



Kandidutkielma

Tietojenkäsittelytieteen kandiohjelma

Toimialalähtöinen ohjelmistokehitys ja Elixir vikasietoisten ohjelmistojen tuotannossa

Rolf Wathén

21.11.2023

MATEMAATTIS-LUONNONTIETEELLINEN TIEDEKUNTA
HELSINGIN YLIOPISTO

Yhteystiedot

PL 68 (Pietari Kalmin katu 5)
00014 Helsingin yliopisto

Sähköpostiosoite: 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 vikasietoisten 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	November 21, 2023	25 pages	
Tiivistelmä — Referat — Abstract			
<p>level.</p> <p>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</p>			
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			
Software study track			

Sisällys

1	Johdanto	1
2	Toimialaperusteinen ohjelmistokehitys	3
2.1	Menetelmän kuvaus	3
2.2	Implementointeja käytännössä	9
3	Elixir	14
3.1	Elixir-prosessit	15
3.2	Elixir-prosessien virhesietoisuus	15
3.3	Laajennettavuus	18
3.4	Vahdit	20
4	Toimialamallit Elixirillä	21
5	Johtopäätökset	22
	Lähteet	23

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 uudelleenasetusten 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 suorite-tilanteissa, 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 tehty 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 toimiala-spesifiseksi kieleksi ja sisältää ominaisuuksia joilla ohjelmakoodista voidaan tehdä virhe- ja häiriösietoinen.

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

Tutkielma jakautuu viiteen lukuun. Luku 2 esittelee toimialaperusteisen suunnittelun, käy läpi liittyviä julkaisuja ja pohtii sen käyttöä vikasietoisuuden näkökulmasta. Luku 3 käsit-

tee Elixir-ohjelmointikieltä ja sillä kirjoitettujen ohjelmien vikasietoisuutta. Luku 4 pohdii Elixirin-käyttöä toimialaperusteisen suunnittelun implementointityökaluna. Viimeinen luku esittää tutkielman johtopäätökset.

2 Toimialaperusteinen ohjelmistokehitys

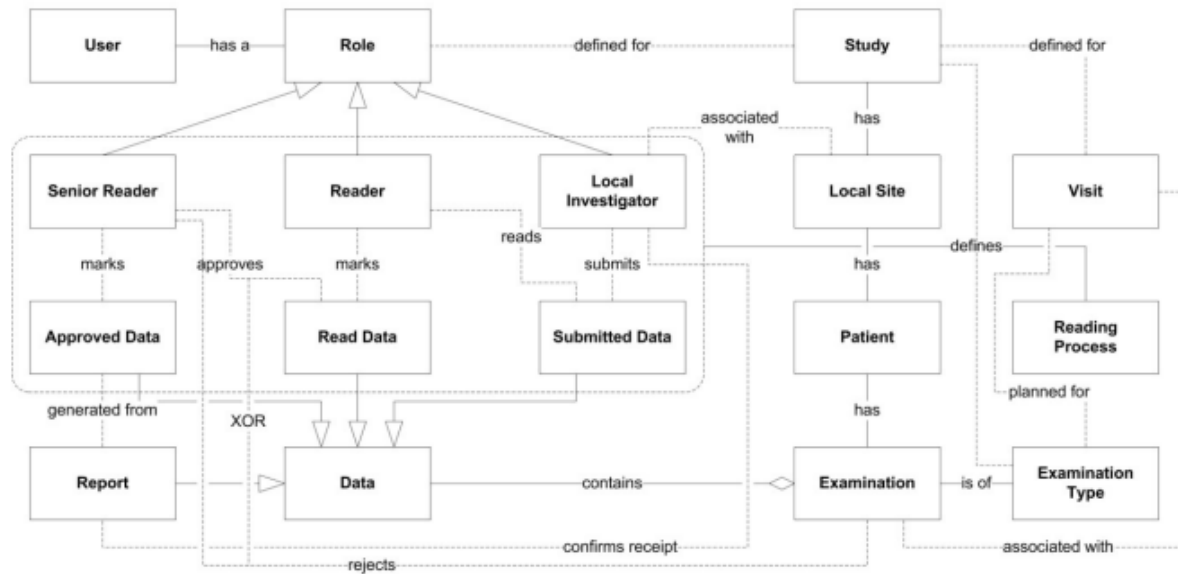
Toimialaperusteinen suunnittelu kuuluu malliperusteisiin ohjelmistokehityskäsitteisiin. Siihen läheisesti liittyviä muita menetelmiä ovat malliperusteinen arkkitehtuuri (Model-Driven Architecture, MDA) (OMG, 2023), malliperusteinen tuotanto (Model-Driven Engineering, MDE (Boronat, 2019)) ja malliperusteinen kehitys (model-driven development, MDD (Selic, 2003)). Toimialaperusteisessa suunnittelussa, eng. Domain Driven Design tai DDD, ohjelmistosuunnittelun keskiössä ovat toimialasta (eng. domain) luodut mallit. Evansin vuonna 2003 (Evans, 2003) julkaisemaan 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). Malliperusteisia menetelmiä kuten MDA, MDE ja MDD käsitellään kirjallisuudessa yleisesti laajasti ja niillä on paljon yhtymäkohtia toimialaperusteiseen suunnitteluun.

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 edesauttaa ohjelmistojen toiminnallisuutta, kattavuutta ja loppukäyttäjän tarpeiden toteutumista. Tämän lisäksi mallinnusvaiheessa voidaan ottaa huomioon skaalautuvuus, häiriöalttius, turvallisuus ja laajennettavuus. Jos nämä on toteutettu ohjelmakoodiin ilman kunnollista suunnittelua ja mallinnusta, voi muutosten teko olla vaikeaa ja kallista.

Malli on selkeästi organisoitu abstraktio toimialasta, johon on tarkasti valittu sen tärkeimmät elementit (Evans, 2003). Mallissa 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 tarkoituksenmukainen 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 klinisten kokeiden tulosten analysointia harjoittavan laitoksen toimintaprosessista (Lotz et al., 2010).

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

On tärkeää, että työryhmään kuuluvat kommunikoivat keskenään samoilla termeillä. Evans käyttää termiä *ubiquitous language* kuvaamaan tätä kieltä. *Ubiquitous* kääntyy "kaikkialla läsnäolevaksi". Yhteinen 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 toimialosaajien kanssa on korostetun oleellista. Molemmiin puoliin oppiminen on tärkeää. Kun ohjelmoija ymmärtää mitä on tekemässä ja miksi, hänen on helpompaa 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 implementointi 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).

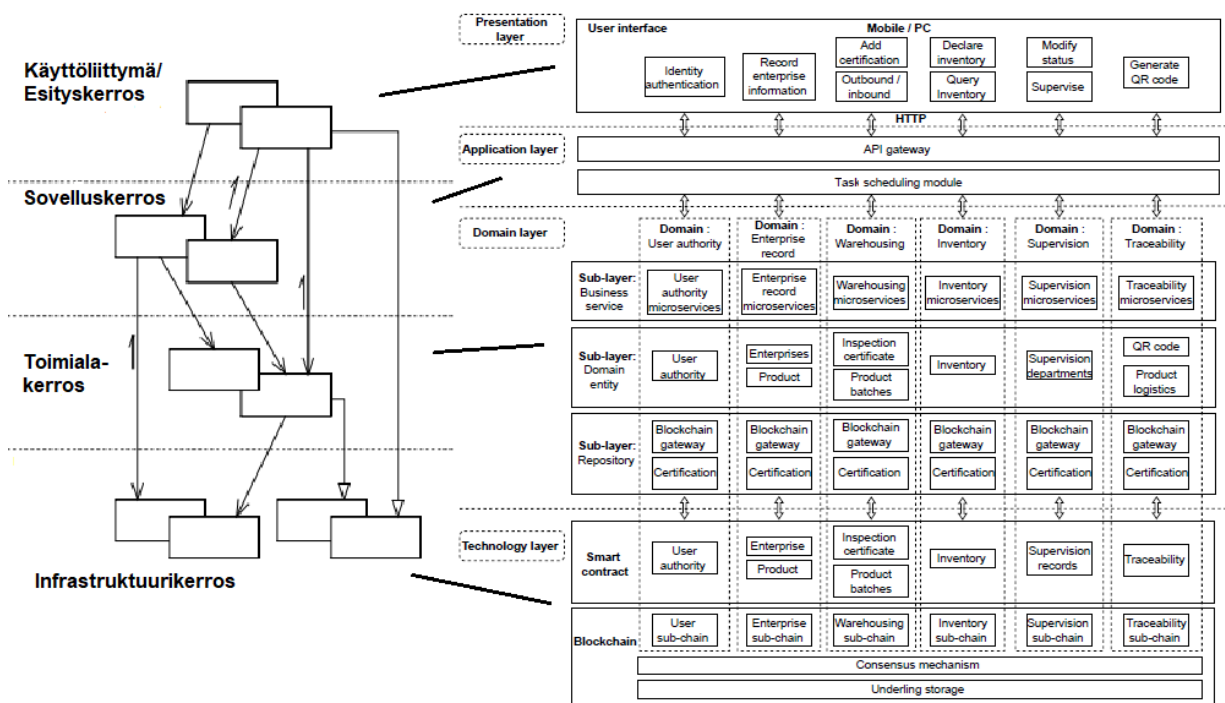
On tärkeää että mallin sisältö on tarkkaan rajattu ja mallien väliset rajat ovat selkeät, kirjallisuus käyttää termiä rajattu konteksti (bounded context) (Evans, 2003; Özkan et al., 2023; Vural ja Koyuncu, 2021; Siegel, 2014). Yksi toimialamalli voi siis muodostua useasta rajatusta kontekstista. Näiden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

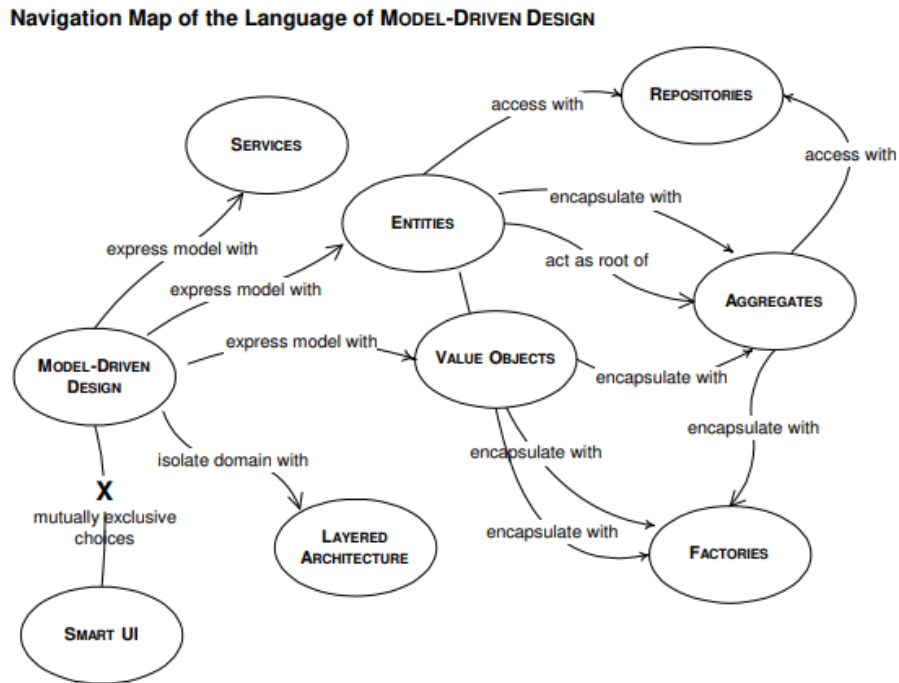
taa virheherkempään toteutukseen (Zhong et al., 2022; Zhong et al., 2023). Tämän vuoksi mallien oikeellisuus on mahdollisuuksien mukaan varmistettava esimerkiksi simuloimalla. Tapauksissa joissa koodi on kirjoitettu ilman DDD-prosessia tai muuta mallinnusmetodiikkaa, mutta joissa halutaan ottaa sellainen käyttöön, voidaan käyttää työkaluja jotka luovat malleja olemmassa olevasta koodista (Boronat, 2019).

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 käyttöliittymä/esityskerros (user interface/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 ydin kerrokseen tehdyt muutokset välittyvät automaattisesti muihin kerroksiin (Läuffer, 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 on toiminto joka suoritetaan kutsusta. Entiteetti on yksikkö jolla on yksilöllinen identiteetti, esimerkiksi pankkitili(numero) tai henkilö jonka identifioi 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 olio jonka muu ohjelma näkee. Tehdas (factory) on se osa ohjelmakoodia joka luo halutut oliot ja aggregaatit. Repositoriot tallentavat ja toimittavat pyydettyä 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 kielensä, domain-specific language (jäljempänä DSL). Näissä kielissä käytetään toimialan sanastoa ja rakenteita (Evans, 2003). Toimialakohtaisia kieliä käyttäen toimiala-asiantuntijat voivat helpommin osallistua ohjelmakoodin tuottamiseen (Hoffmann et al., 2022). Implementoinnissa DSL tavallisesti käännetään jollekin yleiselle ohjelmointikielelle. DSL voidaan myös

rakentaa jonkun yleisen ohjelmointikielen päälle. Tämä vaatii kyseiseltä kieleltä muokattavuutta (Evans, 2003).

Riippuen siitä millä tekniikalla mallit ovat toteutettu, on mahdollista tuottaa suoritettava ohjelmakoodi suoraan mallista (Siegel, 2014). Näin on mahdollista välttää ohjelmakoodiin tulevia vikoja. Mallit voivat myös mahdollistaa simuloinnin mahdollistaen vikojen havaitsemisen lisäksi haitallisten sivuvaikutusten ja prosessin pullonkaulojen olemassaolon (Siegel, 2014). Esimerkiksi Simulink-ohjelmalla voidaan mallintaa ja simuloida teollisuusprosessi uudelleen käytettävissä olevista yksikköprosesseista*.

2.2 Implementointeja käytännössä

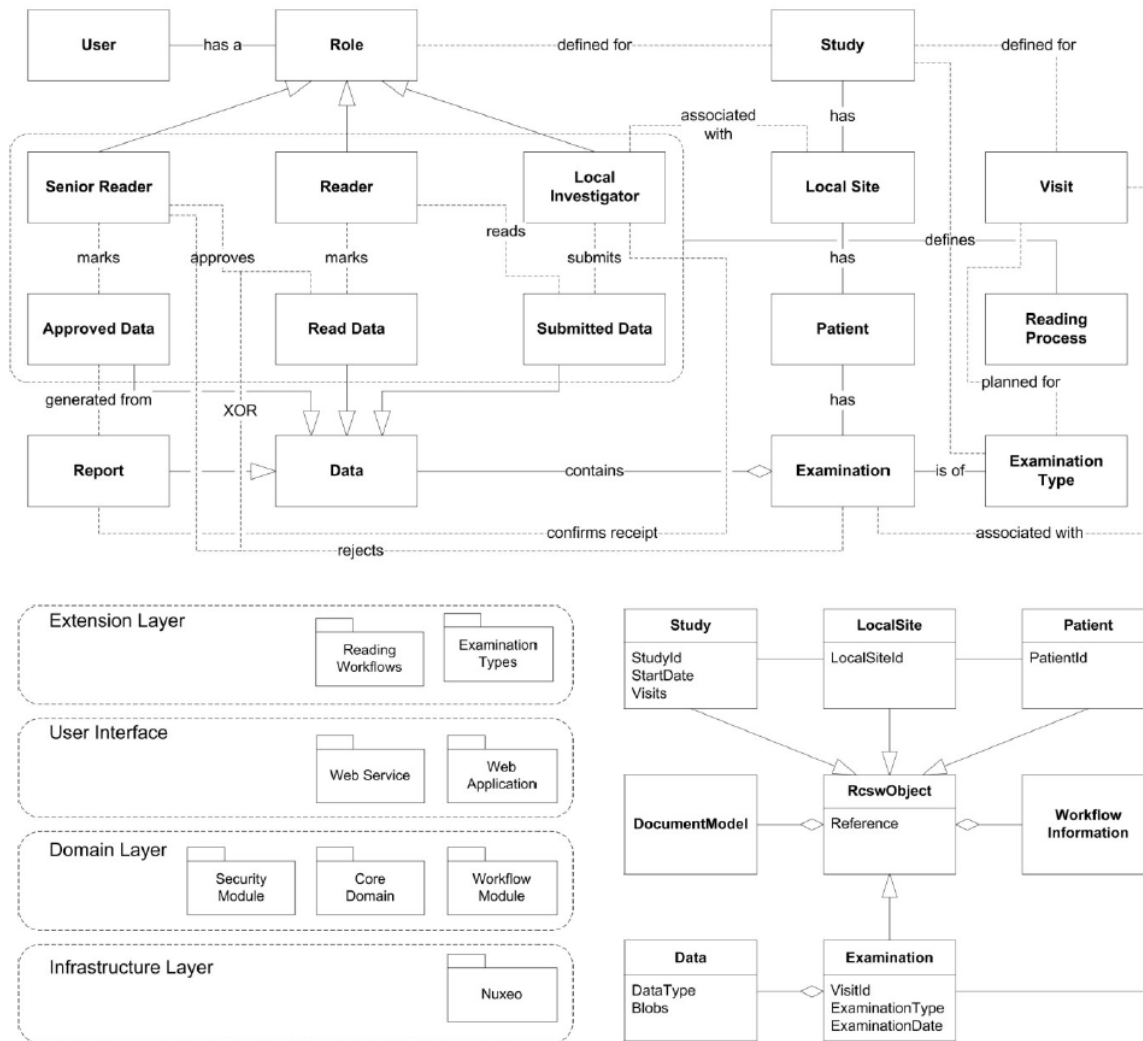
Kliinisten kokeiden tulostenanalysointiohjelmisto

Lots et al. (2010) ovat tehneet kliinisten kokeiden tuloksia analysoivalle analysointikeskustelle 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 tekijöiden implementoinnin ohjelmistoon. Voi päätellä hyvin mallinnettujen koedatan kulun ja käyttöoikeuksien määrittelyn pienentävän 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 toteutettiin 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.

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

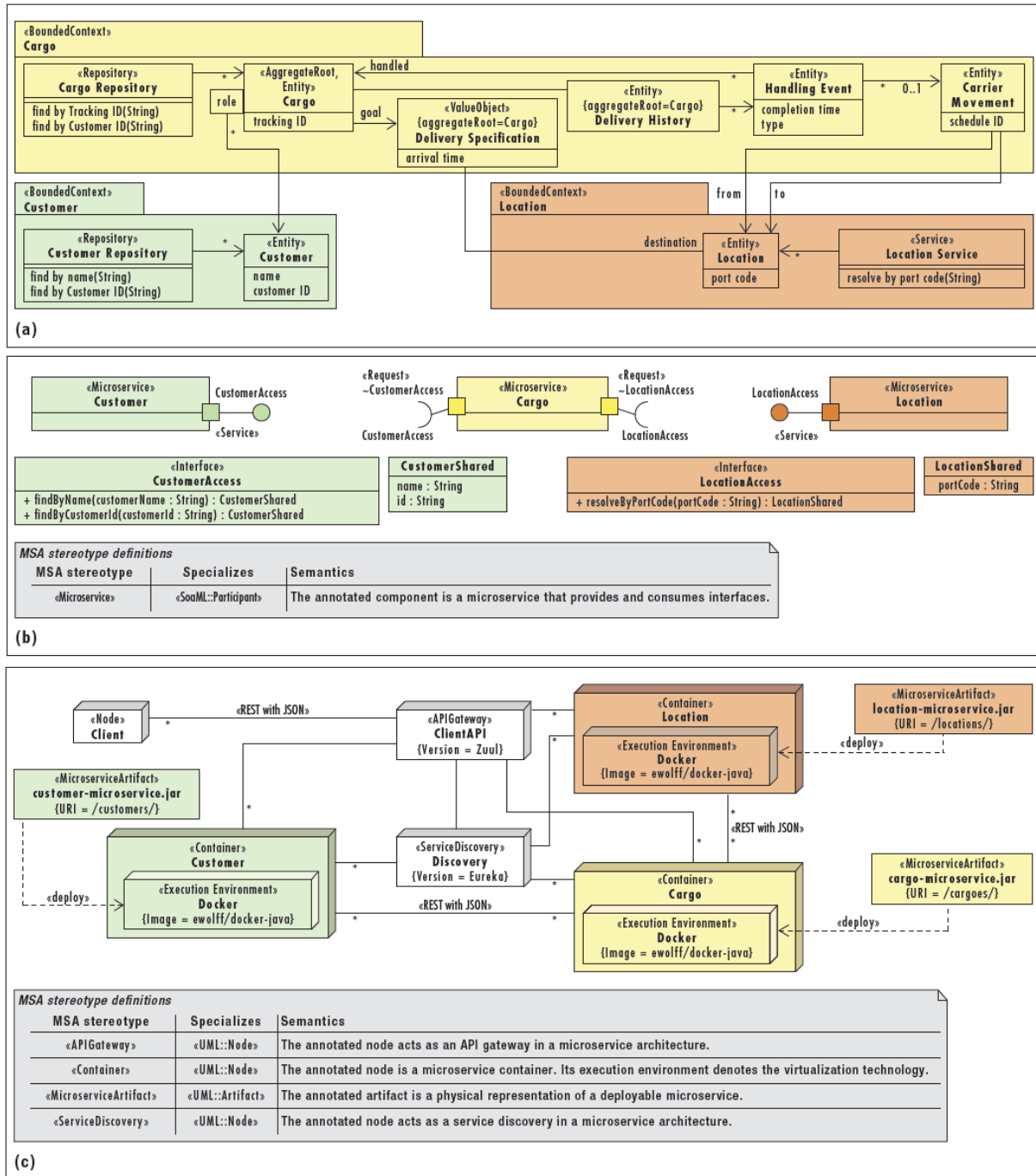
[†]<https://doc.nuxeo.com/nxdoc/quick-overview/>



Kuva 2.4: Toimialamalli klinisten kokeiden analysointia harjoittavan keskuksen toimintaprosessista (yl-
lä) jossa laatikossa käytetty yhteisen kielen (ubiquitous language) termejä. Vasemmalla alhaalla toteutuk-
sen kerroksellinen rakenne ja oikealla alhaalla toimialamallin toteutus (Lotz et al., 2010).

Mikropalvelut

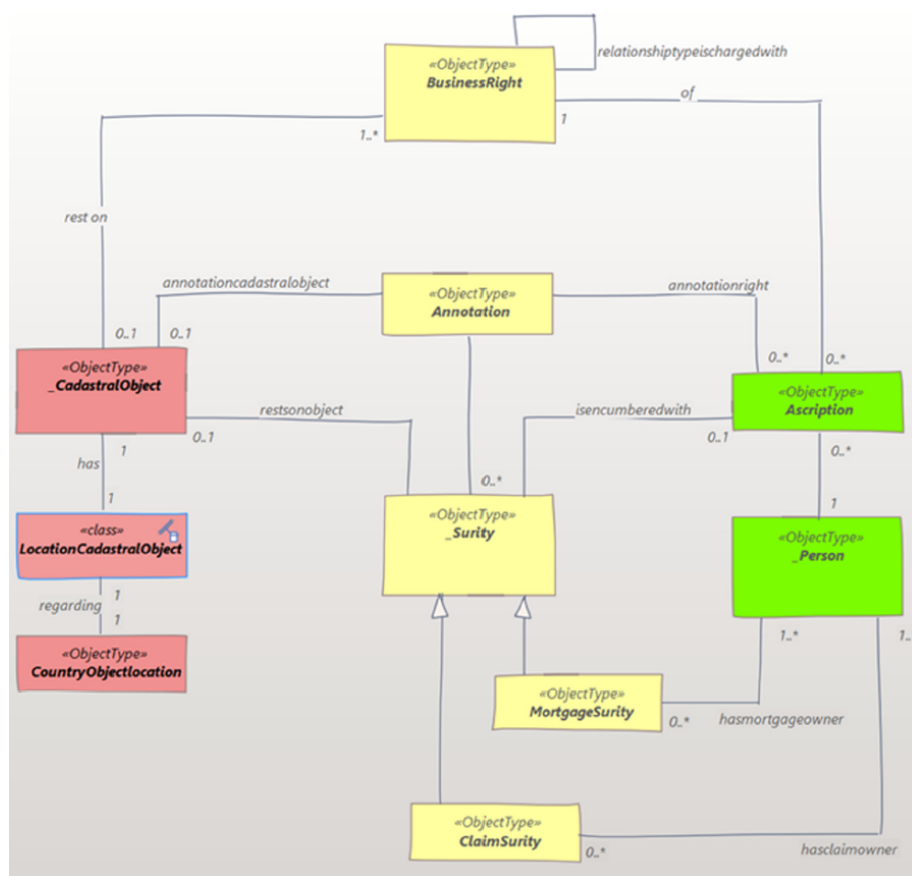
Mikropalveluarkkitehtuuriin perustuvissa ohjelmistoissa omissa proseisseissaan ajetut keskenään kommunikoivat mikropalvelut muodostavat sovelluksen (Lewis ja Fowler, 2014). Kuvassa 2.6 on esitetty toimialaperusteinen malli logistiikalle sekä siitä johdetut mikropalvelumallit rajapintoihin. Vaikka toimialaperusteinen suunnittelu soveltuu mikropalveluarkkitehtuurin omaavien ohjelmistojen suunnitteluun, siinä on tiettyjä haasteita (Rademacher et al., 2018). Toimialaperusteisessa suunnittelussa mallit eivät yleensä ole yksityiskohtaisia. Vaikka toimialapohjaiset mallit olisivat määriteltyjä UML-diagrammeina, mallit ovat epätarkkoja ja eivät 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, jos kahteen eri rajoitettuun kontekstiin kuuluvan toimialakäsitteen suhdetta ei ole selkeästi määritelty. Tästä esimerkkinä kuvassa 2.6 a Delivery Specification ja Location. Toisaalta Vular ja Koyuncu 2021 totesivat DDD:n soveltuvan mikropalveluiden optimaalisen modulaarisuuden määrittämiseen.



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. Mallinnettu käyttäen palveluorientoituneen arkkitehtuurin mallinnuskieltä (Service Oriented Architecture Modeling Language, SoaML). c) Toimialamallista johdettu malli mikropalvelujen toteutukselle, UML-diagrammi mikropalveluarkkitehtuurin toteutukselle.

Lainhuutojen rekisteröintijärjestelmäohjelmisto

Hollannin Suomen maanmittauslaitosta vastaava Kadaster * on käyttänyt toimialaperusteista suunnittelua suunnitellessaan ja toteuttaessaan omistustietojärjestelmäänsä korvaamaan vanha Cobolilla kirjoitettu järjestelmä (Oukes et al., 2021). Toimialan mallinnus tehtiin UML-malleilla jotka muunnettiin Java-luokiksi. ISO 19152 esittää standardimallin maanhallinnan toimialalle. Tätä voitiin käyttää ja luodulla mallilla on yhtäläisyyksiä standardimallin. Toimialaperusteisen suunnittelun todettiin soveltuvan kyseisen toimialan mallintamiseen. Tärkeä tekijä oli toimialaosajien ja ohjelmoijien yhteiden kieli.



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

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

3 Elixir

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 ohjelmointi koostuu lähinnä funktioiden määrittelemisestä, joiden palauttama arvo riippuu vain funktion argumenteista. Funktiolla ei myöskään ole muihin laskuihin vaikuttavia sivuvaikutuksia.(Chambers, 2014) Olio-ohjelmoinnissa taas kaikki laskenta tehdään tarkoituksenmukaisen rakenteen omaavien 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 läheisesti 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.

[†]<https://elixirforum.com/>

[‡]<https://elixir-lang.org/install.html>

[§]<https://hexdocs.pm/elixir/1.16/introduction.html>,<https://elixirschool.com/en>

[¶]<https://discord.com/>

3.1 Elixir-prosessit

Elixirissä koodi ajetaan prosesseiksi kutsutuissa yksiköissä (Elixir, 2023; Ballou, 2015). Ne muistuttavat käyttöjärjestelmien säikeitä (thread), mutta ovat näihin verrattuna keveitä. 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. Kuva 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 pysyy 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 itse asiassa 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 täytyessä, esimerkiksi kun jostain tapahtumasta on kulunut liian pitkä aika.

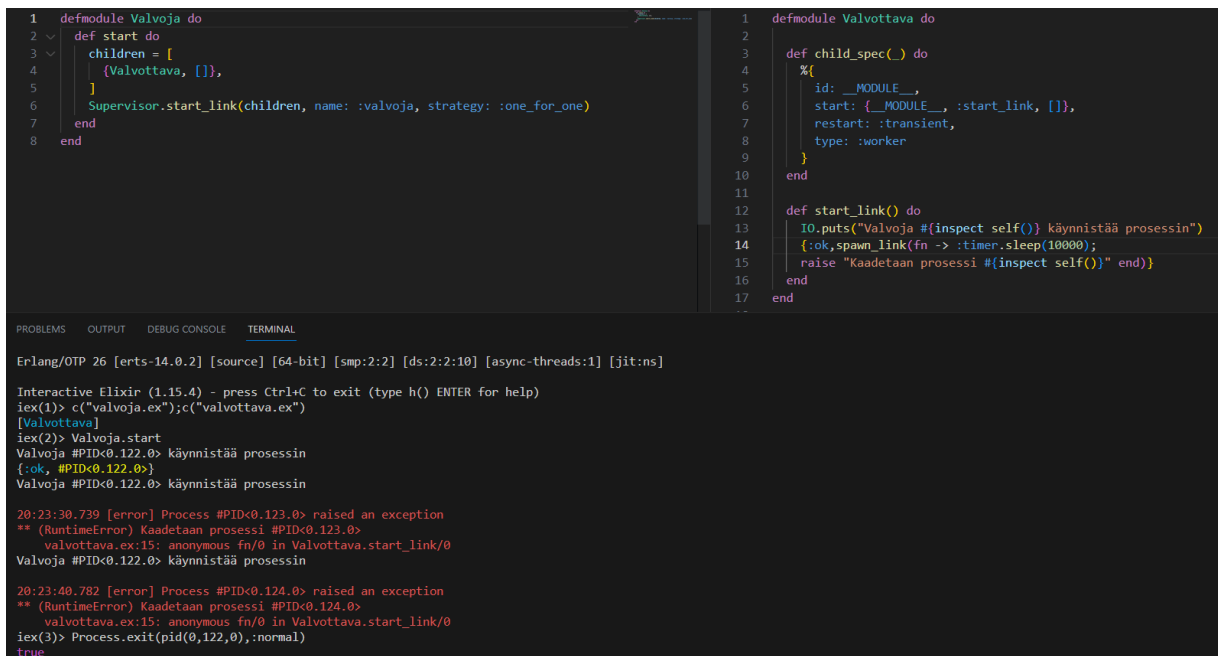
Kuvassa 3.2 on esittetty yksinkertainen supervisor-prosessi*. Käytetty uudelleen käynnistysstrategia on `:one_for_one`. Terminaalista näkyy kuinka prosessi kaatuu kymmenen sekunnin välein ja supervisor käynnistää uuden vastaavan prosessin.

Prosessin uudelleen käynnistäminen auttaa tilanteissa joissa kyseessä ei ole systemaattinen jokaisella suorituskerralla tapahtuva ohjelmointivirhe. Esimerkki on verkkoyhteyden häiriintyminen. Uudelleen käynnistämisen voi ajatella mahdollisesti piilottavan ohjelmointivikoja, joiden aiheuttamat virheet johtavat prosessin pysähtymisen vain tietyissä erikoistilanteissa. Tämä on huomioitava testauksessa, jossa prosessit on testattava useilla erilaisilla syötteillä.

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[†].

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

[†]<https://hexdocs.pm/elixir/1.16/try-catch-and-rescue.html#errors>



```

1 defmodule Valvoja do
2   def start do
3     children = [
4       {Valvottava, []},
5     ]
6     Supervisor.start_link(children, name: :valvoja, strategy: :one_for_one)
7   end
8 end

1 defmodule Valvottava do
2
3   def child_spec(_) do
4     %{
5       id: __MODULE__,
6       start: {__MODULE__, :start_link, []},
7       restart: :transient,
8       type: :worker
9     }
10  end
11
12  def start_link() do
13    IO.puts("Valvoja #{inspect self()} käynnistää prosessin")
14    {:ok, spawn_link(fn -> :timer.sleep(10000);
15                     raise "Kaadetaan prosessi #{inspect self()}" end)}
16  end
17 end

```

```

Erlang/OTP 26 [erts-14.0.2] [source] [64-bit] [smp:2:2] [ds:2:2:10] [async-threads:1] [jit:ns]

Interactive Elixir (1.15.4) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> c("valvoja.ex");c("valvottava.ex")
[Valvottava]
iex(2)> Valvoja.start
Valvoja #PID<0.122.0> käynnistää prosessin
{:ok, #PID<0.122.0>}
Valvoja #PID<0.122.0> käynnistää prosessin

20:23:30.739 [error] Process #PID<0.123.0> raised an exception
** (RuntimeError) Kaadetaan prosessi #PID<0.123.0>
    valvottava.ex:15: anonymous fn/0 in Valvottava.start_link/0
Valvoja #PID<0.122.0> käynnistää prosessin

20:23:40.782 [error] Process #PID<0.124.0> raised an exception
** (RuntimeError) Kaadetaan prosessi #PID<0.124.0>
    valvottava.ex:15: anonymous fn/0 in Valvottava.start_link/0
iex(3)> Process.exit(pid(0,122,0),:normal)
true

```

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> joka kaadetaan 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.

```

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)> 

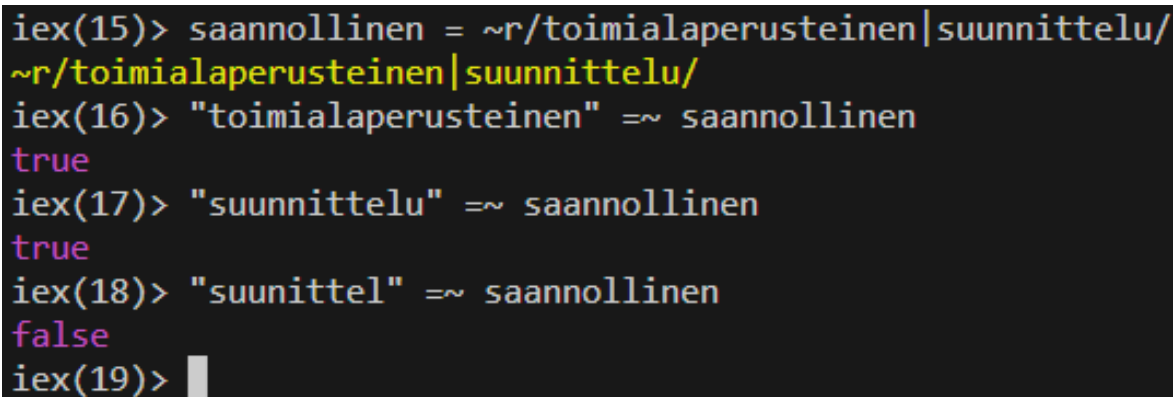
```

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

3.3 Laajennettavuus

Elixiriä on mahdollisuus laajentaa toimialakohtaiseksi kieleksi (DSL). Elixiriin on tehty DSL:ä pidettäviä laajennuksia esimerkiksi tietokantojen käsittelyyn* ja koneoppimiseen †.

Elixiriä voi laajentaa makroilla tai *sigileillä*. Hyvä ohjelmointitapa on käyttää funktioita makrojen sijaan jos mahdollista (Elixir, 2023). Sigilejä Elixirissä on valmiina useita. 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)> "suunnittel" =~ 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: Sinetti `~KAANNA` joka kääntää annetun luvun

3.4 Vahdit

4 Toimialamallit Elixirillä

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

Toimialan ali-toimialat ovat erotettu rajoitettuihin konteksteihin, jotka voidaan toteuttaa Elixir-prosesseina. Prosessien välinen viestintämekanismi mahdollistaa datan ja funktioiden välittämisen prosessilta toiselle. Rajattu konteksti voi koostua yhdestä tai useammasta prosessista, mutta kommunikaation muihin rajattuihin konteksteihin tulee tapahtua vain yhden prosessin kautta. Rajattujen kontekstien välisen tiedonsiirron formaatin tulee olla määriteltä. 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

Toimialapohjai

Toimialaspesifisen kielen luominen geneerisen ohjelmointikielen kuten Elixirin päälle vaatii tältä laajennettavuutta. Elixir on laajennettava. Yksi esimerkki tästä on sinettien käyttö jota käsiteltiin luvussa 3.

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

5 Johtopäätökset

Toimialaperusteisessa suunnittelussa toimialasta tehdään malli joka implementoidaan ohjelmakoodiksi. Ohjelmakoodi noudattaa rakenteeltaan ja käsitteiltään mallia. Prosessi on iteratiivinen, ohjelmakoodin perusteella voidaan myös muuttaa mallia. Jos malli on virheellinen, se voi johtaa vikaherkempään koodiin.

Mallien avulla voi ohjelman toimintaa simuloida ja paljastaa mahdolliset vikakohdat. Tapauksissa joissa ohjelmakoodi tuotetaan suoraan malleista satunnaisten virheiden määrä vähenee. Jos malli on virheellinen, on toteutus todennäköisesti vikaherkempi.

Toimialaperusteisessa suunnittelussa toimialaosajaat ovat isossa roolissa. Ohjelmistoarkkitehtien, ohjelmoijien ja muiden ohjelmistoprojektiin osallistuvien on omaksuttava yhteinen kieli, joka määrittelee mallissa ja ohjelmassa olevat käsitteet ja toiminnot. Toimialoja varten voidaan tehdä oma toimialaspesifinen kielensä joilla voidaan tehdä sekä malli että implementoida varsinainen ohjelma.

Toimialaperusteista suunnittelua on käytetty usein mikropalveluarkkitehtuuriin perustuvien ohjelmistojen toteutuksessa. Näissä ohjelmistoissa omissa prosesseissaan ajatut keskenään kommunikoivat mikropalvelut muodostavat sovelluksen. Elixirissä voidaan ajaa useita prosesseja samanaikaisesti, tarvittaessa hajautetusti. Elixirin prosesseilla ovat valmiina mekanismit joilla ne pystyvät kommunikoimaan keskenään. Elixir näyttää tarjoavat alustan mikropalveluarkkitehtuuriin perustuville sovelluksille jotka on suunniteltu toimialaperusteista suunnittelua käyttäen.

Toimialakohtaisilla kielillä ovat mahdollisia sekä toimialan mallintaminen että mallin implementointi ohjelmakoodiksi. Jos toimialakohtainen kieli on oikein implementoitu, sen käyttö vähentää satunnaisia ohjelmistovirheitä. Toimialakohtainen kieli implementoidaan usein käyttäen alustana jotain yleistä ohjelmointikieltä. Käytetyltä alustalta vaaditaan muokattavuutta. Elixir on muokattava ohjelmointikieli jolle on toteutettu toimialakohtaisia kieliiä. Eräs mekanismi toimialakohtaisen kielen toteutukseen on `sigil_{merkki}`-rakenteen rakenteen käyttö.

Virheellinen malli voi johtaa vikaherkempään toteutukseen.

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](https://doi.org/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](https://doi.org/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). URL: <https://survey.stackoverflow.co/2023/#overview> (viitattu 04. 11. 2023).
- Swadia, J. (2020). *Domain-Driven Design with Elixirg*. 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](https://doi.org/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/JNWPsa04PNM?si=rGaHZ9Qe0_0aMOEp (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: Practice 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: Practice 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).

