Pro gradu -tutkielma

Tietojenkäsittelytiede

Uudelleenkäytön odotuksien toteutuminen internetaikakauden ohjelmistokehityksessä

Arto Kiviluoto

17.1.2020

Matemaattis-luonnontieteellinen tiedekunta

Helsingin yliopisto

Ohjaaja(t)

toht. N. Mäkitalo, prof. T. Mikkonen

Tarkastaja(t)

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
Matemaattis-luonnontieteellinen tiedekunta	Tietojenkäsittelytiede

Tekijä — Författare — Author

Arto Kiviluoto

Työn nimi — Arbetets titel — Title

Uudelleenkäytön odotuksien toteutuminen internetaikakauden ohjelmistokehityksessä

Ohjaajat — Handledare — Supervisors

toht. N. Mäkitalo, prof. T. Mikkonen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Pro gradu -tutkielma	17.1.2020	62 sivua

Tiivistelmä — Referat — Abstract

Uudelleenkäytöllä ohjelmistokehityksessä tarkoitetaan aiemmin kehitettyjen artefaktien hyödyntämistä uudessa kohteessa. Artefaktit voivat olla käsitteellisiä, kuten suunnittelupäätökset, tai konkreettisia, kuten ohjelmakoodi, ja niitä voidaan hyödyntää ohjelmistotuotantoprosessin eri vaiheissa.

Uudelleenkäyttöä tutkittiin jo vuosikymmeniä sitten, ennen internetin yleistymistä, mutta sen ei koettu olevan muodostunut tavanomaiseksi käytännöksi. Tutkielman tavoitteena on koostaa ennen internetaikakautta tunnistettuja uudelleenkäytön muotoja ja niille asetettuja odotuksia, sekä arvioida niiden toteutumista nykypäivän uudelleenkäytössä.

Tässä tutkielmassa lähestytään uudelleenkäytön muotoja ja niihin kohdistettuja odotuksia sekä niiden toteutumista kirjallisuuskatsauksena. Tutkielma on kaksiosainen ja osien jakajana toimii internetin yleistymisen ajankohta, joka tässä tutkielmassa on ajoitettu vuosituhannen vaihteeseen. Nykypäivän uudelleenkäytön menetelmien tarkastelun lähtökohtana toimi Charles W. Kruegerin käyttämä nelikko: abstrahointi, valinta, erikoistaminen ja integrointi.

Tutkielmasta ilmenee, että uudelleenkäytön asema ohjelmistokehityksessä on vakiintunut. Monet nykypäivän uudelleenkäytön menetelmät vastaavat hyvin historiassa esitettyjä menetelmiä ja odotukset uudelleenkäytön menestymisestä näyttävät toteutuneen. Vaikka uudelleenkäyttö on hyväksytty vakiintuneeksi käytännöksi, ei sen hyödyistä näytä olevan luotettavaa näyttöä vahvistamaan kaikkia sille asetettuja odotuksia. Tutkielmassa havaitaan myös, että internetin myötä ohjelmistokehitys on kokenut suuria muutoksia, joiden vaikutuksia menneinä vuosikymmeninä tuskin voitiin ennakoida.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Designing software → Software implementation planning → Software design techniques
Software and its engineering → Software creation and management → Software development techniques → Reusability

Avainsanat — Nyckelord — Keywords

uudelleenkäyttö, ohjelmistokehitys

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Ohjelmistojärjestelmien erikoistumislinja

Sisältö

1 Johdanto **1**

2 Internetaikakautta edeltävä uudelleenkäyttö **3**

2.1 Mitä uudelleenkäytöstä ajateltiin 3

2.2 Mitä muotoja uudelleenkäytöstä löydettiin 5

2.2.1 Klassiseen ohjelmointiin keskittyneet muodot 7

2.2.2 Klassisesta ohjelmoinnista eteenpäin 10

2.2.3 Muuhun, kuin ohjelmointiin keskittyneet menetelmät 12

2.2.4 Yhteenveto internetin yleistymistä edeltävistä uudelleenkäytön muodoista 14

2.3 Uudelleenkäytön oletettuja hyötyjä 15

3 Tutkimus **21**

3.1 Tutkimusmenetelmä 21

3.2 Tutkimuskysymykset 21

3.3 Tutkimuksen toteuttaminen 22

3.4 Näkökulma nykypäivän uudelleenkäytön muotoihin 23

3.4.1 Abstrahointi 23

3.4.2 Valinta 24

3.4.3 Erikoistaminen 25

3.4.4 Integrointi 26

4 Uudelleenkäyttö nykypäivänä **28**

4.1 Näkökulma nykypäiväisen uudelleenkäytön tutkimiseen 28

4.2 Uudelleenkäyttö määrittelyssä 29

4.3 Uudelleenkäyttö suunnittelun aikana 31

4.3.1 Arkkitehtuurit ja suunnittelumallit 31

4.3.2 Prosessit 33

4.4 Uudelleenkäyttö toteutuksessa 34

4.4.1 Nykypäivän yleiset ohjelmointikielet 34
4.4.2 Paketinhallintajärjestelmät 36
4.4.3 Uudelleenkäytettävät räätälöitävät elementit ja komponentit 38
4.4.4 Ohjelmistokehykset ja kirjastot 40
4.4.5 Alustariippumaton ohjelmistokehitys 42
4.4.6 Muita uudelleenkäytön menetelmiä toteutusvaiheessa 44
4.5 Uudelleenkäyttö käyttöönotossa 47
4.6 Uudelleenkäyttö ylläpidossa 48
4.7 Nykypäivän uudelleenkäytön muodot kootusti 49
4.8 Nykypäivän uudelleenkäytön vaikutuksia 51

5 Yhteenveto **55**

Kirjallisuus **59**

1 Johdanto

Uudelleenkäytöllä ohjelmistokehityksessä tarkoitetaan uusien ohjelmistojen tuottamista hyödyntäen aiemmin kehitettyjä osia, joiden alkuperäinen käyttötarkoitus voi olla toinen ohjelmistotuote tai uudelleenkäyttö. Uudelleenkäyttöä on tavalla tai toisella esiintynyt tietojenkäsittelyn syntymästä lähtien, minkä tekee luonnolliseksi ihmisten taipumus hyödyntää hyväksi havaittuja toimintatapoja ja toisaalta synnynnäinen halu tehdä asiat mahdollisimman helposti. Uudelleenkäytettävät osat voivat olla lähes mitä tahansa prosessimalleista ja arkkitehtuuripäätöksistä aina lähdekoodin osiin asti. Koska uudelleenkäytettävät osat voivat olla luonteeltaan varsin erilaisia, voidaan niitä Kruegerin [19] tavoin kutsua yleisesti uudelleenkäytettäviksi *artefakteiksi*.

Uudelleenkäytön odotettiin tuovan helpotusta ohjelmistokehityksessä vallitsevaan kriisiin jo tietojenkäsittelyn ensimmäisten vuosikymmenten aikana [19]. Tällä kriisillä tarkoitettiin ohjelmistokehitystä vaivaavia ongelmia ja ohjelmistokehityksen tarpeen suurta lisääntymistä, joka oli havaittavissa jo ennen internetin yleistymistä [18]. Kriisi ilmeni muun muassa kustannusten kasvuna ja aikataulujen viivästymisenä, puutteina vaatimusten tyydyttämisessä, sekä saatavilla olevien ohjelmistoalan ammattilaisten riittämättömyytenä [18].

Uudelleenkäytön odotettiin olevan ratkaisu varsin merkittävään ongelmaan ohjelmistokehityksessä, minkä vuoksi on mielenkiintoista tutkia, miten vuosikymmeniä sitten esitetyt odotukset ovat käyneet toteen. Jonesin [16] arvion mukaan vuonna 1983 maailmassa oli noin 3,25 miljoonaa ammattimaista ohjelmistokehittäjää ja tänä päivänä ohjelmistokehittäjien määrä on arviolta yli 20 miljoonaa[1,2], joista suurin osa tekee ohjelmistokehitystä ammattimaisesti. Arvioiden mukaan ohjelmistokehittäjien määrä on yhä kasvussa ja sa-

[1]Evans Data Corporation — Developer Population Growth Shifts Toward China, India and Emerging Countries — Press Release, https://evansdata.com/press/viewRelease.php?pressID=268, luettu 28.12.2019

[2]IDC's Worldwide Developer Census, 2018: Part-Time Developers Lead the Expansion of the Global Developer Population, https://www.idc.com/getdoc.jsp?containerId=US44363318, luettu 28.12.2019

malla mediassa on esillä pula ohjelmistokehittäjistä[3,4].

Tutkielma lähestyy aihetta esittelemällä teoriapohjaa uudelleenkäytöstä ja konkretisoimalla sitä uudelleenkäytön menetelmillä, joita tunnistettiin kirjallisuudessa internetin yleistymistä edeltävinä vuosikymmeninä. Konkreettisten esimerkkien kautta käydään läpi uudelleenkäytön tuomia todettuja, sekä odotettuja hyötyjä. Näitä hyötyjä peilataan nykypäivänä esiintyviin uudelleenkäytön muotoihin ja analysoidaan, kuinka vuosikymmenten saatossa hyödyt ovat realisoituneet ja muuttuneet. Nykypäivän uudelleenkäytön muotojen tunnistamisen lisäksi tutkitaan, miten uudelleenkäyttö on kehittynyt ja kuinka se vastaa alkuperäisiä ajatuksia.

Toisessa luvussa käsitellään uudelleenkäytön menetelmiä, joita oli tunnistettu vuosikymmeninä ennen internetin syntyä ja yleistymistä. Luvussa esitellään uudelleenkäytön muotoja, niistä saatavia todettuja ja odotettuja hyötyjä, haittoja ja odotuksia uudelleenkäytön tulevaisuudesta.

Kolmannessa luvussa esitellään tutkielman taustoja ja tekoprosessia. Luvussa käydään läpi tutkimuskysymykset, joihin tämä tutkielma pyrkii vastaamaan, sekä menetelmä, joilla vastaukset on saatu. Luvussa käydään läpi myös ennen internetaikakautta kirjoitetussa aineistossa esiin nousseita uudelleenkäytettävien artefaktien ominaisuuksia, joiden avulla nykypäivän uudelleenkäytön muotoja tarkastellaan. Nämä ominaisuudet ovat abstrahointi, valinta, erikoistaminen ja integrointi.

Neljännessä luvussa käydään läpi nykypäivän ohjelmistokehitystä uudelleenkäytön näkökulmasta. Luvussa tutkitaan, missä muodoissa uudelleenkäyttö nykypäivänä esiintyy eri ohjelmistokehitysprosessin vaiheissa ja kuinka yleistä se on. Nykypäivän muotoja peilataan internetaikakautta edeltävän ajan uudelleenkäyttöön esittelemällä yhtäläisyyksiä ja eroavaisuuksia.

Viidennessä luvussa nostetaan esiin keskeisimmät tutkimustulokset ja pohditaan tutkielman onnistumista sekä tulosten luotettavuutta. Luvussa ehdotetaan myös mahdollisia uudelleenkäyttöä käsitteleviä jatkotutkimusaiheita.

[3] Koodareista on Suomessa huutava pula – helpotusta haetaan sieltä, missä pippuri kasvaa, https://www.kaleva.fi/uutiset/talous/koodareista-on-suomessa-huutava-pula-helpotusta-haetaan-sielta-missa-pippuri-kasvaa/805268/, luettu 28.1.2019

[4] Jopa 10 000 työpaikkaa koodareille, mutta tekijät puuttuvat – ”Vaatii kaikkien osapuolten aktivoitumista”, https://yle.fi/uutiset/3-10669492, luettu 28.12.2019

2 Internetaikakautta edeltävä uudelleenkäyttö

Tietojenkäsittely oli ehtinyt kehittyä ja yleistyä joitain vuosikymmeniä ennen vuotta 1983, jolloin uudelleenkäyttö nousi keskeiseksi puheenaiheeksi ja tutkimuksen kohteeksi. Tuolloin Yhdysvalloissa järjestettiin konferenssi, jonka keskeinen teema oli uudelleenkäyttö [16]. Ohjelmistoalaa vaivasivat moninaiset ongelmat, kuten pitämättömät aikataulut, ohjelmistokehityksen alati kasvavat kustannukset, tuotetun ohjelmiston heikkolaatuisuus ja riittämättömät henkilöstöresurssit, jotka liittyivät vahvasti myös ohjelmistokriisiksi kutsuttuun ilmiöön [18]. Kiinnostus uudelleenkäyttöä kohtaan oli suuri, sillä sen odotettiin tuovan ohjelmistokehitykseen mullistavia hyötyjä ja siten jopa ratkaisevan alaa vaivaavat ongelmat [19, 18, 14]. Tässä luvussa syvennytään ennen internetin yleistymistä esitettyihin näkemyksiin ja kokemuksiin uudelleenkäytöstä.

2.1 Mitä uudelleenkäytöstä ajateltiin

Uudelleenkäytöllä on uskottu olevan merkittäviä hyötyjä uutta ohjelmistotuotetta koskeviin resurssihaasteisiin, mitä tukee Lanerganin ja Grasson [21] raportoima keskimäärin 60% uudelleenkäyttöaste valmistuneissa ohjelmistotuotteissa. Raportoidut uudelleenkäyttöasteet koskivat Raytheon-nimisen yrityksen omia ohjelmistoja, ja tarkoittavat sitä, että 60% uudesta ohjelmistotuotteesta sisälsi jo aiemmin jotain muuta käyttötarkoitusta varten kehitettyjä ohjelmistojen osia [21, 19].

Lanerganin ja Grasson [21] tapauksessa on tosin syytä huomioida, että uudelleenkäyttö koski yksittäisen yrityksen sisäisiä ohjelmistotuotteita, jolloin synergiaetuja voi olla paljon, eivätkä tulokset ole suoraan yleistettävissä. Jonesin [16] arvion mukaan 1980-luvun alussa mahdollisesti vain 15% sovelluskoodista oli yksittäiseen sovellukseen erikoistunutta ja loput yleisluontoisempaa eli uudelleenkäytettävää. Jonesin [16] tulokset viittaisivat siis siihen, että Lanerganin ja Grasson [21] ilmoittamat luvut eivät ole täysin sidoksissa yksittäisen yrityksen sisäisiin ohjelmistotuotteisiin. Hänen arvionsa mukaan jopa 85% uudelleenkäyttöaste olisi uudelleenkäytön tutkijoiden tavoitetila.

Krueger [19] kuvasi uudelleenkäyttöä työkaluna *kognitiivisen etäisyyden* (engl. cogniti-

ve distance) lyhentämiseksi ohjelmistotuotannossa. Kognitiivisella etäisyydellä tarkoitetaan ajatustyön määrää, joka vaaditaan ohjelmiston viemiseen yhdestä vaiheesta seuraavaan [19]. Eri uudelleenkäytön muodoilla on erilainen suhde ja vaikutus kognitiiviseen etäisyyteen, mutta uudelleenkäytettävyyden kuvaajana se toimii varsin hyvin.

Loven [22] mukaan 1980-luvun lopulla uudelleenkäyttö ei ollut yleistynyt ohjelmistokehityksen työkaluna siitä syystä, että uuden ohjelmakoodin kirjoittaminen oli helpompaa, kuin uudelleenkäytettävän koodikomponentin hyödyntäminen. Hyödyntämistä vaikeuttivat Loven [22] mukaan lähinnä tekniset rajoitteet, eikä kyse ollut niinkään asenteista uudelleenkäyttöä kohtaan. Ratkaisuna näihin teknisiin ongelmiin hän näki oliot (engl. object), tai olio-ohjelmoinnin (engl. object-oriented programming), joiden käytön yleistyminen johtaisi uudelleenkäytön kynnyksen madaltumiseen. Syy siihen, miksi oliot mahdollistavat uudelleenkäytön, on niiden merkittävä vaikutus sovelluksen rakenteeseen ja se, että olioissa tiedon käsittely on tyyppisidonnaisempaa [22].

Hieman ristiriitaisesti vain muutamaa vuotta aiemmin Lanergan ja Grasso [21] esittivät, että ohjelmistokehitystä vaivasi ajatusmaailma, jossa liiketoimintaa tukevien sovellusten koettiin tyypillisesti olevan niin uniikkeja, että ainoana vaihtoehtona nähtiin niiden kehittäminen täysin alusta alkaen. Liiketoimintaa tukevissa sovelluksissa oli heidän mukaansa vain kuusi keskeistä mahdollista toimintoa: datan järjestäminen, muokkaaminen, yhdistäminen, tuhoaminen, päivittäminen tai sen raportointi. Vaikka tietojenkäsittely onkin syntyjään varsin datalähtöistä, ja Lanerganin ja Grasson [21] tiivistys liiketoimintaa tukevista toiminnoista on useimpien sovelluksien ytimessä, vaikuttaa esitetty näkemys nykypäivänä varsin kapeakatseiselta.

Coomer ja kumppanit [5] puolestaan totesivat, että ulkoistetuissa ohjelmistoprojekteissa uudelleenkäytettävien komponenttien kehittäminen olisi ohjelmistoyrityksille epätaloudellista, sillä ne saattaisivat vähentää tulevaisuudessa tilattavien töiden tuomaa kassavirtaa, eivätkä yritykset välttämättä ole halukkaita käyttämään toisten ohjelmistoyritysten kehittämiä ratkaisuja. He totesivat myös, että kyseisissä ohjelmistoprojekteissa työskentelevät projektipäälliköt eivät välttämättä halua ottaa kontolleen uudelleenkäytettävien artefaktien kehittämisen tuomia mahdollisia lisäkustannuksia, mikäli ne eivät suoraan hyödyttäisi kyseistä projektia. Uudelleenkäytön menestystä vaikutti tuolloin hankaloittavan siis sekä asenteelliset että teknologiset rasitteet.

2.2 Mitä muotoja uudelleenkäytöstä löydettiin

Ensimmäisinä vuosikymmeninään tietojenkäsittely ei ollut saavuttanut samanlaista läsnäoloa tai monimuotoisuutta, millaisena se tällä hetkellä esiintyy. Siitä syystä 1990-luvulla ja sitä ennen tehdyt tutkimukset uudelleenkäytön saralla eivät kykene kattamaan kaikkia kehityksen tuomia mahdollisuuksia. Kyseisiä tutkimuksia läpikäymällä käy kuitenkin varsin pian selväksi, että tuolloin uudelleenkäytön näkökulmasta tutkitut aiheet ovat vielä tänäkin päivänä varsin osuvia.

Uudelleenkäyttöä pyrittiin hyödyntämään erilaisissa artefakteissa abstrakteista konkreettisiin. Abstrakteimmat artefaktit olivat muun muassa erilaisia suunnittelumalleja, joissa toteutustason yksityiskohtiin otetaan kantaa varsin karkealla tasolla ja pääosassa ovat loogiset kokonaisuudet [19, 16]. Konkreettisimpia artefakteja taas edustavat lähellä fyysisen laitteistotason komponentteja olevat suorituksen aikana hyödynnetyt konekäskyt [19]. Painotus uudelleenkäytön tutkimisessa ei kuitenkaan aineiston perusteella aluksi ollut kovinkaan keskittynyt abstraktin ja konkreettisen välisen skaalan ääripäihin, vaan sovelluskehittäjän kirjoittamaan sovelluskoodiin – erityisesti korkean tason ohjelmakoodiin.

Ohjelmakoodiin painottumisessa mielenkiintoista on se, että Kim ja Stohr [18] toteavat Freemaniin (Freeman, 1987) viitaten ohjelmoinnin osuuden olevan vain pieni osa ohjelmiston tuottamisen kokonaiskustannuksista. Jos suurin osa uudelleenkäytön tutkituista hyödyistä tapahtuu lähinnä alueella, joka kattaa kokonaiskustannuksista vain kolmanneksen, herää kysymys, kuinka merkittävää uudelleenkäyttö näissä muodoissa on? Muutamat esitetyt uudelleenkäytön menetelmät toisaalta mahdollistivat ohjelmistokehitykseen muutoksia esimerkiksi siinä, kuka ohjelmistokehittäjä oikeastaan on, kun tietyt menetelmät pyrkivät abstrahoimaan ohjelmakoodia niin pitkälle, että teknistä osaamista ei juurikaan enää tarvita [13]. Ohjelmiston kehittäjän tarvitsisi abstrahoinnin myötä tuntea ohjelmiston vaatimukset loogisella tasolla ja lopullinen ohjelma voitaisiin tuottaa koneellisesti loogisten määritysten perusteella. On siis myös mahdollista, että uudelleenkäytön avulla koko ohjelmistokehitysprosessi saattaisi kokea rakenteellisia muutoksia, jotka heijastuvat myös kustannusrakenteeseen.

Alla on Horowitzin ja Munsonin [14] tutkimuksessa esitetty taulukko 2.1, jossa ohjelmiston kehitysvaihe on jaettu osiin. Jokaiselle osalle on annettu prosentuaalinen arvo kuvaamaan sen osuutta kokonaiskustannuksista ja kirjoittajien arvio lähitulevaisuuden tehostamismahdollisuuksista, sekä lopullinen kustannus tehostamisen jälkeen. Ohjelmointi yhdistettynä koodin testaamiseen kattaa myös Horowitzin ja Munsonin mukaan karkeas-

ti kolmanneksen koko ohjelmiston kehitysvaiheen kustannuksista, mutta heidän arvionsa mukaan suurimmat mahdollisuudet tehokkuuden parantamiseen yksittäisillä alueilla (yli 50%) sijaitsevat juuri tällä sektorilla. Tämä huomioituna on loogista, että juuri tähän sektoriin panostetaan uudelleenkäytön tutkimisessa. Kaikki tehostamiset huomioon ottaen lopullinen vaikutus kokonaiskustannukseen olisi sen pienentyminen 60%:iin alkuperäisestä. Huomioitavaa taulukossa on, että järjestelmään kohdistuvissa vaatimuksissa ei nähty mahdollisuuksia parannukselle, mikä myöhemmässä osiossa osoittautuu puutteelliseksi oletukseksi.

	Kustannus	Parannus %	Loppukustannus
Järjestelmävaatimukset	2	0	2
Laitteistovaatimukset	8	25%	6
Ohjelmistovaatimukset	10	20%	8
Ohjelmistosuunnittelu	12	40%	7
Ohjelmointi	13	75%	3
Yksikkötestaus	24	50%	12
Integraatiotestaus	13	30%	9
Dokumentaatio	6	30%	4
Järjestelmätestaus	12	25%	9
	100	**40%**	**60**

Taulukko 2.1: Ohjelmistokehityksen vaiheita, kustannuksia ja mahdollisuuksia tehokkuuden parantamiseksi, suomennettu [14].

Taulukossa 2.2 Horowitz ja Munson [14] esittävät ohjelmiston kustannuksia sen elinkaaren aikana. Taulukossa edellä esitetyt ohjelmiston kehityksestä aiheutuvat kustannukset kattavat vain noin neljänneksen ohjelmiston kokonaiskustannuksista, jolloin tietyllä kehityksenaikaisella osa-alueella tehdyt säästöt muuttuvat suuremmassa kuvassa tarkasteltuna vähemmän merkityksellisiksi. Kuten myöhemminkin todetaan, eri osiot eivät ole kuitenkaan täysin irrallisia toisistaan. Esimerkiksi tietyt panostukset kehityksen aikana voivat heijastua muun muassa ylläpidon alla olevaan vikojen poistamiseen.

Vaihe	Kustannus	Parannus %	Loppukustannus
Järjestelmän kehitys	100	40%	60
Asennus	15	20%	12
Ylläpito			
Vian poisto	60	75%	15
Ympäristömuutokset	60	30%	42
Parannukset	180	40%	108
	415	**43%**	**237**

Taulukko 2.2: Ohjelmiston kustannukset sen elinkaaren aikana, suomennettu [14].

Ohjelmiston kehittämiseen keskittyneitä uudelleenkäytön muotoja on kuvattu kattavasti Kruegerin [19] tutkimuksessa, jossa hän tunnisti kahdeksan kategoriaa, joihin uudelleenkäyttöä voidaan jakaa:

- korkean tason ohjelmointikielet
- mallien ja koodin kaivelu
- lähdekoodikomponentit
- ohjelmistoskeemat
- sovellusgeneraattorit
- erittäin korkean tason ohjelmointikielet
- muunnosjärjestelmät
- ohjelmistoarkkitehtuurit

Kategorioihin syvennytään seuraavissa alaluvuissa ja niiden lisäksi esitellään muutamia muita aineistossa esiintyneitä uudelleenkäytön kannalta oleellisia artefakteja.

2.2.1 Klassiseen ohjelmointiin keskittyneet muodot

Ensimmäisten havaittujen uudelleenkäytön hyötyjen joukossa oli Kruegerin [19] mukaan ohjelmoinnissa tapahtunut siirto symbolisesta konekielestä (Assembly) *korkean tason ohjelmointikieliin*. Symboliset konekielet ovat lähellä suorittimien ymmärtämää konekieltä,

mutta sovelluskehittäjälle symbolinen konekieli ei välttämättä ole ohjelmakoodin kirjoittamisen kannalta ihanteellisin ja tehokkain ohjelmointikieli. Symbolinen konekieli perustuu vahvasti ulkoa muistettaviin komentoihin, eikä niiden ilmaisu ole Kruegerin mukaan ole yhtä luonnollista, kuin korkean tason ohjelmointikielissä. Hän toteaa, että tutkimalla yleisesti käytettyjä käskyjä ja näiden yhteyksiä toisiinsa voitiin muodostaa kokoelmia ja toimintakaavoja, jotka toimivat pohjana korkean tason ohjelmointikielten kehittämiselle. Krueger viittaa Brooksiin [Brooks, 1975], jonka mukaan siirryttäessä symbolisista konekielistä korkean tason ohjelmointikieliin, havaittiin ohjelmakoodin kirjoittamisessa jopa viisinkertainen nopeutus. Näin uudelleenkäytön avulla syntyi ohjelmointikielten luokka, joka muodostui ohjelmoinnin ja uudelleenkäytön uudeksi "pohjatasoksi". Krueger toteaa, että symbolisten konekielten toistuvuuksien hyödyntämistä ei usein mielletä uudelleenkäytöksi, mutta verrattaessa niitä tapoihin, joilla korkean tason kieliä abstrahoidaan uudelleenkäytön nimissä, voidaan niiden välillä nähdä vahva yhtäläisyys.

Korkean tason ohjelmointikielten rakenteiden hyödyntämisellä eri mittakaavoissa ja eri tavoilla saavutetaan monenlaisia uudelleenkäytettäviä artefakteja [19, 16]. Luonnollinen jatkumo symbolisten konekielten hyödyntämisestä korkean tason ohjelmointikielten kehitykseen on käyttää ohjelmakoodissa korkean tason kielissä esiintyviä rakenteita ja toistuvuuksia uudelleenkäytettävinä artefakteina. Ohjelmointikielissä on valmiina sisällytetty kontrollirakenteita, kuten esimerkiksi *if* ja *for*, jotka kuuluvat ohjelmistokehityksessä arkisimpaan työkalustoon, mutta kuten Joneskin [16] totesi saman tarpeen täyttäviä koodilohkoja, moduuleita, luokkia ynnä muita *lähdekoodikomponentteja* saattaa toistua monissa rinnakkaisissa sovelluksissa uniikkeina toteutuksina. Nämä koodilohkot voivat esimerkiksi toteuttaa jonkin matemaattisen laskutoimituksen tai käsitellä merkkijonoja.

Useassa tutkimuksessa todettiin, että vahvasti formaaleihin määritelmiin perustuville toimialoille, kuten matematiikkaa hyödyntäville, kehitetyt ohjelmistot ovat omiaan edellä mainitulle lähdekoodikomponentteihin perustuvalle uudelleenkäytölle. Tarkat määritelmät johtavat siihen, että ohjelmistosta, toteutuksesta ja ympäristöstä riippumatta samalla syötteellä tuloksen tulisi aina perustua samoihin lainalaisuuksiin ja täten olla sama [19, 16, 21, 14]. Matemaattiset uudelleenkäytettävät artefaktit toimivat hyvänä esimerkkinä myös sen takia, että niiden tarvetta voi esiintyä lähes minkä toimialan ohjelmistoissa tahansa; esimerkiksi teknologian, talouden, maantieteiden ja ilmastotietotieteiden.

Kruegerin [19] mukaan matemaattisen artefaktin abstraktion taso voi olla hyvinkin korkea, sillä käyttäjän ei välttämättä tarvitse tietää artefaktista enempää, kuin sen edustaman konseptin idea ja nimi. Kun konsepti on ennalta tuttu, nimen kaltaisen hyvin korkean

tason tunnisteen avulla voi tarkasti ymmärtää miten ja mihin artefaktia voi hyödyntää. Artefaktin valinta voi siis parhaimmillaan olla erittäin helppoa ja kognitiivinen etäisyys on varsin lyhyt vaaditun työn määrän ollessa käsitteen ja toteutuksen välillä todella pieni.

Horowitzin ja Munsonin [14] mukaan suuren määrän tarkkaan määriteltyjä käsitteitä sisältävien toimialojen olemassaolo herätti McIlroyssa [McIlroy, 1968] toiveita siitä, että ohjelmistoalalle voisi syntyä kokonaan uusi haara, joka keskittyisi uudelleenkäytettävien koodikomponenttien valmistukseen. Ohjelmistokehittäjät olisivat siis jakautuneet kahteen joukkoon, joista toinen pyrkisi kehittämään mahdollisimman paljon laadukkaita ja tarpeellisia artefaktikirjastoja, joita toinen taas hyödyntäisi ohjelmistojen kehittämisessä. Tämänlaisia komponenttikirjastoja esitteli tutkimuksessaan muun muassa Jones [16], mutta lähtökohdat näille oli joko tietyn yrityksen sisäisen kehityksen tarpeet tai kapeahkot käyttökohteet, kuten matematiikka.

On olemassa myös organisoimattomampi tapa hyödyntää edellisen kaltaisia komponentteja, jota Krueger [19] kutsui *mallien ja koodin kaiveluksi* (engl. design and code scavenging). Koodin kaivelussa etsitään valmiista ohjelmakoodista osia, jotka voivat olla loogisia rakenteita ja kokonaisuuksia tai konkreettista ohjelmakoodia, joita voidaan hyödyntää uudessa käyttötarkoituksessa [19]. Nämä osat voisivat oikein abstrahoituina ja paketoituina olla uudelleenkäytettäviä artefakteja artefaktikirjastossa, mutta sellaisenaan ne ovat vain heikosti erottuvia osia suuressa kokonaisuudessa.

Erityisesti mallien kaivelussa korostuu kehittäjän oma kokemus ja ammattitaito [4]. Lähtökohtaisesti hyödynnettävissä artefakteissa ei ole mainittavaa abstraktiota ja valinta perustuu kehittäjän omiin muistikuviin artefaktin olemassaolosta [19]. Paikannus tapahtuu tilanteen mahdollistamilla keinoilla, kuten lähdekoodista etsimällä. Malleja, eli aiempia suunnittelupäätöksiä hyödyntämällä artefaktin konkretisointi muuttuu yhä vaikeammaksi, sillä mallit koostuvat Kruegerin [19] tulkinnassa laajoista koodikokonaisuuksista, joita kehittäjä joutuu muokkaamaan. Mallit ovat siis ohjelmakoodista implisiittisesti tulkittavissa ja abstraktion taso on täten heikko.

Valinta eri suunnittelupäätösten välillä voi olla varsin työläs tehtävä, mikäli vaihtoehtojen välinen vertailu vaatii paljon ohjelmakoodin analysointia. Mallien ja koodin kaivelusta tekee epätehokkaan uudelleenkäytön muodon se, että se perustuu vahvasti kokemukseen. Vaikka vietetty aika ohjelmistokehityksen parissa lisää kokemusta, liittyy siihen myös artefaktin lähteiden ja sovelluskohteiden vastaavuus, mikä ei välttämättä suoraan korreloi ajan kanssa. Tapaukset, joissa mallien ja lähdekoodin kaivelua voi hyödyntää, voivat siis olla varsin sattumanvaraisia.

Ammattimaiseen ohjelmistokehitykseen liittyy oleellisesti laadun varmistaminen, jolle yleinen ilmenemismuoto on ohjelmakoodin testaaminen. Koodilohkojen uudelleenkäytön yhtenä etuna on historiassa esitetty se, että mikäli lohkoa on käytetty jo aiemmin jonkin sovelluksen kehittämisessä on oletettavaa, että sen toiminnallisuus on jo aiemmin testattu, todettu virheettömäksi ja laadullisesti hyväksyttäväksi [19, 5]. Valmiiksi testatun koodin hyödyt ovat kaksivaikutteiset; sen lisäksi, että sovelluksen valmistumisen jälkeisten käytönaikaisten virheiden määrä vähenee, kehittäjiltä kuluu vähemmän aikaa kehityksen aikana ilmenevien toteutusvirheiden etsimiseen ja korjaamiseen. Virheiden etsiminen ja korjaaminen ovat Horowitzin ja Munsonin [14] mukaan voineet olla hyvin yleisiä ja pitkäkestoisia toimenpiteitä.

2.2.2 Klassisesta ohjelmoinnista eteenpäin

Korkean tason kielistä askel abstraktimpaan suuntaan on *sovellusgeneraattorit*. Sovellusgeneraattoreilla mahdollistetaan matalampi vaatimustaso sovelluksen kehittäjältä, jonka Horowitzin ja kumppaneiden [13] mukaan ei välttämättä tarvitse olla enää korkean tason ohjelmointikieliä ymmärtävä ohjelmoija.

Kruegerin [19] tutkimuksessa sovellusgeneraattoreilla tarkoitetaan työkaluja, joiden ulosanti on suorituskelpoinen, valmis sovellus. Sovellusgeneraattori ottaa syötteenään määrittämänsä syntaksin, abstraktion tason ja syöttömekanismin avulla käyttäjältä kuvauksen sovelluksen toimintalogiikasta ja tuottaa tämän perusteella toimivan sovelluksen. Koska toimintalogiikka ja syöttömekanismit voivat olla korkean abstrahoinnin tuloksia on mahdollista, että kehittäjän ei tarvitse olla tietoinen esimerkiksi sovelluksen kääntämisestä, toteutustason teknisestä arkkitehtuurista tai ohjelmakoodista, vaan ne ovat tältä piilossa [19]. Krueger kiteyttää, että sovellusgeneraattoria käyttäessään kehittäjälle on tärkeämpää vastata kysymykseen; *mitä* tarpeita sovelluksen tulisi tyydyttää. *Miten* sovellus kyseiset tarpeet täyttää perustuu sovellusgeneraattorin toteutuspäätöksiin.

Biggerstaffin ja Richterin [4] mukaan sovellusgeneraattoreissa on nähty ongelmalliseksi niiden rajallisuus toimialojen suhteen. Ohjelmistoilla voi olla varsin erilaisia tarpeita sovellettavasta toimialasta riippuen, eikä yksittäisen artefaktin ratkaisu ole välttämättä jokaiselle toimialalle tarpeellinen tai toimiva. Jotta ratkaisu olisi toimialalle soveltuva, se saattaa vaatia toimialakohtaista erikoistamista, mikä on Kruegerin [19] mukaan sovellusgeneraattoreissa hyvin yleistä. Erikoistaminen on johtanut siihen, että kehityksen päätteeksi sovellusgeneraattori on muodostunut hyvin toimialakohtaiseksi.

Kaikkien toimialojen huomioiminen on ongelma, jota voi tuskin koskaan luotettavasti väittää ratkaistuksi. Toimialojen kehittyminen tuo jatkuvasti uusia tarpeita ja jo pienikin määrä erilaisia toimialoja voi aiheuttaa vaatimuslistan kasvun niin suuriin kokoluokkiin, ettei kaikkien toteuttaminen ole järkevässä ajassa mahdollista.

Erittäin korkean tason ohjelmointikielet (engl. Very High Level Languages, VHLL), tai *neljännen sukupolven ohjelmointikielet* (engl. Fourth Generation Languages) Krueger [19] asettaa korkean tason ohjelmointikielten ja sovellusgeneraattoreiden välimaastoon. VHLL:t jatkavat korkean tason ohjelmointikielten ajatusta ohjelmointikielten abstrahoinnista ja tarjoavat ohjelmoijalle hyväksi havaittuja valmiita rakenteita, joiden avulla ohjelmakoodi kirjoitetaan. VHLL:n tyypillinen ero sovellusgeneraattoriin on hänen mukaansa se, että niiden tarjoamat rakenteet ovat yleisluontoisempia, eivätkä yhtä toimialaan sidottuja, kuten sovellusgeneraattorit tapaavat olla. Kääntöpuolena sille, että VHLL:t eivät ole rajoittuneet tietylle toimialalle on se, että niillä kehittäminen saattaa olla tehottomampaa juuri toimialaan heikomman sopivuuden vuoksi [19]. VHLL:t ovat siis uudelleenkäytön näkökulmasta melko ristiriitaisia, sillä tehokkuus on yksi merkittävä asia, mikä tekee uudelleenkäytöstä houkuttelevampaa.

SETL-nimistä VHLL:ää verrattiin korkean tason ohjelmointikieliin siten, että myös korkean tason ohjelmointikieliä kritisoitiin aluksi niiden tehottomuudesta symboliseen konekieleen verrattuna [19]. Tämän pohjalta Krueger [19] esitti ajatuksen, että myös VHLL:t voisivat muodostua valtavirran ohjelmointikieliksi, kuten korkean tason kieletkin alkoivat muodostua. Korkean tason kielien menestyksessä auttoivat optimoidut kääntäjät, kaiken aikaa tehokkaammaksi kehittyvä laitteisto ja erityisesti tuottavuuden kasvu, joka seurasi liiallisen teknisen haastavuuden ja ulkoa muistamisen määrän vähenemisestä [19]. Kaikki edellä mainitut seikat ovat yhtä lailla mahdollisia VHLL:ien tapauksessa, minkä perusteella usko niiden yleistymiseen on perusteltua.

Laajemman kirjon ohjelmointikielet (engl. Wide-Spectrum Language, WSL) ovat Kruegerin [19] esittelemä VHLL:ien erikoistapaus, joka yhdistäisi korkean tason ohjelmointikielten ja VHLL:ien hyvät puolet. Laajemman kirjon ohjelmointikielissä kehittäjällä olisi käytössään VHLL:n pitkälle abstrahoidut rakenteet, mutta tarpeen vaatiessa ohjelman suorituskykyä voisi parantaa käyttämällä korkean tason rakenteita ja siten väistää VHLL:n sudenkuoppia [19].

Muunnosjärjestelmät (engl. transformational systems) ovat VHLL:ien tapaan pitkälle abstrahoituja sovelluskehityksen työkaluja, joista Krueger [19] toteaa, että Zaven [Zave, 1984] mukaan kehittäjä keskittyy ensisijaisesti siihen, *mitä* lopullisen ohjelman tulisi tehdä.

Muunnosjärjestelmän vastuulla on *miten* se tapahtuu. VHLL:stä poiketen muunnosjärjestelmällä tehtyä korkean tason ”ohjelmakoodia” ei Kruegerin mukaan käännetä täysin koneellisesti, vaan alkuperäiseen koodiin kohdistetaan ihmisen ohjaamana muunnoksia, joilla koodi saadaan muunnos kerrallaan mahdollisimman tehokkaaseen muotoon. Viimeisten muunnosten jälkeinen lopullinen ohjelmisto vastaa korkean tason ohjelmointikielillä toteutetun ohjelmiston tehokkuutta. Toiminnallisesti nämä muutokset eivät aiheuta muutoksia, vaan koodi pysyy *semanttisesti identtisenä*, jolloin sovellukseen kohdistuvat muutostarpeet voidaan tehdä melko abstraktilla tasolla ja säästyä näin eri puolille koodikantaa hajautuneen toteutuksen muokkaamiselta [19, 2].

Vielä 1990-luvun alussa automaattiset muunnokset olivat Kruegerin [19] mukaan kääntäjille liian haastava ongelma, minkä takia muunnokset oli suoritettava ihmisavusteisesti. Ajatus siitä, että ajan saatossa kehittyvä laskentakyky mahdollistaisi automaattisten muunnosten tekemisen oli siis jo olemassa.

Ohjelmistoarkkitehtuureja hyödyntämällä artefakti koostuu hyvin laajamittaisesta kokonaisuudesta. Kruegerin [19] tutkimuksessa ohjelmistoarkkitehtuurilla tarkoitettiin ohjelmistokehyksiä, jotka pitivät sisällään laajoja rakenteita ja alijärjestelmiä. Tällä menetelmällä uudelleenkäytettävä artefakti koostuu useasta ennalta kehitetystä ohjelmiston palasta. Ohjelmistoarkkitehtuurit muistuttavat sovellusgeneraattoreita muun muassa siten, että lopullisessa sovelluksessa rakenne on laajalti uudelleenkäytettyä, mutta toisaalta osin kehittäjän itse toteuttamaa täsmäkieltä käyttäen ja vaatii siten teknisempää osaamista.

2.2.3 Muuhun, kuin ohjelmointiin keskittyneet menetelmät

Coomer ja kumppanit [5] jakavat uudelleenkäyttöön kohdistetut ratkaisut kahteen ryhmään, joista ensimmäinen käsittää edellä mainittuja konkreettisia artefakteja ja toinen abstraktimpia malleja (engl. design), ideoita ja tietoa. Jälkimmäiseen osioon kuuluvat ovat luonteeltaan käsitteellisiä, eikä niitä voida siten integroida kohteeseen suoraan, kuten konkreettisia artefakteja, vaan ne vaativat jonkinlaista toteutustyötä.

Kruegerin [19] tutkimuksessa käsitteellisimpiä artefakteja edustavat *ohjelmistoskeemat*. Ohjelmistoskeemoilla Krueger tarkoittaa malleja, jotka ovat hänen mukaansa formaali jatke lähdekoodikomponenteille. Ohjelmistoskeemat toimivat hänen mukaansa ratkaisuina ennalta tunnistettuihin ongelmiin ja ne koostuvat algoritmeista ja tietorakenteista, mutta eivät lähdekoodista. Artefaktin käyttäjät etsivät abstrakteja ohjelmistoskeemoja,

jotka tarjoavat ratkaisun formaalissa muodossa ilmaistuun ongelmaan ja ohjelmistoskeeman konkretisointi jää käyttäjän vastuulle.

Verrattuna lähdekoodikomponentteihin ohjelmistoskeemat ovat soveltuvampia useampaan tilanteeseen ja eri ohjelmointikieliin, sillä artefaktin käyttäjälle tarjotaan tietoa artefaktin rakenteesta menemättä liikaa toteutustason yksityiskohtiin [19], eikä teknisiä rajoitteita ole lähdekoodikomponentteihin verrattuna tällöin yhtä paljon. Toisaalta lähdekoodikomponenteissa ohjelmointikielen ollessa soveltuva kognitiivista etäisyyttä voidaan pitää lyhyempänä, kun artefaktin käyttäjän ei tarvitse olla sen sisällöstä tietoinen. Kruegerin [19] mukaan ohjelmistoskeemoja voidaan myös tarjota järjestelmien kautta, jotka auttavat ohjelmistoskeeman toteuttamisessa, mutta aiempiin menetelmiin nojaten tämä lähestymistapa asettaisi enemmän teknisiä rajoitteita esimerkiksi yhteensopivien ohjelmointikielten kanssa.

Kruegerin [19] tutkimuksessakin esiintyvä Kernighanin [17] käsittelemä *Unix-putki* (engl. Unix pipeline), mahdollistaa kahden sovelluksen toiminnan yhdistämisen. Unix-putki muuntaa kokonaisia sovelluksia uudelleenkäytettäviksi artefakteiksi, kun näiden yhdistelmillä voidaan täyttää uusia tarpeita. Alla olevassa esimerkissä 2.1 *esimerkki.txt* on 253 rivin mittainen tekstitiedosto, jonka pituuden käyttäjä voi saada näkyviin käyttämällä *less*-sovellusta "putkitettuna" *wc*-sovelluksen kanssa. less mahdollistaa tekstitiedoston lukemisen komentoriviltä, mutta | -putkimerkin avulla tuloste annetaan putkimerkkiä seuraavan sovelluksen syötteeksi. wc-sovellus tulostaa *-l* -valitsimella komentoriville syötteensä rivimäärän.

```
1  ~$: less esimerkki.txt | wc -l
2  253
3  ~$:
```

Esimerkki 2.1: Esimerkki Unix-putken käytöstä.

Putkitettuna täysin irralliset ohjelmat muodostuvat täten uudelleenkäytettäviksi artefakteiksi ja tämä uudelleenkäytön muoto on Unix-ympäristöissä vielä vuosikymmenien jälkeenkin varsin keskeisessä roolissa käyttäjien keskuudessa. Coomer ja kumppanit [5] nostavat tärkeäksi putkituksen ominaisuudeksi sen korkean abstraktion tason, sillä monimutkainen tekninen toteutus on saatu abstrahoitua varsin helppokäyttöiseksi komentorivityökaluksi. Kernighan [17] huomauttaa, että monet käyttäjien usein putkittamat sovellukset ovat päätyneet uudelleenkäytettäviksi, vaikka niitä ei alun perin oltu kehitetty uudelleenkäyttöä varten.

Maiden ja Sutfcliffe [24] esittivät, että ohjelmistojen vaatimusmäärittelyssä voisi hyödyntää uudelleenkäyttöä vastaavuuden kautta. Tällöin myös vaatimusmäärittelyn uudelleenkäytössä ongelmakenttä tulisi saada mahdollisimman hyvin abstrahoitua, jolloin eri toimialueiden vaatimuksista pystytään matalan tason eroavaisuuksista huolimatta löytämään korkean tason vastaavuuksia. Vaatimuksia olisi mahdollista muodostaa artefakteiksi katalogeihin, joissa ne esitetään abstraktion kautta [24].

Vaatimusmäärittelyn uudelleenkäyttö eroaa useimmista aiemmin esitetyistä menetelmistä siten, että katalogista ei suoranaisesti etsitä ratkaisua tarjoavaa artefaktia johonkin tiettyyn ongelmaan, vaan Maidenin ja Sutcliffen [24] mukaan toimialueen abstraktiolle tarjotaan mahdollisesti hyödyllisiä artefakteja. Näistä artefakteista käyttäjä voisi halutessaan valita omaan tilanteeseensa sopivia ehdokkaita haluamansa määrän. Koska vaatimuksien käyttö ei ole yhtä rajallista, kuin ongelmanratkaisukeskeisessä uudelleenkäytössä, voi artefaktien valinta edellyttää enemmän asiantuntemusta. Maidenin ja Sutcliffen mukaan vaatimusartefaktien etsintä ja valinta ovat lähtökohtaisesti ihmisen toteuttamia tehtäviä, sillä työkalua, jolla on sovelluksista ja toimialueista oikeanlaista ymmärrystä ja vastaavuuden havainnointikykyä, voi olla vaikea toteuttaa. Heidän mukaansa kokemuksesta on vaatimusmäärittelyssä apua, minkä vuoksi sen uudelleenkäytöstä suurta hyötyä voisivat saada erityisesti kokemattomat vaatimusmäärittelyn tekijät. Vaatimusartefaktien löytämisessä ja valitsemisessa voidaan silti tarvita ja hyödyntää käyttäjää valitsemisessa tukevia työkaluja [24].

Pääsääntöisesti sovellukset hyödyntävät *dataa*, jossa Jones [16] näki vuonna 1984 mahdollisuuksia uudelleenkäytölle. Jonesin [16] mukaan uudelleenkäytettävän datan edellytyksenä oli standardoitujen esitysmuotojen olemassaolo, mutta työ datan uudelleenkäytön mahdollistamiseksi tuolloin vasta alkutekijöissään. Koska datan käsittely liittyy lähes jokaiseen ohjelmistoon, voisi se uudelleenkäytön muotona olla varsin laaja-alaista.

2.2.4 Yhteenveto internetin yleistymistä edeltävistä uudelleenkäytön muodoista

Uudelleenkäytön muodot ennen internetin yleistymistä painottuivat vahvasti ohjelmakoodin tuottamiseen ja toteutusvaiheeseen. Taulukossa 2.3 on tiivistetty edellä esitettyjä historiassa tunnistettuja uudelleenkäytön muotoja ja nostettu niistä esiin oleellisia seikkoja.

	Mitä uudelleenkäytetään	Vaatimukset käyttäjältä	Huomioitavaa
Korkean tason kielet	Symbolisten konekielten rakenteita ja toistuvuuksia	”Klassista” ohjelmointiosaamista	Uudelleenkäytön uusi ”pohjataso”
Lähdekoodikomponentit	Lähdekoodikokonaisuuksia	Komponenttien integrointi sovellettavaan ohjelmistoon	Yleiskäyttöiset kirjastot hankala toteuttaa
Mallien ja koodin kaivelu	Suunnittelumalleja, ohjelmakoodia	Aiempaa kokemusta nykyistä vastaavista tarpeista	Sattumanvaraista ja kokemukseen pohjautuvaa
Sovellusgeneraattorit	Suunnittelumalleja, ohjelmakoodia (alijärjestelmät ym.)	Vaatimusten syöttäminen pitkälle abstrahoidulla syntaksilla	Yleiskäyttöiset generaattorit hankala toteuttaa
Erittäin korkean tason kielet	Korkean tason kielten laajempia yleiskäyttöisiä kokonaisuuksia	Ohjelmointia laajemmista kokonaisuuksista koostuvilla komponenteilla	Seuraava askel korkean tason kielistä eteenpäin?
Laajemman kirjon ohjelmointikielet	Korkean tason kielten laajempia yleiskäyttöisiä kokonaisuuksia	Ohjelmointia laajemmista kokonaisuuksista koostuvilla komponenteilla ja ”klassista” ohjelmointiosaamista tarpeen vaatiessa	Yhdistää VHLL:n ja korkean tason ohjelmointikielten etuja
Muunnosjärjestelmät	Optimoituja muunnoksia	Ohjelmiston toimintalogiikan kuvaus korkealla tasolla ja muunnoksien ohjaaminen	Toimintalogiikkaan kohdistuvat muutokset tehdään korkealla tasolla
Ohjelmistoarkkitehtuurit	Ohjelmistojen ”kehikkoja”, ohjelmakoodia (alijärjestelmät ym.)	Sopivan arkkitehtuurin valinta ja sen erikoistaminen täsmäkielellä	Erikoistaminen vaatii kyvykkyyttä käytössä
Ohjelmistoskeemat	Formaaleja ratkaisuja tunnettuihin ongelmiin	Ongelman kuvaamista formaalisti	Ohjelmistoskeeman lopullinen toteuttaminen tapahtuu käyttäjän toimesta
Ohjelma (Unix-putki)	Valmiita itsenäisiä ohjelmia	Yksittäisten sovellusten ja ”väliohjelmiston” tuntemista	”Suunnittelematonta” uudelleenkäyttöä
Data	Jonkin ohjelmiston käsittelemä data	*Ei esiinny aineistossa*	Datan esitysmuotojen standardointi puutteellista
Vaatimusmäärittely	Abstrahoituja vaatimuksia	Abstraktioiden vastaavuudet jäävät ihmisen tulkittaviksi	Avuksi erityisesti kokemattomille määrittelijöille

Taulukko 2.3: Uudelleenkäytön muotoja ennen internetin yleistymistä.

2.3 Uudelleenkäytön oletettuja hyötyjä

Loven [22] mukaan ohjelmiston kehittäminen alusta alkaen on taloudellisesta näkökulmasta tarkasteltuna yhtä järkevää, kuin tiettyyn tarkoitukseen tarvittavan tietokoneen rakentaminen alusta alkaen. Yksi ilmiselvä uudelleenkäytön hyöty on säästö käytetyssä ajassa ja sitä kautta ohjelmistotuotantoprosessin kokonaiskustannuksissa, kun artefaktia ei tarvitse luoda tyhjästä. Artefaktien hyödyntäminen ei kuitenkaan tuo sataprosenttista ajallista säästöä, sillä artefakti täytyy paikantaa, valita ja integroida, sekä erikoistaa sovelluskohteeseen sopivaksi.

Artefakti voi säästää kehityksessä käytettyä aikaa monitahoisesti. Jos ajatellaan normaalitapauksena sitä, että kehittäjä suunnittelee ja toteuttaa jonkin komponentin, voivat mahdolliset säästöt tuottavuudessa olla esimerkiksi Kimin ja Stohrin [18] koostaman taulukon 2.4 mukaisesti 50% luokkaa. Uudelleenkäytetyssä artefaktissa on kuitenkin myös se etu,

että se on jo aiemmin validoitu toiminnallisesti ja voi siten estää mahdollisia virhetilanteita. Seppäsen [30] mukaan paremman laadun myötä säästöä tulee edellä mainitun 50% tehokkuuslisän lisäksi myös virheiden korjaamiseen kuluvalta ajalta säästymisestä.

Uudelleenkäytön aste	Projekteja	Tuottavuuden parannus
48%	useita	8 - 9%/vuosi
25%	yksi	130 henkilötyökuukauden säästö
60%	useita	50% kasvu
35%	useita	250 henkilötyöpäivän säätö / kuukausi
42%	useita	33.9 koodiriviä (ei kommentti-) päivässä
14%	useita	1,5 miljoonan dollarin säästö
65%	useita	796% (lisätietoa alla)

Taulukko 2.4: Uudelleenkäyttöä tilastoituna, muokattu ja suomennettu [18].

Taulukossa 2.4 Kim ja Stohr [18] ovat koonneet eri tilanteissa havaittuja tuottavuuden parannuksia suhteessa projektin uudelleenkäyttöasteeseen. Taulukosta nähdään, että uudelleenkäytöllä on useassa yhteydessä havaittu olevan huomattavia vaikutuksia ohjelmistokehitykseen eri mittareilla. Toisaalta taulukko havainnollistaa hyvin myös sitä, että uudelleenkäytön hyödyllisyyden analysointi voi olla paikoin haastavaa, sillä eri mittareilla mitattuna tulokset ja hyödyt näyttävät varsin erilaisilta ja siten niitä vaikuttaisi olevan melko haastavaa vertailla keskenään [18]. Viimeisen rivin huomattavan suuresta tuottavuuden parannuksesta Kim ja Stohr huomauttavat, että hyödynnetystä automatiikasta johtuen uudelleenkäytön osuus on epäselvä.

Taulukon 2.4 toinen rivi on Loven [22] tutkimuksesta, jossa hän käsittelee ostettavissa olevien uudelleenkäytettävien komponenttien kustannuksellista ja ajallista tehokkuutta itse kehitettyyn lähdekoodiin verrattuna. Love [22] esittää, että hyödyntämällä uudelleenkäytettäviä komponentteja, jotka hänen mukaansa tyypillisesti sisältävät 10–15 metodia ja koostuvat 200–300 (ei-kommentti-)koodirivistä, 25% uudelleenkäyttöasteella voidaan saavuttaa 130 henkilötyökuukauden säästöt. Henkilötyökuukausien määrä perustuu siihen, että Loven [22] mukaan yhden edellä mainitun kaltaisen komponentin kehittämiseen menee yhdeltä henkilöltä noin yhden työkuukauden kestävä aika. Vähennettyjen henkilötyökuukausien tuomien säästöjen, joihin vaikuttaa uudelleenkäytettyjen komponenttien mahdolliset kustannukset, lisäksi kehitykseen kulutetun ajan lyheneminen tarkoittaa myös tuotteen markkinoille saattamisen nopeutumista ja täten nopeampia tuottoja [22].

Taulukon 2.4 rivillä viisi on Coomerin ja kumppaneiden [5] tutkimuksessa esiintyneet tulokset Nasan ohjelmistotuotantoprojektista. Sen jälkeen, kun NASA oli ottanut käyttöön Ada-ohjelmointikielen, he havaitsivat kolmanteen kyseisellä kielellä tuotettuun sovellukseen mennessä saavuttaneensa 42% uudelleenkäyttöasteen. Raportoitu päiväkohtainen tuottavuus oli 33,9 koodiriviä ja vikojen määrä tuhatta koodiriviä kohden oli vain yksi [5].

Kim ja Stohr [18], sekä Coomer ja kumppanit [5] mainitsivat esimerkkinä Toshiban Fuchu Software Factoryn (taulukko 2.4, rivi yksi), jossa vuonna 1977 alkanut uudelleenkäyttö johti vuosien saatossa noin 8–9% tuottavuuden kasvuun ja tuotetussa koodissa vikojen määrä oli tuhatta koodiriviä kohden kahdesta kolmeen.

Edellä mainitut tulokset antavat uudelleenkäytöstä varsin positiivisen kuvan, kun tuottavuus paranee ja vikojen määrä pysyy varsin maltillisena, mutta tarkasteltuna ne vaikuttavat vain suuntaa antavilta. Raportoiduissa tuloksissa ei ole mainintaa siitä, että uudelleenkäytettävien artefaktien todellista hyödyllisyyttä olisi koestettu kehittämällä rinnakkaisia ohjelmia ilman uudelleenkäyttöä. Rinnakkaisella, mutta semanttisesti identtisellä ohjelmistolla voitaisiin mahdollisesti paremmin arvioida tarpeellisten koodirivien määrää ja siten tehostamisen toteutumista. Syy tähän on siinä, että koodirivien määrä voi ohjelmistokohtaisesti olla varsin muuttuva, vaikka toiminnallisesti sisältö ei muutu. Tätä seikkaa havainnollistetaan alla.

Esimerkeissä 2.2, 2.3 ja 2.4 on esitetty kolme tapaa kirjoittaa JavaScript-koodilla taulukon alkiot järjestävä toiminnallisuus koodiriveihin perustuvien tehokkuuslaskelmien ongelmakohtien havainnollistamiseksi. Esimerkeissä on sama 5-alkioinen kokonaislukutaulukko ja järjestämisen lopputulos on sama. Jokaisen esimerkin alkutilanteessa alustetaan muuttuja a, joka saa arvokseen taulukon, jonka sisältönä ovat luvut yhdestä viiteen epäjärjestyksessä. Taulukolle kutsutaan tämän jälkeen järjestämisfunktiota, joka ensimmäisessä tapauksessa on toteutettu itse ja jälkimmäisissä tapauksessa hyödynnetään kielen omia mekanismeja. Jälkimmäiset tavat ovat muutoin samanlaisia, mutta esimerkissä 2.4 käytetään uudempaa ja tiiviimpää syntaksia[1].

[1] Arrow function expressions, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions, luettu 30.12.2019

```
a = [1,5,2,3,4];

function sort(a) {
    for(var i = 0; i < a.length; i++) {
        for (var j = i + 1; j < a.length; j++) {
            if (a[j] < a[i]) {
                tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
        }
    }
}
sort(a);
```

Esimerkki 2.2: Taulukon järjestäminen 1.

```
a = [1,5,2,3,4];
a.sort(function(a,b) {
  return a - b
});
```

Esimerkki 2.3: Taulukon järjestäminen 2.1.

```
a = [1,5,2,3,4];
a.sort((a,b) => a - b);
```

Esimerkki 2.4: Taulukon järjestäminen 2.2.

Täysin itse kehitetyn komponentin ja artefaktina hyödynnetyn komponentin välillä voi olla edellä esitettyjen esimerkkien 2.2, 2.3 ja 2.4 kaltaisia toteutuseroja. Esimerkin 2.2 koodirivien määrä on merkittävästi suurempi, kuin kummassakaan jälkimmäisessä esimerkissä ja toisaalta useat 2.3 ja 2.4 väliset erot voivat kumuloitua suurissa artefakteissa. Koodiriveihin perustuvia uudelleenkäyttöasteita, tai tehokkuuden parannuksia tarkkailevat mittaukset voivat siis olla hyvinkin epätarkkoja. On syytä huomioida, että kyseiset epätarkkuudet voivat antaa liian suotuisan, mutta myös epäsuotuisan, kuvan todellisesta uudelleenkäyttöasteesta tai tuottavuuden kasvusta.

Jonesin[16] laskelmien mukaan vuonna 1983 sovelluskoodia tuotettiin karkeasti noin 6,5

miljardia riviä. Ammattikunnan kasvu huomioiden esitettiin arvioita, että vuonna 1990 tämä luku yltäisi vuositasolla jo yli 15 miljardiin [16]. Koodirivien laskeminen uudelleenkäytön mittaamisessa on hyvin luonnollista, sillä ohjelmistokehittäjän työstä näkyvin ja helpoiten mitattava asia on koodirivit [14]. Uudelleenkäytettävän artefaktin koodirivit ja niiden määrä on myös hyvin tiedossa ja tunnistettavissa sovelluskohteessa, jolloin niiden tarkastelu uudelleenkäytön asteessa on perusteltua.

Ohjelmakoodin rivien määrään vaikuttaa se, lasketaanko mukaan ”tuottamattomat” rivit, joita ovat esimerkiksi ohjelmakoodin sekaan kirjoitetut dokumentoivat kommenttirivit. Taulukossa 2.4 ainakin Love [22] sekä Coomer ja kumppanit [5] jättivät kommenttirivit huomioimatta, mutta mikäli kommenttirivejä kirjoitetaan itse tehdyssä ja uudelleenkäytetyssä koodissa, onko niiden huomiotta jättäminen perusteltua? Hyvä dokumentaatio on tärkeä asia ammattimaisessa ohjelmistokehityksessä ja esimerkiksi Lanergan ja Grasso [21] käsittelevät uudelleenkäytön hyötyjä myös dokumentaation kautta. Hyvä dokumentaatio voi johtaa säästöihin esimerkiksi koodimuutosten aikana, kun koodin toiminnan ymmärtämiseen kuluvaa aikaa saadaan pienennettyä. Ongelmalliseksi kommenttirivien mukaan laskemisen tai laskematta jättämisen tekee se, että uudelleenkäytettävän koodikomponentin ylläpidettävyyttä parantavat tekijät eivät välttämättä ole sen käyttäjälle mielekkäitä asioita mitata, jos ylläpito ei ole käyttäjän itsensä vastuulla.

Lanerganin ja Grasson [21] tutkimuksessa käsiteltiin Raytheonin ohjelmistoissa käytettyjä Cobol-ohjelmointikielellä tehtyjä loogisia rakenteita. Kyseessä olevat artefaktit olivat organisaation omiin tarpeisiin organisaation sisällä kehitettyjä. Lanergan ja Grasso [21] havainnoivat Raytheonin ohjelmistokehityksessä monia muutoksia loogisten rakenteiden kehittämisen myötä: valmiiden rakenteiden avulla uusien ohjelmistojen rakenteiden suunnittelu helpottui ja toisaalta kehittäjien oli helpompi jäsennellä ratkaisuja mielessään valmiiden kokonaisuuksien avulla. Valmiit rakenteet olivat ennalta testattuja ja täten luotettavia, ne lyhensivät kehitysaikaa, ja olivat valmiiksi dokumentoituja, eikä dokumentaatio ollut yhtä hajautunutta eri ohjelmistoihin. Valmiita rakenteita hyödyntämällä sovelluksien välillä toistuvien osien kehitys muodostui nopeammaksi, jolloin uniikkien osien ohjelmointiin käytetty aika kokonaisajasta korostui halutusti [21].

Suurimmaksi hyödyksi Lanergan ja Grasso [21] kokivat menetelmässään valmiin sovelluksen ylläpidonaikaisten muutosten helppouden. Ohjelmistoratkaisuja tuottavien yritysten käyttämä aika kului heidän mukaansa jopa 60–80 prosenttisesti valmiiden sovellusten muutostarpeisiin vastaamiseen uusien ominaisuuksien kehittämisen sijasta. Rakenteiden omaksuminen vaatii uudelta kehittäjältä aikaa ja vaivaa, mutta tämän jälkeen niin ke-

hittäminen, kuin ylläpitokin on huomattavasti helpompaa. Kun valmiiseen sovellukseen vaaditaan muutoksia, ei ole yhtä suurta merkitystä sillä, kuka sovelluskoodin on alun perin kirjoittanut, koska organisaation sisällä laajasti käytetyt loogiset rakenteet johtavat siihen, että yksittäisten kehittäjien ohjelmakoodit ovat toistensa kaltaisia. Samankaltaisuudesta johtuen muutokset toteuttavan kehittäjän on helppo sisäistää alkuperäisen kehittäjän ohjelmakoodi. Muutosten helpottumisen lisäksi Lanergan ja Grasso [21] arvelivat, että loogisten rakenteiden hyödyntäminen uusien sovellusten kehittämisessä johtaa tuottavuuden kasvuun jopa 50%:lla, kun kehittäjä on omaksunut tietyn rakenteen käytön. Rakenteen käytön omaksuminen vaatisi heidän arvionsa mukaan sen käyttämistä kolme kertaa [21].

Jones [16] näki uudelleenkäytön tuovan hyötyjä myös ohjelmiston elinkaaren päättyessä. Hänen mukaansa yksi suurimmista taloudellisista perusteista uudelleenkäytölle ilmenee vanhojen järjestelmien korvaamisen kautta. Kun järjestelmän koostumus ja vaatimukset ovat hyvin tunnettuja, ne ovat korvattavissa nopeasti sekä matalin kustannuksin.

Taulukossa 2.5 on koostettu tässä luvussa esiin nousseita keskeisimpiä hyötyjä.

	Suunnittelu	Kehitys	Muutokset	Virhetilanteet
Saadut hyödyt	• Valmiiden kokonaisuuksien avulla suunnittelu karkeammalla tasolla • Artefaktit voivat poistaa osan suunnittelutyöstä valmiin arkkitehtuurien myötä	• Valmiit komponentit vähentävät ohjelmointiin kuluvaa aikaa ja siten kustannuksia • Uniikin ohjelmakoodin kirjoittaminen korostuu • Ohjelmistotuote saadaan markkinoille nopeammin	• Muutostarpeet helpompi täyttää kun kehittäjien ohjelmakoodi yhtenevää (komponentit organisaation sisältä) • Hyvin tunnetut vanhentuneet järjestelmät helpommin korvattavissa	• Valmiit komponentit valmiiksi testattuja

Taulukko 2.5: Ennen internetaikaa odotettuja uudelleenkäytön hyötyjä.

3 Tutkimus

Uudelleenkäyttöä ja sen mahdollisuuksia tutkittiin paljon viimeisinä vuosikymmeninä ennen vuosituhannen vaihdetta ja toisinaan se ehdittiin leimata epäonnistuneeksi. Nyttemmin voidaan todeta, että uudelleenkäyttö on ohjelmistokehityksessä erittäin vahvasti läsnä ja tiettyjen teknologisten edistysaskelien myötä sen muodot ovat lisääntynyt valtavasti. Uudelleenkäyttöä ilmenee mitä erilaisemmissa muodoissa, joista osa on sellaisia, mitä vielä 30–40 vuotta sitten ei välttämättä ollut mahdollista ennakoida. Tästä syntyi ajatus tutkia, mitä uudelleenkäytön muotoja historiassa tunnistettiin, mitä hyötyjä niistä havaittiin sekä odotettiin ja myös, mitä koettiin uudelleenkäytön erilaisten muotojen mahdollisiksi ongelmiksi. Tätä historiatietoa on hyödyllistä peilata nykypäivän toteutuneisiin uudelleenkäytön muotoihin, hyötyihin ja haasteisiin sekä tämän hetken ajatusmaailmaan uudelleenkäytöstä.

3.1 Tutkimusmenetelmä

Tämä Pro gradu -tutkielma on toteutettu kirjallisuuskatsauksena. Tutkimuksen taustalla on tietojenkäsittelyn alalla vuosikymmeniä sitten vallinnut tilanne, johon on tällä hetkellä parhaiten mahdollista perehtyä tuona aikana tehdyn kirjallisen aineiston kautta. Internetin yleistyminen on valittu taitekohdaksi uudelleenkäytön aikakausien välillä, sillä nykypäivän ohjelmistokehityksessä sen rooli on keskeinen. Internetin voidaan ajatella syntyneen 1980- ja 1990-luvuilla, mutta tässä tutkimuksessa sen vaikutusten katsotaan alkaneen näkyä parhaiten vasta vuosituhannen vaihteen jälkeen.

3.2 Tutkimuskysymykset

Tämän tutkielman tavoitteena on vastata seuraaviin tutkimuskysymyksiin:

> $\mathbf{Tk_1}$: Mitkä olivat uudelleenkäytön muotoja tietojenkäsittelyn ensimmäisinä vuosikymmeninä ja mitä odotuksia niille asetettiin?
>
> $\mathbf{Tk_2}$: Miten menneiden vuosikymmenten odotukset uudelleenkäytölle toteutuvat nykypäivän muotojen kautta?

Tutkielman alkupuolella painotus on ensimmäiseen kysymykseen vastaamisessa ja lopussa keskitytään vastaamaan toiseen kysymykseen alkuosan tukemana.

3.3 Tutkimuksen toteuttaminen

Kirjallisuutta on haettu pääasiallisesti merkittävistä digitaalisista kirjastoista, kuten IEEE Digital Library[1] ja ACM Digital Library[2], sekä hyödynnetty Helka-[3] ja Google Scholar[4] -hakupalveluita. Lähdeaineistoa on etsitty muun muassa hakusanoilla ”code reuse”, ”software reuse”, ”specification reuse” ja ”maintenance reuse”. Hakusanoilla löydetyn aineiston lähteiden kautta on laajennettu tämän työn lähdeaineistoa ns. ”lumipallomenetelmällä”, jossa löydettyjen lähdeaineistojen lähdemateriaalia on edelleen tutkittu.

Charles W. Kruegerin [19] (1992), Robert G. Lanerganin ja Charles A. Grasson [21] (1984) sekä Ellis Horowitzin ja John B. Munsonin [14] (1984) tutkimukset muodostuivat varsin keskeisiksi lähteiksi uudelleenkäytön historian osalta. Kruegerin kattava selvitys erilaisista uudelleenkäytön tunnistetuista muodoista 1990-luvun alkuun mennessä tarjosi hyvän lähtökohdan muun materiaalin läpikäymiseen. Lanerganin ja Grasson [21] tulokset uudelleenkäytön hyödyistä esiintyivät useissa lähteissä ja olivat omiaan tuomaan esiin uudelleenkäytön mahdollisia hyötyjä.

Uudelleenkäytön nykypäiväisistä muodoista on hankalampaa löytää tieteellistä aineistoa, mihin osasyitä on muun muassa tietojenkäsittelyn kehittymisen jatkuva nopeutuminen ja laajeneminen eri suuntiin. Akateeminen maailma ei tietojenkäsittelyn alalla edusta aina kehityksen kärkeä, vaan uusimmat ilmiöt nähdään usein ensimmäisenä kaupallisessa ohjelmistokehityksessä. Erityisesti uudempia tapauksia käsiteltäessä on siis toisinaan käytettävä epäformaalimpaa lähdeaineistoa, kuten blogitekstit ja dokumentaatiot, joiden synnyttäminen on nopeampaa ja siten niitä on laajemmin saatavilla. Nykypäiväisiä uudelleenkäytön muotoja on löydetty niin lähdeaineiston, lähdeaineiston ilmiöstä tunnistettujen vastaavuuksien, kuin oman ohjelmistoalalta saadun kokemuksenkin kautta.

---

[1] IEEE Xplore Digital Library, https://ieeexplore.ieee.org, luettu 28.12.2019

[2] ACM Digital Library, https://dl.acm.org/, luettu 28.12.2019

[3] Haun aloitussivu — Helka, https://helka.finna.fi/, luettu 19.12.2019

[4] Google Scholar, https://scholar.google.com, luettu 28.12.2019

3.4 Näkökulma nykypäivän uudelleenkäytön muotoihin

Internetaikakautta edeltävässä aineistossa oli havaittavissa toistuvuutta artefakteissa sekä siinä, miten näitä käsiteltiin. Krueger [19] tarkasteli tutkimuksessaan eri menetelmiä erityisesti neljän ominaisuuden kannalta, joita ovat *abstrahointi*, *valinta*, *erikoistaminen* ja *integrointi*. Nämä ominaisuudet esiintyivät myös muissa lähteissä suorasti tai epäsuorasti. Tässä tutkimuksessa esitellyt nykypäiväiset uudelleenkäytön muodot on pyritty käsittelemään myös näitä ominaisuuksia silmällä pitäen. Ominaisuuksia voi toisaalta myös pitää uudelleenkäytön menetelmien edellytyksinä, sillä jokaisen menetelmän tulee ne jollain tavalla täyttää tahallisesti tai tahattomasti. Edellä mainitut ominaisuudet avataan seuraavaksi.

3.4.1 Abstrahointi

Tutkimuksessaan Krueger [19] korosti abstrahoinnin merkitystä uudelleenkäytössä todeten, että se on uudelleenkäytön menetelmien keskeisin ominaisuus. Abstrahoinnissa uudelleenkäytettävää artefaktia pyritään kuvaamaan korkeammalla tasolla yksinkertaistetussa muodossa piilottamalla liialliset yksityiskohdat, jotta artefakti saadaan helpommin lähestyttäväksi [19]. Abstrahointi on tuttua esimerkiksi korkean tason ohjelmointikielistä, kuten Java, joissa ohjelmoijan ei tarvitse huolehtia laitteiston hallinnasta. Kruegerin [19] mukaan abstrahoinnin avulla voidaan nopeuttaa ja helpottaa käsityksen muodostamista siitä, mitä varten jokin uudelleenkäytettävä artefakti on suunniteltu, miten sitä voidaan hyödyntää ja minkälaisia tarpeita sillä voidaan täyttää.

Abstrahointi on jopa niin merkittävä asia, että se voi estää uudelleenkäytön onnistumisen. Krueger [19] esitti tutkimuksessaan uudelleenkäyttöön liittyviä truismeja, joista ensimmäinen liittyy abstrahointiin (suomennettu):

> *Jotta uudelleenkäytön menetelmä voi olla tehokas, sen täytyy vähentää kognitiivista etäisyyttä järjestelmän alkuperäisen ajatuksen ja sen lopullisen suoritettavan toteutuksen välillä.*

Truismi vaatii abstraktiolta ilmaisuvoimaa, jotta se voidaan helposti mieltää käsillä olevan ongelman ratkaisun osaksi ja tehokkuutta, jotta tämä voidaan tehdä pienellä vaivalla

mahdollisimman nopeasti. Alla olevissa esimerkkikoodeissa 3.1 ja 3.2 havainnollistetaan yksinkertaisen yhteenlaskufunktion kautta, kuinka abstrahointi voisi toimia käytännössä.

```
function sum(a,b) {
    return a + b;
}
```

Esimerkki 3.1: Yksinkertainen koodiesimerkki yhteenlaskun toteuttavasta funktiosta.

Vaikka *sum(a,b)*-funktion sisäinen toteutus esimerkissä 3.1 on triviaali, on alempana esitetty abstrahoitu versio huomattavan paljon kuvaavampi ja helpompi lähestyä.

```
sum(a, b): Palauttaa argumentteina annettujen arvojen summan
```

Esimerkki 3.2: Summafunktio abstrahoituna.

3.4.2 Valinta

Uudelleenkäytettävä artefakti on valittavissa, tai poissuljettavissa, mahdollisten artefaktien joukosta ensisijaisesti näiden abstraktioita vertailemalla, joten valinta on tiukasti sidottu artefaktin abstraktioon. Toinen Kruegerin [19] truismeista kuvaa hyvin valinnan ja abstraktion suhdetta (suomennettu):

> *Jotta artefakti voidaan valita uudelleenkäytettäviksi, on käyttäjän tiedettävä, mitä se tekee.*

Coomer ja kumppanit [5] toteavat tutkimuksessaan, että artefaktia voidaan hyödyntää vain, jos sen käyttäjä on ensin kykenevä paikantamaan sen. Artefaktin paikantaminen on luonnollisesti edellytys edeltävän truismin toteutumiselle. Paikantamisesta Krueger [19] toteaa kolmannessa truismissaan seuraavasti (suomennettu):

> *Jotta artefaktia voidaan käyttää tehokkaasti, se täytyy voida löytää nopeammin kuin se rakennettaisiin.*

Uudelleenkäytettäviä artefakteja tarjotakseen niiden kehittäjien täytyy huolehtia siitä, että käyttäjille tarjotaan jokin kanava, mistä artefakteja voi tarkastella. Tällaisia kanavia ovat esimerkiksi uudelleenkäytettäviä artefakteja sisältävät kirjastot. Kirjastosta valinta voi olla täysin manuaalista tai automatiikan avustamaa [19].

Mikäli uudelleenkäyttö koskee vain yksittäistä organisaatiota, voi kanavan tarjoaminen olla kohtuullisen helppoa, sillä kaikki päätökset kirjastoa koskien voidaan tehdä organisaation sisällä paikallisesti. Mikäli artefaktien kohdeyleisö on kuitenkin moninainen ja laaja, artefaktien ja artefaktikirjastojen standardointi helpottaa saavutettavuutta merkittävästi. Kimin ja Stohrin [18] mukaan standardointi voi laajemmassa mittakaavassa olla paljon haastavampaa kuin yksittäisen organisaation sisällä ja Love [22] huomasikin, että ulkopuolelta ostettavien artefaktien kanssa yhteensopivuusongelmat nousivat lähes aina esiin. Standardoinnin lisäksi Kim ja Stohr [18] mainitsevat eri organisaatioiden välisen artefaktien uudelleenkäytön mahdolliseksi ongelmaksi myös hankaluudet oikeuksien suhteen.

Artefakteja tarjoavan kanavan täytyy tehdä yksittäiseen artefaktiin perehtyminen mahdollisimman helpoksi. Yksinkertaisissa ja tarkasti määritellyissä tapauksissa artefaktin kuvaaminen voi olla varsin helppoa. Pelkkä nimi voi riittää tarkan käsityksen saamiseksi artefaktista varsinkin silloin, kun se toteuttaa jonkin tarkasti määritellyn tehtävän, joka on laajasti tunnettu ja hyväksytty. Kruegerin [19] mukaan uudelleenkäyttö on menestynyt parhaiten sovelluskohteissa, joissa abstraktiot ovat olleet "yksisanaisia". Tarkkoja määritelmiä löytyy usein esimerkiksi eksakteista tieteistä. Tarkoin määritellyt artefaktit voivat olla helpommin luetteloitavissa, kun ongelma- ja siten ratkaisukenttä on tarkoin tunnettu ja tällöin myös artefaktien joukko voi olla selkeästi rajattavissa.

Hankalammaksi muodostuu tapaukset, jossa artefakti täyttää jonkin tarpeen, joka voi olla hyödynnettävissä monella toimialla ja monessa käyttökohteessa. Kuinka artefaktit tulisi tällöin luokitella? Toimiiko sama luokittelu kaikkiin mahdollisiin käyttökohteisiin, eli saako kaikissa asiayhteyksissä selvän siitä, mitä artefakti tekee? Edellä mainitut truismit alkavat artefakteihin liittyvien rajoitteiden vähentyessä korostua enenevissä määrin.

Krueger [19] toteaa, että kirjastoa varten sen ylläpitäjän tarvitsee kehittää malli artefaktien kuvaamiselle. Tämä malli muodostaa alkuperäisen artefaktin abstraktion päälle toisen abstraktion tason. Esimerkiksi hänen mainitsemassa IMSL-matematiikkakirjastossa on hyödynnetty kolmea erilaista indeksiä artefaktien selaamiseen. Erilaisten indeksien ja kuvaustapojen hyödyntäminen voi osaltaan helpottaa tilannetta siinä, että asiayhteydestä riippumatta yleiskäyttöisen artefaktin sovellusmahdollisuudet tulevat esiin.

3.4.3 Erikoistaminen

Uudelleenkäytettävässä artefaktissa voi monessa tapauksessa olla tarvetta muuntaa sitä tilanteeseen sopivaksi. Tähän tarkoitukseen artefakti voi tarjota muutettavia osia, joi-

den avulla sovittaminen mahdollistetaan [19]. Näiden muuttujien mahdollisten arvojen erilaiset yhdistelmät yhdessä muuttumattomien osien kanssa muodostavat joukon, joka koostuu abstraktion erikoistetuista ilmentymistä [19]. Erikoistamisessa on siis kyse siitä, että uudelleenkäytettävän artefaktin abstraktiosta valitaan jokin sen ilmentymistä [19]. Erikoistamisen yhteydessä Krueger[19] mainitsee jälleen truismin (suomennettu):

> *Jotta uudelleenkäytön menetelmä on tehokas, täytyy artefaktin uudelleenkäytön olla helpompaa kuin sen kehittäminen alusta lähtien.*

Kuten edellä mainittuun truismiinkin sisältyy, artefaktin sovittaminen käyttötarkoitukseen sopivaksi ei saa olla erityisen hankalaa [19, 14]. Erikoistamisessa oleellisia asioita ovat Kruegerin [19] mukaan artefaktin *muuttumattomat* ja *muuttuvat* osat. Yksinkertaistettuna esimerkkinä erikoistamisesta toimii mielivaltainen matemaattinen laskutoimitus, joka toistetaan suorituskerroista riippumatta tismalleen samalla tavalla vain numeeristen arvojen muuttuessa. Tämän laskutoimituksen toteuttava funktio voisi olla uudelleenkäytettävä artefakti, funktion toimintalogiikka ja toteuttava koodi artefaktin muuttumattomia osia, ja argumentteina annettavat muuttujien arvot sen muuttuvia osia.

3.4.4 Integrointi

Integrointi kuvaa keinoja, joilla uudelleenkäytettävä artefakti saadaan liitettyä kohteeseen, jossa artefaktia aiotaan hyödyntää [19]. Tällöin artefaktin täytyy tarjota jonkinlainen rajapinta, minkä kautta vuorovaikutus artefaktin ja sovelluskohteen välillä tapahtuu.

Se miten integrointi käytännössä tapahtuu, tai miten se voi tapahtua, riippuu luonnollisesti hyvin paljon artefaktin luonteesta. Esimerkiksi puhtaasti käsitteellisessä artefaktissa, kuten suunnittelu- tai arkkitehtuurimalleissa integraatio ei ole erityisen konkreettista, kun taas lähdekoodin uudelleenkäyttöön perustuvissa artefakteissa vuorovaikutus artefaktin ja sitä käyttävän sovelluksen välillä täytyy määrittää hyvinkin konkreettisesti.

Konkreettisemmissa artefakteissa integroinnissa voi tulla haasteita muun muassa artefaktien toteutuksen ja sovelluskohteen teknisten valintojen välisistä eroavaisuuksista, kuten Love [22] edellä havaitsi. Edellisessä osiossa esitetyn truismin vaatimus artefaktin käytön helppoudesta on olennainen seikka myös integroinnissa, sillä artefakteja hyödynnettäessä suuri osa työstä sijaitsee tarkoituksenmukaisesti integroinnin puolella. Artefaktin integrointi sovelluskohteeseen jää viime kädessä aina artefaktin käyttäjän vastuulle. Kim ja Stohr [18]

havainnoivat, että useimmiten artefaktin integroinnista muodostuvat ongelmat on jätetty käyttäjän itsensä ratkaistaviksi ilman riittävää tukea artefaktia tarjoavalta taholta.

4 Uudelleenkäyttö nykypäivänä

Tämän vuosituhannen puolella uudelleenkäyttö ohjelmistokehityksessä on yleistynyt merkittävästi monessa eri muodossa. Sitä pidetään edelleen varteenotettavana menetelmänä laadun ja riittävän nopeuden ylläpitämiseksi [29, 6]. Dongarran ja Grossen [8] tutkimuksessa nähtiin internetin potentiaali artefaktien jakamisen avustamisessa jo 1980-luvun puolivälissä. Internetin arkipäiväistyminen, maailmanlaajuinen leviäminen, sekä sen avittama kehityksen siiloutumisen väheneminen ovatkin osoittautuneet erittäin merkittäviksi tekijöiksi tiedon jaon ja uudelleenkäytettävien artefaktien menestymiselle muun muassa jakamisen ja löydettävyyden helpottumisen kautta.

4.1 Näkökulma nykypäiväisen uudelleenkäytön tutkimiseen

Aiemmin uudelleenkäyttöä käsiteltiin pääasiallisesti kooditason ratkaisuna ja erot uudelleenkäytön menetelmissä näkyivät erityisesti siinä, kuinka lähellä ohjelmakoodia käyttäjä on. Oli myös varsin yleistä, että uudelleenkäytön mahdollisuuksia lähdettiin tutkimaan toimiala- tai tieteenalalähtöisesti [16, 19, 21]. Nyttemmin uudelleenkäyttöä voidaan hyödyntää pitkin ohjelmiston elinkaarta erilaisilla tavoilla ja ohjelmistotuotantoprosessin vaiheiden kautta uudelleenkäyttöä voidaan tarkastella erilaisesta näkökulmasta.

Ammattimaisessa sovelluskehityksessä on useita trendejä, jotka esiintyvät muun muassa tiettyjen työkalujen ja menetelmien käyttönä ohjelmistokehityksen eri vaiheissa. Monet näistä työkaluista ja menetelmistä hyödyntävät jollain tapaa uudelleenkäyttöä ja tietyissä tapauksissa ne ovat jopa ohjelmistotuotantoprosessin keskeisimpiä osia. Tässä luvussa esitellään nykypäivänä esiintyviä uudelleenkäytön muotoja ja uudelleenkäytettäviä artefakteja sidottuna seuraaviin ohjelmistotuotannon vaiheisiin: *määrittely, suunnittelu, toteutus, käyttöönotto* ja *ylläpito*.

4.2 Uudelleenkäyttö määrittelyssä

Pääsääntöisesti syy ohjelmistotuotteiden syntyyn ovat niille ilmenneet tarpeet, jotka muotoillaan tuotettavan ohjelmiston määrittelyksi. Ohjelmistotuotteen voidaan ajatella olevan ratkaisu johonkin ongelmaan, jonka määrittely kuvaa. Benittin ja da Silvan [3] mukaan uudelleenkäyttö on sitä tehokkaampaa, mitä aiemmin sitä hyödynnetään ja alkuvaiheessa esiintyvä uudelleenkäyttö saattaa johtaa uudelleenkäytön hyödyntämiseen ohjelmistotuotannon myöhemmissä vaiheissa. Laadukas määrittely voi edesauttaa ohjelmistoprojektin onnistumisessa esimerkiksi ohjelmiston tarpeiden selvän dokumentoinnin kautta. Edellä mainituista syistä määrittely vaikuttaisi olevan mitä otollisin vaihe uudelleenkäytön hyödyntämisen kannalta.

Määrittelynaikaista uudelleenkäyttöä ei vielä tänä päivänä ole laajasti hyödynnetty. Vaatimusmäärittely on yksi määrittelyn vaihe, jossa uudelleenkäyttöä on mahdollista hyödyntää. Schnitzhofer ja kumppanit [29] toteavat, että vaatimukset määritellään usein ohjelmiston tilaavan osapuolen, kuten loppukäyttäjien, toimesta, mutta vaikka tilaavan osapuolen edustajat ovatkin toimialansa ammattilaisia, eivät heidän taitonsa välttämättä riitä tukemaan laadukasta ohjelmistokehitystä teknisen tietämyksen puutteen johdosta. Vaatimusmäärittely on toimenpide, jossa kokemuksella ja ammattitaidolla on suuri merkitys. Ammattimaisten vaatimusmäärittelijöiden ja toimialatietämystä omaavien loppukäyttäjien tietotaitoa yhdistämällä voidaan Schnitzhoferin ja kumppaneiden [29] mukaan saada monia hyötyjä. Yhteistyön tukemiseksi Schnitzhofer ja kumppanit [29] esittävät vaatimusmäärittelyn uudelleenkäyttöön työkalua, jossa määrittelyä voidaan tehdä vaatimusmäärittelyn ammattilaisen tuottamien ja katselmoimien valmiiden pohjien avustuksella. Pohjat ohjaavat loppukäyttäjää tuottamaan määrittelyjä, joissa tarpeelliset tiedot tulevat varmasti täytetyiksi, ja voivat sisältää esitäytettyjä arvoja. Määrittelyyn voidaan lisätä vaatimuksia yhtä lailla pohjien tai valmiiden uudelleenkäytettävien vaatimusten avulla. Uudelleenkäytettäviin vaatimuksiin voi liittyä myös alivaatimuksia, jolloin määrittely vaatimuksineen muodostaa hierarkkisen rakenteen [29, 3]. Menetelmästä saatavia mitattuja hyötyjä ei kuitenkaan ole tiedossa.

Darimont ja kumppanit [6] ehdottavat vaatimusmäärittelyn tueksi ohjelmistotyökalua, jossa vaatimusmäärittelystä tehdään ohjelmistokehitystyökaluista tuttuun tapaan ”projekti”. Projektimuotoiset määrittelyt säilötään määrittelijän toimesta paikallisesti, mutta ne ovat jaettavissa, mikäli määrittely hyväksytään yleiseen käyttöön. Jaettavien artefaktien hyväksynnästä ja laadunvalvonnasta vastaa nimetty henkilö (Reuse Manager) [6]. Kyseinen

lähestymistapa vaatii kohtuullisesti organisointia, kun vastuista ja kommunikoinnin toteutumisesta täytyy huolehtia.

Yksittäisten organisaatioiden sisällä voi olla mahdollista uudelleenkäyttää vanhojen ohjelmistojen vaatimusmäärittelyjä, jos uuden ohjelmiston kohderyhmä, toimialue, tai ohjelmisto itse vastaa vanhaa ohjelmistoa. Benittin ja da Silvan [3] mukaan organisaation sisällä tapahtuvan kehityksen osalta erityisesti tuoteperheiden kehityksessä uusien tuotteiden vaatimusmäärittelyssä uudelleenkäytöstä voidaan saada paljon hyötyä. Luokitellessaan lähdeaineistoaan he havaitsivat, että valituiksi päätyneissä tapauksissa tuoteperheisiin kuuluvien ohjelmistotuotteiden osuus oli suuri. Tuoteperheiden vaatimusmäärittelyssä Maidenin ja Sutcliffen [24] käsittelemä tuotteiden vastaavuus lienee uudelleenkäyttöä merkittävästi edesauttava tekijä.

Benitti ja da Silva [3] tutkivat vaatimusmäärittelyn uudelleenkäyttöä kokeellisesti. Kokeessa tietojenkäsittelyn opiskelijat jaettiin kahteen ryhmään, joissa molemmissa suoritettiin vaatimusmäärittelyä yhdelle kahdesta kuvitteellisesta ohjelmistotuotteesta. Opiskelijat saivat koulutusta vaatimusmäärittelystä ennen kokeen aloittamista. Kummassakin ryhmässä opiskelijat olivat jakautuneet kuudeksi pariksi, joista puolet saivat käyttää vaatimusten uudelleenkäytön hyödyntämiseen kehitettyä SERS-työkalua ja puolet suorittivat vaatimusmäärittelyn ilman työkalua.

Benittin ja da Silvan [3] tutkimuksessa havaittiin, että lähes kaikissa tapauksissa SERS:n avustamat ryhmät saavuttivat merkittävästi paremman, lähes kaksinkertaisen tehokkuuden (löydettyjen vaatimusten määrä käytettyyn aikaan nähden) ja vaikuttavuuden (hyväksyttyjen vaatimusten määrä käytettyyn aikaan nähden) vaatimusmäärittelyn tuloksissa. Kontrolliryhmässä vain yksi pari onnistui pääsemään yksittäistä SERS:iä käyttänyttä paria korkeampaan tehokkuuslukuun saavuttamalla kaikista korkeimman ehdotettujen vaatimusten lukumäärän. Toisaalta hyväksyttyjen vaatimusten määrä jäi kyseiseltä parilta pienemmäksi kuin viidellä kuudesta SERS:iä käyttäneestä parista. Kun kokeessa analysoitiin SERS:iä käyttäneiden ryhmien hyväksyttyjä toiminnallisia ja ei-toiminnallisia vaatimuksia, havaittiin uudelleenkäyttöprosentin olevan useimmissa tapauksissa 50% tai enemmän ja vain yhdellä uudelleenkäyttöprosentti jäi alhaiseksi (ilmoitettu 33%, mutta aiemmin raportoitujen tulosten perusteella tämä vaikuttaa virheellisesti liian suurelta). Kokeen yhteydessä suoritetussa kyselyssä SERS:iä käyttäneet opiskelijat kertoivat pitäneensä työkalua hyödyllisenä vaatimusmäärittelyssä.

Maiden ja Sutcliffe [24] esittivät, että erityisesti kokemattomat henkilöt hyötyisivät vaatimusten uudelleenkäytöstä vaatimusmäärittelyssä. Edellä esitetyt vastaukset ja kokeelli-

set tulokset näyttäisivät tukevan tätä oletusta, sillä opiskelijoiden voidaan olettaa olevan vaatimusmäärittelijöinä melko kokemattomia.

Vaatimusmäärittelyn uudelleenkäyttöä tunnutaan lähestyvän pääsääntöisesti erilaisten tarkoitukseen kehitettyjen ohjelmistotyökalujen kautta. Erikoistamista ja integraatiota on mahdollista tukea työkaluilla, mutta vaatimusmäärittely on viime kädessä dokumentti, johon erikoistaminen ja integrointi eivät tavallisesti liity. Tärkeimmäksi ominaisuudeksi nousee se, miten artefaktit saadaan abstrahoitua ja valittua mahdollisimman helposti. Abstraktio riippuu täysin vaatimusmäärittelyä tukevan työkalun toiminnasta, mutta esimerkiksi kirjastomaisessa työkalussa se voisi tapahtua kuten komponenttikirjastoissakin.

4.3 Uudelleenkäyttö suunnittelun aikana

Benittin ja da Silvan [3] toteamus, että uudelleenkäytön tehokkuus korostuu, mitä aiemmin sitä hyödynnetään, näkyy vahvasti myös suunnittelunaikaisessa uudelleenkäytössä. Seuraavissa aliluvuissa esitellään käsitteellisempiä uudelleenkäytön menetelmiä, joiden merkitys alkaa parhaiten näkymään suunnitteluvaiheessa ja siitä eteenpäin.

4.3.1 Arkkitehtuurit ja suunnittelumallit

Korkealla tasolla tarkasteltuna ohjelmistot koostuvat loogisista kokonaisuuksista, joilla on omat vastuunsa ohjelmiston toiminnassa. Nämä kokonaisuudet yhdessä muodostavat sovelluksen arkkitehtuurin, joka voi toimia myös uudelleenkäytettävänä artefaktina.

Jonesin [16] mukaan Kendall (Kendall, R. C., 1983) toteaa, että vaikuttava uudelleenkäyttö vaatii lähtökohdaksi arkkitehtuurisen näkökulman, eikä pelkkä irrallisten komponenttien yhdistely ole riittävää. Harrison ja Avgeriou [11] puolestaan siteeraavat Clementsiä ja kumppaneita (Clements, P., Kazman, R. ja Klein, M., 2001), joiden mukaan ohjelmiston muokattavuus, tehokkuus, turvallisuus, saavutettavuus ja luotettavuus (engl. modifiability, performance, security, availability, reliability) ovat ominaisuuksia, joita huonon arkkitehtuurin myötä ei voida enää toteuttaa laadukkaasti.

Koska arkkitehtuuri on ohjelmiston osien muodostama kokonaisuus, ohjelmistolla on aina arkkitehtuuri riippumatta siitä, onko se suunnitelmallista vai ei. Valmiin ja toimivaksi todetun arkkitehtuurin ottaminen suunnittelun pohjaksi voi täten olla houkutteleva vaihtoehto. Alkuperäisen toteutuksen aikana arkkitehtuurien uudelleenkäyttö ei välttämättä tuo moneen muuhun artefaktiin verrattavia suoria ajallisia hyötyjä, mutta onnistuneet

suunnittelupäätökset voivat pienentää kustannuksia esimerkiksi ylläpitovaiheessa ja, kuten Harrison ja Avgeriou toteavat [11], ne voivat heijastua asiakastyytyväisyyteen, sekä muista ohjelmistoista erottumiseen.

Krueger [19] havaitsi, että monet uudelleenkäytön menetelmät kapseloivat sisäänsä myös arkkitehtuurin uudelleenkäyttöä. Arkkitehtuurien kapselointia on edelleen nähtävissä esimerkiksi ohjelmistokehyksissä ja -kirjastoissa, joita esitellään tarkemmin luvussa 4.4. Niissä yleistä on muun muassa Malli-Näkymä-Kontrolleri -arkkitehtuurin (engl. Model-View-Controller, MVC), tai jonkin sen muunnelman, hyödyntäminen. Ohjelmistokehyksissä ja kirjastoissa on usein pakotteita tietynlaisiin komponenttihierarkioihin ja komponenttien väliseen vuorovaikutukseen, minkä avulla arkkitehtuuria ohjataan noudattamaan.

Internetin myötä suosituiksi suunnittelumalleiksi ovat nousseet esimerkiksi palvelukeskeinen arkkitehtuuri (engl. Service-Oriented Architecture, SOA) ja mikropalveluarkkitehtuuri (engl. Microservice Architecture). Palvelukeskeisessä arkkitehtuurissa ja mikropalveluarkkitehtuurissa, joka Zimmermannin ja kumppaneiden [38] mukaan on tosin vain palvelukeskeisen arkkitehtuurin erikoistapaus, ohjelmisto jaetaan itsenäisiin kokonaisuuksiin, jotka ovat vuorovaikutuksessa keskenään esimerkiksi verkkoprotokollien ja -rajapintojen (käsitellään kohdassa 4.4.6) avulla. Kun ohjelmisto on hajautettu erillisiin itsenäisiin palveluihin, niitä voidaan kehittää eri teknologioilla ja päivittää yksittäin, jolloin voidaan sanoa palveluiden olevan ”heikosti sidottuja” (engl. loosely coupled) toisiinsa. Hajauttaminen tuo Zimmermannin ja kumppaneiden [38] mukaan mukanaan myös hyötyjä vikasietoisuudessa, kun virheet yksittäisessä palvelussa pääsevät heikommin vaikuttamaan koko ohjelmiston toimintaan.

Tarkkaa ajankohtaista tietoa eri arkkitehtuurien levinneisyydestä on hankala saada, mutta Balalaie ja kumppanit [1] havaitsivat mikropalveluarkkitehtuurin normalisoituneen yleisesti tunnetuksi malliksi muun muassa analysoimalla Google Trendsiä. Heidän mukaansa internethakujen sisältö mikropalveluarkkitehtuureiden osalta on siirtynyt arkkitehtuuriin perehtyvistä hauista teknologiakeskeisempään suuntaan lähemmäs toteutustasoa. He toteavat myös, että tämän hetken tunnetuimmatkin yritykset kehittävät sovelluksiaan mikropalveluarkkitehtuuria noudattaen.

Arkkitehtuuri on lähtökohtaisesti varsin abstrakti artefakti. Sitä voidaan havainnollistaa tiettyä arkkitehtuuria noudattavilla konkreettisilla ohjelmistoesimerkeillä, joihin artefakti on ”integroitu”. Useimpiin historiassa esitettyihin uudelleenkäytön menetelmiin verrattuna suunta on siis päinvastainen; artefakti on alkujaan abstrakti ja kognitiivisen etäisyyden lyhentämiseksi se voidaan kuvata toteutustasolla. Arkkitehtuurien abstraktiudesta joh-

tuen erikoistaminen on hyvin joustavaa, sillä niihin voi tehdä mielivaltaisia tilanteeseen sopivia muutoksia. Erikoistamisen jälkeen pohdittavaksi jää lähinnä se, kuinka tarkasti ohjelmiston erikoistettu arkkitehtuuri vastaa alkuperäistä artefaktia.

4.3.2 Prosessit

Perinteisen ohjelmistokehitysprosessin voidaan ajatella kulkevan vaiheittain kronologisesti alusta loppuun niin sanotun *vesiputousmallin* mukaisesti. Nyttemmin ohjelmistotuotantoprosessit noudattavat usein *ketterän kehityksen* mallia [15]. Prosessimallit eivät ole sidottuja pelkästään ohjelmistoprojektin suunnitteluvaiheeseen, vaan niitä hyödynnetään koko projektin ajan, mutta ketterät menetelmät rikkovat perinteisen vesiputousmallin suunnitteluosion ja levittävät sen lähes koko projektin mittaiseksi. Kuten Huo ja kumppanit [15] toteavat, ketterien menetelmien voidaan ajatella sisältävän paljon pieniä vesiputousmallin kaltaisia toistuvia syklejä.

Eri prosessimallien ja toimintatapojen hyödyntäminen on yleistä toimialasta riippumatta, mutta tämän vuosituhannen puolella erilaiset toimintamallit ovat saaneet varsin vahvan aseman. Scaled Agile Framework (SAFe)[1] ja Scrum Alliance[2] myöntävät sertifikaatteja, jotka luovat osaltaan ketteriin menetelmiin virallisen työkalun tuntua. Erilaisia ketteriä ja *kevyitä* (engl. Lean) prosessimalleja on useita, kuten esimerkiksi Scrum ja Kanban, ja näitä täydentävä koko ohjelmiston elinkaarta käsittävä DevOps. Prosessit voivat olla myös osittain sidoksissa arkkitehtuuriin, mikä on Balalaien ja kumppaneiden [1] mukaan nähtävissä DevOpsin ja mikropalveluarkkitehtuurien kanssa.

Prosessimalleissakin vaarana on aiempien menetelmien tapaan se, että menetelmän todellinen hyöty voi jäädä epäselväksi. Huo ja kumppanit [15] toteavat, että prosessimallien vaikutuksia tuotetun sovelluksen laatuun ei aina ole realistista vertailla, sillä kehityksen lähtökohdat ovat liian erilaiset. Erich ja kumppanit [9] taas huomasivat, että vaikka heidän haastattelemien DevOps-menetelmän käyttäjien kokemukset olivat positiivisia, ei kvantitatiivisia mittaustuloksia hyödyistä ollut saatavilla.

Arkkitehtuurien tapaan prosessit ovat alkujaan abstrakteja ja konkretisoituvat vasta niiden käyttöönoton myötä. Erikoistaminen, eli sovittaminen omaan toimintaan, on joustavaa, mutta valinta eri vaihtoehtojen välillä voi olla hankalaa. Tämä johtunee osaksi siitä,

[1] Scaled Agile Framework – SAFe for Lean Enterprises, https://www.scaledagileframework.com/, luettu 28.12.2019

[2] Scrum Alliance — Transforming the World of Work, https://www.scrumalliance.org/, luettu 28.12.2019

että sovelluskohteena ohjelmistotuotantoprojekti, sen sidosryhmät ja muut liittyvät tekijät voivat olla hankalia kuvata ja vertailla niitä tilanteita vasten, joissa prosessimallit toimivat. Tilannetta helpottaa kuitenkin juuri erikoistamisen helppous, jota kuvaa hyvin Scrumia ja Kanbania yhteen sulattava Scrumban[3].

4.4 Uudelleenkäyttö toteutuksessa

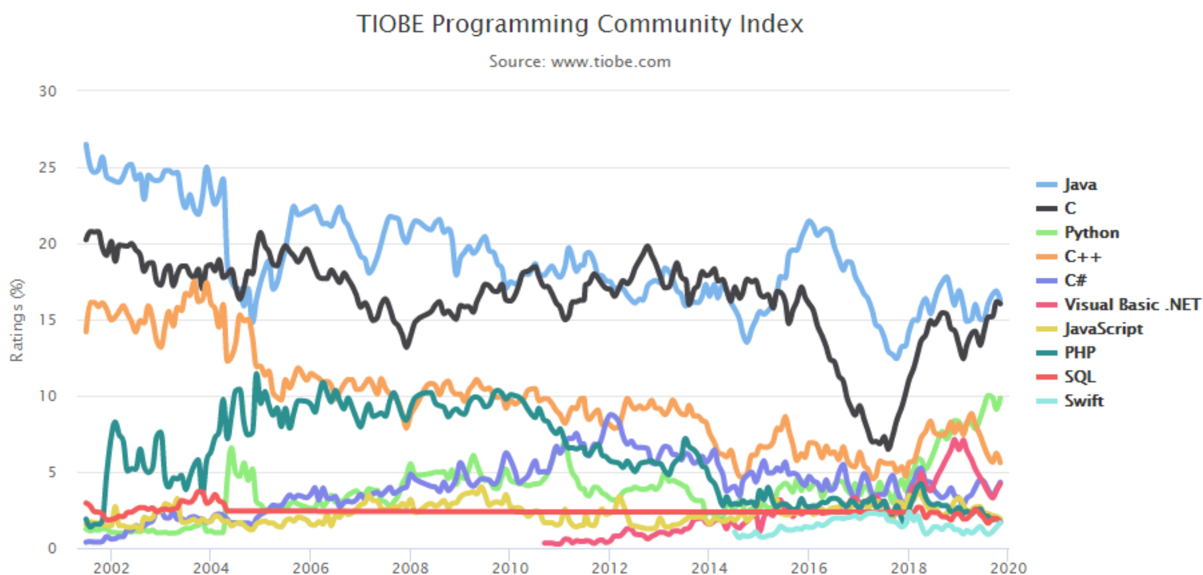
Ennen internetin yleistymistä uudelleenkäytön pääpaino oli sovelluskoodin tuottamisessa ja vaikka ”levittäytymistä” on tapahtunutkin vaikuttaisi siltä, että sama pätee tänäkin päivänä. Love [22] arvioi 1980-luvun lopussa, että oliot ja olio-ohjelmointi olisi ratkaisu niihin esteisiin, joita uudelleenkäytön menestymisen tiellä oli. Nykypäivänä olio-ohjelmointi on varsin laajasti levinnyttä, mutta vaikuttaisi siltä, ettei sen vaikutus uudelleenkäytölle ole loppujen lopuksi niin suuri, kuin mitä Love [22] arveli.

4.4.1 Nykypäivän yleiset ohjelmointikielet

Korkean tason ohjelmointikielten suosio on jatkunut jo vuosikymmeniä, eikä selkeitä viitteitä niistä pois siirtymiseen ole toistaiseksi havaittavissa. Tioben [34] ohjelmointikielten suosiota ylläpitävän indeksin mukaan suosituimmat ohjelmointikielet ovat olleet viime vuosina järjestään korkean tason ohjelmointikieliä, mitä havainnollistaa kuva 4.1. Suurimpina poikkeuksina tilastoissa voitaneen pitää SQL-täsmäkieltä, jolla ei voida kehittää ohjelmistoja, sekä Assembly, joka sitten vuoden 2001 on parhaimmillaan päässyt sijalle 8[4].

[3] What is Scrumban? — Agile Alliance, https://www.agilealliance.org/what-is-scrumban/, luettu 28.12.2019

[4] The Assembly language Programming Language, https://www.tiobe.com/tiobe-index/assembly-language/, luettu 3.1.2020



Kuva 4.1: Suosituimmat ohjelmointikielet kuukausittain ajalla 06/2001 - 11/2019 [34].

Kuva 4.2 esittelee ohjelmointikielten suosion viimeisen 30 vuoden ajalta Tioben [34] koostamana. Kuvasta on tehtävissä mielenkiintoinen havainto, kun tarkastellaan vuoden 2019 kärkisijoja, sillä neljä suosituinta ohjelmointikieltä löytyvät tilastosta jo vuodelta 1999. Erityisesti C ja C++ ovat pystyneet pitämään suosionsa, sillä 30 vuoden ajan C:n huonoin sijoitus on toinen ja C++:n vastaavasti viime vuosien neljäs sija.

Programming Language	2019	2014	2009	2004	1999	1994	1989
Java	1	2	1	1	3	-	-
C	2	1	2	2	1	1	1
Python	3	7	6	6	21	21	-
C++	4	4	3	3	2	2	2
Visual Basic .NET	5	9	-	-	-	-	-
C#	6	5	5	8	16	-	-
JavaScript	7	8	8	9	9	-	-
PHP	8	6	4	5	31	-	-
SQL	9	-	-	89	-	-	-
Objective-C	10	3	25	35	-	-	-
Lisp	32	17	16	13	14	5	3
Pascal	220	15	13	87	6	3	22

Kuva 4.2: Ohjelmointikielten suosio vuosittain viiden vuoden tarkkuudella 1989 - 2019 [34].

Edellisistä taulukoista paljastuu, että merkittävää muutosta ei ole tapahtunut siinä *minkälainen* ohjelmoinnissa käytetty kieli on. Ohjelmointikielten vanhat jättiläiset pitävät pintansa ja uudemmat, mutta kuitenkin tavalla tai toisella toisiaan muistuttavat kielet, syntyvät ja vaihtelevat suosion suhteen. Symboliset konekielet ovat selvästi väistyneet korkean tason ohjelmointikielten tieltä yleisellä tasolla, mutta erittäin korkean tason ohjelmointikielet tai laajemman kirjon ohjelmointikielet loistavat poissaolollaan.

Aiemmin tunnistetuista uudelleenkäytön menetelmistä lähdekoodikomponentit ovat ajan myötä alkaneet osin sulautua korkean tason ohjelmointikieliin. Monien ohjelmointikielten mukana on saatavilla erilaisia kirjastoja, jotka pitävät sisällään monia komponentteja yleisimpien tarpeiden täyttämiseksi. Esimerkiksi Lanerganin ja Grasson [21] mainitsemat päivämääriin liittyvät rutiinit ovat asioita, joiden käsittelyyn Javan luokkakirjastoissa on useita erilaisia luokkia[5,6].

Monet ohjelmointiympäristöt (engl. Integrated development environment, IDE) tukevat useita kieliä ja mahdollistavat monipuolisesti muun muassa lähdekoodin esitäyttöä, kääntämistä ja virheanalysointia Kruegerin [19] esittämien rakennesuuntautuneiden muokkausohjelmien (engl. Structure-oriented editors) mukaisesti. Niiden hyödyllisyys ohjelmistojen toteutuksessa on yhä korostuvampaa uusien ominaisuuksien kautta. Nämä ominaisuudet liittyvät esimerkiksi lähdekoodin visualisointiin, versionhallintaan ja automatiikan avustamaan dokumentaatioon ja niiden kautta ohjelmistojen kehittämiseen käytetyt työkalut keskittyvät yhä enemmän yhden pisteen taakse[7].

4.4.2 Paketinhallintajärjestelmät

Useille ohjelmointikielille on olemassa montaa eri roolia ajavia paketinhallintajärjestelmiä, joista eräs merkittävä rooli on uudelleenkäytettävien artefaktien tarjoaminen sovelluksissa käytettäviksi. Paketinhallintajärjestelmien vastuulla on muun muassa valinnan mahdollistaminen ja helpottaminen tuomalla artefaktin abstraktiot tarkasteltaviksi, sekä tukeminen artefaktin integraatiossa sovellukseen.

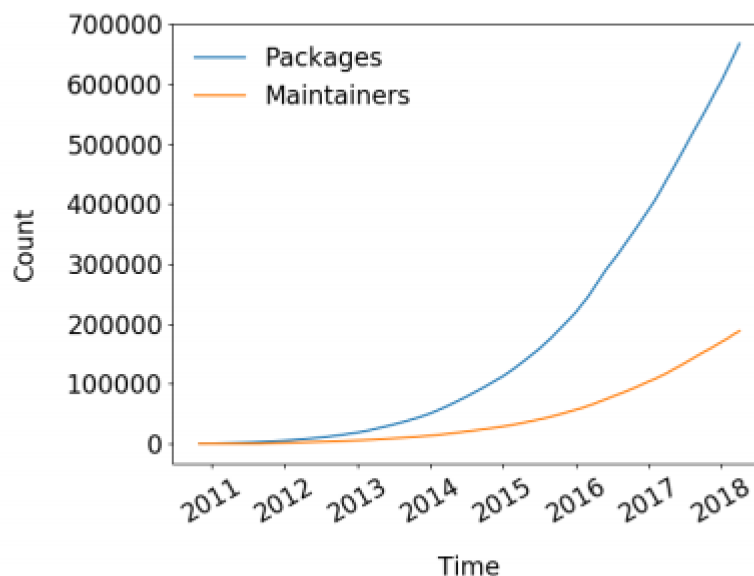
Yksi tällaisista järjestelmistä on Node.js:ää varten kehitetty Node Package Manager tai

[5] Date (Java Platform SE 8), https://docs.oracle.com/javase/8/docs/api/java/util/Date.html, luettu 28.12.2019

[6] GregorianCalendar (Java Platform SE 7), https://docs.oracle.com/javase/7/docs/api/java/util/GregorianCalendar.html, luettu 28.12.2019

[7] Visual Studio Code - Code Editing. Redefined, https://code.visualstudio.com/, luettu 3.1.2020

npm[8]. npm:ää voi hyödyntää JavaScriptilla kirjoitetun selain- tai palvelinpuolen ohjelman koonnissa ja testaamisessa, mutta erityisen hyödyllinen se on kolmannen osapuolen riippuvuuksien hallinnassa [37]. Suhteellisen lyhyessä ajassa npm:stä on muodostunut varsin merkittävä kehitystyökalu JavaScript-ohjelmointikielen tueksi, mitä selventää Zimmermannin ja kumppaneiden [37] koostama kuva 4.3 npm:stä löytyvien uudelleenkäytettävien artefaktien ja niiden kehittäjien määrästä.



Kuva 4.3: NPM-pakettien ja kehittäjien määrä vuositasolla. [37]

Useassa uudelleenkäytön menetelmässä on noussut esiin ajatus komponenttikirjastoista, joiden kautta eri tarkoituksiin kehitettyjä artefakteja voisi jakaa. npm tarjoaa juuri tällaisen kirjaston, minkä kautta miljoonat kehittäjät voivat hyödyntää kuvan 4.3 mukaisesti satojen tuhansien kehittäjien ylläpitämiä satoja tuhansia artefakteja, joita npm:ssä kutsutaan *paketeiksi*. Esimerkiksi npm:n eniten hyödynnetyn paketin, yleishyödyllisiä apufunktioita tarjoavan *lodash*:n, viikkotason lataukset vuonna 2019 ovat kasvaneet paikoin jopa yli 25 miljoonaan[9].

Yleisesti käytetyt paketit perustuvat avoimeen lähdekoodiin ja ovat vapaasti käytettäviä sekä muokattavia, poistaen siten Kimin ja Stohrin [18] huolia ongelmista oikeuksien suhteen. Vastuu pakettien aiheuttamista virheistä tosin jää käyttäjälle, mikä saattaa paketista johtuneissa virhetilanteissa osoittautua kiusalliseksi. Pakettien aiheuttamien virheiden

[8]npm — build amazing things, https://www.npmjs.com/, luettu 28.12.2019

[9]npm, https://www.npmjs.com/browse/depended, luettu 28.12.2019

esiintyessä ei ole takeita siitä, koska ja kenen toimesta virhe poistetaan, sillä niiden kehitys perustuu usein vapaaehtoisuuteen.

Avoimen lähdekoodin lisähyötynä on myös se, että artefaktin kehitykseen voi osallistua satoja henkilöitä. Aiemmin mainitun lodashin kehitykseen on GitHubin[10] mukaan osallistunut lähes 300 henkilöä ja ominaisuuksia, korjauksia ynnä muita sisältäviä *pull requesteja*, on käsitelty yli tuhat. Kyseinen laaja-alainen katselmointi on omiaan muun muassa koodin laadun varmistamiseksi ja yhteisöllinen kehitys voi auttaa edellä mainituissa tilanteissa, kun virheitä esiintyy.

npm:n kautta on havaittavissa ilmiö, jossa artefakteja käytetään monikerroksisesti. npm-paketeissa on mahdollista, että kehittäjän valitseman artefaktin, jota tässä tapauksessa kutsutaan myös kehitettävän ohjelman *riippuvuudeksi*, on riippuvainen muista npm-paketeista. Näillä ”piilossa” olevilla riippuvuuksilla voi olla edelleen riippuvuuksia, jolloin riippuvuusketju voi muodostua hyvinkin pitkäksi. Mikkosen ja Taivalsaaren [25] mukaan nykypäivänä on npm:n kaltaisten työkalujen kautta tavanomaista, että ohjelmistoissa kehittäjien tuntema koodi on vain ”jäävuoren huippu” ohjelmiston lähdekoodista. Kun tähän yhdistetään piilotetut riippuvuudet, voi pitkien riippuvuusketjujen takia olla lähes mahdotonta tietää, minkälaisista artefakteista ohjelmisto koostuu. Uudelleenkäyttö on tällaisessa tapauksessa jo varsin pitkälle vietyä, mutta siitä voi myös syntyä ongelmia, joita käsitellään myöhemmin luvussa 4.8.

4.4.3 Uudelleenkäytettävät räätälöitävät elementit ja komponentit

Web-komponentit ovat yksi ratkaisuehdotus internetsivujen ja selainpohjaisten sovellusten kehittämisen parantamiseksi, missä sovelluskehittäjät voivat sisällyttää uudelleenkäytettäviä komponentteja sovelluskoodiinsa HTML-tagien tapaan [35]. Web-komponentit noudattavat hyvin uudelleenkäytön ideologiaa, sillä web-komponenteissa niiden tarjoaminen kirjastomaisesti on keskeisessä roolissa.

Web-komponentit ottavat kantaa moneen historiassa esiin nousseeseen huoleen ja tarpeeseen. Web-sovelluskehitys on melko sirpaloitunutta selaimien tukiessa web-teknologioita eri tavoilla. Tämän lisäksi erilaisten JavaScript-kirjastojen käytön myötä standardoinnille on tarvetta. Web-komponentit perustuvatkin spesifikaatioihin, joista esimerkiksi *Custom Ele-*

[10]GitHub - lodash/lodash: A modern JavaScript utility library delivering modularity, performance, & extras., https://github.com/lodash/lodash, luettu 28.12.2019

ments ja *Shadow DOM* -spesifikaatioiden[11,12] takaa löytyy kansainväliset web-standardeja ajavat yhteisöt Web Hypertext Application Technology Working Group[13](WHATWG) ja World Wide Web Consortium (W3C)[14].

Web-komponenteissa [35] näkyy hyvin Kruegerin [19] esittämä ominaisuusnelikko, joka uudelleenkäytettävyyden kannalta on huomioitava. Valintaa tukeva komponenttikirjaston käyttöliittymä on helposti saavutettavissa verkkosivulta, mistä tarpeisiin sopivia komponentteja voidaan etsiä joko vapaasti selaamalla tai hakutoiminnoin. Komponenttien selailussa ja vertailussa komponentti on abstrahoitu vain nimen ja lyhyen kuvauksen tasolle, minkä avulla käyttäjälle tarjotaan helppo tapa ymmärtää komponentin tarkoitus, toiminta ja mahdollisuudet yleisellä tasolla. Komponenttikohtaisilla sivuilla tarjotaan tarkempaa tietoa muun muassa komponenttien rajapinnoista erikoistamista varten, alikomponenteista ja riippuvuuksista, lisenssitiedoista ja jopa pääsy komponentin lähdekoodiin. Integrointiin ei tarjota valmista mekanismia suoraan taustalla olevan tahon toimesta, mutta laajalti yleistyneet paketinhallintajärjestelmät mahdollistavat komponenttien sisällyttämisen omiin ohjelmistoprojekteihin, minkä jälkeen käyttäjän vastuulle jää itse komponentin käyttäminen kooditasolla.

Vaikka web-komponentit tuntuvatkin vastaavan ihanteellisesti historiassa esitettyihin vaatimuksiin menestyvälle uudelleenkäytölle, ei tutkielmassa löydetty näyttöä siitä, että web-komponentit olisivat aiheuttaneet erityistä murrosta selainpohjaisten sovellusten kehittämisessä. Selainpohjaisten sovellusten kehitys on toisaalta vielä verrattain nuori ja nopeasti kehittyvä sovelluskehityksen alue, mikä saattaa vaikuttaa web-komponenttien vaikuttavuuteen.

Erilaisia artefakteja on nyttemmin alettu jakaa myös varsin epävirallisin keinoin. Internetissä on lukuisia ohjelmistotuotantoon ja ohjelmointiin vihkiytyneitä sivustoja, joissa tietoalan ammattilaiset keskustelevat ja jakavat tietoa sekä mielipiteitä. Näistä sivustoista ammattilaisten keskuudessa yksi ehkä tunnetuimmista on Mikkosen ja Taivalsaarenkin [25] mainitsema Stack Overflow[15]. Stack Overflow:n foorumi sisältää lukuisia ohjelmistotuotantoon liittyviä kysymyksiä ja vastauksia.

[11]Shadow DOM, `https://w3c.github.io/webcomponents/spec/shadow/`, luettu 23.12.2019
[12]Custom Elements, `https://w3c.github.io/webcomponents/spec/custom/`, luettu 23.12.2019
[13]Web Hypertext Application Technology Working Group (WHATWG), `https://whatwg.org/`, luettu 23.12.2019
[14]World Wide Web Consortium (W3C), `https://www.w3.org/`, luettu 23.12.2019
[15]Stack Overflow - Where Developers Learn, Share, & Build Careers, `https://stackoverflow.com/`, luettu 13.12.2019

Rich ja Waters [27] kirjoittivat vuonna 1988, ettei sen ajan teknologialla ollut mahdollista ottaa käyttäjältä vastaan syötteitä luonnollisilla kielillä, kuten englanti, ja koneellisesti kääntää näitä ohjelmistoiksi. Heidän näkökulmansa oli melko pitkälle viety siten, että toimialan asiantuntijana loppukäyttäjä kykenisi luonnollisella kielellä antamaan ohjelmiston määrittelyn sovellusgeneraattorille, joka automaattisesti tuottaisi ohjelmiston. Aineistossa ei esiintynyt viitteitä siitä, että kyvykkyyttä siihen olisi vieläkään olemassa, mutta verrattavaa kehitystä on tapahtunut. Stack Overflow:n vastausarkiston kumppanina ja tietoalan ammattilaisten apuna toimivat hakupalvelut, kuten Google, jotka ymmärtävät luonnollisella kielellä tehtyjä hakuja. Hakupalvelut kykenevät tulkitsemaan haun ja yhdistämään käyttäjän Stack Overflow:n uudelleenkäytettäviin artefakteihin.

Satunnaisten vapaamuotoisiin kysymyksiin ja vastauksiin pohjautuvien artefaktien käyttö on kuitenkin melko epäkypsä uudelleenkäytön menetelmä. Abstraktion taso eri artefaktien välillä on mielivaltainen, mutta usein melko matala ja integrointi omaan ohjelmistotuotteeseen tapahtuu täysin tilanteesta riippuvasti käyttäjän toimesta. Erikoistaminen tämänlaisissa ad hoc -ratkaisuissa on joustavaa, vaikkakaan erikoistamismahdollisuuksia ei varsinaisesti ole määritelty ja artefaktien valinnassa sattuma on suuressa roolissa, sillä varsinaista katalogia tai vastaavaa ei ole. Mikkosen ja Taivalsaaren [25] mukaan tänä päivänä muun muassa Stack Overflow:n selaaminen valmiiden ratkaisujen toivossa on kuitenkin rutiininomaista.

Edellä mainittu uudelleenkäyttö nykyisessä muodossaan ei ole kovin lähellä Richin ja Watersin [27] kattavuutta, kun syötteen antaa loppukäyttäjän sijasta tuotteen kehittäjä ja artefaktikin on useimmiten vain koodikomponentin tai mallin tavoin hyödynnettävä lopullisen ohjelmiston osa. Se kuitenkin kertoo siitä, kuinka tämänhetkinen kehityksen suunta voi tulevaisuudessa johtaa heidän visioimiin uudelleenkäytön menetelmiin.

4.4.4 Ohjelmistokehykset ja kirjastot

Ohjelmistokehykset ja kirjastot ovat uudelleenkäytön menetelmä, jossa ohjelmistokehittäjät hyödyntävät laajaa kokoelmaa valmiiksi tehtyä käyttökelpoista ohjelmakoodia, joka tulee sisältymään kehitettävään ohjelmistoon [7].

Perusperiaatteeltaan ohjelmistokehys on artefakti, joka on kuin sovelluksen raamit, joka mahdollistaa ohjelmiston kehittämisen moneen tarpeeseen ja ottaa jossain määrin kantaa ohjelmiston yleiseen rakenteeseen. Tämän lisäksi ne sisältävät paljon hienomman tason komponentteja, joita kehittäjät voivat käyttää tarpeen mukaan.

Kehittäjän vastuulla on täydentää näitä raameja omalla ohjelmistokohtaisella ohjelmakoodilla, joka on samaa, yleensä korkean tason ohjelmointikieltä, kuin millä ohjelmistokehys on toteutettu. Kirjasto on hieman ohjelmistokehystä yksinkertaisempi siten, että se ei ohjaa ohjelmiston rakennetta yhtä vahvasti, vaan tarjoaa lähinnä tarpeen mukaan käytettäviä komponentteja. Kirjastot ovat mielenkiintoinen menetelmä, sillä aiemmin juuri niiden sisältämiä komponentteja ajateltiin artefakteiksi, mutta nyttemmin kokonaiset kirjastot ovat muodostuneet ohjelmistoon sisällytettäväksi artefaktiksi.

Historiassa mainittu laadun parantaminen on yksi painavista syistä ohjelmistokehysten ja kirjastojen hyödyntämiseen ja varsinkin web-sovellusten kehityksessä ohjelmistokehysten ja kirjastojen hyödyntäminen on erittäin yleistä[16]. Ocarizan ja kumppaneiden [26] tutkimuksen perusteella yleisyys johtuu osaksi siitä, että nämä abstrahoivat taakseen DOM API:n (engl. Document Object Model Application Programming Interface) ja vähentävät täten virhetilanteiden määrää.

Ohjelmistokehyksiä ja kirjastoja on kehitetty monelle kielelle, kuten esimerkiksi Angular[17], React[18] ja Vue.js[19] JavaScriptille ja Spring[20], Struts[21], sekä Hibernate[22] Javalle. Uuden ohjelmistotuotteen kehityksen aloitus voi olla hyvin suoraviivaista ja nopeaa, kun jotkin ohjelmistokehykset tarjoavat yksinkertaisen sovelluspohjan luonnin lähes ”nappia painamalla”.

Ohjelmistokehysten yksi negatiivinen puoli on se, että ne voivat sisältää runsaasti toissijaista koodia (engl. boilerplate code). Toissijaisella koodilla tarkoitetaan koodia, jota jonkin toiminnon täyttävä ydinkoodi tarvitsee toimiakseen [23]. Ohjelmistokehyksissä toissijaista koodia voi esiintyä esimerkiksi silloin, kun määritellään uusi ohjelmistokehyksen mukainen komponentti. Ohjelmistokehykset ja kirjastot voivat myös olla melko kookkaita, mistä voi luonnollisesti seurata runsas määrä niiden mukana tulevaa koodia, jota ohjelmistossa ei hyödynnetä. Tämänlainen koodi voi kasvattaa ohjelmiston kokoa ja esimerkiksi web-sovelluksissa se näkyy tarpeettoman suurina tiedonsiirtomäärinä.

Ohjelmistokehysten ja kirjastojen käyttö saattaa aiheuttaa epävarmuustekijöitä sovelluksen tulevaisuuden suhteen. Kehitettävistä ohjelmistoista tulee Digin ja Johnsonin [7] mu-

[16] Front-end Frameworks - Overview, https://2018.stateofjs.com/front-end-frameworks/overview/, luettu 28.12.2019
[17] Angular, https://angular.io/, luettu 28.12.2019
[18] React – A JavaScript library for building user interfaces, https://reactjs.org/, luettu 28.12.2019
[19] Vue.js, https://vuejs.org/, luettu 28.12.2019
[20] Spring, https://spring.io/, luettu 28.12.2019
[21] Welcome to the Apache Struts project, https://struts.apache.org/, luettu 28.12.2019
[22] Hibernate. Everything data. - Hibernate, https://hibernate.org/, luettu 28.12.2019

kaan riippuvaisia niiden hyödyntämistä ohjelmistokehyksistä ja kirjastoista. Kuten Robbes ja kumppanitkin [28] toteavat, kehitystyön myötä ohjelmistokehyksien ja kirjastojen rajapinnat voivat muuttua. Kun ohjelmistokomponentteihin, joista kehitettävä ohjelmisto on riippuvainen, kohdistuu muutoksia, ne heijastuvat niistä riippuvaisiin ohjelmistoihin tai ohjelmiston osiin. Tällöin riippuvaiseen sovellukseen kohdistuu siitä itsestään johtumattomia muutostarpeita [28]. Näihin muutostarpeisiin joudutaan useimmiten vastaamaan manuaalisesti [7], jolloin uudelleenkäytön tuomat säästöt pienenevät.

4.4.5 Alustariippumaton ohjelmistokehitys

Coomer ja kumppanit [5] korostivat käyttöjärjestelmän ja ohjelmointikielen valintaa uudelleenkäytön toteuttamisen yhteydessä. Käyttöjärjestelmään ja ohjelmointikieleen vahvasti liittyen on nykyään tarjolla ratkaisuja, joissa näiden merkitystä on saatu vähennettyä.

Erityisesti älypuhelinten yleistymisen myötä mobiililaitteiden käyttämille alustoille suuntautunut ohjelmistokehitys muodostui varsin merkittäväksi markkinaksi. Mobiilisovellusten, tai *appien*, suosion voidaan katsoa alkaneeksi vuodesta 2008, jolloin Apple julkaisi älypuhelinaikakauden käynnistäneelle iPhone-puhelimelleen App Store -mobiilisovelluskaupan. Nykyään alkuperäisten joidenkin satojen sovellusten sijasta App Store tarjoaa jo miljoonia sovelluksia[23].

Älypuhelinten ja mobiilisovellusten suuresta suosiosta syntyi mobiiliekosysteemien kilpailu, jossa kaikilla on omat sovelluskauppansa, mutta monet sovelluskehittäjät joutuivat tarjoamaan sovelluksiaan niistä useimmissa. Vielä vuonna 2013 ekosysteemejä, ja siten sovelluskauppoja, oli olemassa kuusi, joilla oli yli prosentin osuus kaikista sovelluskaupoista yhteensä tehdystä 81 miljoonan arvioidusta latauksesta [36]. Vuonna 2018 mobiilisovellusten latauskertojen kokonaismäärä oli jo lähellä 200 miljardin rajaa[24]. Elinvoimaisten ekosysteemien määrä on ollut kuluneen kuuden vuoden aikana varsin vaihteleva ja suurimpina alustoina ovat tuon ajanjakson säilyneet Googlen Android ja Applen iOS[25]. Vaihtelusta huolimatta, ja toisaalta sen takia, yhtenä ongelmana on ollut se, kuinka yhdellä sovelluksella voi päästä mahdollisimman hyvin osaksi koko mobiilisovellusmarkkinaa. Mobiilia-

[23]Apple's App Store now has over 2 million apps - The Verge, https://www.theverge.com/2016/6/13/11922926/apple-apps-2-million-wwdc-2016, luettu 23.12.2019

[24]Annual number of mobile app downloads worldwide 2018 — Statista, https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/, luettu 23.12.2019

[25]IDC - Smartphone Market Share - OS, https://www.idc.com/promo/smartphone-market-share/os, luettu 3.1.2020

lustoille kohdistunutta sovelluskehitystä on vaivannut alustojen ja niiden hyödyntämien teknologioiden sirpaloituminen [36]. Sirpaloitumisesta johtuen mahdollisimman menestyksekäs sovellus on täytynyt tarjota useassa eri versiossa eri ekosysteemit huomioiden.

Uudelleenkäyttö on tarjonnut alustarajat ylittävään sovelluskehitykseen ratkaisuja. React Native [10] on Facebookin tukema avoimen lähdekoodin[26] ohjelmistokehys, joka on suunniteltu natiivien mobiilisovellusten kehittämiseen. React Nativen suosiosta kertovat muun muassa GitHubin vuoden 2019 tilastot[27], joissa React Native oli kuudentena, kun mitataan GitHubissa kehitykseen osallistuneiden henkilöiden määrää. React Nativella kehitetyt sovellukset ovat rinnastettavissa Xanthopoulosin ja Xinogalosin [36] kuvaukseen *tulkituista sovelluksista*. React Nativen kaltaisilla ohjelmistokehyksillä kehittäjät kirjoittavat sovelluksensa toimintalogiikan jollakin ohjelmointikielellä, React Nativen tapauksessa JavaScriptilla, ja tämän perusteella ohjelmistokehys tuottaa natiivia ja kehyskohtaista ohjelmointikieltä hyödyntävän sovelluksen eri alustoille [36].

React Nativen verkkosivujen [10] mukaan, se pohjautuu Reactiin, joka on JavaScript-kirjasto interaktiivisten web-sovellusten ja käyttöliittymien luomiseen. Ensisijaisesti kehittäjät luovat React Nativella kehitetyt sovelluksensa käyttämällä JavaScript-ohjelmointikieltä. Mobiililaitteiden suosion myötä monet palveluntarjoajat joutuvat keskittymään perinteisemmän internetsivun lisäksi myös mahdollisimman sulavaan mobiilikäyttökokemukseen. Kun React Native pohjautuu Reactiin, on kehittäjien helppo siirtyä joko verkkosivujen kehittämisestä mobiilisovellusten kehittämiseen, tai toiseen suuntaan.

React Nativen kaltainen alustarajat ylittävä ohjelmistokehys hyödyntää Kruegerin [19] esittelemistä ohjelmistoarkkitehtuureista tuttuja uudelleenkäytön menetelmiä. Ohjelmistokehyksen kehittäjät ovat luoneet paljon valmiita komponentteja, joita kehyksen käyttäjät hyödyntävät kehittäessään sovellustaan ja täydentävät ohjelmakoodia uniikin toteutuksen osalta. Suoritettava ohjelmakoodi on yhdistelmä toistensa kanssa vuorovaikutuksessa olevaa natiivia ja ei-natiivia ohjelmakoodia. Tällöin sovellukset pääsevät hyödyntämään suoritusympäristön rajapintoja rajoituksetta ja tehokkuus ei jää vajavaiseksi, mikä Xanthopoulosin ja Xinogalosin [36] mukaan on ongelmana muissa alustarajat ylittävissä ratkaisuissa.

Sen lisäksi, että mobiilisovellusohjelmointi on helpompaa aloittaa, kun Android- ja iOS-

[26] GitHub - facebook/react-native: A framework for building native apps with React., https://github.com/facebook/react-native, luettu 23.12.2019

[27] The State of the Octoverse — The State of the Octoverse celebrates a year of building across teams, time zones, and millions of merged pull requests., https://octoverse.github.com/, luettu 23.12.2019

kehityksen tapauksessa kehittäjän ei tarvitse tuntea molempien ympäristöjen natiivia ohjelmointikieltä, myös aikaa säästyy huomattavasti. React Nativen kaltaisella kehyksellä kahta täysin erillistä, mutta näennäisesti identtistä suoritettavaa sovellusta voidaan kehittää yhdellä projektilla kahden sijasta. Ajansäästö uuden tuotteen kehityksen aloituksessa ja uusien ominaisuuksien lisäämisessä voi olla merkittävä.

Tulkituissa sovelluksista löytyy yhtymäkohtia myös laajemman kirjon ohjelmointikieliin, sillä natiivit API:t on saatu abstrahoitua uudelleenkäytettävien komponenttien taakse, mutta React Native -sovellusta on lisäksi mahdollista erikoistaa omilla natiivilla kielellä toteutetuilla moduuleilla siinä tapauksessa, että ohjelmistokehys ei tarjoa valmista vaihtoehtoa johonkin tarpeeseen [10]. Xanthopoulos ja Xinogalos [36] käsittelevät myös muun muassa *tuotettuja sovelluksia*, joissa kehityskieli on jokin täsmäkieli ja muunnosjärjestelmien tavoin alkuperäinen ohjelmakoodi muunnetaan eri muotoon, lopulta natiiviksi ohjelmakoodiksi. Kruegerin [19] mukainen ihmisohjaavuus on tosin korvaantunut automatisoiduilla muunnoksilla ja formaaliin muotoon abstrahointia ei tapahdu.

4.4.6 Muita uudelleenkäytön menetelmiä toteutusvaiheessa

Toteutusvaiheessa voidaan hyödyntää uudelleenkäyttöä myös menetelmin, joita ei selkeästi voida luokitella aiempiin alilukuihin. Näitä menetelmiä esitellään kootusti seuraavaksi.

Sivustonluontityökalut

Horowitz ja kumppanit [13] totesivat tutkimuksessaan, että sovellusgeneraattoreiden myötä sovellusten kehittäminen voi siirtyä loppukäyttäjille, joiden tietotekninen osaaminen ei ole korkealla tasolla. Nykypäivän esimerkkinä kyseisestä tilanteesta toimii selainpohjaiset sovellukset, joissa sovelluksen käyttäjät voivat itse rakentaa omia tarpeitaan palvelevan internetsivuston[28, 29].

Palvelut hyödyntävät uudelleenkäyttöä siten, että luotavan sivuston lopullinen lähdekoodi on abstrahoitu sivun luojalta ja sen sijasta käyttäjä luo sivuston erilaisia valitsimia ja käyttöliittymäelementtejä käyttämällä, sekä voi valita valmiita pohjia sivustolla käytettäväksi. Tällaiset *sivustonluontityökalut* ovat itsessään ohjelmistotuote, jotka abstrahoinnin avulla mahdollistavat internetsivustojen luomisen henkilöille, joilta vaadittava tekninen

[28] Build a Website – Website Builder – Squarespace, https://www.squarespace.com/, luettu 23.12.2019

[29] Free Website Builder — Create a Free Website — Wix.com, https://www.wix.com/, luettu 23.12.2019

osaaminen puuttuu. Lopullinen uudelleenkäyttöä hyödyntävä ohjelmistotuote on tässä tapauksessa sivustonluontityökalulla kehitetty internetsivusto.

Kehitystyötä on siis saatu siirrettyä ammattimaisilta kehittäjiltä pois, mutta sivustonluontityökalujen tarjoamat mahdollisuudet ovat huomattavasti rajallisemmat verrattuna siihen, että internetsivusto toteutettaisiin siihen tarkoitettuja ohjelmointikieliä hyödyntäen. On kuitenkin huomioitava, että sivustonluontityökalujen käyttö lienee harvoin ammattimaista, vaan sivustoja tehdään käyttäjän omiin tarpeisiin.

Integraatioväylät

Paljon ohjelmistoratkaisuja tehdään myös menetelmin, joissa ohjelmakoodin tuottaminen ei ole pääasiallinen kehitystapa. Hieman sovellusgeneraattoreita mukaillen voidaan nykyään tuottaa ratkaisuja muun muassa yrityksen tiedonvälitys- ja automaatiotarkoituksiin erilaisilla integraatioratkaisuilla.

Digitalisaation myötä monia osa-alueita yritysten toiminnasta on ratkaistu tietojenkäsittelyn keinoin, mutta näiden osa-alueiden välillä on tärkeää jakaa tietoa. Tiedon jakamista on kyetty ratkaisemaan erilaisilla järjestelmäintegraatioilla, jotka Landin ja kumppaneiden [20] mukaan sisältävät jo itsessään uudelleenkäyttöä monella tavalla eri järjestelmien yhdistelyn kautta. Sen sijasta, että jokaisen keskenään kommunikoivan kahden järjestelmän muodostaman parin välille tehtäisiin ad hoc -ratkaisuja tai pisteestä pisteeseen -integraatioita (engl. point-to-point / P2P integration), voidaan sovellusten välinen kommunikaatio mahdollistaa *integraatioväylien* avulla.

Integraatioväyliä voidaan kehittää niitä tarjoavien tuotteiden avulla, kuten esimerkiksi Mulesoftin Mule[30]. Itse väylän lisäksi Mule tarjoaa ohjelmointiympäristön, jonka pohjalla on Digin ja Johnsoninkin [7] tutkimuksessa käsitelty avoimen lähdekoodin Eclipse. Mulen ohjelmointiympäristöllä integraatioita voidaan konfiguroida graafisia käyttöliittymäkomponentteja hyödyntäen. Graafisessa käyttöliittymässä käyttäjä valitsee työkalupaletista tarvitsemiaan komponentteja, joita lisätään työskentelyalueelle. Komponentit voivat mahdollistaa muun muassa yhteyksiä eri järjestelmiin erilaisin tiedonsiirtoprotokollin, tiedon muunnoksia tai muunlaista käsittelyä. Komponentteja konfiguroidaan ensisijaisesti valmiiksi määritellyillä kentillä, jolloin kehitys on vahvasti ohjattua. Graafinen käyttöliittymä on vain yksi abstraktion taso, sillä varsinainen konfiguraatio tapahtuu Mulen tapauksessa

[30]MuleSoft — Integration Platform for Connecting SaaS and Enterprise Applications, https://www.mulesoft.com/, luettu 23.12.2019

XML-muodossa, jota ohjelmointiympäristö muodostaa käyttöliittymässä tehtyjen valintojen perusteella.

Mulekaan ei ole täydellisesti kaikkia tarpeita täyttävä, jonka vuoksi kehittäjillä on mahdollisuus erikoistaa ratkaisujaan. Erikoistaminen on mahdollista esimerkiksi itse kehitetyillä Java-komponenteilla, joita integraatiokonfiguraatiossa voidaan hyödyntää Mulen omien komponenttien tapaan. Edellä mainittu tapa kehittää on tuttu laajemman kirjon ohjelmointikielistä, kun graafisen käyttöliittymän tai käsin konfiguroidun erittäin korkean tason kieltä muistuttavaa integraatiokonfiguraatiota voidaan sovittaa omiin tarpeisiin korkean tason kielen avulla.

Verkkorajapinnat

Internet ja verkottuminen ovat mahdollistaneet tiedon laajamittaisen jakamisen myös datana. Jonesin [16] peräänkuuluttama datan uudelleenkäyttö on internetin myötä ottanut valtavan harppauksen niin jakokanavien kuin standardoitujen esitysmuotojenkin suhteen ja on siten arkipäiväistynyt.

Hartmann ja kumppanit [12] ovat tutkimuksessaan haastatelleet web-kehittäjiä, jotka ovat luoneet uusia sovelluksia hyödyntämällä jo olemassa olevien palveluiden jakamaa dataa verkkorajapintojen (engl. Web API) kautta. Ohjelmistokehittäjien on helppo käyttää esimerkiksi JSON- (JavaScript Object Notation) tai XML-muotoista (Extensible Markup Language) dataa vapaavalintaisella ohjelmointikielellä ja yhdistää erilaista tietoa muodostaen näiden avulla täysin uudenlaisia palveluita. Verkkorajapintojen yhdistelyssä Sillitti ja kumppanit [31] nostivat merkittäviksi eduiksi sen, että ne mahdollistavat epäyhteensopivissa ympäristöissä suoritettavien komponenttien yhteensovittamisen, sekä sen, ettei verkkorajapintoja tarjoavien palveluiden pystyttäminen vaadi niiden käyttäjiltä lainkaan työpanosta.

Verkkorajapintojen kasvu tuo uusia mahdollisuuksia ohjelmistokehitykselle. Hartmannin ja kumppaneiden [12] tutkimuksen raportoimasta, ProgrammableWeb:n[31] vuonna 2008 listaamasta alle tuhannesta API:sta on vuonna 2019 päästy jo yli 22 000 API:n lukemaan[32] ja suunta vaikuttaisi olevan edelleen kasvava. Verkkorajapintojen tarjoamaa dataa hyödyn-

[31] ProgrammableWeb - APIs, Mashups and the Web as Platform, https://www.programmableweb.com/, luettu 28.12.2019

[32] APIs show Faster Growth Rate in 2019 than Previous Years — ProgrammableWeb, https://www.programmableweb.com/news/apis-show-faster-growth-rate-2019-previous-years/research/2019/07/17, luettu 28.12.2019

tämällä ja yhdistelemällä voidaan luoda monenlaisia uusia palveluita, kuten esimerkiksi Hartmannin ja kumppaneiden [12] esittämät karttapohjaiset ravintola- ja sääpalvelut. Edellisen kaltaisissa *yhdistelysovelluksissa* (engl. mashup) pääosassa on datan ja uudelleenkäytettävien komponenttien yhdistäminen uudella tavalla ja itse sovelluskohtainen uniikki koodi voi olla toissijaista.

Tässä tutkielmassa verkkorajapintoja tarkastellaan siten, että niiden oletetaan toimivan lähinnä datan lähteenä. Verkkorajapinnat voivat myös tarjota monenlaista tiedon käsittelyä, jolloin ne ovat helpommin verrattavissa aiemmin esitettyihin ulkoisiin ohjelmistokomponentteihin.

Verkkorajapinnat ovat varsin pitkälle abstrahoitu, sillä niiden toteutuksesta ei tarvitse olla tietoinen, mutta toisaalta se ei ole edes merkityksellistä, sillä käyttäjä on todennäköisesti kiinnostunut vain verkkorajapinnasta haettavan datan sisällöstä ja muodosta. Integrointi muihin ohjelmistoihin tapahtuu kyselyillä, jotka noudattavat selkeitä ennalta määrättyjä sääntöjä ja vastauksen käsittely on kutsuvan ohjelmiston vastuulla. Erikoistaminen verkkorajapinnoissa tarkoittaa enemmänkin suodatusta, kun kyselyparametreilla annetaan lisätietoa siitä, millä ehdoin ja minkälaista dataa halutaan hakea. Valinta lienee yksi tärkeimmistä uudelleenkäytön ominaisuuksista verkkorajapinnoissa, sillä verkkorajapinta on helppo julkaista, mutta sen saattaminen kohderyhmän tietoisuuteen voi olla haastavampaa.

Krueger [19] käsitteli tutkimuksessaan artefaktien tarjoamista kirjastomaisesti ja totesi, että mitä suurempi määrä artefakteja kirjastossa on, sen vaikeampaa sopivan artefaktin löytäminen on. Verkkorajapintojen valinnassa juuri tämä on yksi haasteista, sillä esimerkiksi ProgrammableWeb:n kaltaisissa laajoissa sovelluskohderiippumattomissa luetteloissa vähemmän tunnetut verkkorajapinnat ovat helposti vaarassa jäädä monelta mahdolliselta käyttäjältä löytämättä.

4.5 Uudelleenkäyttö käyttöönotossa

Teknologinen kehitys on mahdollistanut tehokkaita uudelleenkäytön muotoja, kun kokonaisia suoritusympäristöjä tai niiden osia voidaan tehdä uudelleenkäytettäviksi ja monistettaviksi.

Säiliöintityökalut, kuten Docker[33] mahdollistavat käyttöjärjestelmätason virtualisoinnin,

[33]Docker, https://www.docker.com/, luettu 23.12.2019

jonka vaikutukset ohjelmistokehitykseen näkyvät jo kehityksen aikana, mutta erityisesti sen jälkeen. Säiliöinti toimii erinomaisesti mikropalveluarkkitehtuurin kanssa, kun erilaiset palvelut voidaan pakata *säiliöihin* tai *kontteihin*, minkä jälkeen ne ovat helpommin suoritettavissa missä tahansa ympäristössä, jolle kyseinen säiliöintituote on asennettavissa. Zimmermann [38] totesi säiliöintiteknologioiden olevan jopa yksi mikropalveluarkkitehtuurin dogmeista.

Coomer ja kumppanit [5] esittelivät vuonna 1990 uudelleenkäytön tavoitteiksi muun muassa laitteistoriippumattomuuden ja siirrettävyyden. Säiliöintiteknologia on edennyt jo pitkälle laitteistoriippumattomamman suoritusympäristön suuntaan, kun kerran kehitetty ja säiliöity ohjelmistotuote, tai sen osa, voidaan siirtää lähes muuttumattomana eri käyttöjärjestelmien välillä.

Sen lisäksi, että ohjelmistokehittäjät voivat säiliöidä itse kehittämiään sovelluksia, on myös mahdollista uudelleenkäyttää valmiita ratkaisuja. Ohjelmistoihin voidaan ladata suorituskelpoisia säiliöitä moneen tarkoitukseen, jotka ovat erikoistettavissa omiin tarkoituksiin konfiguroimalla. Säiliöintiteknologioiden suosiota kuvastaa esimerkiksi Docker Hub:ssa[34] saatavilla olevien tietokanta- ja HTTP-palvelin -säiliöiden latausmäärät, jotka liikkuvat yli kymmenessä miljoonassa. Docker Hubin kaltaiset kirjastot ovat monin tavoin kuten aiemmin käsitellyt komponenttikirjastot, mutta tässä tapauksessa komponentit ovat itsenäisiä ja kokonaisia palveluita sovelluksen kokonaisarkkitehtuurissa.

4.6 Uudelleenkäyttö ylläpidossa

Ohjelmiston ylläpito vaikuttaa olevan vaihe, jossa sille ominaisia uudelleenkäytön menetelmiä on haasteellista löytää. Toisaalta monet aiemmista vaiheista kuitenkin pyrkivät siihen, että ohjelmiston ylläpito olisi mahdollisimman mutkatonta. Lanergan ja Grasso [21] esittivät vuonna 1984, että 60–80 prosenttia ohjelmistotalojen käyttämästä ajasta menisi koodin muokkaamiseen uuden kehittämisen sijasta, jolloin juuri ylläpidon vaikutuksen minimointi olisi erityisen tärkeää.

Luvussa 4.3.1 mainittu mikropalveluarkkitehtuuri jakaa ohjelmiston hallittavampiin komponentteihin, mikä mahdollistaa sen, että ohjelmistoon kohdistuvat muutokset heijastuvat pienemmälle alueelle ja helpottaa näin ylläpidettävyyttä. Luvussa 4.3.2 esitetty DevOps tähtää siihen, että kehityksen jälkeisen irrallisen vaiheen sijasta ylläpito olisi-

[34] Docker Hub, https://hub.docker.com/, luettu 28.12.2019

kin tiivisti sidoksissa kehitykseen, jolloin muutokset sovellukseen voivat tapahtua nopeammalla aikataululla[35]. Luvun 4.4 esittelemät menetelmät tähtäävät kaikki osaltaan ylläpidettävyyteen; uudelleenkäytettävissä komponenteissa ylläpitovastuu on ulkoistettu ja historiassa esitettyjen ajatusten mukaisesti laadun tulisi olla lähtökohtaisesti varmennettu. Ohjelmistokehykset ja kirjastot abstrahoivat suurelta osin osia toimintalogiikasta vähentäen virheiden mahdollisuuksia, alustariippumattomissa ohjelmistokehitysmenetelmissä muutoksia saadaan yhdellä kertaa tehtyä useampaan eri suoritusympäristöön.

4.7 Nykypäivän uudelleenkäytön muodot kootusti

Kuten aiemminkin, nähtävissä on edelleen uudelleenkäytön painottuminen ohjelmiston toteutusvaiheeseen. Muutosta on kuitenkin havaittavissa, kun uudelleenkäytön menetelmät ovat alkaneet levitä myös muihin ohjelmistotuotantoprosessin vaiheisiin ja yksittäiseen vaiheeseen liittyvillä menetelmillä on useimmiten kauaskantoisia vaikutuksia. Taulukossa 4.1 esitellään tiivistettynä tässä luvussa käsiteltyjä uudelleenkäytön muotoja, kerrotaan mihin ohjelmistotuotantoprosessin vaiheeseen ne oleellisimmin liittyvät ja nostetaan esiin niiden keskeisiä ominaisuuksia.

[35]What is DevOps? — Atlassian, https://www.atlassian.com/devops, luettu 12.12.2019

	Määrittely	Suunnittelu	Toteutus	Käyttöönotto	Ylläpito
Vaatimusartefaktit	Vastaavissa ohjelmistoissa voi hyödyntää samoja vaatimuksia				
Arkkitehtuurit & Suunnittelumallit		Arkkitehtuuri koostuu suunnittelupäätöksistä, joiden pohjalta ohjelmisto toteutetaan			Hyvä arkkitehtuuri helpottaa muutostyötä
Prosessimallit	Ketterien kehitysmallien myötä ohjelmistotuotannon vaiheet ovat toistuvia				
Ohjelmointikielet			Ohjelmointikielten kirjastot sisältävät artefakteja ja monet artefaktit ovat kieliriippuvaisia		
Paketinhallinta-järjestelmät			Mahdollistaa mm. artefaktien liittämisen ohjelmistoon		
Räätälöitävät elementit ja komponentit			Standardeihin perustuvia komponentteja kehitykseen		
Ohjelmistokehykset ja kirjastot		Ohjelmistokehykset ohjaavat suunnittelussa / toteutuksessa käytettävissä komponentteja			
Tulkitut ja tuotetut sovellukset			Yhdestä lähdekoodista sovellus usealle ohjelmointikielelle ja/tai alustalle		
Sivustonluontityökalut			Internetsivusto omaan käyttöön itse luotuna		
Integraatioväylät			Järjestelmien väliset integraatiot kehitetään korkealla tasolla		
Verkkorajapinnat	Synnyttää uusia ohjelmistoja ja mahdollistaa datan uudelleenkäyttöä				
Säiliöinti		Pienet säiliöidyt palvelut ovat joustavia ottaa käyttöön ja päivittää			

Taulukko 4.1: Uudelleenkäyttötapoja ohjelmistotuotannon eri vaiheissa.

Monet edellä esitetyistä menetelmistä ovat toisiaan tukevia ja niitä voidaan käyttää yhdessä. Lähtökohtaisesti uudelleenkäytön ajatuksena on se, että ohjelmistotuotannossa hyödynnetään artefakteja, jotka vähentävät vaaditun työn määrää ja kustannuksia. Uudelleenkäytöllä alkukustannus pysyy pienenä vaaditun työn vähyyden myötä. Lopullinen ohjelmisto on abstraktiotasoltaan korkeampi ja joustamattomampi myös kehittäjien näkökulmasta, eikä se ole täysin sen kehittäjien hallinnassa. Hartmann ja kumppanit [12] kuitenkin toteavat, että uudelleenkäytettäviä artefakteja on mahdollista korvata tarpeen vaatiessa itse kehitetyillä ratkaisuilla, jolloin alusta asti kehittämisen korkea kustannus realisoituu vasta silloin, kun sille on todellinen tarve.

Nykypäiväisten uudelleenkäytön muotojen uskotaan edelleen vähentävän työn määrää ja parantavan laatua. Vaikuttaisi kuitenkin siltä, että uudelleenkäytön hyötyjä ja haittoja käsitellään nykyäänkin kohtuullisen pinnallisesti ja uskomuspohjaisesti, ilman riittäviä objektiivisia tutkimuksia todellisista säästöistä. Hartmannin ja kumppaneiden [12] tutkimuksessa haastateltavat henkilöt totesivat, että komponenttien etsimiseen menee merkittävästi aikaa ja joillakin se oli kehityksessä jopa työläin vaihe. Vaikka etsintään ja

hyödyntämiseen kulutettu aika kirjallisuudessa tunnistetaankin, ei järkevästi mitattavaa vertailua itse kehitettyjen ja uudelleenkäytettävien artefaktien välillä tunnu edelleenkään esiintyvän. Syy saattaa löytyä esimerkiksi siitä, että tosimaailman tilanteessa tehty mittaus kahden vaihtoehdon välillä toisi ylimääräisiä kustannuksia ja resurssihaasteita, joita uudelleenkäytön hyödyntämisellä koitetaan juuri ehkäistä.

4.8 Nykypäivän uudelleenkäytön vaikutuksia

Uudelleenkäytön nykypäiväiset muodot ovat tuoneet monia muutoksia siihen, miten ja millaisena ohjelmistotuotanto tänä päivänä esiintyy. Uudelleenkäytön tarkoituksena on parantaa ohjelmistotuotantoprosessia sen eri osa-alueilla, mutta uusien muotojen kautta myös negatiivisia vaikutuksia on nähtävissä. Seuraavaksi tarkastellaan uudelleenkäytön vaikutuksia edellisessä luvussa esiteltyjen uudelleenkäytön muotojen kautta ryhmiteltynä niissä havaittaviin ilmiöihin, jotka tutkimuksen tekijän mielestä nousevat keskeisimmiksi.

Kehityksen nopeutuminen

Yksi usein esiintyvä uudelleenkäytön tavoite on tehtävän työn abstraktiotason nostaminen, jolloin osaamisvaatimukset työn tekemiseen eivät ole niin korkeat. Tästä seuraa luonnollisesti se, että tehtävä työ on helpompaa ja sen kykenee suorittamaan yhä useampi henkilö tai jopa kone.

Horowitzin ja kumppaneiden [13] tutkimuksessa esitetty mahdollisuus siihen, että esimerkiksi sovellusgeneraattoreiden myötä ohjelmistoja eivät kehittäisi vain ohjelmistokehittäjät, ei vielä ole havaittavissa. Aiemmin esitellyt sivustonluontityökalut ja integraatioväylät toki mahdollistavat ohjelmistojen kehittämisen ilman syvää ohjelmointiosaamista, mutta sivustonluontityökaluissa kohderyhmä on omiin tarkoituksiinsa kehitettäviä sivustoja luovat henkilöt, eikä kehitys ole järin ammattimaista. Integraatioväyliä kehittävät henkilöt toisaalta ovat ammattilaisia, mutta integraatioiden kehittämistä voi tuskin täysin rinnastaa yleiseen käsitykseen ohjelmistojen kehittämisestä. Lisänä edellisiin mainittakoon mahdollisuus kehittää tulkittuja ja tuotettuja sovelluksia, jolloin kahdelle eri ohjelmointikieltä käyttävälle alustalle saadaan kehitettyä ohjelmistoja lähes kokonaan kolmannella kielellä vähentäen kehittäjiltä vaadittujen ohjelmointikielten osaamista.

Ohjelmistotuotannossa, jossa ohjelmakoodin tuottaminen on keskiössä, on kuitenkin tapahtunut suuria muutoksia siinä, kuka lähdekoodin on tuottanut. Ohjelmistojen arkkiteh-

tuurit voivat noudattaa mikropalveluarkkitehtuuria, jossa esimerkiksi HTTP-palvelimen ja tietokannan pohjalla on erikoistettu valmis säiliö. Ohjelman lähdekoodi voi myös pohjautua suurelta osin kolmannen osapuolen tuottamiin komponentteihin, jotka ovat ulkoisten vastuukehittäjien ja yhteisön ylläpitämiä. Valmiiden komponenttien myötä omassa tuotteessa voidaan hyödyntää laajan kehittäjäjoukon tietotaitoa ilmaiseksi ja jopa sisällyttää tuotteeseen ominaisuuksia, joiden kehittämiseen ohjelmiston omalla kehittäjäjoukolla olisi puutteelliset taidot. Myös korkean tason ohjelmointikielten omien kirjastojen ja kyvykkyyksien jatkuva kehittyminen mahdollistaa ohjelmistokehittäjiä monipuolisemman ohjelmakoodin tuottamiseen pienemmällä vaivalla.

Säiliöidyt pienet palvelut mahdollistavat ohjelmistojen nopean käyttöönoton eri ympäristöissä ja valmiit arkkitehtuurimallit helpottavat ylläpitotyötä. Ylläpitovaiheen keveyden myötä, ketterien menetelmien tukemana, ohjelmistoihin päästään kehittämään uusia ominaisuuksia nopeammin, eivätkä Lanerganin ja Grasson [21] arviot ylläpitovaiheen raskaudesta pääse toteutumaan.

Ohjelmiston koostumuksen hämärtyminen

Ulkoisten riippuvuuksien kautta ohjelmistoon tuotu ohjelmakoodi on voinut tarkoittaa hyvinkin suuren koodimäärän liittämistä ohjelmistoon, minkä sisältöä ei ole järkevällä työpanoksella mahdollista käydä läpi. Toisaalta vaikka koodi olisikin mahdollista käydä läpi, olisi uudelleenkäytön tuoma hyöty tällöin pienempi [19].

Uudelleenkäytettävien komponenttien lähdekoodi jäänee varsin monelta ohjelmistokehittäjältä piiloon abstraktion taakse, mikä koodikomponenteissa on toisaalta myös tarkoituksenmukaista. Lähdekoodin lisäksi ohjelmistokehittäjiltä voi monikerroksisen uudelleenkäytön myötä jäädä tiedostamattomaksi myös se, mitä komponentteja ohjelmistossa on uudelleenkäytetty. Uudelleenkäytettäviin komponentteihin perehtyminen heikentää uudelleenkäytöstä saatuja hyötyjä sen vaatiman ajan vuoksi, mutta Mikkosen ja Taivalsaaren [25] mukaan komponentteihin perehtymättä jättämiseen on myös nykypäiväiseen kehittäjäkulttuuriin liittyviä asenteellisia syitä. On siis havaittavissa asennemuutos internetiä edeltävän ja seuraavan aikakauden välillä, kun aiemmin jokaista ohjelmistoa pidettiin lähtökohtaisesti uniikkina ja ”alusta asti ohjelmoitavana”, ja nykyään ohjelmistoja koostetaan valmiista osista kiinnostumatta siitä, mitä nämä osat todellisuudessa ovat.

Tietoturva ja riskit

Ohjelmiston hyödyntämät uudelleenkäytettävät artefaktit saattavat olla liitettävissä ohjelmistoon kokonaisuuden mukana tai niitä voidaan tarjota jonkin työkalun kautta. Tällöin ohjelmistoon aukeaa uusi väylä mahdollisille tietoturvaongelmille ja muille riskeille.

Ulkoiset riippuvuudet voivat tuoda sovellukseen mukanaan uusia tietoturvauhkia ja riskejä. Esimerkiksi marraskuussa 2018 npm-paketinhallintajärjestelmän *event-stream*[36] -pakettiin kohdistui tietoturvapoikkeama [32], joka paljastaa useita npm-pakettien kaltaisiin avoimiin uudelleenkäytettäviin artefakteihin liittyviä ongelmia.

npm:n blogissa [32] kerrotaan, kuinka event-streamin, kuten monen muunkin paketin, kehitys perustui vapaaehtoisuuteen. Alkuperäinen kehittäjä halusi kuitenkin luopua paketin ylläpitovastuusta. Tämä mahdollisti tietoturvahyökkäyksen, jossa hyökkääjä otti paketin itselleen ylläpidettäväksi, minkä jälkeen hänen oli mahdollista suoraan käsitellä paketin lähdekoodia ilman rajoituksia. Haitallinen koodi ei kuitenkaan ollut event-stream -paketissa itsessään, vaan tämän varta vasten lisätyssä uudessa riippuvuudessa, eli käyttäjän näkökulmasta riippuvuuden riippuvuudessa [32]. Pakettien epäsuorat riippuvuudet, eli jonkin artefaktin hyödyntämät artefaktit, voivat monelta käyttäjältä helposti jäädä huomiotta. Esimerkiksi event-stream on 1738 paketin suora riippuvuus ja sillä on itsellään seitsemän suoraa riippuvuutta. Riippuvuusketju jatkuu molempiin suuntiin.

event-streamin tapauksessa hyökkäys oli kohdennettu tietynlaisiin ympäristöihin ja tarkoituksena oli saada tietoja kryptovaluuttatileistä, joten kaikille paketin käyttäjille ei hyökkäyksestä ollut merkittävää haittaa [32]. Mahdollisia kohteita oli kuitenkin huomattavan suuri määrä, kun event-streamin viikoittaiset latauskerrat ovat viimeisen vuoden aikana liikkuneet noin 1–2 miljoonan maastossa.

npm:n historiassa löytyy muitakin riippuvuusketjuista syntyneitä ongelmia. npm-pakettien ylläpitäjät voivat itse nimetä pakettinsa, mutta päällekkäisyyksiä voi helposti syntyä, kun pakettien määrät liikkuvat sadoissa tuhansissa (kuva 4.3). Vuonna 2016 kahden npm-paketin ylläpitäjän välille syntyi nimikiista, kun tunnetun tavaramerkin nimi esiintyi jo olemassa olevassa paketissa ja tavaramerkin omistaja näki tämän johtavan mahdollisiin sekaannuksiin [33].

npm:n julkaisema blogikirjoitus [33] kertoo, kuinka organisaatio antoi kiistassa päätöksensä tavaramerkin omistajan hyväksi. Seuraukset olivat arvaamattomat ja laajamittaiset, kun päätöksestä johtuen kiistan hävinnyt osapuoli poisti kaikki pakettinsa npm:stä, joista *left-*

[36] event-stream - npm, https://www.npmjs.com/package/event-stream, luettu 28.12.2019

pad nimisen paketin häviäminen aiheutti riippuvuusketjujen takia tuhansien projektien koonnin epäonnistumisen. Robbesin ja kumppanien [28] kuvaama heijastumisongelma tuli näkyviin, kun riippuvuusketjusta hävisi paketti. Hävinneestä paketista välittömästi riippuvien pakettien koonti ei enää ollut mahdollista, mikä edelleen heijastui riippuvuusketjussa eteenpäin [33]. Myös nopea yritys tilanteen korjaamiseksi paljasti ongelman, kun poistuneen paketin tilalla julkaistiin uudelleen identtinen paketti eri versionumerolla, mutta useat left-padista riippuvaiset paketit oli lukittu vanhempaan juuri poistettuun versioon. Uudelleenjulkaisu samalla versionumerolla oli tehty mahdottomaksi, mikä toisaalta turvaa normaalitilanteessa riippuvaisten pakettien kehitystä versioimattomia muutoksia estämällä. Jotta normaaliin toimintaan päästiin takaisin, jouduttiin turvautumaan varmuuskopion palautukseen [33].

Riippuvuuksissa havaittujen turvallisuusongelmien havaitsemisen helpottamiseksi paketinhallintatyökalut, kuten npm, voivat tarjota ratkaisuja. Esimerkiksi npm tarjoaa auditointityökalun[37], jolla pakettien käyttäjät ja ylläpitäjät voivat helposti auditoida ohjelmistonsa riippuvuudet mahdollisten tunnettujen turvallisuusongelmien varalta. Ei kuitenkaan ole tiedossa, kuinka nopeasti turvallisuusongelmat ovat työkalun kautta löydettävissä.

Riskejä ohjelmistoon syntyy myös ohjelmistojen kehittyvästä luonteesta. Kun kehitettävään ohjelmistoon tuodaan vierasta ohjelmakoodia erityisesti valmiiden komponenttien kautta, muodostuu riippuvuussuhde kolmanteen osapuoleen. Kimin ja Stohrin [18] mukaan uudelleenkäytön tulisi vähentää testaamisen tarvetta, sillä uudelleenkäytetyn koodin pitäisi olla laadukasta ja valmiiksi testattua. Vaikka uudelleenkäytöllä korvattua ohjelmakoodia ei tarvitsekaan testata, tuo uudelleenkäyttö mukanaan uusia testaustarpeita. Kuten Robbes ja kumppanit [28] totesivat ohjelmiston riippuvuuksiin kohdistuneet muutokset heijastuvat myös niitä käyttävään ohjelmistoon ja voivat aiheuttaa tilanteita, joissa ohjelmakoodista tulee toimimatonta. Näissä tapauksissa testaus on yksi tapa, jolla tilanteen aiheuttamat ongelmat voidaan havaita ajoissa.

[37] npm-audit — npm Documentation, https://docs.npmjs.com/cli/audit, luettu 26.12.2019

5 Yhteenveto

Tämä tutkielma käsitteli uudelleenkäyttöä ohjelmistokehityksessä, uudelleenkäytön muotoja ja sille asetettuja odotuksia ennen internetin yleistymistä ja peilasi näitä nykypäiväiseen ohjelmistokehitykseen, sekä siinä esiintyvään uudelleenkäyttöön. Uudelleenkäyttö koettiin 1980-luvulla erityisen lupaavaksi työkaluksi ratkaisemaan ohjelmistoalaa vaivaavaa kriisiä, joka ilmeni suurena kysyntänä, pitämättöminä aikatauluina ja kustannuksina, ohjelmistojen heikkolaatuisuutena sekä osaavien työntekijöiden riittämättömänä määränä.

Uudelleenkäytön ajateltiin ratkaisevan kriisin, koska uudelleenkäytön tutkijoiden mielestä useat ohjelmistot sisälsivät runsaasti näiden välillä toistuvaa ohjelmakoodia. Kehittäjäkunnassa havaittiin toisaalta vastakkaisia mielipiteitä, kun kehityksen alla olevia ohjelmistotuotteita pidettiin uniikkeina. Mikäli toistuvaa ohjelmakoodia voitaisiin hyödyntää uusien ohjelmistojen kehityksessä, olisi uusia ohjelmistoja mahdollista kehittää pienemmillä kustannuksilla, mutta silti laadukkaammin. Kustannukset pienenisivät ohjelmointityöhön vaaditun ajan ja siten kehittäjien määrän vähenemisen myötä ja laatu paranisi, sillä uudelleenkäytettävät artefaktit ovat ennalta testattuja.

Uudelleenkäytölle oli ennen internetin yleistymistä tavanomaista, että sitä hyödyntävät menetelmät olivat varsin ohjelmakoodilähtöisiä. Internetiä edeltävän ajan kirjallisuudessa esiintyvissä menetelmissä toistuivat eri laajuiset koodikomponentit ja niistä koostuvat kirjastot, sovellusgeneraattorit ja eri abstraktion tasolla kirjoitettavat ohjelmointikielet. Myös ohjelmakoodin taustalla olevat mallit ja ratkaisut tunnistettiin tuolloin oleellisiksi uudelleenkäytettäviksi artefakteiksi ja jopa määrittelyn uudelleenkäytön mahdollisuus tunnistettiin yhdessä teoksessa.

Internetin vaikutus ohjelmistokehitykseen ja uudelleenkäyttöön on ollut mullistava. Tiedon välitys ihmisten kesken sekä ohjelmistotason kommunikaatio ovat kietoutunut vahvasti internetin ympärille ja lähes kaikki tutkielmassa esitellyistä nykypäivän uudelleenkäytön menetelmistä hyötyvät siitä. Tästä mullistuksesta johtuen on tuskin järkevää odottaa, että internetiä edeltävänä aikakautena olisi voitu nähdä ennalta kaikkia mahdollisia uudelleenkäytön vaikutuksia.

Nykypäivänä uudelleenkäyttöä esiintyy laajemmin ohjelmistokehityksen eri vaiheissa. Ohjelmistokehitysprosessin vaiheista määrittelyyn, suunnitteluun, toteutukseen, käyttöönottoon ja ylläpitoon voidaan kaikkiin sitoa erilaisia uudelleenkäytön menetelmiä, joista useat

vaikuttavat moneen vaiheeseen. Jo vuosikymmeniä sitten todettujen menetelmien kaltaisia artefakteja hyödynnetään tänäkin päivänä runsaasti. Ulkoiset riippuvuudet, ohjelmistokehykset ja muuta kautta kehittäjätahon ulkopuolelta tullut ohjelmakoodi muodostaa tänä päivänä kehitetystä ohjelmistosta merkittävän osan. Tämä viittaisi siihen, että aiemmin esitetyt väitteet ohjelmistojen välillä vallitsevasta toistuvuudesta pitävät paikkansa. Ohjelmakoodin uudelleenkäyttö on jopa niin yleistä, että ohjelmistojen kehittäjillä ei välttämättä ole käsitystä uudelleenkäytetyn koodin määrästä omassa ohjelmistossaan. Ohjelmistoalalla vallitsee kulttuuri, jossa ohjelmistoon liitettyjen artefaktien määrästä ja sisällöstä ei olla edes järin kiinnostuneita.

Internetin myötä ohjelmistokehitys on saanut historiassa esitettyjen odotusten suhteen ristiriitaisia muotoja. Datan uudelleenkäyttö on normalisoitunutta, mutta esimerkiksi verkkorajapintojen kautta se on mahdollistanut täysin uudenlaista ohjelmistokehitystä, mikä taas kasvattaa kehitettyjen ohjelmistojen määrää. Monet muodot myös lupaavat kehityksen nopeutta ja parempaa laatua, mutta konkreettinen hyöty tuntuu silti jäävän oletuspohjalle. Myös ohjelmistoalan osaajista näyttäisi edelleen olevan pulaa. Uudelleenkäytön odotettiin helpottavan ohjelmistoalaa vaivaavaa kriisiä, mutta tämän toteutumisesta ei ole selkeää näyttöä ohjelmistoalan kasvaessa yhä mobiili- ja selainpohjaisten sovellusten ja muiden aluevaltausten myötä. Sen sijaan on olemassa viitteitä siitä, että kriisi oireilee edelleen.

Tarkastelu ja jatkotutkimusajatukset

Tämä tutkielma toteutettiin kirjallisuuskatsauksena ja prosessin myötä oli havaittavissa, että lähemmäs nykypäivää siirryttäessä soveltuvan kirjallisuuden määrä vähenee. Mahdollisia syitä tähän voi olla esimerkiksi se, että ohjelmistokehityksen ja uudelleenkäytön menetelmien laajentuessa myös ongelmakenttä laajenee ja koostavaa tutkimusta on hankalampi tehdä. Toisaalta uudelleenkäytöstä vaikuttaisi tulleen alalla normi ja tutkimuksen fokus on saattanut siirtyä.

Uudelleenkäytön odotusten toteutumisen arviointia hankaloitti se, ettei luotettavaa ja vertailukelpoista mittaustulosta saaduista hyödyistä löytynyt. Intuitiivisesti ajateltuna monet uudelleenkäytön menetelmät vaikuttavat tuovan merkittäviä etuja ohjelmistokehitykseen, mutta olisi perusteltua tehdä empiirisiä mittauksia semanttisesti yhtenevistä ohjelmistoista, jotka eroavat vain uudelleenkäytön hyödyntämisen suhteen.

Tässä tutkimuksessa nykypäiväisiä uudelleenkäytön muotoja käytiin läpi yksilötasolla

pintapuolisesti ja keskityttiin laajempaan kokonaisuuteen uudelleenkäytön tilan kuvaamiseksi. Myös tutkielmassa käsittelemättömiä nykypäivän uudelleenkäytön muotoja lienee paljon. Koska uudelleenkäyttö esiintyy tänä päivänä varsin monessa muodossa, olisi hyödyllistä perehtyä tarkemmin yksittäisiin muotoihin niiden haittojen ja hyötyjen kartoittamiseksi. Lisäksi tutkimuksen teemaa jatkaen voisi olla hyödyllistä tutkia, mihin uudelleenkäytön uskotaan tämänhetkisen tiedon valossa ohjelmistokehitystä vievän.

Kirjallisuus

[1] A. Balalaie, A. Heydarnoori ja P. Jamshidi. ”Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture”. *IEEE Software* 33.3 (toukokuu 2016), s. 42–52.

[2] R. Balzer. ”A 15 Year Perspective on Automatic Programming”. *IEEE Transactions on Software Engineering* SE-11.11 (marraskuu 1985), s. 1257–1268.

[3] F. Benitti ja R. da Silva. ”Evaluation of a Systematic Approach to Requirements Reuse”. *Journal of Universal Computer Science* 19.2 (2013), s. 254–280.

[4] T. Biggerstaff ja C. Richter. ”Reusability Framework, Assessment, and Directions”. *IEEE Software* 4.2 (maaliskuu 1987), s. 41–49.

[5] T. Coomer, J. Comer ja D. Rodjak. ”Developing reusable software for military systems, why it is needed, why it isn't working”. *ACM SIGSOFT Software Engineering Notes* 15.3 (heinäkuu 1990), s. 33–38.

[6] R. Darimont, W. Zhao, C. Ponsard ja A. Michot. ”Deploying a Template and Pattern Library for Improved Reuse of Requirements Across Projects”. Teoksessa: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. Syyskuu 2017, s. 456–457.

[7] D. Dig ja R. Johnson. ”How do APIs evolve? A story of refactoring”. *Journal of Software Maintenance and Evolution: Research and Practice* 18.2 (maaliskuu 2006), s. 83–107.

[8] J. Dongarra ja E. Grosse. ”Distribution of mathematical software via electronic mail”. *SIGNUM Newsl.* 20.3 (heinäkuu 1985), s. 45–47.

[9] F. M. A. Erich, C. Amrit ja M. Daneva. ”A qualitative study of DevOps usage in practice”. *Journal of Software: Evolution and Process* 29.6 (kesäkuu 2017).

[10] Facebook Inc. *React Native · A framework for building native apps using React.* 2020. URL: https://facebook.github.io/react-native/.

[11] N. Harrison ja P. Avgeriou. ”Pattern-Based Architecture Reviews”. *IEEE Software* 28.6 (marraskuu 2011), s. 66–71.

[12] B. Hartmann, S. Doorley ja S. Klemmer. ”Hacking, Mashing, Gluing: Understanding Opportunistic Design”. *IEEE Pervasive Computing* 7.3 (heinäkuu 2008), s. 46–54.

[13] E. Horowitz, A. Kemper ja B. Narasimhan. ”A survey of Application Generators”. *IEEE Software* 2.1 (tammikuu 1985), s. 40–54.

[14] E. Horowitz ja J. Munson. ”An Expansive View of Reusable Software”. *IEEE Transactions on Software Engineering* SE-10.5 (syyskuu 1984), s. 477–487.

[15] M. Huo, J. Verner, L. Zhu ja M. Babar. ”Software quality and agile methods”. Teoksessa: *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* Syyskuu 2004.

[16] T. Jones. ”Reusability in Programming: A Survey of the State of the Art”. *IEEE Transactions on Software Engineering* SE-10.5 (syyskuu 1984), s. 488–494.

[17] B. Kernighan. ”The Unix System and Software Reusability”. *IEEE Transactions on Software Engineering* SE-10.5 (syyskuu 1984), s. 513–518.

[18] Y. Kim ja E. Stohr. ”Software reuse: Issues and research directions”. Teoksessa: *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences.* Vol. 4. Tammikuu 1992, s. 612–623.

[19] C. Krueger. ”Software reuse”. *ACM Computing Surveys* 24.2 (kesäkuu 1992), s. 131–183.

[20] R. Land, I. Crnković, S. Larsson ja L. Blankers. ”Architectural Reuse in Software Systems In-house Integration and Merge – Experiences from Industry”. 3712 (2005), s. 123–139.

[21] R. Lanergan ja C. Grasso. ”Software Engineering with Reusable Designs and Code”. *IEEE Transactions on Software Engineering* SE-10.5 (syyskuu 1984), s. 498–501.

[22] T. Love. ”The economics of reuse (of software)”. Teoksessa: *Digest of Papers. COMPCON Spring 88 Thirty-Third IEEE Computer Society International Conference.* Helmikuu 1988, s. 238–241.

[23] R. Lämmel ja S. Jones. ”Scrap your boilerplate: a practical design pattern for generic programming”. Teoksessa: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation.* New York, NY, USA, tammikuu 2003, s. 26–37.

[24] N. Maiden ja A. Sutcliffe. ”Reuse of analogous specifications during requirements analysis”. Teoksessa: *Proceedings of the Sixth International Workshop on Software Specification and Design*. Lokakuu 1991, s. 220–223.

[25] T. Mikkonen ja A. Taivalsaari. ”Software Reuse in the Era of Opportunistic Design”. *IEEE Software* 36.3 (toukokuu 2019), s. 105–111.

[26] F. Ocariza, K. Pattabiraman ja A. Mesbah. ”Detecting Inconsistencies in JavaScript MVC Applications”. Teoksessa: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Toukokuu 2015, s. 325–335.

[27] C. Rich ja R. Waters. ”Automatic programming: myths and prospects”. *Computer* 21.8 (elokuu 1988), s. 40–51.

[28] R. Robbes, M. Lungu ja D. Röthlisberger. ”How do developers react to API deprecation?: the case of a smalltalk ecosystem”. Teoksessa: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE ’12. New York, NY, USA, marraskuu 2012.

[29] P. Schnitzhofer, F. Schnitzhofer ja R. Ramler. ”Tool Support for Reuse-Driven Elicitation and Specification of User Requirements”. Teoksessa: *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. Elokuu 2014, s. 267–268.

[30] V. Seppänen. ”Acquisition, organisation and reuse of software design knowledge”. *Software Engineering Journal* 7.4 (heinäkuu 1992), s. 238–246.

[31] A. Sillitti, T. Vernazza ja G. Succi. ”Service Oriented Programming: A New Paradigm of Software Reuse”. 2319 (huhtikuu 2002), s. 269–280.

[32] THE NPM BLOG. *Details about the event-stream incident*. Blog. 2018. URL: https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident.

[33] THE NPM BLOG. *kik, left-pad, and npm*. Blog. 2016. URL: https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm.

[34] TIOBE Software BV. *TIOBE Index for November 2019*. 2019. URL: https://www.tiobe.com/tiobe-index/.

[35] *webcomponents.org*. 2020. URL: https://www.webcomponents.org/.

[36] S. Xanthopoulos ja S. Xinogalos. ”A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications”. Teoksessa: *Proceedings of the 6th Balkan Conference in Informatics*. New York, NY, USA, syyskuu 2013, s. 213–220.

[37] M. Zimmermann, C.-A. Staicu, C. Tenny ja M. Pradel. ”Small World with High Risks: A Study of Security Threats in the npm Ecosystem”. Teoksessa: *Proceedings of the 28th USENIX Security Symposium*. Elokuu 2019, s. 995–1010.

[38] O. Zimmermann. ”Microservices tenets”. *Computer Science - Research and Development* 32 (2016), s. 301–310.