OAMK OULUN AMMATTIKORKEAKOULU

Petteri Kivelä

CI-JÄRJESTELMÄN KÄYTTÖÖNOTTO YRITYKSESSÄ

Jatkuvan integraation järjestelmä tukiasematesterin ohjelmistokehitykseen

CI-JÄRJESTELMÄN KÄYTTÖÖNOTTO YRITYKSESSÄ

Jatkuvan integraation järjestelmä tukiasematesterin ohjelmistokehitykseen

Petteri Kivelä
Opinnäytetyö
Kevät 2023
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, Ohjelmistokehityksen suuntautumisvaihtoehto

---

Tekijä(t): Petteri Kivelä
Opinnäytetyön nimi: CI-järjestelmän käyttöönotto yrityksessä
Työn ohjaaja(t): Teemu Leppänen
Työn valmistumislukukausi ja -vuosi: Kevät 2023 Sivumäärä: 33

Opinnäytetyön taustana oli työnantajayrityksen tarve ottaa käyttöön jatkuvan integraation järjestelmä ohjelmistokehityksessä. Työn aikana hankittua tietoa ja osaamista oli tarkoitus hyödyntää myös tulevissa projekteissa ja lisäksi olla tukena muulle ohjelmistokehitystiimille. Tavoitteena työssä oli luoda toimiva CI-järjestelmä tukiasematesterin ohjelmistokehitykseen ja aloittaa testien kehitys järjestelmän toimivuuden varmistamiseksi.

Työn teoriaosuudessa tutustutaan ensin jatkuvaan integraatioon kuuluviin asioihin ja vaiheisiin. Testiautomaatio on tärkeimpiä asioita jatkuvassa integraatiossa. Testiautomaation vaatimista resursseista, tavoitteista ja haasteista kerrotaan työn alkuvaiheessa. Lyhyesti käydään läpi myös erilaisia testaustapoja ja -vaiheita. Työn aikana tutustuttiin kahteen CI-alustaan, joiden ominaisuuksia ja soveltuvuutta yrityksen käyttöön tutkittiin. Teoriaosuudessa esitellään yleistä tietoa näistä alustoista.

Seuraava oleellinen osa jatkuvaa integraatiota on versionhallinta. Versionhallintaan liittyy tarvittavia toimintatapoja ohjelmistokehittäjiltä riittävän hyödyn saamiseksi ja CI-järjestelmän toiminnan varmistamiseksi. Työssä esitellään tämän hetken suosituin versionhallintajärjestelmä Git ja kerrotaan sen toiminnasta.

Tutkittaviksi CI-alustoiksi valikoituivat Jenkins ja GitHub Actions. Jenkinsiä tutkittiin ensisijaisena vaihtoehtona sen yleisyyden ja monipuolisuuden ansiosta, mutta lopulta päädyttiin käyttämään GitHub Actionsiä. GHA on suosiotaan kasvattava ja helppokäyttöinen GitHubiin integroitu CI-palvelu. GitHub on Git-koodirepositorioiden tallennuspaikka. Vaatimuksena CI-järjestelmälle olikin yhteensopivuus GitHubin kanssa, joka oli helpoin toteuttaa GHA:n avulla. Järjestelmän muina vaatimuksina oli mahdollisuus kehittää yksikkötestejä Visual Studio-ympäristössä kehitetylle tukiasematesterin ohjelmistolle.

Työn toteutusvaiheessa aloitettiin GitHub Actionsin käyttö yrityksen organisaatiolle luodulla GitHub-tilillä ja testattiin ohjelmistokoodin rakentamista GHA:n testiympäristössä. Seuraavana vaiheena perehdyttiin yksikkötestien kehitykseen Visual Studiolla ja tukiasematesterin testaamiseen vaativiin asioihin. Testiautomaation kehittämiseen sisältyi runsaasti opeteltavaa aikaisemman kokemuksen puuttuessa, mutta tavoitteen mukaisesti testien kehitys päästiin aloittamaan. Loppuvaiheessa perehdyttiin ohjelmiston vaatimien ohjelmistopakettien hallintaan GitHub Packages -palvelun avulla.

Asiasanat: Testiautomaatio, ohjelmistokehitys, jatkuva integraatio, versionhallinta, ohjelmiston laatu

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Option of Software Development

---

Author(s): Petteri Kivelä
Title of thesis: Implementation of CI system in a company
Supervisor(s): Teemu Leppänen
Term and year when the thesis was submitted: Spring 2023
Number of pages: 33

The background for the thesis was the employer's need to implement a continuous integration system for software development. The knowledge and skills acquired during work were intended to be used in future projects and to support other members of the software development team. The goal of the work was to create a functional CI system for the development of a base station tester and to start developing tests to ensure the system's functionality.

This thesis includes a theoretical section and an empirical section. The theoretical section provides an overview of continuous integration, explaining its key features and concepts. The empirical section delves into the planning and requirements of a continuous integration system and finally implementation of the system.

The requirement for the CI system was compatibility with GitHub repositories, which was easiest to achieve with GitHub Actions. Creating unit tests for base station tester software was successfully started with MSTest-framework as was requested by the employer. CI system is automatically triggered by software commits to the repository.

Earlier studies didn't include continuous integration related topics so in the beginning of the thesis there was a lot of new things to learn. Creating test automation was also challenging due to lack of previous experience. In future, this thesis will help the company's employers to continue using CI system and create unit testing for different projects.

Keywords: Test automation, software development, continuous integration, version control, software quality

SISÄLLYS

1 JOHDANTO 6
2 JATKUVA INTEGRAATIO OHJELMISTOKEHITYKSESSÄ 7
2.1 Testiautomaatio 7
2.2 Ohjelmistokoodin jatkuva integraatio 9
2.2.1 Jenkins 11
2.2.2 GitHub Actions 13
2.3 Versionhallinta 16
3 JÄRJESTELMÄN SUUNNITTELU 19
3.1 Käyttöönoton suunnittelu 19
3.2 CI-järjestelmän vaatimukset 20
3.3 Järjestelmäarkkitehtuuri 22
4 JÄRJESTELMÄN TOTEUTUS 24
5 POHDINTA 31
LÄHTEET 32

1 JOHDANTO

Jatkuva integraatio on ohjelmistokehitykseen liittyvä toimintatapa, jossa uutta ohjelmistokoodia liitetään valmiiseen ohjelmistotuotteeseen. Tulevan ohjelmistokoodin toimivuus varmistetaan automatisoiduilla testeillä ennen sen päivittämistä versionhallintajärjestelmään. Toimintatapa parantaa ohjelmiston laatua ja sen avulla voidaan löytää mahdolliset ohjelmistovirheet mahdollisimman pian.

Opinnäytetyön tarkoituksena on rakentaa työnantajayrityksen käyttöön jatkuvan integraation järjestelmä, joka testaa ohjelmistokoodia automatisoidusti sitä versionhallintaan päivitettäessä. Järjestelmä liittyy yrityksessä kehitettävään tukiasematesteriin ja siihen liittyvään ohjelmistoon. Tukiasematesteri on rakennettu räkkiin, johon kuuluu testeri-PC. PC (engl. personal computer) suorittaa testisekvenssejä, jotka testaavat tukiasemaa erilaisilla mittauksilla. Testisekvensseihin liittyy runsaasti ohjelmistokoodia, jonka automatisoitu testaaminen on tärkeää ennen sen käyttöönottoa virheiden ja yhteensopivuusongelmien havaitsemiseksi.

Opinnäytetyön teoriaosuudessa käsitellään jatkuvan integraation ja versionhallinnan käsitteitä sekä niiden merkitystä ohjelmistokehitykselle. Ensimmäisenä on syytä ymmärtää jatkuvan integraation ja versionhallinnan hyödyt sekä niihin liittyviä toimintamalleja. Työssä käsitellään yleisellä tasolla jatkuvaa integraatiota prosessina sekä ohjelmistokehitystiimiltä tarvittavia toimintatapoja, jotta jatkuvan integraation ohjelmistokehitys toimisi ongelmitta. Teoriaosuudessa esitellään opinnäytetyössä käytettäviä teknologioita sekä kerrotaan niihin kuuluvista ominaisuuksista ja erilaisista tavoista hyödyntää niitä. Opinnäytetyön työvaiheen alussa testausautomaation alustaksi, eli palvelimeksi, verrataan eri vaihtoehtoina avoimen lähdekoodin Jenkinsiä ja GitHubiin integroitua GitHub Actionsiä. Versionhallintajärjestelmänä käytetään Gitiä ja versionhallinnan käyttöliittymänä ja tallennuspaikkana GitHubia.

Teoriaosuuden jälkeen esitellään järjestelmän käyttöönoton vaatimukset sekä niiden pohjalta tehty suunnitelma. Työvaiheessa pyritään löytämään paras tapa hyödyntää käytettävissä olevia teknologioita tukiasematesterin ohjelmiston testaamiseksi. Järjestelmän käyttöönoton jälkeen aloitetaan automatisoitujen testien kehittäminen ohjelmistokoodien testaamiseksi.

2 JATKUVA INTEGRAATIO OHJELMISTOKEHITYKSESSÄ

Tässä luvussa esitellään jatkuvaan integraatioon kuuluvia tekijöitä sekä toimintatapoja, joilla järjestelmä saadaan toimimaan sujuvasti.

2.1 Testiautomaatio

Ohjelmistokehityksessä testaus on yleisin tapa ohjelmistokoodin laadunvarmistukseen ja -hallintaan ja se on tärkeä osa kehitysprosessia. Testauksen ansiosta saadaan tietoa ohjelmiston toiminnasta käytännössä verrattuna sille asetettuihin vaatimuksiin. Ohjelmistojen testaus manuaalisesti on kallista ja aikaa vievää. Tätä voidaan hallita testiautomaation avulla, jossa ohjelmisto testataan automatisoiduilla testeillä ihmisen sijasta. Kokonaisuutena testiautomaatio on iso investointi ohjelmistokehitykseen liiketoiminnan näkökulmasta. Tämä investointi ei välttämättä maksa itseään takaisin, jos testausjärjestelmä ei toimi odotetulla tavalla. (1.)

Testiautomaatiota voidaan käyttää testausprosessin vaikuttavuuden parantamiseksi vähentämällä riskiä ihmisen tekemille virheille ja tekemällä testeistä toistettavia lyhyessä ajassa tai testausprosessin tehokkuuden parantamiseksi vapauttamalla henkilöstöä tärkeämpiin tehtäviin. Testiautomaatio on ketterän ohjelmistokehityksen tärkeimpiä komponentteja, jonka ansiosta ohjelmistokehittäjät saavat välitöntä palautetta tekemänsä ohjelmiston toiminnasta. Testiautomaation aktiivinen käyttö tekee siitä kriittisen osan ohjelmistokehitystä: jos automaatio lakkaa toimimasta, koko kehitysprosessi hidastuu tai pysähtyy. (1.)

Suurimpia riskejä testiautomaation kehittämisessä on projektille asetetut epärealistiset odotukset ja mahdollinen automaation hylkääminen, jos toivottuja tuloksia ei saavuteta odotetussa ajassa. Kun investoidaan testiautomaation kehittämiseen, on muistettava, että se ei tuota kovin nopeasti ja järjestelmän ylläpidettävyyden takia tuloksia on odotettava pitkäjänteisesti. Hyvin suunniteltu strategia testiautomaatioon on tärkeää projektinhallinnan kannalta ja jotta ymmärretään, mitä järjestelmällä voidaan tehdä ja mitä ei. Jos testiautomaatio ei vastaa odotuksia ja tuottaa pettymyksen projektin sidosryhmille, se voidaan nähdä turhaksi toiminnaksi, vaikka se tuottaisikin arvoa organisaatiolle. (1.)

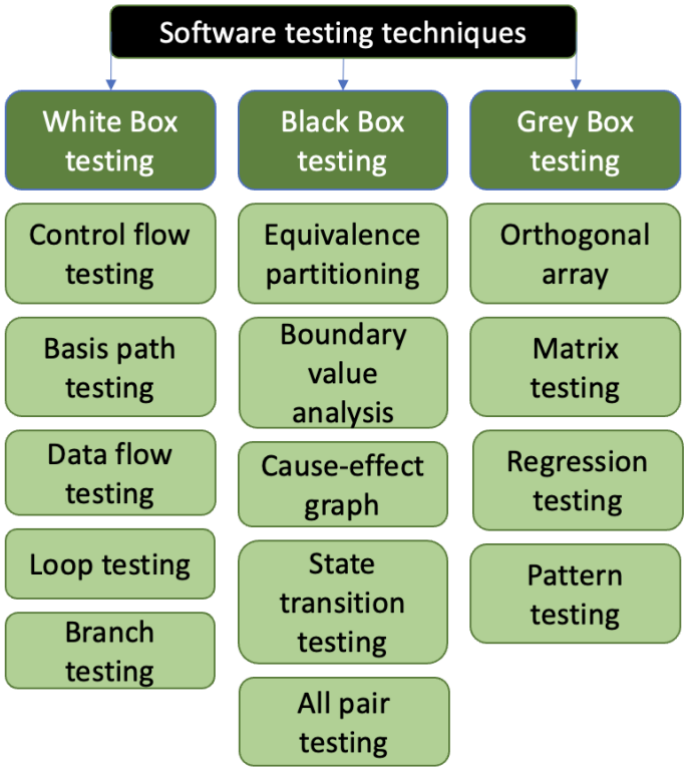
Testiautomaation kehittäminen vaatii osaavaa henkilöstöä, joilla on kokemusta ohjelmistokehityksestä ja testauksesta sekä ymmärrys testattavan järjestelmän toiminnasta. Ohjelmistotestaajien näkökulmasta manuaalisen testaamisen korvaaminen automaatiolla vaikuttaa testaajien päivittäiseen työhön ja vaatii koulutusta ja uudenlaista priorisointia. Kuten tavallisessakin ohjelmistoprojektissa, lisäksi tarvitaan johtamis- ja suunnittelutaitoja omaavia ihmisiä. Ilman päteviä henkilöitä projektin lopputuloksena tulee usein huonompaa laatua aikataulujen ja budjettien ylittyessä. (1.)

Ohjelmistotuotetta testatessa kaikki testaukseen kuuluvat työkalut ja teknologiat yhdistetään testiympäristöön, jota ohjelmistotestaajat käyttävät. Ympäristö yhdistetään automaattisten testiskriptien kanssa, joka muodostaa testijärjestelmän. Riittävän testauskattavuuden saavuttamiseksi järjestelmän on usein oltava lähes yhtä monimutkainen kuin testattavan järjestelmän. Tästä syystä järjestelmää on suunniteltava ja kehitettävä riittävän suurella prioriteetilla. Huonolaatuinen testijärjestelmä aiheuttaa ongelmia testaajille ja lisää riskejä testatun tuotteen laadulle. Järjestelmässä ilmenevät ongelmat kasvattavat tarvetta testata ohjelmistoa manuaalisesti, mikä puolestaan vie enemmän resursseja projektilta. (1.)

Ohjelmistotestaus sisältää tiettyjä tasoja ja vaiheita. Perusvaiheisiin kuuluu yksikkötestaus, joka kuuluu yleensä ohjelmistokehittäjän vastuulle ohjelmistoon kehitettyjen uusien osien (moduulien) jälkeen. Seuraavana perusvaiheena on integraatiotestaus, jossa ohjelmistoon kehitetyt moduulit integroidaan yhteen ja toiminnallisuutta testataan yhdessä. Lopullinen vaihe on systeemitestaus, jossa ohjelmistoa testataan kokonaisuutena eri näkökulmista ja eri tilanteissa. Ohjelmistotestaus varmistaa myös, että integroidut uudet moduulit eivät häiritse muiden moduulien toimintaa. Testausta suorittavat kehittäjien lisäksi testaukseen erikoistuneet ohjelmistotestaajat. (1.)

Testauksessa merkittävimmät tekniikat ovat mustan laatikon, valkoisen laatikon ja harmaan laatikon testausmenetelmät (engl. black box, white box, gray box). Valkoisen laatikon tekniikalla testataan ohjelmiston toiminnan lisäksi myös sen sisäisen rakenteen toimivuutta. Testien tekeminen valkoisen laatikon tekniikalla vaatii myös ohjelmointitaitoja, koska siinä testataan koodin eri osien toimivuutta hyvin yksityiskohtaisesti. Tätä tapaa voidaan hyödyntää kaikilla testaustasoilla. Mustan laatikon tekniikassa testataan ohjelmiston toimivuutta yleisesti syventymättä toteutustasolla sen yksityiskohtiin. Tällä tavalla saadaan selville, täyttääkö ohjelmiston ominaisuudet sen tulevan käyttäjän asettamat vaatimukset. Musta laatikko on yksinkertaisin ja yleisin testaustapa. Harmaan laatikon tekniikka on aiemmin mainittuja tapoja

yhdistelevä menetelmä. Harmaan laatikon tekniikassa testataan sovelluksen toimivuutta, mutta tässä menetelmässä testaaja on tietoinen ohjelmiston sisäisestä rakenteesta ja ottaa sen huomioon testauksessa. Kuvassa 1 esitellään eri testausmenetelmiä ja niihin kuuluvia tekniikoita (1.)



KUVA 1. Ohjelmistotestauksen eri tekniikoita (1)

2.2 Ohjelmistokoodin jatkuva integraatio

Jatkuva integraatio (engl. Continuous Integration, CI) on ohjelmistokehityksen toimintamalli, jossa ohjelmistokehittäjät integroivat luomaansa koodia toimivaan ohjelmistotuotteeseen. Koodia voidaan integroida projektiin tarvittaessa useita kertoja päivässä. Jokaisen integraation toimivuus varmistetaan automatisoiduilla testeillä virheiden löytämiseksi. Järjestelmän toiminnan kannalta kehitystiimin jäsenten välinen kommunikaatio on tärkeää, jotta toimivaan ohjelmistoon tehdyistä muutoksista ollaan tietoisia tiimin sisällä. Perimmäinen tarkoitus jatkuvassa integraatiossa on välittömän palautteen saaminen. (2.)

Suurin hyöty jatkuvan integraation toimintamallissa on riskien vähentäminen. Järjestelmän ansiosta voidaan aina tietää, missä tilassa ohjelmisto on, mitkä asiat toimivat ja mitkä eivät. Virheet

ohjelmistossa usein sotkevat aikatauluja ja aiheuttavat lisätyötä kehittäjille, koska virheet on saatava korjattua ennen siirtymistä uusien ominaisuuksien kehittämiseen. Jatkuva integraatio ei kokonaan poista ohjelmiston vikoja, mutta sen avulla ne voidaan löytää helpommin ja korjata. Ohjelmistossa olevien vikojen määrä lisääntyy usein kumulatiivisesti, joten niiden määrän kasvaessa niiden korjaaminen on entistä vaikeampaa. Jatkuvan integraation ansiosta vikojen määrä voi vähentyä merkittävästi. Tämä tosin riippuu vahvasti koodille tehtyjen testien laadusta. (2.)

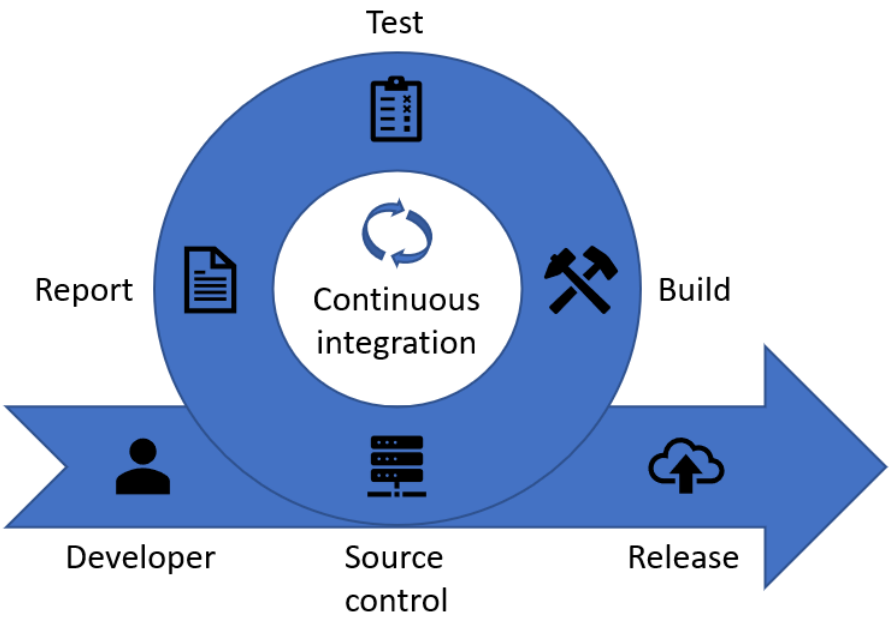
CI-järjestelmä vastaa ketterän ohjelmistokehityksen vaatimuksiin. Muutokset voidaan integroida nopeasti, mikä mahdollistaa esimerkiksi nopean reagoinnin asiakkaan muuttuviin vaatimuksiin. Järjestelmää voisi kutsua prosessin automatisoinniksi, jossa ihmisen tekemän ohjelmistotestauksen sijasta kehitysympäristöön sisällytetään automatisoidut mekanismit suorittamaan testaus. Ohjelmistosuunnittelussa jatkuva integraatio nähdään jo yhtä tärkeänä kuin versionhallinta. Ilman CI-järjestelmää ohjelmistokoodi voidaan katsoa toimimattomaksi, ennen kuin se on testattu manuaalisesti ja integroitu ohjelmistoprojektiin. Jatkuvalla integraatiolla sekä riittävän kattavilla testeillä ohjelmistomuutokset voidaan todistaa toimivaksi. (3.)

Jatkuvan integraation järjestelmä pyrkii automaattisesti rakentamaan ja asentamaan ohjelmiston ajoympäristössä, missä ohjelmistolle tehdään kokoelma automatisoituja testejä aina ennen, kun ohjelmistokoodi päivitetään versionhallintaan. Tämän koko prosessin automatisoiminen lisää tuottavuutta ja ohjelmistopäivitysten laatua, sillä jatkuvan integraation avaintekijä on lopulta testaaminen yksikkö- ja integraatiotestein. (3.)

Yksi suurimpia haasteita jatkuvan integraation käyttöönotossa on kunnollisen testaustrategian luominen. Automatisoidut testit ovat tärkein osa jatkuvaa integraatiota ja riittävä testien automatisoiminen voi usein epäonnistua. Testivetoinen kehitys helpottaa automaation kehittämistä, mutta ohjelmistokoodin osien ja niistä vastaavien testien väliset riippuvuudet tekevät integraatiosta monimutkaista. Testiautomaation lisäksi tarvitaan usein ohjelmiston laadusta vastaavia henkilöitä, jotka testaavat ohjelmistoa manuaalisesti sen luotettavuuden varmentamiseksi. (4.)

Toinen merkittävä haaste järjestelmän käyttöönotossa on testien huono laatu. Esimerkkejä huonosta laadusta ovat epäluotettavat testit, testitapausten suuri määrä, matala testikattavuus ja liian pitkäkestoiset testit. Nämä laatuongelmat vähentävät käyttöönoton ongelmien lisäksi organisaatioiden luottamusta automatisoituun ohjelmistokoodin integrointiin. Suuressa

koodikannassa testien määrä kasvaa nopeasti ja pitkäkestoinen testivaihe hidastaa kehittäjän saamaa palautetta. (4.)

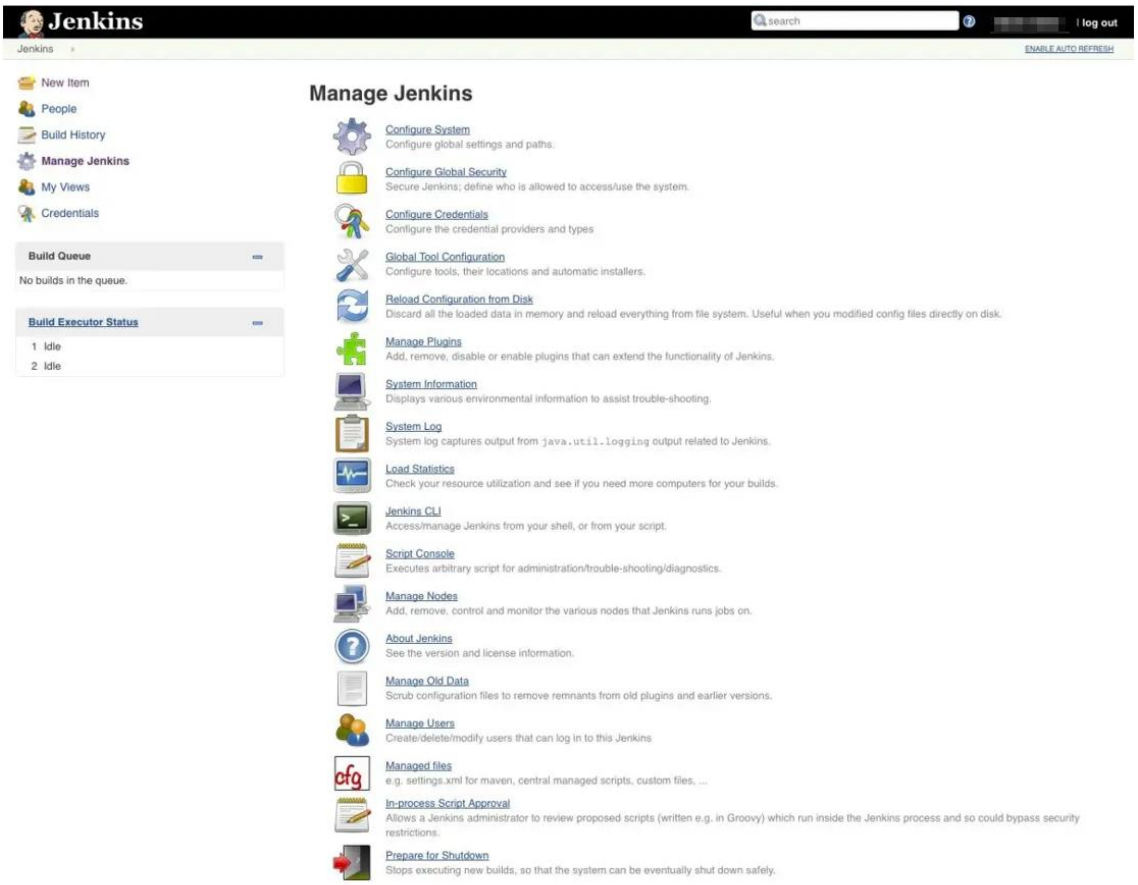


KUVA 2. Yksinkertainen kaavio jatkuvan integraation prosessista (4)

Jatkuvaa integraatiota varten tarvitaan palvelin, joka havaitsee ohjelmistokoodien säilytyspaikkaan eli repositorioon tehdyt muutokset ja rakentaa ohjelmistoprojektin palvelimelle. CI-palvelimien avulla on mahdollista eri tavoin käynnistää ohjelmistoprojektin rakennus ja yksikkötestien suoritus. Jenkins on viime vuosina ollut suosituin vaihtoehto jatkuvan integraation palvelinohjelmistoksi. Aliluvuissa esitellään Jenkinsin lisäksi GitHub-palveluun integroitu GitHub Actions.

2.2.1 Jenkins

Jenkins on avoimen lähdekoodin palvelinohjelmisto, jota käytetään jatkuvaan integraatioon monen kokoisissa organisaatioissa. Jenkins on alustariippumaton ja se on kehitetty Java-ohjelmointikielellä. Jenkins tuo monenlaisia mahdollisuuksia automaation kehittämiseen, koska se on laajennettavissa erilaisilla laajennusosilla. Jenkins-yhteisö on kehittänyt asennuspaketit kaikille suosituimmille käyttöjärjestelmille. (5, s. 3.)



KUVA 3. Kuvakaappaus Jenkinsin käyttöliittymästä

Jenkins tarjoaa yksinkertaisen tavan luoda ympäristö jatkuvalle integraatiolle ja toimitukselle, joka tukee erilaisten ohjelmointikielten ja koodirepositorioiden yhdistelmiä. Se ei poista tarvetta tehdä komentoskriptejä yksittäisiä vaiheita varten, mutta se helpottaa integraatioon tarvittavien ohjelmistojen ja työkalujen yhdistämistä. (6.)

Jenkins-projektin aloitti Kohsuke Kawaguchin vuonna 2004 jatkuvan integraation ja testaamisen mahdollistamiseksi. Nykyään Jenkinsillä on mahdollista organisoida ohjelmistokehityksen koko prosessi, jota kutsutaan jatkuvaksi toimitukseksi (engl. Continuous Deployment, CD). Jenkinsistä on tullut suosituin vaihtoehto sen laajennettavuutensa ja aktiivisen yhteisönsä ansiosta. Sille on kehitetty yli 1600 laajennusosaa, joilla Jenkins voidaan integroida erilaisiin käyttötarpeisiin. (6.)

Jenkinsin päätoimintaperiaate on konfiguroitava työ (engl. job), joka voi käynnistää suoritettavaksi myös muita töitä. Jokaista työn suoritusta kutsutaan koontiversioksi, jolla on oma numerotunnuksensa. Jokainen työ koostuu yhdestä tai useammasta koontiversion esi-, rakennus- tai jälkirakennusvaiheesta. Esi- ja jälkirakennusvaiheet on tarkoitettu ympäristön luomiseen ja

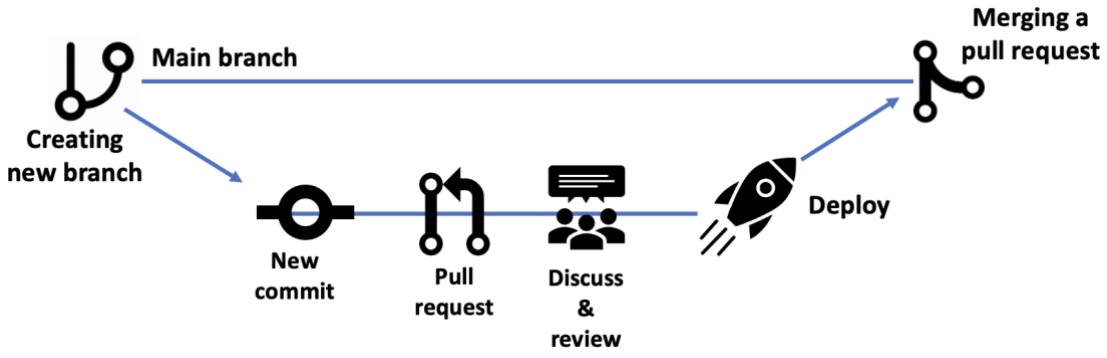
purkamiseen ohjelmiston suorittamista varten. Rakennusvaihe voi olla millainen tahansa automatisoitu tehtävä, esimerkiksi skriptin suorittaminen. Jenkins mahdollistaa myös koontiversioiden rakentamisen hajauttamisen, jossa työmäärä jaetaan ”aliympäristöille”. Tällöin yksi Jenkins-asennus voi hallinnoida useita erilaisia ympäristöjä testausta varten. Tämä mahdollistaa testien rinnakkaisen suorittamisen eri ympäristöissä, mikä nopeuttaa testausprosessia. (7.)

Jenkinsin huonoina puolina pidetään nykyään sen käyttöliittymän vanhanaikaisuutta ja ominaisuuksien monimutkaista käyttöönottoa. Monet Jenkinsille kehitetyistä laajennusosista ovat vanhentuneita eikä niitä enää ylläpidetä. Jenkinsin vanhanaikaisuuden takia siihen voi joutua asentamaan useita liitännäisiä sen modernisoimiseksi. Jenkinsin konfigurointiin ei myöskään ole ylläpidettyä YAML-käyttöliittymää, kuten useilla nykyaikaisemmilla CI/CD-alustoilla. Jenkinsiin liittyvä dokumentaatio on osin vanhentunutta, mikä hankaloittaa sen käyttöönottoa. (8.)

2.2.2 GitHub Actions

GitHub Actions on Microsoftin omistamaan GitHubiin lisätty ominaisuus, joka tuo koodirepositorioiden ylläpitäjille mahdollisuuden suorittaa automaattisia testisekvenssejä. Testisekvenssissä suoritetaan sarja käyttäjän määrittelemiä komentoja. Sekvenssi suoritetaan tietyn tapahtuman jälkeen, esimerkiksi koodirepositorioon tulleiden muutosten takia tai repositorion sivuhaaran yhdistyessä päähaaraan. Testisekvenssin komennot määritellään repositorion GitHub/workflows-hakemistossa käyttämällä YAML-syntaksia. Ohjelmistokehittäjät voivat luoda sekvenssiin omia toimintoja luomalla kustomoitua koodia, joka on vuorovaikutuksessa repositorion kanssa. Testisekvenssin suorittamisen jälkeen sen tulokset voidaan käsitellä eri tavoin. (9.)

GitHub Actions on uutena ohjelmistona vielä melko vähän käytössä verrattuna Jenkinsiin. Ohjelmistokehittäjät ovat kuitenkin ottaneet Actionsin myönteisesti vastaan, mikä ilmenee siitä käydyistä keskusteluista, joissa todetaan sen hyvien puolien ylittävän huonot puolet. Sen käyttöönotto näyttää lisänneen hylättyjen vetopyyntöjen (engl. pull request) määrää, mikä osoittaisi sen helpottavan ongelmien havaitsemista kontribuutioissa. (9.)



KUVA 4. Korkean tason kuvaus koodirepositorion muutosprosessista GitHubissa (9)

GitHubin ollessa suosituin Git-repositorioiden tallennuspaikka ei ole yllättävää, että GitHub Actions on nousemassa nopeasti suosituimpien CI/CD-palveluiden joukkoon. Tutkimukset osoittavat, että GitHub Actionsin suosion kasvu näkyy muiden CI/CD-palveluiden käytön vähenemisenä. Sen käyttöönotto on huomattavasti yksinkertaisempaa kilpailijoihin verrattuna. Sitä varten luodaan työnkulkutiedosto (engl. workflow) oikeaan kansioon repositoriossa, johon määritellään testiympäristö ja suoritettavat komennot. Työnkulussa olevia toimintoja/komentoja voidaan myös jakaa julkiseen käyttöön GitHub Marketplacessä. Merkittävä osa GitHubin repositorioista on siirtynyt käyttämään GitHub Actionsiä pääohjelmointikielestä riippumatta. Jenkinsin tapaan myös GitHub Actions tukee useita eri käyttöjärjestelmiä (10.)

Kuten muistakin CI-järjestelmistä, myös GitHub Actionsiin liittyen on raportoitu joistakin tietoturvaongelmista. Joitakin esimerkkejä ovat vetopyyntöjen manipulointi salaisuuksien varastamiseksi ja mielivaltaisen koodin syöttäminen repositorioon erilaisilla komennoilla. Varsinkin avoimen lähdekoodin julkisissa repositorioissa on oltava varovainen. Julkisista repositorioista tai GitHub Marketplacesta saatujen toimintojen käyttäminen lisää riskiä haavoittuvuushyökkäyksille. Organisaatioiden yksityisissä repositorioissa kannattaa käyttää vain luotettavaksi todettuja toimintoja. Pienentääkseen riskiä vaarantuneiden toimintojen käytölle GitHub suosittelee viittaamaan toimintoihin käyttämällä ”commit SHA” -salausta. (10.)

Kuvassa 5 esitellään GitHub Actionsin työnkulkutiedosto, joka sisältää testisekvenssin käynnistäjän (engl. trigger), testien ajoympäristön ja erilaisia testivaiheita (kuva 5). Esimerkissä sekvenssi käynnistetään muutoksien ilmestyessä suoraan repositorion päähaaraan tai sivuhaarojen yhdistyessä siihen. Esimerkin build-kohdassa havaitaan, että testit suoritetaan Windows-käyttöjärjestelmän vuoden 2019 versiossa ja siinä käytetään dotnet-kirjaston versiota 5.0.x. Env-kohdassa voidaan määritellä erilaisia polkuja repositorion ohjelmistoprojekteille. Tämän

jälkeen sekvenssissä suoritetaan varsinaiset testivaiheet. Checkout-toiminnolla repositorion kopioidaan ”työtilaan”, jotta sekvenssille saadaan pääsy repositorion tiedostoihin. Sen jälkeen ajoympäristöön asennetaan tässä esimerkissä tarvittavia ohjelmistoja kuten .NET Core ja MSBuild koontiversion suorittamiseen. GitHub Actionsillä voi suorittaa myös powershell-tiedostoja erilaisiin tarkoituksiin. NuGet-pakettienhallintaohjelman avulla asennetaan projektille tarvittavat riippuvuudet. Lopuksi dotnet build -komennolla rakennetaan repositoriossa oleva Visual Studio solution -ohjelmistopaketti ja dotnet test -komennolla suoritetaan projektille kehitetyt yksikkötestit.

```
on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
jobs:
  build:
    strategy:
      matrix:
        configuration: [Release]
        dotnet-version: ['5.0.x']
    runs-on: windows-2019
    env:
      SolutionPath: [redacted] sln
      Project_Path: [redacted] .csproj
      Test_Project_Path: [redacted] .csproj
    steps:
    - name: Checkout
      uses: actions/checkout@v2
      with:
        fetch-depth: 0
    - name: Setup .NET Core SDK ${{ matrix.dotnet-version }}
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: ${{ matrix.dotnet-version }}
    - name: Setup MSBuild.exe
      uses: microsoft/setup-msbuild@v1.1
    - name: Check no OpenTAP dll
      run: powershell -file .\Powershell\CheckNoOpenTAPDll.ps1 -TapPath D:\a\Github-Actions-
    - name: Install dependencies
      run: dotnet restore
    - name: Restore nuget packages
      run: nuget restore
    - name: Build solution
      run: dotnet build --configuration "Release"
    - name: Run unit tests
      run: dotnet test --configuration "Release"
```

KUVA 5. Esimerkki lyhyestä GitHub Actionsin työnkulkutiedostosta

2.3 Versionhallinta

Versionhallinta on oleellinen osa jatkuvaa integraatiota. Se auttaa ohjelmistokehittäjiä lähdekoodin hallinnoimisessa sekä tekemään yhteistyötä muiden kehittäjien kanssa mahdollisimman tehokkaasti. Versionhallinta on järjestelmä, jonka tehtävä on hallita ohjelmiston kehitysvaiheita eli tallentaa kehittäjien tekemät muutokset. Ohjelmiston kasvaessa ja kehittyessä koodien ja tiedostojen hallinnasta tulee vaikeampaa. Ilman versionhallintaa tietokoneella joutuisi säilyttämään koodin eri versioita ja riski väärien tiedostojen muokkaamiseen ja ohjelmiston rikkomiseen on suuri. (11.)

Versionhallintaan on kaksi erilaista lähestymistapaa, joita ovat keskitetty versionhallintajärjestelmä (CVCS, Centralized Version Control System) ja hajautettu versionhallintajärjestelmä (DVCS, Distributed Version Control System). Keskitetyssä mallissa ohjelmistokoodille on ainoastaan yksi keskeinen säilytyspaikka eli repositorio, joka toimii serverinä. Hajautetussa mallissa ohjelmistokehittäjillä on tietokoneellaan oma repositorio eli paikallinen kopio kehitettävästä ohjelmistosta. Hajautettu malli on nykyisin suositumpi tapa sekä avoimien että suljettujen lähdekoodien projekteihin. Keskitetyn mallin riskinä on mahdollinen yhteyden menettäminen serveriin, jolloin työskentely hankaloituu huomattavasti. (11.)

Versionhallinta todistetusti yksinkertaistaa ja nopeuttaa ohjelmistokehitysprosessia. Se vapauttaa kehittäjiä työskentelemään ilman pelkoa toisten tekemän koodin vahingoittamisesta. Järjestelmä havaitsee mahdolliset koodiin syntyneet konfliktit, jos kehittäjät ovat tehneet muutoksia samoihin tiedostoihin. Tehtyjen muutosten jälkeen kehittäjät tekevät luovutuksen (engl. commit) eli tallentavat uusimman version versionhallintaan. Järjestelmään jää samalla tieto muutosten tekijästä ja ajankohdasta sekä mahdollinen viesti muutoksista kehitystiimiä varten. Jos toimimaton koodi päätyy vahingossa versionhallintaan, edellisen toimivan version palauttaminen on helppoa. (11.)

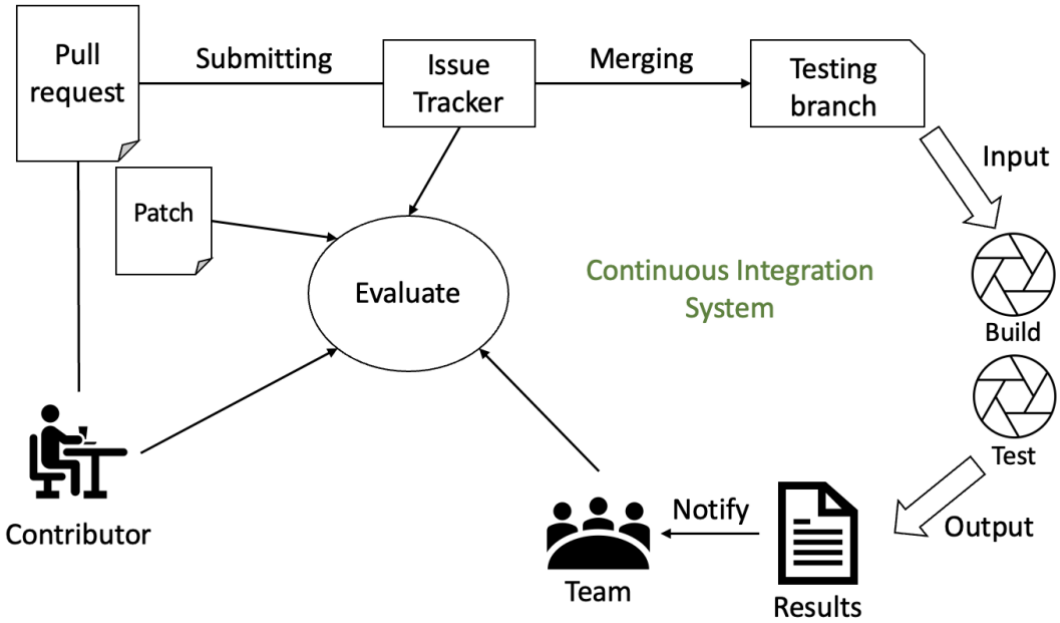
Git on ilmainen ja avoimen lähdekoodin versionhallintajärjestelmä, joka toimii Windows-, Mac- ja Linux-alustoilla. Git lisää uusia toiminnallisuuksia tietokoneen tiedostojärjestelmään erilaisilla komennoilla, jotka mahdollistavat tiedostojen ja niiden historian seurannan. Git ei muokkaa tai siirrä tiedostoja millään tavalla, joten niitä voidaan käyttää normaalisti. Kun Gitin käyttäjä tekee tiedostoihin muutoksia, niiden ”tila” voidaan tallettaa Gitin versiohistoriaan, joista edelliset versiot voidaan tarvittaessa palauttaa. Gitin suurimmat hyödyt tulevat esiin, kun projektissa on mukana

useampi henkilö, jotka voivat työskennellä samojen tiedostojen parissa ylikirjoittamatta aikaisempia versioita. Tämä vaatii jokaiselta ohjelmistokehittäjältä oman sivuhaaran luomisen repositorioon, johon muutoksien teko voidaan aloittaa. Muutoksia ei kannata tehdä suoraan repositorion päähaaraan, jotta toimimatonta ohjelmistokoodia ei vahingossa päädy toimivaan versioon. Gitiä voi käyttää graafisen käyttöliittymän kautta tai käyttöjärjestelmän komentokehotteeseen syötettävillä komennoilla. (12.)

Työn organisoiminen itsenäisiksi projekteiksi eli repositorioiksi on Gitin tärkeimpiä toimintaperiaatteita. Repositorio tarkoittaa kansiota tietokoneella, jonka tapahtumia Git-versionhallinta seuraa. Kun repositorion tiedostoihin tehdään muutoksia, Git havaitsee niiden muuttuneen niiden aikaisemmasta tilastaan. Jos tiedostojen muutoksiin ollaan tyytyväisiä, ne voidaan lisätä Gitin kokoamisalueelle (engl. staging area) ja tehdä luovutus uuteen versioon. Epätoivottujen tiedostojen päätyminen versionhallintaan voidaan estää gitignore -tekstitiedoston avulla. Epätoivottuja tiedostoja voivat olla ohjelmiston koontiversion rakentamisesta syntyneet tiedostot. Repositorioon tulisi ensisijaisesti sijoittaa sisältää vain ohjelmistokoodia sisältävät tiedostot. (12.)

Ohjelmistokehityksessä vetopohjaiset muutospyynnöt ohjelmistoprojektille ovat muodostuneet vakiintuneeksi käytännöksi. Git-versionhallinnan tuoma käytäntö mahdollistaa tiimin ohjelmistokehittäjien esittää muutosehdotuksia koodiin muokkaamatta lainkaan toimivaa ohjelmistoversiota repositoriossa. Kehittäjät voivat tehdä toimivasta versiosta paikallisen kloonin koneelleen ja muutoksien jälkeen tehdä muutospyynnön repositorion päähaaraan tekemällä vetopyynnön. Verrattuna esimerkiksi avoimen lähdekoodin yleiseen työtapaan, jossa lähetetään korjaustiedostoja ja hyväksytään postituslistojen kautta, vetopohjainen malli tuo useita etuja: tiedon keskittämisen ja korjaustiedostot sijaitsevat samassa repositoriossa versionhallinnassa. (3.)

Kuvan 6 esimerkissä ohjelmistokehittäjän projektille tekemän vetopyynnön jälkeen tiedot kirjautuvat ongelmanseurantajärjestelmään ja tehdyt muutokset yhdistetään repositorion testaushaaraan. Testaushaarassa olevasta ohjelmistosta rakennetaan koontiversio ja suoritetaan sille kehitetyt automatisoidut testit. Testeistä saadut tulokset ja raportti arvioidaan kehitystiimissä. Tiimin arvioitua raportin päätetään kehitettyjen muutosten hyväksymisestä ohjelmistoon.



KUVA 6. Yleiskatsaus vetopyynnön arviointiprosessiin (3)

3 JÄRJESTELMÄN SUUNNITTELU

Tässä luvussa käsitellään järjestelmän suunnitteluvaihetta ja sille asetettuja vaatimuksia. Lopuksi käydään läpi lopullisen järjestelmän arkkitehtuuri ja ominaisuudet.

3.1 Käyttöönoton suunnittelu

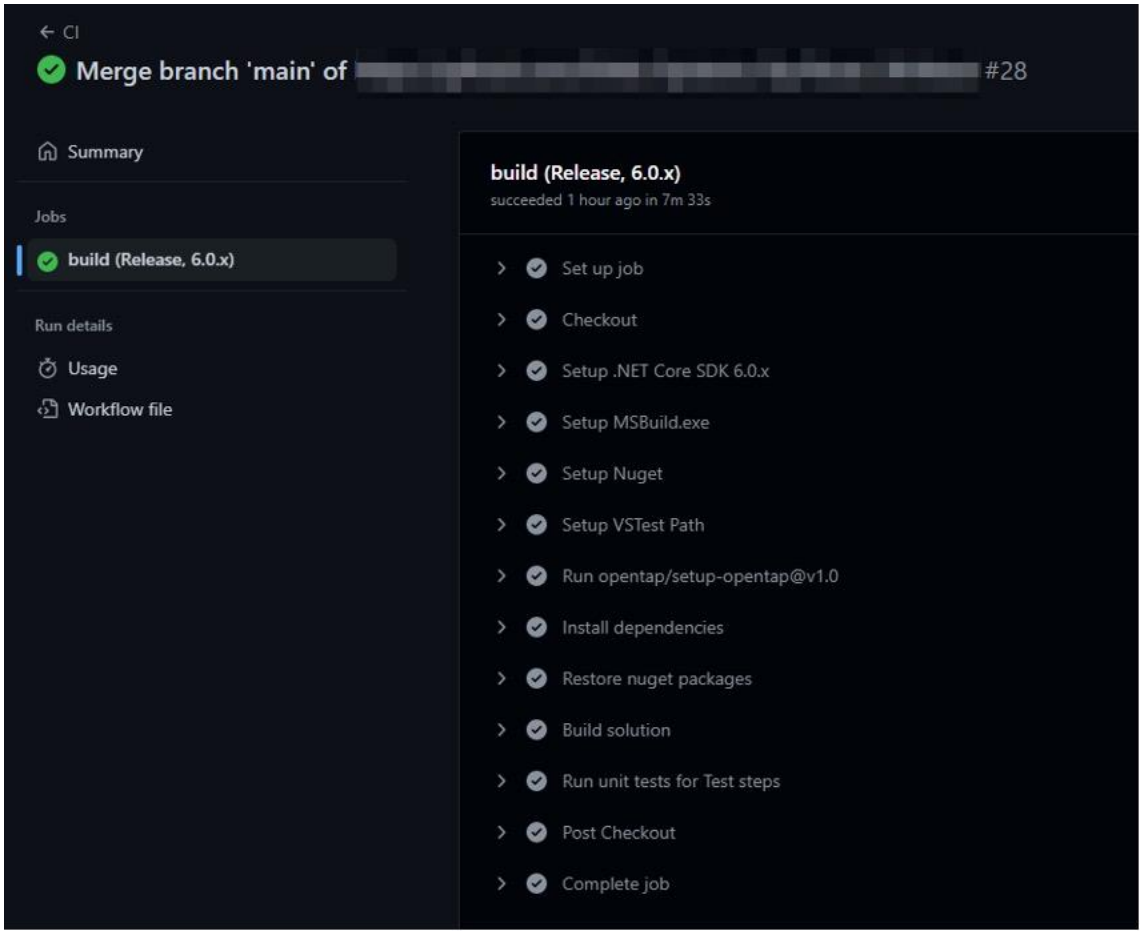
Jatkuvan integraation järjestelmän käyttöönoton suunnittelu alkoi työhön sopivien teknologioiden ja työkalujen selvittämisellä. Työn alkuvaiheessa yrityksessä käytiin läpi vaihtoehtoisia eri teknologioita ja alustoja, joille voisimme alkaa rakentamaan testausympäristöä. Selvitystyön edetessä löysimme kaksi eri vaihtoehtoa, joiden eroja selvitimme ja tutkimme niiden sopivuutta omaan ohjelmistoprojektiimme, jossa kehitämme tukiasematesterin ohjelmistoa. Ensimmäinen vaihtoehto oli Jenkins, joka perinteisempänä ohjelmistona oli tutumpi muullekin ohjelmistokehitystiimille. Toisena vaihtoehtona tutkimme GitHub Actionsiä, joka hieman tuntemattomampana ja uudempana palveluna olisi mielenkiintoinen vaihtoehto.

Suunnitteluvaiheeseen kuului lisäksi tukiasematesterin ohjelmistoon perehtyminen. Työssä täytyi selvittää, mitä tarvitaan ohjelmistokoodin rakentamiseksi testiympäristössä ja miten yksikkötestejä voidaan kehittää ohjelmistoprojektille. Yksikkötestejä oli pystyttävä suorittamaan komentokehotteen kautta, jotta komentoja voisi käyttää myös testeissä. Ohjelmistoprojektiin kuuluvien erilaisten NuGet-pakettien ja riippuvuuksien toimintaan saattaminen testiympäristössä oli suunnittelua vaativa asia.

Jenkinsin testaaminen ja automaatioserverin käyttöönotto oli huomattavasti haastavampaa, joten sen käyttöönotto opinnäytetyötä varten rajallisella aikataululla ja resursseilla olisi haastavampi tehtävä. Jenkinsiin kuuluvien laajennusosien suuri määrä voi olla ensikertalaiselle sekavaa ja niiden tehokas käyttäminen vaatisi runsaasti aikaa perehtymiseen. Jenkins olisi kuitenkin monipuolisuutensa ansiosta varteenotettava vaihtoehto. Jenkins on erittäin suosittu ja käytössä monissa yrityksissä, joten sen hallitseminen olisi hyödyllistä urakehityksen kannalta.

GitHub Actionsin käyttöönottoon riittää ainoastaan työnkulkutiedoston luominen, johon määritellään testausympäristö ja sekvenssin laukaisijat (engl. trigger). Sen jälkeen sekvenssiin

lisätään erilaisia haluttuja komentoja, joita järjestelmän halutaan suorittavan. GitHubin tarjoamien testiympäristöjen lisäksi, GHA:ssa voi käyttää myös itse luotuja ympäristöjä (engl. runner).



KUVA 7. Kuvakaappaus läpimenneestä GitHub Actionsin sekvenssistä

3.2 CI-järjestelmän vaatimukset

Vaatimuksena työssä oli ottaa käyttöön jatkuvan integraation järjestelmä yrityksen kehittämälle ohjelmistoprojektille, jossa kehitetään testausohjelmistoa tukiasematesterille. Järjestelmän tuli olla yhteensopiva testeriin kehitetyn ohjelmiston kanssa, jotta kehitystyön jatkuessa voidaan tarvittaessa helposti kehittää lisää testejä laadun parantamiseksi. Järjestelmän tuli olla yhteensopiva myös aikaisemmin käyttöönotetun versionhallinnan Gitin ja tallennuspaikkana toimivan GitHubin kanssa. Järjestelmän alustalle eli automaatioserverille ei ollut alkuvaiheessa vaatimuksia, joten teknologioiden näkökulmasta käyttöönotossa oli melko vapaat kädet. Taulukossa 1 eritellään järjestelmälle työn alkuvaiheessa asetetut vaatimukset.

TAULUKKO 1. Vaatimukset CI-järjestelmälle

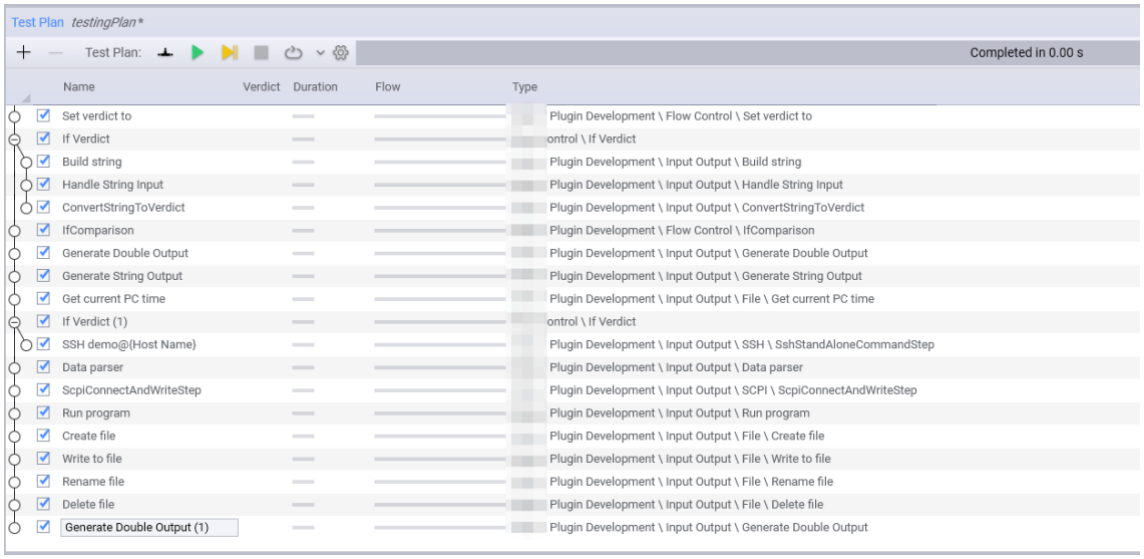
Versionhallinta	Versionhallintana käytetään Gitiä. Vaatimuksena yhteensopivuus Gitin avulla tehdyille kontribuutioille.
CI-palvelin	Vaatimuksena yhteensopivuus koodirepositorioiden tallennuspaikkana toimivan GitHubin kanssa.
Testiautomaatio	Järjestelmän mahdollistettava automatisoitujen yksikkötestien suunnittelu tukiasematesterin ohjelmistoon.
Koodirepositorioiden tallennus	Ohjelmistokoodien tallennuspaikka on GitHub. Vaatimuksena järjestelmälle havaita GitHubin repositorioihin tehdyt muutokset ja käynnistää testit.

Opinnäytetyön opintopistemäärän puitteissa kovin kattavaa testausautomaatiota ei ollut mahdollista kehittää, joten tarkoituksena olikin selvittää tarvittavat teknologiat, ottaa järjestelmä käyttöön ja aloittaa testien kehitys. Yrityksen muuta ohjelmistotiimiä ajatellen olikin hyödyllistä selvittää keinot, miten järjestelmä saadaan käyttöön, jotta myös tulevissa ohjelmistoprojekteissa voidaan hyödyntää hankittua osaamista.

Tukiasematesterissä käytettävä ohjelmisto on melko laaja ja ohjaa erilaisia testeriin kuuluvia instrumentteja. Tämä aiheuttaa haasteita testiautomaation kehittämiseen, sillä instrumentit joudutaan virtualisoimaan testien suorittamiseksi ohjelmistolle. Tukiasematesteriin kuuluvaa ohjelmistoa ei voida suorittaa millä tahansa alustalla. Ohjelmistoon kuuluu suuri määrä erilaisia kirjastoja, joten CI-palvelimien tiedostorajoitukset voivat tuoda haasteita ohjelmistokoodin kääntämiseen ja testien suorittamiseen palvelimella. Yhtenä vaihtoehtona oli myös oman testiserverin luominen ja käyttäminen testialustana.

Tukiasematesterin ohjelmisto koostuu suurimmaksi osaksi OpenTAPille suunnitelluista laajennuksista (engl. plugin). OpenTAP on avoimen lähdekoodin projekti automatisoitujen testien ja testisekvenssien luomiseen. OpenTAPin laajennusosat ovat testisekvenssiin suunniteltuja testisteppejä. Laajennukset voivat olla esimerkiksi testilaitteistoon kuuluvien instrumenttien

ohjaamiseen tai datan käsittelyyn tarkoitettuja. Laajennuksia voi ohjelmoida C#-kielellä Microsoftin Visual Studio -ohjelmointiympäristössä. Vaatimuksena CI-järjestelmälle on, että laajennusosille suunniteltujen yksikkötestien suorittaminen on mahdollista testiympäristössä. Lisäksi toivottu ominaisuus olisi OpenTAPin testisekvenssien (TapPlan) testaaminen niiden toimivuuden varmistamiseksi. OpenTAPin testisekvenssejä voi muokata ja kehittää siihen tarkoitetun editorin avulla (kuva 8).



KUVA 8. Kuvakaappaus OpenTAPin TapPlanista editorissa

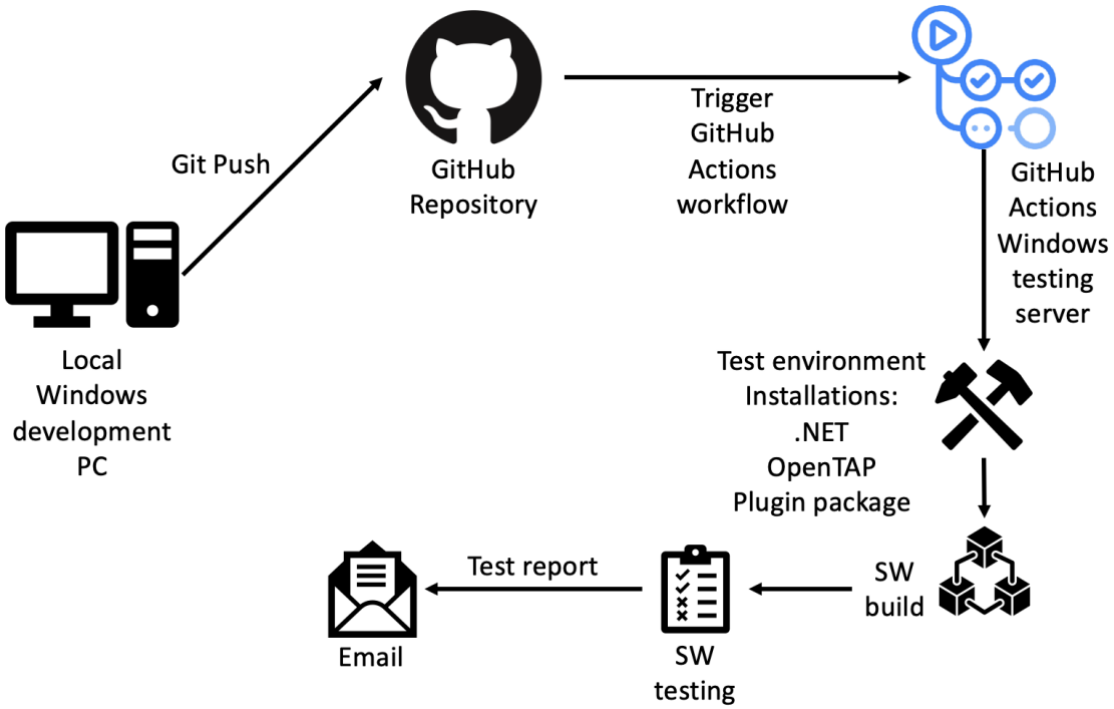
3.3 Järjestelmäarkkitehtuuri

GitHub Actions osoittautui työn aikana helppokäyttöiseksi ratkaisuksi. Yrityksessä oli jo aikaisemmin käyttöönotettu versionhallinta Git ja projekteille tallennuspaikaksi GitHub, joten GitHub Actionsin käyttäminen ja testisekvenssin luominen oli hyvin yksinkertainen prosessi. GitHub Actions on lyhyen olemassaolonsa aikana kasvattanut suosiotaan nopeasti ja yleistyy koko ajan eri organisaatioiden käytössä. GitHub tarjoaa testisekvenssien suorittamiseen erilaisia ympäristöjä Windowsista Linuxiin ja ohjelmistojen/kirjastojen asennukseen tarvittavia toimintoja löytyy jo melko kattavasti. Haasteena GHA:n käytölle on tiedostokokojen rajoitukset ja käyttöön liittyvät aikarajoitukset.

Ohjelmistoprojektiimme kuuluvan Visual Studio solution -ohjelmistopaketin suorittaminen ja koontiversion rakentaminen onnistui GitHub Actionsissä. Tämä vaati oikeiden konfiguraatioiden selvittämistä GitHub Actionsin testisekvenssiin ja tarvittavien asennuskomentojen lisäämisen työnkulku-tiedostoon. GitHub Actions havaitsee automaattisesti koodirepositorioihin tehdyt

muutokset, jotka käynnistävät sekvenssin suorituksen. Työnkulkutiedoston voi konfiguroida aktivoitumaan tiettyyn haaraan tulleista muutoksista tai repositorioon tulleesta vetopyynnöstä.

Visual Studio -projektissa tarvittavat NuGet-ohjelmistopaketit täytyi sisällyttää GitHub-repositorioon, jotta ne saatiin myös testiympäristön käyttöön GitHub Actionsissä. Vaihtoehto ohjelmistopakettien hallintaan on GitHubin tarjoama Packages-paketinhallintapalvelu. GitHub Packages tarjoaa mahdollisuuden erilaisten ohjelmistopakettien hostaamiseen julkisesti tai yksityiseen käyttöön. GitHub Packagesin käyttöä varten GitHubissa täytyy luoda käyttöoikeustietue (engl. access token) joko organisaatiolle tai käyttäjän henkilökohtaiselle tilille. Tokenille asetetaan käyttöoikeudet ohjelmistopakettien lukemiseen tai kirjoittamiseen GitHubin asetuksissa.



KUVA 9. Työssä suunnitellun CI-järjestelmän arkkitehtuuri

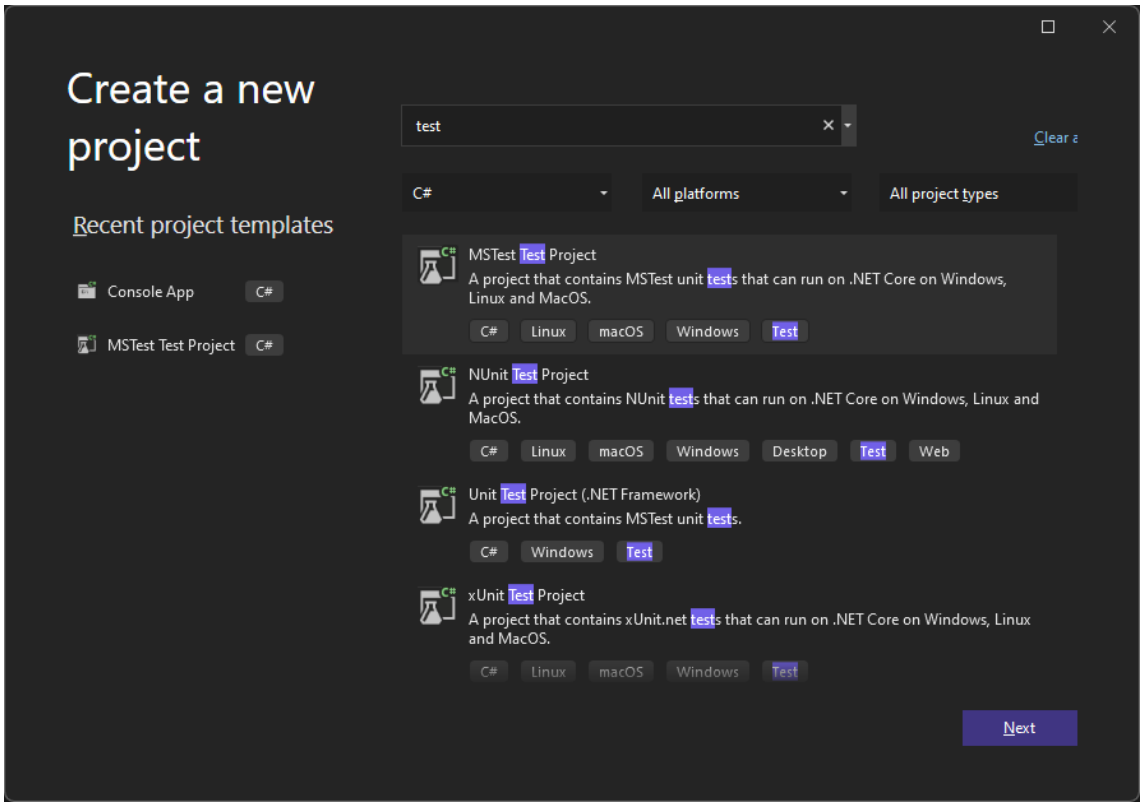
4 JÄRJESTELMÄN TOTEUTUS

Työvaiheet järjestelmän luomiseen:

1. Testiprojektin luominen
2. Testien suoritus Visual Studiossa
3. Testien suoritus komentokehotteessa
4. GitHub Actionsin työnkulkutiedoston luominen
5. Testien suoritus GHA:ssa
6. NuGet-ohjelmistopakettien hostaaminen GitHubiin
7. Yksikkötestien suunnittelu
8. OpenTAPin testisekvenssin suoritus GHA:ssa

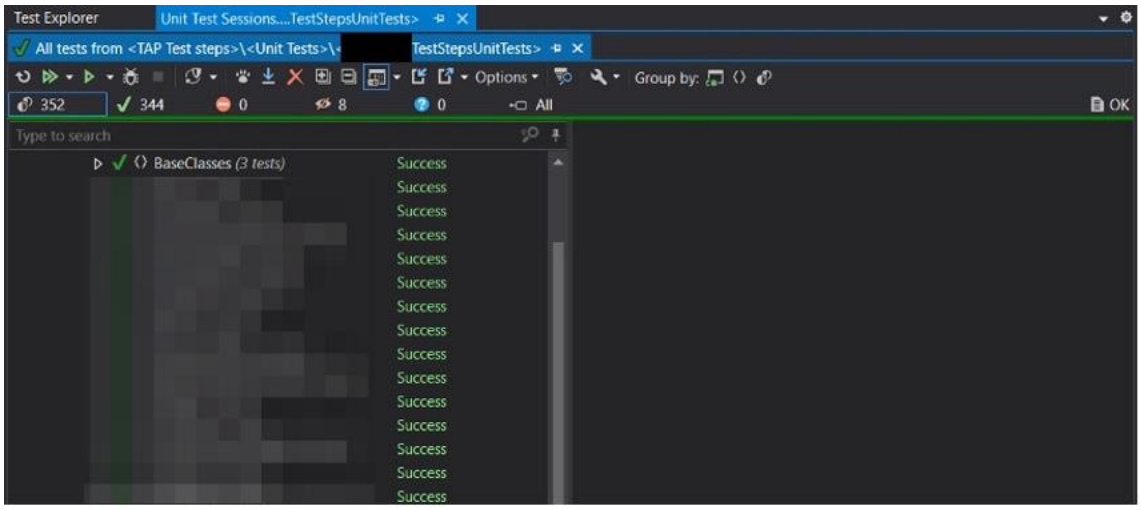
Tukiasematesterin ohjelmistossa käytetään Microsoftin Visual Studio -ympäristöä ja C#-ohjelmointikieltä. Aikaisemman kokemuksen puuttuessa testiautomaation kehittämisestä testiprojektien ja testien luominen oli aluksi haastavaa. Visual Studio -projekteissa käytetään usein Microsoftin MSTest-ohjelmistokehystä (engl. framework) testien kehittämiseen. Toisena vaihtoehtona tutkin myös avoimen lähdekoodin NUnitia, mutta Visual Studion mukana tuleva natiivi MSTest osoittautui sopivaksi ratkaisuksi. Visual Studion ja vanhojen .NET-ohjelmistokehyksien yhteensopivuus testiprojektien kanssa aiheutti jonkin verran yhteensopivuusongelmia.

Microsoftin Visual Studio solution -ohjelmistoille testiprojektien luominen on kohtuullisen yksinkertaista. Ohjelmistolle luodaan uusi projekti ja listalta haluttu projektin tyyppi (kuva 10). Testiprojektin luomiseksi hakusanan avulla voidaan hakea erilaisia projekteja halutulle ohjelmistokehykselle. Ensimmäisinä vaihtoehtoina ehdotetaan MSTestiä ja NUnitia. Testiprojektin valitsemisen jälkeen voidaan valita myös, mille .NET-kehykselle projekti luodaan. Projektin luomisen jälkeen on lisättävä myös tarvittavat riippuvuudet esimerkiksi ohjelmiston muihin projekteihin, jotta niiden käyttäminen onnistuu testiprojektissa. Kun projekti on luotu, Visual Studio luo automaattisesti UnitTest-tiedoston, johon yksikkötestejä voi alkaa kirjoittamaan.



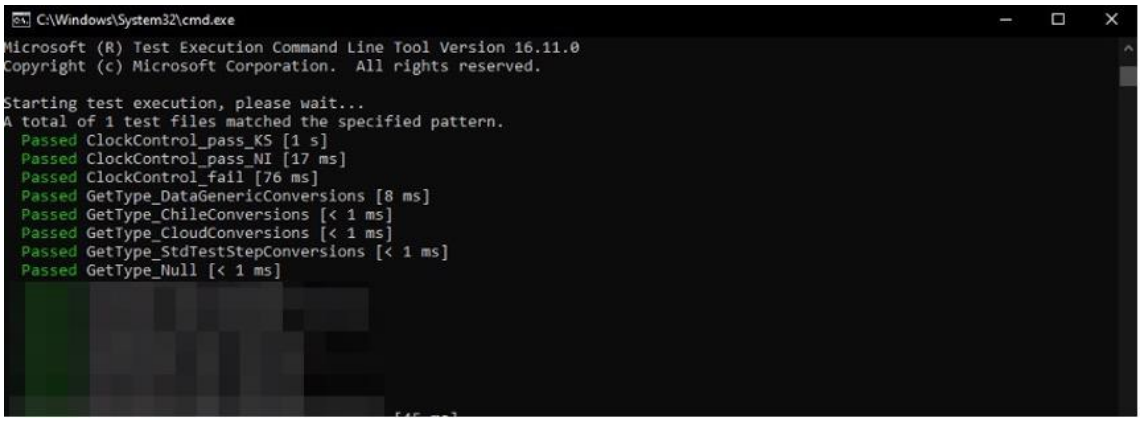
KUVA 10. Testiprojektin luominen Visual Studiossa

Visual Studiolla suunnitellut yksikkötestit voi suorittaa avaamalla testiselaimen (engl. Test Explorer). Testiselaimessa voi suorittaa kaikki ohjelmistoon kuuluvat testit tai valita halutut testit suoritettavaksi. Visual Studiossa on mahdollista suorittaa sisäänrakennetun MSTestin lisäksi myös muilla ohjelmistokehyksillä rakennettuja testejä. Suoritettujen testien läpimeno/epäonnistuminen näkyy reaaliaikaisesti testiselaimessa (kuva 11).



KUVA 11. Kuvakaappaus ajetuista testeistä ja tuloksista VS:n Text Explorerissa

Ohjelmistoprojektille suunniteltuja testejä voidaan suorittaa myös komentokehotteen avulla dotnet test -komennolla (kuva 12). Komento rakentaa ohjelmiston ensin testikelpoiseen tilaan, suorittaa testit komentokehotteessa ja tulostaa lyhyen raportin testituloksista. Testien ajaminen epäonnistuu, jos ohjelmistopaketista havaitaan minkäänlaisia virheitä. Komentoa voidaan muokata erilaisilla parametreillä, esimerkiksi halutaanko projekti rakentaa Debug- vai Release-konfiguraatiolla tai suorittaa vain tietyt testit. Vanhemmissa Visual Studio-pohjaisissa ohjelmistoissa voi joutua käyttämään dotnet vstest -komentoa, joka käyttää komentopohjaista VSTest.Console -ohjelmistoa yksikkötestien ajamiseen.

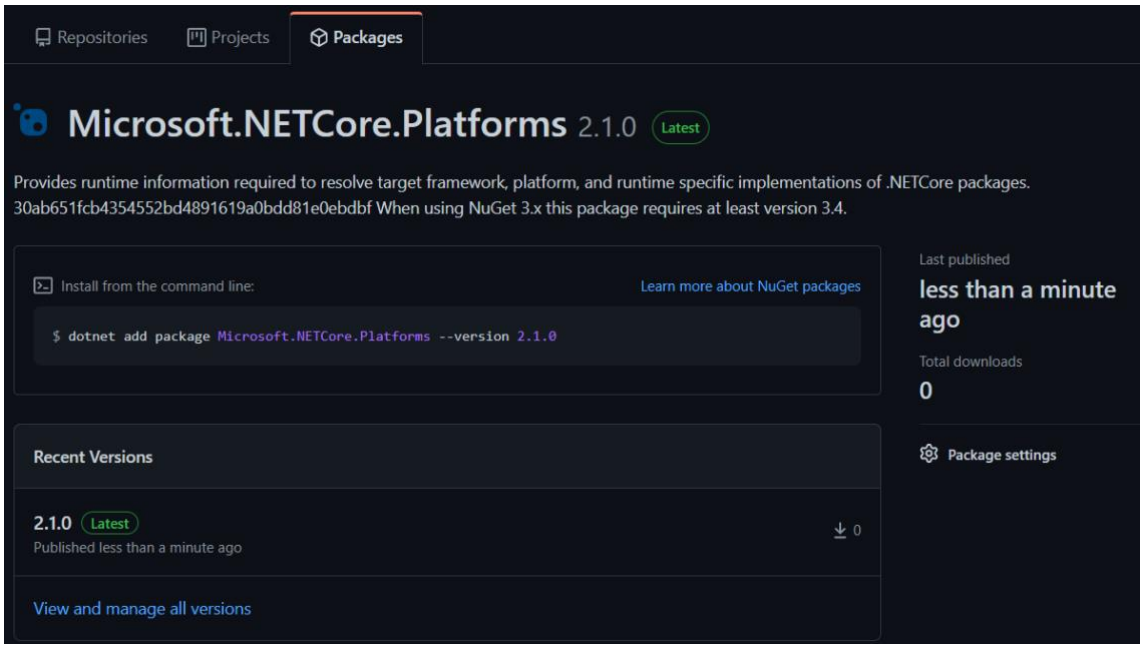


KUVA 12. Kuvakaappaus suoritetuista yksikkötesteistä komentokehotteessa

GitHub Action työnkulku eli suoritettava sekvenssi suunnitellaan soveltuvaksi testattavana olevaa ohjelmistoa varten. Sekvenssiin määritellään testien suoritusympäristö ja tarvittavat konfiguraatiot. Vaihtoehtoina käyttöjärjestelmälle ovat Windows, Mac ja Ubuntu. Tukiasematesterin ohjelmisto on luotu Microsoftin Visual Studiolla, joten Windows oli luonteva valinta testiympäristöksi. Muita vaadittavia konfiguraatioita ovat .NET-ohjelmistopaketin ja OpenTAPin asennuskomennot testiympäristöön. Seuraava vaihe on ohjelmiston koontiversion rakentaminen, jotta yksikkötestien suorittaminen olisi mahdollista. Tässä vaiheessa paljastuvat ohjelmistossa olevat mahdolliset virheet, jotka aiheuttavat sekvenssin suorituksen epäonnistumisen.

Testien ajaminen onnistuu helposti GitHub Actionsissä samalla komennolla. Yksikkötestien tulokset voi nähdä GHA:ssa tarkastelemalla testisekvenssin testiajon suoritusvaiheita. GHA:n työnkulku epäonnistuu, jos ohjelmistolle tehdyt yksikkötestit eivät mene läpi. Yksikkötestien ajaminen GHA:ssa vaatii ohjelmistoprojektin koontiversion rakentamisen testiympäristöön. Jos ohjelmistokoodi sisältää virheitä, koontiversion rakentaminen epäonnistuu eikä yksikkötestienkään suorittaminen ole mahdollista.

Ohjelmistoprojektiin kuuluvia NuGet-paketteja voi julkaista (engl. publish) yrityksen GitHub-tilille, joita voi halutessaan yhdistää repositorioihin ja hyödyntää niitä riippuvuuksina Visual Studio -ohjelmistoprojekteissa (kuva 13). Jotta voidaan muuttaa Visual Studio ohjelmiston käyttämien Nuget-pakettien sijaintia, täytyy muokata projektikansiossa olevaa nuget.config -tiedostoa ja asetettava pakettien sijainniksi GitHub Packagesiin tallennettujen pakettien internet-osoite. GitHub Actionsin työnkulkuun voi lisätä komennon, joka lisää Packagesin myös GHA:n käyttöön. Siten myös testiympäristö saa pääsyn Packagesiin tallennettuihin ohjelmistopaketteihin.



KUVA 13. Kuvakaappaus GitHub Packagesiin tallennetusta NuGet-paketista

Oleellinen osa automatisoitujen yksikkötestien luomista on jäljitelmätekniikka (engl. mock). Jäljitelmätekniikassa yksikkötestien suorituksissa käytetään ohjelmistossa olevista luokista simuloituja olioita (kuva 14). Yksikkötestin suoritukseen tarvittaviin riippuvuuksiin voi käyttää simuloituja olioita, jolloin yksikkötesti testaa ainoastaan testattavaa ohjelmistokomponenttia. CI-järjestelmän kohteena olevaan tukiasematesterin ohjelmistoon kuuluu runsaasti luokkia, esimerkiksi järjestelmään kuuluville instrumenteille, laitteille sekä erilaisille RF-mittauksille (engl. radio frequency). Yksikkötesteissä tukiasematesteriin kuuluvista instrumenteista voidaan luoda luokkien ja jäljitelmätekniikan avulla simuloituja olioita. Tämä mahdollistaa ohjelmiston toimivuuden testaamisen ainakin osittain ilman tukiasematesteriin kuuluvia instrumentteja. Haastavimmissa tapauksissa automatisoitu testaaminen ei välttämättä ole mahdollista ja ominaisuutta joudutaan testaamaan manuaalisesti. Käytin opinnäytetyössä jäljitelmätekniikan testaamiseen avoimen

lähdekoodin NSubstitute-ohjelmistoa. NSubstituten voi asentaa Visual Studio -projektille käyttämällä NuGet-pakettienhallintaohjelmaa.

```
private IPsu _psu;

[TestInitialize]
73 references | Kivela, 4 days ago | 1 author, 1 change
public override void TestInitialize()
{
    _psu = Substitute.For<IPsu>();
    base.TestInitialize();
}

[TestCleanup]
0 references | Kivela, 4 days ago | 1 author, 1 change
public void TestCleanup()
{
}

[TestMethod]
| 0 references | Kivela, 4 days ago | 1 author, 1 change
public void CurrentTest_Pass()
{
    //setting result to be within limits
    _psu.IsConnected.Returns(true);
    _psu.MeasureCurrent().Returns(5);
```

KUVA 14. Kuvakaappaus NSubstituten käytöstä yksikkötestissä

OpenTAPin testisekvenssin (TapPlan) suorittaminen onnistui komentokehotteessa avulla sopivan komennon löydyttyä (kuva 15). Tämän ansiosta testisekvenssin voi suorittaa CI:n testiympäristössä saman komennon avulla. Testeissä kävi ilmi, että GitHub Actionsin testiajo havaitsee sekvenssissä olevat virheet eivätkä testit mene läpi. Tämä tapa ei kuitenkaan testaa ohjelmistoa yksityiskohtaisesti, vaan testaa ainoastaan sekvenssin toimivuuden ja paljastaa mahdolliset virheet OpenTAPin testistepeissä.

```
10  OpenTAP Command Line Interface 9.19.5+6210d651
11
12  Loaded test plan from D:\a\Github-Actions-TEST\Github-Actions-TEST\GithubActionTEST.TapPlan [274 ms]
13  Test Plan: GithubActionTEST
14  ----------------------------------------------------------------
15  Starting TestPlan 'GithubActionTEST' on 02/08/2023 20:17:10, 11 of 11 TestSteps enabled.
16  "Generate Double Output" started.
17  "Generate Double Output" completed with verdict 'Pass'. [10.2 ms]
18  "Generate String Output" started.
19  Output Value: Some initial text
20  "Generate String Output" completed with verdict 'Pass'. [551 us]
21  "Handle String Input" started.
22  Input String Value: Some initial text
23  "Handle String Input" completed with verdict 'Pass'. [1.91 ms]
24  "Delay" started.
25  "Delay" completed. [105 ms]
26  "Log Output" started.
27
28  "Log Output" completed. [705 us]
29  "Repeat" started.
30  "Repeat \ Get current PC time" started.
31  "Repeat \ Get current PC time" completed with verdict 'Pass'. [784 us]
32  "Repeat" completed with verdict 'Pass'. [8.60 ms]
33  "IfComparison" started.
34  "IfComparison" completed. [5.45 ms]
35  "Check FPGA measurements" started.
36  "Check FPGA measurements" completed. [9.08 ms]
37  "ConvertStringToVerdict" started.
38  "ConvertStringToVerdict" completed with verdict 'Pass'. [1.51 ms]
39  -------- Summary of test plan started 02/08/2023 20:17:10 --------
40   Generate Double Output                          10.2 ms  Pass
41   Generate String Output                           551 us  Pass
42   Handle String Input                             1.91 ms  Pass
43   Delay                                            105 ms
44   Log Output                                       705 us
45   Repeat                                          8.60 ms  Pass
46     Get current PC time                            784 us  Pass
47   IfComparison                                    5.45 ms
48   Check FPGA measurements                         9.08 ms
49   ConvertStringToVerdict                          1.51 ms  Pass
50  ----------------------------------------------------------------
51  ----------- Test plan completed successfully in 273 ms -----------
```

KUVA 15. Kuvakaappaus OpenTAPin testisekvenssin ajamisesta GitHub Actionsissä

Tukiasematesterin ohjelmisto saatiin rakennettua GitHub Actionsin windows-testiympäristössä ilman virheitä. Onnistuneiden testien ansiosta päädyin käyttämään CI-alustana GHA:ta vaikeakäyttöisemmän Jenkinsin sijaan, vaikka Jenkins onkin yleisessä käytössä yrityksissä ja olisi hyödyllinen taito osata hallita sitä. Jenkins voi olla vielä monipuolisuutensa ja lukuisten laajennusosiensa ansiosta parempi ratkaisu joihinkin tapauksiin. Rajallisten resurssien takia GHA:n käyttö voi olla pienemmille yrityksille nopeampi tapa tuoda jatkuva integraatio ohjelmistoprojekteihin. GHA on helppokäyttöisyytensä ansiosta käytännöllinen palvelu. GHA:n käyttäjämäärän kasvaessa siitä voi tulla jopa yhtä suosittu CI-alusta kuin Jenkins. GitHubin

maksullisissa versioissa GHA:n käyttöön liittyvät rajoitukset vähenevät ja tällöin GHA sopii vieläkin paremmin yritysten tarpeisiin.

Työlle asetetut vaatimukset täytettiin onnistuneesti. Jatkuvan integraation järjestelmä saatiin yhteensopivaksi versionhallinnan kanssa. Järjestelmä aktivoi testien suorituksen Gitin avulla tehtyjen kontribuutioiden jälkeen. Repositorioiden tallennuspaikkana toimivan GitHubin kanssa GHA toimii sujuvasti. GHA havaitsee repositorion sivuhaaraan tehdyt muutokset tai vaihtoehtoisesti päähaaraan tehdyt vetopyynnöt. Automatisoitujen testien kehitys aloitettiin onnistuneesti käyttämällä Microsoftin MSTest-ohjelmistokehystä. OpenTAPin testisteppien testaus onnistui luomalla testisekvenssi OpenTAPillä ja suorittamalla sen GHA:n testiympäristössä. Tämä vaati OpenTAPin ja yrityksen kehittämän laajennuspaketin asentamisen testiympäristöön.

5 POHDINTA

Jatkuvan integraation järjestelmän käyttöönotto oli aiheena opiskelijalle haastava. Järjestelmän suunnittelu täytyi aloittaa lähes alusta, sillä työnantajayrityksessä ei ole ollut aikaisemmin käytössä vastaavanlaista järjestelmää. Työn alkuvaiheeseen kuului paljon selvitystyötä jatkuvaan integraatioon kuuluvista asioista, toimintatavoista ja vaatimuksista. Työn aihe tuli opiskelijalle melko uutena, vaikka se käsitteenä olikin tuttu. Opintoihini ei sisältynyt opetusta siitä, mitä jatkuva integraatio kokonaisuudessaan tarkoittaa ja mitä asioita siihen kuuluu. Testiautomaation tekemisestä oli opinnäytetyön alkuvaiheessa hieman aiempaa kokemusta, joten testien suunnittelussa tästä oli hyötyä. Versionhallinnasta oli myös aiempaa kokemusta opintojen/työn ansiosta, joten haasteena olikin järjestelmään kuuluvien asioiden yhteensovittaminen ja sopivien teknologioiden/työkalujen löytäminen.

Tukiasematesterin ohjelmistokoodin laajuus ja monimutkaisuus teki sopivan testiympäristön luomisesta haastavaa. Testeriin kuuluu erilaisia instrumentteja, joten niiden virtualisoiminen ja testien luominen on aloittelevalle ohjelmistokehittäjälle hankala tehtävä. Ohjelmistoprojekteissa tulisi heti alusta lähtien panostaa testien kehittämiseen ja antaa sille resursseja. Laajalle ohjelmistoprojektille testiautomaation kehityksen aloittaminen kesken projektin on kelle tahansa vaikea tehtävä, etenkin juuri valmistuvalle opiskelijalle. Testiautomaation kehitys ei tosin ollutkaan työn päätavoite vaan järjestelmän käyttöönotto, joka hyödyttää yrityksen ohjelmistotiimiä jatkossa.

Opinnäytetyön kirjoitusprosessi eteni sujuvasti alusta alkaen. Aiheesta löytyy teoriatietoa runsaasti ja erilaisten tutkimuksien löytäminen käytettävistä teknologioista oli kohtuullisen helppoa. Työn osuudesta kirjoittaminen oli hieman vaikeampaa, kun opinnäytetyön julkisuuden takia joutui rajoittamaan työhön liittyvistä asioista ja yksityiskohdista kertomista.

Opinnäytetyön jälkeen yrityksen ohjelmistokehittäjien olisi kannattavaa tehdä automatisoituja testejä samalla, kun ohjelmistoja kehitetään laadun parantamiseksi. Se vaatii aikaa ja resursseja, mutta olisi varmasti lopulta kannattavaa mahdollisten virheiden määrän vähentyessä. Asiakkaalle päätyvän ohjelmistokoodin laadun parantuessa asiakastyytyväisyys lisääntyy ja kehittäjien virheiden korjaamiseen kuluva aika vähenee. Ohjelmistotestaamiseen erikoistuneiden henkilöiden palkkaaminen voi olla myös toimiva ratkaisu.

LÄHTEET

1. Wiklund, Kristian & Eldh, Sigrid & Sundmark, Daniel 2017. Wiley Online Library. Impediments for software test automation: A systematic literature review. Hakupäivä 21.1.2023. https://onlinelibrary.wiley.com/doi/full/10.1002/stvr.1639
2. Fowler, Martin 2006. MartinFowler.com. Continuous Integration. Hakupäivä 16.1.2023. https://martinfowler.com/articles/continuousIntegration.html
3. Vasilescu, Bogdan & Yu, Yue & Wang, Huaimin & Devanbu, Premkumar & Filkov, Vladimir 2015. Quality and productivity outcomes relating to continuous integration in GitHub. Hakupäivä 18.1.2023. https://yuyue.GitHub.io/res/paper/fse2015.pdf
4. Shahin, Mojtaba & Ali Babar, Muhammad & Zhu, Liming 2017. IEEE Xplore. Continuous Integration, Delivery and Deployment: A Systematic review on Approaches, Tools, Challenges and practices 22.3.2017. Hakupäivä 28.1.2023. https://ieeexplore.ieee.org/document/7884954
5. McAllister, Jonathan 2015. Mastering Jenkins. The Jenkins platform architecture and configuration techniques. Hakupäivä 11.2.2023. https://media.oiipdf.com/pdf/5bbf1ad3-2782-40e2-ad97-99c5110c1aaf.pdf
6. Heller, Martin 2020. InfoWorld. What is Jenkins? The CI server explained. Hakupäivä 11.2.2023. https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html
7. Pool, Tonis 2013. University of Tartu. Faculty of Mathematics and Computer Science. Continuous Integration & Feature Branches. Hakupäivä 11.2.2023. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=023be4e368e261696e75b79d778a61a1a3d2f1a7
8. Osterman, Erik 2020. Cloud Posse. Jenkins Pros & Cons (2020). Hakupäivä 15.2.2023. https://cloudposse.com/devops/cicd/jenkins-pros-cons-2020/
9. Kinsman, Timothy & Wessel, Mairieli & Gerosa, Marco A. & Treude, Christoph 2021. arXiv. How Do Software Developers Use GitHub Actions to Automate Their Workflows?. Hakupäivä 11.2.2023. https://arxiv.org/pdf/2103.12224.pdf
10. Decan, Alexandre & Mens, Tom & Mazrae, Pooya Rostami & Golzadeh, Mehdi 2022. University of Mons. On the Use of GitHub Actions in Software Development Repositories. Hakupäivä 11.2.2023. https://decan.lexpage.net/files/ICSME-2022.pdf

11. Zolkifli, Nazatul Nurlisa & Ngah, Amir & Deraman, Aziz 2018. ResearchGate. Version Control System: A Review. Hakupäivä 3.2.2023. https://www.researchgate.net/publication/327291907_Version_Control_System_A_Review
12. Vuorre, Matti & Curley, James P. 2017. SAGE Journals. Curating Research Assets: A tutorial on the Git Version Control System. Hakupäivä 7.2.2023. https://journals.sagepub.com/doi/pdf/10.1177/2515245918754826