: '1'
          applyTemplate: countdown-tick-breathe
    workout:
    - exercise: Breathing training
      steps:
   - title:
            - lang: 'en'
              value: 'Task: Inhale with abdomen, exhale through pursed lips'
            - lang: 'lv'
              value: 'Uzdevums: Ieelpo ar vēderu, izelpo, savelkot lūpas tūtiņā'
          video:
             - lang: 'en'
              url: /media/3-breathing-training/instruction_EN.mp4
             - lang: 'lv'
     url: /media/3-breathing-training/instruction _LV.mp4
          duration: 27
        - applyTemplate: countdown-321-breathe
```

Att.3.17. **Treniņa konfigurācijas fragmenta piemērs no datnes "application.yml"**

Vispirms treniņa konfigurācijā ir uzskaitīti soļu šabloni – *templates*. Lai izmantotu tos turpmāk treniņa konfigurācijā (sākas ar "*workout*"), jauna soļa vietā lietotājam ir jāraksta `applyTemplate: templateName`. Pēc soļu šabloniem tiek uzskaitīti soļu grupas – lai tās izmantotu treniņā, vēlamajā vietā lietotājam ir jāuzraksta `applyGroup: groupName` – tajā vietā tiks ievietots nevis viens kārtējais solis, bet soļu grupa.

Lai nodrošinātu, ka sistēma izpilda prasību par sistēmas lokalizāciju (lietotājstāsts *S3_WORKOUT_LOCALIZATION*), treniņa konfigurācijas lauki `title`, `video` un `audio` ir atsevišķas klases, katrs no kurām manto no "WorkoutConfig" klases iekšējās abstraktās klases `Localized`, kurai ir divi lauki – `value` un `lang`.

Iekšējas klases "Step" (sk. *Att.3.18.*) lauki `title`, `video` un `audio` ir saraksti no klases `Localized` objektiem, nevis simbolu virknes, jo vērtības jāglabā dažādās valodās.



Att.3.18. **Klases "WorkoutConfig" un to iekšējo klašu struktūra**

Iekšējas klases "Step" metode `getLocalized(List<T> values, String lang)` kā parametru saņem lietotāja izvēlēto sistēmas valodu un kādu no datu tipiem, kas manto no `Localized` – tas ir, `title`, `video` vai `audio`. Metode mēģina iegūt vērtību vēlamajā valodā, bet, ja tās treniņa konfigurācijā nav, metode ņem vērtību angļu valodā – tātad, ja solim ir `title`, `video` vai `audio`, angļu valodas vērtībai ir jābūt obligāti norādītai. Tika nolemts paņemt vienu – šajā gadījumā angļu – valodu kā noklausīto, tāpēc, ka var gadīties, ka solim piesaistītās instrukcijas būs vienādas visās sistēmas valodās.

Klase "WebSocketConfig"

Moduļa "Config" klase "WebSocketConfig" implementē Spring saskarni *WebSocketMessageBrokerConfigurer*, kas, savukārt, definē metodes ziņojumu apstrādes konfigurēšanai no *WebSocket* klientiem, izmantojot vienkāršus ziņojumapmaiņas protokolus, tādus kā STOMP.

Klase "WebSocketConfig" ir apzīmētu ar Spring anotāciju *@Configuration*, lai norādītu, ka tā ir Spring konfigurācijas klase, kā arī ir apzīmēta ar anotāciju *@EnableWebSocketMessageBroker*. Ziņojumu starpnieks nodrošina *WebSocket* ziņojumu virzīšanu un apstrādi.

Klases metode `configureMessageBroker()` implementē saskarnes *WebSocketMessageBrokerConfigurer* metodi, lai konfigurētu ziņojumu starpnieku. Metodes realizācijā vispirms tiek izsaukta metode `enableSimpleBroker()`, lai ļautu vienkāršam uz atmiņas balstītām ziņojumu starpniekam sūtīt ziņojumus klientam uz galamērķiem, kuriem ir prefikss */topic*. Kā arī, metode apzīmē */exchange* prefiksu ziņojumiem, kas ir saistīti ar metodēm, kas anotētas ar Spring anotāciju *@MessageMapping* – šie ziņojumi atrodas apakšsistēmas "Java-Application" moduļa "Web" klasē "WebSocketBroker".

3.4.3.5. Tīmekļa lappuse: modulis "Web"

Balstoties uz sistēmas "ARehab Portable" funkcionālajām prasībām, sistēmas lietotājam ir jānodrošina iespēja pēc sistēmas palaišanas pārlūkprogrammā redzēt visus treniņa soļus, uzlikt sensoru iestatījumus un palaist treniņu (lietotājstāsts *S3_ WEB_WORKOUT*).

Modulis "Web" ir apakšsistēmas "Java Application" modulis, kura nolūks ir pārvaldīt tīmekļa lappusi un nodrošināt tās funkcionalitāti atbilstoši sistēmas prasībām. Modulis ļauj lietotājam palaist treniņu un pārvaldīt tās gaitu.

Moduļa nolūks

Moduļa nolūks ir nodrošināt pieprasīto tīmekļa lapas funkcionalitāti – treniņa soļu attēlošanu, iespēju norādīt sensoru iestatījumus un robežvērtības piepūles veidiem, treniņa palaišanu, soļu progresa indikatoru dinamisku attēlošanu pārlūkprogrammā.

Pēc sistēmas prasībām, tīmekļa lapā ir jānodrošina iespēja:

- redzēt visus treniņa soļus, katram solim – tā tekstu, ilgumu (vai informāciju par to, ka solim ir manuāla pāreja), piepūles veidu, ja treniņa konfigurācijas datnē tā bija noradīta šim treniņa solim (lietotājstāsts *S3_ WEB_WORKOUT)*
- katram piepūles veidam norādīt pareizo sensora kanāla numuru, kuru sistēmai jāizmanto datu iegūšanai no elpošanas vai kustību sensora (lietotājstāsts *S3_SENSOR_CONFIG_WEB*);
- katram piepūles veidam spēt norādīt robežvērtības (minimālo un maksimālo vērtību) relatīvās vērtības aprēķināšanai (lietotājstāsts *S3_SENSOR_CONFIG_WEB*);
- noklikšķināt uz pogas "*Start!*", lai palaistu treniņu - soļu automātiskā (pēc to ilgumiem) pārslēgšana (ja solim ir manuāla pāreja – pārslēgt var tikai lietotājs) un apstrādāto sensoru datu pārraide (lietotājstāsti *S3_ WEB_WORKOUT*, *S4_ MULTICAST*);
- brīvi pārslēgt soļus abos virzienos (lietotājstāsts *S3_ WEB_WORKOUT*);
- kārtējam ieslēgtam solim redzēt tā dinamisku progresa indikatoru procentos (lietotājstāsts *S3_ WEB_WORKOUT*).

Lai nodrošinātu treniņa palaišanu un tā gaitas kontroli caur tīmekļa lapu, saziņas realizācijā starp tīmekļa lapu un pārējo sistēmu tika izmantots *WebSocket* protokols, kas nodrošina divvirzienu interaktīvu saziņu. Dažādu ziņojumu pārsūtīšana notiek, izmantojot DTO objektus, un DTO modeļi tiek aprakstīti moduļa "Web" pakotnē "dto".

Pakotne "dto"

Katra pakotnes klase ir ienākošā vai izejošā (no sistēmas skatupunkta) ziņojuma objekta modelis, kas pārsūta kaut kādu informāciju par treniņa progresu vai sensoru iestatījumiem. Pakotnes struktūra ir parādīta klašu diagrammā zemāk (sk. *Att.3.19.*).

OutgoingSensorConfigMessage
equals(Object) boolean
hashCode() int
of(SensorConfig) OutgoingSensorConfigMessage
toString() String
breatheChannelIndex Integer
breatheLowerBound Double
breatheUpperBound Double
muscleChannelIndex Integer
muscleLowerBound Double
muscleUpperBound Double

IncomingSensorConfigUpdateMessage
getChannelByType(PatientEffortType) Integer
getLowerBoundByType(PatientEffortType) Double
getUpperBoundByType(PatientEffortType) Double
breatheChannelIndex Integer
breatheLowerBound Double
breatheUpperBound Double
muscleChannelIndex Integer
muscleLowerBound Double
muscleUpperBound Double

OutgoingStepSwitchMessage
currentStepId int
manual boolean
previousStepId int

IncomingStepSwitchMessage
stepId int

OutgoingStepProgressMessage
percentage int
stepId int

OutgoingWorkoutStateMessage
workoutInProgress boolean

IncomingWorkoutStateMessage
workoutInProgress boolean

Att.3.19. **Moduļa "Web" pakotnes "dto" struktūra**

Klase "IncomingSensorConfigUpdateMessage"

Modelis apraksta ienākošo ziņojumu par sensoru konfigurācijas atjaunošanu. Tāds ziņojums tiek sūtīts, kad sistēmas lietotājs ievada un apstiprina ievadītos sensoru iestatījumus. Abiem piepūles veidiem atbilst trīs lauki šajā modelī – sensora kanāla indekss, minimālā vērtība šī piepūles veida mērījumam un maksimālā vērtība (mērķis, ko pacientam jācenšas sasniegt) šī piepūles veida mērījumam.

Klase "IncomingStepSwitchMessage"

Modelis apraksta ienākošo ziņojumu par treniņa soļa pārslēgšanu. Tas ir signāls sistēmai, ka treniņā solis jāpārslēdz uz nākamo. Ziņojuma mērķis ir nodrošināt sistēmas lietotājam kontroli treniņa soļu pārslēgšanā jebkurā brīdī un virzienā. Kā arī, šī ziņojuma saņemšana ir vienīgais veids, kā sistēma var pārslēgt soli ar manuālu pāreju.

Klase "IncomingWorkoutStateMessage"

Modelis apraksta ienākošo ziņojumu par treniņa stāvokli. Šis ziņojums informē sistēmu par to, vai treniņš ir palaists.

Klase "OutgoingSensorConfigMessage"

Modelis apraksta izejošo ziņojumu par sensoru konfigurāciju. Šis ziņojums tiek pieprasīts no sistēmas, lai, pārlādējot tīmekļa lapu (ja sistēmas darbība netika apstādināta), tīmekļa lapas lauki (katram piepūles veidam – sensora kanāla numurs, minimālā un maksimālā vērtība) tiktu automātiski aizpildīti ar sistēmā esošām sensoru konfigurācijas vērtībām.

Klase "OutgoingStepProgressMessage"

Modelis apraksta izejošo ziņojumu par kārtējā treniņa soļa progresu. Šī modeļa nolūks ir nodrošināt dinamisku treniņa soļu progresa indikatoru.

Klase "OutgoingStepSwitchMessage"

Modelis apraksta izejošo ziņojumu par treniņa soļa pārslēgšanu. Sistēma nodrošina soļu pārslēgšanu pēc to ilgumiem un sūta šo ziņojumu, lai informētu tīmekļa lapu, kas attēlo treniņa gaitu, ka solis ir vizuāli jāpārslēdz uz nākamo.

Klase "OutgoingWorkoutStateMessage"

Modelis apraksta izejošo ziņojumu par treniņa stāvokli. Ziņojums informē tīmekļa lietotni, vai treniņš ir palaists. Piemēram, kad kārtējām solim nav nākamā, pēc tekošā soļa izpildes treniņš sistēmai ir jāaptur.

Pārējās moduļa "Web" klases, kas nav DTO modeļi, ir klases-kontrolleri "WebSocketBroker" un "ARehabController".

Klase "WebSocketBroker"

Klase ir apzīmēta ar Spring anotāciju *@Controller*. Šis kontrolleris ir ziņojumu starpnieks, kas nodrošina *WebSocket* ziņojumu virzīšanu un apstrādi.

Kontrollera metode `listen()` izsauc moduļa "Workout" klases "WorkoutManager" metodi `registerUpdateCallback` un kā parametru padod anonīmu klasi, kas implementē moduļa "Workout" saskarni "WorkoutStateUpdatedCallback":

- metodē `onWorkoutStarted()` uz galamērķi *"/topic/workout-progress"* tiek sūtīts ziņojums – klases "OutgoingWorkoutStateMessage" instance, parametrs `workoutInProgress` ir patiess;
- metodē `onWorkoutFinished()` uz galamērķi *"/topic/workout-progress"* tiek sūtīts ziņojums – klases "OutgoingWorkoutStateMessage" instance, parametrs `workoutInProgress` ir aplams;
- metodē `onStepSwitched(Step currentStep, Step previousStep)` uz galamērķi *"/topic/step-switch"* tiek sūtīts ziņojums – klases "OutgoingStepSwitchMessage" instance;
- metodē `onStepProgress(Step step, double progress)` uz galamērķi *"/topic/step-progress"* tiek sūtīts ziņojums – klases "OutgoingStepProgressMessage" instance ar tekošo soļa kārtas numuru un tā progresa indikatoru procentos.

Pārējās kontrollera metodes saņem ienākošos ziņojumus un apstrādā tos:

- `workoutStateMessageAction (IncomingWorkoutStateMessage message)` - metode ir apzīmēta ar Spring anotāciju *@MessageMapping("/workout-progress").* Ja ienākošajā ziņojumā ir teikts, ka treniņš ir palaists, tad tiek izsaukta klases

"WorkoutManager" metode `startWorkout()`. Ja treniņš nav palaists, tad tiek izsaukta klases "WorkoutManager" metode `stopWorkout()`;

- `stepSwitchMessageAction (IncomingStepSwitchMessage message)` - metode apzīmēta ar Spring anotāciju *@MessageMapping("/step-switch").* Tiek izsaukta klases "WorkoutManager" metode `switchToStep`, ka parametrs tiek padots ienākošajā ziņojumā norādītais soļa kārtas numurs;
- `sensorConfigUpdateMessageAction (IncomingSensorConfigUpdateMessage message)` - metode ir apzīmēta ar Spring anotāciju *@MessageMapping("/update-sensor-config").* Tiek izsaukta klases "SensorConfigManager" metode `setConfig(message)`, ka parametrs tiek padots ienākošais ziņojums pēc tā apstrādes ar klases "SensorConfigImpl" metodes `of(IncomingSensorConfigUpdateMessage message)` palīdzību.

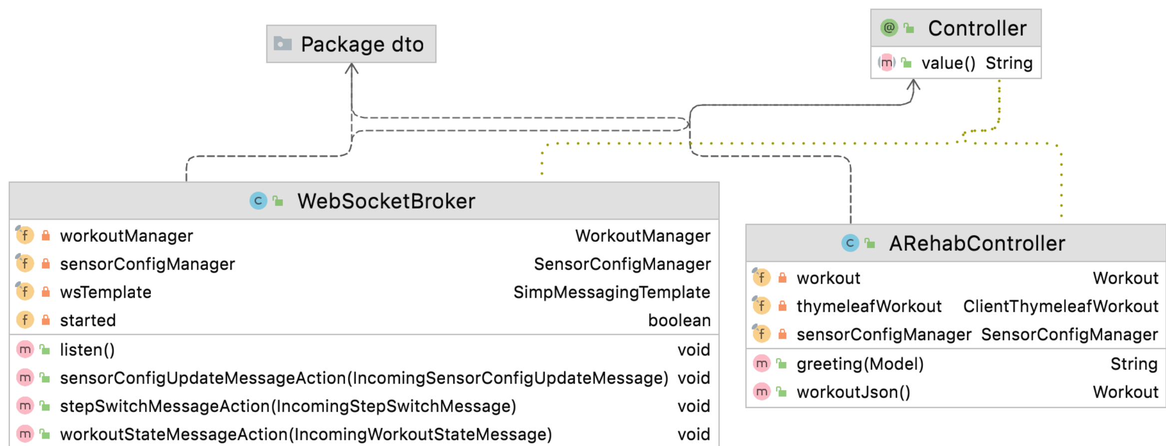
Klase "ARehabController"

Klase ir apzīmēta ar Spring anotāciju *@Controller*. Kontrolleris "ARehabController" ir galvenā moduļa klase, kas pārvalda tīmekļa lappusi. Klases lauki ir klašu "Workout", "ThymeleafWorkout" un "SensorConfigManager" instances, kuru klātbūtni vienā eksemplārā nodrošina Spring satvars sistēmas palaišanas brīdī, "būvējot" lietotnes komponentus, kontrollerus, konfigurācijas.

Kontrollera metode `greeting(Model model)` ir apzīmēta ar Spring anotāciju *@GetMapping*, norādītais vietrādis URL ir *"/workout"*. Metode paņem treniņa vingrojumus no klases "ThymeleafWorkout" un klases "SensorConfigManager" metodes `getConfig()` atgriezto vērtību, kas ir klases "OutgoingSensorConfigMessage" instance. Treniņa vingrojumus un sensoru konfigurāciju metode pievieno kā atribūtus modelim. Metode atgriež tīmekļa lapas šablonu *"workout.html"*, kas atrodas apakšsistēmas "Java-Application" mapē "*resources*".

Kontrollera metode `workoutJson()`, savukārt, ir apzīmēta ar Spring anotāciju *@GetMapping*, norādītais vietrādis URL ir *"/workout",* un metode atgriež klases "Workout" instanci.

Klašu "ARehabController" un "WebSocketBroker" struktūra ir parādīta klašu diagrammā zemāk (sk. *Att.3.20.*).



Att.3.20. **Klases "ARehabController" un "WebSocketBroker"**

3.5. Treniņa konfigurācijas uzglabāšana: apakšsistēma "Etcd"

"Etcd" ir sistēmas "ARehab Portable" apakšsistēma, kuras mērķis ir uzglabāt Java lietotnes uzģenerēto treniņa konfigurāciju, kuru izmantos AR lietotne. Apakšsistēmas mērķis ir realizēt daļu no sistēmas prasības par konfigurācijas redzamību AR lietotnei (lietotājstāsts *S2_ WORKOUT_CONFIG_AR*).

Līdz ar to, ka treniņa konfigurācijas saglabāšana sākas no gala lietotāja datora, bet lasīšana – no AR brillēm, ir jānodrošina piemērotā datu piekļuve.

Par pamatu treniņa konfigurācijas saglabāšanai tika paņemta atslēgu-vērtību datu glabātava Etcd, kas droši saglabā datus veidā "atslēga-vērtība" un ļauj tiem piekļūt dalītajai sistēmai vai mašīnu klasterim. Etcd datu glabātuve tiek palaista kā atsevišķs Docker konteineris, kas veidojas no Docker attēla "*bitnami/etcd:latest*" [6] bez jebkādiem papildus slāņiem. Apakšsistēma "Java-Application" vēršas tieši pie Etcd datu glabātuves, lai uzreiz pēc sistēmas "ARehab Portable" palaišanas saglabātu uzģenerēto treniņa konfigurāciju.

Etcd datu glabātuves korektai darbībai ir izšķiroša nozīme sistēmas "ARehab Portable" darbībā, tāpēc Docker komponēšanas konfigurācijas datnē *"docker-compose.yml"* tika norādīts, ka Docker konteineri "java-application" un "etcd-proxy" ir atkarīgi no konteinera "etcd", kas nozīmē, ka konteineram "etcd" ir jābūt vispirms palaistam, lai varētu palaist pārējos.

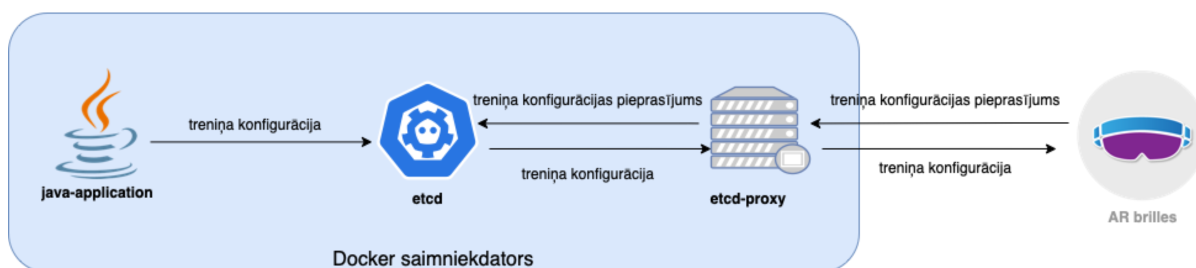
3.6. Treniņa konfigurācijas pieprasījumu apstrāde: apakšsistēma "Etcd-Proxy"

"Etcd-Proxy" ir sistēmas "ARehab Portable" apakšsistēma, kuras mērķis ir ļaut AR lietotnei piekļūt Java lietotnes uzģenerētajai treniņa konfigurācijai. Apakšsistēmas nolūks ir

realizēt daļu no sistēmas prasības par konfigurācijas redzamību AR lietotnei (lietotājstāsts *S2_ WORKOUT_CONFIG_AR*).

Apakšsistēma "Etcd-Proxy" ir starpniekserveris starp Etcd datu glabātavu un AR lietotni, kas pieprasa treniņa konfigurāciju. "Etcd-Proxy" pārķer šo treniņa konfigurācijas pieprasījumu, griežas pie Etcd datu glabātuves, pēc atslēgas saņem treniņa konfigurāciju un attēlo tās saturu AR lietotnei.

Attēlā 3.21. ir shematiski attēlota konteineru "java-application", "etcd", "etcd-proxy" un AR lietotnes mijiedarbība treniņa konfigurācijas datu plūsmā.



Att.3.21. **Konteineru "java-application", "etcd", "etcd-proxy" un AR lietotnes mijiedarbība**

Lai nodrošinātu saziņu starp izolētiem Docker konteineriem, tie ir jāapvieno vienā Docker tīklā. Tomēr, lai nezaudētu konteineru izolāciju, ir nepieciešami divi tīkli - pirmajā apvienojot Docker konteinerus "java-application" un "etcd" (konteineru saziņas mērķis – treniņa konfigurācijas saglabāšana) un otrajā – konteinerus "etcd" un "etcd-proxy" (konteineru saziņas mērķis – treniņa konfigurācijas iegūšana). Abi tīkli ir veidoti, izmantojot Docker tīkla tiltus (*bridge*).

"Etcd-Proxy" pamatā ir Nginx starpniekserveris, kas apstrādā treniņa konfigurācijas pieprasījumu no AR lietotnes. Lai vērsos pie Etcd datu glabātuves, "Etcd-Proxy" implementācijā tika izmantota bibliotēka "*lua-resty-etcd*" [7], un Nginx konfigurācijas datnē tika ievietots koda bloks programmēšanas valodā Lua [8], kas nodrošina treniņa konfigurācijas iegūšanas loģiku:

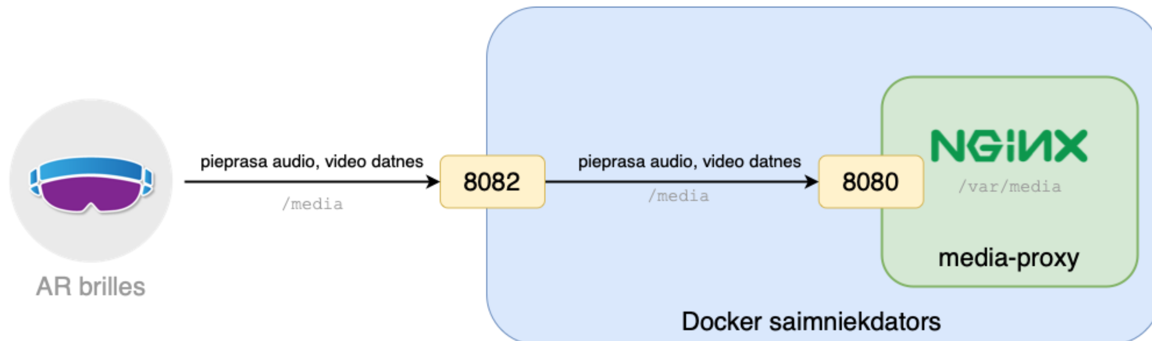
1. "Etcd-proxy" vēršas pie Etcd, pieprasot atslēgas *"config"* vērtību;
    1.1. Ja iegūst atbildi, tad Nginx izvada tās vērtību;
    1.2. Ja atbildes nav, tad Nginx izvada tukšu JSON.

3.7. Lokālu video un audio datņu izmantošana treniņa konfigurācijā: apakšsistēma "Media-Proxy"

"Media-Proxy" ir sistēmas "ARehab Portable" apakšsistēma, kuras nolūks ir kalpot par starpniekserveri starp AR lietotni, kas pieprasa treniņa konfigurācijā iekļautās audio un video datnes, un īsto serveri, kas šajā gadījumā ir gala lietotāja dators, uz kura glabājas iepriekšminētās datnes. Apakšsistēma "Media-Proxy" saglabā visas treniņa konfigurācijā izmantotās datnes sava Docker konteinera pārvaldītā saimniekdatora direktorijā, pārķer visus ar audio un video datnēm saistītos pieprasījumus un atgriež pieprasītās datnes. Apakšsistēmas mērķis ir realizēt daļu no sistēmas prasības par konfigurācijas redzamību AR lietotnei (lietotājstāsts *S2_ WORKOUT_CONFIG_AR*).

Apakšsistēmas "Media-Proxy" pamatā ir Nginx starpniekserveris, kas apstrādā pieprasījumus. Audio un video datņu glabāšanu Docker konteinera pārvaldītā saimniekdatora direktorijā nodrošina attiecīgais Docker sējums, kā norādīts koda fragmentā zemāk. Lokālu audio un video datņu saglabāšana Docker pārvaldītā direktorijā notiek konteinera "media-Proxy" dzīves cikla sākumā jeb tā palaišanas brīdī.

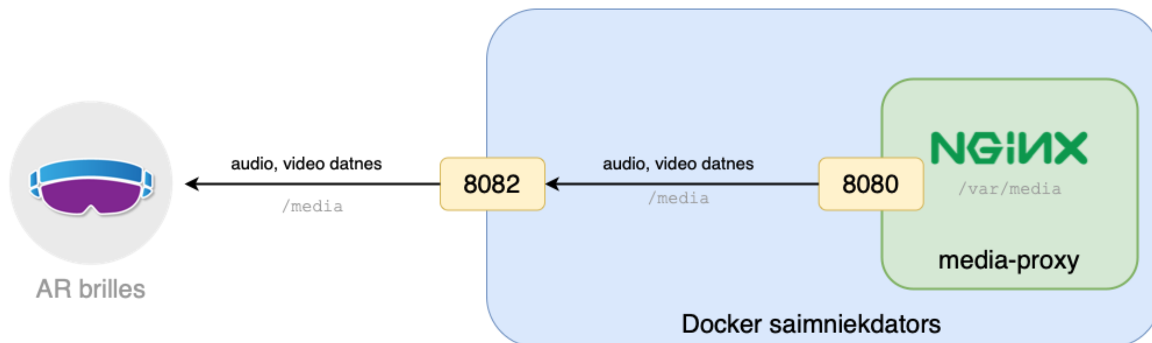
Attēlā zemāk (sk. *Att.3.22.*) ir redzams, kā AR lietotnes pieprasījums nonāk līdz Docker konteinerim "media-proxy", izmantojot ārējo portu 8082 un iekšējo portu 8080.



Att.3.22. **Pieprasījums no AR lietotnes uz Docker konteineru "media-proxy"**

AR lietotne pieprasa audio un video datnes no saimniekdatora, griežoties pie direktorijas "*/media*" un saimniekdatora porta - šajā gadījumā tas ir ports ar numuru 8082. Saimniekdatora Docker dzinējs pārķer šo pieprasījumu un novirza to uz konteinera "Media-Proxy" portu – šajā gadījumā ar numuru 8080. Docker konteinera uzglabāšanas sistēmā pastāv direktorija *"/var/media*", kurā glabājas visas konfigurācijā izmantotās audio un video datnes. Apakšsistēmā "Media-Proxy" strādājošs Nginx starpniekserveris, saņemot pieprasījumu, griežas pie šīs direktorijas un atgriež datni ar pieprasījumā norādīto nosaukumu, ja tāda ir.

Attēlā 3.23. ir redzams, ka “Media-Proxy” līdzīgā veidā atgriež pieprasītas datnes, sūtot tos uz ārējo saimniekdatora portu.



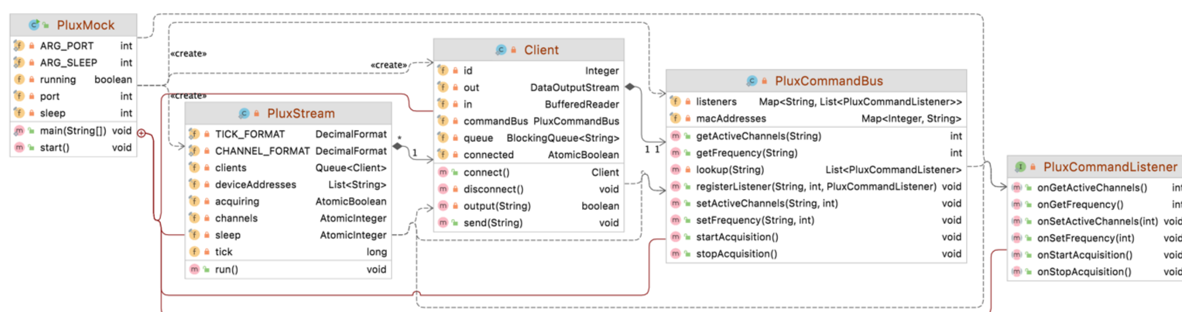
Att.3.23. **Docker konteineris “media-proxy” atgriež pieprasītās datnes**

3.8. Plux sensora modelis: programma “PluxMock”

Pēc sistēmas prasībām, pirmā sprinta ietvaros jārealizē sensora modelis-aizbāznis, kas atdarina “OpenSignals” saskarni. Programmas nolūks ir realizēt šo sistēmas prasību (sprinta uzdevums *S1_PLUX_MOCK*).

Programmai ir viena klase – izpildāma klase “PluxMock”. Klases izpildāma metode `main(String[] args)` kā parametru pieņem porta numuru, uz kura jāpalaiž sensora modelis.

Klasei “PluxMock” ir vairākas iekšējas klases un saskarnes - diagramma zemāk ir parādīta klases struktūra (sk. *Att.3.24.*).



Att.3.24. **Klases “PluxMock” struktūra**

Klases “PluxMock” metodē `start()` vispirms tiek mēģināts atvērt servera kontaktligzdu, veidojot Java klases *ServerSocket* objektu, uz parametrā saņemtā porta. Tad tiek veidota klientu rinda ar datu tipu *ConcurrentLinkedQueue*, iekšējās klases “PluxCommandBus” objekts, iekšējās klases “PluxStream” objekts, kas kā konstruktora parametrus saņem klientu rindu, “PluxCommandBus” instanci un sarakstu jebkādā garumā ar izstrādātāja vai testētāja

izdomātām sensoru MAC adresēm. Klases “PluxStream” objektam tiek izsaukta metode `start()`. Tad mūžīgā cilpā tiek mēģināts pieņemt jaunu klientu savienojumu, veidojot jaunu Java *Socket* klases instanci un pievienojot to klientu rindai.

Lietojumprogrammatūras “OpenSignals” Plux komandas [5], kuras jārealizē sensora modelī, ir:

- komanda `start`,
- komanda `stop`,
- komanda `config,{MAC|DEVICE ID},samplingfreq`,
- komanda `config,{MAC|DEVICE ID},activechannels`,
- komanda `config,{MAC|DEVICE ID},samplingfreq,{VALUE}`,
- komanda `config,{MAC|DEVICE ID},activechannels,{VALUE}`

(sprinta uzdevums *S1_PLUX_MOCK*). Klases “PluxMock” iekšējā saskarne “PluxCommandListener” piedāvā metožu komplektu, kuru jārealizē, implementējot Plux komandu “klausītāju” – metodes `onStartAcquisition()`, `onStopAcquisition()`, `onGetActiveChannels()`, `onSetActiveChannels(int)`, `onGetFrequency()`, `onSetFrequency(int)`.

Klases “PluxMock” iekšējās klases “PluxCommandBus” metode `registerListener(String macAddress, int order, PluxCommandListener listener)` ievieto atslēgu-vērtību sarakstā `macAddresses` saņemto MAC adresi, kā arī piesaista saņemto “PluxCommandListener” objektu pie šīs MAC adreses atslēgu-vērtību sarakstā `listeners`. Klase arī realizē metodes, `startAcquisiton()`, `stopAcquisition()`, `getActiveChannels(String id)`, `setActiveChannels(String id, int mask)`, `getFrequency(String id)`, `setFrequency(String id, int frequency)`. Šo metožu parametru skaits atbilst “OpenSignals” komandu [5] parametru skaitam. Metodēs katram “PluxCommandListener” objektam no `listeners` tiek izsaukta attiecīgā “PluxCommandListener” metode.

Klases “PluxMock” iekšējā klase “Client” apraksta datu pārraides klienta mijiedarbību ar programmu. Klases lauki ir Java klašu *DataOutputStream*, *BufferedReader* un “PluxMock” iekšējās klases “PluxCommandBus” objekti. Tiek veidota rinda `queue` ar Java datu tipu *LinkedBlockingQueue*. Klases metodē `connect()` tiek palaisti divi pavedieni – pirmais pavediens pieslēgtam klientam pārbauda, vai klients vēl nav atslēdzies, mēģinot nolasīt kaut ko no rindas; ja klients ir atslēdzies, tiek izsaukta metode `disconnect()`, kas savukārt notīra rindu. Otrais pavediens pārvalda klienta ievadītās komandas, vispirms ielasot tos no *BufferedReader* objekta un pēc tam interpretējot tos atbilstoši programmas prasībām (sprinta

uzdevums *S1_PLUX_MOCK*), izsaucot atbilstošās klases "PluxCommandBus" objekta metodes atkarībā no ievadītās komandas un izvadot rezultātu rindā ar *DataOutputStream* objekta palīdzību.

Klases "PluxMock" iekšējā klase "PluxStream" manto no Java *Thread* klases un ir atbildīga par nejaušu sensoru datu ģenerēšanu un noformēšanu katram no lietotāja ievadītām izdomātām sensoru MAC adresēm. Klases lauki ir rinda no klases "Client" objektiem; saraksts ar sensoru MAC adresēm; *AtomicBoolean* tipa mainīgais `acquiring`, kas nosaka, vai dotajā brīdī notiek datu ģenerēšana; *AtomicInteger* tipa mainīgais `channels`, kas reprezentē sensoru kanālu bināro masku, kur 1 nozīmē, ka kanāls attiecīgajā pozīcijā ir ieslēgts, bet 0 – ka nav ieslēgts; *AtomicInteger* tipa mainīgais `sleep`, kas nosaka, cik ilgu pauzi ir jātaisa pavedienam (tāpēc sanāk, `1000 / sleep` ir sensora "frekvence"). Klases konstruktorā, katrai sensoru MAC adresei no saraksta `macAddresses` klases "PluxCommandBus" objektam tiek izsaukta metode `registerListener(String macAddress, int order, PluxCommandListener listener)`, realizējot saskarni "PluxCommandListener" ar anonīmu klasi – realizācijas katrā metodē programmas konsolē tiek izvadīts paziņojums par veikto darbību un tiek attiecīgi mainīti klases lauki `channels` un `sleep`. Klases-pavediena metodē `run()` tiek aprakstīta pavediena pauzes taisīšana atkarībā no `sleep` vērtības un datu ģenerēšana atkarība no sensoru kanālu binārās maskas `channels`. Ja kārtējā kanāla vērtība tā pozīcija maskā ir 1, datu masīvam tiek pievienota nejauši uzģenerēts šī sensora kanāla "mērījums" atbilstoši "OpenSignals" dokumentācijā aprakstītajām sensoru datu formātam [5].

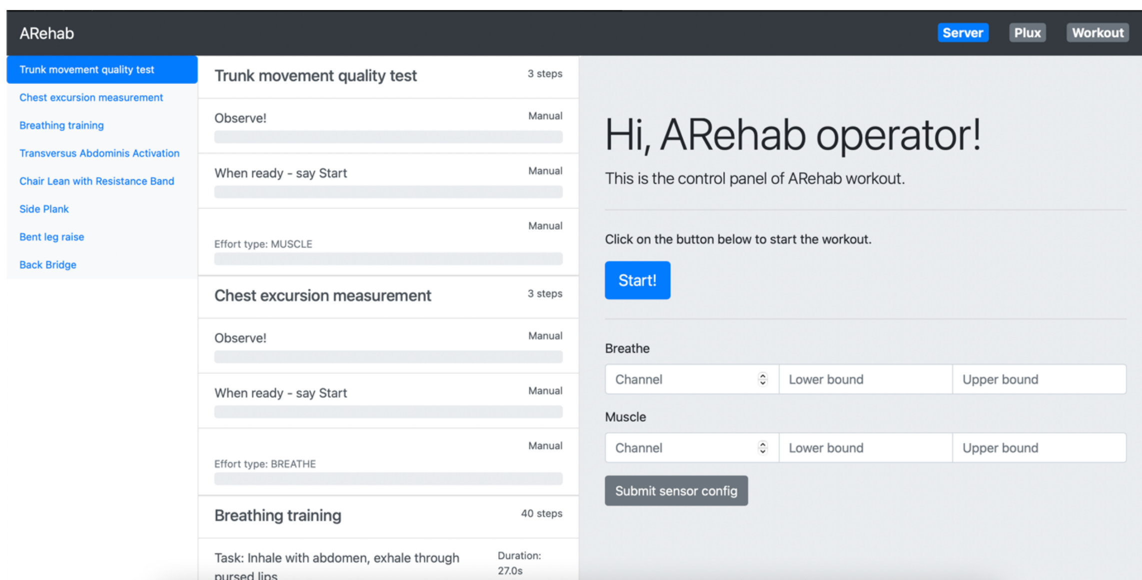
Zemāk ir ievietots fragments no "PluxMock" uzģenerētajiem datiem; fragments tika saīsināts labākai uzskatāmībai (sk. *Att.3.25.*).


```
{"returnCode": 0, "returnData": {"00:00:00:00:00:00": [[1.0, 0.25362,
0.06723, 0.00001, 0.50785, 0.25362, 0.06723, 0.00001], [2.0, 0.26049,
0.07122, 0.00002, 0.51571, 0.26049, 0.07122, 0.00002], [3.0, 0.26741,
0.07531, 0.00016, 0.52355, 0.26741, 0.07531, 0.00016], [4.0, 0.27439,
0.07951, 0.00041, 0.53140, 0.27439, 0.07951, 0.00041]],
"11:11:11:11:11:11": [[151.0, 0.98187, 0.98309, 0.85129, 0.84796,
0.98187, 0.98309, 0.85129], [152.0, 0.97971, 0.98505, 0.85683, 0.84227,
0.97971, 0.98505, 0.85683], [153.0, 0.97744, 0.98690, 0.86229, 0.83651,
0.97744, 0.98690, 0.86229]]}}
```


Att.3.25. **Saīsināts fragments no programmas "PluxMock" uzģenerētajiem datiem**

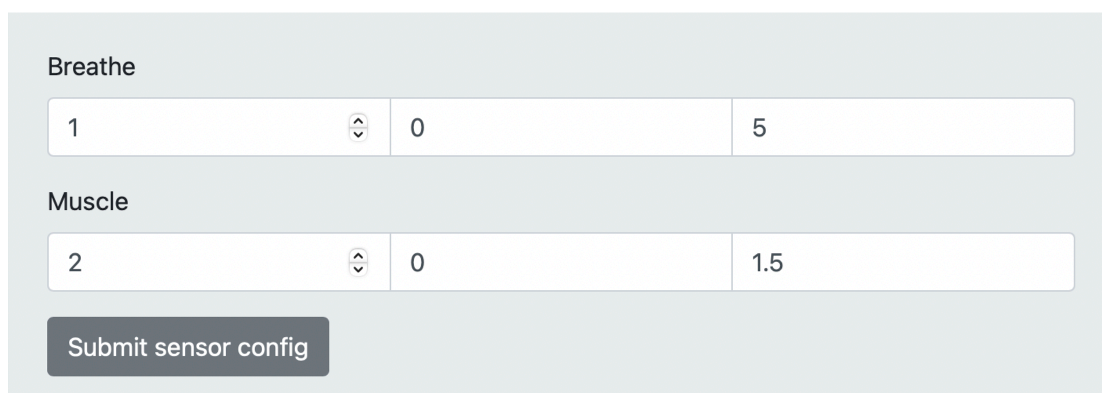
3.9. Lietotāja saskarnes projektējums

Pēc sistēmas četru Docker konteineru palaišanas lietotājs var doties uz tīmekļa lapu, lai ierakstītu sensoru iestatījumus, palaistu treniņu un pārvaldītu to. Attēlā zemāk ir parādīts tīmekļa lapas stāvoklis uzreiz pēc sistēmas palaišanas (sk. *Att.3.26.*).



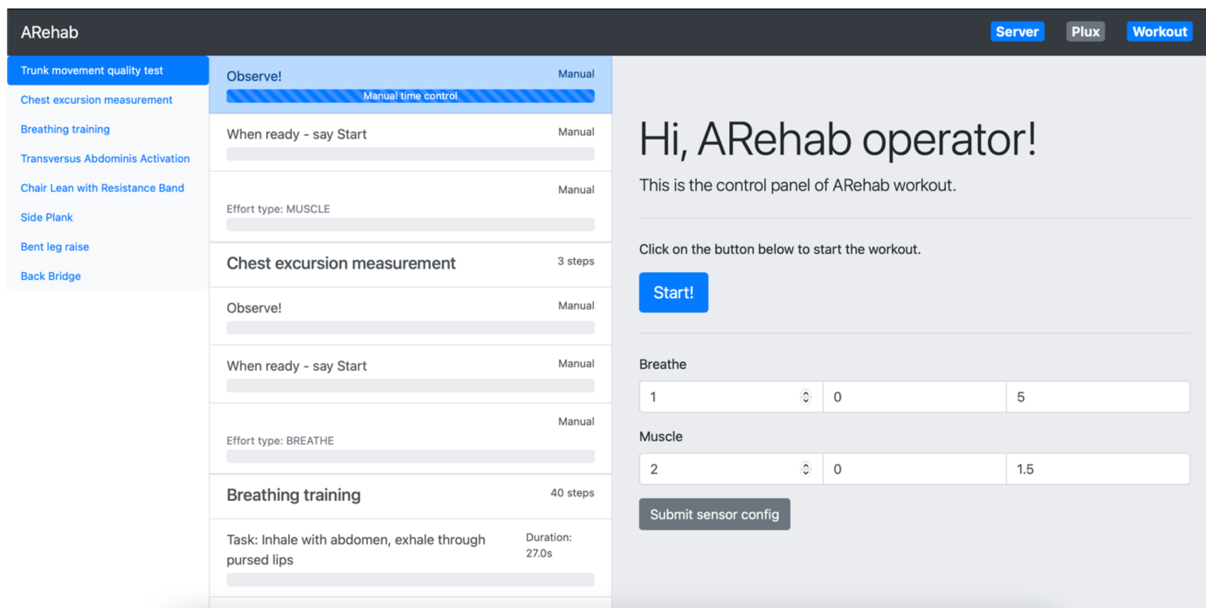
Att.3.26. **Tīmekļa lapas stāvoklis pēc sistēmas palaišanas**

Tīmekļa lapas kreisajā pusē ir saraksts ar visiem treniņa vingrojumiem pēc katras. Lapas centrā atrodas saraksts ar visiem treniņa soļiem, kas ir sagrupēti pa vingrojumiem. Par katru soli var redzēt to tekstuālu saturu, ilgumu vai "*Manual*", ja solim ir manuāla pāreja uz nākamo. Lapas labajā pusē atrodas poga "*Start!*" treniņa palaišanas, zem tās – forma sensoru iestatījumiem (sk. *Att.3.27.*), kas lietotājam jāaizpilda pirms treniņa palaišanas. Kad sensoru iestatījumi ir ievadīti, lietotājam jānoklikšķina uz pogas "*Submit sensor config*".



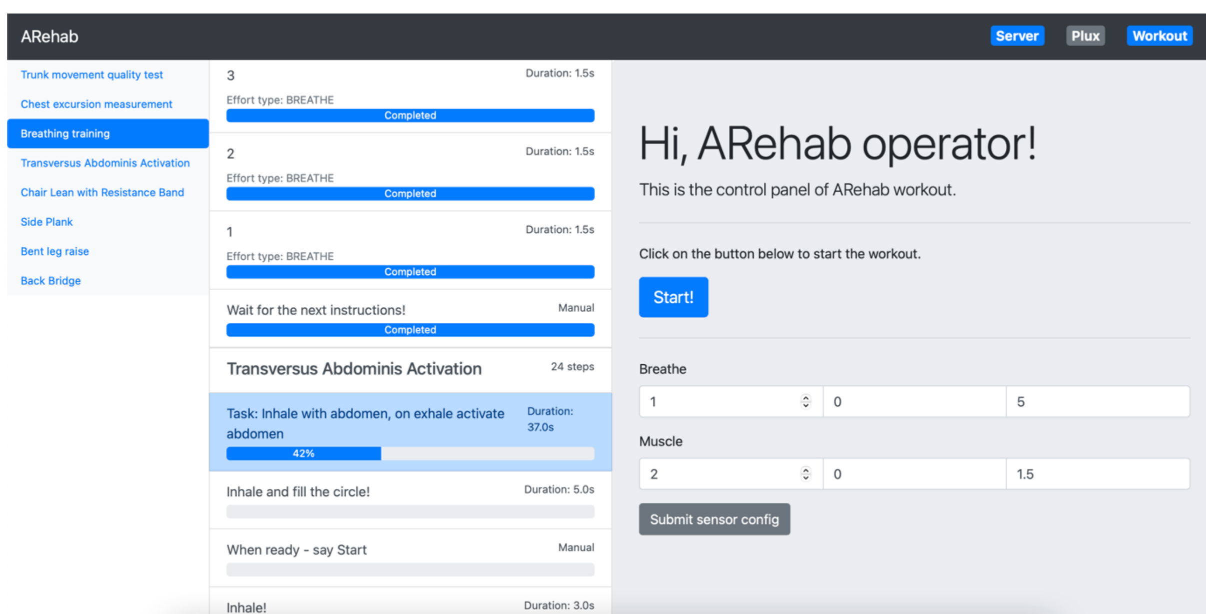
Att.3.27. **Forma, kurā lietotājs ievada sensora iestatījumus pirms treniņa palaišanas**

Kad tas ir izdarīts, var laist treniņu. Attēlā zemāk ir parādīts tīmekļa lapas stāvoklis uzreiz pēc treniņa palaišanas (sk. *Att.3.28.*).



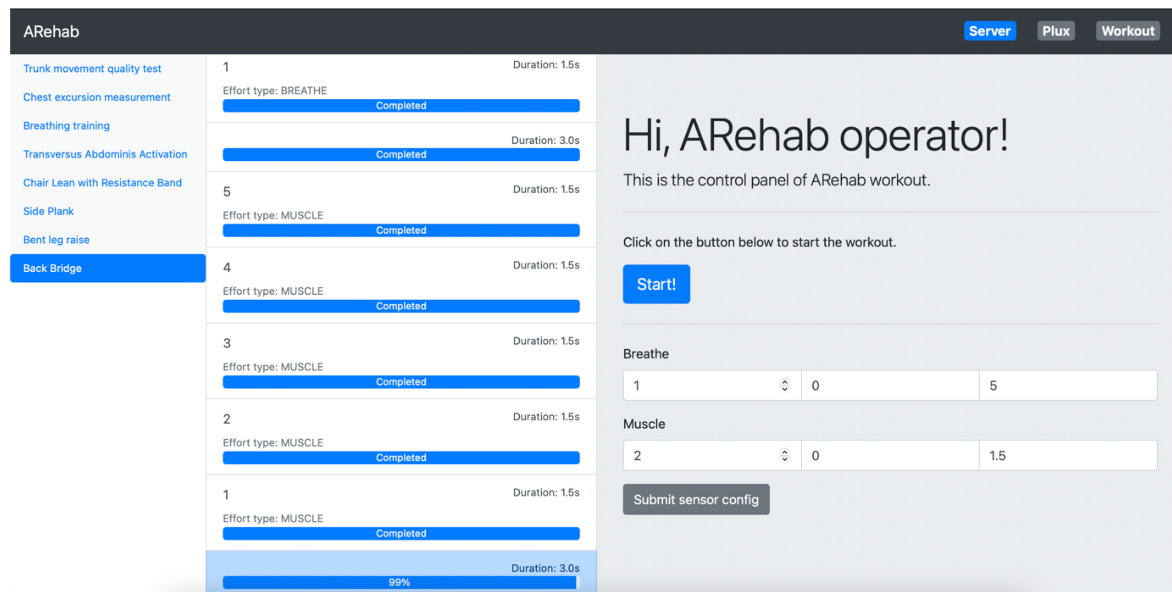
Att.3.28. **Tīmekļa lapas stāvoklis pēc pogas "*Start!*" nospiešanas**

Tekošajam treniņa solim, ja tam ir norādīts ilgums, var redzēt tā dinamisku progresa indikatoru procentos. Visi iepriekšējie soļi ir apzīmēti kā "*Completed*". Attēlā 3.1.4. ir parādīts tīmekļa lapas stāvoklis treniņa ar ieslēgto treniņu (sk. *Att.3.29.*). Lietotājs var brīvi pārslēgt soļus abos virzienos jebkurā kārtībā.



Att.3.29. **Tīmekļa lapas stāvoklis ar ieslēgto treniņu**

Attēlā zemāk ir parādīts tīmekļa lapas stāvoklis, kad visi treniņa soļi ir izieti un treniņš ir beidzies (sk. *Att.3.30.*). Atkārtoti uzspiežot "*Start!*", treniņš palaidīsies no jauna.



Att.3.30. **Tīmekļa lapas stāvoklis pēc treniņa beigām**

3.10. Sistēmas palaišana

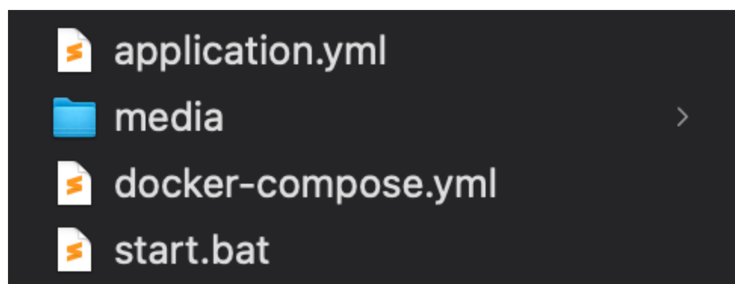
Sistēmas piegāde gala lietotājam tiek nodrošināta ar Docker attēlu reģistra "DockerHub" palīdzību. Apakšsistēmu "Java-Application", "Etcd-Proxy" un "Media-Proxy" Docker attēlu pēdējās versijas tiek augšupielādētas Docker attēlu reģistra "DockerHub" privātā repozitorijā, kuram gala lietotājam ir piekļuve.

Gala lietotāja datorā jābūt direktorijai, no kuras tiks palaista sistēma (sk. *Att.3.31.*). Direktorijā jābūt:

1. Treniņa un sistēmas konfigurācijas datnei "*application.yml*";
2. Mapei "*media*", kurā tiek glabātas treniņā izmantotās audio un video datnes;
3. Docker komponēšanas datnei "*docker-compose.yml*";
4. Teksta datnei "*start*" (Windows operētājsistēmai tā var būt, piemēram, *.bat* palaižamā datne) ar kodu, kuram jāizpildās, lai palaistu sistēmu. Datnē ir sekojošas koda rindas:

```
docker login -u <DockerHub lietotāja ar piekļuves tiesībām lietotājvārds>
docker pull arehab-java-application:latest
docker pull arehab-etcd-proxy:latest
docker pull arehab-media-proxy:latest
docker-compose up
```

Ar šo kodu lietotājs pieteicas “DockerHub” sistēmā un tiek lejupielādētas Docker attēlu pēdējās versijas. Pēc komandas `docker-compose up` no attēliem tiek veidoti un palaisti Docker konteineri.

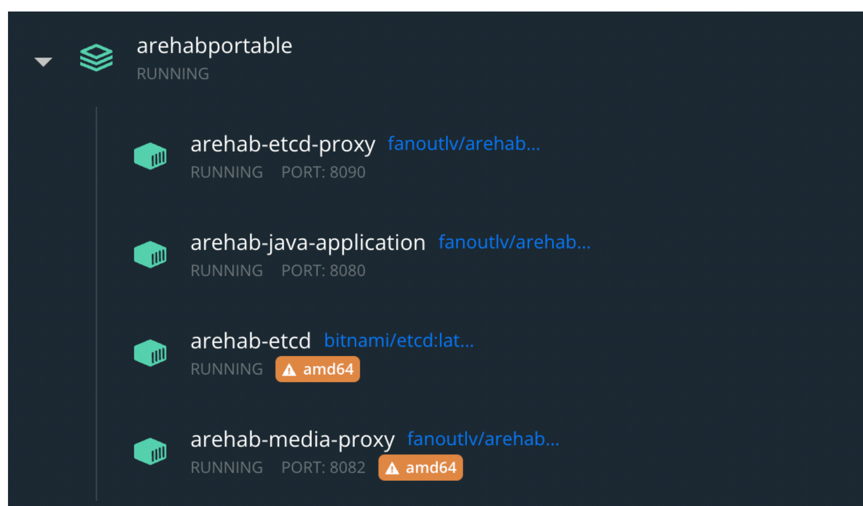


Att.3.31. **Direktorija sistēmas palaišanai**

Papildus prasības sistēmas palaišanai ir:

1. Palaista lietojumprogramma “OpenSignals” ar pieslēgtiem Plux sensoriem un ar atvērto TCP savienojumu (pēc apraksta “OpenSignals” dokumentācijā [5]);
2. Treniņš un sistēmas palaišanai nepieciešamie parametri datnē “*application.yml*”.

Palaižot kodu no datnes “*start*” (ja tā ir *.bat* datne Windows operētājsistēmā – uzklikšķinot uz tās divas reizes, pārējos gadījumos - iekopējot kodu konsolē), tiek palaisti četri Docker konteineri. Lietotājs var izvēlēties, kur viņam ir ērtāk pārvaldīt konteinerus - tos var redzēt, apstādināt un atkārtoti palaist arī no lietojumprogrammas “Docker Desktop” (Linux operētājsistēmās – “Docker”) (sk. *Att.3.32.*).



Att.3.32. **Palaisti sistēmas “ARehab Portable” Docker konteineri**

Uzreiz pēc konteineru palaišanas konsolē var redzēt dažādus paziņojumus no konteineriem (sk. *Att.3.33.*).

```
arehab-java-application | Started Application in 1.581 seconds (JVM running for
1.918)
arehab-java-application | IP address resolved...
arehab-java-application | Generating config...
arehab-java-application | Config saved...
arehab-java-application | Connecting to Plux...
arehab-java-application | Plux connection established
arehab-java-application | Patient effort accumulator started
arehab-java-application | Multicast server started on port 8888
```

Att.3.33. **Paziņojumi no konteinera “java-application” pēc sistēmas palaišanas**

Kad konteineri ir palaisti, lietotājs var atvērt pārlūkprogrammu un doties uz tīmekļa lapu (*localhost:8080/workout*), kur viņš var ierakstīt sensoru iestatījums, palaist un pārvaldīt treniņa gaitu (tīmekļa lapas projektējums aprakstīts nākamajā dokumenta sadaļā).

Pēc treniņa palaišanas konsolē var redzēt apstrādātu sensoru datu un treniņa soļu pārraides ziņojumus, kas reāllaikā tiek sūtīti uz AR brillēm (sk. *Att.3.34.*).

```
ARehab Portable — docker-compose-v1 ‹ docker-compose up — 80×24
arehab-java-application | [1, 0.53, 1, 1]
arehab-java-application | [1, 0.54, 1, 1]
arehab-java-application | [1, 0.55, 1, 1]
arehab-java-application | [1, 0.56, 1, 1]
arehab-java-application | [1, 0.58, 1, 1]
arehab-java-application | [1, 0.59, 1, 1]
arehab-java-application | [1, 0.60, 1, 1]
arehab-java-application | [1, 0.61, 1, 1]
arehab-java-application | [1, 0.62, 1, 1]
arehab-java-application | [1, 0.64, 1, 1]
arehab-java-application | [1, 0.65, 1, 1]
arehab-java-application | [1, 0.66, 1, 1]
arehab-java-application | [1, 0.67, 1, 1]
arehab-java-application | [1, 0.69, 1, 1]
arehab-java-application | [1, 0.70, 1, 1]
arehab-java-application | [1, 0.71, 1, 1]
arehab-java-application | [1, 0.73, 1, 1]
arehab-java-application | [1, 0.73, 1, 1]
arehab-java-application | [1, 0.75, 1, 1]
arehab-java-application | [1, 0.77, 1, 1]
arehab-java-application | [1, 0.78, 1, 1]
arehab-java-application | [1, 0.80, 1, 1]
arehab-java-application | [1, 0.81, 1, 1]
```

Att.3.34. **Treniņa datu pārraides ziņojumi reāllaikā**

4. TESTĒŠANAS DOKUMENTĀCIJA

4.1. Testēšanas metodes

Sistēmas “ARehab Portable” testēšana notika divos veidos – apakšsistēmas “Java-Application” klases tika testētas ar automatizētiem vienībtestiem, kas tika rakstītas, izmantojot Java satvaru *JUnit*, savukārt sistēmas prasības tika testētas ar manuāliem testiem.

Testēšanas dokumentācijas mērķis ir veicināt sistēmas “ARehab Portable” vienībtestēšanu, lai pārbaudītu sistēmas darbības pareizību un atbilstību izvirzītajām prasībām.

4.2. Automatizētā apakšsistēmas “Java-Application” klašu vienībtestēšana

Sistēmas automatizētai testēšanai par testējamām vienībām tika uzskatītas apakšsistēmas “Java-Application” klases un to metodes. Tabulā zemāk (sk. *Tab.4.1.*) ir apkopoti “Java-Application” klašu automatizēti *JUnit* vienībtesti un to rezultāti.

Tab.4.1. Apakšsistēmas “Java-Application” klašu vienībtesti un testēšanas rezultāti

Testējamā klase	**Vienībtests**	**Testa rezultāts**
WorkoutManager	*whenSwitchToStepIsCalled_thenStepIsSwitched()*	✓
Client ThymeleafWorkout	*whenMethodOfIsCalled_thenWebWorkoutIsCreated()*	✓
ClientUnityWorkout	*whenMethodOfIsCalled_thenUnityWorkoutIsCreated()*	✓
Exercise	*whenExercisesAreCreated _thenComparisonWorksCorrectly()*	✓
LinkedStep	*whenLinkedStepIsCreated_thenItPointsToNextOne()*	✓
Step	*whenStepsAreCreated _thenComparisonWorksCorrectly()*	✓
	whenStepIsCreated_thenHasDurationWorksCorrectly()	✓
	whenStepIsCreated_thenGetDurationMSWorksCorrectly()	✓
Workout	*whenWorkoutConfigIsPassed _thenParseMethodWorksCorrectly()*	✓
IncomingSensor ConfigUpdate Message	*whenGetChannelByTypeIsCalled _thenCorrectChannelNumberIsReturned()*	✓
	whenGetLowerBoundByTypeIsCalled _thenCorrectLowerBoundIsReturned()	✓
	whenGetUpperBoundByTypeIsCalled _thenCorrectUpperBoundIsReturned()	✓
OutgoingSensor ConfigMessage	*whenOfMethodIsCalled _thenCorrectSensorConfigMessage IsGeneratedBasedOnSensorConfig()*	✓
ARehabConfig	*whenGetMulticastTickDurationMSIsCalled _thenCorrectValueIsReturned()*	✓
	whenGetAccumulatedTickDurationMSIsCalled _thenCorrectValueIsReturned()	✓

	whenGetMuscleAccumulatorFrameLengthIsCalled_ thenCorrectValueIsReturned()	✓
	whenGetBreatheAccumulatorFrameLengthIsCalled _thenCorrectValueIsReturned()	✓
	whenDevicesAreRequestedBySupportedType_ thenCorrectDevicesAreReturned()	✓
PluxTcpConnection	*whenTryToEstablishTcpConnection _thenAttemptIsSuccessful()*	✓
	whenConnectionIsEstablished_thenItIsActive()	✓
	whenConnectionIsActive_thenCommandIsSent()	✓
	whenNoConnection_thenCommandCannotBeSent()	✓
BreatheEffort Accumulator	*whenBreatheDataIsPassed _thenCorrectAccumulatedValueIsReturned()*	✓
MuscleEffort Accumulator	*whenMuscleDataIsPassed _thenCorrectAccumulatedValueIsReturned()*	
PatientEffortCalculator	*whenCalculateMethodIsCalledForBreatheStep _thenCorrectValueIsCalculated()*	✓
	whenCalculateMethodIsCalledForMuscleStep _thenCorrectValueIsCalculated()	✓
PatientActivity StateManager	*whenLastEffortIsSet_thenItIsStored()*	✓
	whenLastAccumulatedEffortIsSet_thenItIsStored()	✓
PatientActivity MessageImpl	*whenGetAsStringIsCalled _thenCorrectMessageFormatIsReturned()*	✓
SensorConfigImpl	*whenConfigUpdateMessageIsPassed _thenCorrectConfigIsCreated()*	✓
SensorConfigManager	*whenInboundByType_thenCorrectValueIsReturned()*	✓
ListAverage StrategyImpl	*whenListOfValuesIsPassed_thenAverageIsCalculated()*	✓
RMSReduceStrategy	*whenListOfValuesIsPassed_thenRMSIsCalculated()*	✓

4.3. Manuālā sistēmas prasību testēšana

Sistēmas prasību manuālā testēšana notika vairākkārt uz dažādām vidēm. Katra sistēmas prasība – lietotājstāsts vai sprinta uzdevums – tika ņemta kā testējamā vienība. Prasību testēšanai tika izmantota gan izstrādei un testēšanai paredzētā izpildāmā programma “PluxMock”, gan “OpenSignals” lietojumprogramma datu iegūšanai no īstiem Plux sensoriem.

Tabulās zemāk (sk. *Tab.4.2. - Tab.4.13.*) ir apkopoti testpiemēri katrai sistēmas prasībai. Katram testpiemēram ir norādīts testpiemēra numurs, soļi testpiemēra palaišanai un testa sagaidāmais rezultāts.

Tab.4.2. Testpiemēri sistēmas prasības

"*S1_SENSOR_ DATA_ ACQUISITION*" testēšanai

Tests	**Soļi testpiemēra palaišanai**	**Sagaidāmais rezultāts**
T1_1	1. Palaist sistēmu bez ieslēgtiem sensoriem.	Sistēma katru sekundi izdod paziņojumu "*Plux connection failed. Retrying in 1s...*".
T1_2	1. Palaist sistēmu bez ieslēgtiem sensoriem. 2. Kad sistēma izdod vismaz vienu paziņojumu veidā "*Plux connection failed. Retrying in 1s...*", ieslēgt "OpenSignals" sensoru datu pārraidi.	Kad ir uzstādīts savienojums ar sensoru datu plūsmu no "OpenSignals", sistēma izdod paziņojumu "*Plux connection established*".
T1_3	1. Palaist sistēmu bez ieslēgtiem sensoriem. 2. Kad sistēma izdod vismaz vienu paziņojumu veidā "*Plux connection failed. Retrying in 1s...*", ieslēgt programmas "PluxMock" datu pārraidi.	Kad ir uzstādīts savienojums ar "PluxMock" datu plūsmu, sistēma izdod paziņojumu "*Plux connection established*".

Tab.4.3. Testpiemēri sistēmas prasības

"*S1_PLUX_MOCK*" testēšanai

Tests	**Soļi testpiemēra palaišanai**	**Sagaidāmais rezultāts**
T2_1	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*start*".	Programma izvada nejauši uzģenerētos sensoru datus formātā, kas atbilst OpenSignals "TCP/IP Module Guide" [5] dokumentācijai.
T2_2	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*start*". 4. Kad programma sāka izvadīt uzģenerētos sensoru datus, ierakstīt komandu "*stop*".	Programma pārtrauc nejauši uzģenerēto sensoru datu izvadi.
T2_3	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*config,0,samplingfreq*". 4. Kad programma izvada rezultātu, ierakstīt komandu "*config,0,samplingfreq,100".* 5. Ierakstīt komandu "*config,0,samplingfreq*".	Programma izvada frekvenci, kas atbilst uzstādītai frekvences vērtībai – 100.
T2_4	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*config,0,activechannels".*	Programma izvada aktīvo sensora kanālu bitu masku, kas atbilst uzstādītai maskas vērtībai – 11110000.

	4. Kad programma izvada rezultātu "11111111", ierakstīt komandu "*config,0,activechannels, 11110000*". 5. Ierakstīt komandu "*config,0,activechannels*".	
T2_5	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*config,0,activechannels".* 4. Kad programma izvada rezultātu "11111111", ierakstīt komandu "*config,0,activechannels, 11110000*". 5. Ierakstīt komandu "*start*".	Programma izvada nejauši uzģenerētos sensoru datus, katrā datu blokā ir 5 vērtības – pirmā vērtība ir skaitītājs, pārējās 4 ir atbilstošu ieslēgto kanālu dati.
T2_6	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*devices".*	Programma izvada paziņojumu "*Not implemented.*
T2_7	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*config*".	Programma izvada paziņojumu "*Wrong param count*".
T2_8	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*config,0,a*".	Programma izvada paziņojumu "*Unknown command*".
T2_9	1. Palaist programmu "PluxMock". 2. Ar *netcat* pieslēgties portam, uz kura ir palaista programma. 3. Ierakstīt komandu "*command*".	Programma izvada paziņojumu "*Undefined command*".

Tab.4.4. Testpiemērs sistēmas prasības "*S2_WORKOUT _CONFIG*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T3_1	1. Atvērt sistēmas treniņa konfigurācijas datni "*application.yml*". 2. Apskatīt treniņa konfigurācijā esošos laukus.	Par katru treniņa vingrojumu un vingrojuma soli treniņa konfigurācijas datnē ir iespējams glabāt, apskatīt un labot visu informāciju, kas ir pieminēta sistēmas prasības ar identifikatoru *S2_WORKOUT _CONFIG* akceptēšanas kritērijos.

Tab.4.5. Testpiemēri sistēmas prasības "*S2_STEPS_TEMPLATES_GROUPS*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T4_1	1. Atvērt sistēmas treniņa konfigurācijas datni "*application.yml*". 2. Apskatīt treniņa konfigurāciju.	Treniņa konfigurācijas datnē soļi tiek apvienoti grupās un veido šablonus.
T4_2	1. Palaist sistēmu. 2. Ar HTTP klientu (piemēram, *Postman*) izpildīt POST pieprasījumu uz "*/client/api/v1/workout*".	Treniņa konfigurācijā soļu grupas un šabloni tika aizstāti ar korektiem atsevišķiem soļiem.

Tab.4.6. Testpiemēri sistēmas prasības "*S2_WORKOUT_CONFIG_AR*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T5_1	1. Palaist sistēmu.	Pēc palaišanas sistēma izdod paziņojumi "*Generating config...*" un "*Config saved…*".
T5_2	1. Palaist sistēmu. 2. Ieraugot paziņojumu "*Config saved…*", ar HTTP klientu (piemēram, *Postman*) izpildīt POST pieprasījumu uz "*/client/api/v1/workout*".	Dotais pieprasījums atgriež korektu treniņa konfigurāciju.

Tab.4.7. Testpiemēri sistēmas prasības "*S2_DEVICES_CONFIG*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T6_1	1. Atvērt sistēmas treniņa konfigurācijas datni "*application.yml*". 2. Apskatīt sistēmas konfigurācijas lauku "*devices*".	Par katru sensoru datnē var glabāt visu informāciju, kas pieminēta sistēmas prasības ar identifikatoru *S2_DEVICES_CONFIG* akceptēšanas kritērijos.

Tab.4.8. Testpiemēri sistēmas prasības "*S3_WEB_WORKOUT*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T7_1	1. Palaist sistēmu. 2. Atvērt tīmekļa lietotni adresē "*localhost:8080/workout*".	Pārlūkprogrammā var redzēt visus treniņa soļus pēc kārtas, sagrupētus pa treniņa vingrojumiem, atbilstoši treniņa konfigurācijas datnes saturam. ar katru vingrojumu var redzēt tā nosaukumu. Par katru soli var

		redzēt tā tekstu un ilgumu (vai MANUAL) un piepūles veidu (ja norādīts) formā "*BREATHE*" vai "*MUSCLE*".
T7_2	1. Palaist sistēmu. 2. Atvērt tīmekļa lietotni adresē "*localhost:8080/workout*". 3. Uzstādīt sensoru iestatījumus un noklikšķināt uz pogas "*Submit sensor config*". 4. Noklikšķināt uz pogas "*Start!*".	Treniņa soļi sāk automātiski pārslēgties pēc kārtas atbilstoši soļos noradītajiem ilgumiem. Ja solī nav norādīts ilgums, sistēma gaida ievadi no lietotāja, lai pārslēgtu soli. Var pārslēgt soļus abos virzienos manuāli. Ieslēgtajam solim var redzēt dinamisku progresa indikatoru procentos.

Tab.4.9. Testpiemēri sistēmas prasības "*S3_SENSOR_CONFIG_WEB*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T8_1	1. Palaist sistēmu. 2. Atvērt tīmekļa lietotni adresē "*localhost:8080/workout*".	Katram piepūles veidam formātā "*Breathe*" vai "*Muscle*" ir iespējams norādīt skaitli, kas atbilst sensora kanālam, no kura jālasa dati. Katram piepūles veidam formātā "*Breathe*" vai "*Muscle*" ir iespējams norādīt divus skaitļus - minimālo un maksimālo vērtību.
T8_2	1. Palaist sistēmu. 2. Atvērt tīmekļa lietotni adresē "*localhost:8080/workout*". 3. Uzstādīt sensoru iestatījumus un noklikšķināt uz pogas "*Submit sensor config*". 4. Noklikšķināt uz pogas "*Start!*".	Sistēma izmanto norādītos kanāla numurus datu lasīšanai atkarībā no kārtējā soļa piepūles veida. Sistēma izvada apstrādātos datus, kārtējā apstrādātā vērtība ir skaitlis no 0 līdz 1 ar diviem cipariem aiz komata, kas tika izrēķināta, izmantojot norādītās robežvērtības.

Tab.4.10. Testpiemēri sistēmas prasības "*S3_WORKOUT_LOCALIZATION*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T9_1	1. Atvērt sistēmas treniņa konfigurācijas datni "*application.yml*".	Treniņa konfigurācijā var redzēt visus treniņa soļus pēc kārtas

	2. Pārliecināties, ka treniņa soļos ir norādītas "*title*", "*video*", "*audio*" parametru vērtības angļu ("*en*") un latviešu ("*lv*"; vismaz daļēji, lai varētu pārbaudīt) valodās. 3. Konfigurācijas datnes laukā "*userLanguage*" norādīt "*lv*". 4. Palaist sistēmu. 5. Ar HTTP klientu (piemēram, *Postman*) izpildīt POST pieprasījumu uz "*/client/api/v1/workout*".	atbilstoši treniņa konfigurācijas datnes saturam. Soļu parametri "*title*", "*video*", "*audio*" atbilst konfigurācijas datnē norādītajām latviešu valodas vērtībām vai angļu valodas vērtībām, ja "*lv*" vērtība konkrētajā solī netika norādīta.

Tab.4.11. Testpiemēri sistēmas prasības "*S4_PATIENT_EFFORT_ACCUMULATOR*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T10_1	1. Palaist sistēmu. 2. Atvērt tīmekļa lietotni adresē "*localhost:8080/workout*". 3. Uzstādīt sensoru iestatījumus un noklikšķināt uz pogas "*Submit sensor config*". 4. Noklikšķināt uz pogas "*Start!*".	Sistēma uzsāk apstrādāto datu un informācijas par treniņu pārraidi. Elpošanas treniņa soļos sensoru dati tiek apstrādāti pēc "*Root Mean Square"* metodes. Kustību treniņa soļos sensoru dati tiek apstrādāti pēc "*Moving Average*" metodes.

Tab.4.12. Testpiemēri sistēmas prasības "*S4_MULTICAST*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T11_1	1. Palaist sistēmu. 2. Atvērt tīmekļa lietotni adresē "*localhost:8080/workout*". 3. Uzstādīt sensoru iestatījumus un noklikšķināt uz pogas "*Submit sensor config*". 4. Noklikšķināt uz pogas "*Start!*". 5. Ieslēgt AR lietotni AR brillēs.	Sistēma sāk apstrādāto sensoru datu pārraidi atbilstoši treniņa soļiem. Pārsūtītie dati ir redzami sistēmas Docker konteinera konsolē. AR brilles pieslēdzas datu plūsmai un attēlo treniņa vingrinājumu soļus atbilstoši palaistā treniņa gaitai pārlūkprogrammā – kad kārtējais solis ieslēdzās, AR lietotne uzreiz to attēlo brillēs.

Tab.4.13. Testpiemēri sistēmas prasības
"*S4_MULTICAST_MESSAGE*" testēšanai

Tests	Soļi testpiemēra palaišanai	Sagaidāmais rezultāts
T12_1	1. Palaist sistēmu. 2. Atvērt tīmekļa lietotni adresē "*localhost:8080/workout*". 3. Uzstādīt sensoru iestatījumus un noklikšķināt uz pogas "*Submit sensor config*". 4. Noklikšķināt uz pogas "*Start!*".	Sistēma sāk apstrādāto sensoru datu pārraidi. Pārsūtīto datu formāts Docker konteinera konsolē atbilst formātam *[stepNr, sensorValue, prevSensorValue, maxSensorValue]*, kur *stepNr* ir tekošā treniņa soļa kārtas numurs atbilstoši treniņa konfigurācijai, *sensorValue* ir korekti apstrādāta kārtējā sensora vērtība, *prevSensorValue* ir 1 un *maxSensorValue* ir 1.

Tabulā zemāk (sk. *Tab.4.14.*) ir apkopoti sistēmas prasību manuālās testēšanas rezultāti.

Tab.4.14. Sistēmas "ARehab Portable" prasību manuālās testēšanas rezultāti

Testējamā sistēmas prasība	Testa numurs	Testa rezultāts
S1_SENSOR_ DATA_ ACQUISITION	**T1_1**	✓
	T1_2	✓
	T1_3	✓
S1_PLUX_MOCK	**T2_1**	✓
	T2_2	✓
	T2_3	✓
	T2_4	✓
	T2_5	✓
	T2_6	✓
	T2_7	✓
	T2_8	✓
	T2_9	✓
S2_WORKOUT_CONFIG	**T3_1**	✓
S2_STEPS_TEMPLATES_GROUPS	**T4_1**	✓
	T4_2	✓
S2_WORKOUT_CONFIG_AR	**T5_1**	✓
	T5_2	✓
S2_DEVICES_CONFIG	**T6_1**	✓
S3_WEB_WORKOUT	**T7_1**	✓
	T7_2	✓
	T8_1	✓

S3_SENSOR_CONFIG_WEB	**T8_2**	✓
S3_WORKOUT_LOCALIZATION	**T9_1**	✓
S4_PATIENT_EFFORT_ACCUMULATOR	**T10_1**	✓
S4_MULTICAST	**T11_1**	✓
S4_MULTICAST_MESSAGE	**T12_1**	✓

5. PROJEKTA ORGANIZĀCIJA

Sistēmas “ARehab Portable” izstrādi, dokumentācijas rakstīšanu un sistēmas testēšanu veica darba autore. Projekta, kura ietvaros tika izstrādāta sistēma “ARehab Portable”, plānošanā un īstenošanā piedalījās vairāki cilvēki. Projekts tika īstenots uzņēmumā “Fanout SIA” un paredzēja sistēmas “ARehab Portable” un AR lietotnes izstrādi.

Projekta plānošanā un īstenošanā iesaistītās puses:

- projekta vadītājs (uzņēmuma vadītājs),
- sistēmas pasūtītājs (sistēmas lietotājs),
- sistēmas “ARehab Portable” izstrādātājs (kvalifikācijas darba autore),
- Unity AR lietotnes izstrādātājs.

Sistēmas “ARehab Portable” (turpmāk - sistēma) izstrāde notika saskaņā ar spējās izstrādes metodoloģiju. Sistēmas prasībās tika sākotnēji apspriestas pirms izstrādes sākuma. Lietotājstāstus rakstīja darba autore sadarbībā ar sistēmas pasūtītāju, savukārt sprinta uzdevumus izstrādātājam iedeva projekta vadītājs.

Izstrādes process tika sadalīts četros 2-3 nedēļu ilgos sprintos. Pirms katra sprinta tika organizēta sprinta plānošana, kurā piedalījās darba autore, projekta vadītājs un sistēmas pasūtītājs. Katra sprinta ietvaros trīs reizes nedēļā notika sarunas ar projekta vadītāju un sistēmas pasūtītāju, kuros darba autore stāstīja par izstrādes gaitu, tika uzdoti jautājumi un veikti precizējumi prasībās. Katra sprinta beigās tika organizētas retrospektīvas, kuru gaitā tika apspriesti kārtējā sprinta rezultāti.

Visā sistēmas izstrādes laikā tika organizētas papildus sarunas ar sistēmas pasūtītāju, kurās darba autore veicināja vēlākā sistēma lietotāja izglītošanu darbā ar sistēmu, konfigurācijas datni, tika sniegtas atbildes uz jautājumiem par sistēmas izmantošanu, palaišanas procedūrām un izstrādes gaitu.

Sistēmas izstrādes gaitā tika organizētas tikšanās uzņēmuma ofisā starp darba autori, projekta vadītāju un sistēmas pasūtītāju, uz kuriem sistēmas pasūtītājs ņēma līdzi elpošanas, kustību sensorus, AR brilles un datoru, un notika sistēmas palaišana un testēšana ar sistēmas lietotāja ierīcēm.

Paralēli ar sistēmas “ARehab Portable” izstrādi projekta ietvaros notika AR lietotnes AR brillēm “HoloLens” plānošana un izstrāde. Lietotne no “ARehab Portable” saņem datu plūsmu un attēlo datus AR brillēs. AR lietotnes izstrādi veicināja Unity AR lietotnes

izstrādātājs, ar kuru regulāri tika organizētas sarunas, kuru gaitā tika apspriests pārsūtīto datu formāts un “ARehab Portable” sistēmas piedāvātā saskarne datu iegūšanai.

5.1. Konfigurāciju pārvaldība

Sistēmas “ARehab Portable” apakšsistēmām “Java-Application”, “Etcd-Proxy” un “Media-Proxy” tika izveidoti atsevišķi privāti Docker attēlu repozitoriji, kuru versiju kontrolei tika izmantota sistēma “DockerHub”. Tā ir Docker attēlu versiju kontroles sistēma, kas nodrošina Docker attēlu kopīgošanu un glabāšanu repozitorijos. Katras apakšsistēmas Docker attēla kārtējā versija tika vispirms uzbūvēta lokāli uz darba autores datora un tad tika publicēta “DockerHub” repozitorijā ar atzīmi “*latest*”. Jaunu Docker attēlu versiju publicēšana tika saskaņota ar projekta vadītāju un sistēmas lietotāju. Lai piegādātu jaunu sistēmas versiju līdz sistēmas lietotājam, nekādas papildus procedūras no lietotāja nav nepieciešamas, jo sistēmas palaišanas procedūras iekļauj koda darbināšanu, kas pirms konteineru palaišanas pārbauda, vai datorā ir jaunākās attēlu versijas, un nepieciešamības gadījumā lejupielādē jaunākās apakšsistēmu Docker attēlu versijas no “DockerHub” repozitorijiem.

Apakšsistēmas “Java-Application” pirmkoda versiju kontrolei tika izmantota Git versiju kontroles sistēma un tīmeklī balstīta platforma “GitHub”.

5.2. Kvalitātes nodrošināšana

Izstrādātās sistēmas kvalitātes nodrošināšanai tika veikti sekojoši pasākumi:

- Standartu un programmēšanas principu ievērošana - rakstot apakšsistēmas “Java-Application” kodu programmēšanas valodā Java, tika ievērotas *Clean Code* arhitektūras [9], *SOLID* un *DRY* programmēšanas principi. Līdz ar to, ka sistēmā apstrādāti sensoru un treniņa dati tiek rakstīti un lasīti reāllaikā, sistēmas izstrādē tika ievēroti daudzpavedienu programmēšanas principi sacensību stāvokļu novēršanai. Izstrādātāja mērķis bija rakstīt sevi dokumentējošu, viegli saprotamu kodu, kuru būtu viegli uzturēt. Sistēmas dokumentācijas rakstīšanā tika ievēroti standarti *LVS 68:1996* [3] un *LVS 72:1996* [4];
- Pirmkoda apskates - sistēmas izstrādes gaitā tika organizētas regulāras neformālās tehniskās koda apskates projekta vadītāja vadībā;
- Sistēmas testēšana – sistēma tika vairākkārt darbināta dažādās vidēs, moduļiem tika rakstīti vienībtesti, tika veikta manuālā prasību testēšana, izstrādes laikā veiktās testēšanas rezultātā atklātie defekti tika apkopoti, analizēti un izlaboti;

- Izglītošana – sistēmas izstrādes procesā notika gan izstrādātāja izglītošana prasību vākšanā, programmatūras izstrādē, testēšanā, uzturēšanā un dokumentācijas rakstīšanā, gan sistēmas lietotāja izglītošana darbā ar sistēmu un tās palaišanai un izmantošanai nepieciešamajām tehnoloģijām;
- Risku pārvāldība – plānojot sistēmas izstrādi, tika novērtēti projekta riski saistībā ar klienta gatavību un apņēmību izmantot sistēmu, sistēmas izmantošanas un projekta rezultāta svarīgumu visām iesaistītajām pusēm, sistēmas prasību pilnīgu izpratni, komandas esošajām un nepieciešamajām prasmēm, komandas pieredzi paredzēto sensoru tehnoloģiju izmantošanā. Klients kā lietotāju pārstāvis tika pilnība iesaistīts prasību atrašanā un aprakstīšanā, izstrādes procesa apspriešanā, sistēmas regulārā testēšanā, izglītošanā darbā ar sistēmu.

5.3. Darbietilpības novērtējums

Izstrādātās sistēmas pirmkoda normatīvās darbietilpības noteikšanai tika izmantota komercsabiedrības QSM *(Quantitative Software Management)* uzkrātā informācija par pabeigtiem projektiem. Šī informācija ir apkopota un publiski pieejama QSM etalontabulās [10]. Etalontabulas "*Business Systems Implementation Unit (New and Modified IU) Benchmarks*" pirmā aile rāda informāciju par 25% mazākajiem no 550 jeb pirmās kvartiles projektiem, kas vidēji ilguši 3,2 mēnešus, kuru programmkoda lielums bija starp 131 un 3115 loģiskām rindiņām un kuru lieluma mediāna bija 1889 loģiskās rindiņas. Šādus projektus vidēji izstrādāja 1,57 cilvēki.

Izmantojot rīku CLOC [11], tika noskaidrots, ka sistēmas "ARehab Portable" programmkoda lielums ir 3048 koda rindiņas, neieskaitot tukšas rindiņas un komentārus. Ņemot vērā programmēšanas stilu, tika noskaidrots, ka sistēmas "ARehab Portable" izstrādātā programmkoda loģisko rindiņu skaits varētu būt starp 2200 un 2600 rindiņām, kas ir tuvu QSM etalontabulā aprakstīto pirmās kvartiles projektu lieluma mediānai. Līdz ar to var secināt, ka izstrādātā programmkoda normatīvā darbietilpība sasniedz 3 personmēnešus, kas ir kvalifikācijas darba prasībās norādītais minimālais apjoms darba ietvaros izstrādātajām programmatūras produktam.

REZULTĀTI UN DISKUSIJA

Darba rezultātā tika izstrādāta strādājošā reāllaika sistēma sensoru datu iegūšanai, apstrādei, pārraidei un pacientu treniņu veidošanai, palaišanai un pārvaldei. Izstrādātā sistēma atbilst izvirzītajām prasībām, kā arī tika izstrādāta sistēmai atbilstošā dokumentācija.

Sistēma tiek plaši pielietota darbā ar rehabilitācijas pacientiem. Projekta ietvaros notiek sistēmas "ARehab Portable" pasūtītāja organizētās demonstrācijas fizikālās medicīnas un rehabilitācijas ārstiem, tiek veikta zinātniska izpēte par sensoru un AR tehnoloģiju izmantošanu pacientu motivācijas, pašapziņas un progresa uzlabošanai. Projekta ietvaros notiek virtuālās realitātes lietotnes izstrādes plānošana ar nolūku veikt turpmāku sistēmas "ARehab Portable" testēšanu ar divām pacientu grupām – pacienti ar paplašinātās realitātes brillēm un pacienti ar virtuālās realitātes brillēm; testēšanas mērķis ir salīdzināt rezultātus atkarībā no briļļu veida, saprast katra veida priekšrocības un trūkumus un izprast lietotāju pieredzi.

Turpmākajos plānos saistībā ar sistēmu "ARehab Portable" ietilpst tīmekļa lietotnes funkcionalitātes paplašināšana, medicīnas sensoru izmantošana sistēmā pulsa, sirdsdarbības, skābekļa saturācijas mērījumiem un mākslīgā neironu tīkla izstrāde pacientu atpazīšanai pēc iegūtiem datiem.

SECINĀJUMI

Kvalifikācijas darba izstrādes laikā tika iegūta vērtīga pieredze programmēšanā valodās Java un Lua, Spring un JUnit satvaru pielietošanā, darbā ar Gradle, Docker, Docker attēliem, konteineriem, sējumiem, komponēšanu, Nginx, Etcd, izstrādes vidi IntelliJ IDEA, pārraides vadības protokolu, Linux un MacOS komandrindām. Darba izstrādes laikā tika pilnveidotas prasmes lietotājstāstu rakstīšanā, sistēmas plānošanā un projektēšanā, risinājumu izvēlē, testēšanā, atkļūdošanā, darbā saskaņā ar spējās izstrādes metodoloģiju, koda rakstīšanā saskaņā ar labo programmēšanas praksi, ievērojot *Clean Code*, *SOLID* un *DRY* programmēšanas principus. Kvalifikācijas darba izstrādes laikā tika iegūta vērtīga pieredze darbā ar sistēmas lietotāju un sistēmas lietotāja izglītošanā. Darba rezultātā tika izstrādāta aktīvi izmantojamā sistēma. Darba izstrādes laikā darba autore pilnveidoja spēju pielietot iegūtās teorētiskās zināšanas praktiskajā darbā, prasmi plānot savu darbu un prasmi rakstīt sistēmai atbilstošu dokumentāciju.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Informācija par pētījumu "Design research for user-friendly guidance of complex whole-body rehabilitation for lower extremity amputees by means of extended reality and advanced wearables data processing" [tiešsaiste] – [atsauce 30.12.2021.] Pieejams: https://va.lv/en/research/research/design-research-user-friendly-guidance-complex-whole-body-rehabilitation-lower
2. PLUX wireless biosignals [tiešsaiste] – [atsauce 06.01.2022.] Pieejams: https://plux.info/
3. Programmatūras prasību specifikāciju ceļvedis, *LVS 68:1996, 1996*.
4. Ieteicamā prakse programmatūras projektējuma aprakstīšanai, *LVS 72:1996*.
5. Lietojumprogrammas "OpenSignals" dokumentācija "TCP/IP Module Guide" [tiešsaiste] – [atsauce 06.01.2022.] Pieejams: http://biosignalsplux.com/downloads/apis/OpenSignals_TCP_IP_Manual.pdf
6. Etcd Docker attēls "*bitnami/etcd*" [tiešsaiste] – [atsauce 06.01.2022.] Pieejams: https://hub.docker.com/r/bitnami/etcd/
7. Lua Etcd bibliotēka priekš OpenResty [tiešsaiste] – [atsauce 06.01.2022.] Pieejams: https://github.com/api7/lua-resty-etcd
8. Lua 5.4 Reference Manual [tiešsaiste] – [atsauce 06.01.2022.] Pieejams: https://www.lua.org/manual/5.4/
9. Martin, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. Upper Saddle River, NJ: Prentice Hall, 2009.
10. QSM Benchmark Tables [tiešsaiste] – [atsauce 06.01.2022.] Pieejams: http://www.qsm.com/resources/qsm-benchmark-tables
11. CLOC rīks [tiešsaiste] – [atsauce 06.01.2022.] Pieejams: http://cloc.sourceforge.net/

PIELIKUMI

1. pielikums – klases “**Workout**” koda fragments

```
@RequiredArgsConstructor
// inner class for workout config parsing logic
public static class WorkoutConfigParser {
    private final WorkoutConfig config;
    private final IdSequence idSequence = new IdSequence();

  // inner class that sets ids and sequence numbers for exercises and steps
    public static class IdSequence {
        private int exerciseId = 1;
        private int exerciseOrder = 0;
        private int stepId = 1;
        private int stepOrder = 0;
        public int nextExerciseId() {
            return exerciseId++;
        }
        public int nextExerciseOrder() {
            return exerciseOrder++;
        }
        public int nextStepId() {
            return stepId++;
        }
        public int nextStepOrder() {
            return stepOrder++;
        }
    }

    // method parses all exercises and steps and returns a workout instance
    public Workout parse(String userLang) {
        Set<Exercise> exercises = new TreeSet<>();
        Set<Step> steps = new TreeSet<>();
        for (WorkoutConfig.Exercise exercise : config.getWorkout()) {
            int exerciseId = idSequence.nextExerciseId();
            int exerciseOrder = idSequence.nextExerciseOrder();
            Set<Step> parsedSteps = exercise.getSteps()
                    .stream()
                    .map(s -> WorkoutConfigParser.this.parseStep(s,
exerciseId, userLang))
                    .flatMap(Collection::stream)
                    .collect(Collectors.toCollection(TreeSet::new));
            // adds parsed exercises and steps to corresponding tree sets
            exercises.add(new Exercise(exerciseId, exerciseOrder,
exercise.getTitle()));
            steps.addAll(parsedSteps);
        }
        return new Workout(exercises, steps);
    }

 // method parses one step at a time and returns a parsed step as a
singleton list
    private List<Step> parseStep(WorkoutConfig.Step step, int exerciseId,
String lang) {
```

```
        // stores step template
        WorkoutConfig.Step overrides = new WorkoutConfig.Step();
        if (step.getApplyTemplate() != null) {
            overrides = getTemplate(step.getApplyTemplate());
        }
        // applies a step group if applyGroup is present
        if (step.getApplyGroup() != null) {
            return applyGroup(step.getApplyGroup(), exerciseId, lang);
        }
        // if step has a step template, its values are applied
        return Collections.singletonList(
                new Step.StepBuilder()
                        .id(idSequence.nextStepId())
                        .order(idSequence.nextStepOrder())
                        .exerciseId(exerciseId)
                        .title(Optional.ofNullable
                                        (overrides.getTitle(lang))
                                .orElse(step.getTitle(lang)))
                        .graph(Optional.ofNullable
                                        (overrides.getGraph())
                                .orElse(Optional.ofNullable
                                                (step.getGraph())
                                        .orElse(false)))
                        .effortType(Optional.ofNullable
                                        (overrides.getPluxSource())
                                .map(PatientEffortType::getByName)
                                .orElse(Optional.ofNullable
                                                (step.getPluxSource())
                                        .map(PatientEffortType::getByName)
                                        .orElse(null)))
                        .duration(Optional.ofNullable
                                        (overrides.getDuration())
                                .orElse(Optional.ofNullable
                                                (step.getDuration())
                                        .orElse(-1f))) // manual transition
                        .video(Optional.ofNullable
                                        (overrides.getVideo(lang))
                                .orElse(step.getVideo(lang)))
                        .audio(Optional.ofNullable
                                        (overrides.getAudio(lang))
                                .orElse(step.getAudio(lang)))
                .build());
    }

    // method finds step template by template name string
    private WorkoutConfig.Template getTemplate(String templateName) {
        return config.getTemplates()
                .stream()
                .filter(t -> t.getName().equals(templateName))
                .findFirst()
                .orElseThrow(() -> new RuntimeException(
                        String.format("Template %s not found",
templateName)));
    }
```

```java
    // method finds step groups by group name string,
    // applies step group and returns a list of parsed steps
    private List<Step> applyGroup(String groupName, int exerciseId, String
lang) {
        WorkoutConfig.Group group = config.getGroups()
                .stream()
                .filter(t -> t.getName().equals(groupName))
                .findFirst()
                .orElseThrow(() -> new RuntimeException(
                        String.format("Group %s not found", groupName)));
        return group.getSteps()
                .stream()
                .flatMap(s -> this.parseStep(s, exerciseId, lang).stream())
                .collect(Collectors.toList());
    }
}
```

2. pielikums – klases “**PatientActivityTcpMulticast**” koda fragments

```java
@Slf4j
@RequiredArgsConstructor
public class PatientActivityTcpMulticast {
    private final PatientActivityMessageBroker messageBroker;
    private final PatientActivityStateManager activityStateManager;
    // port number from config file
    private final int multicastPort;
    //  duration depends on multicastTicksPerSecond from config file
    private final int multicastTickDurationMS;
    private boolean started = false;

    // method launches TCP multicast
    public void start() {
        // must be started exactly once
        if (started) {
            throw new RuntimeException("Multicast is already stared");
        }
        started = true;

        // attempts to create multicast server socket
        final ServerSocket multicastSocket;
        try {
            multicastSocket = new ServerSocket(multicastPort);
            log.info("Multicast server started on port {}", multicastPort);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

        // thread takes last calculated value, composes a multicast message
and published it to clients
        new Thread(() -> {
            for (;;) {
                AccumulatedPatientEffort lastAccumulatedEffort =
activityStateManager.getLastAccumulatedEffort();
                PatientActivityMessageImpl message = new
```

```
PatientActivityMessageImpl(
                        lastAccumulatedEffort.getStepId(),
lastAccumulatedEffort.getAccumulatedValue());
                messageBroker.publish(message);
                // logs sent multicast message if workout has been started
                if (message.getStepId() != -1) {
                    log.info(message.getAsString());
                }
                try {
                    Thread.sleep(multicastTickDurationMS);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }).start();
        // starts a thread for managing client connections
        new Thread(new PatientActivityTcpClientHandler(multicastSocket,
messageBroker)).start();
    }
}
```

3. pielikums – klases "**PluxAdapter**" koda fragments

```
@RequiredArgsConstructor
@Slf4j
public class PluxAdapter {
    // plux hostname from config file
    private final String hostname;
    // plux port from config file
    private final int port;
    // plux connection and data callbacks
    private final PluxSensorDataCallback sensorDataCallback;
    private final PluxConnectionEstablishedCallback
connectionEstablishedCallback;
    private final PluxConnectionFailedCallback connectionFailedCallback;
    // plux devices list
    private final BitalinoDeviceRegistry deviceRegistry;

    private boolean started = false;

    // method launches plux adapter
    public void start() {
        // can be started only once
        if (started) {
            throw new RuntimeException("Only single connection is
supported");
        }
        started = true;
        // thread manages plux tcp connection and logs corresponding
messages
        new Thread(() -> {
            PluxTcpConnection connection = new PluxTcpConnection();
            PluxOutputCallback outputCallback = new
PluxOutputCallback((PluxDeviceRegistry) deviceRegistry,
sensorDataCallback);
```

```
            int tries = 0;
            for (;;) {
                if (!connection.isActive()) {
                    if (tries++ == 0) {
                        log.info("Connecting to Plux...");
                    }
                    boolean connected = connection.establish(hostname,
port, outputCallback);
                    if (connected) {
                        log.info("Plux connection established");
                        connectionEstablishedCallback.onEstablished();
                        // sends plux command that starts sensor data
acquisition
                        connection.sendCommand(() -> "start");
                    } else {
                        log.info("Plux connection failed. Retrying in
1s...");
                        connectionFailedCallback.onFail();
                    }
                }
                // pause for 1s
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }).start();
    }
}
```

4. pielikums – klases “**SensorConfigManager**” koda fragments

```
public class SensorConfigManager {
    // default null sensor config
    private SensorConfig config = new DefaultConfigImpl();

    public synchronized SensorConfig getConfig() {
        return config;
    }

    public synchronized void setConfig(SensorConfig config) {
        this.config = config;
    }

    // method returns relative value according to sensor config
    public synchronized double inboundByType(PatientEffortType effortType,
PluxSensorData sensorData) {
        Integer channelIndex = config.getChannelIndex(effortType);
        Bounds bounds = config.getBounds(effortType);
        // config must be submitted before multicast is started
        if (null == channelIndex || null == bounds) {
            throw new IllegalStateException("No config");
        }
        // gets raw sensor value from corresponding channel number
        double original = sensorData.getChannelValue(channelIndex);
```

```
        Double lower = bounds.getLower();
        Double upper = bounds.getUpper();
        if (null == lower || null == upper) {
            return original;
        }
        if (original < lower) {
            return 0;
        }
        if (original > upper) {
            return 1;
        }
        return (original - lower) / (upper - lower); // relative value
    }
}
```

5. pielikums – klases "**WebSocketBroker**" koda fragments

```
@Controller
@RequiredArgsConstructor
public class WebSocketBroker {
    private final WorkoutManager workoutManager;
    private final SensorConfigManager sensorConfigManager;
    private final SimpMessagingTemplate wsTemplate;
    private boolean started = false;

    // listener method manages workout state update callbacks
    // sends messages upon state changes
    public void listen() {
        // must be started exactly once
        if (started) {
            throw new RuntimeException("WebSocket broker already started");
        }
        started = true;

        // registers anonymous callback implementation
    workoutManager.registerUpdateCallback(new WorkoutStateUpdatedCallback()
{
            @Override
            public void onWorkoutStarted() {
                // sets workoutInProgress to true
                    wsTemplate.convertAndSend("/topic/workout-progress",
                        new OutgoingWorkoutStateMessage(true));
            }
            @Override
            public void onWorkoutFinished() {
                // sets workoutInProgress to false
                wsTemplate.convertAndSend("/topic/workout-progress",
                        new OutgoingWorkoutStateMessage(false));

            }
            @Override
// sends new step id, previous step id (if present) and true if current is
manual transition
           public void onStepSwitched(Step currentStep, Step previousStep) {
```

```
                wsTemplate.convertAndSend("/topic/step-switch",
                        new OutgoingStepSwitchMessage(currentStep.getId(),
                        Optional.ofNullable(previousStep)
                       .map(Step::getId)
                       .orElse(-1), !currentStep.hasDuration()));
            }
            @Override
// sends step progress percentage
            public void onStepProgress(Step step, double progress) {

                wsTemplate.convertAndSend("/topic/step-progress",
                        new OutgoingStepProgressMessage(step.getId(), (int)
(100 * progress)));
            }
        });
    }

// methods receive action messages and trigger workout and sensor config
managers accordingly
    @MessageMapping("/workout-progress")
    public void workoutStateMessageAction(IncomingWorkoutStateMessage
message) {
        // starts if workout in progress
        if (message.isWorkoutInProgress()) {
            workoutManager.startWorkout();
        } else {
            workoutManager.stopWorkout();
        }
    }

    @MessageMapping("/step-switch")
    public void stepSwitchMessageAction(IncomingStepSwitchMessage message)
{
        workoutManager.switchToStep(message.getStepId());
    }

    @MessageMapping("/update-sensor-config")
    public void
sensorConfigUpdateMessageAction(IncomingSensorConfigUpdateMessage message)
{
        sensorConfigManager.setConfig(SensorConfigImpl.of(message));
    }
}
```

6. pielikums – klases "**PatientActivityPluxConnector**" koda fragments

```
@Slf4j
@RequiredArgsConstructor
public class PatientActivityPluxConnector {

    private final PatientActivityStateManager activityStateManager;
    private final PatientEffortCalculator patientEffortCalculator;
    private final WorkoutManager workoutManager;
    // plux devices list
    private final BitalinoDeviceRegistry deviceRegistry;
```

```
    // hostname from config file
    private final String pluxHostname;
    // port from config file
    private final int pluxPort;

    private boolean started = false;

    // method starts TCP connector with callbacks
    public void start(PluxConnectionEstablishedCallback
connectionEstablishedCallback,PluxConnectionFailedCallback
connectionFailedCallback) {
        // must be started exactly once
        if (started) {
            throw new RuntimeException("Plux connector already started");
        }
        started = true;
        // anonymous callback implementations
        PluxSensorDataCallback sensorDataCallback = new
PluxSensorDataCallback() {
            @Override
            public void onSensorData(PluxSensorData sensorData) {
                BitalinoDevice device =
deviceRegistry.get(sensorData.getDeviceId());
 // ignores until step effort type corresponds with any of supported device
types
                workoutManager.getCurrentStep().ifPresent(step -> {
                    if (step.getEffortType() != null &&
!device.supports(step.getEffortType())) {
                        return;
                    }
                    // calculates and sets processed plux value
                    PluxPatientEffort effort =
patientEffortCalculator.calculate(step, sensorData);
                    activityStateManager.setLastPluxEffort(effort);
                });
            }
        };
        PluxConnectionFailedCallback connectionFailedLocalCallback = new
PluxConnectionFailedCallback() {
            @Override
            public void onFail() {
                activityStateManager.setLastPluxEffort(new
NullPatientEffort());
                connectionFailedCallback.onFail();
            }
        };

        // creates and starts plux adapter
        PluxAdapter pluxAdapter = new PluxAdapter(pluxHostname, pluxPort,
sensorDataCallback,
                connectionEstablishedCallback,
connectionFailedLocalCallback, deviceRegistry);
        pluxAdapter.start();
    }
}
```

```
// runnable inner class that implements anonymous command listener,
// generates and sends random sensor data to clients
private static class PluxStream extends Thread {
    // tick number display format
    private static final DecimalFormat TICK_FORMAT = new
DecimalFormat("0.0");
    // plux channel value display format
    private static final DecimalFormat CHANNEL_FORMAT = new
DecimalFormat("0.00000");

    private final Queue<Client> clients;
    // list of plux devices MAC addresses
    private final List<String> deviceAddresses;
    // default data acquisition logical value
    private final AtomicBoolean acquiring = new AtomicBoolean(false);
    // default binary plux channel mask
    private final AtomicInteger channels = new AtomicInteger(0b11111111);
    // default plux sleep time in ms
    private final AtomicInteger sleep = new AtomicInteger(1000);
    private long tick = 0;

    public PluxStream(Queue<Client> clients, PluxCommandBus commandBus,
List<String> deviceAddresses) {
        this.clients = clients;
        this.deviceAddresses = deviceAddresses;
        // registers anonymous implementation of command listener for each
device
        for (int i = 0; i < deviceAddresses.size(); ++i) {
            String macAddress = deviceAddresses.get(i);
            // command listener changes values and logs action info
            commandBus.registerListener(macAddress, i, new
PluxCommandListener() {
                @Override
                public void onStartAcquisition() {
                    acquiring.set(true);
                    System.out.println("Acquisition started");
                }
                @Override
                public void onStopAcquisition() {
                    acquiring.set(false);
                    System.out.println("Acquisition stopped");
                }
                @Override
                public int onGetActiveChannels() {
                    System.out.println(macAddress + ": active channels
requested");
                    return channels.get();
                }
                @Override
                public void onSetActiveChannels(int mask) {
                    channels.set(mask);
                    System.out.println(macAddress + ": active channels
updated");
                }
```

```
                    @Override
                    public int onGetFrequency() {
                        System.out.println(macAddress + ": frequency
requested");
                        return 1000 / sleep.get();
                    }
                    @Override
                    public void onSetFrequency(int frequency) {
                        sleep.set(1000 / frequency);
                        System.out.println(macAddress + ": frequency updated");
                    }
                });
            }
        }

    @Override
    // runnable method that generates arbitrary sensor data
    public void run() {
        StringBuilder data = new StringBuilder();
        while (true) {
            // pause if no acquisition in progress
            if (!acquiring.get()) {
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                continue;
            }
            data.setLength(0);
            // formatting
            data.append("{\"returnCode\": 0, \"returnData\": {");
            for (int i = 0; i < deviceAddresses.size(); ++i) {
                if (i != 0) {
                    data.append(", ");
                }
                data.append("\"" + deviceAddresses.get(i) + "\": [");
                for (int j = 0; j < 150; ++j) {
                    if (j != 0) {
                        data.append(", ");
                    }
                    data.append("[");
                    // gets next tick number
                    data.append(TICK_FORMAT.format(++tick));
                    // checks if each channel in the binary mask is set to
active
                    // if it is active, appends random sensor value at its
position
                    if (((channels.get() >> 8) & 1) == 1) {
                        data.append(",
").append(CHANNEL_FORMAT.format((Math.sin((Math.PI / 200) * tick) + 1) /
2));
                    }
                    if (((channels.get() >> 7) & 1) == 1) {
                        data.append(",
```

```
").append(CHANNEL_FORMAT.format((Math.sin((Math.PI / 200) * tick + 100) +
1) / 2));
                    }
                    if (((channels.get() >> 6) & 1) == 1) {
                        data.append(",
").append(CHANNEL_FORMAT.format((Math.sin((Math.PI / 200) * tick + 200) +
1) / 2));
                    }
                    if (((channels.get() >> 5) & 1) == 1) {
                        data.append(",
").append(CHANNEL_FORMAT.format((Math.sin((Math.PI / 200) * tick + 300) +
1) / 2));
                    }
                    if (((channels.get() >> 4) & 1) == 1) {
                        data.append(",
").append(CHANNEL_FORMAT.format((Math.sin((Math.PI / 200) * tick) + 1) /
2));
                    }
                    if (((channels.get() >> 3) & 1) == 1) {
                        data.append(",
").append(CHANNEL_FORMAT.format((Math.sin((Math.PI / 200) * tick + 100) +
1) / 2));
                    }
                    if (((channels.get() >> 2) & 1) == 1) {
                        data.append(",
").append(CHANNEL_FORMAT.format((Math.sin((Math.PI / 200) * tick + 200) +
1) / 2));
                    }
                    if (((channels.get() >> 1) & 1) == 1) {
                        data.append(",
").append(CHANNEL_FORMAT.format((Math.sin((Math.PI / 200) * tick + 300) +
1) / 2));
                    }
                    data.append("]");
                }
                data.append("]");
            }
            data.append("}}");
            String toSend = data.toString();
            // sends data block
            for (Client client : clients) {
                client.send(toSend);
            }
            // pause
            try {
                Thread.sleep(sleep.get());
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}
```

Kvalifikācijas darbs „***Sensoru datu apstrādes un fiziskās aktivitātes analīzes sistēma fizikālās medicīnas un rehabilitācijas ārstiem***” izstrādāts Latvijas Universitātes Datorikas fakultātē.

Ar savu parakstu apliecinu, ka darbs izstrādāts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: ***Jekaterina Jevtejeva*** __________________ **__.01.2022**.

Rekomendēju darbu aizstāvēšanai

Darba vadītāja: ***Dr.dat. Vineta Arnicāne*** _____________ **__.01.2022.**

Recenzents: ***Dr.phys. Modris Bērzonis***

Darbs iesniegts 10.01.2022.

Kvalifikācijas darbu pārbaudījumu komisijas sekretārs: ________________________

Darbs aizstāvēts kvalifikācijas darbu pārbaudījuma komisijas sēdē

___.01.2022. prot. Nr. __________

Komisijas sekretārs(-e): ____