Kanditutkielma

Tietojenkäsittelytieteen kandiohjelma

Toimialalähtöinen ohjelmistokehitys ja Elixir virhesietoisten ohjelmistojen tuotannossa

Rolf Wathén

8.12.2023

Matemaattis-luonnontieteellinen tiedekunta

Helsingin yliopisto

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähkopostiosoite: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Bachelor's Programme in Computer Science

Tekijä — Författare — Author

Rolf Wathén

Työn nimi — Arbetets titel — Title

Toimialalähtöinen ohjelmistokehitys ja Elixir virhesietoisten ohjelmistojen tuotannossa

Ohjaajat — Handledare — Supervisors

Lea Kutvonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Bachelor's thesis	December 8, 2023	29 pages

Tiivistelmä — Referat — Abstract

level.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Designing software → Software design engineering
Applied computing → Enterprise computing → Business process management → Business process modeling
Software and its engineering → Software notations and tools → Context specific languages → Domain specific languages

Avainsanat — Nyckelord — Keywords

domain-driven design, DDD, model based methods, fault tolerance, Elixir, monitoring

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Sisällys

1 Johdanto

Ihmisen tekemät asiat ovat harvoin täysin virheettömiä ja täydellisiä. Ainakaan ensimmäisenä versionaan. Tämä luonnollisesti, ehkä jopa erityisesti, pätee tietokoneohjelmiin. Näihin kirjoitettaessa jää usein vikoja, joiden etsintä ja poisto, debuggaus, on oleellinen osa ohjelmointia. Vioista johtuvat ohjelmistovirheet voivat aiheuttaa myös suuria taloudellisia menetyksiä. Vuonna 2009 Yhdysvalloissa vuosittaisten korjauspäivitysten ja tarvittavien uudelleenasennusten hinnaksi on arvioitu 60 miljardia dollaria vuosittain (Zhivich ja Cunningham, 2009). Nyky-yhteiskunnan infrastruktuuri on myös lähes täysin riippuvainen ohjelmistoista. Koillis-Yhdysvallat vuonna 2003 pimentänyt ohjelmistovirheestä johtunut sähkökatko aiheutti muiden kuviteltavissa olevien hankaluuksien lisäksi 7-10 miljardin dollarin kustannukset (Zhivich ja Cunningham, 2009). Taloudellisten menetysten lisäksi ohjelmistoviat voivat aiheuttaa esimerkiksi turvallisuusuhkia (Weber et al., 2005).

Tietokoneohjelman suorittamisen keskeyttävät virheet ja häiriöt voivat tapahtua suoritetilanteissa, joita ohjelmoija ei ole ennakoinut. Ne voivat tapahtua ohjelman ulkopuolisista syistä. Esimerkkeinä ovat muistipiirin pettäminen, massamuistin lukuvirhe, toisen ohjelman aiheuttamaa häiriö tai verkkoyhteyden pettäminen.

Ohjelma voidaan kirjoittaa sellaiseksi, että sen suoritus ei keskeydy virheeseen tai häiriöön. Ohjelmistoarkkitehtuurin suunnittelu, valitut työkalut ja implementointimetodit vaikuttavat tähän tavoitteeseen. Toimialaperusteinen suunnittelu (Evans, 2003) on yksi mahdollisuus ohjelmistoarkkitehtuurin suunnittelumetodiksi. Siinä keskiössä ovat toimialasta tehdyt mallit, joiden pohjalta varsinainen ohjelmisto toteutetaan. Joissain tapauksissa suoritettava ohjelmakoodi tuotetaan suoraan malleista, toisissa mallit implementoidaan toimiala-spesifisillä ohjelmointikielillä tai yleisillä ohjelmointikielillä. Elixir (Elixir, 2023) on verrattain uusi, suosiotaan kasvattava yleinen ohjelmointikieli (StackOverflow, 2023). Se on laajennettavissa toimialakohtaiseksi ohjelmointikieleksi ja sisältää ominaisuuksia, joilla ohjelmakoodista voidaan tehdä virhe- ja häiriösietoinen.

Tämän tutkielman tarkoituksena on tarkastella toimialamallien implementointia Elixir-ohjelmointikielellä niin, että tuloksena olevat ohjelmistot olisivat mahdollisimman vikavapaita ja robusteja erilaisia ajonaikaisia häiriöitä vastaan.

Tutkielma jakautuu viiteen lukuun. Luku 2 esittelee toimialaperusteisen suunnittelun ja pohtii sen käyttöä vikasietoisuuden näkökulmasta. Luku 3 käsittelee Elixir-ohjelmointikieltä

ja sillä kirjoitettujen ohjelmien virhe- ja häiriösietoisuutta. Luku 4 pohtii Elixirin-käyttöä toimialaperusteisen suunnittelun implementointityökaluna. Viimeinen luku esittää tutkielman johtopäätökset.

2 Toimialaperusteinen ohjelmistokehitys

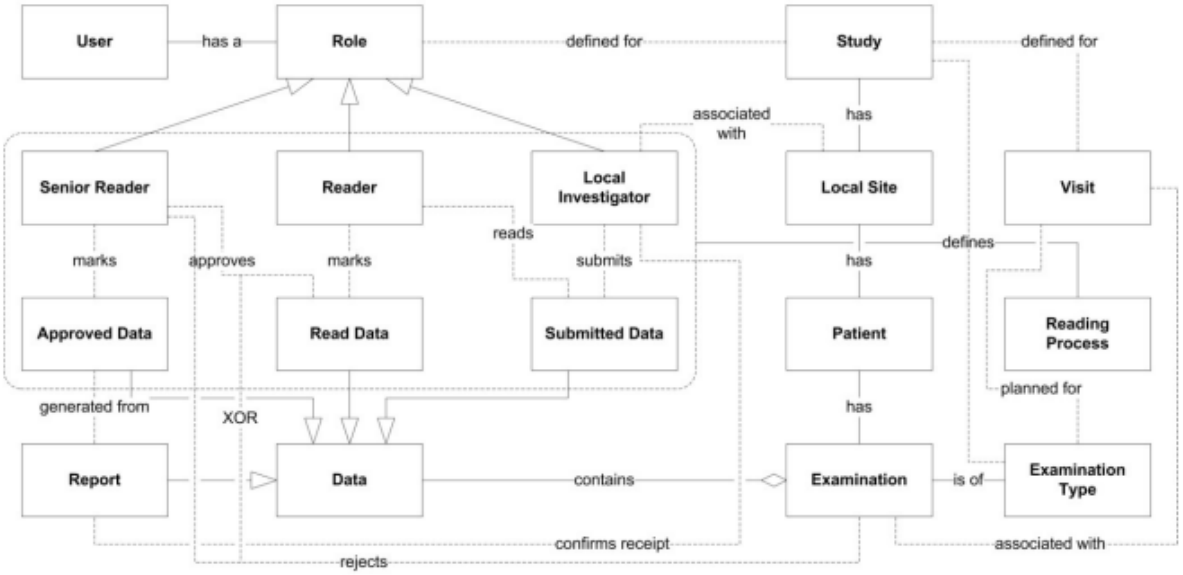
Toimialaperusteinen suunnittelu kuuluu malliperusteisiin ohjelmistokehityskäsitteisiin. Siihen läheisesti liittyviä muita menetelmiä ovat malliperusteinen arkkitehtuuri (Model-Driven Architecture, MDA) (OMG, 2023), malliperusteinen tuotanto (Model-Driven Engineering, MDE) (Boronat, 2019) ja malliperusteinen kehitys (Model-Driven Development, MDD (Selic, 2003)). Toimialaperusteisessa suunnittelussa (Domain Driven Design, DDD) ohjelmistosuunnittelun keskiössä ovat toimialasta (domain) luodut mallit. Evansin (2003) kirjaan Domain-Driven Design: Tackling Complexity in the Heart of Software viitataan useassa lähteessä toimialaperusteisen suunnittelun lähdeteoksena (Läufer, 2008; Lotz et al., 2010; Wlaschin, 2018; Rademacher et al., 2018; Vural ja Koyuncy, 2021; Oukes et al., 2021; Wang et al., 2022; Zhong et al., 2022; Zhong et al., 2023; Özkan et al., 2023). Tämän jälkeen aihetta ovat kokonaisuutena käsitelleet ainakin Khonovov (2022) ja Millett ja Tune (2015).

2.1 Menetelmän kuvaus

Mallinnus on oleellinen osa isoja ohjelmistoprojekteja (UML, 2005; Evans, 2003). Ohjelmistomallien voi osittain ajatella vastaavan talon piirustuksia rakentamisessa. Mallien avulla voidaan kuvata ohjelmistojen toiminnallisuutta sekä tarkastella kattavuutta ja loppukäyttäjän tarpeiden toteutumista. Tämän lisäksi mallinnusvaiheessa kannattaa huomioida skaalautuvuus, häiriöalttius, turvallisuus ja laajennettavuus. Jos nämä on toteutettu ohjelmakoodiin ilman kunnollista suunnittelua ja mallinnusta, muutosten teko on vaikeaa ja kallista.

Toimialamalli on selkeästi organisoitu abstraktio toimialasta, johon on tarkasti valittu sen tärkeimmät elementit (Evans, 2003). Toimialamallissa käytetään usein diagrammeja ja kaavioita kuten kuvassa 2.1. Yleisesti käytettyjä diagrammeja ovat UML diagrammit (Booch et al., 1998; Rademacher et al., 2018; UML, 2005), kuten luokkadiagrammit. Malli ei välttämättä ole sama kuin diagrammi (Wlaschin, 2018). Se voi olla myös esimerkiksi luonnollisella kielellä selkeäsi kirjoitettu kuvaus, ohjelmistokoodia tai muu tarkoituksen-

mukainen esitystapa (Wlaschin, 2018; Evans, 2003). Mallien liika yksityiskohtaisuus hidastaa niiden tekoa ja heikentää selkeyttä (Evans, 2003). Toisaalta yksityiskohtaisuuden puute voi hidastaa mallien implementointia (Rademacher et al., 2018).



Kuva 2.1: Toimialamalli kliinisten kokeiden tulosten analysointia harjoittavan laitoksen toimintaprosessista (Lotz et al., 2010).

Muihin mallipohjaisiin menetelmiin verrattuna toimialaperusteisessa suunnittelussa keskeistä ovat toimialasta, esimerkiksi pankkitoiminnasta, muodostetut mallit. Niitä muodostavat työryhmät, joihin kuuluvat esimerkiksi suunnittelijat, toimialaosaajat ja varsinaiset ohjelmoijat. Luotujen mallien rakennetta noudatetaan varsinaisessa ohjelmakoodissa. Malleja voidaan muodostaa eri näkökulmista, esimerkiksi painottaen tietoturvaa. Mallien tekijöiden tulisi osallistua myös ohjelmointiin (Evans, 2003). Jos organisaatiorajat eri ihmisryhmien välillä ovat liian jyrkät, tietoa katoaa rajapinnoissa.

On tärkeää, että työryhmään kuuluvat kommunikoivat keskenään samoilla termeillä. Evans käyttää termiä kaikkialla läsnäoleva kieli (ubiquitous language) kuvaamaan tätä kieltä. Tämä kieli syntyy iteratiivisesti malleja luodessa ja kehittäessä. On tärkeää varmistaa, että kaikki osallistujat ymmärtävät kaikki termit ja käsitteet samalla tavalla. Kielessä käytettyä sanastoa käytetään ohjelmistokoodissa esimerkiksi luokkien, metodien ja muuttujien nimeämisessä (Evans, 2003; Gray ja Tate, 2019). Tästä käytetään nimeä tarkoitukset paljastava rajapinta (intention-revealing interface). Zhong et al. (2023) ovat näyttäneet,

että parempi kommunikaatio työryhmässä johtaa parempiin malleihin. Nämä taas oikein implementoituina johtavat vähemmän virheherkkään ja helpommin ylläpidettävään ohjelmakoodiin. Huono suunnittelu ja siitä johtuvat ohjelmistovirheet voivat johtaa ohjelman epätoivottuun toimintaan tai esimerkiksi vakaviin tietovuotoihin (Weber et al., 2005; Kärkkäinen ja Hujanen, 2023; Cao, 2020)

Yhteisen kielen saavuttamisen jälkeenkin ohjelmoijien säännöllinen keskustelu toimialaosaajien kanssa on korostetun oleellista. Molemmin puolinen oppiminen on tärkeää. Kun ohjelmoija ymmärtää mitä on tekemässä ja miksi, hänen on helmpompi keskittyä oleellisiin asioihin. Vaikeasti ylläpidettävät ohjelmistot syntyvät yleensä, kun toimialaa tai käyttöympäristöä ei ole kunnolla ymmärretty ja mallinnettu (Evans, 2003; Zhong et al., 2022).

Mallit kehittyvät projektin edetessä. Mallinnus ja niiden implemointi kulkevat rinnakkain. Mallia tehdessä on mietittävä sen implementointia (Evans, 2003). Implementointi taas saattaa tuoda uusia ideoita mallinnukseen. Jos tehty ohjelma ei pääpiirtettäin noudata tehtyä mallia, on mallin arvo kyseenalainen. Kyseenalaistaa voi myös toimiiko koodi silloin oikein. Kun mallin ja koodin yhteys ei ole riittävän selkeä, kumpikaan ei tue toisen ymmärtämistä. Jos osio mallista on mahdotonta implementoida ohjelmistoteknisesti sitä noudattaen, täytyy mallia päivittää (Evans, 2003).

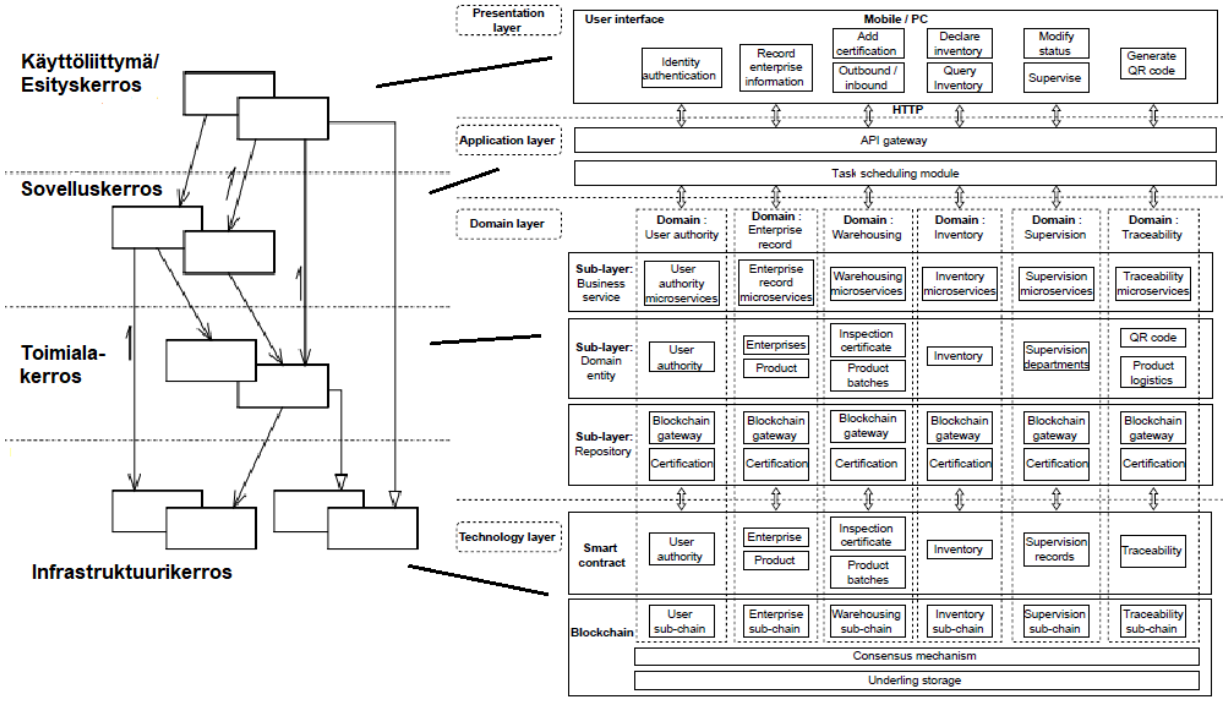
On tärkeää että mallin sisältö on tarkkaan rajattu ja mallien väliset rajat ovat selkeät, kirjallisuus käyttää termiä rajattu konteksti (bounded context) (Evans, 2003; Özkan et al., 2023; Vural ja Koyuncy, 2021; Siegel, 2014). Yksi toimialamalli voi siis muodostua useasta rajatusta kontekstista. Näiden väliin on hyödyllistä joissain tapauksissa tehdä suojakerros (anti-corruption layer), joka varmistaa, että yhdestä rajatusta kontekstista tuleva data on oikeanlaista ja oikeassa muodossa mennessään käsittelyyn toiseen rajattuun kontekstiin. Kontekstien suhdetta toisiinsa voidaan esittää kontekstikartoilla (context map). On oleellista, että jokaista kontekstia ja näiden alaosioita kohti on yhteisymmärrys siitä, minkä mallin perusteella suoritetaan implementointi (Evans, 2003).

Toimialamalleissa säilyy kuvaus ohjelmakoodin toiminnasta (Siegel, 2014). Ohjelmistoprojektien pitkittyessä henkilökunta usein vaihtuu, kuten vaihtuvat myös ohjelmistojen käyttäjät ja yllpitäjät. Tällöin mallit luonnollisesti tekevät uusille henkilöille ohjelmistojen päivityksestä ja ylläpidosta helpompaa (Evans, 2003).

Vikojen ja niiden aiheuttamien ohjelmistovirheiden mahdollisuus pienenee, kun ohjelman rakenne noudattaa kaikkien hyväksymästä mallia (Evans, 2003). Virheellinen malli voi johtaa virheherkempään toteutukseen (Zhong et al., 2022; Zhong et al., 2023). Tämän

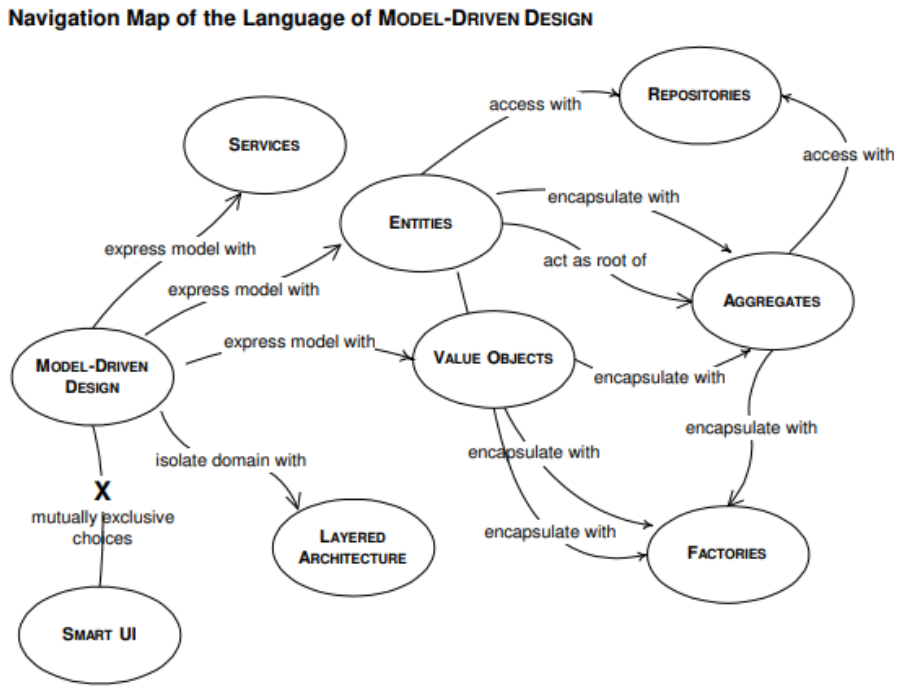
vuoksi mallien oikeellisuus on mahdollisuuksien mukaan varmistettava esimerkiksi simuloimalla. Mallien luonti olemassa olevasta koodista on mahdollista (Boronat, 2019). Tämä on hyödyllistä, jos koodi on kirjoitettu ilman toimialaperusteista suunnittelua tai muuta mallinnusmetodiikkaa.

Toimialaperusteisessa suunnittelussa ohjelmat suunnitellaan käyttämään kerroksellista arkkitehtuuria (Snoeck et al., 2000; Evans, 2003). Kaikki varsinainen toimialaan liittyvä ohjelmakoodi eristetään omaan kerrokseensa. Kerroksellisen arkkitehtuurin ideaa on on kuvattu kuvassa 2.2. Kerroksellinen arkkitehtuuri helpottaa ohjelman ylläpitoa ja suunnittelua (Dooley, 2011). Tyypilliset kerrokset, joihin ohjelma on jaettu ovat käyttöliittymäkerros (user interface layer, presentation layer), sovelluskerros (application layer), toimialakerros (domain layer) ja infrastruktuurikerros (infrastructure layer) (Evans, 2003). Käyttöliittymäkerros kommunikoi joko käyttäjän tai toisen tietojärjestelmän kanssa, ottaa vastaan näiden pyynnöt ja palauttaa ja esittää tulokset. Yleensä kevyenä pidettävä sovelluskerros välittää komentoja ja dataa käyttöliittymäkerroksen ja toimialakerroksen välillä sekä ohjaa toimialakerroksen toimintaa. Toimialakerrokseen on keskitetty kaikki toimialalogiikka. Infrastruktuurikerroksessa ovat tekniset toiminnot kuten tietokantojen käsittely, viestien lähetys ja käyttöliittymän piirto. Toimialaperusteisen suunnittelun yhteyteen on kehitetty kehysympäristöjä (framework) kuten Naked Objects, joilla ydinkerrokseen tehdyt muutokset välittyvät automaattisesti muihin kerroksiin (Läufer, 2008).



Kuva 2.2: Malli kerrokselliselle arkkitehtuurille (Evans, 2003) ja sitä soveltaen tehty referenssiarkkitehtuuri lohkoketjupohjaisille seurantajärjestelmille(Wang et al., 2022).

Malli tuodaan ohjelmakoodiin palveluiden (services), entiteettien (entities) ja arvo-olioiden (value objects) avulla. Palvelu (service) on toiminto, joka suoritetaan kutsusta. Entiteetti on yksikkö, jolla on yksilöllinen identiteetti. Esimerkkejä ovat pankkitili tai henkilö. Ensimmäisen identifioi pankkitilinumero ja toisen sosiaaliturvanumero. Arvo-olioissa merkittävää ovat niiden arvot, ei identiteetti. Esimerkiksi pankkitiliin voidaan liittää arvo-olio tilityyppi, jossa annetaan arvo korolle ja talletusajalle. Aggregaatti (aggregate) on joukko toisiinsa liittyviä olioita. Jokaiseen aggregaattiin liittyy yksi entiteetti, joka toimii sen juurena (root) ja on ainoa muun ohjelman näkemä olio. Tehdas (factory) luo halutut oliot ja aggregaatit. Repositoriot tallentavat ja toimittavat pyydettäessä tarvitut oliot. Mallin tuonti ohjelmakoodiin on esitetty kuvassa 2.3.



Kuva 2.3: Evansin 2003 esittämä käsitekartta toimialamallin tuomisesta ohjelmakoodiin olio-ohjelmoinnissa. Smart UI on vaihtoehtoinen tapa DDD:lle.

Mallinnusta ja implementointia varten voidaan luoda oma toimialakohtainen ohjelmointikielensä (domain-specific language, DSL). Näissä kielissä käytettään toimialan sanastoa ja rakenteita (Evans, 2003). Toimialakohtaisia kieliä käyttäen toimiala-asiantuntijat voivat helpommin osallistua ohjelmakoodin tuottamiseen (Hoffmann et al., 2022). Implementoinnissa toimialakohtainen ohjelmointikieli tavallisesti käännetään jollekin yleiselle ohjel-

mointikielelle. Toimialakohtainen ohjelmointikieli voidaan myös rakentaa jonkun yleisen ohjelmointikielen päälle. Tämä vaatii kyseiseltä kieleltä muokattavuutta (Evans, 2003).

Riippuen millä tekniikalla mallit ovat toteutettu, on mahdollista tuottaa suoritettava ohjelmakoodi suoraan mallista (Siegel, 2014). Näin on mahdollista välttää ohjelmakoodiin tulevia vikoja, koska ohjelmoijan huolimattomuus- ja ajatusvirheistä johtuvat viat jäävät pois. Edellytyksenä on, että ohjelmakoodin mallista tuottava ohjelma toimii oikein. Simulointi mallien avulla mahdollistaa vikojen havaitsemisen lisäksi haitallisten sivuvaikutusten ja prosessin pullonkaulojen olemassaolon (Siegel, 2014). Esimerkiksi Simulink-ohjelmalla voidaan mallintaa ja simuloida teollisuusprosessi uudelleen käytettävissä olevista yksikköprosesseista*. Tunnetuilla syöte-vaste-pareilla voidaan tarkistaa mallin oikea toiminta. Toisaalta voidaan mitata jonkin osion laskentaan kulunut aika.

2.2 Implementointeja käytännössä

Toimialaperusteista suunnittelua on mahdollista käyttää monenlaisten ohjelmistojen suunnitteluun. Implementointi käytännössä antaa parhaan käsityksen soveltuvuudesta, toteutustavoista ja menetelmän tärkeimmiksi koetuista osioista. Esimerkkejä käytännön implementoinneista ovat kliinisten kokeiden tulostenanalysointiohjelmisto (Lotz et al., 2010), mikropalvelut (Rademacher et al., 2018; Vural ja Koyuncy, 2021) ja lainhuutojen rekisteröintijärjestelmäohjelmisto (Oukes et al., 2021).

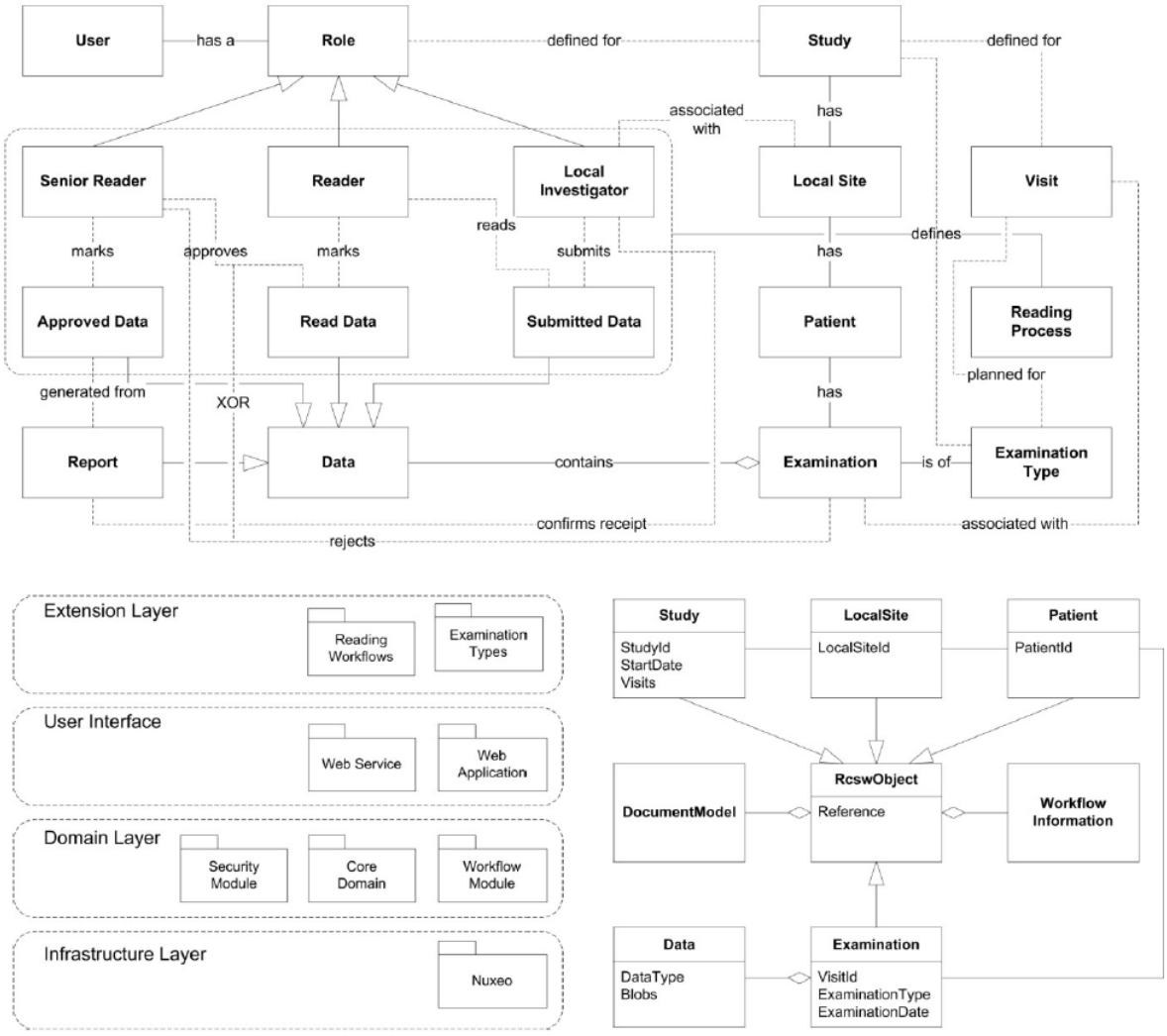
Kliinisten kokeiden tulostenanalysointiohjelmisto

Lots et al. (2010) ovat tehneet kliinisten kokeiden tuloksia analysoivalle analysointikeskukselle koemateriaalia ja sen analyysiä hallinnoivan sovelluksen käyttäen toimialaperusteista suunnittelua. He päätyivät toimialaperusteisen suunnittelun käyttöön toimialan monimutkaisuuden takia, taustalla eivät olleet ohjelmistotekniset haasteet. Toimialaperusteisen suunnittelun käyttö johti toimialan parempaan ymmärtämiseen ja mahdollisti kaikkien oleellisten tekijoiden implementoinin ohjelmistoon. Hyvin mallinnettu koedatan kulku ja käyttöoikeuksien määrittely pienentää todennäköisyyttä koedatan vaarantumisesta ohjelmistoviasta aiheutuvan virheen vuoksi.

Luotu toimialamalli, sen implementaatio ja ohjelmiston kerroksellinen rakenne on esitetty kuvassa 2.4. Käytetyssä kerroksellisessa arkkitehtuurissa ohjelmiston alin kerros to-

*https://se.mathworks.com/help/simulink/

teutettiin Nuxeo sisällönhallinta-alustaa käyttäen*, joka toimii repositoriona. Seuraavana kerroksena on toimialakerros, jossa on toteutettuna toimialamallin logiikka. Toimialakerroksen päällä on käyttöliittymä toteutettuna verkkosovelluksella ja palvelimella REST rajapintoja käyttäen. Ylimpänä on laajennuskerros esimerkiksi uusien tutkimustyyppien (Examination Type) implementointia varten.

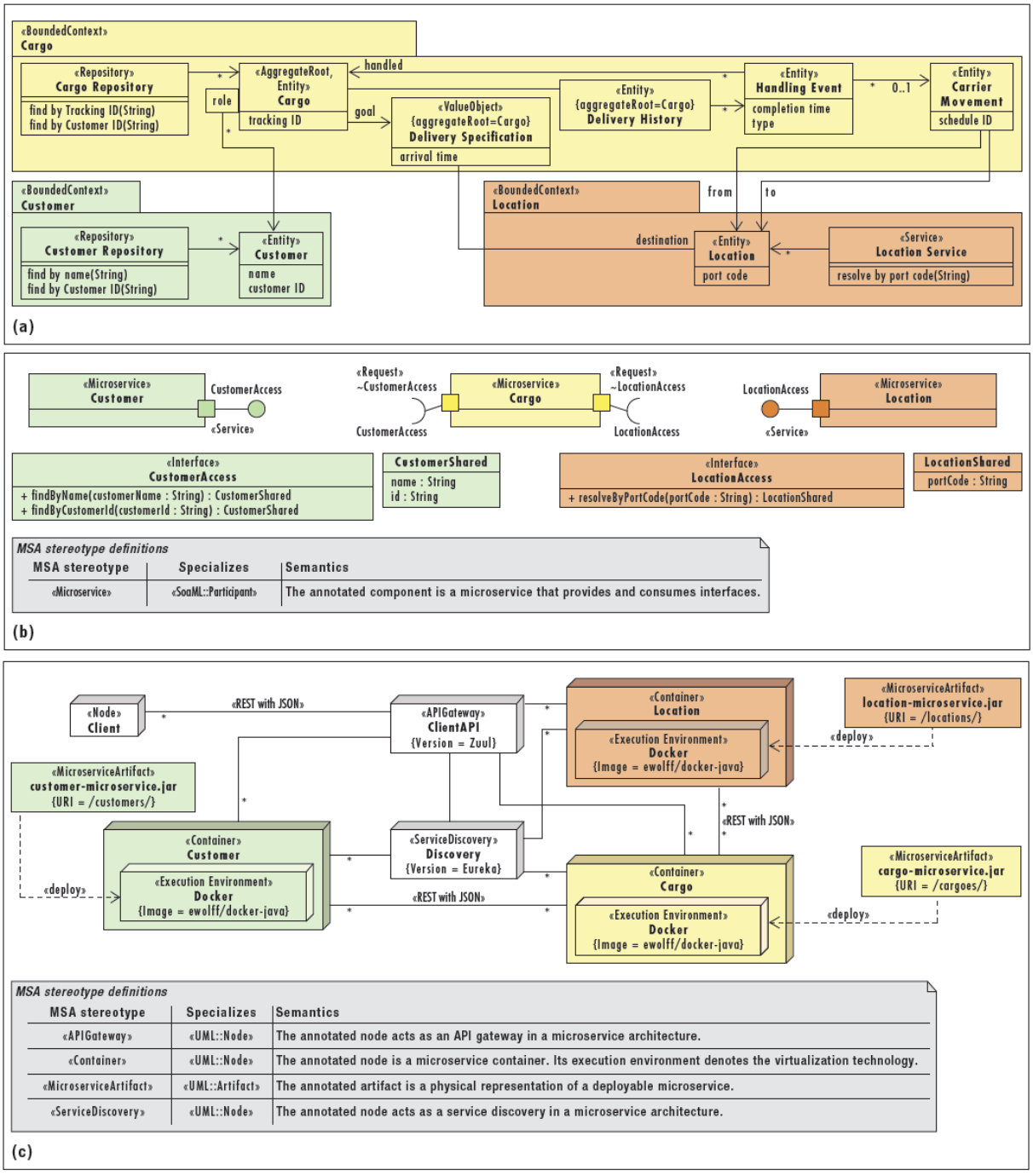


Kuva 2.4: Toimialamalli kliinisten kokeiden analysointia harjoittavan keskuksen toimintaprosessista (yllä), jossa laatikossa käytetty yhteisen kielen (ubiquitos language) termejä. Vasemmalla alhaalla toteutuksen kerroksellinen rakenne ja oikealla alhaalla toimialamallin toteutus (Lotz et al., 2010).

*https://doc.nuxeo.com/nxdoc/quick-overview/

Mikropalvelut

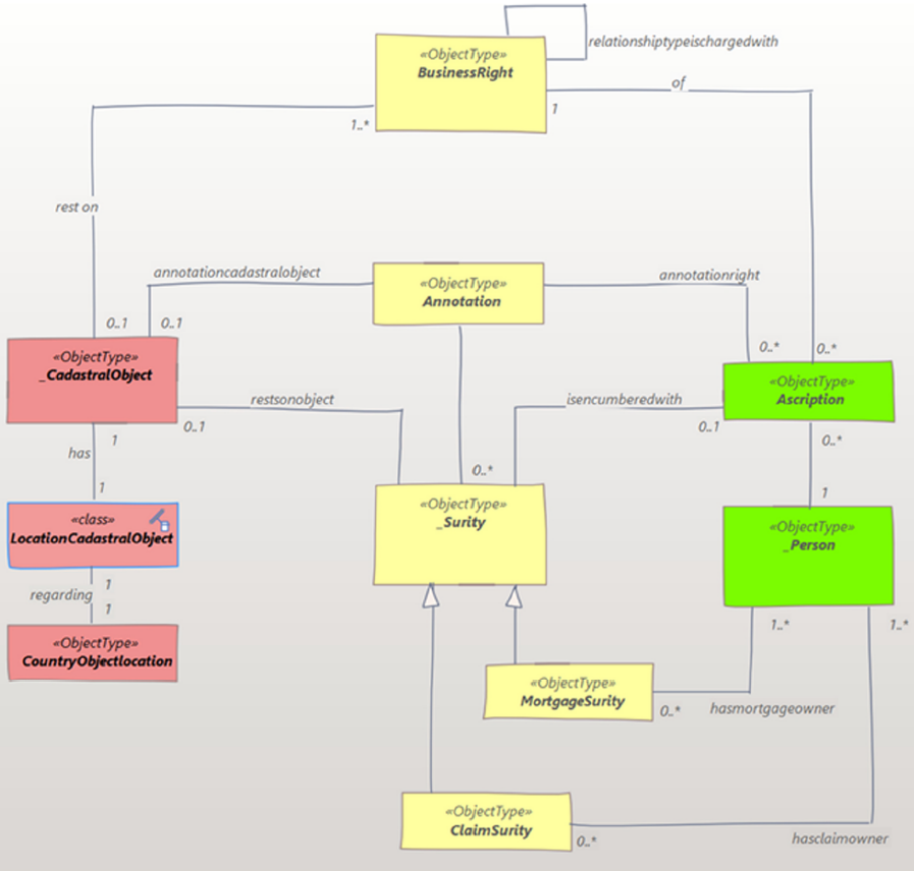
Mikropalveluarkkitehtuuriin perustuvissa ohjelmistoissa omissa proseisseissaan ajetut keskenään kommunikoivat mikropalvelut muodostavat sovelluksen (Lewis ja Fowler, 2014). Kuvassa 2.6 on esitetty toimialaperusteinen malli logistiikalle sekä siitä johdetut mikropalvelumallit rajapintoineen. Vaikka toimialaperusteinen suunnittelu soveltuu mikropalveluarkkitehtuuria käyttävien ohjelmistojen suunnitteluun, siinä on haasteita (Rademacher et al., 2018). Toimialaperusteisessa suunnittelussa mallit eivät yleensä ole yksityiskohtaisia. Vaikka toimialapohjaiset mallit olisivat määriteltyjä UML-diagrammeina, mallit ovat abstrakteja eivätkä sisällä esimerkiksi rajapintojen määrittelyjä, attribuuttien tyyppejä ja metodien paluuarvojen tyyppejä (Rademacher et al., 2018). Toimialamallien luomisen kannalta tämä on tarkoituksenmukaista. Haasteita toimialamallien toteutukseen mikropalveluna tuo myös kahteen eri rajoitettuun konktekstin kuuluvan toimialakäsitteen suhdetta epäselvä määritelty. Tästä esimerkkinä pelkkä assosiaatioviiva kuvassa 2.6a Delivery Specificationin ja Locationin välillä. Toisaalta toimialaperusteinen suunnittelu soveltuu hyvin mikropalveluiden moduulijaon suunnitteluun (Vural ja Koyuncy, 2021).



Kuva 2.5: Esimerkki mikropalveluiden määrittämisestä toimialapohjaisesta mallista(Rademacher et al., 2018). a) Lastinkuljetustoimialamalli (Cargo) UML-diagrammina käyttäen toimialaperusteisen suunnittelun käsitteitä. b) Toimialamallista johdettu välimalli mikropalvelujen rajapintojen määrittämiseksi. Mallinnetu käyttäen palveluorientoituneen arkkitehtuurin mallinnuskieltä (Service Oriented Architecture Modeling Language, SoaML). c) Toimialamallista johdettu malli mikropalvelujen toteutukselle.

Lainhuutojen rekisteröintijärjestelmäohjelmisto

Hollannin maanmittauslaitos Kadaster* on käyttänyt toimialaperusteista suunnittelua toteuttaessaan omistustietojärjestelmäänsä korvaamaan vanha Cobolilla kirjoitettu järjestelmä (Oukes et al., 2021). Toimialan mallinnus tehtiin UML-malleilla, jotka muunnettiin Java-luokiksi. Työssä hyödynnettiin ISO 19152 esittämää standardimallia maanhallinnan toimialalle. Toimialaperusteisen suunnittelun todettiin soveltuvan kyseisen toimialan mallintamiseen. Tärkeä tekijä oli toimialaosaajien ja ohjelmoijien yhteiden kieli.



Kuva 2.6: Oukes et al. (2021) luoma toimialamalli kiinteistöjen omistuksen rekisteröinnille Hollannissa esitettynä UML-luokkadiagrammina.

Toimialaperusteisen suunnittelun käytännön esimerkeissä nousee esille työryhmän yhteisen kielen tärkeys. Toimialan mallinnus ja jako rajattuihin konteksteihin auttavat hahomottamaan osioiden yhteyksiä ja toiminnallisuutta. Toimialamallien toteutuksessa UML-

*https://www.kadaster.nl/about-us

diagrammit ovat yleinen toteutustapa. Ykstyiskohtaisina mallit sitovat enemmän toteutustapaa, mutta toisaalta selkeyttävät sitä.

3 Elixir-ohjelmointikieli

Elixir on kohtuullisen uusi suosiotaan kasvattava ohjelmointikieli (Elixir, 2023; Ballou, 2015), jolla on aktiivinen virtuaaliyhteisö[†]. Sille on verkosta löydettävissä sekä asennustiedostot useille käyttöjärjestelmille[‡] että useita tutoriaaleja[§].

Elixir on funktionaalinen ohjelmointikieli. Ideaalisessa funktionaalisessa ohjelmoinnissa ohjelma koostuu funktioista, joiden palauttama arvo riippuu vain funktion argumenteista. Funktiolla ei myöskään ole muuhun laskentaan vaikuttavia sivuvaikutuksia (Chambers, 2014). Olio-ohjelmoinnissa taas kaikki laskenta tehdään olioiden ja niiden metodien avulla (Chambers, 2014). Esimerkiksi Mukundin kahden artikkelin kokonaisuus esittää tiiviin johdatuksen funktionaaliseen ohjelmointiin(Mukund, 2007a; Mukund, 2007b).

Sosiaalisen median alusta Discord[¶] on kertonut vuonna 2020 käyttävänsä Pythonin lisäksi Elixiriä ohjelmistonsa toteuttamiseen (Elixir, 2023). Järjestelmä koostuu monoliittisestä Python APIsta ja runsaasta kahdestakymmenestä Elixir-palvelusta. Viesti-infrastuktuuri koostuu yli neljästäsadasta Elixir-palvelimesta ja pystyy käsittelemään miljoonia samanaikaisia käyttäjiä. Discordin lisäksi Elixiriä käyttävät esimerkiksi Pepsico ja Pintarest (Elixir, 2023) ja siihen läheisesi liittyvää Erlangia Whatsapp (Whatsapp, 2012).

Sovelluksia varten Elixir-ohjelmat käännetään tavukoodiksi, jota ajetaan BEAM-virtuaalikoneessa (Erlang, 2023). BEAM on alunperin kirjoitettu Erlang-ohjelmointikieltä varten (Erlang, 2023). Elixir pohjautuu Erlangiin, on yhteensopiva tämän kanssa sekä käyttää samoja kirjastoja. Tavukoodiksi kääntämisen lisäksi Elixiriä voidaan käyttää interaktiivisessa iEx-ympäristössä (interactive elixir) (Elixir, 2023). Tämä mahdollistaa interaktiivisen ohjelmistokehityksen ja testauksen.

3.1 Elixir-prosessit

Elixirissä koodi ajetaan prosesseiksi kutsutuissa yksiköissä (Elixir, 2023; Ballou, 2015). Ne muistuttavat käyttöjärjestelmien säikeitä (thread), mutta ovat näihin verrattuna keveitä.

[†] https://elixirforum.com/
[‡] https://elixir-lang.org/install.html
[§] https://hexdocs.pm/elixir/1.16/introduction.html,https://elixirschool.com/en
[¶] https://discord.com/

Vastakäynnistetyn prosessin koko on 338 sanaa, josta keko vie 233 sanaa (Erlang, 2023). Yksi sovellus voi koostua lähes rajattomasta määrästä samanaikaisia prosesseja. Se tekee Elixir-ohjelmista helposti skaalautuvia. Prosessit kommunikoivat keskenään viesteillä. Kuvassa 3.1 on esitetty yksinkertainen prosessin käynnistäminen, sille viestin lähettäminen iEx-prosessista ja prosessin sulkeutuminen.

```
iex(26)> self()
#PID<0.109.0>
iex(27)> uusi_pid=spawn(fn->receive do {:terve, lahettaja} -> IO.puts("Terve #{inspect lahettaja}") end end)
#PID<0.116.0>
iex(28)> Process.alive?(uusi_pid)
true
iex(29)> send(uusi_pid,{:terve,self()})
Terve #PID<0.109.0>
{:terve, #PID<0.109.0>}
iex(30)> Process.alive?(uusi_pid)
false
iex(31)>
```

Kuva 3.1: Uuden Elixir-prosessin käynnistäminen, sille viestin lähettäminen ja prosessin sulkeutuminen iEx-ympäristössä.

3.2 Elixir-prosessien virhesietoisuus

Elixir-sovelluksissa virheeseen tai häiriöön peruuttamattomasti keskeytyneen prosessin ei tarvitse johtaa koko ohjelmiston keskeytymiseen. Oikein kirjoitettuna Elixir-ohjelma pystyy käynnistämään keskeytyneen prosessin helposti uudelleen. Tämä tehdään toimivaksi tiedetystä alkutilasta. Keskeytymisten havainnointi ja uudelleenkäynnistys tehdään Elixirissä monitorointiprosessien avulla, joita kutsutaan tavallisesti supervisor-prosesseiksi. Ne valvovat niille määrättyjä prosesseja ja virhetilanteen sattuessa käynnistävät ne uudelleen. Supervisor-prosessit voivat itse olla toisten supervisor-prosessien valvonnassa, muodostaen supervision-puita.

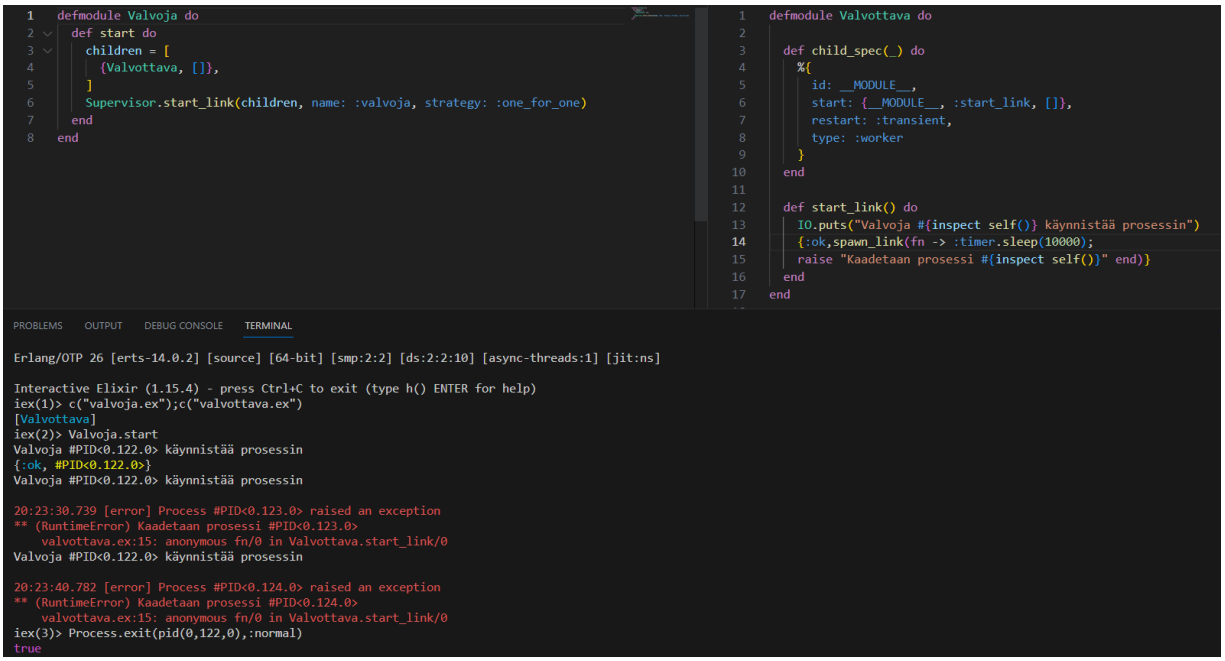
Virhetilanteessa Elixirin supervisor-prosessi voi valvomansa prosessin suhteen noudattaa kolmea eri strategiaa:

- :one_for_one -> Valvotun prosessin kaatuessa vain se käynnistetään uudelleen.
- :one_for_all -> Valvotun prosessin kaatuessa kaikki valvotut prosessit päätetään ja käynnistetään uudelleen.

- :rest_for_one -> Valvotun prosessin kaatuessa se ja sen jälkeen käynnistetyt prosessit päätetään ja käynnistetään uudelleen.

Keskeytyessään tai muuten päättyessään Elixir-prosessi lähettää viestin sitä monitoroivalle prosessille. Elixirin supervisor-rakenteet ovat abstraktioita matalamman tason rakenteille. Näiden rakenteiden avulla voi keskeytyksille tehdä myös räätälöityjä reagointeja, toisin sanoen myös muut kuin yllämainitut strategiat ovat mahdollisia. Prosessin voi ohjelmallisesti määrätä keskeytymään jonkun tietyn ehdon täyttyessä, esimerkiksi kun jostain tapahtumasta on kulunut liian pitkä aika.

Kuvassa 3.2 on esittety yksinkertainen supervisor-prosessi*. Käytetty uudelleenkäynnistysstrategia on :one_for_one. Terminaalista näkyy, kuinka prosessi kaatuu kymmenen sekunnin välein ja supervisor käynnistää uuden vastaavan prosessin.



Kuva 3.2: Yksinkertaistettu supervisor-prosessi Elixirissä. Funktio Valvoja.start käynnistää monitoroivan prosessin #PID<0.122.0>. joka käynnistää monitoroidun prosessin #PID<0.123.0>. Tämä pysäytetään kymmenen sekunnin kuluttua. Supervisor-prosessi #PID<0.122.0> käynnistää sen uudelleen, nyt uudella prosessitunnuksella #PID<0.124.0>. Supervisor-prosessi lopetetaan, jolloin monitoroitukin prosessi poistuu.

Prosessin uudelleen käynnistäminen auttaa tilanteissa, joissa kyseessä ei ole systemaatti-

*Muokattu https://gist.github.com/gkaemmer/12a536f7c859c576200e974235e2f923

nen jokaisella suorituskerralla tapahtuva ohjelmointivirhe. Tällainen tilanne on esimerkiksi verkkoyhteyden häiriintyminen. Uudelleen käynnistämisen voi piilottaa ohjelmointivikoja, joiden aiheuttamat virheet johtavat prosessin pysähtymisen vain tietyissä erikoistilanteissa. Tämä on huomioitava testauksessa, jossa prosessit on testattava sekä useilla erilaisilla syötteillä että mahdollisesti järjestämällä erikoistilanteita.

Elixirissä on myös Pythonin `try/except`-lohkojen kaltainen `try/rescue`-rakenne, mutta sitä käytetään harvemmin*. Esimerkiksi epäonnistunut tiedoston avaaminen voidaan käsitellä `File.read/1` antaman paluuarvon mukaan. Kuvassa 3.3 on esitetty esimerkki `try/rescue`-rakenteesta ja eräs tapa käsitellä epäonnistunutta tiedoston lukua*.

```
iex(4)> virheeseen_pysahtyva=spawn(fn -> raise "Virhe!" end)
#PID<0.113.0>

20:02:59.558 [error] Process #PID<0.113.0> raised an exception
** (RuntimeError) Virhe!
    iex:4: (file)
iex(5)> virheesta_selviava=spawn(fn -> try do raise "Virhe!" rescue RuntimeError -> IO.puts("Selvisin!") end end)
Selvisin!
#PID<0.114.0>
iex(6)> case File.read("Pahkasian_parhaat.pdf") do
...(6)> {:ok,teksti} -> IO.puts("Onnistui: #{teksti}")
...(6)> {:error, syy} -> IO.puts("Virhe: #{syy}")
...(6)> end
Virhe: enoent
:ok
iex(7)>
```

Kuva 3.3: Esimerkit Elixirin try/rescue-rakenteesta ja tavasta käsitellä epäonnistunutta tiedoston lukua.

*https://hexdocs.pm/elixir/1.16/try-catch-and-rescue.html

3.3 Laajennettavuus toimialakohtaiseksi kieleksi

Elixiriä on mahdollisuus laajentaa toimialakohtaiseksi kieleksi. Elixiriin on tehty toimialakohtaisena kielenä pidettäviä laajennuksia esimerkiksi tietokantojen käsittelyyn* ja koneoppimiseen †.

Elixiriä voi laajentaa makroilla ja *sigil*eillä. Hyvä ohjelmointitapa on käyttää funktioita makrojen sijaan, jos mahdollista(Elixir, 2023). Sigilejä Elixirissä on valmiina merkkijonojen, säännöllisten lauseiden, sanalistojen ja päivämäärärakenteiden käsittelyyn. Ne alkavat ∼-merkillä, jota seuraa yksi pieni kirjain tai yksi tai useampi iso kirjain. Näiden jälkeen tulee erotin. Viimeisen erottimen jälkeen tulevat valinnaiset modifikaattoorit. Esimerkkinä kuvassa 3.4 ∼r sigil säännöllisille lauseille. Omia sigilejä voi luoda funktioilla jotka implementoivat sigil_{merkki}-rakenteen. Esimerkkinä ∼KAANNA kuvassa 3.5 joka antaa käänteisluvun.

```
iex(15)> saannollinen = ~r/toimialaperusteinen|suunnittelu/
~r/toimialaperusteinen|suunnittelu/
iex(16)> "toimialaperusteinen" =~ saannollinen
true
iex(17)> "suunnittelu" =~ saannollinen
true
iex(18)> "suunittel" =~ saannollinen
false
iex(19)>
```

Kuva 3.4: Esimerkki sigilistä ∼r jonka avulla Elixirissä voidaan käsitellä säännöllisiä lauseita

*https://hexdocs.pm/ecto/getting-started.html
†https://github.com/elixir-nx/scholar

```
iex(1)> defmodule Oma do
...(1)> def sigil_KAANNA(numero,[]), do: 1/String.to_integer(numero)
...(1)> end
{:module, Oma,
 <<70, 79, 82, 49, 0, 0, 5, 184, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 192,
   0, 0, 0, 18, 10, 69, 108, 105, 120, 105, 114, 46, 79, 109, 97, 8, 95, 95,
   105, 110, 102, 111, 95, 95, 10, 97, 116, ...>>, {:sigil_KAANNA, 2}}
iex(2)> import Oma
Oma
iex(3)> ~KAANNA(8)
0.125
iex(4)> ~KAANNA(17)
0.058823529411764705
iex(5)>
```

Kuva 3.5: Sigil ~KAANNA joka kääntää annetun luvun

3.4 Datan validointi vahdeilla

Elixirissä olevilla vahdeilla (guard) voidaan varmistaa, että funktioon tuleva data on oikeanlaista ja oikea versio funktiosta otetaan käyttöön. Tätä on demonstroitu kuvassa 3.6 `when`-ilmaisulla. Vahdeista voidaan tehdä huomattavasti monimutkaisempiakin.

```
iex(10)> defmodule Laske do
...(10)> def summa(a,b) when is_integer(a) and is_integer(b) do
...(10)> a+b
...(10)> end
...(10)> def summa(a,b) do
...(10)> "Lasken vain kokonaislukuja."
...(10)> end
...(10)> end
warning: variable "a" is unused (if the variable is not meant to be used, prefix it with an underscore)
  iex:14: Laske.summa/2

warning: variable "b" is unused (if the variable is not meant to be used, prefix it with an underscore)
  iex:14: Laske.summa/2

{:module, Laske,
 <<70, 79, 82, 49, 0, 0, 5, 200, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 171,
   0, 0, 0, 18, 12, 69, 108, 105, 120, 105, 114, 46, 76, 97, 115, 107, 101, 8,
   95, 95, 105, 110, 102, 111, 95, 95, 10, ...>>, {:summa, 2}}
iex(11)> Laske.summa(1,2)
3
iex(12)> Laske.summa(1,0.4)
"Lasken vain kokonaislukuja."
iex(13)> Laske.summa("a",2)
"Lasken vain kokonaislukuja."
iex(14)>
```

Kuva 3.6: Esimerkki Elixir-vahdista joka tarkastaa funktion parametrit.

4 Toimialamallit Elixirillä

Evansin (2003) mukaan toimialapohjaisen suunnittelun tuottamien mallien implementointi tietokoneella vaatii olio-ohjelmointiin tai logiikkaan perustuvan ohjelmointikielen. Toisaalta Wlaschin 2018 demonstroi toimialapohjaisen suunnittelun mallien implementointia funktionaalisella #F ohjelmointikielellä. Elixir aiheisissa konfrenseissa on pidetty useita esitelmiä toimialaperusteisen suunnittelun käytöstä Elixir-ohjelmistojen toteutuksessa (Velasco, 2023; Swadia, 2020; Martin, 2017). Esimerkit siis kumoavat Evansin näkemyksen.

Toimialaperusteisessa suunnittelussa oleellinen yhteisen kielen saavuttaminen työryhmälle on luonnollisesti oleellista, kun mallit toteutetaan Elixirillä. Tämän kielen käyttö tietorakenteiden, muuttujien ja funktioiden nimissä on sekä mahdollista että suotavaa (Gray ja Tate, 2019; Velasco, 2023). Ohjelmiston kerroksellisen rakenne ja toimialalogiikan toteutus omassa kerroksessaan on myös osa Elixirillä toteutettuja toimialamalleja (Gray ja Tate, 2019; Swadia, 2020). Toimialan jakaminen rajattuihin konteksteihin ja niiden välisten suhteiden kuvaaminen kontekstikartoilla on analogista Elixir-toteutuksessa.

Virhe- ja häiriösietoisuuden kannalta alitoimialat (rajatetut kontekstit) voidaan suorittaa Elixirissä omissa prosesseissaan ja muodostaa näille tarkoituksenmukaisia supervisionpuita. Prosessien välinen viestintämekanismi mahdollistaa datan ja funktioden välittämisen rajatulta kontekstilta toiselle. Rajattujen kontekstien välinen kommunikointi voi tapahtua myös kutsumalla yhteen rajattuun kontekstiin kuuluvia funktioita toisesta. Kommunikaation muihin rajattuihin konteksteihin tulee ideaalisti tapahtua vain yhden prosessin tai funktion kautta. Rajatettujen kontekstien välisen tiedonsiirron formaatin tulee olla määritelty. Kun prosessi on saanut dataa toista rajattua kontekstia edustavalta prosessilta, sen tulee tarkastaa datan oikeellisuus ennen käsittelyyn laittoa. Samoin ulos rajatusta kontekstista lähetetty data on tarkistettava. Lähetettävän datan rakenteen on myös oltava sellainen, ettei se aiheuta häiriötä vastaanottavassa prosessissa. Tulevan datan oikeellisuus ja oikean käsittelyn varmistaminen on tehtävissä Elixirin vahdeilla. Näillä voidaan toteuttaa kevyt toimialaperusteisen suunnittelun määrittelemä suojakerros. On mahdollista, että tulevan datan tietorakenne on erilainen, kuin mitä vastaanottava rajattu konteksti pystyy käsittelemään. Tällöin tarvitaan monimutkaisempi suojakerros.

Kun toimialapohjaisen suunnittelun mallit toteutetaan olio-ohjelmointiin perustuvalla ohjelmointikielellä, entiteetit ja arvo-oliot ovat toteutettavissa luonnollisesti olioina. Näin ei

toimita funktionaalisessa ohjelmoinnissa. Entiteetit ja arvo-oliot toteutetaan tietorakenteina, joita palveluina toimivat funktiot kutsuttuina käsittelevät (Velasco, 2023; Gray ja Tate, 2019). Tehtaat voivat luoda nämä tietorakenteet. Repositorioina voivat toimia funktiot, jotka välittävät entiteettejä ja arvo-olioita toimialakerroksen ja infrastruktuurikerroksen välillä. Infrastuktuurikerros käsittelee varsinaiset tietokannat ja voidaan toteutaa Ecton* avulla.

Elixirin laajennettavuus mahdollistaa toimialakohtaisten ohjelmointikielten luomisen. Näissä käytetään ilmaisujen nimeämiseen toimialaosaajien ja ohjelmoijien projektia varten määrittelemää yhteistä kieltä. Tämä tekee ohjelmakoodista helpommin ymmärrettävää toimialaosaajille. Kielen laajennus Elixirissä on mahdollista makroja ja sigilejä käyttäen.

Toimialapohjaista suunnittelua käytetään mikropalveluarkkitehtuuriin perustuvien sovellusten kehityksessä (Özkan et al., 2023; Vural ja Koyuncy, 2021). Näissä ohjelmistoissa omissa proseisseissaan ajetut keskenään kommunikoivat mikropalvelut muodostavat sovelluksen (Lewis ja Fowler, 2014). Elixir-sovellukset ajetaan luonnostaan prosesseissa ja ympäristö sisältää prosessien välisen kommunikaatiomekanismin. Elixsirin ominaisuudet muodostavat alustan toimialapohjaista suunnittelua käyttäville mikropalveluarkkitehtuuriin perustuville sovelluksille.

*https://hexdocs.pm/ecto/getting-started.html

5 Johtopäätökset

Tämä tutkielma tarkastelee toimialaperusteista suunnittelua ja sillä tuotettujen toimialamallien implementointia Elixir-ohjelmointikielellä siten, että tuloksena olevat ohjelmistot ovat mahdollisimman vikavapaita ja robusteja erilaisia ajonaikaisia häiriöitä vastaan.

Ohjelmistoissa olevat viat voivat aiheuttaa suoritusaikana virheitä ja häiriöitä, joiden negatiiviset vaikutukset voivat olla huomattavat. Soveltuvilla suunnittelumenetelmillä voidaan ohjelmistovikojen määrää vähentää sekä varautua häiriötilanteisiin. Toimialaperusteinen suunnittelu on mahdollisuus tällaiseksi menetelmäksi.

Ohjelmistovikojen, niistä johtuvien virheiden ja muiden ohjelmakoodin häiriöaltistusten vähentämisessä toimialaperusteista suunnittelua käyttäen korostuvat

- luodun mallin oikeellisuus,
- ohjelmakoodin tuotto suoraan mallista,
- työryhmän yhteinen kieli,
- toimialakohtaiset ohjelmointikielet,
- kerroksellinen arkkitehtuuri,
- suojakerros ja
- rajatut kontekstit eli alatoimialat.

Toimialaperusteisessa suunnittelussa toimialasta tehdään malli, joka implementoidaan ohjelmakoodiksi. Tyypillinen mallinnustapa on UML-diagrammien käyttö, mutta myös muita vaihtoehtoja on. Valitusta mallinnustavasta riippumatta ohjelmakoodi noudattaa rakenteeltaan ja käsitteiltään mallia. Mallinnusprosessi on iteratiivinen. Jos ohjelmakoodia ei pysty toteuttamaan mallin mukaan, muutetaan mallia vastaamaan ohjelmakoodia. Jos malli on virheellinen, se voi johtaa vikoja sisältävään ja siten virheherkempään koodiin. Mallien avulla voi ohjelman toimintaa simuloida ja paljastaa mahdolliset vikakohdat. Tapauksissa, joissa ohjelmakoodi tuotetaan automaattisesti malleista, satunnaisten ohjelmoijien virheistä johtuvien vikojen määrä vähenee.

Toimialaperusteisessa suunnittelussa toimialaosaajat ovat isossa roolissa. Näiden sekä ohjelmistoarkkitehtien, ohjelmoijien ja muiden ohjelmistoprojektiin osallistuvien on omaksuttava yhteinen kieli, joka määrittelee mallissa ja ohjelmassa olevat käsitteet ja toiminnot. Ohjelmointityötä varten toimialalle voidaan tehdä oma toimialakohtainen ohjelmointikielensä, jota voidaan käyttää sekä mallien teossa että joissain tapauksissa implementoida varsinainen ohjelma. Projektissa osallistujien väliseen kommunikaatioon luotu yhteinen kieli ja ohjelmointiin luotu toimialakohtainen ohjelmointikieli voivat käyttää samoja termejä. Toimialakohtaisen ohjelmointikielen käyttö ohjelmakoodissa auttaa toimialaosaajia ymmärtämään paremmin ohjelmakoodia ja siten tunnistamaan toimialaan liittyvät viat tai mahdolliset virhekohdat. Sen käytöllä on mahdollisuus vähentää ohjelmoijien satunnaisia virheitä.

Kerroksellinen arkkitehtuuri selkeyttää ohjelman rakennetta. Toimialapohjaisessa suunnittelussa kaikki toimialaan liittyvä ohjelmakoodi keskitetään omaan kerrokseensa. Tämä kerros jakautuu rajattuihin konteksteihin eli alatoimialaoihin. Näiden suhdetta kuvataan kontekstikartoilla. Ohjelmistoprojektin laajuudesta riippuen, rajattuja kontekseja voivat kehittää omat työryhmänsä. Rajattujen konstekstien välisen kommunikaation tulee olla hyvin suunniteltua. Olio-ohjelmointia käytettäessä yksi rajattuun kontekstiin kuuluva olio toimii juurena, jonka kautta konstekstien välinen kommunikaation toimii. Funtionaalista ohjelmointia käytettäessä juurena voi toimia jokin rajattuun kontekstiin kuuluva funktio, tai Elixiriä käytettäessä prosessi. Rajattujen kontekstien väliin voidaan tehdä suojakerros, joka varmistaa, että saapuva data on oikeanlaista ja käsitellään oikeassa muodossa. Tämä vähentää virheiden ja häiriöiden mahdollisuutta.

Ohjelmakoodi on mahdollista kirjoittaa sellaiseksi, että sen suoritys ei pysähdy virhe- tai häiriötilanteessa. Elixir-ohjelmointikielessä on valmiina monitorointirakenteet, joilla tämä voidaan toteuttaa. Kaikki ohjelmakoodi Elixirissä ajetaan prosesseissa, jotka muistuttavat käyttöjärjestelmien säikeitä, mutta ovat näihin verrattuna huomattavan keveitä. Prosesseja voi olla käynnissä yhtäaikaisesti lähes rajaton määrä. Tämä mahdollistaa ohjelmistojen virhesietoisen toteutuksen. Elixirin sisältämiä prosessien monitoirointirakenteita kutsutaan supervisoreiksi. Supervisor-prosessi valvoo sille määrättyjä prosesseja. Jos jokin näistä prosesseista pysähtyy virheeseen tai häiriöön, se lähettää viestin supervisor-prosessille, joka käynnistää pysähtyneen prosessin ja valitusta monitorointistrategiasta riippuen muita prosesseja uudelleen . Jos supervisor-rakenteiden valmiit uudelleenkäynnistysstrategiat eivät ole soveltuvia, voidaan monitorointi toteuttaa myös alemman tason rakenteilla ja määrittää soveltuvat strategiat uudelleenkäynnistykseen.

Elixiriä voidaan käyttää toimialaperusteisen suunnittelun tuottamien mallien toteutukseen. Oleellisin ero olio-ohjelmointiin perustuviin ohjelmointikieliin on toimialaperusteisen suunnittelun entiteettien ja arvo-olioiden toteuttaminen tietorakenteina olioiden sijaan. Ohjelmistojen kerroksellinen rakenne, toimialaan liittyvän ohjelmakoodin eristäminen omaan kerrokseensa, rajatut kontekstit ja suojakerros ovat kaikki toteutettavissa Elixirissä ilman vaikeuksia. Suojakerros voidaan toteuttaa vahdeilla. Elixirin laajennettavuus mahdollistaa toimialakohtaisten ohjelmointikielten luomisen, joita voidaan käyttää toimialamallien toteutuksessa. Laajennukset voidaan toteuttaa makroja tai sigilejä käyttäen.

Toimialaperusteisen suunnittelun käyttö vähentää ohjelmakoodissa olevia vikoja ja auttaa paikantamaan häiriöalttiita kohtia ohjelmistoissa. Elixiriä käyttäen ohjelmat voidaan tehdä virhesietoisiksi. Toimialaperusteinen suunnittelu ja Elixir muodostavat yhdistelmän, jonka käyttö mahdollistaa vähävikaisempien, virhe- ja häiriösietoisten ohjelmistojen tuotannon.

Toimialaperusteinen suunnittelu liittyy muihin malliperusteisiin menetelmiin. Yleisimpien malliperusteisten menetelmien yhtenäisyydet, eroavuudet, edut ja rajoitteet olisi edullista selvittää. Elixir soveltuu tämän tutkimuksen perusteella toimialamallien toteutukseen. Vertailu saman toimialamallin toteutuksesta jollakin olio-ohjelmointikielellä ja Elixirillä olisi arvokas molempien vahvuuksien selvittämiseksi.

Lähteet

Ballou, K. (2015). *Learning Elixir*. Packt Publishing.

Booch, G., Jacobson, I. ja Rumbaugh, J. (1998). *The Unified Modeling Language user guide*. Addison-Wesley.

Boronat, A. (2019). "Code-First Model-Driven Engineering: On the Agile Adoption of MDE Tooling". *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, s. 874–886.

Cao, H. (2020). "A Systematic Study for Learning-Based Software Defect Prediction". *IOP Conf. Series: Journal of Physics: Conf. Series 1487*. DOI: doi:10.1088/1742-6596/1487/1/012017.

Chambers, J. M. (2014). "Object-Oriented Programming, Functional Programming and R". *Statistical Science* 29.2, s. 167–180. DOI: 10.1214/13-STS452.

Dooley, J. (2011). *Software Development and Professional Practice*. Apress.

Elixir (2023). URL: https://elixir-lang.org (viitattu 23. 10. 2023).

Erlang (2023). URL: https://www.erlang.org/ (viitattu 28. 10. 2023).

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Gray, J. E. ja Tate, B. (2019). *Designing Elixir Systems with OTP Write Highly Scalable Self-Healing Software with Layers*. The Pragmatic Programmers.

Hoffmann, B., Urquhart, N., Chalmers, K. ja Guckert, M. (2022). "An empirical evaluation of a novel domain-specific language – modelling vehicle routing problems with Athos". *Empirical Software Engineering* 27.27. DOI: https://doi.org/10.1007/s10664-022-10210-w.

Khononov, V. (2022). *Learning Domain-Driven Design*. O'Reilly.

Kärkkäinen, H. ja Hujanen, M. (2023). *Mikä oli Ville Tapion IT-ymmärrys? 5 avointa kysymystä Vastaamosta*. URL: https://www.is.fi/digitoday/tietoturva/art-2000009428901.html (viitattu 03. 11. 2023).

Lewis, J. ja Fowler, M. (2014). *Microservices a definition of this new term*. URL: https://martinfowler.com/articles/microservices.html (viitattu 09. 11. 2023).

Lotz, G., Peters, T., Zrenner, E. ja Wilke, R. (2010). "A Domain Model of a Clinical Reading Center - Design and Implementation". *32nd Annual International Conference*

of the IEEE EMBS Buenos Aires, Argentina, August 31 - September 4, 2010, s. 4530–4533.

Läufer, K. (2008). "A Stroll through Domain-Driven Development with Naked Objects". *Computing in Science Engineering* May/June, s. 76–83.

Martin, R. (2017). *Perhap: Applying Domain Driven Design and Reactive Architectures to Functional Programming*. URL: https://youtu.be/kq4qTk18N-c?si=IMD9JNyDe2Hl85Nw (viitattu 20. 11. 2023).

Millett, S. ja Tune, N. (2015). *Patterns, Principles, and Practices of Domain-Driven Design 1st Edition*. Wrox.

Mukund, M. (2007a). "A taste of functional programming — 1". *Resonance* 12.8, s. 27–48.

– (2007b). "A taste of functional programming — 2". *Resonance* 12.9, s. 40–63.

OMG (2023). *MDA® - THE ARCHITECTURE OF CHOICE FOR A CHANGING WORLD*. URL: https://www.omg.org/mda/ (viitattu 09. 11. 2023).

Oukes, P., Andel, M. van, Folmer, E., Bennett, R. ja Lemmen, C. (2021). "Domain-Driven Design applied to land administration system development: Lessons from the Netherlands ". *Land Use Policy* 104.

Rademacher, F., Sorgalla, J. ja Sachweh, S. (2018). "Challenges of Domain-Driven Microservice Design A Model-Driven Perspective". *IEEE software* 35.3, s. 36–43.

Selic, B. (2003). "The Pragmatics of ModelDriven Development". *IEEE Software* 20.5, s. 19–25.

Siegel, J. M. (2014). *Object Management Group Model Driven Architecture (MDA) MDA Guide rev. 2.0 OMG Document ormsc/2014-06-01*. URL: https://www.omg.org/cgi-bin/doc?ormsc/14-06-01 (viitattu 10. 11. 2023).

Snoeck, M., Poelmans, S. ja Dedene, G. (2000). "A Layered Software Specification Architecture". *19th International Conference on Conceptual Modeling Salt Lake City, Utah, USA, October 9-12, 2000 Proceedings*, s. 454–469.

StackOverflow (2023). *StackOverflow platform's 2023 Developer survey, n>90000*. URL: https://survey.stackoverflow.co/2023/#overview (viitattu 04. 11. 2023).

Swadia, J. (2020). *Domain-Driven Design with Elixir*. URL: https://youtu.be/fx3BmpzitUg?si=Qcu2_zVz-LeQZ5Xp (viitattu 20. 11. 2023).

UML (2005). *INTRODUCTION TO OMG'S UNIFIED MODELING LANGUAGE™ (UML®)*. URL: https://www.uml.org/what-is-uml.htm (viitattu 16. 11. 2023).

Wang, Y., Li, S., Liu, H., Zhang, H. ja Pan, B. (2022). "A Reference Architecture for Blockchain-based Traceability Systems Using Domain-Driven Design and Microservices".

29th Asia-Pacific Software Engineering Conference (APSEC). DOI: 10.1109/APSEC57359.2022.00039.

Weber, S., Karger, P. A. ja Paradkar, A. (2005). "A Software Flaw Taxonomy: Aiming Tools At Security". *Proceedings of the 2005 workshop on Software engineering for secure systems—building trustworthy applications*.

Velasco, G. (2023). *Using DDD concepts to create better Phoenix Contexts*. URL: https://youtu.be/JNWPsaO4PNM?si=rGaHZ9QeO_OaMOEp (viitattu 20. 11. 2023).

Whatsapp (2012). URL: https://blog.whatsapp.com/1-million-is-so-2011 (viitattu 28. 10. 2023).

Wlaschin, S. (2018). *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. Pragmatic Bookshelf; 1st edition.

Vural, H. ja Koyuncy, M. (2021). "Does Domain-Driven Design Lead to Finding the Optimal Modularity of a Microservice?" *IEEEAccess* 9, s. 32721–32733.

Zhivich, M. ja Cunningham, R. K. (2009). "The Real Cost of Software Errors". *IEEE SECURITY PRIVACY* 7.2, s. 87–90.

Zhong, C., Huang, H., Zhang, H. ja Li, S. (2022). "Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation". *Software: Pracice and Experience* 52.12, s. 2574–2597.

Zhong, C., Zhang, H., Huang, H., Chen, Z., Li, C., Liu, X. ja Li, S. (2023). "DOMICO: Checking conformance between domain models and implementations". *Software: Pracice and Experience*.

Özkan, O., Babur, Ö. ja Brand, M. van den (2023). "Domain-Driven Design in Software Development: A Systematic Literature Review on Implementation, Challenges, and Effectiveness". URL: https://arxiv.org/ftp/arxiv/papers/2310/2310.01905.pdf (viitattu 11. 11. 2023).