Lappeenrannan teknillinen yliopisto
LUT School of Engineering Science
Tietotekniikan koulutusohjelma

Diplomityö

Joonas Maksimainen

OHJELMISTOTESTAUSTYÖKALUJEN OPETUSKÄYTÖN ARVIOINTI TARKASTELEMALLA OPETUSMETODEJA, OHJELMISTOTESTAUSKURSSILLE HYÖDYNNETTÄVIÄ TYÖKALUJA JA NIIDEN POHJALTA LAADITTAVIA KURSSITEHTÄVÄIDEOITA

Työn tarkastaja(t): dos. Uolevi Nikula
DI Timo Hynninen

Työn ohjaaja(t): dos. Uolevi Nikula
DI Timo Hynninen

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
LUT School of Engineering Science
Tietotekniikan koulutusohjelma

Joonas Maksimainen

Ohjelmistotestaustyökalujen opetuskäytön arviointi tarkastelemalla opetusmetodeja, ohjelmistotestauskurssille hyödynnettäviä työkaluja ja niiden pohjalta laadittuja kurssitehtäväideoita

Diplomityö
2019
122 sivua, 4 taulukkoa, 14 kuvaa
Työn tarkastajat:
dos. Uolevi Nikula
DI Timo Hynninen

Hakusanat: ohjelmistotestaus, opetusmetodit, testaustyökalut

Tässä työssä arvioidaan ohjelmistotestaustyökalujen hyödyntämistä opetuksessa tarkastelemalla ohjelmistotestauksen opetusmetodeja, LUT-yliopiston kurssille CT60A0220 C-ohjelmoinnin ja testauksen periaatteet hyödynnettäviä testaustyökaluja ja niiden pohjalta luotuja kurssitehtäväideoita. Tarkoitus on pohtia ohjelmistotestaustyökalujen käytön mahdollisuutta yliopiston kursseilla ja tapoja parantaa ohjelmistotestauksen opetusta. Tutkimuksessa työkaluvalinnat tehtiin työkaluvaatimusten sekä testauksesta ja sen opetuksesta kerätyn aineiston pohjalta. Kurssitehtäväideat laadittiin työkalujen sekä kurssin sisällön ja tutkimusaineiston perusteella. Kurssin päätyttyä työkalu- ja tehtäväideavalintojen onnistuvuutta arvioitiin asiantuntijaraadin kanssa, joka antoi omat kommenttinsa ja arvionsa tutkimukseen ja sen tuloksiin liittyen. Vaikka työkaluja ei ehditty kunnolla hyödyntää kohdekurssilla, johtuen pääosin tehtävien pienestä määrästä, tarjosi tutkimus ideoita hyödyntää testaustyökaluja muilla kursseilla sekä loi pohjaa ja materiaalia tuleville työkalu- ja tehtävävalinnoille.

ABSTRACT

Lappeenranta University of Technology
LUT School of Engineering Science
Software Engineering Degree Program

Joonas Maksimainen

Evaluation of utilizing software-testing tools in teaching by analysing teaching methods, testing tools to be utilized in software testing course and course work ideas

Master's Thesis
2019
122 pages, 4 tables, 14 figures
Examiners:
Associate professor Uolevi Nikula
M.Sc. (Tech.) Timo Hynninen

Keywords: software testing, teaching methods, testing tools

In this work, the utilization of software-testing tools is evaluated by analyzing teaching methods of software testing, software testing tools and coursework ideas of LUT University's course CT60A0220 Principles of C-programming and software testing. Based on these, the main goal of this study is to analyze possibilities of utilizing software testing tools in university's courses and ways to improve the teaching of software testing. In the research, the tool choices were based on tool requirements and the research material of software testing and teaching software testing. Coursework ideas were made by analyzing testing tools, course material and previous research material. After the course, professors and experts evaluated the choices for the tools and coursework ideas and gave their own comments and critiques of the research and its results. Although the testing tools weren't properly utilized in the target course mainly due to small amount of coursework based on them, the research offered lots of ideas to utilize testing tools in later courses, and created layout and material for future tool and coursework choices.

ALKUSANAT

Työ on tehty LUT-yliopiston tietotekniikan diplomi-insinöörin tutkintoa varten. Kiitän kaikkia minua työssä tukeneita, perhettä, sukulaisia ja ystäviä. Kiitos.

SISÄLLYSLUETTELO

SYMBOLI- JA LYHENNELUETTELO

ACM = Association of Computing Machinery

CentOS = Community Enterprise Operating System

COTS = Commercial-Of-The-Shelf

CSV = comma separated values

ESW = Embedded Software

GQM = Goal-Question-Metric

IEEE = Institute of Electrical and Electronics Engineers

LTI = Learning Tools Interoperability

PECA = **P**lanning the evaluation, **E**stablishing the criteria, **C**ollecting the data, **A**nalyzing the data

SWA = Semantic Web Application

TDD = Test-driven development

1 JOHDANTO

1.1 Tausta

Tietotekniikan kehitys luo jatkuvasti haasteita niin alan yrityksille kuin sen opetukselle. Suurin ongelma ei kuitenkaan ole siinä, etteivät opetuslaitokset pysy uusien teknologioiden tahdissa, vaan vaikeudet saada kurssien tarjonta kohtaamaan yritysten tarpeiden kanssa käytännössä sekä opiskelijoiden taidot muistaa ja hyödyntää kurssin antia jälkeenpäin. Vaikka opiskelija olisi saanut tietoja yrityksien kaipaamista teknologioista ja käytännöistä teoriassa, voi hänen konkreettiset käytännön taidot aiheesta olla vajaita. Esimerkiksi opeteltaessa ohjelmistotestausta, voi opiskelijoilla olla suuria vaikeuksia hakea työpaikkaa alan yritykseltä, mikäli kurssista on kulunut kohtalaisesti aikaa ja opitut taidot olivat hyvin teoriapainotteisia. Vaikka ohjelmien testaus käytännössä voidaan oppia ohjelmia koodatessa, ne taidot eivät välttämättä vastaa niitä ohjelmistotestauksen taitoja ja työkalujen käyttöä, joita esimerkiksi alan yritykset tarvitsevat. Siksi on tärkeää, että tietotekniikan opetuksessa opiskelija saa teorian lisäksi paljon konkreettista käytännön oppia häntä kiinnostavan alan työkaluista, jotta hänellä on vähintään peruspohja parantaa taitojaan jälkeenpäin opiskellessa lisää ja lopulta yrityksessä.

1.2 Tavoitteet ja rajaukset

Diplomityön tavoitteena on arvioida mahdollisuuksia hyödyntää ohjelmistotestaustyökaluja samaa aihetta käsittelevien kurssien opetuksessa vertailemalla ohjelmistotestauksen opetusmetodeja, etsimällä sopivat testaustyökalut ja laatimalla testaukseen keskittyvä harjoitustehtävämateriaali ohjeistuksineen käytettäväksi Lappeenrannan teknillisen yliopiston kurssilla CT60A0220 C-ohjelmoinnin ja testauksen periaatteet. Lisäksi tavoitteena on arvioida työkalujen käyttöönoton ja harjoitustehtävien sisällön onnistumista kurssin päätyttyä asiantuntija-arvioinnilla työn ulkopuolisen asiantuntijan kautta. Kohteena olevan kurssin tavoitteena testauksen osalta on, että opiskelija tuntee tavallisimmat ohjelmistotestauksen työmenetelmät sekä testauksen työvaiheet ja työkalut, omaa valmiudet tehdä ohjattua testaustyötä itsenäisesti, tai suunnitella ja valmistella testaustyötä osana organisaatiota. Lisäksi opiskelija tietää miten ohjelmistotestausta tehdään ja kuinka testaustoiminta ja ohjelmistokehitys liittyvät toisiinsa.

Diplomityön harjoitusmateriaaliosio kattaa harjoitustehtävät, harjoituksissa käytettävät työkalut sekä työkalujen käytön ohjeet ja opastusvideot. Harjoitustehtävissä on tarkoituksena oppia eri testausmetodien ja – työkalujen käyttöä, jotka keskittyvät testauksen v-mallin mukaisesti yksikkö-, integraatio- ja järjestelmätestaukseen. Testausharjoituksissa testauksen kohteena toimii c-ohjelmointikielellä koodattu yksinkertainen ohjelma, jota kurssin osanottajat testaavat eri metodein ja työkaluin etsien virheitä. Näitä löydettyjä virheitä verrataan tehtäviä laatineiden kurssin vetäjien kylvämiin virheisiin, jolloin voidaan arvioida opiskelijoiden suorituksia harjoituksissa.

1.3 Työn rakenne

Rakenteeltaan tutkimus käsittää teorian ja käytännön osuuden, joista käytännön osuus kattaa valitut työkalut, niistä laaditut tehtävät sekä opastusvideot ja ohjeet työkalujen käyttöön. Tutkimuksen teoriaosuudessa toisessa luvussa tarkastellaan lyhyesti sovellustestausta ja sovellustestauksen opetusta, jonka jälkeen esitetään kolmannessa tutkimus- ja valintaprosessin kehitys valittaessa työkaluja ja harjoitustehtäviä. Tämän jälkeen neljännessä luvussa esitellään tutkimuksen kohdekurssi ja lopputulokset työkalu- ja tehtävävalintojen osalta, joka käsittää esittelyt valituista työkaluista ja laadituista tehtävistä. Lisäksi arvioidaan tutkimuksen onnistumista asiantuntija-arvion kautta ja tarkastellaan tulevaisuuden näkymiä. Tällä on tarkoitus selvittää kurssimateriaalin positiivisia ja negatiivisia seikkoja sekä arvioida, miten tulevien kurssien materiaalia voisi edelleen parantaa. Tutkimuksen viidennessä luvussa suoritetaan yhteenveto.

2 OHJELMISTOTESTAUS JA SEN OPETUS

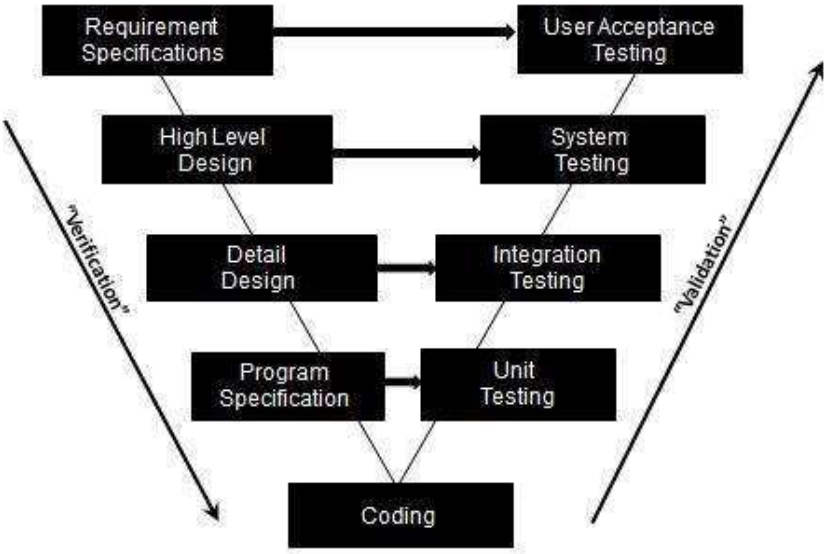
Vuosikymmenten saatossa ohjelmistotestaus on muuttunut yhä vaikeammaksi mutta samalla myös helpommaksi kuin koskaan. Muun muassa ohjelmointikielet, käyttöjärjestelmät ja alustat ovat kehittyneet ja lisänneet ohjelmistojen käyttö- ja vastuualueita, jolloin ohjelmiston toiminnallisuuden varmistaminen on muodostunut yhä tärkeämmäksi. Erityisesti ohjelmistojen käyttäjäkunnan ollessa maailmanlaajuista, aiheuttaa tämä suurta turhautumista ja resurssien tarvetta kehitykseen ja testaukseen tarvittavassa työssä. Kuitenkin samalla ohjelmistojen ja käyttöjärjestelmien ollessa kehittyneempiä ja niiden tarjotessa jo valmiiksi kehitettyjä ja testattuja rutiinisovelluksia, ei ohjelmiston kehitystä tarvitse aloittaa alusta asti. Esimerkiksi graafista käyttöliittymää kehitettäessä on nykyään tarjolla paljon erilaisia kirjastoja, joista voi käyttää valmiiksi debugattuja ja testattuja objekteja, jolloin muun muassa testaukseen tarvittava aika vähenee. [1, s.1]

2.1 Johdanto ohjelmistotestaukseen

Lyhyesti ohjelmistotestaus terminä tarkoittaa Myersin, Sandlerin ja Badgettin teoksen mukaan prosessia tai sarjaa prosesseja, joiden tarkoituksena on varmistaa ohjelmointikoodin toimivuus sille osoitetulle ja suunnitellulle tehtävälle ja toisaalta estää, ettei se tee mitään odottamatonta. Ottaen huomioon monet ohjelmistotestauksen kuvailevat termit, voidaan ohjelmistotestaus katsoa tuhoavaksi prosessiksi löytää virheitä ohjelmasta. Vaikka ideaalissa tilanteessa haluttaisiin testata jokainen sovelluksen permutaatio, ei tällainen ole ollenkaan mahdollista, saati käytännöllistä. [1, s.5-8]

Kuitenkin yllättäen, puhuttaessa ohjelmistotestauksen tavoitteista, eivät monet ohjelmistokehittäjät ole ollenkaan varmoja omista testaamistavoitteistaan. Tämä taas voi johtua siitä, ettei testauksessa aina ymmärretä eroa verifikaation ja validoinnin välillä, toinen niistä jätetään huomioimatta ja testauksen perimmäinen tavoite on ymmärretty väärin. IEEE:n standardien mukaan testauksessa verifikaatio tarkoittaa prosessia, joka määrittelee, milloin tuotteet ohjelmiston kehitysprojektissa tietyssä vaiheessa täyttävät aiemmassa vaiheessa asetetut vaatimukset. Validointi taas tarkoittaa prosessia, jossa arvioidaan sovellusta ohjelmistokehityksen loppuvaiheessa varmistaen tarkoitetun käytön toimiminen vaatimusten mukaan. Kun verifikaation keskittyy enemmän tietoihin yksittäisistä sovellusartefakteista, vaatimuksista ja spesifikaatioista, riippuu validointi kohteesta, jolle sovellusta ollaan tekemässä. [2, s.8-9]

Ohjelmistotestaus on välttämätön toiminto informaatioteknologiassa ja kattaa laajoja toimenpiteitä niin pienen koodinpätkän testaavasta yksikkötestistä suuren informaatiojärjestelmän asiakashyväksyntään ja verkkokeskeisen palvelusovelluksen tarkkailuun. Monella tapaa testaaminen voi tähdätä erilaisiin tavoitteisiin esimerkiksi paljastamalla poikkeamia käyttäjävaatimuksissa tai mittaamalla sovelluksen rasituksen sietokykyä. Lisäksi itse testausprosessi voidaan suorittaa joko tarkasti kontrolloidun prosessin mukaan sisältäen suunnittelun ja dokumentoinnin tai tilapäisesti joustavammin. 70-luvun lopulta lähtien uraauurtavat tutkimukset ja julkaisut sovellustestauksen teoriasta ovat muovautuneet erilaisiksi tekniikoiksi ja työkaluiksi, jotka taas ovat muuttuneet kehitys- ja testausprosesseiksi. Monista yrityskäyttöön otetuista prosesseista tunnetuin on Kuva1.:n esittämä V-malli, jonka eri variaatiot ovat luoneet kuvauserot eri testaustasoille, joita ovat yksikkö-, integraatio-, systeemi- ja hyväksymistestaus. [3]

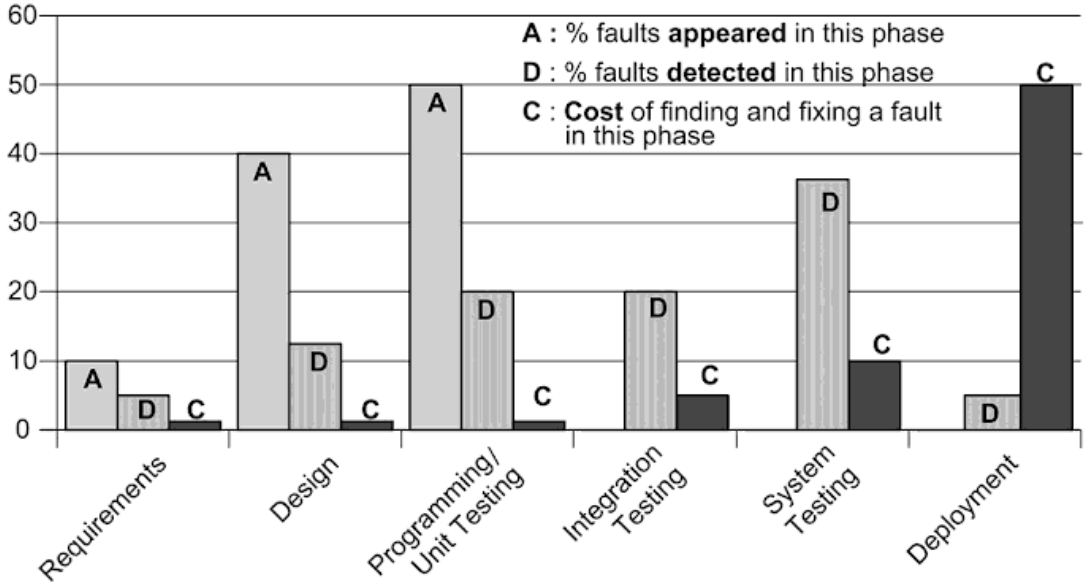


Kuva 1. V-malli Tutorialspoint. [62]

Huolimatta testauksen V-mallin asemasta, on A. Bertolinon mukaan 2000-luvulta lähtien väitetty V-mallia, perustuen sen formaaliin dokumentointiin, joiltakin tahoilta tehottomaksi ja turhan byrokraattiseksi, ja on näin lisännyt ketterämpien prosessien suosiota. Tämän takia, koskien juuri testauksen merkitystä, erityisesti testausvetoinen kehitysprosessi eli *test-driven development* (TDD) on saanut paljon huomiota. [3] Testausvetoisessa kehitysprosessissa yksikkötestit laaditaan ja koodataan samalla tai heti kehitysvaiheessa kohteina olevien sovellusyksiköiden jälkeen, jolloin yksikön testaus voidaan suorittaa heti sen kehityksen jälkeen. Näin pystytään myöhemmin testaamaan sovellus automaattisesti ja nope-

ammin, kun kaikki luodut yksikkötestit ajetaan automaattisesti läpi käyttäen hyväksi työkaluja, jotka ovat toisena tekijänä lisänneet ketterämpien prosessien suosiota. Lisäksi automaattisilla yksikkötesteillä voidaan arvioida heti päivitysten vaikutuksia, kun sovellukseen tehdään muutoksia. Vaikka TDD katsotaan yksinomaan kehitys- eikä testausprosessiksi, avaa se silti uusia näkökulmia kehitettäessä ja etsittäessä tehokkaampia testausmenetelmiä erityisesti testausautomaatioon liittyen. [4]

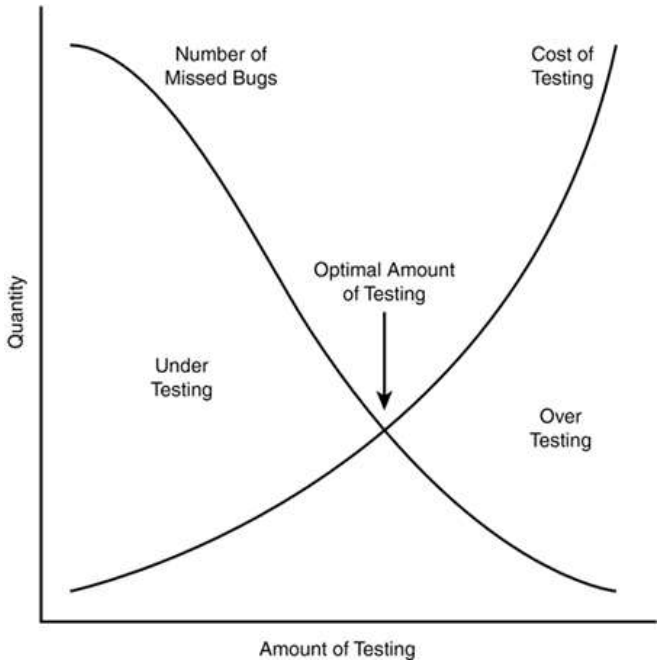
Huolimatta siitä millaista prosessia testauksessa käyttää, on testauksen merkitys suuri ohjelmistokehityksessä erityisesti kustannusten kannalta. Yleisesti sanottuna virheiden korjaamisesta aiheutuvat kustannukset kasvavat aina sitä suuremmaksi mitä myöhemmin ne havaitaan kehitysprosessin aikana. Kuvasta 2 voi havaita miten suureksi kustannukset voivat kasvaa, mikäli virhe havaitaan vasta käyttöönottovaiheessa verrattuna muihin kehitysvaiheisiin. Vaikka täydellistä testausta ei voida koskaan saavuttaa, voidaan säästää kustannuksissa huomattavasti, mitä varhaisemmin virheet havaitaan ja korjataan. [2, s.12]



Kuva 2. Myöhäisen testauksen kustannukset [2]

Sen lisäksi, että sovelluksen 100-prosenttinen testaaminen on mahdotonta, siihen ei kannata myöskään pyrkiä. Tietysti tavoitteena on löytää ja korjata virheitä sovelluksesta varhain mahdollisimman paljon, mutta täydellisyyteen pyrkiminen voi haitata projektia. Vaikka virhe, joka paljastuu vasta asiakkaan kohdalla, tulee kalliiksi korjata, vaatii testaaminen valtavasti resursseja, ellei sitä tehdä optimaalisesti. Kuva 3 näyttää, miten kustannustehottomaksi testaaminen muuttuu yritettäessä testata kaikkea, kun testauksista

aiheutuneet kulut ovat tehneet kohteena olleen projektin tai tuotteen kannattamattomaksi. Ohjelmistoprojekteissa tuote on julkaistava viimeistään jonakin päätettynä ajankohtana, jonka takia myös testaus on lopetettava, mutta ei kuitenkaan liian aikaisin. Tämän takia on tärkeää oppia järjestämään suuri testien määrä hallittavaan muotoon ja pystyä priorisoimaan tärkeimmät testit, jolloin tavoitteena testataan optimaalinen määrä eikä liian paljon tai liian vähän. [5, s.39-40]



Kuva 3. Jokaisella ohjelmistoprojektilla on optimaalinen testaustavoite [5]

2.1.1 Testausmetodit

Testauksen suorittamiseen on monia lähestymistapoja, jotka voidaan jakaa staattisiin, jossa koodia tarkastellaan sitä suorittamatta ja dynaamisiin, jossa koodi suoritetaan luotujen testitapausten mukaan. Testatessaan projektiratkaisua testaajat suorittavat joko toiminnallisia tai rakenteellisia testejä ja koska aiemmin todettiin, ettei täydellistä 100-prosenttista testausta voida tehdä, tietäen testauksen keskeisimmät tavoitteet, jää haasteeksi valita sopivat testausmetodit ja –strategiat, jotta testaus voidaan suorittaa mahdollisimman optimaalisesti. Kaksi tunnetuinta strategiaa tämän haasteen voittamiseksi ovat ”musta laatikko”-testaus eli *black box testing*, joka vastaa toiminnallista testausta sekä ”valkoinen laatikko”-testaus eli *white box testing*, joka taas vastaa rakenteellista testausta. Kumpikin näistä testausmetodeista käyttää erilaisia testaustekniikoita sisältäen sekä etuja että haittoja, mutta molempia testausmetodeja ja –tekniikoita tarvitaan, jotta sovellusten tai systeemien testaus voidaan hoitaa tehokkaasti. [6, s.69]

2.1.1.1 Black-box-testaus

Mustalaatikkotestauksella tarkoitetaan toiminnallista testausta, jossa, testatessa sovelluksen toimivuutta, ei tiedetä mitään sen sisäisestä logiikasta tai rakenteesta. Esimerkiksi tarkastellessa matemaattisen funktion toimivuutta sovelluksessa ei keskitytä siihen, miten funktio laskee lopputuloksen, vaan antaako funktio tietyn tuloksen tietyillä arvoilla. Rakenteen ja logiikan sijaan mustalaatikkotestauksessa keskitytään löytämään ne olosuhteet, joissa sovellus ei toimi sille asetettujen spesifikaatioiden vaatimalla tavalla. Myös testidata tässä testausmetodissa otetaan ylipäätään juuri spesifikaatioista. [1, s.9]

Toiminnallisessa testauksessa sovelluksen suoritus toimintojen tarkistamiseksi on välttämätöntä, jonka takia mustalaatikkotestaus luetaan validoinnin kategoriaan. Mustalaatikkotestauksella voidaan suunnitella ja toteuttaa hyvä määrä tehokkaita ja toimivia testitapauksia virheiden löytämiseksi sovelluksesta, koska testausmetodi simuloi varsinaisen sovelluksen käyttöä eikä tee oletuksia sovelluksen rakenteesta. [6, s.69]

Koska mustalaatikkotestauksella testataan oikeiden ja virheellisten syötteiden vaikutusta ohjelmaan, metodin sisältämiä tekniikoita voi käyttää kaikissa eri sovellustestauksen tasoissa, joita ovat myöhemmin esitettävät yksikkö-, integraatio-, systeemi- ja hyväksymistestaus. [7, s.38] Kuitenkin kun testataan useiden eri syötteiden vaikutuksia samalla sovelluksella, lisää tämä mahdollisuuksia jättää huomaamatta sovelluksen loogiset virheet ja nostaa esille tilanteen turhasta testailusta. [6, s.69] Erityisesti tilanteet, joissa halutaan testata epärealistisesti kaikki mahdolliset syötteet, sorrutaan väsyttävään syöttötestaukseen, mikä ei ole vain mahdotonta vaan myös aiemmin mainittuna epätaloudellista ottaen huomioon kaikkien sovellukseen laadittavien testitapausten valtava määrä. [1, s.9-10]

2.1.1.2 White-box-testaus

Valkolaatikkotestaus, jota kutsutaan välillä rakenteelliseksi testaukseksi, on vastakohta mustalaatikkotestaukselle, koska siinä testitapausten laatimiseksi ja testauksen suorittamiseksi täytyy tuntea sovelluksen sisäinen rakenne ja logiikka. Tästä syystä valkolaatikkotestauksen rakennetestit käyttävät verifikaatiotekniikoita. Esimerkiksi jos sovelluksen kehitystiimi luo koodikappaleen, joka prosessoi informaatiota tietyllä tavalla, täytyy testaustiimin verifioida tämä rakenteellisesti lukemalla koodia ja katsoa että koodi voisi toimia järkevästi. Mikäli on mahdollista, testaustiimi voi myös kytkeä koodin sovellukseen ja suorittaa sen, jolloin koodi pystytään rakenteellisesti validoimaan. [8]

Vaikka valkolaatikkotestauksen rakenteellisessa testauksessa pääsee testaamaan sovelluksen logiikkaa ja rakenteellisia ominaisuuksia esimerkiksi koodin tehokkuutta, voi liika logiikan tarkastelu aiheuttaa erkaantumisen kohdekäyttäjän vaatimuksista. Esimerkiksi kun keskitytään testaamaan ja muokkaamaan sovelluksen logiikkaa, voi virhearvio aiheuttaa sovelluksen toiminnan muutoksen käyttäjälle ei-halutulla tavalla. Lisäksi testattaessa logiikkaa, eivät muutokset tai testitapaukset välttämättä vastaa todellisen elämän tilanteita. [6, s.69]

Myös halu testata kaikki loogiset polkuvalinnat sovelluksessa voi osoittautua aiemmin mainitun väsyttävän syöttötestauksen tapaan sekä käytännössä että taloudellisesti mahdottomaksi. Mikäli loogisia polkuvalintoja halutun tavoitteen saavuttamiseksi sovelluksessa on runsaasti, on kaikkien näiden testitapausten luonti ja läpikäynti kannattamatonta. Toiseksi tässä väsyttävässä polkutestauksessa ei aina huomata kaikkia mahdollisia polkuja halutun tavoitteen saavuttamiseksi ja kolmanneksi kaikkia dataherkkyys virheitä ei välttämättä havaita testailusta huolimatta. [1, s.11-13]

Siinä missä toiminnalliset testaustekniikat mustalaatikkotestauksessa ovat validoivia ja sijoittuvat testauksen eri tasoille, ovat rakenteellisen testauksen testit verifioivia katsauksia. Verifikaatiota hyödyntävistä testeistä ensimmäisenä ovat soveltuvuuskatsaus, jossa varmistetaan sovellusyksikön logiikka tarkastelemalla yksikön rakenteellista toimintaa. Vaatimuskatsauksessa varmistetaan sovelluksen ominaisuudet, esimerkiksi tarkistetaan miten suuren rasituksen muun muassa suuresta samanaikaisesta käyttäjämäärästä jokin järjestelmä kestää. [8]

2.1.2 Testaustasot

Kuten aiemmin mainittiin, suoritetaan sovelluksen testaaminen yleisesti eri tasoilla, joita ovat yksikkö-, integraatio-, järjestelmä- ja hyväksymistestaus, joista 3 ensimmäistä suoritetaan testaajien ja jälkimmäinen asiakkaiden ja/tai käyttäjien taholta. Jokaisella testaustasolla on omat testaustavoitteensa ja tekniikat voivat vaihdella verifikaation ja validoinnin välillä, jonka lisäksi testaamiseen tarvitaan monia erilaisia työkaluja eri testaustasoilta. Jokaisen löydetyn virheen kohdalla lähdekoodia debugataan syiden löytämiseksi, mikä on merkittävin asia testaustoiminnoista mutta kuluttaa valtavasti sekä resursseja että aikaa sovelluksen julkaisulta. [7, s.368]

2.1.2.1 Yksikkötestaus

Yksikkötestauksessa testausta suoritetaan käyttämällä toiminnallisia ja rakenteellisia testejä. Testauksen kohteena ovat yksittäiset sovelluksen osat, joita voidaan kutsua muun muassa komponenteiksi, moduuleiksi, prosesseiksi tai funktioiksi, ja jokaisen niistä oletetaan toimivan niille määritellyn toiminnallisuuden mukaan. [7, s.369] A. Bertolino ja E. Marchetti ovat määritelleet yksikön artikkelissaan näin: ”Yksikkö on pienin testattavin sovelluksen osa, joka voi koostua sadoista tai vain muutamasta rivistä koodia, ja edustaa yleisesti yhden tai muutaman kehittäjän aikaansaamaa lopputulosta. Yksikkötestin tarkoituksena on varmistaa, että yksikkö täyttää sen toiminnalliset vaatimukset ja/tai että sen toteutettu rakenne täsmää yhteen sen suunnitellun rakenteen kanssa.” Lisäksi yksikkötestejä voidaan käyttää tarkistamaan käyttöliittymiä, eli parametrit välitetään oikeassa järjestyksessä, parametrien määrä vastaa argumentteja ja ne ovat samanlaisia. Muita tarkistuskohteita ovat lokaalin datan rakenne, eli sopimaton kirjoitus, väärä muuttujan nimi ja epäjohdonmukainen muuttujatyppi, sekä rajaolosuhteet. [9, s. 4]

Ongelmat yksikkötestien kanssa liittyvät paljolti siihen, voidaanko tiettyä yksikköä suorittaa itsenäisesti, sillä monesti yksiköt voivat tarvita muita yksiköitä toimiakseen kunnolla ja edelleen nämä voivat tarvita niitä seuraavia yksiköitä. Lisäksi yksikön suorittamiseen voi joutua kirjoittamaan lisää lähdekoodia, jolloin kohdeyksikölle tehdään sen toimintoja hallitseva ajuri ja muut kohdeyksikön toimintoa tarvittavien yksiköiden vastaavat tyngät. Tätä ajurin ja tynkien suunniteltua yhdistelmää kutsutaan rakennustelineiksi, jotka tulisi poistaa kohdeyksiköstä aina yksikkötestin suorituksen jälkeen virheiden paikallistamisen helpottamiseksi johtuen yksiköiden pienestä koosta. [7, s. 369]

Muut ongelmat yksikkötestauksessa liittyvät yksikön määritelmään, sillä esimerkiksi nykyisin yleisessä olio-ohjelmoinnissa on ongelmana valita testattavaksi yksiköksi joko luokka tai luokan metodit. Mikäli yksiköiksi valitaan luokan metodit, voi tämä tehdä testaamisesta yksinkertaisempaa ja perinteistä vaatimuksiin perustuvaa yksikkötestausta, mutta samalla lisää tarvetta käyttää edellä mainittuja rakennustelineitä testien suorittamiseksi ja vaikeuttaa integraatiotestien suoritusta. Jos taas yksiköiksi luetaan luokat, voi tämä helpottaa tulevaa integraatiotestausta, kun luokkia yhdistetään. Kuitenkin samalla luokkien perimän testaaminen vaikeutuu, kun esimerkiksi abstrakteja luokkia ei voida testata, koska niitä ei voida toteuttaa. Lisäksi luokilla, jotka ovat periytyneet suuremmista luokista, ei välttämättä ole kaikkia toimintoja, jotta testaus voitaisiin suorittaa kattavasti. Tämän takia joudutaan litistämään periytyviä luokkia yhdeksi isoksi luokaksi, jotka sisältävät perimän kaikki metodit ja mahdollisesti lisäämään muita testauksen suorittamiseksi. [10, s. 305]

Yksiköiden pienen koon takia monet valkolaatikkotestauksen tekniikat ovat sopivia käyttää yksikkötason testauksissa, sillä koodin tarkastelu ei vaadi niin paljon resursseja verrattuna integraatio- tai systeemitason kohteisiin. Lisäksi mikäli yksiköt pystyttäisiin suunnittelemaan siten, ettei niiden testaamiseen tarvitsisi luoda aiemmin mainittuja rakennustelineitä, tekisi se testaamisesta hyvin tehokasta. Kuitenkin käytännössä tällainen on hyvin vaikeaa, jonka takia ajurit ja tyngät tulisi pitää mahdollisimman pieninä ja niiden tarvetta minimoida kustannusten vähentämiseksi. Tämä taas riippuu yksikön toiminnallisuudesta ja sen jakautumisesta muiden yksiköiden kesken. [7, s.369]

2.1.2.2 Integraatiotestaus

Yleisesti integraatio on prosessi, jossa sovelluksen osat tai komponentit yhdistetään suuremman komponentin luomiseksi. Integraatiotestauksessa tähdätään erityisesti niihin ongelmiin, jotka aiheutuvat tässä vaiheessa. Vaikka yksittäiset sovelluksen osat suoriutuisivat läpi niille tehdyistä yksikkötesteistä eristyksessä muista, voivat ne aiheuttaa vääriä tai olettamattomia tuloksia komponenttien välillä. Tästä syystä integraatiotestauksella varmistetaan, että jokaisen komponentin vuorovaikutus toimii sille suunnittelussa asetettujen vaatimusten mukaan, mikä käytännössä merkitsee keskittymistä kommunikaatiorajapintoihin integroitujen komponenttien välillä. [9, s.5]

Koska integraatiotestauksessa keskitytään rajapintoihin yksikköjen välillä, kannattaa kiinnittää huomiota siihen, miten ja millä tavalla yksiköt ovat riippuvaisia toisistaan eli millainen liitos yksiköiden välillä on. Näitä liitostyyppejä parhaimmasta huonoimpaan ovat data-, leima-, hallinta,- ulko-, yleis- ja sisältöliitos. Mitä enemmän yksiköt jakavat dataa tai tekevät kutsuja toistensa välillä, sitä vahvempi liitos on, mikä voi lisätä virheiden potentiaalista määrää ja vaikeuttaa testausta erillisinä yksiköinä. Lisäksi liialliset riippuvuudet vaikeuttavat ja lisäävät työmäärää yksiköiden yhdistämiseksi ja laskevat suunnittelun koodin laatua. Tämän takia on parempi suosia mahdollisimman pieniä riippuvuuksia yksiköiden välillä, mutta jos liitos kuitenkin on suuri, tulee siihen kohdistuvaa rajapintaa testata useammalla eri testitapauksella. [7, s.370]

Kirjallisuudessa integraatiotestaukseen ei ole monia formalisoituja tapoja ja käytännön metodit perustuvat varsinaisesti hyvään suunnittelutaitoon ja testaajien intuitioon. Perinteisten systeemien testaus on tehty yleisesti joko lisääntyvällä tai ei-lisääntyvällä lähestymistavalla, joista jälkimmäisessä kaikki yksiköt linkitetään toisiinsa ja testataan kerralla. Lisääntyvässä lähestymistavassa käytetään joko ”top-down”-strategiaa, jossa moduulit yhdistetään yksi kerrallaan ja edetään testauksessa pääsovelluksesta hierarkiassa alaspäin pienempiin moduuleihin tai ”bottom-up”-strategiaa, jossa edetään testeissä ja moduulien liittämisessä päinvastaisessa järjestyksessä pienemmästä suurempaan. Käytännössä testauksessa käytetään sekalaista lähestymistapaa, joka määritellään ulkoisten tekijöiden kuten muun muassa moduulien saatavuuden, julkaisukäytännön ja testaajien käytettävyyden mukaan. [9, s.5]

Olio-ohjelmoinnissa integraatiotestaus on vähiten ymmärretty testauksen taso ja edellyttää täyttä yksikkötason testausta ja aiemmin mainitulla testattavan yksikkökohteen valinnalla, joko luokan metodit yksittäin tai koko luokka, on merkitystä. Mikäli metodit on valittu testattaviksi yksiköiksi, täytyy integraatiotestaus suorittaa kahdessa osassa, joista ensimmäisessä integroidaan ja testataan luokan sisäiset operaatiot yhtenä täytenä luokkana. Toisessa osuudessa integroidaan ja testataan luokka muiden luokkien kanssa. Jos taas testattavaksi yksiköksi valittiin koko luokka, joudutaan litistetyt luokat ”venyttämään” takaisin normaaleiksi ja testaamisen tarvitut lisämetodit on poistettava. Kun pohja integraatiotestaukselle on saatu valmiiksi, suunnitellaan ja tunnistetaan testattavat kohteet tarpeen mukaan ja testataan kohteet tilanteen mukaan joko staattisesti koodia tarkastelemalla tai dynaamisesti testaamalla aiemmin mainittujen tekniikoiden mukaan. [10, s.311]

2.1.2.3 Systeemitestaus

Kun yksikkö- ja integraatiotestit on suoritettu, ruvetaan suorittamaan systeemitestausta, jossa kokonaista sovellusta testataan sille oletetussa ympäristössä, käyttäen yleensä toiminnallisia testaustekniikoita, vaikka joitakin rakenteellisia testaustekniikoita voi käyttää. Itse systeemiksi määritellään sovelluksen, laitteiston ja muiden osaaottavien osien yhdistelmää, jotka tuottavat tuoteominaisuudet ja ratkaisut. Systeemitestauksella varmistetaan, että jokainen systeemin toiminto toimii odotetusti ja että jokainen systeemin ei-toiminnollinen vaatimus esimerkiksi suorituskyky ja turvallisuus testataan. Tästä syystä systeemitestaus on ainoa vaihe, jossa testataan sekä toiminnallisia että ei-toiminnallisia systeemin vaatimuksia. Testauksen suorittaa erillinen testaustiimi testijohtajan alaisuudessa, jonka lisäksi kaikki sovellukseen liittyvät manuaalit ja dokumentointi tarkastetaan, mikä verifikaatiotoimena parantaa lopullisen tuotteen laatua ja on näin yhtä tärkeää. [7, s.373]

Koska systeemitestauksessa perimmäinen tarkoitus on verrata systeemiä tai sovellusta sen varsinaisiin tavoitteisiin, on systeemitestauksesta 2 seuraavaa implikaatiota:

1. ”Systeemitestaus ei rajoitu systeemeihin. Jos tuote on sovellus, on systeemitestaus prosessina yritys osoittaa, kuinka sovellus ei kokonaisuutena täytä sille asetettuja vaatimuksiaan.”
2. ”Systeemitestaus, määritelmän mukaan, on mahdotonta, jos tuotteella ei ole minkäänlaisia sille määrättyjä, kirjoitettuja ja mitattavia tavoitteita.”

Huolimatta näistä määrittelyistä on systeemitestaus väärinymmärretyin ja vaikein testausprosessi, sillä monet käsittävät sen olevan vain valmiin systeemin tai sovelluksen funktioiden testausta, mikä sellaisenaan olisi turhaa, koska funktioiden testausta tehdään jo aikaisemmissa testaustasoissa. [1, s.130]

Vaikeaksi systeemitestauksen tekee, huolimatta keskeisestä tavoitteesta, siitä puuttuvat tunnistettavat metodiikat, jotka sisältäisivät kokoelman määriteltyjä testitapauksia. Sen sijaan joudutaan turvautumaan epämääräisiin mutta hyödyllisiin suosituksiin kirjoitettaessa testitapauksia, joilla voidaan edelleen näyttää sovelluksen epäjohdonmukaisuus tavoitteiden kanssa. Tämän takia systeemitestaus vaatii huomattavan määrän luovuutta, kenties jopa enemmän kuin itse sovelluksen ohjelmointi, suunnitellessa hyviä eri kategorioiden testitapauksia. G. J. Myersin teoksessa näitä testitapauskategorioita on 15, joita kannattaa

käydä läpi suunnitellessa testitapauksia systeemitestaukseen, vaikka teos ei väitäkään niiden sopivan jokaiseen ohjelmaan. [1, s.132]

Kategorioita ovat:

1. *Facility testing* eli laitteistotestaus: Onko jokainen vaatimuksen toiminto implementoitu?
2. *Volume testing* eli volyymitestaus: Pystyykö systeemi käsittelemään suurta datamäärää?
3. *Stress testing* eli stressitestaus: Pystyykö systeemi käsittelemään suurta datamäärää lyhyessä ajassa?
4. *Usability testing* eli käytettävyystestaus: Pystyykö systeemi vastaamaan ihmistekijöihin käytettävyydessä? Sisältää useita alempia kysymyksiä ja vaihtelee kohdekäyttäjäryhmän mukaan.
5. *Security testing* eli turvallisuustestaus: Tarkoituksena horjuttaa systeemin turvallisuutta eri testitapauksilla korjauksien tekemiseksi.
6. *Performance testing* eli suorituskykytestaus: Selviytyykö sovellus sille asetetuista suorituskyky- tai tehokkuusvaatimuksista?
7. *Storage testing* eli tallennustilatestaus: Kuinka paljon pää- ja väliaikaismuistia ohjelma käyttää ja mikä on väliaikais- tai työtiedostojen koko.
8. *Configuration testing* eli asetustestaus: Mitkä ovat sovelluksen tai systeemin minimi- tai maksimiasteukset ja millä asetuksilla ja alustalla kohdetuote toimii?
9. *Compatibility/Configuration/Conversion testing* eli yhteensopivuus-/konfiguraatio-/konversiotestaus: Miten hyvin sovellus suoriutuu muunnoksesta uudempaan versioon? Tuleeko datan siirrosta virheitä ja löytyykö yhteensopivuusongelmia?
10. *Installability testing* eli asennettavuustestaus: Onko sovelluksella monimutkainen asennus ja tuleeko asennuksessa virheitä?
11. *Reliability testing* eli luotettavuustestaus: Kaikki testit tukevat luotettavuutta mutta onko jotain tiettyjä suoraan vaatimuksiin liittyviä ja testattavia luotettavuuksia?
12. *Recovery testing* eli palautumistestaus: Pystyykö sovellus, esimerkkinä käyttöjärjestelmä, suoriutumaan ja palautumaan sille aiheutetuista virheistä, miten nopeasti ja millä tavoin?
13. *Serviceability testing* eli palvelutestaus: Onko ohjelmalla vaatimuksia palvelutarjonnan ja ylläpidon osalta ja liittyykö niihin virheitä?

14. *Documentation testing* eli dokumentaatiotestaus: Systeemitestauksessa ollaan huolestuneita myös käyttäjän dokumentoinnin tarkkuudesta. Vastaako dokumentaatio ohjeistus systeemin käytöstä systeemin todellista käyttöä?
15. *Procedure testing* eli proseduuri testaus: Toimivatko määrätyt ihmisproseduurit, esimerkiksi proseduurit systeemioperaattorille, tietokantaylläpitäjälle tai loppukäyttäjälle, toivotulla tavalla? Onko ylläpitotoimenpiteissä ongelmia esimerkiksi käyttöoikeuksissa tai itse toiminnoissa? [1, s.133-143]

Yleisesti tiivistettynä systeemitestaus sisältää testaamiset suorituskykyyn, turvallisuuteen, luotettavuuteen, stressitestaukseen ja palautumiseen. Lisäksi testejä ja systeemitesteistä saatua dataa voidaan käyttää määrittelemään toimivuutta, kun halutaan tukea tilastollista analyysia systeemin toimivuudesta. [9, s.5] Kuitenkin havaittaessa virheitä systeemitestauksessa, tulee asiassa noudattaa ehdotonta varovaisuutta ja suorittaa tarkka vaikutusanalyysi virheestä ennen korjausta. Joskus, jos systeemi sen sallii, virheen korjaamisen sijaan ne dokumentoidaan ja mainitaan tunnettuina rajoitteina, sillä tietyissä tapauksissa virheen korjaaminen voi muun muassa vaatia hyvin paljon aikaa tai teknisesti se ei ole mahdollista nykyisessä suunnittelussa. [7, s.373]

2.1.2.4 Hyväksymistestaus

Seuraava testitaso eli hyväksymistestaus, on yleensä asetettu aiempien testaustasojen yläpuolelle, vaikka hyväksymistestaus ei varsinaisesti ole oma tasonsa vaan enemmänkin jatkumoa systeemitestaukselle. [9, s.5] Kun testaustiimi arvioi, että tuote on valmis asiakkaille, kutsutaan asiakkaat demonstraatioon, jonka jälkeen he voivat testata tuotetta arvioiden käytön mielekkyyttä ja varmuutta. Arviointi taas voi vaihdella tilapäisestä käytöstä pitkäaikaiseen ja tarkasti suunniteltuun käyttöön, joka on ehdottoman tärkeää ennen lopullisen tuotteen hyväksymistä. Testauksen suorittajat voivat olla joko itse asiakkaita tai asiakkaan valtuuttamia henkilöitä ja testauspaikka taas joko kehittäjän tai asiakkaan riippuen yhteisistä sopimuksista. [7, s.373]

Hyväksymistestauksen ei pitäisi tapahtua vain kehitysprosessin lopussa vaan sen pitäisi olla jatkuvaa toimintaa, joka testaa tilapäisiä ja lopullisia tuotteita, jotta aikaa ei käytetä tarpeettomasti korjauksien tekemiseen, jotka osoittautuvat hylätyiksi loppukäyttäjän taholta. Päällimmäisenä tavoitteena hyväksymistestauksessa on varmistaa, että muutettu ap-

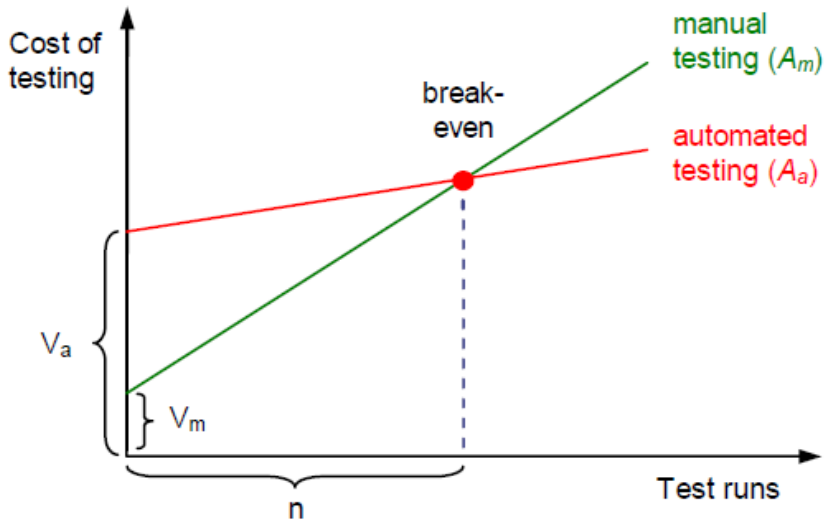
plikaatio toimii kunnolla sille tarkoitetussa ympäristössä. [6, s.493] Yleensä hyväksymistestaus suoritetaan asiakkaan luona ja vain silloin testattava tuote on kehitetty juuri tätä asiakasta varten. Jos kehitteillä on ”standardoitu” sovellus suurelle määrälle anonyymejä asiakkaita, esimerkiksi käyttöjärjestelmät, joudutaan mahdolliset käyttäjät identifioimaan laittamalla oletetut kohteet testaamaan sovellusta. Tätä kutsutaan alfa/beta-testaukseksi, joista beta-testauksessa mahdolliset asiakkaat testaavat sovellusta ilman kehittäjien osallistumista. Alfa-testauksessa mahdolliset asiakkaat suorittavat testauksen kehittäjien luona kehittäjien testaajien ohjauksen ja valvonnan alla. [7, s.374]

2.1.3 Testausautomaatio ja työkalut

Testaus on kenties kallein toimenpide sovellusprojekteissa, viemällä yhden arvion mukaan jopa yli 50 prosenttia ohjelmistoprojektin resursseista, jonka lisäksi huonosta testauksesta aiheutuvat jälkikustannukset, muun muassa huono sovelluksen laatu ja sen aiheuttamat virheet, tuottavat lisäkustannuksia kehittäjille. Sen takia sovelluksen laadun ja testausprosessin tehokkuuden parantaminen ovat tehokas tapa vähentää kustannuksia pitkällä tähtäimellä. Yksi ratkaisu testausprosessin tehokkuuden parantamiseen on automaation lisääminen testaukseen, missä testaajat voivat keskittyä kriittisempiin sovelluksen toimintoihin jättäen toistettavat työtehtävät testiautomaatiosysteemille. Tällä tavalla on mahdollista käyttää ihmisresursseja tehokkaammin esimerkiksi kattavampaa testaukseen tai muuten säästää kuluja yleisesti testausprosessissa ja sovelluskehityksessä. [11, s.1]

C. Meudecin mukaan testausautomaatio voidaan jakaa kolmeen pääkategoriaan, joista ensimmäisenä on hallintatehtävien automaatio, joihin kuuluvat muun muassa testausvaatimusten ja lopputulosten dokumentointi ja testausraporttien luominen. Toisena ovat mekaanisten ja toistettavien tehtävien automatisointi, jotka sisältävät sovelluksen suorittamisen ja tarkkailun sille testaukseen määrätyssä ympäristössä sekä laitteiston tallennus ja uudelleen toisto. Kolmantena kategoriana ovat testien luomistehtävien automatisointi [12, s.2], joka taas sisältää kolme alempaa tutkimusmenettelytapaa. Ensimmäisessä menettelytavassa osalla saadusta lähdekoodista generoidaan automaattisesti testisyötteet, joilla on tarkoitus selvittää sovelluksen eri tasot ja valintapolut. Toisessa menettelytavassa on tarkoitus, sovelluksen lähdekoodia ja oletettu toimintaa tuntien, generoida algoritmi, joka luo automaattisesti syötteet ja tarkistaa tulosteet. Kolmannessa menettelytavassa on tarkoitus saadulla formaalilla vaatimuksella automaattisesti generoida testitapaukset, jotta voidaan näin implementoida haluttu vaatimus. [13, s.1]

Kahdelle ensimmäiselle testausautomaation kategorialle löytyy paljon kaupallisia työkaluja, jonka takia automaattista testausta pidetään usein synonyyminä testien suorittamiselle automaattisesti, vaikka testisyötteiden varsinainen generointi tapahtuu yleensä vielä manuaalisesti. Vaikka poikkeuksena tähän onkin satunnaisten arvojen valinta, on testisyötteiden, puhumattakaan testitapausten, valinta ja luonti automaattisesti työnkalujen kehittäjille yhä haaste. [12, s.2] Koska automaatiotestausta käytetään toistettavien tehtävien suorittamiseen, esimeriksi yksikkötestaukseen tai regressiotestaukseen, joissa testitapaukset suoritetaan aina muutosten jälkeen, ovat tyypilliset automaatiotestauksen tehtävät testausskriptien kehittämistä ja kirjoittelua sekä testaustulosten verifiointia. Ne testaukset, jotka sisältävät vähän toistamista, esimerkiksi tutkiva testaus tai myöhäisen kehityksen verifikaatiotestaus, ovat sopivampia manuaaliselle testaukselle ja suurempana toimenpiitenä olevan automaation rakentaminen parempi useamman toiston testitapauksille. [11, s.2]



Kuva 4. Kriittinen piste automaatiotestaukselle [14]

Kuitenkin jako manuaalisen ja automaatiotestauksen välillä ei ole niin suoraa käytännössä kuin se näyttää, sillä mikä tahansa huonosti tehty koodi voi olla mahdotonta testata luotettavasti ja tehden näin automaatiotestauksesta kelpaamatonta. [11, s.2] Lisäksi vaikka testausautomaation käyttöönotto voi laskea kustannuksia testauksessa pitkällä tähtäimellä, ei sillä voida korvata kaikkea manuaalista testausta, eikä sen käyttöönoton tuomat kustannukset ole niin yksinkertaisesti laskettavissa tai halvempia, jos arvioitaisiin kuluja esimerkiksi kuvan 4 mallilla. Erityisesti R. Ramlerin ja K. Wolfmaierin artikkelissa on arvioitu ja otettu kantaa liian yksinkertaisten mallien käytöstä tehtäessä päätöksiä automaatiotestauksen käyttöönotossa. Heidän mukaansa ongelmia kustannusmalleissa manuaalisen ja auto-

maattisen testauksen vertailusta ovat vain kulujen arviointi hyötyjen jäädessä pois, vertaamattomien näkökulmien vertaus, kaikkien testitapausten tulkinta yhtä tärkeinä, projektin kontekstin, eritoten budjetin, huomiotta jättäminen ja lisäkustannustekijöiden puuttuminen analyysista. [14]

2.1.4 Esimerkkitestausprosessi lyhyesti

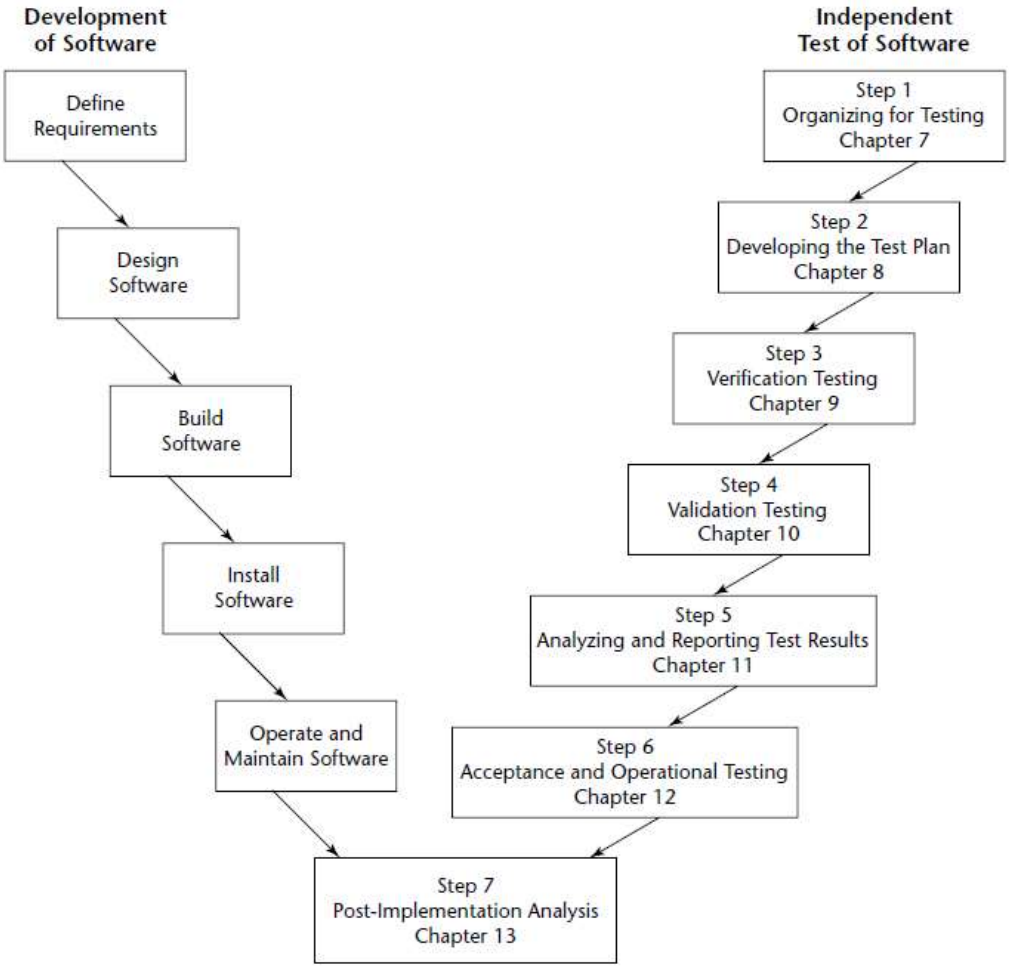
Sovelluksen testaukseen ja sovellusprojektin läpivientiin ei ole parhainta prosessia, sillä yritykset suorittavat testausprosesseja ja sovelluksen kehitystä eri tavoin ja menetelmin. Kuitenkin W. E. Perryn teoksessa esiteltävä seitsemän askeleen prosessi yhdistää monien eri prosessien hyviä aspekteja ja perustuu yli 1000:een Laadunvarmistusinstituutin eli *Quality Assurance Institute* kanssa tekemisissä olevien organisaatioiden ohjelmistotestausprosesseihin. Vaikka testausprosessit voivat vaihdella, tekee niiden ymmärtäminen ja käyttö testauksesta johdonmukaista, prosessi on helpompi opettaa muille, niitä voi parantaa ajan kuluessa ja testimanagerin on helpompi johtaa prosessia. [6, s.153-154]

Testauksessa on kaksi yleistä kategoriaa: esi-implementaatio- ja jälki-implementaatio testaus, joista ensimmäisessä tavoitteena on testata systeemin toimiminen asetettujen vaatimusten mukaan ja että systeemin virheet on poistettu ennen systeemin asettamista tuotantoon. Toisessa vaiheessa testaus tapahtuu, kun systeemi menee käyttöön ja katsotaan osaksi systeemin ylläpitoa. Kuten jo aikaisemmin mainittiin, on kustannusten kannalta parasta, että systeemin virheet havaitaan mahdollisimman varhain ensimmäisessä vaiheessa. [6, s.154]

2.1.4.1 Seitsemän askeleen ohjelmistotestausprosessi

Seitsemän askeleen ohjelmistotestausprosessi seuraa V-mallin konseptia testauksessa, jonka voi huomata kuvasta 5, jossa esitetään sekä sovelluksen kehitys- ja testausprosessi. Kummatkin prosessit alkavat ja jatkuvat samaan aikaan projektin loppuun asti ja projektin viimeinen vaihe, jälki-implementaatioanalyysi, tapahtuu sekä kehitys että testausprosessissa. Analyysin tarkoituksena on määritellä, voidaanko kehitys ja/tai testausprosessi suorittaa tehokkaammin tulevaisuudessa. Lisäksi kuvassa testausosuuden vaiheisiin ovat merkitty W. E. Perryn teoksen kappaleet, jotka kaikki sisältävät mallin kuvaillen testaustiimin suoritettavia proseduureja testauksen eri vaiheissa. [6, s.156] Ennen prosessin aloittamista on myös otettava huomioon, että prosessia edeltää testausympäristön luominen, jossa

testaus on tehokasta ja taloudellista. Tämä tarkoittaa yrityksien kohdalla johdon puolesta sopivien testaustyöprosessien valitsemista, testaustyökalujen ja niiden toiminta-alustan hankkimista sekä taitavan sovellustestaushenkilöstön palkkaamista ja kouluttamista. [6, s.60]



Kuva 5. Seitsemän askeleen ohjelmistotestausprosessi [6]

7:n askeleen testausprosessin ensimmäisessä vaiheessa on tarkoitus, pääasiassa testimanagerin taholta, organisoida testaus ja se jakautuu kolmeen pienempään osaan. Ensimmäisessä osassa määritellään testauksen laajuus eli mitä testataan ja minkälaista testausta aiotaan tehdä. Tämän jälkeen organisoidaan testaustiimi, eli päätetään ketkä suorittavat valitun testauksen, jonka jälkeen arvioidaan tulevaa testaussuunnitelmaa ja tämän hetkistä tilannetta esimerkiksi testaukseen tarvittavien resurssien osalta. [6, s. 165] Testausprosessin toisessa vaiheessa kehitetään testaussuunnitelma, jossa määritellään, kuinka testaus on tarkoitus suorittaa. Aluksi suoritetaan riskianalyysi, jonka jälkeen laaditaan testaussuunnitelma. Testaussuunnitelman rakenne tulisi olla aina sama, vaikkakin sen sisältö voi vaihdella perustuen niihin riskien vakavuuksiin ja vaatimuksiin, joita testaajat hahmottavat heille testattavassa sovelluksessa. [6, s. 210]

Kolmantena testausprosessin vaiheena on verifikaatiotestaus, joka niin ikään jakautuu kolmeen osaan. Näistä ensimmäisessä testataan sovelluksen vaatimuksia, jotka keskeneräisinä, epätarkkoina tai riittämättöminä johtavat useimpiin sovelluksen kaatumisiin ja tarkistamattomina lisäävät toteuttamisvaiheessa kustannuksia huomattavasti. Tätä varten testaajat verifikaatiolla varmistavat, että vaatimukset ovat tarkkoja ja ne eivät ole ristiriitaisia. Toisessa osassa tarkoitus on testata sovelluksen suunnittelua, eli sekä ulkoinen että sisäinen suunnittelu testataan läpi käyttäen verifikaatiotekniikoita. Näin saadaan varmistetuksi, että suunniteltu sovellus saavuttaa projektin tavoitteet sekä on tehokas ja tuottaa tulosta sille tarkoitetulla laitteistolla. Verifikaatiotestauksen kolmannen osan testit riippuvat siitä, millä tekniikalla sovellus on rakennettu ja koodattu sekä onko sovelluksessa käytetty valmiita ratkaisuja. Jos joku sovelluksen funktio on koodattu manuaalisesti itse, pitää sen toiminta verifioida. [6, s. 291-292]

Testausprosessin neljäs vaihe eli validointitestaus jakautuu kahteen osaan, joista ensimmäisessä suoritetaan niin ikään validointitestaus. Tämä tarkoittaa koodin testausta dynaamisesti, jossa testisuunnitelmassa merkittyjä lähestymistapoja, metodeja ja spesifioituja työkaluja käytetään varmistamaan, että suoritettavat koodit täyttävät merkityt sovellusvaatimukset ja suunnittelun rakenteelliset spesifikaatiot. Neljännen vaiheen toisena osana on testitulosten tallennus eli testeistä saadut tulokset dokumentoidaan. [6, s. 409-410] Testausprosessin viidentenä vaiheena on testitulosten analysointi ja raportointi, jossa ensimmäiseksi arvioidaan saatujen tulosten ongelmakohtia varianssilla ”mitä on” ja ”mitä pitäisi olla”. Tämän jälkeen laaditaan testiraportit, joissa ilmoitetut ongelmakohdat ja muut huolenaiheet on parasta saada mahdollisimman nopeasti niistä vastaavien osapuolten tietoon, jotta ne voidaan korjata mahdollisimman nopeasti. [6, s. 459-460]

Testausprosessin kuudentena vaiheena on hyväksymys- ja operaatiotestaus, joka jakautuu kolmeen osaan: hyväksymystestin suoritukseen, sovelluksen asennuksen testaukseen ja sovellusmuutosten testaukseen. Hyväksymistestit mahdollistavat loppukäyttäjille tilaisuuden arvioida sovelluksen soveltuvuutta ja käytettävyyttä suoritettaessa jokapäiväisiä työtoimintoja. Lisäksi hyväksymistestit testaavat sitä, että sovellus toimii kohdekäyttäjien toiveista laadittujen ohjelmistovaatimusten mukaisesti. Tämän jälkeen tulee testata sovelluksen asennus ja toimiminen sille tarkoitetussa ympäristössä, kun testaustiimi on varmistanut sovelluksen valmiuden tuotantoon. Tällä testataan käyttöliittymää operoiviin

sovelluksiin, liittyviin sovelluksiin ja operoiviin proseduureihin. Kuudennen vaiheen viimeisessä osassa testataan muutosten vaikutusta sovellukseen esimerkiksi tilanteessa, kun suoritetaan huoltoa ja halutaan varmistaa sovelluksen toimivuus muutosten ja päivitysten jälkeen. [6, s.491-492]

7:n vaiheen testausprosessin viimeisen vaiheen eli jälki-implementaatioanalyysin tavoite on tulevaisuuspainotteinen ja siinä on tarkoitus arvioida testauksen onnistumista ja mahdollisia parannuksia testaukseen. Parhaiten tuloksia testauksen kehittämisessä saa tekemällä arvion jokaisen sovelluskehityksen lopussa ja sisällyttämällä vaiheeseen myös kehittäjät, loppukäyttäjät ja laadunvarmistamisen ammattilaiset. Kun aina selvitetään testauksen tehokkuus ja siihen liittyvät syyt, voidaan kehittää testausta parempaan suuntaan yhä enemmän ja näin edelleen parantaa kehitettyjen sovellusten laatua. [6, s.581]

2.1.5 Laajempi perspektiivi testaukseen

Sovellustestaukseen liittyy monia kehitystavoitteita ja tavoitesaavutuksia, joiden tarkoituksena on parantaa testausta entisestään. A. Berentonin artikkelissa *Software Testing Research: Achievements, Challenges, Dreams* on listattu yleisimmät sovellustestauksen tutkimuksen tavoitehaaveet ja niitä kohtaavat haasteet. Ensimmäisenä näistä haaveista on universaali testausteoria, josta universaalilla tarkoitetaan vain yhtä koherenttia ja täsmällistä kehystä. Tätä kehystä testaajat voivat edelleen käyttää ymmärtääkseen olemassa olevien testaustekniikoiden vahvuuksia ja heikkouksia, ja kehys opastaa valitsemaan sopivimman tekniikan tai tekniikoiden yhdistelmän. Lisäksi unelmana olisi omata testauskoneisto, joka sitoo sen tavoitteeseen valita tehokkain testaustekniikka tai niiden yhdistelmä mukautuen sille tehtyihin oletuksiin. [3, s.7]

Ensimmäisenä haasteena universaalissa testausteoriassa on täsmällisen testausoletuksen eli testaushypoteesin löytyminen. Vaikka sovellus testattaisiin ja sen toiminta todistettaisiin oikeaksi sille tehdyllä oletuksella, voi sovelluksen toiminta vielä olla väärä. Tämän takia yhtä isoa ja samaa testausoletusta ei ole, vaan se jakautuu pienempiin luokiteltuihin oletuksiin riippuen testaustavoitteesta. [3, s.7] Toisena haasteena on testauksen tehokkuus, sillä vaikka jokin testaustekniikka olisikin todella tehokas, sisältävät kaikki testaustekniikat eri tyypin heikkouksia. Lisäksi testaustekniikat kärsivät kyllästymisilmiöstä, jossa uusien virheiden löytymisen määrä vähenee, kun testaustekniikan soveltuvuusalue kasvaa. Tämän takia on parempi käyttää useampia kuin yhtä testaustekniikkaa. [15] Kolmantena haasteena

on rakenteellisen testaus, sillä kun sovellukset rakentuvat useasta pienemmästä yksiköstä, pitää yleisessä testausteoriassa olla kappaleita pienempien yksiköiden testaukseen puhumattakaan yksiköiden yhdistämiseen tarvittavasta integraatiosta. [3, s.8] Neljäntenä haasteena universaalissa testausteoriassa ovat empiirisen todistusaineiston puute, sillä jokaisessa ohjelmistotekniikan aiheen tutkimuksessa tarvitaan empiirisiä tutkimuksia ehdoteltujen tekniikoiden ja käytäntöjen arviointiin. [16]

Toisena tavoitehaaveena ohjelmistotestauksessa on testauspohjainen mallinnus, mikä tarkoittaa ideaalin mallin rakentamista, jotta sovellus voidaan testata tehokkaasti. Tämä taas on testaajan näkökulmaan perustuva käännös mallipohjaisesta testauksesta, johon nykyään keskitetään suuria määriä resursseja. Mallipohjainen testaus tarkoittaa sovelluskehityksessä määriteltyjen mallien käyttöä testausprosessin ajoon, erityisesti testitapausten luomiseen automaattisesti. [17, s.7] Toisen haaveen ensimmäisiin haasteisiin kuuluvat juuri itse mallipohjaisen testauksen käyttöönoton käytännön ongelmat, joita ovat muun muassa eri tyylisten mallien yhdistäminen sekä integrointi nykyisiin ohjelmistoprosesseihin, missä testauksen hallinnassa mallien tulisi olla mahdollisimman abstrakteja ja samalla säilyttää kyky luoda suoritettavia testejä. Muita haasteita testauspohjaisessa mallinnuksessa on mallien puuttuminen kokonaan tai niihin pääsy on estetty muun muassa COTS-peräisissä sovelluksissa sekä arviot testien lopputulosten onnistuvuudesta. [3, s.9-10]

Kolmantena tavoitehaaveena ohjelmistotestauksessa on testauksen suorittaminen täysin automatisoidusti. Sovellusten monimutkaistuessa myös niiden kehittämistä on automatisoitu erilaisilla kehitystyökaluilla, joilla voidaan luoda monimutkaista koodia suuria määriä pienellä vaivalla. Tämä taas on lisännyt huolta testauksen jäämisestä jälkeen kehityksessä, jonka takia iso osa nykyisistä testauksen kehittäjistä on pyrkinyt parantamaan testauksen automatisointia kasvavalla asteella esimerkiksi kehittämällä tekniikoita testisyötteiden luomiseen automaattisesti. Lisäksi kehityksessä on pyritty vielä edemmäksi testien luomisesta ja etsimällä innovatiivisia tukiprosesseja automatisoida testausprosesseja. Lupaavina askeleina täydellisen automatisoinnin saavuttamiseksi ovat antaneet yksikkötestit, jotka ovat välttämättömiä sovelluksen laadun varmistamiseksi ja antavat jo alussa paljon tietoa vaikeasti myöhemmin havaittavista virheistä. Yksikkötestaus tosin vaatii kustannuksiin nähden ylimääräistä koodausta suoritusympäristön simuloimiseksi ja yksikön tulosteiden tarkistamista. Haasteina täysin automatisoidun testauksen saavuttamisessa ovat syötteiden

luominen, aluekohtaiset testauksen lähestymistavat ja testaus ajantasaisesti esimerkiksi verkossa. [3, s. 11]

Pääasiallisena tavoitteena ohjelmistotestauksen kehittämisessä on kustannustehokkaasti kehittää käytännöllisiä testausmetodeja, -työkaluja ja prosesseja korkealaatuisen sovelluksen kehittämiseksi. [18] Tästä syystä neljänenä tavoitehaaveena ohjelmistotestauksen kehityksessä on tehokkuuden maksimointi, jonka tekee vaikeaksi ensimmäisenä haasteena oleva alati kasvava modernien sovellusten monimutkaisuus ja sovelluksen koodin laadun ylläpito. Toisena haasteena tehokkuuden maksimoimisessa on käyttäjäpopulaation ja resurssien hallinta, esimerkiksi miten kerätä sovelluksen käyttötietoa sekä hallita ja muuttaa tämä raaka data relevantiksi informaatioksi. Kolmantena ja neljäntenä haasteena ovat testausmallien hallinta oikean lähestymistavan valitsemiseksi ja testauskustannusten ymmärtäminen, sillä monesti testaamista katsotaan kuluneutraalista näkökulmasta eikä oteta huomioon kulujen vaikutusta rajoitteina testauksessa. [3, s.13-15]

Viimeisenä haasteena ohjelmistotestauksen tehokkuuden maksimoimisessa on ohjelmistotestauksen opetus. Niin kuin saatavilla olevat tekniikat, työkalut ja prosessit, myös testaajien taidot, omistautuminen ja motivaatio vaikuttavat suuresti onnistuneen ja tehottoman testausprosessin eroihin, jonka takia on työskenneltävä yhtä paljon niin tekniikan kuin ihmispotentiaalin saralla. Testaajat tulee kouluttaa ymmärtämään sekä testaamisen peruskäsityksiä ja rajoituksia että saatavilla olevien tekniikoiden mahdollisuuksia. Samalla kun tekniikka kehittyy, tulee myös seuraavan sukupolven testaajien olla tietoisia ja ottaa näitä tekniikoita käyttöön, jotta myös testaamisen käytännön taso etenee. Tästä syystä koulutuksen on oltava jatkuvaa, jotta voidaan pysyä kehittyvän testaamisen mukana. [3, s. 15]

2.2 Ohjelmistotestauksen opetus

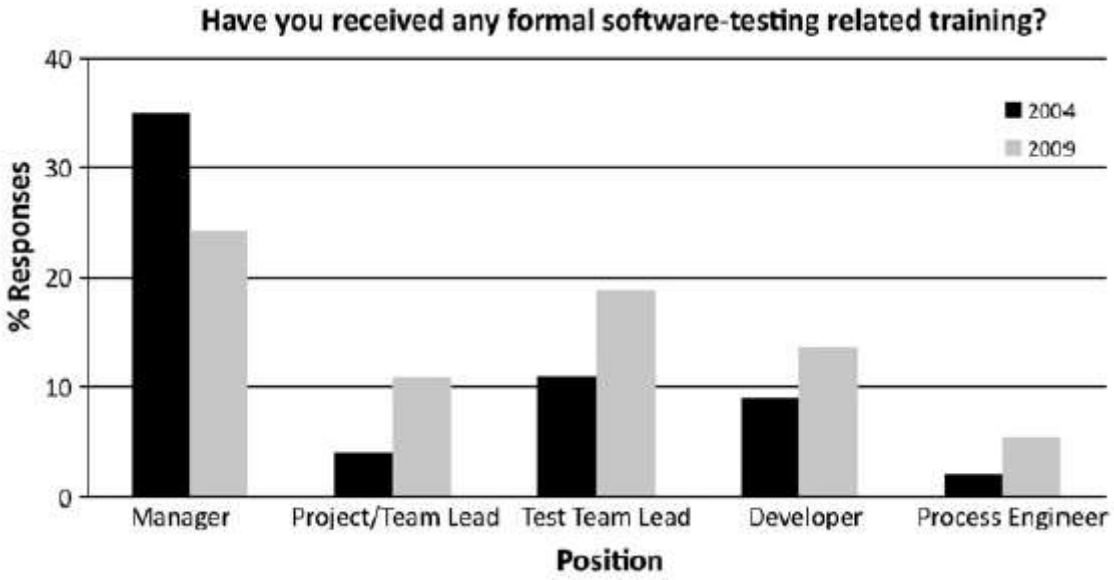
Viimeisen neljänkymmenen vuoden aikana lukemattomat systeemien kaatumiset ovat johtaneet katastrofaalisiin seurauksiin niin kustannusten kuin ihmishenkien menetyksen osalta, jotka kaikki olivat selvitetty johtuneen systeemeissä olevista virheistä. Yhtenä syistä virheiden suureen määrään ja niistä aiheutuviin kustannuksiin voidaan pitää testauksen puutetta kehityksessä ja koska sovellusten laatu on dominoiva onnistumiskriteeri ohjelmistoalalla, on ohjelmistotestaus tärkeä prosessi sovellusten laadun parantamisessa. [19] Jotta informaatiotekniikan alalla voidaan tuottaa korkealaatuisia tuotteita, yritykset tarvitsevat uusia insinööriopiskelijoita, joilla ovat hyvät ongelmanratkaisu-, debuggaus- ja

analyyttiset kyvyt. Tietokonesysteemien kasvaessa ja monimutkaistuessa testaamisesta on tullut yhä vaikeampaa, sillä vaikka ohjelmistokehityksessä kehitysperiaatteiden, -metodien ja työkalujen edistyvä kehitys on tehnyt ohjelmoijista näkyvämpiä kuin koskaan ennen, eivät työkalut laadun varmistamiseen ole pysyneet vauhdissa. Puhumattakaan testauksen formaalin opetuksen puuttumisesta, kun testaustaidot opitaan usein sovellusten implementaatioon painottuvilla kursseilla. [20]

Ohjelmistotestauksen opetuksessa kohdataan monia ongelmia ja niiden korjaamiseen on asetettu useita erilaisia tavoitteita. Kuten aiemmin mainittiin, informaatiotekniikan alalla korkealaatuisten tuotteiden kehittämiseksi yritykset tarvitsevat palkattavaksi juuri-valmistuneita hakijoita, joilla ovat hyvät ongelmanratkaisu-, debuggaus- ja analyyttiset kyvyt. Monet tietotekniikan ja tietokonetieteen vasta-valmistuneet kyllä omaavat astuessaan työelämään erinomaiset kehitystaidot, mutta ovat puutteellisia taidoiltaan testauksessa, debuggauksessa ja analyyttisissa taidoissa. Tähän osittain syynä on nykyisten akateemisten opetussuunnitelmien painotus sovelluksen kehitykseen testauksen kustannuksella virallisena tekniikan tieteenhaarana. Suurin osa tämän päivän opetussuunnitelmista painottaa varhaisia vaiheita ohjelmistokehityksen elämäsyklissä, kuten: vaatimusten kerääminen, arkkitehtuurin suunnittelu ja toteutus. Lisäksi joissakin tapauksissa on olemassa kursseja keskittyen ainoastaan yhteen edellä mainituista vaiheista, mutta valtaosa opetussuunnitelmista keskittyy suunnitteluun ja toteutukseen. [20]

Ensimmäisenä ongelmana testauksen opetuksessa on aiemmin mainittu taitojen puute ohjelmistotestauksen osalta vastavalmistuneilla. Testauksen opetusta tarjotaan sekä akateemisessa että teollisessa kontekstissa ja itse opetuksen ja harjoittelun tasosta tehdyissä julkaisuissa [21-23] voidaan havaita perustelut testaustaitojen puutteille akateemisen koulutuksen jälkeen. [24] Kuitenkin kohentumista formaalin koulutuksen osalta voidaan havaita, kun verrataan kuvassa 6. eri työntekijöiden saamaa formaalia ohjelmistotestaukseen liittyvää koulutusta vuosina 2004 ja 2009 tehtyjen tutkimusten välillä. Verrattuna vuoteen 2004, pienempi osa managereista on saanut koulutusta, mikä taas johtuu vähenevistä työtehtävistä suorittaa testausta itse. [25] Taitojen puutteille on monia syitä, joita ovat muun muassa luokkaopetuksen ja työelämän tehtävien sisällön erilaisuus sekä vajavaisuus laboratoriotilojen ja käytännön harjoittelun osalta. Nämä seikat asettavat opiskelijat

alalyöntiasemaan työnantajien kanssa ja kokemuksen puute työkalujen osalta estää opiskelijoita pääsemästä kiinni testausprosesseista ja testauksen vaiheista. [26]



Kuva 6. Formaali koulutus ohjelmistotestauksessa. [24]

Sen lisäksi, että monien yliopistojen tietotekniikan osastojen ohjelmistotestauskurssit eivät ole itsenäisiä omalla tavallaan, vaan sisällytetty sovellustekniikkaan, keskitytään joissakin yliopistoissa testauskursseilla liikaa testausteorian selittämiseen ja testausmetodien esittämiseen. Tällöin opiskelijat jäävät vaille systemaattista harjoittelua, kun testauksen opettamisessa ei keskitytä sovellustestaukseen käytännössä. Kun teorian ja käytännön välillä opetuksessa ei löydy kunnollista yhteyttä, johtaa tämä opiskelijoiden kiinnostuksen puutteeseen testauksesta. [26] Lisäksi perinteinen opetustekniikat eivät välttämättä motivoi tarpeeksi opiskelijoita, jolloin kiinnostusta pyritään nostamaan esimerkiksi testaukseen liittyvillä peleillä ja pelillistämisellä, jotka rohkaisevat opiskelijoita jatkamaan testauksen opiskelua luokkien ja luentojen ulkopuolella. [27]

Jotta ohjelmistotestauksen opetuksessa opiskelijat saadaan ratkaisemaan konkreettisia tosi elämän ongelmia, täytyy käytettävien työkalujen, harjoitusten, projektien ja kurssitehtävien olla mahdollisimman käytännöllisiä ja realistisia. Tämän takia testaukseen opetuksessa, erityisesti vain testaukseen keskittyvillä kursseilla, on pyritty käyttämään mahdollisimman realistisia SUT-, eli *Systems Under Test*, -järjestelmiä sekä käyttämällä kaupallisia testaustyökaluja, sillä julkisesti käytettävissä olevia opetusohjelmistoja, joita kouluttajat

voivat adaptoida ja kustomoida, on vähän. Kuitenkin ongelmana tässä on, etteivät olemassa olevat testauslaboratoriot ole päivitettyjä viimeisimmillä ja uusimmilla työkaluilla sekä teknologioilla. Lisäksi laboratorioiden testausympäristöä ei ole rakennettu perustuen realistisiin SUT-järjestelmiin vaan käyttäen ”lelu”-esimerkkejä kyseisistä järjestelmistä. Vaikka kaupallisten työkalujen ja realististen SUT-järjestelmien käyttöönottoa ei voida yleistää, on Vahid Garousin artikkeli antanut hyviä ennakkotapauksia esimerkiksi Calgaryn yliopiston ottaessa käyttöön toimialan tehokkuuden tasoisia työkaluja ja ison skaalan SUT-järjestelmiä, sillä Calgaryn kaupungilla on hyvin aktiivinen testaustoimiala ja kurssin osanottajien odotettiin työskentelevän ohjelmistotestausalalla. [28]

2.3 Opetuskehykset ja -metodit

Yhteenvetona ja ratkaisuna edellisiin ongelmiin on yhä ilmeisempi ohjelmistotestauksen opetuksen kehittäminen, jossa välttämätöntä on kehittää testausympäristöjä ja –työkaluja sovellustestauksen harjoittelun ja opetuksen helpottamiseksi ja näin tarjoamalla opiskelijoille ja ammattilaisille taidot ja motivaation työskennellä sovellustestauksessa. Kehityksen toteuttamiseksi ja sovellustestauksen integroimiseksi osana kandidaattiopintojen kurssia on käytetty monia eri työkaluja, opetuskehyksiä ja –metodeja, joista voidaan luokitella monia erilaisia päälähestymistapoja. Näitä ohjelmistotestausken opetusta tukevia lähestymistapoja on Pedro Vallen, Ellen Barbosan ja Jose Maldonadon artikkelin mukaan 11, joita ovat: koulutukselliset moduulit, koulutukselliset pelit, testausjohtoinen kehitys, ohjelmistokehityksen ja –testauksen integroitu opetus, vertaisarvioinnit, tietovisat, tutoriaalit, ongelmapohjainen oppiminen, sosiaaliset verkostot, ohjelmistoresidenssit sekä suorituspohjainen oppiminen. [27] Pääasiallisesti eniten käytettyjä lähestymistapoja ovat aiemmin mainituista 4 ensimmäistä, joiden lisäksi yhtenä uutena metodina käyttöön on ilmestynyt projektipainotteinen CDIO eli *Conceive – Design- Implement – Operate.* [26]

2.3.1 Koulutukselliset moduulit

Ensimmäisenä pääasiallisena lähestymistapana ovat koulutukselliset moduulit, jotka ovat tiiviitä opetusyksiköitä. Nämä opetusyksiköt taas koostuvat teoriasisällöstä yhdistettynä käytännön aktiviteetteihin ja arviointeihin, joita kaikkia tuetaan teknologisilla ja tietokonepohjaisilla resursseilla. [30] Gustavo Lopezin, F. Cocozzan, A. Martinezin ja M. Jenkinsin julkaisemassa artikkelissa on tutkittu ja arvioitu lähestymistavan käyttöä ohjelmistotestausharjoituskurssilla, jossa opetuksen kohteena oli ryhmä ohjelmistokehittäjiä, joilla oli vähän tai ei ollenkaan taustaa ohjelmistotestauksessa. Kurssin ensimmäinen moduuli oli

teorian opiskelua, joka koostuu ohjelmistotestauksen periaatteista, testaustyypeistä, -tasoista ja suunnittelutekniikoista. Toinen moduuli oli käytännöllistä workshop-harjoittelua, jonne opiskelijat toteuttivat ensimmäisen moduulin teoreettisia konsepteja käyttäen erikoistyökalua, joka taas tuki koko testausprosessia. Modulaarinen rakenne oli valittu ajatellen soveltamista eri teollisuuden haaroihin ja kurssia itseään oli arvioitu koulutettavan, kouluttajan ja managerin näkökulmasta. [31]

Tutkimuksen antamassa yhteenvedossa huomioitiin monia varteenotettavia seikkoja ajatellen tulevia kursseja, joista ensimmäisenä oli lyhyiden luokassa käytävien harjoitusten lisääminen, sillä luokka-aikaa ei katsottu riittäväksi teollisen harjoittelun asetuksilla koskien korkeasti teknisiä asioita kuten ohjelmistotestausta. Lisäksi teorian ja käytännön osioiden huomattiin tarvitsevan opetusta yhdessä kuin erikseen, minkä takia tuleville kursseille kaavailtiin opetusrakenteeksi kahden session jakoa, joista ensimmäisessä esiteltäisiin tietty opetettava aihe. Toinen viikoittainen harjoitussessio koostuisi harjoitustehtävistä, kyselysessioista ja aiheen yhteenvedosta. Kolmantena huomioitavana seikkana oli, että testauksen hallintaan ja johtamiseen liittyvät aiheet, vaikkakin välttämättömiä testausprosessin ymmärtämiseksi, eivät olleet testejä suorittavien henkilöiden mielestä tärkeitä. Tämän takia tulevilla kursseilla tulee keskittyä artikkelin mukaan enemmän teknisiin aiheisiin. [31]

Edellä mainittujen seikkojen on artikkelissa arvioitu parantavan kurssia niin, jotta sitä voidaan tarjota muille ohjelmistokehitysorganisaatioille ja näin edelleen lisätä tietoa ja kokemusta opetettaessa yrityksiä. Kuitenkin huomioitavaa ja lopullisena mainintana artikkelissa on kurssilla käytettävien ja tarvittavien sovellusten ja laitteistoinfrastruktuurin kokoaminen sekä ylläpito, mikä vaatii paljon aikaa ja vaivaa puhumattakaan rahallisista kustannuksista. Siksi välttämättömän huomion kiinnittäminen infrastruktuuriin tulee tässä olemaan aina kriittinen kurssin onnistumisen kannalta. [31]

2.3.2 Koulutukselliset pelit

Monesti motivaation puute on opiskelijoilla esteenä oppia ohjelmistotestauksen sisältöä, minkä johtavina syinä ovat teorian ja käytännön sisällön epäyhteneväisyys, luokassa opetetut asiat eivät täsmää työelämässä vaadittuihin, opiskelijoilla on hankala ymmärtää testausprosessia ja –tasoja sekä kokemuksen puute ohjelmistokehityksessä. Tästä syystä koulutusta on pyritty saamaan kiinnostavammaksi erityisesti koulutuksellisilla peleillä ja

kurssien pelillistämisellä sekä samalla lisäämään opiskelijoiden halua opiskella itse ja aktiivisesti. [32] Koulutuksellisilla peleillä on tarkoitus helpottaa konseptin ja ideoiden oppimista tarjoamalla houkuttelevat kasvatukselliset toimintatavat. Näissä toimitavoissa käyttäjät voivat oppia paljon aktiivisemmin, dynaamisemmin ja olemalla motivoituneita, ja näin pystyvät saamaan tietoa hauskuuden kautta. On kuitenkin huomioitava, että useasti pelit opettavat vain toiminnallisen testauksen konsepteja, eikä niillä välttämättä ole yhtenevää opetusmetodiikkaa muille kursseille. Tällöin opiskelijoilla ei ole yhtenevää näkemystä ohjelmistotestauksen konsepteista muiden oppiaineiden kanssa. [29]

Ensimmäisenä esimerkkinä koulutuksellisten pelien käytössä on A. Soskan, J. Mottokin ja C. Wolffin artikkelin kokeellisen korttipelin käyttö ohjelmistotestauksen opetuksessa. Korttipeli on kilpailullinen, jossa tulee kerätä tietty määrä pisteitä pelaten ja suorittaen korteissa olevia tehtäviä ja näitä tehtäviä voi pelata yksinään tai ryhmässä riippuen pelin tilanteesta. Artikkelin mukaan integroimalla pelin luentoihin opiskelijat tulevat vaihtamaan kokemuksia ja tietoa toistensa kanssa, adoptoimaan pelin pelaamisen osaksi itsenäistä opiskelun etenemistä, tulevat enemmän motivoituneiksi pelata uudestaan ja lisätä tietotasoaan sekä sisäistävät opittua tietoa selittämällä sitä muille. Korttipelin käytön tuloksista kerrottaessa pelin kokeilu todisti monia perinteisen opetuksen heikkouksia ja toi etuja ryhmäopiskelun ja mukavamman opiskelun muodossa. Lisäksi peli näytti yleisesti motivoivan opiskelijoita ja syventämään heidän tietämystään. Tosin ensimmäisistä lupaavista tuloksista huolimatta pelin integroiminen vaatii vielä muokkausta muuan muassa korttien antamien bonuspisteiden tasoittamisessa ja yleisessä pelattavuudessa. [33]

Toisena esimerkkinä koulutuksellisten pelien käytöstä opetuksessa on G. Fraserin, A, Gambin ja J. M. Rojasin artikkelissa julkaistu raportti *Code Defenders*-pelin käytöstä testauksen opetuksessa. Peliä käytettiin Passaun ylipistossa 120 opiskelijan kurssilla, joka sisälsi 10 kahden tunnin mittaista pelisessiota. *Code Defenders* on verkossa vapaasti käytettävissä oleva ja Java-ohjelmointikieltä hyödyntävä peli, jota voi pelata kaksin vastakkain tai joukkueina. Pelissä pelaaja toimii joko hyökkääjän tai puolustajan roolissa, joista hyökkääjän näkymä on kuvassa 7 ja puolustajan kuvassa 8. Hyökkääjän tarkoituksena on luoda virheitä sisältävää koodia, joka on pelissä testauksen alla olevaa virheellinen Java-luokka eli ”mutantti.” Puolustajan tarkoitus on taas tuhota nämä mutantit luomalla virheitä havaitsevia JUnit-työkalun yksikkötestejä. Pisteitä osapuolet saavat tuhotuista ja läpipäässeistä

mutanteista ja saatujen pisteiden määrä vaihtelee muun muassa mutanttien sisältämän virheiden määrän ja vaikeuden perusteella. [34]

Kuva 7. Hyökkääjän näkökulma Code Defenderissä [34]

Kuva 8. Puolustajan näkökulma Code Defenderissä [34]

Vaikka artikkeli ei sisällä yksityiskohtaista analyysia opiskelijoiden suorituksista, innokkuudesta ja havainnoista, kokonaisuudessaan opiskelijat nauttivat saamastaan kokemuksesta ja olivat päällisin puolin hyvin kiinnostuneita. Opiskelijoiden koodaamisen taitotasojen eroista huolimatta hyötyivät myös perusyksikkötestien kirjoittamisen kanssa tus-

kailevat opiskelijat joukkuepohjaisessa pelimuodossa, sillä seuratessa oman joukkueen testejä nämä opiskelijat saivat hyvin neuvoa ja ohjeistusta kokeneemmilta joukkuetovereiltaan. Muina hyötyinä pelin käytöstä olivat opiskelijoiden kasvava halu parantaa itseään ja taitojaan pelin kilpailuhenkisyyden ja pistelistan kautta sekä suurempi tuotetun koodin määrä verrattuna perinteisiin kursseihin. Parannuksina pelin hyödyntämisessä testauksen opetuksessa pidettiin artikkelissa ensimmäisenä testien lisääminen hyökkäyksiin sattumanvaraisten mutaatioiden välttämiseksi. Toisena parannuksena oli monimutkaisuuden lisääminen peliin muun muassa kehittyneempiä testaustekniikoita käyttämällä, esimerkiksi lisäämällä integraatiotestauksen ja tietokantojen käytön testaukseen. [34]

2.3.3 Testausjohtoinen kehitys

Testausjohtoinen kehitys, eli *test-driven development* (TDD), on laajasti tunnettu tehokkaana sovelluskehitysmuotona parantaa ohjelmiston luotettavuutta. Testausjohtoisessa kehityksessä kehittäjät testaavat koodiaan sitä mukaa kun he kirjoittavat sitä, mikä tapahtuu yleensä kirjoittamalla ja suorittamalla yksikkötestejä, joissa taas generoidaan testien syötteet, luodaan odotetut tulosteet, suoritetaan annetut syötteet ja verrataan saatuja tulosteita odotettuihin. Suuren työtaakan välttämiseksi ja yksikkötestien suorittamiseksi automaattisesti käytetään erilaisia testaustyökaluja muun muassa testien automatisointiin ja syötteiden luomiseen automaattisesti. Etuina testausjohtoisessa kehityksessä ovat suurempi luottamus sovellusyksikön toimivuuteen testin alla ja virheiden korjaamisesta aiheutuneiden kustannusten väheneminen, kun virheet havaitaan aikaisin. [35]

Jotta testausjohtoista kehitystä voidaan opettaa tehokkaasti, tarvitaan tähän sekä tukea työkalujen käytön osalta että riittävän taitotason ohjelmistokehitystaitoja. Tästä syystä kouluttajien tulee rakentaa tehokkaita tapoja kumpienkin taitojen opettamiseksi opiskelijoille. [35] S. Kollanuksen ja V. Isomöttösen julkaisemassa artikkelissa TDD:n soveltamisesta koulutukseen, Jyväskylän yliopistossa järjestettiin kurssi 52 opiskelijalle ja kurssi oli tähdätty neljännen vuoden maisteritason opiskelijoille. Vaikka kurssin osanottajien koulutuksen vuositaso vaihteli ja TDD:n hyödyntäminen oli tuolloin monille vielä uutta, omasivat monet kurssin esivaatimuksia paremmat ohjelmointitaidot. Kurssissa TDD esitettiin pikaisesti yhdellä luennolla sisältäen pieniä esimerkkejä, jonka jälkeen opiskelijat suorittivat kaksi kurssitehtävää. Ensimmäinen kurssitehtävä oli helpompi lämmittelytehtävä TDD:aa hyödyntämällä ja käyttämällä JUnitin sisältävää Eclipse-testaustyökalua. Toinen

kurssitehtävä oli varsinainen kurssityö, jossa tuli kehittää ja toteuttaa yksinkertainen http-palvelin hyödyntäen TDD:n käyttöä. [36]

Kurssin jälkeisessä kyselyssä ja tutkimuksen lopputuloksissa todettiin, että TDD:n hyödyntämisen suurimmat ongelmat opiskelijoilla olivat teknisiä, mitkä vaikuttivat heidän kykyynsä hyödyntää TDD:aa sulavasti kurssitehtävissä. Lisäksi koska TDD itsessään on paljon työtä vaativa kehitystapa, sen käyttö oli erittäin vaikeaa opiskelijoille samanaikaisesti teknisesti haastavissa tehtävissä. Vaikka opiskelijat totesivat jälkikyselyissä kehitystavan parantaneen koodinsa laatua ja TDD:n vaikutukset yrityskäytössä ovat kiistattomia, opetuskäytön suunnittelu ja soveltaminen tuovat runsaasti haasteita. [36] Muun muassa K. Keefen, J. Sheardin ja M. Dickin artikkelissa julkaistussa tutkimuksessa sovellettiin useita XP- eli *Extreme Programming*-kehitystapoja, joista TDD katsottiin kaikista vaikeimmaksi soveltaa opiskelijoihin. [37] Tästä syystä Kollanuksen ja Isomöttösen artikkelissa ehdotetaan TDD:n käytön opettelua aikaisemmin opintosuunnitelmissa, mutta ei liian aikaisin. TDD:n aikaisempi soveltaminen tekisi siitä rutiininomaisempaa eikä sen käyttö olisi siten väkinäistä kurssitehtävissä. [36]

Toisena esimerkkinä testausjohtoisen kehityksen käytöstä opetuksessa on S. H. Edwardsin julkaisema artikkeli, jossa otetaan erityisesti kantaa opiskelijoiden tapaan käyttää ”yritys ja epäonnistuminen”-menetelmää testauksessa sekä syitä opiskelijoiden haluttomuuteen hyödyntää testauskäytäntöjä harjoituksissa, mitä voivat olla esimerkiksi opiskelijoiden vajavaiset taidot ohjelmoinnissa sekä ajan puute ja suuri työmäärä. Jotta opiskelijat sisällyttäisivät ohjelmistojen kehitykseen testausta ja laatisivat enemmän testitapauksia, ehdotetaan artikkelissa TDD:n sisällyttämistä opetukseen testauksessa, mikä tosin vaatii tukea testaustyökalujen osalta sekä jatkuvaa arviointia ja kommentointia opiskelijan sen hetkisestä suorituksesta. TDD:n käytön tekemiseksi käytännölliseksi, käytettiin artikkelin kurssilla Web-CAT-systeemiä, eli *Web-based Center for Automated Testing*, joka arvostelee opiskelijan koodia ja testejä perustuen koodin ”oikeuteen”, testien koodikattavuuteen sekä testien täydellisyyteen ja validiteettipisteisiin ongelmasta johtuen. Web-CAT:n käytettäessä testattava koodi ja testi tulee palauttaa samalla kertaa. [38]

Edwardsin artikkelin lopussa julkaistuissa lopputuloksissa TDD:n soveltamisesta opetuskäyttöön todettiin kurssin antamien kokemusten olleen varsin positiivisia. Monesti

tietotekniikan opiskelijat eivät tule saavuttamaan haluttua analyyttista ajattelua opintojensa aikana, kun vasta myöhemmin, mutta TDD-aktiviteeteilla voidaan ylläpitää jatkuvaa harjoittelua sekä käsityskyvyn ja analyysien parantumista ohjelmointitehtävien edetessä. Lisäksi arvosanojen ja palautteen generoinnin automatisoinnin huomattiin olevan myös suuressa arvossa, sillä helposti saatavilla olevan jatkuvan palautteen huomattiin rohkaisevan opiskelijoita Web-CAT:in käyttöön. Tämä yhdessä TDD:n kanssa osoitti taulukon 1 perusteella huomattavaa koodin laadun parantumista, minkä lisäksi opiskelijoiden itsevarmuus muuttaessaan koodiaan nousi. Erityisesti 73,5% opiskelijoista halusi artikkelin mukaan käyttää Web-CAT:iä ja TDD:iä tulevien kurssien ohjelmointitehtävissä, vaikka se ei olisi vaadittavaa. Kuitenkin artikkelin yhteenvedossa todetaan, että toimiakseen tämä kokonaisuus vaatii ehdottomasti hyvin tuetut ja saatavilla olevat systeemit ja työkalut. [38]

Taulukko 1. Tulosvertailu molempien ryhmien välillä (lihavoidut erot ovat huomattavia).) [38]

Comparison	Spring 2001 Without TDD	Spring 2003 With TDD	t-score Assuming Unequal Variances	Critical t-value p = 0.05
Recorded grades	**90.2%**	**96.1%**	t(df = 62) = **2.67**	2.00
TA assessment	98.1%	98.2%	t(df = 65) = 0.06	2.00
Curator assessment	93.9%	96.4%	t(df = 71) = 1.36	1.99
Web-CAT assessment	**76.8%**	**94.0%**	t(df = 61) = **4.98**	2.00
Time from first submission until assignment due	2.2 days	4.2 days	t(df = 112) = **3.15**	1.98
Test case failures from master suite (out of 1064)	**390 (36.7%)**	**265 (24.9%)**	t(df = 84) = **3.48**	1.99
Estimated Defects/KSLOC	**70.0**	**38.3**		

2.3.4 Ohjelmistotestauksen ja ohjelmoinnin yhdistetty opetus

Neljäntenä vaihtoehtona ohjelmistotestauksen opetuksessa on testauksen ja ohjelmoinnin yhdistetty opetus, jossa molempien aiheiden peruskonseptit opetetaan yhdessä. E. Barbosan, C. Corten ja J. Maldadon julkaisemien artikkelien tutkimuksissa [39][40][41] on näytetty yhdistetyn opetuksen tarjoavan etuja kehittämällä opiskelijoiden analyyttisia taitoja ja ymmärrystä, sillä testaamisessa on välttämätöntä, että opiskelijat tietävät ohjelmansa toiminnan ja semantiikan. [29] Lisäksi yhdistetty opetus auttaa opiskelijoita ymmärtämään kehittäjien ja testaajien näkemyksiä ja asenteita, sillä kun esimerkiksi kehittäjät näkevät sovelluksessa tekemänsä koodin, testaajat eivät tätä näe. Erityisesti asenteissa kehittäjillä on parantamisen varaa, sillä kun päätavoite on koodin loppuun saaminen ja testaaminen nähdään vain aktiviteettina todistaa koodin toimivuus, on testauksen taso tällöin valitettavan heikkoa. Yhdistetyssä opetuksessa kehittäjät ja testaajat näkevät toistensa näkökulmat ja kurssi opettaa käyttämään kummankin osapuolen testaustekniikoita. [42]

Ensimmäisenä esimerkkinä yhdistetyn opetuksen hyödyntämisessä on N. Harrisonin artikkelissa julkaistu tutkimus, jossa kurssissa testausta opetetaan sekä kehittäjän että testaajan näkökulmasta. Kurssi jakautuu kolmeen vaiheeseen, joista ensimmäisessä on opiskelijoiden tehtävänä kehittää ja testata projektiohjelma kehittäjän roolissa. Tämän jälkeen toisessa vaiheessa opiskelijat saavat testattavakseen toisen opiskelijan kehittämän sovelluksen ja testaavat sen nyt testaajan roolissa. Kummassakin testauksessa opiskelijat laativat kohdeohjelmalle testaussuunnitelman ja testitapaukset ja työt suoritetaan yksin, jotta opiskelijat pääsevät demonstroimaan omat testauskykynsä eivätkä ryhmätyöskentelyn ongelmat aiheuta häiriöitä. Lisäksi opiskelijoille annetut projektit olivat erilaisia, joista eri tyyppien määrä oli puolet osanottajien määrästä. [42]

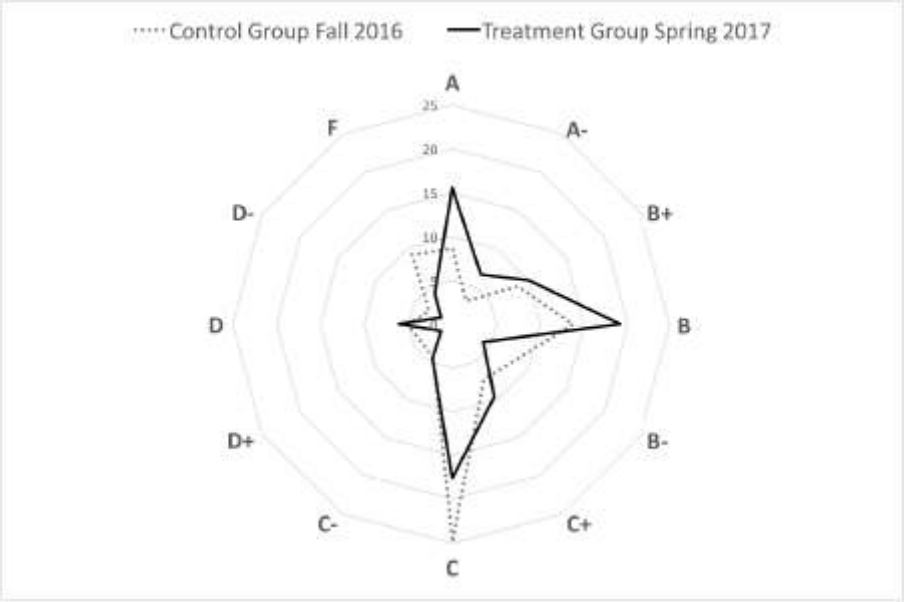
Koska testattavat ohjelmat ja projektitehtävät vaihtelevat laadultaan opiskelijoiden välillä, ei testaussuorituksia arvioida löytyneiden virheiden, vaan testaussuunnitelmien ja –raporttien laadun mukaan. Ohjelman oikeus ja toiminta arvioidaan vain kehitysvaiheessa. Kolmannessa vaiheessa opiskelijat kuvailevat ja perustelevat lyhyessä raportissa lähestymistapojaan testauksessa, jonka jälkeen heidän tulee vertailla molempia käyttämiään testausmenetelmiä. Tämä antaa opiskelijoille mahdollisuuden analysoida lähestymistapojaan kriittisesti, jonka jälkeen he pääsevät oppimaan muiden käyttämistä menetelmistä, kun kaikki kurssin osanottajat antavat luokkaesityksen projekteistaan niin testaavan kehittäjän kuin testaajan roolissa. [42]

N. Harrisonin artikkelin kuvailemaa tapaa on käytetty viiden vuoden ajan ja kurssilla on ollut suuri onnistumisprosentti opiskelijoiden keskuudessa. Opiskelijat oppivat käyttämään ja arvostamaan testaajien ja kehittäjien käyttämiä eri testaustekniikoita, lähestymistapoja ja laadun varmistamisen aktiviteetteja. Lisäksi kurssi avaa mahdollisuuden oppia eri tyypin projekteista ja niiden lähestymistavoista. Tosin kurssissa ongelmina ovat projektien erilaisuuden takia työmäärän jakaminen tasaisesti ja samanlaisen oppimiskokemuksen tarjoaminen opiskelijoille, mitkä voivat myös aiheuttaa vaikeuksia ja epämukavia kokemuksia opiskelijoille. Jos taas projektitehtäviin liittyy graafisten käyttöliittymien käyttöä, vähentää tämä mahdollisuuksia syötevirheiden määrässä. Seuraava iso ongelma opiskelijoiden huonoon suoritustasoon on opiskelijoiden huonot suunnitelmat ja projektitehtävien jättäminen viimeiselle minuutille. Lisäksi testaamisesta saatetaan tietyissä asioissa laistaa, mikä tosin myöhemmin kostautuu ja toimii näin hyvänä opettajana. Vaikka kursseilla löytyy aina

viimeiseen minuuttiin jättäviä, voidaan aiemmat mainitut ongelmat ehkäistä pitämällä tarkkaan silmällä projekteja ja varmistamalla niihin tarvittava kehityksen ja testauksen työmäärä. [42]

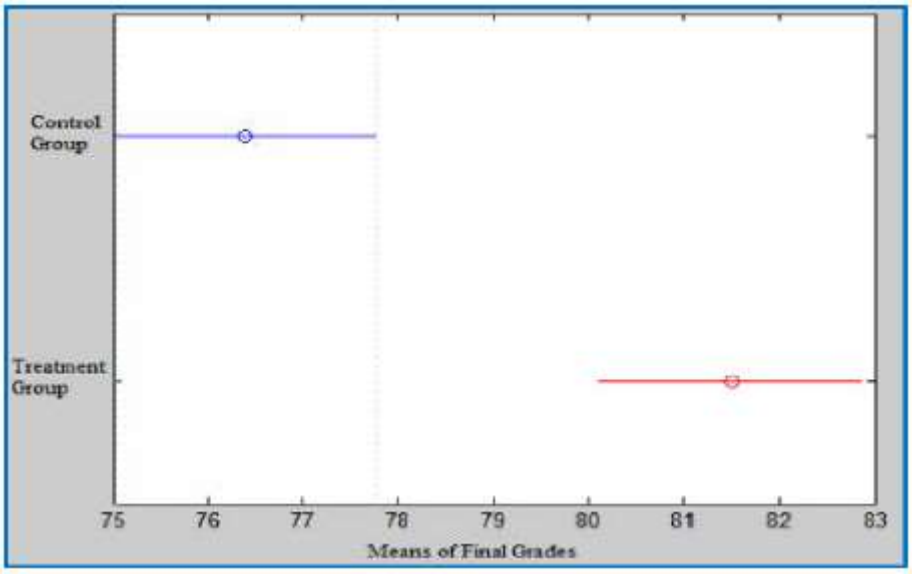
Toinen esimerkki ohjelmistokehityksen ja –testauksen yhdistämisestä on E. Barbosan, M: Silvan, C. Corten ja J. Maldonadon julkaisema artikkeli, jossa on mainittu monien yliopistotason perinteisten tietotekniikan opetuskurssien jättävän testauksen turhan usein taka-alalle ja aloitettavaksi liian myöhään sekä unohtaen käytännön opetuksen. Tästä syystä artikkelissa ehdotetaan PROGTEST-systeemin kehittämistä ja käyttöä, joka on verkkopohjainen ympäristö koodaustehtävien, jotka perustuvat testausaktiviteetteihin, palautukseen ja automaattiseen arviointiin, minkä tavoitteena on antaa riittävää palautetta arvioimaan opiskelijoiden suorituskykyä ohjelmoinnissa ja testauksessa. Vaikka alustaa ei ole vielä kehitetty, antaa artikkeli silti hyviä viiteideoita ja toteaa kriittisimmän asian ohjelmoinnin perusteiden ja testauksen yhdistetyssä opetuksessa olevan tarkoituksenmukaisen palautteen antaminen oppijan suorituskyvyn arvioimiseksi. [40]

Kolmantena esimerkkinä ohjelmistokehityksen ja –testauksen yhdistämisestä on V. Ramasamyn, H. Alomarin, J. Kiperin ja G. Potvinin julkaisema artikkeli, jossa ohjelmoinnin ja testauksen yhdistämisessä käytetään verkkopohjaista SEP-CyLE-systeemiä, joka tarkoittaa termiä *Software Engineering and Programmin CyberLearning Environment*. SEP-CyLE-järjestelmän tarkoituksena on parantaa testauksen opetusta ja integrointia ohjelmointikursseille tarjoamalla testaukseen liittyvää verkko-opintomateriaalia, joka tarkoittaa oppimisobjekteja sekä testaustyökalujen ohjeistuksia ja käyttöohjeita. Oppimisobjektit koostuvat opintomateriaalista ja tehtävistä, jotka ovat kaksiosaisia ja sisältävät käytännön harjoitustehtävän ja yhteenvedon. Tehtävien tekemiseen menee suunnilleen 5-15 minuuttia ja järjestelmään luotavien tunnuksien kautta voi ansaita pisteitä tehtävien suorittamisesta. Kursseille valittavat järjestelmän tarjoamat tehtävät valitaan kurssien järjestäjien toimesta riippuen kurssista, tehtävien vaikeudesta ja järjestäjän suosikeista. Järjestelmä on avoin niin kandidaatti- kuin maisteritason opiskelijoille ja kirjautumalla järjestelmään voi myös kommentoida ja keskustella keskustelupalstalla kanssaopiskelijoiden kanssa. [43]



Kuva 9. Suorituskyvyn arviointi loppuarvosanoja vertailemalla lisäopetusta saamattoman ja sitä saaneen välillä. [43]

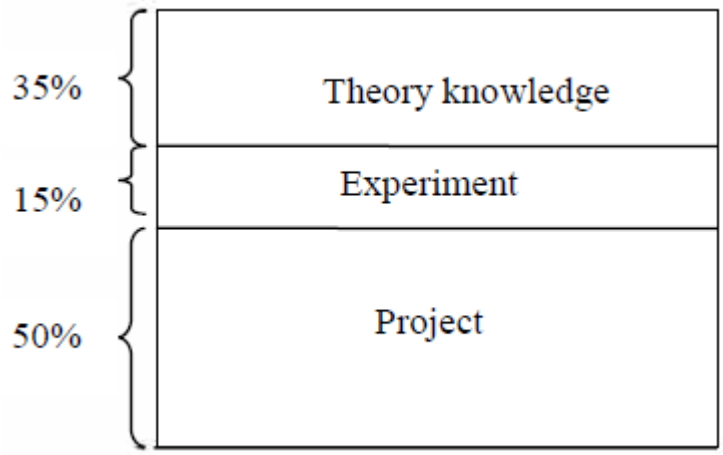
Artikkelin tutkimuksessa verrattiin SEP-CyLE-järjestelmän vaikutusta oppimiseen, kun toiselle kahdesta ryhmästä opiskelijoita annettiin samalla ohjelmointikurssilla lisäopetusta ohjelmistotestauksesta käyttäen SEP-CyLE-järjestelmän verkkotehtäviä ja ohjeistuksia. Keskikokoisen yliopiston ensimmäinen ryhmä suoritti kurssin syksyllä 2016 ilman testauksen lisäopetusta sekä SEP-CyLE-järjestelmää ja edellä mainittuja hyödyntävä ryhmä keväällä 2017. Tutkimuksen tuloksissa saatiin varsin positiivisia näkymiä, vaikkakin työmäärä lisääntyi toisella ryhmällä perinteisen ja verkko-opetuksen yhdistelmän muodossa. Opiskelijoiden tietämys testauksesta, joka liittyi ohjelmointiin, kasvoi ohjelmointikurssin opetuksen aikana huomattavasti, vaikka verkkotehtäviin oli käytetty keskimäärin vain 2 tuntia. Tämän lisäksi erot järjestelmää käyttäneiden ja käyttämättömien arvosanoissa olivat huomattavia, minkä voi havaita kuvista 9 ja 10. Niistä voidaan havaita, miten perustana oleva testauksen opetus paransi käsityksiä ja helpotti opiskelijoiden kykyä oppia ohjelmointikielen perusteita tehokkaammin. Yhteenvedossa todettiin ohjelmoinnin perusteiden parantamisen lisäksi testauksen lisäopetuksen helpottavan, ei vain välttämättömien testausstrategioiden oppimista ohjelmoinnin aikana, vaan myös inspiroivan opiskelijoita oppimaan ohjelmistotestauskursseilla. [43]



Kuva 10. Huomattava ero lisäopetusta saamattoman ja sitä saaneen välillä. [43]

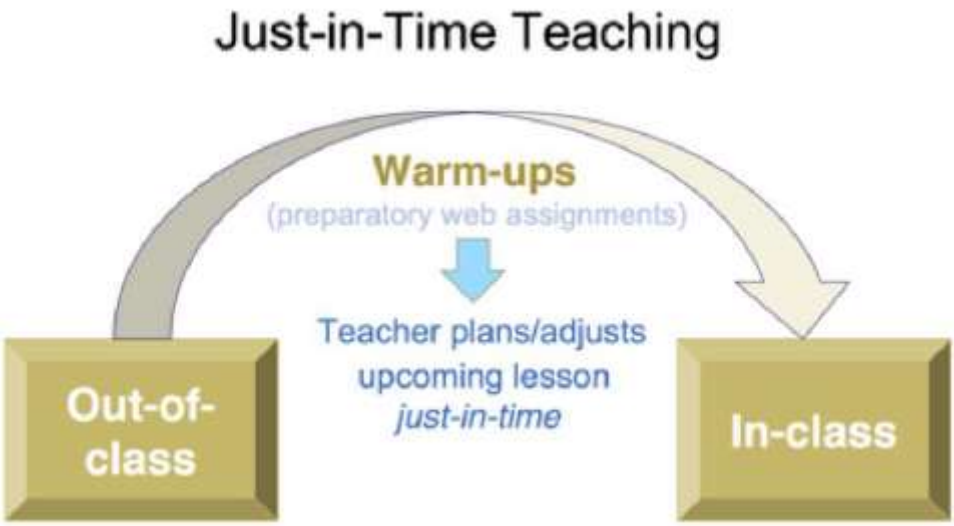
2.3.5 Muita opetuskehyksiä

Muita mainittavia opetuskehyksiä ohjelmistotestauksessa ovat aiemmin mainittu CDIO, JiTT eli Just-in-Time Teaching ja oikeiden ohjelmistoyritysten testausprojektien käyttö testausopetuksessa. CDIO-opetuskehyksessä pyritään testauksen opettamiseen suorittamalla pääosin ryhmätyönä pienempiä käytännön projekteja, joilla opiskelija tulee saamaan oikeaa kokemusta projektien suorittamisesta ja soveltamaan oppimiansa teorioita käytännössä. Samalla opiskelijat oppivat projektien kautta kommunikointi-, ryhmätyö- sekä innovaatio- ja ongelmanratkaisutaitoja. [26] Toisaalta CDIO-kehystä käytettäessä on otettava huomioon ja tarkkailtava monia projektien aikana tapahtuvia tekijöitä, joita ovat muun muassa opiskelijoiden testausympäristön valmistelu, valmiina olevien mallien ja testaustyökalujen käyttö, testien tulokset sekä raportoinnin ja palautteiden aikataulutus. Lisäksi opiskelijoille tulee järjestää kattava testaustyökalupaketti ja testausympäristö, joiden aikaansaamiseksi suositaan työasemien ja laboratorioiden järjestämistä projektien tekoon, mitä kuvaillaan esimerkiksi Z. Binin ja Z. Shimingin artikkelissa. Kuvan 11 mallista voi nähdä miten CDIO-malliin perustuva kurssin sisältö jakautuu ja miten paljon erityisesti käytännön harjoituksiin on panostettu. [44]



Kuva 11. Ohjelmistotestauskurssin sisältömalli perustuen CDIO-malliin (Zhu, Bin & Zhang, Shiming 2014 s. 489) [44]

Alexandra Martinezin artikkelin esittelemässä JiTT-menetelmässä tähdätään opiskeluun ennen varsinaista opettamista, mikä tarkoittaa tietyn kappaleen lukemista ja verkkoalustan nettitehtävien tekemistä ennen opetusta. Lukutestit tapahtuvat päivää ennen opetusta, sillä kurssin pitäjä katsoo vastaukset läpi ja suunnittelee seuraavan opetuksen sopivaksi niiden mukaan, mikä voidaan nähdä kuvasta 12. Näin päästään perille opiskelijoiden vahvuuksista ja heikkouksista sekä tekemällä tehtävistä avoimen esseen tehtäviä voidaan parantaa opiskelijoiden kirjoittamistaitoja, eli organisoida ideoita paremmin ja kirjoittaa selvemmin. [45]



Kuva 12. "Just-in-Time Teaching"-mallin komponentit ja työn kulku (Martinez, Alexandra 2018 s. 111) [45]

Havaittuja vahvuuksia kahdella järjestetyllä kurssilla JiTT-mallin käytöstä kurssin pitäjän osalta olivat opiskelijoiden lisääntynyt sitoutuminen luokkaopetuksiin sekä syvempien ja kiinnostavampien kysymysten esittäminen luokassa. Lisäksi opiskelijat pystyivät itse arvioimaan ymmärrystään oppiaiheista ja ilmoittamaan tästä ongelmatilanteissa. Opiskelijoiden mielestä heidän oppimisensa parani ja lukutestit olivat hyviä varmistamaan opittavat asiat. Ongelmina JiTT-mallin käytössä olivat kurssin pitäjälle kohdistuva työmäärä aiheutuen lukutestien tarkistamisesta ja luennon suunnittelu niiden pohjalta sekä arviointi lukutestien sopivuudesta ja vaikeudesta opiskelijoille. [45]

Toinen projekteja hyödyntävä opetuskehys on aitojen testausprojektien tekeminen yrityksille, mikä tarjoaa loputtomia mahdollisuuksia. Hyvin järjestettynä tämä opetuskehys on eduksi sekä opiskelijoille että yhteistyössä oleville yrityksille, sillä opiskelijat saavat kokemusta aitojen systeemien testaamisesta ja yritykset tuotteensa systemaattisesti testatuiksi. Projektit johtavat myös syvenevään yhteistyöhön yliopistojen ja yritysten välillä ja niistä voi saada paljon tutkimusmateriaalia uudempien testaustekniikoiden ja – teknologioiden kehittämiseksi. [46]

Hyödyistä huolimatta yritysten testausprojektien hyödyntämisessä kohdataan myös monia vaikeuksia ja ongelmia, joita ovat luottamuksen saaminen yrityksen ja yliopiston välille, yritystietojen salassapitovelvollisuudet, yritysten antama tekninen tuki ja projektien riskitasot. Vandi Garousin artikkelin tutkimuksessa havaittiin yritysprojekteja suoritettaessa, että yritykset eivät olleet antaneet kohdennettuja resursseja opiskelijoille testaamiseen ja että melkein kaikissa tapauksissa opiskelijat olivat ainoita testausresursseja vastakehitettyjen järjestelmien testauksessa. Vaikeuksista huolimatta kursseilla oli suuri vaikutus mukana olleiden opiskelijoiden uravalintoihin ja monet siirtyivät kehityspuolelta testauspuolelle. [46]

2.4 Yhteenveto ohjelmistotestauksen opetuksesta

Kuten jo aiemmin mainittiin, on testaukseen panostaminen välttämätöntä ottaen huomioon sen vajavaisuudesta ja ohjelmistojen virheistä aiheutuneiden kustannusten suuruuden. Testauksen tehokkuuden maksimoimisessa opetuksen kehittäminen on suuri haaste, jossa ongelmina ovat opiskelijoiden taitojen ja kiinnostuksen puute, sekä vanhanaikaiset työkalut ja tehtävät nykytekniikkaan ja –teknologiaan nähden. [19] Näiden ongelmien ratkaisemiseksi on järjestetty monia eri opetuskehyksiin ja metodeihin perustuvia kursseja ja tutkimuksia yliopistoissa ja julkaistu artikkeleita näistä tutkimuksista ja tuloksista. Vaikka pää-

asiallisia opetusmetodeja on neljä, on vuosien saatossa ilmestynyt monia muita. Useat tutkimusartikkelit ovat tuoneet esille eri opetuskehysten ja –metodien etuja ja haittoja, mutta myös niiden käyttöön kohdistuvia rajoituksia, mitkä voivat ikävällä tavalla estää käyttämästä opetusmetodia tai –kehystä riippumatta huomattavista eduista. [29]

Ensimmäinen koulutusmetodi eli koulutukselliset moduulit, joissa testauksen opetusta tehdään käyttämällä teoriaa ja käytännön opetusta sisältäviä opetusyksiköitä. Aihetta käsittelevässä artikkelissa opetuksen kohteena oli ryhmä ohjelmistokehittäjiä, joilla ei ollut lainkaan kokemusta ohjelmistotestauksesta ja kurssin opetus oli jaettu teorian ja käytännön osiin. Tutkimuksen tuloksissa havaittiin testauksen opetuksessa teorian ja käytännön opetuksen yhdistämisen tärkeäksi ja opetuksessa kaivattiin lisää käytännön harjoittelua ja keskittymistä teknisiin aiheisiin. Lisäksi erityismainintana olivat opetukseen tarvittavat työkalut ja laitteistoinfrastruktuuri, joiden järjestäminen, vaikkakin välttämätöntä, vaatii paljon resursseja. [31]

Toisena koulutusmetodina koulutukselliset pelit ja kurssien pelillistäminen tähtäävät pääosin opiskelijoiden kiinnostuksen lisäämiseen valitusta aiheesta sekä halua oppia aktiivisesti ja itsenäisesti, sillä monet syyt muun muassa käytännön ja teorian opetuksen eriäväisyys latistaa intoa oppia. Vaikka pelejä testauksen opetukseen on julkaistu useita, opettavat ne monesti vain toiminnallisen testauksen konsepteja, eivätkä ole yhteneväisiä opetusmetodiikan kanssa muilla kursseilla. [29] Ensimmäisessä aiheesta kertovassa artikkelissa tutkittiin kilpailullisen korttipelin käyttöä, mikä muun muassa joukkuepelinmahdollisuuden takia sisälsi etuja ryhmäopiskelun muodossa ja syvensi opiskelijoiden tietämystä. Vaikka tutkimuksen tulokset olivat lupaavia, vaativat pelin tulevat integroimiset vielä päivittämistä. [33] Toisen artikkelin tutkimuksessa hyödynnettiin *Code Defender*-nimistä testauspeliä, missä opiskelijat yrittivät vastakkain kehittää ja testata koodia saaden pisteitä. Vaikka tuloksissa huomattiin pelin tarvitsevan joitakin päivityksiä esimerkiksi tietokantojen käyttöä testauksessa, pelin kilpailullisuuden vuoksi opiskelijoiden halu parantaa itseään ja tuotetun koodin määrä kasvoi. [34]

Vaikka testausjohtoinen kehitys eli TDD ei olekaan testaus- vaan kehitysmetodi, sen hyödyntäminen testauksen opetuksessa on havaittu parantavan ohjelmiston luotettavuutta lukuisilla kehityksen aikana luoduilla yksikkötesteillä. TDD:n opetuskäyttöä koskevissa artikkeleissa havaittiin, että kehitysmetodin opetuskäyttöön soveltaminen on erittäin

haastavaa, vaatii kohtalaisia kehitystaitoja ja lisää huomattavasti opiskelijoihin kohdistuvaa työmäärää. Ensimmäisessä artikkelin tutkimuksen tuloksissa todettiin kehitysmenetelmän parantavan koodin laatua mutta opiskelijoiden mukaan työmäärä oli huomattavan suuri, mikä osin johtui heidän kokemattomuudestaan hyödyntää TDD-metodia. Tämän takia artikkelin yhteenvedossa pohdittiin TDD:n soveltamista aikaisemmin sekä automatisoinnin ja työkalujen tärkeyttä, jotta kehitysmetodin käyttöön ehtii tottua. [35] Toisessa TDD-metodia hyödyntävän tutkimuksen artikkelissa hyödynnettiin kyseisen opetusmetodin lisäksi Web-CAT-systeemiä, jota hyödynnettiin kurssissa opiskelijoiden koodin arvosteluun ja kommentointiin. Tutkimuksen tuloksissa havaittiin TDD:n käytön muun muassa laskeneen löytyneiden virheiden määrää opiskelijoiden koodissa melkein puolella ja TDD-aktiviteettien huomattiin parantaneen opiskelijoiden käsityskykyä ja analyyttisia taitoja ohjelmointitehtävien edetessä. Vaikka TTD:n vaikutukset todettiin artikkeleissa positiivisiksi, vaatii opetusmetodin hyödyntäminen kokonaisuutena hyvin tuetut systeemit ja työkalut. [36]

Neljäntenä pääasiallisena ohjelmistotestauksen opetusmetodina on ohjelmistokehityksen ja –testauksen yhdistetty opetus, jolla on haluttu laajentaa opiskelijoiden ymmärrystä testauksesta niin kehittäjän kuin testaajan näkökulmasta. Opetusmetodista kertovan ensimmäisen artikkelin tutkimuksen kurssissa opiskelijoiden tehtävänä oli kehittää oma sovellus kehittäjän roolissa, jonka jälkeen heidän tuli testata toisen opiskelijan sovellus testaajan roolissa. Opiskelijoiden vaihtelevien töiden ja erilaisten tehtävänantojen takia töitä arvioitiin laadittujen dokumenttien muun muassa testaussuunnitelman laadun mukaan eikä koodatun sovelluksen mukaan. Vaikka artikkelissa mainittiin kurssilla olleen suuri onnistumisprosentti viiden vuoden ajan ja sen avanneen opiskelijoilla uusia näkökulmia testauksen osalta, todettiin kurssin järjestäminen haastavaksi johtuen opiskelijoille jaettavista useista erilaisista projekteista ja työmäärän vaihtelevuudesta. [41]

Muissa ohjelmistokehityksen ja –testauksen opetuksen yhdistämistä tutkivissa artikkeleissa esiteltiin erilaisten systeemien käyttöä kyseisessä metodissa. Näitä olivat kehitteillä oleva verkkopohjainen ProgTest-systeemi testausaktiviteetteihin perustuvien koodaustehtävien palautukseen sekä automaattiseen arviointiin. Artikkelin tutkimuksessa arvioitiin systeemin auttavan eritoten palautteen nopeaan antamiseen ja töiden arviointiin, mitkä taas olivat artikkelin mukaan tämän opetusmetodin kriittisimmät tekijät kurssin onnistumisen kannalta. [39] Toisessa artikkelissa esitellyn SEP-CyLE-järjestelmän tarkoituksena on taas

tarjota opiskelijoille verkkomateriaalia tehtävien sekä testaustyökaluohjeistusten muodossa. Vaikka opiskelijoiden työmäärä lisääntyi, heidän arvosanansa paranivat verrattuna systeemiä käyttämättömiin. Lisäksi SEP-CyLE-järjestelmän käyttö ja lisäopetuksen huomattiin parantavan ohjelmoinnin perusteita ja inspiroimaan opiskelijoita oppimaan enemmän ohjelmistotestauskursseilla. [43]

Muita artikkeleissa mainittuja opetuskehyksiä ovat pienempien testausprojektien järjestämiseen perustuva CDIO, oikeiden testausprojektien suoritus kurssilla yritysten ja yliopistojen yhteistyönä sekä JiTT-metodi, jossa opetus tapahtuu järjestämällä luennot verkossa tehtävien ennakkoharjoitusten ja lukumateriaalin perusteella. Vaikka CDIO ja eritoten oikea testausprojekti tarjoavat artikkeleiden tulosten perusteella paljon käytännön kokemusta ja taitoja testauksesta, puhumattakaan vaikutuksista töiden saatiin ja erikoistumiseen, kohdataan kurssien järjestämisessä huomattavia vaikeuksia. CDIO:ssa ongelmia tuottavat kurssin järjestämisestä aiheutuva työmäärä, projektitöiden arviointi ja arvostelu sekä testaustyökalujen ja –ympäristön järjestäminen kurssille. [44] Oikeiden testausprojektien järjestämisessä yrityksillä ongelmana ovat vaitiolovelvollisuudet yrityssalaisuuksissa sekä käyttävätkö yritykset tosissaan opiskelijoita isompaan projektiin tai vain yksinkertaisen ja toistettavien testaustoimenpiteiden tekoon. [46] JiTT-menetelmässä ongelmana taas on kurssin järjestäjiin kohdistuva suuri työmäärä, kun luentoja järjestetään verkkotehtävistä saatujen tulosten perusteella. [45]

3 TUTKIMUS

Tutkimuksen suorittamiseksi laadittiin tutkimusprosessi, joka luotiin muodostamalla ja valitsemalla johtavat tutkimuskysymykset, työskentelymetodit sekä lähde- ja aineistopaketti. Lisäksi kartoitettiin ja kerättiin tutkimusta ja loppuratkaisua hidastavat ja rajoittavat vaatimukset ja rajoitteet. Tutkimusprosessin ja tutkimuksen tavoitteena oli selvittää, onko mahdollista löytää ja hyödyntää LUT-yliopiston ohjelmistotestausta sisältävän kurssin opetuksessa sopivaa testaustyökalupakettia tai –kokonaisuutta ja millä tavoin tämä voidaan suorittaa. Kohdekurssi, jolle työkalut oli tarkoitus valita ja kurssitehtävät laatia, oli LUT-yliopiston järjestämä kurssi *CT60A0220 C-ohjelmoinnin ja testauksen periaatteet*.

3.1 Tutkimuksen tavoitteet ja tutkimuskysymykset

Tutkimuksen tavoitteen ja suunnan muodostavista tutkimuskysymyksistä ensimmäisenä tutkimuskysymyksenä oli selvittää, voidaanko ohjelmistotestauksen kurssilla hyödyntää kaupallisia ja/tai avoimen lähdekoodin testaustyökaluja ja millä tavoin? Tästä tarkentavina kysymyksinä oli, miten ohjelmistotestauksen opetusta voi kehittää näitä työkaluja hyödyntämällä ja millaista kurssimateriaalia, esimerkiksi kurssi- ja harjoitustyöideoita, voidaan laatia näiden työkalujen ja muiden vaikuttavien rajoitteiden, esimerkiksi kurssin tavoitteiden ja esivaatimuksien, pohjalta. Onko mahdollista luoda työkaluilla ohjelmistotestaustehtäviä IT-yritysten työelämään liittyen? Viimeisinä tarkentavina kysymyksinä olivat, minkälaiset olivat kurssin lopputulokset, onnistuivatko kurssin työkalu- ja tehtävävalinnat ja mikäli onnistui tai ei onnistunut niin miksi. Lopuksi haluttiin vielä selvittää, miten LUT-yliopiston ohjelmistotestauksen opetusta ja kursseja voidaan kehittää saatujen tulosten perusteella.

3.2 Lähteiden ja aineiston valinta

Tutkimusprosessin muodostamiseksi kerättiin testauksesta ja testauksen opetuksesta runsaasti aineistoa, joista testauksesta oli pääasiassa LUT-yliopiston kirjaston ja verkosta saavat kirjalliset julkaisut. Ohjelmistotestauksesta kertovasta aineistosta pohjana toimi tutkimuksen toisena ohjaajana ja kurssin opetusassistenttina toimineen Timo Hynnisen tarjoama ja ohjelmistotestauksen opetuksesta kertova tutkimusartikkelipaketti, jonka lisäksi lähteiksi oli valittu muita verkossa, pääosin ACM:än eli *Association of Computing Machinery*-, *Springer Link*– ja *IEEE Xplore* -sivustojen digitaalisissa kirjastoissa julkaistuja tutkimusartikkeleita.

3.3 Prosessien muodostuminen

Mikäli halutaan luoda tehokas ja aikaansaava testausprosessi, tarvitaan siihen komponentiksi oikean testausympäristön perustaminen. Pääasiallisesti hyvä testausympäristö koostuu ohjelmistotestaustyöprosesseista, -työkaluista ja pätevästä testaushenkilöstöstä. [6, s.60] Perustuen tähän, päätettiin testauksen opetusprosessi ja sen avainkomponenttina oleva testauksen opetusympäristö järjestää samalla tavalla, jossa testaustyöprosesseina olivat kurssi ja siinä ohjelmistotestauksen opetusmetodina käytetty ohjelmoinnin ja testauksen yhdistettyä opetus. Työkaluina toimisivat opetuksessa käytettävät testaustyökalut ja testausympäristön ”testaushenkilöstönä” toimisivat LUT-yliopiston opetushenkilöstö ja kurssille osallistuvat opiskelijat. Testaustyökalujen valitsemiseksi päätettiin keskittyä kurssin sisältöön ja testauksen V-mallin eri tasojen työkaluihin. Työkalujen ja tehtävien valinta- ja laadintaprosessien tarkoituksena oli saada aikaan toimiva kokonaisuus, jotta testauksen opetus suoriutuisi tehokkaasti.

3.3.1 Rajoitteet

Ennen työkalujen valintaa ja tehtävien laatimista piti ottaa huomioon suurimmat ja yleisimmät niihin vaikuttavat rajoitteet ja vaatimukset, jonka takia laadittiin lista työkaluja koskevista vaatimuksista sekä käyttötapaukset työkalujen käytöstä. Näistä laadittu liitepaketti sisälsi kaikki työkaluja koskevat rajoitteet, joista monet liittyivät kurssiin ja kurssin opetusmateriaalin sisältöön. Muita huomattavia rajoitteita työkaluja valittaessa olivat kurssin puitteiden lisäksi resurssit työkalujen hankinnassa ja käyttöönotossa. Kurssitehtäviä koskevat rajoitteet liittyivät edellä mainittujen lisäksi opiskelijoiden tieto- ja taitotasoon perustuen aiempiin kursseihin sekä heihin ja opetushenkilöstöön kohdistuvan työmäärään. Työkalujen ja erityisesti kurssitehtävien valinnan ja laadinnan aikana tuli esille uusia rajoitteita, jonka takia monia kurssitehtäväideoita jouduttiin sulkemaan pois ottaen huomioon myös niiden suorittamisesta aiheutuvan työmäärän. Lisäksi koska kurssin rakenne, sisältö ja opetusmetodi eli ohjelmoinnin ja testauksen yhdistetty opetus olivat jo ennalta määrättyjä, ei esimerkiksi testausjohtoiseen kehitykseen eli TDD:hen perustuvaa opetusta ja tehtäviä voitu järjestää.

3.3.2 Työkalujen valinta ja ideat

Työkalujen valinnassa pohdittiin aluksi vaihtoehtoa sopimusyhteistyöstä ohjelmistotestaustyökaluja kehittävien tai niitä käyttävien IT-yritysten kanssa. Vaikka yritysten käyttämien testaustyökalujen käyttö yliopistossa toisi työkalujen kehittäjille näkyvyyttä, tulisi se vaatimaan huomattavasti enemmän aikaa ja järjestelyä osapuolten välillä. Myös teetetyn työkalusovellusratkaisun kehittäminen opetuskäyttöön yliopistolle vaatisi huomattavasti resursseja ajallisesti ja rahallisesti. Tästä syystä päätettiin soveltaa COTS-ratkaisun tekemistä, jossa tarvittava testaustyökalupaketti rakennetaan yhdistelemällä valmiita kaupallisia ja/tai avoimen lähdekoodi työkaluja.

Työkalujen valitsemiseen pohdittiin aluksi useamman työkalun testaamista jokaisesta V-mallin testaustasosta vaatimusten ja käyttötapausten mukaan, eli testauksenhallinta-, yksikkötestaus-, integraatiotestaus-, järjestelmätestaustyökalut, joiden lisäksi muita kurssilla opetettavia työkaluja muun muassa automaatiotestaustyökaluja. Tällöin työkaluja voitaisiin arvioida ja vertailla paremmin niiden ominaisuuksien perusteella ja laatia tarkka ja kattava yhteenveto työkalujen eroista ja niiden sopivuudesta opetuskäyttöön. Kuitenkin koska tarkoituksena on rakentaa useamman työkalun yhdistelmä, ei COTS-ohjelmilla ole aina takuita niiden yhteensopivuudesta muiden työkalujen kanssa huolimatta niiden ominaisuuksista. [6, s.689]

Ottaen huomioon kattavaan testaukseen tarvittavat resurssit, ei suuren työkalumäärän testaamisen ollut tarpeeksi aikaa puhumattakaan rahallisista resursseista. Vaikka kaupalliset työkalut sisältävät kokeiluversioita, eivät ne aina anna kaikkia ominaisuuksia käytettäväksi, jotta työkalun testaus oli kattavaa. Edellä mainittujen seikkojen takia työkalukokonaisuus päätettiin rakentaa vähitellen testauksenhallintatyökalusta aloittaen ja edeten V-mallin mukaisesti yksikkötestauksesta järjestelmätestaukseen ja automaatioon. Työkalujen testaus suoritettiin käymällä läpi työkalua käyttäen sille laaditut vaatimukset ja käyttämällä virtuaalikonetta, joka hyödynsi samaa LUT-yliopiston harjoitusluokassa hyödynnettävää Linux-pohjaista Ubuntu-käyttöjärjestelmää. Testauksenhallintatyökalun testauksessa käytettiin Linux-pohjaista CentOS- eli ***C****ommunity* ***Ent****erprise* ***O****perating* ***S****ystem*-käyttöjärjestelmää.

Muita työkalujen käyttöönottoa rajoittavia mainittavia tekijöitä olivat opiskelijoiden tieto- ja taitotasot, sillä muun muassa monet testaustyökaluista olivat Java-pohjaisia ja kurssin osanottajat olivat pääosin aloittelevia C-kielen opiskelijoita. Lisäksi tiettyjä työkaluja esimerkiksi Seleniumia käytettäessä käyttöliittymätestaukseen, on testaajalla oltava www-sovellusten tietämystä, mitä ensimmäisen vuosikurssin opiskelijoilla ei oleteta vielä olevan.

3.3.3 Tehtävävaihtoehtojen ideointi ja arviointi

Tehtävien laatiminen ja valinta suoritettiin opetettavan kurssin sisällön puitteissa ja käymällä keskusteluja kurssin järjestäjien ja opetusvastaavien kanssa. Näihin kuuluivat myös tutkimustyön ohjaajina toimineet kurssin vastuuhenkilö ja opetusassistentti. Koska kurssin harjoitusten pitäminen oli opetusassistentin vastuulla, sovittiin keskustelut tehtävien käytännön järjestämisestä pääasiassa hänen kanssaan. Keskusteluissa sovittiin aikarajat eri harjoitustehtävien palauttamiseen opetusassistentin käyttöön ennen harjoitusten alkamista ja nämä tehtävät perustuivat harjoituksia edeltävien luentojen sisältöön. Ennen tehtävän määräaikaa laadittiin ideointi harjoitustehtäväehdotuksista, jonka jälkeen käytiin keskustelu opetusvastaavien kanssa, valittiin sopiva idea ja idea tämä työstettiin edelleen harjoitustehtäväksi. Mikäli ideoissa tai harjoitustehtävissä oli huomautettavaa muun muassa sisällön tai työn määrän kannalta, suoritettiin toimenpiteet ja muokkaukset niiden mukaan. Harjoitustyön valmistuttua tarkistettiin se vielä opetusassistentin toimesta, jonka jälkeen tehtävä otettiin käytettäväksi sille suunnitellulle harjoituspäivälle.

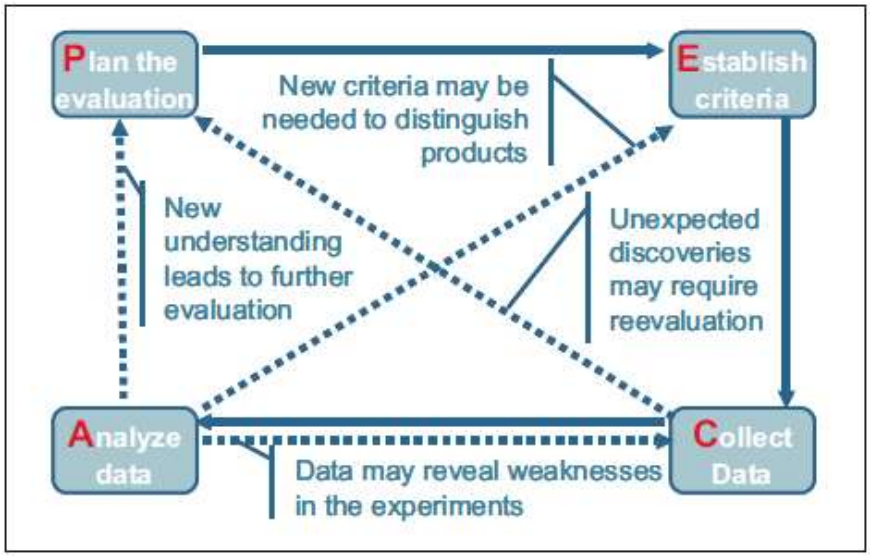
Ideoita harjoitustehtäviksi oli useita, mutta koska kurssin sisältö, materiaali ja opetusmetodi oli jo ehditty valita, ei niitä ollut enää mahdollista järjestää. Lisäksi kurssin osanottajien ollessa enimmäkseen ensimmäisen vuosikurssin opiskelijoita, ei heiltä voida vaatia sellaisten harjoitustehtävien tekemistä, joissa vaaditaan tietämystä myöhemmin tulevilta kursseilta. Vaikka esimerkiksi pienimuotoinen ohjelmistotestausprojekti olisi antanut paljon ensimakua ja käytännön tietoa testauksen kulusta, olisi tämä vaatinut heiltä tietämystä teoriapuolelta ohjelmistoprojekteista. Myös kurssin järjestäminen olisi ollut erityisen haastavaa ja lisännyt opetuksesta vastaavien työtä muun muassa projektien tarkistuksissa. Tämän takia projektiluontoiset tehtävät tuli sulkea pois vaihtoehdoista. Lisäksi TDD:n käyttö, vaikkakin tehdessä testauksesta rutiininomaista ja parantaessa koodin laatua opiskelijoilla, vaatii kohtalaisen hyviä koodaustaitoja onnistuakseen. [35]

3.4 COTS-sovellukset

Koska teetetyn testaustyökalusovellusratkaisun kehittäminen ja toteutus ei olisi ollut resurssien takia mahdollista lyhyellä aikavälillä, päätettiin tutkimuksessa käyttää työkalujen hankinnassa valmiita COTS-pohjaisia testaustyökaluja. Vaikka COTS- eli Commercial-Off-The-Shelf-sovellukset ovat valmiita muiden kehittäjien tekemiä ratkaisuja, tuo niiden käyttäminen komponentteina suuremman kokonaisratkaisun kasaamiseksi omat haasteensa. Vaikeudet johtuvat yleensä COTS-sovellusten ja –komponenttien täydellisestä ”musta laatikko”-luonteesta, sillä suuremman ratkaisun kehittäjillä ei ole pääsyä katsomaan ja muuttamaan lähdekoodia ja COTS-sovellukset on usein tarkoitettu suorittaa itsenäisesti. [47]

3.4.1 Yleiskatsaus ja sovellusten arviointi

COTS-sovelluksia hyödyntäessä käyttöön otetaan valmiisi kehitetty sovellus, jolloin ei tarvitse käyttää resursseja saman tyyppisen sovelluksen kehittämiseen alusta asti. COTS-pohjaisten ohjelmien hyödyntäminen on saavuttanut suosiota pitkin IT-teollisuutta ja erityisesti pienemmät ja keskisuuret organisaatiot ovat yhä enemmän riippuvaisia COTS-komponenteista. [48] Kasattaessa suurempaa kokonaisuutta COTS-komponenteista on välttämättömänä toimenpiteenä arvioida sovelluksen soveltuvuutta ja käyttöä isommassa kohdejärjestelmässä. Tästä huolimatta arviointiprosessissa tehdään useita virheitä, joita ovat riittämättömät ponnistelut, uudempien sovellusversiojulkaisujen uudelleenarvioinnin laiminlyönti, ”parhaimmasta päästä”-listan käyttö, joka ei vastaa systeemin tarpeita ja kuvausta, riittämätön osakkaiden osallisuus ja puuttuva käytännön kokeilu. [49]



Kuva 13. PECA-prosessi. [49]

Jotta COTS-sovelluksen soveltuvuuden arviointi olisi mahdollisimman selkeä ja korkealaatuinen, on tähän ratkaisuksi ehdotettu S. Cornella-Dornan, J. C. Deanin, E. Morrisin ja P. Oberndorfin artikkelissa PECA-prosessia. Prosessin nimi muodostuu prosessin vaiheista: ***P****lanning the evaluation* eli arvioinnin suunnittelu, ***E****stablishing the criteria* eli kriteerein luominen, ***C****ollecting the data* eli datan kerääminen ja ***A****nalyzing the data* eli datan analysointi. Vaikka PECA alkaa COTS-sovellusten arvioinnista ja päättyy suositusten antamisiin päätöksenteossa, se on prosessina erittäin ketterä ja suuntaa antava. Kuvasta 13 voi huomata, ettei prosessi etene aina peräkkäin, vaan monet eri arviointitapahtumat kuten yllättävät kriteerien muutokset ja havainnot voivat aloittaa esimerkiksi uuden iteraation. Tämän takia PECA on tarkoitettu muokattavaksi organisaatioiden käyttöön ja tarpeisiin. Lisäksi erilaisia tekniikoita voi hyödyntää prosessin eri vaiheissa, esimerkiksi ketterää GQM eli *Goal-Question-Metric*-tekniikkaa voi käyttää luodessa COTS arviointikriteereitä. [49]

3.4.2 Hyödyt ja haitat verrattuna teetettyyn sovellusratkaisuun

Kaksi suurinta eroa COTS-sovelluksen ja teetetyn sovelluksen välillä ovat vaatimusten laatimisessa ja mahdollisuudesta vaikuttaa sovelluksen kehitykseen ja lopulliseen versioon. Kun COTS-kokonaisuutta suunnitellessa kehittäjä laatii komponenttien vaatimukset, laatii teetetyssä sovelluksessa vaatimukset ohjelmistoa tarvitseva yritys, joka pystyy vaikuttamaan kehitykseen testausvaiheessa. COTS-ratkaisua etsivä ei taas pysty vaikuttamaan löytämiinsä sovelluksiin millään tavalla. COTS-komponenttien ja –sovellusten hyödyntäminen tuo mukanaan paljon hyötyjä ja haittoja. COTS-sovelluksen käyttöönotolla vältetään teetetyn ratkaisun tuomat kehityskustannukset, vähennetään kehityksessä ilmeneviä riskejä ja voidaan ottaa sovellus käyttöön teetettyä ratkaisua nopeammin. Lisäksi, riippuen sovelluksen käyttökohteista, on COTS-ratkaisuja tarjolla suurempi valikoima, niiden laatu on korkeampi ja pysyäkseen kilpailukykyisinä, pyrkivät COTS-sovellusten kehittäjät hyödyntämään aina uudempaa teknologiaa. Vaikka COTS-ratkaisujen integroiminen on suuri haaste, voi teetetyillä sovellusratkaisuilla olla vielä suurempia integraatiohaasteita, kun niiden käyttöliittymät eivät välttämättä ole standardisoituja. [6, s. 686-687]

Haittoja COTS-sovelluksilla on oikean kokonaisuuden valitseminen ja kokoaminen, sekä näiden komponenttien löytäminen suuren tarjonnan joukosta. Erityisen suurta haittaa COTS-ratkaisujen käyttöönotossa on mahdottomuus vaikuttaa niiden kehityssuuntaan ja laatuun sekä saada täysin vaatimuksia vastaava ratkaisu. Lisäksi on erittäin vaikea tietää

käytössä olevien tuotteiden tulevaisuudesta, sillä COTS-sovelluksien tuki ja päivitys voivat päättyä kehittäjän taholta tai sovelluksiin voi tulla huomattaviakin muutoksia. Nämä muutokset tuovat mukanaan monia haasteita, mitä voivat olla tuki- ja lisenssioikeuksien muutokset, integrointi vanhempien sovellusten kanssa ja tulevien päivitysten jatkuva testaaminen. [6, s. 687-688] Lisäksi kun testaamista joudutaan yleensä suorittamaan musta laatikko-periaatteella, eivät sovelluksen testaajat ja hankkijat pääse näkemään sovelluksen sisäistä perspektiiviä. [47]

3.4.3 Käyttöönoton riskit ja testaushaasteet

Kun COTS-sovellusta otetaan käyttöön, voi tähän liittyä monia riskejä, sillä jotkin sovelluksen toiminnot eivät välttämättä toimi ollenkaan tai sille halutulla tavalla ja riippuen sovelluksesta niiden tietoturva voi olla puutteellinen. Muut riskit COTS-sovellusten käyttöönotossa liittyvät yhteensopivuudessa haluttuun ympäristöön, sovelluksen tuen päättymisessä esimerkiksi kehittäjän mentyä konkurssiin ja lisenssisopimuksen solmimisessa. Lisäksi monissa COTS-pohjaisissa projekteissa saatetaan usein olettaa virheellisesti, että sovellus on testattu jo kehittäjän puolelta, jolloin sovellusta ei tarvitse testata kuin minimaalisesti, mikä taas voi johtaa suurin ongelmiin käyttöönoton jälkeen. Näiden kaikkien riskien takia olisi suotavaa, että kohteena oleva COTS-sovellus sisältää palauteosion, jonne on mahdollista raportoida käyttäjäkohtaisia ongelmia. Tällöin kehittäjät voisivat puuttua ongelman korjaamiseen ja ilmoitta tästä muille käyttäjille. [6, s.688-689]

Haasteet COTS-sovellusten testaamisessa liittyvät usein pakonomaisuuteen käyttää ulkoista mustan laatikon testausta, sillä ilmaisia ja avoimen lähdekoodin ohjelmia lukuun ottamatta COTS-sovellusten sisäistä rakennetta ei tunneta eikä sitä päästä tarkastelemaan. [47] Muita haasteita testauksessa ovat COTS-komponenttien integraation varmistaminen, komponenttien uudempien versioiden ilmestyminen, jatkuva regressiotestaus kokonaisuuden toiminnan varmistamiseksi ja teknologian vaihtuminen. Jatkuvan sovellusten ja teknologian kehityksen takia COTS-sovellusten testauksen automatisointia on toivottu, mutta tähän on hankala päästä juuri samoista syistä johtuen. Se aikajana, jolloin testejä ei ole mahdollista automatisoida, on juuri se aika, jolloin testejä haluttaisiin automatisoitavan. [6, s. 689-690]

4 TUTKIMUKSEN TULOKSET

4.1 Kohdekurssi

Tutkimuksen kohdekurssina oleva LUT-yliopiston järjestämä kurssi *CT60A0220 C-ohjelmoinnin ja testauksen periaatteet* jakautui kevätlukukauden kolmannelle ja neljännelle periodille sekä oli laajuudeltaan 6 opintopisteen kurssi. LUT-yliopiston opinto-oppaan mukaan kurssin tavoitteena oli opettaa C-ohjelmoinnin ja testauksen periaatteet, joiden lisäksi opiskelija osaa kurssin jälkeen tehdä C-kielisiä ohjelmia ryhmän jäsenenä hyvää ohjelmointitapaa noudattaen ja jakaa ohjelman toiminnallisiin kokonaisuuksiin. Testauksen osalta opiskelija tuntee kurssin jälkeen tavallisimmat ohjelmistotestauksen työmenetelmät sekä testauksen työvaiheet ja heillä on valmiudet tehdä ohjattua testaustyötä itsenäisesti, tai suunnitella ja valmistella testaustyötä osaksi organisaatiota. Lisäksi opiskelija tulee tietämään miten ohjelmistotestausta tehdään ja kuinka testaustoiminta ja ohjelmistokehitys liittyvät toisiinsa. Taulukossa 2 näkyy opinto-oppaan ilmoittama kurssin sisältö ja suoritustavat. [50]

Taulukko 2. Kurssin CT60A0220 sisältö ja suoritustavat. [59]

Sisältö	
C-ohjelmointikielen kielioppi ja rakenteet, erityisesti tietotyypit, osoittimet, dynaaminen muistinhallinta ja rekursio sekä käytännön C-ohjelmoinnin periaatteet	
Ohjelmoinnin perustyökalut kuten editori, kääntäjä, virheenjäljittimet ja versionhallinta.	
Ohjelmistotestauksen työkalut ja tavallisimmat dokumentit, yksikkötestaus, integrointitestaus, järjestelmätestaus.	
Testaus käytännössä	
Suoritustavat	
3. periodi	4. periodi
Luentoja 14 h	Luentoja 14 h
Omatoiminen opiskelu 7 h	Omatoiminen opiskelu 7 h
Pakollisten harjoitustehtävien ja projektien teko 50 h	Pakollisten harjoitustehtävien ja projektien teko 54 h
Tenttiin valmistautuminen 7 h ja tentti 3 h. Kokonaiskuormitus 156 h.	

Kurssille arvioitiin osallistuvan noin 150 opiskelijaa ja kurssin esitietovaatimuksena olivat kurssi CT60A0200 Ohjelmoinnin perusteet tai sitä vastaava tiedot. Kurssin työmäärässä ja suoritustavoissa painotettiin harjoitustehtävien ja projektien tekoja, joiden lisäksi kurssilla järjestettiin luentoja ja opiskelijoiden vastuulla oli omatoimista opiskelua. Kurssin arvioinnissa arvosanasta puolet oli tentistä ja toinen puoli pakollisista viikkotehtävistä ja harjoitustyöstä. Lisäksi kurssin pääasiallisena oppimateriaalina oli Jussi Kasurisen teos *C-kieli ja käytännön ohjelmointi osa 1 Ohjelmistotestauksen käsikirja*. [50]

4.2 Tutkimuksen työkalu- ja tehtäväratkaisut

Tutkimuksessa työkalut valittiin perustuen testauksen V-mallin tasoihin ja kurssin testausosion luentojen sisältöön. Perustuen aiemmin mainittuihin rajoitteisiin ja laadittuihin vaatimuksiin, päädyttiin työkalujen valinnoissa seuraaviin työkaluvalintoihin, joissa suosittiin resurssien säästämiseksi ilmaisia ja avoimen lähdekoodin ohjelmia. Vaikka suoraa yhteistyötä yritysten kanssa tai pienempää ohjelmistotestausprojektia ei pystytty järjestämään johtuen rajoitteista, haluttiin opiskelijoille antaa ensimakua IT-yrityksien testauskäytännöistä opettamalla testauksenhallintatyökalun käyttöä muiden V-mallin testaustyökalujen lisäksi. Työkaluvaihtoehtoja ja niiden muodostamaa kokonaisratkaisua lähdettiin rakentamaan aloittamalla testauksenhallintatyökaluista ja edeten V-mallin mukaisesti yksikkötestauksesta järjestelmätason testaustyökaluihin.

4.2.1 Testauksen johtaminen: Tarantula

Ensimmäisenä työkalukohteena oli testauksenhallintatyökalun valinta, jossa vaatimusten ja rajoitteiden pohjalta käyttöön valittiin suomalaisen Prove Expertisen kehittämä ketterään testauksenhallintaan pohjautuva ja ilmainen avoimen lähdekoodin omaava Tarantula. Tarantulan toimintoina ovat testitapausten, -vaatimusten ja -jonojen luominen sekä testitapausten suorittamisen ja näiden kaikkien toimintojen hallinnointi. Lisäksi työkaluun on mahdollista integroida vaatimusten ja virheiden jäljittämiseksi Jira-, Bugzilla- ja Doors-työkaluja. [51] Vaikka Tarantulan tukeminen kehittäjän taholta on päättynyt, johtuen maksavien asiakkaiden puutteesta, on työkalu nyt ilmainen avoimen lähdekoodin työkalu ja sen voi ottaa käyttöön lataamalla sen GitHubin verkkosivulta ja asentamalla palvelimen omalle koneelle. [52] Vaikka kehittäjä ei tarjoa työkalulle enää tukipalveluita, voi apua ja neuvoja hakea yhteisön ylläpitämältä verkkosivulta ja foorumilta. [53]

Tarantulan käyttö aloitetaan asentamalla ja luomalla työkalulle oma palvelin, luomalla kohdekäyttäjille tilit ylläpitäjän roolissa ja kirjautumalla luoduilla tunnuksilla nettiselaimen kautta palveluun. Luoduille tunnuksille on mahdollista jakaa eri testausprojektien rooleja, jotka vaikuttavat työkalun toimintojen käyttöoikeuksiin, jolloin voidaan tehdä jako opiskelijoiden, opetusvastaavien ja palvelimen ylläpitäjien välillä. Lisäksi uuden käyttäjät saavat ilmoituksen heille luodusta tilistä sähköposti kautta, sillä luodessa uusia tilejä niille tulee lisätä perustietojen lisäksi sähköpostiosoitteet. Mikäli halutaan luoda ryhmäjakoa opiskelijoiden välillä, voidaan tätä varten luoda työkalulla ryhmää kohti oma projekti ja asettaa opiskelijoiden oikeudet tietyille projekteille. Projektien etenemisen ja käyttäjien edistymisen opetusvastaavat voivat tarkistaa raportointinäkymästä. [51]

4.2.2 Yksikkö- ja integraatiotestaus: CUnit

Toinen opetukseen valituista testaustyökaluista rajoitteiden ja vaatimusten jälkeen oli yksikkö- ja integraatiotestauksen käsittävä Cunit, jolla voidaan kirjoittaa, hallita ja suorittaa yksikkötestejä C-ohjelmointikielellä. Cunit on ilmainen ja se käyttää yksinkertaista kehysrakennetta testirakenteiden luomiseen sekä tarjoaa laajan valikoiman testiväitekomentoja yleisten datatyyppien testaamiseen. Lisäksi työkalu tarjoaa useita erilaisia käyttöliittymiä testien suorittamiseen ja tuloksien raportointiin. Näitä käyttöliittymiä ovat automatisoidut käyttöliittymät koodihallittuun testaamiseen ja raportointiin sekä interaktiiviset käyttöliittymät antaen käyttäjille mahdollisuudet suorittaa ja katsella testauksen tuloksia dynaamisesti. Rakenteeltaan Cunit on yhdistelmä alustaitsenäistä kehystä lukuisilla käyttöliittymillä, jotka määrittelevät interaktion kehyksen kanssa testien suorittamiseksi ja tulosten tarkistamiseksi. [54]

```
                    Test Registry
                          |
          ------------------------------
          |                            |
       Suite '1'      . . . .       Suite 'N'
          |                            |
    ---------------             ---------------
    |             |             |             |
 Test '11' ... Test '1M'     Test 'N1' ... Test 'NM'
```

Kuva 14. Cunit-yksikkötestauksen kehyksen rakenne. http://cunit.sourceforge.net/doc/introduction.html [54]

Rakenteeltaan Cunit on organisoitu kuvan 14 mukaisesti, jossa yksitäiset testit on pakattu testisarjoihin, jotka taas rekisteröidään aktiivisella testirekisterillä. Testisarjoilla on omat järjestely- ja purkamistoiminnot, jotka kutsutaan automaattisesti ennen ja jälkeen jonon testien suorittamisen. Lisäksi yksittäiset testit ja sarjat rekisterissä voidaan suorittaa käyttämällä yhtä funktiokomentoa tai sitten voidaan suorittaa kaikki testit ja sarjat kerralla. Jotta Cunitin komennot eroaisivat C-kielen komennoista, käytetään työkalussa ”CU_”-alkuisia komentoja. Tyypillisesti Cunit-kehyksen käyttö etenee taulukon 1 mukaisesti ja työkalun käyttöliittymänä on komentorivi. Lisäksi Cunit on yhteensopiva Ubuntu-käyttöjärjestelmän kanssa, ja työkalulla on foorumi ongelmatilanteista kysymiseen. [54]

Taulukko 3. Tyypillinen CUnit-kehyksen käytön vaiheiden järjestys. [60]

1. Kirjoitetaan testeille funktiot (ja jonon aloitus- ja puhdistusfunktiot testisarjoille jos tarpeellista.)
2. Käynnistetään testirekisteri – CU_initialize_registry()
3. Lisätään sarjat testirekisteriin – CU_add_suite()
4. Lisätään testit testisarjoihin – CU_add_test()
5. Suoritetaan testit käyttäen haluttua käyttöliittymää, esimerkiksi – CU_console_run_tests
6. Tyhjennetään testirekisteri – CU_cleanup_registry

Koska yksikkö ja integraatiotestit eroavat toisistaan vain siinä suhteessa, että integraatiotesteissä testataan useamman yksikön funktioiden yhteistoimintaa, voidaan tämä suorittaa myös Cunitilla kääntämällä testattavan funktion lisäksi siinä hyödynnettävät ulkoisten tiedostojen funktiot ja käyttämällä tiedostoviitteitä. [54] Vaikka Cunitia ei pystytä varsinaisesti integroimaan testauksenhallintatyökalu Tarantulaan, voidaan niiden käyttö yhdistää hyödyntämällä Cunitin ominaisuutta tulostaa suoritettujen testien tulokset XML-tiedostoon automatisoidussa käyttöliittymässä ja Tarantulan mahdollisuutta tallentaa, sillä luotuihin testitapauksiin tiedostoja. Esimerkiksi suunnitellaan ja luodaan aluksi testitapaus, jonka pohjalta koodataan sitä vastaava yksikkötesti. Tämä yksikkötesti suoritetaan ja siitä saadut tulokset tallennetaan XML-tiedostoon, joka taas tallennetaan alkuperäiseen testitapaukseen Tarantulassa. Lopuksi suoritetaan Tarantulassa kyseisen testitapauksen testisuoritus ja merkitään XML-tiedostossa olevat tulosten pohjalta testitapauksen onnistuminen. Näin voidaan pitää kunnolla kirjaa testitapausten suorituksista ja onnistumista niin opiskelijoiden kuin opetusvastaavien taholta.

4.2.3 Järjestelmätestaus: Valgrind

Alkuun työkaluksi järjestelmätestaukseen pohdittiin Seleniumia, sillä isomman järjestelmän esimerkiksi nettisuvun testaus, varsinkin työkalun tarjoaman käyttöliittymä- ja automaatiotestauksen kautta, olisi mielenkiintoinen aihe ja kohde olisi tarpeesi iso. Seleniumin käyttö vaatii kuitenkin kohtalaista WWW-sovellusten rakenteen tietämystä, jonka takia tämä työkaluvaihtoehto jouduttiin sulkemaan pois. Lisäksi koska järjestelmätestausta voi tehdä myös muilla tavoin, kurssissa käsitellään linkitettyjä listoja ja muistivuodot ovat usein C- ja C++-kielisissä ohjelmissa ongelmana, päädyttiin näiden syiden, vaatimusten ja rajoitteiden takia Valgrind-testaustyökaluun.

Valgrind on instrumentaatiokehys dynaamisten analyysityökalujen kehittämiseen ja se sisältää jo ennestään joukon työkaluja. Näillä työkaluilla voidaan tehdä tietynlaista profilointia, debuggausta tai saman tyyppisiä toimintoja, joilla voi parantaa kohdeohjelman suorituskykyä. Valgrindin arkkitehtuuri on modulaarinen, joten uusia työkaluja on helppo tehdä vaikuttamatta olemassa olevaan työkalukehykseen. [55] Valgrindissa olevista työkaluista todennäköisemmin opetuskäytössä, perustuen kurssin sisältöön, on muistivirheitä havaitseva Memcheck. Memcheck-työkalu helpottaa havaitsemaan taulukossa 2 lueteltuja ongelmia, jotka ovat yleisiä C- ja C++-kielen ohjelmissa. Kyseisiä ongelmat, joita on hankala havaita muilla tavoin, voivat olla huomaamatta pitkiäkin aikoja ja aiheuttaa satunnaisia vaikeasti diagnosoitavia ohjelman kaatumisia. [56]

Taulukko 4. Ongelmat, jotka voidaan havaita Valgrindin Memcheck-työkalulla. [61]

1. Pääsy muistiin, johon ei pitäisi olla oikeuksia esimerkiksi kirjoittamalla yli muistiosioita.
2. Käytetään määrittämättömiä arvoja
3. Väärä muistin vapautus esimerkiksi muistiosion vapauttaminen kahdesti.
4. Päällekkäiset src- ja dst-pointterit memcpy- ja muissa siihen liittyvissä funktioissa
5. Epäilyttävän (olettaen negatiivisen) arvon lähettäminen size-parametrille muistinvaraus-funktiossa
6. Muistivuodot eli unohdetaan vapauttaa varattu muisti käytön jälkeen funktioissa.

Muita Valgrindin sisältämiä työkaluja on ovat Cachegrind, Callgrind, Helgrind, DRD, Massif, DHAT, SGcheck ja BBV, joilla voidaan muun muassa suorittaa välimuistin ja haarautumisen ennustamisen profilointi, mikä taas helpottaa ohjelmia toimimaan nopeammin. [55] Vaikka Valgrindin Memcheck-työkalua ei ehditty soveltaa kunnolla laadituissa harjoitustehtävissä, on se silti erittäin hyödyllinen C- ja C++-pohjaisten ohjelmien testaukseen johtuen aiemmin mainituista muistinkäyttöongelmista. Työkalulla voidaan näyttää opiskelijoille tarkkaavaisuuden tärkeys kirjoitettaessa muistinvarausta hyödyntävissä sovelluksissa ja miten paljon muistivuodot voivat vaikuttaa suorituskykyyn.

4.2.4 Muut työkalut ja ratkaisut

Rajoitteista ja vaatimuksista johtuen monia työkaluvaihtoehtoja opetuskäyttöön jouduttiin sulkemaan pois. Suurimpina rajoitteina työkalujen valinnassa olivat erityisesti resurssien puute ja yhteensopimattomuus työkalujen välillä, joiden takia erityisesti käyttöliittymä- ja automaatiotestauksen toteuttamiseen jouduttiin soveltamaan Ubuntun komentorivin komentoja, joilla saataisiin toteutetuksi useamman testin käynnistäminen ja suorittaminen kerralla. Käyttöliittymätestauksen automatisoinnin toteuttamiseksi kehiteltiin menetelmä lukea tekstitiedosto ja käyttää siinä esiintyviä komentoja, jotka käyvät läpi sekä yksikkötestien suorittamisen että testattavan ohjelman käyttöliittymän toiminnot läpi.

Tietysti testatessa suuria yksikkötestimääriä, mitä yritysmaailmassa tehdään, olisi tällaisen testausjärjestelyn hallinta aivan liian työlästä ja siihen tarvittaisiin oikea työkalu yksikkötestien automatisointiin. Kuitenkin pienempien ohjelmien testaamiseen, joita opiskelijat varsinkin ensimmäisellä ohjelmointi- ja testauskurssilla saavat aikaan, on helpompi soveltaa tällaista järjestelyä. Myöhemmillä kursseilla, kun opiskelijoiden ohjelmat kasvavat ja he oppivat muita ohjelmointikieliä esimerkiksi Javaa, on suositellumpaa käyttää varsinaisia työkaluja niin automatisointiin kuin käyttöliittymätestaukseen. Näin voidaan vähentää työtaakkaa testauksessa ja yhteensopivampia työkaluja ohjelmointikielen osalta on helpompi löytää.

Muista työkaluvaihtoehdoista käyttöliittymä- ja automaatiotestaukseen nousivat Selenium ja Jenkins. Selenium on kokoelma avoimen lähdekoodin työkaluja, joilla voidaan automatisoida verkkopohjaisten sovellusten testausta. Työkaluilla voidaan muun muassa kirjoittaa ja suorittaa nettisivua käyttävä skripti, joka käy läpi verkkosivun toiminnot ja

käyttöliittymän testauksen automaattisesti. Vaikka työkalun käyttäminen antaisi paljon kokemusta opiskelijalle suuremman järjestelmän testaamisesta ja käyttöliittymätestauksesta, ei sitä voitu laittaa kurssille käytettäväksi johtuen mahdollisuudesta käyttää työkalua vain verkkopohjaisten sovellusten testaukseen. [57] Lisäksi työkalukokoelman käyttämiseen vaaditaan tietämystä verkkopohjaisten sovellusten rakenteesta, sillä muun muassa testausskriptejä kirjoitettaessa on tiedettävä testattavista objekteista. Koska kurssin osanottajilla ei ole esitietoja WWW-sovelluksista aiempien kurssien kautta, ei heiltä voida vaatia Seleniumin käyttöä.

Jenkins on avoimen lähdekoodin automaatiopalvelin, jota voi käyttää monien eri tehtävien automatisointiin, kuten koodin kehittämiseen, analysointiin, testaamiseen tai lähettämiseen ja käyttöönottoon. Jenkinsin asentaminen tulee suorittaa joko systeemipakettien, Docker-virtuaalisointi-ohjelman tai yksittäisenä JRE:n eli Java Runtime Environment-palvelun kanssa. Vaikka Jenkinsiin voi asentaa monia lisäosia ja sen käyttö testien suorittamisen automatisointiin on yksi sen ominaisuuksista, on Jenkins Java-pohjainen ja sille oli hankala löytää C-kieleen yhteensopivaa lisäosaa testien automatisointiin. Lisäksi Jenkins ei tukenut ollenkaan Cunit-yksikkötestaustyökalua. [58] Vaikka Jenkinsia ei pystytty käyttämään tutkimuksen kohdekurssilla, on työkalu silti erittäin käyttökelpoinen pohdittaessa projektipohjaisia kursseja tai jatkuvaa integrointia, jota tapahtuu käytettäessä TDD-kehitysmuotoa.

4.2.5 Tehtävät

Tutkimuksen kohdekurssille laaditut tehtävät tehtiin valittuihin työkaluihin ja kurssin sisältöön perustuen. Koska kurssin opetustavaksi oli jo valittu ohjelmoinnin ja testauksen yhdistetty opetus, ei projektiluontoisia tai TDD:een perustuvia tehtäviä ja niistä rakentuvaa opetusta voitu enää järjestää. Muut syyt aiemmin mainittujen opetusmetodien tehtävien hylkäämiseen olivat opiskelijoiden riittämättömät ennakkokurssivaatimukset ja taidot projektiosaamisen ja yleisen ohjelmointitason arvioinnin kannalta, sillä ohjelmistoprojekteista opettavia kursseja ensimmäisen vuositason opiskelijat eivät olleet vielä käyneet. Lisäksi perustuen S. Kollanuksen ja V. Isomöttösen TDD:n opetuskäytöstä saatuihin tutkimustuloksiin, ei TDD:aa suositella käytettäväksi liian aikaisin opetuksessa, sillä kehitysmetodin käyttö tehtävissä lisäisi huomattavasti opiskelijoiden työmäärä ja kehitysmetodi vaatisi jo kohtalasia ohjelmointitaitoja. [36]

Koska kurssin luento- ja harjoitusrakenne sekä opetusmetodi olivat jo valittuja, muotoiltiin tehtävät normaaleiksi harjoitustehtäviksi, jotka sisälsivät ja painottivat valittujen työkalujen käyttöä. Erityisesti Tarantulan käytön opettelulla haluttiin opiskelijoiden saavan ensimakua IT-yritysten arjesta, jossa myös testauksen raportointi ja hallinta ovat tärkeitä kokonaistoiminnan kannalta. Koska kurssi jakautui ohjelmointiin ja testaukseen, pohdittiin mahdollisuutta käyttää kurssin ohjelmointiosuudella opiskelijoiden aikaansaamia harjoitustöitä testauksen kohteina, jolloin opiskelijat näkisivät ja oppisivat sekä muiden tekemiä ratkaisuja että niiden sisältämiä virheitä. Kuten N. Harrisodin tutkimusartikkelissa todettiin, ovat sovelluksen kehittäjät monesti ”sokeita” omille virheilleen, jolloin toisten harjoitustöiden testaaminen ja muiden opiskelijoiden antamat testausraportit toimisivat oivana herätyksenä opiskelijoille ohjelmiensa ”virheettömyydestä” tai mahdottomuudesta löytää kaikkia virheitä. Tällöin harjoitustöitä ei voitaisi arvioida löytyneiden virheiden vaan testaussuoritusten ja niistä laadittujen dokumenttien perusteella. Kuitenkin opiskelijoiden työmäärä voi vaihdella rajusti, mikäli yhdellä osapuolella on testattavana paljon virheitä sisältävä ja miltei toimimaton sovellus ja toisella osapuolella täysin päinvastainen ratkaisu. [42]

Tehtävien vaikeutta kurssille ja opiskelijoiden osaavuutta oli hankala arvioida, sillä vaikka kurssin sisällöstä oltaisiin tietoisia, ei opiskelijoiden kyvystä omaksua kurssin ohjelmointitaitoja ollut varmuutta. Tämän ja aiemmin mainittujen syiden takia päädyttiin harjoitustehtävissä käyttämään testauskohteena yksinkertaista laskinsovellusta, jota testattaisiin testaustyökaluilla ja muilla testausratkaisulla sekä toiminta tallennettaisiin hallintatyökaluun. Vaikka tehtävän- ja sovelluksen jako tapahtuisi LUT-yliopiston Moodle-palvelun ja muiden yliopiston palveluiden kautta, tapahtuisi testauksen tulosten raportointi Tarantulaan. Kuitenkin ennen harjoitustehtävien käyttöönottoa, niiden sopivuudesta neuvoteltiin harjoitusluentojen järjestämisestä vastaavan opetusassistentin kanssa, jonka jälkeen opetusassistentti teki vielä lopulliset muutokset muun muassa virheiden kylvämisen osalta.

4.2.6 Ohjeistusvideot ja käyttöohjeet

Jotta työkalujen käytön opettelu olisi helpompaa ja nopeampaa, kuvattiin työkalujen käytöstä opetusvideot ja laadittiin kirjalliset käyttöohjeet sekä kurssin osanottajille että opetusvastaaville. Käyttöohjeita suunniteltiin käytettäväksi opetuksessa alkuarviolta 5 vuoden ajan, jonka lisäksi käyttöohjeita ja videoita pohdittiin annettavaksi muiden opetuslaitosten käyttöön. Käyttöohjeet eriteltäisiin opetusvastaavien ja opiskelijoiden kesken, jolloin

opiskelijat eivät pääsisi näkemään erityisesti Tarantulan hallintaan ja tietoturvaan liittyviä seikkoja, jotka auttaisivat heitä huijaamaan tai aiheuttamaan vaikeuksia kurssissa.

Työkalujen opastusvideot kuvattiin nauhoittamalla ne ilmaisella iSpring Free Cam 8-videotaltiointityökalulla ja suorittamalla työkalujen toimenpiteet virtuaalikoneella. Alkuun pohdittiin opastusvideoiden lataamista ja tallentamista videopalvelu Youtubeen yksittäiselle tilille, jolloin videot olisivat helposti opiskelijoiden saatavissa. Kuitenkin epävarmuus videoiden säilyvyydestä palvelussa, jos tiliä ei enää aiota käyttää ja ottaen huomioon myös vain opetusvastaaville tarkoitetut videot, päätettiin kaikki videot kirjallisine käyttöohjeineen lähettää kurssin järjestäjän haltuun. Sen jälkeen videot annettaisiin opiskelijoiden käyttöön videopalvelu Echo360:n kautta. Videoista kuvattiin kahdet versiot, joista ensimmäisiä käytettiin kevään opetuksessa ja toiset videot tehtiin kurssin jälkeen. Vaikka ensimmäiset videoversiot sopivat opetukseen, tehtiin uudemmat versiot selkeyden lisäämiseksi ja viittausten poistamisesta muihin videoihin. Näin halutessaan tietää tietyn toiminnon suorittamisen työkalulla, ei katsoja hämmenny viittauksesta muihin videoihin ja hän osaa valita ja katsoa vain tarvittavan videon.

4.3 Vaikutukset

Tutkimuksessa valittujen työkalujen ja harjoitustehtäväideoiden vaikutuksien mittaamiseen järjestettiin asiantuntijahaastattelu, jossa tutkimuksen aikaansaamat työkaluvalinnat ja harjoitustehtäväideat esiteltiin LUT-yliopiston tietotekniikan kandidaatinohjelman opetusvastaavalle, tutkimustyön ohjaajille ja seuraavan kurssin opetusassistentille. Alkuperäisenä ideana oli analysoida kurssin osanottajien jälkikyselyssä antamaa palautetta ja verrata näitä tuloksia aiempien kurssien palautteeseen. Lisäksi haluttiin verrata opiskelijoiden saamia arvosanoja nykyisen ja aiempien kurssien välillä. Ongelmana jälkikyselyn tulosten hyödyntämisessä oli niiden kattavuuden ja autenttisuuden varmuus. Vaikka kysely tuottaisi runsaasti palautetta, ei se varmista sitä, että opiskelijat ovat pohtineet tarpeeksi pitkään vastauksia ja vastanneet kysymyksiin ajatuksen kanssa. Vaikka keskiarvot toimisivat suuntaa antavina, voivat pikaisesti annetut ääripäävastaukset vaikuttaa huomattavasti virheellisen johtopäätöksen antamiseen. Loppujen lopuksi opiskelijapalautteen käytöstä jouduttiin luopumaan, koska jo palautteen määrä jäi vajaaksi eikä se ollut kattava.

4.3.1 Asiantuntijahaastattelu

Koska kurssin loppupalaute jäi opiskelijoiden taholta vajaaksi, järjestettiin tutkimuksen työkaluvalintojen ja tehtäväideoiden arviointiin asiantuntijahaastattelu, joka pidettiin 10. joulukuuta 2018 LUT-yliopiston tiloissa neuvotteluhuoneessa 7630. Asiantuntijaraatiin kuuluivat tutkimustyön ohjaajina toimineet kurssivastaava ja dosentti Uolevi Nikula sekä kurssin opetusassistentti ja tietotekniikan diplomi-insinööri Timo Hynninen. Työn ulkopuolisina asiantuntijoina haastattelussa toimivat LUT-yliopiston tietotekniikan kandidaattiohjelman opetusvastaava ja apulaisprofessori Jussi Kasurinen sekä tulevan kurssin opetusassistentti. Haastattelu käytiin aikavälillä 11:30-13:00, jota ennen haastatteluun osallistuneille oli annettu luettavaksi ennakkoon tutkimustyön silloinen versio.

Haastattelussa käytiin läpi ja esiteltiin tutkimuksessa kurssille valitut työkalut ja niiden pohjalta luodut harjoitustehtäväideat. Hallintatyökalu Tarantulan käyttöä demonstroitiin ja yksi työkalun käytön opetukseen tarkoitetuista videoista näytettiin haastattelijoille. Lisäksi haastattelussa perusteltiin asiantuntijaraadille syyt työkalu- ja harjoitustehtävävalintoihin, joista painotettiin erityisesti resurssien sekä opiskelijoiden ennakkotietojen puutteita. Lopuksi esiteltiin mahdollisuuksia ja ideoita testauksen opetuksen kehittämiselle tulevaisuudessa. Näitä olivat tutkimuksessa pohdittujen mutta kurssilla hyödyntämättömien työkalujen esimerkiksi Jenkinsin ja Seleniumin käyttö myöhemmillä kursseilla. . Haastattelun aikana kävivät ilmi syyt vajaaseen opiskelijapalautteeseen, sillä Timo Hynnisen mukaan testauksenhallintatyökalu Tarantulaa ei ehditty ottaa kurssille käyttöön ja näin ollen siihen liittyvistä tehtävistä jouduttiin luopumaan.

4.3.2 Haastattelun tulokset

Haastattelun jälkeen asiantuntijaraati kommentoi tutkimusratkaisua ja toi esille niin positiivisia kuin negatiivisia seikkoja. Positiivisina seikkoina tutkimuksen ratkaisusta asiantuntijaraati kommentoi tutkimuksen tavoitetason olevan hyväksyttävä ja että siinä keskityttiin työkaluilla paljon tekniseen suorittamiseen ja laadunhallintaan. Myös vaikeutta löytää sopivia työkaluja niin opiskelijoiden taitotason kuin resurssien puutteiden takia pidettiin yleisenä ongelmana esitellyssä tutkimuksessa, jonka takia mahdollisuutta hyödyntää tutkimuksessa esitettyjä työkaluja, esimerkiksi Seleniumia WWW-sovelluksia käsittelevillä kursseilla, katsottiin todennäköiseksi.

Vaikka tutkimuksessa valittujen työkalujen ja niiden pohjalta laadittujen harjoitustehtäväideoiden sisältö vastasi kurssin opetusohjelmaa, kommentoi asiantuntijaraati negatiivisina seikkoina työkaluvalintojen ja niistä laadittujen tehtävien tarvitsevan vielä viilausta erityisesti tehtävien määrän saralla. Lisäksi testausautomaation käytön osalta esitettiin lisää pohdintaa ja viilausta, sillä kuten myös esittelyssä todettiin, ei Ubuntun komennoilla ja tekstitiedostoilla voida varsinaisesti hoitaa testauksen automaatiota erityisesti hallitessa suurempia kokonaisuuksia. Myös opiskelijoiden massaimportointi testauksenhallintatyökalu Tarantulaan katsottiin ongelmaksi, sillä vaikka usean käyttäjien lisääminen voidaan hoitaa ulkoisesti CSV-tiedostolla, hidastaa sisäisen massaimportointitoiminnon puuttuminen työkalusta itsestään sen käytön tehokkuutta ja lisää kurssin opetusvastaavien työtä.

4.3.3 Haastattelun yhteenveto

Vaihtelevista positiivisista ja negatiivisista seikoista huolimatta asiantuntijaraati totesi, ettei työkalujen valinnassa opetuskäyttöön ole täysin oikeaa vastausta ja että on turhaa yrittää tunkea kaikkea materiaalia testauksesta yhdelle kurssille. Tämän takia katsottiin, johtuen myös kurssin suuntautumisesta ensimmäisen vuosikurssin opiskelijoille, että on sopivampaa opettaa enemmän testauksen konsepteja ja saada opiskelijat tajuamaan testauksen tärkeys ennen siirtymistä tarkempaan testauksen opetukseen myöhemmillä kursseilla. Näin ollen katsottiin mahdollisuudeksi panostaminen monimutkaisempien työkalujen käytön demoamiseen, jossa opiskelijat eivät välttämättä tee työkaluilla harjoitustehtäviä mutta pääsevät tutustumaan käytännön kautta työkalun hyödyntämiseen ennalta. Työkalujen varsinainen ja täysimittainen käyttö voidaan hoitaa myöhemmillä kursseilla, kun opiskelijoilla on enemmän kokemusta erityisesti ohjelmoinnin puolelta ja heiltä voidaan odottaa enemmän työskentelyn osalta. Näitä ovat esimerkiksi projektiluontoisten testauskurssien tarjoaminen maisteritason opiskelijoille, jossa kurssitehtävänä on oikean testausprojektin läpisuoritus työkaluineen ja dokumentteineen.

Haastattelun yhteenvedossa todettiin tutkimuksen keskittyvän enemmän testauksen opetusmetodien analysointiin ja opetuksen kehittämiseen kuin kurssin harjoitustehtäviin. Kuitenkin tutkimuksen vaikutuksista opetukseen ja sen jälkeiseen työpaikan saantiin testauspuolella, on asiantuntijaraadin mielestä hankala saada vielä kunnollista näyttöä. Miten muutettu opetus vaikuttaa opiskelijoilla työpaikan saamiseen, voidaan nähdä vasta noin 5 vuoden päästä, kun kohdekurssin käyneet todennäköisesti valmistuvat. Ja silloinkin tähän

vaikuttavat monet muut tekijät, esimerkiksi muilta kursseilta saadut kokemukset ja opitut tiedot ja taidot.

Vaikka tutkimus jäi käytännön, pääasiassa tehtävien, osalta vajaammaksi, tarjosi se paljon ideoita ja vaihtoehtoja hyödyntää tehtäviä ja työkaluja muilla ja myöhemmillä kursseilla. Tällöin opiskelijat omaavat suuremmat ennakkotiedot ja heiltä voidaan odottaa enemmän erityisesti maisteritason kursseja suunnitellessa. Esimerkiksi työlään testausjohtoisen kehityksen käyttö opetuksessa on todettu lisäävän koodin laatua opiskelijoilla, jonka lisäksi TDD:tä hyödyntävällä kurssilla testaushallinta- ja automaatiotestaustyökalujen käyttö on myös luonnollisempaa, kun halutaan hallita ja suorittaa suuria määriä yksikkötestejä nopeasti.

5 YHTEENVETO

Vaikka tutkimuksessa tulokset työkalujen hyödyntämisestä kurssilla jäivät laihoiksi johtuen tehtävien pienestä määrästä eikä testauksenhallintatyökalua ehditty käyttää, tarjoaa se silti pohjaa ja materiaalia tulevien kurssien työkalu- ja kurssitehtävävalinnoille. Lisäksi yhteenvetona voidaan todeta, ettei sopivien työkalujen valitsemiseen ja niiden hyödyntämiseen ohjelmistotestauksen opetuksessa ole oikeaa vastausta. Työkalun valintaan vaikuttivat monet tekijät, joista suurimmat olivat valittu testauksen opetusmetodi, kurssin tavoitteet ja päämäärä, käytössä olevat resurssit sekä kohderyhmän ennakkotiedot ja – taidot. Erityisesti ajan ja rahan puutteen takia maksullisia työkaluvaihtoehtoja jouduttiin sulkemaan pois ja turvautumaan avoimen lähdekoodin työkaluihin, missä taas muun muassa kurssilla painottuva ohjelmointikieli rajasi vaihtoehtoja entisestään. Kuitenkin jatkokehittämistä tehtävien laadun ja määrän sekä työkalujen valinnan osalta tulee suorittaa, jotta opetus saataisiin kehitettyä mahdollisimman kattavaksi ja yhteensopivaksi käytännössä, jolloin opiskelijoilla syntyy kiinnostusta ohjelmistotestausta kohtaan.

Koska ensimmäisillä testauskursseilla painotetaan pääasiassa testauksen periaatteiden ja konseptien oppimista, eikä kaikkea tietoa voi tunkea yhdelle kurssille, ei testaustyökalujen täysiaikainen käyttö siinä ole välttämätöntä. Vaikka opetus joudutaan jakamaan useammalle kurssille, luo ensimmäinen pohjat kiinnostumiselle aiheesta, joten vähintään kattava käytännön demonstraatio, jossa opiskelija pääsee itse kokeilemaan työkalun käyttöä, on erittäin suositeltavaa. Alussa on tärkeää herättää opiskelijoiden kiinnostus sekä iskostaa heille testauksen periaatteet ja tavoitteet, jotta opiskelijat ryhtyvät konkreettisesti suorittamaan testausta vaaditulla tavalla heiltä sitä pyydettäessä eikä vain nopeaa käyttöliittymän selailua pikaisesti. Myöhemmillä kursseilla, eikä vain ohjelmistotestauksen kursseilla, kun opiskelijalla on enemmän kokemusta ja ennakkotietoja aiemmilta kursseilta sekä heiltä voidaan odottaa enemmän, voidaan ruveta hyödyntämään laajempaa testaustyökaluvalikoimaa täysinäisesti. Lisäksi voidaan soveltaa muita ohjelmistotestauksen opetusmetodeja, joissa yritysten hyödyntämät työkalut ovat jopa tervetulleita. Esimerkiksi TDD:n käytössä testausautomaatiotyökalut yksikkötestien suorittamiseksi nopeasti tai projektiluontoisilla kursseilla testauksenhallintatyökalut yritysprojektivaikutelman luomiseksi. Testauksen laiminlyönti johtaa todistetusti suuriin kustannuksiin, ellei jopa ihmishenkien menetykseen, jonka takia tulevaisuuden tekijöitä tullaan aina tarvitsemaan.

6 LÄHTEET

1. Myers, Glenford J. revised and updated by Badgett, Tom & Thomas, Todd M. & Sandler Corey (2004) Art of software testing. Second edition.
   USA: John Wiley & Sons, Inc.
2. Ammann, Paul & Offutt, Jeff (2016) Introduction to software testing, Cambridge University Press
3. Bertolino, Antonia (2007) Software testing research: achievements, challenges, dreams. FOSE ’07 2007 Future of software engineering. 85 – 103.
4. Janzen, David & Saiedian, Hossein (2005) Test-driven development concepts, taxonomy, and future direction. Computer, 38(9):43-50
5. Kaner, Cem & Bach, James & Pettichord, Bret (2001) Lessons learned in software testing: a context-driven approach.
   USA: John Wiley & Sons, Inc.
   https://www.slideshare.net/Softwarecentral/the-realities-of-software-testing
   https://flylib.com/books/en/4.223.1.32/1/
6. Perry, William (2006) Effective methods for software testing. Third edition
   USA: John Wiley & Sons, Inc.
   https://www.softwaretestinggenius.com/download/EMFST.pdf
7. Singh, Yogesh (2011) Software testing, Cambridge University Press
8. Nidhra, Srinivas & Dondeti, Jagruthi (2012) Black box and white box testing techniques – a literature review, International Journal of Embedded Systems and Applications (IJESA) Volume 2, no.2
9. Bertolino, Antonia & Marchetti, Eda (2004) A brief essay on software testing, Technical report: 2004-TR-36
10. Jorgensen, Paul C. (2013) Software testing: a craftsman’s approach. Fourth edition. Auerbach Publishers Inc.
    http://webpages.iust.ac.ir/azgomi/Courses/ST/eBook2%20-%20Software%20Testing%20-%20A%20Craftsman,s%20Approach.pdf
11. Kasurinen, Jussi & Taipale, Ossi & Smolander Kari (2009) Software test automation in practice: empirical observations, Advances in software engineering Volume 2010
    https://www.hindawi.com/journals/ase/2010/620836/abs/

12. Meudec, Christophe (2001) ATGen: automatic test data generation using constraint programming and symbolic execution, Software testing, verification and reliability 2001, 11:81–96
https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.225
13. Polo, Macario & Tendero Sergio & Piattini Mario (2006) Integrating techniques and tools for testing automation, Software testing, verification and reliability 2007, 17:3–36
https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.348
14. Ramler, Rudolf & Wolfmaier, Klaus (2006) Economic perspectives in test automation: balancing automated and manual testing with opportunity cost, Proceedings of the 2006 international workshop on automation of software test, 85-91
https://dl.acm.org/citation.cfm?id=1138946
15. Lyu, Michael R. (1996) Handbook of software reliability engineering. USA, NJ, Hightstown: McGraw-Hill, Inc.
16. Do, Hyunsook & Elbaum, Sebastian & Rothermel, Gregg (2005) Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, Empirical Software Engineering, Volume 10, Issue 4, 405-435
https://link.springer.com/article/10.1007/s10664-005-3861-2
17. Utting, Mark & Legeard, Bruno (2010) Practical model-based testing: a tools approach, Elsevier
18. Harrold, Mary Jean (2000) Testing: a roadmap, Proceedings of the conference on the future of software engineering, 61-72,
https://dl.acm.org/citation.cfm?id=336532
19. Kazemian, Fereydoun & Howles, Trudy (2005) A software testing course for computer science majors, ACM SIGCSE Bulletin, volume 37, 50-53
20. Astigarraga, Tara & Dow, Eli M & Lara, Christina & Prewitt, Richard & Ward, Maria R (2010) The emerging role of software testing in curricula, Transforming Engineering Education: Creating Interdisciplinary Skills for Complex Global Environments, 2010 IEEE, 1-26
21. Geras, Adam M. & Smith, M. R. & Miller, J. (2004) A survey of software testing practices in Alberta, Canadian Journal of Electrical and Computer Engineering, volume 29, no. 3, 183-191

22. Chan, F. T. & Tang, W. H. & Chen, T. Y. (2005) Software testing education and training in Hong Kong, International Conference on Quality Software, 313-316
23. Timoney, Joseph & Brown, Stephen & Ye, Deshi (2008) Experiences in software testing education: some observations from an international cooperation, International Conference for Young Computer Scientists, 2686-2691
24. Garousi, Vahid & Mathur, Aditya (2010) Current state of the software testing education in North American academia and some recommendations for the new educators, Software Engineering Education and Training (CSEE\&T), 2010 23rd IEEE Conference on, 89-96
25. Garousi, Vahid & Varma, Tan (2010) A replicated survey of software testing practices in the Canadian province of Alberta: What has changed from 2004 to 2009?, The Journal of Systems and Software 83, 2251-2262
26. Jia, Songhao & Yang, Cai (2013) Teaching software testing based on CDIO, World Transactions on Engineering and Technology Education, volume 11, 476-47
27. Fu, Yujian & Clarke, Peter (2016) Gamification based cyber enabled learning environment of software testing, 123rd American Society for Engineering Education (ASEE)-Software Engineering Constituent
28. Garousi, Vahid (2010) An open modern software testing laboratory courseware, 23rd IEEE Conference on Software Engineering Education and Training, 177-184
29. Valle, Pedro Henrique Dias and Barbosa, Ellen Francine and Maldonado, Jose Carlos (2015) CS curricula of the most relevant universities in Brazil and abroad: Perspective of software testing education, Computers in Education (SIIE), 2015 International Symposium on, 62-68
30. Barbosa E. & Maldonado J. (2011) Collaborative development of educational modules: a need for lifelong learning, E-infrastructures and technologies for lifelong learning: next generation environments, 2011, 175-211
31. Lopez, Gustavo & Cocozza, Francisco & Martinez, Alexandra & Jenkins, Marcelo (2015) Design and implementation of a software testing training course, 122nd ASEE Annual Conference \& Exposition
32. Valle, Pedro Henrique Dias & Toda, Armando Maciel & Barbosa, Ellen Francine & Maldonado, José Carlos (2017) Educational games: A contribution to software testing education, Frontiers in Education Conference (FIE), 1-8

33. Soska, Alexander & Mottok, Jürgen & Wolff, Christian (2016) An experimental card game for software testing: Development, design and evaluation of a physical card game to deepen the knowledge of students in academic software testing education, Global Engineering Education Conference (EDUCON), 2016 IEEE, 576-584
34. Fraser, Gordon & Gambi, Alessio & Rojas, José Miguel (2018) A preliminary report on gamifying a software testing course with the Code Defenders testing game, Proceedings of the 3rd European Conference of Software Engineering Education, 50-54
35. Xie, Tao & de Halleux, Jonathan & Tillmann, Nikolai & Schulte, Wolfram (2010) Teaching and training developer-testing techniques and tool support, Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, 175-182
36. Kollanus, Sami & Isomöttönen, Ville (2008) Test-driven development in education: experiences with critical viewpoints, ACM SIGCSE Bulletin, volume 40, 124-127
37. Keefe, Karen & Sheard, Judithe & Dick, Martin (2006) Adopting XP practices for teaching object oriented programming, Eighth Australasian Computing Education Conference (ACE2006)
38. Edwards, Stephen H. (2004) Using software testing to move students from trial-and-error to reflection-in-action, SIGCSE ’04 Proceedings of the 35th SIGCSE technical symposium on Computer science education, 26-30
39. De Souza, Draylson. M. & Oliveira, Bruno H. & Maldonado, José C. & Souza, Simone R. S. & Barbosa, Ellen F. (2014) Towards the use of an automatic assessment system in the teaching of software testing, Proc. of FIE 2014, 1-8
40. Barbosa E. & Silva M. & Corte C. K. D. & Maldonado J. C. (2008) Integrated teaching of programming foundations and software testing, Frontiers in Education Conference, 22-25
41. Corte C. K. D. and Maldonado J. C. (2006) Integrated teaching of programming foundations and software testing, Master's thesis, University of São Paulo, Institute of mathematics and computer science
42. Harrison, Neil B (2010) Teaching software testing from two viewpoints, Journal of Computing Sciences in Colleges, volume 26, 55-62, Consortium for Computing Sciences in Colleges

43. Ramasamy, Vijayalakshmi & Alomari, Hakam W. & Kiper, James D. & Potvin, Geoffrey (2018) A minimally disruptive approach of integrating testing into computer programming courses, SEEM ’18 Proceedings of the 2nd International Workshop on Software Engineering Education for Millenials, 1-7
44. Zhu, Bin & Zhang, Shiming (2014) Experiment teaching reform for software testing course based on CDIO, Computer science & education (ICCSE), 2014 9th international conference on, 488-491
45. Martinez, Alexandra (2018) Use of JiTT in a graduate software testing course: an experience report, Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training, 108-115
46. Garousi, Vahid (2011) Incorporating real-world industrial testing projects in software testing courses: Opportunities, challenges, and lessons learned, Software Engineering Education and Training (CSEE\&T), 24th IEEE-CS Conference on, 396-400
47. Vidger, Mark R. & Dean, John (1997) An architectural approach to building systems from COTS software components, CASCON ’97, Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, p. 22
48. Benguria, Gorka & García, Ana, Belén & Sellier, David & Tay, Sandy (2002) European COTS user working group: analysis of the common problems and current practices of the European COTS users, COTS-based software systems, Proceedings of the First International Conference, ICCBSS 2002, 44-53
49. Comella-Dorda, Santiago & Dean, John C. Dean & Morris, Edwin & Oberndorf, Patricia (2001) A process for COTS software product evaluation, COTS-based software systems, Proceedings of the First International Conference, ICCBSS 2002, 86-96
50. LUT-yliopiston WebOodi-sivusto, Opintojakson CT60A0220 *C-ohjelmoinnin ja testauksen periaatteet* kuvaus. <https://weboodi.lut.fi>. Viitattu 25.11.2018
51. Testauksenhallintatyökalu Tarantulan vanha WWW-sivu, <http://www.testiatarantula.com/old/>. Viitattu 25.11.2018
52. Prove Expertise Oy:n luoma verkkosivu informaation antamiseen testauksenhallintatyökalu Tarantulan nykyisestä tilasta, <http://www.testiatarantula.com/>. Viitattu 25.11.2018

53. Testauksenhallintatyökalu Tarantulan yhteisöverkkosivu ja -foorumi avun ja neuvojen saatiin työkalun osalta, <https://getsatisfaction.com/prove>. Viitattu 25.11.2018
54. Cunit-yksikkötestaustyökalun WWW-sivu työkalun käyttöohjeiden ja muun dokumentaation saamiseen, <http://cunit.sourceforge.net/>, Viitattu 26.11.2018
55. Valgring-työkalukehyksen WWW-sivu työkalun käyttöohjeiden ja muun dokumentaation saamiseen, <http://www.valgrind.org/>, Viitattu 26.11.2018
56. Valgring-työkalukehyksen manuaalin WWW-sivun Memcheck-työkalun osio. <http://valgrind.org/docs/manual/mc-manual.html> Viitattu 26.11.2018
57. Selenium-työkalun WWW-sivu, Seleniumin esittelyosio. <https://www.seleniumhq.org/docs/01_introducing_selenium.jsp#introducing-selenium>, Viitattu 27.11.2018
58. Jenkins-työkalun WWW-sivu, Jenkinsin esittely- ja dokumenttiosio. <https://jenkins.io/doc/>, Viitattu 27.11.2018
59. Opintojakson kuvaus opinto-oppaan mukaan, Kurssin sisältö ja suoritustavat. LUT-yliopisto, WebOodi-sivusto, Opintojakson CT60A0220 kuvaus. <https://weboodi.lut.fi/oodi/opintjakstied.jsp?MD5avain=&Kieli=1&OpinKohd=23417172&OnkoIlmKelp=0&ooo_SortJarj=2&vl_tila=4&takaisin=vl_kehys.jsp&Opas=65&Org=1&haettuOpas=65&haeOpintJaks=haeopintojaksot>. Viitattu 25.11.2018
60. A typical sequence of steps for using the CUnit framework, Cunit-yksikkötestaustyökalun WWW-sivu työkalun käyttöohjeiden ja muun dokumentaation saamiseen, <http://cunit.sourceforge.net/doc/introduction.html>, Viitattu 26.11.2018
61. Taulukko 4. Following problems that are common in C and C++-programs, Valgring-työkalukehyksen manuaalin WWW-sivun Memcheck-työkalun osio. <http://valgrind.org/docs/manual/mc-manual.html> Viitattu 26.11.2018
62. Kuva 1. V-model. Tutorialspoint. Viitattu 19.9.2018. Saatavissa: https://www.tutorialspoint.com/software_testing_dictionary/v_model.htm

LIITE 1: TYÖKALUJEN KÄYTTÖTAPAUKSET JA VAATIMUKSET

SISÄLLYSLUETTELO

1 TYÖKALUJEN YLEISVAATIMUKSET

Vaatimus ID	CR001
Prioriteetti	keskitaso
Skenaario	Työkalun hinta
Kuvaus	Mikäli työkalu on maksullinen, sen hankintahinta on oltava sopiva ottaen huomioon ostajan budjetti, hintatoiveet ja opiskelijoiden määrä, joita on arvioitu olevan noin 150 henkilöä.

Vaatimus ID	CR002
Prioriteetti	korkea
Skenaario	Avoimen lähdekoodin työkalu
Kuvaus	Työkalu on avoimen lähdekoodin työkalu, jolloin sen käyttöönotto ei maksa mitään.

Vaatimus ID	CR003
Prioriteetti	korkea
Skenaario	Työkalun yhteensopivuus käyttöjärjestelmään.
Kuvaus	Työkalu on yhteensopiva niiden tietokoneiden käyttöjärjestelmän kanssa, joita kurssin osanottajat käyttävät. Tässä tapauksessa Linux Ubuntu.

Vaatimus ID	CR004
Prioriteetti	korkea
Skenaario	Työkalun yhteensopivuus muiden testaustyökalujen kanssa.
Kuvaus	Työkalu on yhteensopiva muiden työkalujen kanssa, joita kurssin osanottajat käyttävät.

Vaatimus ID	CR005
Prioriteetti	korkea
Skenaario	Työkalun tuki verkossa ongelmatilanteisiin
Kuvaus	Työkalulla on nettisivu/foorumi, josta voi hakea tietoa ongelmatapauksissa.

Vaatimus ID	CR006
Prioriteetti	korkea
Skenaario	Työkalun käyttöohjeet ja toimintojen selitykset verkossa
Kuvaus	Työkalun nettisivulla/foorumilla löytyvät käyttöohjeet ja toimintojen selitykset työkalun käyttöön.

Vaatimus ID	CR007
Prioriteetti	korkea
Skenaario	Mahdollisuus työkalun käyttöön muualla kuin yliopiston koneilla.
Kuvaus	Työkalun käyttö- ja asennuslupia voidaan jakaa edelleen opiskelijoille, jolloin heillä on mahdollista työkalun asennukseen ja käyttöön omilla koneillaan, eikä vain yliopiston koneilla.

Vaatimus ID	CR008
Prioriteetti	korkea
Skenaario	Työkalun helppo asentaminen.
Kuvaus	Työkalu on helppo asentaa käytettäväksi omalle koneelle opiskelijoiden taholta.

Vaatimus ID	CR009
Prioriteetti	keskitaso
Skenaario	Visuaalinen käyttöliittymä
Kuvaus	Työkalua voidaan käyttää graafisen käyttöliittymän kautta.

Vaatimus ID	CR010
Prioriteetti	korkea
Skenaario	Työkalulla aktiivinen tuki verkossa päivityksiin
Kuvaus	Työkalulla on aktiivinen ylläpito ja sille on saatavissa päivityksiä.

Vaatimus ID	CR011
Prioriteetti	keskitaso
Skenaario	Helppokäyttöinen käyttöliittymä
Kuvaus	Työkalun käyttöliittymä on yksikertainen ja sen käyttö on helppo opettaa uusille käyttäjille, joita ovat muuan muassa opiskelijat, opetusvastaavat ja ylläpitäjät.

2 TESTAUKSEN HALLINTA

2.1 Käyttötapaukset

Käyttäjien massaimportit	
Käyttötapaus ID	TM001
Prioriteetti	korkea
Kuvaus	Työkalulla on mahdollista suorittaa toimintoja koskien suurta käyttäjäryhmää esim. luoda tunnukset ja antaa lupa työkalun käyttöön usealle käyttäjälle yhdellä kertaa. Eri tyypin käyttölupia on myös oltava. Päällimmäisenä vaihtoehtona on CSV –tiedoston käyttö eksportoimalla se tietokannasta ja luomalla Excelillä. Massaimportointiin voidaan käyttää tämän vaihtoehdon puuttuessa eri työkaluja.
Käyttäjä	Ylläpitäjä, opetusassistentti, kurssin pitäjä
Alkutila	Työkalu on asennettu koneeseen. Ylläpitäjälle on luotu jo omat tunnukset. Listat kurssin osanottajien opiskelijanumeroista ja sähköposteista ovat tiedossa. (Keksitty lista käy aluksi.) Listasta on luotu CSV –tiedosto. Massaimportointiin tarvittava automatisoiva työkalu on asennettu ja työkalulle tarvittava skripti on luotu.
Vaiheet	
1.	Käyttäjä syöttää tunnukset ja salasanan, jonka jälkeen kirjautuu järjestelmään.
2.	Käyttäjä valitsee työkalun asetuksia ja käyttäjiä käsittelevän osion.
3.	Käyttäjä valitsee osion uusien käyttäjien rekisteröintiin.
4.	Käyttäjä avaa työkalun botin käyttämiseen uusien käyttäjien rekisteröimiseksi automaattisesti.
5.	Käyttäjä syöttää botin toimintoa ohjailevan skriptin.
6.	Käyttäjä käynnistää botin, joka skriptin ja testauksenhallintatyökalun avulla luo uudet tunnukset ja lähettää tiedot uusien käyttäjien sähköposteihin.
Lopputila	
Uudet käyttäjät ovat saaneet sähköpostiin viestin ja tunnukset testaushallintatyökalun käyttöön.	

Qualitymanager-mahdollisuus	
Käyttötapaus ID	TM002
Prioriteetti	korkea
Kuvaus	Työkalulla on mahdollista suorittaa laadunvalvontaa, mikä tarkoittaa muiden käyttäjien työn edistymisen seurantaa ja muun seurantainformaation tarjoamista muun muassa opetusassistentille töiden tarkastamista varten. Nopea laadunvalvontamahdollisuus helpottaa huomattavasti esimerkiksi kurssitöiden etenemisen tarkkailua.
Käyttäjä	Opetusassistentti, kurssin pitäjä
Alkutila	Työkalu on asennettu koneeseen. Kurssivastaavalle/opetusassistentille on luotu jo omat tunnukset. Muilla käyttäjillä on tunnukset järjestelmässä. Työkalulla on luotu projekti. Työkaluun on integroitu muita projektista informaatiota tarjoavia työkaluja esim. bugitietokantaa ylläpitävä työkalu.
Vaiheet	
1.	Käyttäjä syöttää tunnukset ja salasanan, jonka jälkeen kirjautuu järjestelmään.
2.	Käyttäjä valitsee tietyn käyttäjäryhmän projektin.
3.	Käyttäjä valitsee projektin edistymistä seuraavan osion
4.	Käyttäjä avaa hallintatyökaluun integroidun osan muiden tarvittavien informaatioiden tarkistamiseksi.
5.	Käyttäjä käynnistää integroidut työkalut, jotka tarkistavat mm. projektin testijonot ja päivittävät bugitietokannan.
6.	Työkalu palauttaa raportin integroitujen työkalujen tekemän ajon jälkeen projektista.
Lopputila	
Käyttäjä näkee seurantasivulla projektin edistymisen ja integroitujen työkalujen antamat raportit.	

Käyttäjien oikeuksien jakaminen	
Käyttötapaus ID	TM003
Prioriteetti	korkea
Kuvaus	Työkalulla on mahdollista jakaa ja muuttaa erilaisia käyttöoikeuksia, jolloin voidaan estää opiskelijaryhmien pääsyn toistensa projekteihin estäen mm. töiden kopioimisen mahdollisuus. Työkalussa on eri käyttäjätyypin vaihtoehtoja ja käyttöoikeuksia esimerkiksi käyttäjien ja ryhmien välityksellä.
Käyttäjä	Ylläpitäjä, opetusassistentti, kurssin pitäjä
Alkutila	Työkalu on asennettu koneisiin. Käyttäjälle on luotu jo omat tunnukset. Muilla käyttäjillä/ryhmillä on tunnukset järjestelmässä. Työkalulla on luotu projekteja ja projekteja on muokattu hiljattain. Ylläpitäjällä on lista ryhmistä ja niihin kuuluvista käyttäjistä.
Vaiheet	
1.	Käyttäjä syöttää tunnukset ja salasanan, jonka jälkeen kirjautuu järjestelmään.
2.	Käyttäjä valitsee asetukset ja käyttäjiä koskevan osion.
3.	Käyttäjä valitsee käyttäjiä osion
4.	Käyttäjä avaa hallintatyökaluun integroidun osan muiden tarvittavien informaatioiden tarkistamiseksi.
5.	Käyttäjä käynnistää integroidut työkalut, jotka tarkistavat mm. projektin testijonot ja päivittävät bugitietokannan.
6.	Työkalu palauttaa raportin integroitujen työkalujen tekemän ajon jälkeen projektista.
Lopputila	
Käyttäjä näkee seurantasivulla projektin edistymisen ja muiden työkalujen antamat raportit.	

Koodikattavuus-ominaisuus	
Käyttötapaus ID	TM004
Prioriteetti	korkea
Kuvaus	Työkalulla on itsellään tai integroitavissa oleva koodikattavuuden mittaava ominaisuus. Työkalulla on mahdollista tarkistaa, miten suuren rivimäärän koodia ajettu testi on käynyt läpi. Koodikattavuus ominaisuus mahdollistaa kurssin osanottajien testien kattavuuden arvioinnin, kun nähdään, kävikö testausjono todella halutut koodirivit läpi.
Käyttäjä	Ylläpitäjä, opetusassistentti, kurssin pitäjä, kurssin osanottaja
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Ylläpitäjälle/ on luotu jo omat tunnukset. Työkaluun on integroitu testauksen suorittava ja koodikattavuutta mittaava työkalu. Projekti sisältää testattavan ohjelman ja sen testaavan skriptin.
Vaiheet	
1.	Käyttäjä tarkistaa työkalun ostosivulta tuotteen koodikattavuus ominaisuudet
2.	Käyttäjä tarkistaa integroitavien työkalujen verkkosivuilta koodikattavuus ominaisuudet
3.	Käyttäjä syöttää tunnukset ja salasanan, jonka jälkeen kirjautuu järjestelmään.
4.	Käyttäjä valitsee tietyn käyttäjäryhmän projektin.
5.	Käyttäjä valitsee projektista testattavan koodin ja sitä testaavan skriptin
6.	Käyttäjä avaa koodikattavuutta mittaavan työkalun/Varmistaa, että työkalu on toiminnassa ennen testin ajoa.
7.	Käyttäjä avaa hallintatyökaluun integroidun testaustyökalun. (Yksikkötestaus)
8.	Käyttäjä ajaa yksikkötestauksen skriptin.
9.	Työkalu palauttaa raportin ohessa tiedot testiajon koodikattavuudesta
Lopputila	
Käyttäjä näkee raportista rivimäärän, jonka testausskripti on käynyt läpi.	

Bugitietokanta	
Käyttötapaus ID	TM005
Prioriteetti	korkea
Kuvaus	Työkalulla on itsellään tai integroitavissa oleva bugitietokanta-ominaisuus. Työkalulla on mahdollista tarkistaa ja ylläpitää listaa, miten suuren määrän virheitä testattava koodi pitää sisällään. Lisäksi listan tulee päivittyä. Näin esimerkiksi opetusassistentti voi katsoa, miten paljon kurssin osanottajat löysivät virheitä eri testauskerroilla.
Käyttäjä	Ylläpitäjä, opetusassistentti, kurssin pitäjä
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneisiin. Ylläpitäjälle/laadunvalvojalle on luotu jo omat tunnukset. Työkaluun on integroitu testauksen suorittava ja koodikattavuutta mittaava työkalu. Projekti sisältää testattavan ohjelman ja sen testaavan skriptin. Ohjelmasta on versiot ennen ja jälkeen muokkauksen.
Vaiheet	
1.	Käyttäjä tarkistaa työkalun verkkosivulta tuotteen bugitietokanta-ominaisuudet
2.	Käyttäjä tarkistaa integroitavien työkalujen verkkosivuilta bugitietokanta-ominaisuudet
3.	Käyttäjä syöttää tunnukset ja salasanan, jonka jälkeen kirjautuu järjestelmään.
4.	Käyttäjä valitsee tietyn käyttäjäryhmän projektin.
5.	Käyttäjä valitsee projektista testattavan koodin vanhemman version.
6.	Käyttäjä avaa bugitietokantaa ylläpitävän työkalun/Varmistaa, että työkalu on toiminnassa ennen testin ajoa.
7.	Käyttäjä katsoo bugitietokannasta koodin virheiden määrän.
8.	Käyttäjä valitsee projektista testattavan koodin uudemman version.
9.	Käyttäjä katsoo bugitietokannasta koodin virheiden määrän.
10.	Käyttäjä vertaa virheiden määrää keskenään.
Lopputila	
Käyttäjä näkee bugitietokannasta koodin eri versioiden virhemäärät.	

2.2 Vaatimukset

Vaatimus ID	TMR001
Prioriteetti	korkea
Skenaario	Suuri käyttäjämäärä ja kapasiteetti
Kuvaus	Työkalulla on mahdollista luoda tunnukset 150:lle kurssiin osaaottavalle käyttäjälle ja kapasiteettia suuren käyttäjämäärän ylläpitoon.

Vaatimus ID	TMR002
Prioriteetti	korkea
Skenaario	Yksikkötestaustyökalun integroituvuus ja yhteensopivuus
Kuvaus	Hallintatyökaluun on integroitavissa kurssilla käytettävä yksikkötestaustyökalu tai se on muuten yhteensopiva. Yksikkötestaus näkyy hallintatyökalussa.

Vaatimus ID	TMR003
Prioriteetti	korkea
Skenaario	Integraatiotestaustyökalun integroitavuus ja yhteensopivuus
Kuvaus	Hallintatyökaluun on integroitavissa kurssilla käytettävä integraatiotestaustyökalu tai se on muuten yhteensopiva. Integraatiotestaus näkyy hallintatyökalussa.

Vaatimus ID	TMR004
Prioriteetti	korkea
Skenaario	Järjestelmätestaustyökalun integroitavuus ja yhteensopivuus
Kuvaus	Hallintatyökaluun on integroitavissa kurssilla käytettävä järjestelmätestaustyökalu tai se on muuten yhteensopiva. Järjestelmätestaus näkyy hallintatyökalussa.

Vaatimus ID	TMR005
Prioriteetti	korkea
Skenaario	Massaimportointi CSV -tiedostolla
Kuvaus	Työkalulla on mahdollista luoda tunnukset 150:lle kurssiin osaaottavalle käyttäjälle käyttämällä CSV -tiedostoa, joka on eksportoitu tietokannasta tai luotu Microsoft Excelillä.

Vaatimus ID	TMR006
Prioriteetti	keskitaso
Skenaario	Massaimportointi LTI –linkityksellä
Kuvaus	Työkalulla on mahdollista luoda tunnukset 150:lle kurssiin osaaottavalle käyttäjälle käyttämällä LTI –linkitystä, mikäli importointi ei ole mahdollista CSV:llä.

Vaatimus ID	TMR007
Prioriteetti	korkea
Skenaario	Kirjautuminen ja henkilökohtaiset tunnukset
Kuvaus	Käyttäjä kirjautuu järjestelmään omalla tunnuksellaan ja salasanallaan.

Vaatimus ID	TMR008
Prioriteetti	korkea
Skenaario	Ryhmien luonti
Kuvaus	Työkalulla on mahdollista luoda ryhmiä ja lisätä/vaihtaa niihin eri käyttäjiä.

Vaatimus ID	TMR009
Prioriteetti	korkea
Skenaario	Projektien luonti ja vastuunjako projekteihin
Kuvaus	Työkalulla on mahdollista luoda projekteja ja asettaa ryhmiä/käyttäjiä niistä vastuullisiksi.

Vaatimus ID	TMR010
Prioriteetti	keskitaso
Skenaario	Testaussuunnitelmien luonti
Kuvaus	Työkalulla on mahdollista luoda projekteihin testaussuunnitelmia ja vastuita niistä voidaan jakaa eri käyttäjille/ryhmille.

Vaatimus ID	TMR011
Prioriteetti	korkea
Skenaario	Testitapausten luonti
Kuvaus	Työkalulla on mahdollista luoda projekteihin testitapauksia, joita voidaan siirtää testijonoihin ja vastuita testitapauksista voi jakaa eri käyttäjille/ryhmille.

Vaatimus ID	TMR012
Prioriteetti	korkea
Skenaario	Testijonojen luonti
Kuvaus	Työkalulla on mahdollista luoda projekteihin testijonoja, niihin voidaan liittää testitapauksia ja vastuita testijonoista voi jakaa eri käyttäjille/ryhmille.

Vaatimus ID	TMR013
Prioriteetti	korkea
Skenaario	Testivaatimusten luonti
Kuvaus	Työkalulla on mahdollista luoda projekteihin testivaatimuksia, niihin voidaan liittää testitapauksia ja –jonoja, ja vastuita suorituksista voi jakaa eri käyttäjille/ryhmille.

Vaatimus ID	TMR014
Prioriteetti	korkea
Skenaario	Testien suoritusten merkkaaminen
Kuvaus	Työkalulla on mahdollista merkitä suoritettuja testitapauksia ja –jonoja.

Vaatimus ID	TMR015
Prioriteetti	korkea
Skenaario	Testien suoritusten raportointi
Kuvaus	Työkalulla on mahdollista luoda ja tulostaa raportteja suoritetuista testitapauksista ja –jonoista.

Vaatimus ID	TMR016
Prioriteetti	keskitaso
Skenaario	Projektien luonti
Kuvaus	Työkalulla on mahdollista luoda ja tulostaa raportteja projektin tilasta

Vaatimus ID	TMR017
Prioriteetti	keskitaso
Skenaario	Dokumenttien ja artefaktien linkitys
Kuvaus	Työkalulla on mahdollista linkittää dokumentteja/artefakteja projekteihin esimerkiksi muistiinpanoiksi.

Vaatimus ID	TMR018
Prioriteetti	keskitaso
Skenaario	Yksittäisten tehtävien jakaminen
Kuvaus	Työkalulla on mahdollista antaa yksittäisiä tehtäviä ryhmien ja käyttäjien tehtäviksi. Annetut tehtävät näkyvät niistä vastuullisille.

3 YKSIKKÖTESTAUS

3.1 Käyttötapaukset

Yksikkötestin kirjoitus	
Käyttötapaus ID	UT001
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan kirjoittaa yksikkötestejä ja käyttää niissä työkalun tarjoamia testikomentoja. Yksikkötestin voi luoda joko työkalulla itsellään tai kirjoittaa sen itse ja käyttää luotavassa testitiedostossa työkalukehyksen tarjoamia komentoja.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Luotavalle yksikkötestille on ennestään luotu sitä ohjaava testitapaus. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta.
Vaiheet	
1.	Avataan työkalu työpöydältä tai komentorivillä.
2.	Valitaan luotavaksi uusi yksikkötestitiedosto.
3.	Aloitetaan testitiedoston kirjoittaminen lisäämällä työkalun tarvitsemat kirjastot.
4.	Kirjoitetaan viittaukset muihin tarvittaviin kirjastoihin ja testattavien funktioiden tiedostoihin.
5.	Kirjoitetaan testattavat funktiot, mikäli niitä ei voi saada viittauksien avulla testattavaksi halutun testitapauksen mukaan.
6.	Kirjoitetaan työkalun käyttämät testitapausfunktiot ja –komennot.
7.	Kirjoitetaan main-funktio, joka sisältää kaikki yksikkötestin suoritukseen tarvittavat ja työkalun tarjoamat komennot. Näitä voivat olla muun muassa testirekisterin aktivointi.
8	Kirjoitetaan yksikkötestin raportointitapa ja raportin nimi, mikäli sitä ei valita erikseen jälkeenpäin.
9.	Kirjoitetaan yksikkötestin suoritustapa, mikäli sitä ei valita erikseen jälkeenpäin.
10.	Tallennetaan yksikkötesti haluttuun kansion valitsemalla tallennusvaihtoehto.
Lopputila	
Yksikkötesti on tallennettu kohdekansioon ja on valmis suoritettavaksi.	

Yksikkötestin suoritus komentorivillä tai graafisella käyttöliittymällä	
Käyttötapaus ID	UT002
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan suorittaa yksikkötestejä joko itse työkalulla graafisen käyttöliittymän kautta tai komentorivillä.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta. Suoritettavalle yksikkötestille on ennestään luotu sitä ohjaava testitapaus. Suoritettava yksikkötesti on luotu työkalulla jo aiemmin.
Vaiheet	
1.	Aukaistaan komentorivi työpöydältä.
2.	Siirrytään kohdekansioon, jossa suoritettava yksikkötesti sijaitsee.
3.	Käännetään testitiedosto ja siinä käytettävät muut tiedostot.
4.	Ajetaan komentorivillä käännetty testitiedosto.
5.	Katsotaan komentoriviltä työkalun yksikkötestin antamat tulokset
tai	
1.	Aukaistaan yksikkötestaustyökalu työpöydältä.
2.	Valitaan työkalun valikosta yksikkötestin avaaminen.
3.	Siirrytään kohdekansioon, jossa suoritettava yksikkötesti sijaitsee.
4.	Avataan suoritettava yksikkötesti.
5.	Valitaan valikosta yksikkötestin suoritus.
6.	Valitaan yksikkötestin suorituksen asetukset.
7.	Käynnistetään yksikkötestin suoritus.
8.	Katsotaan yksikkötestistä saadut tulokset uudesta ikkunasta.
9.	Tallennetaan ja tulostetaan halutessa saadut tulokset raporttiin valikosta.
Lopputila	
Yksikkötestin tulokset näkyvät ruudulla joko graafisella käyttöliittymällä tai komentorivillä. Komentorivillä saatu raportti on jo tulostettu kohdekansioon, mikäli raportin tulostukseen tarvittava komento on lisätty testitiedostoon. Graafisessa käyttöliittymässä raportin tulostamiseen ja tallentamiseen ilmestyy valintavalikko.	

Testiraportin tulostus ja selaus komentorivillä	
Käyttötapaus ID	UT003
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan tulostaa yksikkötesteistä saadut tulosraportit erilliseen tiedostoon. Kyseistä raporttia voi selata jälkeenpäin.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Luotavalle yksikkötestille on ennestään luotu sitä ohjaava testitapaus. Suoritettava yksikkötesti on luotu työkalulla jo aiemmin. Mikäli yksikkötesti on suoritettu komentorivillä, raporttitiedoston luontiin vaadittu komento on lisätty suoritettavaan yksikkötestiin. Mikäli testien tuloksia haluaa selata komentorivin kautta, pitää vaadittu komento olla lisättynä suoritettavaan yksikkötestiin. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta.
Vaiheet	
1.	Aukaistaan komentorivi työpöydältä.
2.	Siirrytään kohdekansioon, jossa suoritettava yksikkötesti sijaitsee.
3.	Käännetään testitiedosto ja siinä käytettävät muut tiedostot.
4.	Ajetaan komentorivillä käännetty testitiedosto.
5.	Katsotaan komentoriviltä työkalun yksikkötestin antamat tulokset
6.	Valitaan komennot haluttujen tietojen katsomiseksi raportista.
7.	Avataan raporttitiedosto kansiosta, jonne sen oli testitiedostossa määrätty luoda.
Lopputila	
Käyttäjälle näkyy raportti tiedostosta ja komentorivillä näkyy käyttäjän valitsemat tiedot testausraportista.	

Testiraportin tulostus ja selaus graafisella käyttöliittymällä	
Käyttötapaus ID	UT004
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan tulostaa yksikkötesteistä saadut tulosraportit erilliseen tiedostoon. Kyseistä raporttia voi selata jälkeenpäin.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Luotavalle yksikkötestille on ennestään luotu sitä ohjaava testitapaus. Suoritettava yksikkötesti on luotu ja suoritettu työkalulla jo aiemmin. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta.
Vaiheet	
1.	Aukaistaan yksikkötestaustyökalu työpöydältä.
2.	Valitaan työkalun valikosta yksikkötestin avaaminen.
3.	Siirrytään kohdekansioon, jossa suoritettava yksikkötesti sijaitsee.
4.	Avataan suoritettava yksikkötesti.
5.	Valitaan valikosta yksikkötestin suoritus.
6.	Valitaan yksitestin suorituksen asetukset.
7.	Käynnistetään yksikkötestin suoritus.
8.	Selataan työkalun antamaa ja ruudulla näkyvää raporttia.
8.	Valitaan valikosta raportin tallennus.
9.	Valitaan tiedot, jotka halutaan liittää ja tallentaa luotavaan tulosraporttiin.
10.	Hyväksytään raportin tallennus kohdekansioon.
Lopputila	
Käyttäjälle näkyy luotu raporttitiedosto ja käyttäjä voi edelleen selata työkalulla näkyvää raporttia.	

3.2 Vaatimukset

Vaatimus ID	UTR001
Prioriteetti	korkea
Skenaario	Työkalun yhteensopivuus C-kielelle
Kuvaus	Työkalulla voidaan tehdä yksikkötestejä C-ohjelmointikielellä koodatuille sovelluksille.

Vaatimus ID	UTR002
Prioriteetti	korkea
Skenaario	ASSERT-komennot
Kuvaus	Työkalulla on kattava kirjasto komentoja funktioiden testailuun.

Vaatimus ID	UTR003
Prioriteetti	korkea
Skenaario	Komentorivi käyttöliittymänä
Kuvaus	Työkalua voidaan käyttää komentorivin kautta.

Vaatimus ID	UTR004
Prioriteetti	korkea
Skenaario	Testausraportin tulostus näytölle
Kuvaus	Työkalulla voidaan tulostaa suoritetun yksikkötestin tulokset käyttöliittymän ruudulle.

Vaatimus ID	UTR005
Prioriteetti	korkea
Skenaario	Testausraportin tulostus tiedostoon
Kuvaus	Työkalulla voidaan tulostaa suoritetun yksikkötestin tulokset uuteen tiedostoon

Vaatimus ID	UTR006
Prioriteetti	korkea
Skenaario	Raportin tiedot virheiden paikkojen osalta.
Kuvaus	Työkalun tulostamalla testausraportilla näkyy havaittujen virheiden paikat ajetussa yksikkötestissä.

Vaatimus ID	UTR007
Prioriteetti	keskitaso
Skenaario	Yksikkötestien suoritus erikseen testijonoissa
Kuvaus	Työkalulla voidaan tehdä suorittaa testijonojen yksikkötestejä valikoivasti.

Vaatimus ID	UTR008
Prioriteetti	keskitaso
Skenaario	Raportin tiedot virheiden paikkojen osalta.
Kuvaus	Työkalun tulostamalla testausraportilla näkyy havaittujen virheiden paikat ajetussa yksikkötestissä.

Vaatimus ID	UTR009
Prioriteetti	korkea
Skenaario	Työkalun yksikkötestien komentojen selitykset käyttöohjeissa
Kuvaus	Työkalun nettisivun/foorumin käyttöohjeissa löytyvät selitykset yksikkötesteissä käytettävistä komennoista.

Vaatimus ID	UTR010
Prioriteetti	korkea
Skenaario	Työkalun testausraporttien virheiden selitykset
Kuvaus	Työkalun nettisivun/foorumin käyttöohjeissa löytyvät selitykset testausraportin virheiden tyypeistä.

4 INTEGRAATIOTESTAUS

4.1 Käyttötapaukset

Integraatiotestin kirjoitus	
Käyttötapaus ID	IT001
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan kirjoittaa integraatiotestejä ja käyttää niissä työkalun tarjoamia testikomentoja. Integraatiotestin voi luoda joko työkalulla itsellään tai kirjoittaa sen itse ja käyttää luotavassa testitiedostossa työkalukehyksen tarjoamia komentoja.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Luotavalle integraatiotestille on ennestään luotu sitä ohjaava testitapaus. Integroitavat yksiköt ovat ennestään testattu ja ne ovat toimivia. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta.
Vaiheet	
1.	Avataan työkalu työpöydältä tai komentorivillä.
2.	Valitaan luotavaksi uusi integraatiotestitiedosto.
3.	Aloitetaan testitiedoston kirjoittaminen lisäämällä työkalun tarvitsemat kirjastot.
4.	Kirjoitetaan viittaukset muihin tarvittaviin kirjastoihin ja testattavien funktioiden tiedostoihin.
5.	Kirjoitetaan testattavat funktiot, jotka hyödyntävät molempien yksiköiden funktioita ja käytetään apuna viitteitä näiden funktioiden käyttämiseen.
6.	Kirjoitetaan työkalun käyttämät testitapausfunktiot ja –komennot.
7.	Kirjoitetaan main-funktio, joka sisältää kaikki yksikkötestin suoritukseen tarvittavat ja työkalun tarjoamat komennot. Näitä voivat olla muun muassa testirekisterin aktivointi.
8	Kirjoitetaan integraatiotestin raportointitapa ja raportin nimi, mikäli sitä ei valita erikseen jälkeenpäin.
9.	Kirjoitetaan integraatiotestin suoritustapa, mikäli sitä ei valita erikseen jälkeenpäin.
10.	Tallennetaan integraatiotesti haluttuun kansion valitsemalla tallennusvaihtoehto.
Lopputila	
Integraatiotesti on tallennettu kohdekansioon ja on valmis suoritettavaksi.	

Integraatiotestin suoritus komentorivillä tai graafisella käyttöliittymällä	
Käyttötapaus ID	IT002
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan suorittaa integraatiotestejä joko itse työkalulla graafisen käyttöliittymän kautta tai komentorivillä.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta. Suoritettavalle integraatiotestille on ennestään luotu sitä ohjaava testitapaus. Integroitavat yksiköt ovat ennestään testattu ja ne ovat toimivia. Suoritettava integraatiotesti on luotu työkalulla jo aiemmin.
Vaiheet	
1.	Aukaistaan komentorivi työpöydältä.
2.	Siirrytään kohdekansioon, jossa suoritettava yksikkötesti sijaitsee.
3.	Käännetään testitiedosto ja siinä käytettävät muut tiedostot.
4.	Ajetaan komentorivillä käännetty testitiedosto.
5.	Katsotaan komentoriviltä työkalun integraatiotestin antamat tulokset
tai	
1.	Aukaistaan integraatiotestaustyökalu työpöydältä.
2.	Valitaan työkalun valikosta integraatiotestin avaaminen.
3.	Siirrytään kohdekansioon, jossa suoritettava integraatiotesti sijaitsee.
4.	Avataan suoritettava integraatiotesti.
5.	Valitaan valikosta integraatiotestin suoritus.
6.	Valitaan integraatiotestin suorituksen asetukset.
7.	Käynnistetään integraatiotestin suoritus.
8.	Katsotaan integraatiotestistä saadut tulokset uudesta ikkunasta.
9.	Tallennetaan ja tulostetaan halutessa saadut tulokset raporttiin valikosta.
Lopputila	
Integraatiotestin tulokset näkyvät ruudulla joko graafisella käyttöliittymällä tai komentorivillä. Komentorivillä saatu raportti on jo tulostettu kohdekansioon, mikäli raportin tulostukseen tarvittava komento on lisätty testitiedostoon. Graafisessa käyttöliittymässä raportin tulostamiseen ja tallentamiseen ilmestyy valintavalikko.	

Testiraportin tulostus ja selaus komentorivillä	
Käyttötapaus ID	IT003
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan tulostaa integraatiotesteistä saadut tulosraportit erilliseen tiedostoon. Kyseistä raporttia voi selata jälkeenpäin.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Luotavalle integraatiotestille on ennestään luotu sitä ohjaava testitapaus. Suoritettava integraatiotesti on luotu työkalulla jo aiemmin. Mikäli integraatiotesti on suoritettu komentorivillä, raporttitiedoston luontiin vaadittu komento on lisätty suoritettavaan integraatiotestiin. Mikäli testien tuloksia haluaa selata komentorivin kautta, pitää vaadittu komento olla lisättynä suoritettavaan integraatiotestiin. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta.
Vaiheet	
1.	Aukaistaan komentorivi työpöydältä.
2.	Siirrytään kohdekansioon, jossa suoritettava integraatiotesti sijaitsee.
3.	Käännetään testitiedosto ja siinä käytettävät muut tiedostot.
4.	Ajetaan komentorivillä käännetty testitiedosto.
5.	Katsotaan komentoriviltä työkalun integraatiotestin antamat tulokset
6.	Valitaan komennot haluttujen tietojen katsomiseksi raportista.
7.	Avataan raporttitiedosto kansiosta, jonne sen oli testitiedostossa määrätty luoda.
Lopputila	
Käyttäjälle näkyy raportti tiedostosta ja komentorivillä näkyy käyttäjän valitsemat tiedot testausraportista.	

Testiraportin tulostus ja selaus graafisella käyttöliittymällä	
Käyttötapaus ID	UT004
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan tulostaa yksikkötesteistä saadut tulosraportit erilliseen tiedostoon. Kyseistä raporttia voi selata jälkeenpäin.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Luotavalle integraatiotestille on ennestään luotu sitä ohjaava testitapaus. Suoritettava integraatiotesti on luotu ja suoritettu työkalulla jo aiemmin. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta.
Vaiheet	
1.	Aukaistaan integraatiotestaustyökalu työpöydältä.
2.	Valitaan työkalun valikosta integraatiotestin avaaminen.
3.	Siirrytään kohdekansioon, jossa suoritettava integraatiotesti sijaitsee.
4.	Avataan suoritettava integraatiotesti.
5.	Valitaan valikosta integraatiotestin suoritus.
6.	Valitaan integraatiotestin suorituksen asetukset.
7.	Käynnistetään integraatiotestin suoritus.
8.	Selataan työkalun antamaa ja ruudulla näkyvää raporttia.
8.	Valitaan valikosta raportin tallennus.
9.	Valitaan tiedot, jotka halutaan liittää ja tallentaa luotavaan tulosraporttiin.
10.	Hyväksytään raportin tallennus ja tulostus.
Lopputila	
Käyttäjälle näkyy luotu raporttitiedosto ja käyttäjä voi edelleen selata työkalulla näkyvää raporttia.	

4.2 Vaatimukset

Vaatimus ID	ITR003
Prioriteetti	korkea
Skenaario	Työkalun yhteensopivuus C-kielelle
Kuvaus	Työkalulla voidaan tehdä yksikkötestejä C-ohjelmointikielellä koodatuille sovelluksille.

Vaatimus ID	ITR004
Prioriteetti	korkea
Skenaario	ASSERT-komennot
Kuvaus	Työkalulla on kattava kirjasto komentoja funktioiden testailuun.

Vaatimus ID	ITR005
Prioriteetti	korkea
Skenaario	Komentorivi käyttöliittymänä
Kuvaus	Työkalua voidaan käyttää komentorivin kautta.

Vaatimus ID	ITR007
Prioriteetti	korkea
Skenaario	Testausraportin tulostus näytölle
Kuvaus	Työkalulla voidaan tulostaa suoritetun integraatiotestin tulokset käyttöliittymän ruudulle.

Vaatimus ID	ITR008
Prioriteetti	korkea
Skenaario	Testausraportin tulostus tiedostoon
Kuvaus	Työkalulla voidaan tulostaa suoritetun integraatiotestin tulokset uuteen tiedostoon

Vaatimus ID	ITR009
Prioriteetti	korkea
Skenaario	Raportin tiedot virheiden paikkojen osalta.
Kuvaus	Työkalun tulostamalla testausraportilla näkyy havaittujen virheiden paikat ajetussa integraatiotestissä.

Vaatimus ID	ITR010
Prioriteetti	korkea
Skenaario	Raportin tiedot virheiden paikkojen osalta.
Kuvaus	Työkalun tulostamalla testausraportilla näkyy havaittujen virheiden paikat ajetussa integraatiotestissä.

Vaatimus ID	ITR014
Prioriteetti	korkea
Skenaario	Työkalun integraatiotestien komentojen selitykset käyttöohjeissa
Kuvaus	Työkalun nettisivun/foorumin käyttöohjeissa löytyvät selitykset integraatiotesteissä käytettävistä komennoista.

Vaatimus ID	ITR015
Prioriteetti	korkea
Skenaario	Työkalun testausraporttien virheiden selitykset
Kuvaus	Työkalun nettisivun/foorumin käyttöohjeissa löytyvät selitykset testausraportin virheiden tyypeistä.

5 JÄRJESTELMÄTESTAUS

5.1 Käyttötapaukset

Graafisen käyttöliittymän testaus	
Käyttötapaus ID	ST001
Prioriteetti	korkea
Kuvaus	Työkalulla voidaan testata graafista käyttöliittymää kirjoittamalla skripti valintojen merkitsemiseksi ja/tai nauhoittamalla toiminnot ja suorittamalla uudestaan.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu verkkosivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu verkkosivulta. Työkalu on asennettu koneeseen. Luotavalle systeemitestille on ennestään luotu sitä ohjaava testitapaus. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta. Mikäli käyttöliittymää halutaan testata kirjoittamalla skirpti, täytyy tuntea kohteen rakenne esimerkiksi lähdekoodi ja objektit
Vaiheet	
1.	Avataan työkalu työpöydältä.
2.	Valitaan työkalun valikosta graafisen käyttöliittymän testaava toiminto.
3.	Siirrytään työpöydälle ja avataan testattava sovellus tai ohjelmisto.
4.	Siirrytään takaisin työkaluun ja aloitetaan skriptin teko käyttöliittymän testaukseen.
5.	Nauhoitetaan ennalta määrätty toimintapolku valitsemalla käyttöliittymän objekteja testitapauksen antaman järjestyksen mukaan.
6.	Suoritetaan ja tarkistetaan saatu nauhoitusskripti, jotta se vastaa täsmälleen testitapausta.
7.	Tallennetaan valmiiksi saatu skripti myöhempää suoritusta varten.
tai	
1.	Avataan työkalu työpöydältä.
2.	Valitaan työkalun valikosta graafisen käyttöliittymän testaava toiminto.
3.	Siirrytään työpöydälle ja avataan testattava sovellus tai ohjelmisto.
4.	Avataan testattavan sovelluksen tai ohjelmiston lähdekoodi tai koodiosuus.
5.	Kirjoitetaan skripti ja niissä valittavat objektit sekä toiminnot halutussa järjestyksessä käyttäen apuna lähdekoodia tai koodiosuutta.
6.	Suoritetaan ja tarkistetaan kirjoitettu skripti, jotta se vastaa täsmälleen testitapausta.
7.	Tallennetaan valmiiksi saatu skripti myöhempää suoritusta varten.
Lopputila	
Graafisen käyttöliittymän testaukseen tarvittava skirpti on tallennettu ja valmis myöhempää käyttöä varten.	

Automaattinen testaus	
Käyttötapaus ID	ST002
Prioriteetti	korkea
Kuvaus	Tehdään automaattinen testaus suorittamalla aiemmin luodut yksikkötestit tai graafisen käyttöliittymätestauksen skriptit kerralla. Näin voidaan samalla suorittaa testausta regression varalta.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Testattava sovelluksen tiedosto löytyy koneelta. Luotavalle systeemitestille on ennestään luotu sitä ohjaava testitapaus. Tietokoneelle on tallennettu aiempia yksikkötestejä tai suoritettavia skriptejä esimerkiksi graafisen käyttöliittymän testausta varten. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta. Mikäli automaattista testausta on suoritettu yksikkötesteillä tai skripteillä aiemmin ja testaus on raportoitu, voidaan suorittaa regressiotestausta.
Vaiheet	
1.	Avataan työkalu työpöydältä.
2.	Valitaan työkalun valikosta testien suorittaminen automaattisesti.
3.	Valitaan suoritettavat yksikkötestit tai graafisen käyttöliittymätestauksen skriptit.
4.	Käynnistetään testien automaattinen suoritus.
5.	Käydään läpi saadut testien tulokset ja verrataan niitä aiemmin saatuihin tuloksiin, mikäli testausta on suoritettu aiemmin.
6.	Raportoidaan ja tallennetaan saadut tulokset.
Lopputila	
Automaatiotestaus on suoritettu ja saadut tulokset raportoitu.	

Muistin vapautuksen testaus	
Käyttötapaus ID	ST003
Prioriteetti	korkea
Kuvaus	Suoritetaan työkalulla testaus sovelluksen muistin varauksesta ja vapautuksesta. Näin voidaan testata sovelluksen tehokkuutta ja muistivuotoja.
Käyttäjä	Kurssin pitäjä, opiskelija
Alkutila	Työkalun ominaisuudet ovat tarkistettu tuotesivulta. Työkaluun integroitavien työkalujen ominaisuudet on tarkistettu tuotesivulta. Työkalu on asennettu koneeseen. Testattava sovelluksen tiedosto löytyy koneelta. Luotavalle systeemitestille on ennestään luotu sitä ohjaava testitapaus. Tietokoneelle on tallennettu aiempia yksikkötestejä tai automaattisesti suoritettavia skriptejä esimerkiksi graafisen käyttöliittymän testausta varten. Työkalun käyttöohjeet ovat saatavilla painetusta teoksesta tai nettisivulta.
Vaiheet	
1.	Avataan työkalu työpöydältä.
2.	Valitaan työkalun valikosta toiminto muistivuotojen tarkistukseen.
3.	Valitaan testattava kohde tiedostoista.
4.	Käynnistetään ja suoritetaan kohde samalla kun muistinvuototyökalu toimii taustalla.
5.	Suoritaan testitapauksessa ennalta määrätyt toiminnot kohdesovelluksessa. Mikäli kohdesovelluksen toimintojen testaukselle on luotu aiempia skriptejä, voidaan käyttää niitä.
6.	Tarkistetaan ja arvioidaan saatuja tuloksia.
7.	Raportoidaan ja tallennetaan saadut tulokset
Lopputila	
Tulokset työkalun havaitsemista muistivuodoista ovat raportoitu ja tallennettu.	

5.2 Vaatimukset

Vaatimus ID	STR001
Prioriteetti	korkea
Skenaario	Työkalun yhteensopivuus C-kielelle
Kuvaus	Työkalulla voidaan tehdä yksikkötestejä C-ohjelmointikielellä koodatuille sovelluksille.

Vaatimus ID	STR002
Prioriteetti	korkea
Skenaario	Kurssille tarvittavat testaustyypit
Kuvaus	Työkalulla on kattava kirjasto erilaisia testaustyyppejä, jotka täsmäävät kurssin sisällön kanssa.

Vaatimus ID	STR003
Prioriteetti	korkea
Skenaario	Testausraportin tulostus näytölle
Kuvaus	Työkalulla voidaan tulostaa suoritetun systeemitestin tulokset käyttöliittymän ruudulle.

Vaatimus ID	STR004
Prioriteetti	korkea
Skenaario	Testausraportin tallennus tiedostoon
Kuvaus	Työkalulla voidaan tallentaa suoritetun systeemitestin tulokset uuteen tiedostoon

Vaatimus ID	STR005
Prioriteetti	korkea
Skenaario	Raportin tiedot virheiden paikkojen osalta.
Kuvaus	Työkalun tulostamalla testausraportilla näkyy havaittujen virheiden paikat suoritetussa systeemitestissä.

Vaatimus ID	STR006
Prioriteetti	korkea
Skenaario	Raportin tiedot virheiden paikkojen osalta.
Kuvaus	Työkalun tulostamalla testausraportilla näkyy havaittujen virheiden paikat ajetussa integraatiotestissä.

Vaatimus ID	STR007
Prioriteetti	korkea
Skenaario	Työkalun testausraporttien virheiden selitykset
Kuvaus	Työkalun nettisivun/foorumin käyttöohjeissa löytyvät selitykset testausraportin virheiden tyypeistä.

Vaatimus ID	STR008
Prioriteetti	keskitaso
Skenaario	Graafisen käyttöliittymän testaus
Kuvaus	Työkalulla on mahdollista suorittaa graafisen käyttöliittymän testausta.

Vaatimus ID	STR009
Prioriteetti	korkea
Skenaario	Automaattinen testaus
Kuvaus	Työkalulla on mahdollista suorittaa automaattista testausta.

Vaatimus ID	STR010
Prioriteetti	keskitaso
Skenaario	Toimintakyvyn testaus
Kuvaus	Työkalulla on mahdollista suorittaa sovelluksen toimintakyvyn testausta

Vaatimus ID	STR011
Prioriteetti	keskitaso
Skenaario	Automaatiotestausskriptin kirjoittamisen helppous
Kuvaus	Työkalulla suoritettavaan automaatiotestaukseen tarvittavan skriptin kirjoittaminen/tuottaminen on helppoa.

LIITE 2: TYÖKALUJEN KÄYTTÖOHJEET

SISÄLLYSLUETTELO

Työkalujen käyttöohjeet –liitteen kasaus, korjaus ja taitto:

Joonas Maksimainen

Alkuperäinen englanninkielinen CUnit -dokumentointi:

Jerry St. Clair

Alkuperäinen englanninkielinen Valgrind -dokumentointi:

Valgrind™ Developers

Alkuperäinen englanninkielinen Ubuntu -dokumentointi:

Ubuntu Documentation Project

Toteutuksen ohjaus sekä tarkastus:

Uolevi Nikula

Timo Hynninen

Lappeenrannan teknillinen yliopisto, LUT School of Engineering Science

Lappeenranta 14.12.2018

1 TERMILUETTELO

- CSV eli *comma-separated-values* = Tiedostomuoto, jolla tallennetaan taulukkomuotoista tietoa tekstitiedostoon.
- testialue = Tarantula-työkalulla merkittävä testaustyyppi tai testauksen muoto, jolla voidaan luokitella ja jakaa testausprojekteja ja niissä työskenteleviä työntekijöitä haluttuun toimintaan.
- testitapaus eli *test case* = spesifikaatio syötteitä, suoritustiloja, testausprosessia ja odotettuja tuloksia, joilla on tarkoitus määritellä yksittäisen testin suoritus ja näin saavuttaa tietty ohjelmistotestauksen tavoite.
- testijono eli *test suite* = kokoelma testitapauksia, joilla on tarkoitus testata tiettyä sovelluksen osaa tai toimintoa.
- testivaatimus eli *test requirement* = Vaatimus, jonka pohjalta testitapaus tai -tapaukset määritellään. Testivaatimuksen tulee olla yksityiskohtainen.
- priorisointi = Asioiden asettamista tärkeysjärjestykseen, mikä testauksessa tarkoittaa keskittymistä tärkeämmäksi tulkittujen sovelluksen osien kattavampaan testaamiseen.
- testisuoritus = Tarantula-työkalun toiminto, jolla on tarkoitus suorittaa valitut testitapaukset käymällä tapausten vaiheet läpi yksitellen ja merkkaamalla virheelliset vaiheet.
- testiobjekti = Tarantula-työkalun toiminto, jolla voi erotella eri testiskenaarioita toisistaan.

2 TARANTULA

2.1 Käyttäjän luominen

1. Aukaise Tarantulan nettisivu
2. Kirjaudu sisään syöttämällä ylläpitäjän (admin) käyttäjätunnus, salasana ja paina ”Login”.
3. Ryhmien luomiseksi täytyy aluksi luoda käyttäjiä. Käyttäjän luomiseksi valitse toimintovaihtoehtoista ”Admin” ja alavalikosta ”Users”.
4. Jokaisessa toiminnossa on sivulla ”Explorer”-palkki, josta voi selailla jo luotuja kohteita, mitkä ”Admin”-vaihtoehdossa ovat jo luodut projektit ja käyttäjät. Näitä voi muokata klikkaamalla kohdetta ja valitsemalla ”Edit”.
5. Tarantulalla ei ole omaa massaimportointiominaisuutta käyttäjien luomisessa, jonka takia käyttäjien luonti tulee tehdä joko manuaalisesti tai ulkopuolista työkalua käyttäen. Ulkopuolista työkalua käytettäessä luodaan skripti toimintojen suorittamiseksi verkkosivulla automaattisesti ja henkilötietoja syöttämiseksi käytetään csv-tiedostoa, josta luetaan syötettävät tiedot. Työkaluvaihtoehtoja tietojen syöttämiseen automaattisesti ovat muun muassa ilmainen Selenium IDE ja siitä kehityt uudemmat versiot. Näillä voidaan automatisoida toistuvia suorituksia verkkosivulla sekä lukea ja syöttää tietoja csv-tiedostoista.
6. Luodaan uusi käyttäjä valitsemalla ”New”-vaihtoehto, jonka jälkeen syötetään käyttäjän tiedot.
    a. Selkeyden vuoksi käyttäjätunnukseksi ”Login”-kohtaan laitetaan opiskelijoilla heidän opiskelijanumero.
    b. ”Realname”-kohtaan laitetaan käyttäjän nimi esim. Olli Opiskelija.
    c. ”Email”-kohtaan uuden käyttäjän LUT:n sähköpostiosoite.
    d. Puhelinnumero ei ole pakollinen.
    e. Ylläpitäjää luodessa tulee ”Admin”-kohta merkitä, jotta he saavat ylläpitäjän oikeudet kaikkiin työkalun toimintoihin.
    f. Lopuksi tulee syöttää ja varmistaa uuden käyttäjän salasana.
7. Tallenna uusi käyttäjä valitsemalla ”Save”-kohta. Tallennuksen jälkeen uuden käyttäjän sähköpostiin tulee ilmoitus tilistä Tarantulassa.
8. Tallennetut käyttäjät voidaan poistaa etsimällä ne ”Explorer”-palkista ”Users”-kohdasta ja valitsemalla ”Delete”. Tällöin poistettu käyttäjä siirtyy ”Explore”-kohdassa ”Deleted”-kansioon, josta sen voi edelleen palauttaa. Täydellistä puhdistusta ei voi tehdä, sillä käyttäjät ovat usein linkitettyinä luomiinsa testitapauksiin ja tämä voi aiheuttaa käyttäjän poistaessa ongelmia. Käyttäjän nimiä ja sähköpostia voi kuitenkin muuttaa.

2.2 Projektin luominen ja oikeudet

1. Työkalulla ei voi luoda varsinaisia ryhmiä, mutta projekteja voi luoda, jolloin voidaan estää töiden kopioimisen mahdollisuus opiskelijoiden välillä. Käytetään projektin luontia käyttäjien jakamisessa ryhmiin, siirtämällä käyttäjät valikoituihin projekteihin.
2. Valitse ”Admin”-vaihtoehto, josta ”Projects”, jolloin siirrytään näkymään projektien luomiseksi.
3. Sivulla olevasta ”Explorer”-palkista voidaan muokata projekteja ja käyttäjiä valitsemalla kohde ja seuraavaksi ”Edit”.
4. Uusi projekti luodaan valitsemalla ”New”, jonka jälkeen syötetään tietoja projektin, mikä tässä tapauksessa tarkoittaa ryhmää.
    a. Kirjoitetaan projektin nimi.
    b. Kirjoitetaan projektin kuvaus, joka voisi sisältää esimerkiksi tiedot, mitä projektissa tehdään tai mitä käyttäjiä projektissa on.
    c. Merkataan projektin käyttö testitapaus kirjastona, jolloin projektin käyttäjien tapaukset tallentuvat tähän projektiin.
    d. Merkataan mitä testialuetta projekti koskee. Esimerkiksi ohjelman osaa tai testaustyyppiä.
    e. Valitaan bugitietokanta, mikäli sellainen on käytössä.
5. Valitaan tuotteet tai tuote, jota projekti käsittelee.
6. Valitaan projektille käyttäjät, mikä tässä tapauksessa tarkoittaa ryhmän jäseniä, valitsemalla ”New User”.
    a. Valitaan ”Login”-kohdasta käyttäjä.
    b. Valitaan käyttäjän rooli.
        i. Test engineer: Käyttäjällä on oikeus vain testien suorittamiseen.
            1. Dashboard
            2. Test
        ii. Test designer: Voi luoda testitapauksia, -jonoja, -objekteja ja suorittaa testiajoja. Tämä valinta on opiskelijalle paras.
            1. Dashboard
            2. Design
            3. Test
            4. Report
        iii. Manager: Käyttöoikeudet kaikkiin toimintoihin paitsi ylläpitoon.
        iv. Manager view only: Pystyy vain katsomaan projektia, muttei muokkaamaan sitä.
    c. Valitaan halutessa pakotettu testialue, johon käyttäjä keskittyy.
7. Tallenna projekti valitsemalla ”Save”. Projekteja voi poistaa etsimällä ne ”Explorer”-palkista ”Projects”-kohdasta ja valitsemalla ”Delete”. Tällöin poistettu projekti siirtyy ”Explore”-kohdassa ”Deleted”-kansioon, josta sen voi edelleen palauttaa. Täydellistä poistoa ei voi tehdä, sillä projektit ovat usein linkitettyinä niihin luotuihin testitapauksiin. Projektin voi kuitenkin tyhjentää resursseista valitsemalla ”Purge”-vaihtoehdot.

2.3 Testijonon luominen

1. Valitse ”Design” ja siitä ”Sets” testijonon luomiseksi.
2. Valitse ”New” uuden testijono tekemiseksi. Vanhoja testijonoja voi muokata selaamalla niitä ”Explorer” palkista ja valitsemalla ”Edit”.
3. Nimeä testijono. Varmista että nimet ovat merkityksellisiä ja selviä, jotta testijonot pysyvät järjestyksessä ja niitä on helpompi selata. Käytä numerointia, jos testijono on osa suurempaa segmenttiä esimerkiksi Harjoitustestijono1.
4. Valitse päivämäärä ja prioriteetti testijonolle. Tärkeys riippuu testijonon kohteesta, sillä prioriteetti testijonolle tai tapaukselle, jonka kohteena on esim. yrityslogiikka tai moduulin keskeinen toiminnallisuus on korkea tai normaali. Prioriteetti testijonolle sisältäen pieniä tapauksia, esimerkiksi visuaaliseen käyttöliittymään liittyen, on matala.
5. Kirjoita ja/tai valitse merkkilappu ”Explorer”-valikosta ”Sets”-kohdasta, joka viittaa luotavaan testijonoon. Tämä helpottaa testijonon löytämistä myöhemmin, esim. keskeneräisen testijonon.
6. Valitse ”Cases”-vaihtoehto ”Explorer”-valikosta, jonka jälkeen vedä ja pudota haluamasi testitapaukset luomaasi testijonoon. Valitse ne testitapaukset, jotka testaavat samaa moduulia tai koodia järjestelmässä. Tämä helpottaa testijonojen ja -tapausten selailua jälkeenpäin.
7. Tallenna luomasi testijono painamalla ”Save”.

2.4 Testivaatimuksen luominen

1. Paina ”Design” ja valitse ”Requirements”.
2. Paina ”New” luodaksesi uuden testivaatimuksen. Voit muokata vanhempia testivaatimuksia selaamalla niitä ”Explorer” valikosta ja valitsemalla ”Edit”.
3. Nimeä testivaatimus ja valitse päivämäärä. Varmista että nimet ovat merkityksellisiä ja selviä, jotta testivaatimukset pysyvät järjestyksessä ja niitä on helpompi selata.
4. Kirjoita testivaatimuksen tunnus. Tunnus voi koostua lyhenteestä liittyen tiettyyn moduuliin tai toiminnallisuuteen, jonka jälkeen tulee numero. Esimerkiksi HTV001 liittyen vaatimukseen Harjoitustestivaatimus1.
5. Valitse prioriteetti testivaatimukselle. Tärkeys riippuu testivaatimuksen kohteesta, sillä prioriteetti testivaatimukselle, jonka kohteena on esim. yrityslogiikka tai moduulin keskeinen toiminnallisuus on korkea tai normaali. Prioriteetti testivaatimukselle pieneen toimintoon, esimerkiksi visuaaliseen käyttöliittymään liittyen, on matala.
6. Kirjoita ja/tai valitse merkkilappu ”Explorer”-valikosta ”Cases”-kohdasta, joka viittaa luotavaan testivaatimukseen. Tämä helpottaa vaatimuksen selaamista ja muokkaamista myöhemmin, esim. vaatimus on merkitty liittyvän tiettyyn ryhmään tai testitapaukseen.
7. Valitse mahdollinen testialue, mitä kyseinen testivaatimus edustaa.
8. Kirjoita kuvaus testivaatimuksesta, jossa selitään tarkemmin vaatimus ja siihen liittyviä tietoja.
9. Lisää testivaatimuksen liitteeksi tiedosto esim. dokumentti liittyen testivaatimukseen.

10. Valitse ”Cases”-vaihtoehto ”Explorer”-valikosta, jonka jälkeen vedä ja pudota haluamasi testitapaukset, jotka liittyvät luomaasi testivaatimukseen. Testitapausten lisääminen voidaan tehdä jälkeenpäin vaatimusta muokkaamalla, mikä helpottaa kokonaisuuksien hahmottamista testaussuunnitelmia tehdessä.
11. Tallenna luotu testivaatimus valitsemalla “Save”.

2.5 Testitapauksen luominen

1. Paina “Design” ja valitse “Cases”-vaihtoehto.
2. Paina “New” luodaksesi uuden testitapauksen. Voit muokata vanhempia testitapauksia selaamalla niitä ”Explorer” valikosta ja valitsemalla ”Edit”.
3. Nimeä testitapaus. Varmista, että nimet ovat merkityksellisiä ja selviä, jotta testitapaukset pysyvät järjestyksessä ja niitä on helpompi selata. Käytä numerointia, jos testitapaus on osa suurempaa testijonoa esimerkiksi Harjoitustestitapaus1.
4. Valitse päivämäärä ja prioriteetti testitapaukselle. Tärkeys riippuu testitapauksen kohteesta, sillä prioriteetti testitapaukselle, jonka kohteena on esim. yrityslogiikka tai moduulin keskeinen toiminnallisuus on korkea tai normaali. Prioriteetti testitapaukselle pieneen toimintoon, esimerkiksi visuaaliseen käyttöliittymään liittyen, on matala.
5. Kirjoita testitapaukselle suunniteltu aikataulu, joka menee testitapauksen suorittamiseen. Tämä riippuu testitapauksen monimutkaisuudesta ja mahdollisista virheistä, joita testitapauksen suorittaminen tuo tullessaan.
6. Kirjoita ja/tai valitse merkkilappu ”Explorer”-valikosta ”Cases”-kohdasta, joka viittaa luotavaan testitapaukseen. Tämä helpottaa testitapausten selaamista ja muokkaamista myöhemmin, esim. testitapaus on merkitty liittyvän tiettyyn testijonoon tai testattavan ohjelman osaan.
7. Kuvaile testitapauksen tavoite. Mitä tällä testitapauksella on tarkoitus saavuttaa. Mitä tiettyä moduulia tai toimintoa testitapauksen on tarkoitus testata esim. palauttaako funktio tai moduuli oikeat arvot oikealla syötteellä.
8. Kirjoita testidata, jota aiot käyttää syötteenä testitapauksessa.
9. Kirjoita, mitkä ovat testitapauksen esivaatimukset ja oletukset. Mitä ehtoja tulee täyttää, ennen kuin testitapauksen voi suorittaa ja mitä tulee tapahtumaan, kun tapaus suoritetaan?
10. Aina kun testitapauksiin tehdään muutoksia, vaatii työkalu kommenttia muutoksesta, jotta niistä voidaan pitää kirjaa mm. regression varalta.
11. Lisää testitapauksen liitteeksi tiedosto tai kuvankaappaus ajetusta testausskriptin tuloksista, joka on ajettu ja laadittu CUnitilla kyseisen testitapauksen mukaan. Näin voidaan varmistaa, että testattava kohde on varmasti kunnolla testattu eikä vain ”katsottu” läpi.
12. Kirjoita testitapauksen vaiheet, joissa kuvataan testitapauksen eteneminen tarkasti alusta loppuun. Kirjoita aluksi toiminto ja toiminnon lopputulos. Esim. testatessa tekstin lisäämistä tiedostoon pitää ensiksi selvittää tiedoston olemassaolo.
13. Lisää testitapaukseen liittyvät vaatimukset vetämällä niitä ”Explorer”-valikosta kohdasta ”Requirements”.
14. Tallenna luotu testitapaus valitsemalla “Save”.

2.6 Testisuorituksen luominen

1. Paina ”Design” ja valitse ”Executions”-vaihtoehto.
2. Paina ”New” luodaksesi uuden testiajon. Voit muokata vanhempia testiajoja selaamalla niitä ”Explorer”-valikosta ja valitsemalla ”Edit”.
3. Syötä ja lisää testiajoa koskevat tiedot.
    a. Kirjoita testisuorituksen nimi.
    b. Kirjoita päivämäärä.
    c. Valitse testiobjekti, jota testiajo koskee.
    d. Merkitse onko testiajo jo suoritettu, jolloin se poistuu käyttäjille osoitetuista tehtävistä osiossa ”My Tasks” ja testilistoista. Tämän voi tehdä jälkeenpäin, kun testiajo on suoritettu ja se on täysin onnistunut.
    e. Merkitse mihin testialueeseen testiajo sijoittuu.
    f. Kirjoita ja/tai valitse merkkilappu ”Explorer”-valikosta ”Executions”-kohdasta, joka viittaa luotavaan testiajoon. Tämä helpottaa testiajon löytämistä myöhemmin esim. suoritetun testiajon merkkaamiseen.
    g. Testiajosta voi laskea halutessaan raportin siihen kulutetusta työmäärästä.
4. Valitse ”Explorer”-valikosta testitapauksia, jonka jälkeen vedä ja pudota haluamasi testitapaukset luomaasi testiajoon. Valitse ne testitapaukset, jotka testaavat samaa moduulia tai koodia järjestelmässä. Voit valita yksittäisiä testitapauksia ”Cases”-kohdasta tai useampia testitapauksia sisältäviä testijonoja ”Sets”-osiosta. Laittaessasi testitapauksia voit merkata niitä ryhmän eri käyttäjien vastuulle.
5. Tallenna luotu testiajo valitsemalla ”Save”.

2.7 Testiobjektin luominen

1. Paina “Design” ja valitse “Test Objects”-vaihtoehto. Testiobjektilla on tarkoitus erotella testien suoritukset toisistaan. Esimerkiksi, jos kyseessä on uusi sovellusversio SW4 ja haluat suorittaa sille erilaisia testiskenaarioita, niin testiobjektilla pystyy erottamaan nämä testiskenaariot vanhempien versioiden skenaarioista.
2. Paina “New” luodaksesi uuden testiobjektin. Voit muokata vanhempia testiobjekteja selaamalla niitä ”Explorer” valikosta ja valitsemalla ”Edit”.
3. Syötä ja lisää testiobjektia koskevat tiedot ja liitteet. Pakollisten ohella muilla ei ole merkitystä kuin dokumentoinnin kannalta ja jos kyseisillä tekijöillä on suurikin vaikutus esim. testien ajamiseen.
    a. Kirjoita testiobjektin nimi.
    b. Kirjoita päivämäärä.
    c. Kirjoita, mikä on testiobjektin sulautettu ohjelmisto, mikäli systeemillä tai moduulilla on sellainen.
    d. Kirjoita, mikä on testiobjektin semanttinen web-applikaatio, mikäli systeemillä tai moduulilla on sellainen.
    e. Kirjoita, millä teknisellä alustalla systeemi tai moduuli toimii.
    f. Kirjoita millä mekaniikalla testiobjekti toimii.
    g. Kirjoita lyhyt kuvaus testiobjektista.
    h. Kirjoita ja/tai valitse merkkilappu ”Explorer”-valikosta ”Test Objects”-kohdasta, joka viittaa luotavaan testiobjektiin. Tämä helpottaa testiobjektin löytämistä myöhemmin, esim. keskeneräisen testiobjektin.
    i. Valitse mitä testialuetta testiobjekti käsittelee.
    j. Lisää testiobjektiin viittaavia liitteitä, mikäli niillä on suuri merkitys, esim. erillistä dokumentointia testiobjektista testien suorittamisen kannalta.
4. Tallenna testiobjekti valitsemalla ”Save”.

2.8 Testisuoritusten ajaminen

1. Valitaan kohta ”Test” ja josta edelleen ”Test”.
2. Valitaan ”Explorer”-valikosta ”Test”-kohdasta haluttu testisuoritus. Nimen vieressä näkyy miten pitkälle suoritus on edennyt.
3. Valitaan testitapaus ”Explorer”-valikosta ja käydään sen jokainen vaihe yksitellen läpi merkitsemällä vaiheen onnistuminen.
    a. Valitaan testitapauksen vaihe joko valitsemalla se ikkunasta suoraan tai klikkaamalla valintanuolia.
    b. Merkitään vaiheen onnistuminen valitsemalla:
        i. ”Pass”: Vaihe onnistui.
        ii. ”Fail”: vaihe epäonnistui.
        iii. ”Skip”: vaihe ohitettiin.
        iv. ”Not Implemented”: vaihetta ei ole toteutettu.
    c. Merkitään vaiheeseen liittyvä virhe käyttäen apuna bugitietokantaa valitsemalla ”Defect”, jonka jälkeen valitaan bugitietokannasta vaiheen epäonnistumisen aiheuttama virhe.
    d. Merkitään kommentti vaiheeseen liittyen valitsemalla ”Comment”, jonka jälkeen kirjoitetaan kommentti ja valitaan ”Ok”.
    e. Siirrytään seuraavaan vaiheeseen valitsemalla vaihe ikkunasta tai valitsemalla ”Step”-kohdasta ”Next”.
    f. Käydään loput testitapauksen vaiheet läpi.
4. Siirrytään seuraavaan testitapaukseen valitsemalla se joko ”Explorer”-valikosta tai valitsemalla ”Case”-kohdasta ”Next”. Toistetaan prosessi vaiheiden läpikäymisessä kunnes kaikki testitapaukset on käyty läpi.
5. Poistetaan läpikäyty testisuoritus ”Test”-valikosta.
    a. Valitaan ”Design”-valikosta kohta ”Executions”.
    b. Valitaan ”Explorer”-valikosta haluttu testisuoritus ja valitaan ”Edit”.
    c. Merkitään kohta ”Compeleted”, jolloin testisuoritus poistuu testikohdasta.
    d. Tallennetaan muutos valitsemalla ”Save”.

3 CUNIT

3.1 Testausskriptin kirjoittaminen ja ajaminen

1. Lisätään CUnitin kirjastot.

```
#include <CUnit/Cunit.h>
#include <CUnit/Basic.h>
#include <CUnit/Automated.h>
#include <CUnit/Console.h>
#include <CUnit/CUCurses.h> /* Käytetään jos systeemissä on curses-toiminto
*/
```

2. Kirjoitetaan viittaukset kirjastoihin joita aiotaan käyttää.

```
// Header tiedostoon josta otetaan testattava funktio
#include  ”max.h”
```

3. Kirjoitetaan testisarjojen aloitus eli ”init_suite()” - ja puhdistusfunktiot eli ”clean_suite()”.

```
//Luodaan testisarjat
int init_suite(void) { return 0;}
int clean_suite(void) { return 0;}
```

4. Kirjoitetaan testattavat funktiot. Funktion voi ottaa käyttöön myös toisesta tiedostosta, mutta silloin tarvitaan header-tiedostoa.

```
//Katsotaan onko annettu luku parillinen vai pariton. Palauttaa arvon 1 eli ”true”
//jos annettu luku on parillinen ja jakojäännöstä ei synny.
int is_even (int x)
{
	return (x % 2 = = 0);
}
```

5. Kirjoitetaan testitapausfunktiot, joilla testataan halutun kohdefunktion toimivuus.

```
//CU_ASSERT tarkistaa onko sen saama syöte tosi.
void test_is_even (void)
{
	CU_ASSERT(is_even(1) = = 0);
	CU_ASSERT(is_even(2) = = 1);
	CU_ASSERT(is_even(3) = = 0);
	CU_ASSERT(is_even(4) = = 0);
}
```

6. Laaditaan testien suoritusosuus, jossa käynnistetään testirekisterit, lisätään testisarjat rekisteriin ja testitapaukset testisarjoihin. Tämän jälkeen suoritetaan testien suorittaminen tietyn käyttöliittymän mukaan ja lopuksi puhdistetaan testirekisteri.

```
int main (void)
{
	CU_suite pSuite = NULL;

	// Käynnistetään CUnitin testirekisteri
	if (CE_SUCCESS != CU_initialize_registry())
		return CU_get_error();

	// Lisätään testisarja rekisteriiin
```

```
	pSuite = CU_add_suite("Basic_Test_Suite", init_suite, clean_suite);
	if (NULL = = pSuite) {
	CU_cleanup_registry();
	return CU_get_error();
	}

	// Lisätään testitapaus testisarjaan
	if ((NULL = = CU_add_test(pSuite, "test_is_even", test_is_even)))
	{
		CU_cleanup_registry();
		return CU_get_error();
	}

	// Tässä kohtaa voi lisätä uudet testisarjat ja tapaukset.

	// Suorita testit käyttäen peruskäyttöliittymää, mikä tässä tapauksessa on
	// komentorivi
	CU_basic_run_tests();

	// Suorita testit käyttäen automaattista käyttöliittymää, mikä tarkoittaa
	// tulosten viemistä XML-tiedostoon. Tiedoston voi nimetä.
	CU_set_output_filename("Test_results");
	CU_automated_run_tests();

	// Suorita testit interaktiivisessa konsolimoodissa komentorivillä
	CU_console_run_tests();
	// Suorita testit interaktiivisessa Curses-moodissa.
	// CU_curses_run_tests();
	// Puhdista rekisteri ja palataan
	CU_cleanup_registry();
	return CU_get_error();
}
```

7. Tallennetaan tiedosto kohdekansioon.
8. Mikäli lncurses-paketti puuttuu, asennetaan se komennolla
    a. *$ sudo apt-get install libncurses5-dev*
9. Aukaistaan komentorivi ja siirrytään kohdekansioon.
    a. Komennolla "*$ cd ~/Kohdekansio*" päästään suoraan oletettuun kansioon.
10. Käännetään testitiedosto ja siinä käytettävät muut tiedostot
    a. *$ gcc testitiedosto.c max.c –o testitiedosto –lcunit –lcurses*
11. Ajetaan käännetty testitiedosto
    a. *$ ./testitiedosto*
12. Katsotaan tulokset komentoriviltä ja virhekohdat.
13. Mikäli testauksessa on käytetty komentoa *CU_console_run_tests()*, voi testijonoja käydä ja ajaa läpi yksitellen sekä tulostaa tulokset uudestaan.
14. Voidaan katsoa XML-tiedosto läpi samasta kansiosta. Avautuu automaattisesti nettiselainnäkymään, mutta näkyy paremmin gedit-toimintoa käytettäessä. XML-tiedosto voidaan tallentaa jälkeenpäin hallintatyökaluun.

4 VALGRIND

4.1 Valgrindin käyttö

1. Avataan komentorivi ja siirrytään tiedoston sisältävään kansioon.
2. Asennetaan Valgrind
    a. *$ sudo apt-get instal valgrind*
3. Käännetään ohjelma ”main.c” debuggaus merkeillä
    a. *$ gcc –o suoritettava –std=c11 –Wall –ggdb3 main.c*
4. Käynnistetään Valgrind komentorivillä. Tulokset voi saada tekstitiedostoon halutessaan käyttämällä komentoa ”--log-file=tiedostonimi.txt”
    a. *valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes --verbose --log-file=valgrind-out.txt ./suoritettava*
5. Mikäli Ubuntu ei anna ohjata Valgrindia tiedostoon, pitää suoritettavan osuuden oikeuksia muuttaa.
    a. *chmod a+x ./suoritettava*
6. Tarkistetaan tulokset komentoriviltä.
7. Tarkista virheilmoitukset tarvittaessa sivulta
   http://valgrind.org/docs/manual/mc-manual.html#mc-manual.errormsgs
   http://valgrind.org/docs/manual/manual.html

5 UBUNTUN KOMENTOJEN HYÖDYNTÄMINEN

5.1 Sovelluksen syötteet tekstitiedostoa hyväksikäyttäen.

1. Luodaan syötteet sisältävä tekstitiedosto eli esimerkiksi *”luvut.txt”*. Lukujen järjestyksessä ja syötteissä tulee olla tarkkana. Mikäli sovelluksen toiminnat valitaan numeroita syöttämällä, voi tiedoston sisältö olla esim. näin:

```
1
52
23
2
6
45
0
```

   (Laskintiedostossa testattiin aluksi yhteenlasku (valinta 1) ”52 + 23” ja vähennyslasku (valinta 2) ”6 – 45” ja lopulta sovelluksen sulku (valinta 0))
2. Siirrytään ohjelman sisältävään kansioon.
   ”*$ cd ~/Kohdekansio*”
3. Käännetään suoritettava sovellus ja tiedostot joita sovellus hyödyntää:
   ”*$ gcc –Wall HeadCalculator.c BasicCalculator.c CompareCalculator.c CompareCalculator2.c –o laskin*
4. Suoritetaan sovellus käyttäen tekstitiedoston syötteitä:
   ”*$ ./laskin < luvut.txt”*
   Mikäli halutaan saada komentorivin tulosteet tekstitiedostoon esim. *”results.txt”*, käytetään komentoa:
   ”*$ ./laskin < luvut.txt > results.txt”*

5.2 Komentorivin syötteet tekstitiedostoa hyväksikäyttäen.

1. Luodaan komentorivikomennot sisältävä tekstitiedosto eli tässä tapauksessa *”testiauto.txt”*. Komentojen järjestyksessä pitää muistaa olla tarkkana ja ensimmäisen komennon tulee olla tietynlainen. Tiedoston sisältö voi olla alla olevalla tavalla, kun suoritetaan aluksi yksikkötestin kääntö, suoritus sekä lopuksi edellinen käyttöliittymän läpikäynti.

```
#!/bin/sh
gcc –Wall BasicCalculatorTester.c BasicCalculator.c –o perustestaus -lcunit
./perustestaus
gcc –Wall HeadCalculator.c BasicCalculator.c CompareCalculator.c
CompareCalculator2.c –o laskin
./laskin < luvut.txt > results.txt
```

2. Siirrytään ohjelman ja tekstitiedoston sisältävään kansioon. Muistetaan varmistaa, että kaikki tarvittavat tiedostot ovat samassa kansiossa.
   ”*$ cd ~/Kohdekansio*”
3. Tehdään tekstitiedostosta suoritettava komennolla “chmod +x”.
   ”$ chmod +x testiauto.txt”
   Mikäli tiedosto sijaitsee järjestelmän kansioissa, joudutaan käyttämään ”sudo”-komentoa.
   ”$ sudo chmod +x testiauto.txt”
4. Suoritetaan tekstitiedosto ja sen sisältämät komennot
   ”*$ ./testiauto.txt”*

LIITE 3: KURSSITEHTÄVÄIDEAT

SISÄLLYSLUETTELO

1 KURSSITEHTÄVÄEHDOTUKSIA

1.1 Tehtävä 1. Testauksen hallinta

Tarantulan käytön opettelu: halutaan tallentaa tiedot testauksesta, jotta työnantaja/asiakas näkee tulokset ja todistetaan funktioiden toimivuus.

Testattavina kohteina ovat 3-4 pientä c-koodin ohjelmaa sisältäen testattavia funktioita. Käytetään tarvittaessa aikaisempia opiskelijoiden tekemiä CUnit tiedostoja, mikäli on vähemmän aikaa tai vaatii liikaa opiskelijalta. Tämä voi muuttaa testauksen luonnetta, eli halutaanko vain merkitä muistiin testien tulokset luomalla testitapaukset Tarantulaan jälkikäteen vai tehdä suunnittelua luomalla testitapaukset ennen testien kirjoittamista ja muokataan niitä jälkeenpäin lisäämällä niihin tulokset sisältävät XML-tiedostot.
Testauksien tuloksista voidaan luoda XML-tiedostot CUnitilla joko etukäteen, jolloin opetellaan enimmäkseen raportointia tai sitten opiskelijat suorittavat testaukset itse ja tallentavat tulokset jälkikäteen Tarantulaan.

- Valmiiksi luodut tiedostot voivat sisältää virheitä, jolloin nämä joudutaan merkitsemään myös Tarantulaan ajaessa testisuorituksia.

Tehtävä voi edetä seuraavien vaiheiden kautta:

1. Testien luominen CUnitilla ja tulokset

- Kirjoitetaan CUnitilla testit, joissa testataan kohteena olevien koodien funktioiden toimivuus.
- Käytetään header-viittauksia, jolloin testattavia funktioita voi hyödyntää suoraan
- Kirjoitetaan testausskripti siten, että testi palauttaa tuloksista XML-tiedoston. Tämä tiedosto voidaan liittää testitapauksiin Tarantulassa.
- Toisena vaihtoehtona on tehdä testitapaukset Tarantulaan ennen CUnitin testien tekemistä suunnittelumielessä ja XML-tiedosto liitetään myöhemmin testitapaukseen muokkaamalla niitä.
- Kolmas vaihtoehto on se, että CUnitilla saadut tulokset on jo luotu valmiiksi ja tehtävä on enää raportoida luomalla testitapaukset Tarantulalla ja liittämällä XML-tiedostot niihin.

2. Testitapausten luominen Tarantulalla

- Testitapauksia luodessa tulee muistaa nimeämisen tärkeys
- Luodaan 3 – 4 testitapausta ja yksi per testattava koodi tai XML-tiedosto, jonka jälkeen tehdään jako testijonoihin jälkeenpäin. Testitapausten määrä vaihtelee vaaditun työmäärän perusteella.
- Testitapauksissa vaiheet voivat olla ne testattavat funktiot, joita kukin testattava koodi pitää sisällään. Tällöin testitapausten vaiheiden määrä vastaa testattavan tiedoston funktioiden määrää. Toisena vaihtoehtona on merkitä vaiheiksi kaikki CUnit -tiedostossa olevat ASSERT -komennot. Vaikka vaiheiden onnistumisen merkitseminen olisi silloin tarkkaa, tämä voisi aiheuttaa valtavasti työtä riippuen ASSERT -komentojen määrästä.

3. Testijonojen luominen Tarantulalla

- Siirretään moduulikohtaiset testitapaukset niille nimettyihin testijonoihin
- Nimeämisen tärkeys

4. Testisuoritusten tekeminen ja ajaminen Tarantulalla

- Testitapausten vaiheiden läpikäyminen testisuorituksessa
- Mikäli testitapauksen vaiheet vastaavat CUnit-tiedoston funktioita, merkataan vaiheet testisuorituksen ajossa seuraavalla tavalla:
    - ”Pass”: Hyväksytty: Funktion testaavat ASSERT -komennot onnistuivat.
    - ”Fail”: Hylätty: Funktion testaavat ASSERT -komennot epäonnistuivat.
    - ”Skip”: Ohitettu: Funktion testaava ASSERT -komento löytyi CUnit -tiedostosta mutta komennon suorituksessa tapahtui virhe tai se ohitettiin.
    - ”Not Implemented”: Ei toteutettu: Funktion testaavia ASSERT -komentoja ei löydy CUnit tiedostosta.

1.2 Tehtävä 2. Testausautomaatio

Testausautomaation opettelu: kirjoitetaan c-kielinen ohjelma, joka suorittaa useamman komentorivin komennon. Jokainen näistä tiedoston komennoista kääntää ja suorittaa oman CUnit-tiedoston. Lisäksi voidaan lisätehtävänä käyttää Valgrindia muistivuotojen varalta.
Mikäli tehtävämäärä ei ole liian suuri, voidaan tehtäväksi antaa myös CUnit-tiedostojen kirjoittaminen ja kohdeohjelman testaaminen itse, joissa pitää hyödyntää testidatana syötteenä tekstitiedostoa, joka taas sisältää listan testattavia arvoja. Esimerkiksi laskutoimitusfunktion testaamiseksi hyödynnetään listaa erilaisista numeroista tai syöteistä. Mikäli opiskelijoilla on tallessa aiempia CUnit-tiedostoja, voidaan tehtäväksi valita niiden suoritus automaattisesti.

Tehtävä voi edetä seuraavien vaiheiden kautta:

1. Testien kirjoittaminen CUnitilla ja testidatan hyödyntäminen

- Kirjoitetaan 3-4 ajettavaa CUnit-tiedostoa, jotka voivat myös olla ennalta tehtyjä, mikäli tehtävä on liian suuri. Tai sitten osa ajettavista testeistä on valmiiksi luotu ja osa pitää kirjoittaa itse. Yhdessä näistä on suuren testidatamäärän hyödyntäminen käyttämällä tekstitiedostoa, joka sisältää useita testattavia arvoja.

2. Testiautomaatio ja Valgrindin käyttö:

- Automaatiokoodin kirjoittaminen, joka suorittaa komentorivillä komennoilla aiemmin luotujen CUnit-tiedostojen kääntämisen ja ajamisen. Tulokset tallennetaan XML-tiedostoihin.
- Valgrindin hyödyntäminen: Mikäli testattavat koodit hyödyntävät muistinvarausta esim. mallock-komennon avulla, voidaan tehtäväksi antaa myös Valgrindin käyttö koodien testauksessa. Valgrindilla tulokset on mahdollista tallentaa tekstitiedostoon.

2 LOPULLISET KURSSITEHTÄVÄIDEAT

2.1 Tehtävä 1.

Tämän tehtävän tarkoituksena on opetella testauksenhallintatyökalu Tarantulan käyttöä ja oppia testauksen raportointia. Testauksen kohteena on yksinkertainen laskinsovellus, joka hyödyntää header-tiedostojen avulla muiden tiedostojen funktioita. Pääsovellus ”HeadCalculator.c” hyödyntää tiedostoja ”BasicCalculator.c”, ”CompareCalculator.c” ja ” CompareCalculator2.c”.

Tehtävä etenee kolmessa vaiheessa:

1. Luo Tarantulalla testitapaukset jokaisesta tiedostosta, jonka jälkeen liitä ne Tarantulalla luotavaan testijonoon. Pidä mielessä luomiesi testitapausten ja – jonon nimeäminen järkevästi sekä testitapausten sisältämät vaihteet. Tässä tehtävässä jokainen testitapauksen vaihe vastaa yhtä tiedoston funktiota.
   (Kannattaa muistaa että testitapausten kattavuus ja vaiheet riippuvat aina testauksen kohteesta Esim. yksi iso tiedoston funktio voi vastata yhtä testitapausta ja sen vaihteet eri tapoja testata sitä mm. erilaiset muuttujat ja virhesyöttötilanteet.

2. Luo jokaista tiedostoa kohti, joita HeadCalculator.c hyödyntää, oma CUnit -tiedosto. Näillä CUnit-tiedostoilla tulee testata kunkin tiedoston funktioiden toimivuus ja testauksien tulokset tulee siirtää XML-tiedostoon. Nämä XML-tiedostot tulee taas liittää niihin liittyviin testitapauksiin Tarantulassa. Tarkoituksena ei ole korjata koodien virheitä vaan varmistaa funktioiden toimivuus ja löytää eri ASSERT -komennoilla virhekohtia, jotka voidaan XML-tiedostojen avulla esittää ja perustella esim. yrityksen esimiehelle tai asiakkaalle IT-yrityksessä.
   (CUnit-tiedostojen tarvittavat ASSERT -komennot ovat vain numeeristen arvojen ja merkkijonojen vertailua, jottei tehtävä käy liian työlääksi.)

3. Liitä CUnitilla saadut XML-tiedostot niille varattuihin testitapauksiin ja luo testisuoritus. Merkitse epäonnistuneet ASSERT -komennot sisältäneet funktiot.

2.2 Tehtävä 2.

Tämän tehtävän tarkoituksena on havainnollistaa käyttöliittymätestausta ja testausautomaatiota. Tässä opiskelija tajuaa, että on hyödyllistä laatia ohjelmien testaus automaattisesti, jolloin voidaan aina tarkistaa ohjelman toimivuus tekemättä sitä aina manuaalisesti. Näin myös estetään ohjelman regressio sitä päivitettäessä, kun automaatiotestauksella funktiot toimivat yhä samalla tavalla. Testauksen kohteena on yksinkertainen laskinsovellus, joka hyödyntää header-tiedostojen avulla muiden tiedostojen funktioita. Pääsovellus ”HeadCalculator.c” hyödyntää tiedostoja ”BasicCalculator.c”, ”CompareCalculator.c” ja ” CompareCalculator2.c”. Laskinsovelluksessa toimintoina ovat peruslaskutoimitukset, kahden luvun vertailu, parittoman ja parillisen luvun selvitys, alkulukutoiminto sekä palindromiluvun tarkistus.

Tehtävä etenee kahdessa vaiheessa:

1. Testaa laskinsovelluksen käyttöliittymän ja funktioiden toimivuus hyödyntämällä tekstitiedostoa syötteenä. Luo tekstitiedosto, joka sisältää laskinsovellukseen tulevat syötteet ja jotka testaavat kaikkien funktioiden toiminnan komentorivillä. Lisäksi suorita käyttöliittymän testaus siten, että komentoriville ilmestyvät tulosteet tallentuvat toiseen tekstitiedostoon, jolloin voidaan osoittaa työnantajalle/professorille funktioiden toimivuus konkreettisesti. Tässä tapauksessa laskinsovelluksen hyödyntävä tekstitiedosto sisältää listan numeroita, jotka ovat valikon vaihtoehtoja ja laskutoimitusten syötteitä. Tällä halutaan automatisoida käyttöliittymän testausta.
   (Kannattaa olla tarkkana laatiessa tekstitiedoston syötteitä, jottei ei synny ikuista silmukkaa. Tämä aiheuttaa ongelmia, jos saatua tulostetta aiotaan siirtää tekstitiedostoon. Jokaisella rivillä yksi numero jokaista syötettä kohti.)

2. Laadi tekstitiedosto skripti, jolla voidaan suorittaa aikaisemman tehtävän yksikkötestit ja käyttöliittymätestaus yhdellä kertaa komentorivillä. Laadittava tekstitiedosto sisältää testaamisen tekevät komentorivikomennot, jotka taas luetaan ja suoritetaan tekstitiedostosta. Suorita yksikkötestit siten, että niiden tulokset tallentuvat XML-tiedostoihin ja käyttöliittymän testaukset tekstitiedostoon. Koska tarkoitus on suorittaa kaikki testit kerralla, jätä CUnit -toiminto testien selailusta jälkeenpäin pois.

(Ole tarkkana, että suoritat komentorivikomennot oikeassa järjestyksessä eli muistat kääntää ohjelmat ennen niiden suorittamista. Lisäksi muista, että suoritettavissa komennoissa otetaan huomioon sovellusten sijainti, kuten normaalisti käytettäessä komentoriviä. Tästä syystä luo tekstitiedosto kansioon, jossa kaikki testattavat sovellukset sijaitsevat.)