VAASAN YLIOPISTO

TEKNIIKAN JA INNOVAATIOJOHTAMISEN YKSIKKÖ

OHJELMISTOTEKNIIKKA

Janne Vierula

PORTIT JA ADAPTERIT -ARKKITEHTUURIN SOVELTAMINEN KETTERÄSSÄ OHJELMISTOKEHITYKSESSÄ

Diplomityö. joka on jätetty tarkastettavaksi diplomi-insinöörin tutkintoa varten Vaasassa 9.4.2019.

Työn valvoja Professori Jouni Lampinen

Työn ohjaaja DI Arvi Lehesvuo

SISÄLLYSLUETTELO **sivu**

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Janne Vierula

Diplomityön nimi:	Portit ja adapterit -arkkitehtuurin soveltaminen ketterässä ohjelmistokehityksessä
Valvojan nimi:	Professori Jouni Lampinen
Ohjaajan nimi:	DI Arvi Lehesvuo
Tutkinto:	Diplomi-insinööri (DI)
Ohjelma:	Tietotekniikan koulutusohjelma
Suunta:	Ohjelmistotekniikka
Opintojen aloitusvuosi:	2012
Opintojen valmistumisvuosi:	2019 **Sivumäärä:** 69

TIIVISTELMÄ:

Ketterien menetelmien käyttö ohjelmistokehityksessä on nykypäivän trendi. Portit ja adapterit -arkkitehtuuria suositellaan käytettäväksi ketterässä ohjelmistokehityksessä, koska sen käyttö parantaa ohjelmiston ylläpidettävyyttä, joka on tärkeää ketterässä ohjelmistokehityksessä. Tutkimuksessa pyritään todentamaan portit ja adapterit -arkkitehtuurin soveltuvuutta ketterään ohjelmistokehitykseen.

Tutkimus suoritettiin tapaustutkimuksena analysoimalla PC-työkalu projektin Business-logiikan toteutusta Scrum-käyttäjätarinoiden kautta. Projektin tavoitteena oli saada toimitettua kohdeyritykselle asiakasvaatimukset täyttävä PC-työkalu taajuusmuuttajan toiminnallisen turvallisuuden konfigurointiin. Projektissa käytettiin ketterien ohjelmistokehitysmenetelmien käytäntöjä ja PC-työkalun Business-logiikan arkkitehtuurina käytettiin portit ja adapterit -arkkitehtuuria. Business-logiikan tarjoaman rajapinnan formaattina käyttöliittymälle käytettiin JavaScript Object Notation -formaattia. Business-logiikka toteutettiin C#-ohjelmointikielellä.

Projektin tulokseksi saatiin asiakasvaatimukset täyttävä PC-työkalu taajuusmuuttajan toiminnallisen turvallisuuden konfigurointiin. Tutkimuksen aikana todettiin, että portit ja adapterit -arkkitehtuurin soveltuu käytettäväksi ketterässä ohjelmistokehityksessä. Portit ja adapterit -arkkitehtuurin soveltuvuus ketterään ohjelmistokehitykseen johtuu osittain siitä, että portit ja adapterit -arkkitehtuurista löytyy yhtenäisyyksiä suosittuihin ketterässä ohjelmistokehityksessä käytettyihin suunnitteluperiaatteisiin vakaat riippuvuudet ja riippuvuusinversio. Vakaat riippuvuudet ja riippuvuusinversio -suunnitteluperiaatteiden käyttö parantaa muunneltavuutta, joka on ylläpidettävyyden komponentti.

AVAINSANAT: Toiminnallinen turvallisuus, ketterät menetelmät, Ylläpidettävyys, Portit ja Adapterit -arkkitehtuuri

UNIVERSITY OF VAASA
The School of Technology and Innovations
Author: Janne Vierula

Topic of Thesis:	Applying the ports and adapters -architecture to agile software development
Supervisor:	Professor Jouni Lampinen
Instructor:	M.Sc. (Tech.) Arvi Lehesvuo
Degree:	Master of Science in Technology
Degree Programme:	Degree Programme in Information Technology
Option:	Software Engineering
Year of Entering the University:	2012
Year of Completing the Thesis:	2019 **Pages:** 69

ABSTRACT:

Agile software development is a trend nowadays. Usage of Ports and Adapters -architecture is recommended because ports and adapters architecture improves maintainability of the software. This study attempts to verify suitability of ports and adapters -architecture in agile software development.

Study was conducted as a case-study by analyzing implementation of business logic of a PC-tool trough scrum user stories. Target of the project was to deliver a PC-tool for customer which could be used to configure the functional safety of a frequency converter. Agile software development practices were utilized in the project and ports and adapters architecture was selected as architecture for the business logic of the PC-tool. JavaScript Object Notation -format was used in the Interface between the user interface and the business logic. Business logic was implemented using C#-programming language.

Result of the project was customer requirement fulfilling PC-tool which can be used to configure the functional safety of a frequency converter. During the study it was found out that ports and adapters -architecture is suitable to be used in agile software development. Study showed that Suitability of ports and adapters -architecture to agile software development can partly be explained because ports and adapters architecture has similarities to popular agile design principles called Stable Dependencies and Dependency Inversion. Usage of Stable Dependencies and Dependency Inversion design principles increases changeability which is a component of maintainability.

KEYWORDS: Functional safety, Agile software development, Maintainability, Ports and adapters -architecture

1 JOHDANTO

Perinteinen tapa kontrolloida laitteiden, kuten pumppujen ja tuulettimien nopeuksia sähkömoottorin avulla on käyttää vaihteistoa. Vaihteiston käyttö nopeuden kontrolloimiseen on kuitenkin epätarkkaa. Viime aikoina taajuusmuuttajat ovat korvanneet perinteiset tavat kontrolloida laitteiden pyörimisnopeuksia teollisuudessa, koska taajuusmuuttajat ovat energiatehokkaampia, huoltovapaampia ja taajuusmuuttajiin integroitu toiminnallinen turvallisuus vähentää järjestelmän monimutkaisuutta. Toiminnallinen turvallisuus on tärkeää esimerkiksi elintarviketuotannossa, jossa sen tehtävänä varmistaa, että pyörivä laite pysäytetään hallitusti vaaratilanteen sattuessa, esimerkiksi varmistaa, että sokerin tuotannossa käytettävä linko pysäytetään hallitusti siinä tapauksessa, että laitteen käyttäjä tarttuu pyörivään linkoon. (ABB 2016a.)

Ketterä ohjelmistokehitys on nykypäivän trendi, jolla pyritään vähentämään perinteisten ohjelmistokehitysmenetelmien ongelmia suoriutua muuttuvista vaatimuksista. Robertin (Martin 2003: 85) mukaan ketterä ohjelmistokehitys vaatii, että koodi on helposti ylläpidettävää. Ylläpidettävyys koostuu IEC 25023 (ISO/IEC 25023 2016: 25–27) mukaan analysoitavuudesta, testattavuudesta, modulaarisuudesta, muunneltavuudesta ja uudelleenkäytettävyydestä. Alistairin mukaan (Cockburn 2007: 275) portit ja adapterit -arkkitehtuuria suositellaan käytettäväksi ketterissä ohjelmistoprojekteissa, koska se parantaa ohjelmiston testattavuutta. Mahdollisuus tutkia portit ja adapterit -arkkitehtuurin soveltuvuutta ketterään ohjelmistokehitykseen käytännön työelämässä herätti mielenkiinnon paneutua portit ja adapterit -arkkitehtuurin ja ylläpidettävyyden suhteeseen. Työssä pyritään todentamaan portit ja adapterit -arkkitehtuuri soveltuvuutta ketterään ohjelmistokehitykseen tapaustutkimuksen kautta.

Tapaustutkimuksen materiaalina käytettiin projektia, jonka aiheena oli taajuusmuuttajan kanssa kommunikoivan PC-työkalun toteutus Windows ympäristöön. PC-työkalun käyttötarkoitus oli taajuusmuuttajan toiminnallisen turvallisuuden konfigurointi. Toiminnallisen turvallisuuden konfiguroinnilla tarkoitetaan IEC 61800-5-2 määritettyjen turva-

funktioiden parametrien muokkaamista ja turvafunktioiden parametrien arvojen siirtämistä taajuusmuuttajalle. Toteutettu PC-työkalu voidaan jakaa kolmeen osaan, jotka ovat Käyttöliittymä, Business-logiikka ja Kommunikaatiokirjasto. Projektissa käytettiin ketteriä ohjelmistokehitys menetelmiä ja PC-työkalun arkkitehtuurina käytettiin portit ja adapterit -arkkitehtuuria. Tapaustutkimuksen materiaalina käytettiin projektissa C#-ohjelmointikielellä toteutettua PC-työkalun business-logiikkaa.

Kappaleessa 2 käsitellään taajuusmuuttajiin liittyvää toiminnallista turvallisuutta. Kappaleessa 3 käsitellään ketteriä menetelmiä ohjelmistokehityksessä. Kappaleessa 4 käsitellään työn kannalta keskeisimpiä suunnittelumalleja, suunnitteluperiaatteita ja arkkitehtuureja. kappaleessa 5 käydään läpi projektin lähtötilanne ja tutkimusmenetelmä. Kappaleissa 6 käydään läpi PC-työkalun käyttäjätarinoiden toteutuksia ja analysoidaan käyttäjätarinoiden tuloksiksi saatujen komponenttien ylläpidettävyyttä. Kappaleessa 7 analysoidaan tutkimuksen tulosten paikkaansa pitävyyttä vertaamalla tuloksia teoriaan, sekä käydään läpi mahdollisia jatkotutkimus ideoita.

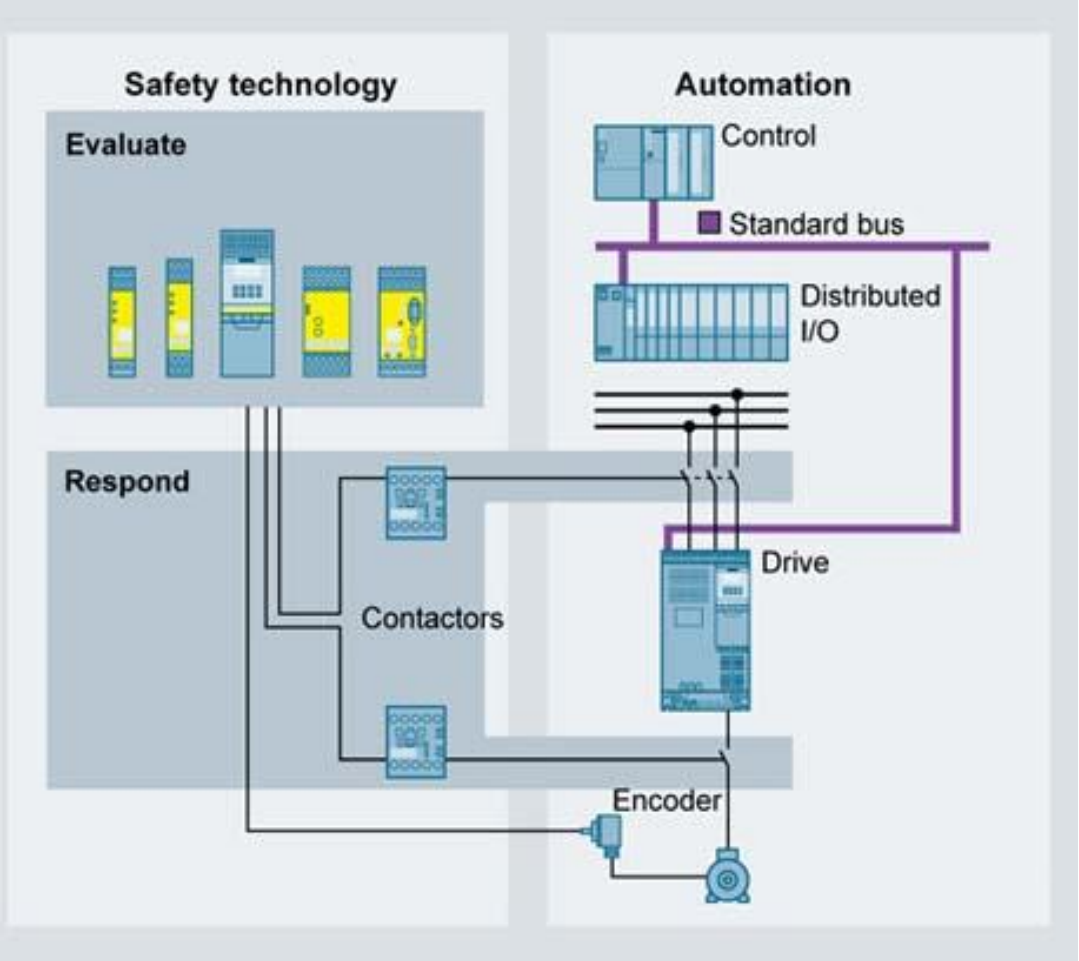
2 TAAJUUSMUUTTAJAT

Perinteinen tapa kontrolloida esimerkiksi ilmanvaihtokonetta sähkömoottoria käyttäen on pyörittää moottoria vakionopeudella ja säätää ilman virtausnopeutta mekaanisella kuristimella. Perinteinen tapa ei ole energiatehokas, koska täyttä ilmavirtausnopeutta tarvitaan vain 1–5 % käyttöajasta. Taajuusmuuttajan avulla sähkömoottoria pyöritetään aina vaaditulla nopeudella, jolloin energiakustannukset pienenevät ja moottorin elinikä kasvaa. (Piper 2009.)

2.1 Toiminnallinen turvallisuus

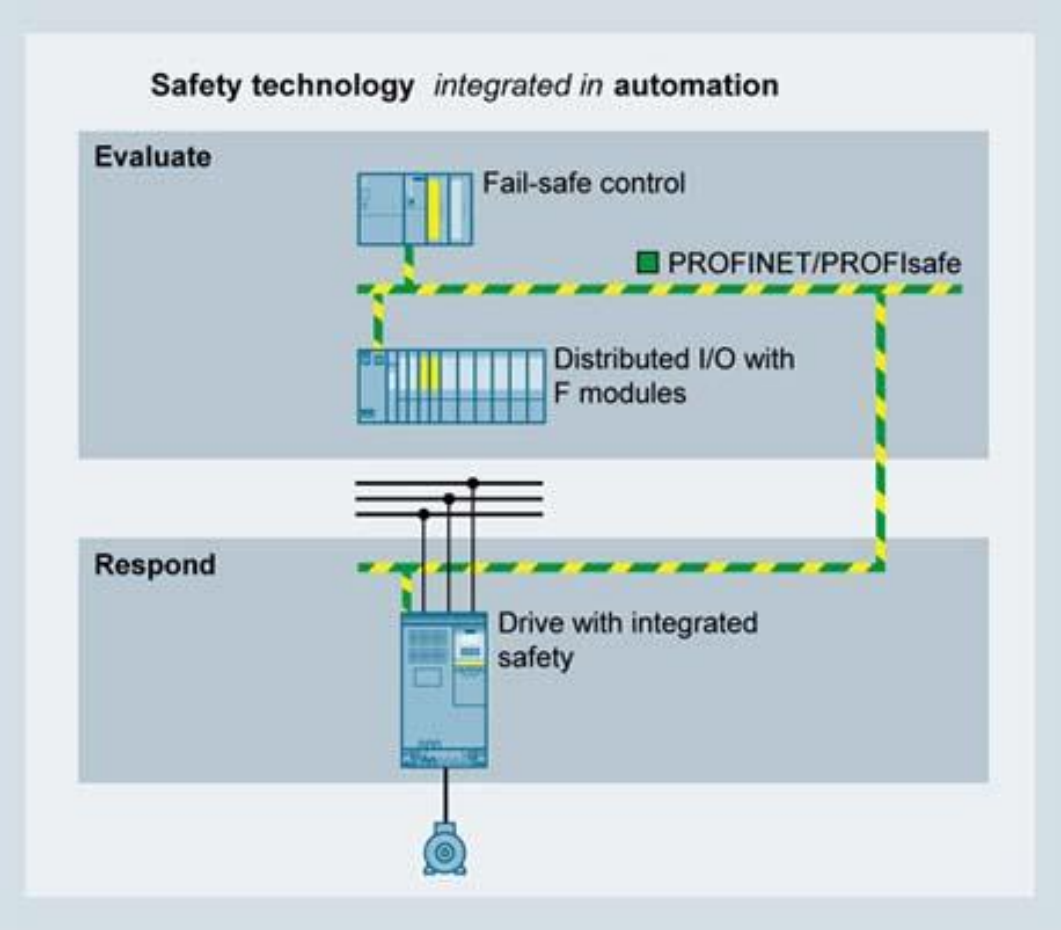
Toiminnallisen turvallisuuden tehtävä on tunnistaa vaaratilanne ja estää vaaratilanteesta aiheutuva vahinko aktivoimalla korjaava tai suojaava laite. Toiminnallinen turvallisuus tuotetaan aktiivisen järjestelmän avulla, esimerkiksi palo-ovea ei luokitella aktiiviseksi järjestelmäksi. Automaattinen tulen sammutus järjestelmä, joka tunnistaa tulipalon savun perusteella ja aktivoi sprinklerit palon sammuttamiseksi luokitellaan aktiiviseksi järjestelmäksi. (IEC 2016.)

Perinteisissä moottorinohjaussovelluksissa toiminnallinen turvallisuus on toteutettu taajuusmuuttajan ulkopuolelle käyttämällä turvareleitä. Taajuusmuuttajaan integroitu toiminnallinen turvallisuus helpottaa moottorinohjaussovellusten turvajärjestelmien toteuttamista, koska sen avulla voidaan yksinkertaistaa turvajärjestelmää jättämällä pois perinteisissä turvajärjestelmissä käytettäviä komponentteja.



Kuva 1 Perinteisen moottorinohjaussovelluksen turvajärjestelmä. (Siemens 2016a.)

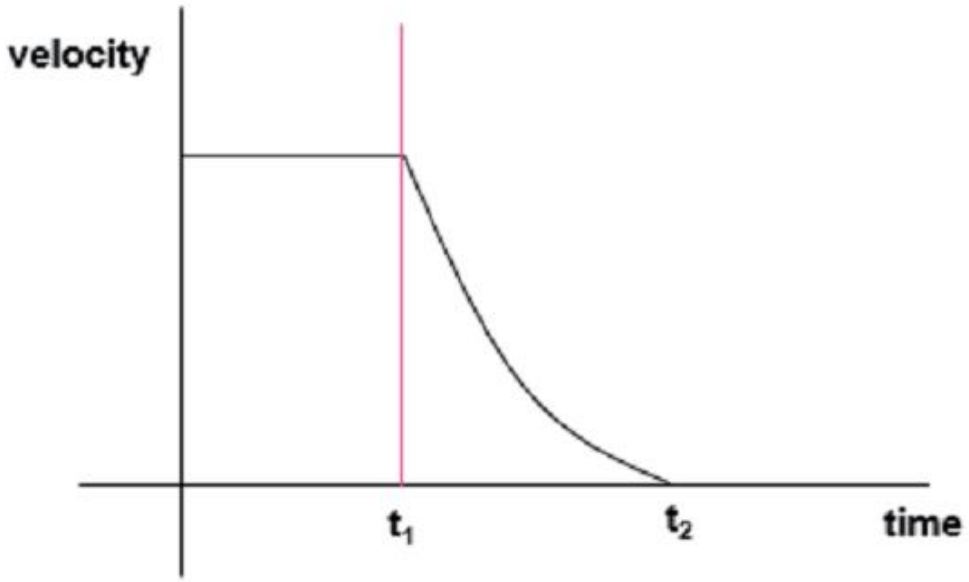
Kuvassa Kuva 1 on esitetty perinteinen moottorinohjaussovelluksen turvajärjestelmä ja kuvassa Kuva 2 moottorinohjaussovelluksen turvajärjestelmä, joka hyödyntää taajuusmuuttajaan integroitua toiminnallista turvallisuutta. (ABB 2016b; Meany 2016: 1361.)



Kuva 2 Taajuusmuuttajaan integroitu toiminnallinen turvallisuus osana moottorinohjaussovelluksen turvajärjestelmää. (Siemens 2016a.)

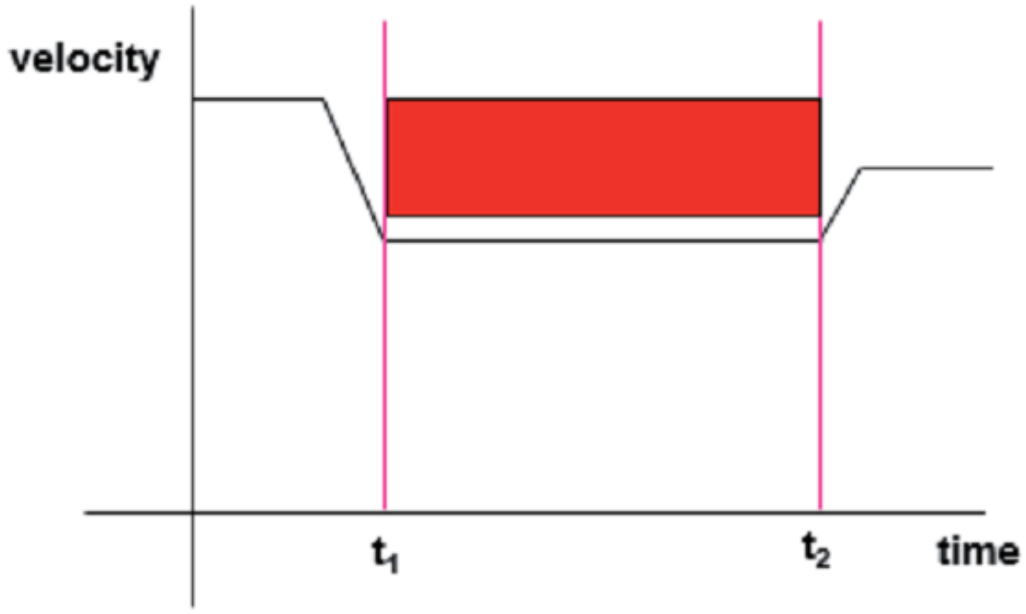
IEC 61800-5-2 määrittää taajuusmuuttajien toiminnallisen turvallisuuden toteuttamiseen käytettäviä turvafunktioita, joita ovat esimerkiksi STO (Safe torgue off) ja SLS (Safety limited speed). (Meany 2016: 1362.)

Kuvassa Kuva 3 on esitetty STO-turvafunktion toiminta. STO-turvafunktion avulla taajuusmuuttajan ohjaaman sähkömoottorin pysäytetään estämällä taajuusmuuttajan tuottaman sähköenergian pääsy sähkömoottoriin, jolloin sähkömoottori rullaa vapaasti pysähdyksiin. STO-turvafunktiota käytetään esimerkiksi hätäpysäytyspainikkeen toteuttamiseen. (Siemens 2016b; Meany 2016: 1363.)



Kuva 3 Safe torgue Off -turvafunktio. (Meany 2016: 1363.)

Kuvassa Kuva 4 on esitetty SLS-turvafunktion toiminta. SLS-turvafunktio avulla sähkömoottorin nopeus pidetään asetetuissa rajoissa, jos sähkömoottorin nopeus ylittää asetetut rajat suoritetaan asetetut toimenpiteet. SLS-turvafunktion avulla voidaan esimerkiksi rajoittaa sähkömoottorin pyörimisnopeus kuljettimen puhdistamisen ajaksi. (Meany 2016: 1363.)



Kuva 4 Safely limited speed -turvafunktio. (Meany 2016: 1363.)

3 KETTERÄT MENETELMÄT

Perinteiset ohjelmistokehitysmenetelmät, kuten vesiputousmalli suoriutuvat huonosti ohjelmistokehityksessä tilanteista, joissa vaatimukset muuttuvat. Ketterät ohjelmistokehitysmenetelmät koostuvat käytännöistä ja suoriutuvat paremmin muuttuvista vaatimuksista, koska ketterien ohjelmistokehitysmenetelmien käytännöt kasvattavat projektissa työskentelevien ihmisten kanssakäymistä. Tunnetuimpia ketteriä ohjelmistokehitysmenetelmiä ovat Scrum ja Extreme Programming. Yleisesti ohjelmistoprojekteissa käytetään ketteristä ohjelmistokehitysmenetelmistä muutamaa käytäntöä, koska käytäntöjen sisäistäminen on etujen saavuttamisen kannalta tärkeää. (Xiaodan & Stacie 2014: 912.)

3.1 Säännölliset tilannepalaverit käytäntö

Säännölliset tilannepalaverit käytännössä projektitiimin jäsenet pitävä säännöllisin väliajoin tilannepalaverin, jossa keskustellaan projektin tilanteesta. Esimerkiksi Scrum-menetelmässä tilannepalavereja pidetään kerran päivässä. Scrum-menetelmän tilannepalaverissä jokainen tiimin jäsen saa noin minuutin mittaisen puheenvuoron. (Lehtonen, Tuomivaara, Rantala, Känsälä, Mäkilä, Jokela, Könnölä, Kaisti, Suomi, Isomäki & Ylitoiva 2014: 43).

3.2 Iteraatio käytäntö

Iteraatio käytännössä projektitiimin työ jaksotetaan vakio mittaisiin jaksoihin eli sprintteihin. Jokaisen sprintin päätteeksi asiakkaalle toimitetaan julkaisu ohjelmistosta. Riippuen sprintin pituudesta muutokset, joita yhden sprintin aikana saadaan aikaiseksi voivat vaihdella. Tavoite kuitenkin on, että jokaisen sprintin aikana pystytään esimerkiksi toteuttamaan vähintään yksi uusi ominaisuus ohjelmistoon. Sprinttien pituudet voivat

vaihdella projektin koon mukaan 1 viikosta 4 viikkoon. Iteraatio käytäntöä käytetään sekä Scrum, että Extreme Programming -menetelmissä. (Lehtonen ym. 2014: 4–6, 35).

Scrum-menetelmässä sprintin sisältö suunnitellaan suunnittelupalaverissa projektitiimin jäsenet valitsevat priorisoidusta kehitysjonosta vaatimuksia, jotka halutaan toteuttaa sprintin aikana. Vaatimuksille annetaan työmääräarviot, joiden avulla projektitiimi pystyy päättelemään, kuinka paljon vaatimuksia sprintin aikana ehditään toteuttamaan. (Lehtonen ym. 2014: 37).

Scrum-menetelmässä sprintin päätteeksi pidetään katselmointipalaveri, jossa käydään läpi suunnittelupalaverissa valitut vaatimukset ja vaatimusten tila. Asiakkaalle esitellään valmiit vaatimukset, joita asiakas kommentoi. Keskeneräisistä vaatimuksista käydään läpi syyt, miksei vaatimuksia ole ehditty toteuttamaan. Vaatimusten tilat päivitetään kehitysjonoon seuraavaan sprintin suunnittelupalaveria varten. (Lehtonen ym. 2014: 39).

4 OHJELMISTOARKKITEHTUURIT

Ohjelmistoarkkitehtuuri asettaa rajat, joiden puitteissa ohjelmistoa tulisi ylläpitää ja kehittää. Ohjelmistoarkkitehtuuri määrittää säännöt joiden avulla ohjelmisto jaetaan komponentteihin, säännöt joiden avulla määritetään komponenttien väliset suhteet. Sääntöjen lisäksi ohjelmistoarkkitehtuuri määrittää ohjelmiston kehityksessä käytettävät suunnittelumallit. (Koskimies & Mikkonen 2005: 18–19.)

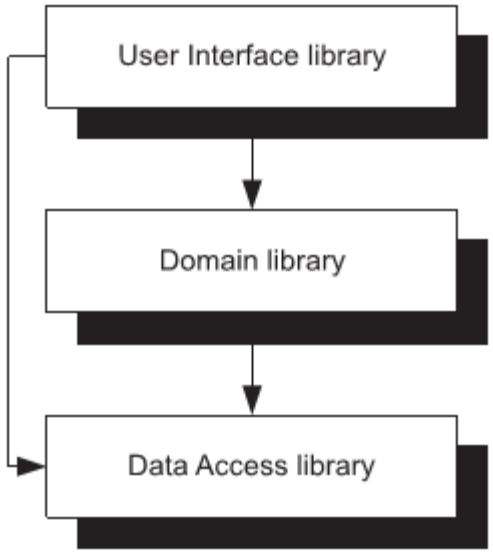
4.1 Kerrosarkkitehtuuri

Kerrosarkkitehtuuri on yleinen malli, jonka avulla voidaan yksinkertaisesti esittää monimutkaisemmankin ohjelmiston arkkitehtuuri jakamalla ohjelmiston komponentit tietyllä logiikalla kerroksiin. Kuvassa Kuva 5 on esitetty yksinkertainen kerrosarkkitehtuuri, jossa ylimpänä kerroksena on käyttöliittymä ja alimpana kerroksena tietokanta. (Koskimies & Mikkonen 2005: 130–131.)

Kuva 5 Yksinkertainen kerrosarkkitehtuuri. (Koskimies & Mikkonen 2005: 128.)

Yleisesti kerrosarkkitehtuuria käytetään toteuttamisen apuna niin, että ohjelmiston eri kerrosten komponentit sidotaan suoraan toisiinsa. Kerrosten sitominen toisiinsa aiheuttaa riippuvaisuuksia komponenttien välille vaikuttaen negatiivisesti ohjelmiston komponenttien uudelleenkäytettävyyteen. Kuvassa Kuva 6 User interface library -

komponentilla on riippuvuudet Domain library -komponenttiin ja Data Access library -komponenttiin. (Seeman 2012: 31–41.)



Kuva 6 Esimerkki komponenttien riippuvaisuuksista käytettäessä kerrosarkkitehtuuria. (Seeman 2012: 37.)

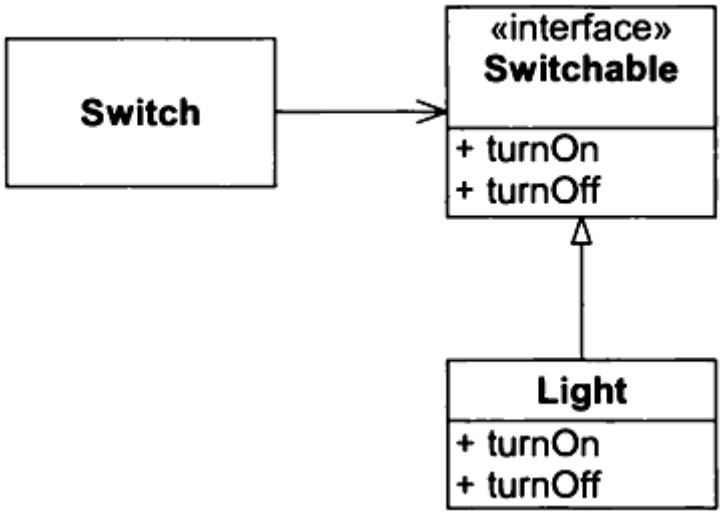
4.2 Suunnittelumallit ja periaatteet

Suunnittelumallit ovat dokumentoituja hyviksi todettuja ratkaisuja yleisiin ohjelmistoteknisiin ongelmiin. Suunnittelumallien käyttämisellä pyritään parantamaan ohjelmiston laatuominaisuuksia, joita ovat esimerkiksi testattavuus, muunneltavuus ja uudelleenkäytettävyys. Suunnittelumalli koostuu kolmesta osasta, jotka ovat ongelma, ongelmayhteys ja ratkaisu. Ongelma on kuvaus ohjelmointikieliriippumattomasta yleisestä ohjelmiston suunnitteluongelmasta, joka ilmenee monenlaisissa järjestelmissä. Ongelmayhteys määrittää tilanteen, jossa kyseistä suunnittelumallia on mahdollista käyttää ja ohjelmiston laatuominaisuuden, jota ratkaisun tulisi parantaa. Ratkaisu on luokkakaavio ja seliteteksti ongelman ratkaisusta sekä mahdollista lisäinformaatioita, esimerkiksi jos suunnittelumallin käyttö heikentää jotakin ohjelmiston laatuominaisuutta. (Koskimies & Mikkonen 2005: 102–105.)

Suunnitteluperiaatteet ovat hyväksi todettuja periaatteita, joita käyttämällä eliminoidaan suunnitteluvirheitä. Suunnitteluvirheet tunnistetaan oireiden perusteella, suunnitteluvirheiden oireita ovat jäykkyys, hauraus, muuttumattomuus, tahmeus, turha monimutkaisuus, turha toistaminen sekä sameus. Yleensä suunnitteluvirheet johtuvat jonkin suunnitteluperiaatteen laiminlyönnistä. (Martin 2003: 85–86.)

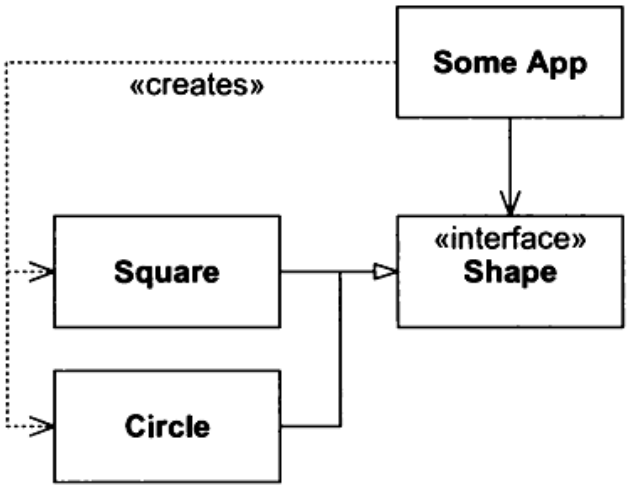
Yksi yleisimmistä tavoista kahden tai useamman komponentin riippuvaisuuksien poistamiseen on rajapinnan määrittäminen komponenttien välille. Tätä tapaa Robert kutsuu Abstract Server -malliksi. Kuvassa Kuva 7 on esitetty yksinkertainen luokkakaavio, jossa **Switch**-luokka on riippuvainen **Light**-luokasta. Ongelmaksi tämä muodostuu siinä vaiheessa, kun **Switch**-luokkaa halutaan käyttää jonkin muun, kuin **Light**-luokan kanssa. Kuvassa Kuva 8 on esitetty sama tilanne, mutta **Switch** ja **Light** -luokkien välinen riippuvuus on poistettu määrittämällä **Switchable**-rajapinta. (Martin 2003: 318.)

Kuva 7 Switch ja Light -luokkien välinen riippuvuus. (Martin 2003: 318.)

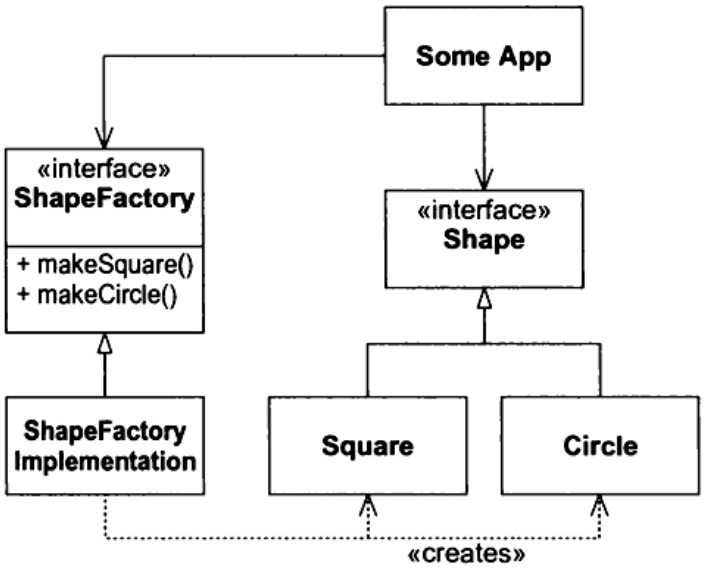


Kuva 8 Switch ja Light -luokan riippuvuus poistettu Switchable-rajapintaa käyttäen. (Martin 2003: 318.)

Toinen hyödyllinen suunnittelumalli luokkien välisten riippuvuuksien vähentämiseen on abstract factory -suunnittelumalli. Kuvassa Kuva 9 esitetty **Some App** -luokka on riippuvainen **Square** ja **Circle** -luokista, koska **Some App** -luokka on vastuussa **square** ja **circle** -objektien luomisesta. Kuvassa Kuva 10 on **Some App** -luokan riippuvuudet **square** ja **circle** -luokkiin poistettu abstract factory -suunnittelumallin avulla. Kuvassa Kuva 10 **square** ja **circle** -objektien luominen on siirretty **ShapeFactory**-rajapinnan toteutukseen, jolloin **Some App** -luokalla in riippuvuus vain **ShapeFactory**-rajapintaan.



Kuva 9 Some App -luokan riippuvuus Square ja Circle -luokkiin. (Martin 2003: 270.)



Kuva 10 Some App -luokan square ja circle -luokkien riippuvuudet poistettu Abstract Factory -suunnittelumallin avulla. (Martin 2003: 270.)

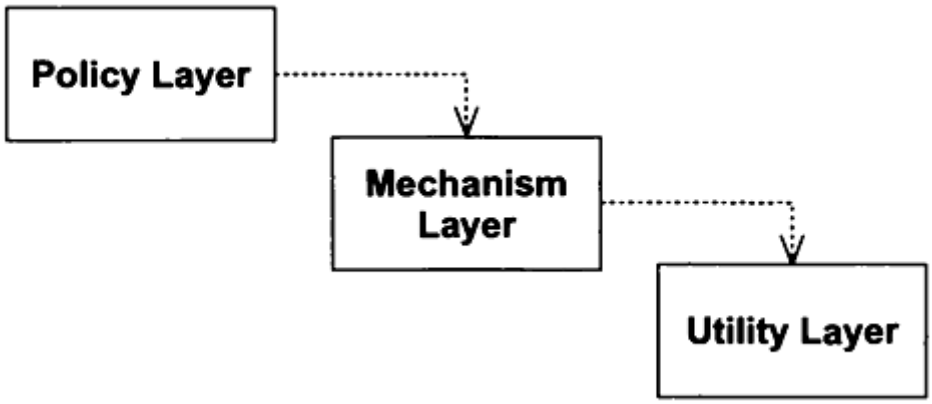
Riippuvuusinversio suunnitteluperiaatetta (Dependency inversion principle) voidaan myös käyttää komponenttien välisten riippuvaisuuksien vähentämiseen. Suunnitteluvirheiden oireista jäykkyys ja muuttumattomuus voidaan päätellä, että sisältääkö arkkitehtuuri komponenttien välisiä riippuvaisuuksia. Dependency Inversion -periaatetta (DIP) käytetään jäykkyyden ja muuttumattomuuden eliminointiin, jolloin sen käyttö vähentää riippuvaisuuksia ja parantaa muunneltavuutta. (Martin 2003: 85, 134.)

Dependency inversion -periaate suosittelee käyttämään kahta heuristiikkaa, jotka ovat:

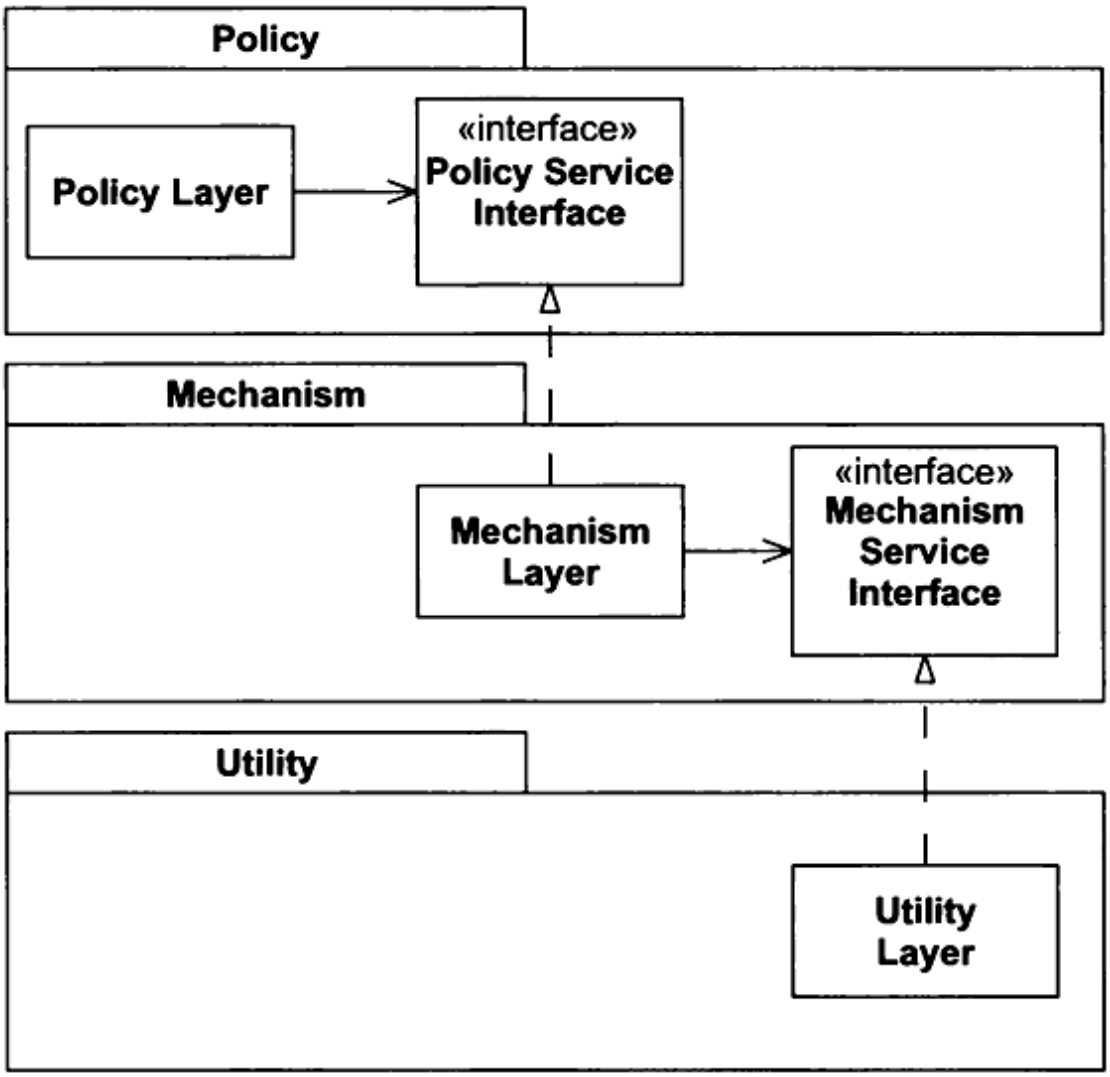
- *"High-level modules should not depend on low-level modules. Both should depend on abstractions."* (Martin 2003: 127.)

- *"Abstractions should not depend on details. Details should depend on abstractions."* (Martin 2003: 127.)

Dependency inversion -suunnitteluperiaate suosittelee määrittelemään rajapinnat ylemmän kerroksen vaatimusten mukaan. Kuvassa Kuva 11 on esitetty yksinkertainen kerrosarkkitehtuuri, jossa ylemmät kerrokset riippuvat alemmista kerroksista. Kuvassa Kuva 12 on määritetty rajapinnat ylemmän tason komponenttien tarpeiden perusteella ja alemman tason komponentit toteuttavat nämä rajapinnat.



Kuva 11 Kerrosarkkitehtuuri, jossa kerrokset riippuvat toisistaan. (Martin 2003: 128.)

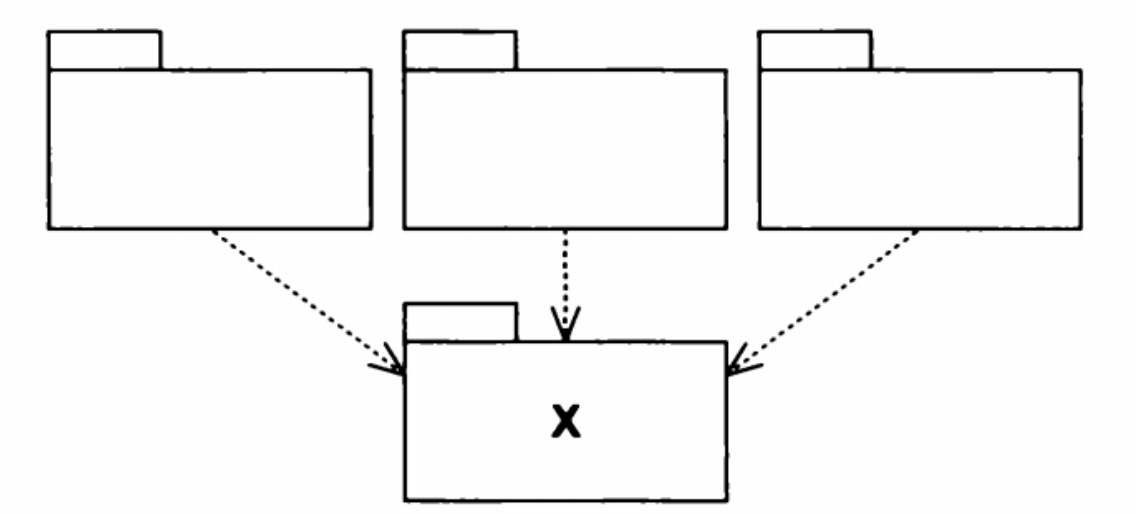


Kuva 12 Kerrosarkkitehtuuri, johon on sovellettu Dependency inversion -periaatetta. (Martin 2003: 129.)

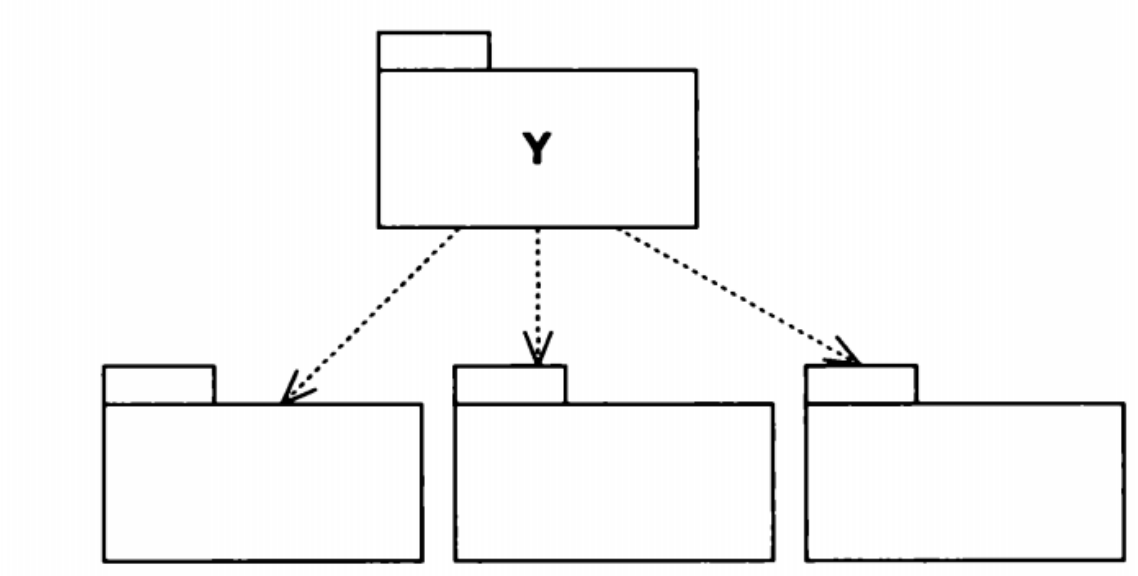
Dependency inversion -suunnitteluperiaate kieltää ylemmän tason komponentteja olemasta riippuvaisia alemman tason komponenteista, suosittelee erottamaan komponenttien toteutukset toisistaan rajapintojen avulla ja määrittelemään rajapinnat käyttävän komponentin näkökulmasta. (Martin 2003: 128–129.)

Vakaat riippuvuudet suunnitteluperiaatetta (Stable Dependencies principle) voidaan käyttää komponenttien välisten riippuvaisuuksien optimointiin. Stable Dependencies -suunnitteluperiaate (SDP) suosittelee, että komponenttien tulisi olla aina riippuvaisia vakaammista komponenteista. Kuvassa Kuva 13 on esitetty komponentti X, joka on

vakaa komponentti. Komponentti X on vakaa, koska se ei ole riippuvainen mistään muusta komponentista ja kolme muuta komponenttia ovat riippuvaisia siitä. Kuvassa Kuva 14 on esitetty komponentti Y, joka epävakaa, koska se riippuu kolmesta muusta komponentista eikä mikään muu komponentti riipu siitä. Komponenttien muunneltavuus on päinvastainen komponentin vakauteen nähden, eli vakaammat komponentin ovat huonommin muunneltavia kuin epävakaat komponentit. (Martin 2003: 261, 263.)



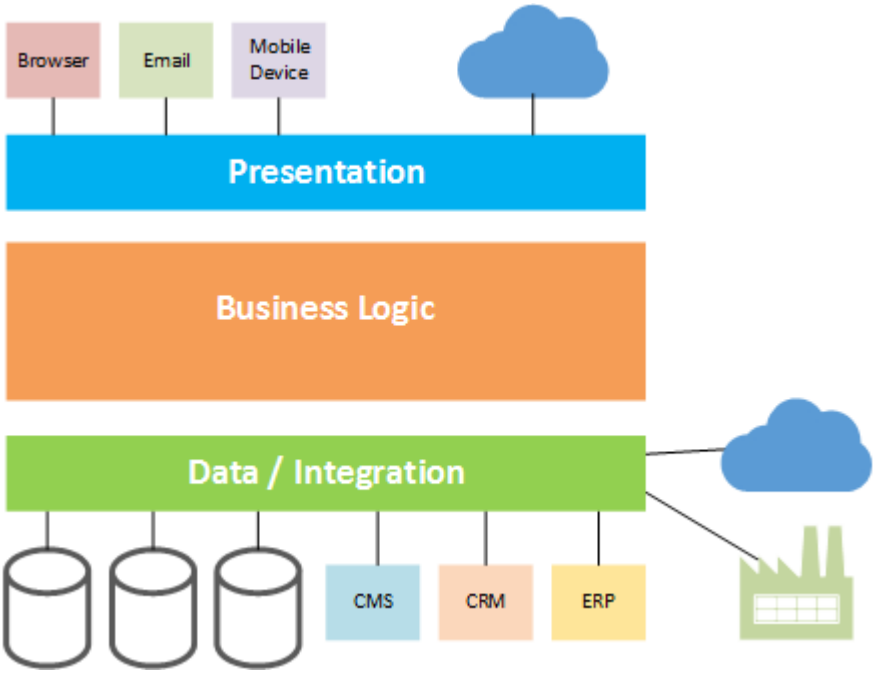
Kuva 13 Vakaa komponentti. (Martin 2003: 261.)



Kuva 14 Epävakaa komponentti. (Martin 2003: 262.)

4.3 Portit ja adapterit -arkkitehtuuri

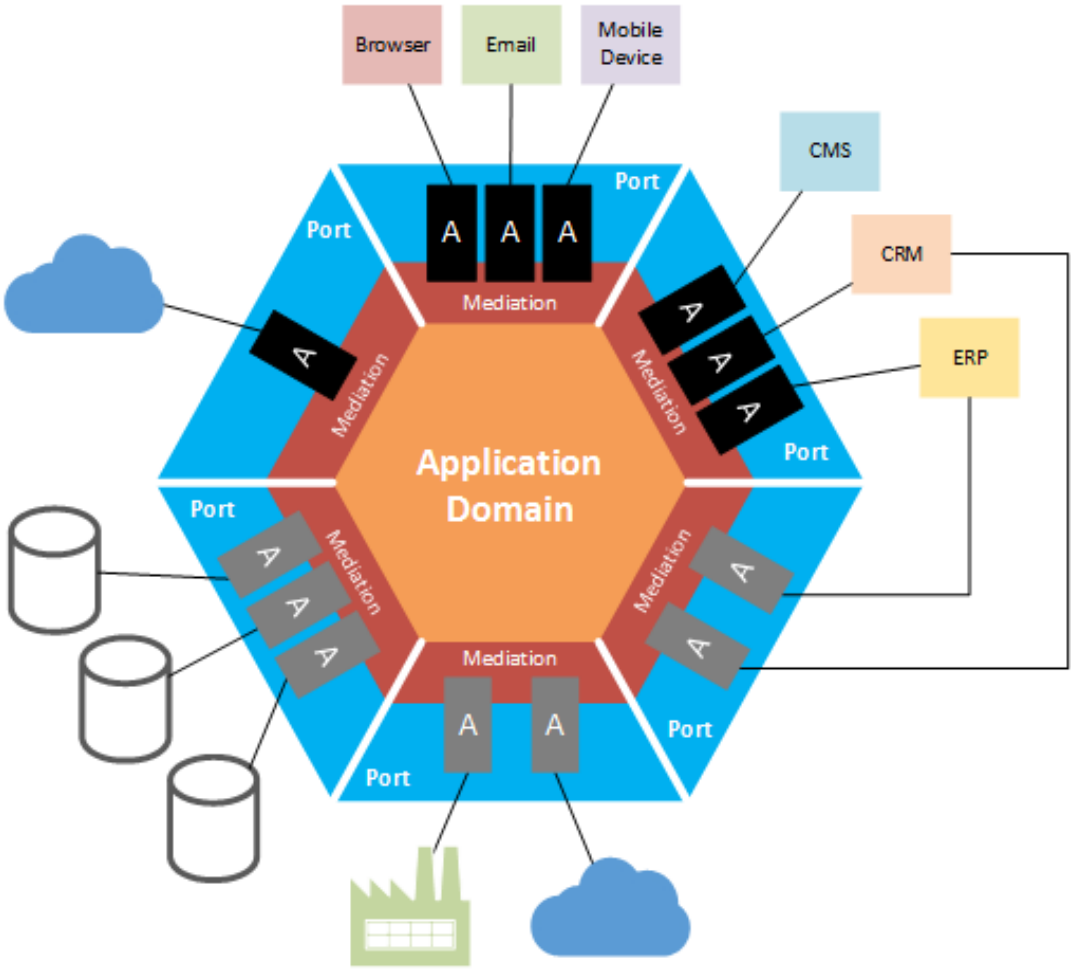
Kerrosarkkitehtuurin käyttäminen aiheuttaa pitkällä aikavälillä kerrosten välisten rajojen hämärtymisen, joka johtaa koodin vuotamiseen vääriin kerroksiin. Riippuen kerrosten toiminnallisuuden jaosta voi olla vaikeaa päättää että, mihin kerrokseen tietty toiminnallisuus tulisi integroida. Kuvassa Kuva 15 on esitetty esimerkki sovelluksen eri kerroksista. Esimerkiksi Twitter syötteen käsittelyn integrointi kuvan Kuva 15 mukaiseen kerrosarkkitehtuuriin voitaisiin tehdä sekä presentation-kerrokseen, että data-kerrokseen. (Young 2017.)



Kuva 15 Perinteinen kerrosten nimeämisen käytäntö. (Young 2017.)

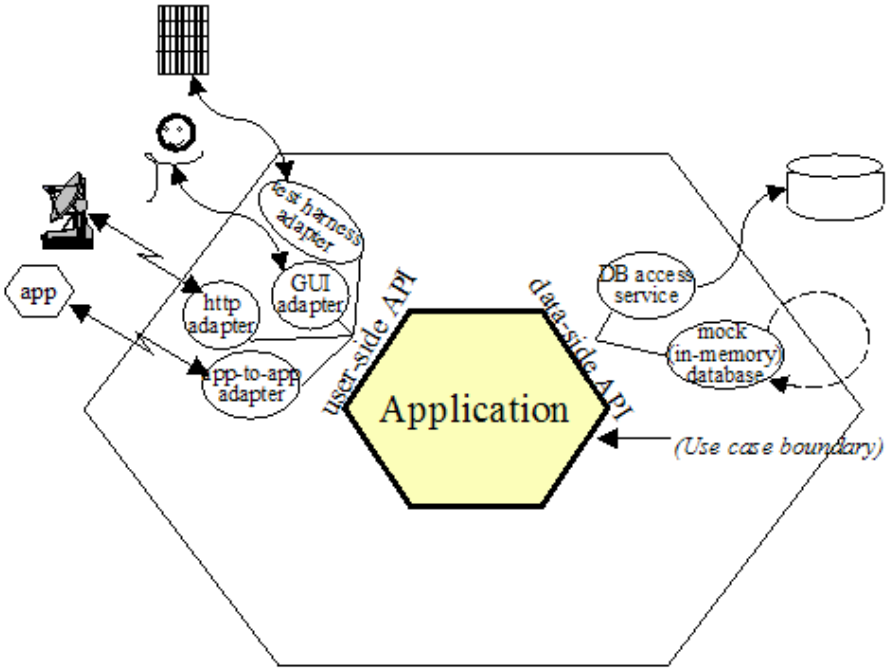
Hexagonal-arkkitehtuurissa yllä mainittuihin ongelmiin on pyritty löytämään ratkaisut eristämällä business logiikka (Application Domain) kuvan Kuva 16 esittämällä tavalla. Hexagonal-arkkitehtuuri tunnetaan myös nimellä portit ja adapterit -arkkitehtuuri. Markin (Seeman 2013) mukaan kerrosarkkitehtuurin pystyy muuntamaan portit ja adapterit -arkkitehtuuriksi käyttämällä dependency inversion -suunnitteluperiaatetta (Young 2017; Cockburn 2005.)

Kuvasta Kuva 16 on poistettu kuvassa Kuva 15 olevat kerrokset Presentation ja Integration. Kuvassa Kuva 16 business logiikka (Application Domain) sisältää mallin jonkin liiketoimintakohtaisen ongelman ratkaisuun eikä sillä ole riippuvaisuuksia ulkopuolelle. Komponentit kommunikoivat business logiikan kanssa porttien kautta, porteilla on kaksi tehtävää. Porttien ensimmäinen tehtävä on suojata business logiikkaa koodin vuotamiselta business logiikan ja komponenttien välillä. Porttien toinen tehtävä on varmistaa, että komponentit ryhmitellään käyttötarkoituksen mukaan. Portit määrittävät rajapinnat joiden kautta komponentit kommunikoivat business logiikan kanssa ja joiden kautta business logiikka kommunikoi komponenttien kanssa. Portit voidaan jakaa kahteen ryhmään ensisijaisiin ja toissijaisiin. Ensisijaisten porttien kautta komennetaan business logiikka suorittamaan jokin toiminto. Toissijaisten porttien kautta business logiikka komentaa komponentteja suorittamaan jonkin toiminnon. (Young 2017; Garrido de Paz 2018.)



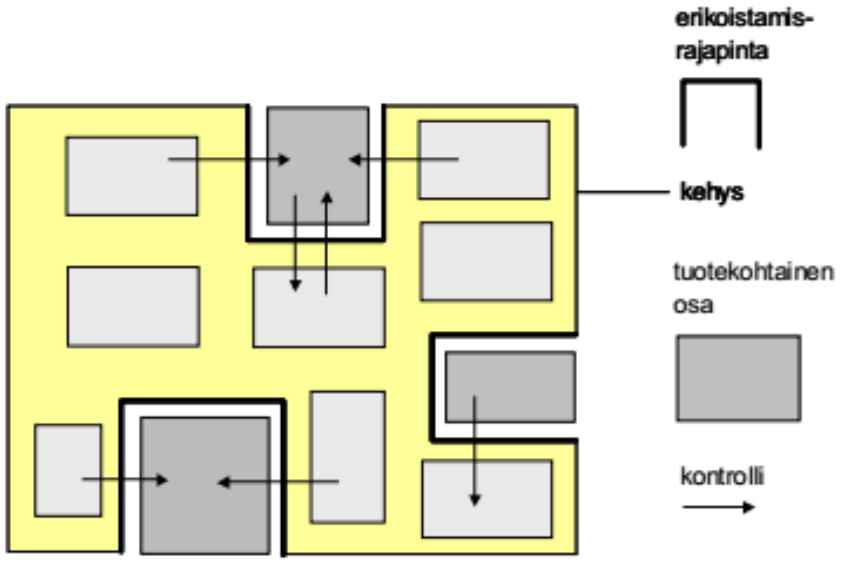
Kuva 16 Portit ja adapterit -arkkitehtuuri. (Young 2017.)

Komponentit, jotka kommunikoivat business logiikan kanssa porttien kautta kutsutaan sovittimiksi. Sovittimia on myös kahden tyyppisiä ensisijaisia ja toissijaisia. Ensisijaiset sovittimet käyttävät ensisijaisten porttien määrittämiä rajapintoja. Toissijaiset sovittimet toteuttavat toissijaisten porttien määrittämiä rajapintoja. Yhtä porttia kohden voi olla monia sovittimia. Portit ja adapterit -arkkitehtuuria voidaan käyttää hyväksi kuvan Kuva 17 mukaisesti testauksessa. Toteuttamalla user-side API -porttiin testipenkki-sovitin ja data-side API -porttiin mock-tietokanta -sovitin pystytään sovelluksen business-logiikan testaus automatisoimaan. (Young 2017; Cockburn 2005.)



Kuva 17 Portit ja adapterit -arkkitehtuuri testauksessa. (Cockburn 2005.)

Portit ja adapterit -arkkitehtuuri jakaa ominaisuuksia tunnetuista arkkitehtuureista ohjelmistokehyksien kanssa. Ohjelmistokehykset eivät itsessään ole suorituskelpoisia. Ohjelmistokehykset vaativat koodia kehyksen ulkopuolelta (Kuva 18), että kehyksestä saadaan suorituskelpoinen sovellus tai komponentti. Portit ja adapterit -arkkitehtuurissa ei business logiikka itsessään ole suorituskelpoinen. Portit ja adapterit -arkkitehtuurissa business logiikka vaatii vähintään toissijaisten porttien toteutukset (toissijaisen sovittimen) ollakseen suorituskelpoinen komponentti ja myös ensisijaisia portteja käyttävän komponentin (ensisijaisen sovittimen) ollakseen suorituskelpoinen sovellus. Ohjelmistokehyksen erona portit ja adapterit -arkkitehtuuriin voidaan pitää sitä, että portit ja adapterit -arkkitehtuurissa business logiikka pitää olla eristetty rajapintojen avulla, ohjelmistokehyksissä tätä rajausta ei suoranaisesti tehdä. (Koskimies & Mikkonen 2005: 190–192)



Kuva 18 Ohjelmistokehyksen vaatimat toteutukset kehyksen ulkopuolelta. (Koskimies & Mikkonen 2005: 190.)

5 TUTKIMUSMENTELMÄ JA LÄHTÖTILANNE

Tutkimusmenetelmänä päätettiin käyttää tapaustutkimusta, koska se soveltuu tosielämän ilmiöiden tutkimiseen. Tapaustutkimuksessa tutkitaan tosielämän ilmiötä ja se vastaa parhaiten tutkimuskysymyksiin, kuinka ja miksi. (Sjoberg, Dyba & Jorgensen 2007).

Tutkimuksessa päätettiin tutkia projektissa toteutetun PC-työkalun business-logiikassa käytetyn portit ja adapterit -arkkitehtuurin soveltuvuutta ketterään ohjelmistokehitykseen. Portit ja adapterit -arkkitehtuuri parantaa ohjelmiston ylläpidettävyyttä, joka on tärkeää ketterässä ohjelmistokehityksessä. Portit ja adapterit -arkkitehtuurin soveltuvuutta ketterään kehitykseen päätettiin tutkia muunneltavuuden kautta seuraavien tutkimuskysymyksien avulla:

- Parantaako portit ja adapterit -arkkitehtuuri ylläpidettävyyttä muunneltavuus komponentin kautta.

- Miksi portit ja adapterit -arkkitehtuuri parantaa ylläpidettävyyttä, muunneltavuus komponentin kautta.

Ensimmäiseen kysymykseen pyrittiin löytämään vastaus analysoimalla valittujen käyttäjätarinoiden aikana tehtyä toteutusta. Käyttäjätarinat valittiin projektin alkupäästä, koska projektin alkupään aikana toteutettiin suurin osa tärkeimmistä komponenteista. Toteutuksen muunneltavuutta analysoitiin riippuvaisuuksien avulla, koska niin kuin aiemmin kappaleessa 4.2 todettiin suorat riippuvaisuudet kahden toteutusta sisältävän komponentin tai luokan välillä vaikuttavat negatiivisesti muunneltavuuteen. Toteutuksen muunneltavuutta analysoitiin yksinkertaistetulla versiolla Stephenin ja Pao-Shengin (Yau Chang 1988: 374) esittämästä muunneltavuuden mittaamisen mallista. Yksinkertaistetussa muunneltavuuden mittaus mallissa toteutuksen sisäistä ja ulkoista muunneltavuutta luokkien ja komponenttien välillä analysoitiin riippuvaisuuksien kautta. Sisäisellä muunneltavuudella tarkoitetaan luokkien välistä muunneltavuutta komponentin

sisällä ja ulkoisella muunnettavuudella komponenttien välistä muunneltavuutta. Muunneltavuudella kuvataan muutosten tekemisen helppoutta komponenttiin ja todennäköisyyttä sille, että muutos toisessa komponentissa aiheuttaa muutoksia muihin komponentteihin. Toiseen kysymykseen pyrittiin löytämään vastaus vertailemalla PC-työkalun business-logiikan komponenttien välisiä riippuvaisuuksia Robertin (Martin 2003: 95–135, 253–268) mainitsemiin ketterien suunnitteluperiaatteiden asettamiin suosituksiin.

Ennen työn aloitusta projektia olivat vieneet eteenpäin kaksi muuta samassa projektissa työskentelevää kollegaa, projektin tilanne ennen työn aloitusta oli seuraava:

- Käyttöliittymästä oli käyttöliittymä spesifikaatioiden mukainen toteutus olemassa, jossa näytettiin demo dataa. Käyttöliittymää pystyi testaamaan nettiselaimen kautta.

- Business logiikan runko oli valmiina, joka tarkoitti, että käyttöliittymän ja business logiikan välisen rajapinnan toiminnot oli dokumentoitu samalle tasolle käyttöliittymän tarpeen kanssa, mutta toteutus puuttui business logiikasta. Business logiikan arkkitehtuuriksi oli valittu portit ja adapterit -arkkitehtuuri.

- PC-työkalulla oli toimiva Windows asennusohjelma, johon oli toteutettu vain perusominaisuudet.

- PC-työkalussa oli Windows komentorivi käyttöliittymä, jolla oli mahdollista ladata binääriformaatissa oleva turvafunktioiden parametri tiedosto taajuusmuuttajalle.

- Kommunikaatiokirjastoon oli toteutettu osa tarvittavista toiminnoista taajuusmuuttajan kanssa kommunikoimiseen.

- Taajuusmuuttajan kommunikaatio rajapinta oli keskeneräinen.

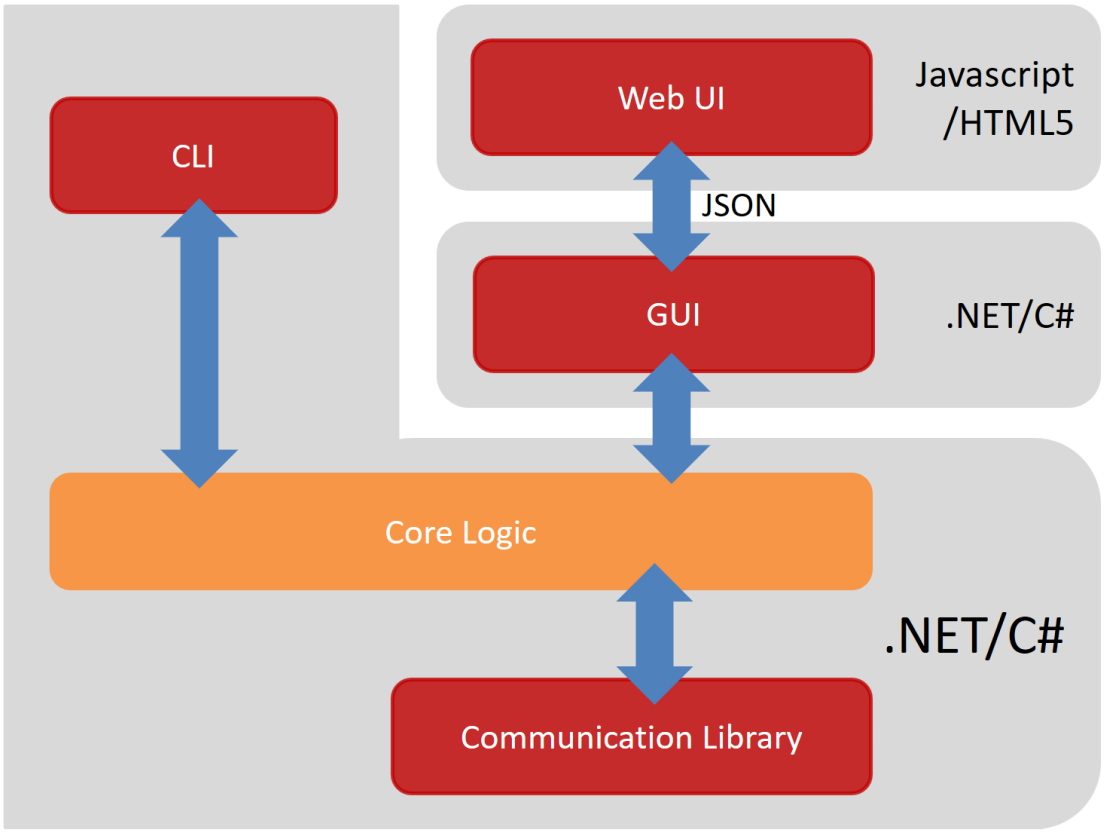
Tämän työn tekijä työskenteli kahden projektissa olevan kollegan kanssa. Toisen kollegan vastuualue oli käyttöliittymän toteuttaminen. Toinen kollega avusti business logiikan ja kommunikaatiokirjaston toteuttamisessa tarpeen mukaan.

Projektissa käytettiin ketteriä ohjelmistokehitysmenetelmiä. Scrum-menetelmästä käytettiin sprintti ja säännölliset tilannepalaverit käytäntöä. Sprintin pituutena käytettiin kuutta viikkoa ja tilannepalavereita pidettiin kerran viikossa. Ajallisesti projekti kesti 12-kuukautta ja siihen mahtui 8-sprinttiä. Taulukossa Taulukko 1 on esitetty jokaisen sprintin sisältö käyttäjätarinoiden avulla ja *kursivoitu* käyttäjätarinat, joiden toteutusta käytettiin tutkimuksen materiaalina.

Taulukko 1 Yleiskuva projektin sprinttien(S) ja käyttäjätarinoiden(KT) sisällöstä

S	KT	Kuvaus
1	1	Komentorivi liittymän perus rakenne ja parametritiedoston lataus komento.
1	2	Arkkitehtuurin suunnittelu ja PC-Työkalun rungon toteutus.
1	3	Diagnostiikka tiedon tallentaminen.
2	1	JSON-rajapinnan käyttöönotto käyttöliittymässä.
2	2	*Parametritiedoston eri versioita tukevan rajapintarakenteen määrittäminen.*
2	3	*Parametritiedoston rakenteen ensimmäisen version toteutus.*
2	4	*JSON-rajapinnan toimintojen toteuttaminen Business logiikkaan.*
2	5	*Demo datan generointi ja taajuusmuuttajan simulointi.*
3	1	*JSON-rajapinnan yhteensopivuuden varmistaminen business logiikassa.*
3	2	JSON-rajapinnan yhteensopivuuden varmistaminen käyttöliittymässä.
3	3	Taajuusmuuttajan kommunikaatioprotokollan toteutus kommunikaatiokirjastoon.
3	4	*Kommunikaatiokirjaston integrointi PC-työkaluun.*
4	1	Parametritiedoston avaaminen ja tallentaminen pc:n tiedostojärjestelmään
4	2	Turvafunktioiden riippuvaisuuksien kuvaaminen parametritiedostossa.
4	3	Turvafunktioiden riippuvaisuuksien esittäminen käyttöliittymässä.
4	4	Loppujen komentorivi komentojen toteutus.
4	5	Yhteyden katkaiseminen laitteesta.
5	1	Toimintojen käsittelijöiden integraatiotestien toteutus.
5	2	Manuaalisten testitapausten dokumentointi.
5	3	Turvafunktioiden parametrien riippuvaisuuksien kuvaaminen parametritiedostossa.
6	1	Korjauksia käytettävyystesteihin.
6	2	Aktiivisuuslokin toteutus käyttöliittymään,
6	3	Aktiivisuuslokin toteutus business-logiikkaan.
6	4	Käyttöönottoraportin toteutus.
7	1	Laitetietojen liittäminen käyttöönottoraporttiin.
7	2	Raportoitujen bugien korjailua.
8	1	Aktiivisuuslokin tallennus tiedostoon.
8	2	Asennusohjelman viimeistely.
8	3	Käyttöönottoraportin tulostaminen.

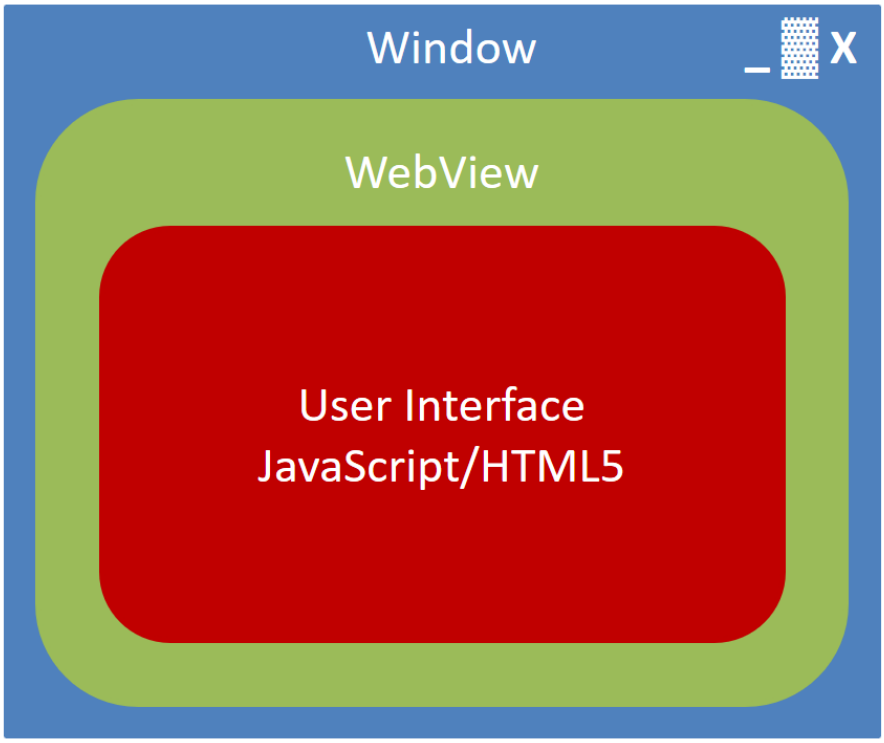
Kuvassa Kuva 19 on esitetty ohjelmiston korkean tason rakenne kerrosarkkitehtuuria käyttäen. Graafisen käyttöliittymän (Web UI) toteutustekniikkana käytettiin Javascript-ohjelmointikieltä. Business logiikan ja kommunikaatiokirjaston toteutustekniikkana käytettiin C#-ohjelmointikieltä. Käyttöliittymän ja business logiikan välisessä rajapinnassa käytettiin JSON (JavaScript Object Notation) -formaattia, joten rajapinta nimettiin JSON-rajapinnaksi. JavaScript Object Notation -formaatti on tekstimuotoinen ja sitä käytetään JavaScript-ohjelmointikielessä tiedon säilömiseen ja siirtoon. Business logiikalla (Business Logic) viitataan kaikkiin muihin komponentteihin paitsi kommunikaatiokirjastoon (Communication Library) ja graafisen käyttöliittymän Web UI -osaan.



Kuva 19 Ohjelmiston arkkitehtuuri esitettynä kerrosarkkitehtuuria käyttäen

JavaScript-käyttöliittymä upotettiin C#-ohjelmointikielen tarjoamaan webview-käyttöliittymäkomponenttiin kuvan Kuva 20 mukaisesti. WebView-komponentti on sovellusalustakohtainen käyttöliittymäkomponentti, joka pystyy www-selaimen tavoin

suorittamaan javascript-ohjelmakoodia ja esittämään html-sisältöä. (Hazarika, Rahul, & Seshubabu 2014: 1589).



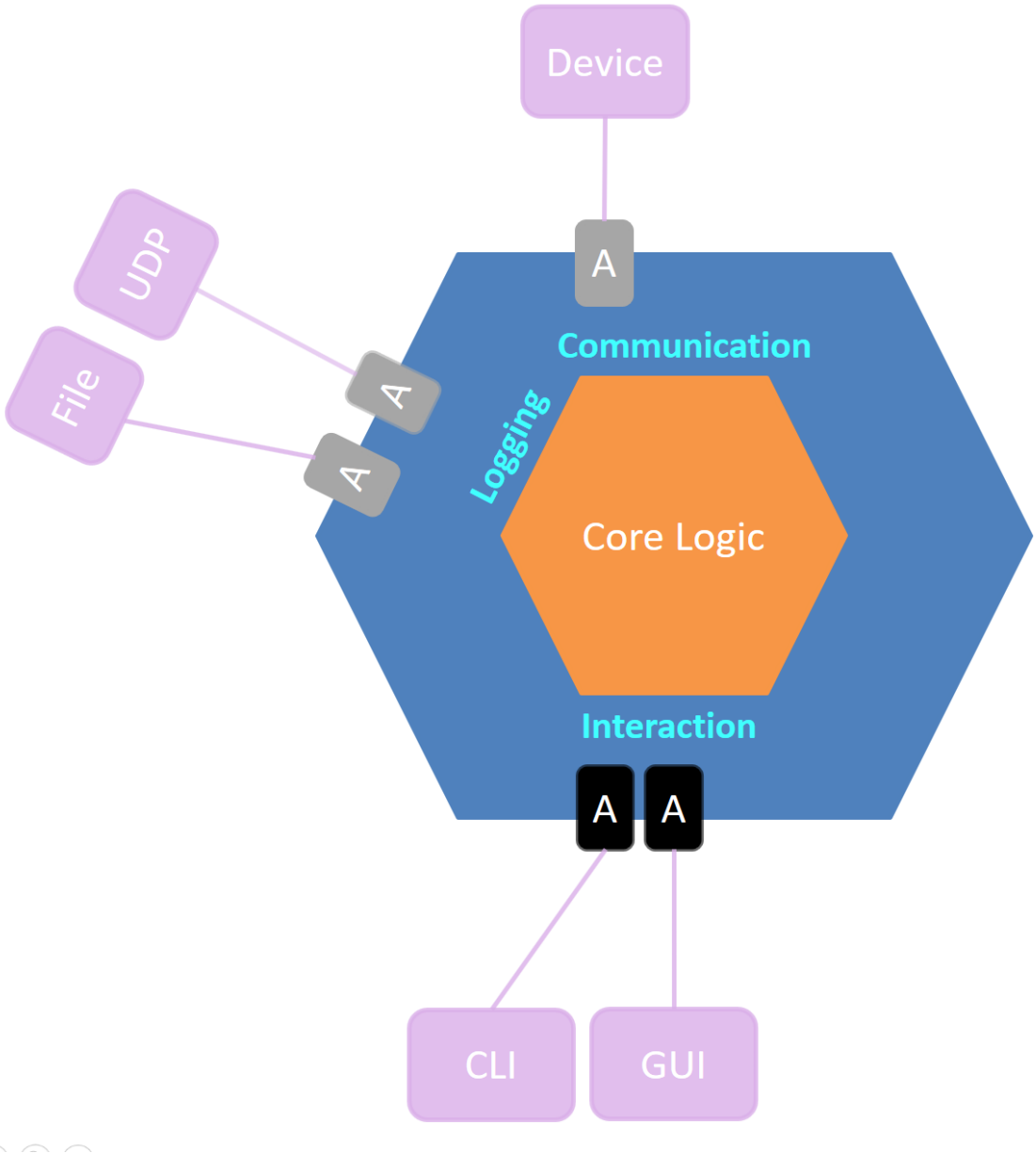
Kuva 20 JavaScript käyttöliittymä WebView-komponentissa

Kuvassa Kuva 21 on esitetty alkutilanne portit ja adapterit arkkitehtuuria käyttäen. Ohjelmiston business-logiikkaan oli määritetty kolme porttia, jotka olivat Communication, Logging ja Interaction.

Interaction-portin oli ensisijainen portti, jonka käyttötarkoitus oli välittää käyttäjän komennot business logiikalle. Interaction-portiin oli aloitettu toteuttamaan kahta sovitinta, jotka olivat GUI-sovitin ja CLI-sovitin. CLI-sovittimen tehtävä oli tarjota käyttäjälle komentorivi käyttöliittymä. GUI-sovittimen tehtävä oli tarjota käyttäjälle JavaScript-ohjelmointikielellä toteutettu graafinen käyttöliittymä.

Communication-portti oli toissijainen portti, jonka käyttötarkoitus oli mahdollistaa business logiikan kommunikointi taajuusmuuttajan kanssa. Communication-porttiin oli

aloitettu toteuttamaan Device-sovitinta, jonka tehtävä oli kommunikoida taajuusmuuttajan kanssa käyttäen kommunikaatiokirjastoa.



Kuva 21 Ohjelmiston alkutilanne esitettynä portit ja adapterit -arkkitehtuurilla.

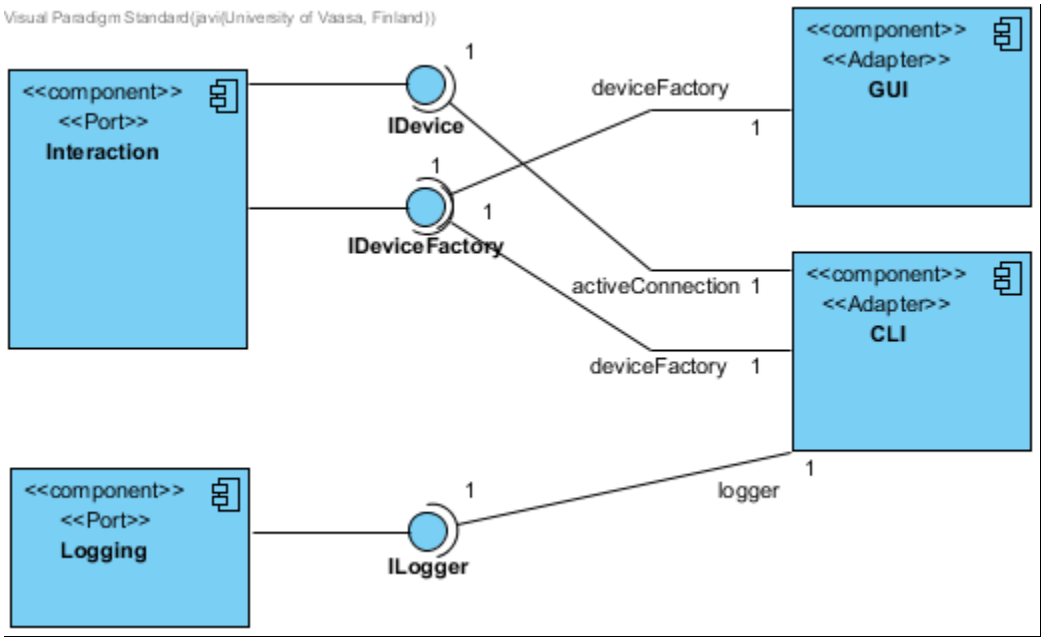
Logging-portti oli toissijainen portti ja sen käyttötarkoitus oli mahdollistaa diagnostiikka tiedon siirtäminen business logiikasta ulospäin. Logging-porttiin oli toteutettu kaksi sovitinta File ja UDP, koska kehitys käytössä diagnostiikka dataa haluttiin vastaanottaa reaaliajassa Log4Net-muodossa ja tuotanto käytössä diagnostiikka data haluttiin tallen-

taa tekstitiedostoihin. File-sovittimen tehtävä oli kirjoittaa diagnostiikka dataa määrättyyn tekstitiedostoon. UDP-sovittimen tehtävä oli lähettää diagnostiikka dataa käyttäen IP-pinon UDP-protokollaa ennalta määrättyyn osoitteeseen. UDP on yhteydetön IP-protokollan kuljetuskerroksen tiedonsiirto protokolla, jota käytetään esimerkiksi suoratoistopalveluissa videokuvan siirtämiseen. (IPv6 2017).

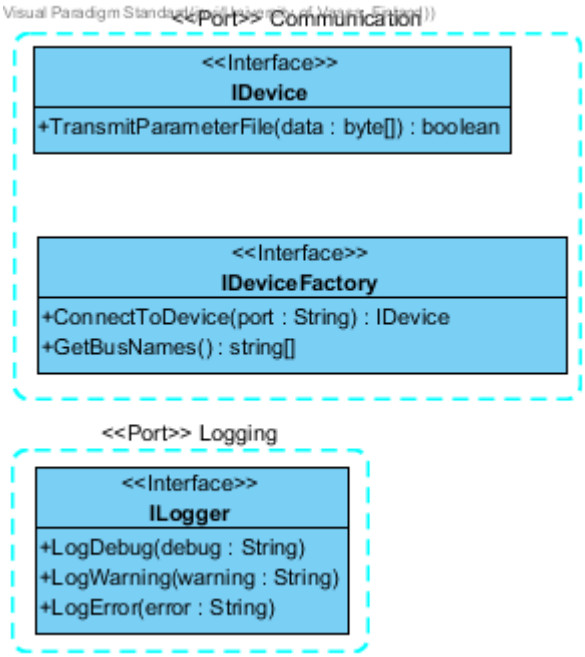
Communication ja Interaction -porttien rajapintojen määrittäminen oli juuri aloitettu, mutta Logging-portin rajapinta oli valmis. Logging-portin sovittimien toteutukset File ja UDP olivat valmiit. Communication-portin Device-sovittimen, Interaction-portin GUI ja CLI -sovittimien toteutus oli juuri aloitettu.

5.1 Keskeneräiset portit ja sovittimet ennen työn aloitusta

Interaction ja Communication -portteja ja sovittimia oli ehditty määrittämään hiukan ennen työn aloitusta. Logging-portin rajapinta oli valmis. Kuvissa Kuva 22 ja Kuva 23 on esitetty Interaction ja Logging -porttien rajapinnat ja niiden sisältämät funktiot työn aloitushetkellä. Interaction-portti tarjosi Kaksi rajapintaa, jotka olivat **IDeviceFactory** ja **IDevice. IDeviceFactory**-rajapinnan tehtävä oli luoda aktiivisia yhteyksiä laitteisiin ja **IDevice**-rajapinnan avulla laiteelle pystyi lataamaan binaarimuodossa olevan parametritiedoston. Logging-portti tarjosi vain yhden rajapinna **ILogger**, jonka avulla muut komponentit pystyivät välittämään diagnostiikka dataa käyttäjille. CLI-sovittimesta käytettiin kaikkia interaction ja logging -porttien tarjoamia rajapintoja, mutta GUI-sovittimesta käytettiin vain **IDeviceFactory**-rajapinan GetBusNames-funktiota.



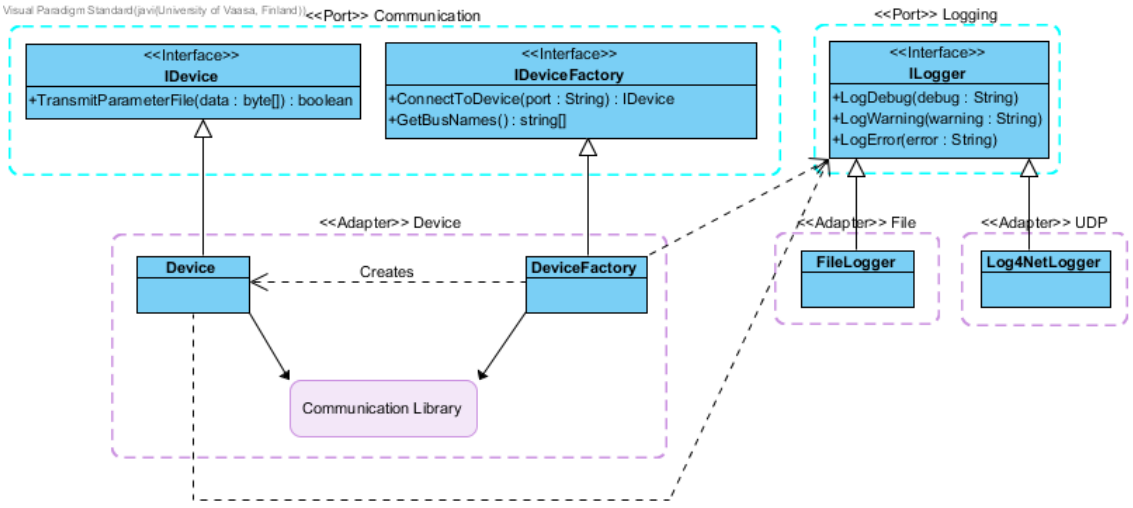
Kuva 22 Porttien tarjoamat rajapinnat työn aloitusvaiheessa.



Kuva 23 porttien tarjoamien rajapintojen toiminnot työn aloitusvaiheessa.

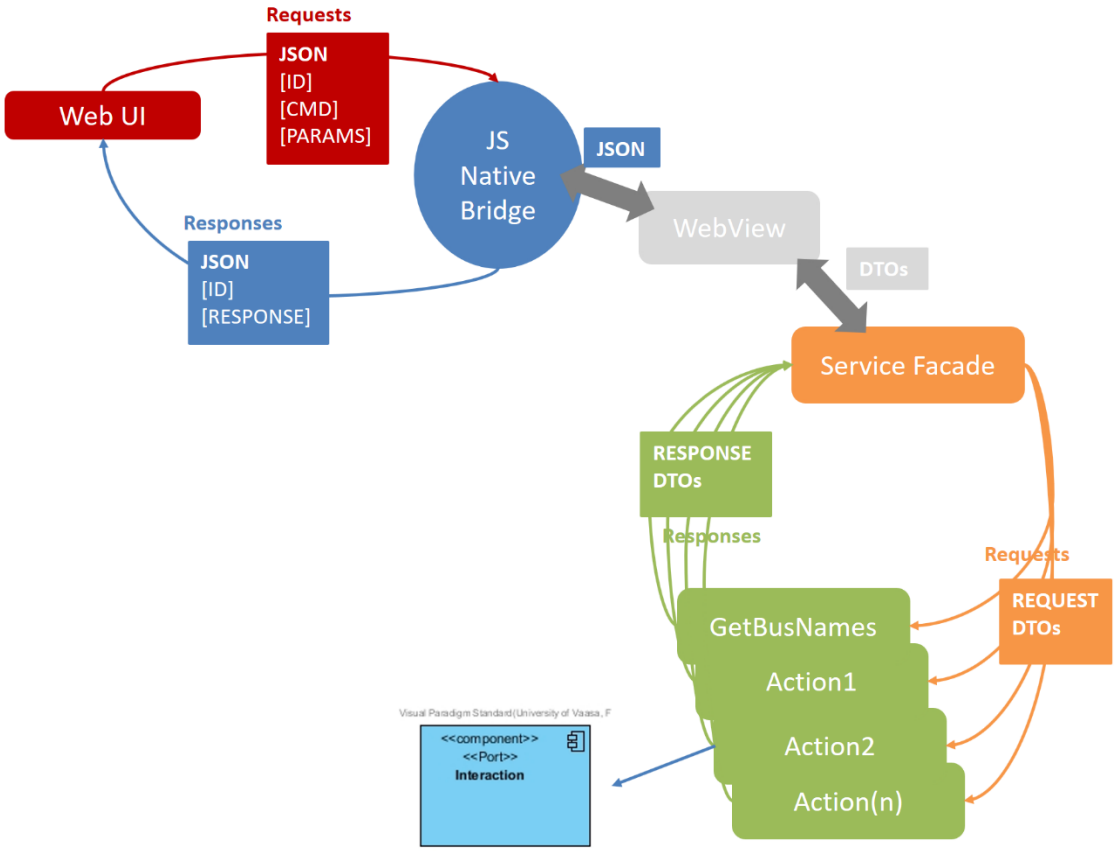
Kuvassa Kuva 24 on esitetty Communication-portin rajapinnat, jotka Device-sovitin toteuttaa käyttäen Communication library -komponenttia ja Logging-portin rajapinta, jonka toteuttavat File ja UDP -sovittimet. Interaction ja Communication -porttien raja-

pinnat olivat samat, koska työn aloitusvaiheessa niiden välille ei ollut vielä toteutettu mitään toiminnallisuutta.



Kuva 24 Communication-portin Device-sovittimen toteutus ennen työn aloitusta.

GUI-sovittimeen oli toteutettu peruslogiikka, jonka avulla Javascript käyttöliittymän komennot välitetään C# toiminnoille (Actions) ja toimintojen käsittelijöiden vastaukset välitetään takaisin käyttöliittymälle. Kuvassa Kuva 25 on esitetty GUI-sovittimen pääkomponentit **Web UI**, **JS Native Bridge** ja **Service Facade. Web UI** -komponentti sisältää Javascript-kielellä toteutetun käyttöliittymän, jonka tehtävä on välittää käyttäjän komennot **JS Native Bridge** -komponentin kautta **Service Facade** -komponentille. **JS Native Bridge** -komponentti mahdollistaa JavaScript ja C# -koodin välisen kommunikaation. **Service Facade** -komponentilla on kaksi tehtävää pyyntö JSON-objektien käsittely ja vastaus C#-objektien käsittely. Pyyntö JSON-objektien käsittelyssä pyyntö tyyppiset JSON-objektit muunnetaan C#-objekteiksi (Request-Dto) ja välitetään oikealle toiminnolle (Action) käsiteltäväksi. Vastaus C#-objektien (Response-Dto) käsittelyssä C#-objektit muunnetaan JSON-objekteiksi ja välitetään takaisin **JS Native Bridge** -komponentille.



Kuva 25 GUI-sovittimen pääkomponentit

Pyyntö JSON-objektit sisältävät kolme kenttää ID, CMD ja PARAMS. ID-kenttä sisältää uniikin tunnisteen, jonka avulla vastaus pystytään linkittämään oikeaan pyyntöön. CMD-kenttä kertoo **Service Facade** -komponentille, että mille toiminnolle pyyntö lähetetään käsiteltäväksi. PARAMS-kenttä sisältää toimintokohtaiset parametrit. Vastaus JSON-objektit sisältävät kaksi kenttää ID ja RESPONSE. ID-kentän tehtävä on sama kuin pyyntöobjektilla. RESPONSE-kenttä sisältää toimintokohtaisen vastauksen. Toiminnot käyttävät interaction-portin tarjoamia rajapintoja.

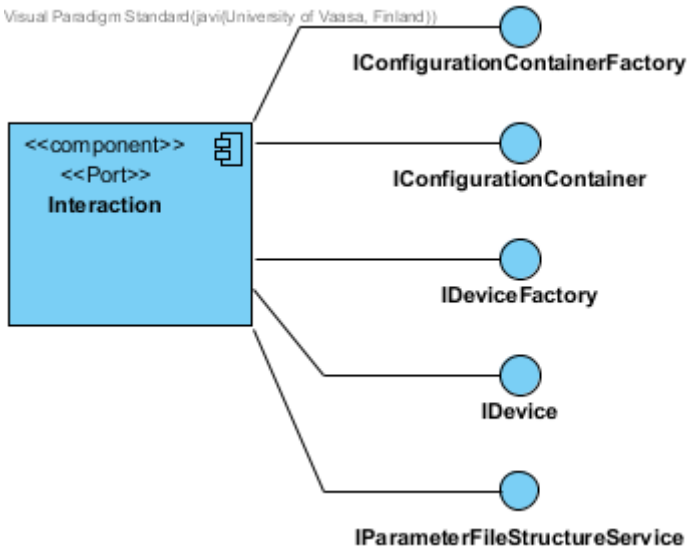
6 TYÖN OSUUS

Seuraavissa kappaleissa käydään läpi aiemmin taulukossa Taulukko 1 esitellyistä käyttäjätarinoista valitut ja analysoidaan niiden toteutuksen sisäistä ja ulkoista muunneltavuutta.

6.1 Parametritiedoston rakenne

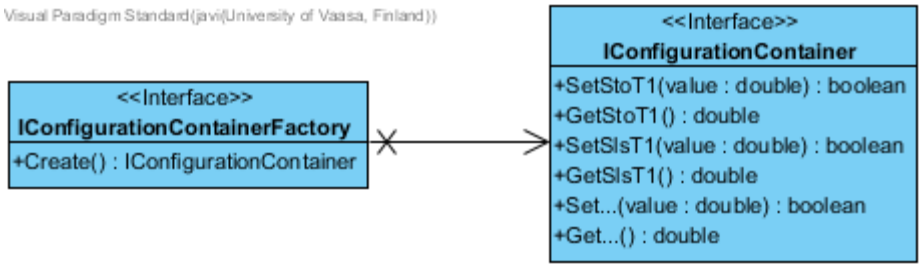
Toisen sprintin toisessa ja kolmannessa käyttäjätarinassa oli tarkoitus toteuttaa business logiikkaan tuki parametritiedoston rakenteen esittämiselle ja versioinnille. Parametritiedoston rakenne määritti turvafunktiot ja parametrit, joita PC-työkalulla pystyttäisiin konfiguroimaan. Asiakasvaatimuksen epäselvyyden vuoksi parametritiedoston rakenteen, ensimmäisen version toteutus ja rakenteen arvojen editointi toteutettiin business logiikan core logic -komponenttiin. Tämän lisäksi interaction-porttiin lisättiin tarvittavat rajapinnat parametriarvojen editointiin ja parametritiedoston rakenteen palauttamiseen.

Parametrienarvojen lukemista ja kirjoittamista varten interaction ja communication -portteihin määritettiin **IConfigurationContainer** ja **IConfigurationContainerFactory** -rajapinnat, joiden kautta uuden parametritiedoston luonti ja parametrien arvojen lukeminen ja kirjoittaminen onnistuisi. Interaction-porttiin lisättiin myös **IParameterStructureService-**rajapinta parametritiedoston rakenteen palauttamista varten käyttöliittymälle. Kuvassa Kuva 26 on esitetty Interaction-portin tarjoamat rajapinnat muutosten jälkeen.



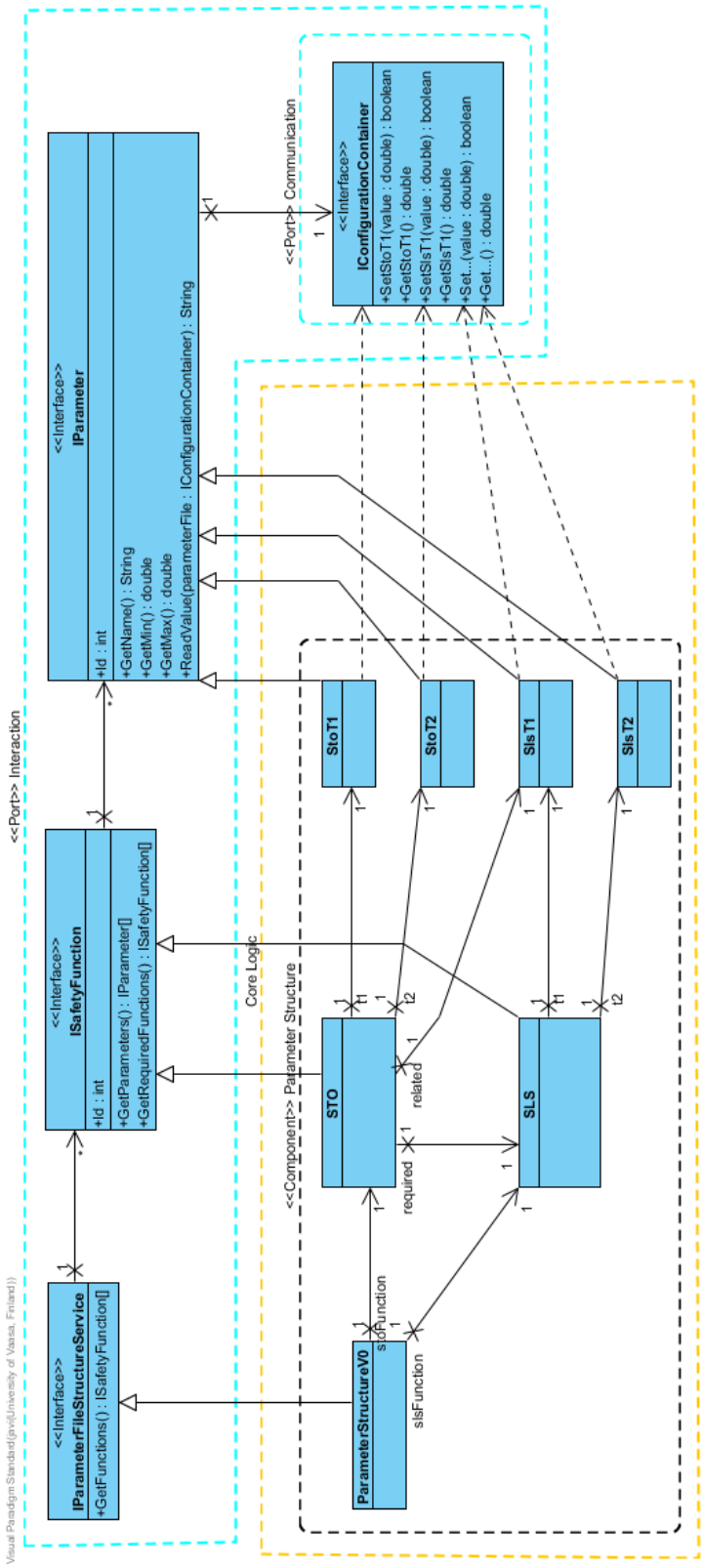
Kuva 26 Interaction-portin tarjoamat rajapinnat muutosten jälkeen

Kuvassa Kuva 27 on esitetty **IConfigurationContainer** ja **IConfigurationContainer-Factory** -rajapintojen metodit. **IConfigurationContainerFactory** -rajapinnan *Create*-metodilla luodaan uusia instansseja **IConfigurationContainer**-rajapintojen toteutuksista.



Kuva 27 IConfigurationContainer ja IConfigurationContainerFactory -rajapintojen metodit

Kuvassa Kuva 28 on esitetty tarkemmin parametritiedoston rakenteen rajapinnat interaction-portissa ja toteutukset core logic -komponentissa. parametritiedoston rakenteen kuvaus koostuu kolmesta eri rajapinnasta **IParameterStructureService**-rajapinta, **ISafetyFunction**-rajapinta ja **IParameter**-rajapinta.



Kuva 28 Parametritiedoston rakenteen rajapinnat ja toteutus

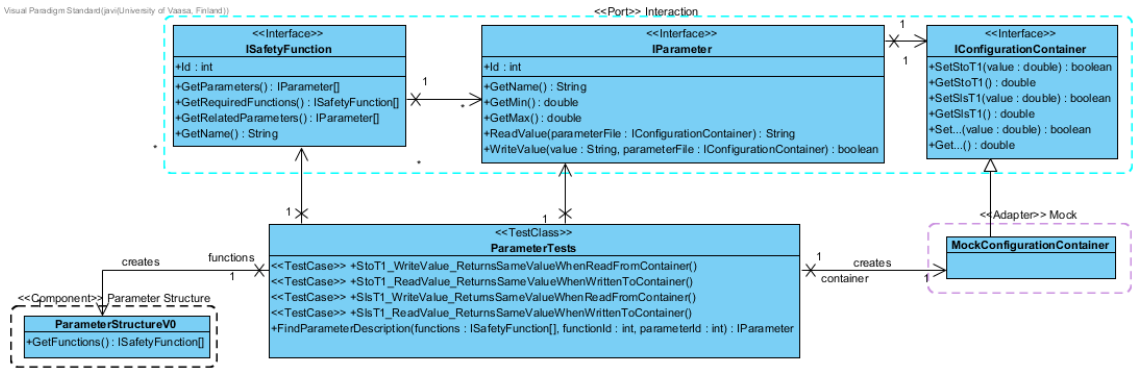
IParameterStructureService-rajapinnalla on GetFunctions-metodi, joka palauttaa määritetyn parametrirakenteen. **ISafetyFunction**-rajapinta mahdollistaa eri turvafunktioiden määrittämisen, turvafunktioilla voi olla riippuvaisuuksia muihin turvafunktioihin ja yksittäisiin parametreihin. **IParameter**-rajapinta mahdollistaa turvafunktioiden parametrien määrittämisen ja parametriarvojen sitomisen turvafunktioihin. Turvafunktioiden parametrit käyttävät Communication-portin **IConfigurationContainer**-rajapintaa parametrin arvon lukemiseen ja kirjoittamiseen.

Parametritiedoston rakenne -komponentin sisäinen toteutus on sidottu yhteen, joka tekee komponentin sisäisestä toteutuksesta vaikeasti muunneltavan. **IParameterStructureService** -rajapinnan toteutus riippuu STO ja SLS -turvafunktioiden toteutuksista, jotka riippuvat niiden parametrien toteutuksista. Muutokset turvafunktioiden parametreihin aiheuttavat todennäköisesti muutoksia turvafunktioiden toteutuksiin, koska turvafunktiot muodostuvat parametreista. Esimerkiksi muutos StoT1-luokan konstruktorin parametreihin aiheuttaisi muutoksen STO-luokkaan, koska STO-luokka on vastuussa StoT1-objektin luomisesta, sama pätee STO ja ParameterStructureV0 -luokkien välillä.

Parametritiedoston rakenne -komponentti riippuu vain Interaction portista, joten muutokset Parametritiedoston rakenne -komponenttiin eivät aiheuta muutoksia muihin komponentteihin ja ainoastaan muutokset Interaction porttiin voivat aiheuttaa muutoksia Parametritiedoston rakenne -komponenttiin. Parametritiedoston rakenne -komponentin riippuvuus ainoastaan Interaction portista johtaa siihen, että komponenttitasolla Parametritiedoston rakenne -komponentti on helposti muunneltava. Käytännössä helposti muunneltavuus tarkoittaa sitä, että muutos vain **IConfigurationContainer**-rajapinnasa voi aiheuttaa muutoksia turvafunktioiden parametreissa.

Parameter structure -komponentin moduulitestit toteutettiin **ParameterTests**-luokaan, kuvassa Kuva 29 on esitetty **ParameterTests**-luokan riippuvaisuudet. Parameter structure -testejä varten interaction-portin **IConfigurationContainer**-rajapinnasta luotiin Mock-toteutus (**MockConfigurationContainer**), jota käytettiin testeissä. Jokaiselle parametrille tehtiin Kaksi moduulitestiä, joista toinen testasi parametrin kirjoitus ja toinen parametrin lukua. Molemmissa testeissä tarkistettiin, että **ParameterStructureV0**-

objektin kautta löydetty tietyn id-arvon omaava **IParameter**-objekti vastasi tiettyä parametria **MockConfigurationContainer**-objektissa.



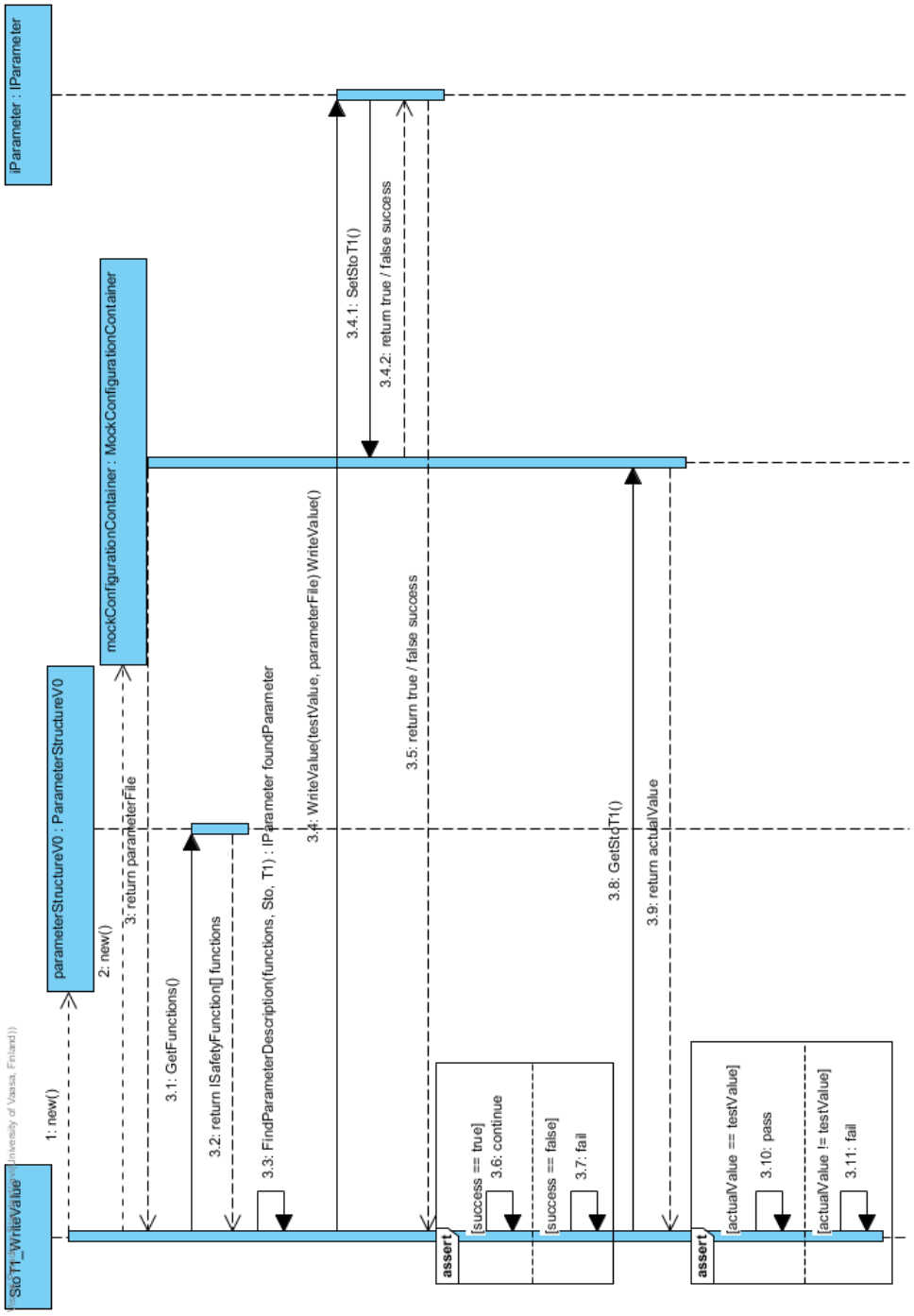
Kuva 29 Parameter structure -komponentin moduulitestien riippuvaisuudet

Kuvissa Kuva 30 ja Kuva 31 on esitetty esimerkki sekvenssit parametrin arvon kirjoittamisen ja lukemisen moduulitesteistä. Sekvenssikaavioista on jätetty pois **FindParameterDescription**-funktion kuvaus koska se toimii testauksen apufunktiona ja sen tehtävä on etsiä annetusta turvafunktiolistasta annetun turvafunktion id:n ja parametrin id:n perusteella testattava parametri. Myös **ISafetyFunction**-rajapinta on jätetty pois tilan säästämisen takia koska rajapinnan funktioita kutsutaan vain **FindParameterDescription**-funktiosta. Parametriarvon kirjoittamisen testauksen sekvenssi oli seuraava:

- Haetaan parametrirakenne **ParameterStructureV0**-objektilta ja etsitään testattava **IParameter**-objekti parametri rakenteesta kohdat 3.1 – 3.3 kuvassa Kuva 30

- Kirjoitetaan parametrin arvo käyttämällä **IParameter**-objektin **WriteValue**-metodia ja tarkistetaan **WriteValue**-metodin paluuarvo kohdat 3.4 – 3.7 kuvassa Kuva 30

- Luetaan parametrin arvo oletetusta **MockConfigurationContainer**-objektin kentästä ja tarkistetaan, että kirjoitettu arvo on päätynyt oikeaan kenttään **MockConfigurationContainer**-objektiin kohdat 3.8 – 3.11 kuvassa Kuva 30



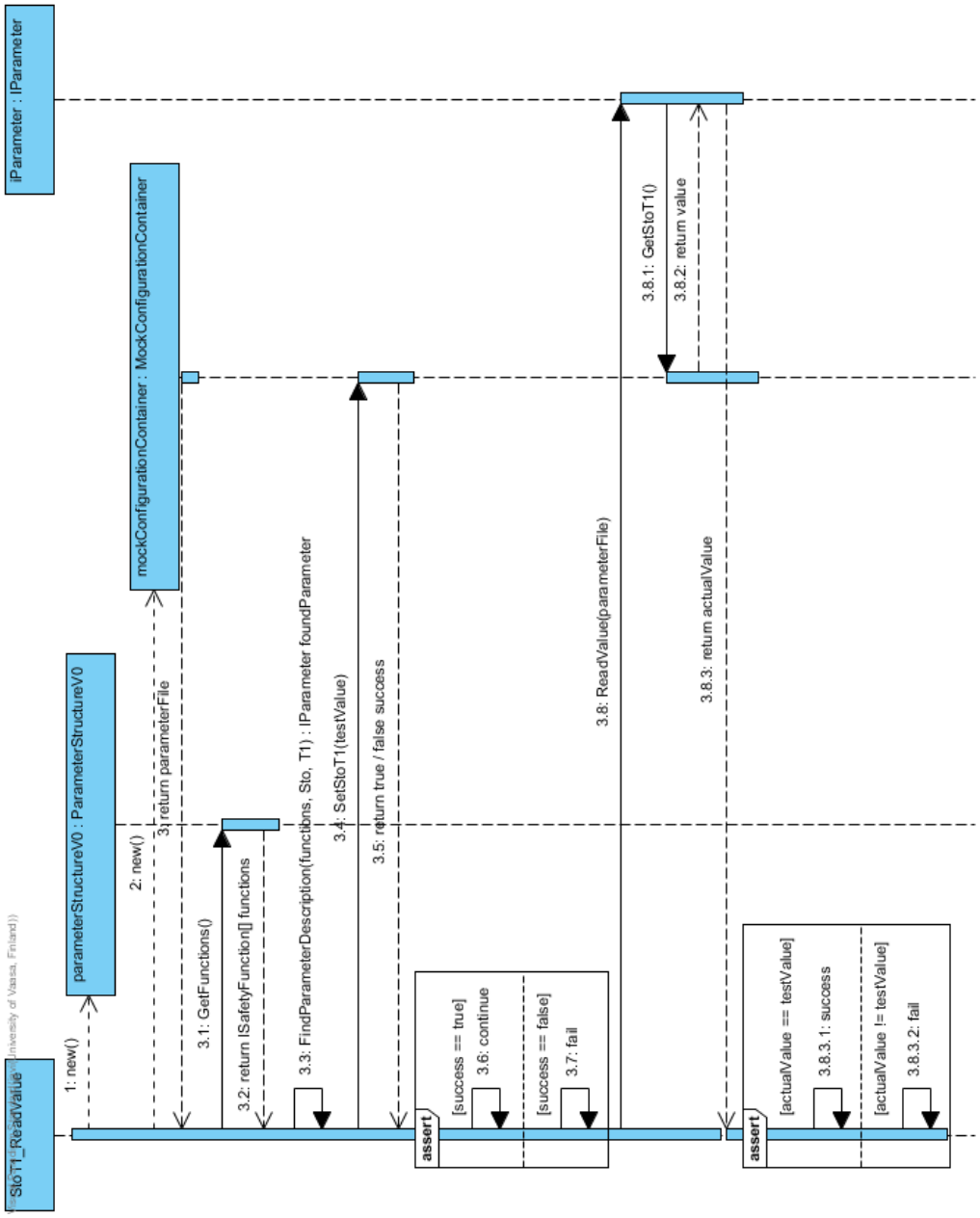
Kuva 30 Esimerkki sekvenssi parametriarvon kirjoittamisen testauksesta

Parametriarvon lukemisen testauksen sekvenssi noudatti samaa kaavaa kuin kirjoittamisen poiketen tästä hieman:

- Kohdat 3.1 ja 3.3 ovat samat kuin kirjoittamisessa.

- Kirjoitetaan arvo oletettuun **MockConfigurationContainer**-objektin kenttään ja tarkistetaan kirjoituksen paluuarvo kohdat 3.4 – 3.7 kuvassa Kuva 31.

- Luetaan parametriarvo **IParameter**-objektin **ReadValue**-funktiota käyttäen ja tarkistetaan että **ReadValue**-funktion palauttama arvo on sama kuin mitä kirjoitettiin **MockConfigurationContainer**-objektiin.



Kuva 31 Esimerkki sekvenssi parametriarvon lukemisen testauksesta

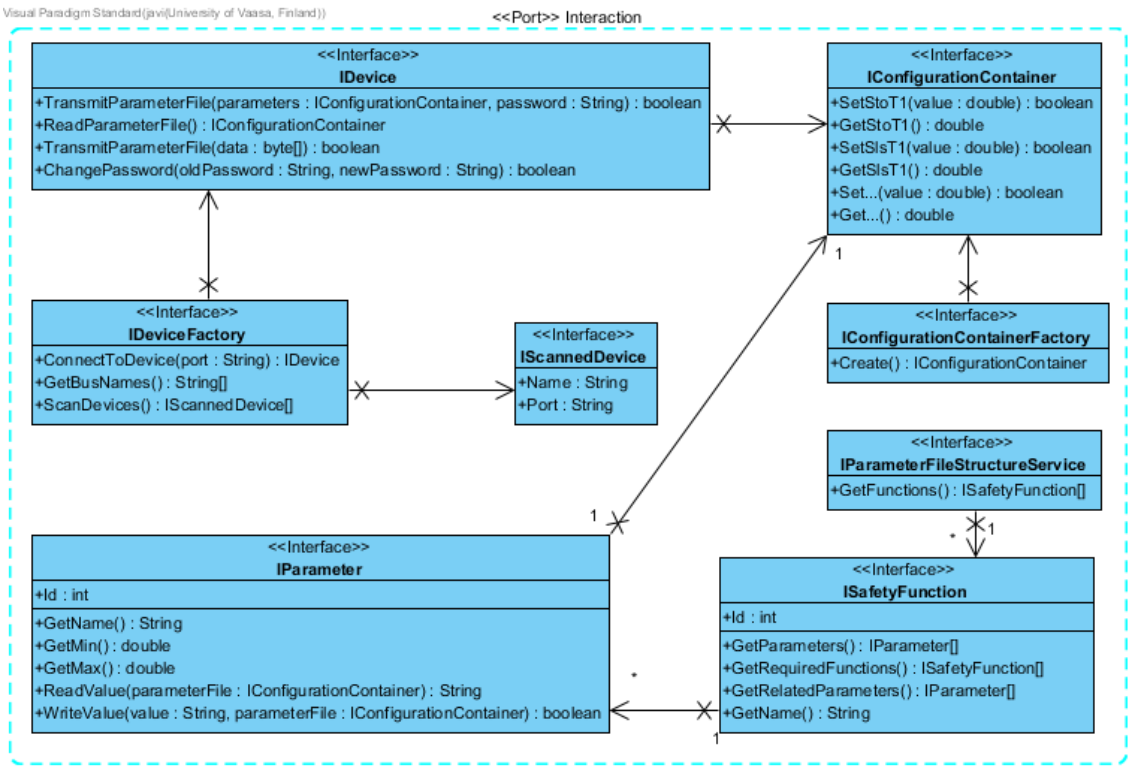
6.2 Toimintojen käsittelijät

Toisen sprintin neljännessä käyttäjätarinassa oli tarkoitus toteuttaa JSON-rajapinnan toiminnot PC-työkalun interaction-portin GUI-sovittimeen. Tavoitteena oli mahdollistaa käyttöliittymän kommunikointi business-logiikan kanssa. Taulukossa Taulukko 2 on esitetty JSON-rajapinnan toiminnot ja niiden kuvaukset.

Taulukko 2 JSON-rajapinnan toiminnot ensimmäisen sprintin aikana

Toiminto	**Kuvaus**
Vaihda salasana	Vaihtaa taajuusmuuttajan salasanan.
Yhdistä laitteeseen	Muodostaa yhteyden taajuusmuuttajaan.
Vie parametritiedosto	Siirtää PC-työkalulla generoidun turvafunktioparametritiedoston taajuusmuuttajalle.
Tuo parametritiedosto	Hakee taajuusmuuttajalla olevan parametritiedoston.
Hae yhteysväylät	Palauttaa käytettävissä olevat taajuusmuuttajan kanssa kommunikointiin käytettävät kommunikaatioväylät.
Hae parametritiedoston kuvaus	Palauttaa parametritiedoston rakenteen, jonka avulla käyttöliittymän näkymät rakennetaan.
Etsi laitteita	Etsii taajuusmuuttajat joihin yhteys voidaan muodostaa.

JSON-rajapinnan toimintojen toteuttaminen vaati muutoksia interaction-portin rajapintoihin **IDevice** ja **IDeviceFactory.** kuvassa Kuva 32 on esitetty lisätyt rajapinnat ja niiden metodit. Taulukossa Taulukko 3 on esitetty JSON-rajapinnan toimintojen käyttämät metodit interaction-portin rajapinnoista.

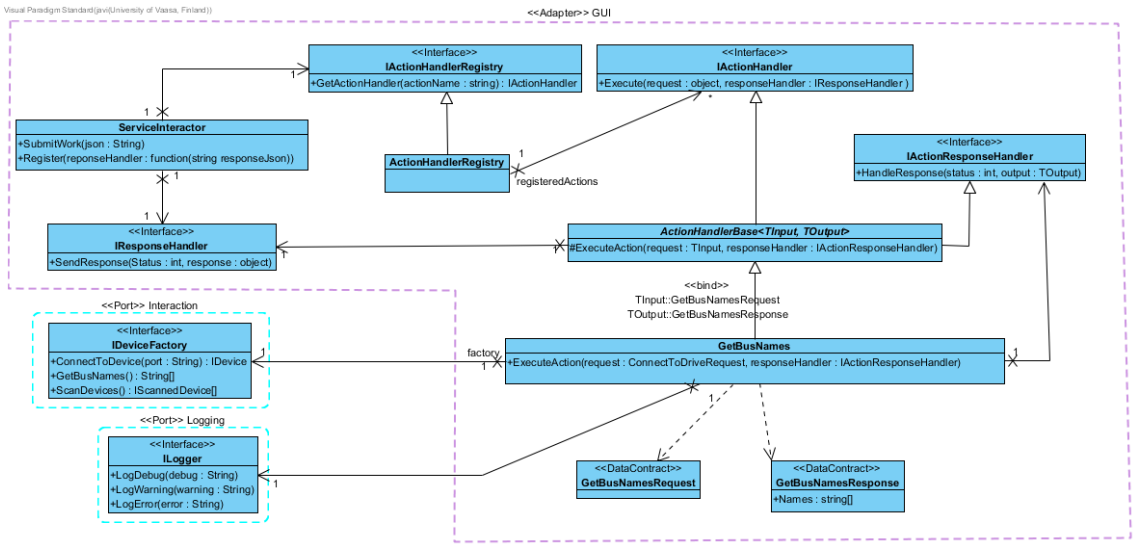


Kuva 32 IDevice ja IDeviceFactory -rajapintojen muutokset.

Taulukko 3 JSON-rajapinnan toimintojen käyttämät interaction-portin metodit.

Toiminto	**Rajapinta**	**Metodi**
Vaihda salasana	IDevice	ChangePassword(...)
Yhdistä laitteeseen	IDeviceFactory	ConnectToDevice(...)
Vie parametritiedosto	IDevice	TransmitParameterFile(parameters..,)
	IParameterFileStructureS…	GetFunctions(…)
	IConfigurationContainerFact…	Create(…)
	IParameter	WriteValue(…)
Tuo parametritiedosto	IDevice	ReadParameterFile(...)
	IParameterFileStructureS…	GetFunctions(…)
	IParameter	ReadValue(…)
Hae yhteysväylät	IDeviceFactory	GetBusNames(...)
Hae parametritiedoston kuvaus	IParameterFileStructureS…	GetFunctions(…)
Etsi laitteita	IDeviceFactory	ScanDevices(...)

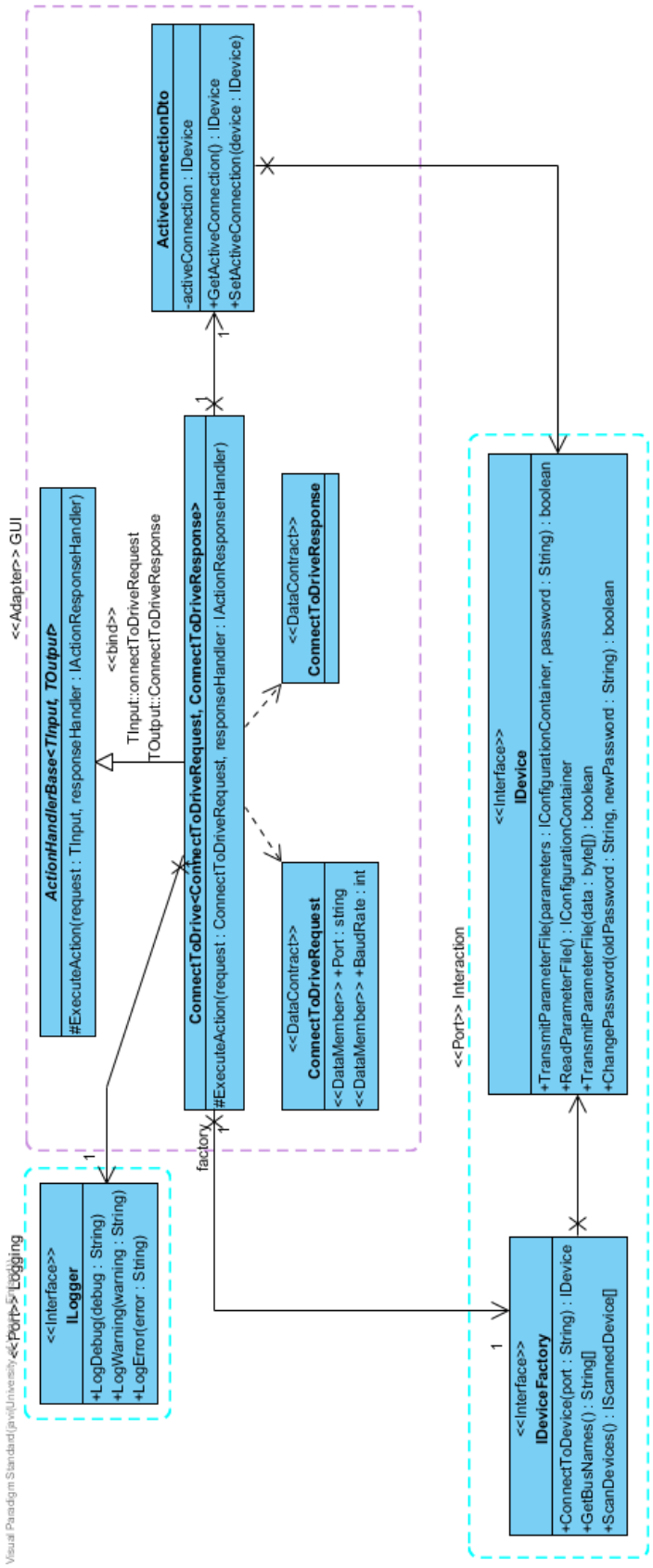
GUI-sovittimen Service Facade -komponentissa oli valmius uusien toimintojen käsittelijöiden (Actions) toteuttamiseen periyttämällä uudet toiminnot **ActionhandlerBase** luokasta. Kuvassa Kuva 33 on esitetty yksityiskohtaisempi kuva Service Facade -komponentista. **ActionhandlerBase**-luokka tarjoaa toiminnoille mahdollisuuden paluuarvon välittämiselle takaisin Javascript käyttöliittymälle **IResponseHandler**-rajapinnan kautta.



Kuva 33 Service Facade -komponentin luokkakaavio.

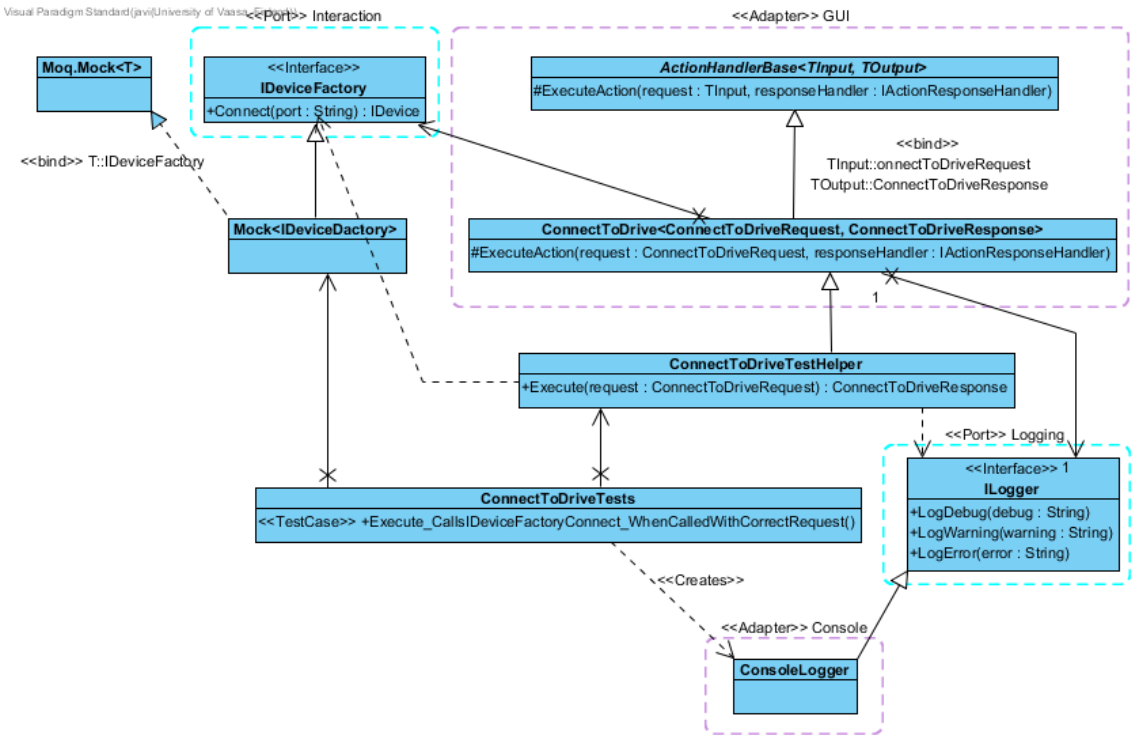
Käyttöliittymältä saapuvat toimintojen suorituspyynnöt Saapuvat **ServiceInteractor**-luokalle, jonka vastuulla on aloittaa toiminnon suorittaminen **IActionHandlerRegistry**-rajapinnan avulla. **IActionHandlerRegistry**-rajapinnan toteutuksen vastuulla on pitää kirjaa kaikista toiminnoista, joita on mahdollista kutsua JSON-rajapinnan kautta. Jokaiselle JSON-rajapinnan toiminnolle määritettiin vastaava **ActionHandlerBase**-luokasta periytyvä toteutus ja määritettiin luokat, jotka kuvaavat pyyntö ja vastaus dto-objektit. Kuvassa Kuva 34 on esitetty yhdistä laitteeseen JSON-rajapinnan toiminnon toteutuksen luokkakaavio.

Yhdistä laitteeseen toiminto toteutettiin **ConnectToDrive**-luokkaan, kuvassa Kuva 34 on esitetty **ConnectToDrive**-luokan riippuvuudet. **ConnectToDrive**-luokan tehtävänä on luoda yhteys käyttäen interaction-portin **IDeviceFactory**-rajapintaa ja välittää **Connect**-funktion palauttama **IDevice**-rajapinta **ActiveConnectionDto**-luokalle.



Kuva 34 **ConnectToDrive**-luokan riippuvuudet.

Kuvassa Kuva 35 on kuvattu **ConnectToDrive**-luokan moduulitestien toteutus. Moduulitestit toteutettiin **ConnectToDriveTests**-luokkaan. **ConnectToDrive**-luokka on riippuvainen **IDeviceFactory** ja **ILogger** -rajapinnoista. Logging-portille lisättiin Console-sovitin, jotta testien aikainen diagnostiikka data saatiin kaapattua talteen. Moduulitesteissä. **IDeviceFactory**-rajapinnan toteutus korvattiin Moq-kirjaston tynkä toteutuksella ja **ILogger**-rajapinnan toteutuksena käytettiin **ConsoleLogger**-luokkaa console-sovittimesta.



Kuva 35 **ConnectToDrive**-luokan moduulitestien toteutus.

ConnectToDrive-luokasta periytettiin **ConnectToDriveTestHelper**-luokka. **ConnectToDriveTestHelper**-luokan tehtävä on paljastaa **ExecuteAction**-metodi testeille. **ConnectToDriveTests**-luokka luo **ConnecToDriveTestHelper**-objektin välittäen **IDeviceFactory**-rajapinnan tynkä toteutuksen ja **ConsoleLogger**-objektin **ConnectToDrive**-objektille. **ConnecToDriveTestHelper**-objektin luonnin jälkeen objekti on valmis testattavaksi. Moduulitestit määritettiin julkisiksi funktioiksi **ConnectToDriveTests**-luokkaan. Moduulitestien nimeämisessä sovellettiin Microsoftin suosituksia. Microsoftin (Microsoft 2011) suosituksissa moduulitestin nimen tulisi sisältää neljä elementtiä,

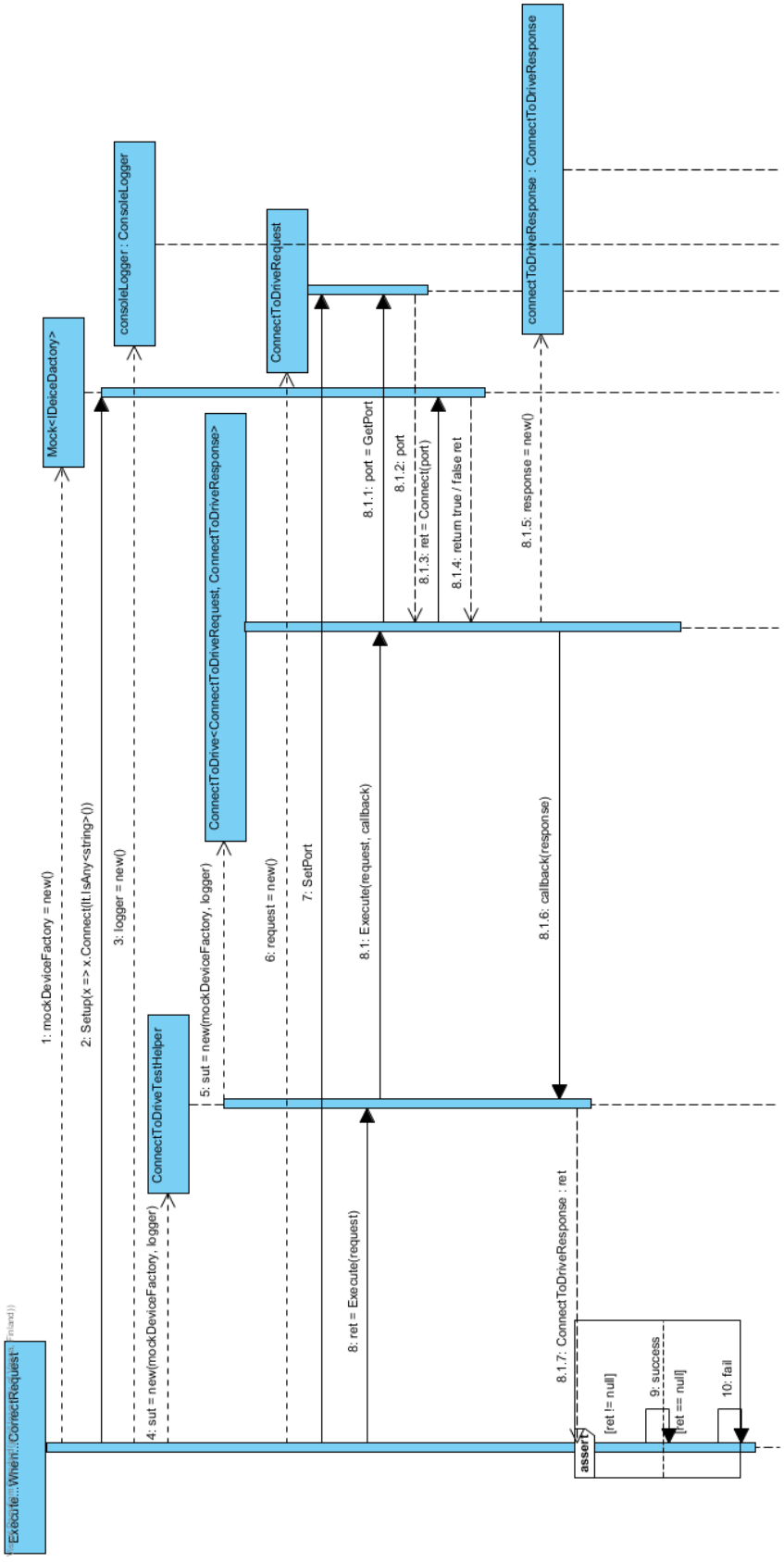
jotka ovat luokan nimi, testattavat funktion nimi, odotettu käytös ja skenaario. Microsoftin suosituksista poiketen luokan nimi jätettiin pois testien nimistä, koska sitä ei nähty tarpeelliseksi. Muiden JSON-toimintojen moduulitestit toteutettiin vastaavalla tavalla, toiminto luokkien toteutusten riippuvuudet korvattiin Moq-kirjaston tynkä toteutuksilla ja **ILogger**-rajapinnan toteutuksena käytettiin aina **ConsoleLogger**-luokkaa.

Kuvassa Kuva 36 on esitetty yhden **ConnectToDrive**-luokan testin sekvenssi. Testin vaiheet ovat seuraavat:

- Luodaan tarvittavat **ConnectToDrive**-riippuvuudet ja konfiguroidaan **IDeviceFactory**-rajapinnan mock-toteutus kohdat 1 – 3.

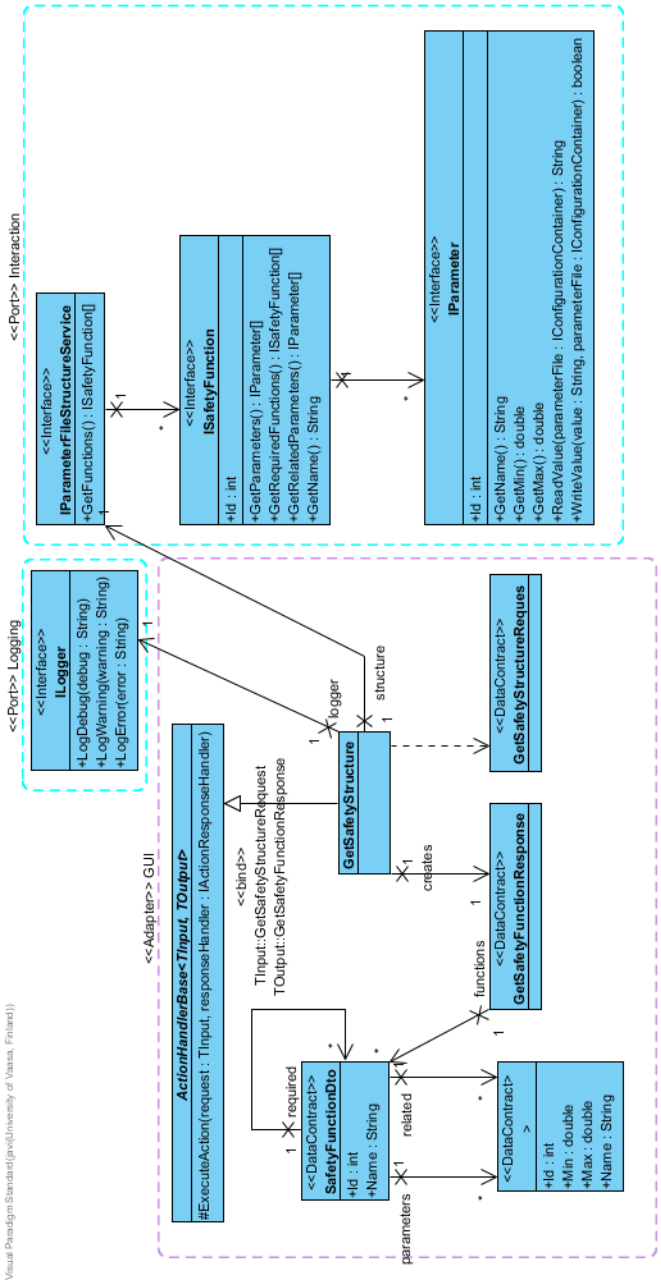
- Luodaan **ConnectToDriveTestHelper**-objekti, **ConnectToDriveRequest**-objekti ja asetetaan portti parametri **ConnectToDriveRequest**-objektille kohdat 4 – 7.

- Kutsutaan **Execute**-metodia **ConnectToDriveTestHelper**-objektilta ja annetaan **ConnectToDriveRequest**-objekti parametrina, tarkastetaan, että funktion paluuarvo ei ole null-tyyppinen kohdat 8 – 10.



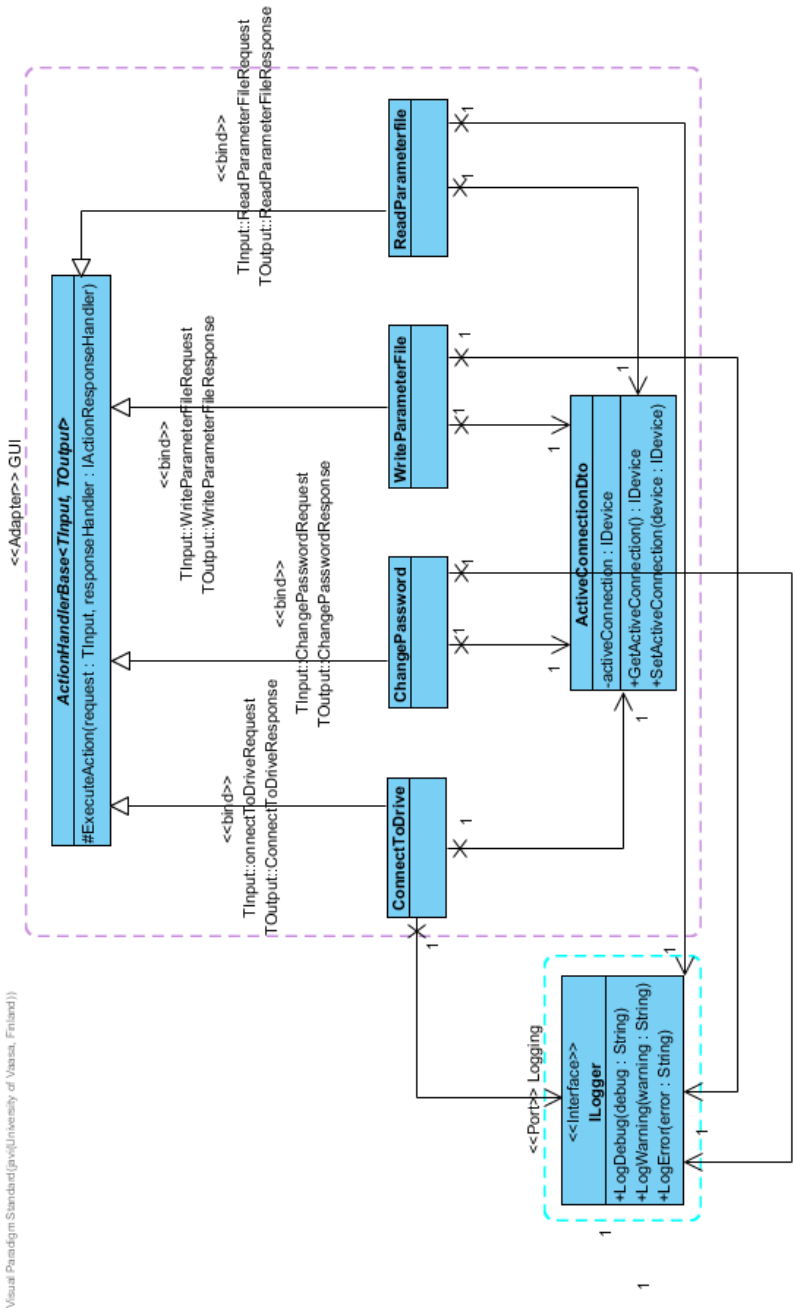
Kuva 36 ConnectToDrive-testin sekvenssi.

Hae parametritiedoston kuvaus toiminto toteutettiin **GetSafetyStructure**-luokkaan, **GetSafetyStructure**-luokan riippuvuudet on esitetty kuvassa Kuva 37. **GetSafetyStructure**-luokan tehtävä on hakea parametritiedoston kuvaus käyttäen interaction-portin **IParameterStructureService**-rajapintaa, muuntaa parametritiedoston rakenne vastaus Dto-objekteiksi ja palauttaa vastaus Dto-objekti rakenne **Service Facade**-komponentille.



Kuva 37 GetSafetyStructure-luokan riippuvuudet.

Muut toiminnot toteutettiin vastaavalla tavalla, eli toiminto käsittelee pyyntö dto -objektin, käyttää jotain metodeja interaction-portin rajapinnoista, tallentelee ja tai muuttaa jonkin sisäisen Dto-objektin tilaa ja palauttaa vastauksen Service Facade-komponentille. Toiminnot eivät ole riippuvaisi keskenään vaan tarvittavan informaation välitys toimintojen käsittelijöiden välillä hoidettiin Dto-objekteilla.



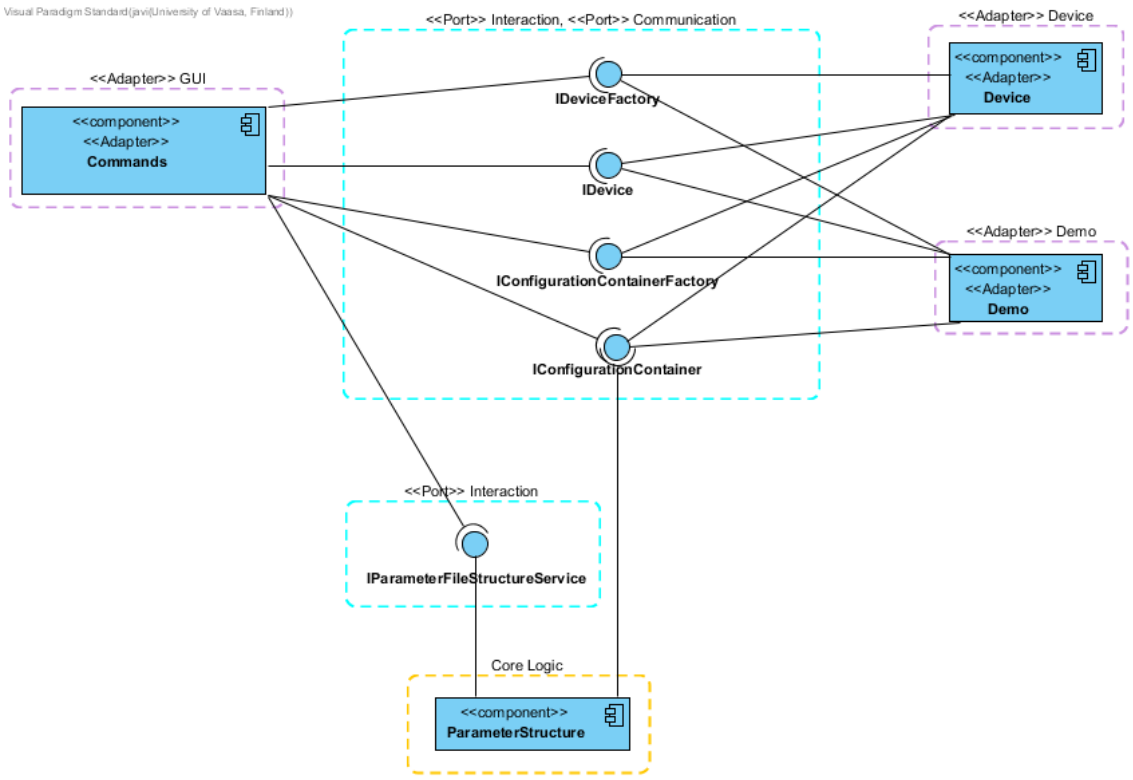
Kuva 38 Toimintojen käsittelijöiden kommunikointi Dto-objektien kautta

Kuvassa Kuva 38 on esitetty toimintojen käsittelijöiden riippuvuudet Dto-objekteista. Projektin tässä vaiheessa toimintojen käsittelijät jakoivat vain yhden Dto-objektin **ActiveConnectionDto**, jonka tehtävä on välittää aktiivista yhteyttä toimintojen käsittelijöiden kesken.

Toimintojen käsittelijöiden toteutukset GUI-sovittimessa ovat helposti muunneltavissa, koska niillä ei ole riippuvaisuuksia toisiinsa ja ne ovat riippuvaisia vain Interaction portin rajapinnoista. GUI-sovitin komponentti riippuu vain Interaction portista, joten GUI-sovitin on myös komponentti tasolla helposti muunneltava. Toimintojen käsittelijät ovat myös helposti moduulitestattavissa, koska niiden toteutukset riippuvat Dto-objekteista ja interaction-portin rajapinnoista.

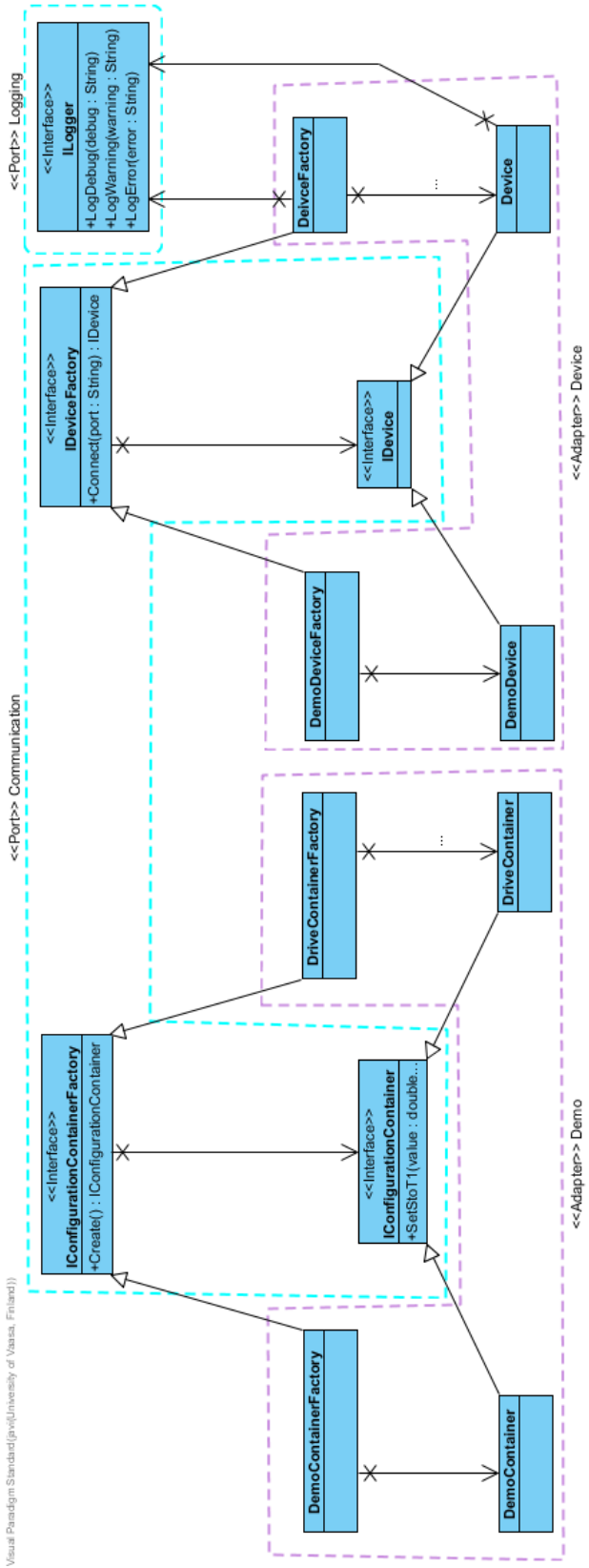
6.3 Demo datan generointi

Toisen sprintin viidennessä käyttäjätarinassa oli tarkoitus toteuttaa business logiikkaan demo datan generointi ja demo datan siirtäminen **JSON**-rajapinnan toimintojen kautta käyttöliittymälle. Tarkoitus oli simuloida taajuusmuuttajan toimintaa business logiikassa, jotta PC-työkalun testaaminen olisi mahdollista ilman yhteyttä taajuusmuuttajaan. Tavoite oli myös, että business logiikassa olisi helppo vaihdella toteutusta demo toteutuksen ja taajuusmuuttajan toteutuksen välillä. Demo ja taajuusmuuttajan -toteutuksen välinen helppo vaihdettavuus ratkaistiin määrittämällä communication-porttiin Demo-sovitin. Kuvassa Kuva 39 **JSON**-rajapinnan toimintojen käsittelijät, jotka on toteutettu **Commands**-komponenttiin käyttävät interaction-portin **IConfigurationContainerFactory** ja **IDeviceFactory** -rajapintoja, eivätkä ole tietoisia siitä kumpi sovittimista on käytössä. Tässä vaiheessa projektia communication ja interaction -portit olivat vielä niin lähellä toisiaan, ettei nähty tarpeelliseksi lähteä toteuttamaan kerrosta niiden väliin.



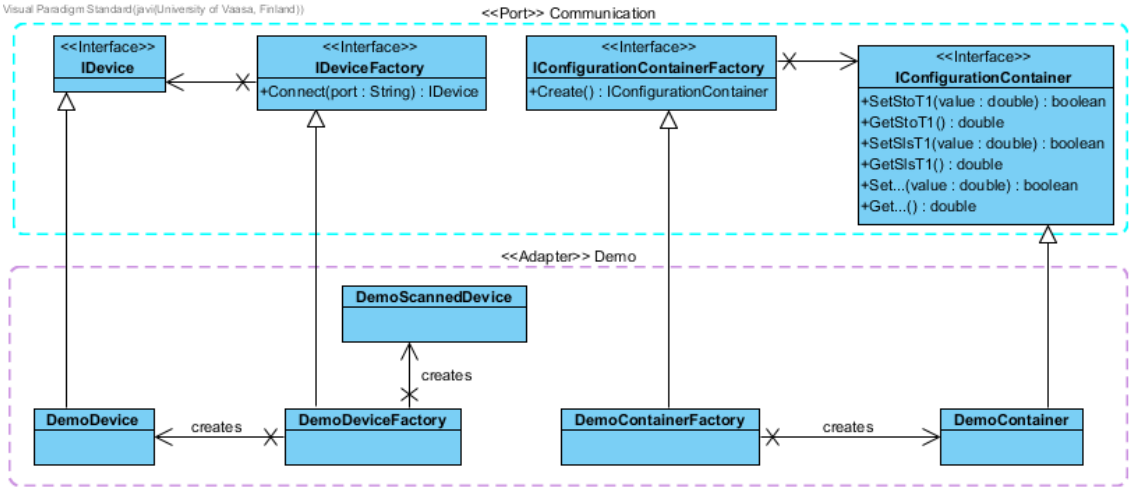
Kuva 39 Demo ja taajuusmuuttajan -toteutukset ja rajapinnat.

IDeviceFactory ja **IConfigurationContainerFactory** -rajapintojen toteutukset **Demo** ja **Device** -sovittimissa luovat eri toteutuksia **IDevice** ja **IConfigurationContainer** -rajapinnoista kuvan Kuva 40 mukaisesti. Kuvassa Kuva 40 **IDevice**-rajapinta määrittää kommunikaatio operaatiot taajuusmuuttajalle kuten parametritiedoston kirjoittamisen ja lukemisen ja **IConfigurationContainer** -rajapinta määrittää operaatiot parametritiedoston käsittelyyn, kuten yksittäisen parametrin arvon kirjoittamisen ja lukemisen.



Kuva 40 IDeviceFactory ja IConfigurationContainerFactory toteutukset Demo ja Device -sovittimissa.

Kuvassa Kuva 41 on esitetty Demo-sovittimen toteutus.



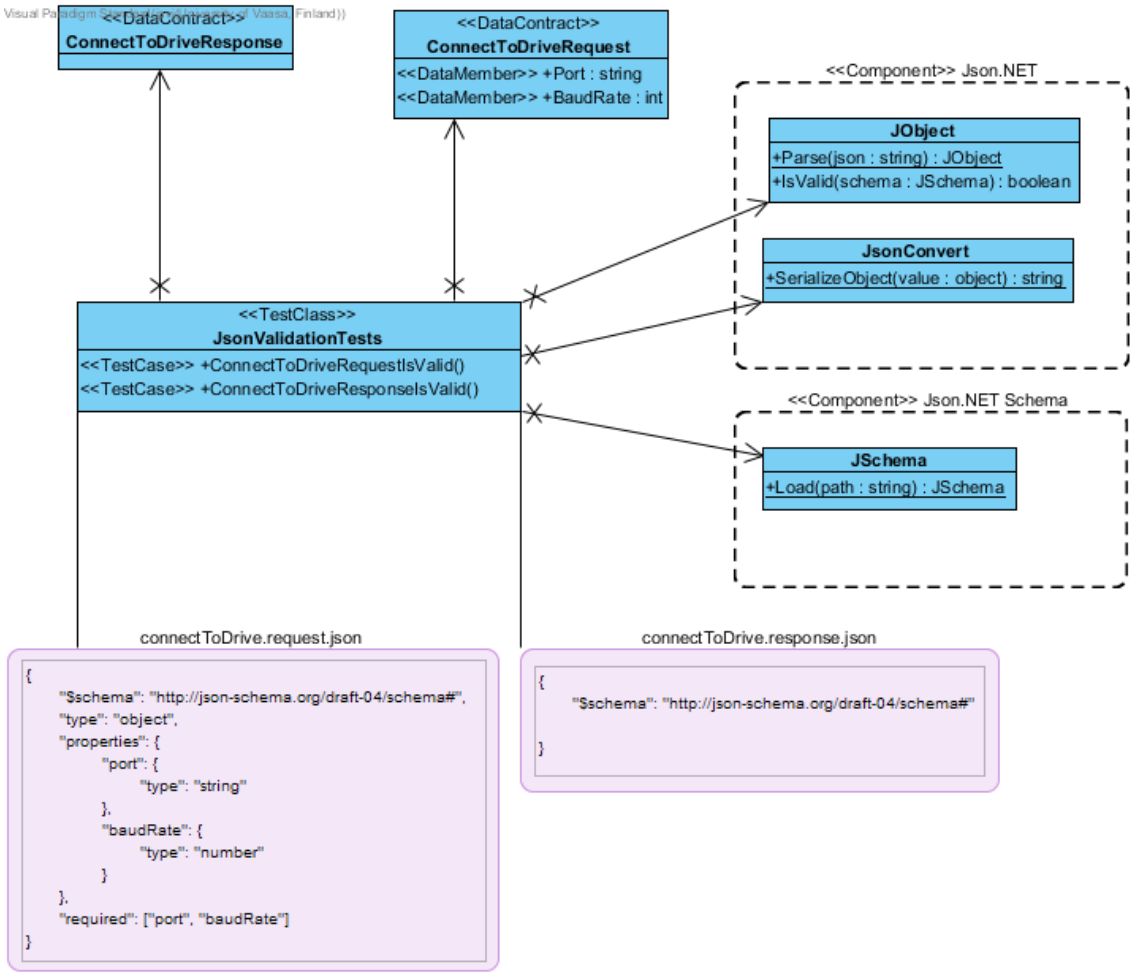
Kuva 41 Demo-sovittimen toteutus

Demo-sovittimen sisäinen toteutus on sidottu tiukasti yhteen, josta seuraa, että komponentin sisäisen toteutus on vaikeasti muunneltava. Komponentti riippuu vain communication portista, joten Demo-sovitin on komponentti tasolla helposti muunneltava.

6.4 JSON-rajapinnan validointi

Kolmannen sprintin ensimmäisessä käyttäjätarinassa oli tarkoitus kehittää tapa, jolla **JSON**-rajapinnan yhteensopivuus pystyttäisiin varmistamaan business logiikan ja käyttöliittymän välillä. **JSON**-rajapinnan yhteensopivuus käyttöliittymän ja business logiikan välillä varmistettiin määrittämällä **JSON**-rajapinnan toiminnoista rajapintakuvaukset. Toimintojen rajapintakuvaukset sisälsivät toiminnon paluuarvon kuvauksen ja toiminnon parametrien kuvaukset. Business logiikkaan toteutettiin moduulitestit jokaiselle **JSON**-rajapinnan toiminnon pyyntö ja vastaus -dto-objektille. Moduulitestit varmistivat, että business logiikan toimintojen paluuarvot ja parametrit vastasivat toimintojen rajapintakuvauksia. **JSON**-rajapinnan yhteensopivuuden varmistavia moduulitestejä kutsuttiin validaatio-testeiksi. Kuvassa Kuva 42 on esitetty **JSON**-rajapinnan validaatiotestien periaate.

Toimintojen rajapintakuvausten formaatiksi valittiin **JSON**-schema formaatti, koska useimmat **JSON**-kirjastot tukevat **JSON**-rakenteen eheyden varmistamista **JSON**-schema formaatin perusteella. **JSON**-schema formaatilla kuvataan haluttu **JSON**-rakenne.



Kuva 42 JSON-rajapinnan validaatio-testien periaate.

Toimintojen pyyntö ja vastaus dto-objektit testattiin kuvan Kuva 43 mukaisella sekvenssillä. Kaikkien pyyntö ja vastaus dto-objektien testaus noudatti samaa sekvenssiä yhdistä laitteeseen dto-objektien testauksen kanssa. Yhdistä laitteeseen dto-objektin testauksen sekvenssi oli seuraava:

1. Luo dto-objekti (ConnectToDriveRequest).

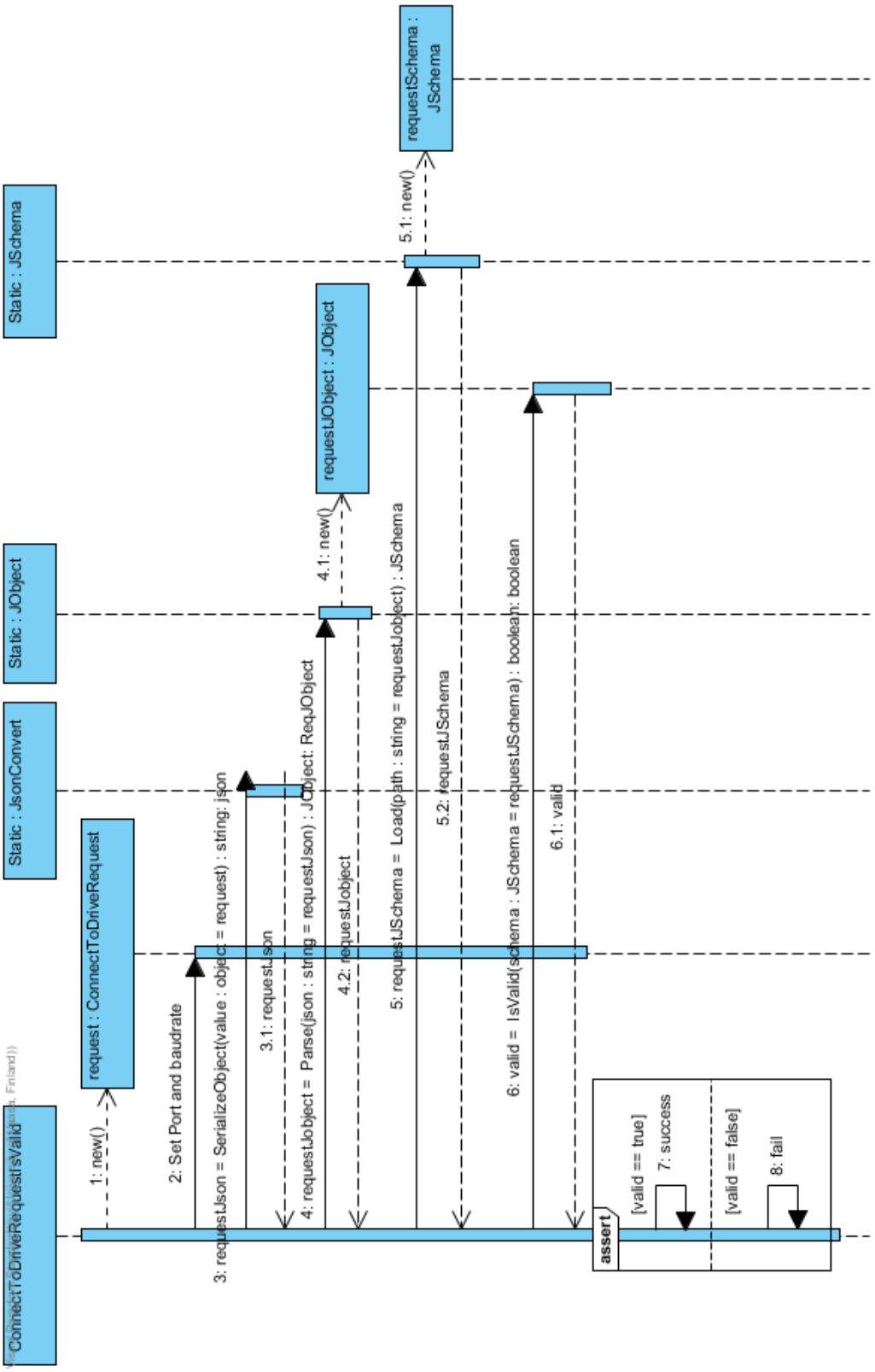
2. Aseta tarvittavat arvot dto-objektille.

3. Muunna dto-objekti JSON muotoiseksi tekstiksi.

4. Muunna dto-objektin JSON muotoinen teksti JObject-objektiksi.

5. Lataa dto-objektin JSON-schema.

6. Tarkasta dto-objektin eheys JObject-objektin IsValid metodia käyttäen.

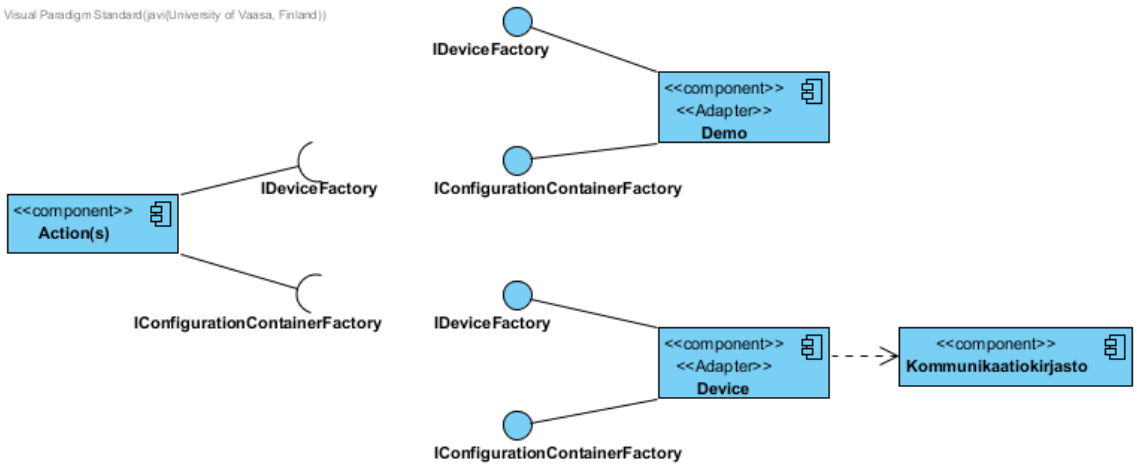


Kuva 43 Yhdistä laitteeseen pyyntö dto-objektin testaus

6.5 Kommunikaatiokirjaston integraatio

Kolmannen sprintin Neljännessä käyttäjätarinassa oli tarkoitus integroida kommunikaatiokirjasto PC-työkaluun. Käyttäjätarinan tavoitteena oli mahdollistaa PC-työkalu käyttöliittymältä luodun parametritiedoston siirtäminen taajuusmuuttajalle ja taajuusmuuttajalta ladatun parametritiedoston näyttäminen ja muokkaaminen käyttöliittymällä.

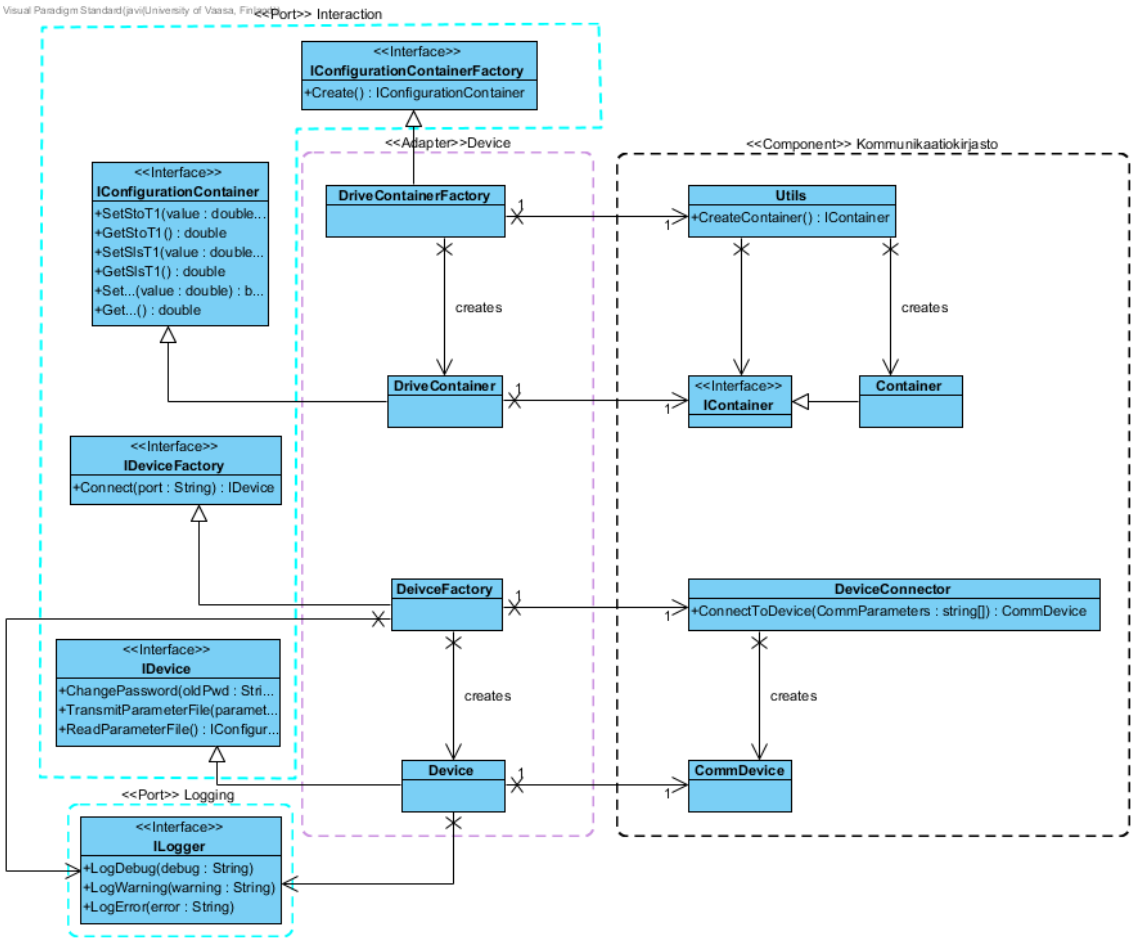
Kommunikaatiokirjasto sisälsi taajuusmuuttajan protokolla toteutuksen, jonka avulla PC-työkalun taajuusmuuttajaa käyttävät toiminnot toteutettiin. Kommunikaatiokirjasto integroitiin PC-työkaluun kuvassa Kuva 44 näkyvään **Device**-sovittimeen.



Kuva 44 Kommunikaatiokirjaston integraatioperiaate.

Kuvassa Kuva 45 on esitetty tarkempi kuvaus **Device**-sovittimesta ja sen riippuvaisuuksista kommunikaatiokirjastoon. **DriveContainerFactory**, **DriveContainer**, **DeviceFactory** ja **Device** -luokat toteutettiin sovittimiksi, joiden avulla kommunikaatiokirjaston toteutukset integroitiin business logiikkaan. Device-sovittimen sisäinen toteutus on sidottu tiukasti yhteen kommunikaatiokirjaston kanssa, joten sovittimen ulkoinen muunneltavuus jäi välttävälle tasolle. Muutokset kommunikaatiokirjaston aiheuttavat hyvin todennäköisesti muutoksia Device-sovittimeen. Sisäinen muunneltavuus jäi kuitenkin hyvälle tasolle, koska esimerkiksi muutokset device-sovittimen **Device** ja **DriveCon-**

tainer -luokkiin eivät aiheuta muutoksia muihin Device-sovittimen luokkiin, koska niihin ei viitata suoraan mistään komponentin sisällä.

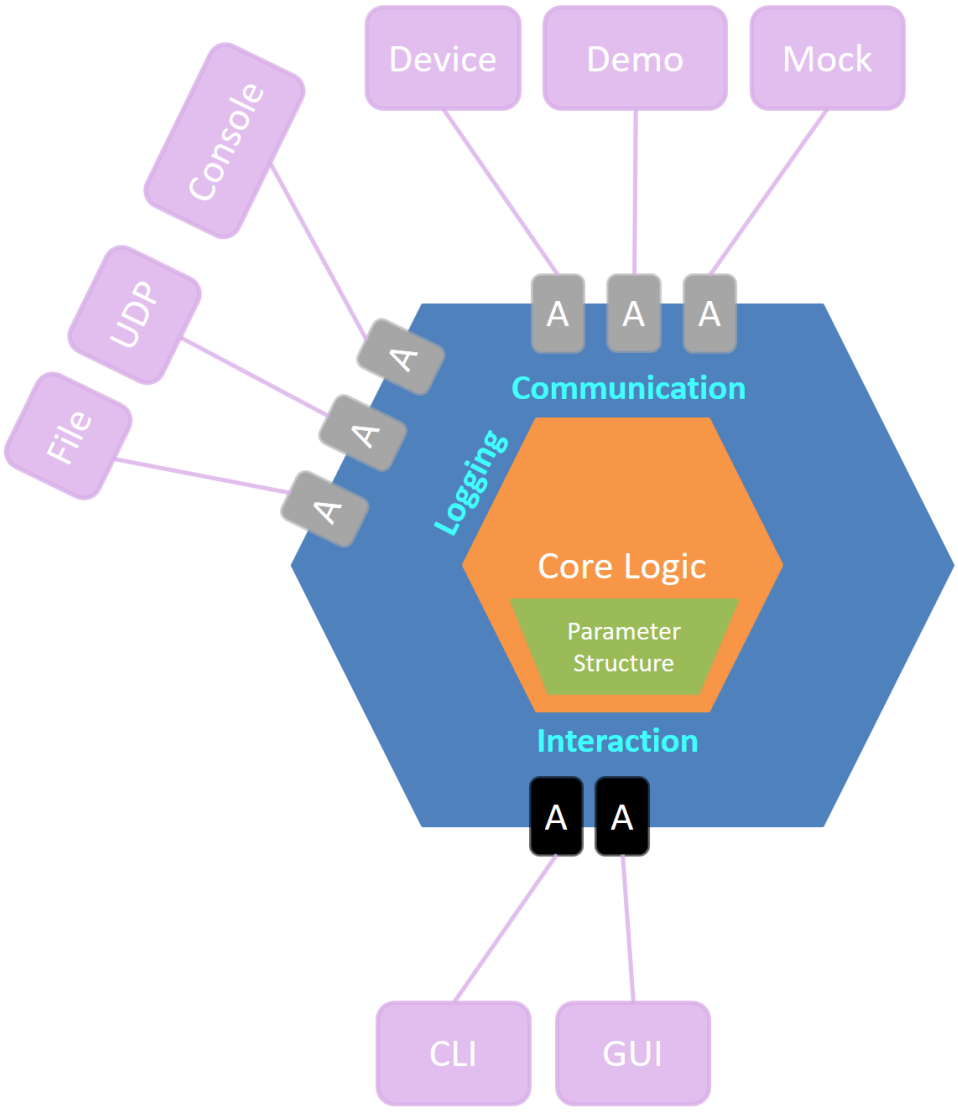


Kuva 45 Kommunikaatiokirjaston riippuvuudet Device-sovittimessa.

6.6 Tulokset

Analysoitujen käyttäjätarinoiden tuloksiksi saatiin toimiva integraatio käyttöliittymän, business-logiikan ja taajuusmuuttajan välille. GUI-sovittimeen toteutettiin toimintojen käsittelijät JSON-rajapinnan toiminnoille. Communication-porttiin toteutettiin Demo-sovitin ja Device-sovitin. Demo-sovitin generoi käyttöliittymälle demo dataa simuloi-

malla taajuusmuuttajan toimintaa, joka mahdollisti PC-työkalun käyttöliittymän testaamisen business logiikan tuottamalla demo datalla. Device-sovittimen tehtävä oli mahdollistaa parametritiedoston siirtäminen PC-työkalun ja taajuusmuuttajan välillä. Moduulitestien määritysten yhteydessä Communication-portin sovittimena käytettiin Mock-sovitinta, jonka toteutus generoitiin ajon aikaisesti käyttäen Moq-kirjastoa. Core Logic -komponenttiin toteutettiin ensimmäinen versio parametritiedoston rakenteesta. Logging-porttiin toteutettiin Console-sovitin, joka mahdollisti diagnostiikka datan keräämisen testejä ajettaessa.



Kuva 46 Arkkitehtuuri käyttäjätarinoiden toteutuksen jälkeen.

Taulukkoon Taulukko 4 on kerätty toteutettujen komponenttien sisäiset ja ulkoiset muunneltavuudet. Kuvasta Kuva 46 nähdään komponenttien väliset riippuvaisuudet GUI-sovitin käyttää Interaction -porttia, eikä riipu muista komponenteista. Demo-sovitin toteuttaa Communication-portin eikä riipu muista komponenteista. Device-sovitin toteuttaa Communication-portin ja käyttää kommunikaatiokirjasto-komponenttia. Parameter Structure -komponentti toteuttaa Integration-portin rajapintoja ja käyttää Communication-porttia. Demo-sovittimen ja Parameter Structure -komponentin sisäiset toteutukset olivat tiukasti sidottu toisiinsa, josta seurasi, että Demo-sovittimen ja Parameter Structure -komponentin sisäinen muunneltavuus jäi huonolle tasolle. GUI-sovittimen sisäinen toteutus oli hyvin eroteltu, joka puolestaan johti siihen, että komponentin sisäinen muunneltavuus jäi hyvälle tasolle. Device-sovittimen sisäinen muunneltavuus jäi hyvälle tasolle, koska sen sisäisillä luokilla on hyvin vähän riippuvaisuuksia keskenään. Device-sovittimen ulkoinen muunneltavuus jäi välttävälle tasolle, koska se on eriytetty muusta sovelluksesta rajapintojen avulla, mutta on kuitenkin samanaikaisesti tiukasti sidottu kommunikaatiokirjastoon. Kaikkien toteutettujen komponenttien paitsi Device-sovittimen ulkoinen muunneltavuus saavutti hyvän tason, koska komponentit eivät ole riippuvaisia toisistaan, vaan riippuvat ainoastaan Core Logic -komponenttiin määritetyistä rajapinnoista. Taulukon Taulukko 4 voidaan tulkita, että portit ja adapterit -arkkitehtuurin käytöllä saavutettiin lähes hyvä ulkoinen muunneltavuus ja välttävä sisäisen muunneltavuus, josta voidaan päätellä, että portit ja adapterit -arkkitehtuurin käyttö parantaa ulkoista muunneltavuutta.

Taulukko 4 Komponenttien sisäinen ja ulkoinen muunneltavuus

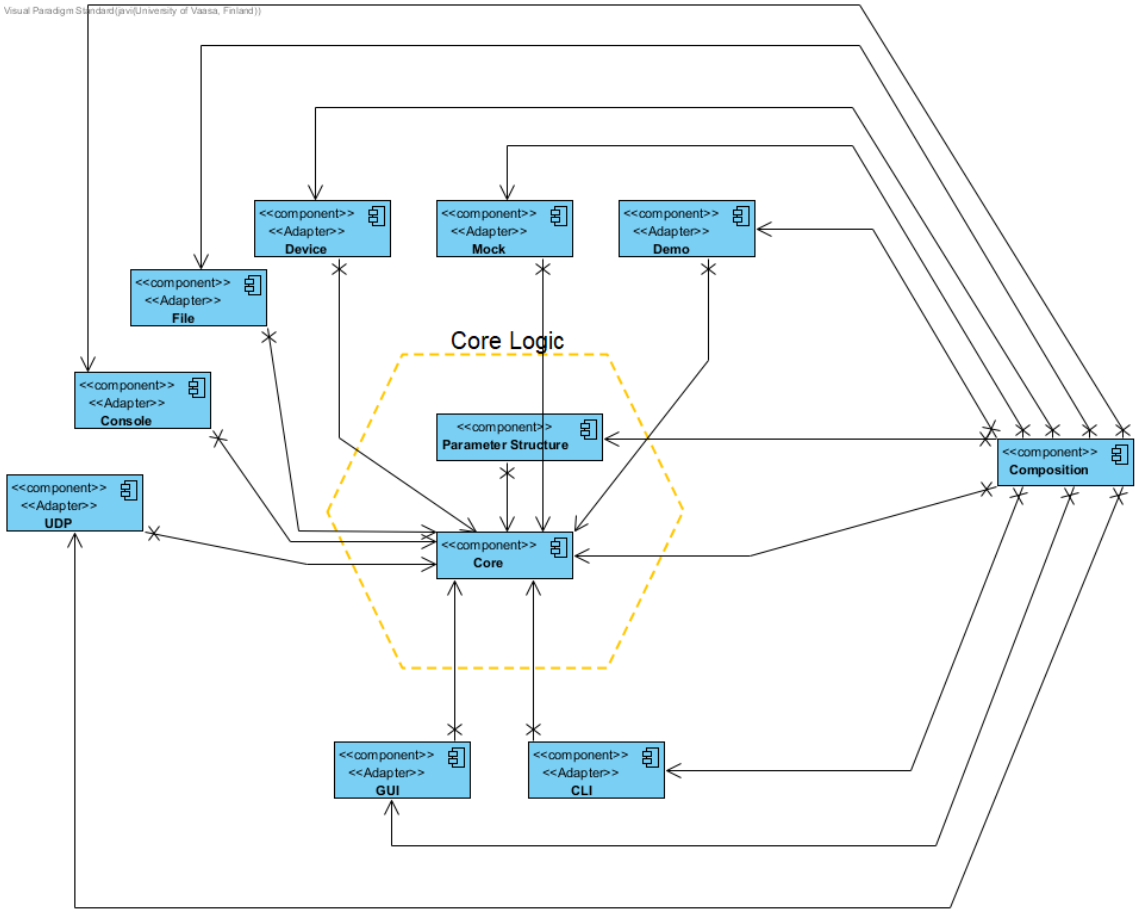
Komponentti	**Sisäinen muunneltavuus**	**Ulkoinen muunneltavuus**
GUI-sovitin	Hyvä	Hyvä
Demo-sovitin	Huono	Hyvä
Parameter structure -komponentti	Huono	Hyvä
Device-sovitin	Hyvä	Välttävä

Toiseen tutkimuskysymykseen pyrittiin löytämään vastauksia kuvan Kuva 47 avulla. Kuvassa Kuva 47 on esitetty komponenttien väliset riippuvaisuudet, mutta kuvasta on jätetty pois sovittimien riippuvuudet kolmannen osapuolen komponentteihin selkeyden vuoksi. Todellisuudessa kaikki muut sovittimet paitsi Demo-sovitin riippuvat muista komponenteista, esimerkiksi Device-sovitin riippuu kommunikaatiokirjastosta, CLI-sovitin riippuu Windowsin komentorivi integraatioista ja niin edelleen. Alla olevasta kuvasta voidaan tehdä seuraavia havaintoja:

1. Kaikki komponentit riippuvat (viittaavat) Core-komponenttiin, jossa on määritettynä abstraktiot, joita muut komponentit toteuttavat tai käyttävät.

2. Kaikki komponentit viittaavat ulkoa sisäänpäin, jos huomioon ei oteta sovittimien riippuvaisuuksia kolmannen osapuolen komponentteihin.

3. Ainoastaan Composition-komponentti on riippuvainen kaikista komponenteista, koska sen tehtävänä on rakentaa koko sovellus. Sovelluksen rakentaminen tarkoittaa toteutusten sitomista abstraktioihin ja abstraktio viittausten välittämistä muille komponenteille. Composition-komponentin tehtävä on esimerkiksi päättää, mitä kolmesta Logging-portin toteutuksesta käytetään File, Console vai UDP.



Kuva 47 Komponenttien väliset riippuvuudet

Vertailemalla yllä mainittuja havaintoja Robertin (Martin 2003: 95–135, 253–268) käsittelemiin ketteriin suunnitteluperiaatteisiin huomataan, että portit ja adapterit -arkkitehtuurista löytyy yhtenäisyyksiä SDP ja DIP suunnitteluperiaatteisiin, jotka Robertin (Martin 2003: 134, 264) mukaan parantavat muunneltavuutta.

Stable Dependencies -suunnitteluperiaate (SDP) suosittelee, että komponentit ovat riippuvaisia vakaammista komponenteista, joka on yhtenäinen toisen yllä mainitun havainnon kanssa, jossa komponentit viittaavat ulkoa sisäänpäin. SDP vastaa ulkopuolelta sisäänpäin viittaamista, koska kuvan Kuva 47 mukaisessa arkkitehtuurissa ytimen keskelle päätyvät keskeisimmät rajapinnat ja keskeisin toteutus mitkä luonnollisesti ajan kuluessa muuttuvat vakaimmiksi.

Dependency Inversion -suunnitteluperiaate (DIP) kieltää ylemmän tason komponentteja olemasta riippuvaisia alemman tason komponenteista, suosittelee erottamaan komponenttien toteutukset toisistaan rajapintojen avulla ja määrittelemään rajapinnat käyttävän komponentin näkökulmasta. DIP on osittain yhtenäinen ensimmäisen havainnon kanssa, koska ensimmäisessä havainnossa komponentit ovat riippuvaisia vain abstraktioista poikkeuksena Composition-komponentti, joka vastaa DIP suositusta erottelemaan komponentit toisistaan rajapintojen avulla. Havainto puoltaa Markin (Seeman 2013) mainintaa siitä, että portit ja adapterit -arkkitehtuuri perustuu dependency inversion -suunnitteluperiaatteen käyttöön.

Portit ja adapterit -arkkitehtuurin yhtenäisyydet SDP ja DIP -suunnitteluperiaatteisiin voi hyvinkin selittää miksi portit ja adapterit -arkkitehtuuri näyttää parantavan ulkoista muunneltavuutta.

7 JOHTOPÄÄTÖKSET

Tutkimuskohteena käytetyn projektin tuloksiksi saatiin asiakasvaatimukset täyttävä PC-työkalu taajuusmuuttajan toiminnallisen turvallisuuden konfigurointiin. Portit ja adapterit -arkkitehtuurin käytöllä PC-työkalun business logiikassa saavutettiin välttävä sisäinen muunneltavuus ja hyvä ulkoinen muunneltavuus. Tutkimuksen tuloksena saatiin näyttöä siitä, että portit ja adapterit -arkkitehtuuri soveltuu käytettäväksi ketterässä ohjelmistokehityksessä, koska se parantaa ylläpidettävyyttä ylläpidettävyyden testattavuus komponentin lisäksi myös muunneltavuus komponentin kautta.

Portit ja adapterit -arkkitehtuurin käytöstä aiheutuva hyvä ulkoinen muunneltavuus johtuu siitä, että arkkitehtuurissa sovelluksen keskeinen toiminnallisuus eristetään rajapintojen avulla muista komponenteista. Keskeisen toiminnallisuuden eristämisen toteutus täyttää osittain ketterien suunnitteluperiaatteiden suosituksia, joka aiheuttaa tuloksissa mainitut yhtenäisyydet arkkitehtuurin ja SDP ja DIP -suunnitteluperiaatteiden välillä. Tulosten perusteella voidaan päätellä, että portit ja adapterit -arkkitehtuurin käyttö noudattaa SDP-suunnitteluperiaatteita täysin ja DIP-suunnitteluperiaatetta osittain. Portit ja adapterit -arkkitehtuurin osittainen samankaltaisuus DIP-suunnitteluperiaatteeseen johtuu siitä, että DIP-suunniteluperiaate suosittelee määrittämään rajapinnat käyttävän komponentin näkökulmasta, kun taas portit ja adapterit -arkkitehtuuri ei ota kantaa rajapintojen määrittämisen näkökulmaan, vaan vastuu tästä päätöksestä jätetään arkkitehtuurin käyttäjälle. Portit ja adapterit -arkkitehtuurin käytöstä aiheutuva välttävä sisäinen muunneltavuus johtuu siitä, että portit ja adapterit -arkkitehtuuri ei ota kantaa ensisijaisten ja toissijaisten sovitin-komponenttien sisäiseen rakenteeseen, vaan jättää vastuun arkkitehtuurin käyttäjälle.

Tutkimuksen tarkkuuteen voi vaikuttaa analyysiin valittujen käyttäjätarinoiden määrä, valitut käyttäjätarinat ja objektiivisuuden taso. Analyysiin valittiin kuusi käyttäjätarinaa projektin alkupäästä. Käyttäjätarinoiden määrä olisi voinut ehkä olla hieman suurempi tai sitten käyttäjätarinoita olisi voinut valita laajemmalta alueelta, esimerkiksi projektin

alkuosasta, keskiosasta ja loppuosasta. Käyttäjätarinoiden valinta laajemmalta alueelta olisi ehkä aiheuttanut arkkitehtuurin kokonaiskuvan hämärtymisen, mutta toisaalta olisi ollut hyvä, jos valituissa käyttäjätarinoissa olisi ollut mukana muutoksia aikaisemmin toteutettuihin komponentteihin. Muutoksien toteuttaminen olemassa oleviin komponentteihin olisi näyttänyt todelliset muutosten tarpeet muissa komponenteissa. Tutkimusta tehdessä pyrittiin ottamaan mahdollisimman objektiivinen näkökulma, mutta koska tutkimuksen tekijä on ollut mukana tutkimusmateriaalina käytetyssä projektissa, täytyy tämä ottaa huomioon tulosten tulkinnassa.

Jatkotutkimuksena ehdotetaan portit ja adapterit -arkkitehtuurin vertailua Robertin (Martin 2003: 95–135, 253–268) mainitsemiin ketteriin suunnitteluperiaatteisiin esimerkiksi ohjelmistosuunnittelijan sisäistämisen näkökulmasta. Kaiken kaikkiaan Mark mainitsee kirjassa 11 erilaista suunnitteluperiaatetta mukaan lukien SDP ja DIP olisi mielenkiintoista tietää kuinka suuri ero on portit ja adapterit -arkkitehtuurin sisäistämisessä verrattaessa 11:sta ketterän kehityksen suunnitteluperiaatteeseen sekä kuinka paljon enemmän arvoa 11:sta suunnitteluperiaatteen noudattaminen tuo ketterään ohjelmistokehitykseen verrattuna portit ja adapterit -arkkitehtuuriin.

LÄHDELUETTELO

ABB (2016a). *Drives deliver safety for food and beverage machinery* [Verkkodokumentti]. ABB. [Viitattu 5.12.2016]. Saatavissa: http://new.abb.com/drives/segments/food-and-beverage/drives-deliver-safety-for-food-and-beverage

ABB (2016b). *Cost and time savings with drive-based functional safety* [Verkkodokumentti]. ABB. [Viitattu 22.12.2016]. Saatavissa: http://new.abb.com/drives/functional-safety/cost-and-time-savings-with-drive-based-functional-safety

Cockburn, A. (2005). *Hexagonal architecture.* [Verkkodokumentti]. Cockburn, A. [Viitattu 28.10.2018]. Saatavissa: https://web.archive.org/web/20180822100852/http://alistair.cockburn.us/Hexagonal+architecture

Cockburn, A. (2007). *Agile Software Development. The Cooperative Game.* Second Edition. Boston: Pearson Education. Inc. ISBN 0-321-48275-1

Garrido de Paz, JM. (2018). *Ports and Adapters Pattern (Hexagonal Architecture)*. [Verkkodokumentti]. Garrido de Paz, JM. [Viitattu 21.12.2018]. Saatavissa: https://softwarecampament.wordpress.com/portsadapters/

Hazarika, P., Rahul, R. CP. & Seshubabu, T. (2014). Recommendations for Webview Based Mobile Applications on Android. In: *2014 IEEE International Conference on Advanced Communication Control and Computing Technologies (ICACCCT)*, 1589-1592. Ramanathapuram: IEEE. Saatavissa: http://ieeexplore.ieee.org.proxy.uwasa.fi/document/7019375

IEC (2016). *Functional Safety* [Verkkodokumentti]. IEC. [Viitattu 22.12.2016]. Saatavissa: http://www.iec.ch/functionalsafety/explained/

ISO/IEC 25023 (2016). *(SQuaRE) - Measurement of system and software product quality*. Systems and software engineering - Systems and software Quality Requirements and Evaluation.

IPv6 (2017). *UDP User Datagram Protocol.* [verkkodokumentti]. Ipv6.com Inc. [Viitattu 6.7.2017]. Saatavissa: http://ipv6.com/articles/general/User-Datagram-Protocol.htm

Koskimies, K. & Mikkonen, T. (2005). *Ohjelmistoarkkitehtuurit*. Tampere: Talentum.

Lehtonen, T., Tuomivaara, S., Rantala, V., Känsälä, M., Mäkilä, M., Jokela, T., Könnölä, K., Kaisti, M., Suomi, S., Isomäki, M & Ylitoiva, M. (2014). *Sulautettujen järjestelmien. Ketterä käsikirja.* Turku: Turun yliopisto. ISBN 978-951-29-5838-2

Martin, C. R. (2003). *Agile Software Development. Principles, Patterns and Practices.* New York: Pearson Education. Inc. ISBN 0-13-597444-5

Meany, Tom (2016). Functional safety for integrated circuits used in variable speed drives. In: *PCIM Europe* 2016, 1361-1368. Berlin: VDE. Saatavissa: http://ieeexplore.ieee.org.proxy.uwasa.fi/document/7499513

Microsoft (2011). *General Service Testing Best Practices* [Verkkodokumentti]. Microsoft. [Viitattu 9.7.2018]. Saatavissa: https://msdn.microsoft.com/en-us/library/hh323702(v=vs.100).aspx

Piper, James (2009). *The Benefits of VFDs In HVAC Systems* [Verkkodokmentti]. FacilitiesNet. [Viitattu 21.12.2016]. Saatavissa: http://www.facilitiesnet.com/hvac/article/The-Benefits-of-VFDs-In-HVAC-Systems-Facilities-Management-HVAC-Feature--11278#

Siemens (2016a). *Drive Technology Advantages* [Verkkodokumentti]. Siemens. [Viitattu 22.12.2016]. Saatavissa: http://www.industry.siemens.com/topics/global/en/safety-integrated/machine-safety/product-portfolio/drive-technology/Pages/safety-drives.aspx

Siemens (2016b). *Safe stopping functions* [Verkkodokumentti]. Siemens. [Viitattu 19.03.2017]. Saatavissa: http://www.industry.siemens.com/topics/global/en/safety-integrated/machine-safety/product-portfolio/drive-technology/Pages/safety-drives.aspx?tabcardname=functions

Seeman, M. (2012). *Dependency Injection in .NET*. New York: Manning. ISBN 9781935182504

Seeman, M. (2013). *Layers, Onions, Ports, Adapters: it's all the same*. [Verkkodokumentti]. Seeman, M. [Viitattu 10.07.2018]. Saatavissa: http://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/

Sjoberg, D. I. K., Dyba, T. & Jorgensen, M. (2007). The Future of Empirical Methods in Software Engineering Research. In: *Future of Software Engineering, 2007*, 358-378. Minneapolis: IEEE. Saatavissa: http://ieeexplore.ieee.org.proxy.uwasa.fi/document/4221632/

Yau, S. S. & Chang, P. (1988). A metric of modifiability for software maintenance. In: *Proceedings. Conference on Software Maintenance, 1988,* 374-381. Scottsdale: IEEE. Saatavissa: https://ieeexplore-ieee-org.proxy.uwasa.fi/document/10190/

Young, C. (2017). *Hexagonal Architecture–The Great Reconciler?*. [Verkkodokumentti]. Young, C. [Viitattu 26.5.2017]. Saatavissa: http://geekswithblogs.net/cyoung/archive/2014/12/20/hexagonal-architecturendashthe-great-reconciler.aspx

Xiaodan. Y & Stacie. P. (2014). Understanding agile software development practices using sharedmental models theory. *Information and Software Technology* 56: 6, *911-921*. Saatavissa: http://dx.doi.org.proxy.uwasa.fi/10.1016/j.infsof.2014.02.010