SeAMK
SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Saara Ala-Korte

Yksikkötestaustyökalu Epecin kirjastoprojekteille

Opinnäytetyö
Kevät 2023
Insinööri (AMK), Automaatiotekniikka

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Tutkinto-ohjelma: Insinööri (AMK), Automaatiotekniikka

Suuntautumisvaihtoehto: Koneautomaatio

Tekijä: Saara Ala-Korte

Työn nimi: Yksikkötestaustyökalu Epecin kirjastoprojekteille

Ohjaaja: Jyri Lehto

Vuosi: 2023 Sivumäärä: 37 Liitteiden lukumäärä:

Opinnäytetyön tavoitteena oli selvittää, soveltuuko co♻e-yksikkötestaustyökalu Epecin kirjastoprojektien testaamiseen. CODESYS-sovelluksien testaamiseen on saatavilla toinen testaustyökalu, jonka käyttö vaatii maksullisen lisenssin. Lisenssin vuoksi sitä ei ole mahdollista käyttää kaikissa asiakasprojekteissa. Tarkoituksena oli saada selville, voidaanko testaustyökalu korvata co♻e-testaustyökalulla.

Aluksi tutustuttiin ohjelmistotestaukseen yleisellä tasolla. Ohjelmistotestauksesta käydään läpi testaustasoja, testaustyyppejä ja niiden ohessa myös erilaisia testausmenetelmiä. Opinnäytetyössä myös tutustuttiin siinä käytettyyn CODESYS-ohjelmointiympäristöön ja sillä käytettäviin testaustyökaluihin co♻e ja Test Manager. Työkaluun tutustumien alkoi tekemällä sillä yksinkertaisia testejä ja jatkui suunnittelemalla tavanomaisia yksikkötestejä erilaisia testaustekniikoita käyttäen.

Työn lopputuloksena todennettiin co♻e-yksikkötestaustyökalun soveltuvan kirjastoprojektien testaamiseen. Työn aikana tehtiin esimerkkitestausprojekti, jota voidaan käyttää myöhemmin yrityksessä perehdytyksessä.

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Degree programme: Automation Engineering

Specialisation: Machine Automation

Author: Saara Ala-Korte

Title of thesis: Unit test tool for the Epec library projects

Supervisor: Jyri Lehto

Year: 2023 Number of pages: 37 Number of appendices:

The objective of the thesis was to investigate the suitability of co♻e unit test tool for the library projects of Epec in customer projects. There was another test tool for testing with CODESYS programs, but that test tool required a license that was subject to a fee. The test tool was not applicable for all customer projects because of the license fee. The intention of the thesis was to investigate if the test tool can be replaced with the co♻e unit test tool.

In the beginning the thesis examined software testing in general. The studied topics were test levels, test types and their various techniques. The thesis also examined the CODESYS programming environment and its test tools, co♻e and Test Manager. Familiarization with co♻e was started by making simple tests and continued by making common unit tests by using various test techniques.

As the result of the thesis Co♻e unit test tool was found suitable for testing the library projects. An example project was realized during the thesis. The example project can be used later for the introduction of the test tool.

SISÄLTÖ

Kuva-, kuvio- ja taulukkoluettelo

Käytetyt termit ja lyhenteet

PDU	Tehonjakeluyksikkö (eng. power distribution unit), jonka tarkoitus on jakaa teho sen lähtöjen kautta siihen yhdistettyihin komponentteihin.
CI/CD Pipeline	Jatkuva integraatio (eng. continous integration) ja jatkuva toimitus (eng. continous delivery) pipeline on työkalu menetelmään, jolla pyritään pitämään sovellus kääntymis- sekä julkaisukelpoisena koko ohjelmistokehityksen elinkaaren ajan.
Testitapaus	Testitapauksella (eng. test case) todennetaan sovelluksen oikeellinen toiminta testaamalla toiminnallisuuksia. Testitapauksessa kerrotaan, miten testi tehdään ja mikä on testin odotettu lopputulos.
Käyttäjätarina	(eng. user case) Käyttäjän näkökulmasta tehty toiminnallisuuden kuvaus.
Käyttötapaus	(eng. use story) Asian toiminnallinen kuvaus tapahtumista vaihe vaiheelta.

1 JOHDANTO

1.1 Työn tausta

Ohjelmistotestaus on osa ohjelmistosuunnittelun elinkaarta. Testauksella pyritään varmentamaan laatu sekä vaatimusten mukainen toiminta. Työn aihe tuli Epec Oy:n tarpeesta löytää soveltuva yksikkötestaustyökalu asiakasprojektien ohjelmistokehitystyöhön. Yksikkötestauksella pyritään löytämään ohjelmavirheet kehitystyön alkuvaiheessa, jolloin niistä ei seuraa isompia virheitä myöhemmässä vaiheessa.

CODESYS Test Manager -työkalu soveltuu hyvin projektien yksikkötestaukseen, mutta sen käyttö vaatii maksullisen lisenssin. Tämän vuoksi sen käyttö kaikissa asiakasprojekteissa ei ole mahdollista. Työn tarkoituksena oli selvittää, löytyykö Test Manager -työkalulle vaihtoehtoista työkalua yksikkötestien implementointiin ja ajamiseen.

1.2 Työn tavoite

Työn tavoitteena oli selvittää, soveltuuko co♻e-yksikkötestaustyökalu Epecin projektikirjastojen testaamiseen. Projektikirjastoja kehitetään CODESYS-pohjaisiin sovelluksiin. Työkalua voisi käyttää kirjastokoodin testaamiseen ajamalla koodi testisovelluksessa. Testisovelluksella saataisiin nopea palaute koodin oikeellisesta toiminnasta, kun kirjastoprojektiin on tullut muutoksia.

1.3 Työn rakenne

Johdannossa esitellään työn tausta ja tavoitteet sekä esitellään opinnäytetyön toimeksiantajana toiminut yritys Epec Oy. Johdannon jälkeen kerrotaan ohjelmistotestauksesta. Ohjelmistotestauksesta esitellään V-malli, ohjelmistotestaustyypit ja ohjelmistotestauksen tasot. Sen jälkeen siirrytään CODESYS-ohjelmointiympäristön esittelyyn. Esittelyssä kerrotaan myös opinnäytetyössä käytetystä co♻e-testaustyökalusta

ja CODESYS Test Manager -työkalusta. Tämän jälkeen kerrotaan työn kulusta lyhyesti ja esitellään vaihe vaiheelta esimerkkiprojekti ja siihen tehtyjen muutoksien vaikutukset.

Yhteenveto ja pohdinta luvussa kerrataan työn tavoitteet ja verrataan lopputulosta niihin. Luvussa kerrotaan myös työn aikana tulleet havainnot sekä pohditaan jatkokehitystä.

1.4 Epec Oy

Opinnäytetyön toimeksiantajana oli Epec Oy. Epec Oy perustettiin vuonna 1978 (Epec, i.a-a). Epec Oy on valmistava järjestelmätoimittaja, joka on erikoistunut hyötyajoneuvojen ja työkoneiden sulautettuihin järjestelmiin. Yritysesittelyssä (Epec, sisäinen tietolähde, 2023) kerrotaan, että yrityksen tuotantotilat sijaitsevat päätoimipisteessään Seinäjoella. Valmistettavia tuotteita ovat mm. kuvassa 1 olevat ohjausyksiköt ja näytöt, joista yksi on kuvassa 2. Yritys valmistavaa myös autonomisia ja avustavia järjestelmiä, sähkönjakeluyksiköitä (PDU, eng. power distribution unit), telematiikkayksiköitä ja akunhallintajärjestelmiä. Toimialaan kuuluvat myös ohjelmistokehitykseen käytettävät ohjelmistot. Yritys myös kehittää ohjausjärjestelmiä asiakasyrityksille. Epec Oy:n asiakkaita ovat mm. Sandvik, Metso ja Tana.

Kuva 1. Epecin ohjausyksiköitä (Epec, i.a.-b).

Kuva 2. Epecin 6107-näyttö (Epec, i.a.-b).

Epec Oy on osa Ponsse Group -konsernia (Epec, i.a.-a). Yritysesittelyn (Epec, sisäinen tietolähde, 2023) mukaan Epec Oy on toiminut metsäkoneyhtiö Ponssen tytäryhtiönä vuodesta 2004 lähtien. Epec Oy valmistaa myös Ponsselle sähköisiä järjestelmiä ja laitteita. Yksi esimerkki Ponssen ja Epecin yhteistyönä syntyneestä innovaatiosta on kuvassa 3 oleva sähkökäyttöinen metsäkone EV1 (Ponsse, 2022). EV1-konseptikoneessa kaikki ajovoimansiirtoon käytettävä energia tulee akuilta. Koneen akut latautuvat polttomoottorilla. Sähköisessä voimansiirtoratkaisussa käytetään Epecin PDU:ta ja HCU:ta. PDU:lla voidaan mahdollistaa sähkömoottorien, eri toimilaitteiden ja akkujen integroituminen järjestelmään. HCU:lla eli hybridiohjausyksiköllä saadaan mm. ohjattua järjestelmän sähköisen voimalinjan toimintaa ja energiakulusta.

Kuva 3. Ponssen sähkökäyttöinen EV1-metsäkone (Ponsse, 2022).

Yritysesittelyssä (Epec, sisäinen tietolähde, 2023) kerrotaan Seinäjoen toimipisteen lisäksi muiden toimipisteiden sijaitsevan Tampereella, Turussa, Kuopiossa, Shanghaissa ja Milwaukeessa. Vuoden 2023 alussa työntekijöiden lukumäärä on noin 190. Henkilöstömäärä on kasvanut lähes 70 % vuosina 2019–2022 (Kylä-Kaila, 2022). Seinäjoelle rakennetaan uusia toimitiloja. Kuvassa 4 olevassa uudessa älykkäässä tehtaassa pyritään käytön aikaiseen hiilineutraaliuteen. Yksi ratkaisuista siihen on katolle asennettavat aurinkopaneelit. Päätoimipiste ja tuotanto siirtyvät uusiin toimitiloihin vuoden 2023 lopussa.

Kuva 4. Seinäjoelle rakentuva uusi tehdas (Ponsse, 2022).

2 OHJELMISTOTESTAUS OSANA LAADUN VARMISTUSTA

Ohjelmistotestauksen tarkoitus on testata sovellusta ohjelmistokehityksen aikana ennen kuin sovellus siirtyy loppukäyttäjille käytettäväksi (Homès, 2012, s. 1). Sovellusta testattaessa voi tulla esiin ohjelmistovirheitä, jotka voidaan korjata vielä varhaisessa vaiheessa kehitystä (ISTQB, 2012, s. 14, 16). Kun mahdolliset ohjelmistovirheet löydetään testaamalla, pystytään tarjoamaan loppukäyttäjälle tuote tai sovellus, joka jo valmiiksi vastaa sille asetettuja laatuvaatimuksia ja toimii halutulla tavalla. Jos viat löytyvät myöhemmin, siitä voi aiheutua asiakastyytymättömyyttä ja muita haittavaikutuksia.

Vika ohjelmistokoodissa voi aiheuttaa pahimmassa tapauksessa suuria menetyksiä. Vuonna 1996 4. kesäkuuta avaruusraketti Ariane 5 laukaistiin lentoon (ESA-CNES, 1996, s. 12). Ohjelmistovian vuoksi raketti suistui lentoraiteeltaan ja räjähti noin 40 sekunnin kuluttua laukaisusta. Tapaturman onnettomuusraportissa (ESA-CNES, 1996, s. 12) kerrotaan, että vika olisi voinut löytyä aiemmin, jos olisi panostettu riittävään analysointiin ja testaamiseen kehitysvaiheessa.

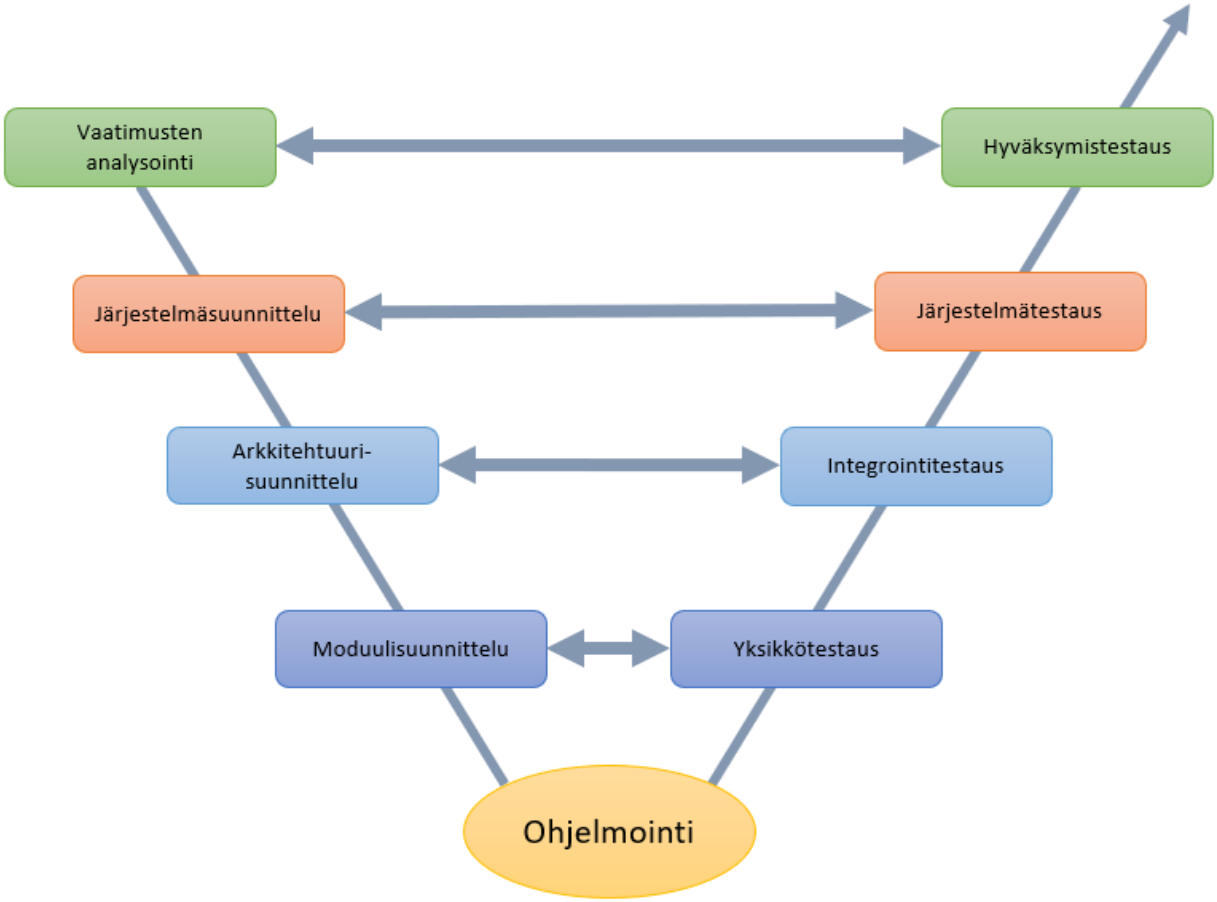
Testauksessa tavoitteena on löytää viat mahdollisimman varhaisessa vaiheessa ohjelmistokehitystä (ISTQB, 2012, s. 16, 46). Yksi ohjelmavirhe voi myöhemmin aiheuttaa useamman ohjelmavirheen. Mitä aikaisemmin vika löytyy, sitä vähemmän sen korjaaminen vaatii kustannuksia. Testaamisella pyritään varmistamaan, että sovellus toimii sille asetettujen vaatimusten mukaisesti sekä käyttäjien ja muiden sidosryhmien odotusten mukaisesti. Testaamisella pyritään myös varmentamaan laatuvaatimusten mukainen toiminta. Joskus testaamisella varmistetaan standardien mukainen toiminta.

Testaamisen suorittaa pääosin joku muu kuin itse ohjelmistokehittäjä (Myers ym., 2012, s. 14). Jos ohjelmistokehittäjä testaisi oman koodinsa, saattaisi jäädä osa virheistä huomaamatta. Omaa koodiaan testatessa helposti voi ikään kuin sokaistua omalle koodilleen samalla tavalla kuin kirjoittaja itse kirjoittamalleen tekstille. Voi myös olla hankalaa katsoa omaa koodiaan erilaisella tavalla kriittisin silmin, kun on sen jo tehnyt aiemmin suunnittelu- ja toteutusvaiheessa. Riskinä on myös, että ohjelmistokehittäjä ymmärtää jonkun asian väärin kehitysvaiheessa ja suorittaa testit samalla ymmärryksellä, jolloin väärinymmärrys jää huomaamatta.

2.1 V-malli

Ohjelmistokehitys sisältää paljon erilaisia työvaiheita, jotka muodostavat elinkaarimallin (ISTQB, 2018, s. 28). Yksi mahdollisista elinkaarimalleista on V-malli. V-mallille ominaista on se, että testaus yhdentyy ohjelmistokehityksen eri vaiheisiin koko ohjelmistokehitysprosessin aikana. V-mallissa painotus on testauksen varhaisessa vaiheessa ja kehitysprosessin edetessä testaustaso on yhtenäinen kehittämistoiminnan kanssa.

V-mallissa toteutus etenee vaihe vaiheelta (W3schools of Technology, i.a.). Seuraava vaihe alkaa, kun edellinen on lopetettu. Työvaihe voidaan lopettaa, kun sille asetetut testit on läpäisty. Työvaiheet toteutetaan kuvion 1 mukaisessa järjestyksessä.



Kuvio 1. V-malli (soveltaen W3schools of Technology, i.a.).

2.2 Testaustyypit

Tässä luvussa esitellään testaustyypit white-box, black-box, funktionaalinen ja ei-funktionaalinen testaus. Mikään testaustyyppi ei ole sidottuna vain yhteen tasoon, vaan niitä voidaan käyttää monella eri testaustasolla (ISTQB, 2018, s. 39, 42). Testaustyyppi määritellään sen mukaan, mihin ominaisuuteen, alueeseen tai asiaan testauksessa keskitytään.

2.2.1 Funktionaalinen testaus

Funktionaalisessa testauksessa keskitytään siihen, millä tavalla koodin toiminnallisuuden tai ominaisuuden kuuluisi käyttäytyä (ISTQB, 2018, s. 39) Testitapaukset tehdään koodille asetettujen vaatimusten pohjalta. Vaatimukset voidaan määrittää dokumentoiduista

toiminnallisista määrittelyistä, käyttötapauksista (usecase) ja käyttäjätarinoista (user story). Testitapaukset suunnitellaan selvittämällä vaatimuksista, minkälainen toiminnallisuus on ja miten sen odotetaan käyttäytyvän. Testitapauksia suunnitellessa voi myös tulla ilmi puutteita ja epäkohtia sovelluksen vaatimusten määrittelyissä (Pezzè ym., 2008, s. 162). Chopra (2018, s. 241) painottaa kirjassaan, että hyvien vaatimusten ja testitapausten taustalla on hyvin toteutettu yhteistyö asiakkaan tai sidosryhmän kanssa sekä ymmärrys tuotteesta.

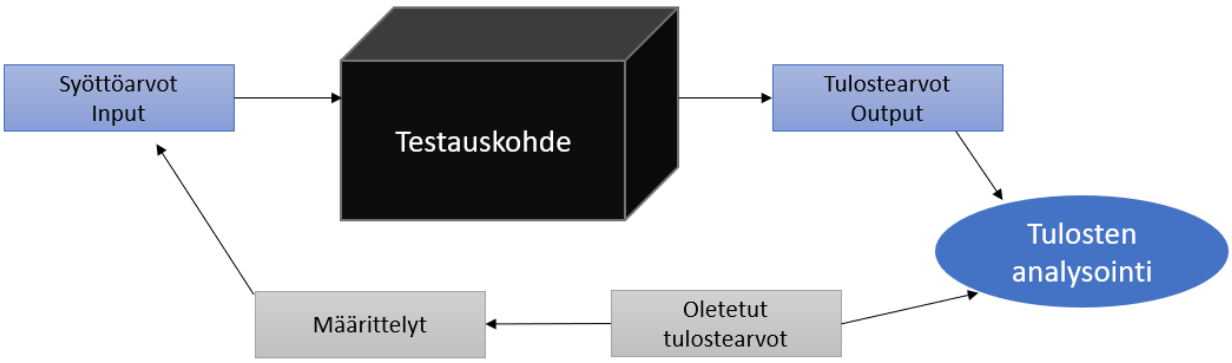
2.2.2 Ei-funktionaalinen testaus

Ei-funktionaalisessa testauksessa arvioidaan sovelluksen käytettävyyttä keskittyen suorituskykyyn, turvallisuuteen ja käytettävyyteen. Tarkoituksena on testata, kuinka hyvin testauskohde käyttäytyy (ISTQB, 2018, s. 40). Ei-funktionaalisessa testauksessa huomioidaan myös laadulliset tekijät, kuten luotettavuus ja skaalautuvuus (Chopra, 2018, s. 241).

Yksi esimerkki ei-funktionaalisesta testauksesta on skaalautuvuustestaus. Skaalautuvuustesteillä voidaan muun muassa testata, kuinka suuri määrä käyttäjiä voi käyttää samaa verkkosovellusta tai kuinka kauan sovelluksella kestää käsitellä suurta datamäärää (Hamilton, 2022b).

2.2.3 Black-box

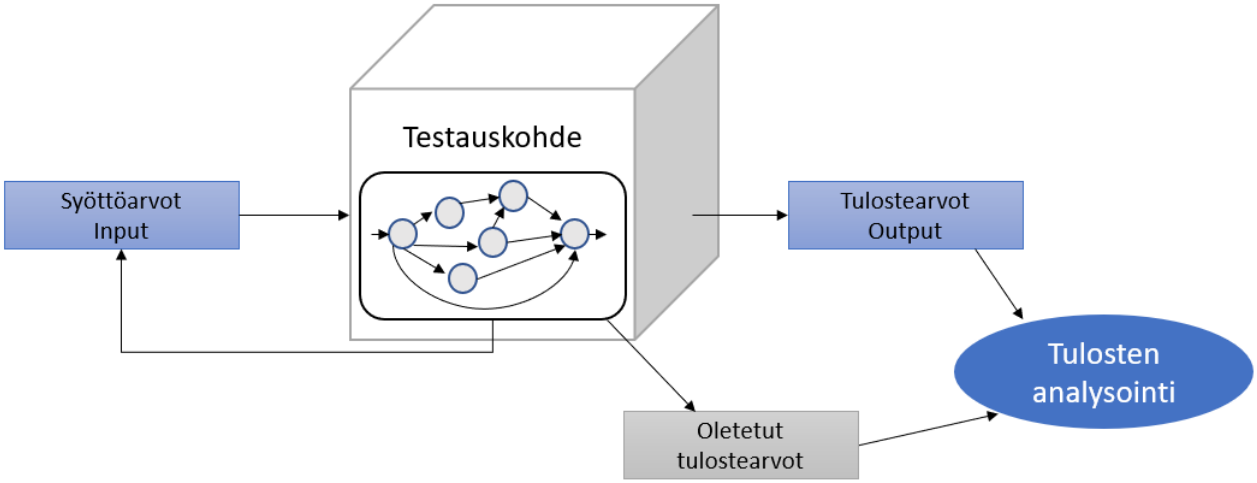
Black-box-testausmenetelmässä tutkitaan kohteen tulostearvoja (output) antamalla erilaisia syöttöarvoja (input) (Myers ym., 2012, s. 8–9). Testauskohde on ikään kuin ”musta laatikko”, jolloin ei keskitytä sisäisiin toiminnallisuuksiin tai niistä ei ole tietoa. Testitapaukset luodaan määrittelyjen perusteella. Tavoitteena on varmistaa kohteen toiminnallisuuksien oikeellisuus vertaamalla tulostearvoja oletettuihin tulostearvoihin. Kuviossa 2 on havainnollistettu black-box-menetelmä.



Kuvio 2. Black-box-testausmenetelmän toiminnallisuus

2.2.4 White-box

White-box-menetelmä poikkeaa black-box-testauksesta siten, että testit perustuvat kohteen sisäisiin ominaisuuksiin ja rakenteeseen (Myers ym., 2012, s. 10). Tarkoituksena on antaa erilaisia syöttöarvoja sen perusteella, minkälaisia mahdollisia suorituspolkuja on koodin sisällä. Toisin kuin black-box-menetelmässä white-box-menetelmässä koodi ja sen rakenne on nähtävillä. Kuviossa 3 on havainnollistettu white-box-menetelmä.

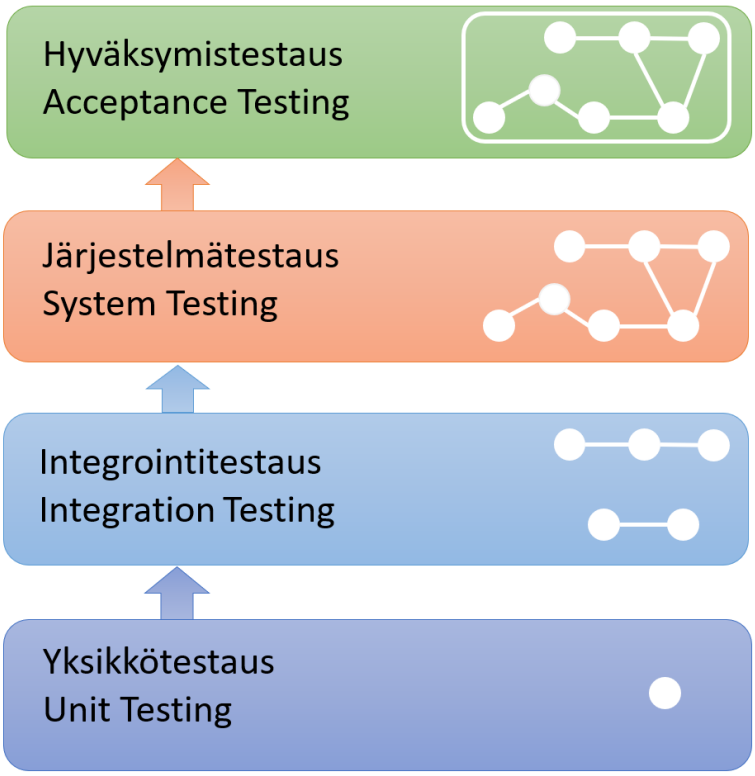


Kuvio 3. White-box-testausmenetelmä (soveltaen Myers ym., 2012, s. 11).

2.3 Ohjelmistotestauksen tasot

Ohjelmistokehityksen elinkaari sisältää erilaisia vaiheita, joiden mukaisesti myös testausta suoritetaan eri tasoilla. Jokaisessa tasossa testausprosessi toteutetaan sille sopivin menetelmin ja tavoitteiden mukaisesti (ISTQB, 2018, s. 30).

Tässä kappaleessa esitellään neljä ohjelmistotestauksen tasoa: yksikkötestaus, integrointitestaus, järjestelmätestaus sekä hyväksymistestaus, joiden toiminnallisuutta on havainnollistettu kuviossa 4. Yksikkötestauksessa keskitytään ohjelmistokoodin pieniin osiin ja yksittäisiin toimintoihin (ISTQB, 2018, s. 31–32, 34, 36). Integrointitestauksessa keskitytään toiminnallisuuksien ja osien yhdistämiseen. Järjestelmätestauksessa testataan järjestelmän toiminnallisuutta kokonaisuutena. Hyväksymistestauksessa testataan, kuinka sovellus toimii sille tarkoitetussa käyttökohteessa. Tässä opinnäytetyössä testaustasona on yksikkötestaus.

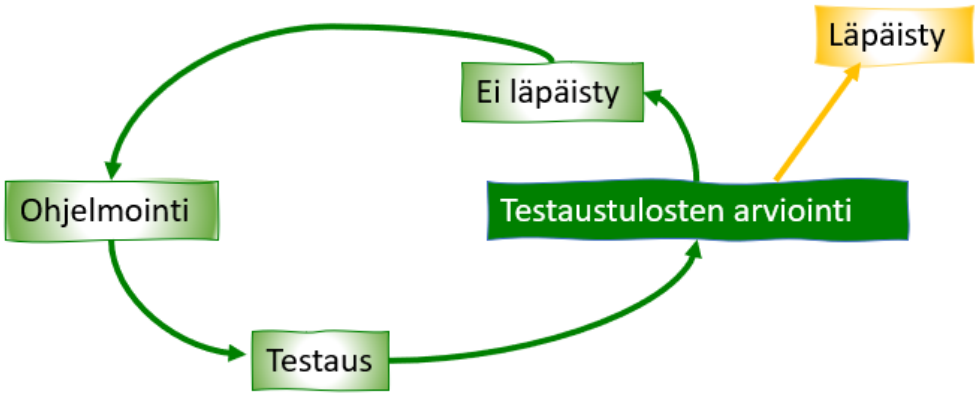


Kuvio 4. Testaustasot (soveltaen Chopra, 2018, s.227).

2.3.1 Yksikkötestaus

Yksikkötestauksessa eristetään yksittäinen osa tai toiminto koodista ja tutkitaan, toimiiko se oikealla tavalla (Hamilton, 2022a). Yksikkötestauksessa testattavana kohteena eli yksikkönä voivat olla koodin yksittäiset toiminnot, metodit, objektit, moduulit tai aliohjelmat. Yksikkötestauksessa käytetään yleensä white-box-menetelmää (Myers ym., 2012, s. 86).

Aiemmin luvussa 3 mainittiin, että koodin testaajan olisi lähtökohtaisesti hyvä olla joku muu kuin koodin tehnyt ohjelmistokehittäjä. Yksikkötestauksessa tästä usein poiketaan, jolloin ohjelmistokehittäjä suorittaa koodinsa testauksen itse (Hamilton, 2022a). Yksi yksikkötesteissä käytetty menetelmä on Test Driven Development eli TDD. TDD-kehitysprosessin eteneminen havainnollistetaan kuviossa 5. Ohjelmistokehittäjä luo yksikölle ensin testit, minkä jälkeen vasta luodaan yksikkö (Bender & McWhether, 2011, s. 8) Jos yksikkö ei läpäise testejä, siihen tehdään muutoksia. Muutoksien jälkeen yksikkö testataan. Ohjelmointi- ja testausprosessia jatketaan niin monta kertaa, kunnes yksikkö on läpäissyt sille asetetut testit.



Kuvio 5. Test Driven Development -prosessin kulku

Manuaalisen testaamisen lisäksi yksikkötestejä voi tehdä myös testaustyökalulla (Hamilton, 2022a). Yksi niistä on opinnäytetyössä tutkittu co♻e-yksikkötestaustyökalu.

Myers mainitsee kirjassaan kymmenen testauksen periaatetta, joista toisena on, että ohjelmoijan tulee välttää oman koodinsa testaamista (Myers ym., 2012, s. 14). Omaa koodia testattaessa voi olla hankalaa katsoa omaa jälkeään kriittisin silmin, minkä

testaaminen vaatii. Riskinä voi olla, että tekee testit sillä ajatuksella, että ne menevät läpi. Riskinä voi myös olla, että ohjelmistokehittäjä ei ole täysin ymmärtänyt koodille asetettuja vaatimuksia. Tällöin myös testit tehdään samalla ymmärryksellä, jolloin koodiin saattaa jäädä virheitä. On myös tavanomaista, että ohjelmistokehittäjä sokaistuu omalle koodilleen samalla tavalla kuin kirjoittaja kirjoitukselleen. Riskit voidaan minimoida katselmoimalla testitapaukset ennen testausta (mts. 109–110).

Testaamiseen on erilaisia tekniikoita. Testattavalle yksiköllе valitaan tekniikka sen ominaisuuksien mukaan. Esimerkiksi testattaessa ehtoa, jonka toteutuminen tapahtuu saadessaan arvon raja-arvojen väliltä, yksi sopiva tekniikka on boundary value analysis eli raja-arvojen analysointi (ISTQB, 2018, s. 58). Boundary value analysis on black-box-testaustekniikka, jossa annetaan syöttöarvoiksi raja-arvot ja juuri raja-arvojen ulkopuolelle jäävät arvot. Esimerkiksi jos ehdon toteutumiseksi raja-arvot ovat 5–20, testataan se syöttöarvoilla 4, 5, 20 ja 21.

2.3.2 Integrointitestaus

Integrointitestaus perustuu ohjelmakoodin yksiköiden ja osioiden yhteensopivuuteen (ISTQB, 2018, s. 32). Integrointitestauksessa tutkitaan käyttäytymistä, joka tapahtuu kahden tai useamman asian ollessa yhdistyneinä eli integroituneina toisiinsa. Tavoitteena on varmistaa laatu rajapintojen välillä. Tarkoituksena on myös todentaa suunnitelmien ja määrittelyjen mukainen käyttäytyminen funktionaalisten sekä ei-funktionaalisten toimintojen rajapinnoilla. Integrointitestaustasolla vikoja voi esiintyä rajapintojen lisäksi myös järjestelmissä sekä yksiköissä.

ISTQB-CTFL-Syllabus (2018, s. 32) esittelee kaksi integrointitestauksen eri tasoa: komponenttien integrointitestaus ja järjestelmien integrointitestaus. Komponenttien integrointitestauksessa tutkitaan vaihetta, jolloin eri yksiköt ovat liitoksissa toisiinsa. Tämä vaihe on vuorossa testattavan kohteen yksikkötestien jälkeen.

Järjestelmien integrointitestauksessa taas tutkitaan järjestelmien, ohjelmistopakettien ja mikropalveluiden välillä tapahtuvaa vuorovaikusta sekä rajapintoja. Lisäksi järjestelmien

integrointitestausta tehdään, kun järjestelmään liitetään jokin ulkoisen organisaation tarjoama ominaisuus tai palvelu, esimerkiksi verkkopalvelu.

2.3.3 Järjestelmätestaus

Järjestelmätestauksessa keskitytään koko järjestelmän tai tuotteen käyttäytymiseen ja suorituskykyyn (ISTQB, 2018, s. 34). Tarkoituksena on varmentaa järjestelmän laatu kokonaisuutena ja todentaa, ovatko funktionaaliset sekä ei-funktionaaliset ominaisuudet suunnitelmien ja määrittelyjen mukaisia.

Järjestelmätestauksen suorittavat yleensä testaajat, joilla on syvää tuntemusta sovelluksen toiminnallisuuksille asetetuista tavoitteista (ISTQB, 2018, s. 36). Testaus toteutetaan sillä ajatuksella, kuinka käyttäjät voisivat sovellusta tai tuotetta käyttää ja miten se toimisi ympäristössään. Testausta voidaan toteuttaa esimerkiksi simuloidussa ympäristössä. Myers mainitsee kirjassaan (2012, s. 122) erilaisia osa-alueita, joilla järjestelmätestausta toteutetaan. Osa-alueissa mainitaan mm. erilaisia suorituskyvyn testauskohteita sekä hyvin yleisiä sovelluksien toiminnallisuuksia, kuten esimerkiksi sovelluksen asentaminen ja uuden julkaisupaketin mukautuvuus edelliseen julkaisupakettiin. Suorituskykyä todennetaan esimerkiksi testaamalla suurien datamäärien siirtymistä ja käsittelyä prosesseissa.

2.3.4 Hyväksymistestaus

Hyväksymistestauksen tarkoituksena on varmentaa sovelluksen tai ohjausjärjestelmän oikeellisuus sille tarkoitetussa kohteessa ennen, kuin se siirretään tuotantoympäristöön (Hamilton, 2022c). Hyväksymistestauksen suorittaa loppukäyttäjä tai asiakas. Testauksessa keskitytään siihen, vastaako sovellus sille asetettuja asiakkaan määrittelemiä vaatimuksia (Myers ym., 2012, s. 131–132).

Hyväksymistestauksessa testaustapa perustuu käyttäjän toimintaan. Se testataan sillä ajatuksella, miten käyttäjä tulee käyttämään sovellusta. Haluttujen toiminnallisuuksien testaamisen lisäksi varmennetaan myös mm., onko sovelluksen käyttö selkeästi ymmärrettävää (Myers ym., 2012, s. 144–145). Varmistetaan esimerkiksi, onko

käyttöliittymän toiminta sellaista, mistä käyttäjälle tulee tapahtumien kulku ilmi oikein. Sovelluksen käyttäjälle ilmaistaan tapahtumien kulkua mm. pop-up-ilmoituksilla, latauksen etenemisen ikoneilla ja painikkeiden värien muutoksilla.

3 CODESYS OHJELMOINTIYMPÄRISTÖNÄ

3.1 CODESYS

CODESYS on ohjelmointiympäristö, joka on kehitetty IEC 61131-3 -standardin mukaan (CODESYS, 2023). Ohjelmointiympäristön on kehittänyt vuonna 1994 perustettu 3S-Smart Software Solutions GmbH. Vuodesta 2020 lähtien yhtiön nimi on ollut CODESYS GmbH. Ohjelmointiympäristöä käytetään kaikilla eri automaatioteollisuuden aloilla erilaisissa sovelluksissa.

CODESYS-ohjelmistoympäristössä sovelluksia voidaan kehittää kaikilla IEC 61131-3 -standardin ohjelmointikielillä (CODESYS, 2023). Ohjelmointikielet ovat function block diagram FBD, ladder diagram LD, instruction list IL, structured text ST ja sequential function chart SFC (CODESYS, 2022b; Huang ym., 2020, s. 4). Mainitut kielet ovat PLC-ohjelmoinnin kieliä. PLC-ohjelmointi on logiikkaohjelmointia, jota käytetään yleisimmin teollisuuden laitteissa (Huang ym., 2020, s. 4). CODESYS-ohjelmointiympäristö sisältää IEC 61131-3 -standardin työkaluja, jotka mahdollistavat mm. integrointiliitännät kaikkiin yleisiin kenttäväyliin, turvaratkaisuihin, liikeratatoimintoihin ja kommunikointikytkentöihin (CODESYS, 2022b).

3.2 Co♻e-testaustyökalu

Co♻e on CODESYS-ohjelmointiympäristöön kehitetty ilmainen yksikkötestaustyökalu (CODESYS, 2022a). Co♻e-työkalun ohjelmointikirjasto sisältää erilaisia funktioita testien suorittamiseen, analysoimiseen ja testaustulosten dokumentointiin. Co♻e käyttää Python scriptejä testien ajamiseen. Työkalu generoi testitulokset XML-tiedostolle.

Co♻e on myös mahdollista integroida CI/CD pipelinelle. CI tarkoittaa jatkuvaa integraatiota ja CD tarkoittaa jatkuvaa toimitusta (continuous delivery) (Microsoft, 2023). Co♻e-testaustyökalun käyttöönotto CI/CD pipelinelle mahdollistaisi automaattisesti testien ajamisen sovellusmuutoksien jälkeen. Tällöin saataisiin nopea suora palaute siitä, onko koodimuutoksilla vaikutusta sovelluksen toimintaan.

3.3 CODESYS Test Manager

CODESYS Test Manager on työkalu, jolla voidaan ohjelmoida ja suorittaa automaatiotestejä kaikenlaisille CODESYS-ohjelmointiympäristössä kehitetyille sovelluksille ja ohjelmointikirjastoille (CODESYS, 2022e). Test manager -työkalulla voidaan ajaa automaattitestejä.

Ohjelman käyttö vaatii maksullisen lisenssin, joka tulee uusia vuosittain (CODESYS, 2022c). Lisenssi sisältää Test managerin lisäksi myös muita sovelluskehityksen työkaluja, kuten esimerkiksi työkalun CODESYS static analysis. Static analysis -työkalulla on mahdollista löytää sovelluksesta vikoja jo ennen testien suorittamista (CODESYS, 2022d). Sen toiminta perustuu aiemmin määriteltyihin sääntöihin, joita verrataan koodiin. Kun koodista löytyy asia, joka ei vastaa sääntöjä, työkalu ilmoittaa, missä se sijaitsee ja miksi se ei vastaa sääntöjä. Määriteltyjä sääntöjä voivat olla esimerkiksi muuttujien nimet ja datatyypit.

4 YKSIKKÖTESTAUS KIRJASTOPROJEKTILLE

4.1 Työn kulku

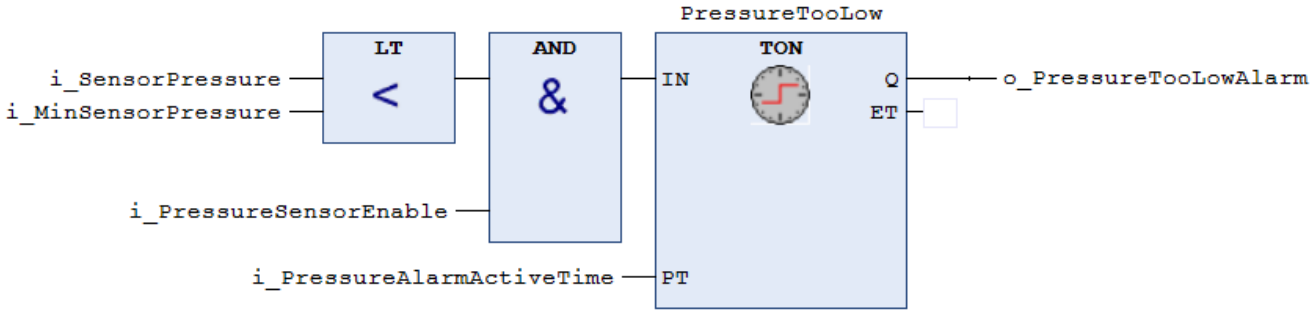
Työ alkoi tutustumalla co♻e-työkaluun ja testaamalla sen käyttöä CODESYS-ohjelmointiympäristössä. Kun työkalu oli ladattu ja lisätty CODESYS-ympäristöön, sen käyttöä kokeiltiin hyvin yksinkertaisilla funktioilla. Näitä yksinkertaisia testejä kokeiltiin ajaa työkalun molemmilla vaihtoehtoisilla testien ajofunktioilla. Co♻e-testaustyökalulla on mahdollista ajaa testitapaukset joko rinnan tai sekvenssissä. Sekvenssissä seuraava testi alkaa edellisen päätyttyä. Ajattaessa rinnan kaikki testit suoritetaan samanaikaisesti. Työkalua kokeiltiin myös kirjastolle, joka oli luotu Epecin moduulille.

Käyttöön perehtymisen jälkeen alettiin suunnittelemaan yksikkötestejä, joita työkalulla voitaisiin mahdollisesti kirjastoprojektien testaamiseen käyttää. Suunnittelu alkoi perehtymällä testaustekniikoihin. Suunnitteluvaiheessa yritys antoi perehdytyksen CODESYS Test Manager -työkaluun. Vaikka työkalua ei opinnäytetyössä käytetä, perehdytyksestä oli hyötyä testien suunnittelussa. Perehdytyksessä sai näkemystä siihen, kuinka työkalua käytetään, minkälaisia testejä sillä ajetaan ja millä tavoin.

Erilaisien yksikkötestauskokeilujen jälkeen luotiin esimerkkiprojekti, jota voidaan käyttää myöhemmin yrityksessä esimerkiksi perehtymiseen. Esimerkkiprojekti sisältää kirjastoprojektin ja sen testisovelluksen. Testaussovelluksen toiminnan kulku käydään läpi myöhemmissä kappaleissa. Kirjastoprojektiin luotiin ajastimen sisältämä painehälytysfunktio. Painehälytysfunktio luotiin siksi, että sille on mahdollista suorittaa erilaisia testejä ja funktio vastaa kirjastoprojektien tyypillisiä ominaisuuksia.

4.2 Esimerkkikohteen esittely

Kirjastoprojektin esimerkkikohteena on funktio, jossa on kuvion 6 mukainen hälytys, joka aktivoituu paineen alittaessa minimiarvon. Tarkoituksena on, että paineanturilla seurataan painetietoja. Jos paineanturin käyttöönotto on sallittu ja anturin mittaama paine on alle minimiarvon hälytyksen aktivointiajan, hälytys alhaisesta paineesta aktivoituu.



Kuvio 6. Alhaisen paineen hälytys

4.3 Kirjastoprojektin testaus testaussovelluksessa

Kirjastoprojektia testataan testaussovelluksessa, johon on lisätty kirjastoprojektin lisäksi co♻e-kirjasto. Sovellus sisältää pääohjelman lisäksi kuvion 7 toimilohkon FB_PressureTooLowNormalFunction_Test, jossa suoritetaan yksikkötestit esimerkkikohteelle. Toimilohko on laajennettu CoUnit.FB_TestSuite-toimintolohkolla, joka mahdollistaa co♻e-työkalun ominaisuuksien käytön.

Esimerkkikohteen testitapauksien prosessit etenevät lähes samalla kaavalla. Tämän vuoksi testaustoimilohkolle tehtiin testausmetodi PressureTooLowAlarm_Test, jossa testausprosessi käydään alusta loppuun käyttäen sille asetettuja syöttöarvoja. Testausmetodia toistetaan silmukassa, jossa jokaisella kierroksella ajetaan seuraava testitapaus taulukosta, joka alustetaan testaustoimilohkossa.

FB_PressureTooLowNormalFunction_Test

```
FUNCTION_BLOCK FB_PressureTooLowNormalFunction_Test EXTENDS FB_TestSuite
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
    PressureSensorLow: UnitTestLibrary.PressureSensorDiag;
    TestCases: ARRAY [1..Number_of_cases] OF PressureTooLowNormalFunction_STRUCT := [
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality1', FailedTestMessage:='Input valu
        TimerDelay:= T#2S, SensorPressure:=9, MinSensorPressure:=10, PressureSensorEnable:=TRUE,
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality2', FailedTestMessage:='Alarm is a
```

```
// Run test array
FOR i:=1 TO Number_of_cases BY 1 DO
    PressureTooLowAlarm_Test(
                            i_TestName:=TestCases[i].TestName,
                            i_FailedTestMessage:=TestCases[i].FailedTestMessage,
                            i_Expected:=TestCases[i].Expected,
                            i_TimerDelay:= TestCases[i].TimerDelay,
                            i_SensorPressure:=TestCases[i].SensorPressure,
                            i_MinSensorPressure:=TestCases[i].MinSensorPressure,
                            i_PressureSensorEnable:=TestCases[i].PressureSensorEnable,
                            i_PressureAlarmActiveTime:=TestCases[i].PressureAlarmActiveTime);
END_FOR
```

Kuvio 7. Esimerkkikohteen testaaminen testaussovelluksessa

4.3.1 Testimetodi

Kuvion 8 metodissa PressureTooLowAlarm_Test testi alkaa testin nimeämisellä. Nimeämisen jälkeen logiin kirjoitetaan statustiedoksi, että testi aloitetaan.

Funktio testataan antamalla sille syöttöarvot. Koska paineenhälytysfunktiossa käytetään ajastinta, sen testaamista varten myös metodissa on ajastin. Ajastimella voidaan testata, tapahtuuko asia oletetussa ajassa.

Kun ajastimen aika on tullut päätökseen, analysoidaan tulostearvot. Tulostearvot analysoidaan käyttämällä metodia AssertEquals. AssertEquals-metodilla verrataan tulostearvoa odotettuun tulostearvoon. Jos tulostearvo ei vastaa odotettua tulosarvoa, kirjoitetaan logiin tieto epäonnistuneesta testistä. Tiedossa lukee, miksi testi on epäonnistunut. Tämän jälkeen logiin kirjoitetaan tieto testinlopetuksesta ja testi saatetaan päätökseen funktiolla TEST_FINISHED.

```
IF TEST_ORDERED(i_TestName) THEN
    IF i_TestName <> Tmpstr THEN
        Tmpstr:=i_TestName;
        // Write test name to the log
        WRITE_DEVICE_LOG(1, 'Test start: %s', CoUnit.GVL_coUnit.CurrentTestNameBeingCalled);
        PressureSensorLow(i_PressureSensorEnable:=FALSE);
    END_IF
    PressureSensorLow(i_SensorPressure:=i_SensorPressure,
                      i_MinSensorPressure:=i_MinSensorPressure,
                      i_PressureSensorEnable:=i_PressureSensorEnable,
                      i_PressureAlarmActiveTime:=i_PressureAlarmActiveTime,
                      o_PressureTooLowAlarm=>PressureTooLowAlarm);

    // Delay to check the test results
    ResultsDelay(IN:=TRUE, PT:=i_TimerDelay);
    IF (ResultsDelay.Q) THEN // Check results after timer
        ResultsDelay(IN:=FALSE);
        AssertEquals( Expected:=i_Expected,
            Actual:=PressureTooLowAlarm,
            Message:=i_FailedTestMessage);
            WRITE_DEVICE_LOG(1, 'Test finished: %s' , coUnit.GVL_coUnit.CurrentTestNameBeingCalled);
            TEST_FINISHED();
```

Kuvio 8. Testimetodin aloitus ja testaustulosten analysointi ajastimen jälkeen

Metodin IF-lause jatkuu testaustulosten analysoinnilla. Kuviossa 9 testataan tilannetta, kun hälytys aktivoituu ennen ajastimen ajan päättymistä ja testin oletettuna tuloksena hälytyksen ei kuulu aktivoitua.

Seuraavaksi metodissa analysoidaan tilanne, kun hälytys on aktivoitunut ennen kuin ajastuksen aika on tullut päätökseen. Tällöin metodissa testi epäonnistuu ja virheviestiin kirjataan oletetusta tuloksesta jäljellä oleva aika hälytyksen aktivoimiseksi.

```
23
24              // If expected results is FALSE and alarm is activated
25              ELSIF i_Expected=FALSE AND NOT(ResultsDelay.Q) THEN
26                  IF i_Expected <> PressureTooLowAlarm THEN
27                      ResultsDelay(IN:=FALSE);
28                      AssertEquals( Expected:=i_Expected,
29                          Actual:=PressureTooLowAlarm,
30                          Message:='Alarm is activated!');
31                      WRITE_DEVICE_LOG(1, 'Test finished: %s' , coUnit.GVL_coUnit.CurrentTestNameBeingCalled);
32                      TEST_FINISHED();
33                  END_IF
34              ELSE
35              // If Alarm is activated before timer expired, test shall fail
36                  TimeLeft:=(i_TimerDelay)-(ResultsDelay.ET);
37                  EtString:=TIME_TO_STRING(TimeLeft);
38                  Failmessage:=CONCAT('Alarm is activated too early! Time left to the alarm: ', EtString);
39                  nExpected:=NOT(i_Expected);  // True to False
40                  IF PressureSensorLow.o_PressureTooLowAlarm=TRUE AND NOT(ResultsDelay.Q) THEN
41                      ResultsDelay(IN:=FALSE);
42                      AssertEquals( Expected:=nExpected,
43                          Actual:=PressureTooLowAlarm,
44                          Message:=Failmessage);
45                      Number:=NumberOfAssert.GetNumberOfArrayAssertsForTest(i_TestName);
46                      WRITE_DEVICE_LOG(1, 'Test finished: %s' , coUnit.GVL_coUnit.CurrentTestNameBeingCalled);
47                      TEST_FINISHED();
48                  END_IF
49      END_IF
50  END_IF
51
```

Kuvio 9. Ajastuksen analysointi testimetodissa

4.3.2 Testitapaukset esimerkkiprojektille

Testitapauksien kirjoitusta varten tehtiin kuviossa 10 oleva struct-taulukko. Kuviossa 11 testaussovelluksen alustuksessa kutsutaan taulukkoa syöttäen sen riveille testitapausten muuttujien arvot. Taulukko sisältää 11 testitapausta, joista kolme on tarkoituksella epäonnistuvia.

PressureTooLowNormalFunction_STRUCT

```
1   TYPE PressureTooLowNormalFunction_STRUCT :
2   STRUCT
3       TestName                    :   T_MaxString;
4       FailedTestMessage           :   T_MaxString; //Message when the test fails
5       Expected                    :   BOOL; //Expected result
6       TimerDelay                  :   TIME; //Delay to check the results
7       SensorPressure              :   UINT;
8       MinSensorPressure           :   UINT;
9       PressureSensorEnable        :   BOOL;
10      PressureAlarmActiveTime     :   TIME;
11  END_STRUCT
12  END_TYPE
```

Kuvio 10. Testaustaulukon rakenne

FB_PressureTooLowNormalFunction_Test

```
FUNCTION_BLOCK FB_PressureTooLowNormalFunction_Test EXTENDS FB_TestSuite
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR
VAR
    PressureSensorLow: UnitTestLibrary.PressureSensorDiag;
    TestCases: ARRAY [1..Number_of_cases] OF PressureTooLowNormalFunction_STRUCT := [
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality1',
        FailedTestMessage:='Input values does not activate alarm!', Expected:=TRUE, TimerDelay:= T#2S,
        SensorPressure:=8, MinSensorPressure:=10, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#2S),
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality2',
        FailedTestMessage:='Alarm is activated without sensor enable!', Expected:=FALSE, TimerDelay:= T#4S,
        SensorPressure:=9, MinSensorPressure:=10, PressureSensorEnable:=FALSE,PressureAlarmActiveTime:=T#2S),
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality3',
        FailedTestMessage:='Alarm activated with incorrect limit value!', Expected:=FALSE, TimerDelay:= T#3S,
        SensorPressure:=11, MinSensorPressure:=10, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#3S),
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality4',
        FailedTestMessage:='Alarm activated too early!', Expected:=FALSE, TimerDelay:= T#2S,
        SensorPressure:=9, MinSensorPressure:=10, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#3S),
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality5',
        FailedTestMessage:='Value is under minimum value and alarm is not activated!', Expected:=TRUE, TimerDelay:= T#3S,
        SensorPressure:=0, MinSensorPressure:=1, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#3S),
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality6',
        FailedTestMessage:='Pressure value is equal with minimum limit value!', Expected:=FALSE, TimerDelay:= T#5S,
        SensorPressure:=0, MinSensorPressure:=0, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#3S),
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality7',
        FailedTestMessage:='Pressure value is not under minimum limit value!', Expected:=FALSE, TimerDelay:= T#4S,
        SensorPressure:=1, MinSensorPressure:=0, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#3S),
        (TestName:='Test_PressureTooLowAlarm_ThisTestFail1',
        FailedTestMessage:='Alarm is not activated!', Expected:=TRUE, TimerDelay:= T#4S,
        SensorPressure:=0, MinSensorPressure:=1, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#5S),
        (TestName:='Test_PressureTooLowAlarm_BasicFunctionality8',
        FailedTestMessage:='Pressure does not active without timer!!', Expected:=TRUE, TimerDelay:= T#0S,
        SensorPressure:=0, MinSensorPressure:=1, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#0S),
        (TestName:='Test_PressureTooLowAlarm_ThisTestShouldFail2',
        FailedTestMessage:='Alarm is activated!', Expected:=FALSE, TimerDelay:= T#3S,
        SensorPressure:=0, MinSensorPressure:=1, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#2S),
        (TestName:='Test_PressureTooLowAlarm_ThisTestShouldFail3',
        FailedTestMessage:='Timer is activated too early!', Expected:=TRUE, TimerDelay:= T#3S,
        SensorPressure:=0, MinSensorPressure:=1, PressureSensorEnable:=TRUE, PressureAlarmActiveTime:=T#2S)
        ];
```

Kuvio 11. Testitapauksien muuttuja-arvot taulukossa

Testitapaukset on suunniteltu niille sopivimpia tekniikoita käyttäen. Yksi tekniikoista oli use case testing eli käyttötapaustestaus. Käyttötapaustestaus on funktionaalista testausta, jossa testit suunnitellaan käyttötapauksen perusteella (ISTQB, 2018, s. 59). Esimerkkikohteessa tekniikkaa käytetään suunnittelemalla testejä mukaillen kohteen perustoiminnallisuutta.

Boundary value analysis -tekniikkaa käytetään testattaessa kohteen numeraalisia arvoja. Yhtenä ehdoista alhaisen paineen hälytyksen aktivoimiseksi on paineen laskeminen alle minimirajan. Ehtoa testataan antamalla paineelle syöttöarvot käyttäen raja-arvona juuri minimirajan alittavaa arvoa. Boundary value analysis -tekniikkaa käytetään myös testattaessa aktivointiaikaa.

Taulukon tarkoituksella epäonnistuvat testit on toteutettu seuraavalla tavalla. Ensimmäisessä epäonnistuvassa testitapauksessa luetaan tulos liian aikaisin. Odotetuksi

tulokseksi on asetettu TRUE, eli hälytys on aktiivinen. Hälytyksen aktivointi on asetettu tapahtuvan viiden sekunnin viiveen jälkeen ja tulos luetaan neljän sekunnin kohdalla. Toisessa epäonnistuvassa testitapauksessa odotetuksi tulokseksi on asetettu FALSE, mikä on virheellinen. Kolmannessa epäonnistuvassa testitapauksessa odotettu tulos on TRUE, mutta se tapahtuu odotettua aiemmin.

4.3.3 Testien ajo sovelluksessa

Esimerkkikohde ja sen testimetodi sisältävät ajastimia, joiden vuoksi tulee olla varma siitä, että suoritus etenee suoraviivaisesti. Jotta päällekkäisyyksiä ei tapahtuisi, tulisi yksittäisten testien aloitusta ja lopetusta pystyä hallitsemaan. Siitä syystä sekvenssissä ajaminen on parempi ratkaisu verrattuna saman aikaiseen ajoon, kun tavoitteena on tehdä tarkkoja ja mahdollisimman oikeellisia testaustuloksia sovelluksessa.

Ajettaessa testitapauksia sekvenssissä seuraava testi alkaa edellisen loputtua. Testitapaus aloitetaan asettamalla sen nimi CoUnit.TEST_ORDERED-funktiolle. Tällä funktiolla valitaan seuraava testitapaus, jos edellinen testi on saatu päätökseen. Edellinen testi tulkitaan päättyneeksi, kun sen testauksen jälkeen on suoritettu CoUnit.TEST_FINISHED-funktio. Kuviossa 12 pääohjelma ajaa funktion CoUnit.RUN_IN_SEQUENCE, jolloin testitapaukset ajetaan sekvenssissä.

```
1  PROGRAM MAIN
2  VAR
3      PressureLowAlarmTest: FB_PressureTooLowNormalFunction_Test;
4  END_VAR
5
```

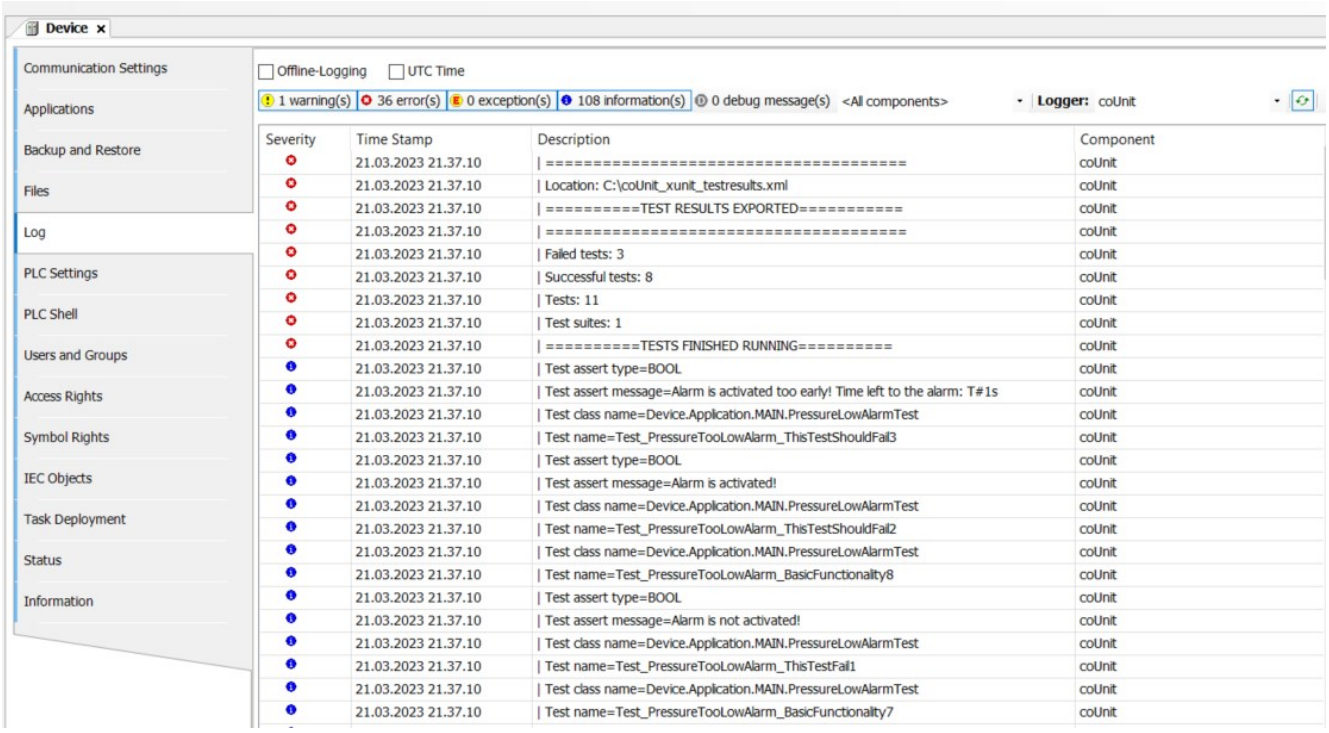
```
1  CoUnit.RUN_IN_SEQUENCE();
2
```

Kuvio 12. Testien ajaminen pääohjelmassa

Testitapausten ajoa voidaan seurata CODESYS Device Log -näkymästä. Logiin tulee ajon aikana tietoa, koska testi alkaa ja koska se päättyy. Lopuksi logiin tulee testien tulostiedot, kuten kuviossa 13.

Co♻e-kirjasto luo myös XML-muotoisen tiedoston. Kuviossa 14 oleva tiedosto sisältää testisovelluksen ajamat testitulokset. Testitapauksista kaikki eivät menneet läpi

onnistuneesti. Epäonnistuneista testitapauksista tulee näkyviin niille taulukossa määritetyt selitteet, jotka kertovat syyn epäonnistuneelle testille. Viimeisellä testillä tämä selite ei ole sama kuin sille on taulukossa määritetty, koska sen tulos on odotettua ajastusta aiemmin. Selitteessä lukee syy ja se, kuinka paljon testin tulos on aikaansa edellä.



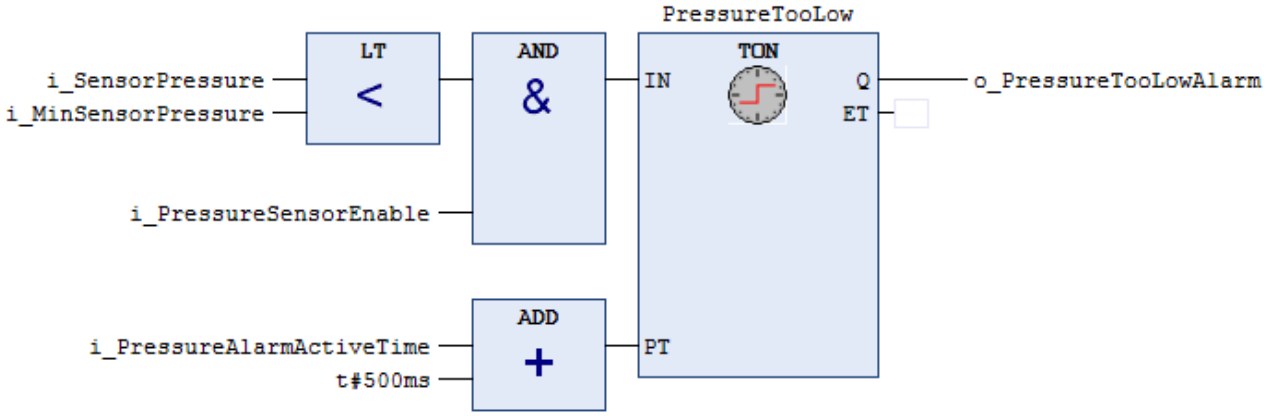
Kuvio 13. Ajonaikaisia logitietoja

```
<testsuites disabled="" failures="3" tests="8">
  <testsuite id="0" name="Device.Application.MAIN.PressureLowAlarmTest" tests="11" failures="3">
    <testcase name="Test_PressureTooLowAlarm_BasicFunctionality1" classname="Device.Application.MAIN.PressureLowAlarmTest"
    status="PASS"/>
    <testcase name="Test_PressureTooLowAlarm_BasicFunctionality2" classname="Device.Application.MAIN.PressureLowAlarmTest"
    status="PASS"/>
    <testcase name="Test_PressureTooLowAlarm_BasicFunctionality3" classname="Device.Application.MAIN.PressureLowAlarmTest"
    status="PASS"/>
    <testcase name="Test_PressureTooLowAlarm_BasicFunctionality4" classname="Device.Application.MAIN.PressureLowAlarmTest"
    status="PASS"/>
    <testcase name="Test_PressureTooLowAlarm_BasicFunctionality5" classname="Device.Application.MAIN.PressureLowAlarmTest"
    status="PASS"/>
    <testcase name="Test_PressureTooLowAlarm_BasicFunctionality6" classname="Device.Application.MAIN.PressureLowAlarmTest"
    status="PASS"/>
    <testcase name="Test_PressureTooLowAlarm_BasicFunctionality7" classname="Device.Application.MAIN.PressureLowAlarmTest"
    status="PASS"/>
    <testcase name="Test_PressureTooLowAlarm_ThisTestFail1" classname="Device.Application.MAIN.PressureLowAlarmTest" status="FAIL">
      <failure message="Alarm is not activated!" type="BOOL"/>
    </testcase>
    <testcase name="Test_PressureTooLowAlarm_BasicFunctionality8" classname="Device.Application.MAIN.PressureLowAlarmTest"
    status="PASS"/>
    <testcase name="Test_PressureTooLowAlarm_ThisTestShouldFail2" classname="Device.Application.MAIN.PressureLowAlarmTest" status="FAIL">
      <failure message="Alarm is activated!" type="BOOL"/>
    </testcase>
    <testcase name="Test_PressureTooLowAlarm_ThisTestShouldFail3" classname="Device.Application.MAIN.PressureLowAlarmTest" status="FAIL">
      <failure message="Alarm is activated too early! Time left to the alarm: T#1s" type="BOOL"/>
    </testcase>
  </testsuite>
</testsuites>
```

Kuvio 14. Testitulokset XML-tiedostossa

4.4 Muutos esimerkkikohteessa

Testataan testisovellusta muutetulle kirjastoprojektille ja katsotaan, millainen vaikutus sillä on testien tuloksiin. Kirjastoprojektissa on tehty muutoksia esimerkkinä olleelle funktiolle. Kuviossa 15 ajastimeen on lisätty 500 ms viivettä i_PressureAlarmActiveTime-tulon lisäksi.



Kuvio 15. Muutos esimerkkikohteessa

4.4.1 Testimetodien ajo muutoksen jälkeen

Esimerkkikohteen muutoksien jälkeen ajetaan testikoodi. Kuviossa 16 olevista testaustuloksista voidaan huomata, että pienellä koodimuutoksella voi olla selkeä vaikutus tuloksiin. Epäonnistuneiden testitulosten ilmetessä tulee miettiä, onko vika testitapauksessa vai testattavassa sovelluksessa. Testisovellus voi vaatia myös muutoksia sovelluskehityksen ohella.

```
▼<testsuites disabled="" failures="6" tests="5">
  ▼<testsuite id="0" name="Device.Application.MAIN.PressureLowAlarmTest" tests="11" failures="6">
    ▼<testcase name="Test_PressureTooLowAlarm_BasicFunctionality1" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="FAIL">
       <failure message="Input values does not activate alarm!" type="BOOL"/>
     </testcase>
     <testcase name="Test_PressureTooLowAlarm_BasicFunctionality2" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="PASS"/>
     <testcase name="Test_PressureTooLowAlarm_BasicFunctionality3" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="PASS"/>
     <testcase name="Test_PressureTooLowAlarm_BasicFunctionality4" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="PASS"/>
    ▼<testcase name="Test_PressureTooLowAlarm_BasicFunctionality5" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="FAIL">
       <failure message="Value is under minimum value and alarm is not activated!" type="BOOL"/>
     </testcase>
     <testcase name="Test_PressureTooLowAlarm_BasicFunctionality6" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="PASS"/>
     <testcase name="Test_PressureTooLowAlarm_BasicFunctionality7" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="PASS"/>
    ▼<testcase name="Test_PressureTooLowAlarm_ThisTestFail1" classname="Device.Application.MAIN.PressureLowAlarmTest" status="FAIL">
       <failure message="Alarm is not activated!" type="BOOL"/>
     </testcase>
    ▼<testcase name="Test_PressureTooLowAlarm_BasicFunctionality8" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="FAIL">
       <failure message="Pressure does not active without timer!!" type="BOOL"/>
     </testcase>
    ▼<testcase name="Test_PressureTooLowAlarm_ThisTestShouldFail2" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="FAIL">
       <failure message="Alarm is activated!" type="BOOL"/>
     </testcase>
    ▼<testcase name="Test_PressureTooLowAlarm_ThisTestShouldFail3" classname="Device.Application.MAIN.PressureLowAlarmTest"
     status="FAIL">
       <failure message="Alarm is activated too early! Time left to the alarm: T#498ms" type="BOOL"/>
     </testcase>
   </testsuite>
 </testsuites>
```

Kuvio 16. Testitulokset muutoksen jälkeen

4.4.2 Jatkuva integraatio ohjelmistokehityksen mukana

Suuria kokonaisuuksia sisältävät ohjausjärjestelmät rakentuvat pienistä osista. Osat voivat rakentua samanaikaisesti eri kehittäjien toimesta (Microsoft, 2022). Haasteena on, että koodimuutokset kumoavat toisensa. Jatkuva integraatio eli CI (eng. continuous integration) on versiohallintamenetelmä, jolla pystytään estämään muutosten kumoaminen ja jatkamaan ketterää kehitystä. Menetelmässä uudet muutokset lisätään versiohallintaan mahdollisimman nopeasti niiden valmistuttua. Jatkuvassa integraatiossa pyritään pitämään sovellus sellaisessa kunnossa, että sitä voidaan aina kääntää.

Kuten aiemmassa kappaleessa huomataan, muutos kirjastoprojektissa voi muuttaa testien tuloksia. Jatkuvaa integraatiota tulisi pitää myös yllä testisovelluksissa.

5 YHTEENVETO JA POHDINTA

Opinnäytetyön tarkoituksena ja tavoitteena oli selvittää, soveltuuko co♻e-yksikkötestaustyökalu Epecin kirjastoprojektien testaamiseen. Co♻e-yksikkötestaustyökalu soveltui kirjastoprojektien testaamiseen. Sen käyttöönotto ja testitapausten luominen oli helppoa. Yksikölle voidaan tehdä testimetodi, jota voidaan käyttää useampaan eri testitapaukseen asettamalla taulukosta metodiin syöttö- ja tuloarvot. Testikoodia yksikölle ei siis välttämättä tarvitse luoda kuin kerran. CODESYS-ohjelmointiympäristössä ajaminen ei onnistunut simulaatiossa.

Co♻e-työkalua käytettiin myös kirjastoprojektiin, joka oli luotu Epecin ohjausyksiköille. Kirjastoprojekti sisälsi käyttöjärjestelmäriippuvaisia muuttujia ja ominaisuuksia, joiden vuoksi testisovelluksen ajo ei onnistunut tietokoneella. Ajaminen vaati ohjausyksikön. Testisovelluksen ajaminen Epecin ohjausyksiköllä onnistui suoraviivaisesti. Sovellus suoritti yksikkötestin testitapaukset, ja tulosten raportointi onnistui odotetulla tavalla.

Käyttöjärjestelmäriippuvaista kirjastoa testatessa tehtiin havainto, joka on hyvä ottaa huomioon testaustyökalua käyttäessä. Testaussovellus voi vaatia paljon muistitilaa, koska sovelluksen lisäksi se sisältää co♻e-kirjaston ja kirjastoprojektin, jota sovelluksessa testataan. Joihinkin ohjausyksiköihin sitä ei voitu integroida riittävän muistitilan puuttumisen vuoksi. Testaussovelluksen kokoa voidaan pienentää pilkkomalla sovellusta pienempiin osiin ja tekemällä useampi testisovellus.

Työn tarkoituksena oli selvittää, voisiko co♻e-testaustyökalu korvata CODESYS Test Manager -testaustyökalun. Co♻e ja CODESYS Test Manager eroavat toisistaan siten, että Test Manager vaatii maksullisen lisenssin. Co♻e on saatavilla ilmaisena. Co♻e -testaustyökalulla voidaan luoda testejä samankaltaisesti black-box-tyyppisesti antamalla syöttö- ja tuloarvot taulukon riveille. Test Managerin käyttöä edellyttävä lisenssi sisältää myös muita työkaluja Test Managerin lisäksi. Co♻e-yksikkötestaustyökalu soveltuu yksikkötestaamiseen samalla tavalla, kuten Test Manager, jos lisenssin tarjoamia lisäominaisuuksia ei tarvita.

Tulevaisuudessa jatkokehityksenä testaustyökalu voitaisiin integroida osaksi sovelluskehitystä. Opinnäytetyössä luotiin esimerkkiprojekti, jota voidaan käyttää myöhemmin perehdyttämiseen projekteissa. Työkalua voitaisiin käyttää CI/CD pipelinen yhteydessä. Jatkuvalla integraatiolla pyritään pitämään versiohallinta mahdollisimman hyvin ajan tasalla päivittämällä muutokset usein. Jatkuvalla toimituksella pyritään pitämään sovellus versiohallinnassa aina julkaisukelpoisena. Automatisoitua prosessia voidaan toteuttaa CI/CD pipelinella, jolla voidaan kääntää ohjelmakoodia ja luoda julkaisupaketteja. Prosessiin voidaan liittää myös automaattitestausta. CI/CD pipeline säästää automatisoitujen ominaisuuksien vuoksi aikaa julkaisupaketin kasaamisessa sekä vähentää siinä tapahtuvien virheiden mahdollisuutta. Tulevaisuudessa olisi mielenkiintoista selvittää, olisiko co♻e-testaustyökalua mahdollista integroida CI/CD pipelinelle ja siten lisätä entistä enemmän sen laatuvarmuutta.

LÄHTEET

Bender, J., & McWhether, J. (2011) *Professional test-driven development with C# developing real world applications with TDD.* John Wiley & Sons.

Chopra, R. (2018). *Software Quality Assurance: A Self-Teaching Introduction.* Mercury Learning & Information.

CODESYS (2022a). *Co♻e: A unittest framework for CODESYS.* Haettu 28.3.2023, https://forge.codesys.com/lib/counit/home/Home/

CODESYS (2022b). *CODESYS Development System V3*. Haettu 28.3.2023, https://store.codesys.com/en/codesys.html

CODESYS (2022c). *CODESYS Professional Development Edition*. Haettu 28.3.2023, https://store.codesys.com/en/codesys-professional-developer-edition.html

CODESYS (2022d). *CODESYS Static Analysis.* Haettu 28.3.2023, https://store.codesys.com/en/codesys-static-analysis.html

CODESYS (2022e). *CODESYS Test Manager*. Haettu 28.3.2023, https://store.codesys.com/en/engineering/codesys-test-manager.html

CODESYS (2023). *THE COMPANY: CODESYS GROUP.* https://www.codesys.com/company.html

Epec. (i.a.-a). *Company*. https://epec.fi/company/

Epec. (i.a.-b). *Telematics and displays*. Haettu 17.4.2023, https://epec.fi/epec-oy-products/connectivity/telematics/

ESA-CNES. (1996). *Ariane 501 Inquiry Board report*. https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

Hamilton, T. (2022a). *Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE.* Guru99. https://www.guru99.com/unit-testing-guide.html

Hamilton, T. (2022b). *What is Scalability Testing? Learn with Example.* Guru99. https://www.guru99.com/scalability-testing.html

Hamilton, T. (2022c). *What is User Acceptanse Testing (UAT)? with Examples*. Guru99. https://www.guru99.com/user-acceptance-testing.html

Homès, B. (2012). *Fundamentals of software testing*. ISTE; John Wiley & Sons.

Huang, Y., Shi, J., Xiong, J., & Zhu, G. (2020). *User-Friendly Verification Approach for IEC 61131-3 PLC Programs*. https://mdpi-res.com/electronics/electronics-09-00572/article_deploy/electronics-09-00572-v2.pdf?version=1586328528

International Software Testing Qualifications Board (ISTQB). (2018). *Certified Tester Foundation Level*. https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFL_Syllabus_2018_v3.1.1.pdf

Kraeling, M., & Oshana, R. (2013). *Software Engineering for Embedded Systems – Methods, Practical Techniques, and Applications*. Elsevier.

Kylä-Kaila, J. (21.12.2022). *Ponsse-konserniin kuuluva teknologiayhtiö Epec rakentaa vastuullisen ja älykkään tehtaan*. Ponsse. https://www.ponsse.com/fi/yhtio/uutiset/a_p/P4s3zYhpxHUQ/c/ponsse-group-technology-company-epec-builds-a-responsible-and-smart-facto-1#/

Microsoft (28.11.2022). *Use continuous integration.* https://learn.microsoft.com/en-us/devops/develop/what-is-continuous-integration

Microsoft. (12.4.2023). *What is Azure Pipelines?* https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops

Myers, G.J., Sandler, C., & Badgett, T. (2012), *The art of software testing*. (3. painos). John Wiley & Sons.

Pezzè, M., & Young, M. (2008). *Software testing and analysis: Process, Principles and Techniques*. John Wiley & Sons.

Ponsse. (17.8.2022). *Ponsselta teknologialanseeraus: sähkökäyttöinen metsäkone*. https://www.ponsse.com/fi/yhtio/uutiset/a_p/P4s3zYhpxHUQ/c/ponsse-launches-new-technology-an-electric-forest-machine#/

W3schools of Technology. (i.a.). *SDLC V-Model*. https://www.w3schools.in/sdlc/v-model