Malli- ja toimialalähtöiset ohjelmistokehitymetodit ja -kielet

Rolf Wathén

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. 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 työssä käsitellään ns. toimialaperusteista suunnittelua ja mallinnusta ohjelmakoodin kirjoitukseen. Näiden päämääränä on tuottaa mahdollisimman vikavapaata ohjelmakoodia. Näihin läheisesti liittyyen käsitellään toimialaspesifejä ohjelmointikieliä käyttäen esimerkkinä Elixir-ohjelmointikieltä, jossa vikasietoisuus on ollut oleellisena kriteerinä ohjelmointikieltä ja sen edeltäjiä kehitettäessä.

Toimiala- ja malliperusteiset ohjelmistokehityskonseptit

Domain-driven design

Domain sanalla on useita suomennoksia jotka tarkottavat eri asioita. Tässä yhteydessä paras suomenkielinen käsite lienee toimiala. Domain driven design voisi ehkä parhaiten suomentaa toimialalähtöinen suunnittelu. Siinä keskeistä on jatkuva kommunikaatio kyseisen toimialan toimijoiden ja erityisesti asiantuntijoiden kanssa. Ohjelmoijien on parhaansa mukaan opittava ymmärtämään toimialalla käytettyjä käsitteitä ja toimintamalleja ja käyttää näitä käsitteitä ja malleja omassa koodissaan. Näin tekemällä toimiala-asiantuntijoiden on helpompi seurata työn etenemistä ja antaa hyödyllisiä rakentavia mielipiteitä koodin kehitykseen [Wikipedia]. DDDssä keskiössä on alussa määrittää mitä tehdään ja miksi, ei miten tehdään. DDDn alkuna pidetään Eric Evansin kirjaa Domain-Driven Design: Tackling Complexity in the Herat of Software [Evans 2003]. Khononov on tehnyt äskettäin helppotajuiselta vaikuttavan kirjan aiheeseen nimeltä Learning Domain-Driven Design Aligning Software Architecture and Business Strategy [Khononov 2022]. Evansin kirjassa mainitaan myös verkkosivu www.domaindrivendesign.org, mutta sen sisältö näyttäisi nykyään olevan sekavaa harhautuen tämän työn aihepiiristä.

Konsepti joka ei tarvitse omaa kappalettaan tässä työssä mutta ansaitsee tulla mainituksi DDDn yhteydessä on domain engineering, joka on prosessi jossa käytetään aiemmin kertynyttä toimialaosaamista ja ohjelmakoodia uusissa projekteissa [Wikipedia].

Domain-specific modeling

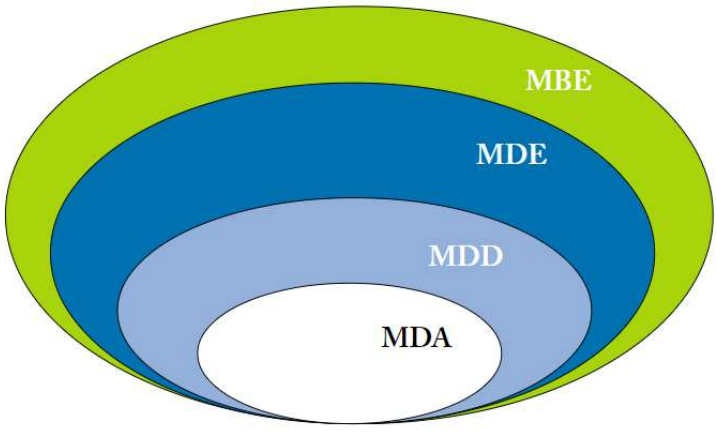
DMS on ohjelmistotuotantometodologia systeemien suunnitteluun ja kehitykseen. Siinä toimialaspesifisten kielten käyttö on oleellisessa osassa. Tässä lähestymistavassa toimialaspesifinen ohjelmointikieli luodaan tapauskohtaisesti oman organisaation tai tietyn projektin käyttöön sen sijaan että se hankittaisiin joltain toimittajalta joka tekee niitä tämän toimialan käyttöön.

Model-driven architecture (MDA)

MDA on lähestymistapa ohjelmistokehitykseen määritellen suuntaviivat ja rakenteen sille millaisia ohjelmistokehityksessä olevien mallien tulisi olla. MDA on laajempi ja yleisempi konsepti kuin käsitelty MDE. Se on itseasiassa Object Management Groupin omistama tavaramerkki. OMGn verkkosivuilta löytyy (9/2023) useitakin artikkeleita aiheesta, viimeisin MDAn yleisesti kuvaava on Object Management Group Model Driven Architecture (MDA) MDA Guide rev. 2.0 [https://www.omg.org/cgi-bin/doc?ormsc/14-06-01]. Näyttää että tämä on viimeisimpiä julkaisuja OMGn puolesta ja siellä aihepiiriä ei ole enää juurikaan kehitetty.

Model-driven engineering (MDE)

MDE on ohjelmistokehitysmetodi jossa ytimessä on toimialakohtaisten mallien luominen ja käyttö. Siinä pyritään esittämään ratkaistava tehtävä (ja sen ratkaisu) ja siinä tarvittava tietovirta malleina jotka ovat kuvaavat käsiteltävän toimialan reaalimaailmaan. Tämän vastakohtaina olisi keskittyä esittämään ja korostamaan algoritmejä joilla tehtävä varsinaisesti ratkaistaan. Luotujen mallien perusteella voidaan jopa ohjelmakoodi, dokumentaatio ja testit luoda algortimien avulla automaattisesti. Jos aloritmit tähän on luotu huolella ja testattu, tämä on omiaan vähentämään suoritettavassa koodissa olevia virheitä sekä nopeuttamaan ohjemistokehitystä koska luotuja malleja voidaan käyttää useissa projekteissa. (Wikipedia)



Kuva 1. Erilaisten malli(model)-perusteisten ajatus- ja kehitysmallien suhde [Brambilla et al. 2017].

Termien ja käsitteiden viidakko erilaisissa malliperusteisissa raameissa ja metodeissa on tiheä. Brambilla et al. Esittävät kuvan 1 selventämään käsitteitä malliperusteisten lyhenteiden ja käsitteiden osalta. Kuvan MDA vastaa edellä kuvattua, kuten myös MDE. MDD, model-driven development on lähellä MDEtä, mutta on työkaluiltaan ja laajuudeltaan suppeampi. MBEssä, model-based engineering, itse mallien kehitys ei välttämättä ole ohjelmistokehitystä ajava voima vaikka toiminta sinällään perustuu malleihin. Esimerkkinä tästä voidaan pitää Matlab ohjelmiston

Simulink-lisäosaa (https://se.mathworks.com/products/simulink.html). Valmistaja Mathworks tosin käyttää tästä termiä Model-Based Design, mutta elementit ovat samat.

Toimialaspesifiset kielet (domain-specific language, dsl)

Wikipedian mukaan toimialaspesifinen kieli on suunnattu tietylle toimialalla. Toimialalla voidaan tarkoittaa esimerkiksi vakuutusalaa, pankkitoimintaa tai telekommunikaatiota. Toisaalta englannin sana domain tässä yhteydessä käsitetään myös sovellusalana, esimerkiksi HTMLää pidetään dsl:nä, sillä tehdään verkkosivuja tai SQL:ää, jolla käsitellään tietokantoja. Tässä työssä käsite toimiala (tai domain) käsitetään ensin mainitussa merkityksessä, esimerkiksi siis pankkitoimintana.

Tässä yhteydessä on syytä mainita myös UML, unified modeling language. Sen tarkoituksena on olla, tai ainakin on ollut olla, standardisoitu kieli visualisoida ohjelmiston rakenne. Aiemmin mainittu OMG on ajanut UMLn käyttöä mutta kuten MDAnkin kanssa, menestys on ollut vaatimatonta ja näyttäisi että yleisestikkään UML sinällään ei ole valtavirtaa. Sen sijaan diagrammit joita joilla UML esitää prosessin tai ohjelman osien vuorovaikutuksia ovat yleisesti käytössä. Näitä ovat esimerkiksi rakkennediagrammi, käyttäjädiagrammi ja sekvenssidiagrammi [Fehre&Ketels 2015].

Toimialaspesifisen ja yleisen ohjelmointikielen välinen raja on usein veteen piirretty viiva. Vaikka ohjelmistoprojekti suoritettaisiin yleisellä ohjelmointikielellä, esimerkiksi Pythonilla, on silti mahdollista käyttää ideoita toimialakohtaisia ohjelmistokehitysmetodeita. Hyvin yksinkertainen tapa tähän on nimetä muuttujat ja oliot kuvaavasti sillä sanastolla jota käytetään kohteena olevalla toimialalla. Esimerkiksi pankkitilin saldon nimeäminen pankkitilin_saldo on projektiin osallistuvalle pankkihenkilölle huomattavasti kuvaavampaa kuin x. Tähän kuriositettina on jopa esitetty että muuttujan nimien tulisi olla niin pitkiä että ne varmasti kuvaavat mistä on kysymys, toisin sanoen ei olisi liian pitkää muuttujan nimeä [Gray&Tate 2019]. Myös ohjelman rakenteessa tiedon kulku käsittelijltä toiselle tulisi kulkea kuten se kulkee reaalimaailmassakin.

Elixir

Elixir on kohtuullisen uusi funktionaalinen ohjelmointikieli. Sen tuottaman tavukoodi pyörii Erlang virtuaalikoneessa, joka verkkosivun mukaan on tunnettu siitä että sillä voidaan ajaa nopeita, vikasietoisia ja hajautettuja ohjelmistoja. Myös Erlang on funktionaalinen ohjelmointikieli jonka historia on Ericssonilla ja telekommunikkaatiossa. Yleisen perehtymisen perusteella on syntynyt tunne, että Elixirin käyttö on kasvavaa ja yhteisö on hyvin aktiivinen ja kehittyvä.

Elixir on siis funktionaalinen ohjelmointikieli. Kuten käsitekin antaa ymmärtää, funktionaalisessa ohjelmointikielessä (ja ohjelmoinnissa) ohjelma koostuu lähinnä 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 pystyy olemaan käynnissä samanaikaisesti lähes määrätön määrä. Nämä prosessit ovat kuin ultrakevyt versio käyttöjärjestelmien threadeistä ja ne pystyvät kommunikoimaan keskenään viesteillä. Kriittisenä ominaisuutena kielessä on se etät yhden prosessin kaatumisen ei tarvitse johtaa koko ohjelmiston

kaatumiseen, vaan ohjelma pystyy käynnistämään kaatuneen prosessin helposti uudelleen. Tämä tekee Elixir ohjelmasta oivan alustan toimialoille kuten telecom, johon sen juuret johtavatkin, tai web-kauppa, mm. Jossa se nykyään on käytössä. Yleisesti Elixirin sisäänrakennettu kyky ajaa hajautettua ohjelmakoodia ja toipua helposti virhetiloista tekee siitä käytännöllisen pohjan moneen kohteeseen jossa toimialakohtaiset ohjelmistokehitysmetodit ja kielet ovat käyttökelpoisia.

Johtopäätökset

Tässä esseessä käsiteltiin pintapuolisesti ohjelmistokehitysmetodeja joissa lähtökohdaksi otetaan toimiala johon haluttu ohjelma tehdään tai jonka piirissä ongelma halutaan ratkaista. Lisäksi tehtiin katsaus toimialaspesifisiin kieliin. Yhteistä näille kaikille ovat konseksuksen etsiminen käytetystä kielestä ja ohjelman toiminnan kuvaus malleilla jotka muistuttavat mahdollisimman paljon sovellustkohdetta reaalimaailmassa. Edellisen lauseen kielellä viitataan sanastoon jota ohjelmoijat ja toimialaosaajat käyttävät keskenään. Kun nämä kaksi ryhmää kommunikoivat samalla kielellä, toimialaosaajat pystyvät paremmin edistämään ohjelmistokehittäjien työtä, jotka taas paremmin ymmärtävät mitä ovat tekemässä. Malleilla taas viitataan siihen mitä käsiteltävän systeemin eri osat tekevät, miten ne on nimetty ja mikä on niiden suhde toisiinsa. Näiden mallien kuvaukseen on olemassa omia kieliään kuten UML, mutta niihin ei spesifisesti paneuduttu.

Toimialaspesifeillä ohjelmointikielillä pystytään tietylle alalle luomaan tehokkaasti ymmärrettävää ja helpommin debugattavaa ohjelmakoodia. Näissä on ymmärrettävä ja eroteltava mitä tarkoitetaan toimialalla, englanniksi domain. Esimerkiksi HTMLllä toimiala on sinällään ovat verkkosivut, mutta verkkosivut ja niiden toiminnallisuus on usein tehty jollekkin liiketoiminnalle, ts. toimialalle. Tässä työssä ajateltiin toimialaspesifisen kielen tarkoittavan pikemminkin ohjelmointikieltä jolla on tarkoitus luoda sovellus jollekkin tietylle toimialalle.

Tehdyn katsauksen perusteella selkeintä ja kiinnostavinta on syventyminen domain-driven design konseptiin ja sen implementointiin Elixir ohjelmointikielellä pitäen fokuksena mahdollisimman vikasietoisen lopullisen ohjelmakoodin tuottoa.

Viitteet:

Wikipedia hakusanalla 9/2023

Gray, James Edward, Tate, Bruce A., Designing Elixir Systems with OTP Write Highly Scalable, Self-Healing Softwware with Layers, Pragmatic Programmers, 2019, 222p.

www.omg.com, 9/2023

elixir-lang.org 9/2023

erlang.org 9/2023

Fehre, Philipp, Ketels, Toon, JavaScript Domain-Driven Design : Speed up Your Application Development by Leveraging the Patterns of Domain-Driven Design, Packt Publishing, 2015, 181p.

Brambilla, Marco, Cabot, Jordi, Wimmer, Manuel, Model-Driven Software Engineering in Practice, Second Edition, Springer Nature Switzerland, 2022.

Khononov, Vlad, Learning Domain-Driven Design, O’Reilly, 2022.

Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003.