Kanditutkielma

Tietojenkäsittelytieteen kandiohjelma

Toimialalähtöinen ohjelmistokehityksen ja Elixir-ohjelmointikielen käyttö vikasietoisen ohjelmakoodin tuotannossa

Rolf Wathén

6.11.2023

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 ohjelmistokehityksen ja Elixir-ohjelmointikielen käyttö vikasietoisen ohjelmakoodin 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 6, 2023	14 pages

Tiivistelmä — Referat — Abstract

Write your abstract here.

In addition, make sure that all the entries in this form are completed.

Finally, specify 1–3 ACM Computing Classification System (CCS) topics, as per https://dl.acm.org/ccs. Each topic is specified with one path, as shown in the example below, and elements of the path separated with an arrow. Emphasis of each element individually can be indicated by the use of bold face for high importance or italics for intermediate level.

ACM Computing Classification System (CCS)
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords

algorithms, data structures

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Sisällys

1 Johdanto **1**

2 Toimialaperusteinen ohjelmistokehitys **2**

2.1 Menetelmän kuvaus 2

2.2 Implementointeja käytännössä 6

2.3 Vaikutus vikasietoisuuteen 6

3 Elixir **8**

3.1 Elixir-prosessien vikasietoisuus 8

4 Elixir mallien implementointivälineenä **10**

5 Johtopäätökset **11**

Lähteet **12**

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 ohjelmankirjoittaja ei ole osannut ennakoida. Ne voivat tapahtua myös ohjelman ulkopuolisista syistä. Esimerkkeinä ovat muistipiirin pettämistä, 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 kappaleeseen joista ensimmäinen on Johdanto. Kappale kaksi esittelee toimialaperusteisen suunnittelun, käy läpi liittyviä julkaisuja ja pohtii sen käyttöä vikasietoisuuden näkökulmasta. Kappale kolme käsittelee Elixir-ohjelmointikieltä ja sillä kirjoitettujen ohjelmien vikasietoisuutta. Neljäs kappale vetää yhteen edelliset kaksi kappaletta ja pohtii Elixirin-käyttöä toimialaperusteisen suunnittelun implementointityökaluna. Viimeinen kappale esittää tutkielman johtopäätökset.

2 Toimialaperusteinen ohjelmistokehitys

Evansin vuonna 2003 (Evans, 2003) julkaisema kirja Domain-Driven Design: Tackling Complexity in the Heart of Software mainitaan 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). Özkan et al. 2023 ovat tehneet kirjallisuustutkimuksen DDDn implementoinnista, haasteista ja tehokkuudesta.

2.1 Menetelmän kuvaus

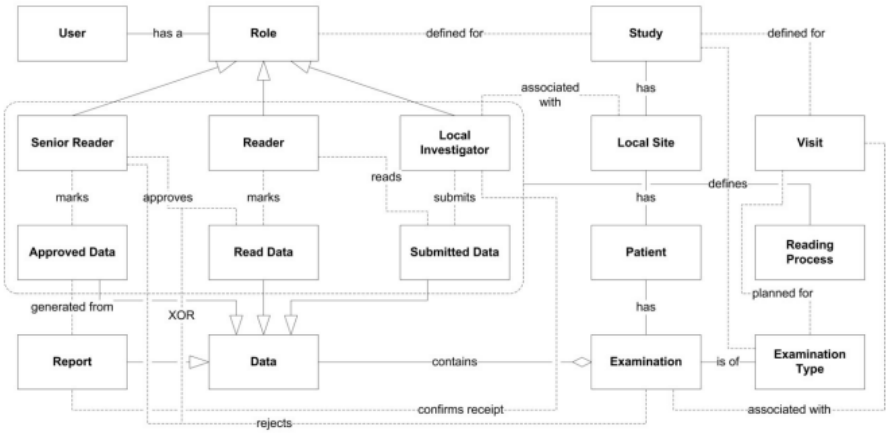
DDD:ssä keskeistä on toimialasta, esimerkiksi pankkitoiminnasta, koko työryhmän yhdessä muodostamat mallit. Näiden mallien rakennetta noudatetaan varsinaisessa ohjelmakoodissa. Työryhmään kuuluvat sekä suunnittelijat, toimialaosaajat että varsinaisen ohjelmakoodin kirjoittajat. Mallien tekijöiden tulisi 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 yllä pidettävään ohjelmakoodiin.

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).



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

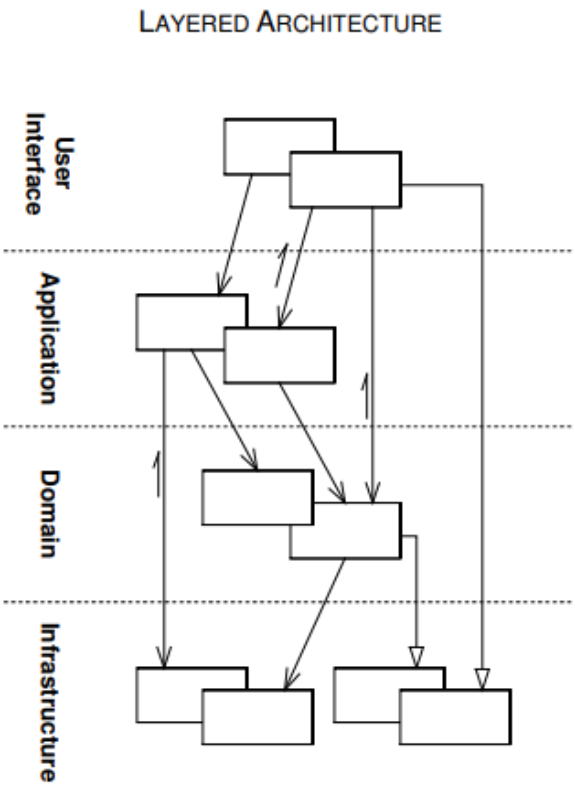
Mallit kehittyvät projektin edetessä. Mallinnus ja niiden implemointi kulkevat rinnakkain. Mallia tehdessä on mietittävä sen implementointia (Evans, 2003). Implementointi taas saattaa tuoda uusia ideoita mallinnukseen. Jos tehty ohjelma ei pääpiirtettäin noudata tehtyä mallia, on mallin arvo kyseenalainen. 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 bounded context (Evans, 2003; Özkan et al., 2023; Vural ja Koyuncy, 2021), rajattu konteksti. 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.

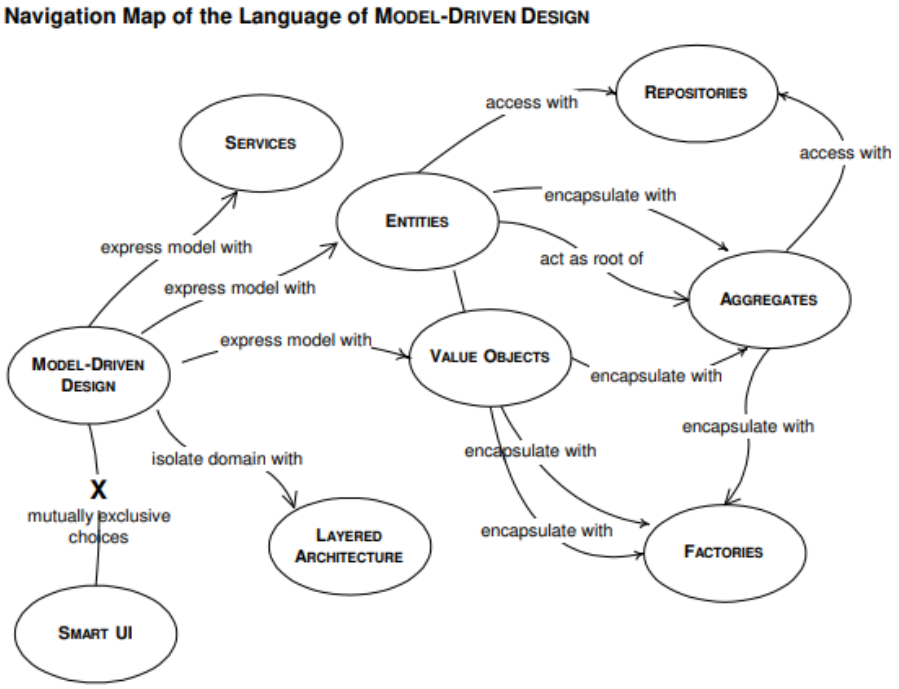
Toimialamallit säilövät tietoa kuvatessaan miten ohjelmakoodi toimii. Projektien pitkittyessä henkilökunta usein vaihtuu, kuten vaihtuvat myös ohjelmistojen käyttäjät ja ylläpitäjät. Tällöin mallit luonnollisesti tekevät uusille henkilöille ohjelmistojen päivityksestä ja ylläpidosta helpompaa (Evans, 2003).

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).



Kuva 2.2: Kerroksellinen arkkitehtuuri (Evans, 2003).

Kuvan 2.3 mukaan 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 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.



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 käytettään toimialan sanastoa ja rakenteita (Evans, 2003). Toimialakohtaisia kieliä käyttäen toimiala-asiantuntijat voivat helpommin osallistua ohjelmakoodin tuottamiseen (Hoffmann1 et al., 2022). Implementoinnissa DSL tavallisesti käännetään jollekkin yleiselle ohjelmointikielelle. DSL:n rakentaminen yleisen ohjelmointikielen päälle vaatii tältä kieleltä muokattavuutta(Evans,

2003).

2.2 Implementointeja käytännössä

Lots et al. 2010 ovat tehneet kliinisten kokeiden tuloksia analysoivalle analysointikeskukselle koemateriaalia ja sen analyysiä hallinnoivan sovelluksen käyttäen DDDn periaatteita. Näitä pidetään hyvin soveltuvina. Mallinnusta tarvittiin toimialan monimutkaisuuden vuoksi. Esimerkiksi lainsäädäntö ja koedatan luottamuksellisuus tuovat omat reunaehtonsa. Voi päätellä hyvin mallinnettujen koedatan kulun ja käyttöoikeuksien pienentävän todennäköisyyttä koedatan vaarantumisesta ohjelmistovirheen tai suunnitteluvirheen vuoksi.

Julkaisujen määrästä päätellen, DDD:n käytön mikropalveluarkkitehtuuriin perustuvien sovellusten kehtiyksessä on havaittu yleistyneen vuodesta 2017 alkaen (Özkan et al., 2023). Tällaiseen arkkitehtuuriin perustuvissa ohjelmistoissa omissa proseisseissaan ajetut keskenään kommunikoivat mikropalvelut muodostavat sovelluksen (Lewis ja Fowler, 2014). Vular ja Koyuncu 2021 totesivat DDD:n soveltuvan mikropalveluiden optimaalisen modulaarisuuden määrittämiseen. Wang et al. 2022 ovat käyttäneet DDD:a ja mikopalveluita kehysympäristön (framework) luomiseen lohkoketjuihin perustuville seurantajärjestelmille.

Hollannin Suomen maanmittauslaitosta vastaava Kadaster * on käyttänyt DDD:a suunnitellessaan uutta tietojärjestelmäänsä korvaamaan vanha Cobolilla kirjoitettu järjestelmä (Oukes et al., 2021). Samaan aikaan järjestelmän uusimisen kanssa päivitettiin myös toimintamallia. DDDn todettiin soveltuvan hyvin toimialan mallintamiseen ja sillä todettiin yhtäläisyyksiä ISO 19152 esittämään standardimalliin maanhallinnan toimialalaa.

2.3 Vaikutus vikasietoisuuteen

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

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

ohjelman toimintaa. Jos häiriötilanne johtuu virheestä toimintalogiikassa, voi ohjelman keskeytyminen olla vianetsinnän kannalta hvyödyllistä.

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).

3 Elixir

Elixir on kohtuullisen uusi funktionaalinen 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 esimerksi JavaScriptillä tai Pythonilla (StackOverflow, 2023).

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 iExympäristössä (interactive elixir)(Elixir, 2023).

Funktionaalisena ohjelmointikielenä Elixir-koodin toimintalogiikka poikkeaa esimerkiksi normaalista Python-koodista. Esimerkiksi Mukundin kahden artikkelin kokonaisuus esittää tiiviin johdatuksen funktionaaliseen ohjelmointiin(Mukund, 2007a; Mukund, 2007b). Elixir-tutoriaaleja [‡] ja asennustiedostot [§] useille käyttöjärjestelmille ovat saatavilla verkosta.

3.1 Elixir-prosessien vikasietoisuus

Elixirissä koodi ajetaan prosesseiksi kutsutuissa yksiköissä(Elixir, 2023; Ballou, 2015). Ne muistuttavat käyttöjärjestelmien threadejä. Yksi sovellus voi koostua lähes rajattomasta määrästä samanaikaisia prosesseja. Se tekee Elixir-ohjelmista helposti skaalautuvia. Pro-

[†] https://discord.com/

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

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

sessit 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 Supervisor-prosessien avulla. Nämä prosessit 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.

4 Elixir mallien implementointivälineenä

Kappale 2 toteaa toimialaperusteisen suunnittelun perustuvan toimialan mallintamiseen. Evansin (Evans, 2003) mukaan DDD ja mallien tuoteutus tietokoneella vaatii olio-ohjelmointiin perustuvan ohjelmointikielen. Toisaalta ideaaliksi implementointivälineeksi DDDlle Evans mainitsee Prologin, logiikkaperusteisen ohjelmointikielen. Funktionaalista ohjelmointia ei mainita. 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 DDD ei vaadi implementointivälineekseen olio-ohjelmointiin perustuvaa ohjelmointikielä.

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

DDD:n 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

5 Johtopäätökset

Tähän 1-2 sivua tekstiä mihin johtopäätöksii tultiin ja mitä täytysi tehdä seuraavaksi.

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.

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

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

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

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.

Hoffmann1, B., Urquhart2, N., Chalmers3, K. ja Guckert1, 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.

Kärkkäinen, H. ja Hujanen, M. (2023). *Artikkeli "Mikä oli Ville Tapion IT-ymmärrys? 5 avointa kysymystä Vastaamosta – oikeudessa kuultiin hämmästyttäviä väittämiä"' lainaa poliisin esitutkintapöytäkirjaa: "POLIISIN esitutkintapöytäkirjan mukaan Vastaamon tietoturvassa oli vakavia puutteita: Vastaamon IT-järjestelmiä suojaava palomuurilaite päästi läpi kaiken verkkoliikenteen, asiakastietokantapalvelin oli näkyvissä internetiin puolentoista vuoden ajan, salasanasuojaus oli kehno tai sitä ei ollut, potilastietoja ei anonymisoitu, etäyhteydet palvelimelle oli sallittu, muttei asianmukaisesti salattu."* URL: https://www.is.fi/digitoday/tietoturva/art-2000009428901.html.

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

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.

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.

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.

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.

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.

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.