Petri Jerohin

Sovellustestaus osana tuotekehitystä

Tietotekniikka
2024

VAASAN AMMATTIKORKEAKOULU
Tietotekniikka

TIIVISTELMÄ

Tekijä	Petri Jerohin
Opinnäytetyön nimi	Sovellustestaus osana tuotekehitystä
Vuosi	2024
Kieli	suomi
Sivumäärä	48
Ohjaaja	Jani Ahvonen

Tässä opinnäytetyössä käsitellään Danfossin sovellus- ja ohjelmistotestausta, jossa pyritään parantamaan Smoke-testausprosessin tehokkuutta ja kattavuutta. Projektin taustana on kehittää automaattista testausympäristöä päivittämällä testausjärjestelmien laitteistoa ja optimoimalla prosesseja. Tutkimusongelmana on selvittää, miten testijärjestelmien rakennetta ja toimintaa voidaan optimoida Danfossin tuotekehitysympäristössä.

Teoreettisessa viitekehyksessä tarkastellaan keskeisiä käsitteitä, kuten sovellustestauksen V-mallia, CI/CD-prosessia, automaattitestausta ja 3D-tulostuksen hyödyntämistä testausympäristön rakennuksessa. Tutkimuksessa käytettiin menetelmänä laadullista analyysia ja aineistona toimivat käytännön kokeet ja havainnot testiympäristöä rakennettaessa.

Keskeisiä havaintoja oli CLI-pohjaisten ohjelmien käytön hankaluus testiympäristöstä käsin, ja testijärjestelmien sijoituspaikaksi määriteltyjen laboratoriokaappien sähkökytkentöjen puutteellisuus. Opinnäytetyö tarjoaa tietoa ja havaintoja sovellus- ja ohjelmistotestauksen käytännöistä ja kehittämisestä Danfossilla ja laajemmin tuotekehityksessä.

Avainsanat	sovellustestaus, automaattitestaus, 3D-tulostus, CI/CD, pipeline

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietotekniikka

ABSTRACT

Author	Petri Jerohin
Title	Sovellustestaus osana tuotekehitystä
Year	2024
Language	Finnish
Pages	48
Name of Supervisor	Jani Ahvonen

This thesis paper will be focusing on Danfoss's software and application testing. It is aiming to improve efficiency and coverage of the Smoke testing process. This Project's background involves developing an automated testing environment by upgrading the hardware of the test systems and optimizing the processes involved. The research problem is to determine how the structure and functionality of the test systems could be optimized in the Danfoss's software testing environment.

The theoretical framework explores the key concepts such as: V-model of application testing, CI/CD-process, automated testing and the usage of 3D-printing in building the testing environment. The research employed qualitative analysis, using practical experiments and observations during the construction of the testing environment as its main data sources.

The key findings include challenges in using CLI-based programs from the testing environment, and the lacking electrical connections inside the laboratory cabinets that were designed for the testing systems. This thesis provides knowledge and observation about the practices and development of software and applications testing at Danfoss and more broadly withing production development.

Keywords	software testing, automated testing, 3D-printing, CI/CD, pipeline

SISÄLLYS

TIIVISTELMÄ

ABSTRACT

1 JOHDANTO ... 10

2 DANFOSS ... 11

3 SOVELLUS- JA OHJELMISTOTESTAUS ... 12

3.1 Yleiset testausvaiheet ... 13

3.2 Continuous Integration, -Delivery ja -Deployment ... 17

3.2.1 Pull Request-prosessin vaiheet ... 18

3.2.2 CI/CD-prosessi Pull Requestien avulla ... 20

3.3 Pipeline CI/CD-prosessin tukena ... 22

3.4 Smoke testit osana CI/CD-prosessia ... 25

4 TYÖN TOTEUTUS ... 26

4.1 Smoke-testijärjestelmän rakenne ... 26

4.2 Käytetyt mittalaitteet ja ohjelmistot/ohjelmointikielet ... 28

4.3 Manuaalisesta ohjelmistopäivityksestä automaattiseen ... 29

4.4 3D-tulostuksen hyödyntäminen testiympäristön rakentamisessa ... 30

4.4.1 AM-teknologia ... 31

4.4.2 FDM-menetelmä ... 31

4.4.3 SLA-menetelmä ... 32

4.4.4 AM-teknologian tavoitteet projektissa ... 32

4.5 Testijärjestelmän sijoitus ja kaapin muutokset ... 34

4.6 Testijärjestelmien tietokoneiden asennus ... 36

4.7 YAML pipeline ... 37

4.8 Ongelmat työn toteutuksen kanssa ... 40

5 TULOSTEN ANALYSOINTI JA JOHTOPÄÄTÖKSET ... 41

5.1 Aikavaatimukset verrattuna manuaaliseen testaamiseen ... 41

5.2 Testausympäristön arviointi ... 43

5.3 Automatisoidun testauksen kehitys ... 44

5.4 Työn johtopäätökset ja yhteenveto ... 46

LÄHTEET 47

KUVALUETTELO

Kuva 1 - Sovellustestauksen tasot V-mallina (Ammann & Offutt, 2008, s. 6) 13

Kuva 2 – GIT-haara visuaalisesti (GitHub Docs, Pull requests, 2024). 18

Kuva 3 - Pull Request-testien prosessikaavio 19

Kuva 4 - Azure Pipeline-malli (Azure DevOps Services, 2024). 23

Kuva 5 – Testijärjestelmän piirikorttien looginen kaavio 27

Kuva 6 - Emulaattorikortin 3D-tulostettu hylly.. 33

Kuva 7 - 3D-tulostettu hyllynkiinnike.. 34

Kuva 8 - DIN-kisko laboratoriokaapin hyllyllä 35

Kuva 9 - Laboratoriokaapin kytkentäkuvat.. 36

Kuva 10 - YAML inline ominaisuus (Azure DevOps Services, 2024)..................... 38

Kuva 11 - GitHub Actions script ominaisuus (GitHub Actions, 2024). 39

Avainsanaluettelo

Avoin lähdekoodi	Avoin lähdekoodi eli ns. ”open source” tarkoittaa ohjelmistoa, jonka lähdekoodi on vapaasti saatavilla, muokattavissa ja jaettavissa. Kehittäjät voivat käyttää, tutkia, muokata ja jakaa ohjelmaa omiin tarpeisiinsa. Avoimen lähdekoodin ohjelmistot tukevat yleensä yhteisölähtöistä kehitystä.
Azure DevOps	Azure DevOps on Microsoftin kehittämä pilvipalvelu alusta, jossa käyttäjä tai yritys voi valita itselleen parhaat ominaisuudet ja palvelut oman ideansa kehityksen helpottamiseksi. Automaattitestauksessa voidaan käyttää Azure-ympäristöstä hyväksi esimerkiksi GIT-tietovarasto- ja Pipeline-ominaisuuksia. (Microsoft Azure, 2024)
CLI	CLI (Command Line Interface) tarkoittaa komentorivipohjaista käyttöliittymä, jossa käyttäjä antaa komentoja tekstimuodossa. CLI mahdollistaa suoran vuorovaikutuksen ohjelmiston tai käyttöjärjestelmän kanssa ilman graafista käyttöliittymää. Esimerkki tästä on Linuxin käyttöjärjestelmät, mihin ei ole asennettu graafista käyttöliittymää.
COM-portti	COM-portti on sarjaportti, jota käytetään tietokoneiden ja laitteiden välisen tietoliikenneyhteyksien muodostamiseen. COM-porttiliitäntä on yleinen esimerkiksi tulostimille tai muille tietokoneen oheislaitteille.
Debug-data	Debug datalla tarkoitetaan informaatiota, jota kerätään ohjelmiston tai järjestelmän virheenkorjausprosessin aikana. Se sisältää virheilmoituksia, muuttujien arvoja, suoritustietoja ja lokeja, jotka auttavat kehittäjää tunnistamaan ja ratkaisemaan ongelmia ohjelman toiminnassa.
IDE	IDE (Integrated Development Environment) eli integroitu kehitysympäristö on ohjelmointityökalu, joka yhdistää erilaisia ominaisuuksia, kuten koodieditorin, kääntäjän, virheenkorjaustyökalut ja käyttöliittymän. IDE helpottaa ohjelmistokehitystä tarjoamalla kehittäjälle tehokkaita työkaluja koodin kirjoittamiseen, testaamiseen ja hallintaan yhdestä ohjelmasta.

Koodin käännös (compile)

Koodin käännöksellä tarkoitetaan prosessia, jossa jollain ohjelmointikielellä kirjoitetusta tiedostosta, jonka ihminen voi ymmärtää, muunnetaan eli käännetään esimerkiksi binääri- tai heksadesimaalimuotoinen tiedosto, jota tietokone ja muut elektroniset laitteet ymmärtävät.

Kommunikaatioväylät

Kommunikaatioväylällä tarkoitetaan eri komponenttien tai laitteiden välistä tiedonsiirtokanavaa, minkä avulla voidaan vaihtaa tietoa laitteesta toiseen. Yleisiä kommunikaatioväylien protokollia ovat esim. Modbus, Profinet, PCIe, CAN. Näiden lisäksi on olemassa useita muita protokollia eri tarkoituksiin.

Pakettimanageri

Pakettimanageri tai paketinhallintaohjelma on työkalu, joka asentaa, päivittää tai hallinnoi ohjelmistoja ja niiden riippuvuuksia automaattisesti. Se helpottaa ohjelmistojen ylläpitoa ajan tasalla varmistamalla, että oikeat versiot ja tarvittavat kirjastot ovat käytettävissä.

Pull Request (PR)

GIT-ympäristössä tapahtuva pyyntö/ilmoitus muille koodin kehittäjille, ominaisuuden lisäämisestä GIT-ympäristön päähaaraan. Vaatii yleensä katselmoinnin ja hyväksymisen muilta kehittäjiltä.

Repository

Repositorio, versionhallintavarasto, tietovarasto tai tämän opinnäytetyön asiayhteydessä koodivarasto, on yleensä verkkopalvelimelle määritelty tallennuspaikka, jossa ohjelmistokehittäjät säilyttävät ja hallinnoivat kooditiedostojaan. Ne on rakennettu yleensä versionhallintajärjestelmien, kuten GITin, avulla. Se mahdollistaa muutosten seurannan, koodin yhteiskehittämisen sekä ohjelmistoversioiden hallinnan.

Skripti

Sanalla skripti, englanniksi ”Script” tarkoitetaan ohjelmointikielellä kirjoitettua sarjaa käskyjä tai komentoja, joilla on jokin toiminnallisuus. Skriptien tarkoituksena on automatisoida tiettyjä tehtäviä tai toimintoja, ja ne ovat yleensä kevyempiä ja yksinkertaisempia kuin suuret sovellukset.

Smoke-testaus Smoke-testaus on nopea ja pinnallinen testausvaihe, joka keskittyy varmistamaan, että ohjelmisto tai järjestelmä toimii ilman vakavia virheitä. Tämä voi sisältää laitteiston perustoimintojen sekä keskeisten kommunikointiväylien testaamisen. Smoke-testausvaihe on yleensä ensimmäinen vaihe, jossa kaikki laitteen moduulit testataan yhdessä.

YAML YAML ei ole varsinaisesti ohjelmointikieli, vaan se on tiedostomuoto, joka on suunniteltu olemaan ihmiselle luettava. Sitä käytetään konfiguraatioiden määrittelyyn ja tietojen sarjallistamiseen säilyttäen yksinkertaisen ja helposti luettavan rakenteen verrattuna XML- tai JSON-tiedostomuotoihin.

Yhteisö (community) Yhteisö, kuten Python-yhteisö, koostuu ohjelmoijista, ohjelmistonkehittäjistä ja käyttäjistä, jotka jakavat tietoa, kehittävät työkaluja ja edistävät esim. ohjelmointikielen kehitystä. Yhteisö toimii avoimen lähdekoodin ympäristöissä, jossa se tarjoaa tukea ja resursseja muille käyttäjille.

1 JOHDANTO

Tämä opinnäytetyö toteutettiin Danfoss Drives Oy:n toimeksiannosta. Projektin tavoitteena oli päivittää vanha sovellustestausjärjestelmän laitteisto ja tehdä optimoivia muutoksia jo olemassa olevaan prosessiin. Päivitys pitää sisällään vanhojen laitteiden ja piirikorttien vaihtamisen uudempaan revisioon, uusien laitteiden lisäämisen ja järjestelmien lukumäärän nostamisen vastaamaan lisääntynyttä kuormitusta. Sovellustestausjärjestelmä on osa isompaa testauskokoanisuutta, mutta tämä projekti keskittyy pääsääntöisesti koko testausprosessin alussa tapahtuvaan smoke-testausvaiheeseen.

Smoke-testausvaiheen tarkoituksena on testata ja varmistaa, että tehdyt muutokset Danfossin laitteiden ohjelmistoihin ovat vakaita ja toimivia yleisellä tasolla. Tällä varmistetaan, että ohjelmisto voidaan ottaa tarkempaan testaukseen myöhemmässä testausvaiheessa (Chauhan, 2014). Testit on suunniteltu ajettavaksi automaattisesti silloin, kun ohjelmistokehittäjä tekee ohjelmistomuutoksen ja luo siitä uuden PR:n (Pull Request).

Smoke-testit on suunniteltu varmistamaan laitteiston tärkeimpien ominaisuuksien sekä kommunikaatioväylien toiminta muiden laitteiden kanssa. Testejä voidaan toteuttaa monenlaisissa ympäristöissä hyödyntäen erilaisia ohjelmistoja ja ohjelmointikieliä. Opinnäytetyössä esitellään esimerkkejä suosituista pipeline-työkaluista, käytetyistä ohjelmistotestauksen kehyksistä ja niissä hyödynnetyistä ohjelmointikielistä.

Projekti hyödyntää jo aikaisemmin kehitettyjä käsitteitä ja strategioita, kuten yleistä sovellustestauksen teoriaa yhdessä uusien ideoiden kanssa. Esimerkiksi kuinka AM-teknologiaa (Additive Manufacturing) eli 3D-tulostusta voidaan hyödyntää testijärjestelmien rakentamisessa.

2 DANFOSS

Danfoss on globaalisti toimiva teollisuusyritys, jonka Mads Clausen perusti Tanskassa vuonna 1933. Yrityksen ensimmäinen toiminimi oli "Dansk Køleautomatik og Apparatfabrik" ja tärkeimmät innovaatiot olivat termostaattiset paisuntaventtiilit, vesiventtiilit, termostaatit, painekytkimet ja suodatinkuivaimet. (Danfoss, 2024.) Nykyään Danfoss toimii globaalisti työpaikkana yli 41 000 henkilölle ja tekee kauppaa jopa 100 eri maan kanssa.

Danfoss toimii usealla eri markkina-alueella, kuten autoteollisuus, energia ja luonnonvarat, teollisuus tai meri- ja offshore-teollisuus. Yhteisenä tekijänä kaikilla markkina-alueilla on kestävät innovaatiot. Innovaatioiden tarkoitus on lisätä energiatehokkuutta, parantaa koneiden tuottavuutta, vähentää päästöjä, mahdollistaa sähköistämistä ja vähentää hiilidioksidipäästöjä yhdessä asiakkaiden kanssa johtavan sovellusosaamisen avulla.

Suomessa Danfossin toimipisteitä sijaitsee Vaasassa, Tampereella, Helsingissä ja Lappeenrannassa. Tämä opinnäytetyö on tehty Vaasan toimipisteellä (entinen Vacon), jossa työskentelee hieman alle 1000 henkilöä ja se keskittyy taajuusmuuttajien tuotekehitykseen ja tuotantoon. Vaasan toimipisteeltä tulleita taajuusmuuttajia voi löytää esimerkiksi uuden Wasaline hybridilautan, Aurora Botnian konehuoneesta. (Danfoss, 2024.)

3 SOVELLUS- JA OHJELMISTOTESTAUS

Sovellukset ja ohjelmistot ovat olennainen osa nykypäivän teknologiaa ja niitä käytetään laajasti sekä arkipäivän elämässä että automatisoiduissa työtehtävissä. Ohjelmistoja löytyy monista eri ympäristöistä, kuten maata kiertävistä satelliiteista ja laivojen GPS- ja viestintäjärjestelmistä. Useimmille ihmisille lähin ohjelmistoa sisältävä laite on oma matkapuhelin. Kuten Ammann ja Offutt (2008, s.3) kuvaavat:

> *Software is an essential component of embedded applications that control exotic applications such as airplanes, spaceships, and air traffic control systems, as well as mundane appliances such as watches, ovens, cars, DVD players, garage door openers, cell phones, and remote controllers.*

Olennainen vaihe sovellusten ja ohjelmistojen kehityksessä on testaus, jonka avulla varmistetaan kehitettävän laitteen, ohjelmiston tai ohjelman toiminnallisuus erilaisten testien avulla. Vaikka ohjelmistotestaus saatetaan toisinaan mieltää pelkästään koodissa olevien virheiden etsimiseksi ja löytämiseksi, sen tarkoitus on paljon laajempi. Testauksen keskeisiä osa-alueita ovat:

Verification eli varmennus tai verifikaatio, joka suoritetaan jokaisen ohjelmistokehitysvaiheen jälkeen. Verifikaatio selvittää, täyttääkö tuote tai ohjelmisto sen edellisessä ohjelmistokehitysvaiheessa määritellyt vaatimukset.

Validation eli validointi, joka suoritetaan ohjelmiston kehitysvaiheen lopussa. Validoinnilla varmistetaan, että ohjelmiston toiminnallisuus vastaa sen suunniteltua käyttötarkoitusta ja se täyttää loppukäyttäjien tarpeet ja odotukset.

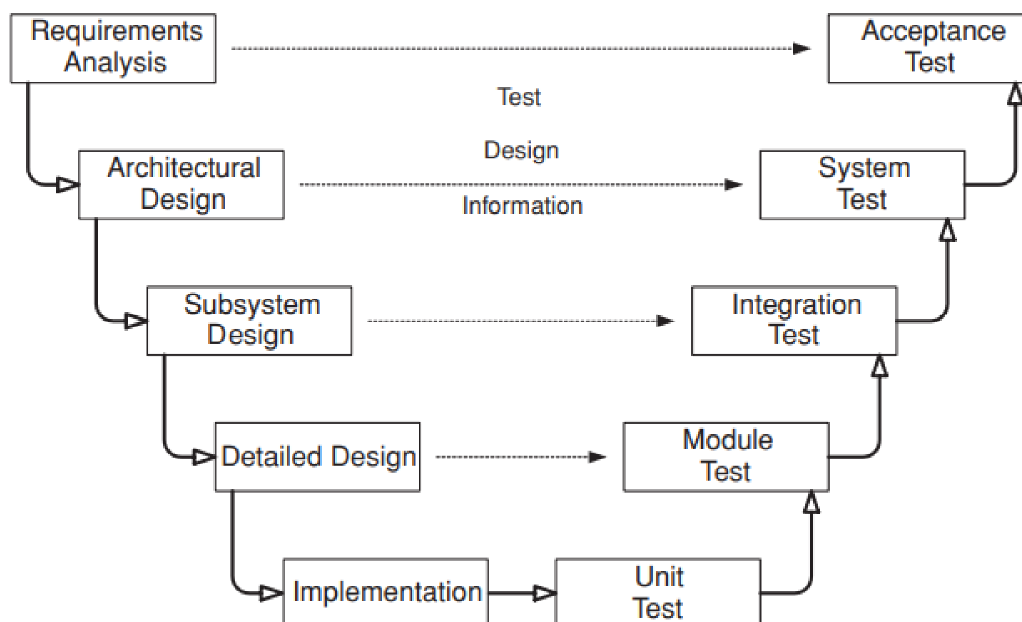
Defect detection eli vikojen havaitseminen, jossa testeillä pyritään tunnistamaan toimintahäiriöitä aiheuttavat virheet ja viat ohjelmiston koodissa.

Performance eli suorituskyky, jolla varmistetaan ohjelmiston toiminta tehokkaasti myös suuresti kuormitettuna esim. suurilla tiedonsiirtomäärillä.

3.1 Yleiset testausvaiheet

Ohjelmistotestaus on kokonaisuutena pitkä ja työläs prosessi. Sekä suurissa että pienissä projekteissa on tavallista jakaa testaus ns. eri tasoihin. Tämä mahdollistaa yhden testi-insinöörin kuormituksen laskua. Eri testitasoja voidaan myös tietyissä tapauksissa ajaa samaan aikaan. Yleisin tapa jakaa testitasoja perustuu eri ohjelmiston kehitysprosessin vaiheisiin. Tarkoituksena on, että testaus seuraa ohjelmistokehityksen luonnollisia vaiheita, jotta tuotetta voidaan tarvittaessa muokata tai hienosäätää vaihe vaiheelta. Yrityksen näkökulmasta katsottuna, mitä aikaisemmin virheet tai ongelmat löydetään, sitä helpommin ja halvemmalla ne pystytään vielä korjaamaan. Tästä syystä myös kehitysprosessin alussa tapahtuva testaus on tärkeä osa ohjelmistotestauksen prosessia.

Jakamalla koko tuotekehitysprosessi vaiheisiin, eristämällä kehityksen vaiheet ja testaamalla ne erikseen muodostuu siitä ns. ”V-Model” eli V-testausmalli, joka on esitelty kuvassa 1.



Kuva 1 - Sovellustestauksen tasot V-mallina (Ammann & Offutt, 2008, s. 6)

Yleinen neuvo testauksessa on suunnitella testit samanaikaisesti kunkin kehitystoiminnan vaiheen kanssa, vaikka ohjelmisto ei vielä olisikaan sellaisessa vaiheessa, että sitä voidaan ajaa laitteessa. Perusteluna neuvolle on, että pelkästään testien suunnittelu jo tässä vaiheessa voi paljastaa suunnittelupäätöksissä syntyneitä virheitä, jotka olisivat muuten voineet jäädä huomaamatta.

Requirements Analysis eli vaatimusten analysointivaihe tutkii ja käy läpi asiakkaalta tulleet tarpeet ja vaatimukset. Tätä vastaava Acceptance Test eli hyväksymistestaus on suunniteltu määrittämään, täyttääkö valmis ohjelmisto kaikki asiakkaan tarpeet ja toimiiko se loppukäyttäjän näkökulmasta oikein. Tämä testivaihe vaatii syvällistä osaamista aihealueesta ja asiakkaan näkökulmasta ohjelmistoon. (Ammann & Offutt, 2008, s. 5)

Architectural design eli arkkitehtinen suunnittelu on vaihe, jossa valitaan käytettävät komponentit ja liittimet, jotka yhdessä muodostavat järjestelmän. Järjestelmän on sen jälkeen tarkoitus täyttää edellisessä vaiheessa määritellyt tavoitteet. Ohjelmistokehityksessä komponenteilla ja liittimillä voidaan tarkoittaa elektronisten komponenttien lisäksi myös esimerkiksi funktioita (C/C++) / metodeja (Python/Java) tai rajapintoja eri järjestelmien välillä. System Test eli järjestelmätestaus on suunniteltu testaamaan, täyttääkö kasattu tuote kaikkia asetettuja järjestelmävaatimuksia. (Ammann & Offutt, 2008, s. 6)

Järjestelmätestauksessa oletetaan, että yksittäiset, ei vielä kasaan kootut laitteet tai ohjelmiston osat toimivat erikseen tarkoitetuillaan tavoilla, joten vaiheessa voidaan keskittyä pelkästään tarkistamaan, toimivatko eri osat yhteen kasattuna niin kuin pitäisi. Yksi järjestelmätestauksen vaihe on tässä opinnäytetyössä päivitettävä Smoke Testing-vaihe.

Järjestelmätestausvaihe etsii yleensä suunnittelu- tai spesifikaatio-ongelmia, ja se on yrityksen näkökulmasta kallis paikka löytää alemman tason virheitä, koska kehitysprosessi on jo pitkällä ja virheiden korjaaminen on todella työlästä. Järjestelmätestausta suorittaa yleensä erikseen määritelty testaustiimi, pikemminkin kuin

ohjelmoijan. Tämä mahdollistaa sen, että ohjelmoijien resurssit voidaan käyttää paremmin alemman tason testaukseen ja itse ohjelman parempilaatuiseen kehittämiseen. (Ammann & Offutt, 2008, s. 6)

Subsystem design eli alijärjestelmän- tai osajärjestelmän suunnittelu on sovelluskehityksen vaihe, joka määrittelee osajärjestelmän rakenteen ja käyttäytymisen. Jokaisella osajärjestelmällä on oma tehtävänsä osana isompaa kokonaisuutta. Esimerkkinä osajärjestelmästä voidaan käyttää puhelimen kameramoduulia, joka on oma osansa koko puhelimen toiminnallista kokonaisuutta. Kameramoduulille on kehitysvaiheessa luotu omat suunnitelmat sen käyttäytymisen, rakenteen ja ominaisuuksien osalta. Myöhemmin se on lisätty muiden osajärjestelmien rinnalle, minkä kautta siitä on tullut osa puhelinta. Osajärjestelmät ovat yleensä aiemmin kehitettyjen ohjelmistojen mukautuksia. (Ammann & Offutt, 2008, s. 6)

Integration Test eli integraatiotestaus on suunniteltu arvioimaan moduulien tai osajärjestelmien välisiä rajapintoja ja kommunikaatiota. Se arvioi, onko kommunikaatio kahden eri moduulin välillä tasaista ja siirtyykö data niiden välillä oikein. Esimerkiksi jos kännykän kameramoduulilla otetaan kuva, siirtyvätkö kuvan kaikki tiedot oikein muistimoduulille? Entä jos kyseessä onkin video? Jos videon siirto muistiin ei ole tasaista tai siirron aikana tapahtuu virheitä, voi video katsottaessa näyttää sumealta tai pahimmassa tapauksessa pelkältä mustalta ruudulta.

Integraatiotestauksessa oletetaan, että yksittäisissä alijärjestelmissä ei ole vikoja, koska ne on testattu jo aikaisemmassa vaiheessa. Tästä syystä testaus keskittyy pääsääntöisesti yllä mainittuihin asioihin. Joissain tilanteissa integraatiotestauksella voidaan tarkoittaa järjestelmätestausta tai toisinpäin, mutta toisin kuin järjestelmätestaus, integraatiotestaus tapahtuu yleensä ohjelmiston kehittäjätiimin toimesta. (Ammann & Offutt, 2008, s. 6)

Detailed design eli yksityiskohtaisen suunnittelun vaihe, on tarkoitettu yksittäisten moduulien tai ohjelmiston osien rakenteen ja toiminnallisuuden määrittelyyn.

Program Unit eli ohjelmayksikkö, jota voidaan kutsua myös Funktioksi C/C++ ohjelmointikielellä tai metodiksi Python/Java ohjelmointikielellä tarkoittaa kooditasolla yhtä tai useampaa riviä koodia, jolla on nimi ja jota muut osat koodista voivat kutsua. Moduulit koostuvat useista eri ohjelmayksiköistä, ja niitä voidaan kooditasolla kutsua tiedostoksi tai kirjastoksi C-ohjelmointikielessä tai nimellä ”Class” eli luokka Java/Python/C++ ohjelmointikielillä. (Ammann & Offutt, 2008, s. 6)

Module Test eli moduulitestaus on suunniteltu arvioimaan yksittäisten moduulien toimintaa irrotetussa ympäristössä muista moduuleista. Tarkoituksena on testata, kuinka eri ohjelmayksiköt toimivat toistensa kanssa ja liikkuuko data yksiköstä toiseen oikein ja oikeassa muodossa. Moduulitestaus jätetään useimmissa yrityksissä ohjelmiston kehittäjälle eli ohjelmoijalle. (Ammann & Offutt, 2008, s. 6)

Implementation eli implementaatio ohjelmistokehityksessä on vaihe, jossa varsinainen ohjelmointi eli ohjelmakoodin kirjoittaminen tapahtuu. Tässä vaiheessa ohjelmoija käyttää hyväkseen aiemmissa vaiheissa tehtyjä dokumentteja ja spesifikaatioita. Ne kertovat ohjelmoijalle, mitä ohjelman kuuluu tehdä ja mitä ominaisuuksia sillä pitää olla. Implementaatiovaiheessa luodaan aiemmin moduulitestausvaiheessa mainitut ohjelmayksiköt.

Unit Testing eli yksikkötestaus on testausmenetelmä, jossa tarkastellaan implementaatiovaiheessa kehitettyjä ohjelmayksikköjä. Yksikkötestaus on ns. alin mahdollinen taso, jossa ohjelmistoja testataan. Yksikkötestausta on mahdollista suorittaa myös tapauksissa, joissa ollaan luomassa yleiskäytössä olevia kirjastoja, eikä vielä tiedetä, millaisessa ympäristössä niitä tullaan käyttämään. Yksikkötestaus suuremmissa yrityksissä jätetään yleensä moduulitestauksen tapaan ohjelmoijan vastuulle. (Ammann & Offutt, 2008, s. 7)

Yksikkötestauksen helpottamiseksi on kehitetty useita työkaluja eri ohjelmointikielille. Esimerkiksi C-kielessä käytettävästä GoogleTest-ympäristöstä ja Pythonin "pytest"-koodikirjastosta löytyy molemmista "mocker"-ominaisuus, jota suomeksi kutsutaan matkijaksi. Mockerin avulla voidaan jäljitellä koodikirjaston, funktion tai

luokan toimintoja ilman, että niitä käytetään oikeasti. Tämä on erityisen hyödyllistä yksikkötestauksessa, kun työskennellään ympäristöissä, joissa käytetään esimerkiksi sarjaporttiyhteyksiä tai muita kommunikointiprotokollia muiden laitteiden kanssa.

Mockerin avulla voidaan näissä tapauksissa jäljitellä sarjaporttiyhteyden toimintoja, kuten yhteyden muodostamista, datan lähettämistä ja vastaanottamista sekä yhteyden sulkemista, ilman fyysistä yhteyttä toiseen laitteeseen. Tämän ansiosta ohjelmayksiköistä voidaan testata ominaisuuksia, joita ei normaalisti voisi testata pelkästään yhden yksikön avulla. Ilman mockerin apua nämä testit pitäisi suorittaa vasta moduulitestauksen yhteydessä. Mocker mahdollistaa virheiden löytymisen aikaisemmin, mikä pitkällä aikavälillä säästää yritykseltä merkittävästi aikaa, rahaa ja ohjelmistokehittäjien resursseja.

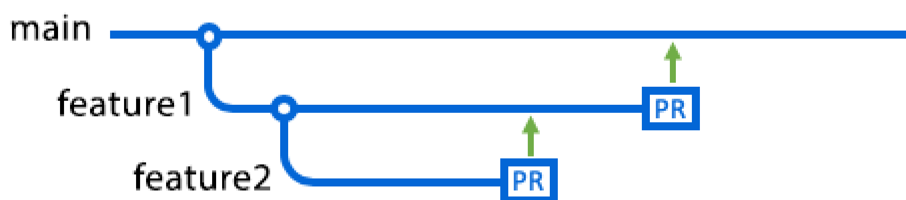
3.2 Continuous Integration, -Delivery ja -Deployment

Kun ohjelmiston eri testausvaiheet ovat tiedossa ja testattava kokonaisuus on määritelty, seuraava vaihe on päättää, millä tavoin testejä tullaan käytännössä suorittamaan. Manuaalinen testaus tarjoaa yleensä kehittäjälle tai testaajalle paremman käsityksen ohjelmiston sisäisestä toiminnasta. Manuaalinen lähestymistapa vaatii paljon aikaa ja resursseja erityisesti silloin, kun testien tuloksia vertaillaan käsin. Koska nykyaikaisessa ohjelmistokehityksessä resurssit halutaan jakaa tehokkaammin tehtäville, joita on vaikea tai jopa mahdotonta automatisoida, ovat jatkuvan integraation (CI) ja jatkuvan toimituksen (CD) käsitteet tulleet entistä tärkeämmiksi käytännöiksi.

Continuous Integrationilla eli jatkuvalla integraatiolla tarkoitetaan integraatioprosessia, jossa useat ohjelmistonkehittäjät integroivat ohjelmistomuutoksia yhteen jaettuun koodivarastoon (repository). Jokainen integraatio varmennetaan automaattisesti rakennus- ja testiprosessilla. Prosessin tarkoituksena on tunnistaa ja ratkaista ohjelmiston integraatio-ongelmat jo kehityksen aikaisessa vaiheessa,

jotta niiden korjaaminen olisi helpompaa. Tämä vähentää mahdollisia ongelmia kehitysprosessin myöhemmissä vaiheissa.

Prosessi alkaa siitä, kun ohjelmistokehittäjä haluaa tallentaa omasta koodivaraston haarastaan muutokset kaikkien yleiskäytössä olevaan haaraan. Tämä on yleinen käytäntö käytettäessä GIT-koodivarastoympäristöä, jossa yleisessä käytössä olevasta haarasta tehdään kopio. Kopioidussa haarassa ohjelmistokehittäjä voi muokata, lisätä ja poistaa koodia oman tarpeensa mukaan. Näin varmistetaan, ettei kehittäjän työ vaikuta muiden ohjelmoijien työhön, vaikka kaikki ei heti toimisikaan niin kuin pitäisi. Kuva 2 visualisoi GIT-haaran luonnin ja Pull Request-prosessin.



Kuva 2 – GIT-haara visuaalisesti (GitHub Docs, Pull requests, 2024).

3.2.1 Pull Request-prosessin vaiheet

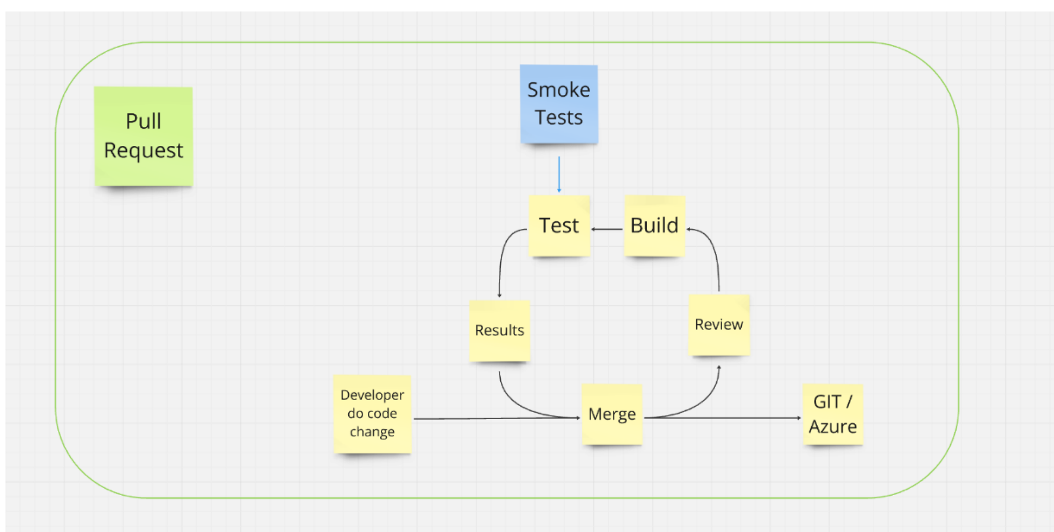
Kun kehittäjä on tehnyt tarvittavat koodimuutokset omassa haarassaan, luo hän vetopyynnön (Pull Request, PR) pyytääkseen koodimuutosten yhdistämistä pää-haaraan. Tämä prosessi koostuu yleensä eri vaiheista kuten koodin katselmointi, jossa muut kehittäjät tai/ja tiimin jäsenet tarkistavat koodimuutokset lukemalla ne läpi. Katselmoinneissa tarkastellaan yleensä koodin laatua, mahdollisia virheitä, yhteensopivuutta projektin vaatimuksien kanssa tai kommenttien laatua. Katselmoinnissa voidaan vaatia myös lisämuutoksia, jotka pitää korjata tai lisätä ennen hyväksymistä.

PR toimii myös automaattisten CI/CD testien laukaisimena (trigger), jossa pyyntöä ei pysty hyväksymään, jos ennalta määritellyt testit eivät mene hyväksytysti läpi.

Automaattisiin testeihin voi kuulua yksikkötestejä, integraatiotestejä ja myös Smoke-testit ovat usein osa CI/CD-automaattitestejä. Danfossilla PR aloittaa myös koodin ulkoasuun liittyviä testejä ja koodin automaattisen dokumentaation. Tämä varmistaa, että eri ohjelmistokehittäjät kirjoittaisivat mahdollisimman samantyylistä koodia, jotta Danfossin koodikanta pysyy järjestelmällisenä ja hyvin dokumentoituna.

Kun katselmointi on hyväksytty ja kaikki automaattitestit ovat onnistuneet, PR hyväksytään virallisesti ja koodimuutokset yhdistetään päähaaraan. Jos kirjoitetussa koodissa havaitaan testien avulla virheitä, pitää ohjelmistokehittäjän korjata ne, ja ajaa koko prosessi uudestaan.

Kuva 3 havainnollistaa visuaalisesti PR-prosessin. Kuvaan on lisätty Danfossilla käytössä oleva Smoke Test-vaihe osana Pull Request-testejä. Smoke-testit toimivat järjestelmätestausvaiheena PR-prosessissa, joka varmistaa Danfossin laitteiston toimivuuden uuden ohjelmistoversion kanssa.



Kuva 3 - Pull Request-testien prosessikaavio

3.2.2 CI/CD-prosessi Pull Requestien avulla

Ohjelmistokehittäjän luodessa Pull Requestin jatkuvan integraation (CI)-järjestelmä käynnistyy ja aloittaa automaattisen testauksen. Prosessi alkaa luomalla lokaali kopio tämänhetkisestä jaetusta päähaarasta ja lisäämällä kehittäjän oman haaran muutokset sen päälle. Tämän jälkeen koodi käännetään (compile) eli muutetaan ihmiselle luettavasta ohjelmointikielen koodista esimerkiksi binäärimuotoiseksi koodiksi, jota tietokone ymmärtää. Tämän jälkeen eri testivaiheet kuten, yksikkötestit, integraatiotestit ja smoke testit tarkistavat, että kaikki muutokset toimivat ilman suurempia ongelmia jo olemassa olevan ohjelmistokannan kanssa.

Jatkuvalla integraatiolla pyritään saamaan ohjelmistokehittäjät tekemään pienempiä muutoksia useasti päivän ja viikon aikana, sen sijaan että kaikki muutokset tehtäisiin yhtenä suurena muutoksena. Jatkuvalla integraatiolla vältetään myös ns. ”integration hell”, jolla viitataan suuren projektin lopussa tehtävään integraatiotyöhön. Jos integraatiotyö aloitetaan vasta projektin lopussa, on se todella aikaa vievää ja virheitä tapahtuu helposti. Jatkuvalla integraatiolla siis jaetaan projektin lopun työ pitemmälle aikavälille ja pienemmille koodimuutoksille kerrallaan. (Duvall ja muut, 2007, s.8)

Continuous Delivery eli jatkuva toimitus puolestaan on prosessi, joka seuraa jatkuvaa integraatiota. Sillä tarkoitetaan integraation jälkeistä vaihetta, joka keskittyy ohjelmistomuutosten automaattiseen käyttöönottoon myöhemmissä testausympäristöissä ja tuotannossa. Tarkoituksena on siis pitää ohjelmistokanta niin sanotussa ”Non-Stop deployable state” eli jatkuvasti toimitettavassa tilassa. Hyväksikäyttämällä CI-prosessissa tapahtuvaa koodin kääntämisprosessia voidaan ohjelmistokanta pitää teoreettisesti asiakkaalle lähetettävässä tilassa. Koska ohjelmisto on valmiiksi käännetty, on myöhemmin tehtävä testaus helpompi ja nopeampi toteuttaa. (Duvall ja muut, 2007, s.190)

CD-prosessissa tehtävä testaus tapahtuu yleensä järjestelmä- ja hyväksymistestaustasolla. Esimerkiksi Danfoss Drives Oy toteuttaa osan näistä testeistä määrävälein toistuvilla automaattitesteillä. Näiden testien ajo suoritetaan joka päivä aikataulutetusti, ja niiden tarkoituksena on ajaa syvällisempiä ja enemmän aikaa vieviä testejä. Aikataulutettujen testien ja muiden CD-prosessissa käytettävien testien avulla varmistetaan, että ohjelmisto on turvallinen, yhteensopiva ja valmiina ladattavaksi oikeille laitteille lisätestausta ja tuotantoa varten.

Vaikka CD-prosessi automatisoi suurimman osan käyttöönottoprosessista, jättää se silti viimeisen tärkeän vaiheen eli manuaalisen hyväksymisen tekemättä. Manuaalisen hyväksymisen vaiheen on tarkoituksena antaa tiimille aikaa tehdä viimehetken tarkistuksia ennen kuin ohjelmisto lähetetään käytettäväksi tuotantoon. Manuaalisen hyväksymisen vaihe jätetään viimeiseksi, koska siinä ohjelmistokehittäjät ja tiimin muut jäsenet varmistavat, ettei automaattisessa testauksessa tai missään muussa prosessissa ole tullut mitään ongelmia, jotka voisivat haitata laitteen toimintaa tuotannossa.

Jatkuvan Integraation ja toimituksen lisäksi on olemassa myös kolmas käytäntö, Continuous Deployment eli jatkuva käyttöönotto, josta myös käytetään lyhennettä ”CD”. Kun puhutaan jatkuvasta käyttöönotosta, tarkoitetaan sillä täysin automatisoitua prosessia ohjelmiston julkaisuun. Tämä käytäntö ottaa automaattisesti ohjelmistomuutokset käyttöön tuotantoympäristössä sen jälkeen, kun ne ovat läpäisseet kaikki vaaditut testit. Prosessi tapahtuu ilman manuaalista työtä.

Jatkuva käyttöönotto on hyvä ratkaisu vähäriskisille sovelluksille, joissa on vakaa testauskehys, koska se mahdollistaa nopean ja luotettavan ohjelmiston toimituksen. Tämä tarkoittaa, että uudet ominaisuudet, parannukset ja virheenkorjaukset voivat näkyä loppukäyttäjällä jopa muutamassa kymmenessä minuutissa. Nykypäivänä jatkuvaa käyttöönottoa voidaan hyödyntää esimerkiksi nettisivujen ylläpitämisessä tai videopelin virheen korjaamisessa. Prosessi lähtisi käyntiin siitä, että joku huomaa virheen esim. nettisivun linkissä ja ilmoittaa siitä sivun ylläpitäjälle. Sivun ylläpitäjä käy sen jälkeen tekemässä nopean muutoksen, joka korjaa linkin.

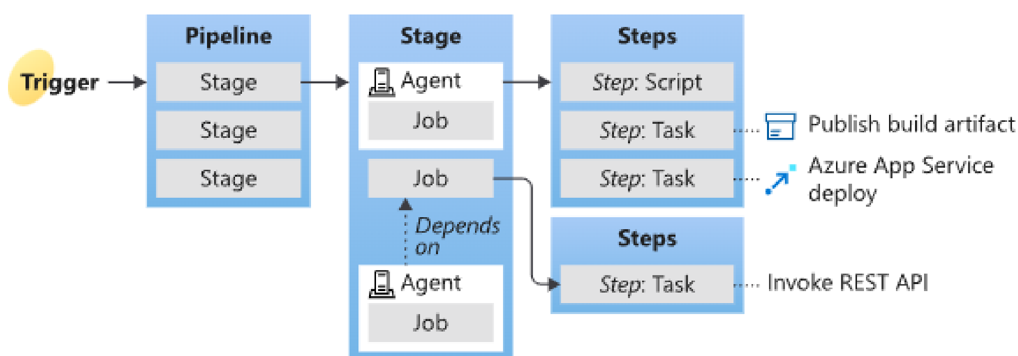
Muutoksen tallennus aloittaisi sen jälkeen automaattisen testausprosessin. Jos testausprosessi läpäisee kaikki tarvittavat testit, se siirtäisi muutoksen automaattisesti tuotantoserverille, joka korjaisi linkin kaikille nettisivun käyttäjille nopeasti ja tehokkaasti.

3.3 Pipeline CI/CD-prosessin tukena

Puhuttaessa CI/CD-järjestelmistä ja niiden toteutuksesta, yleinen käytäntö on rakentaa järjestelmä pipeline-muotoon. Suomeksi pipelinelle voi käyttää korvaavaa sanaa ohjelmistoputki, mutta sana "pipeline" on myös yleisesti käytössä. Tässä opinnäytetyössä tullaan käyttämään englanninkielistä lainasanaa suomalaisen sanan sijaan. Sovelluskehityksessä ja -testauksessa käytettävä pipeline on työkalu, jonka avulla voidaan määritellä erilaisia prosesseja tai tehtäviä, joita käytetään isomman prosessin automatisointiin. Pipeline voi sisältää esimerkiksi CI/CD-järjestelmissä käytettyjä vaiheita ja prosesseja.

Pipeline-työkaluja on saatavilla useilta eri palveluntarjoajilta, ja ne ryhmitellään usein käyttöympäristön mukaan. Esimerkiksi Microsoftin Azure DevOps-ympäristön Pipeline-työkalu ja GitHubin GitHub Actions ovat pilvipohjaisia ratkaisuja. Sen sijaan avoimen lähdekoodin Jenkins-automaatiopalvelin ja JetBrainsin kehittämä TeamCity ovat paikallisesti hallittavia työkaluja. Tässä opinnäytetyössä esitellyt pipeline-esimerkit ja toimintatavat on toteutettu Microsoftin Azure-ympäristön syntaksia noudattaen.

Pipelinen kehittämiseen Azure DevOps ympäristössä on käytetty YAML tiedostomuotoa, jonka avulla voidaan määritellä pipelinen konfiguraatio paremmin ihmiselle ymmärrettävässä muodossa. Konfiguraatio koostuu tasoista (stage), ja jokaisen tason sisällä on yksi tai useampi työ (job), joka puolestaan sisältää yhden tai useamman askeleen (step). Kuva 4 havainnollistaa Azure Pipelinen yleismallin ja siihen liittyvät vaiheet:



Kuva 4 - Azure Pipeline-malli (Azure DevOps Services, 2024).

"Stage" eli taso määrittelee loogisen rajan pipeline-prosessin sisällä ja sitä voidaan hyödyntää erottamaan eri päätehtäviä. Näitä päätehtäviä voivat olla esimerkiksi ohjelmiston käännösprosessi tai testaus. Jos pipelinen asetukset pidetään oletustilassa ja pipeline sisältää useamman tason, suoritetaan tasot järjestyksessä. Tason suorittamiseen voidaan myös määritellä tiettyjä ehtoja, kuten edellisen tason valmistuminen tai sen tila (Pass/Fail).

Azure Pipeline mahdollistaa myös tasojen rinnakkaisen ajon, mikäli tasoilla ei ole keskinäisiä riippuvuuksia. Tason riippuvuudet voidaan määritellä YAML-konfiguraatiossa, jolloin pipeline suorittaa ne oikeassa järjestyksessä. Lisäksi pipeline-tasot voidaan ajastaa tai käynnistää automaattisesti, esimerkiksi koodin päivityksen yhteydessä.

Jokainen taso koostuu yhdestä tai useammasta työstä (Job). Työ on agentille määritelty prosessi, joka sisältää yhden tai useamman askeleen (Step). Työn tarkoituksena on jakaa tason päätehtävä pienempiin prosesseihin. Esimerkiksi jos tason tehtävänä on testaus, yksi työ voisi olla tietyn testivaiheen, kuten järjestelmätestauksen, suorittaminen. Työstä syntyvät artifaktit, kuten käännetty ohjelmistokoodi tai testiraportit, voidaan tallentaa ja siirtää seuraaville tasoille jatkokäsittelyä varten. Tämä mahdollistaa saumattoman tiedonsiirron pipeline-vaiheiden välillä.

Askeleet ovat Azure Pipelinen pienimpiä yksiköitä. Ne koostuvat yhdestä tai useammasta skriptistä tai tehtävä (task), jotka suorittavat yksittäisiä toimenpiteitä. Tehtävät ja skriptit muodostavat pipelinen automaation ytimen. Esimerkki askeleesta voisi olla virtuaaliympäristön pystyttäminen testejä varten. Tehtävänä voisi toimia komentosarja, joka lataa ohjelmiston version koodivarastosta ja asentaa tarvittavat kirjastot tai apuohjelmat virtuaaliympäristöön.

Azure Agentti on palvelu, joka asennetaan tietokoneelle ajamaan pipeline-prosesseja kyseisen tietokoneen resurssien avulla. Resurssit voivat sisältää esimerkiksi mittalaitteita, testattavaa laitteistoa tai ohjelmistoja, joita tarvitaan testien suorittamiseen. Suuremmissa yrityksissä käytössä voi olla useita agentteja, joille voidaan määritellä erilaisia resursseja riippuen siitä, mitä laitteistoa niille on liitetty. YAML-konfiguraatiossa voidaan määrittää, mitä resursseja kukin työ tarvitsee, ja pipeline valitsee sen perusteella sopivan agentin.

Tehtävät pipelinen sisällä voivat olla riippuvaisia toisistaan tai itsenäisiä. Yhden työn sisällä olevat tehtävät ovat yleensä toisiinsa nähden joko riippuvaisia tai liittyvät samaan asiapiiriin. Esimerkiksi ensimmäinen työ voi asentaa tarvittavat lisäohjelmistot testiympäristön ajamista varten, minkä jälkeen seuraava tehtävä voi ajaa konfiguraatiossa määritellyt testit. Vaiheiden jakaminen eri agenteille mahdollistaa useiden testikonfiguraatioiden ajamisen samanaikaisesti toisistaan riippumatta.

Eri pipeline-työkalujen sisäinen syntaksi ja tapa jakaa tehtäviä loogisiin kokonaisuuksiin ovat yleisesti ottaen samankaltaisia, mutta niissä on myös merkittävästi työkalukohtaisia eroja. Esimerkiksi Azure DevOps pipeline-malli (Stage → Job → Step) on yksi tapa järjestää prosesseja, mutta tämä lähestymistapa ei ole kaikille työkaluille yhtenäinen standardi. Tästä syystä ohjelmistotestaajan on tärkeää perehtyä käytettävän työkalun dokumentaatioon huolellisesti, vaikka hänellä olisi aiempaa kokemusta muiden pipeline-työkalujen käytöstä.

3.4 Smoke testit osana CI/CD-prosessia

Jo aiemmin mainitut Smoke Testit ja niiden testijärjestelmät ovat merkittävässä roolissa CI/CD-järjestelmän prosessissa. Niiden tärkeimpänä tarkoituksena on varmistaa, onko ohjelmisto vielä uusien muutosten jälkeen tarpeeksi vakaa myöhemmin tulevia testivaiheita varten. Smoke-testit ovat osa järjestelmätestausvaihetta, ja ne ovat ensimmäiset testit, jotka ajetaan koko ohjelmistolle yhdessä. Termi smoke testaamiselle puhutaan olevan peräisin elektronisen laitteiston testauksesta: *"The phrase smoke test comes from electronic hardware testing. You plug in a new board and turn on the power. If you see smoke coming from the board, turn off the power. You don't have to do any more testing."(Kaner ja muut. 2002. s.15)*

Smoke-testien tarkoituksena ei ole havaita kaikkia puutteita ja ongelmia, mitä uudesta ohjelmistoversiosta voi löytyä. Siihen on olemassa myöhemmässä vaiheessa tarkempia ja laajempia testejä. Smoke-testeissä sen sijaan testataan järjestelmän kaikista kriittisimmät ominaisuudet ja mahdolliset kommunikaatioväylät samassa järjestelmässä olevien laitteiden välillä. Koska Smoke testit ajetaan PR-prosessin aikana, niiden ajallinen kesto ei saa olla pitkä. Yleisesti testien pituudessa puhutaan muutamista minuuteista tuntiin, mutta suuremmissa järjestelmissä ja testiympäristöissä pelkästään uuden ohjelmistoversion päivitys saattaa kestää 45 minuuttia. Tästä syystä suuremmissa testijärjestelmissä smoke testit voivat kestää kauemmin.

4 TYÖN TOTEUTUS

Kuten aiemmissa luvuissa on mainittu, tämän opinnäytetyön projekti keskittyy smoke testausvaiheen parannukseen taajuusmuuttajien tuotekehityksessä. Projektin päämääränä on päivittää vanhat testijärjestelmät uusiin versioihin. Tämä tarkoittaa vanhojen piirikorttien revisioiden päivittämistä ja muutamien uusien piirikorttien lisäämistä, jotka puuttuivat vanhoista järjestelmistä. Tämän lisäksi testijärjestelmien lukumäärä nostetaan vastaamaan lisääntynyttä kuormitusta. Projekti on suoritettu Danfoss Drives Oy:n toimeksiannosta Vaasassa vuonna 2024

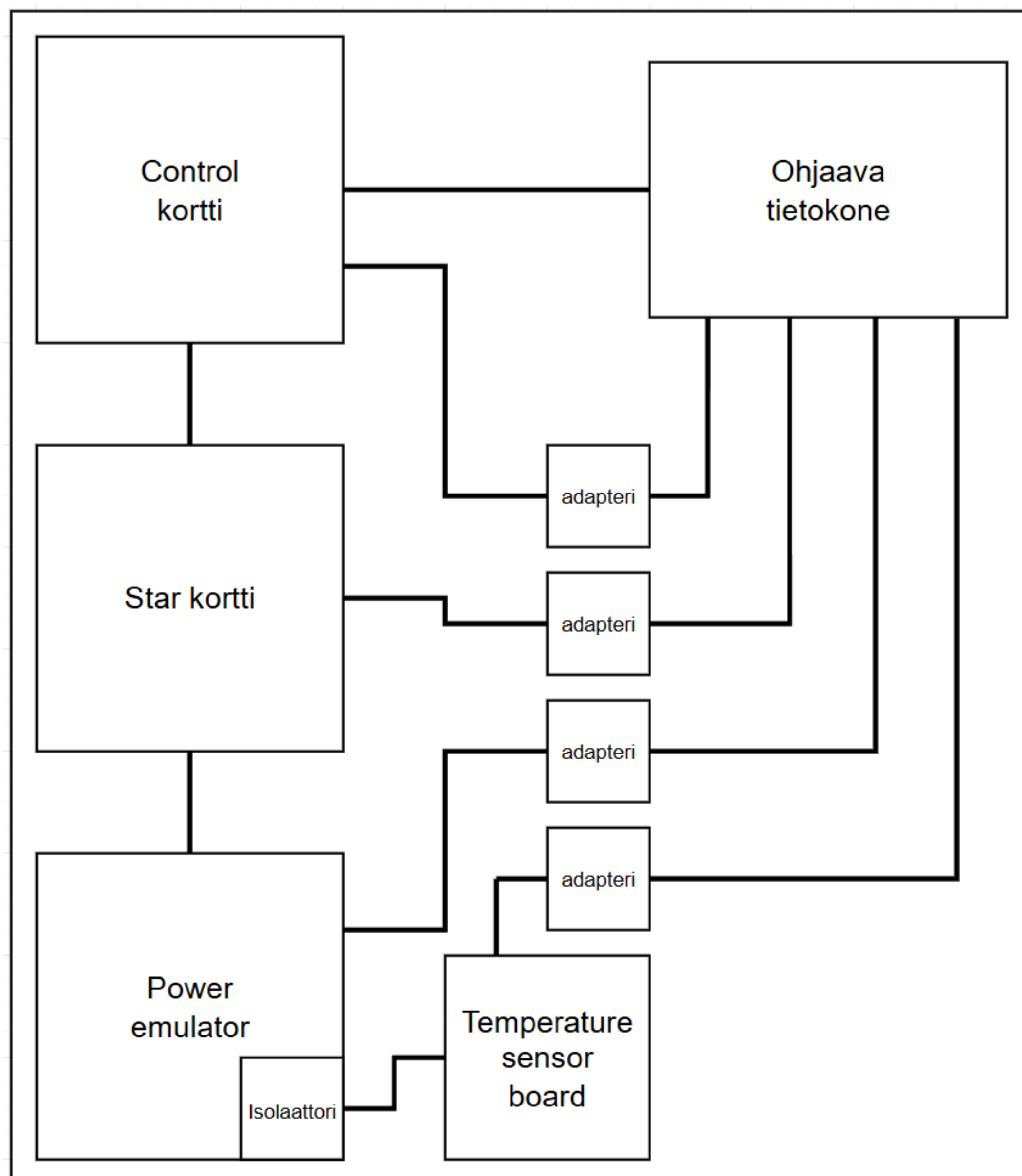
4.1 Smoke-testijärjestelmän rakenne

Puhuttaessa smoke testijärjestelmästä tarkoitetaan sillä yhtä laitekokonaisuutta, joka sisältää tietokoneen, useita Danfossin taajuusmuuttajan sisältä löytyviä piirikortteja ja muita samaan kokonaisuuteen liittyviä lisäpiirikorttia. Lisäpiirikortit toimivat avustavina adaptereina ja emulaattorikortteina, joiden tehtävänä on lukea ja lähettää Danfossin laitekorteille erimuotoista dataa.

Laitekorteista tärkeimmät ovat Control- ja Star-kortit. Control-kortti toimii ohjainyksikkönä koko laitteelle. Yhdistämällä erilaisia lisäkortteja Control-korttiin taajuusmuuttajaa voidaan käyttää monissa eri tilanteissa ja ympäristöissä. Star-kortti on tarkoitettu jakamaan yhdeltä Control-kortilta tulevat ohjeet usealle eri tehomoduulille samanaikaisesti. Tämä mahdollistaa yhden sähkömoottorin ohjauksen useasta rinnakkain kytketystä tehomoduulista vain yhden Control-kortin avulla. Star-kortteja on käytössä esimerkiksi Wasaline Aurora Botnia-hybridilautan konehuoneessa, jossa Danfossin taajuusmuuttaja säätelee kahta 6MW:n propulsiomoottoria ja kahta 1.5MW:n keulapotkuria (Danfoss, 2024).

Testausjärjestelmissä käytettävät lisäpiirikortit ovat Power-Emulator, jota käytetään emuloimaan tehomoduulin ja sähkömoottorin toimintaa. Isolaattori-piirikortti, joka on osana Power-Emulaattoria. Isolaattori-kortin taakse yhdistettävä

Temperature Sensor Board, joka pystyy mittaamaan laitteen lämpötiloja eri mittapisteistä. Kuva 5 havainnollistaa testijärjestelmän piirikorttien kytkentöjä.



Kuva 5 – Testijärjestelmän piirikorttien looginen kaavio

Testijärjestelmässä käytetään myös muita laitteita, jotka mahdollistavat laitteiden ohjauksen ja tietojen keräämisen. Näitä laitteita ovat esimerkiksi relekortti, jota käytetään laitekorttien käynnistämiseen ja sammuttamiseen ohjelmallisesti ja kommunikaatioadapteri, jonka avulla päästään lukemaan eri piirikorttien käynnistys ja debug-dataa tietokoneella, sekä ohjelmistopäivitysadapteri, joka muuttaa

tietokoneelta tulevan datan laitteen ymmärtämään muotoon. Muutetun datan avulla voidaan pakottaa ohjelmistopäivitys piirikorteille.

Star-kortti, Isolaattori ja Temperature Sensor Board ovat laitekortteja, jotka eivät olleet osana vanhoja Smoke-testijärjestelmiä. Niiden puuttuminen aiheutti ongelmia myöhemmissä testivaiheissa, sillä kortteihin liittyvät vakavat ongelmat havaittiin vasta tarkemmassa testauksessa. Tämä voi pahimmillaan johtaa tilanteisiin, joissa tarkemmassa testauksessa käytettävät testijärjestelmät rikkoutuvat. Testijärjestelmien rikkoutuminen puolestaan pysäyttää testausprosessin, kunnes virhe on korjattu.

Virheen löytäminen voi olla haastavaa, sillä tarkempia testejä ei välttämättä suoriteta samana päivänä, jolloin muutokset on tehty. Usein tarkempia testiajoja suoritetaan vasta öisin, jolloin virheellisten muutoksen päälle on voitu tehdä jo uusia muutoksia. Tämä johtaa tilanteeseen, jossa kaikki kyseisen päivän aikana tehdyt muutokset on tarkistettava manuaalisesti virheen löytämiseksi. Pieni virhe tai unohtunut kommentoitu koodirivi voi viedä testi-insinöörin resursseja jopa useita päiviä, jolloin automaattisia testejä ei voida ajaa rikkoutuneiden testijärjestelmien vuoksi.

4.2 Käytetyt mittalaitteet ja ohjelmistot/ohjelmointikielet

Danfoss Drives Oy:n testijärjestelmät koostuvat useista erilaisista piirikorteista ja lisäadaptereista, jotka vaativat toimiakseen erityisiä ohjelmia ja ohjelmointikielien rajapintoja. Koska näihin ohjelmiin ja ohjelmointikieliin liittyvä tieto on yrityksen tuotekehityksen kannalta luottamuksellista, on se rajattu pois tästä dokumentista.

Seuraavaksi käsitellään yleisesti käytössä olevia ohjelmointikieliä sekä niiden testauskehyksiä ja lisäosia. Ohjelmointikielen valinnassa tulee huomioida muun muassa sen ominaisuudet, yhteensopivuus valittuun testauskehykseen (framework) sekä integraatio kehitysympäristöön ja sopivaan pipeline-työkaluun.

Automaattitestauksessa yleisimmin käytettyjä ohjelmointikieliä ovat Python, Java ja C#. Python soveltuu monipuolisesti eri testausalueille, kuten yksikkötestaukseen, järjestelmätestaukseen ja automaatiojärjestelmien kehittämiseen. Java on erityisen hyvä yritys- ja verkkosovellusten testaukseen, kun taas C# soveltuu parhaiten Windows-sovellusten, Unity-pohjaisten pelien ja yrityssovellusten testaukseen.

Pythonin tunnettuja testauskehyksiä ovat muun muassa Robot Framework, joka tarjoaa helposti laajennettavia ratkaisuja erityisesti hyväksymistestaukseen. PyTest on joustava ja monipuolinen kehys, joka sopii erityisesti yksikkötestaukseen. Selenium puolestaan on suunnattu verkkosovellusten automaatiotestaukseen.

Javalle ominaisia testauskehyksiä ovat esimerkiksi JUnit, joka on suosittu yksikkötestauskehys, sekä TestNG, joka tukee edistyneitä ominaisuuksia, kuten parametrisoitua testausta. Lisäksi Javaan on saatavilla Selenium-lisäosa verkkosovellusten automaatiotestauksen tueksi.

C#-ohjelmointikielen testauskehyksiin kuuluvat muun muassa NUnit, joka keskittyy yksikkötestaukseen, sekä Microsoftin tarjoama MSTest. Myös C#:lle on olemassa Selenium-lisäosa, joka on suunniteltu verkkosovellusten automaatiotestaukseen.

4.3 Manuaalisesta ohjelmistopäivityksestä automaattiseen

Projekti käynnistettiin keväällä 2024, Q2-kauden alussa, Danfossin tarpeesta päivittää Smoke-testijärjestelmiä. Tavoitteena oli rakentaa uusia testijärjestelmiä vanhojen rinnalle. Lisäksi projektiin kuului uuden piirikorttirevision käyttöönotto sekä uusien laite- ja lisäkorttien lisääminen, jotta testijärjestelmät pystyisivät suorittamaan kattavampia testejä jo Smoke-testauksen vaiheessa. Projektin ensimmäisessä vaiheessa keskityttiin uusien järjestelmien spesifikaatioiden laatimiseen. Keskeisiä kysymyksiä oli kaksi: Mitä uudella järjestelmällä halutaan testata ja mitä

laitekortteja järjestelmiin lisätään, jotta testit ovat riittävän kattavia, mutta samalla tehokkaita ja ajallisesti hallittavissa?

Projektin toteutus aloitettiin tarvittavien resurssien hankinnalla. Koska projekti oli laaja ja vaati useita resursseja, joita tilattiin ulkoisilta toimittajilta, käytettiin projektin alkuvaihe hyödyksi tutustumalla testausjärjestelmän toimintaan. Tämä tehtiin omalla työpöydällä käyttäen erilaisia laitekortteja, mikä antoi hyvän pohjan järjestelmän ymmärtämiselle ennen varsinaisen työn alkamista. Ensimmäinen versio, jota käytettiin demonstroimaan tuotantotiimille uuden testijärjestelmän toimivuutta, koostui useista piirikorteista, jotka oli yhdistetty toisiinsa erivärisillä johdoilla. Tämä prototyyppi tarjosi alustavan käsityksen järjestelmän rakenteesta ja toimintaperiaatteista. Prototyyppi tarjosi hyvän alustan järjestelmän manuaalisen päivittämisen ja käsin suoritettavien testien harjoitteluun. Tuotantotiimin suositusten mukaisesti projektin alkuvaiheessa keskityttiin ohjelmistopäivitysten ja yksinkertaisten testien manuaaliseen suorittamiseen. Tämä manuaalinen vaihe auttoi ymmärtämään paremmin, mitä osia järjestelmästä tullaan myöhemmin automatisoimaan samalla, se loi vahvemman perustan jatkokehitykselle.

4.4 3D-tulostuksen hyödyntäminen testiympäristön rakentamisessa

Projektia varten tilattujen resurssien saavuttua voitiin aloittaa lopullisten testijärjestelmien rakentaminen. Osien asettelussa oli tärkeää huomioida eri laitekorttien väliset liitännät, laitteiden käyttöjännitteen syöttö relekortin kautta sekä adapterikorttien sijoittelu. Laitekortit Control ja Star sekä Star-korttiin yhdistettävä emulaattorikortti yhdistettiin sarjassa valokuitukaapelilla. Tämä asetti haasteita sopivan asettelun löytämiselle, koska valokuitukaapeli on herkkä vaurioitumaan eikä sitä saa taivuttaa liian jyrkästi.

Lisähaasteita aiheuttivat emulaattorikortti ja useat eri adapterikortit. Oikeissa taajuusmuuttajissa, jotka menevät asiakkaille, näitä kortteja ei käytetä, vaan ne ovat ainoastaan sovelluskehityksen tukena. Koska kortit eivät ole osa lopullista tuo-

tetta, niille ei ollut olemassa valmiita koteloita tai kiinnitysmekanismeja. Aiemmissa testijärjestelmissä emulaattori- ja adapterikortit oli kiinnitetty alustalevyyn ruuveilla, kierretangoilla ja muttereilla, ja tilanpuutteen vuoksi kortteja oli jouduttu kasaamaan päällekkäin. Tämä asettelu aiheutti ongelmia erityisesti silloin, kun alimmaisin piirilevy hajosi, sillä sen vaihtamiseksi piti purkaa kaikki päälle asetetut toimivat kortit.

Tässä projektissa päätettiin kokeilla ongelmaan uutta lähestymistapaa eli AM-teknologian hyödyntämistä. AM-teknologia eli Additive Manufacturing, tai yleisemmin 3D-printtaus, perustuu materiaalin lisäämiseen kappaleen valmistamiseksi. 3D-printtausta voidaan hyödyntää yksityiskohtaisten ja monimutkaisten prototyyppien valmistamisessa sekä nopeassa, tuotantolinjamaisessa kappaleiden tuotannossa.

4.4.1 AM-teknologia

Gebhardtin ja muiden (2019) määritelmän mukaan lisäysvalmistus (AM) on automaattinen prosessi, jossa tuotetaan kolmiulotteisia fyysisiä esineitä suoraan 3D CAD-tiedoista (tietokoneavusteinen suunnittelu). Prosessi perustuu kerroksittaiseen valmistukseen eikä vaadi osakohtaisia työkaluja, kuten jyrsimien tai porien käyttöä. Osat valmistetaan rakentamalla ja yhdistelemällä volyymielementtejä. Alun perin tätä prosessia kutsuttiin nopeaksi prototyypittämiseksi, ja tämä nimitys on yhä yleinen.

4.4.2 FDM-menetelmä

3D-printtauksen menetelmiä on useita, mutta yleisin pöytätason tulostimissa on FDM (Fused Deposition Modeling). FDM-tulostin toimii puristamalla muovifilamenttia kerros kerrokselta rakennusalustalle. Tämä menetelmä on kustannustehokas ja nopea tapa valmistaa fyysisiä malleja, mutta valmiilla tuotteilla on usein

suhteellisen karkea pinta. Lisäksi tulostusorientaatio voi vaikuttaa kerrosten väliseen kestävyyteen, mikä saattaa johtaa heikkoon kappaleen kestävyyteen. (Ahart, 2019.)

4.4.3 SLA-menetelmä

Toinen yleisesti teollisuudessa käytetty menetelmä on SLA (Stereolithography) -tyylinen 3D-printtaus. SLA tunnetaan hartsiperusteisena tulostusteknologiana, jossa hartsi kovetetaan UV-valon avulla kiinteäksi kappaleeksi. Tätä menetelmää käytetään konseptimallien, kosmeettisten prototyyppien ja monimutkaisten osien, joilla on erityisiä geometrisia muotoja, luomiseen. SLA-teknologialla voidaan saavuttaa erittäin korkeita tarkkuuksia ja erinomaisia pintaviimeistelyitä.

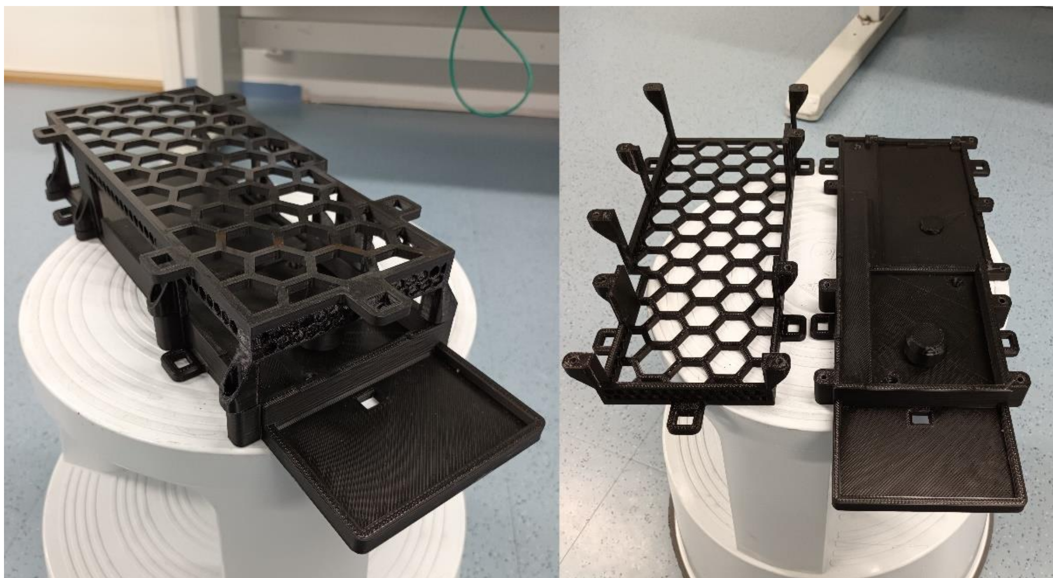
Erinomaisten pintaviimeistelyjen ansiosta SLA-teknologialla tulostetuilla kappaleilla ei esiinny samanlaisia ongelmia kerrosten välisessä kestävyydessä kuin FDM-teknologian kappaleissa. Tästä syystä SLA on suositumpi vaihtoehto erityisesti valmiiden kappaleiden tuotannossa teollisuudessa. (Ahart, 2019.)

4.4.4 AM-teknologian tavoitteet projektissa

Projektin yhtenä tavoitteena oli helpottaa testijärjestelmien huoltotoimenpiteitä, ja tämä toteutettiin AM-teknologian avulla. Emulaattorikortille ja eri adapterikorteille suunniteltiin 3D-mallinnusohjelmalla hyllykköratkaisuja, jotka mahdollistivat saman tyyppisten korttien pinottavan ja kiinteän asettelun. Hyllykköratkaisu pitää kortit järjestyksessä ja tekee niistä helposti irrotettavia huoltotarpeen ilmetessä.

Suunnitteluprosessissa otettiin huomioon eri korttien mitat ja liitännät, jotta hyllyn käyttö olisi mahdollisimman vaivatonta ja tehokasta. Tämän ratkaisun myötä huoltotoimenpiteet yksinkertaistuivat, mikä paransi testijärjestelmien käytettävyyttä ja vähensi huoltamiseen kuluvaa aikaa.

Kuvassa 6 on esitetty emulaattorikorttia varten suunniteltu hylly. Se mahdollistaa emulaattorikortin asennuksen ja purkamisen vaikuttamatta muihin testijärjestelmän laitteisiin ja emulaattorikortteihin. Hyllyn päädyssä oleva taso on tarkoitettu emulaattorikortin adapterilaitetta varten.



Kuva 6 - Emulaattorikortin 3D-tulostettu hylly

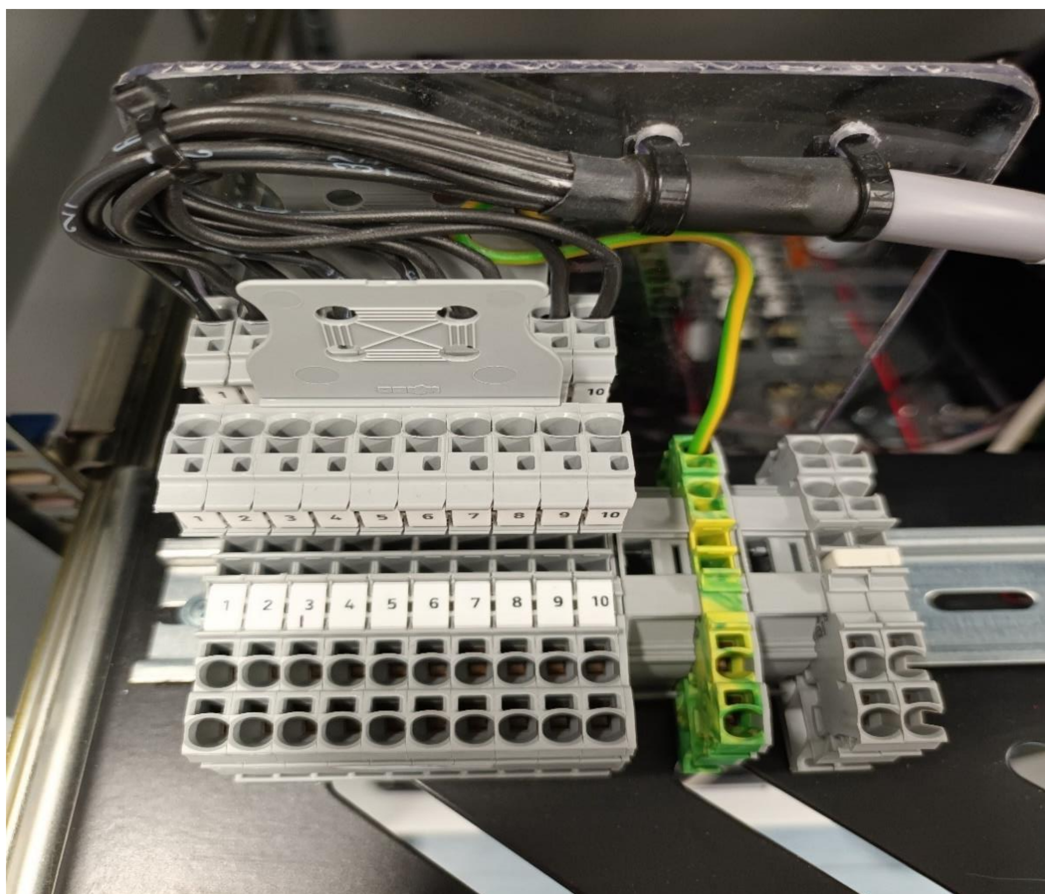
Emulaattorikortin hyllynkansi on suunniteltu kuusikulmiokuviolliseksi emulaattorikorttiin liitettävien johtojen kiinnityksen helpottamiseksi. Tämän lisäksi kuviointi parantaa myös emulaattorikortin jäähdytystä, koska laboratoriokaappiin sisäänrakennettu tuuletin pystyy kierrättämään enemmän ilmaa hyllyn läpi, verrattuna kokonaan koteloituun emulaattorikorttiin ja sen hyllyyn.

Emulaattorikortin hyllyyn on suunniteltu myös viisi uloketta, joiden avulla hyllyjä voidaan kasata päällekkäin. Hylly kiinnitetään toisiinsa siihen suunnitellulla kiinnityspalalla (Kuva 7). Kiinnityspala on tulostettu PLA-muovilla, joka antaa yhdessä vaakatasoisen tulostusorientaation kanssa juuri tarpeeksi joustavuutta, että pala on helppo kiinnittää ja irrottaa.

Kuva 7 - 3D-tulostettu hyllynkiinnike

4.5 Testijärjestelmän sijoitus ja kaapin muutokset

Kaikkien testijärjestelmien lopullinen sijoituspaikka tulee olemaan niille kaavailluissa kahdessa laboratoriokaapissa. Yksi kaappi sisältää neljä vikavirtasuojakytkintä, kahdeksan 230VAC-eristettyä pistorasiaa, sisäänrakennetun +24VDC/15A-jännitelähteen ja tuuletusjärjestelmän. Kaapin kaikki ominaisuudet ovat oman vikavirtasuojakytkimensä takana. Lisäksi kaapissa on useita hyllyä, joista vain osalle on kytketty laitteen käyttöjännitteenä toimivan +24VDC ja nollataso GND. Kytkentä on tehty käyttäen WAGO merkkisiä DIN-kiskoon yhdistettäviä liittimiä. (Kuva 8)

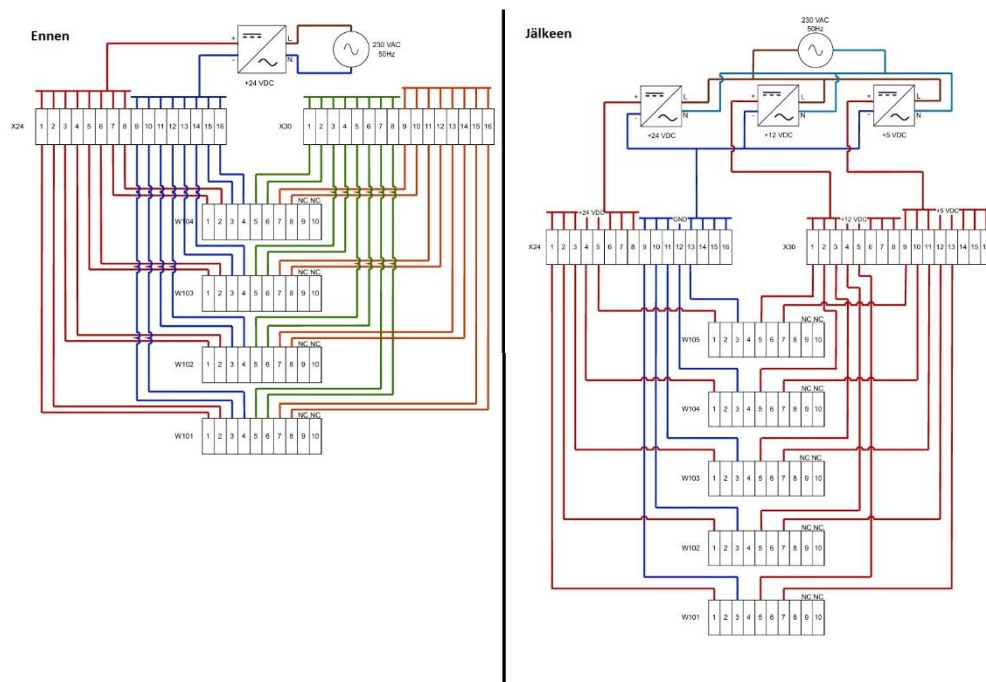


Kuva 8 - DIN-kisko laboratoriokaapin hyllyllä

Lisäksi testijärjestelmät tarvitsevat +5VDC ja +12VDC käyttöjännitteen muiden järjestelmän lisälaitteiden käyttöön. Aiemmin nämä jännitteet on toteutettu laitteiden omilla jännitemuuntajilla, jotka on yhdistetty kaapin pistorasioihin. Koska yhteen kaappiin tullaan sijoittamaan useita testijärjestelmiä, pistorasioiden määrä ei riittänyt kaikkien lisälaitteiden tarpeeseen. Tämä ratkaistiin lisäämällä kaappeihin kaksi uutta vikavirtasuojakytkintä, joiden taakse kytkettiin tarvittavat jännitelähteet.

Lisälaitteiden käyttöjännitteet yhdistettiin DIN-kiskon liittimiin hyllyillä. Samalla lopuillekin hyllyille asennettiin omat DIN-kiskonsa, jotta jokaiselle testijärjestelmälle saatiin oma kiskonsa, eikä niitä tarvinnut jakaa hyllyjen välillä.

Kuva 9 esittää kytkentäkaavion laboratoriokaapin ennen ja jälkeen tilanteista. DIN-kiskossa eri jännitetasojen väliin jätettiin yksi tyhjä liitinpaikka, jotta tarpeen vaatiessa järjestelmää voidaan laajentaa jälkikäteen tukemaan uusia laitteita.



Kuva 9 - Laboratoriokaapin kytkentäkuvat

4.6 Testijärjestelmien tietokoneiden asennus

Testijärjestelmien tietokoneet asennettiin järjestelmän aluslevylle niille suunnitelluille paikoille. Tietokoneiden kiinnityskehikko oli muokattu alkuperäisesti näytön taakse kiinnitettävästä metallikehikosta siten, että siitä oli sahattu pala pois. Palan sahaus varmisti, että kehikko sopi korkeudeltaan pystysuoraan asentoon. Tietokoneet sijoitettiin pystysuoraan asentoon tilan säästämiseksi aluslevyllä.

Tietokone liitettiin testijärjestelmään RJ45-ethernet-kaapelin kautta, ja eri adapterikortteihin yhteys toteutettiin USB-kaapeleilla. Koska tarvittavien USB-laitteiden määrä ylitti tietokoneen omien porttien kapasiteetin, ongelma ratkaistiin

käyttämällä USB-hubia. USB-hub jakaa yhden tietokoneeseen liitettävän USB-liitännän useisiin eri USB-portteihin, mikä mahdollisti kaikkien tarvittavien laitteiden kytkemisen.

Lisälaitteiden toiminnan varmistamiseksi tarvittiin ajureita, jotka ladattiin jokaiselle tietokoneelle laitevalmistajien virallisilta verkkosivuilta. Suurin osa ohjelmistoista, kuten ohjelmointikieli ja sen testauskehys sekä muut lisäohjelmistot, asennettiin Chocolatey-nimisen pakettimanagerin kautta. Se latasi asennustiedostot suoraan Danfossin verkkopalvelimelta ja suoritti niiden asennuksen tietokoneelle. Lisäpalveluiden asennus suoritettiin manuaalisesti lataamalla ne palveluntarjoajan verkkosivuilta, koska niiden asennuspaketteja ei ollut saatavilla Danfossin verkkopalvelimella.

Pipeline Agentin asennuksen yhteydessä tietokoneen agentille määriteltiin uniikki nimi, jonka avulla se voidaan liittää myöhemmin oikeaan agenttiryhmään (Agent Pool). Agenttiryhmä toimii pipelinen resurssiryhmänä, jota käytetään määrittämään, millä tietokoneella (agentilla) pipelinen testit suoritetaan. Agenttiryhmät helpottavat agenttien hallintaa, koska yksi agentti voi kuulua useampaan eri ryhmään sen resurssien perusteella. Tämä optimoi resurssien käyttöä monen pipelinen välillä. (Azure DevOps Services, 2024.) Pipeline-agentit ovat käytössä lähes kaikilla pipeline-työkaluilla, mutta ne on yleensä nimetty eri tavalla riippuen palveluntarjoajasta. Esimerkiksi GitHub- ja GitLab-ympäristöissä agenteista käytetään nimitystä runners.

4.7 YAML pipeline

Tietokoneiden asennuksen jälkeen, pipeline-agenteille luotiin oma agenttiryhmä, johon agentit lisättiin aiemmin määriteltyjen uniikkien nimien perusteella. Pipeline-kehitys aloitettiin manuaalisen pipeline-ratkaisun kehityksellä. Manuaalinen pipeline tarkoittaa, että sen ajoa ei aloiteta automaattisesti, kuten Pull Request-pipeline, joka käynnistyy uuden koodivarastoon tehdyn Pull Requestin yhteydessä. Manuaali ajo alkaa käyttäjän antamasta komennosta.

Pipelinen YAML-rakenne voi koostua yhdestä suuresta tiedostosta, jossa on kaikki pipelinen toiminnallisuus, tai useista niin kutsutuista ”template”-tiedostoista. Template-tiedostot helpottavat koodin uudelleenkäyttöä muuttamalla tiedostoon syötettäviä parametreja (input parameters). Lisäksi template-tiedostojen käyttö tekee YAML-koodista jäsennellympää, koska pipelinen ominaisuudet voidaan jakaa helposti omiin tiedostoihinsa. Tämä helpottaa myös koodin muokkausta myöhemmin, koska kaikki tiettyyn toiminnallisuuteen liittyvät koodirivit löytyvät samasta tiedostosta, eikä niitä tarvitse etsiä yhdestä suuresta tiedostosta.

Pipelinen toiminnallisuudet voidaan toteuttaa ajamalla YAML-ympäristöstä ohjelmatiedostoja, jotka on kirjoitettu muilla ohjelmointikielillä. Ohjelmatiedostot voivat olla esimerkiksi jonkin ohjelmointikielen testikehyksellä ohjelmoituja testitiedostoja. Toinen tapa on käyttää YAML task-ominaisuutta nimeltä ”inline-script”. Sen avulla YAML-koodiin voidaan lisätä komentoja muista ohjelmointi- tai komentokielistä, kuten PowerShell tai Python. Kuvissa 10 ja 11 esitellään esimerkki Microsoftin Azure pipelinen ja GitHubin actions-ympäristöissä erilaisten scriptien ajamisesta.

Write a warning

yml Copy

```yml
- task: PowerShell@2
  inputs:
    targetType: 'inline'
    script: Write-Host "##vso[task.LogIssue type=warning;]This is the warning"
    # Writes a warning to build summary and to log in yellow text
```

Kuva 10 - YAML inline ominaisuus (Azure DevOps Services, 2024).

```yaml
- uses: actions/github-script@v7
  id: set-result
  with:
    script: return "Hello!"
    result-encoding: string
- name: Get result
  run: echo "${{steps.set-result.outputs.result}}"
```

Kuva 11 - GitHub Actions script ominaisuus (GitHub Actions, 2024).

Pipeline päätettiin toteuttaa ensin testiympäristöön manuaalisena, jotta testijärjestelmän toiminnallisuutta voitaisiin kokeilla paremmin. Manuaalinen pipeline helpottaa myös ohjelmistokehitystä, kuten testijärjestelmien ohjelmistopäivityksissä käytettävän apuohjelman ohjelmointia. Ohjelmistopäivitys olisi voitu toteuttaa myös YAML-koodissa, mutta prosessissa on mukana vaiheita, jotka suoritetaan ainoastaan tapauksissa, joissa edellinen vaihe epäonnistuu. Tämä loogisten vaiheiden hallinta on helpompi toteuttaa ohjelmointikielen testauskehyksen avulla kuin YAML-koodilla.

Toiminnallisuuden kehittäminen YAML-koodin avulla olisi ollut erittäin työlästä, koska pilvipalvelupohjaista pipeline koodia ei pysty ajamaan paikallisesti. Se suoritetaan aina pilvipalvelun kautta kommunikaationa paikallisen agentin kanssa. Tämä tarkoittaa, että jokainen muutos on ensin vietävä koodivaraston pilvessä olevaan haaraan. Kun muutos on lisätty haaraan, pipeline on ajettava alusta asti, koska sen käyttämä ympäristö alustetaan aina ajon alussa. Alustusprosessi voi kestää jopa useita minuutteja riippuen siitä, paljonko ohjelmia ja ohjelmointikielen eri lisäosia tietokoneelle on asennettu jo valmiiksi. Pipeline aloittaa halutun toiminnallisuuden vasta ympäristön alustuksen jälkeen, jonka jälkeen sitä pääsee testaamaan. Tämän prosessin kesto oli toinen tärkeä syy siihen, miksi toiminnallisuus päätettiin toteuttaa ohjelmointikielen testauskehyksen avulla. Koska testauskehys on ohjelmointikielen lisäosa, pystyy sen avulla luotuja tiedostoja ajaa paikallisessa ympäristössä, mikä vähentää merkittävästi toiminnallisuuden testaamiseen kuluvaa aikaa.

Kun manuaalisen pipelinen toiminnallisuus saadaan toteutettua halutulla tavalla testiympäristössä, muokataan se seuraavaksi automaattisen pipelinen muotoon. Juuri muutettu automaattinen pipeline pidetään vielä testiympäristössä, jotta sen toimintaa voidaan seurata rinnakkain vanhojen Smoke-testijärjestelmien kanssa. Järjestelmän pito testiympäristössä mahdollistaa testausprosessin olevien vikojen ja muiden muutosten tekemisen vaikuttamatta tuotannossa olevaan koodivarastoon.

Kun automaatti-pipelinen ja testijärjestelmien toiminnallisuus on varmistettu, vanhat testijärjestelmät korvataan uusilla. Vanhoille testijärjestelmille ei ole määritelty vielä uusia tehtäviä, mutta ne tullaan luultavasti päivittämään uusilla laitteilla, minkä jälkeen ne siirretään toiseen vaiheeseen testausprosessia, jotta nykyisten testijärjestelmien kuormitusta voidaan pienentää.

4.8 Ongelmat työn toteutuksen kanssa

Projektin aikana ilmeni useita haasteita, jotka vaikuttivat sen etenemiseen. Emulaattorikortille suunnitellussa 3D-tulostetussa hyllyssä ei otettu huomioon FDM-tulostustyylin toleransseja verrattuna SLA-tyyliseen tulostukseen. Koska hyllyprototyyppi tulostettiin SLA-tyylisellä tulostimella aikarajoitusten ja laitteiden saatavuuden vuoksi, emulaattorikortti ei mahtunut valmiiseen FDM-tyylisesti tulostettuun hyllyyn. Ongelma ratkaistiin suurentamalla 3D-mallin toleransseja ja hiomalla jo tulostetuista kappaleista ylimääräistä muovia pois pienellä viilalla.

Laboratoriokaappien kytkennän muokkaaminen oli myös haaste, jota ei osattu ennakoida projektin alussa. Kaapit oli alun perin tarkoitettu neljälle erityyppiselle testijärjestelmälle, joissa käytetään toista lisälaitetta. Lisälaite puuttuu smoke-testijärjestelmiltä kokonaan, ja sen toiminnallisuus on korvattu toisella lisälaitteella. Lisäksi kaappeihin tullaan sijoittamaan viisi järjestelmää neljän sijaan. Tämä aiheutti ongelmia, jotka ratkaistiin lisäämällä kaappeihin jälkikäteen +5VDC- ja +12VDC-jännitelähteet. Ongelman olisi voinut ennaltaehkäistä tilaamalla Smoke-järjestelmille tarkoitettuja kaappeja.

5 TULOSTEN ANALYSOINTI JA JOHTOPÄÄTÖKSET

Vaikka projekti on edelleen kesken, tähän mennessä saavutetut tulokset tarjoavat tärkeää tietoa ja suunnan jatkotyöskentelylle. Tämänhetkiset havainnot ja analyysit ovat antaneet mahdollisuuden arvioida testijärjestelmien toimivuutta sekä tunnistaa mahdollisia kehityskohteita. Seuraavissa luvuissa käsitellään saavutettuja tuloksia ja niiden vaikutuksia projektin loppuvaiheisiin. Lopuksi käydään myös läpi, miten projekti jatkuu tästä eteenpäin.

5.1 Aikavaatimukset verrattuna manuaaliseen testaamiseen

Manuaalisen testaamisen aikarakenne koostuu järjestelmän ohjelmistopäivityksestä, adapterilaitteiden ja terminaaliyhteyden muodostuksesta laitteeseen, testien suorittamisesta, tulosten analysoinnista ja raportoinnista.

Jos ohjelmistopäivitys onnistuu ilman ongelmia, se kestää yhtä kauan kuin automaattitestauksessa. Mikäli päivityksen aikana ilmenee virheitä ja ohjelmisto täytyy pakottaa laitteeseen, päivitysprosessi pitkittyy. Lisäaika kuluu adapterin kytkemiseen, ohjelmistopäivityksen tekevän ohjelman käynnistämiseen sekä ohjelmiston binääritiedostojen lataamiseen Danfossin pilvipalvelusta.

Adapterilaitteiden ja terminaaliyhteyksien muodostaminen vie yhtä paljon aikaa sekä manuaalisessa että automaattisessa testauksessa. Testien suorittaminen sen sijaan on huomattavasti hitaampaa manuaalitestauksessa. Vaikka manuaalitestauksessa suoritetaan samat komennot kuin automaattitesteissä, jo pelkkä komentojen syöttäminen ja tulosten tarkastus vie merkittävästi enemmän aikaa verrattuna automaattiseen testausprosessiin.

Tulosten analysointi voi olla nopeampaa manuaalisessa testauksessa, koska analyysiä voidaan suorittaa samanaikaisesti testien kanssa. Automaattitesteissä tulosten analysointi tapahtuu vasta testien päätyttyä lukemalla testiraporttia. Manuaalisen testiraportin kirjoittaminen, vaikka sitä kirjoitettaisiin testien aikana, vie myös merkittävästi enemmän aikaa kuin automaattitesteissä.

Automaattisen testauksen aikarakenne koostuu YAML-pipeline-ympäristön alustuksesta, johon sisältyy koodivarastosta uuden version lataaminen, tarvittavien ohjelmistojen ja lisäosien asennus ja laitteisiin yhdistäminen. Lisäksi automaattitestit vievät aikaa ohjelmiston päivitykseen, testien ajamiseen, tulosten tallentamiseen ja testiympäristön alasajoon.

Vaikka pipeline-ympäristön alustus kestää useita minuutteja, muut testauksen vaiheet säästävät silti huomattavasti enemmän aikaa. Ohjelmistopäivityksen kesto on, kaiken sujuessa hyvin, sama kuin manuaalisessa testauksessa. Jos kuitenkin ilmenee ongelmia, automaattitestit ovat nopeampia, koska kaikki tarvittavat resurssit ladataan valmiiksi jokaista testiä varten riippumatta siitä, käytetäänkö niitä vai ei. Tämä vähentää ylimääräistä viivettä päivityksen aikana. Lisäksi ohjelmistopäivityksen tekevän ohjelman CLI-versio on käyttöliittymää nopeampi.

Automaattitestien suorittaminen on nopeampaa ja tehokkaampaa, sillä tietokone ei tarvitse ylimääräistä aikaa komentojen syöttämiseen, ja tulosten tarkistus sujuu nopeasti. Tulosten tallennus kestää yleensä vain muutamia kymmeniä sekunteja, ja aika voi vaihdella tallennuskohteen mahdollisen ruuhkan mukaan. Tallennus ei kuitenkaan koskaan kestä yli 30 sekuntia.

Testiympäristön alasajo on nopea prosessi, johon sisältyy avointen yhteyksien sulkeminen, debug- ja käynnistysdatan tallentavien laitteiden sammutus sekä koko järjestelmän alasajo. Automaattitestauksessa raportin analysointi voi kestää kauemmin kuin manuaalitestauksessa, koska raportti on käytävä läpi kokonaisuudessaan, ja sieltä voi olla haastavaa tunnistaa, missä vaiheessa testit ovat epäonnistuneet.

Vaikka testausprosessi on lyhyempi automatisoidussa ympäristössä kuin manuaalisessa, automaattitestijärjestelmien käyttöönotossa on huomioitava myös niiden suunnittelu, toteutus ja ylläpito. Erityisesti suuret ja monimutkaiset järjestelmät

vaativat paljon aikaa ja resursseja suunnitteluun ja rakentamiseen. Testijärjestelmän valmistuttua on luotava kattavat testit, ja vasta testispesifikaation laatimisen jälkeen voidaan aloittaa varsinaisten testien kehittäminen.

Kun testijärjestelmien ajettavat testit on luotu, on ne vielä muokattava automaattisesti ajettaviksi. Tämä toteutetaan yleensä jonkin pilvipalvelupohjaisen pipeline-työkalun, kuten Microsoft Azuren tai GitHub Actionsin avulla. Kun tämä vaihe on valmis, automaattisia testijärjestelmiä voidaan alkaa käyttämään. Testijärjestelmien käyttöönottamisen jälkeen niiden ylläpito on suhteellisen vaivatonta.

Järjestelmien ylläpitoon kuuluu ohjelmistojen ja järjestelmien päivittäminen, satunnaiset huoltotoimenpiteet sekä rikkinäisten laitteiden vaihto. Haasteita ylläpidossa aiheuttavat erityisesti testattavat ohjelmistot, jotka rikkovat ohjainlaitteita tai muita osia, sekä tilanteet, joissa jokin laite saavuttaa elinkaarensa lopun ja vaatii vaihtamista. Vaikka tällaiset tilanteet ovat harvinaisia, niiden korjaaminen voi olla työlästä, erityisesti jos testijärjestelmä on rakennettu niin, että esimerkiksi ohjelmistopäivitysadapterin liittimelle ei pääse helposti käsiksi.

5.2 Testausympäristön arviointi

Opinnäytetyössä käytetty testauskehys on tehokas ja käyttäjäystävällinen testausympäristö, joka soveltuu erinomaisesti erilaisten testien automatisointiin. Sen käyttöönottaminen on helppoa, ja pelkkä testauskehyksen perustana olevan ohjelmointikielen käyttökokemus riittää alkuun pääsemiseen. Hyvä dokumentaatio sekä lukuisat verkosta löytyvät esimerkit tekevät oppimisesta vaivatonta. Vaikka testauskehys on yksinkertainen työkalu, se on silti erittäin monipuolinen. Testien ohjelmointiin on useita vaihtoehtoja: testejä voi kirjoittaa yksittäisinä Test Case-muotoisina tai hyödyntää mallipohjia (template), jotka mahdollistavat saman tyylisten testien ajon parametrien avulla samasta mallipohjasta. Tämä joustavuus auttaa optimoimaan testausprosessia ja parantamaan koodin uudelleenkäytettävyyttä.

Testauskehyksen käyttöön liittyvä merkittävä etu on resurssitiedostojen hyödyntäminen. Resurssitiedostoon kerätään tyypillisesti laitekohtaiset muuttujat, testauksessa tarvittavat avainsanat sekä muut testiympäristöön liittyvät asiat, jotka voivat vaihdella sen mukaan, millä järjestelmällä testejä ajetaan. Tämä mahdollistaa yhden yhteisen resurssitiedoston luomisen kaikille Smoke-testijärjestelmille. Resurssitiedostoon on myös mahdollista ohjelmoida toiminnallisuus, jonka avulla avainsana tunnistaa käytössä olevan laitteen. Tämän ansiosta uusilla, juuri rakennetulla järjestelmillä on mahdollista suorittaa myös aiemmin kehitettyjä testitiedostoja ilman ongelmia.

Kehittäessämme Smoke-testijärjestelmien ohjelmistopäivitystiedostoja havaitsimme merkittävän haasteen, joka liittyi ulkoisten CLI-ohjelmien yhteensopivuuteen. Vaikka testauskehys toimii ohjelmointikielen päällä, ei se tue täysin ohjelmointikielelle tarkoitettua ohjelmakoodia, vaan testauskehykselle on kehitetty oma syntaksi. Tämä tarkoitti, että kun CLI-ohjelmaa kutsuttiin useammalla kuin yhdellä asetuslipulla samaan aikaan, esimerkiksi jos samaan komentoon piti liittää laitemääritelmä, ohjelmistotiedosto ja ohjelmistotiedoston apuparametrit. Tämä lähestymistapa oli haastava testauskehyksen avulla, vaikka vastaava toteutus olisi ollut huomattavasti helpompaa pelkästään alkuperäisellä ohjelmointikielellä.

Mielestäni käytetyn testauskehyksen vaihtaminen johonkin toiseen ei olisi parantanut tai heikentänyt testien suorituskykyä. Kaikissa ohjelmointikielien päälle rakennetuissa testauslisäosissa on omat haasteensa. Käytetty testauskehys on kuitenkin vakaa ja monikäyttöinen työkalu, jota voidaan hyödyntää erilaisissa testaustarkoituksissa.

5.3 Automatisoidun testauksen kehitys

Automatisoiduissa testeissä on lähes aina mahdollisuuksia kehittämiselle. Testien kattavuutta voidaan laajentaa esimerkiksi testaamalla sovellusten eri osa-alueita

kattavammin. Tämä voidaan saavuttaa lisäämällä uusia testitapauksia tai laajentamalla olemassa olevia testityyppejä, kuten suorituskyky- tai kuormitustestejä. Lisäksi testien ylläpidettävyyttä voidaan parantaa selkeyttämällä testiskriptejä, päivittämällä olemassa olevaa dokumentaatiota tai muokkaamalla koodia modulaarisempaan muotoon, mikä helpottaa sen uudelleenkäyttöä eri testitilanteissa.

Opinnäytetyöprojektin yhtenä tavoitteena oli testien suoritusajan optimointi. Tämä toteutettiin luomalla ohjelmiston päivitysprosessi erityisesti näille testijärjestelmille. Tällä tavalla vältettiin yleiskäytössä olevan prosessin käyttö, joka olisi yrittänyt päivittää myös kortteja, joita ei ollut yhdistetty testijärjestelmään. Lisäksi testijärjestelmien kuormitusta pyrittiin optimoimaan nostamalla testijärjestelmien lukumäärä vastaamaan lisääntynyttä kuormitusta. Tämä muutoksen myötä testijärjestelmille ei synny yhtä paljon ruuhkaa kuin aikaisemmin, erityisesti tapauksissa, joissa useat ohjelmistokehittäjät yrittävät tallentaa omia muutoksiaan koodivaraston päähaaraan. Vanhoissa järjestelmissä ei ollut tarpeeksi resursseja ja ne suorittivat tarvittavia testejä hitaammin, mikä johti pitkiin jonotusaikoihin.

Pipeline-järjestelmässä havaittiin myös mahdollisia ongelmia. Pipeline järjestää pipeline-agentit aakkosjärjestykseen ja ohjaa testit aina ensimmäiselle vapaalle agentille tämän listan perusteella. Tämä tarkoittaa, että ruuhka-aikoina listan alkupään laitteet ovat jatkuvassa käytössä, vaikka listan loppupään laitteita ei välttämättä hyödynnetä lainkaan. Lisäksi vaikka kaikki testijärjestelmät on rakennettu samalla tavalla ja niille on asennettu samat ohjelmistot ja lisäosat, on silti mahdollista, että yksi järjestelmä päästää joitain vikoja läpi, kun taas toinen ei. Tämä voi aiheuttaa merkittäviä ongelmia erityisesti, jos viallinen laite kuuluu aakkosjärjestyksessä listan alkuun.

Vaikka tällainen tilanne on hypoteettinen, eikä sitä oleteta tapahtuvan, olisi silti suositeltavaa jakaa laitteiden kuormitus tasaisesti kaikkien laitteiden kesken.

Ongelman ratkaisemiseksi voitaisiin edelleen käyttää aakkosjärjestyksessä olevaa listaa, mutta sen sijaan, että uusi testi ohjattaisiin aina listan ensimmäiselle agentille, voitaisiin toteuttaa vuoronumeropohjainen järjestelmä. Tämä järjestelmä määrittäisi jonon ensimmäisen testin suoritettavaksi edellisen testin suorittaneesta järjestelmästä seuraavalle. Ruuhkatilanteissa, joissa kaikki järjestelmät ovat käytössä, testit ohjattaisiin aina ensimmäiselle järjestelmälle, jonka edelliset testit valmistuvat.

5.4 Työn johtopäätökset ja yhteenveto

Opinnäytetyöprojektissa päästiin osittain projektille asetettuihin tavoitteisiin. Testijärjestelmien rakennusvaihe on suoritettu, jonka ansiosta testijärjestelmien lukumäärä vastaa nyt lisääntynyttä kuormitusta. Testijärjestelmien ohjelmistopäivitykseen tarkoitettu skripti on myös kehitetty. Sen avulla pipelinen testiympäristön alustus saatiin optimoitua nopeammaksi kuin aikaisemmin. Smoke-testijärjestelmien pipeline on kehitetty manuaali-pipelineksi ja sen toiminnallisuudet on tarkistettu.

Opinnäytetyöprojekti on vielä kesken, mutta tähän mennessä tehty työ tarjoaa erittäin hyvän pohjan työn jatkamiselle ja raportin perusteella kehityksen suunta on selkeä. Seuraavaksi kehitystyö projektissa keskittyy manuaalisen pipelinen muutokseen automaattiseksi ja sen integroimiseen vanhojen Smoke-testijärjestelmien rinnalle. Testijärjestelmien vertailu rinnakkain auttaa selvittämään, millaisia testejä uusille järjestelmille tullaan kehittämään. Vertailu parantaa myös käsitystä prosessin kestosta ja siitä, paljonko prosessien optimointi vaikutti testien kestoon. Mahdolliset ongelmat tullaan myös ottamaan huomioon ja selvittämään, onko käytetyssä pipeline-ympäristössä ominaisuutta kuormittaa järjestelmiä tasasisemmin ja hallitusti.

LÄHTEET

Ammann, P., & Offutt, J. (2008). *Introduction to software testing*. Cambridge University Press.

Chauhan, V. K. (2014). Smoke Testing. *International Journal of Scientific and Research Publications*. Viitattu 10.9.2024 osoitteesta https://www.ijsrp.org/research-paper-0214.php?rp=P262302

Danfoss. (2024). Danfossin kotisivut. Viitattu 2.9.2024 osoitteesta https://www.danfoss.com/en/

Danfoss. (2024). Maailman ensimmäinen iC7-Marine-taajuusmuuttajasta tehonsa saava lautta: Aurora Botnia. Viitattu 2.9.2024 osoitteesta https://www.danfoss.com/fi-fi/service-and-support/case-studies/dds/danfoss-ic7-series-powers-wasaline-hybrid-electric-ropax-ferry/

Duvall, M. P., Matyas, S., & Glover, A. (2007). *Continuous integration: Improving software quality and reducing risk*. Pearson Education.

Gebhardt, A., Kessler, J., & Thurn, L. (2019). *3D printing: Understanding additive manufacturing* (toinen painos). Hanser Publications.

GitHub Actions. (2024). GitHub Actions kotisivut. Viitattu 22.11.2024 osoitteesta https://github.com/actions/github-script

GitHub Docs. (2024). About Branches. Viitattu 24.9.2024 osoitteesta https://docs.github.com/en/enterprise-cloud@latest/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches

Kaner, C., Bach, J., & Pettichord, B. (2002). *Lessons learned in software testing*. Wiley Computer Publishing.

Matt Ahart. (2019). Types of 3D printing. Protolabs. Viitattu 15.10.2024 osoitteesta https://www.protolabs.com/resources/blog/types-of-3d-printing/

Microsoft Azure. (2024). Azuren kotisivut. Viitattu 10.9.2024 osoitteesta https://azure.microsoft.com/en-us

Microsoft Azure. (2024). Key concepts for new Azure Pipelines users. Viitattu 2.10.2024 osoitteesta https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/key-pipelines-concepts?view=azure-devops

Microsoft Azure. (2024). What is Azure CLI. Viitattu 14.10.2024 osoitteesta https://learn.microsoft.com/en-us/cli/azure/what-is-azure-cli

Microsoft Azure. (2024). Create and manage agent pools. Viitattu 16.10.2024 osoitteesta https://learn.microsoft.com/en-us/azure/devops/pipelines/agents/pools-queues?view=azure-devops&tabs=yaml%2Cbrowser

Microsoft Azure. (2024). PowerShell@2 – Powershell v2 task. Viitattu 16.10.2024 osoitteesta https://learn.microsoft.com/en-us/azure/devops/pipelines/tasks/reference/powershell-v2?view=azure-pipelines

Python Software Foundation. (2024). Python.org. kotisivut. Viitattu 14.10.2024 osoitteesta https://www.python.org