Tampereen yliopisto

Miko Mäkinen

VÄHÄISEN KOODIN KEHITYSYMPÄRISTÖT

Tieto- ja sähkötekniikan tiedekunta
Kandidaatintyö
6/2021

TIIVISTELMÄ

Miko Mäkinen: Vähäisen koodin kehitysympäristöt
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
6/2021

Vähäisen koodin kehitysalustat (englanniksi Low-Code Development Platforms) ja näihin liittyvät kehitysympäristöt pyrkivät tehostamaan ohjelmistokehitystä vähentämällä tarvetta kirjoittaa ohjelmakoodia. Ohjelmakoodin kirjoittamisen sijasta vähäisen koodin kehitysympäristöissä kehitys tapahtuu pääsääntöisesti sijoittelemalla valmiita elementtejä graafisen käyttöliittymän kautta. Tämän työn tarkoitus on vertailla perinteisiä ja vähäisen koodin kehitysympäristöjä toisiinsa ja pohtia, onko jälkimmäisille tarvetta. Tämä vertailu tehtiin dokumentaation ja muun kirjallisuuden pohjalta.

Työ jakautuu viiteen osaan ja näistä ensimmäisessä käsitellään vähäisen koodin kehityksen historiaa. Tässä osiossa käydään läpi tekniikoita, jotka mahdollisesti johtivat vähäisen koodin kehityksen syntymiseen. Seuraavassa osiossa käydään läpi kehitysympäristötyyppien eroja: mitä ja miten niillä kehitetään sekä listataan esimerkkejä kehitysympäristötyyppien edustajista. Tästä jatketaan pohtimalla kehitysympäristötyyppien erojen vaikutusta kehitettyjen sovellusten tekijänoikeuteen sekä vertailemalla lisenssien hintoja. Hintojen vertailusta siirrytään tutkimaan kustannustehokkuutta, jonka jälkeen lopuksi määritellään joitakin ketteryyden osa-alueita ja vertaillaan, miten hyvin eri kehitysympäristötyypit suoriutuvat näillä.

Työn keskeisimmät havainnot ovat ensimmäiseksi se, miten vähäisen koodin kehitys on vain uusin iteraatio pitkään jatkuneissa ponnisteluissa helpottaa ja tehostaa ohjelmistotuotantoa. Siirryttäessä kehitysympäristötyyppien erojen vertailuun huomattiin, että mitä isompia osia ohjelmakoodia kehitysympäristö luo automaattisesti ja mitä enemmän kehitys painottuu näihin valmiisiin elementteihin, sitä kapea-alaisemmaksi kehitys kyseisellä kehitysympäristöllä muuttuu. Seuraavaksi tutkittaessa lisenssejä ja pohdittaessa tekijänoikeudellisia ulottuvuuksia havaittiin perinteisten kehitysympäristöjen olevan huomattavasti edullisempia käyttää. Toisaalta kustannustehokkuutta tarkastellessa kävi ilmi, että vähäisen koodin kehitys on mahdollisesti kuitenkin edullisempaa, kun huomioidaan sovelluskehityksen parantunut tehokkuus sekä vähentynyt tarve ohjelmistotuotantoon erikoistuneille työntekijöille. Lopuksi erilaisia ketteryyden osa-alueita tarkasteltaessa selvisi, että perinteisillä kehitysympäristöillä yksittäinen kehitysympäristö taipuu useampaan eri tarkoitukseen ja on tarvittaessa helpompi korvata toisella kehitysympäristöllä, kun taas vähäisen koodin kehitysympäristöt tuottavat uusia sovelluksia nopeammin.

Avainsanat: Ohjelmistokehitys, kehitysympäristö, kehitysalusta, Low-Code, vähäinen

SISÄLLYSLUETTELO

KÄÄNNÖKSET

Suomennos	Alkuperäinen
Vähäisen koodin kehitysympäristö	Low-code development platform
Loppukäyttäjän ohjelmointi	End-user programming
Luonnollinen ohjelmointikieli	Natural-language programming
Visuaalinen ohjelmointi	Visual programming
Taulukkolaskenta	Spreadsheets
Lisäyslauseke	Include statement
Toimialuekohtainen ohjelmointikieli	Domain specific language
Asetusfunktio	Setter (function)
Palautusfunktio	Getter (function)
Kehittäjä	Developer
Kansalaiskehittäjä	Citizen Developer
Teknologia hallinto	Technology Management
Sovelluskauppa	App Store
Järjestelmäriippumaton	Cross-platform
Vähennetty informaatioteknologiavaiva	Reduced IT-effort
kolmansien osapuolten sovellukset	Third-party applications
yksinkertaistetut liiketoimintamenettelyt	Streamlined business processes
ketterä kehitys	Agile programming
takaisinmallinnustyökalu	Reverse engineering tool
Ohjelmistokehityspaketti	Software development kit
Ohjelmointirajapinta	Application Programming Interface
liitännäinen	Plug-in
taustajärjestelmäkehittäminen	Back-end development

1. JOHDANTO

Vähäisen koodin kehitysalustat (Low-Code Development Platforms) ovat suunniteltu siten, että kehittäjän tarvitsisi kirjoittaa mahdollisimman vähän varsinaista koodia. Sen sijaan tuote syntyy kehitysalustaan liittyvän kehitysympäristön graafisen käyttöliittymän kautta sijoittamalla valmiita elementtejä haluttuihin paikkoihin. Kehitysympäristö muodostaa koodin automaattisesti, eikä kehittäjän tarvitse kirjoittaa koodia kuin harvinaisemmissa tapauksissa, joita kehitysympäristön tekijä ei ole osannut nähdä ennalta ja johon kehitysympäristö ei siis taivu.

Kuten edellä on mainittu, englanninkielinen termi Low-Code Development Platforms kääntyy suomeksi vähäisen koodin kehitysalustoina. Tässä työssä kuitenkin käsitellään aihetta pääsääntöisesti vertailemalla perinteisiä ja vähäisen koodin kehitysympäristöjä toisiinsa, joten tässä työssä viitataan vähäisen koodin kehitykseen enimmäkseen kehitysympäristötasolla. Tämä ero on lähinnä semanttinen, sillä vähäisen koodin kehitysalustoille on ominaista, että niihin liittyy vain alustan tarjoajan tarjoama kehitysympäristö, eikä kehitysalustoilla ole kehitysympäristöjen välistä kilpailua.

Perinteisillä kehitysympäristöillä tarkoitetaan tässä työssä kehitysympäristöjä, joissa kehittäjä työskentelee tekstipohjaisesti pääsääntöisesti tuottaen koodin itse. Myös perinteisissä kehitysympäristöissä voi olla kehittäjää avustavaa automaatiota, kuten esimerkiksi ennustava tekstinsyöttö, mutta pääpaino on kehittäjän omassa koodin tuottamisessa.

Tämän työn tarkoitus on vertailla perinteisiä ja vähäisen koodin kehitysympäristöjä toisiinsa tietyillä osa-alueilla dokumentaation sekä muun kirjallisuuden pohjalta ja tätä kautta pohtia tarvetta vähäisen koodin kehitysympäristöille. Aihe on melko uusi: vähäisen koodin kehitysympäristöt ovat ottaneet vasta viime vuosina tuulta purjeisiinsa ja niistä on julkaistu vain verrattain vähän tieteellisiä artikkeleita, vaikka joitakin toki löytyy.

Tässä työssä vertailu on jaettu siten, että tarkasteltavat osa-alueet muodostavat omat lukunsa ja tarpeen mukaan osa-alueen alla on alaluvut perinteisille ympäristöille ja vähäisen koodin kehitysympäristöille tai osa-alue on jaettu alalukuihin muulla tavoin. Lukujen ja alalukujen sisältö jakautuu siten, että luvun tai alaluvun käsitellessä perinteisiä ja vähäisen koodin kehitysympäristöjä erikseen perinteisten kehitysympäristöjen tarkastelu tulee ensin.

Aluksi luvussa kaksi tarkastellaan vähäisen koodin kehitysympäristöjen historiaa ja taustaa. Tästä jatketaan lukuun kolme, jossa pohditaan ympäristötyyppien eroja yleisemmällä tasolla. Luvussa 4 vertaillaan kehitysympäristötyyppien eroja lisensoinnin kannalta ja luvussa 5 vertaillaan kehitysympäristötyyppien tehokkuutta. Luku 6 puolestaan keskittyy ketteryyteen, luvussa 7 tehdään johtopäätökset ja lopuksi luvussa 8 tehdään yhteenveto.

2. VÄHÄISEN KOODIN KEHITYKSEN HISTORIA

Tässä luvussa käydään läpi vähäisen koodin kehitysympäristöjen historiaa ja taustaa. Tässä yhteydessä vähäisen koodin kehitysympäristöjen vertaileminen perinteisiin kehitysympäristöihin ei ole kovin mielekästä, sillä kehitysympäristöt eivät tietenkään kilpaile historioillaan, vaan pikemminkin kehitysympäristöt ovat osa ohjelmistotuotannon yhteistä historiaa. Ne siis jakavat osan yhteistä historiaa, josta sitten haarautuvat omiksi osikseen ohjelmistotuotannon historiassa. Tämän vuoksi tätä lukua ei jaeta vertailuun perinteisten ja vähäisen koodin kehitysympäristöjen välille, vaan vähäisen koodin kehitysympäristöjen historia käydään läpi osana ohjelmistotuotannon historian kokonaisuutta.

Edellä mainitusta historian kokonaisuudesta voikin olla haastavaa erottaa, mikä kuuluu juurikin osaksi vähäisen koodin kehitysympäristöjen historiaa ja missä raja historian eri osa-alueilla menee. Itse termi on kuitenkin verrattain helppo ajoittaa: SDTimesin editori Rob Marvin haastatteli Forresterin analyytikkoa Clay Richardsonia, joka on kirjoittanut useita papereita vähäisen koodin kehitysalustoista ja kehitellyt termin. Tässä haastattelussa Richardson kertoo termin juontavan juurensa vuoteen 2011, mutta termi on otettu varsinaisesti käyttöön vasta vuonna 2013 tehdyssä tutkimuksessa. [1] Tämän jälkeen myös muut tutkijat ovat ottaneet termin käyttöön ja termi on lähtenyt leviämään [2][3][4][5].

Konseptitasolla vähäisen koodin kehitysympäristöt ovat kuitenkin olleet olemassa huomattavasti termiä pidempään. Forbesin journalisti Joaquim Madrinha väittääkin 2018 vuonna julkaistussa artikkelissaan vuonna 2001 perustettua OutSystemsiä vanhimmaksi vähäisen koodin kehitysympäristöksi [6], jolloin vuotta 2001 voitaisiin pitää vähäisen koodin kehitysympäristöjen syntymävuotena. Toisaalta esimerkiksi Dell Boomi on peräisin vuodelta 2000 [7] ja tätäkin ainakin nykyisin kutsutaan vähäisen koodin kehitysympäristöksi [8]. Lisäksi on täysin mahdollista, että on olemassa sellainen ennen vuotta 2000 kehitetty ohjelmistokehitysratkaisu, joka sopisi vähäisen koodin kehitysympäristön määritelmään, mutta joka on jäänyt pois käytöstä ennen vuotta 2014, jolloin kukaan ei vain ole nimittänyt ratkaisua vähäisen koodin kehitysympäristöksi.

Tarkan syntymävuoden määrittäminen on siis jo valmiiksi hieman haastavaa, mutta jos lisäksi harkitaan, mikä mahdollisesti olisi toiminut vähäisen koodin kehitysympäristöjen esikuvana tai mitkä ideat olisivat saattaneet vaikuttaa vähäisen koodin kehitysympäristöjen syntyyn, niin vähäisen koodin kehitysympäristöjen lähtökohdan määrittäminen vaikeutuu entisestään. Esimerkiksi loppukäyttäjän ohjelmointia (End-User Programming) voidaan pitää eräänlaisena yläkäsitteenä, jonka alle myös vähäisen koodin kehitysympäristöt kuuluvat. Loppukäyttäjän ohjelmoinnissa loppukäyttäjä tuottaa ohjelmia tai laajentaa niiden toimintaa käyttämällä esimerkiksi luonnollista ohjelmointikieltä (Natural-language Programming), visuaalista ohjelmointia (Visual programming) tai taulukkolaskentaa (spreadsheets). Loppukäyttäjän ohjelmoinnin muista työkaluista mielestäni visuaalinen ohjelmointi muistuttaakin

erityisen paljon vähäisen koodin kehitystä ja jotkin vähäisen koodin kehitysympäristöt, kuten OutSystems [9], saattavatkin sisältää visuaalista ohjelmointia. Näin ollen ensimmäisenä visuaalista ohjelmointia hyödyntänyttä Helix tietokantahallintatyökalua voitaisiin pitää eräänlaisena vähäisen koodin kehitysympäristöjen esikuvana. Helix on vuodelta 1983. [10]

Tässä työssä käytetty termin suomennos puolestaan on tehty tämän työn yhteydessä, kun en löytänyt valmista käännöstä. Suomentaessani termiä harkitsin termin osia ja kokonaisuutta:

- ”Low” voitaisiin kääntää matala, vähän tai vähäinen. ”Matala” olisi voinut toimia viittauksena madallettuun kynnykseen, joka sallii kansalaiskehittäjien käyttää alustaa, mutta tämä olisi kuitenkin ollut hyvin kömpelö käännös. En myöskään kokenut, että ”vähän koodattava” tai ”vähän koodaava” toimisi erityisen hyvin. Mielestäni ”vähäisen (koodin kehitysympäristö)” oli loogisin valinta, sillä tarkoitus on että kehittäjä kirjoittaa vain vähäisen määrän koodia ympäristöä käytettäessä.
- ”Code” voitaisiin kääntää vielä virallisemmin ”ohjelmakoodi,” mutta tämä tekisi jo valmiiksi pitkästä termistä vielä pidemmän ja koen, että konteksti kertoo riittävän hyvin millaisesta koodista on kysymys.
- ”Development Platform” puolestaan voisi alkaa joko ”ohjelmointi” tai ”kehitys” ja päättyä ”alusta”. Vaihtoehtoisesti se voitaisiin kääntää kokonaisuutena myös ohjelmistokehitysalustaksi. Kun kuitenkin otetaan huomioon, että tarkoitus on vähimmäistää ohjelmakoodin kirjoittaminen, mielestäni ”ohjelmointi” olisi tässä yhteydessä huonompi valinta. Käännös ”kehitysalusta” olisi siis osuva, mutta vertailu perinteisten kehitys**ympäristöjen** ja vähäisen koodin kehitys**alustojen** välillä voisi olla hieman hämäävä. Toisaalta, vähäisen koodin kehityksen yhteydessä kehitysympäristö liittyy aina tiettyyn kehitysalustaan, joten näillä kahdella ei ole merkittävää eroa.

Lopulta päädyin siis kääntämään termin ”vähäisen koodin kehitysalustaksi”, mutta vertailussa viittaamaan lähinnä ”vähäisen koodin kehitysympäristöihin”. Tämä oli mielestäni kaikin puolin paras ratkaisu.

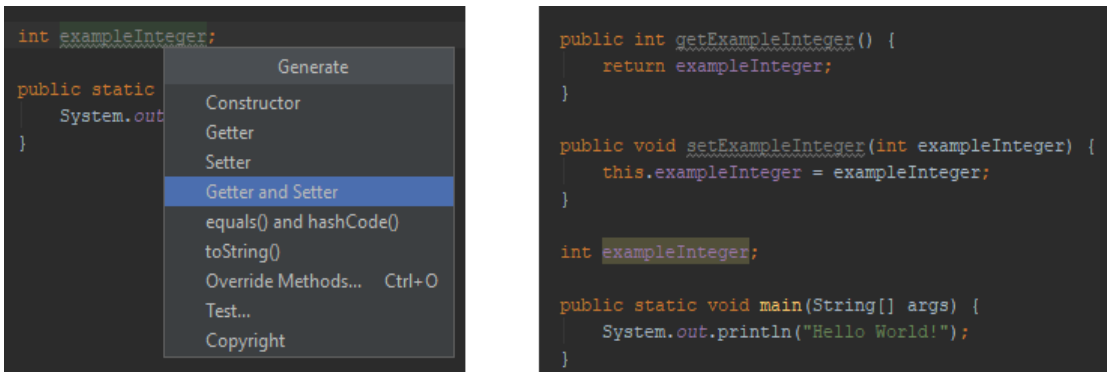
3. KEHITYSYMPÄRISTÖTYYPPIEN EROT

Tässä luvussa käydään läpi ympäristötyyppien teknisiä eroja. Samalla avataan johdannossa esitettyjä kehitysympäristöjen määritelmiä ja pohditaan niitä tarkemmin.

Luku jakautuu kahteen alalukuun, joista kumpikin keskittyy omaan kehitysympäristötyyppiinsä. Lukujen lopuissa listataan muutamia esimerkkejä kehitysympäristöistä.

3.1. Perinteiset kehitysympäristöt

Perinteiset kehitysympäristöt lähtevät olettamuksesta, että kehittäjä itse tuottaa tarvittavan koodin. Tässä vaiheessa modernimpia kehitysympäristöjä käyttäneet kehittäjät saattavat kuitenkin huomata, että perinteistenkin kehitysympäristöjen modernit versiot voivat myös automaattisesti tuottaa koodia. Kehitysympäristö voi esimerkiksi tarjota valmiin "Hello World"-ohjelman, jolloin kehittäjän ei itse tarvitse muistaa main-funktion syntaksia. Jotkin modernit kehitysympäristöt, kuten Intellij IDEA, voivat lisäksi automatisoida myös muun muassa kirjastojen lisäämisen (include statement) kehittäjän lisätessä muuttujan, jonka tyyppiä ei vielä löydy käytetyistä kirjastoista, mutta jonka tyyppi löytyy kehitysympäristön tuntemasta kirjastosta, tai jopa tuottaa pieniä määriä rutiininomaista koodia, kuten asetus- ja palautusfunktioita.



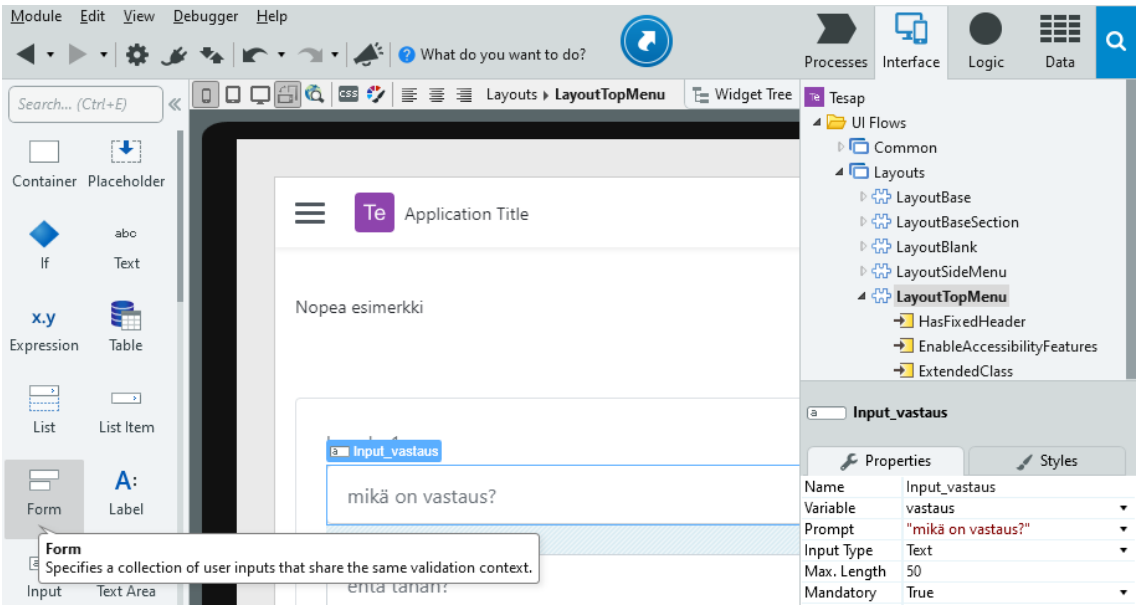
Kuva 1: IntelliJ IDEA asetus- ja palautusfunktio automaatioesimerkki

Tämä voi hämärtää rajaa vähäisen koodin kehitysympäristöjen ja perinteisten kehitysympäristöjen välillä, mutta pääpiirteittäin eron voisi muotoilla sanoiksi siten, että perinteisissä kehitysympäristöissä kehitys on tekstipohjaista ja vähäisen koodin kehitysympäristöissä graafista. Toisaalta perinteisten kehitysympäristöjen enenevä automatisoituminen viittaa siihen, että vähäisen koodin kehitys on ideana validi.

Perinteisten kehitysympäristöjen tehokas käyttö vaatii kattavaa osaamista ohjelmakehityksessä ja niiden käyttäjien odotetaan tuntevan käytettävä ohjelmointikieli. Esimerkkejä perinteisistä kehitysympäristöistä ovat Visual Studio, IntelliJ Idea, CLion, Eclipse ja Notepad++.

3.2. Vähäisen koodin kehitysympäristöt

Kuten jo mainittu, ohjelmistojen tuottaminen vähäisen koodin kehitysympäristöillä pääasiallisesti tapahtuu graafisen käyttöliittymän kautta. Richardson ja Rymer listaavat tarkemmin vähäisen koodin kehityksen sisältävän visuaalista datan mallinnusta, vuokaavioita, ”What You See Is What You Get” -käyttäjäkokemusta sekä deklaratiivisia työkaluja, joihin voi sisältyä toimialuekohtainen ohjelmointikieli (Domain Specific Language) [11]. Toimialuekohtainen ohjelmointikieli tässä tarkoittaa minikieltä, joka on luotu tiettyä tehtävää tai ohjelmaa varten, kuten esimerkiksi Unixin shell skriptit tai R statistiikkakieli.



Kuva 2: OutSystemsin käyttöliittymä. Kuvaa on muokattu pienemmäksi siirtelemällä elementtejä siten, että kaikki oleelliset elementit ovat nähtävissä.

Kuvasta 2 nähdään esimerkki vähäisen koodin kehitysympäristön käyttöliittymästä. Kuvan tilanteessa kuvitteelliseen Tesap testiapplikaatioon on lisätty lomake *Form* lomaketyökalulla vetämällä elementti paikalleen kuvan keskellä olevaan esikatselunäkymään. Tähän lomakkeeseen on lisäksi lisätty *Input* syöttökenttiä, joista Input_vastaus on muokattavana kuvan oikeassa alalaidassa. Oikeassa ylälaidassa on lisäksi välilehdet suoritettavien prosessien määrittämiselle, ohjelman logiikan muokkaamiselle ja ohjelman tietokantarakenteen määrittämiselle.

Vähäisen koodin kehitysympäristöt ovat perinteisiä ympäristöjä kapeampialaisia, eli ne ovat yleensä suunniteltu tietynlaista kehitystä varten. Richardson ja Rymer jakavatkin vähäisen koodin kehitysympäristöt viiteen eri kategoriaan käyttötarkoituksen perusteella:

- **Yleiskäyttöiset kehitysympäristöt** tarjoavat laajan kirjon internet- ja mobiiliapplikaatioita. Ne pyrkivät korvaamaan Javan, .NETin sekä muut perinteiset kehitysalustat. Niitä markkinoidaan suurille, keskeisille ja liiketoimialakohtaisille yrityksille. Yleiskäyttöiset alustat täyttävät applikaatioiden tietokantatarpeet ja lisäksi tarjoavat kattavan valikoiman deklaratiivisia työkaluja, jotka kattavat applikaation koko elinkaaren.

- **Prosessiapplikaatioiden kehitysympäristöt** ovat suunnattu applikaatioille, jotka vaativat yhteistyötä ja koordinaatiota useiden eri työntekijä- ja asiakasroolien välillä. Ne tarjoavat prosessiautomaatio-, palveluohjaus- sekä sosiaalisen kanssakäymisen ominaisuuksia. Ne on suunnattu ensisijaisesti liiketoimiala- ja teknologiahallinto-ostajille. Ne kykenevät prosessien, lomakkeiden ja tehtävien visuaaliseen mallintamiseen sekä tarjoavat vastauksen monimutkaisiin prosessi- ja yhteistyöskenaarioihin.
- **Tietokanta-applikaatioiden kehitysympäristöt** ovat suunnattu applikaatioille, jotka keräävät, hakevat ja näyttävät relaatiotietokantoihin tallennettua tietoa. Niitä markkinoidaan ensisijaisesti liiketoimiala- ja pienyrityksille. Ne tarjoavat perus ”What You See Is What You Get”- ja tietojenkäsittelytyökalut.
- **Pyyntöjä käsittelevien applikaatioiden kehitysympäristöt** ovat suunnattu applikaatioille, jotka ottavat vastaan, prosessoivat ja seuraavat pyyntöjä. Niitä markkinoidaan infrastruktuurin ja operaatioiden ammattilaisille, liiketoimialajohtajille sekä ratkaisukehittäjille. Ne tarjoavat palvelukatalogin, joka tukee supporting-request ja submit-and-track prosesseja. Ne tarjoavat ratkaisun pyyntöjen hallinnointimallille.
- **Mobiiliapplikaatioiden kehitysympäristöt** pyrkivät yksinkertaistamaan järjestelmäriippumatonta mobiilikehitystä. Niitä markkinoidaan ensisijaisesti kuluttajakeskeisille yrityksille. Ne tarjoavat mobiiliwidgettejä sekä sovelluskauppajulkaisutyökaluja. [11]

Käyttäjäkunta näillä kehitysympäristöillä on kirjava. Eri tahojen mukaan vähäisen koodin kehitysympäristöjä käyttää tai ne soveltuvat muun muassa niin ammattilaisille kuin kansalaiskehittäjille [2] tai esimerkiksi ammattilaisille käyttäjäkokemussuunnittelijoille [3] ja sallivat jopa bisnesasiantuntijoiden osallistuvan kehitykseen [12]. Tämä käy järkeen ottaen huomioon, että yksi vähäisen koodin kehitysympäristöjen olemassaolon keskeisistä tavoitteista mainitaan usein olevan vähentää kehittäjältä vaadittavaa ohjelmointikielen ymmärrystä [2][4][3] ja toisaalta nopeuttaa ammattilaisten kehittäjien työtä [2]. Esimerkkejä vähäisen koodin kehitysympäristöistä ovat Appian, OutSystems, Google App Maker (lopetettu 2021), MS PowerApp ja Salesforce.

4. TEKIJÄNOIKEUS JA LISENSOINTI

Nyt kun kehitysympäristöjen erot ovat selviä, siirrytään puhumaan tekijänoikeudesta ja lisensoinnista. Tämä luku jakautuu muiden lukujen tapaan yleiseen pohdintaan tekijänoikeudesta ja lisensoinnista, jonka jälkeen alaluvuissa tarkastellaan lisensointia ja tekijänoikeutta ensin perinteisissä kehitysympäristöissä ja sitten vähäisen koodin kehitysympäristöissä.

Kun vähäisen koodin kehittäjä käyttää alustan tarjoamia työkaluja automaattisesti generoimaan käyttöliittymä elementtejä, näiden taustalla oleva ohjelmakoodi ei synny tyhjästä vaan vähäisen koodin kehitysympäristön alkuperäinen kehittäjä on kirjoittanut koodille pohjan, josta alusta generoi alkuperäisen kehittäjän määräämällä tavalla ja mahdollisesti hyvin vähäisillä muutoksilla lopullisen koodin kehittäjälle. Herää siis kysymys, että kuka omistaa vähäisen koodin kehitysympäristöllä luodun koodin tekijänoikeuden?

Esimerkiksi Suomen tekijänoikeuslaki sanoo, että tietokoneohjelmaa pidetään kirjallisena teoksena, tekijällä on yksinomainen oikeus määrätä teoksesta valmistamalla siitä kappaleita muuttamattomana tai muutettuna ja että ”kappaleen valmistamisena pidetään sen valmistamista kokonaan tai osittain, suoraan tai välillisesti”. [13] Tältä kannalta vaikuttaisi siis siltä, että tekijänoikeus vähäisen koodin kehitysympäristön välillisesti tuottamaan koodiin kuuluu vähäisen koodin kehitysympäristön alkuperäisellä kehittäjällä, eikä ympäristöä käyttävällä kehittäjällä.

Ohjelmistokehitysyhtiöiden liikeideaan kuitenkin usein kuuluu myydä käyttöoikeus tuotettuun ohjelmaan loppukäyttäjille. On siis tarkasteltava, että minkälaisia lisenssejä vähäisen koodin kehitysympäristön käyttöön liittyy ja miten alustalla tuotettuja ohjelmia voi myydä eteenpäin. Toisaalta myös perinteiseen ohjelmistokehitykseen voi liittyä lisenssejä, esimerkiksi käytettäessä muiden kehittäjien kirjoittamia kirjastoja.

4.1. Lisensointi perinteisessä ohjelmistokehityksessä

Perinteisissä kehitysympäristöissä lisenssi maksetaan ohjelman käytöstä ja pois lukien mahdollisten kolmansien osapuolten kirjastojen käyttöoikeuksien ostaminen, yritys tai kehittäjä omistaa tekijänoikeuden kirjoittamaansa ohjelmaan. Ainakin vielä toistaiseksi perinteiset kehitysympäristöt vastaavat enemmän työkalua, kuten kynää kirjoittajalle tai lastaa muurarille, joten ne eivät pysty luomaan tekijänoikeudellisia vaateita ympäristöllä tuotettuun ohjelmaan ympäristön alkuperäiselle kehittäjälle. Vaikka jonkinlaista automaatiota jo on, kuten kuvasta 1 nähdään, niin tämä automaatio on kuitenkin hyvin vähäistä eivätkä automaatiolla tuotetut hyvin lyhyet, rutiininomaiset metodit todennäköisesti täytä tekijänoikeuslaissa mainittua teoksen määritelmää [13]. Tilanne voi kuitenkin muuttua, jos kehitysympäristöihin sisällytetään esimerkiksi tekoälyä, joka ottaa vastuulleen suurempien osuuksien rakentamisen ohjelmistokehityksessä.

Jetbrainsin kauppasivun mukaan IntelliJ IDEA Ultimaten organisaatioille tarkoitettu lisenssi maksaa vuosittain 499,00 € per käyttäjä. Yksityiset käyttäjät saavat lisenssin 150.00 € vuosimaksulla ja esimerkiksi opiskelijat saavat kehitysympäristön käyttöönsä ilmaiseksi. [14] IntelliJ IDEA:n käyttöön ei sisälly muita maksuja.

Visual Studio Professional, joka on yrityksille suunnattu versio, maksaa Microsoftin kauppasivun mukaan 37,95 € kuussa per käyttäjä, eli 455,4 € vuodessa. [15] Microsoft tarjoaa myös ilmaisen version yksityisille käyttäjille, Visual Studio Community, mutta tämä saattaa erota ominaisuuksiensa puolesta. Visual Studion käyttöön ei myöskään sisälly muita maksuja.

Eclipse on puolestaan täysin ilmainen. Sen käyttäminen ei maksa mitään, se ei vaadi kertaostosta, eikä Eclipsellä tuotetuista ohjelmista tarvitse maksaa tekijänoikeuskorvauksia Eclipse Foundationille. Tähän ei myöskään vaikuta se, että käytetäänkö Eclipseä harrastusmielessä vai kaupalliseen tarkoitukseen. [16]

Notepad++ on Eclipsen tavoin täysin ilmainen yksityiseen sekä kaupalliseen käyttöön. Se oikeastaan menee vielä hieman pidemmälle, sillä myös Notepad++:n lähdekoodia saa käyttää GPL:n ehtojen mukaisesti. [17]

4.2. Lisensointi vähäisen koodin kehitysympäristöissä

Vähäisen koodin kehitysympäristöissä kehitysympäristön alkuperäisellä kehittäjällä voi olla enemmän vaikutusta kehitysympäristöllä tuotetun lopputuotteen lähdekoodiin ja tämä teoriassa voisi aiheuttaa jonkinlaisen tekijänoikeuden alkuperäiselle kehittäjälle lopputuotteeseen. Alkuperäinen kehittäjä voisi teoriassa esimerkiksi kirjoittaa kirjaston, jota kaikki kehitysympäristön graafisessa käyttöliittymässä luodut elementit hyödyntävät. Ohjelmakirjastot ovat laajoja itsenäisiä kokonaisuuksia, joita voidaan saattaa muiden käytettäväksi erilaisten lisenssien kanssa. Nämä lisenssit sisältävät erilaisia ehtoja kirjaston käytölle [18] ja ovat laillisesti sitovia, vaikka lähdekoodi olisi jaettu vapaasti [19][20]. Teoriassa siis vähäisen koodin kehitysympäristön alkuperäinen kehittäjä voisi ensin lisensoida itse vähäisen koodin kehitysympäristön, laskuttaa sen käytöstä ja lisäksi pyytää tekijänoikeuskorvauksen jokaisesta myydystä lopputuotteesta, jossa käytetään kehitysympäristön kehittäjien kirjoittamaa kirjastoa.

Esimerkiksi Salesforce Platform maksaa joko 25 $ per käyttäjä + 25 $ per käyttäjä, jos käytetään Lightning Console -palvelua. Tämä on kuukausihinta, eli 300 $ tai 600 $ vuodessa. Lisäksi lopulliset ohjelmat ovat rajoitettuja kymmeneen mukautettuun dataobjektiin. Jos dataobjekteja tarvitaan enemmän, käyttö maksaa 100 $ per käyttäjä. [21]

Appian puolestaan laskuttaa 54 € per normaali käyttäjä per kuukausi, 16 € kuukaudessa per käyttäjä, joka kehittää korkeintaan neljänä 24 tunnin jaksona kuukaudessa tai 1,80 € käyttäjiltä, jotka eivät kuulu asiakasorganisaatioon mutta joiden tarvitsee lähettää kyselyitä. Lisäksi käyttäjiä pitää olla vähintään 100. Näin ollen Appian vaikuttaisi ehkä jopa markkinoiden kalliimmalta ratkaisulta, mutta Appian väittää muilla tarjoajilla olevan piilotettuja kuluja, jotka tulevat esiin vasta kun asiakas on jo sitoutunut kehitysympäristöön [22].

Microsoftin Powerapps maksaa 10 $ per käyttäjä per sovellus per kuukausi tai 40 $ per käyttäjä per kuukausi rajattomalla määrällä sovelluksia. Lisäksi jotkin ominaisuudet myydään

erikseen, esimerkiksi tekoälyn sisällyttäminen sovelluksiin maksaa 500 $ per miljoona palvelukrediittiä per kuukausi, 100,000 sivunlatausta maksaa 100 $ kuukaudessa ja 100 käyttäjän kirjautumissessiota maksaa 200 $ kuukaudessa. [23]

OutSystems tarjoaa rajoittamattoman määrän applikaatioita joko tuhannelle loppukäyttäjälle hintaan 4,000 $ per kuukausi, tai rajoittamattomalle määrälle loppukäyttäjiä hintaan 10,000 $ per kuukausi [24]. Mielenkiintoisen tästä hinnoittelusta tekee se, että tämän 10,000 $ per kuukausi Standard Editionin voi vaihtoehtoisesti asentaa OutSystemsin pilvipalvelimen sijasta asiakkaan omalle pilvipalvelimelle. Omien resurssien käyttäminen ei kuitenkaan vähennä hintaa, ennemminkin päinvastoin sillä 4,000 $ Basic Editionia ei voi asentaa asiakkaan omalle palvelimelle, vaikka haluaisi.

Vaikuttaa siltä, että vähäisen koodin kehitysympäristöille on tyypillistä olla yhdistettynä jonkinlaiseen pilvipalveluun. Tämä ei kuitenkaan välttämättä ole vähäisen koodin kehitysympäristöjen synnynnäinen ominaisuus, vaan mahdollisesti seurausta siitä, että vähäisen koodin kehitysympäristöt ovat nousseet suosioon pilvipalvelujen aikakaudella. Toisaalta tämä on myös loistava tapa vähäisen koodin kehitysympäristön tarjoajalle enimmäistää oma kannattavuutensa.

5. KUSTANNUSTEHOKKUUS

Tässä luvussa käsitellään tehokkuutta lähinnä tiettyjen vähäisen koodin kehitysympäristöjen kannalta siten, että perinteiset kehitysympäristöt tarjoavat lähtöarvon. Perinteistenkin ympäristöjen välillä toki on eroja, esimerkiksi työrutiini, jossa Notepad++:lla kirjoitetaan ohjelma, tallennetaan, käännetään ohjelma komentorivillä ja lopuksi ajetaan ohjelma ei ole yhtä tehokas kuin rutiini, jossa IntelliJ Ideassa tehdään sama painamalla 'aja'. Nämä erot eivät kuitenkaan ole oleellisia tämän työn kannalta.

Vähäisen koodin kehitysympäristöjen tehokkuuden selvittämistä vaikeuttaa se, ettei aiheesta juurikaan ole tehty tutkimuksia. Tällä hetkellä laadukkain tutkimus vaikuttaisi olevan Forresterin 2019 vuonna tekemä The Total Economic Impact of Power Apps and Power Automate, jossa Jonathan Lipsitz ja Jon Erickson selvittivät Microsoftin Power Apps vähäisen koodin kehitysympäristön ja Power Automate UI automaatio sovelluksen tuottamia säästöjä sovelluskehityksessä [25]. Tätä tutkimusta tarkastellessa on hyvä huomioida, että tutkimus on tehty Microsoftin, jonka palveluja Power Apps ja Power Automate ovat, tilauksesta ja ennen raportin julkaisemista Microsoft on käynyt raportin läpi ja antanut palautetta sen sisällöstä. Forrester kuitenkin vakuuttaa säilyttäneensä toimituksellisen valvonnan itsellään, eikä suostuisi julkaisemaan harhaanjohtavaa materiaalia. [25]

Forrester tutki kolmen Pohjois-Amerikassa ja yhden Iso-Britanniassa sijaitsevan yrityksen säästöjä ja tehokkuuden lisäystä siirryttäessä perinteisistä kehitysympäristöistä vähäisen koodin kehitysympäristöön, tarkemmin sanottuna Power Appsiin. Yritykset toimivat eri aloilla ja niillä oli 430:stä 3500:aan Powerratkaisujen käyttäjää. Näiden pohjalta Forrester koosti yhdistelmäyrityksen, jolla on 2000 työntekijää sekä 21 kehitysprojektia ja käytti tätä esimerkkiyritystä tulosten ilmaisemiseen. [25] Valitettavasti Forrester ei sen tarkemmin eritellyt alkuperäisiltä yrityksiltä kerättyä tietoa; ei miten paljon yksittäiset yritykset käyttivät perinteisillä kehitysympäristöillä eikä sitä paljonko ne käyttivät Power Appsillä ja Power Automatella, joten tuloksista ei nähdä onko eri toimialoilla eroja eikä sitä, että vaikuttaako yrityksen koko lineaarisesti vai eksponentiaalisesti.

Tutkimuksessaan Forrester jakoi vähäisen koodin kehitysympäristöjen tuottaman hyödyn kolmeen eri kategoriaan, joissa hyötyä kuvataan syntyneiden säästöjen kannalta. Nämä kolme kategoriaa ovat vähennetty informaatioteknologiavaiva (IT-vaiva, Reduced IT effort), korvatut kolmansien osapuolten sovellukset (Retired third-party applications) sekä yksinkertaistetut liiketoimintamenettelyt (Streamlined business processes). [25] Vaikka kaksi jälkimmäistä kategoriaa ovat mainitsemisen arvoisia, näistä kolmesta merkityksellisin tämän työn kannalta on vähennetty IT-vaiva.

Yhdistelmäyrityksen kokonaissäästön siirryttäessä perinteisistä kehitysympäristöistä vähäisen koodin kehitysalustoille Forrester arvioi olevan 8,9 miljoonaa dollaria nykyrahassa 10 % riskipainotuksella kolmen vuoden tarkasteluajanjaksolla. Tästä 3,5 miljoonaa syntyy

vähennetystä IT-vaivasta ja 5,3 miljoonaa yksinkertaistetuista liiketoimintamenettelyistä, kuten nopeammasta hinta-arvioiden, raporttien ja tuloslaskelmien tuottamisesta. [25]

Vähennetty IT-vaiva puolestaan jakautuu vähennettyihin perinteisen kehityksen kuluihin sekä vähennettyihin IT hallinnon ja ylläpidon kuluihin. Tässä Forrester esittää vähäisen koodin kehityskulujen olevan 200 tuhatta dollaria per projekti vähemmän kuin perinteiset kehityskulut ja yhdistelmäyrityksellä olevan alun perin yksi kehitysprojekti, joka ensimmäisenä vuotena muuttuu neljäksi ja toisena kahdeksaksi jona se pysyy kolmannen vuoden. Eli kolmena vuonna syntyy säästöä yhteensä 4,1 miljoonaa ensimmäisen projektin ollessa huomattavasti normaalia pienempi. Tämän lisäksi Forrester esittää ylläpidon kulujen olevan 20 % edellisen vuoden kehityskulujen summasta, jolloin ylläpitokulut vähenisivät 20 tuhatta dollaria ensimmäisenä vuotena, 180 seuraavana ja 500 tuhatta dollaria kolmantena vuotena eli yhteensä 700 tuhatta dollaria. Forrester laskee vähennetyn IT-vaivan sekä vähennetyt hallinnon ja ylläpidon kulut yhteen, arvioi inflaation ja tekee 10 % riskipainotuksen, jolloin päästään tuohon 3,5 miljoonan summaan. [25]

Tämä 3,5 miljoonan dollarin säästö vuodessa voi kuulostaa melko suurelta väitteeltä. Kuitenkin, jos huomioidaan suurimman osan tutkimuksessa tarkastelluista yrityksistä sijaitsevan Pohjois-Amerikassa, jossa ohjelmistokehittäjien mediaanipalkka on noin 93 tuhatta dollaria vuodessa [26], 3,5 miljoonan tai 8,9 miljoonan vuosisäästö 2000 hengen yritykselle ei kuulosta enää kovin ihmeelliseltä.

Valitettavasti muut tahot eivät varsinaisesti ole tutkineet vähäisen koodin kehitysympäristöjen tai -alustojen tehokkuutta, eivätkä mahdollisesti aiheutuvia säästöjä. Yleisesti hyväksytty mielipide kuitenkin on, että vähäisen koodin kehitysympäristöt ovat perinteisiä nopeampia [2][27][28] ja siten tehokkaampia.

6. KETTERYYS

Seuraavaksi käsitellään ketteryyttä. Ensin pohditaan hieman, että mitä ketteryys on ja minkälaisiin osa-alueisiin se jakautuu. Tämän jälkeen käydään osa-alueet läpi siten, että jokainen osa-alue alkaa perinteisten kehitysympäristöjen tarkastelulla, josta jatketaan vähäisen koodin kehitysympäristöjen tarkasteluun.

Ohjelmistotuotannon kontekstissa ketteryydellä voidaan tarkoittaa useita eri asioita [29]. Tässä ketteryyttä tarkastellaankin kolmella eri tavalla:

i. Joustavuus, eli taipuisuus [30], on eräänlainen ketteryyden osa-alue [29]: jos jokin asia on ketterä, se todennäköisesti on ennemminkin taipuisa kuin kankea. Ensimmäisessä tavassa pohditaan siis sitä, että kuinka hyvin ohjelmistokehitystyökalu taipuu eri tarkoituksiin
ii. Toisaalta ketteryys on myös nopeaa liikkuvuutta [29], joka ohjelmistotuotannon kontekstissa tarkoittaa uusien iteraatioiden tuottamista nopeasti. Tarkastellaan siis eri kehitystyökalujen nopeutta.
iii. Kolmanneksi tarkastellaan toisenlaista taipuisuutta, korvattavuutta, eli sitä, että kuinka helposti asiakaskehittäjä voi siirtää projektinsa kehitysympäristöstä toiselle.

Muitakin tapoja ymmärtää ketteryys toki on, mutta tässä kontekstissa nämä kolme ovat merkittävimpiä. Lisäksi on myös hyvä huomioida, että tässä yhteydessä sanalla ketteryys ei viitata *ketterään kehitykseen* (Agile Development) eikä vertailun tarkoitus siis ole esimerkiksi pohtia eri kehitysympäristöjen soveltuvuutta ketterän kehityksen työkaluksi, vaikkakin vertailusta voisi mahdollisesti vetää johtopäätöksiä asiaan liittyen.

6.1. Tapa i – taipuisuus

Tavalla (i) tarkasteltuna, perinteiset kehitysympäristöt ovat todella ketteriä. Esimerkiksi Oracle väittää, että vuonna 2016 15 biljoonaa laitetta käyttivät Javaa [31]. Vaikka tämä luku on hyvin suuri, noin kaksi kertaa planeetan väkiluku, se on kuitenkin uskottava, sillä Java on toiseksi yleisin ohjelmointikieli [32] ja maailmassa oli 2016 jo 17,68 biljoonaa pelkästään esineiden internetin laitetta [33]. Javaa onkin käytetty kaikkeen rakettitieteen ratkaisuista [34] viihdeteollisuuden tarkoituksiin [35], eli toisin sanoen kehitysympäristö, joka sallii näin taipuisan ohjelmointikielen vapaan käytön, on puolestaan myös hyvin taipuisa. Vaikka ohjelmointikielten välillä on eroja, jotkin kielet sopivat paremmin tiettyihin tarkoituksiin, niin myös muut kielet ja niillä kehittämiseen tarkoitetut perinteiset kehitysympäristöt ovat verrattain taipuisia.

Vähäisen koodin kehitysympäristöjen kantava ajatus on, ettei kehittäjän tarvitsisi kirjoittaa kovinkaan paljoa ohjelmakoodia. Joskus tämä on kuitenkin välttämätöntä halutun toiminnon aikaansaamiseksi. Tarkastellaan siis, kuinka hyvin tämä onnistuu eri esimerkkiympäristöissä:

- OutSystems mahdollistaa HTML:n käytön suoraan kehitysympäristön käyttöliittymäkomponentilla, mutta sallii myös käyttäjäkehittäjän luomien .NET-komponenttien hyödyntämisen [36]. Lisäksi OutSystems tarjoaa ulkopuolisten järjestelmien ja tietokantojen helpon integroinnin automaattisella takaisinmallinnustyökalulla (reverse engineering tool) [36].
- Appian puolestaan tarjoaa ohjelmistokehityspaketin (Software Development Kit, SDK), jolla asiakaskehittäjä pystyy yhdistämään sovelluksen esimerkiksi Salesforcen, Googlen tai Amazonin palveluihin. Lisäksi Appian tarjoaa ohjelmointirajapinnan (Application Programming Interface, API), jota hyödyntäen asiakaskehittäjä voi vapaasti luoda liitännäisiä (Plug-in). [37]
- Salesforce Platform tarjoaa erilaisia toimialuekohtaisia ohjelmointikieliä käyttöliittymän mukauttamiseen ja taustajärjestelmäkehitykseen (Back-end development). Lisäksi Salesforce tarjoaa useita ohjelmointirajapintoja eri osa-alueiden toiminnan laajentamiseksi, kuten esimerkiksi *SOAP* ohjelmointirajapinnan yhdistämään asiakaskehittäjän sovelluksen Salesforce Platformin ulkopuolisiin sovelluksiin tai *Tooling* ohjelmointirajapinnan mukautettujen työkalujen kehittämiseen Platformsovelluksille. [38]
- Microsoft Powerapp saattaa asiakaskehittäjän käyttöön hyvin kattavan valikoiman erilaisia ohjelmointirajapintoja ulkopuolisten sovellusten liittämiseksi asiakaskehittäjän sovellukseen [39]. Tämän lisäksi Microsoft mahdollistaa myös asiakaskehittäjän sovelluksen laajentamisen erilaisilla liitännäisillä sekä .NET kokoonpanolla (Assembly) [39].

Vaikka alkuun voisi vaikuttaa siltä, että vähäisen koodin kehitysympäristöt ovat hyvin rajoittuneita ohjelmakoodin kirjoittamisen kannalta, esimerkeistä nähdään, että kehitysympäristöstä riippuen myös vähäisen koodin kehitysympäristöt sallivat asiakaskehittäjän kirjoittaa mielivaltaista koodia melko vapaasti. Erilaisten liitännäisten kirjoittaminen ei kuitenkaan salli ohjelman ytimen luonteen muuttamista, eli vähäisen koodin kehitysympäristöllä luodut sovellukset ovat kuitenkin sidottuja kyseisen kehitysympäristön alkuperäiseen tarkoitukseen. Toisin sanoen, vähäisen koodin kehitysympäristöt ovat tavalla i tarkasteltuina myös melko taipuisia, mutta vain omilla aloillaan.

6.2. Tapa ii – nopeus

Tavalla (ii) tarkasteltuna perinteiset kehitysympäristöt ovat hieman kömpelömpiä. Forresterin tutkimuksessa Richardson ja Rymer tarkastelivat eräiden kehitysprojektien työmääriä: Yhdysvaltojen hallituksen *Affordable Care Act*in dokumenttien määräystenmukaisuus moduulin (Document compliance module) kehitykseen meni perinteisillä ympäristöillä 100 henkilötyökuukautta, erääseen puhelinpalveluoperaattorin sovellukseen 4 kuukautta ja Espanialaisen vakuutusyhtiön verkkokanavaan & hallintatyökaluun noin 2,7 vuotta [11]. Perinteisillä kehitysympäristöillä uusien iteraatioiden kehittäminen onkin usein niin hidasta, että varsinkin alkuvaiheessa uudet projektit saattavat hyödyntää erilaisia koekappaletyökaluja, kuten esimerkiksi paperi- ja rautalankaprototyyppejä [40].

Toisaalta tässä suhteessa vähäisen koodin kehitysympäristöt tuottavat verrattain parempia tuloksia. Edellä mainittuihin Forresterin tarkastelemiin kehitysprojekteihin meni vähäisen koodin kehitysympäristöillä huomattavasti vähemmän aikaa: *Affordable Care Act*in dokumenttien määräystenmukaisuus moduuliin vain 5 henkilötyökuukautta, puhelinpalveluoperaattorin sovellukseen 3 viikkoa ja vakuutusyhtiön verkkokanavaan & hallintatyökaluun vain 13 viikkoa [11]. Kuvasta 2 nähdäänkin, että esimerkiksi OutSystemsin vähäisen koodin kehitysympäristö muistuttaa erilaisia rautalankaprototyyppityökaluja [41][42], paitsi että lopputulos on jo valmiiksi toimiva ohjelma.

6.3. Tapa iii – korvattavuus

Tavalla (iii) tarkasteltuna perinteisen kehitysympäristön asiakaskehittäjällä on vaihtoehtoja. Perinteiset kehitysympäristöt saattavat luoda projektille kehitysympäristökohtaisen projektitiedoston, joka kertoo pääfunktion sijainnin ja mahdollisesti joitakin ohjelman kääntämiseen liittyviä parametreja, mutta tämä on yksinkertainen tekstipohjainen tiedosto, jonka sisältö on verrattain helppoa muuttaa toisen samalle ohjelmointikielelle luodun perinteisen kehitysympäristön projektitiedostoksi. Toisin sanoen, jos asiakaskehittäjälle tulee tarve siirtyä yhdestä perinteisestä kehitysympäristöstä toiseen, tämä on täysin mahdollista ja jopa helppoa. Riippuen kohteena olevasta vähäisen koodin kehitysympäristöstä, voi olla myös mahdollista siirtää projekti kohtuullisella vaivalla perinteisestä vähäisen koodin kehitysympäristöön.

Vähäisen koodin kehitysympäristöä voi kuitenkin olla hankalampaa korvata toisella. Tarkastellaan esimerkkikehitysympäristöjä:

- Outsystems sallii moduulien tallentamisen vain oml-tiedostoina (Outsystems Modeling Language) tai myös Outsystemsin omassa muodossa xif-tiedostoina (Extension and Integration Framework) ja kokonaisten applikaatioiden tallentamisen vain oap-tiedostoina (Outsystems Application Pack) [43]. Asiakaskehittäjällä ei siis ole pääsyä koko lähdekoodiin muodossa, jossa sen voisi siirtää toiselle kehitysalustalle.
- Appian puolestaan mahdollistaa sovellusten viennin zip-tiedostoina [44], joiden sisältö on normaalimmissa tiedostomuodoissa [45]. Näin ollen asiakaskehittäjän voi olla helpompaa viedä Appianilla kehitetty sovellus toiseen kehitysympäristöön, mutta tällöin tulee toki muistaa, ettei asiakaskehittäjällä välttämättä ole oikeutta hyödyntää Appianin ohjelmakirjastoja, joihin suoraan Appianista viety ohjelmakoodi saattaa viitata.
- Myös Salesforce Platformista on mahdollista viedä kokonaisia sovelluksia muodossa, jossa sovelluksen voi siirtää toiseen kehitysympäristöön hyödyntämällä Ant-migraatiotyökalua [46]. Tällöin toki pätevät samat tekijänoikeudelliset huomiot kuin Appianin kanssa.
- Microsoftin Powerplatformilta on tässä työssä tarkastelluista vähäisen koodin kehitysympäristöistä mahdollisesti helpointa viedä sovellus toiseen kehitysympäristöön. Powerplatform ei vain tarjoa työkalua sovelluksen vientiin [47], vaan Microsoftilla on myös todella kattava dokumentaatio siitä, että mitä Powerplatformin omat kirjastot tekevät [48].

Vaikuttaa siis siltä, että ainakin teknisesti suurimmassa osassa tapauksia vähäisen koodin kehitysympäristössä kehitetyn sovelluksen lähdekoodi on mahdollista siirtää kokonaan tai osin toiseen kehitysympäristöön. Tällöin tulee kuitenkin aina huomioida tekijänoikeus, lisenssit sekä mahdollisesti vähäisen koodin kehitysympäristön tarjoajan kanssa tehdyt sopimukset.

7. JOHTOPÄÄTÖKSET

Tarkastaessa vähäisen koodin kehityksen historiaa nähdään, että vähäisen koodin kehitys ei välttämättä ole ideana aivan uusi, vaan ennemminkin pitkään jatkuneen kehityksen uusin iteraatio. Vähäisen koodin kehitysympäristöt eivät siis ole ensimmäinen yritys madaltaa ohjelmistokehitykseen tarvittavaa taitokynnystä tai tehostaa ohjelmistotuotantoa. Lisäksi havaitaan, että kun asialle annetaan nimi, niin tästä asiasta aletaan tehdä enemmän tutkimuksia tai ainakin nämä tehdyt tutkimukset ovat helpompia löytää.

Toisaalta, vertailtaessa vähäisen koodin kehitysympäristöjä perinteisiin kehitysympäristöihin, huomataan, että vaikka sekä perinteiset että vähäisen koodin kehitysympäristöt saattavat hyödyntää samoja ohjelmistokehitystä tehostavia tekniikoita, kuten automaattista lähdekoodin generointia, niin ne hyödyntävät kyseisiä tekniikoita hyvin eriävissä määrin. Samalla huomataan, että näitä tekniikoita enemmän hyödyntävät vähäisen koodin kehitysympäristöt jakautuvat alalajeihin sen mukaan, millaisia ohjelmia niillä on tarkoitus kehittää. Tästä nähdään kuinka näiden tekniikoiden laajempi hyödyntäminen, vaikkakin mahdollisesti tehostaa ohjelmistokehitystä enenevissä määrin, niin se kuitenkin samalla tekee kehitysympäristöllä tuotettavista ohjelmista yhä kapea-alaisempia.

Taulukko 1 – Lisensointi

	Kehitysympäristö	Hinta
Perinteiset Kehitysympäristöt	IntelliJ Idea Ultimate	499,00 € / ammattikäyttäjä / vuosi 150,00 € / yksityinen käyttäjä / vuosi
	Visual Studio Professional	455,40 € / ammattikäyttäjä / vuosi
	Visual Studio Community	Ilmainen, yksityisille
	Eclipse	Ilmainen, kaikille
	Notepad++	Ilmainen, kaikille
Vähäisen Koodin Kehitysympäristöt	Salesforce Platform	Ensimmäiset 10 dataobjektia 300 $ / käyttäjä / vuosi, Lightning Console palvelu lisäksi 300 $ / käyttäjä / vuosi 110 dataobjektiin asti 1200 $ / käyttäjä / vuosi sisältäen Lightning Consolen
	Appian	648 € / normaalikäyttäjä / vuosi 192 € / kevytkäyttäjä / vuosi 21,6 € / organisaation ulkopuolinen käyttäjä / vuosi Käyttäjiä oltava vähintään 100
	Microsoft Power Apps	120 $ / käyttäjä / sovellus / vuosi 480 $ / käyttäjä / vuosi (rajattomasti sovelluksia) Tekoäly 500 $ / miljoona palvelukrediittiä / kuukausi 100,000 sivunlatausta 100 $ / kuukausi 100 käyttäjän kirjautumissessiota 200 $ / kuukausi
	Outsystems	1000 loppukäyttäjää 48000 $ / vuosi Rajattomasti loppukäyttäjiä 120000 $ / vuosi

Lisensoinnin kannalta perinteiset kehitysympäristöt ovat, kuten Taulukosta 1 – Lisensointi nähdään, edullisempia käyttää. Lisäksi asiakaskehittäjä voi, tai joutuu, kilpailuttamaan verkkoisännöintipalvelun erikseen, siinä missä vähäisen koodin kehitysympäristöillä tuotetut ohjelmistoratkaisut ovat pääsääntöisesti sidottuja kyseiseen vähäisen koodin kehitysalustan pilvipalveluun, josta vähäisen koodin kehitysympäristön alkuperäinen kehittäjä laskuttaa asiakaskehittäjältä. Toisaalta vähäisen koodin kehitysympäristöjen väitetysti parempi kustannustehokkuus johtuu juurikin kehityksen suoraviivaistamisesta ja yksinkertaistamisesta, johon myös kätevien pakettiratkaisujen tarjoaminen lukeutuu.

Taulukko 2 – Ketteryys

Tapa	Perinteinen kehitysympäristö	Vähäisen koodin kehitysympäristö
Taipuisuus	Todella taipuisia, esimerkiksi vuonna 2016 15 biljoonaa laitetta käyttivät Javaa kirjaviin tarkoituksiin	Tehty tiettyyn käyttötarkoitukseen, toiminnallisuutta voidaan tarvittaessa laajentaa ohjelmointirajapintojen kautta
Nopeus	Hitaampi, esimerkiksi *Affordable Care Act*in dokumenttimoduuliin kului 100 henkilötyökuukautta	Nopeampi, sama moduuli toteutettiin 5 kuukaudessa
Korvattavuus	Helppo korvata, vaihtoehtoisia kehitysympäristöjä löytyy ja ne tulkitsevat koodia samalla tavalla	Hankala korvata, sovelluskohtaiset tiedostotyypit hankaloittavat siirtymistä alustalta toiselle eikä alustoilla ole kilpailevia kehitysympäristöjä

Taulukkoon 2 on summattu luvun 6 vertailun tulokset. Näistä tuloksista nähdään, että sekä perinteiset että vähäisen koodin kehitysympäristöt ovat omilla tavoillaan ketteriä, eivätkä kummatkaan ympäristötyypit ole ainakaan tältä kannalta ehdottomasti toisia ympäristötyyppejä parempia. Toisaalta, jos ajatellaan ketterää kehitystä, niin tässä mielessä verratuista osa-alueista tärkein on nopeus. Vaikka ketterän kehityksen kanssa on mahdollista, että asiakkaan vaatimukset ja tätä kautta toteutettava ohjelma muuttuu merkittävästi kehityksen aikana, joka taipuisuuden kannalta voisi suosia perinteisiä kehitysympäristöjä, niin ketterän kehityksen kantava ajatus on kuitenkin tuottaa uusia iteraatioita nopeasti ja tehokkaasti. Niin suuret muutokset vaatimuksissa, että ne pakottaisivat vaihtamaan vähäisen koodin kehitysympäristöltä ja -alustalta toiselle, lienevät myös melko harvinaisia. Vähäisen koodin kehitysympäristöt voisivat siis tukea varsinkin ketterää kehitystä. Tässä on kuitenkin hyvä huomioida, että kuten taulukosta 1 nähdään, niin joidenkin vähäisen koodin kehitysalustojen käyttöön ottaminen voi olla erityisen kallista, eikä tämänkaltainen sitoutuminen välttämättä sovi kaikille ketterän kehityksen työryhmille tai projekteille.

Kaiken kaikkiaan sen lisäksi, että vähäisen koodin kehitysympäristöt soveltuvat tehostamaan suurten yritysten ohjelmistotuotantoa, ne voisivat ominaisuuksiensa puolesta soveltua myös erityisen hyvin pienille yrityksille, joilla ei ole varaa ylläpitää erillistä ohjelmistokehitysyksikköä, jossa ohjelmistokehityksen erityisosaajat huolehtisivat yrityksen ohjelmistokehitystarpeista, vaan yrityksen ohjelmistokehitysvastuu jakautuu yrityksessä vähemmän erikoistuneille monialaisille työntekijöille. Tältä kannalta on kuitenkin hieman vaistonvastaista, että vähäisen koodin kehitysalustojen alkupään kustannukset ovat perinteisiä kehitysympäristöjä ja -alustoja huomattavasti suuremmat. Joka tapauksessa, vähäisen koodin kehitysympäristöt pystyvät tuomaan erikokoisille yrityksille tietyissä tilanteissa huomattavat säästöt ja tehostamaan ohjelmistokehitystä, joten vähäisen koodin kehitysympäristöille on tarvetta. Vähäisen koodin

kehitysympäristöt soveltuvat kuitenkin vain niille tarkoitettuihin tilanteisiin, joten ne eivät kykene täysin syrjäyttämään perinteisiä kehitysympäristöjä ainakaan nykyisellään.

8. YHTEENVETO

Tämän työn tarkoitus oli vertailla perinteisiä kehitysympäristöjä ja vähäisen koodin kehitysympäristöjä sekä pohtia, että onko jälkimmäisille tarvetta. Tämä vertailu tehtiin kehitysympäristötyyppien perusominaisuuksista, lisensoinnista, kustannustehokkuudesta sekä ketteryydestä ja samalla havaittiin vähäisen koodin kehitysympäristöille olevan tarvetta. Eri osa-alueisiin ei uppouduttu kovin syvällisesti, mutta tutkimuskysymyksen kannalta laaja-alaisempi pinnallinen tarkastelu tuotti paremman lopputuloksen.

Varsinainen työ lähti liikkeelle vähäisen koodin kehityksen historiasta, jonka yhteydessä nähtiin vähäisen koodin kehityksen olevan jatkumoa pidempään jatkuneille ponnisteluille tehostaa ja helpottaa ohjelmistokehitystä. Tästä jatkettiin kehitysympäristötyyppien eroihin, joista merkittävimpinä nostettiin esiin perinteisten kehitysympäristöjen kykenevän tuottamaan hyvin erilaisia sovelluksia tekstipohjaisella lähdekoodin tuottamisella siinä missä vähäisen koodin kehitysympäristöt jakautuvat kapeammille sovelluskehityksen aloille ja tuottavat sovelluksia enemmän visuaalisin työkaluin. Luvussa 4 vertailtiin lisensointia ja pohdittiin tekijänoikeudellisia seikkoja. Lisenssejä vertailtaessa huomattiin vähäisen koodin kehitysympäristöjen vaikuttavan huomattavasti kalliimmilta, mutta siirryttäessä lukuun 5, eli kustannustehokkuus, havaittiin vähäisen koodin kehityksen kuitenkin mahdollisesti tulevan edullisemmaksi kun huomioidaan sovelluskehityksen parantunut tehokkuus sekä vähentynyt tarve ohjelmistotuotantoon erikoistuneille työntekijöille. Lopuksi tarkasteltiin kehitysympäristötyyppejä niiden ketteryyden kannalta. Tässä ketteryys jaettiin kolmeen osa-alueeseen: taipuisuuteen, nopeuteen ja korvattavuuteen. Perinteiset kehitysympäristöt havaittiin taipuisammiksi, vaikkakin vähäisen koodin kehitysympäristöt olivat myös yllättävän taipuisia. Jälkimmäiset olivat myös nopeampia, siinä missä perinteiset kehitysympäristöt olivat helpommin korvattavissa.

Tätä työtä tehdessä nousi esille, että vähäisen koodin kehitysympäristöt ovat ennalta määrätyn luonteensa vuoksi aina rajoittuneita tietyntyyppisiin sovelluksiin. Vähäisen koodin kehityksellä ei voisi siis saada aikaan aivan mitä tahansa, ei ainakaan ilman toiminnallisuuden laajentamista applikaatiorajapinnan kautta mahdollisesti tukeutuen perinteisiin kehitysympäristöihin. Tämän rajoituksen poistaminen tekisi vähäisen koodin kehitysympäristöistä entistä parempia työkaluja, mutta valmiin elementin luominen jokaiseen tilanteeseen mitä asiakaskehittäjä saattaa keksiä olisi käytännössä mahdotonta. Olisi siis mielenkiintoista tutkia, olisiko mahdollista hyödyntää tekoälyä jollakin tavalla, esimerkiksi kääntämään yksinkertaiset ihmiskieliset kuvaukset ohjelmaelementtien lähdekoodeiksi.

LÄHTEET

1. SDTimes, 2014, How low-code development seeks to accelerate software delivery, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://sdtimes.com/application-development/low-code-development-seeks-accelerate-software-delivery/

2. Naeem Ahmad Sattar, 2018, Selection Of Low-Code Platforms Based On Organization And Application Type, Lappeenranta University Of Technology

3. Mariana Bexiga et. al., 2020, Closing the Gap Between Designers and Developers in a Low Code Ecosystem, ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems

4. Apurvanand Sahay et. al., 2020, Supporting the understanding and comparison of low-code development platforms, Universit`a degli Studi dell'Aquila

5. Information Age, 2018, 4 requirements of a Low-Code development platform, Verkkolähde, Saatavissa (viitattu 10.2.2020): http://www.information-age.com/4-requirements-Low-Codedevelopment-platform-123469520/

6. Forbes, 2018, Is Outsystems Portugal's Latest Unicorn, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.forbes.com/sites/forbesinternational/2018/07/13/is-outsystems-portugals-latest-unicorn/#6e41ac2c3633

7. Traction on Demand, 2019, The Purpose of Dell Boomi Lives in the Company Name: TractionForce 2019 Sponsor Highlight, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://tractionondemand.com/blog/dell-boomi-tractionforce-sponsor/

8. TechRepublic, 2020, COVID-19 triggering a massive shift in adoption of low-code platforms, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.techrepublic.com/article/covid-19-triggering-a-massive-shift-in-adoption-of-low-code-platforms/

9. Outsystems, 2019, What Is Visual Programming, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.outsystems.com/blog/posts/what-is-visual-programming/

10. Macintosh Repository, 2016, Double Helix, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.macintoshrepository.org/7324-double-helix

11. Clay Richardson & John R. Rymer, 2016, Vendor Landscape: The Fractured Fertile Terrain of Low-Code Application Platforms, Forrester

12. Emma Van Pelt, 2019, Large Enterprises Succeeding With Low-Code, Forrester

13. Tekijänoikeuslaki 2018/849. Annettu Helsingissä 12.11.2018. Saatavissa (viitattu 07.02.2020) https://www.finlex.fi/fi/laki/ajantasa/1961/19610404#L1P1

14. Claudia de O. Melo et. al., 2017, A Method for Evaluating End-User Development Technologies, Organizational Transformation & Information Systems (SIGORSA)

15. Microsoft, 2021, Buy Visual Studio, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://visualstudio.microsoft.com/vs/pricing/

16. Eclipse, 2017, Eclipse Foundation Legal Frequently Asked Questions (FAQ), Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.eclipse.org/legal/legalfaq.php

17. Notepad++, 2021, What is Notepad++, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://notepad-plus-plus.org/

18. Paul B. de Laat, 2005, Copyright or Copyleft? An Analysis of Property Regimes for Software Development, University of Groningen

19. Daniel B. Ravicher & Aaron K. Williamson, 1995, Free Software Foundation Inc. V Cisco Systems Inc. complaint, United States District Court Southern District of New York

20. Free Software Foundation, 2009, FSF Settles Suit Against Cisco, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.fsf.org/news/2009-05-cisco-settlement.html

21. Salesforce, 2021, Platform Pricing, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.salesforce.com/editions-pricing/platform/

22. Appian, 2021, Low-code Automation Platform Pricing, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.appian.com/pricing/

23. Microsoft, 2021, Power Apps pricing, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://powerapps.microsoft.com/en-us/pricing/

24. Outsystems, 2021, Pricing and Editions, Verkkolähde, Saatavissa (viitattu 03.03.2021): https://www.outsystems.com/pricing-and-editions/

25. Jonathan Lipsitz & Jon Erickson, 2019, The Total Economic Impact Of Power Apps And Power Automate, Forrester

26. CareerExplorer, 2018, Software engineer salary, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://www.careerexplorer.com/careers/software-engineer/salary/

27. Cecilie Ness & Marita Eltvik Hansen, 2019, Potential of Low-Code in the healthcare sector, Norwegian School of Economics

28. Raquel Sanchis et. al., 2019, Low-Code as enabler of digital transformation in manufacturing industry, Applied Sciences

29. Kielitoimiston sanakirja, 2021, Ketterä, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://www.kielitoimistonsanakirja.fi/#/ketter%C3%A4

30. Kielitoimiston sanakirja, 2021, Joustava, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://www.kielitoimistonsanakirja.fi/#/joustava

31. Oracle, 2020, Moved By Java Timeline, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://www.oracle.com/java/moved-by-java/timeline/

32. Statistics Times, 2020, Top Computer Languages, Verkkolähde, Saatavissa (viitattu 25.03.2021): http://statisticstimes.com/tech/top-computer-languages.php

33. Statista, 2016, Number of IoT devices 2015-2025, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

34. JAXenter, 2014, Developing NASA's mission software with Java, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://jaxenter.com/netbeans/developing-nasas-mission-software-with-java

35. Minecraft, 2020, Minecraft Java Edition, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://www.minecraft.net/en-us/store/minecraft-java-edition/

36. Outsystems, 2020, Extend Logic with Your Own Code, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/Extend_Logic_with_Your_Own_Code

37. Appian, 2021, Extending Appian, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://docs.appian.com/suite/help/20.4/extending-appian.html

38. Salesforce, 2021, Platform Development Basics, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://trailhead.salesforce.com/content/learn/modules/platform_dev_basics

39. Microsoft, 2021, Connectors, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://docs.microsoft.com/en-us/connectors/connectors

40. Fraktio, 2020, Selkeytä työtäsi prototyypeillä – se on helppoa, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://www.fraktio.fi/blogi/selkeyta-tyotasi-prototyypeilla-se-on-helppoa

41. HotGloo, 2021, HotGloo, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://www.hotgloo.com/

42. Pidoco, 2021, Picodo – Online Wireframe and UX Prototyping Tool, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://pidoco.com/en

43. Outsystems, 2020, Sharing a Project, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://success.outsystems.com/Support/Forge_Components/Forge_FAQs/Sharing_a_Project

44. Appian, 2021, Import and Export Applications, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://docs.appian.com/suite/help/19.2/import_export_applications.html

45. Appian, 2021, Package Structure, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://docs.appian.com/suite/help/20.4/reference-structure.html

46. Salesforce, 2021, Ant Migration Tool, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://developer.salesforce.com/docs/atlas.en-us.daas.meta/daas/meta_development.htm

47. Microsoft, 2020, Work with solutions using the SDK APIs, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://docs.microsoft.com/en-us/power-platform/alm/solution-api#create-an-unmanaged-solution

48. Microsoft, 2021, About table/entity reference for Microsoft Dataverse, Verkkolähde, Saatavissa (viitattu 25.03.2021): https://docs.microsoft.com/en-us/powerapps/developer/data-platform/reference/about-entity-reference