Kanditutkielma

Tietojenkäsittelytieteen kandiohjelma

Toimialalähtöinen ohjelmistokehitys ja Elixir vikasietoisten ohjelmistojen tuotannossa

Rolf Wathén

14.11.2023

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 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 14, 2023	19 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

Software study track

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 virheitä, joiden etsintä ja poisto, debuggaus, on oleellinen osa ohjelmointia.Virheet voivat aiheuttaa myös suuria taloudellisia menetyksiä. Jo 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 ohjelmistovirheet voivat aiheuttaa esimerkiksi turvallisuusuhkia (Weber et al., 2005).

Tietokoneohjelman suorittamisen keskeyttävät virheet tapahtuvat usein suoritetilanteissa, joita ohjelmoija ei ole osannut ennakoida. Ne voivat tapahtua myös ohjelman ulkopuolisista syistä. Esimerkkeinä ovat muistipiirin pettäminen, massamuistin lukuvirhe, toisen ohjelman aiheuttamaa häiriö tai verkkoyhteyden pettäminen. Tietokoneohjelmassa olevia virheitä voi pyrkiä vähentämään kirjoitusvaiheessa erilaisilla tekniikoilla.

Ohjelma voidaan kirjoittaa sellaiseksi, että sen suoritus ei keskeydy virheeseen. Ohjelman rakenteen suunnittelu ja valitut työkalut kuten käytetty ohjelmointikieli vaikuttavat tähän tavoitteeseen. Toimialaperusteinen suunnittelu (Evans, 2003) on yksi mahdollisuus suunnittelumetodiksi. Elixir-ohjelmointikieli (Elixir, 2023) on lähtökohtaisesti kehitetty sisältämään ominaisuuksia, jotka tekevät suoritetusta ohjelmasta vikasietoisemman.

Tässä tutkielmassa käsitellään toimialaperusteisen suunnittelun ja Elixir-ohjelmointikielen yhteiskäyttöä vikasietoisen ohjelmakoodin kirjoitukseen.

Tutkielma jakautuu viiteen lukuun. Luku 2 esittelee toimialaperusteisen suunnittelun, käy läpi liittyviä julkaisuja ja pohtii sen käyttöä vikasietoisuuden näkökulmasta. Luku 3 käsittelee Elixir-ohjelmointikieltä ja sillä kirjoitettujen ohjelmien vikasietoisuutta. 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 (MDE, Model-Driven Engineering) 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 (jäljempänä DDD) perusteoksena (Läufer, 2008; Lotz et al., 2010; Wlaschin, 2018; Rademacher et al., 2018; Vural ja Koyuncy, 2021; Oukes et al., 2021; Özkan et al., 2023; Zhong et al., 2022; Zhong et al., 2023). Kirja on yhteenveto julkaisuaan edeltäneen kahden vuosikymmenen aikana kehittyneistä käytännöistä ja periaatteista (Evans, 2003). Tämän jälkeen aihetta ovat kokonaisuutena käsitelleet ainakin Khonovov (2022) ja Millett ja Tune (2015).

2.1 Menetelmän kuvaus

Toimialaperusteisessa suunnittelussa keskeistä ovat toimialasta, esimerkiksi pankkitoiminnasta, muodostetut mallit. Mallit muodostavat työryhmät joihin kuuluvat sekä suunnittelijat, toimialaosaajat että varsinaiset ohjelmoijat. Luotujen mallien rakennetta noudatetaan varsinaisessa ohjelmakoodissa. 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 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). Zhong et al.

(2023) ovat näyttäneet, että parempi kommunikaatio työryhmässä johtaa parempiin malleihin. Nämä taas oikein implementoituina johtavat vähemmän vikaherkkään 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)

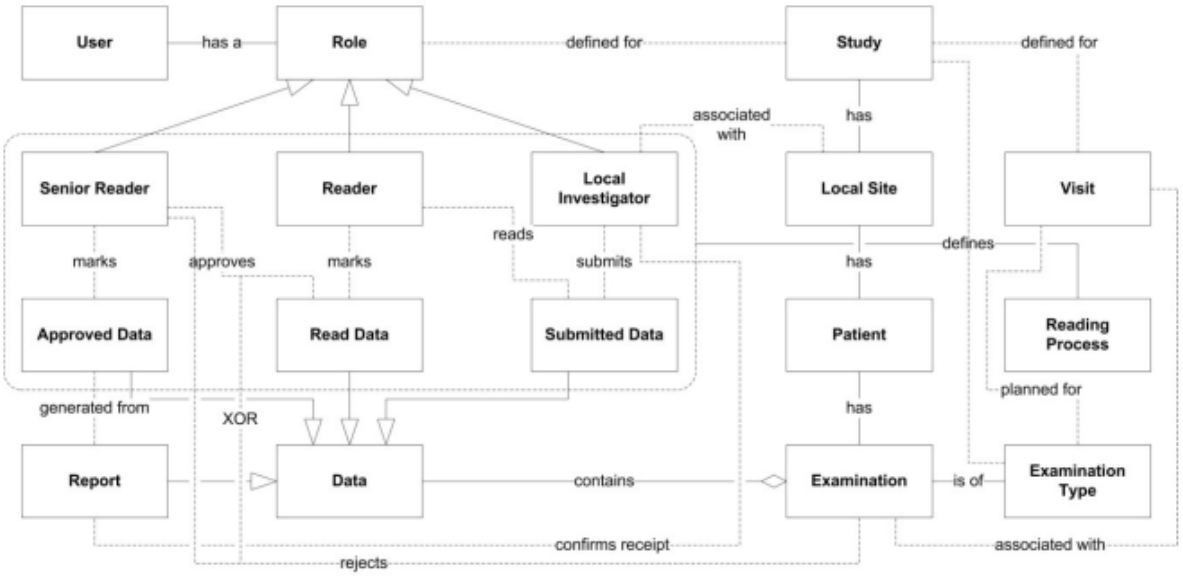
Ohjelmoijien säännöllinen keskustelu toimialaosaajien kanssa on korostetun oleellista. Oppiminen on tärkeässä roolissa sekä ohjelmoijalle että toimiala-asiantuntijalle. Kun ohjelmoija ymmärtää mitä on tekemässä ja miksi, hänen on helmpompi keskittyä oleellisiin asioihin. Vaikeasti ylläpidettävät ohjelmistot syntyvät yleensä, kun toimialaa tai käyttöympäristöä ei ole kunnolla ymmärretty ja mallinnettu (Evans, 2003; Zhong et al., 2022).

Mallli 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) kuten luokkadiagrammit. Malli ei ole välttämättä sama kuin diagrammi, eikä diagrammien tulisi olla liian yksityiskohtaisia (Evans, 2003). Malli voi olla myös esimerkiksi luonnollisella kielellä selkeäsi kirjoitettu kuvaus tai ohjelmistokoodia(Evans, 2003). Virheellinen malli voi johtaa vikaherkempään toteutukseen(Zhong et al., 2022; Zhong et al., 2023).

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. Toisaalta jos 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; paper, 2014). On oleellista, että malleja yhtä osiota kohti on yksi, se jonka perusteella se implementoidaan ohjelmistoon (Evans, 2003). Osiot muodostavat suurempia kokonaisuuksia, joille ovat omat mallinsa.

Toimialamalleissa säilyy kuvaus ohjelmakoodin toiminnasta(paper, 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).

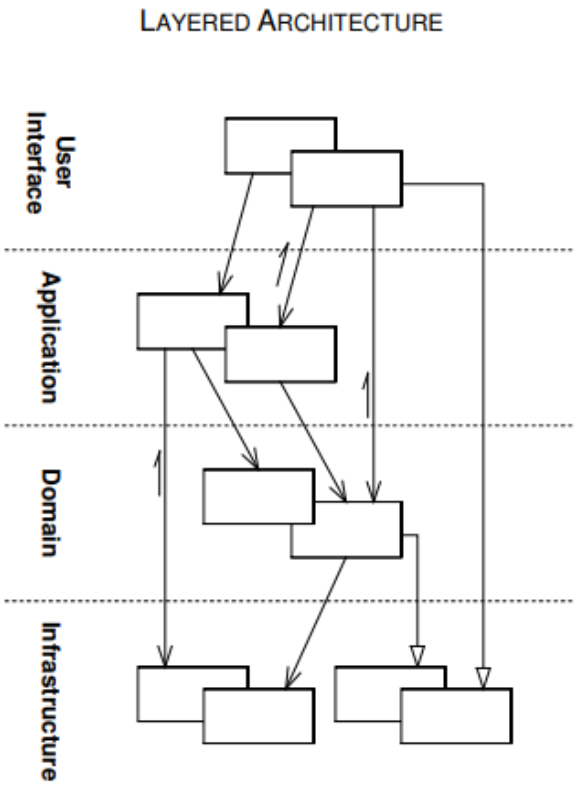


Kuva 2.1: Toimialamalli kliinisten kokeiden tulosten analysointia harjoittavan laitoksen toimintaprosessista (Lotz et al., 2010).

Ohjelmistovirheiden mahdollisuus pienenee kun ohjelman rakenne tulee kaikkien hyväksymästä mallista (Evans, 2003). Virheellinen malli voi johtaa vikaherkempään toteutukseen(Zhong et al., 2022; Zhong et al., 2023). Tämän vuoksi mallien oikeellisuus on mahdollisuuksien mukaan varmistettava. Tapauksissa joissa koodi on jo 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). DDDn yhteyteen on kehitetty kehysympäristöjä (framework) kuten Naked Objects joilla ydin kerrokseen tehdyt muutokset välittyvät automaattisesti muihin kerroksiin (Läufer, 2008).

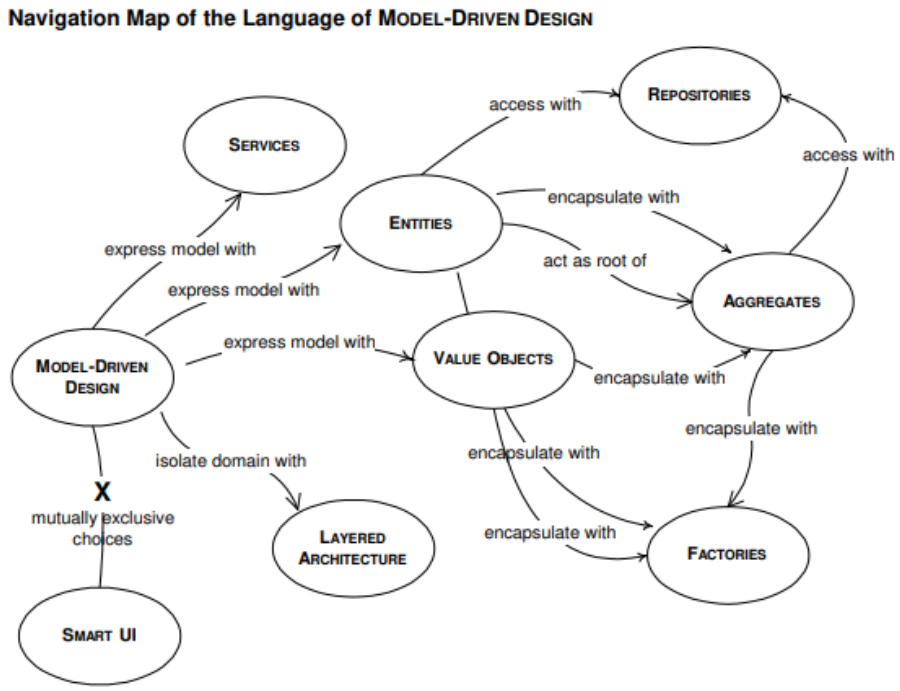
Malli tuodaan ohjelmakoodiin palveluiden (services), entiteettien (entities) ja arvo-olioiden (value objects) avulla. Palvelu on toiminto joka suoritetaan kutsusta. Entiteetti on yksikkö jolla on yksilöllinen identiteetti, esimerkiksi pankkitili(numero) tai henkilö jonka identifioi



Kuva 2.2: Kerroksellinen arkkitehtuuri (Evans, 2003).

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 ja on ainoa olio jonka muu ohjelma näkee. Tehdas (factory) on se osa ohjelmakoodia joka luo halutut oliot ja aggregaatit. Repositoriot tallentavat ja toimittavat pyydettäessä tarvitut oliot. Mallin tuonti ohjelmakoodiin on esitetty kuvassa 2.3.

Mallinnusta ja implementointia varten voidaan luoda oma toimialakohtainen kielensä, domain-specific language (jäljempänä 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 DSL tavallisesti käännetään jollekin yleiselle ohjelmointikielelle. DSL:n rakentaminen yleisen ohjelmointikielen päälle vaatii tältä kieleltä muokattavuutta(Evans, 2003).



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

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

2.2 Implementointeja käytännössä

Lots et al. (2010) ovat tehneet kliinisten kokeiden tuloksia analysoivalle analysointikeskukselle koemateriaalia ja sen analyysiä hallinnoivan sovelluksen käyttäen toimialaperusteista suunnittelua. Sitä käytettiin toimialan monimutkaisuuden takia, taustalla eivät olleet ohjelmistotekniset haasteet. Sen käyttö johti toimialan parempaan ymmärtämiseen

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

ja mahdollisti kaikkien oleellisten tekijoiden implementoinin ohjelmistoon. Voi päätellä hyvin mallinnettujen koedatan kulun ja käyttöoikeuksien pienentävän todennäköisyyttä koedatan vaarantumisesta ohjelmistovirheen tai suunnitteluvirheen 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. Alhaalta lukien seuraavana kerroksena on toimialakerros jossa on toteutettuna toimialamallin logiikka. Seuraavana kerroksena on käyttöliittymä to verkkosovelluksella ja palvelimella REST rajapintoja käyttäen. Ylimpänä on laajennuskerros esimerkiksi uusien tutkimustyyppien (Examination Type)implementointia varten.

Mikropalveluarkkitehtuuriin perustuvissa ohjelmistoissa omissa proseisseissaan ajetut keskenään kommunikoivat mikropalvelut muodostavat sovelluksen (Lewis ja Fowler, 2014). Kuvassa 2.5 on esitetty toimialaperusteinen malli logistiikalle sekä siitä johdetut mikropalvelumallit rajapinnoille ja toteutukselle. 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 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, jos kahteen eri rajoitettuun konktekstin kuuluvan toimialakäsitteen suhdetta ei ole selkeästi määritelty. Tästä esimerkkinä kuvassa 2.5 a Delivery Specification ja Location. Toisaalta Vular ja Koyuncu 2021 totesivat DDD:n soveltuvan mikropalveluiden optimaalisen modulaarisuuden määrittämiseen.

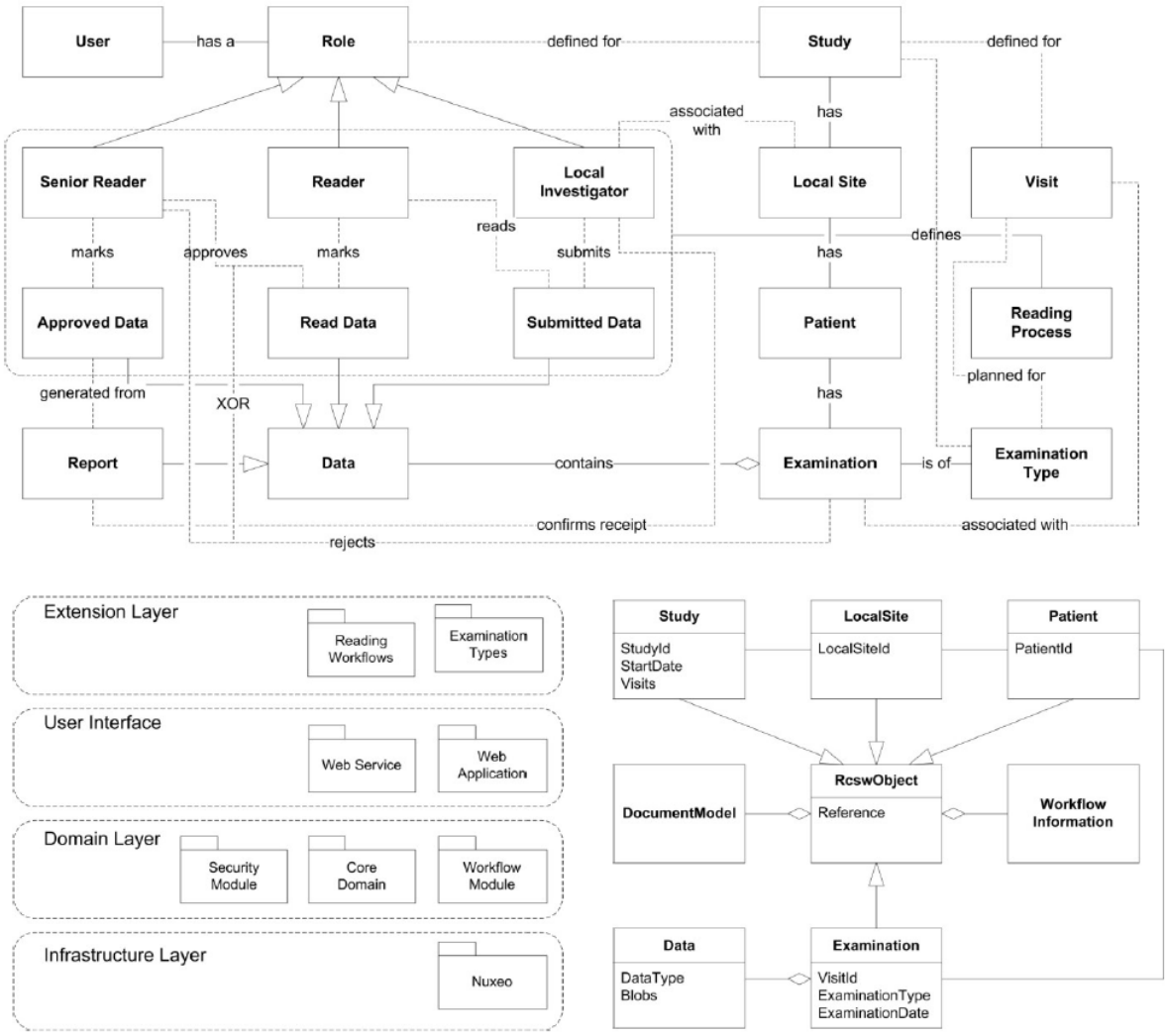
Hollannin Suomen maanmittauslaitosta vastaava Kadaster † on käyttänyt toimialaperusteista suunnittelua suunnitellessaan ja toteuttaessaan omistustietojärjestelmäänsä korvaamaan vanha Cobolilla kirjoitettu järjestelmä (Oukes et al., 2021). Toimialan mallinnus tehtiin UML-malleilla jotka muunnettiin Java-luokiksi. 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 toimialaosaajien ja ohjelmoijien yhteiden kieli. .

Irtonaisia lauseita ehkä myöhempää käyttöön. Wang et al. 2022 ovat käyttäneet DDD:a ja

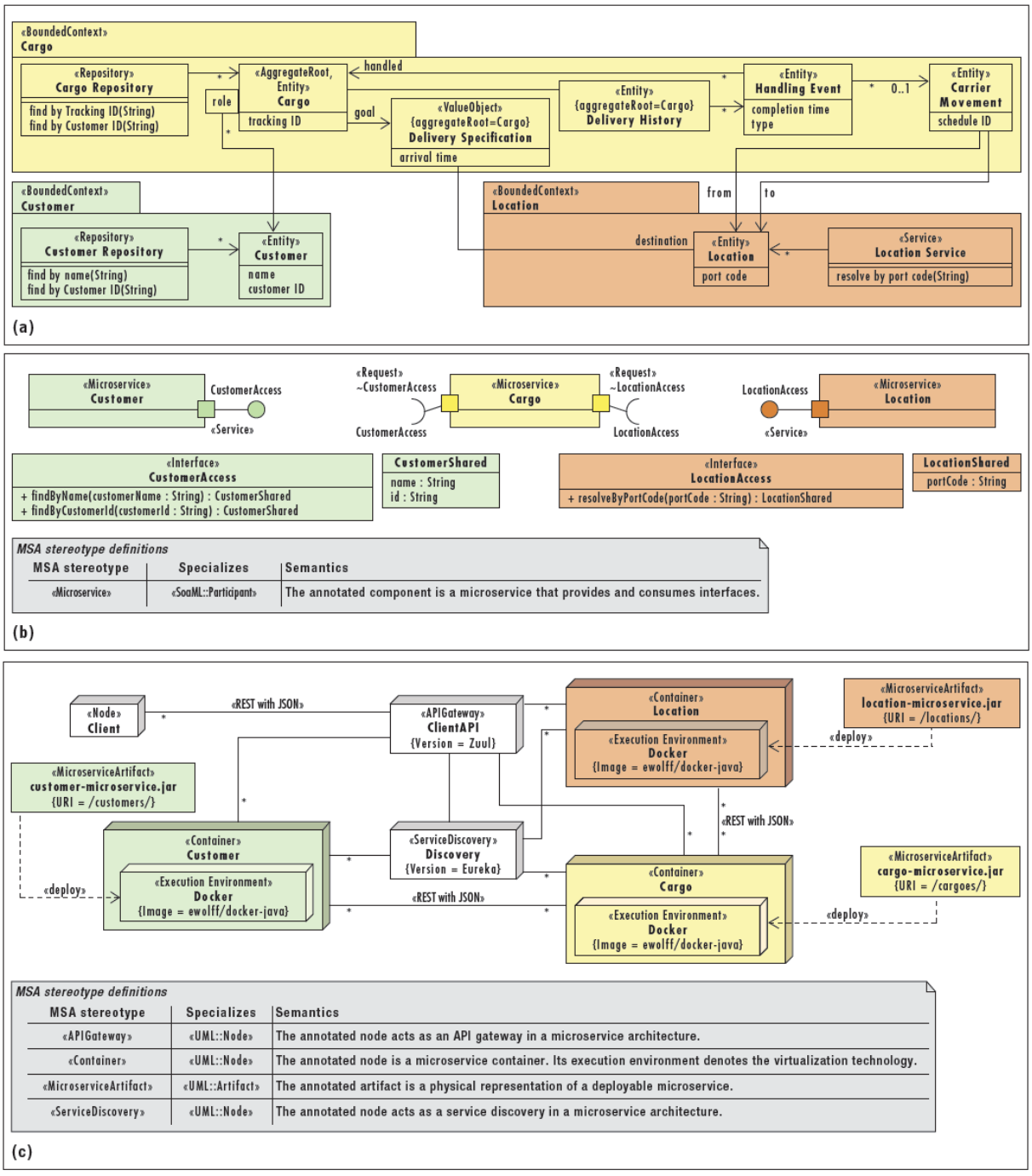
*https://doc.nuxeo.com/nxdoc/quick-overview/

†https://www.kadaster.nl/about-us

mikopalveluita kehysympäristön (framework) luomiseen lohkoketjuihin perustuville seurantajärjestelmille. Özkan et al. 2023 ovat tehneet kirjallisuustutkimuksen DDDn implementoinnista, haasteista ja tehokkuudesta. Tässä työssä vikasietoinen ohjelma on määritelty niin, että ilman toimenpiteitä sen suoritus keskeytyisi vika tai häiriötilanteeseen. Esimerkkitilanteita ovat nollalla jakaminen tai verkkoyhteyden katkeaminen. Ohjelmistovirheet tai huono suunnittelu voivat myös esimerkiksi johtaa ohjelman epätoivottuun toimintaan tai esimerkiksi vakaviin tietovuotoihin (Weber et al., 2005; Kärkkäinen ja Hujanen, 2023), vaikka ne eivät keskeytäkkään ohjelman toimintaa. Jos häiriötilanne johtuu virheestä toimintalogiikassa, voi ohjelman keskeytyminen olla vianetsinnän kannalta hvyödyllistä.



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).



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, UML-diagrammi mikropalveluarkkitehtuurin toteutukselle..

3 Elixir

Elixir on kohtuullisen uusi ohjelmointikieli (Elixir, 2023; Ballou, 2015) jonka suosio on kasvamassa. Stack Overflow-ohjelmointiyhteisön 2023 tekemässä kyselytutkimuksessa noin 4.5% 87150:stä vastaajasta oli viimeisen vuoden aikana käyttäny sitä ja 73% halusi käyttää uudestaan. Jälkimmäinen luku oli korkeampi kuin esimerkksi JavaScriptillä tai Pythonilla (StackOverflow, 2023). Elixir-tutoriaaleja [†] ja asennustiedostot [‡] useille käyttöjärjestelmille ovat saatavilla verkosta.

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 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 tämän kirjastoja. Tavukoodiksi kääntämisen lisäksi Elixiriä voidaan käyttää interaktiivisessa iEx-ympäristössä (interactive elixir)(Elixir, 2023). Tämä mahdollistaa interaktiivisen ohjelmistokehtityksen ja testauksen.

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

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

[§] https://discord.com/

3.1 Elixir-prosessien vikasietoisuus

Elixirissä koodi ajetaan prosesseiksi kutsutuissa yksiköissä(Elixir, 2023; Ballou, 2015). Ne muistuttavat käyttöjärjestelmien säikeitä (thread). 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)>
```

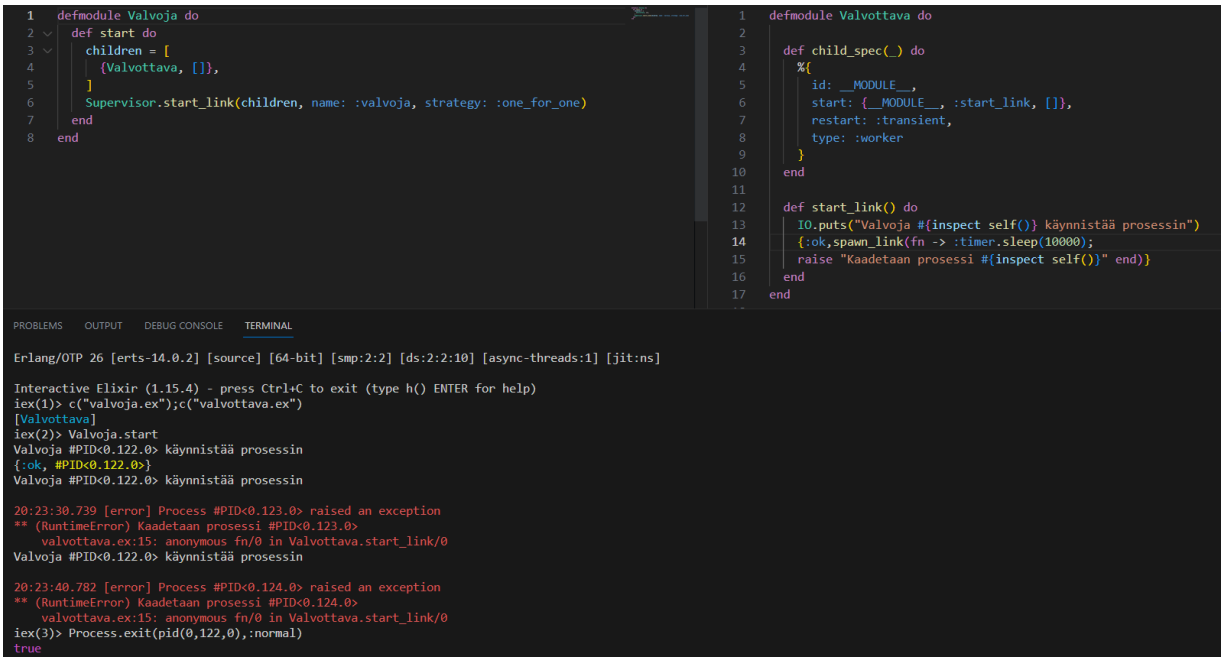
Kuva 3.1: Uuden Elixir-prosessin käynnistäminen, sille viestin lähettäminen ja prosessin sulkeutuminen iEx-ympäristössä.

Yhden prosessin kaatumisen ei tarvitse johtaa koko ohjelmiston kaatumiseen. Oikein kirjoitettuna ohjelma pystyy käynnistämään kaatuneen prosessin helposti uudelleen toimivaksi tiedetystä alkutilasta. Tämä suoritetaan Elixirissä monitorointiprosessien avulla. Näitä kutsutaan Elixirissä 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 supervisor-prosessi voi valvomansa prosessin suhteen noudattaa Elixirissä 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.

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.



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.

Prosessin uudelleen käynnistäminen auttaa tilanteissa joissa kyseessä ei ole systemaattinen jokaisella suorituskerralla tapahtuva ohjelmointivirhe. Esimerkki on verkkoyhteyden häiriintyminen. Uudelleen käynnnistämisen voi ajatella piilottavan ohjelmointivirheet, jotka aiheuttavat prosessin pysähtymisen vain tietyissä erikoistilanteissa. Tämä on huomioitava testauksessa jossa prosessit on testattava useilla erilaisilla syötteillä.

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

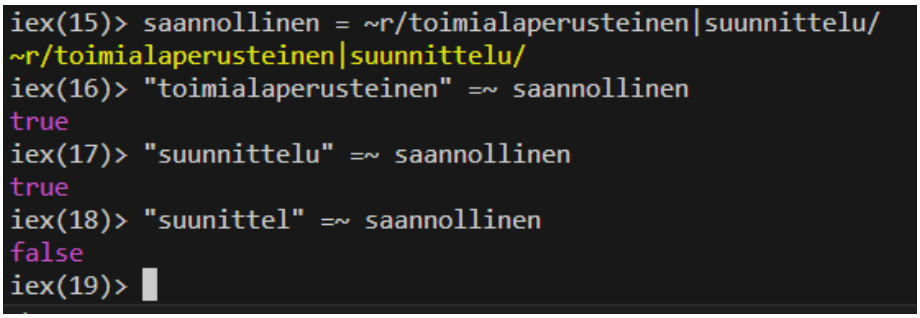
4 Elixir mallien implementointivälineenä

Evansin (2003) mukaan toimialapohajisen suunnittelun tuottamien mallien implementointi tietokoneella vaatii olio-ohjelmointiin perustuvan ohjelmointikielen. Toisaalta ideaaliksi implementointivälineeksi DDDlle Evans mainitsee Prologin, logiikkaperusteisen ohjelmointikielen mutta ei funktionaalista ohjelmointia. Wlaschin 2018 demonstroi toimialapohjaisen suunnittelun mallien implementointia funktionaalisella #F ohjelmointikielellä. Gray ja Tate käyttävä Elixir-ohjelmoinnin oppaassaan konsepteja kuten muuttujien ja funktioiden kuvaava nimeäminen, ohjelman kerroksellinen rakenne ja toimialaan liittyvän koodin eristäminen omaan kerrokseensa (Gray ja Tate, 2019). Voi päätellä, että toimialapohjainen suunnittelu ei vaadi implementointivälineekseen olio-ohjelmointiin perustuvaa ohjelmointikielä.

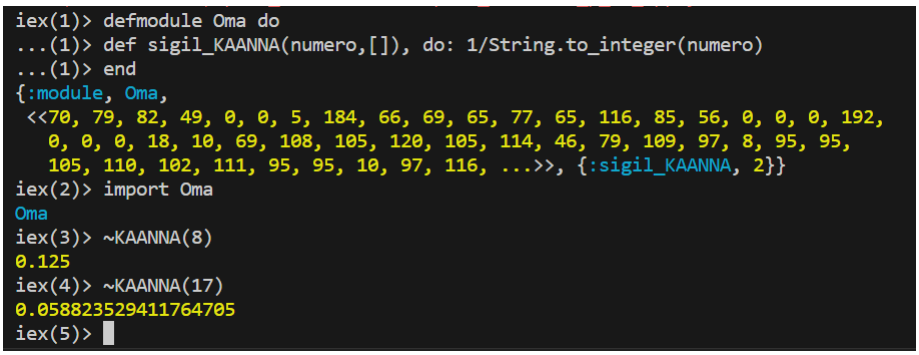
Elixiriä on mahdollisuus laajentaa toimialakohtaiseksi kieleksi (DSL). Elixiriin on tehty DSL:ä pidettäviä laajennuksia esimerkiksi tietokantojen käsittelyyn[†] ja koneoppimiseen [‡]. Yksi mahdollisuus tähän on sinettien (sigil) käyttö. Elixirissä on näitä valmiina useita. Ne alkavat ∼-merkillä, jota seuraa yksi pieni kirjain tai yksi tai useampi iso kirjain. Näiden jälkeen tulee erotin. Viimeisen erottimen jälkeen tulevat valinnaiset modifikaattoorit. Esimerkkinä kuvassa 4.1 ∼r sinetti säännöllisille lauseille. Omia sinettejä voi luoda funktioilla jotka implementoivat sigil_{merkki}-rakenteen. Esimerkkinä ∼KAANNA kuvassa 4.2 joka antaa käänteisluvun.

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 voivat muodostaa alustan DDDllä suunnitelluille mikropalveluarkkitehtuuriin perustuville sovelluksille.

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



Kuva 4.1: Esimerkki sinetistä ~r jonka avulla Elixirissä voidaan käsitellä säännöllisiä lauseita



Kuva 4.2: Sinetti ~KAANNA joka kääntää annetun luvun

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 toimialaosaajat 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 proseisseissaan ajetut 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 kieliä. 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.

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.

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.

paper, O. white (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).

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.

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).

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*.

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).