Teknisen velan ja henkilöriippuvuuksien hallinta end-to-end-testauksella

Turun yliopisto
Tietotekniikan laitos
Diplomityö
Ohjelmistotekniikka
Tammikuu 2025
Aku Lappalainen

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck-järjestelmällä.

TURUN YLIOPISTO
Tietotekniikan laitos

Aku Lappalainen: Teknisen velan ja henkilöriippuvuuksien hallinta end-to-end-testauksella

Diplomityö, 62 s.
Ohjelmistotekniikka
Tammikuu 2025

Monimutkaisten ja vanhojen ohjelmistojen ylläpidossa kehittäjien ymmärrys ohjelmiston toiminnasta on usein rajoittunut, koska alkuperäisiä dokumentaatioita voi olla puutteellisesti saatavilla ja koodin rakenne voi olla epäselvä tai huonosti dokumentoitu. Tämä voi tehdä virheiden jäljittämisestä ja järjestelmän laajentamisesta haasteellista. Tällöin yksittäisten kehittäjien syvällinen ymmärrys ohjelmiston toiminnasta voi olla hyvinkin merkityksellistä uusien ominaisuuksien tai ohjelmiston virheiden korjaamisen kannalta, sillä se mahdollistaa nopeamman ja tarkemman ongelmanratkaisun.
Tutkimuksen tarkoituksena on selvittää tarkemmin mikä on teknisen velan ja henkilöriippuvuuksien suhde, sekä tutkia pystytäänkö kattavilla end-to-end-testeillä vaikuttamaan teknisen velan ja henkilöriippuvuuksien hallintaan.
Tavoitteena on selvittää kirjallisuuslähteiden pohjalta, miten henkilöriippuvuus ja tekninen velka liittyvät toisiinsa, sekä kuinka kattavat end-to-end -testit vaikuttavat sovelluskehityksen teknisen velan ja henkilöriippuvuuksien hallintaan. Lisäksi tutkimus sisältää kyselytutkimuksen, jonka tarkoituksena on selvittää miten ohjelmistokehittäjät suhtautuvat end-to-end-testien käyttöön teknisen velan ja henkilöriippuvuuksien hallinnan välineenä.
Tarkasteltaessa teknisen velan ja henkilöriippuvuuden suhdetta suurin uhkatekijä on kirjallisuuskatsauksen mukaan teknisen velan aiheuttama kontekstin monimutkaistuminen. Monimutkaistuminen taas aiheuttaa merkittäviä regressiohaasteita. E2e-testien pääasiallinen hyöty tässä yhteydessä liittyy niiden dokumentointiominaisuuksiin. Ne kuvaavat tiettyjä järjestelmän kannalta kriittisiä polkuja, mutta eivät ole suoranaisesti teknisen velan hallinnan työkalu. Myös kyselytutkimuksessa e2e-testien hyötynä pidettiin niiden dokumentointiominaisuuksia, mutta niiden roolia teknisen velan hallinnassa ei pidetä kovin merkittävänä. Sen sijaan niiden merkitys korostuu henkilöriippuvuuksien vähentämisessä.

Asiasanat: end-to-end, ohjelmistotestaus, tekninen velka, henkilöriippuvuus, legacy-järjestelmä

Sisällys

1 Johdanto **1**

1.1 Ensimmäinen tutkimuskysymys 2

1.2 Toinen tutkimuskysymys 2

1.3 Tutkimuksen rakenne 3

2 Tekninen velka ja sen merkitys sovelluskehityksessä **5**

2.1 Mitä tekninen velka tarkoittaa 5

2.2 Teknisen velan vaikutukset sovelluskehitysprojekteihin 11

3 Henkilöriippuvuus sovelluskehitysprojekteissa **14**

3.1 Mitä tarkoittaa henkilöriippuvuus 14

3.1.1 Legacy henkilöriippuvuudessa 15

3.1.2 Monoliittisuus henkilöriippuvuudessa 17

3.1.3 Mikropalveluarkkitehtuuri henkilöriippuvuudessa 20

3.2 Henkilöriippuvuuden vaikutukset sovelluskehitysprojekteihin 21

3.3 Henkilöriippuvuuden suhde tekniseen velkaan 24

3.3.1 Henkilöriippuvuuden haasteet nykyhetkessä 24

3.3.2 Henkilöriippuvuuden haasteet tulevaisuudessa 25

3.3.3 Yhteeveto henkilöriippuvuuden suhteesta tekniseen velkaan . . 26

4 End-to-end testaus **29**

4.1 Mitä end-to-end testaus tarkoittaa 29
4.2 Kuinka end-to-end -testeillä voidaan hallita teknistä velkaa 32
4.3 Kuinka end-to-end -testeillä voidaan hallita henkilöriippuvuuksia . . . 35
4.4 Esimerkkejä end-to-end -testauksen hyödyistä ja vaikutuksista 40
4.5 Kustannustehokkuus ja end-to-end -testaus 45

5 Kyselytutkimus 47

5.1 Kysely end-to-end -testauksen vaikutuksista tekniseen velkaan ja henkilöriippuvuuteen 47
5.2 Analyysi end-to-end -testauksen vaikutuksista 51
5.2.1 Kokemukset e2e-testaamisesta 52
5.2.2 Teknisen velan hallinta e2e-testien kautta 53
5.2.3 Henkilöriippuvuuksien hallinta e2e-testien kautta 55
5.2.4 Ajatukset e2e-testaamisesta teknisen velan ja henkilöriippuvuuksien hallinnassa 58

6 Johtopäätökset ja jatkotutkimusaiheet 60

6.1 Yhteenveto tutkimustuloksista ja niiden merkitys 60
6.2 Mahdolliset jatkotutkimusaiheet ja suositukset tuleville tutkimuksille 62

Lähdeluettelo 63

Kuvat

2.1 Teknisen velan nelikenttä [5] 8

3.1 Monoliittisten projektien tuottavuus laskee monimutkaisuuden kasvaessa, kun taas mikropalveluarkkitehtuurin tapauksessa lasku on maltillisempaa [21] 16

3.2 Monoliittisen- ja mikropalveluarkkitehtuurin kuvaus 18

3.3 Pariutumisen ja koheesion ero 19

4.1 Miten eri testit eroavat toisistaan [45] 31

4.2 Playwrightin näkymä, jossa punaisella merkattuna järjestelmään tallentuneen kuvankaappauksen ja uusien muutosten jälkeen tulleet erot. 38

4.3 Playwrightin näkymä, jossa kehittäjä pystyy vertaamaan liukusäätimen avulla järjestelmään tallentuneen kuvankaappauksen ja uusimman version eroja. 39

4.4 E2e-testin kuvakaappaukset kertovat kehittäjälle miten loppukäyttäjän kuuluisi ohjautua käyttöliittymässä ja miltä näkymien pitäisi näyttää 41

4.5 Kuvankaappausten lisäksi kehittäjä pystyy tulkitsemaan haluttua käyttäjän liikehdintää testien koodista 42

4.6 Eri testityyppien ajamiseen vaadittava aika kasvaa suhteessa niiden järjestelmäkattavuuteen [59] 43

4.7 Yksikkötesteissä ominaisuuden ulkopuoliset osat korvataan niitä jäljittelevillä testiosilla [59] 43
4.8 Testaamisen hinta [60] 45

5.1 Koetko, että e2e-testien sisällyttäminen projektiin olisi hidastanut tai nopeuttanut kehitystä? 54
5.2 Koetko, että e2e-testien ylläpito vaatii enemmän vaivaa verrattuna matalamman tason testeihin, kuten yksikkötesteihin? 54
5.3 Koetko, että e2e-testit tukevat teknisen velan hallintaa, erityisesti koodin refaktoroinnin yhteydessä? 55
5.4 Koetko, että e2e-testit ovat tärkeämmässä roolissa teknisen velan hallinnassa verrattuna matalamman tason testeihin, kuten yksikkötesteihin? 55
5.5 Koetko, että e2e-testit helpottavat järjestelmän toiminnan kannalta kriittisten polkujen toiminnan ymmärtämistä? 56
5.6 Koetko, että e2e-testit helpottavat uusien kehittäjien sisäänajamista projektiin? 56
5.7 Koetko, että e2e-testejä voidaan käyttää dokumentoimaan tiettyjä järjestelmän toiminnan kannalta kriittisiä polkuja? 57
5.8 Koetko, että e2e-testeillä voidaan korvata perinteinen dokumentaatio, jossa ominaisuuksien toiminta kuvataan tekstimuodossa? 57
5.9 Koetko, että voisit toteuttaa projektiin e2e-testejä sillä näkökulmalla, että niiden tarkoitus olisi hallita teknistä velkaa? 58
5.10 Koetko, että voisit toteuttaa projektiin e2e-testejä sillä näkökulmalla, että niiden tarkoitus olisi hallita henkilöriippuvuuksia? 59

Taulukot

2.1 Teknisen velan maksamiseen käytetyn ajan osuus [1] 12

4.1 Testaamisen tyypit [43] 30

5.1 Käytetyt testausmenetelmät ja -tyylit 52

5.2 E2e-testauksen hyödyt 53

1 Johdanto

Ohjelmistojen ikääntyessä ja kasvaessa yhä monimutkaisemmaksi muuttuu niiden ylläpito entistä hankalammaksi. Nopeasti muuttuvat alan standardit saattavat altistaa ohjelmistot sille, että niitä joudutaan muokkaamaan poikkeamaan huomattavastikin alkuperäisestä määrittelystä. Lisäksi uudet ominaisuudet ja rajallinen budjetti voivat aiheuttaa järjestelmän turhan teknisen monimutkaistumisen ja teknisen velan kerryttämiseen. Tekninen velka on omiaan entisestään monimutkaistamaan järjestelmän koodia sekä toimintaa.

Monimutkaisten ja vanhojen ohjelmistojen ylläpidossa kehittäjien ymmärrys ohjelmiston toiminnasta on usein rajoittunut, koska alkuperäisiä dokumentaatioita voi olla puutteellisesti saatavilla ja koodin rakenne voi olla epäselvä tai huonosti dokumentoitu. Tämä voi tehdä virheiden jäljittämisestä ja järjestelmän laajentamisesta haasteellista. Tällöin yksittäisten kehittäjien syvällinen ymmärrys ohjelmiston toiminnasta voi olla hyvinkin merkityksellistä uusien ominaisuuksien tai ohjelmiston virheiden korjaamisen, sillä se mahdollistaa nopeamman ja tarkemman ongelmanratkaisun. Tutkimuksen tarkoituksena on selvittää tarkemmin, että mikä on teknisen velan ja henkilöriippuvuuksien suhde, sekä tutkia pystytäänkö kattavilla end-to-end testeillä vaikuttamaan teknisen velan ja henkilöriippuvuuksien hallintaan.

Tämän tutkimuksen tarkoituksena on vastata seuraaviin tutkimuskysymyksiin:

TK1: Miten henkilöriippuvuus ja tekninen velka liittyvät toisiinsa?

TK2: Kuinka kattavat end-to-end -testit vaikuttavat sovelluskehityksen teknisen velan ja henkilöriippuvuuksien hallintaan?

1.1 Ensimmäinen tutkimuskysymys

Teknistä velkaa synnyttää useimmiten ominaisuudet, jotka ovat joko puutteellisen osaamisen, rajallisen ajan sekä rahan tai huolimattomuuden seurauksena toteutettu poiketen parhaasta mahdollisesta tavasta. Tässä tutkimuksessa tarkastellaan, miten teknisen velan kertyminen vaikuttaa järjestelmän henkilöriippuvuuksiin. Esimerkki tällaisesta tilanteesta on kun järjestelmän ominaisuus, joka on tehty teknistä velkaa kerryttäen päätyy henkilölle muokattavaksi tai ylläpidettäväksi, jolla ei ole kokemuksen kerryttämään kontekstiosaamista kyseisestä ominaisuudesta tai jopa koko järjestelmästä. Tällöin saatetaan tehdä virheitä, joilta olisi voitu välttyä, jos niitä olisi ollut tekemässä joku, jolla on tarpeeksi ymmärrystä järjestelmästä.

Tutkimuksessa tarkastellaan teknisen velan ja henkilöriippuvuuksien välistä suhdetta kirjallisuuslähteiden pohjalta. Tavoitteena on selvittää, mitkä tekijät johtavat teknisen velan syntyyn ja mitkä tekijät voivat pahentaa sitä. Lisäksi tarkastellaan teknisen velan vaikutuksia sovelluskehitysprojekteihin. Samalla tutkitaan kirjallisuuslähteiden pohjalta henkilöriippuvuuksien syntymekanismeja ja niiden vaikutuksia kehitysprosessin sujuvuuteen.

1.2 Toinen tutkimuskysymys

Teknistä velkaa syntyy sovelluskehityksen yhteydessä vääjäämättä. Vaikka kehitysvaiheessa ei varsinaisesti velkaa kerrytettäisikään, vanhenee järjestelmä ajan myötä.

Tällöin esimerkiksi kolmannen osapuolen kirjastojen päivittäminen aiheuttaa sen, että järjestelmään syntyy teknistä velkaa vanhentuneiden kirjastojen puolesta. Ajoittain nämä saattavat vaikuttaa siihen, että koodiin joudutaan tekemään muutoksia, joita saatetaan tehdä pienellä budjetilla muiden töiden ohessa. Lisäksi pienten ylläpitopäivitysten tekeminen saattaa siirtyä jollekin, jolla ei ole kontekstiosaamista projektista. Tällaisessa tapauksessa saatetaan päätyä tilanteeseen, jossa järjestelmään tehdään muutoksia, jotka päällisin puolin vaikuttavat järkeviltä, mutta todellisuudessa se rikkoo joitain järjestelmän ominaisuuksia.

Tutkimuksen tarkoituksena on selvittää kirjallisuuslähteiden pohjalta miten end-to-end testit eroavat matalamman tason testaustyyleistä, kuten yksikkötesteistä ja miten niillä voidaan hallita teoriatasolla teknistä velkaa sekä henkilöriippuvuuksia. Tutkimus käy myös läpi esimerkkejä end-to-end testauksesta ja tarkastelee niiden kustannustehokkuutta verrattuna muihin teknisen velan ja henkilöriippuvuuden hallintakeinoiin.

Tutkimuksessa toteutettiin myös kyselytutkimus, jonka tarkoituksena oli selvittää miten ohjelmistokehittäjät suhtautuvat end-to-end testien käyttöön teknisen velan ja henkilöriippuvuuksien hallinnan välineenä. Kyselyn kysymykset kohdistuivat kehittäjien kokemuksiin ja käytäntöihin liittyen testauksen kattavuuteen, teknisen velan syntyyn ja henkilöriippuvuuksien hallintaan testauksen keinoin.

1.3 Tutkimuksen rakenne

Tutkimus pyrkii avaamaan lukijalle kirjallisuuslähteiden pohjalta mitä TK1 ja TK2 kannalta olennaiset termit tekninen velka, henkilöriippuvuus ja end-to-end-testaus tarkoittavat. Terminologian avaamisen lisäksi kappaleet 2, 3 ja 4 tarkastelevat kirjallisuuslähteden pohjalta, miten edellämainitut tekijät ovat yhteydessä toisiinsa.

Kappaleen 2 tarkoitus on avata lukijalle mitä tekninen velka on, miten sitä syntyy ja mitä eri vaikutuksia sillä on sovelluskehitysprojekteihin.

Kappale 3 käsittelee henkilöriippuvuutta. Kirjallisuuslähteiden avulla käsitellään henkilöriippuvuuden syntyyn vaikuttavia tekijöitä, vaikutuksia sovelluskehitysprojekteihin sekä suhdetta tekniseen velkaan.

Kappale 4 tarkastelee end-to-end-testien roolia sovellusten testausprosessissa. Kirjallisuuslähteiden avulla pyritään selvittämään miten end-to-end-testeillä voidaan hallita teknistä velkaa ja henkilöriippuvuuksia. Kappale käsittelee myös end-to-end-testaamisen hyötyjä, sekä sivuaa testausmenetelmän kustannustehokkuutta verrattuna muihin testausmenetelmiin.

Kappaleessa 5 toteutettiin TK2:n tueksi kyselytutkimus, jonka tarkoituksena oli selvittää kehittäjien suhtautumista end-to-end testien käyttöön teknisen velan ja henkilöriippuvuuksien hallinnan välineenä.

Kappalessa 6 tiivistetään tutkimuksen keskeiset havainnot TK1 ja TK2 näkökulmista hyödyntäen sekä kirjallisuusosion että kyselytutkimuksen tuloksia. Kappale käsittelee tulosten lisäksi myös mahdollisia jatkotutkimusaiheita.

2 Tekninen velka ja sen merkitys sovelluskehityksessä

2.1 Mitä tekninen velka tarkoittaa

Ohjelmistokehityksessä teknisellä velalla tarkoitetaan käsitettä, jossa kehityksen aikana ohjelmistoon tehdään ratkaisuja, jotka säästävät rahaa lyhyellä aikavälillä, mutta pitkällä aikavälillä nostavat kokonaiskustannuksia.[1] Terminä tekninen velka on vakiintunut vuonna 1992 Ward Cunnighamin kirjoittamaan artikkeliin pohjautuen. [2] Cunningham kuvaili artikkelissa WyCash+-ohjelmiston kehitystä, jossa tuotetta kehitettiin prototyypistä vaiheittain lopulliseksi tuotteeksi. Kun ohjelmistoa kehitetään, pystytään prosessia nopeuttamaan tekemällä ratkaisuja, jotka koodin kannalta eivät ole laadullisesti hyvällä tasolla, mutta tällä saadaan tuotettua toimivia kokonaisuuksia asiakkaalle. Tällöin kerrytetään teknistä velkaa. WyCash+ oli talouteen liittyvä ohjelmisto ja Cunningham koki asiayhteyteen sopivaksi analogiaksi käyttää termiä “velka” (engl. debt) kuvaamaan ohjelmiston kehityksen aikana tehtyjä päätöksiä. Cunninghamin mukaan velan kerryttäminen ohjelmiston kehityksen aikana nopeuttaa kehitystä, kunhan se maksetaan takaisin. Käytännössä velan takaisinmaksulla tarkoitetaan sitä, että koodi käydään kokonaisuutena läpi ja aikaisemmin toteutetut osat sovitetaan luomaan järkevä yhtenäinen, koodillisesti laadukas kokonaisuus. WyCash+-ohjelmiston kohdalla, vanhempia osioita ohjelmistosta, jotka

olivat jo olleet käytössä, käytiin useaan otteeseen muokkaamassa vastaamaan ohjelmiston senhetkistä tarvetta. Tällöin saatiin luotua olemassaolevista ohjelmiston osista yhtenäisiä kokonaisuuksia ja parannettua koodia laadullisesti.

Vaikka Cunningham käsittelee teknistä velkaa lähinnä koodin näkökulmasta, on tekninen velka käsitteenä laajentunut sitten Cunninghamin vuonna 1992 kirjoittaman artikkelin. Nykyään teknisen velan käsitteen alle voidaan alakäsitteinä listata testivelka, henkilövelka, arkkitehtuurivelka, vaatimusvelka ja dokumentaatiovelka [3].

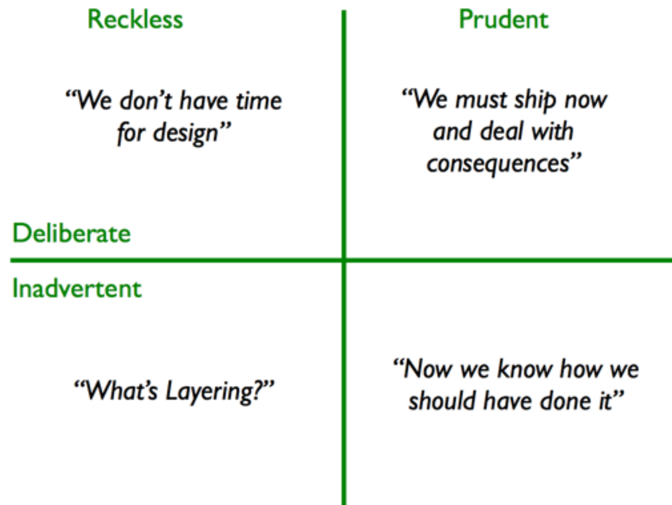
Teknistä velkaa voidaan käytännön esimerkillä verrata jonkin mekaanisen laitteen huoltoon. Mikäli laitetta ei huolleta säännöllisesti, alkavat sen kuluneet osat kuormittaa laitetta entisestään, kunnes saavutaan pisteeseen, jossa laite hajoaa. Sovelluskehityksessä kuluminen ei ole ihan yhtä lineaarista mitä mekaanisissa laitteissa, mutta samaa ajatusmaailmaa pystytään käyttämään myös siinä. Mikäli koodikanta pitää sisällään esimerkiksi nopealla aikataululla toteutettuja ominaisuuksia, joita on tuotettu vastaamaan nopeaa kysyntää asiakkaan suunnalta, alkaa koodin laatu ajan saatossa heikkenemään. Mitä laajempiin ohjelmistokokonaisuuksiin mennään, sitä enemmän niissä on ominaisuuksia, jotka vaativat muita ominaisuuksia toimiakseen. Mikäli näitä ominaisuuksia on lisätty järjestelmään tavalla, joka ei ota huomioon muita olemassa olevia ominaisuuksia tai mahdollisesti tulevaisuudessa vastaan tulevia muutoskohteita, kertyy järjestelmään teknistä velkaa. Tämä tilanne johtaa uusien ominaisuuksien tai muokkausten tekemisen vaikeutumiseen järjestelmässä. Kun muutosten tekeminen vaikeutuu, se nostaa myös muutostöiden kustannuksia. Tässä yhteydessä puhutaan teknisen velan kasvusta. Mikäli teknistä velkaa ei makseta pois, se kasvaa korkoa, mikä puolestaan lisää velan määrää entisestään. Seuraavassa suora lainaus Ward Cunninghamin artikkelista:

> “Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. The

> danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation.”

Kuten edellä on mainittu, voidaan teknisen velan ottamista käyttää hyödyksi ohjelmistokehityksessä, jotta saadaan nopeutettua kehitystä. Tahallinen teknisen velan ottaminen voidaan jakaa kahteen eri pääkategoriaan, lyhyen ja pitkän aikavälin velkaantumiseen. Lyhyen aikavälin velkaantuminen on reaktiivista, eli päätös ottaa velkaa tehdään tietoisesti tiettyä tarkoitusta varten. Lyhyen aikavälin velkaantuminen voidaan myös jakaa kohdistettuun ja hajanaiseen lyhytaikaiseen velkaantumiseen. Kohdistetulla tarkoitetaan sitä, että toteutetaan yksittäinen, tunnistettava pikaratkaisu ja hajanaisella sitä, että on tehty useita pieniä pikaratkaisuja tietyn tavoitteen saavuttamiseksi. Esimerkkinä tahallisesta lyhyeen aikajänteen velasta olisi kovakoodata tiettyyn ohjelmiston näkymään käännökset kahdella eri kielellä, sen sijaan, että otettaisiin käyttöön käännöksiä varten suunniteltu käännöskirjasto. Hajanaisen tästä velasta tekisi se, että järjestelmässä olisi useita näkymiä, jotka pitäisi kääntää usealle eri kielelle. Hajanaisessa ratkaisussa tiedostetaan, että se kasvattaa enemmän velkaa ja lopullisen ratkaisun eli velan hinta kasvaa enemmän mitä lyhyen aikavälin kohdistetussa ratkaisussa. [4]

Martin Fowler jakaa artikkelissaan teknisen velan käsitteen nelikentäksi, joka koostuu kahdesta ulottuvuudesta: holtiton/järkevä (engl.reckless/prudent) ja tahallinen/tahaton (engl. deliberate/inadvertent) (Kuva 2.1). Fowlerin nelikenttämallissa velan kertymistä pohditaan siltä kantilta, että onko velka kertynyt tahallisesti vai tahattomasti ja, että onko kyseinen ratkaisu holtiton vai järkevä. Järkevien ja tahallisten ratkaisuiden voidaan olettaa olevan suunnitelmallisia ja niissä on otettu huomioon se, että velka tulee maksaa takaisin. Mikäli ratkaisun vaikutuksia tulevaisuuteen ei oteta huomioon, voidaan tahallinenkin ratkaisu määritellä holtittomaksi.



Kuva 2.1: Teknisen velan nelikenttä [5]

Mikäli teknistä velkaa otetaan tahattomasti, mutta sen vaikutukset pystytään huomiomaa ja oppimaan niistä, voi tahaton velka olla tällöin järkevää. [5]

Tahallinen pitkän aikavälin velkaantumimen syntyy ennakoivasti strategisista syistä. Tällainen voi olla esimerkiksi alkujaan tiukka projektiaikataulu, jolloin päädytään tietoisesti jättämään kattavien testitapausten kirjoittaminen tekemättä. Tällöin saatetaan saada lopputulos aikaisemmin valmiiksi, mutta tietoisesti ollaan kasvatettu järjestelmän teknistä velkaa. Teknisen velan syntyminen ei aina ole edellä mainitun esimerkin tapaan tahallista. Tahatonta teknistä velkaa syntyy esimerkiksi puutteellisesta määrittelystä, sovellusarkkitehtuurillisista päätöksistä, kontekstiymmärryksen tai ohjelmointitaitojen puutteesta. Tahaton tekninen velka ei välttämättä kuitenkaan johdu poikkeuksetta huonoista valinnoista. Ohjelmiston ikääntyessä voi käydä niin, että esimerkiksi ohjelmistokehys, joka on sovellusta määriteltäessä ollut hyvä, ei enää saakaan päivityksiä ja täten ei enää sovellu ajamaan kaikkia järjestelmän tulevia tarpeita. [6] Tällöin ei voida käyttää esimerkiksi uusimpia kirjastoja, joiden avulla kehittäminen nopeutuu vaan joudutaan tekemään enemmän töitä, jotta uudet ominaisuudet saadaan toteutettua. Huonosti kirjoitettu koodi tulisi kuitenkin erottaa käsitteenä teknisestä velasta. [7]

Tekninen velka ei myöskään liity pelkkään järjestelmän koodikantaan. Velkaa kerryttää myös kaikki järjestelmän ympärillä tapahtuvat asiat, joita on projektin hallinnasta aina prosessiautomaatioon. [8] Esimerkki tällaisesta on esimerkiksi CI/CD (Continuous Integration/Continuous Delivery/Deployment). Continuous Integrationilla (suom. jatkuva kehitys) tarkoitetaan prosessia, jossa joka kerta kun koodia lisätään yhteiseen versionhallintaan kuten Git:iin, CI-järjestelmä kuten CircleCI tarkistaa ohjelmiston toimivuuden ajamalla automaattisesti testejä, kääntämällä ohjelmiston (engl. compile software) ja tarkistamalla sen toimivuuden. Continuous Delivery/Deployment (suom. jatkuva julkaisu) huolehtii siitä, että ohjelmistoprosessi on automatisoitu niin, että CI:n vahvistama toimiva koodi saadaan julkaistua automatisoidusti ilman manuaalisia toimenpiteitä kehittäjältä. Toimivan CI/CD-putken pystyttämättä jättämällä saatetaan säästää aikaa lyhyellä aikavälillä, mutta pitkässä juoksussa se aiheuttaa erinäisiä ongelmia. Ilman toimivaa CI/CD-putkea kehittäjä joutuu manuaalisesti varmistamaan koodin toimivuuden ja tekemään tarvittavat toimenpiteet ohjelmiston päivittämiseksi tuotantoon. Tämä lisää riskiä inhimillisille virheille ja hidastaa päivitysten tuotantoonvientiä. Lisäksi manuaalinen päivittäminen vaatii enemmän aikaa ja resursseja, mikä taas nostaa kehityksen kokonaiskustannuksia ja vähentää yrityksen kykyä reagoida nopeasti markkinamuutoksiin. [9]

Gonzalez-Barahona ym. (2017) tutkimuksessa luoma teoreettinen malli, “technical lag”, luotiin mittaamaan sitä kuinka vanhentunut järjestelmä on. Tästä eteenpäin “technical lag”:iin viitataan termillä tekninen viive.

Kuten aiemmin mainittua, moni tuotannossa oleva järjestelmä käyttää ilmaisia, avoimen lähdekoodin ohjelmia, joihin viitataan usein lyhenteellä FOSS (free, open source software). Synopsys tarkasteli vuonna 2023 1703 kaupallisen ohjelmiston koodikantaa 17 eri toimialalta, minkä havaintona oli, että 96% näistä sisälsi avoimen lähdekoodin ohjelmia. [10] Yksinkertaistaen näihin viitataan jatkossa kirjastoina.

Gonzalez-Barahona ym. erottavat kuitenkin teknisen velan ja teknisen viiveen toisistaan käsitteinä sillä perusteella, että teknisen velan määritelmän mukaan teknistä velkaa on heidän mukaan jokin asia, jota ei alunperin ole tehty parhaalla mahdollisella tavalla, vaan jätetty sellaiseksi, joka vaatii jatkokehitystä. Teknisessä viiveessä kuitenkin järjestelmän laatu heikkenee esimerkiksi haavoittuvuuksien myötä ajan saatossa eikä siksi, että on tehty päätöksiä, jotka heikentävät tuotteen laatua mikäli niiden aiheuttamaa velkaa ei makseta takaisin. [11]

Itse koen kuitenkin, että tietyissä tilanteissa tekninen viive voidaan lukea myös tekniseksi velaksi. Kuten aiemmin Fowlerin nelikenttämallissa kävi ilmi, on tekninen velka jaettavissä useampaan eri kategoriaan eikä se ole aina yhtä yksiselitteistä. Vaikka teknistä viivettä syntyy niin ikään itsestään ajan kuluessa, voidaan teknisen viiveen jatkuva huomiotta jättäminen mielestäni lukea tekniseksi velaksi. Tällainen on esimerkiksi tilanne, jossa järjestelmä on ylläpitovaiheessa ja siihen ei tehdä mitään aktiivista kehitystä. Kuitenkin järjestelmän käyttämät kirjastot saavat jatkuvasti päivityksiä, jotka sisältävät esimerkiksi tietoturva ja bugikorjauksia. Mikäli projektia ylläpitävä taho ei reagoi näihin päivityksiin päivittämällä järjestelmän käyttämiä kirjastojen versioita, vaikeutuu järjestelmän ylläpito tulevaisuudessa. Tällöin voidaan joutua tilanteeseen, jolloin johonkin järjestelmän toimintojen kannalta merkittävään kirjastoon, tässä esimerkissä fiktiiviseen kirjastoon k_1 tulee kriittinen haavoittuvuus, joka on heti paikattava järjestelmän tietoturvan takia. Mikäli kirjasto k_1 on riippuvainen kirjastosta k_2 ja järjestelmässä on käytössä myös kirjasto k_3, joka taas vaatii toimiakseen eri version kirjastosta k_2, voidaan päätyä tilanteeseen, jossa järjestelmän oleellisia toiminnallisuuksia hajoaa, kun kirjastot k_1 ja k_2 päivitetään, sillä kirjasto k_3 vaatii vanhemman version kirjastosta k_2 toimiakseen. Pahimmillaan kirjastojen välisiä riippuvuuseroja ei päivitysvaiheessa edes huomata vaan ongelma ilmenee vasta kun järjestelmän päivitykset menevät loppukäyttäjille. Mikäli kirjastoja päivitetään aktiivisesti, edellämainittujen tilanteiden syntyminen

on harvinaisempaa ja vähintäänkin vältetään useiden eri kirjaston versioiden yli päivittämistä. [12] [13] Kolmannen osapuolen kirjastoihin tehdyt tietoturvaselkkaukset ovatkin yleistyneet ja niistä on jopa koitunut uhkia kriittiselle infrastruktuurille.[14] [15] Täten voidaan päätellä, että ohjelmiston riippuvaisuus tietystä kirjaston versiosta ilman, että sitä pystytään teknisen velan takia päivittämään on myös merkittävä tietoturvauhka.

2.2 Teknisen velan vaikutukset sovelluskehitysprojekteihin

Kaikissa sovelluskeskeisissä järjestelmissä on jonkin verran teknistä velkaa riippumatta niiden alasta tai koosta. Tekninen velka koostuu useista eri tekijöistä, joita ovat esimerkiksi lyhyet koodinpätkät tai testitapaukset, joita on järjestelmään lisätty kumulatiivisesti. Tekninen velka ei ole atomista tai monoliittista (atominen tarkoittaa yksittäisiä, irrallaan muusta olevia ominaisuuksia ja monoliittinen sitä, että asiat on kytketty toisiinsa) vaan koostuu useista asioista, joita voidaan kutsua teknisen velan osiksi.

Teknisen velan vaikutus sovelluskehitykseen on monitasoista. Mikäli teknistä velkaa kerrytetään eikä makseta ajallaan pois on lopputuloksena se, että kehittämiseen käytettävä aika kasvaa. Teknisen velan kertyminen vaikuttaa negatiivisesti koodikannan monimutkaisuuteen ja täten tekee kehittämisestä vaikeampaa. [16] Yritysmaailmassa kehittämiseen käytetty aika korreloi useasti myös sen kanssa paljonko kehittämiseen kuluu rahaa. Besker ym.(2022) mukaan 36% sovelluskehitykseen käytetystä ajasta tuhlataan teknisen velan koron maksamiseen. Tutkimuksessa kävi myös ilmi, että haastatellut kehittäjät käyttivät eniten tästä ajasta ongelmien ymmärtämiseen ja niiden arvioimiseen. (Taulukko 2.1) [1]

Velan maksamisen konkretisoituminen saattaa venyä, koska tekninen velka ker-

Taulukko 2.1: Teknisen velan maksamiseen käytetyn ajan osuus [1]

Activity	***Mean***	***Score 6 (highest)***
Understanding and/or measuring the issues	4.38	20.63 %
Management process of the issues (track, monitor, communicate)	3.89	22.98 %
Finding the issues	3.69	19.68 %
Refactoring the issues	3.59	15.35 %
Deciding which issue to refactor first	2.89	2.05 %
Other	2.60	15.15 %

tyy optimitilanteessa toimivista ratkaisuista eikä sen olemassaolo ilmene loppukäyttäjälle mitenkään. Teknistä velkaa maksettaessa ei järjestelmän toimintojen tulisi muuttua loppukäyttäjän näkökulmasta mitenkään, sillä oletusarvona teknistä velkaa kerryttävä toteutus toimii määrittelyiden mukaisesti, mutta vaatii refaktorointia ohjelmiston ylläpidettävyyden takia. Tämän takia on tärkeää, että teknisen velan ottamisessa ollaan läpinäkyviä koko projektitiimin kesken ja projektista vastuussa olevat henkilöt, kuten projektipäälliköt ovat tietoisia siitä, että toteutetut ratkaisut tulee käydä tulevaisuudessa läpi. Etenkin ongelmallisen tilanteesta tekee se, että projekti on toteutettu esimerkiksi ulkoiselle yritykselle, joka vastaa projektin rahoituksesta. Tällöin haasteita asettaa se, miten saadaan perusteltua refaktoroinnin vaatimat lisäkustannukset ilman, että se vaikuttaa mitenkään loppukäyttäjän kokemukseen järjestelmästä.

Sen lisäksi, että tekninen velka vaikuttaa negatiivisesti koodin laatuun ja sen kehitysmahdollisuuksiin, on teknisellä velalla myös merkitystä työntekijän moraaliin. Vuonna 2020 julkaistussa tutkimuksessa tarkasteltiin teknisen velan ja sen hallinnan vaikutusta työntekijöiden moraaliin ja täten tuottavuuteen. Tutkimuksessa havaittiin, että haastateltavat sovelluskehittäjät kokivat teknisen velan läsnäolon lisäävän tuhlattua aikaa ja resursseja, joka taas vaikutti negatiivisesti kehittäjien moraaliin.

> "As discussed earlier, almost all our interviewees and respondents to the 5-point Likert Scale survey mentioned that the occurrence of TD hinders

their progress and their future activities. Several interviews discussed the amount of development time and resources that are wasted because of the TD accumulated in their products. Therefore, we decided to investigate any potential correlation between the amount of waste that occurs due to TD and developers' morale." [17]

3 Henkilöriippuvuus sovelluskehitysprojekteissa

3.1 Mitä tarkoittaa henkilöriippuvuus

Henkilöriippuvuudella tarkoitetaan mittaria, jolla tarkastellaan sitä miten riippuvainen projektin eteneminen on tietyistä tekijöistä. Mittari kuvaa projektin resilienssiä yhtäkkiselle tilanteelle, jossa projektin kehittäjät vaihtuvat. Englanninkielisissä tutkimuksissa mittariin viitataan termeillä bus factor ja truck factor. Suorana käännöksenä näillä tarkoitetaan sitä, kuinka monta projektin tekijää voivat jäädä bussin alle ennenkuin projektin eteneminen keskeytyy. [18] Pahimmillaan voidaan olla tilanteessa, jossa koko projektin kehitys ja ymmärrys on yhden kehittäjän takana. Tällöin luonnollisesti todennäköisyys merkittävän kehittäjän poisjäämiselle projektista on suurempi kuin isommissa projektiryhmissä. Vuoden 2016 ISBSG dataan perustuen ohjelmistokehitystiimien keskimääräinen kehittäjämäärä on 7,9 henkilöä ja mediaani 5 henkilöä [19]. Tästä voidaan päätellä, että tilanne, jossa koko projekti on pelkästään yhden henkilön kehitettävänä on melko harvinainen.

Vaikka henkilöriippuvuudella terminä viitataan kirjallisuudessa yleisesti “bus factor”-mittariin, jossa projektin eteneminen pysähtyy projektin avainhenkilöiden äkkinäisen poisjäämisen seurauksena, käsitellään tässä henkilöriippuvuutta myös projektin kontekstiymmärryksen kautta. Kontekstiymmärryksellä tässä yhteydessä

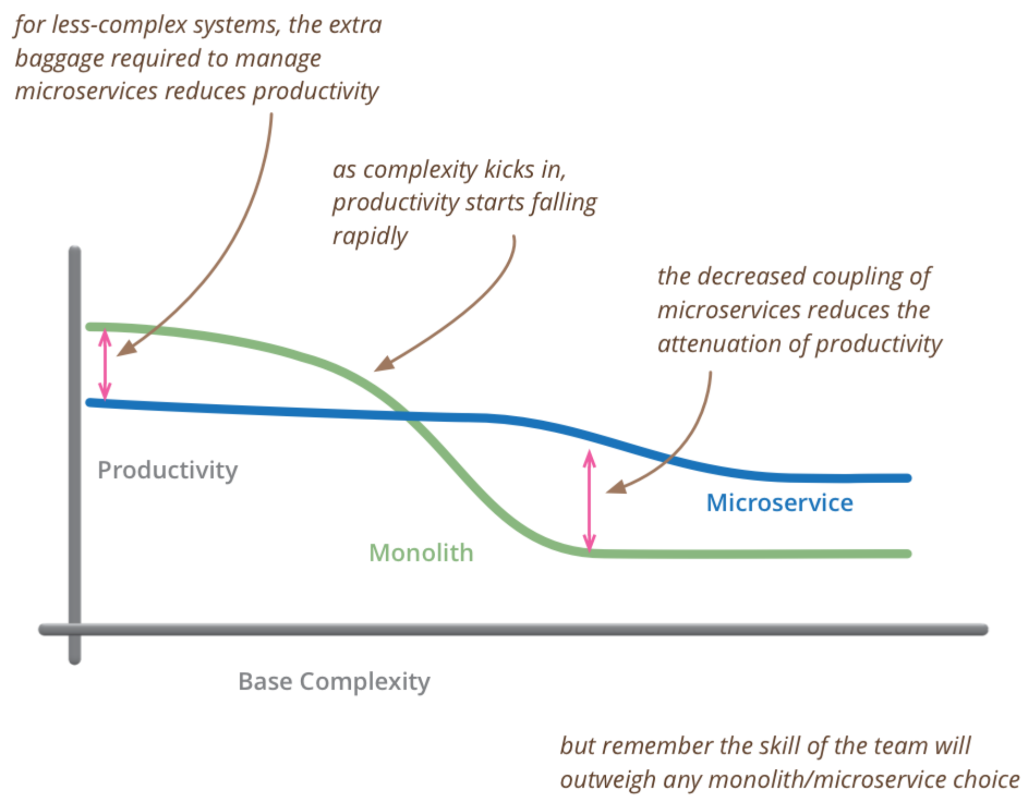
tarkoitetaan kehittäjän tai asiakkaan henkilöstön ajan saatossa luomaa ymmärrystä järjestelmän toimintaa ja integraatioita kohtaan kokonaisuutena.

Sovelluskehityksessä eniten henkilöriippuvuutta kerryttävät legacy- ja monoliittiprojektit[20]. Näitä molempia konsepteja yhdistävä tekijä henkilöriippuvuuden kerryttämisessä on monimutkaisuus ja monoliittisten projektien tapauksessa koodin suuri määrä.

Monoliittisen arkkitehtuurin vastakohtana voidaan pitää mikropalveluarkkitehtuuria, jossa järjestelmän eri ominaisuudet toimivat itsenäisinä yksiköinä. Henkilöriippuvuuden kannalta mikropalveluarkkitehtuuria voidaan pitää monoliittisten projektien vastakohtana, sillä järjestelmän muuttuessa monimutkaisemmaksi, mikropalveluiden ylläpito, kehitys ja täten tuottavuus ei kärsi yhtälailla verrattuna monoliittisiin projekteihin kuten kuvasta 3.1 käy ilmi. Seuraavissa kappaleissa avataan legacyn ja monoliittisten projektien vaikutusta henkilöriippuvuuteen. Tämän lisäksi kappale tarkastelee käsitteenä mikropalveluarkkitehtuuria vastakohtana monoliittiselle projektille henkilöriippuvuuksien näkökulmasta.

3.1.1 Legacy henkilöriippuvuudessa

Legacyllä ei tietotekniikassa niinkään viitata projektin tai koodin varsinaiseen ikään vaan siihen miten projektin kehittäminen etenee. Yksi merkittävä ero legacyn ja ei-legacyn välillä on kattavat testitapaukset. Legacya kehittäessä kehittäjällä voi olla jatkuvasti tunne, että ei ole varma vaikuttaako seuraava muutos koodikannassa johonkin ominaisuuteen johon sen ei ole tarkoitus vaikuttaa. Muutoksia koodiin tehdessä ei olla varmoja siitä, että vaikuttaako muutos haluttuun kohtaan järjestelmää, joka johtaa manuaaliseen testailuun ja epävarmuuteen muutoksien toiminnasta. [22]. Legacy-koodin monimutkaisuus vaikuttaa siihen, että kehittäjillä kestää kauemmin päästä projektissa tasolle, jossa heillä on korkea suorituskyky, joka on suoraan verrannollinen kustannuksiin, sekä hankaloittaa tehtävien vaikeuden tai työmäärän ar-



Kuva 3.1: Monoliittisten projektien tuottavuus laskee monimutkaisuuden kasvaessa, kun taas mikropalveluarkkitehtuurin tapauksessa lasku on maltillisempaa [21]

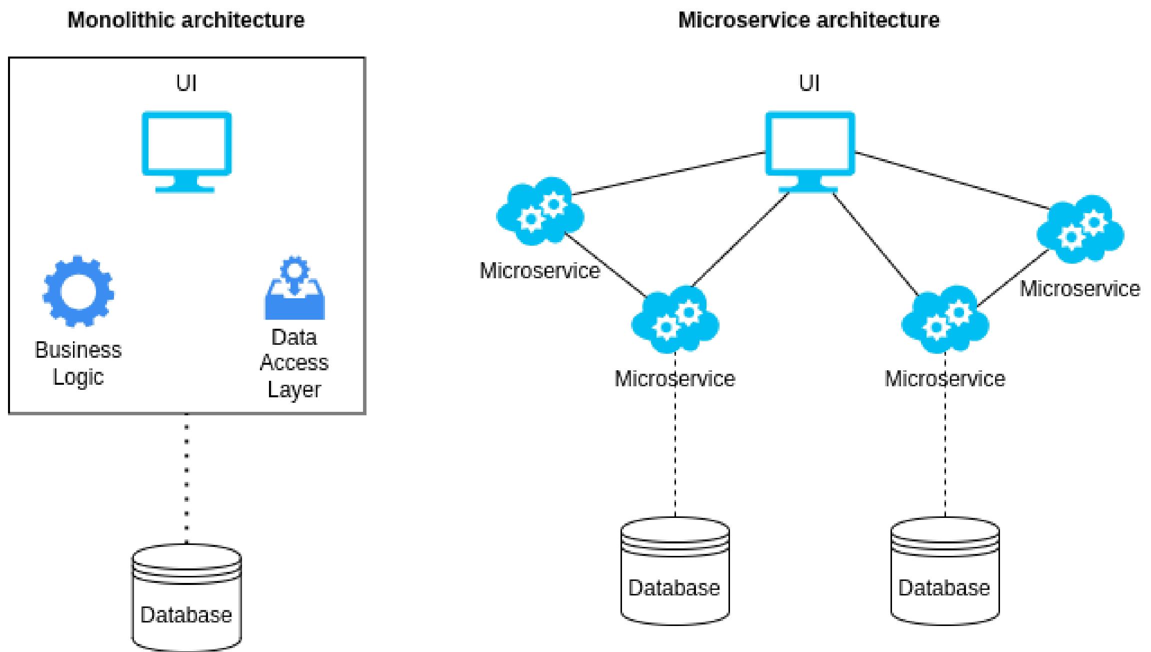
vioimista[23]. Legacyn tapauksessa henkilöriippuvuus on selkeästi nähtävissä siitä, että kauemmin projektissa ollut ja enemmän kontekstiymmärrystä omaava henkilö pystyy tehokkaasti työskentelemään koodikannan kanssa vaikka ulkopuolisen tai uuden kehittäjän näkökulmasta legacyn rajoitteet hankaloittavat kehittämistä merkittävästi.

3.1.2 Monoliittisuus henkilöriippuvuudessa

Monoliittisella arkkitehtuurilla tarkoitetaan projekteja, joissa koko järjestelmän toiminnallisuus on sidottuna yhteen koodikantaan. Tämä yksittäinen koodikanta sisältää oleelliset toiminnallisuudet kuten tietokantojen hallinnan, käyttöliittymän ja bisnelogiikan (Kuva 3.2). Monoliittisten projektien etuina on yleensä nopeampi kehitystahti, käyttöönotto ja testattavuus. Tämä koskee kuitenkin vain projekteja, joiden koodikanta on kohtuullisen pieni. Näiden ominaisuuksien vuoksi monoliittinen arkkitehtuuri on hyvä vaihtoehto projektin ollessa pieni ja silloin kun tiedetään, että projektia ei olla laajentamassa.

Monoliittisen arkkitehtuurin ongelmana on järjestelmän ominaisuuksien riippuvuus toisistaan, kun koodikanta ja projekti laajenee. Tällöin suurimmaksi ongelmaksi voi kehittäjän näkökulmasta muodostua jo olemassa olevien ominaisuuksien toiminnan säilyttäminen. Kun lisätään uusia ominaisuuksia, täytyy ottaa huomioon, että ne eivät vaikuta haluamattomilla tavoilla jo olemassa oleviin ominaisuuksin. Samoin vanhoja ominaisuuksia muokattaessa, vaaditaan kehittäjältä laajempaa kontekstiymmärrystä koko monoliittisesta järjestelmästä ja sen toiminnasta. Mikäli kehittäjällä ei ole tarpeeksi ymmärrystä järjestelmän toiminnasta tai kehityksessä ei ole otettu huomioon tarvittavia testitapauksia, voi pahimmillaan muutokset rikkoa bisneslogiikan kannalta oleellisia kokonaisuuksia. [24][25]

Puhuttaessa ohjelmistokehityksessä esiintyvistä, kahden tai useamman eri ominaisuuden riippuvuksista toisiin, voidaan käyttää termiä “coupling” (suom. pariutu-



Kuva 3.2: Monoliittisen- ja mikropalveluarkkitehtuurin kuvaus

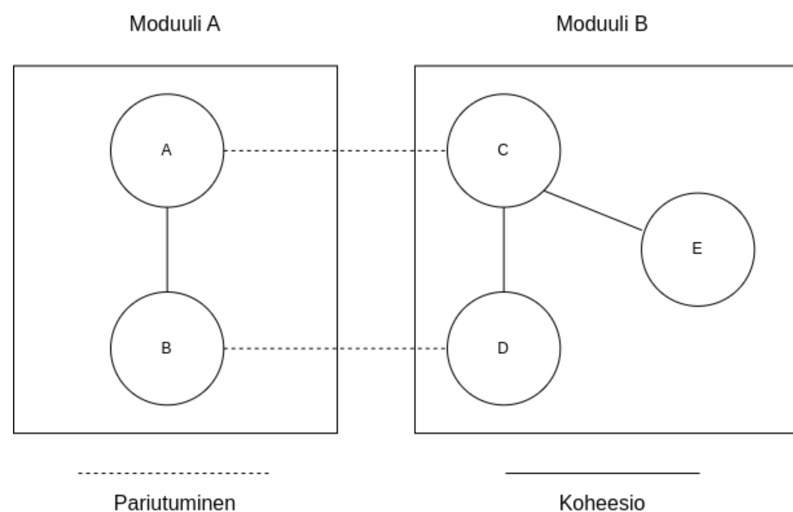
minen). Ohjelmistokehityksessä yksittäisten ominaisuuksien tai moduulien pariutumisessa pariutumisen vahvuus määrittää sen miten helppoa nähin ominaisuuksiin on tehdä muutoksia [26]. Viitaten kuvaan 3.3, mikäli kehittäjä tekee uuden muutoksen moduulin A ominaisuuteen B, on sillä suoria vaikutuksia moduuli B:n ominaisuuden D toimintaan, joka on riippuvainen moduulin A ominaisuudesta B. Pariutuminen tuokin merkittäviä haasteita kehittämiseen, kun tarkastellaan uusien henkilöiden tuomista projektiin. Tällöin ei riitä, että uusi kehittäjä sisäistää yhden moduulin toiminnan vaan moduulin A ominaisuuden B toiminnan muuttaminen vaatiikin kokonaan eri moduulin toimintaan perehtymistä. Moduulin ja koko projektin ymmärtäminen muuttuu tällöin huomattavasti vaikeammaksi, kuten Stevens artikkelissaan toteaa:

> “Coupling is the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by

itself if it is highly interrelated with other modules. Complexity can be reduced by designing systems with the weakest possible coupling between modules.” [26]

Mikäli ominaisuuksien tai moduulien pariutuminen on vahvaa, vaatii ohjelmiston kehittäminen, sekä ylläpito huomattavasti enemmän kontekstiosaamista verrattuna tilanteeseen, jossa pariutuminen on taas heikkoa. Järjestelmän ominaisuuksien ja moduulien pariutumista tarkastellessa, monesti tarkasteluun liitetään myös termi “cohesion” (suom. koheesio/yhteenkuuluvuus). Koheesiolla tarkoitetaan tässä kontekstissa sitä kuinka hyvin saman moduulin sisältämien metodit tai ominaisuudet sopivat yhteen. Tavoitetila koheesion kannalta olisi mahdollisimman korkea koheesio, jolloin yksittäinen moduuli vastaa vain yhdestä tehtävästä. [27]

“Cohesion refers to the “relatedness” of module components. A highly cohesive component is one with one basic function. It should be difficult to split a cohesive component.” [27]



Kuva 3.3: Pariutumisen ja koheesion ero

Monoliittiset projektit pystytään suunnittelemaan ja toteuttamaan niin, että

edellämainitut pariutuminen ja koheesio ovat hyvin huomiotuina. Kehittäjällä voi silti olla ongelmia varmistua siitä, että koodiin tehdyt muutokset eivät vaikuta muihin järjestelmän toimintoihin. Ajan kuluessa järjestelmät kasvavat yhä isommiksi ja monimutkaisemmiksi, jolloin suuri toiminnallisuuksien määrä laskee järjestelmän koheesiota ja kasvattaa pariutumista. [28] [24]

3.1.3 Mikropalveluarkkitehtuuri henkilöriippuvuudessa

Kun verrataan mikropalveluarkkitehtuuria monoliittisiin projekteihin henkilöriippuvuuden näkökulmasta, on sillä joitain selkeitä etuja. Toisin kuin monoliittisissa projekteissa, mikropalveluarkkitehtuurin tapauksessa taas korkea koheesio, sekä heikko pariutuminen ovat ominaisuuksia, jotka ovat huomioituina jo itse arkkitehtuurissa. Etenkin heikko pariutuminen on merkittävä mikropalveluarkkitehtuurin etu verrattuna monoliittiseen arkkitehtuuriin[29]. Henkilöriippuvuutta tarkastellessa tämä tarkoittaa sitä, että kehittäjä, jolla ei ole tarvittavaa kontekstiymmärrystä kehittääkseen tai muokatakseen vastaavaa ominaisuutta monoliittisessa projektissa voi hyvinkin olla kykeneväinen tekemään tarvittavia muutoksia mikropalveluun. Korkean koheesion ja heikon pariutumisen lisäksi mikropalveluiden etuna on niiden pieni koodikoko. Pieni koodikoko yksinkertaistaa kontekstin hallintaa ja vähentää täten riippuvuutta yksittäisestä henkilöstä. Uuden kehittäjän on huomattavasti helpompi sisäistää yksittäisen mikropalvelun toiminta silloin kun järjestelmän sisältämä koodimäärä ja täten logiikan määrä on pienempi. [30] Vaikka mikropalvelutkin nimestään poiketen voivat olla koodimäärältään suuria, tulee huomioida se, että kyseinen ominaisuus monoliittisessa arkkitehtuurissa olisi todennäköisesti vastaavan kokoinen kaiken muun logiikan seassa.

Tarkastellessa mikropalveluiden koheesiota, on koko arkkitehtuurin ajatuksen takana se, että yksittäinen mikropalvelu vastaa vain yhden tehtävän suorittamisesta. Pienimmillään tämä voi olla jopa yksittäinen funktio(FAAS, function as a ser-

vice[31]) tai jokin muu yhtenäinen kokonaisuus. Mikropalveluiden tarkoituksena on vastata tietyn asian suorittamisesta ja välittää siitä saatu informaatio eteenpäin. Tämän takia uuden kehittäjän on huomattavasti helpompi sisäistää mikä tämä asia on ja arvioida siihen tehtävän muutoksen vaikutus muuhun mikropalvelun toimintaan.

Mikropalveluiden matala pariutuminen edesauttaa siihen, että lähtökohtaisesti muutokset vaikuttavat vain kyseisen mikropalvelun toimintaan. Tällöin kehittäjän ei tarvitse huolehtia siitä vaikuttaako mikropalvelun koodikantaan tehdyt muutokset johonkin ei-haluttuun ominaisuuteen. Tämä myös mahdollistaa useamman kehittäjän samanaikaisen työskentelyn palveluiden parissa[32]. Kuten kappaleen 3.1.2 esimerkissä tulee ilmi, on monoliittisissa projekteissa suurempi riski, että ilman kontekstiymmärrystä saatetaan muokata jotain ominaisuutta tavalla, joka vaikuttaa myös ominaisuuksiin joita ei ollut tarkoitus muokata. Mikropalveluiden lähtökohtaisesti matalampi pariutuminen helpottaa uuden kehittäjän työtä siinä, että muutokset todennäköisimmin osuvat oikeaan.

Mikropalveluiden huonona puolena henkilöriippuvuuden näkökulmasta voidaan pitää sitä, että järjestelmän sisältämien osien ymmärtäminen saattaa olla haastavampaa. Siinä missä monoliittinen projekti pitää kaiken sisällään, voi mikropalveluarkkitehtuurissa järjestelmä olla jaettuna todella moneen erilliseen osaan, jolloin kokonaisuuden hahmottaminen vaikeutuu. Lisäksi edellämainitusta syystä kehitysympäristöjen pystyttäminen saattaa aiheuttaa enemmän haasteita.

3.2 Henkilöriippuvuuden vaikutukset sovelluskehitysprojekteihin

Henkilöriippuvuus vaikuttaa sovelluskehitysprojekteihin monella tapaa ja sen hallitsemiseksi tulisikin tehdä toimenpiteitä. Kuten edellä mainittua, pahimmillaan henkilöriippuvuus johtaa tilanteeseen, jossa kehitysryhmästä jää liikaa oleellisia kehittäjiä

pois sairastumisen, työpaikan vaihtamisen tai muun asian seurauksena.

Tämän lisäksi henkilöriippuvuudella on muita merkittäviä vaikutuksia sovelluskehitysprojekteihin. Sovelluskehityksen aikana kehittäjät oppivat tuntemaan järjestelmän ja sen eri toiminnallisuudet tavalla, jonka dokumentoiminen on hyvin raskasta. Lisäksi dokumentoinnin selaaminen laajan kontekstiymmärryksen saavuttamiseksi ei ole käytännössä realistista. Parhaita tapoja hallita henkilöriippuvuutta on dokumentoinnin lisäksi varmistaa projektin jatkuvuus joko kouluttamalla tai mieluiten osallistuttamalla uusia henkilöitä projektiin. Tällöin projektin resilienssi bus factor -mittarilla paranee.

Yksi yleisimmistä kontekstiosaamisen puutteeseen johtavista tilanteista on niinsanottu “handover”, jossa projekti luovutetaan joko yrityksen sisäisesti toiselle kehitystiimille tai henkilölle tai vielä useammin yritykseltä toiselle. Huolimatta siitä, että järjestelmää olisi huolellisesti dokumentoitu ja handoveriin varattaisiin hyvin aikaa toimittajien välille, jää edellämainittu ajan tuoma kontekstiymmärrys handoverin mukana edelliselle toimittajalle. Varsinaisen handoverin lisäksi kontekstiosaamisen puute realisoituu, mikäli projektiryhmään halutaan tuoda uusia henkilöitä mahdollisesti nopeuttamaan kehitystä, muuttuneen markkinatilanteen vuoksi tai tuomaan uusia näkökulmia. Tällaisessa tilanteessa vaaditaan huolellinen perehdytys järjestelmän toimintoihin.

Vaikka yleisimmin handoveria käsittelevät artikkelit keskittyvät siihen, että ohjelmistokehitysprojekti siirtyy eri kehittäjälle tai kehitystiimille, on huomioon otettava asiakkaan puoli tässä prosessissa. Etenkin konsulttiprojekteissa tai muissa tapauksissa, joissa kehityksestä vastaava taho ei omista tuotetta, ole sen ensisijainen käyttäjä tai välittäjä, muodostuu tämän kyseisen tahon henkilöriippuvuudet yhdeksi merkittäväksi uhkatekijäksi järjestelmän kannalta. Asiakkaan puolelta ei välttämättä osata varautua mahdollisiin järjestelmän kehityksen ja ylläpidon kannalta oleellisiin seikkoihin, joihin törmätään, kun esimerkiksi henkilö, joka on vastannut järjestelmän

kehityssuunnasta ja omaa laajan ymmärryksen järjestelmän toiminnasta, sekä sen rajoitteista jää pois projektista.

Projekteissa, joissa kehitetään erilliselle taholle tuotetta on mahdollista, että kehitystiimin ja asiakkaan välillä toimii henkilö, jolla kehittäjien ohella on eniten kontekstitietoa ja ymmärrystä järjestelmän toiminnasta. Agile-menetelmissä, kuten Scrumissa ja Kanbanissa, tällaiseen henkilöön viitataan termillä “product owner”[33]. Product ownereita on lähtökohtaisesti projekteissa vain yksi, joten on selvää, että aiemmin mainittu henkilöriippuvuuden bus factor -mittarilla tarkasteltuna tällaisen henkilön poisjääminen luo luonnollisesti uhkan projektin etenemiselle. Kun kyseessä on vanha projekti, jonka kehitys on jatkunut pitkään, kasvaa haasteet kun projektiin pitäisi saada uusia kehittäjiä.

Kappaleissa 3.1.2 ja 3.1.3 käsitellään monoliittisten projektien ja mikropalveluarkkitehtuurin vaikutuksia henkilöriippuvuuteen. Näiden kahden arkkitehtuurin eroista pystytään päättelemään, että mikropalveluarkkitehtuurilla on omat etunsa mitä tulee henkilöriippuvuden hallintaan. Aiemmin tässä kappaleessa mainittu projektin luovutus yrityksen sisäisesti tai yritykseltä toiselle pystytään tarvittaessa tekemään mikropalveluarkkitehtuurin tapauksessa usein vaivattomammin. Ei ole myöskään poissuljettua, että useampi yritys tai projektiryhmä vastaa saman tuotteen kehityksestä mikropalveluarkkitehtuurissa niin, että heidän vastuullaan ovat tietyt mikropalvelut. Tällaisessa tapauksessa voidaan jopa toimia niin, että uusi yritys tai projektiryhmä ottaa portaittain järjestelmän eri mikropalveluita ylläpitoon ja kehitykseensä, jolloin henkilöriippuvuuden näkökulmasta saadaan kontekstiosaaminen siirtymään porrastetusti uudelle taholle. Vastaavassa tilanteessa taas monoliittisen projektin luovuttamisessa on hyötynä projektin keskittyneisyys, jolla tarkoitetaan sitä, että yksi koodikanta sisältää kaiken mitä projektin kehitys vaatii. Tällöin esimerkiksi kehitysympäristön pystyttäminen on todennäköisesti helpompaa kuin hajautetummassa projektirakenteessa.

3.3 Henkilöriippuvuuden suhde tekniseen velkaan

Henkilöriippuvuuden suhdetta tekniseen velkaan voidaan tarkastella kahdesta näkökulmasta: nykyhetken ja tulevaisuuden haasteista.

3.3.1 Henkilöriippuvuuden haasteet nykyhetkessä

Tarkastellessa henkilöriippuvuuden suhdetta tekniseen velkaan nykyhetkessä, merkittävimpiä riskeinä ovat työntekijän äkillinen sairastuminen, lomailu sekä vanhemmuusvapaa.

Sairastapauksien vaikutus projektien etenemiseen ja ylläpitoon henkilöriippuvuuksien näkökulmasta on melko pieni. Suurin osa sairaslomista on kestoltaan enintään vain muutaman päivän. [34] Näin lyhyen poissaolon vaikutukset eivät yleensä vaadi korvaavan henkilön ajamista sisään projektiin, poislukien kriittiset tietoturvahaavoittuvuudet, sekä muu tukitoiminta palvelutasosopimuksesta(eng. SLA) riippuen.

Suuremman jatkuvan uhkan henkilöriippuvuuksien näkökulmasta aiheuttaa lomailu. Riippuen maasta, jossa työntekijä työskentelee vaihtelee lomien pituudet. Suomessa lain mukaan työntekijä on oikeutettu kerryttämään lomapäiviä kaksi ja puoli arkipäivää kultakin täydeltä lomanmääräytymiskuukaudelta, ellei toisin säädetä. [35] Tämä tarkoittaa käytännössä vakituisessa työsuhteessa olevalle henkilölle kuukauden mittaista kesälomaa, sekä viikon mittaista talvilomaa. Etenkin em. kesäloma vaatii kestonsa puolesta yleensä sitä, että projektitiimin kesken poissaolevan henkilön vastuu on jaettu jo ennen lomien alkamista.

Vanhemmuusvapaiden kesto on usein huomattavasti pidempi, muutamasta kuukaudesta jopa pariin vuoteen. [36] Mikäli tiedossa on, että henkilö työskentelee henkilöriippuvuuden kannalta haasteellisessa projektissa, on varmistuttava siitä, että hänen vastuunsa saadaan siirrettyä toiselle henkilölle tai jyvitettyä muiden projektin työntekijöiden kesken. Tällaisessa pidemmän ajanjakson poissaolossa tulisi myös

tarkastella mahdollisuutta, jossa projektiin siirrettäisiin uusi työntekijä.

3.3.2 Henkilöriippuvuuden haasteet tulevaisuudessa

Merkittävimpänä tekijänä tutkittaessa henkilöriippuvuuden ja teknisen velan välistä suhdetta on henkilöstön vaihtuvuus. Henkilöstön vaihtuvuudella tarkoitetaan organisaatiossa aloittavia ja lopettavia työntekijöitä. IT-maailmassa ja sovelluskehityksessä, yritysten suurin arvo koostuu yleensä aineettomasta pääomasta. Aineettomassa pääomassa yrityksen maine, osaaminen ja asiakassuhteet ovat keskeisiä yrityksen menestymisen kannalta [37]. Henkilöriippuvuudessa voidaankin tarkastella yksittäistä kokenutta kehittäjää yrityksen arvon kannalta merkittävänä tekijänä. Tällöin voidaan olettaa, että yritys haluaa liiketoiminnan näkökulmasta minimoida riskit arvon menettämiselle. Mikäli henkilöriippuvuutta esiintyy paljon, kasvaa myös riskit siihen, että henkilöstön vaihtuvuus laskee näin yrityksen aineetonta pääomaa.

Kuten luvun 2.2 lopussa tulee ilmi, tekninen velka vaikuttaa myös yleisellä tasolla negatiivisesti työntekijöiden moraaliin, joka on omiaan lisäämään henkilöstön vaihtuvuutta.

Kuten aiemmassa kappaleessa tuli ilmi, teknistä velkaa syntyy vääjäämättä projektin edetessä. Vaikka optimitilanteessa teknisen velan ottaminen olisi dokumentoituna, voi juuri tällaisen tiedon omaaminen jäädä yksittäisen kehittäjän taakse. Ajan kuluessa ja henkilöstön vaihtuessa voi teknistä velkaa kerryttävien ominaisuuksien ja toteutusten olemassaolon tiedostaminen kadota täysin. Tällöin saatetaan alkaa rakentamaan uutta toiminnallisuutta sellaisen ominaisuuden päälle, joka jo ennestään kerryttää teknistä velkaa, joko olemalla esimerkiksi huonosti toteutettu tai käyttämällä vanhentunutta teknologiaa.

Tekninen velka monesti myös saattaa hankaloittaa koodin luettavuutta ja ymmärrettävyyttä silloin kun tekninen velka syntyy koodikantaan vaikuttavista tekijöistä. Koodin muuttuessa vaikeammin ymmärrettäväksi henkilöriippuvuuden mer-

kitys korostuu, sillä uusien henkilöiden perehdyttäminen on aikaa vievää ja vaatii paljon resursseja verrattuna koodikantaan, jossa ei ole teknisen velan aiheuttamaa taakkaa. Vuoden 2007 tutkimuksessa kävi ilmi, että työntekijöiden vaihtuvuus on yhtä todennäköistä pitkään organisaatiossa työskenneille, kuin uusille työntekijöille. Erona kuitenkin näiden työntekijöiden välillä on, että pitkään organisaatiossa työskennelleille henkilöille on kertynyt merkittävä määrä ymmärrystä projekteista, jonka takaisinkerryttäminen on hankalaa[38]. Useissa tutkimuksissa on tullut ilmi, että tekijöitä siihen miksi kokeneemmat kehittäjät vaihtavat työpaikkaa on esimerkiksi työn mielenkiintoisuus, merkityksellisyys ja kuormittavuus [39] [40] [41]. Tekninen velka ja sen synnyttämät ongelmat kehityksessä vaikuttavat negatiivisesti kaikkiin edellämainittuihin, sekä lukuisiin muihin tekijöihin.

Kun halutaan minimoida teknisen velan haasteet henkilöriippuvuuden näkökulmasta tulevaisuutta ajatellen, tulisi projektin rakennetta suunnitellessa ottaa huomioon edellämainitut asiat. Mikäli jo projektia suunnitellessa tiedetään, että projektin elinkaari on lyhyt ja voidaan olettaa, että se ei tule altistumaan suurelle henkilövaihtuvuudelle, voi monoliittinen projekti olla oikea valinta. Kuten kuvassa 3.1 tulee ilmi, voittaa monoliittinen projekti yksinkertaisimmillaan mikropalveluarkkitehtuurin kehityksen tuottavuudessa. Sama tuottavuuden ero heijastuu myös henkilöriippuvuuksiin ja täten teknisen velan hallintaan. Projektin monimutkaistuessa, mikropalveluiden tarjoama heikko pariutuminen tarjoaa kehittäjille paremmat mahdollisuudet hallita teknistä velkaa pienemmällä kontekstiosaamisella.

3.3.3 Yhteeveto henkilöriippuvuuden suhteesta tekniseen velkaan

Tarkastellessa kirjallisuuslähteitä kappaleessa 2 teknisen velan ja kappaleessa henkilöriippuvuuden 3 osalta nousee esiin muutama seikka, jotka yhdistävät näitä kahta termiä.

Kuten kappaleessa 2.1 käy ilmi, syntyy teknistä velkaa kun kehityksen aikana tehdään ratkaisuja, jotka lyhyellä aikavälillä säästävät rahaa, eli toisin sanoen ovat nopeita toteuttaa. Toteutuksen nopeuteen positiivisessa mielessä voi vaikuttaa usea seikka. Esimerkiksi toteutuksen dokumentoinnin, testaamisen tai koodin tarkistusmenettelyn(engl. code review) laiminlyönti ovat menetelmiä, joilla saadaan nopeutettua kehitysprosessia, mutta jätetään teknistä velkaa seuraavalle kehittäjälle. Sen vuoksi teknistä velkaa kerryttäessä päädytään monesti tilanteeseen, jossa kyseinen tekninen velka tai tieto sen synnystä on vain yhden henkilön tiedossa voidaan päätellä, että tekninen velka johtaa välillisesti myös henkilöriippuvuuteen. Henkilöriippuvuuden kasvattaminen teknisen velan johdosta taas saattaa aiheuttaa kierteen, jossa koodia tai kontekstia ymmärtämätön kehittäjä tuottaa koodia, joka entisestään kerryttää teknistä velkaa ja täten lisää henkilöriippuvuutta.

Kuten kappaleessa 3.2 todetaan, vaikuttaa henkilöriippuvuus myös muihinkin kuin järjestelmän kehittäjiin. Mikäli järjestelmän kehitysvaiheessa kehittäjä ei ole esimerkiksi huolehtinut dokumentaation päivittämisestä voi järjestelmän ympärillä toimiville henkilöille olla hyvinkin epäselvää tiettyjen kokonaisuuksien toiminta. Esimerkiksi dokumentaation puutteen aiheuttama henkilöriippuvuus on suoraan yhdistettävissä Kruchten ym.[3] artikkelissa mainittuun henkilö- ja dokumentaatiovelkaan. Mikäli järjestelmän muutoksia ei dokumentoida, voi pahimmillaan olla tilanne, jossa esimerkiksi järjestelmän tukihenkilökunnalla ei ole ajantasaista tietoa jonkin tietyn ominaisuuden toiminnasta ja täten eivät virhetilanteessa pysty ohjeistamaan loppukäyttäjää. Lievemmissä tapauksissa tämä saattaa aiheuttaa ongelmia, kun järjestelmään halutaan uusia ominaisuuksia ja näitä aletaan suunnittelemaan dokumentaation pohjalta, joka ei ole ajantasalla. Mikäli samat henkilöt ovat olleet suunnittelemassa järjestelmän aiempia ominaisuuksia saattavat he muistaa, että dokumentaatio ei vastaa tämänhetkistä toteutusta. Kuitenkin tapauksessa, jossa uutta ominaisuutta suunnittelemassa olevat henkilöt eivät ole tietoisia dokumentaation ja

järjestemän tämänhetkisen tilan eroista, voi uuden ominaisuuden toteutus hankaloitua merkittävästi, mikäli jo suunnitteluvaiheessa siihen on määritelty asioita, jotka eivät vastaa järjestelmän tämänhetkistä tilaa.

4 End-to-end testaus

4.1 Mitä end-to-end testaus tarkoittaa

Sovelluskehityksessä testaamisella tarkoitetaan prosessia, jonka avulla pyritään varmistamaan ohjelmistojärjestelmän laatu. [42]

> "The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances." [42]

Sovellustestauksen lajikirjo on hyvin laaja. Karkeasti jaoteltuna testejä voi olla automaattisia tai manuaalisesti ajettavia. Testien ajotavan lisäksi testit voidaan jakaa useaan eri kategoriaan. Tässä tutkimuksessa testejä käsitellään neljän eri kategorian kautta. Nämä kategoriat ovat yksinkertaisimmasta monimutkaisimpaan: yksikkötestit, integraatiotestit, järjestelmätestit ja end-to-end testit. (Taulukko 4.1) (Kuva 4.1)

Luo [43] sisällyttää tutkimuksessaan end-to-end (jatkossa e2e) -testaamisen terminä järjestelmätesteihin. Alan artikkeleissa on kuitenkin nykyään alettu erottamaan järjestelmätestit ja e2e-testit omiksi kokonaisuuksikseen. [44] [45] Microsoftin (2022) mukaan suurimpina eroina järjestelmä- ja e2e-testien välillä on se, että järjestelmätestit keskittyvät laadunvalvontaan, kun taas e2e-testit testaavat koko sovelluksen kulkua (application flow) käyttämällä tuotantoa vastaavaa ympäristöä. Lisäksi Microsoft on määritellyt e2e-testaamisen tapahtuvan manuaalisesti, kun taas

Taulukko 4.1: Testaamisen tyypit [43]

Tyyppi	**Kuvaus**
Yksikkötestit (engl. unit testing)	Alimman tason testejä. Testaa yksittäisen järjestelmän osan (“unit”, “module”, “component”) toimintaa riippumatta muista järjestelmän ominaisuuksista.
Integraatiotestit (engl. integration testing)	Testejä joissa yhdistyy useamman eri järjestelmän osan toiminnallisuudet. Varmistetaan, että tieto liikkuu yksittäisten järjestelmien välillä oikein.
Järjestelmätestit (engl. system testing)	Järjestelmää testataan kokonaisuutena. Varmistetaan, että kaikki ominaisuudet toimii alusta loppuun.
Hyväksyntätestit (engl. acceptance testing)	Järjestelmä siirtyy kehittäjiltä testattavaksi asiakkaalle tai erilliselle testausryhmälle toimivaksi todettuna kokonaisuutena. Keskittyy enemmän järjestelmän toimivaksi toteamiseen eikä niinkään virheiden etsimiseen.

järjestemätestit voidaan toteuttaa myös automatisoidusti. (Kuva 4.1)

E2e-testit voidaan jakaa kahteen eri kategoriaan, vaaka- ja pystysuuntaiseen (engl. horizontal and vertical end to end testing). Vaakasuuntaisella e2e-testauksella tarkoitetaan sellaista testausta, joka kattaa koko sovelluksen. Käytännössä tämä tarkoittaa loppukäyttäjän simuloimista varsinaista tuotantoympäristöä vastaavassa ympäristössä. Vaakasuuntaisilla testeillä pystytään testaamaan tietyn käyttäjän prosessin toimintaa, kuten rekisteröitymisprosessia. Käytännön esimerkkinä horisontaalisesta testistä voidaan rekisteröitymisprosessi jakaa käyttäjän saapumiseen sivustolle, “rekisteröidy”-napin painallus ja käyttäjätietojen täyttö. Pystysuuntaisella e2e-testauksella vastaavassa esimerkissä testataan sitä, tallentuvatko käyttäjätiedot tietokantaan rekisteröinnin yhteydessä ja vastaako rajapintakutsu oikealla tavalla kun käyttäjä yrittää kirjautua sisään uusilla tunnuksilla.[46][47]

Tässä tutkimuksessa e2e-testeillä viitataan testeihin jotka simuloivat oikean käyttäjän toimia käyttöliittymässä ja niillä testataan tietyn järjestelmän kannalta kriittisen toiminnallisuuden toimivuutta. Tutkimus käsittelee lähinnä testejä tapauksissa, joissa on erikseen käyttöliittymäpuoli (engl. frontend), jonka loppukäyttäjä näkee

The table below illustrates the most critical characteristics and differences among Unit, Integration, System, and End-to-End Testing, and when to apply each methodology in a project.

	Unit Test	**Integration Test**	**System Testing**	**E2E Test**
Scope	Modules, APIs	Modules, interfaces	Application, system	All sub-systems, network dependencies, services and databases
Size	Tiny	Small to medium	Large	X-Large
Environment	Development	Integration test	QA test	Production like
Data	Mock data	Test data	Test data	Copy of real production data
System Under Test	Isolated unit test	Interfaces and flow data between the modules	Particular system as a whole	Application flow from start to end
Scenarios	Developer perspectives	Developers and IT Pro tester perspectives	Developer and QA tester perspectives	End-user perspectives
When	After each build	After Unit testing	Before E2E testing and after Unit and Integration testing	After System testing
Automated or Manual	Automated	Manual or automated	Manual or automated	Manual

Kuva 4.1: Miten eri testit eroavat toisistaan [45]

ja jonka kanssa käyttäjä vuorovaikuttaa, sekä palvelinpuoli (engl. backend), joka sisältää bisneslogiikan, määrittelee tietokannan rakenteen ja hallitsee sen toimintaa. Testejä ajetaan kehittäjän toimesta, joko lokaalia ympäristöä vasten tai tuotantoa vastaavaa pilottiympäristöä vasten. Testejä voidaan ajattaa, joko manuaalisesti tai automaattisesti. Esimerkkejä kriittisistä toiminnallisuuksista järjestelmän kannalta, joita edellämainituilla testeillä testataan, on esimerkiksi käyttäjän kulku sivuston etusivun kautta rekisteröitymiseen, siitä kirjautumiseen ja lopulta kirjautuneen käyttäjän näkymään.

4.2 Kuinka end-to-end -testeillä voidaan hallita teknistä velkaa

E2e-testien rooli teknisen velan hallinnassa on ensisijaisesti auttaa kehittäjää maksamaan teknistä velkaa pois. Kuten aiemmin tutkimuksessa on käynyt ilmi, teknistä velkaa syntyy ohjelmistokehityksen yhteydessä ja siihen täytyy aikanaan puuttua. Tämä tarkoittaa käytännössä jo olemassa olevien ominaisuuksien läpikäyntiä ja mahdollisesti suuriakin refaktorointeja. Ohjelmistokehityksessä refaktoroinnilla tarkoitetaan olemassa olevan koodin muokkaamista siten, että muutokset eivät vaikuta ohjelmiston toiminnallisuuteen käyttäjän näkökulmasta. Refaktoroinnin tarkoituksena on parantaa koodin laatua. Vaikka lähtökohtaisesti tavoitteena on ensin suunnitella ja sitten toteuttaa hyvää koodia, kuitenkin ajan ja uusien ominaisuuksien myötä koodin eheys vähitellen heikkenee. Tällöin voidaan harkita refaktoroinnin tarpeellisuutta.[48] Teknisen velan takaisinmaksu voidaankin tulkita monesti refaktoroinniksi.

Kun tehdään muutoksia olemassa oleviin toiminnallisuuksiin tulee varmistua siitä, että järjestelmä toimii edelleen. Järjestelmän toiminnan takaamisessa voidaan käyttää testejä. Testit määrittävät miten järjestelmän kuuluu toimia. Tällaista tes-

taamista kutsutaan regressiotestaamiseksi [49]. Mikäli testi hajoaa jotain ominaisuutta muokattaessa, saattaa se tarkoittaa sitä, että kyseinen ominaisuus ei toimi enää samalla tavalla mitä sen on alunperin suunniteltu toimivan. Refaktoroidessa koodia saattaa kuitenkin tulla tilanteita, joissa osuus järjestelmästä, jota ei refaktoroida on liitoksissa ominaisuuteen tai kokonaisuuteen jota refaktorointi koskee. Tällöin voidaan päätyä tilanteeseen, jossa muokataan ominaisuutta A ja tarkkaillaan saman ominaisuuden A toimintaa esimerkiksi manuaalisella savutestauksella (engl. smoke testing). Savutestauksella tarkoitetaan nopeaa tapaa testata, että ominaisuus toimii edelleen esimerkiksi muokattaessa käyttöliittymässä napin toimintaa, voidaan käydä käyttöliittymässä painamassa kyseistä nappia ja todentaa sen toiminta [50]. Tarkastellessa monimutkaisempia kokonaisuuksia voi kuitenkin olla, että vaikka järjestelmässä on kattavat yksikkötestit ja kehittäjä on kehitysvaihessa testannut käytännössä, että refaktoroitu ominaisuus toimii yhä täysin samalla tavalla kuin aiemmin, voi ominaisuuden A refaktorointi kuitenkin olla vaikuttanut ominaisuuteen B siten, että siinä on tapahtunut muutoksia haluttuun toimintatapaan verrattuna. Etenkin monoliittiset projektit ja projektit joiden koodikanta pitää sisällään huonolaatuista suunnittelua ovat erityisen alttiita tällaiselle. Tällaisessa tilanteessa voidaan puhua niinsanotuista koodin hajuista (engl. code smell), joilla tarkoitetaan huonon suunnittelun ja toteutuksen vaikutuksia tuotetun koodin laatuun [51].

Esimerkkinä refaktoroinnissa on tilanne, jossa koodissa on määritelty tietokantakenttä siten, että käyttäjän nimi tallentuu snake casella kirjoitettuun "user_name" kenttään. Järjestelmän tyyliohje (engl. style guide) kuitenkin määrittelee, että tietokantakenttien nimien tulisi olla määriteltynä camel casella tyyliin "userName" [52][53]. Mikäli kehittäjä muuttaa tämän kentän nimeä on varmistettava, että muutos reflektoituu kaikkialle järjestelmään, jotka kyseistä kenttää käyttävät. Todennäköistä on, että edellä mainitun esimerkin tapauksessa, mikäli järjestelmään on jäänyt vanhoja viittauksia "user_name" kenttään, jäävät ne luultavasti kiinni eri-

näisiin koodin tarkastustyökaluihin, kuten linttereihin, yksikkötesteihin tai muihin CI/CD-putkessa ajettaviin tarkastuksiin [54]. Tällaisessa tilanteessa e2e-testien roolilla on suuri merkitys kun pyritään säilyttämään järjestelmän toiminnat entisellään. Vaikka edellämainitut testit varmistavat backendin toiminnan muutosten jälkeen, jää frontendin ja backendin välinen keskusteluyhteys testaamatta. Kun edellämainitussa tapauksessa frontend yrittää lähettää backendiin käyttäjän nimen muodossa "user_name" tapahtuu virhe, joka pystytään havaitsemaan oikein määritetyllä e2e-testillä.

Tarkastellessa e2e-testauksen roolia teknisen velan hallinnassa, painottuu käyttöliittymän toiminnan testaus suuremmassa roolissa mitä backendin. Kuten aiemmin mainittua, keskittyy e2e-testaamaan sitä miten frontend ja backend toimivat yhdessä. Kuitenkin backendiin toteutetut kattavat yksikkötestit mahdollistavat koodia refaktoroidessa sen, että pystytään varmistumaan bisnesslogiikan pysyvän ennallaan alustamalla tietynlainen jäljitelmä tietokannasta, minkä dataa vasten testit ajetaan. Frontendille pystytään tekemään näitä yksikkötestejä siinä missä backendillekin, mutta koska frontendin toiminta on lähes poikkeuksetta, staattisia nettisivuja tai järjestelmiä lukuunottamatta riippuvaista backendin toiminnasta, on kattavien testitapausten luominen haastavaa. Lähes kaikki järjestelmät hakevat dynaamisesti tietoa ja keskustelevat backendin kanssa siitä mitä käyttöliittymän tulisi näyttää. Tällöin on olennaista, että järjestelmää testattaessa käyttöliittymä pystyy hakemaan tietoa siitä backendin versiosta, mikä on sillä hetkellä järjestelmässä käytössä. Aiempaan esimerkkiin viitaten, mikäli käyttöliitymän yksikkötestillä testataan sitä, että käyttäjän profiilinäkymän oikeassa yläreunassa on elementti johon käyttäjän nimen kuuluu tulla näkyviin, voidaan e2e-testillä testata, että näytetäänkö siinä käyttäjän nimi jolla on kirjauduttu järjestelmään sisään. Aiemmassa esimerkissä käyttöliittymän yksikkötesti tarkastaa vain onko nimelle varattu elementti olemassa, mutta e2e-testi puolestaan ilmoittaisi virheestä, kun aiemmin haetun käyttäjän tiedoista

ei enää palaudukaan tietokantakenttää “user_name”.

4.3 Kuinka end-to-end -testeillä voidaan hallita henkilöriippuvuuksia

Henkilöriippuvuuksien hallinnan näkökulmasta e2e-testit tarjoavat muutaman eri hyödyn. Kuten aiemmin mainittua, kulkee tekninen velka ja sen ymmärrys käsi kädessä henkilöriippuvuuksien kanssa. Mikäli e2e-testit tarjoavat tavan varmistua muutosten toimivuudesta, jopa projektin ulkopuolisen tai kontekstista vähemmän ymmärtävän henkilön toimesta, voidaan todeta niiden hallitsevan omalla tavallaan myös henkilöriippuvuuksia. Kuten kappaleessa 3.2 todetaan, on epärealistista olettaa kehittäjien hankkivan laajaa kontekstiosaamista projektista pelkästään dokumentaatiota lukemalla. Tällöin on tärkeää, että järjestelmän testit tarjoavat tukea siinä, toimivatko muutokset halutulla tavalla koko järjestelmän laajuudella. Siinä missä yksikkötestit mahdollistavat backendiin tehtyjen muutosten toiminnan varmistamisen, voidaan e2e-testeillä varmistaa koko järjestelmän toimintaa. Edellinen kappale 4.1 käsittelee sitä miten e2e-testit kasvattavat varmuutta siihen, että järjestelmä toimii halutulla tavalla myös refaktoroinnin jälkeen. Tarkastellessa e2e-testien suhdetta henkilöriippuvuuteen pystytään toteamaan, että sama logiikka pätee myös siinä, että projektista vähemmän ymmärtävä pystyy tekemään laajempia refaktorointeja tai muutoksia järjestelmään.

E2e-testien etuna henkilöriippuvuuksien hallinnassa verrattuna yksikkötesteihin onkin niiden laajuus. Yksikkötestit nimensä mukaan keskittyvät yhden tietyn asian testaamiseen. E2e-testi saattaa kytkeä sisäänsä suuren sarjan tapahtumia. Esimerkkinä rekisteröitymistä testaava yksikkötesti backendissä saattaa testata, jos rajapinnan tietty päätepiste vastaanottaa oikeanlaisen hyötykuorman niin tietokantaan luodaan käyttäjä vastaamaan payloadin tietoja. Tämän jälkeen järjestelmä voi testa-

ta, että jos kirjautumisesta vastaavaan rajapinnan päätepisteeseen lähetetään oikeat tunnukset, niin palauttaako rajapinta viestin, joka kertoo onnistuneesta kirjautumisesta. Vastaava e2e-testi kuvaa saman prosessin, mutta sisältää myös käyttöliittymän yhtenä elementtinä. Käyttöliittymä todennäköisesti hakee dataa monesta eri rajapinnan endpointista latautuessaan. Viitaten kappaleen 4.1 esimerkkiin tietokantakentän nimenmuutoksesta, voi olla, että kehittäjä on ottanut huomioon sen, että rekisteröitymisen yhteydessä backendista haetaan arvoa “userName”, mutta toinen ominaisuus joka vastaa nimen hakemisesta näytettäväksi käyttöliittymässä kyselee yhä kentän “user_name” arvoa. Tällöin voidaan e2e-testeistä havaita, että käyttäjän nimi ei näy oikein käyttöliittymässä tai käyttöliittymän kysely palauttaa väärän arvon. Rekisteröitymiseen liittyvät testit saattavat mennä läpi, mutta laajempaa kokonaisuutta testaava e2e-testi ilmoittaa virheestä. Lisäksi kehittäjä pystyy varmistumaan siitä, ohjautuuko käyttäjä yhä oikeaan paikkaan muutosten jälkeen ja onko käyttöliittymä yhä ehyt muutoksista huolimatta.

Sen lisäksi, että e2e-testien avulla saadaan supistettua riskejä liittyen henkilöriippuvuuksiin, voidaan testejä käyttää myös dokumentaationa. Aiemmassa kappaleessa 3.2 todetaan, että alati muuttuvan järjestelmän dokumentoinnin olevan raskasta. Siinä missä esimerkiksi rajapintakuvauksiin perinteinen dokumentaatio sopii, on käyttöliittymän ja backendin sisältävän kokonaisuuden dokumentoiminen vastaavalla tavalla hankalaa, sekä resursseja kuluttavaa.

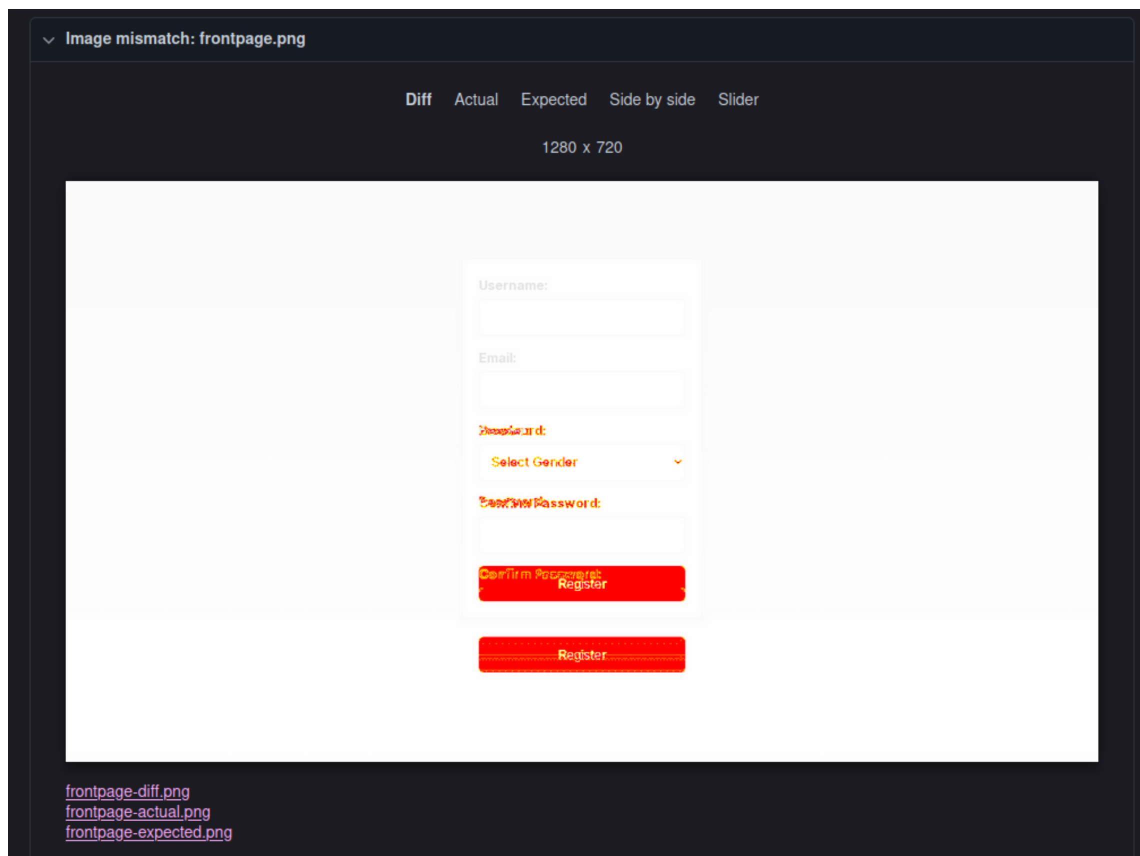
Mikäli järjestelmään on toteutettu e2e-testi testaamaan tietyn kriittisen käyttäjäpolun toimintaa, voi kehittäjä varmistua siitä, että se on todennäköisesti kyseisen käyttäjäpolun haluttu sen hetkinen tapa toimia. Siinä missä perinteiset dokumentaatiot saattavat jäädä kehityksen jalkoihin esimerkiksi ajan tai resurssien puutteen takia, on testit lähes poikkeuksetta ajantasalla halutun toiminnan kanssa. Kuten kappaleessa 2.1 mainitaan, voidaan projektissa käyttää jatkuvan kehityksen prosessia, jossa automaatio ajaa järjestelmään määritellyt testit jokaisen koodi-iteraation

välissä. Tällöin voidaan varmistua jokaisen iteraation välissä siitä, että uudet muutokset eivät ole hajottaneet olemassa olevia ominaisuuksia, joita varten on luotuna testit. Projektin dokumentoinnin ongelmana jatkuvan kehityksen projekteissa on se, että dokumentaatiota pitäisi olla jatkuvasti päivittämässä. Etenkin projekteissa, joissa koodin vaihtuvuus on nopeaa, voi dokumentaation päivittäminen osoittautua monesti tehtäväksi, joka jätetään myöhemmälle ja sen jälkeen unohdetaan.

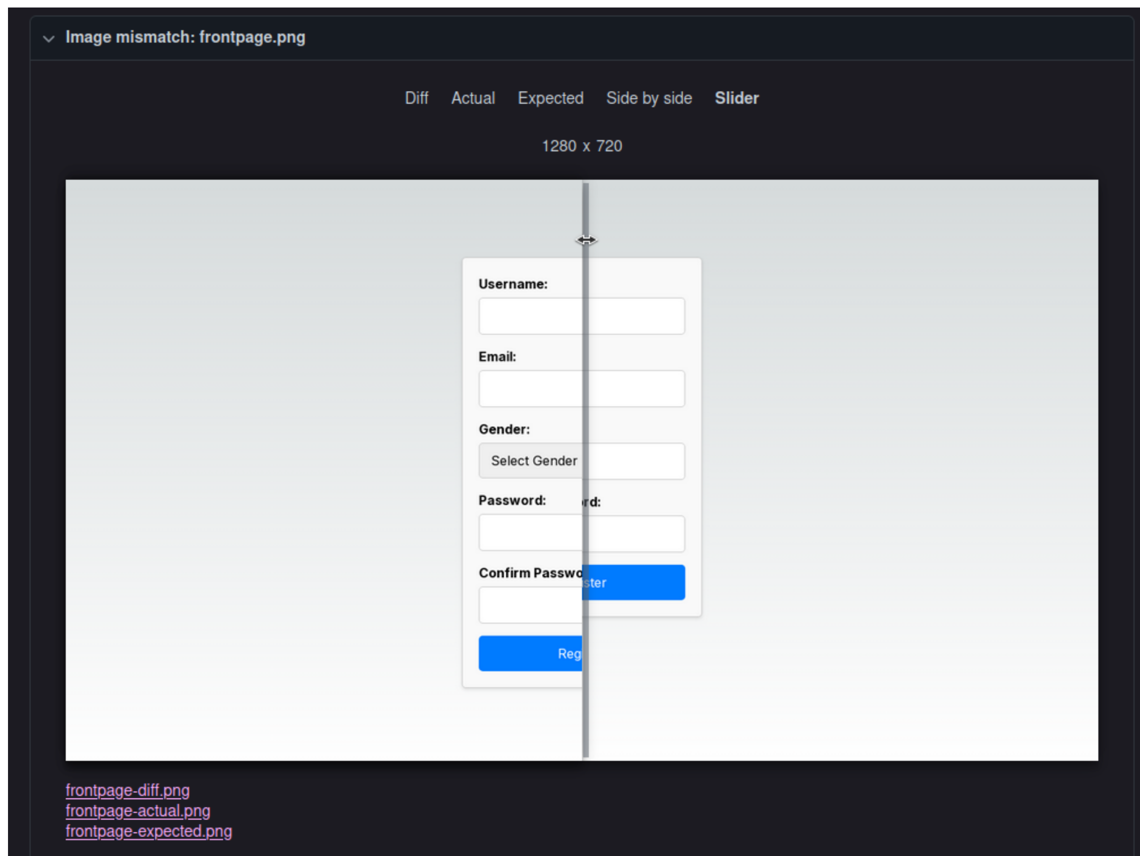
Äärimmäisyyksiin vietynä testauksen roolia kehityksessä kuvaa testivetoinen kehitys (engl. test driven development), jolla tarkoitetaan kehitystapaa, jossa lähtökohtaisesti luodaan ominaisuutta ja sen toimintaa mittaavat testit ennen itse ominaisuutta [55]. Testivetoisen kehityksen hyötynä henkilöriippuvuuksia käsitellessä on se, että jokaista ominaisuutta varten on luotuna testit. Nämä testit itsessään voivat toimia osana dokumentaatiota. Johtuen siitä, että testit ovat osana koodia, on koodialustasta riippuen kehittäjän helpompi löytää niiden perusteella logiikka ominaisuuden toiminnassa verrattuna perinteiseen tekstimuotoiseen dokumentaatioon, joka vain sanallisesti kuvaa ominaisuuden toimintaa. Yleisesti testien käyttämisessä osana dokumentaatiota hyötynä on koodiläheisyys, ja täten ei vaadi kehittäjää poistumaan kehitysympäristöstä kontekstin ääreltä[56]. Hyvä esimerkki kuvaamaan dokumentatiivisten testien toimintaa projektissa henkilöriippuvuuden hallinnan apuna on jo aiemmin mainittu refaktorointi. Mikäli kehittäjällä ei ole kontekstiosaamista projektista, voi refaktorointi vaatia pitkällistä perehtymistä projektin dokumentaatioon. Vaikka projektin dokumentaatio olisi ajantasalla, voi kehittäjä varmistua ominaisuuksien toiminnasta testien avulla. Lisäksi kuten aiemmin mainittua, dokumentatiiviset testit saattavat helpottaa itse toiminnallisuuksien paikallistamista koodikannasta.

Modernit e2e-testauksen työkalut mahdollistavat yhtenä testauksen tapana myös kuvankaappauksien ottamisen ja niiden vertaamisen toisiinsa. Etenkin tällöin e2e-testauksen tapauksessa on testeistä merkittävä hyöty kehittäjälle. Esimerkiksi Playw-

right tarjoaa kehittäjille mahdollisuuden ottaa kuvakaappaus järjestelmän näkymistä ja verrata niitä testien ajojen välillä toisiinsa [57]. Kuvassa 4.2 näkyy miten Playwright raportoi käyttöliittymätason eroista. Tässä esimerkissä käyttäjän rekisteröintikenttään on lisätty uusi kenttä "Gender", minkä takia testi ei enää mene läpi. Kuvassa 4.2 näkyy toinen näkymä edellämainitusta raportista, mistä kehittäjä pystyy vertaamaan miltä näkymä on näyttänyt ennen muutoksia. E2e-testauksen tapauksessa esimerkinkaltaiseen virhetilanteeseen voitaisiin törmätä mikäli käyttöliittymä hakee edellämainitun rekisteröintinäkymän kentät dynaamisesti esimerkiksi rajapinnasta. Kuten kuvasta 4.2 käy ilmi, on kehittäjän helppo verrata alkuperäisen ja muutoksien jälkeisen näkymän eroja. Etu korostuu etenki silloin kun muutokset heijastuvat näkymiin, joita ei ole suoranaisesti muutettu.



Kuva 4.2: Playwrightin näkymä, jossa punaisella merkattuna järjestelmään tallentuneen kuvankaappauksen ja uusien muutosten jälkeen tulleet erot.

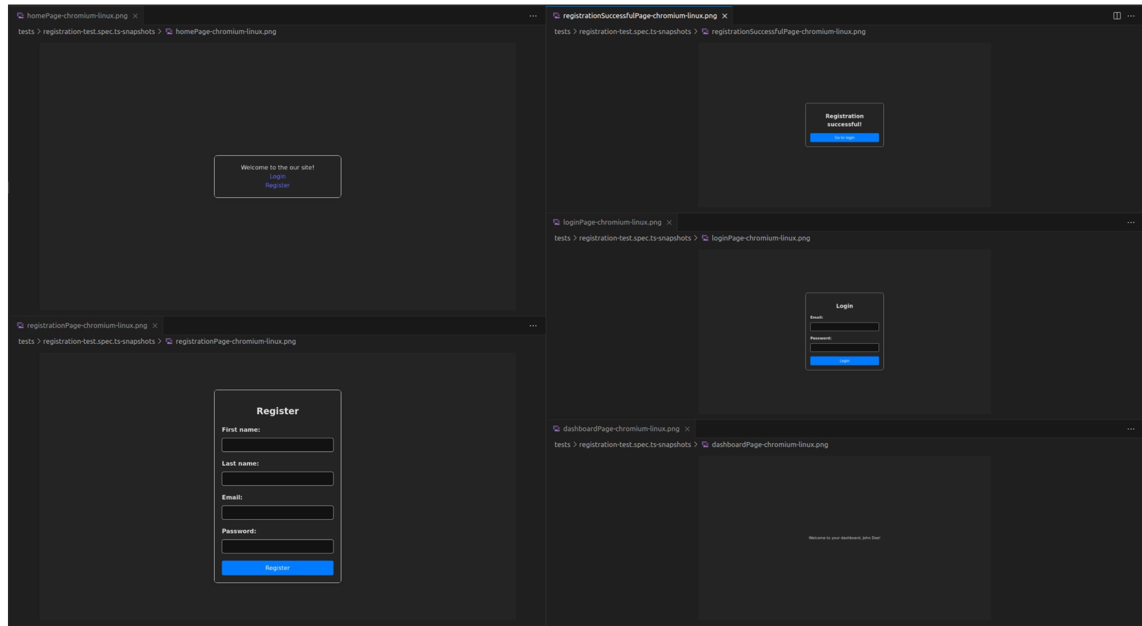


Kuva 4.3: Playwrightin näkymä, jossa kehittäjä pystyy vertaamaan liukusäätimen avulla järjestelmään tallentuneen kuvankaappauksen ja uusimman version eroja.

4.4 Esimerkkejä end-to-end -testauksen hyödyistä ja vaikutuksista

Yksi e2e testien suuri hyöty on niiden mahdollistama tapa järjestelmän toiminnan kulun kuvaamiseen ja sen dokumentoimiseen. Lähtökohtaisesti e2e-testit kuvaavat aina tietyn järjestelmän kannalta kriittisen polun kulkua alusta loppuun. Kuten kappaleissa 4.1 ja 4.2 käy ilmi, voidaan tätä toiminnan kulkua ja dokumentointia käyttää kehittäjien apuna järjestelmän ymmärtämisessä kokonaisuutena. Vaikka e2e-testeillä ei pystytä korvaamaan kirjallista dokumentaatiota, voidaan sitä käyttää sen tukena. Kuvissa 4.4 ja 4.5 näkyy kuinka Playwright-kirjastolla luodulla e2e-testillä pystytään dokumentoimaan haluttu käyttäjän liikehdintä sivulla (engl. happy path/happy flow)[58]. Kuten kappaleessa 4.2 käy ilmi, saattaa dokumentatiiviset testit myös helpottaa kehittäjää paikantamaan mistä virhetilanteet johtuvat, sekä selittämään järjestelmän toimintaa sen ympärillä. Tämä korostuu etenkin, mikäli kehittäjällä ei ole riittävää kontekstiymmärrystä projektista. Tällöin dokumentatiivisilla e2e-testeillä pystytäänkin hallitsemaan henkilöriippuvuuksia.

Toinen e2e-testien hyöty on niiden käyttäminen integraatioiden testauksessa. Kuvassa 4.6 kuvataan eri testityyppien integroitumista muihin järjestelmän osiin, sekä järjestelmän ulkopuolisiin osiin. Kuvasta käy ilmi, että testien järjestelmäkattavuudessa ovat UI-testit, eli käyttöliittymä testit korkeimmalla tasolla ja yksikkötestin vastaavasti alimmalla tasolla. Tässä kontekstissa käyttöliittymätesteihin voidaan lukea myös käyttöliittymän kautta ajettavat e2e-testit. Tässä kontekstissa järjestelmäkattavuudella tarkoitetaan sitä, että yksikkötestejä ajetaan itsenäisinä palikoina, ilman mitään yhteyksiä muihin järjestelmän osiin. Kuvassa 4.7 havainnollistetaan miten yksikkötesteissä järjestelmän ominaisuuden toiminnan kannalta tärkeitä integraatioita voidaan jäljitellä testeissä niihin määritellyillä jäljitelmillä (engl. mock). Tämän takia niillä voidaan saadaan kiinni ongelmia, jotka eivät yksikkötesteistä il-



Kuva 4.4: E2e-testin kuvakaappaukset kertovat kehittäjälle miten loppukäyttäjän kuuluisi ohjautua käyttöliittymässä ja miltä näkymien pitäisi näyttää

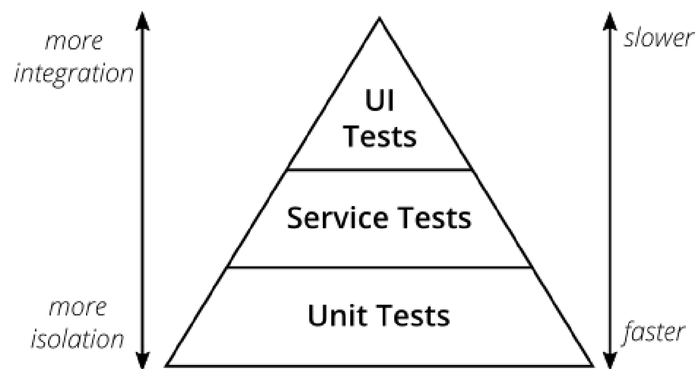
mene.

Esimerkki tällaisesta tapauksesta on, että järjestelmä A on integroitu järjestelmään B. Järjestelmä A sisältää autojen tietoja, kuten rekisterinumeron, merkin, mallin, valmistusvuoden ja värin. Järjestelmä A tekee joka yö haun järjestelmään B, josta se hakee uusien autojen tiedot. Järjestelmä A olettaa, että järjestelmä B palauttaa listan autoista, mikä sisältää avaimet “registration_number”, “make”, “model”, “year”, “color”. Haun jälkeen järjestelmä A käy läpi järjestelmä B:n palauttamat tietueet, rikastaa jo järjestelmään tallennetujen autojen tietoja ja tallentaa uudet autot järjestelmään. Järjestelmässä A on myös testit, joissa järjestelmää B jäljitellään siten, että sinne tehtävä kutsu palauttaa edellämainitun listan autoista ja niiden tiedoista. Jos järjestelmään B tehdään muutos, kuten tietueen avaimen muuttaminen tai esimerkiksi rajapinnan autentikointimetodin vaihtaminen, eivät järjestelmään A toteutetut yksikkötestit päivity automaattisesti vastaamaan näitä muutoksia. Tässä tapauksessa mikäli integraation testaus olisi suoritettuna e2e-testauksella, jossa ote-

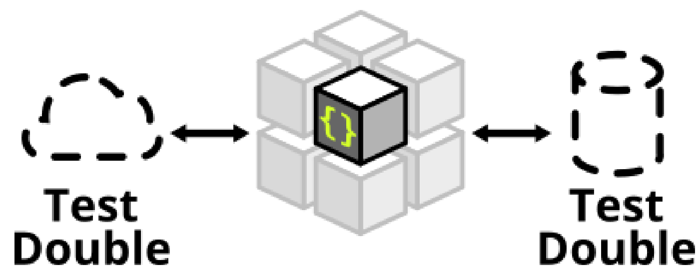
```
registration-test.spec.ts
tests > registration-test.spec.ts > test("Registration and login test") callback
1  import { expect, test } from "@playwright/test"
2
3  test("Registration and login test", async ({ page }) => {
4      await page.goto("http://localhost:5173/")
5      await expect(page).toHaveScreenshot("homePage.png")
6      await page.getByRole("link", { name: "Register" }).click()
7
8      await expect(page).toHaveScreenshot("registrationPage.png")
9      await page.locator('input[name="firstName"]').fill("John")
10     await page.locator('input[name="lastName"]').fill("Doe")
11     await page.locator('input[name="email"]').fill("john.doe@email.com")
12     await page.locator('input[name="password"]').fill("johndoepassword123!")
13
14     await page.getByRole("button", { name: "Register" }).click()
15     await expect(page).toHaveScreenshot("registrationSuccessfulPage.png")
16     await page.getByRole("button", { name: "Go to login" }).click()
17
18     await expect(page).toHaveScreenshot("loginPage.png")
19     await page.locator('input[name="email"]').fill("john.doe@email.com")
20     await page.locator('input[name="password"]').fill("johndoepassword123!")
21     await page.getByRole("button", { name: "Login" }).click()
22
23     await expect(page).toHaveScreenshot("dashboardPage.png")
24 })
25
```

Kuva 4.5: Kuvankaappausten lisäksi kehittäjä pystyy tulkitsemaan haluttua käyttäjän liikehdintää testien koodista

taan oikeasti yhteys järjestelmän B rajapintaan, pystytään havaitsemaan, että testin tulos ei vastaa odotettua. Ongelmana integraatioiden jäljittelyssä on se, että mikäli integroitavaan järjestelmään tulee muutos, täytyy kyseisiä jäljitelmiä muokata. Tällöin ulkoisiin järjestelmiin tulevat muutokset eivät välttämättä vastaa ohjelmistoon jäljiteltyä integraation toimintaa. E2e-testien tapauksessa ulkoisiin palveluihin tulevat muutokset hajottavat testit, jolloin järjestelmän kehittäjät huomaavat helpommin nämä ongelmat.



Kuva 4.6: Eri testityyppien ajamiseen vaadittava aika kasvaa suhteessa niiden järjestelmäkattavuuteen [59]



Kuva 4.7: Yksikkötesteissä ominaisuuden ulkopuoliset osat korvataan niitä jäljittelevillä testiosilla [59]

E2e-testit mahdollistavat myös järjestelmän kannalta kriittisten ominaisuuksien manuaalitestaamisen helpottamisen. Modernit e2e-testauksessa käytettävät työkalut mahdollistavat sen, että testejä voidaan ajaa kehittäjän laitteella niin, että ne suorittavat tietyn tehtävän ja jättävät järjestelmän siihen tilaan. Esimerkiksi jos kehittäjä tekee muutoksia verkkokaupan ostoskorin toimintaan, voi hän halutessaan auto-

matisoida e2e-testillä järjestelmässä käyttäjän luonnin, verkkokaupassa navigoinnin tuotesivulle ja tuotteen lisäämisen ostoskoriin. Tämän jälkeen kehittäjä voi käydä manuaalisesti tarkistamassa muutoksen seurauksia ostoskorinäkymässä ilman, että joutuu käyttämään aikaa manuaalisesti sellaisten järjestelmän osien läpikäymiseen, joita ei juuri nyt haluta testata.

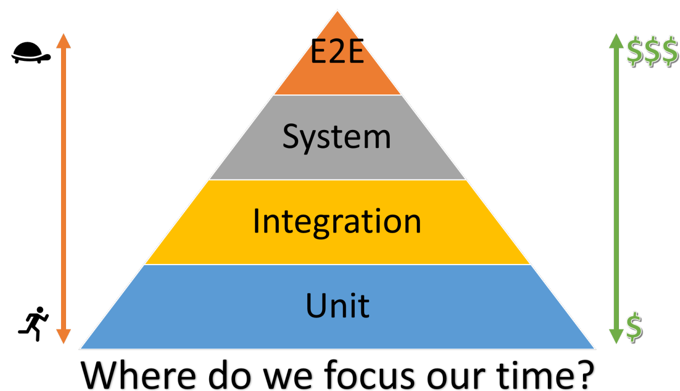
Vaikka edellämainitun esimerkin tilanteessa voitaisiin tehdä järjestelmään kehittäjille suunnattuja ominaisuuksia, kuten nappi, joka automaattisesti luo tietokantaan käyttäjän tietyillä tiedoilla, on siinä tietyt riskit. Esimerkiksi jokin ominaisuus voi muuttua käyttöliittymässä ja tämä automaattisesti luotu demokäyttäjä ei enää vastaa täysin käyttöliittymän kautta luotua käyttäjää. Tällainen manuaalinen testaaminen on hyvin aikaavievää ja kuormittavaa. Esimerkin kaltainen manuaalinen käyttäjien luonti manuaalitestausta varten voidaankin laskea tietyllä tavalla tekniseksi velaksi. Kuten kappaleessa 2.2 käy ilmi, on tällaisella vaikutus työntekijän moraaliin ja täten tuottavuuteen. Ham Vocke viittaakin artikkelissa “The Practical Test Pyramid” siihen miten jatkuva manuaalitestaaminen rasittaa kehittäjää seuraavalla tavalla:

> “It’s obvious that testing all changes manually is time-consuming, repetitive and tedious. Repetitive is boring, boring leads to mistakes and makes you look for a different job by the end of the week.” [59]

Kehittäjien lisäksi e2e-testejä voidaankin edellämainitulla tavalla käyttää myös asiakkaan toimesta. Mikäli yritys kehittää tuotetta toiselle yritykselle, jonka vastuulla on todeta toimiiko uudet ominaisuudet halutulla tavalla, voidaan e2e-testien avulla toteutettua automaatiota käyttää myös tässä apuna. Tällöin asiakas pystyy testaamaan järjestelmän uusia ominaisuuksia ilman, että joutuvat esimerkin tapauksessa tekemään mahdollisesti useita uusia käyttäjiä manuaalisesti.

4.5 Kustannustehokkuus ja end-to-end -testaus

E2e-testien suurimpana ongelmana on niiden toteuttamisen hinta. Sen lisäksi, että e2e-testit ovat hitaita ja kalliita toteuttaa on niiden ylläpito myös työlästä. Kuvassa 4.8 näkyy, että yksikkötestit ovat nopeimpia toteuttaa, kun taas e2e-testit ovat hitaimpia ja kalliimpia testejä toteuttaa. Tästä johtuen korkean tason testejä, kuten e2e-testit, tulisi toteuttaa harkiten ja vain sellaisiin paikkoihin joissa niille on tarvetta. [59]



Kuva 4.8: Testaamisen hinta [60]

E2e-testien ongelmana kustannustehokkuuden näkökulmasta on myös niiden laajuus. Yksikkötestien tapauksessa voidaan testata eristetysti yksittäistä toimintoa. Esimerkki tällaisesta toiminnosta on funktio, jota kutsutaan kun käyttäjä painaa “Lisää ostoskoriin”-nappia verkkokaupassa. Yksikkötestillä voidaan kutsua tätä funktiota ja antaa sille parametrina tuotteen id, jonka perusteella käyttäjän ostoskori päivittyy. Testitapauksessa voidaan kutsua tätä funktiota ja tarkistaa onko tuote päätynyt todella ostoskoriin. Todellisuudessa kuitenkin nappi, jota käytetään kutsumaan tätä funktiota saattaa laukaista useampia eri funktioita suoraan tai epäsuorasti. Esimerkiksi käyttäjälle voidaan näyttää käyttöliittymässä ponnahdusikkuna ostoskorin päivittymisen yhteydessä. Tämän ponnahdusikkunan näyttämisestä vastaava ominaisuus ei välttämättä ole suoraan kytköksissä edellämainittuun funk-

tioon. E2e-testien tapauksessa eri toiminnallisuuksia ei kuitenkaan eriytetä toisistaan vaan ne tapahtuvat aina kuten ne on suunniteltu toimimaan varsinaisen käyttäjän kanssa. Tällöin tulee e2e-testejä suunnitellessa ja toteuttaessa harkita mitä kaikkia järjestelmän polkuja halutaan testata ja kuinka laajasti. Edellämainitussa “Lisää ostoskoriin”-napin esimerkissä voi olla mahdollista, että ponnahdusikkuna näytetään vain, jos verkkokaupassa on käynnissä aktiivinen tarjouskampanja, käyttäjän ip-osoite on tietystä maasta tai muuten sattumanvaraisesti osalle käyttäjistä. Mikäli kaikille mahdollisille tapauksille lähdettäisiin luomaan omaa testitapausta, kuluttaisi se paljon resursseja. Nämä resurssit lähtökohtaisesti olisivat aina pois jostain muusta järjestelmän kehitykseen varatusta ajasta.

E2e-testauksen tapauksessa kannattaakin lähestyä testien luomista sillä näkökulmalla, että testit käsittävät vain oleellisimpia, liiketoiminnan kannalta kriittisimpiä ominaisuuksia, ja näistäkin valitaan vain kaikista oleellisimmat tapaukset, joiden toimintaa mitataan. Esimerkki tällaisesta testauksen ajattelutavasta on kappaleessa 4.3 mainittu “happy path”-testaus. “Happy path”-testauksessa tarkoituksena on testata, että ideaalitilanteessa järjestelmän ydintoiminnallisuus (engl. core feature) toimii. Verkkokaupan tapauksessa tällainen testi voisi testata, että käyttäjä pystyy selaamaan tuotteita, lisäämään niitä ostoskoriin, syöttämään toimivat maksutiedot ja viimeistelemään maksuprosessin [58]. Vaikka järjestelmässä olisi ominaisuuksia, kuten ponnahdusikkuna, mikäli maksukortin tiedot ovat väärin, ei näitä “Happy path”-testauksessa käsiteltäisi. Tämä yksinkertaistaa testejä huomattavasti ja vähentää niihin käytettävää aikaa kuitenkin pitämällä testit relevanttina liiketoiminnan kannalta kriittisiä ominaisuuksia ajatellen. Mikäli tällainen testi menee läpi, on esimerkin tapauksessa käyttäjän mahdollista suorittaa ostotapahtuma onnistuneesti vaikka jotkin, ei testatut käyttäjäpolut saattaisivatkin johtaa epäonnistuneeseen ostoprosessiin.

5 Kyselytutkimus

5.1 Kysely end-to-end -testauksen vaikutuksista tekniseen velkaan ja henkilöriippuvuuteen

Kyselyrunko koostuu viidestä osa-alueesta: kehittäjän kokemus ja tausta, ajatukset ja kokemukset e2e-testaamisesta, teknisen velan hallinta e2e-testeillä, henkilöriippuvuuksien hallinta e2e-testeillä sekä loppuajatukset ja ehdotukset. Kyselyn tavoitteena on saada näkemystä siitä, miten kehittäjät kokevat e2e-testien vaikuttavan teknisen velan hallintaan ja henkilöriippuvuuksien hallintaan, tunnistaa onko e2e-testien käyttöönotossa haasteita, sekä kerätä yleisiä mielipiteitä e2e-testien vaikutuksista sovelluskehitysprojekteihin.

Kysymyksissä e2e-testeillä viitataan testeihin, jotka simuloivat oikean käyttäjän toimia käyttöliittymässä ja niillä testataan tietyn järjestelmän kannalta kriittisen toiminnallisuuden toimivuutta.

Kysymyksissä teknisellä velalla tarkoitetaan käsitettä, jossa kehityksen aikana ohjelmistoon tehdään ratkaisuja, jotka säästävät rahaa lyhyellä aikavälillä, mutta pitkällä aikavälillä nostavat kokonaiskustannuksia. Kun ohjelmistoa kehitetään, pystytään prosessia nopeuttamaan tekemällä ratkaisuja, jotka koodin kannalta eivät ole laadullisesti hyvällä tasolla, mutta tällä saadaan tuotettua toimivia kokonaisuuksia asiakkaalle. Tällöin kerrytetään teknistä velkaa.

Kysymyksissä henkilöriippuvuudella tarkoitetaan mittaria, jolla tarkastellaan si-

tä miten riippuvainen projektin eteneminen on tietyistä tekijöistä. Mittari kuvaa projektin resilienssiä yhtäkkiselle tilanteelle, jossa projektin kehittäjät vaihtuvat.

TK2:n tueksi laaditut kysymykset kohdistuivat kehittäjien kokemuksiin ja käytäntöihin liittyen testauksen kattavuuteen, teknisen velan syntyyn ja henkilöriippuvuuksien hallintaan testauksen keinoin. Kyselytutkimus suunniteltiin siten, että se mahdollisti eri näkökulmien keräämisen kehittäjiltä.

1. Kehittäjän kokemus ja tausta testaamisesta:
   (a) Kuinka monta vuotta kokemusta sinulla on ohjelmistokehittäjänä?
   (b) Kuinka monta vuotta olet työskennellyt ohjelmistojen automaatiotestauksen parissa?
   (c) Sisältyykö testien toteuttaminen nykyiseen työnkuvaasi?
      i. Kyllä
      ii. Jonkin verran
      iii. Ei
   (d) Mitä testausmenetelmiä tai -tyylejä projekteissasi yleensä käytetään? Esimerkiksi yksikkö-, integraatio tai e2e-testit
2. Käsitykset ja kokemukset e2e-testaamisesta
   (a) Minkä asian koet merkittävimpänä hyötynä e2e-testien toteuttamisesta?
   (b) Minkä asian koet merkittävimpänä haittana e2e-testien toteuttamisesta?
   (c) Koetko, että e2e-testien sisällyttäminen projektiin olisi hidastanut tai nopeuttanut kehitystä?
      i. Kyllä, hidastanut
      ii. Kyllä, nopeuttanut
      iii. Ei vaikutusta

(d) Koetko, että e2e-testien ylläpito vaatii enemmän vaivaa verrattuna matalamman tason testeihin, kuten yksikkötesteihin?

i. Kyllä
ii. Jonkin verran
iii. En

3. Teknisen velan hallinta e2e-testien kautta

(a) Kohtaatko nykyisessä työssäsi teknistä velkaa?

i. Kyllä
ii. Jonkin verran
iii. En

(b) Pyritäänkö teknisen velan määrään aktiivisesti puuttumaan?

i. Kyllä
ii. Jonkin verran
iii. Ei

(c) Koetko, että e2e-testit tukevat teknisen velan hallintaa, erityisesti koodin refaktoroinnin yhteydessä?

i. Kyllä
ii. Jonkin verran
iii. En

(d) Koetko, että e2e-testit ovat tärkeämmässä roolissa teknisen velan hallinnassa verrattuna matalamman tason testeihin, kuten yksikkötesteihin?

i. Kyllä
ii. Jonkin verran
iii. En

4. Henkilöriippuvuuksien hallinta e2e-testien kautta
   (a) Kohtaatko nykyisessä työssäsi henkilöriippuvuutta?
      i. Kyllä
      ii. Jonkin verran
      iii. En
   (b) Pyritäänkö nykyisessä työssäsi henkilöriippuvuuksien hallintaan aktiivisesti puuttumaan?
      i. Kyllä
      ii. Jonkin verran
      iii. Ei
   (c) Koetko, että e2e-testit helpottavat järjestelmän toiminnan kannalta kriittisten polkujen toiminnan ymmärtämistä?
      i. Kyllä
      ii. Jonkin verran
      iii. En
   (d) Koetko, että e2e-testit helpottavat uusien kehittäjien sisäänajamista projektiin?
      i. Kyllä
      ii. Jonkin verran
      iii. En
   (e) Koetko, että e2e-testejä voidaan käyttää dokumentoimaan tiettyjä järjestelmän toiminnan kannalta kriittisiä polkuja? Esimerkki kriittisestä polusta on verkkokaupan tapauksessa seuraava: Käyttäjä päätyy etusivulle, lisää tuotteen ostoskoriin, etenee ostoskoriin, viimeistelee tilauksen ja saa sähköpostiinsa tilausvahvistuksen.

i. Kyllä

ii. Jonkin verran

iii. En

(f) Koetko, että e2e-testeillä voidaan korvata perinteinen dokumentaatio, jossa ominaisuuksien toiminta kuvataan tekstimuodossa?

i. Kyllä

ii. Jonkin verran

iii. En

5. Loppuajatukset ja ehdotukset

(a) Koetko, että voisit toteuttaa projektiin e2e-testejä sillä näkökulmalla, että niiden tarkoitus olisi hallita teknistä velkaa?

i. Kyllä

ii. Jonkin verran

iii. En

(b) Koetko, että voisit toteuttaa projektiin e2e-testejä sillä näkökulmalla, että niiden tarkoitus olisi hallita henkilöriippuvuuksia?

i. Kyllä

ii. Jonkin verran

iii. En

(c) Herättikö edellä mainitut kysymykset mitään kysymyksiä?

5.2 Analyysi end-to-end -testauksen vaikutuksista

Kyselyyn vastasi seitsemän henkilöä, joiden ohjelmistokehitystaustan pituus vaihteli viidestä vuodesta kahteentoista vuoteen, keskiarvon ollessa seitsemän vuotta. Kuusi

vastaajista kertoi, että testien toteuttaminen kuuluu heidän nykyiseen työnkuvaansa, yksi arvioi sen kuuluvan “jonkin verran”.

5.2.1 Kokemukset e2e-testaamisesta

Kyselyvastausten perusteella yleisimmin käytetty testausmenetelmä projekteissa on yksikkötestaus, joka mainittiin kaikissa vastauksissa. Integraatiotestausta käytetään myös laajalti, mutta sen käyttö riippuu vastaajan mukaan usein projektin tarpeista. E2e-testaus mainittiin satunnaisesti tai harvemmin käytettynä menetelmänä. Funktionaalinen testaus mainittiin vain kerran, yhdistettynä muihin testausmenetelmiin. Taulukossa 5.1 on koottuna eri testausmenetelmät ja niiden mainintojen lukumäärät vastausten perusteella.

Taulukko 5.1: Käytetyt testausmenetelmät ja -tyylit

Testausmenetelmä	Mainintojen määrä	Huomiot
Yksikkötestaus	7	Mainittiin kaikissa vastauksissa.
Integraatiotestaus	5	Käytetään projekteista riippuen.
End-to-End testaus	5	Käytetään satunnaisesti tai harvemmin.
Funktionaalinen testaus	1	Mainittu kerran, yhdessä muiden testien kanssa.

Tarkastellessa vastauksia e2e-testien hyödyistä yleisimmin vastauksissa nousi esiin e2e-testien kyky simuloida loppukäyttäjän toimia. Käyttäjän simuloinnin koettiin helpottavan esimerkiksi eri laitteilla testaamista, vähentävän manuaalisen testaamisen määrää ja testien vaatimaa ylläpitotyötä. Taulukossa 5.2 on listattuna vastausten perusteella koodut e2e-testien hyödyt ja niiden selitykset. Vastauksista nousi möys esiin vastaajien kokemukset siitä, että e2e-testeillä pystytään niiden laajuuden ansiosta taklaamaan kappaleessa 4.1 esitettyjä regressioita. Regressiotestausta lukuunottamatta vastauksista ei käynyt ilmi, että vastaajat kokisivat e2e-testien olevan merkittävässä roolissa teknisen velan tai henkilöriippuvuuksien hallinnassa.

E2e-testien vaikutusta kehityksen nopeuteen ja ylläpidolliseen työmäärään tar-

Taulukko 5.2: E2e-testauksen hyödyt

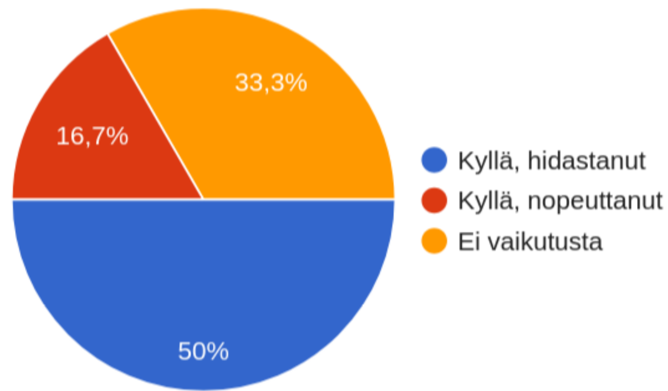
Hyöty	Selitys
Huolettomammat muutokset koodiin	Yleisimpien käyttötapausten toimivuus varmistetaan automaattisesti, mikä lisää kehitysvapautta.
Luottamus järjestelmään ja regressioiden välttäminen	Helpottaa jatkokehitystä ja varmistaa, että aiemmin toimineet toiminnallisuudet toimivat edelleen.
Eri laiteiden simulointi	Pystytään simuloimaan eri laitteita, joka nopeuttaa testaamista.
Käyttäjänäkökulman automatisointi	Vähentää manuaalista testausta, sillä käyttäjän toimintojen simulointi vastaa tosielämän käyttöä.
Ketjutettujen toimintojen testaus	Voidaan testata ketjutettuja toimintoja siten, että data liikkuu usean prosessin läpi saman testin sisällä
Virheiden käsittelyn testaus	Pystytään testaamaan järjestelmän virheidenkäsittelyä.
Käyttötapausten testaus yksittäisten funktioiden sijaan	Käyttäjäpolku ei välttämättä muutu, joka vähentää testien ylläpitotyötä.

kastellessa huomataan, että vastauksissa ilmenee kappaleessa 4.5 todettu e2e-testien ongelma kustannustehokkuudessa. Kuvassa 5.1 nähdään, että puolet vastaajista (50 %) koki e2e-testien hidastavan kehistystä, kun taas 16,7 % koki sen nopeuttavan. Loput vastaajista (33,3 %) ei kokenut e2e-testien sisällyttämisen projektiin vaikuttavan kehityksen nopeuteen. Kuvassa 5.2 näkyy miten vastaajat kokivat e2e-testien ylläpitoon liittyvän vaivan määrän verrattuna matalamman tason testeihin. Vastauksista ilmenee, että kaikki vastaajat kokivat e2e-testien ylläpidon vaativan enemmän vaivaa verrattuna matalamman tason testeihin.

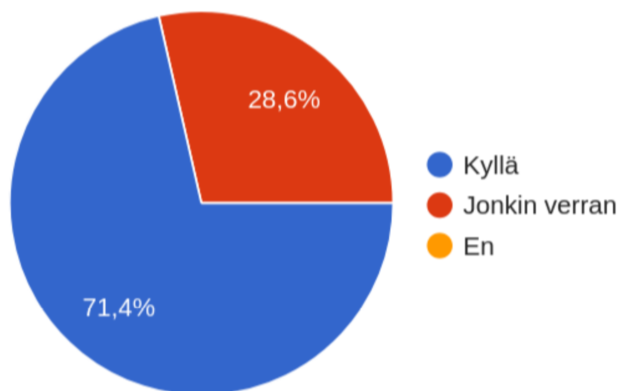
5.2.2 Teknisen velan hallinta e2e-testien kautta

Vastaajista 71,7 % vastasi kohtaavansa teknistä velkaa. 83,3% vastasi teknisen velan kuitenkin olevan asia, johon pyritään aktiivisesti puuttumaan.

Kuvassa 5.3 nähdään, että vastaajista kaikki kokivat e2e-testien tukevan teknisen velan hallintaa. Kuitenkin vastaajista suurin osa (85,8 %) koki, että e2e-testit

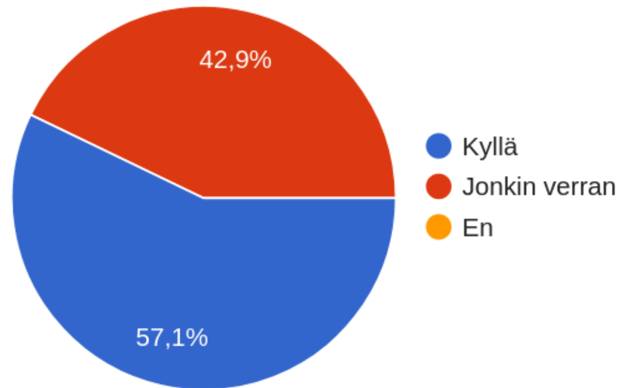


Kuva 5.1: Koetko, että e2e-testien sisällyttäminen projektiin olisi hidastanut tai nopeuttanut kehitystä?

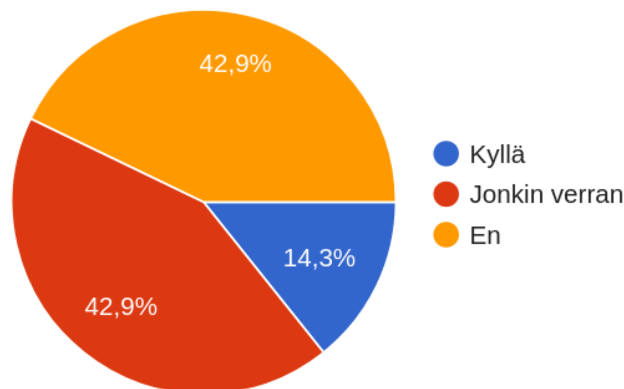


Kuva 5.2: Koetko, että e2e-testien ylläpito vaatii enemmän vaivaa verrattuna matalamman tason testeihin, kuten yksikkötesteihin?

eivät ole merkittävämmässä roolissa teknisen velan hallinnassa verrattuna mitä matalamman tason testit, kuten yksikkötestit (Kuva 5.4).



Kuva 5.3: Koetko, että e2e-testit tukevat teknisen velan hallintaa, erityisesti koodin refaktoroinnin yhteydessä?



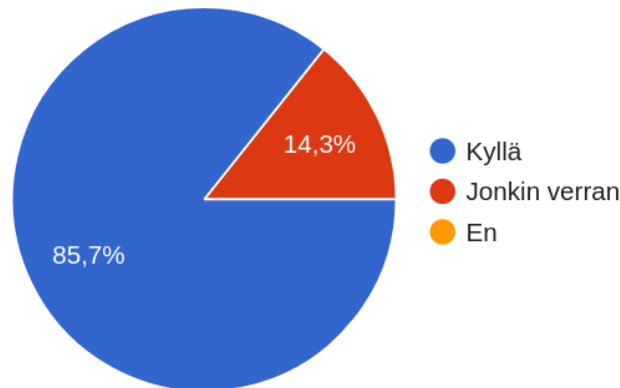
Kuva 5.4: Koetko, että e2e-testit ovat tärkeämmässä roolissa teknisen velan hallinnassa verrattuna matalamman tason testeihin, kuten yksikkötesteihin?

5.2.3 Henkilöriippuvuuksien hallinta e2e-testien kautta

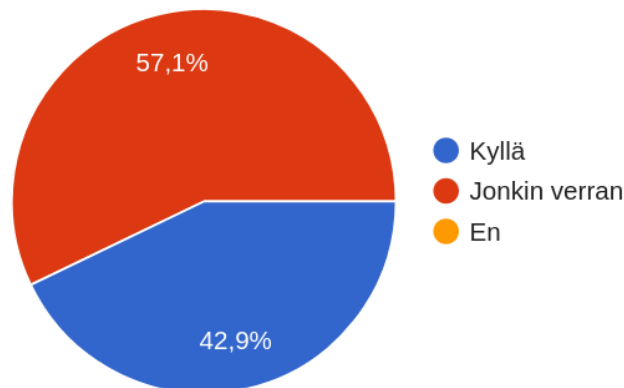
Vastanneista 85,7 % ilmoitti kohtaavansa henkilöriippuvuutta ainakin jonkin verran nykyisessä työssään. 85,7 % vastasi, että henkilöriippuvuuksien hallintaan pyritään aktiivisesti puuttumaan.

Vastaajista kaikki olivat sitä mieltä, että e2e-testit helpottavat ainakin jonkin verran järjestelmän kannalta kriittisten polkujen toiminnan ymmärtämistä (Kuva

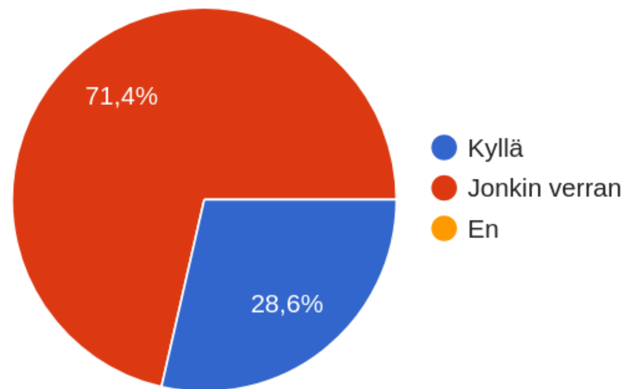
5.5), helpottavat ainakin jonkin verran uusien kehittäjien sisäänajamista projekteihin (Kuva 5.6) ja niitä voidaan käyttää dokumentoimaan tiettyjä järjestelmän toiminnan kannalta kriittisiä polkuja (Kuva 5.7. Kyselyn mukaan e2e-testeillä ei voida täysin korvata perinteistä tekstimuotoista dokumentaatiota. Yksikään vastaaja ei uskonut niiden voivan toimia täytenä korvikkeena. Kuitenkin 42,9 % vastaajista koki, että ne voivat osittain korvata perinteistä dokumentaatiota. Enemmistö, 57,1 %, koki, ettei e2e-testeillä voi korvata dokumentaatiota lainkaan (Kuva 5.8). Tulokset osoittavat, että vaikka e2e-testit voivat tukea dokumentaation laadintaa, niiden rooli ei riitä perinteisen dokumentaation syrjäyttämiseen.



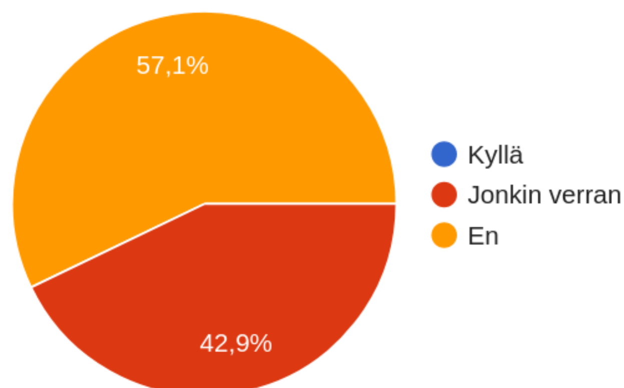
Kuva 5.5: Koetko, että e2e-testit helpottavat järjestelmän toiminnan kannalta kriittisten polkujen toiminnan ymmärtämistä?



Kuva 5.6: Koetko, että e2e-testit helpottavat uusien kehittäjien sisäänajamista projektiin?



Kuva 5.7: Koetko, että e2e-testejä voidaan käyttää dokumentoimaan tiettyjä järjestelmän toiminnan kannalta kriittisiä polkuja?

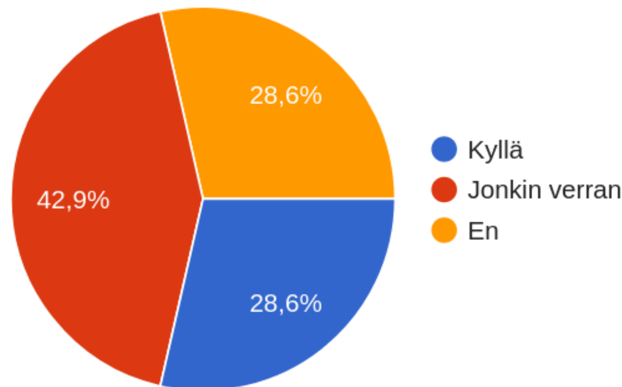


Kuva 5.8: Koetko, että e2e-testeillä voidaan korvata perinteinen dokumentaatio, jossa ominaisuuksien toiminta kuvataan tekstimuodossa?

5.2.4 Ajatukset e2e-testaamisesta teknisen velan ja henkilöriippuvuuksien hallinnassa

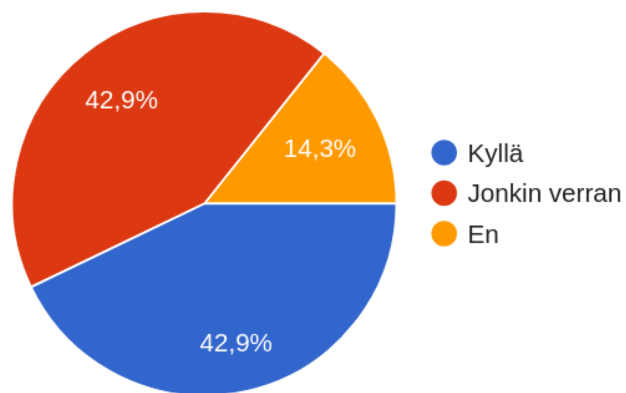
Kyselyn lopussa vastaajilta kysyttiin ajatuksia siitä, voisivatko he alkaa toteuttamaan projekteissaan e2e-testejä sillä näkökulmalla, että niiden tarkoituksena olisi suoraan hallita teknistä velkaa tai henkilöriippuvuuksia.

Tarkastellessa vastauksia liittyen tekniseen velkaan, havaittiin, että vastaukset jakautuivat tasaisesti eri vaihtoehtojen kesken. Kuvassa 5.9 näkyy, että 28,6 % vastaajista kokevat, että voisivat toteuttaa e2e-testejä sillä näkökulmalla, että niiden tarkoitus olisi hallita teknistä velkaa. Jonkin verran tätä mahdollisuutta koki toteuttamiskelpoisena 42,9 % vastaajista. 28,6 % vastaajista koki, etteivät voisi toteuttaa testejä tällä näkökulmalla.



Kuva 5.9: Koetko, että voisit toteuttaa projektiin e2e-testejä sillä näkökulmalla, että niiden tarkoitus olisi hallita teknistä velkaa?

Henkilöriippuvuuden tapauksessa 42,9 % vastaajista koki, että voisivat toteuttaa e2e-testejä projekteihinsa sillä näkökulmalla, että niiden tarkoitus olisi hallita henkilöriippuvuuksia. 42,9 % koki mahdollisuuden jonkin verran toteuttamiskelpoisena ja 14,3 % koki etteivät voisi toteuttaa testejä tällä näkökulmalla (Kuva 5.10).



Kuva 5.10: Koetko, että voisit toteuttaa projektiin e2e-testejä sillä näkökulmalla, että niiden tarkoitus olisi hallita henkilöriippuvuuksia?

6 Johtopäätökset ja jatkotutkimusaiheet

6.1 Yhteenveto tutkimustuloksista ja niiden merkitys

Teknistä velkaa syntyy väistämättä projektien edetessä, kuten kappaleessa 2.1 tarkastelluista kirjallisuuslähteistä käy ilmi. Kappaleessa 3 henkilöriippuvuuden suhdetta tekniseen velkaan tarkastellessa huomattiin, että tekninen velka on paremmin hallittavissa, mitä paremmin kehittäjä ymmärtää koodikantaa.

TK1:n kannalta saatiin selville, että tarkastellessa teknisen velan ja henkilöriippuvuuden suhdetta suurin uhkatekijä on teknisen velan aiheuttama kontekstin monimutkaistuminen. Monimutkaistuminen taas aiheuttaa merkittäviä regressiohaasteita. Eri tekijät, kuten projektin monoliittisuus, vanheneva koodikanta tai heikko dokumentaatio kasvattavat henkilöriippuvuuden riskiä. Kuitenkin teknisen velan rooli kaikissa näissä on hyvin merkittävä, sillä se kasvattaa vaadittavaa kontekstiymmärrystä entisestään monimutkaistamalla järjestelmän toimintoja. Tekninen velka heikentää kehittäjien kykyä ymmärtää projektin kokonaisuutta, mikä puolestaan vaikeuttaa uusien ominaisuuksien kehittämistä ja saattaa jopa johtaa teknisen velan noidankehään.

Koska teknistä velkaa syntyy aina ja sen hallinta helpottuu mitä paremmin ke-

hittäjä ymmärtää ohjelmiston toimintaa, otettiin tutkimuksessa tarkasteluun e2e-testien vaikutus teknisen velan ja henkilöriippuvuuksien hallintaan.

TK2:sta tarkastellessa, kappaleessa 4 käy ilmi, että e2e-testeillä pystytään teoriassa auttamaan kehittäjää ymmärtämään järjestelmän kannalta kriittisten polkujen toimintaa. Mikäli kehittäjät on helpompi saada ymmärtämään järjestelmän toimintaa, vähentää tämä henkilöriippuvuuksia ja täten helpottaa teknisen velan hallintaa. Kappaleessa 5 suoritetun kyselytutkimuksen tarkoituksena oli selvittää, miten kehittäjät kokevat e2e-testien roolin henkilöriippuvuuksien ja teknisen velan hallinnan välineenä.

Kyselyssä vastaajien vastausten perusteella kävi ilmi, että e2e-testien roolia ei koeta kovin merkittävänä tekijänä teknisen velan hallinnan kannalta. Vastauksia tarkastellessa oli kuitenkin havaittavissa, että e2e-testien rooli koettiin merkittävämmäksi henkilöriippuvuuksien kohdalla. Syynä tähän on luultavasti se, että vaikka henkilöriippuvuuksien ja teknisen velan välillä on yhteys, vaatii teknisen velan hallinta syvällisempää järjestelmän ymmärrystä mitä e2e-testit tarjoavat. Kuvasta 5.5 käy ilmi, että 85,7 % vastaajista pitää e2e-testejä hyödyllisinä järjestelmän kannalta kriittisten polkujen toiminnan kuvaamisessa. Tämä osoittaa, että e2e-testit voivat toimia lisätyökaluna teknisen velan hallinnassa.

TK2:n kannalta keskeinen tulos on se, että e2e-testien pääasiallinen hyöty tässä yhteydessä liittyy niiden dokumentointiominaisuuksiin. Ne kuvaavat tiettyjä järjestelmän kannalta kriittisiä polkuja, mutta eivät ole suoranaisesti teknisen velan hallinnan työkalu.

6.2 Mahdolliset jatkotutkimusaiheet ja suositukset tuleville tutkimuksille

Kyselytutkimuksen laajuus oli suppea, vain seitsemän henkilöä. Tutkimuksen laajuutta kasvattamalla saataisiin enemmän aineistoa, minkä pohjalta pystyisi tekemään luotettavampia johtopäätöksiä. Jatkotutkimuksissa olisi myös syytä ottaa tarkasteluun yksikkötestauksen rooli henkilöriippuvuuden ja teknisen velan hallinnassa. Taulukossa 5.1 käy ilmi, että kaikki kyselyyn vastanneet henkilöt mainitsivat vastauksissaan yksikkötestauksen, kun heiltä kysyttiin käytetyistä testausmenetelmistä ja -tyyleistä.

Toinen mielenkiintoinen näkökulma jatkotutkimuksiin on korkeamman tason testaussuunnitelma. Tällä tarkoitetaan sitä, että mikä varsinainen tavoite järjestelmään toteutettavilla testeillä on. Jos testejä toteutetaan järjestelmään ensisijaisesti luottamuksen lisäämiseksi ja regressioiden ehkäisemiseksi, niiden vaikutus voi poiketa merkittävästi siitä, että testit suunniteltaisiin nimenomaan henkilöriippuvuuksien vähentämiseen.

Lähdeluettelo

[1] T. Besker, A. Martini ja J. Bosch, ”The pricey bill of technical debt: When and by whom will it be paid?”, teoksessa *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2017, s. 13–23.

[2] W. Cunningham, ”The WyCash portfolio management system”, *ACM Sigplan Oops Messenger*, vol. 4, nro 2, s. 29–30, 1992.

[3] P. Kruchten, R. L. Nord ja I. Ozkaya, ”Technical debt: From metaphor to theory and practice”, *IEEE software*, vol. 29, nro 6, s. 18–21, 2012.

[4] D. Joyce, L. McPhee, R. Johnston, J. Corrin ja R. Hirsch, ”Toward a Conceptual Framework for Technical Debt in Archives”, *The American Archivist*, vol. 85, nro 1, s. 104–125, 2022.

[5] W. N. Behutiye, P. Rodríguez, M. Oivo ja A. Tosun, ”Analyzing the concept of technical debt in the context of agile software development: A systematic literature review”, *Information and Software Technology*, vol. 82, s. 139–158, 2017, ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2016.10.004`. url: `https://www.sciencedirect.com/science/article/pii/S0950584916302890`.

[6] P. Kruchten ja I. Ozkaya, *Managing technical debt: reducing friction in software development.* Addison-Wesley Professional, 2019.

[7] M. Robert C. ”A Mess is not a Technical Debt”. (2009), url: https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt.

[8] V. Lenarduzzi, J. Daly, A. Martini, S. Panichella ja D. A. Tamburri, ”Toward a Technical Debt Conceptualization for Serverless Computing”, *IEEE Software*, vol. 38, nro 1, s. 40–47, 2021. DOI: 10.1109/MS.2020.3030786.

[9] F. Zampetti, S. Geremia, G. Bavota ja M. Di Penta, ”Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study”, teoksessa *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, s. 471–482.

[10] Synopsys. ”2024 Open Source Security and Risk Analysis Report”. (2024), url: https://www.synopsys.com/software-integrity/engage/ossra/ossra-report (viitattu 11. 04. 2024).

[11] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles ja D. Izquierdo, ”Technical lag in software compilations: Measuring how outdated a software deployment is”, teoksessa *Open Source Systems: Towards Robust Practices: 13th IFIP WG 2.13 International Conference, OSS 2017, Buenos Aires, Argentina, May 22-23, 2017, Proceedings 13*, Springer International Publishing, 2017, s. 182–192.

[12] A. Decan, T. Mens ja E. Constantinou, ”On the evolution of technical lag in the npm package dependency network”, teoksessa *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, s. 404–414.

[13] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab ja N. Tsantalis, ”Dependency smells in javascript projects”, *IEEE Transactions on Software Engineering*, vol. 48, nro 10, s. 3790–3807, 2021.

[14] C. Okafor, T. R. Schorlemmer, S. Torres-Arias ja J. C. Davis, ”Sok: Analysis of software supply chain security by establishing secure design properties”, teoksessa *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, 2022, s. 15–24.

[15] Microsoft. ”Customer Guidance on Recent Nation-State Cyber Attacks”. (2020), url: `https://msrc.microsoft.com/blog/2020/12/customer-guidance-on-recent-nation-state-cyber-attacks/`.

[16] J. Yli-Huumo, A. Maglyas ja K. Smolander, ”How do software development teams manage technical debt? – An empirical study”, *Journal of Systems and Software*, vol. 120, s. 195–218, 2016, ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2016.05.018`. url: `https://www.sciencedirect.com/science/article/pii/S016412121630053X`.

[17] T. Besker, H. Ghanbari, A. Martini ja J. Bosch, ”The influence of Technical Debt on software developer morale”, *Journal of Systems and Software*, vol. 167, s. 110 586, 2020, ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2020.110586`. url: `https://www.sciencedirect.com/science/article/pii/S0164121220300674`.

[18] E. Jabrayilzade, M. Evtikhiev, E. Tüzün ja V. Kovalenko, ”Bus factor in practice”, teoksessa *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, s. 97–106.

[19] D. Rodríguez, M. Sicilia, E. García ja R. Harrison, ”Empirical findings on team size and productivity in software development”, *Journal of Systems and Software*, vol. 85, nro 3, s. 562–570, 2012, Novel approaches in the design and implementation of systems/software architecture, ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2011.09.009`. url: `https://www.sciencedirect.com/science/article/pii/S0164121211002366`.

[20] M. Srinivas, G. Ramakrishna, K. R. Rao ja E. S. Babu, ”Analysis of legacy system in software application development: A comparative survey”, *International Journal of Electrical and Computer Engineering*, vol. 6, nro 1, s. 292, 2016.

[21] ”Microservice Premium”. (2010), url: `https://www.martinfowler.com/bliki/MicroservicePremium.html/` (viitattu 06. 09. 2024).

[22] M. Feathers, ”Working effectively with legacy code”, *Object Mentor, Inc. Available online at http://www. objectmentor. com*, s. 9, 2002.

[23] R. Britto, ”Strategizing and evaluating the onboarding of software developers in large-scale globally distributed legacy projects”, tohtorinväitöskirja, Blekinge Tekniska Högskola, 2017.

[24] M. Kalske et al., ”Transforming monolithic architecture towards microservice architecture”, *University of Helsinki*, 2017.

[25] G. Blinowski, A. Ojdowska ja A. Przybyłek, ”Monolithic vs. microservice architecture: A performance and scalability evaluation”, *IEEE Access*, vol. 10, s. 20 357–20 374, 2022.

[26] W. P. Stevens, G. J. Myers ja L. L. Constantine, ”Structured design”, *IBM systems journal*, vol. 13, nro 2, s. 115–139, 1974.

[27] J. M. Bieman ja B.-K. Kang, ”Cohesion and reuse in an object-oriented system”, *ACM SIGSOFT Software Engineering Notes*, vol. 20, nro SI, s. 259–262, 1995.

[28] N. Dragoni, S. Giallorenzo, A. L. Lafuente et al., ”Microservices: yesterday, today, and tomorrow”, *Present and ulterior software engineering*, s. 195–216, 2017.

[29] W. Jin, T. Liu, Q. Zheng, D. Cui ja Y. Cai, ”Functionality-oriented microservice extraction based on execution trace clustering”, teoksessa *2018 IEEE International Conference on Web Services (ICWS)*, IEEE, 2018, s. 211–218.

[30] S. Habibullah, X. Liu, Z. Tan, Y. Zhang ja Q. Liu, ”Reviving legacy enterprise systems with microservice-based architecture within cloud environments”, 2019.

[31] ”What is FaaS?” (2021), url: `https://www.ibm.com/topics/faas` (viitattu 20. 09. 2024).

[32] S. Panichella, M. I. Rahman ja D. Taibi, ”Structural coupling for microservices”, *arXiv preprint arXiv:2103.04674*, 2021.

[33] ”What is a Product Owner?” (), url: `https://www.scrum.org/resources/what-is-a-product-owner` (viitattu 03. 05. 2024).

[34] ”Yhden ja kahden päivän sairauspoissaolot yleisiä”. (2023), url: `https://stat.fi/tietotrendit/artikkelit/2023/yhden-ja-kahden-paivan-sairauspoissaolot-yleisia/` (viitattu 23. 07. 2024).

[35] ”Vuosilomalaki”. (2005), url: `https://www.finlex.fi/fi/laki/ajantasa/2005/20050162` (viitattu 23. 07. 2024).

[36] ”Raskaus- ja vanhempainvapaa”. (2024), url: `https://tyosuojelu.fi/tyosuhde/muut-vapaat-tyosta/perhevapaat/raskaus-ja-vanhempainvapaa` (viitattu 23. 07. 2024).

[37] H. Laihonen, M. Hannula, N. Helander et al., *Tietojohtaminen*, 2013.

[38] D. Joseph, K.-Y. Ng, C. Koh ja S. Ang, ”Turnover of information technology professionals: A narrative review, meta-analytic structural equation modeling, and model development”, *MIS quarterly*, s. 547–577, 2007.

[39] P. C. B. Lee, ”Turnover of information technology professionals: a contextual model”, *Accounting, Management and Information Technologies*, vol. 10, nro 2, s. 101–124, 2000.

[40] J. Lo, ”The information technology workforce: A review and assessment of voluntary turnover research”, *Information Systems Frontiers*, vol. 17, s. 387–411, 2015.

[41] R. Korsakienė, A. Stankevičienė, A. Šimelytė ja M. Talačkienė, ”Factors driving turnover and retention of information technology professionals”, *Journal of business economics and management*, vol. 16, nro 1, s. 1–17, 2015.

[42] G. J. Myers, C. Sandler ja T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[43] L. Luo, ”Software testing techniques”, *Institute for software research international Carnegie mellon university Pittsburgh, PA*, vol. 15232, nro 1-19, s. 19, 2001.

[44] ”Difference between System Testing and End-to-end Testing”. (2023), url: `https://www.geeksforgeeks.org/difference-between-system-testing-and-end-to-end-testing/` (viitattu 23. 04. 2024).

[45] Microsoft. ”Unit vs Integration vs System vs E2E Testing”. (2022), url: `https://microsoft.github.io/code-with-engineering-playbook/automated-testing/e2e-testing/testing-comparison/` (viitattu 23. 04. 2024).

[46] ”end-to-end testing”. (2023), url: `https://www.techtarget.com/searchsoftwarequalit` `definition/End-to-end-testing` (viitattu 08. 10. 2024).

[47] ”What is E2E? A guide to end-to-end testing”. (2024), url: `https://circleci.com/blog/what-is-end-to-end-testing/` (viitattu 08. 10. 2024).

[48] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[49] S. Yoo ja M. Harman, ”Regression testing minimization, selection and prioritization: a survey”, *Software testing, verification and reliability*, vol. 22, nro 2, s. 67–120, 2012.

[50] ”Smoke Testing – Software Testing”. (2024), url: https://www.geeksforgeeks.org/smoke-testing-software-testing/ (viitattu 12. 08. 2024).

[51] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik ja A. De Lucia, ”Detecting code smells using machine learning techniques: Are we there yet?”, teoksessa *2018 IEEE 25th international conference on software analysis, evolution and reengineering (saner)*, IEEE, 2018, s. 612–621.

[52] ”Snake case”. (2024), url: https://developer.mozilla.org/en-US/docs/Glossary/Snake_case (viitattu 21. 08. 2024).

[53] ”Camel case”. (2024), url: https://developer.mozilla.org/en-US/docs/Glossary/Camel_case (viitattu 21. 08. 2024).

[54] ”linter developer’s guide”. (), url: https://www.sonarsource.com/learn/linter/ (viitattu 21. 08. 2024).

[55] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[56] T. Theunissen, U. van Heesch ja P. Avgeriou, ”A mapping study on documentation in Continuous Software Development”, *Information and software technology*, vol. 142, s. 106 733, 2022.

[57] ”Screenshots”. (), url: https://playwright.dev/docs/screenshots (viitattu 23. 09. 2024).

[58] ”Happy Path Testing”. (2018), url: https://www.h2kinfosys.com/blog/happy-path-testing/ (viitattu 27. 10. 2024).

[59] ”The Practical Test Pyramid”. (2018), url: https://martinfowler.com/articles/practical-test-pyramid.html (viitattu 14. 10. 2024).

[60] Microsoft. "E2E Testing". (2023), url: https://microsoft.github.io/code-with-engineering-playbook/automated-testing/e2e-testing/ (viitattu 23. 04. 2024).