Antero Hirvo

Testaamalla luottamus perintöjärjestelmästä

Regressiotestaustyökalu perintöjärjestelmän kehitykseen

Tekniikan ja innovaatiojohtamisen
akateeminen yksikkö
Kandidaatintutkielma
Automaatio ja tietotekniikka

Vaasa 2025

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen akateeminen yksikkö
Tekijä: Antero Hirvo
Tutkielman nimi: Testaamalla luottamus perintöjärjestelmästä: Regressiotestaustyökalu perintöjärjestelmän kehitykseen
Tutkinto: Tekniikan kandidaatti
Oppiaine: Automaatio ja tietotekniikka
Työn ohjaaja: Timo Mantere
Valmistumisvuosi: 2025 **Sivumäärä:** 50

TIIVISTELMÄ:
Tämän kandidaatintutkielman tavoitteena on kehittää testityökalu, joka varmistaa kohdeyrityksen kehitystyökalun koodin generoinnin toimivuuden. Testityökalu parantaa kehitystyökalun koodin generoinnin kehityksen varmuutta testaamalla regressiotestin tavoin tiedostojen muutoksia kehitystyökalun koodin muutosten välillä.

Tutkielman kirjallisuusosuudessa käsitellään ohjelmistotestausta, perintöjärjestelmiä ja perintöjärjestelmien kanssa työskentelyä.

Tutkielman empiirinen osuus käsittelee testityökalun teknistä toteutusta ja sen toimintaa. Testityökalu kehitettiin Python-ohjelmointikielellä hyödyntäen sen laajaa standardikirjastoa. Pythonin alustariippumaton ajo ja standardikirjaston ulkoisista paketeista riippumaton toteutus mahdollistaa testityökalun helpon käyttöönoton kuin integraation esimerkiksi testiautomaatioon.

Keskeisenä aiheena testityökalun kehityksessä on aiemman toiminnan pitäminen samana kehitystyökalun refaktorointien ja optimointien välillä.

Kehitetty testityökalu mahdollistaa kohdeyrityksen kehitystyökalun koodin generoinnin optimoinnin ja refaktoroinnin testipohjaisella varmuudella, joka ei ole ollut aiemmin mahdollista kehitystyökalun kohdalla. Testiautomaatioon integrointi ja raportoinnin tarkempi analyysi jää testityökalun jatkokehityksen kohteeksi.

AVAINSANAT: ohjelmistotestaus, perintöjärjestelmä, regressiotestaus, ohjelmistokehitys, Python

Sisällys

Kuviot

Taulukot

1 Johdanto

Tutkielman tavoitteena on kehittää testityökalu, joka testaa kohdeyrityksen kehitystyökalun generoiman koodin oikeellisuutta kehitystyökalun muutosten välillä. Generoitujen tiedostojen oikeellisuus varmistetaan regressiotestin tavoin vertailemalla niitä kehitystyökalun aiemman version generointeihin. Tuotetun testityökalun hyödyllisyyttä, toteutusta ja lopullista käyttöä arvioidaan kehitystyökalun käytön ja kehityksen näkökulmasta.

Kohdeyrityksen kehitystyökalu on monien perintöjärjestelmiä kuvaavien piirteiden mukainen. Työkalu on vanha, lähes dokumentoimaton ja ei sisällä testejä sen toiminnan validoimiseksi, mikä tekee sen ylläpidosta ja kehityksestä hankalaa. Testityökalun tarkoitus on toimia yhtenä testinä kattamalla kehitystyökalun koodin generoinnin ja varmistamaan sen toiminnan. Testityökalu antaa konkreettista varmuutta kehitystyökalun toiminnasta ja generoitujen tiedostojen oikeellisuudesta. Tämä mahdollistaa koodin generoinnin optimoinnin ja kehitystyökalun lähdekoodin refaktoroinnin testipohjaisella varmuudella, joka ei ole ollut mahdollista kehitystyökalun aiemmassa kehityksessä.

Tutkielman kirjallisuusosuudessa tutustutaan ohjelmistotestaukseen, perintöjärjestelmiin ja perintöjärjestelmien ylläpitoon. Keskeisenä aiheena kohdeyrityksen kehitystyökalun kehityksessä on perintöjärjestelmän lähdekoodin refaktorointi ja optimointi. Perintöjärjestelmien ylläpidossa keskitytään pitämään olemassa oleva toiminta samana muutosten ja uusien ominaisuuksien välillä. Ohjelmistotestauksen kannalta regressiotestaus eli uudelleentestaus on keskeisessä osassa testityökalun toiminnassa.

1.1 Tausta

Kohdeyrityksen kehitystyökalu on lähes kaksikymmentä vuotta vanha ja sen ylläpito on jatkunut tähän päivään saakka koska se on edelleen aktiivisena työkaluna tuotekehitystiimin käytössä. Standardi, johonka kehitystyökalu perustuu, on vuosien varrella muuttunut uusien ominaisuuksien ja versioiden myötä, joka on pakottanut myös kehitystyökalun pysymään kehityksen mukana. Uusien ominaisuuksien lisääminen ja virheiden korjaaminen osoittautuu erittäin riskialttiiksi, johtuen lähdekoodin vaihtelevasta ylläpidosta ja sen monimutkaisuudesta. Lisäksi kehitystyökalun tasolla koodi on täysin testaamatonta ja heikosti dokumentoitua.

Kehitystyökalu generoi C-kielistä koodia XML-kielisiin spesifikaatioihin perustuen. Generoidut tiedostot ovat spesifikaatioiden määrittelemiä laiteajurifunktioita, jotka toimivat yhtenä osana kohdeyrityksen tuotteessa. Koodin generoinnin lisäksi työkalulla pystyy tarkastelemaan, rakentamaan ja muokkaamaan uusia spesifikaatioita sen graafisen käyttöliittymän kautta. Kehitystyökalu tukee myös siihen erikseen kehitettyjen komentosarjojen komentorivipohjaista ajoa, joita kehitetty testityökalu hyödyntää.

Tarve testityökalulle syntyi kehitystyökalun optimoinnin kautta, jolloin sen tehokkuutta paranneltiin mutta huomaamatta myös sen haluttu toiminta muuttui alkuperäisestä ja tällöin generoidut tiedostot eivät vastanneet enää aiempaa. Modernia versiota kehitystyökalusta on kehitetty mutta se osoittautui haastavaksi kehitystyökaluun vuosien aikana kehitettyjen aiempien ominaisuuksien laajan määrän ja eri standardiversioiden tuen takia.

1.2 Tavoitteet ja tutkimuskysymykset

Tutkielmassa kehitetyn testityökalun tulee pystyä raportoimaan testiaineiston pohjalta generoitujen koodien välillä tapahtuneet muutokset kahden eri kehitystyökalun version välillä. Testityökalun poikkeamaraportin avulla saadaan varmuus kehitystyökalun

muutoksista ja voidaan olla varmoja, etteivät ne rikkoneet kehitystyökalun aiempaa toimintaa.

Tutkimuksen tavoitteita pohditaan seuraavien tutkimuskysymyksien avulla:

1. Miten testityökalu kehitetään vertailemaan kahden eri kehitystyökalun versioiden generoinnin muutoksia?
2. Miten testityökalun avulla varmistetaan, että kohdeyrityksen kehitystyökalun muutokset eivät riko aiempien versioiden toimintaa?
3. Mitkä ovat testityökalun käytön haasteet ja rajoitteet osana kohdeyrityksen kehitystyökalun kehitystä?

1.3 Tutkielman rakenne

Toinen ja kolmas luku kokoaa tutkielman teoreettisen viitekehyksen. Toinen luku käsittelee ohjelmistotestausta ja sen keskeisiä piirteitä. Luvussa tarkastellaan testaustasoja ja menetelmiä kuin myös ohjelmistotestauksen haasteita.

Kolmannessa luvussa käsitellään perintöjärjestelmien teoriaa. Luku jakautuu kolmeen alilukuun, jotka kattavat perintöjärjestelmien keskeiset piirteet, työskentelyn niiden kanssa ja lopuksi arvioi kohdeyrityksen kehitystyökalua ja sen piirteitä perintöjärjestelmänä.

Neljäs luku käsittelee testityökalun teknistä toteutusta ja sen toimintaa. Luku esittelee lyhyesti testityökalun käyttämän ohjelmointikielen ja sen hyödyntämät kirjastot. Luvussa käydään läpi testityökalun toimintaa ja esitellään sen ajoon vaadittavat parametrit kuin myös sen tuottama eroavaisuusraportti.

Viimeisessä luvussa käsitellään tutkielman tuloksia ja johtopäätöksiä. Luvussa arvioidaan testityökalun käyttöä kehitystyökalun kehityksessä, vastaamalla määriteltyihin

tutkimuskysymyksiin. Lisäksi luvussa pohditaan testityökalun käyttötarkoitusta ja annetaan jatkokehitysideoita testityökalun tulevaisuudelle.

2 Ohjelmistotestaus

Ohjelmistotestaus on olennainen ja tärkeä osa vakiintuneita ohjelmistotuotannon käytäntöjä. Kasurinen (2013, luku 1) kiteyttää ohjelmistotestauksen olevan työtä, jolla varmistetaan, että suunniteltu työ tehdään oikein. Ohjelmistotestaus on ohjelmistotuotantoa ohjaavaa työtä, jonka tavoitteena on varmistaa tuotteiden laatu ja toimivuus (Kasurinen, 2013, luku 1). Sen avulla varmistetaan, että tuote vastaa suunnitelmaa kuin myös tunnistetaan mahdolliset poikkeamat (Kasurinen, 2013, luku 1). Kasurinen (2013, luku 1) painottaa määritelmän olevan laaja ja sen toteutustapa vaihtelee eri testauskohteiden välillä sillä esimerkiksi auton osien tai graafisen käyttöliittymän testaus eroaa paljolti. Lisäksi testaajan työtehtävät vaihtelevat testattavan kohteen mukaan, ja ne voivat ulottua testien suunnittelusta ja kirjoittamisesta aina testiryhmän haastattelujen pitämiseen asti (Kasurinen, 2013, luku 1).

Kuitenkaan ohjelmistotestauksen termin ymmärrys ei ole yksikäsitteinen. Myers ja muut (2011, s. 5–6) väittää heikon ohjelmointitestauksen johtuvan sen termin ymmärtämisestä väärin päin. He esittävät määritelmiä ohjelmistotestauksesta, jotka kuvaavat ohjelmistotestauksen olevan tapa esittää ohjelmiston olevan virhevapaa ja varmistaa ohjelman toimivan suunnitellulla tavalla tai että ohjelmistotestaus on tapa saada varmuutta ohjelman toiminnasta. Toisaalta he toteavat näiden määritelmien kuvaavan ohjelmistotestauksen väärin päin. Heidän mukaansa ohjelmistotestaus on ohjelman arvoa kasvattavaa toimintaa, jolla tuotetaan laadukkaampia ja luotettavampia ohjelmia mikä tarkoittaa sen virheiden minimointia.

Mili (2015, luku 1) kertoo modernien ohjelmistotuotteiden ja -projektien olevan kalliita, monimutkaisia ja virhealttiita. Hän kuvastaa ohjelmistokehityksen alalta puuttuvan yhteisiä vaatimuksia, standardeja ja käytäntöjä millä projektit ja tuotteet voitaisiin kehittää, jonka takia ohjelmistokehitys on huomattavasti virhealttiimpi ala muihin insinöörialoihin verrattuna. Voidaan siis olettaa, että ei ole täysin virhevapaata ohjelmistokoodia ja siksi Myers ja muut (2011, s. 6) korostaa, että on parempi testata

ohjelmakoodia, sillä oletusarvolla, että siinä on virheitä ja testattaessa etsiä siitä mahdollisimman monta virhettä.

IEEE Computer Society:n julkaisema SWEBOK (Software Engineering Body of Knowledge) (IEEE Computer Society, 2024) opas kattaa ohjelmistotestauksen termistöä ja määritelmiä laajasti. SWEBOK määrittelee ohjelmistotestauksen seuraavasti:

> Software testing consists of the *dynamic* validation that a *system under test* (SUT) provides *expected* behaviors on a finite set of test cases suitably *selected* from the usually infinite execution domain. (IEEE Computer Society, 2024, luku 5, s. 1)

SWEBOK täydentää määritelmää avaamalla korostetut ohjelmistotestauksen avaintermit.

Dynaamisuus (engl. dynamic) tarkoittaa ohjelman testausta aina erilaisilla syötteillä (engl. input) (IEEE Computer Society, 2024, luku 5, s. 1). Valitut syötteet ei kuitenkaan yksinään varmista ohjelmien toimivuutta, sillä epädeterministiset (engl. nondeterministic) ja monimutkaiset ohjelmat voivat tuottaa samalla syötteellä eri ulostuloja (Cambridge University Press & Assessment, n.d.). Syötteiden valitseminen ja niillä testaaminen on osa staattista testaamista, joka on dynaamisuuden rinnalla tärkeä osa testausta (IEEE Computer Society, 2024, luku 5, s. 1).

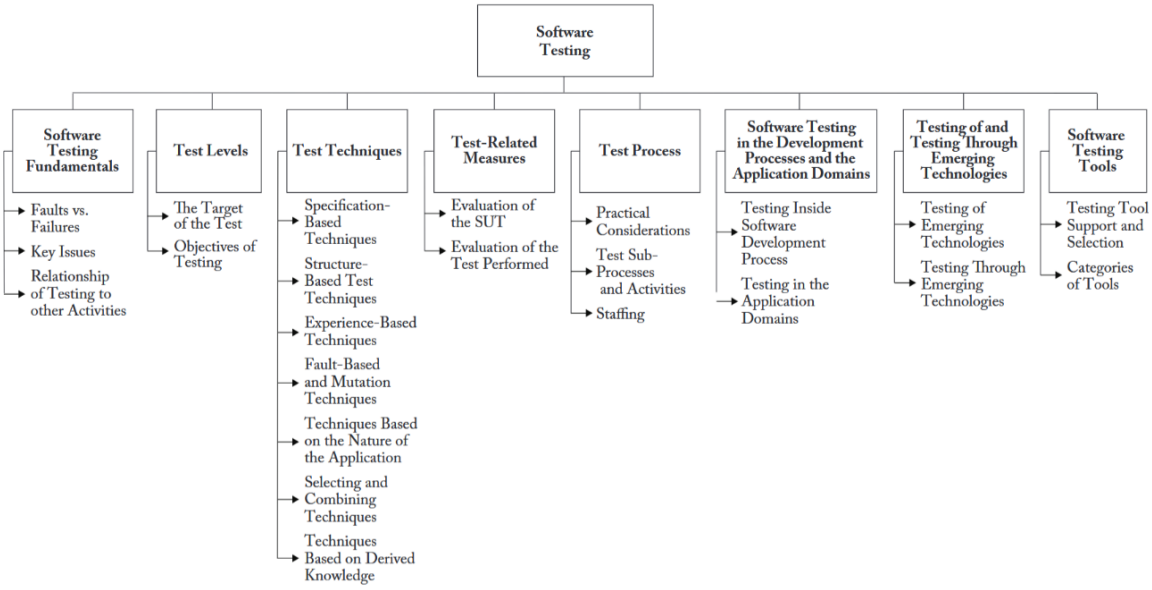
Äärellisyys (engl. finite) tarkoittaa realistisesti mahdollisten testattavien testitapauksien rajallisuutta. Täydellinen testaus (engl. exhaustive testing) tarkoittaa kaikkien mahdollisten syötteiden testausta varmistaakseen ohjelman toiminnan (Myers ja muut, 2011, s. 9) ja jo aivan pienissäkin ohjelmissa se osoittautuu lähes mahdottomaksi, puhumattakaan laajemmista testattavista ohjelmista. Käytännössä täydellinen testiasetelma on siten ääretön ja siksi testatessa kootaan tietty, tarkka, osajoukko äärettömistä mahdollisuuksista (IEEE Computer Society, 2024, luku 5, s. 2. Testattavaa osajoukkoa kootessa testaus osoittautuu riskienarvioinniksi ja kompromissien

tekemiseksi testien tärkeyksien, saatavilla olevien resurssien ja aikataulujen välillä (IEEE Computer Society, 2024, luku 5, s. 2).

Valikointi (engl. selected) tarkoittaa oikeiden testimenetelmien valitsemista testin toimeenpanoa varten. Testausmenetelmiä on paljon ja niillä jokaisella on omat hyvät ja huonot puolensa, jonka testaajan huomioi pohjautuen riskianalyysiin ja ammattitaitoon (IEEE Computer Society, 2024, luku 5, s. 2).

Oletettavuus (engl. expected) tarkoittaa, että testauksen tulokset tulee pystyä määrittelemään onnistuneeksi tai epäonnistuneeksi testin määritelmän mukaan (IEEE Computer Society, 2024, luku 5, s. 2). Ilman, että testauksen tulokset saadaan määriteltyä onnistuneeksi tai epäonnistuneeksi on testaus turhaa.

IEEE:n (IEEE Computer Society, 2024, luku 5, s. 2–3) mukaan ohjelmistotestaus on järjestäytynyt ja integroitunut enemmän ohjelmistokehityksen mukana suoritettavaksi toiminnaksi verrattuna vanhempaan näkökulmaan missä ohjelmistotestaus suoritettiin ohjelmistokehityksen jälkeen. IEEE esittää modernin ohjelmistotestauksen SWEBOK-mallin jakautuvan kahdeksaan päälukuun ja niiden alalukuihin (kuvio 1). Kuitenkaan SWEBOK-malli ei määrittele, että testaus kattaa muutakin kuin mekaanista testaamista ja tähän liittyen (Kasurinen, 2013, s. 48) painotta, että oikealla arkkitehtuurisuunnittelulla ja ammattimaisella kehityksellä saadaan saman verran virheitä pois koodista kuin testauksella.



Kuvio 1. Ohjelmistotestauksen SWEBOK-malli (IEEE Computer Society, 2024, luku 5, s. 3).

Kasurinen (2013, s. 48) esittää ISTQB-testaussertifikaatin (International Software Testing Qualifications Board) testauksen seitsemän perusperiaatetta:

1. Testaus osoittaa vikojen olemassaolon
2. Täydellinen testaus on mahdotonta
3. Testaus tulee aloittaa mahdollisimman aikaisin
4. Viat kasaantuvat tiettyihin osiin tai moduuleihin
5. Testejä tulee ylläpitää
6. Testaus riippuu tilanteesta
7. Testaus ei korvaa hyvää suunnittelua

ISTQB:n perusperiaatteet painottavat hyvää arkkitehtuurin suunnittelua ja testausta osana tuotetta ja sen kehitystä. Lisäksi periaatteet painottavat testien jatkuvaa ylläpitoa ja kriittisyyttä testeihin ja niistä saatavaan varmuuteen. Kuudennen perusperiaatteen mukaisesti on huomioitavaa, että testaus riippuu tilanteesta ja testattavasta tuotteesta. Tutkielman testityökalu keskittyy kehitystyökalun koodin generoinnin osuuteen, joka pyrkii esittämään vikojen olemassaolon ohjelman kyseisessä osiossa kuin myös vähentämään kehityksen aikana huomaamattomasti syntyneiden vikojen kasaantumista.

Seuraavissa aliluvuissa käsitellään ohjelmistotestaukselle keskeisiä perusteita ja menetelmiä. Lisäksi luvussa 2.1 käsitellään virheen, vian ja häiriön määritelmät, jota ovat tärkeä ymmärtää ohjelmistotestauksesta puhuttaessa.

2.1 Virhe, vika ja häiriö

Ohjelmistojen toimintahäiriöille on monia nimityksiä mutta yleisimmät nimitykset vika, virhe ja bugi koetaan arkikielessä synonyymeinä (Kasurinen, 2013, s. 50). Arkikielestä poiketen kuitenkin ohjelmistokehityksessä virhe, vika ja häiriö ovat kolme määriteltävää käsitettä vaikkakin kirjallisuuden mukaan voi tarkat nimitykset vaihdella.

Kasurinen (2013, s. 50) mukaan erehdys tai virhe (engl. fault tai mistake) on ihmisen tuottama poikkeama, jonka takia syntyy vika. Kaner ja muut (2002) korostavat, että vika voi johtua perinpohjaisesta toteutuksesta tai ohjelman suunnittelusta. Vika, tai arkikielisemmin bugi (engl. error), on virheestä johtunut tila missä ohjelma ei pääse toimimaan sille suunnitellulla tavalla, joka aiheuttaa häiriön (engl. failure, crash) (Kasurinen, 2013, s. 50). Häiriöön johtanut tila on tärkeä osata tunnistaa sillä itse häiriö ei ole varsinainen korjattava asia, vaan häiriöön johtanut kriittinen tila ja sitä käsittelevä logiikka (Kaner ja muut, 2002).

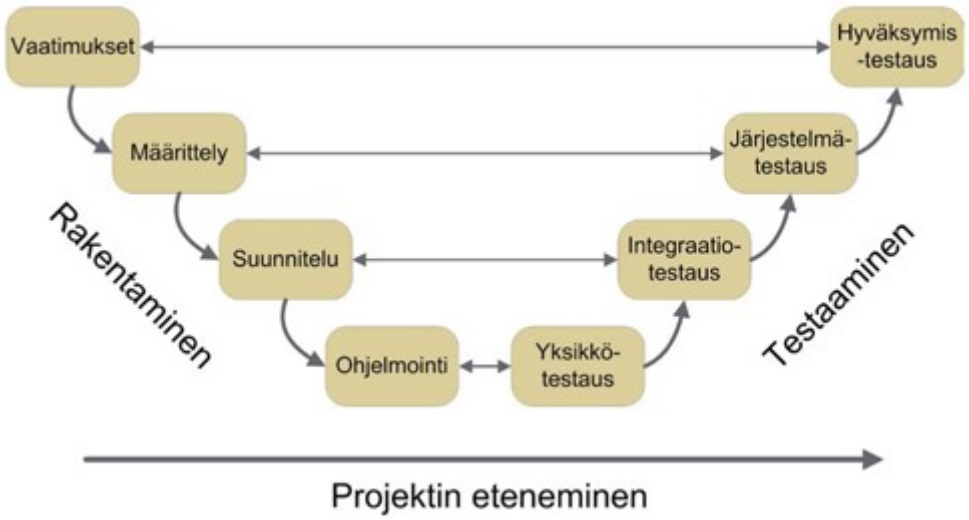
Kaner ja muut (2002) korostavat, että yleisesti käytetty termi bugi (engl. bug) on kaiken kattava termi, jolla voidaan kuvata kaikkia näitä kolmea käsitettä yhtenäisesti. Bugiraportti ohjelmasta voi kuvastaa virhettä, vikaa, häiriötä tai toiminnallista puutetta ilman tarkempaa määritelmää, joka voi osoittautua hankalaksi, kun kaikki virhetyypit käsitellään saman kattotermin alla (Kaner ja muut, 2002).

Ohjelmistoissa voi olla vikoja, jotka eivät aina esittäydy suoraan virheinä, joita kutsutaan oireiksi (engl. symptom) (Kaner ja muut, 2002). Oire voi olla esimerkiksi ohjelman hitaus, joka itsessään enteilee suurempaa ongelmaa (Kaner ja muut, 2002).

Ohjelmistotestauksen tärkeänä päämääränä on löytää nämä virheet ja niiden syyt, joista lopulliset viat johtuvat (IEEE Computer Society, 2024, luku 5, s. 3–4).

2.2 Testaustasot

Ohjelmistotestaus on etenevä prosessi, jota tehdään koko projektin tai tuotteen elinkaaren aikana (Mili, 2015, s. 23). Testaustyötä projektin elinkaaren aikana kuvataan porrastetulla V-mallilla (kuvio 2), josta nähdään projektin vaiheen mukaan vastaavan tason testausvaihe (Mili, 2015, s. 32–33). Mallin vasemmalla reunalla on tuotteen tai projektin testattava kohde (engl. target) ja oikealla puolella nuolella yhdistettynä on tähän vastaava testauksen tavoite (engl. objective) (Mili, 2015, s. 32). V-mallin kolme olennaista tasoa ovat yksikkötestaus (engl. unit testing), integraatiotestaus (engl. integration testing) ja järjestelmätestaus (engl. system testing), jotka ovat suunnattu kehittäjille (Mili, 2015, s. 32). Neljäs ja ylin taso, hyväksyttämistestaus (engl. acceptance testing) poikkeaa edeltävästä kolmesta tasosta koska se on yleisesti toimitettu loppukäyttäjiä varten ja usein myös tehty loppukäyttäjien toimesta varmistaakseen tuotteen toimivan vaatimusten mukaan (Mili, 2015, s. 32).



Kuvio 2. Testauksen V-malli (Kasurinen, 2013, s. 51).

2.2.1 Yksikkötestaus

Yksikkötestauksen tarkoitus on varmistaa yhden ohjelmiston osion kuten yhden funktion tai laajemman aliohjelman toimivuus, joka voidaan rajata omaksi osioksi ilman riippuvuuksia muihin osiin (IEEE Computer Society, 2024, luku 5, s. 6). Yksikkötestit ovat pieniä ja erittäin nopeasti suoritettavia testejä, joiden tarkoitus on välittömästi antaa palautetta testattavan moduulin toiminnasta (Kasurinen, 2013, s. 51–52). Yksikkötestein voidaan testata moduulilta muun muassa sen toimintaa oikeilla ja väärillä syötteillä, muistinhallintaa tai oliopohjaisia näkyvyysrajoja (Kasurinen, 2013, s. 53). Yleisesti yksikkötesteillä on suora pääsy testattavan ohjelman lähdekoodiin, joka ei välttämättä päde muille testaustasoille (IEEE Computer Society, 2024, luku 5, s. 15).

Yksikkötestaus suoritetaan osana ohjelmointia ja yleisesti ohjelmoija itse kirjoittaa ja ajaa yksikkötestejä (IEEE Computer Society, 2024, luku 5, s. 6). Yksikkötestien epäonnistuessa ohjelmoija tietää välittömästi tehneen virheen ja pystyy reagoimaan virheeseen välittömästi (Kasurinen, 2013, s. 53–54). Yksikkötestien kirjoitus ja ajo osana ohjelmointia on perusta testipohjaiselle kehitykselle (engl. Test Driven Development, TDD), joka on tehokas yleisesti käytetty ohjelmistotuotannon menetelmä (Beck, 2002) .

Koskelan mukaan (2013, s. 9) yksikkötestit ovat tärkeässä osassa ohjelmistokehityksessä sillä ne ovat yleisimmin suoritettuja testejä, joilla on suora vaikutus testauksen rahalliseen arvoon. Koskela nostaa esille esimerkissään Googlen tutkimuksen, joka havainnollisti saman virheen korjaamisen hintaa eri testaustasoilla. Tutkimuksen mukaan virheen korjaaminen yksikkötestein maksaa noin viisi dollaria, kun taas korkeammilla tasoilla saman virheen korjaaminen maksaa eksponentiaalisesti enemmän.

2.2.2 Integraatiotestaus

Yksikkötestauksen jälkeen testatut moduulit kootaan yhteen suunnitelluksi järjestelmäksi, jota varten suoritetaan integraatiotestaus testatakseen koko järjestelmän

toimivuutta yhdessä (Kasurinen, 2013, Luku 3). Integraatiotestauksen testausvaihe vastaa ohjelmistokehityksen V-mallissa projektin suunnittelua.

Kasurisen (2013, s. 54) mukaan integraatiotestauksen tärkein tavoite on varmistaa yhdistettyjen järjestelmien osien toimivuus ja yhteensopivuus. Kuitenkaan integraatiotestaus ei vielä testaa koko järjestelmää vaan voi testata esimerkiksi kahden moduulin välistä kommunikointia tai samaa tietokantaa käyttävien moduulien toimintaa (Kasurinen, 2013, s. 54). Samoin kuin yksikkötestaus, on integraatiotestaus jatkuva työtehtävä, joka elää projektin edetessä mutta verrattuna yksikkötestaukseen, integraatiotestauksessa abstrahoidaan alempien tasojen logiikkaa ja keskitytään integroitavien moduulien tasoon (IEEE Computer Society, 2024, luku 5, s. 6–7).

Integraatiotestauksen avulla yksikkötestatut moduulit tuodaan yhteen testaten niiden yhteensopivuutta eri strategioilla kuten alhaalta ylöspäin (engl. bottom up testing) tai ylhäältä alaspäin (engl. top down testing), joissa moduulit integroidaan järjestelmään määritetyissä järjestyksissä (Kasurinen, 2013, s. 55). Pienemmissä sovelluksissa kaikki moduulit voidaan kerralla integroida yhteen ja testata niiden yhteensopivuutta niin sanottuna savutustestinä, jolla arvioidaan nopeasti järjestelmän kokonaisvaltaista toimivuutta (IEEE Computer Society, 2024, luku 4, s. 9). Savutustestin kertaluonteista ja koko järjestelmän kattavaa testausta kutsutaan myös kertarysäystestaukseksi (engl. big bang testing) (IEEE Computer Society, 2024, luku 4, s. 9).

2.2.3 Järjestelmätestaus

Järjestelmätestaus on kokonaisen järjestelmän testausvaihe, kun yksittäiset moduulit on testattu ja integroitu kokonaiseksi järjestelmäksi (Kasurinen, 2013, s. 56–57). Integraatiotestaus on laajempi testauksen käsite ja se kattaa kaiken testauksen, mikä tehdään kokonaiselle tuotteelle missä ei ole käytössä aiempien testitapojen yhteydessä käytettyjä mahdollisia tynkä- tai valeobjekteja (engl. stub ja mock object) (Kasurinen, 2013, s. 56). Yleisesti yksikkö- ja integraatiotestaus löytää suuren osan järjestelmän

mahdollisista virheistä ja siksi järjestelmätestauksella keskitytään määrittelyn ei-toiminnallisiin vaatimuksiin kuten turvallisuuteen, nopeuteen, käytettävyyteen ja luotettavuuteen (IEEE Computer Society, 2024, luku 5, s. 7). Järjestelmätestaukseen kuuluu myös muiden järjestelmien välisen yhteyden, laitteistojen tai käyttöjärjestelmien välinen testaus (IEEE Computer Society, 2024, luku 5, s. 7).

Järjestelmätestaus on yleisesti vielä kehityksen aikana tehtävää musta- tai lasilaatikko testausta, jolla varmistetaan järjestelmän toimivan määrittelyn mukaisesti välittämättä sen sisäisestä toiminnasta (Kasurinen, 2013, s. 57). Ennen lopullista hyväksyttämistestausta, järjestelmätestauksella pyritään löytämään mahdollisimman paljon vielä jäljellä olevia virheitä (Mili, 2015, s. 32). Hyväksyttämistestaukseen siirryttäessä ei järjestelmään oleteta enää tehtävän kovinkaan paljon muutoksia ja siksi on tärkeää, että integraatiotestaus tehdään mahdollisimman laajasti ja lopullista käyttöympäristöä mukaillen (Kasurinen, 2013, s. 57). Riippuen alasta ja projektista, integraatiotestaus voi alkaa hyvin aikaisessa vaiheessa esimerkiksi tuotekehityksen keskittyessä prototyyppeihin ja integraatiotestaus voi erota V-mallista siten, että se ei aina ole projektin loppuvaiheessa tehty työvaihe (Kasurinen, 2013, s. 57).

2.2.4 Hyväksyttämistestaus

Hyväksyttämistestaus on V-mallin viimeinen testausvaihe, jossa yleensä asiakas tai tilaaja testaa lopullisen tuotteen vastaavan haluttuja vaatimuksia (IEEE Computer Society, 2024, luku 5, s. 7). Hyväksyttämistestauksessa keskitytään näyttämään vaadittuja ominaisuuksia ja niiden vaadittua toimintaa (Kasurinen, 2013, s. 57). Virheiden vähyys ei tässä vaiheessa ole enää olennainen osa testausta eikä hyväksyttämistestaukseen siirryttäessä enää oleteta tekevän muutoksia tehtyyn tuotteeseen (Kasurinen, 2013, s. 57). Hyväksyttämistestauksen jälkeen voidaan todeta projekti valmiiksi, jonka jälkeen tehdään loppuraportointi ja arkistointi ylläpitoa ja jatkokehitystä varten (Kasurinen, 2013, s. 57).

2.3 Testausmenetelmät

Kasurisen (2013, Luku 4) mukaan ohjelmistotestaus ja sen menetelmät jakautuvat projektin kehitysvaiheessa esituotannon, kehitysvaiheen ja julkaistavan tuotteen testaukseen. Jokaisessa testausvaiheessa on omanlainen lähestymistapa testattavaan tuotteeseen kuin myös omat työkalut ja tavat niiden toteuttamiseen.

Esituotanto ohjelmistoprojektissa on suunnittelun ja hyvin aikaisen testauksen vaihe, jossa muun muassa testataan projektin konseptiversioita ja kootaan haluttuja vaatimuksia (Kasurinen, 2013, s. 62). Esituotantovaiheessa testaustyö ei keskity varsinaisesti testitapauksiin vaan enemmänkin suunnitteluun ja konseptiversioiden kokeiluun, jota kutsutaan esitestaukseksi (Kasurinen, 2013, s. 63). Esitestauksen tarkoitus on säästää rahaa löytämällä virheitä ja huonoja arkkitehtonisia ongelmia mahdollisimman aikaisin sillä myöhemmin samojen virheiden korjaus on huomattavasti kalliimpaa ja vaikeampaa (Kasurinen, 2013, s. 63). Esitestauksen lisäksi esituotannossa suunnitellaan tuotantovaiheessa tehtävä testaus, jossa ohjelmistoprojektin vaatimukset määrittelevät halutut vaatimukset, joiden perusteella testaus suunnitellaan kahdesta näkökulmasta. Suunnitellut testitapaukset kattavat kaksi näkökulmaa missä varmistetaan, että projekti on tehty oikein ja se toimii halutulla tavalla.

Kehitysvaiheen testauksen lähestymistavan keskisenä osana on testauksen V-malli, jossa jokaisella tasolla keskitytään omanlaiseen testaukseen siihen sopivilla menetelmillä (Kasurinen, 2013, s. 64–65). Kehitysvaiheen testaus jakautuu staattiseen ja dynaamiseen testaamiseen, jotka kuvastavat millaista testausta suoritetaan (Kasurinen, 2013, s. 65). Staattinen testaus on nimensä mukaisesti järjestelmän testausta ilman sen ajamista, jossa tehdään perustason testausta muun muassa koodillisten syntaksivirheiden löytämiseksi ennen dynaamisen testauksen aloittamista (Kasurinen, 2013, s. 65). Staattiseen testaukseen löytyy paljon työkaluja kuten SonarQube, Checkstyle tai Codacy jotka ovat osa integroitu suoraan ohjelmointiympäristöön. Staattisen testauksen vastakohta on dynaaminen testaus, jossa testattava systeemi ajetaan virheiden löytämiseksi (Kasurinen, 2013, s. 65).

Ennen julkaisua suoritettaviin testausmenetelmiin kuuluu useita eri työvaiheita, joita ei välttämättä pystytä tai haluta suorittaa ennen projektin kehityksen valmistuttua (Kasurinen, 2013, s. 70). Testausmenetelmiin kuuluu muun muassa käytettävyystestaus, jonka tarkoitus on varmistaa tuotteen helppokäyttöisyys, jota voidaan testata esimerkiksi käyttökokeiluilla ja haastatteluilla. Käytettävyyden lisäksi tärkeä testauksen osa ennen julkaisua on systeemin kuormitus- ja suorituskykytestaus, joilla varmistetaan tuotteen toimivuus oikeassa käyttöympäristössä (Kasurinen, 2013, Luku 4). Seuraavissa aliluvuissa on tutustuttu tarkemmin työn kannalta keskeisiin testausmenetelmiin.

2.3.1 Musta-, lasi- ja harmaa laatikko -testaus

Musta laatikko testaus (engl. black box testing, input/output driven testing) on perinteisin tapa testata ohjelman toimintaa seuraamalla ohjelman ulostuloa tietämättä ohjelman sisäisestä toteutuksesta mitään (Kasurinen, 2013, s. 65–66). Testitavan yksinkertaisuuden vuoksi keskittyy ohjelmistotestaaja suunnittelemaan testien sisään- ja ulostulot vastaamaan ohjelmiston suoria- ja epäsuoria vaatimuksia (Kaner ja muut, 2002). Musta laatikko testaus on testaustapa, jota hyödynnetään jokaisella testauksen tasolla ja sillä voidaan testata aina yksittäisiä yksikköjä kuin kokonaisia järjestelmiä (Kasurinen, 2013, s. 66). Tärkeä osa musta laatikko testausta on, että ohjelmalle annetut syötteet kattavat tarkasti ja mahdollisimman monimuotoisesti ohjelman toimintaa, joka itsessään osoittautuu omaksi haasteeksi, että testattavaksi valitaan tarvittavat oikeat syötekombinaatiot (Kasurinen, 2013, s. 66). Laajojen syötekombinaatioiden, yksinkertaisten tulosten ja raa'an toiston vähentämiseksi Kasurinen (2013, s. 66) painottaa automaattisen testauksen tärkeyttä.

Musta laatikko nimitys kuvastaa testattavaa kohdetta mustana laatikkona, jonka toteutusta ja sisäistä toimintaa ei voi nähdä ja siksi ohjelman testaus sisään- ja ulostulojen kautta osoittautuu tietyissä tapauksissa kömpelöksi ja mahdottoman laajaksi (Myers ja muut, 2011, s. 9). Lasilaatikkotestaus (engl. white box testing) on musta

laatikko testauksen vastakohta missä sisään- ja ulostulojen lisäksi tarkastellaan ohjelman sisäistä toimintaa (Kasurinen, 2013, s. 67). Lasilaatikkotestauksella voidaan tarkastella syötteiden käyttäviä ohjelmapolkuja aina lähdekoodiin asti, joka toisaalta taas tekee testaajan työstä huomattavasti teknisempää ja vaatii testaajalta ohjelmoinnin ja testattavan projektin tarkkaa tietämystä (Kasurinen, 2013, s. 67–68). Lasilaatikkotestaus ei yksinään toimi sillä se ei erota projektin vaatimusten puutteita tai puuttuvia ominaisuuksia sen logiikkakeskeisen lähestymistavan takia (Kasurinen, 2013, s. 68).

Harmaa laatikko testaus (engl. gray box testing) yhdistää musta- ja lastilaatikkotestauksen hyvät puolet testaamalla samanaikaisesti vaatimuksia ja koodin logiikkaa (Kasurinen, 2013, s. 68). Harmaa laatikko testaus sopii järjestelmiin missä koko toimintaa ei voida testata lasilaatikkotestauksella kuten esimerkiksi verkkopalveluita, jotka hyödyntävät ulkoisia palvelinratkaisuita (Kasurinen, 2013, s. 68). Kehitystyökalun koodin generoinnin testaus tapahtuu musta laatikko -tyylillä, jossa testityökalu tarkastelee kehitystyökalulle annettavia syötteitä ja ulostulotiedostoja, mutta ei analysoi sen sisäistä toimintaa.

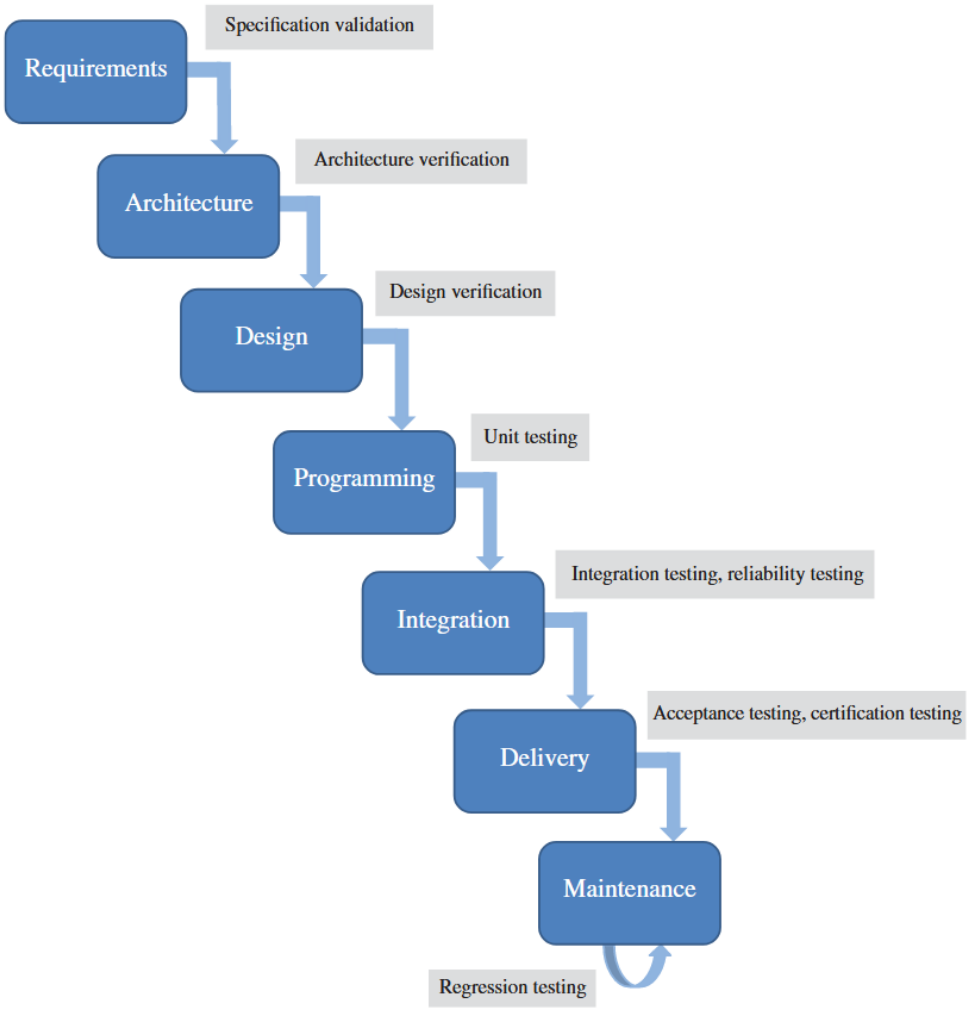
2.3.2 Regressiotestaus

Regressiotestaus on testauksen yleistermi, joka kuvastaa testauksen uudelleen ajoa varmistaakseen järjestelmän edelleen toimivan muutosten jälkeen (Kasurinen, 2013, s. 68). Regressiotestaus on tärkeä työtehtävä varmistamaan ohjelman toimivuuden uusien versioiden ja aktiivisen kehityksen keskellä esimerkiksi kehityshaarojen yhdistämisen jälkeen (Kasurinen, 2013, s. 69). Regressiotestaus ei ole yksinään mihinkään testaustasoon liittyvä ja sitä voidaan suorittaa jo aina yksikkötasolta lähtein, joka tekee siitä otollisen hyödyntämään automaatiota (Kasurinen, 2013, s. 70).

Regressiotestaus on tärkeänä osana ohjelmistoprojektin elinkaaren (kuvio 3) loppupäässä missä regressiotestauksella saatu varmuus on keskeinen työkalu julkaistujen ohjelmiston huollossa (Mili, 2015, s. 25). Huoltotyöt muuttavat ohjelman

toimintoja, jotka ovat jo kehitysvaiheessa testattu ja huoltotöiden päätyttyä kyseiset testit ajetaan uudestaan varmistaakseen ohjelman toimivan vielä vaatimusten mukaisesti ja ettei muutokset ole luoneet tahattomia eroavaisuuksia toiminnassa (Mili, 2015, s. 25). Suurissa ohjelmistoprojekteissa testejä on paljon ja niiden ajo vie aikaa ja yleistä onkin, että regressiotestauksessa ajetaan vain osa testeistä, jotka testaavat vain ohjelman muutettua osiota huomioiden kuitenkin, ettei liikaa testejä jätetä ajamatta ja ettei ohjelman toiminnan varmuus kärsi (IEEE Computer Society, 2024, luku 5, s. 8).

Aktiivisessa kehityksessä regressiotestaus on tärkeänä osana moderneissa ohjelmistotuotannon kehityskäytännöissä kuten DevOps ja ketterissä (Agile) menetelmissä ja testipohjaisessa kehityksessä (TDD) (IEEE Computer Society, 2024, luku 5, s. 8). Regressiotestaus suoritetaan yleisesti integraatiotestauksen jälkeen ennen tuotantoon käyttöönottoa testiautomaatiota hyödyntäen (IEEE Computer Society, 2024, luku 5, s. 24). Regressiotestaus on keskeinen osa testityökalun toimintaa, sillä sen tarjoama testipohjainen varmuus perustuu generoinnin uudelleentestaamiseen vertaamalla uuden version toimintaa aiempaan. Näin varmistetaan, että toiminnallisuus säilyy ennallaan.



Kuvio 3. Ohjelmistoprojektin elinkaari (Mili, 2015, s. 26).

2.4 Testauksen haasteet

Aikaisempi teoria on perustunut paljolti ohjelmistotestauksen teoriaan ja hyviin käytänteisiin, joita parhaan mukaan pyritään käyttämään ohjelmistokehityksessä. Todellisuudessa käytänteiden toimeenpano ja laadukas ohjelmistotestaus on vaikeaa, kallista ja aikaa vievää.

Kumar (2019) painottaa ohjelmistojen huonon laadun olevan suuri ongelma ohjelmistoalalla ja ohjelmistojen kasvaessa niiden testaus osoittautuu vaikeammaksi ja

jatkuvasti kalliimmaksi. Hän painottaa ohjelmistotestauksen keskeisten haasteiden olevan testauskokonaisuuksien väärinymmärrys ja laajuus, vaikeus päättää koska ohjelma on tarpeeksi testattu julkaisua varten ja kuinka testaustyö tehdään aina aikapaineen alla.

Koskela (2013, luku 2) korostaa testien laadun merkitystä erityisesti testikoodin luettavuuden näkökulmasta. Hän painottaa, että testien laatu, luotettavuus ja paikkansapitävyys vaikuttavat suoraan sekä ohjelmiston kokonaislaatuun että kehittäjien työskentelyn tehokkuuteen.

Garousi ja muut (2020) nostaa esille tutkimuksessaan keskeisiksi vaikeuksiksi ohjelmistotestauksessa osaamisen puutteen, vähäisen käytettävän ajan ja resurssien puutteen ja testiautomaation käytön. Garousi ja muut (2020) huomauttavat myös organisaatioiden vaikeuksista testata vanhoja perintöjärjestelmiä (engl. legacy system) vanhempien teknologioiden takia. Garousi ja muiden (2020) mukaan perintöjärjestelmien testaus jätetään vähäiselle huomiolle tutkimuksessa ja koulutuksessa tulevien uusien teknologioiden (engl. greenfield) toimesta sen sijaan, että keskityttäisiin olemassa oleviin teknologioihin (engl. brownfield), joka tuo omat ongelmat uusien ja entisten järjestelmien yhteiselossa.

3 Perintöjärjestelmät

Ohjelmistokehitys ja sen ympärillä pyörivä liiketoiminta elää syklistä kiertoa kehityksen, ylläpidon ja loppukäytön välillä (Annett, 2019, s. 3). Liiketoimintaan liittyvät ratkaisut ja tuotteet kehitetään ohjelmistokehityksen käytäntöjen mukaisesti, sisältäen vaatimusmäärittelyn, suunnittelun, kehityksen ja testauksen vaiheet, jonka jälkeen tuote julkaistaan käyttöön ja siirtyy ylläpitovaiheeseen (Annett, 2019, s. 3). Ohjelmistokehitys suhteutettuna kehitetyn ohjelman elinkaareen on lyhyt ja lopulta ohjelman pitkäaikainen ylläpito osoittautuu olevan ohjelman kallein ja liiketoiminnan kannalta vaikein päätös (Annett, 2019, s. 3). Ylläpitovaiheessa ohjelman kehitykseen ei käytetä rahaa, eikä työtunteja, joka on toisaalta vaihe, jolloin ohjelmalla tehdään rahaa mutta toisaalta se on myös ohjelmalle kriittinen vaihe missä teknologiat kehittyvät mutta ohjelma ei (Annett, 2019, s. 3).

Ohjelmistoprojektit kehitetään ajan parhailla teknologioilla ja kehittäjien parhaalla tavalla ja osaamisella mutta luonnollisesti ajan edetessä teknologiat, yhteiskunta, lait, vaatimukset kuin organisaatiot muuttuvat, jonka perässä ohjelmistot luonnollisesti seuraavat (Rosenkranz ja muut, 2024, s. 8). Systeemiä, joka jää tämän kehityksen jalkoihin kutsutaan perintösysteemiksi (engl. legacy system), joka on arkikielen verrattuna päinvastainen määritelmä saadusta perinnöstä (Annett, 2019, s. 1). Annett (2019, s. 1) määrittelee perintösysteemin olevan menneen ajan teknologioilla kehitettyjä, nykypäivään verrattuna vanhentuneita systeemejä, jotka nykypäivänä kehitettäisiin moderneimmilla teknologioilla. Rosenkranz ja muut (2024) esittää vaihtoehtoisissa määritelmissä perintösysteemien olevan suuria, vanhoja systeemejä, jotka ovat vaikeita ylläpitää mutta kuitenkin ovat kyseiselle liiketoiminnalle kriittisiä.

Perintösysteemeistä ja perintökoodista (engl. legacy code) puhuttaessa, termejä käytetään läheisesti mutta ovat ne silti tärkeä osata erottaa toisistaan. Edellä mainitun pohjalta perintösysteemi tarkoittaa kokonaisuudessaan vanhaksi jäänyttä mutta silti aktiivisesti käytössä olevaa ohjelmistoa tai systeemiä. Perintökoodi taas vastaa vanhaa ja vanhentunutta lähdekoodia, joka on yleisesti vaikeaa muokata ja ymmärtää (Boccara,

2019, s. 6). Tutkimuksen testattava kehitystyökalu on perintösysteemi mutta koodi, jota halutaan testata ja jonka kanssa työskennellään, on koodia, joka täyttää perintökoodin piirteet.

Boccaran (2019) määritelmästä poiketen, Feathers (2015) määrittelee perintökoodin yksinkertaisesti olevan koodia ilman testejä. Hänen näkökulmansa perustuu testeillä saatavaan varmuuteen, jolla koodi voidaan pitää ylläpidettävänä ja muutettavana. Toisaalta Boccaran (2019, s. 6) mukaan määritelmät ovat ristiriidassa, sillä vaikka koodissa olisi testejä, ei se välttämättä takaa, että koodia on helppo muokata. Feathersin näkökulma nojaa enemmän testipohjaiseen varmuuteen, jonka hän on kokenut toimivan perintösysteemien kanssa, kun taas Boccaran määritelmä painottaa hyviä ohjelmointikäytänteitä.

3.1 Perintöjärjestelmien piirteet

Perintöjärjestelmien määritelmien lisäksi on tärkeä osata tunnistaa niiden keskeisiä ongelmia ja piirteitä, joista monet perintösysteemit koostuvat. Useat lähteet yhtyvät samaan perintöjärjestelmän piirteeseen niiden dokumentaation ja ylläpidon kuin kehittämisen osaamisen puutteeseen (ks. Annett, 2019; Boccara, 2019; Rosenkranz ja muut, 2024). Rosenkranz ja muut (2024, s. 11) painottavat perintösysteemien yleisen dokumentaation puuttuvan ja että olemassa olevan dokumentaation harvoin vastaa systeemiä, jota on vuosien varrella ylläpidetty vaihtelevalla osaamisella. Heidän mukaansa puuttuva ja vaillinainen dokumentaatio johtaa kehittäjien hätäratkaisuihin ja opittuihin tapoihin, joita harvoin dokumentoidaan. He perustelevat, että puuttuvan dokumentaation vuoksi ohjelman lähdekoodi toimii ainoana ohjeena sen toiminnasta. Tämä on ongelmallista, sillä perintökoodi on usein vaikeasti luettavaa ja huonosti ylläpidettävää.

Toinen yleinen piirre perintösysteemeille on niiden suuri koko ja laaja kirjo toiminnallisuuksia, jotka ovat kertyneet ohjelmaan pitkän ylläpidon aikana

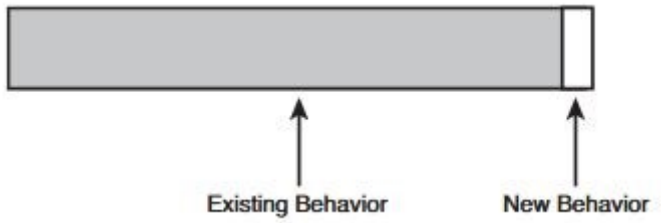
muuttuneiden vaatimusten kuin virheidenkorjauksen myötä (Rosenkranz ja muut, 2024, s. 11). Lisäksi pitkäikäisissä projekteissa liiketoiminnan kasvu lisää myös ohjelmaan kertyvän datan ja ylläpidettävien vaatimusten määrää, joita harvoin poistetaan jatkuvan liiketoimintatarpeen vuoksi (Rosenkranz ja muut, 2024, s. 11). Feathers (2005, s. 7–8) painottaa suurien ohjelmien muuntamisen ja entisen toiminnan ylläpidon vaikeutta, joka toisaalta johtaa ylläpidon jättämistä vähäiselle koska kehittäjät eivät koe tarpeeksi suurta varmuutta tekemään muutoksia. Tähän samaistuu ohjelmistokehityksen yleisesti esille nostettu lausahdus, että on turhaa lähteä muuntamaan ohjelmaa, jos se toimii (Kanat-Alexander, 2012, s. 34).

Lähdekoodin ylläpidon laiminlyönti, johtaa pidemmällä aikavälillä rikkonaiseen ja kerroksittaiseen arkkitehtuuriin, joka tekee lähdekoodista vaikeasti ylläpidettävää tai jopa käyttökelvotonta (Rosenkranz ja muut, 2024, s. 9). Rosenkraz ja muut (2024, s. 9) kuvailevat pitkällä aikavälillä tapahtuvaa koodin laadun huonontumista, entropian kasvua, koodin rappeutumiseksi (engl. code decay), jota esiintyy osittain kaikissa projekteissa, kun muutoksia tehdään projektin alkuperäisen julkaisun jälkeen. Feathers painottaa (2005, s. 7–8) muutoksia tehdessä vaikeimman osuuden olevan olemassa olevan toiminnan pitäminen samana uuden toiminnan rinnalla. Uudet ominaisuudet ovat yleisesti paljon pienemmässä osassa koko ohjelman aiempaan toimintaa nähtynä (kuvio 4) ja täten muutosten tekeminen on luo riskin ohjelman aiemmalle toiminnan muuttumiselle (Feathers, 2005, s. 8). Muutosten luoman riskin pienentämiseksi Feathers (2005, s. 8) esittää kolme kysymystä, jotka tulee huomioida perintöjärjestelmiä muuntaessa:

1. Selvitä, mitä muutoksia joudutaan tekemään.
2. Miten tiedämme, että muutokset on tehty oikein?
3. Miten tiedämme, ettemme ole rikkoneet jotain aiempaa toimintaa?

Kohdeyrityksen kehitystyökalua testaava testityökalu ja siitä haluttu hyöty perustuu näiden kysymysten pohjalle. Testityökalun avulla saadaan varmuus kehitystyökalun aiemman toiminnan oikeellisuudesta kuin myös mahdollisista tahattomista

eroavaisuuksista. Feathersin kolme kysymystä antaa hyvän pohjan perintöjärjestelmien kanssa työskentelyyn, jota käsitellään seuraavassa aliluvussa.



Kuvio 4. Olemassa olevan toiminnan suhde uuteen toimintaa (Feathers, 2005, s. 7).

3.2 Työskentely perintöjärjestelmien kanssa

Aiempaan pohjautuen, perintöjärjestelmien kanssa työskentely vaatii tarkkaa riskienarviointia olemassa olevan toiminnan ylläpitämiseksi kuin myös mahdollisimman tehokkaita testejä varmistamaan ohjelman aiemman toiminnan. Lisäksi perintöjärjestelmien kanssa työskentelyyn löytyy useita kehityskäytänteitä ja strategioita, jotka auttavat toimimaan perintöjärjestelmien rajoitteiden kanssa. Seuraavissa alaluvuissa on esitelty tapoja toimia perintöjärjestelmien kanssa.

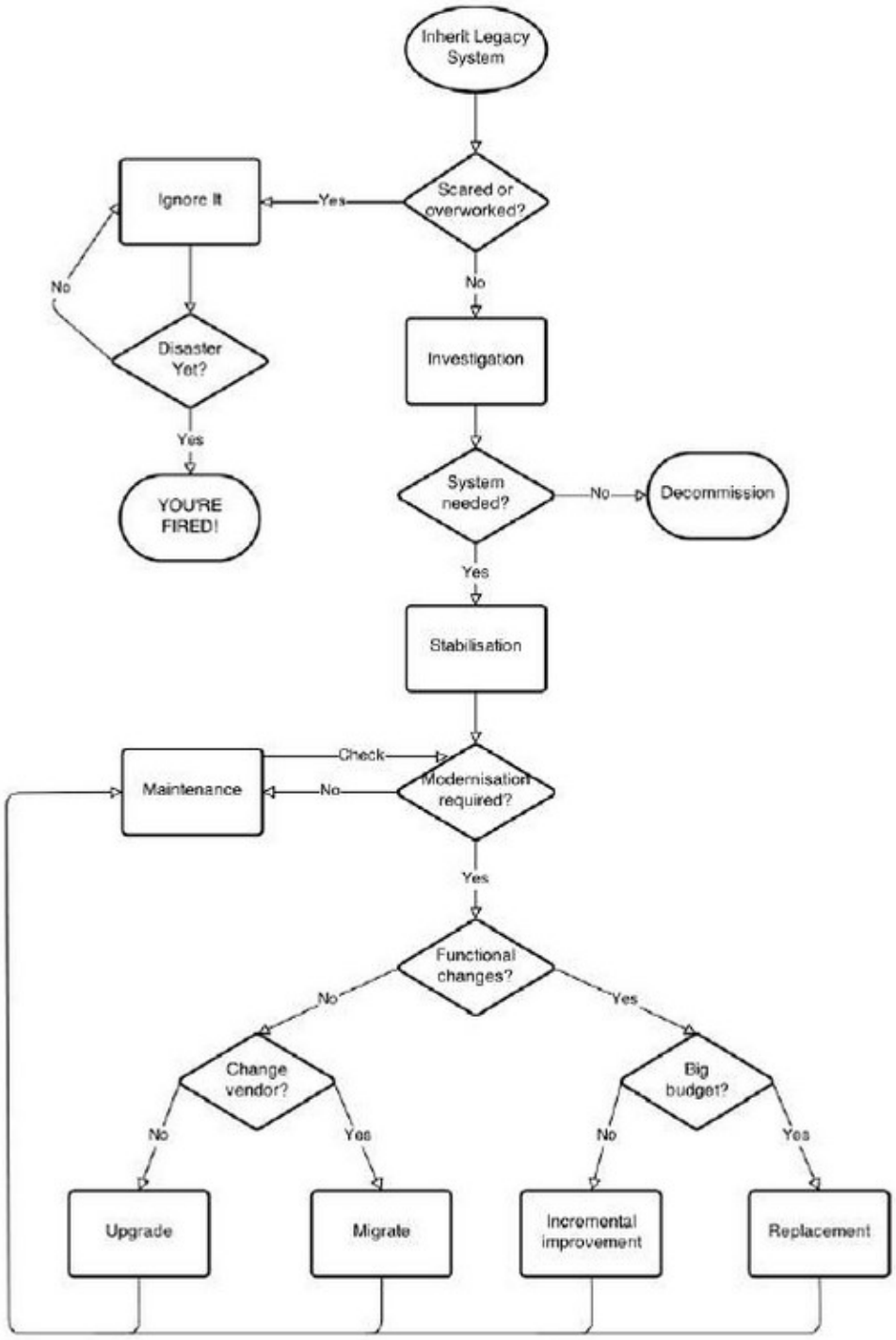
3.2.1 Ylläpitostrategia

Annett (2019, s. 22) esittää perintöjärjestelmien ylläpitoa ja kehitystä kuvaavan kaavion (kuvio 5), joka nosta esille keskeisiä toimintoja ja päätöksentekovaiheita perintöjärjestelmän elinkaaren aikana. Kaavio on muodoltaan aktiviteettikaavio (engl. activity diagram), joiden tarkoitus on kuvailla tietyn prosessin logiikkaa tai työnkulkua (Fowler, 2003, s. 89). Aktiviteettikaaviot koostuvat alku- ja loppupisteistä, joiden väliin haluttu logiikka esitetään toiminta- ja päätöslohkoilla, jotka yhdistetään kulkua kuvaavilla nuolilla (Fowler, 2003, s. 90). Toimintalohkoilla on yleisesti yksi sisään- ja ulostulo, jotka kuvaavat prosessissa tapahtuvaa toimintaa. Päätöslohkoilla on yksi

sisääntulo ja useampi sen loogista operaatiota vastaava ulostulo, jotka kuvaavat prosessissa tehtäviä päätöksiä.

Annett (2019, s. 22) esittämä kaavio jakautuu kahteen keskeiseen osaan: perintösysteemin ylläpitoon ja systeemistä eroon pääsemiseen. Kaavio alkaa yleistyksellä, jonka mukaan on mahdollista tietyissä rajoissa syrjäyttää perintösysteemi ja keskittyä väliaikaisesti muualle. Ennen pitkään kuitenkin systeemille pitää tehdä jotain ja silloin ennen kuin systeemiä aletaan purkamaan tai ylläpitämään, tehdään systeemin toiminnalle ja toteutukselle perinpohjainen analyysi. Analyysin pohjalta osataan tehdä päätöksiä sen mahdollisesti alasajosta kuin jatkokehityksen toimenpiteistä. Analyysin jälkeen keskitytään vakauttamaan ja päivittämään järjestelmä siten että pyritään välttämään järjestelmän jatkuvaa rappeutumista enne seuraavia toimenpiteitä (Annett, 2019, s. 96).

Stabiili järjestelmä arvioidaan ennen mahdollista jatkokehitystä sen modernisoinnin tarpeen mukaan (Annett, 2019, s. 22). Tapauksessa missä systeemi koetaan tarpeeksi elinvoimaiseksi, keskitytään systeemin ylläpitoon ja tasaisin väliajoin palataan takaisin tarkastamaan systeemin modernisoinnin tarve. Kun systeemin koetaan tarvitsevan kehitystä, on tehtävä tärkeä päätös funktionaalisten muutosten tarpeen välillä, jotka voivat johtaa järjestelmän jatkokehitykseen tai suuremmalla budjetilla uuden korvaajan kehitykseen (Annett, 2019, s. 22). Jos systeemi koetaan toiminnallisella tasolla käyttökelpoiseksi mutta kuitenkin kehitystarpeessa olevaksi, keskitytään joko järjestelmän päivittämiseen tai olemassa olevan teknologian muuntamista uudemmalle pohjalle (Annett, 2019, s. 22). Lopulta kaavio ajautuu aina takaisin ylläpitoon, jonka keskeinen ajatus on pitää järjestelmä käyttökelpoisena.



Kuvio 5. Perintöjärjestelmän ylläpitostrategia (Annett, 2019, s. 22).

3.2.2 Refaktorointi

Refaktorointi (engl. refactoring) on työtä, jolla parannetaan koodin laatua muun muassa parantamalla sen luettavuutta ja rakennetta, ilman että sen toiminta muuttuu (Fowler & Beck, 1999, s. 46). Refaktoroimalla vähennetään lähdekoodin mahdollisia virheen aiheuttajia mutta ei kuitenkaan itsessään korjata virheitä, mikä voi kuulostaa erikoiselta. Refaktorointi toimii päinvastoin yleiseen suunnittelu lähtöiseen ohjelmistokehitykseen, jossa suunnitellaan ensin ja koodataan myöhemmin. Refaktoroimalla voidaan muuntaa ja parantaa ohjelman suunnittelua ja arkkitehtuuria suunnittelun jälkeen, joka tekee siitä olennaisen työkalun perintöjärjestelmien kanssa toimimiseen (Fowler & Beck, 1999. s. 9). Feathers (2005, s. 5–6) painottaa, että refaktorointi eroaa koodin optimoinnista tai siistimisestä siten, että refaktoroimalla halutaan konkreettisesti muuttaa ohjelman rakennetta paremmaksi toisin kuin optimoinnissa, jossa keskitytään esimerkiksi ohjelman resurssien säästämiseen tai tehokkuuteen.

Kuten aiemmin on esitetty, ajan myötä järjestelmät ja niiden lähdekoodi, rappeutuu ja ylläpito vaikenee muutosten ja vaihtuvien vaatimusten myötä. Pienin askelin ohjelman lähdekoodi erkaantuu sen alkuperäisestä suunnittelusta ja uusien vaatimusten ja ominaisuuksien tarpeessa koodin kanssa työskentely vaikenee merkittävästi, joka on keskeinen ongelma perintöjärjestelmissä (ks. Fowler & Beck, 1999, s. 9; Rosenkranz ja muut, 2024). Erityisesti perintöjärjestelmien uusien ominaisuuksien kehittäminen edellyttää riskialttiita muutoksia, jotka voivat huonosti testatuissa järjestelmissä aiheuttaa vaikeita sivuvaikutuksia (ks. Annett, 2019, luku 3; Feathers, 2005). Näiden välttämiseksi hyödynnetään refaktorointia.

Refaktorointi ei ole yksi menetelmä vaan laaja kokoelma eri tekniikoita ja työkaluja jäsentelemään koodia parempaan muotoon, joita on kehitetty vuosikymmenien varrella (Fowler & Beck, 1999). Fowler ja Beck (1999, luku 6) ja Feathers (2005, s. 415–419) nostavat esille keskeiseksi tekniikaksi metodin erottamisen (engl. extract method), jolla pilkotaan suuria metodeja pienemmiksi kokonaisuuksiksi. Muita tekniikoita on muun muassa muuttujien pilkkominen, luokan erotus tai ehtolauseen korvaus oliopohjaisella

monimuotoisuudella (Refactoring.Guru, 2015). Refaktorointi ei kuitenkaan ole ennalta määrätty työvaihe verrattuna esimerkiksi ohjelmistotestauksessa suoritettaviin testausvaiheisiin ja onkin enemmän työkalu ja toimintatapa hyvän koodin tuottamiseksi, joka on keskeinen osa ohjelmistokehitystä (Kanat-Alexander, 2012). Nykypäivänä monet kehitysympäristöt sisältävät tehokkaita refaktorointityökaluja, joita voidaan hyödyntää suoraan kehitystyön aikana (IEEE Computer Society, 2024, luku 4, s. 14).

3.2.3 Testaus

Muun muassa refaktoroimalla tehdyt muutokset ovat erityisen vaikeita toteuttaa ilman testejä, jolla saadaan varmuus ohjelman toiminnasta vielä muutosten jälkeen. Ilman toimintaa takaavia testejä Feathers (2005, s. 9) kuvastaa kehityksen olevan muutosten jälkeistä ”toivomista”, että järjestelmä toimii vielä halutulla tavalla. Vaihtoehtoisesti parempi lähestymistapa Feathersin mukaan on ”peitä ja muokkaa”, jossa varmistetaan, että muutettava osuus on testattu ennen muutosten tekoa, jonka jälkeen testillä voidaan varmistaa kohdan toiminta. Regressiotestaus on keskeinen toimintatapa ja tehokas työkalu ohjelman toiminnan varmistamiseksi. Aktiivisessa kehityksessä vain tietyn osajoukon testien ajo mahdollisimman nopeasti korostuu, jossa yksikkötestit ovat tärkeässä osassa antamassa konkreettista varmuutta ohjelman toiminnasta (Feathers, 2005, s. 11).

Testipohjaisella kehityksellä on keskeinen rooli niin uusien järjestelmien kehittämisessä kuin vanhojen perintöjärjestelmien ylläpidossa. Se tukee järjestelmän toimivuuden varmistamista kehitystyön aikana, erityisesti ”peitä ja muokkaa” -lähestymistavassa. Samalla se mahdollistaa kattavan testisarjan (engl. test suite) rakentamisen, joka muodostaa laadukkaan ja komponenttipohjaisen järjestelmän kehityksen ja ylläpidon selkärangan. Laaja, automaattinen ja nopeasti toimiva testisarja toimii tehokkaana virheiden havainnointityökaluna kuin myös vähentää virheiden korjaukseen kuluvaa aikaa (Fowler & Beck, 1999, s. 73–74).

Beckin (2002, s. 9) mukaan testipohjainen kehitys perustuu kahteen perussääntöön:

1. Yhtäkään riviä koodia ei tule kirjoittaa ilman, että sitä vastaa alkuun epäonnistuva automaattinen testi.
2. Kaikki toistuva ja tarpeettoman monimutkainen koodi poistetaan ja refaktoroidaan mahdollisimman selvään muotoon.

Sääntöjen pohjalta testipohjainen kehitys takaa modulaarisen ja testatun systeemin, jonka kehitys ja muutokset perustuvat testeistä saatuun varmuuteen. Itse kehitys koostuu kolmesta vaiheesta, missä ensimmäiseksi kirjoitetaan epäonnistuva testi, jonka jälkeen kirjoitetaan testin läpäisevä koodi ja lopulta testin avulla kirjoitettu koodi refaktoroidaan mahdollisimman tehokkaasti (Beck, 2002, s. 9). Testipohjaisesti kehitetty järjestelmä pidemmällä aikavälillä on huomattavasti varmempi ylläpitää, joka on konkreettinen apu perintöjärjestelmien ylläpidossa (Rosenkranz ja muut, 2024, s. 13–14).

3.2.4 Perintöjärjestelmä muuntoalgoritmi

Feathers (2005, s. 18–20) esittää perintökoodin muuntamiseen soveltuvan algoritmin, jonka tarkoitus on edistää järjestelmän testausta, joka parantaa järjestelmän ylläpidettävyyttä. Algoritmi koostuu viidestä osasta: muutos- ja testauskohtien kartoittamisesta, riippuvuuksien katkaisusta, testien kirjoituksesta ja testattavien kohtien muuntamisesta ja refaktoroinnista.

Algoritmissa riippuvuuksien katkaisulla tarkoitetaan testattavan koodin muuntamista testiympäristössä ajettavaksi (Feathers, 2005, s. 19). Yleinen vaikeus perintöjärjestelmän testauksessa on komponenttien välisten riippuvuuksien tiukka yhteys ja monimutkaisuus, joka tekee komponentin testauksesta vaikeaa (Feathers, 2005, s. 19). Komponenttien saamiseksi toimimaan eristetyissä testiympäristöissä hyödynnetään tekoriippuvuuksia, joiden avulla teeskennellään testattavan komponentin riippuvuuksia, kuten esimerkiksi tietokantayhteyttä (Koskela, 2013, luku 2.6).

Algoritmin viimeisessä vaiheessa Feathers (2005, s. 20) kehottaa testipohjaisen kehityksen hyödyntämistä uusien ominaisuuksien lisäämisessä, käyttäen eri refaktorointimenetelmiä. Testipohjaisen kehityksen käyttö on oleellista vastaamaan Feathersin esittämään dilemmaan, jossa koodin muuntamiseksi tulisi se olla testattua mutta taas testien lisäämiseksi joutuu muuntamaan koodia. Lopulta kuitenkin, aiempaa toimintaa testaavien testien luonti ja koodin muutosten tekeminen on loppukädessä ohjelmoijan ja tiimin vastuulla, jotka tekevät lopullisen riskiarvion tarvittavista muutoksista (Feathers, 2005, s. 8–9).

3.3 Perintöjärjestelmän piirteet kohdeyrityksen kehitystyökalussa

Vaikka kohdeyrityksen kehitystyökalu on aktiivisesti ylläpidetty, on siinä silti selviä perintöjärjestelmän piirteitä, jotka tekevät sen ylläpidosta hankalaa. Ensinnäkin työkalu on lähes kaksikymmentä vuotta vanha ja toteutettu aikansa teknologioilla, jotka ovat nykypäivään verrattuna vanhoja. Rosenkranz ja muut (2024, s. 7) huomauttavat perintöjärjestelmien olevan yleisesti reilusti yli kymmenen vuotta vanhoja ja perustuvat vanhentuneeseen teknologiaan, mikä pätee myös kehitystyökaluun. Toisaalta he lisäävät, että vanhaksi määritelty systeemi ei tarkoita systeemin olevan vanhentunut, joka osaltaan on myös totta kehitystyökalun kohdalla, sillä se on edelleen käytössä. Kehitystyökalu on kehitetty .NET alustan päälle, jonka käytetty versio on 4.8, joka on vielä ainakin toistaiseksi tuettu julkaisu (Microsoft, n.d.). Vaikka kehitystyökalu on vanha, ei se ole vanhentunut tietoturvallisesti ylläpidetyn version vuoksi, mikä on selkeä vahvuus. Yleisesti normaaliksi ohjelman käyttöiäksi koetaan noin 6–8 vuotta ennen sen käytöstäpoistamisvaihetta (engl. end of life) (Assaad & Henein, 2022).

Toinen konkreettinen merkki kohdeyrityksen kehitystyökalussa on sen koodin monimutkaisuus ja dokumentaation puute (Rosenkranz ja muut, 2024, s. 11). Vuosien varrella lisääntyneet ominaisuudet, lukemattomat ylläpitotoimet ja optimoinnit ovat tehneet työkalun lähdekoodista monimutkaista ja kasvattaneet työkalua huomattavasti, joka on yleistä perintöjärjestelmissä (Rosenkranz ja muut, 2024, s. 11). Testien puute ja

vaikeasti ymmärrettävä lähdekoodi, johtaa huolimattomiin muutoksiin, hätäratkaisuihin ja kompromisseihin (Rosenkranz ja muut, 2024, s. 15).

Kolmas kriittinen piirre on perintöjärjestelmän tärkeys yritystoiminnassa. Kehitystyökalu on aktiivisesti käytetty työkalu, joka on olennaisesti osa kehitysympäristöä ja sen korvaus osoittautuu kalliiksi ja vaikeaksi prosessiksi, joka on yleistä perintöjärjestelmissä (Rosenkranz ja muut, 2024, s. 18). Uuden, modernin version kehitys samaisesta työkalusta vaatii aikaa, rahaa ja osaamista, jota harvoin on varsinkin liiketoiminnan kriittisten tuotteiden kehitystyökalujen kehittämiseen varattu (Annett, 2019, s. 33).

4 Tekninen toteutus

Tässä luvussa käsitellään testityökalun teknistä toteutusta ja toimintaa.

4.1 Python

Python on suosittu ja laajasti käytetty tulkattu, korkean tason oliopohjainen ohjelmointikieli, jonka yksinkertainen ja helppo syntaksi tekee siitä tehokkaan työkalun ohjelmistokehityksessä kuin sen tukitoimissa (Python Software Foundation, n.d.). Python on laajasti käytetty sekä koulutuksessa että teollisuudessa sen monien vahvuuksien ansiosta, kuten lähdekoodin luettavuuden, valmiiden tietorakenteiden ja helpon alustariippumaton ajon vuoksi (Nagpal & Gabrani, 2019). Toisaalta tulkattuna kielenä Python ei ole yhtä suorituskykyinen kuin monet muut käännetyt ohjelmointikielet eikä sen ohjelmistopaketit ole välttämättä yhtä kehittyneitä kuin vastaavissa kielissä, joka tulee huomioida uusia projekteja aloittaessa (Nagpal & Gabrani, 2019).

Python valittiin testityökalun kehitykseen sen yksinkertaisuuden ja laajan standardikirjaston takia. Pythonin standardikirjasto on Pythoniin sisäänrakennettu laaja kokoelma työkaluja muun muassa tiedostojen hallintaa, statistiikkaa ja testausta varten (Python Software Foundation, 2025d). Kohdeyrityksessä Pythonia hyödynnetään myös muissa kehitystehtävissä, joka helpottaa testityökalun käyttöönottoa. Koska testityökalu on kohdeyrityksen kehitystyökalusta irrallinen ohjelma, mahdollisti se ohjelmointikielen vapaan valinnan, joista Python osoittautui edellä mainittujen mukaisesti sopivimmaksi. Testityökalun kehitykseen harkittiin myös C#:ia, mutta sitä ei otettu käyttöön testityökalun toteutuksen vaatimien ulkoisten pakettiriippuvuuksien ja monimutkaisuuden vuoksi. Toisaalta C#:n hyöty olisi ollut saman kielen hyödyntäminen kehitystyökalussa ja sen testityökalussa. Seuraavassa luvussa käsitellään tarkemmin käytettyjä standardikirjaston moduuleja ja niiden käyttötarkoitusta.

4.2 Käytetyt Python-kirjastot

Testityökalu hyödyntää Pythonin standardikirjastosta löytyviä apukirjastoja. Pelkästään standardikirjaston hyödyntäminen vähentää mahdollisiin tietoturvauhkiin kajoamista ulkoisten pakettipalveluiden kautta kuin myös helpottaa testityökalun kehitystä ja käyttöönottoa (ks. Alfadel ja muut, 2023; Ruohonen ja muut, 2021). Standardikirjaston käyttö mahdollistaa riippumattomuuden ulkoisista työkaluista ja varmuuden toimivista paketeista (Python Software Foundation, 2025d). Lisäksi kohdeyrityksen kehitystyökalu on toteutettu ilman ulkoisia kirjastoja, joten pelkän standardikirjaston hyödyntäminen oli luonteva valinta.

`Difflib` on apukirjasto laskemaan muutoksia niin merkkijonojen kuin muiden vertailtavien olioiden välillä (Python Software Foundation, 2025b). Generoitujen kooditiedostojen eroja vertaillaan `difflib.unified_diff` metodilla", joka kokoaa kaikkien vertailtavien kohteiden löydetyt erot selkeään ja luettavaan muotoon.

Eroavaisuuksien raportointiin hyödynnetään XML (Extensive Markup Launguage) tiedostomuotoa, joka on tehokas tiedostomuoto säilyttämään rakenteellista dataa ihmisille ja koneille luettavassa muodossa (Alnaqeib ja muut, 2010). XML tiedostojen kirjoittamiseen hyödynnetään standardikirjaston `xml.etree.ElementTree` rajapintaa (Python Software Foundation, 2025e)

Testattava kehitystyökalu ja sille määriteltävät parametrit ajetaan standardikirjaston `subprocess` moduulin `run()` metodilla, joka ajaa parametrina annetun ohjelman (Python Software Foundation, 2025c).

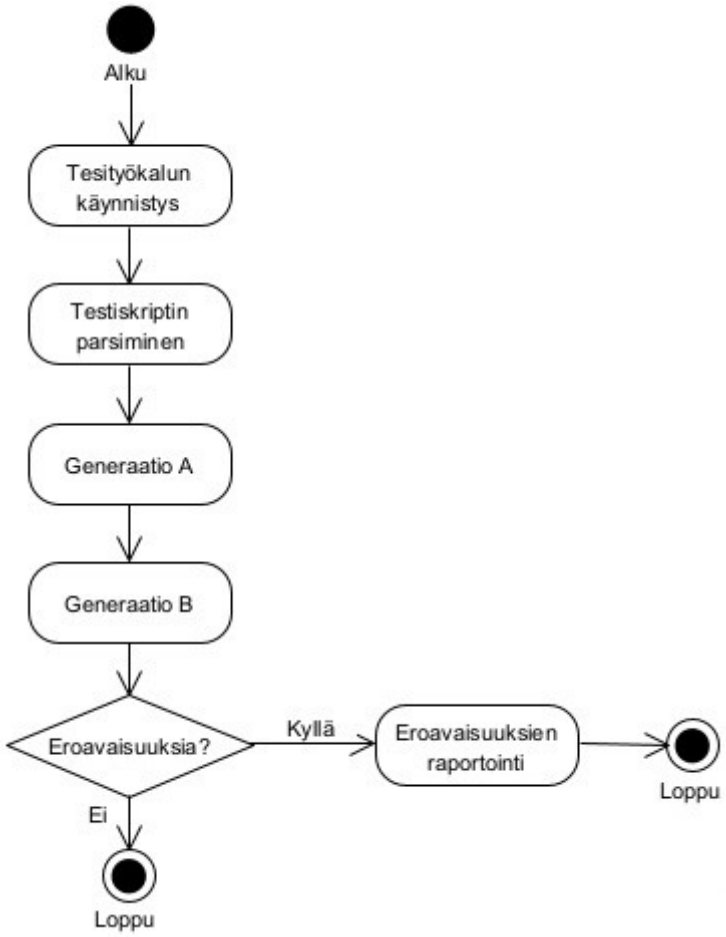
Testityökalun ajoparametrien hallintaan hyödynnetään `argparse` moduulia (Python Software Foundation, 2025a). Parametrien avulla määritellään ajoon tarvittavat tiedostopolut ja halutessaan muokataan ulostuloon vaikuttavia asetuksia.

Kehityksen tukena käytettiin `flake8` kirjastoa, joka staattisesti testaa lähdekoodin vastaavuutta Pythonin PEP-8 tyylistandardiin ja varmistaa myös, ettei lähdekoodi ole liian monimutkaista tai siinä ei ole rakenteellisia virheitä kuten käyttämättömiä muuttujia (ks. Stapleton Cordasco, 2016; van Rossum ja muut, 2001). Lisäksi lähdekoodi formatoitiin `Black` työkalulla, joka asettelee lähdekoodin vastaamaan PEP-8 standardi vaatimaa ulkoasua (Langa, n.d.).

4.3 Testityökalun toiminta

Kuviossa 6 esitellään testityökalun ajon aikana suoritettavat keskeiset vaiheet. Testityökalu ajaa kaksi vertailtavaa versiota samasta kehitystyökalusta, generoi niiden avulla kaksi vastaavaa versiota generoiduista tiedostoista ja vertailee tiedostojen välisiä erovaisuuksia. Ajettavat testigeneroinnit määritellään alkuun parsittavassa testiskriptissä, joka myös sisältää kehitystyökalun generointiin liittyviä määrittelyjä ja asetuksia. Kahden eri version tuottamat tiedostot vertaillaan `difflib` kirjaston

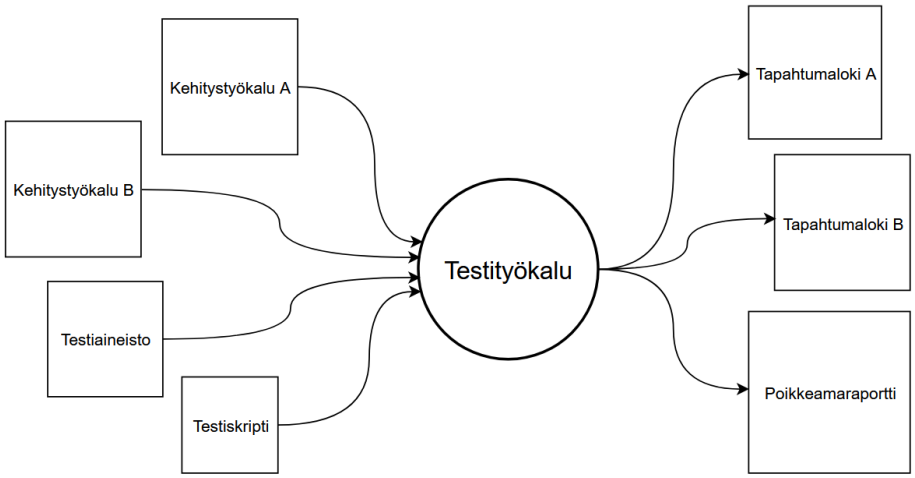
`unified_diff` metodilla ja raportoidaan ajajan konsoliin sekä XML-tiedostoon myöhempää analyysiä varten.



Kuvio 6. Testityökalun toiminnan keskeiset vaiheet**.**

Kuvio 7 kuvaa testityökaluun syötettäviä argumentteja ja sen tuottamaa ulostuloa. Testityökalu on komentoriviltä ajettava Python-ohjelma, jonka ajo määritellään sille annettujen argumenttien pohjalta. Testauksen keskeiset argumentit ovat kahden vertailtavan kehitystyökalun `.exe`-tiedostojen tiedostopolut sekä kehitystyökaluille ajettava testiskripti. Testiskriptiä varten määritellään myös testiaineistojen tiedostopolku, jossa sijaitsevat generointeja määrittelevät spesifikaatiot. XML-raportin lisäksi testityökalu tallentaa molempien kehitystyökalujen generointien lokitiedostot, jotka sisältävät debug-lokeja ajon kulusta ja sen mahdollisista ongelmista. Testityökalu sisältää myös lisäparametrejä, joilla voi määritellä ulostulotiedostojen

tulostuskansioiden tiedostopolkuja ja muuntaa ulostulon asetuksia. Testityökalu ajetaan joko suoraan komentoriviltä kutsumalla Python-tulkilla testityökalua oikeilla parametreillä, tai se voidaan integroida automaattisesti ajettavaan testiautomaatiojärjestelmää.



Kuvio 7. Testityökalun komentoriviargumenttien ja ulostulojen kuvaus.

Generointia varten ajettava testiskripti on kehitystyökalulle suunniteltu skriptitiedosto, joka sisältää sille käynnistyksen yhteydessä suoritettavia komentoja. Skriptin kautta määritellään kehitystyökalun toimintaparametrit, tiedostopolut tarvittaville lisätiedostoille ja lokitiedostoille sekä suoritettavat lisätoiminnallisuudet ja generoinnin testauksen kohteena olevat spesifikaatiot. Skripti käyttää kehitystyökalukohtaista, muista formaateista poikkeavaa syntaksia, joka ei ole yhteensopiva muiden skriptikielten kanssa. Se toimii keskeisenä ohjausmekanismina kehitystyökalun pitkän aikavälin ylläpidossa, sillä sen komentojen avulla hallitaan vuosien aikana lisättyjen ominaisuuksien ja korjausten käyttöä. Testauksessa kaikkia konfiguraatioita ja ominaisuuksia ei kuitenkaan ole aina tarpeen käyttää ja kehitystyön aikana voidaan hyödyntää kevyempiä, rajattua spesifikaatiota ajavia skriptejä nopeaa testipohjaista iterointia varten. Testiautomaatiossa, esimerkiksi yöllä suoritettavassa ajossa, voidaan käyttää laajempaa regressiotestaukseen soveltuvaa testiskriptiä, joka kattaa koko spesifikaatiopoolin mutta sen suorittaminen vie ajallisesti kauemmin.

4.4 Raportointi

`Difflib` tuottama poikkeamaraportti pohjautuu GNU käyttöjärjestelmän `Diffutils` paketin käyttämään eroavaisuuksien yhdistettyyn formaattiin (engl. unified format) (Free Software Foundation, n.d.). Poikkeamaraportin formaatin tarkoituksena on esittää eroavat rivit sekä muutama kontekstiin liittyvä lisärivi luettavuuden parantamiseksi (Free Software Foundation, n.d.). Sama eroavaisuusformaatti on käytössä muun muassa suositussa Git-versionhallintajärjestelmässä, joka tekee siitä helposti lähestyttävän monelle kehittäjälle, jotka ovat ennestään tottuneita Gitin tuottamiin poikkeamaraportteihin (Straub & Chacon, 2014, s. 33–36).

Liitteessä 1 on esitetty GNU:n dokumentaation esittämä esimerkki poikkeamaraportista (Free Software Foundation, n.d.). Poikkeamaraportit koostuvat kolmesta keskeisestä osasta: kaksirivisestä otsikosta, joka ilmoittaa muuttuneen tiedoston ja muutoksen ajankohdan, muuttuneista osioista sekä rivipohjaisista eroavaisuuksista (Free Software Foundation, n.d.). Rivipohjaisten muutosten tunnukset ovat esitetty taulukossa 1.

Taulukko 1. Rivipohjaisten muutosten indikaattorit (mukailtu lähteestä: Python Software Foundation, 2025b).

Indikaattori	Kuvaus
-	Rivi uniikki alkuperäiseen
+	Rivi uniikki muuttuneeseen
(tyhjä)	Rivi esiintyy molemmissa
?	Rivi ei esiinny kummassakaan

Muuttuneen osion alkua kuvastaa erityinen rivi: `@@ from-file-line-numbers to-file-line-numbers @@`, joka ilmaisee, miltä riviväliltä muutokset alkavat ja mihin ne päättyvät. Liitteen 1 esimerkin jälkimmäinen muuttunut osio `@@ -9,3 +8,6 @@` osoittaa, että vanhassa muodossa eroavaisuudet alkavat riviltä 9 ja jatkuu seuraavat kolme riviä, kun taas uudessa muodossa muokattu osio alkaa riviltä 8 ja jatkuu seuraavat

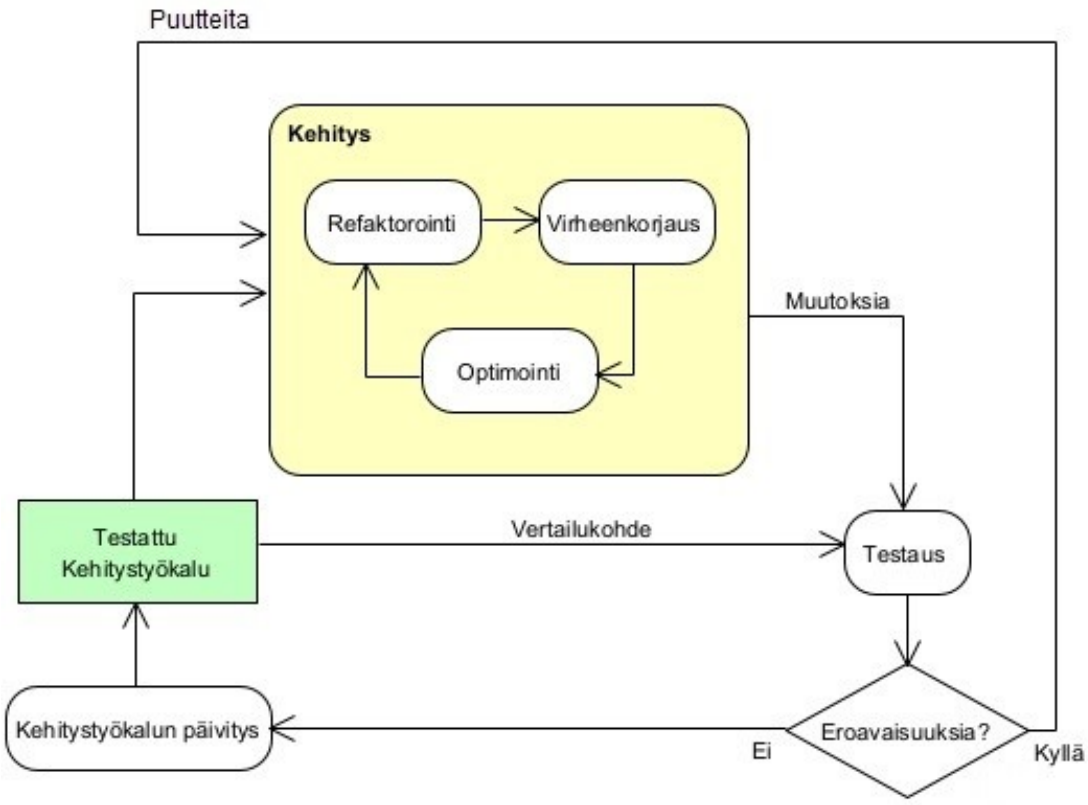
kuusi riviä. Merkintä kuvastaa kolmen rivin lisäystä ja koko muuttuneen osion siirtymistä yhden rivin ylöspäin.

Testityökalun löytämät poikkeamaraportit tallennetaan XML-tiedostoon puumaiseen rakenteeseen, jossa erot asetetaan `<diffs time=”AIKALEIMA”>` lohkon alle ja jokainen testattu spesifikaatio tallennetaan omaan lohkoon `<diff name=SPESIFIKAATION_NIMI>`. Jokainen lohko sisältää testatun tiedostoparin poikkeamaraportin edellä mainitussa formaatissa ja testatun spesifikaation nimen lohkon attribuuttina. Raportin aikaleima tallennetaan attribuuttina `diffs` lohkoon mahdollista arkistointia varten. XML-rakenne mahdollistaa luettavuuden koneellisesti ja samalla ihminen voi tarkastella raporttia XML-tiedostomuotoa tukevalla tekstieditorilla.

5 Tulokset ja johtopäätökset

Tässä luvussa käsitellään tuotetun testityökalun käyttöä kohdeyrityksen kehitystyökalun kehityksessä sekä sen vastaavuutta tutkielmassa määriteltyihin tutkimuskysymyksiin. Ensimmäiseen tutkimuskysymykseen, *Miten testityökalu kehitetään vertailemaan kahden eri kehitystyökalun versioiden generoinnin muutoksia?,* vastattiin aiemmassa teknistä toteutusta käsittelevässä luvussa. Testityökalu toteutettiin Python-ohjelmointikielellä hyödyntäen sen laajaa standardikirjastoa. Kehitystyökalun generoitujen tiedostojen vertailuun hyödynnettiin standardikirjaston `difflib` moduulia, jonka avulla generoitujen tiedostojen eroavaisuudet saadaan raportoitua tehokkaasti GNU:n tunnetuksi tehdyssä eroavaisuusformaatissa. Poikkeamaraportit tallennetaan myöhempää analyysia varten XML-tiedostomuodossa.

Toiseen tutkimuskysymykseen, *Miten testityökalun avulla varmistetaan, että kohdeyrityksen kehitystyökalun muutokset eivät riko aiempien versioiden toimintaa?*, vastatakseen tulee pohtia testityökalun käyttöä kohdeyrityksen kehitystyökalun kehityksessä. Testityökalua hyödyntävä kehitystyökalun testipohjainen kehitys on kuvattu kuviossa 8. Testityökalu voidaan integroida osaksi kehitystyökalun kehitystyötä mahdollistamaan testatun kehityssyklin muun muassa lähdekoodin refaktorointia tai generoinnin optimointia varten. Kuvion 8 kuvaamalla testisyklillä kehitystyökalun viimeisintä versiota kehitetään esimerkiksi optimoimalla sen toimintaa, ja testityökalulla varmistetaan sen toiminta verrattuna edelliseen versioon. Jos testityökalulla huomataan muutoksia optimoidussa versiossa, palataan takaisin kehitysvaiheeseen korjaamaan muutokset ennen kuin ajetaan testit uudestaan, jonka jälkeen testien läpäistessä voidaan turvallisesti päivittää kehitystyökalua.



Kuvio 8. Kehitystyökalun testipohjainen kehitysprosessi.

Kolmannessa tutkimuskysymyksessä, *Mitkä ovat testityökalun käytön haasteet ja rajoitteet osana kohdeyrityksen kehitystyökalun kehitystä?,* keskeisinä haasteina ovat testiajurin nopeus ja testattujen generointien määrä. Mitä enemmän testigenerointeja ajaa, sitä paremman varmuuden saa kehitystyökalun toiminnasta mutta toisaalta testien ajo kestää kauemmin. Pidempi testien ajoaika johtaa kehityksen pidempään palautesykliin, joka vaikuttaa suoraan kehittäjien tehokkuuteen (Koskela, 2013, s. 8–9). Kehitystyökalun aktiivisessa kehityksessä testiskriptiin tulisi valita mahdollisimman pieni ja tehokas määrä testattavia spesifikaatioita, joiden avulla testien ajosykli pysyy mahdollisimman lyhyenä. Lyhyiden kehityssyklien valmistuttua voidaan koko kehitystyökalu testata laajemmalla testiskriptillä, joka testaa mahdollisesti kaikki generoitavat spesifikaatiot esimerkiksi yön aikana testiautomaatiolla ajettuna. Laaja takautuva regressiotesti takaa, että kehityksen aikana tehty kompromissi testien ajon nopeudesta ei ole johtanut testaamattomien generointien sivuvaikutuksiin.

Testityökalu voidaan automaattisia regressiotestejä varten integroida jatkuvan integroinnin (CI) järjestelmään, jossa voidaan automaattisesti testata koodiin tehtyjä muutoksia. Python-toteutus mahdollistaa integraation esimerkiksi Microsoft Azure Pipelines ympäristöön (Microsoft, 2024). Lisäksi Python-applikaatio on mahdollista paketoida omaksi asennettavaksi applikaatioksi, jonka avulla sen käyttöönotosta voisi tehdä vielä aiempaa helpompaa sillä testityökalun ajaminen ei vaatisi erillistä Python asennusta (Python Packaging Authority, 2025). Testiautomaatioon integrointi ja tarkempi testityökalun käyttöönotto jätetään tutkielmassa jatkokehityksen kohteeksi.

Testityökalun rajoituksista on tärkeä huomata, että testityökalu itsessään ei ole integroitu mihinkään testijärjestelmään, vaan se sisältää vain toiminnallisuuden kehitystyökalun generoinnin testaukseen. Kehitystyökalun testauksen laajentuessa voidaan testityökalu integroida siihen ja hyödyntää liittyvän testin ratkaisuun eroavaisuusraportin sisältöä. Jatkokehityksenä testityökaluun voisi lisätä onnistumista kuvaavan palautusarvon helpottamaan testisarjaan integrointia. Lisäksi on huomattavaa, että testityökalu ei testaa kehitystyökalun muita osuuksia kuten sen käyttöliittymän toimintaa. Kehitystyökalun muuta toimintaa ja suorituskykyä voidaan seurata testityökalun luomien lokitiedostojen avulla, mutta perusteellisempi analyysi edellyttää yksikkö- ja integraatiotestien kehittämistä itse kehitystyökalulle.

Lähteet

Alfadel, M., Costa, D. E., & Shihab, E. (2023). Empirical analysis of security vulnerabilities in Python packages. *Empirical Software Engineering*, *28*(3), 1–37. https://doi.org/10.1007/s10664-022-10278-4

Alnaqeib, R., Alshammari, F. H., Zaidan, M. A., Zaidan, A. A., Zaidan, B. B., & Hazza, Z. M. (2010). *An Overview: Extensible Markup Language Technology*. *2*(6).

Annett, R. (2019). *Working with Legacy Systems: A Practical Guide to Looking after and Maintaining the Systems We Inherit*. Birmingham: Packt Publishing, Limited.

Assaad, Z., & Henein, M. (2022). *End-of-Life of Software How is it Defined and Managed?* https://doi.org/10.48550/arXiv.2204.03800

Beck, K. (2002). *Test-Driven Development By Example*.

Boccara, J. (2019). *The Legacy Code Programmer's Toolbox*.

Cambridge University Press & Assessment. (n.d.). *Non-deterministic*. Noudettu 23.2.2025 osoitteesta https://dictionary.cambridge.org/dictionary/english/non-deterministic

Feathers, M. C. (2005). *Working effectively with legacy code*. Prentice Hall Professional Technical Reference.

Fowler, M., & Beck, K. (1999). *Refactoring: Improving the design of existing code*. Addison-Wesley.

Fowler, M. (2003). *UML Distilled* (3. p).

Free Software Foundation. (n.d.). *Unified Format (Comparing and Merging Files)*. Noudettu 20. maaliskuuta 2025, osoitteesta https://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html

Garousi, V., Felderer, M., Kuhrmann, M., Herkiloğlu, K., & Eldh, S. (2020). Exploring the industry's challenges in software testing: An empirical study. *Journal of Software: Evolution and Process*, *32*(8), e2251. https://doi.org/10.1002/smr.2251

IEEE Computer Society. (2024). *Guide to the Software Engineering Body of Knowledge*.

Kanat-Alexander, M. (2012). *Code simplicity*. O'Reilly.

Kaner, C., Bach, J., & Pettichord, B. (2002). *Lessons learned in software testing: A context-driven approach*. John Wiley & Sons.

Kasurinen, J. P. (2013). *Ohjelmistotestauksen käsikirja* (1. p). Docendo.

Koskela, L. (2013). *Effective Unit Testing: A Guide for Java Developers*. Manning Publications Co. LLC.

Kumar, D. G. A. (2019). A REVIEW ON CHALLENGES IN SOFTWARE TESTING. *Journal of Information and Computational Science*, *9*(6).

Langa, Ł. (n.d.). *Black 25.1.0 documentation*. Noudettu 19. maaliskuuta 2025, osoitteesta https://black.readthedocs.io/en/stable/

Microsoft. (n.d.). *Microsoft .NET Framework—Microsoft Lifecycle*. Noudettu 17. maaliskuuta 2025, osoitteesta https://learn.microsoft.com/fi-fi/lifecycle/products/microsoft-net-framework

Microsoft. (2024, heinäkuuta 13). *Customize Python pipelines—Azure Pipelines*. Noudettu 17. maaliskuuta 2025, osoitteesta https://learn.microsoft.com/en-us/azure/devops/pipelines/ecosystems/customize-python?view=azure-devops

Mili, A. (2015). *Software Testing: Concepts and Operations*. John Wiley & Sons, Incorporated.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The Art of Software Testing*. John Wiley & Sons, Incorporated. https://doi.org/10.1002/9781119202486

Nagpal, A., & Gabrani, G. (2019). Python for Data Analytics, Scientific and Technical Applications. *2019 Amity International Conference on Artificial Intelligence (AICAI)*, 140–145. https://doi.org/10.1109/AICAI.2019.8701341

Python Packaging Authority. (2025, maaliskuuta 15). *Creating and packaging command-line tools—Python Packaging User Guide*. Noudettu 20. maaliskuuta 2025, osoitteesta https://packaging.python.org/en/latest/guides/creating-command-line-tools/

Python Software Foundation. (n.d.). *What is Python? Executive Summary*. Python.Org. Noudettu 19. maaliskuuta 2025, osoitteesta https://www.python.org/doc/essays/blurb/

Python Software Foundation. (2025a, maaliskuuta 19). *argparse—Parser for command-line options, arguments and subcommands*. Python Documentation. Noudettu 19. maaliskuuta 2025, osoitteesta https://docs.python.org/3/library/argparse.html

Python Software Foundation. (2025b, maaliskuuta 19). *difflib—Helpers for computing deltas*. Python Documentation. Noudettu 19. maaliskuuta 2025, osoitteesta https://docs.python.org/3/library/difflib.html

Python Software Foundation. (2025c, maaliskuuta 19). *subprocess—Subprocess management*. Python Documentation. Noudettu 19. maaliskuuta 2025, osoitteesta https://docs.python.org/3/library/subprocess.html

Python Software Foundation. (2025d, maaliskuuta 19). *The Python Standard Library*. Python Documentation. Noudettu 19. maaliskuuta 2025, osoitteesta https://docs.python.org/3/library/index.html

Python Software Foundation. (2025e, maaliskuuta 19). *xml.etree.ElementTree—The ElementTree XML API*. Python Documentation. Noudettu 19. maaliskuuta 2025, osoitteesta https://docs.python.org/3/library/xml.etree.elementtree.html

Refactoring.Guru. (n.d.). *Refactoring Techniques*. Noudettu 15. maaliskuuta 2025, osoitteesta https://refactoring.guru/refactoring/techniques

Rosenkranz, S., Staegemann, D., Volk, M., & Turowski, K. (2024). Explaining the Business-Technological Age of Legacy Information Systems. *IEEE Access*, *12*, 84579–84611. IEEE Access. https://doi.org/10.1109/ACCESS.2024.3414377

Ruohonen, J., Hjerppe, K., & Rindell, K. (2021). A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI. *2021 18th International Conference on Privacy, Security and Trust (PST)*, 1–10. https://doi.org/10.1109/PST52912.2021.9647791

Stapleton Cordasco, I. (2016). *Flake8: Your Tool For Style Guide Enforcement—Flake8 7.1.0 documentation*. Noudettu 19. maaliskuuta 2025, osoitteesta https://flake8.pycqa.org/en/latest/

Straub, B., & Chacon, S. (2014). *Pro Git (Second Edition)*. Apress. https://doi.org/10.1007/978-1-4842-0076-6

van Rossum, G., Warsaw, B., & Coghlan, A. (2001, heinäkuuta 5). *PEP 8 – Style Guide for Python Code | peps.python.org*. Python Enhancement Proposals (PEPs). Noudettu 19. maaliskuuta 2025, osoitteesta https://peps.python.org/pep-0008/

Liitteet

Liite 1. Esimerkki `Diffutils` poikkeamaraportti

```
--- lao   2002-02-21 23:30:39.942229878 -0800
+++ tzu   2002-02-21 23:30:50.442260588 -0800
@@ -1,7 +1,6 @@
-The Way that can be told of is not the eternal Way;
-The name that can be named is not the eternal name.
 The Nameless is the origin of Heaven and Earth;
-The Named is the mother of all things.
+The named is the mother of all things.
+
 Therefore let there always be non-being,
   so we may see their subtlety,
 And let there always be being,
@@ -9,3 +8,6 @@
 The two are the same,
 But after they are produced,
   they have different names.
+They both may be called deep and profound.
+Deeper and more profound,
+The door of all subtleties!
```