Kanditutkielma

Tietojenkäsittelytieteen kandiohjelma

Toimialalähtöinen ohjelmistokehityksen ja Elixir-ohjelmointikielen käyttö vikasietoisen ohjelmakoodin tuotannossa

Rolf Wathén

30.10.2023

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	October 30, 2023	12 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

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).

Tietokoneohjelman suorittamisen keskeyttävät virheet tapahtuvat usein suoritetilanteissa joita ohjelmankirjoittaja ei ole osannut ennakoida. Ne voivat tapahtua myös tilanteissa joita ei välttämättä edes olisi voinut ennakoida, esimerkkinä voisi ajatella esimerkiksi muistipiirin pettämistä tai toisen ohjelman aiheuttamaa häiriötä. Tietokoneohjelmassa olevia virheitä voi pyrkiä vähentämään kirjoitusvaiheessa erilaisilla tekniikoilla. Jos ja kun virhe syytä tai toisesta tapahtuu, voidaan ohjelma kirjoittaa myös sellaiseksi, että sen suoritus ei keskeydy virheeseen, vaan suoritusta pystytään jatkamaan tavalla tai toisella. Tässä käsitellään toimialaperusteista suunnittelun ja Elixirin ohjelmointikielen yhteiskäyttöä ohjelmakoodin kirjoitukseen. Toimialaperusteisen suunnittelun päämääränä on tuottaa mahdollisimman vikavapaata ja helposti päivitettävissä olevaa ohjelmakoodia. Tämä ohjelmakoodi voidaan toki toteuttaa monella eri ohjelmointikielellä. Tässä työssä keskitytään Elixir-ohjelmointikieleen, jossa vikasietoisuus on ollut oleellisena kriteerinä ohjelmointikieltä ja sen edeltäjiä kehitettäessä.

2 Toimialaperusteinen ohjelmistokehitys

2.1 Tausta ja periaatteet

Evansin vuonna 2003 (Evans, 2003) julkaisemaan kirjaa Domain-Driven Design: Tackling Complexity in the Heart of Software pidetään kirjaalisuudessa toimialaperusteisen suunnittelun (jäljempänä DDD) peruskirjana, vaikka itse kirjakin sanoo olevansa yhteenveto parinkymmenen vuoden aikana kehittyneistä käytännöistä ja periaatteista.

Mallit kehittyvät kokoajan ja tavalla on siten analogioita ketterään ohjelmistokehitykseen. Mallinnus ja niiden implemointi siis kulkevat käsi kädessä. Mallia tehdessä on myös mietittävä miten se saadaan implementoitua. Implementointi taas saattaa tuoda uusia ideoita mallinnukseen.

Vaikka DDD ei aina takaakkaan varmaa menestystä ohejlmistokehityksessä, vaikeasti ylläpidettävät ohjelmistot syntyvät yleensä kun toimialaa tai käyttöympäristöä ei ole kunnolla ymmärretty ja mallinnettu.

Mallli on selkeästi organisoitu abstraktio toimialasta johon on tarkasti valittu sen tärkeimmät elementit (Evans, 2003). Malli toimialasta ei välttämättä ole diagrammi, se voi olla myös esimerkiksi luonnollisella kielellä selkeäsi kirjoitettu kuvaus tai ohjelmistokoodia.

Kolme käyttökohdetta määrittävät DDDssä käytetyn mallin. 1) malli ja sen implementointi täydentävät toisiaan. Ohjelmakoodin kehittyessä malli kehittyy ja päinvastoin. Näiden kahden suhde auttaa myös jatkokehityksessä ja ylläpidossa, koska ohjelmakoodin toimintaa voidaan tulkita ja ymmärtää mallin avulla. 2) Malli on myös selkäranka käytetylle yhteiselle kielelle jota käyttävät kaikki projektiin osallistujat, niin ohjelmoijat kuin toimiala-asiantuntijatkin. Yhteinen kieli kaikkien osallistujien kesken nousee esiin kirjallisuudessa useissa lähteissä. Osittain tämä johtuu siitä että Evans mainitsee sen tärkeänä. 3) Malli on jalostettua tietoa. Mallilla projektitiimi saa yhtenäisen käsitteyksen toimialasta ja projektin kohteesta ja sen avulla saavutetaan yhteisymmärrys tärkeimmistä ja kiinnostavimmista osista.

Oppiminen on tärkeässä roolissa sekä ohjelmoijalle että toimiala-asiantuntijalle. Kun oh-

jelmoija ymmärtää mitä on tekemässä ja miksi, on helmpompi keskittyä oleellisiin asioihin.

Toimialamallit auttavat säilömään tietoa. Esimerkiksi pitkälle kehittyneet mallit kuvaavat miten ohjelmakoodi toimii. Projektien pitkittyessä henkilökunta usein vaihtuu kuten vaihtuvat myös ohjelmistojen käyttäjät ja yllpitäjät. Tällöin mallit luonnollisesti tekevät uusilee henkilöille ohjelmistojen päivityksestä ja ylläpidosta helpompaa.

Evans käyttää termiä ubiquitous language kuvaamaan yhteistä kieltä ohjelmistokehittäjien ja toimiala-asiantuntijoiden välillä. Ubiquitous kääntyy "kaikkialla läsnäolevaksi". Tässä ei kuitenkaan ole tarvetta ryhtyä raamatulliseksi, vaan oleellista on että kaikki projektiin osallistuvat puhuvat samoista asioista samoilla nimillä. Suomeksi sanonta "nyt puhutaan samaa kieltä"tavoittanee idean hyvin. Yhteisen kielen pohjana taas on toimialamalli, sitä luotaessa ja edelleen kehitettäessä kaikkien tulee käyttää siinä esitettyä termistöä ja jatkuvasti varmistaa että kaikki ymmärtävät kaikki termit ja käsitteet samalla tavalla. Tässä yhteisessä kielessä käytettyä sanastoa tulee käyttää myös ohjelmistokoodissa luokkien, metodien, muuttujien jne. nimeämisessä.

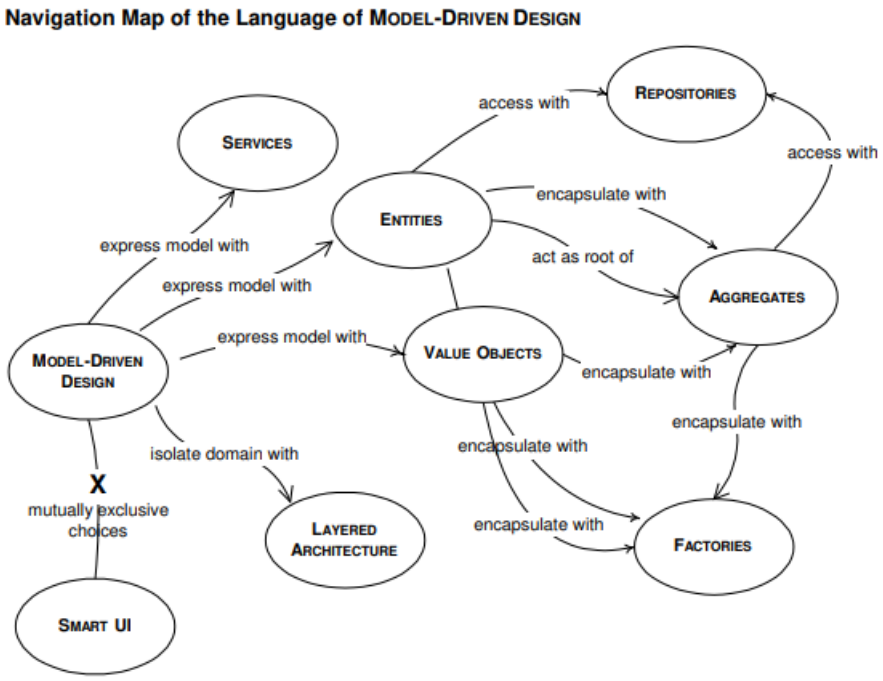
DDDsssä luodaan toimialalle siis malli. Mallissa käytetään usein diagrammeja ja kaavioita. Yleisesti käytettyjä diagrammeja ovat UML diagrammit (Booch et al., 1998) kuten luokkadiagrammit. Evans korostaa että malli ei ole sama kuin diagrammi, eikä diagrammien tulisi olla liian yksityiskohtaisia.

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. On oleellista että pääasiallisia malleja yhtä osiota kohti on yksi. Ohejelmakoodi tulee kirjoittaa alustavan mallin mukaan, ja mallia päivittää niin että se on koodin mukainen. Yksi malli vähentää virheiden mahdollisuutta, sillä ohjelman rakenne tulee nyt yhteisesti hyväksytystä mallista (Evans, 2003).

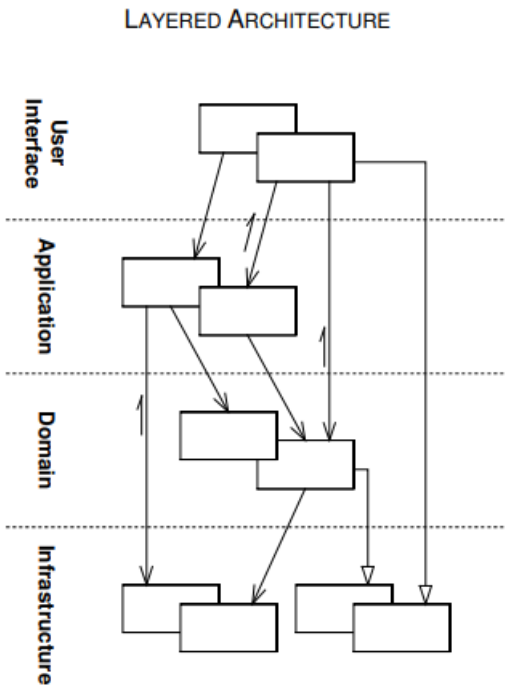
Mallien tekijöiden tulisi osallistua myös ohjelmointiin (Evans, 2003). Jos organisaatiorajat eri ihemisryhmien välillä ovat liian jyrkät, tietoa katoaa rajapinnoissa.

DDDn yhteyteen on kehitetty ohjelmistokehyksiä (framework) kuten Naked Objects joilla ydin kerrokseen tehdyt muutokset välittyvät automaattisesti muihin kerroksiin. Läufer demonstroi tämän käyttöä (Läufer, 2008).

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).



Kuva 2.1: Evansin (Evans, 2003) esittämä kartta DDD:ssä käytetyille käsitteille.



Kuva 2.2: Kerroksellinen arkkitehtuuri (Evans, 2003).

Tässä vielätullaan kuvaamaan kuvien 2.1 ja 2.2 sisältö sanallisesti.

2.2 Toimialaperusteinen ohjelmistokehitys ja vikasietoisuus

Kun malleilla on datan ja prosessin kulku ohjelmistossa mallinnettu, on ohjelmistossa helppo varautua siihen että vastaanotettu data on oikeanlaista ja riittävää. Tapauksissa jossa näin ei ole, voidaan ohjelmisto oikealla suunnittelulla pitää pystyssä. Näin voidaan toki tehdä ilman että projektilla on mitään tekemistä DDDn kanssa, mutta näin siihen on mahdollista paremmin varautua. Jos jokin osio mallista osoittautuu mahdottomaksi tehdä ojelmistoteknisesti mallia noudattaen, on mallia päivitettävä.

Pystytään varautumaan yllättäviin tilanteisiin.

Lots et al. (Lotz et al., 2010)ovat tehneet kliinisiä kokeita analysoivalle analysointikeskukselle koemateriaalia ja sen analyysiä hallinnoivat sovelluksen käyttäen DDDn periaatteita ja pitävät näitä hyvin soveltuvina. Tässä tapauksessa mallia tarvittiin pikemminkin toimialan monimutkaisuuden kuin varsinaisen ohjelmiston tekniseen tekoon liittyvien haasteiden vuoksi. Huomioon on otettava esimerkiksi lainsäädäntö ja koedatan ehdoton luottamuksellisuus. Voi siis hyvin päätellä että kun esimerkiksi koedatan kulku ja käyttöoikeudet ovat hyvin mallinnettuja, koedatan paljastuminen ja joutuminen vääriin käsiin ohjelmistovirheen tai suunnitteluvirheen vuoksi on vähemmän todennäköistä.

Tähän tulee vielä lisää tekstiä, etsitään pari viitettä.

3 Elixir

Elixir on kohtuullisen uusi funktionaalinen ja dynaaminen ohjelmointikieli (Ballou, 2015). Sen tuottaman tavukoodi pyörii BEAM-virtuaalikoneessa joka on alunperin kirjoitettu Erlang ohjelmointikieltä varten. Myös Erlang on funktionaalinen ohjelmointikieli jonka historia on Ericssonilla ja telekommunikkaatiossa ja se on toiminut pohjana Elixirin kehityksessä. Elixir onkin Erlang yhteensopiva ja käyttää tämän kirjastoja. Verkkosivustonsa [†] mukaan Erlangilla voidaa ajaa nopeita, vikasietoisia ja hajautettuja ohjelmistoja. Esimerkiksi Whatapp on ilmoittanut käyttävänsä Erlangia (Whatsapp, 2012).

Funktionaalisessa ohjelmointikielessä (ja ohjelmoinnissa) ohjelma koostuu luonnollisesti funktioista. Nämä voivat ottaa muita funktioita argumentteina ja funktio voi myös palauttaa funktioita. Lähtökohtaisesti silmukat luodaan rekursiolla. Muuttujat taas ovat muuttumattomia (immutable).

Elixirissä kaikki koodi ajetaan prosesseiksi kutsutuissa yksiköissä joita voi olla käynnissä samanaikaisesti lähes määrätön määrä. Prosessi muistuttavat käyttöjärjestelmien threadejä, mutta ovat huomattavasti kevyempiä. Prosessit pystyvät helposti kommunikoimaan keskenään viesteillä, ominaisuus jota tarkasttellaan hieman lähemimmin seuraavassa kappaleessa. Vikasietoisuuden kannalta oleellisena Elixirin ominaisuutena on, että yhden prosessin kaatumisen ei tarvitse johtaa koko ohjelmiston kaatumiseen. Oikein kirjoitettuna ohjelma pystyy käynnistämään kaatuneen prosessin helposti uudelleen. Tämä tekee Elixir ohjelmasta oivan alustan monille toimialoille. Julkisista lähteistä löytyvien tietojen mukaan Elixiriä käyttävät systeemeissään mm. monille tutut Discord, Pepsico ja Pintarest (Valim, 2023)

3.1 Johdatus Elixiriin

Elixirin asennusohjeet ja tiedostot löytyvät helposti internetistä [‡]. Se on saatavilla tätä kirjoitettaessa ainakin macOSlle, Linuxille, Windowsille ja BSDlle joiden lisäksi sitä on mahdollista ajaa Dockerissa ja Rasberry Pilla.Elixiriä asennettaessa asennetaan myös Erlang ja ns. OTP-kirjasto johon huomattava määrä toiminnallisuutta perustuu. OTP tulee

[†] https://www.erlang.org/

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

sanoista Open Telecom Platform, jäänne historiasta.

Elixirin interaktiivisen ohjelmointiympäristön IEx:n saa Windows 11 Powershellissä käynnistettyä komennolla `iex.bat`. Joissain muissa ympäristöissä toimii pelkkä `iex`, mutta Powershellissä `iex` on alias `Invoke-expression`ille komentojen ajamiseen. `iex` ympäristö vastaa esimerkiksi Python konsolia joka yleensä käynnistyy `python3` komennolla.

Elixir ohjelmat voidaan luonnollisesti kirjoittaa tekstieditorilla. Tiedostotunnus on `.exs` tai `.ex`, mutta käytetään tässä esimerkkinä iex-ympäristöä. Lähdetään liikkeelle kuten ohjelmointikielissä aina ”Hello world!” tyyppisestä esimerkistä. Oletetaan että lähdetään käynnistämään järjestelmää.

```
iex(5)> kaynnistysilmoitus="Käynnistetään järjestelmää."
"Käynnistetään järjestelmää."
iex(6)> onnistuiko_ilmoitus=IO.puts(kaynnistysilmoitus)
Käynnistetään järjestelmää.
:ok
iex(7)> onnistuiko_ilmoitus
:ok
iex(8)>
```

Eli muuttujalle käynnistysilmoitus, joka on siis nimetty DDD-filosofian mukaan, annetaan arvoksi = operaattorilla "Käynnistetään järjestelmää.". Interaktiivisessa ympäristössä muuttujan arvo tulostuu heti, mutta `IO.puts` funktiolla se saataisiin tulostumaan myös suoritettavassa ohjelmassa. Ja koska funktionaalisessa ohjelmoinnissa ohjelma palauttaa arvon, `IO.puts` palauttaa arvon `:ok` jota on tietotyyppiä atomi, vakio (englanniksi atom, monissa kielissä symbol). Ilmoituksen tulostus on `IO.puts` funktion sivuvaikutus. Sen palauttaa arvon `:ok` jonka perusteella varsinainen ohjelma jatkaisi suoritustaan.

Operaattoria = on syytä käsitellä hieman enemmän sillä se on Elixirissä match-operaattori jota käytetään Elixirissä ja toisessa funktionaalisessa ohjelmointikielessä merkittävässä roolissa olevassa hahmontunnistuksessa (eng. pattern matching). Näytetää esimerkki

```
iex(12)> case [:laite1,:ok,25] do
...(12)> [:laite1,:ok,x] when x>20 -> "Kova nopeus"
...(12)> [:laite1,:ok,y] -> "Kaikki ok"
...(12)> end
warning: variable "y" is unused (if the variable is not meant to be used,
```

```
prefix it with an underscore)
  iex:14

"Kova nopeus"
```

jossa avainsanalla case analysoidaan listaa [:laite1,:ok,25] ja käytetään hahmotunnistusta ja vahtia when. Nyt analysoitava lista vastaa listaa [:laite1,:ok,x], muuttujalle x tulee arvo 25 jolle vahti when vaatii x>25 toteutumisen. Tämä toteutuu joten palautuksena tulee string "Kova nopeus". Seuraavaan tunnistukseen ei koskaan päästä josta iex varoittaa.

3.2 Elixir ja vikasietoisuus

Funktionaalisena ohjelmointikielenä Elixiriä on sekä ihmisten että tietokoneen kääntäjien suhteellisesti helpompi ymmärtää kuin imperatiivisia ohjelmointikieliä (Ballou, 2015) Varsinkin ohjelmaa päivitettäessä tätä voi pitää tärkeänä tekijänä. Elixiriin on kuitenkin rakennettu sisäisesti ominaisuus joka tekee ohjelmakoodista todella vikasietoisen. Tämä perustuu ohjelmakoodin suoritukseen hajautetusti prosesseissa joiden tilaa voivat valvoa toiset ns. supervisor-prosessit. Häiriötilassa supervisor-prosessi voi päättää valvomansa prosessin ja käynnistää sen välittömästi uudelleen.

4 Elixir ja toimialaperusteinen suunnittelu

Evans (Evans, 2003) antaa lähes paradigman asemassa olevassa kirjassaan selkeästi ymmärtää, että DDD ja mallien tuoteutus tietokoneella yleensä lähes vaatii ohjelmointikielen joka perustuu olio-ohjelmointiin, esimerkkeinä Java, C++ tai Python. Toisaalta ideaaliksi kumppaniksi DDDlle hän mainitsee Prologin, logiikkaperusteisen ohjelmointikielen. Funktionaalista ohjelmointia ei suoraan konseptina mainita, mutta teksti antaa ymmärtää että ne eivät ole tarkoitukseen ideaalisia. Evansilta löytyi perushaulla hyvin vähän julkaisuja kirjansa jälkeen, joten hänen nykyinen kantansa ei ole tiedossa. Vastakkaisia esimerkkejä löytyy, kuten Wlaschinin Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F# (Wlaschin, 2018). F# [†], kuten Elixirkin on funktionaalinen ohjelmointikieli. Teoksessa Designing Elixir Systems with OTP Write Highly Scalable Self-Healing Software with Layers (Gray ja Tate, 2019) käytetään myös selkeästi DDD-ideoita vaikka sitä ei suoraan mitenkään mainita. Esimerkiksi kehoitetaan nimeämään muuttujat ja funktiot kuvaavasti, käytetään kerroksellista (layers) ohjelmarakennetta ja tehdään päätoiminnallisuus (domain) omaan kerrokseensa.

Elixirin dokumenteissa mainitaan mahdollisuus laajentaa kieltä toimialakohtaiseksi kieleksi, englanniksi domain-specific language, DSL. Tämä on asia jossa tarkkaivaisuus on paikallaan. Englannin kielinen sana *domain* on tässä työssä käännetty toimialaksi. DSLn yhteydessä se voi kääntyä myös sovelluskohteeksi. Tässä työssä toimialalla on tarkoitettu esimerkiksi vakuutusalaa, pankkitoimintaa, telekommunikaatiota tai verkkokauppaa. DSLn yhteydessä voidaan siis myös kääntää se sovelluskohdekohtaiseksi kieleksi. Elixiriin on tehty laajennuksia esimerkiksi tietokantojen [‡] ja koneoppimiseen [§], joita voidaan pitää DSLinä. Sellaisia toimialakohtaisia kieliä jotka vastaisivat sanan *domain* tässä työssä käytettyä käännöstä ja jotka toimivat Elixirin päällä ei löydetty.

Vural ja Koyuncu (Vural ja Koyuncy, 2021) ovat tutkineet DDDn käyttöä pilven mikropalveluiden oikeaan määrittelyyn ja todenneet sen hyvin soveltuvaksi. He maninitsevat myös että yhden mikropalvelun kaatuiminen ei saa johtaa koko systeeminin kaatumiseen.

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

Edellä tutkitun tieton perusteellaa voinee päätellä että Elixir soveltuu ainakin tältä osin käytetyksi ohjelmointikieleksi.

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.

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.

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.

Valim, J. (2023). URL: https://elixir-lang.org/cases.html.

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.