SAVONIA
ammattikorkeakoulu

OPINNÄYTETYÖ - AMMATTIKORKEAKOULUTUTKINTO
TEKNIIKAN JA LIIKENTEEN ALA

AUTOMAATTITESTIT OHJELMOINNIN PERUSTEET -KURSSILLE

TEKIJÄ Elias Niskanen

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Tutkinto-ohjelma Tietotekniikan tutkinto-ohjelma	
Työn tekijä(t) Elias Niskanen	
Työn nimi Automaattitestit Ohjelmoinnin perusteet -kurssille	
Päiväys 8.8.2024	Sivumäärä/Liitteet 22
Toimeksiantaja/Yhteistyökumppani(t) Savonia AMK	

Ohjelmistotestauksen avulla yleisesti vähennetään ohjelmistovirheitä, arvioidaan niiden riskiä sekä parannetaan ohjelmiston laatua. Ohjelmistotestausta voidaan suorittaa automatisoidusti, jossa toistuvat testitapaukset suoritetaan automaattisesti. Toimiva automaattitestaus varmistaa ohjelmiston laadun ylläpitoa ja vapauttaa työntekijöiden resursseja. Automaattitestien suunnittelu on tärkeässä roolissa ja sitä varten tarvitaan hyvä kokonaiskuva tehtävistä. Yksikkötestit ovat pieniä ohjelmia suuren kokonaisuuden sisällä. Yksikkötestauksessa ohjelmiston komponentteja tarkistetaan yksittäin. Yksikkötesteillä pyritään havaitsemaan koodin virheet aikaisessa vaiheessa, etteivät ne kertaudu ja ne ovat myös helpommin korjattavissa kehitystyön alkuvaiheessa. Yksikkötestit ohjaavat parhaimmillaan ohjelmiston suunnitteluprosessia.

Tietotekniikan AMK insinööriopintojen Ohjelmoinnin perusteet -kurssi on ensimmäinen osa laajaa ohjelmointikurssien kokonaisuutta. Kurssin osaamistavoitteina on mm. oppia suunnittelemaan ja toteuttamaan ohjelmistoja C#-kielellä sekä oppia luomaan ohjelmallisesti hakemistoja ja tiedostoja, kirjoittaa dataa tiedostoihin ja lukea tietoa tiedostoista. Ohjelmoinnin perusteet -kurssilla opiskelijoiden tekemien ohjelmien tarkastaminen tapahtuu nykyään automaattitestien avulla. Tässä tietotekniikan AMK-insinööriopintojen lopputyössä on tavoitteena kehittää Ohjelmoinnin perusteet -kurssilla käytettäviä automaattitestejä opiskelijalähtöisiksi, jotta opiskelijoiden ymmärrys testaamiseen vahvistuisi jo opiskelujen alkuvaiheessa ja siten oppimisprosessi ohjelmoinnin opiskelussa tehostuisi.

Tämä opinnäytetyö koostuu kirjallisuuskatsauksesta ja päiväkirjamuotoisesta työkuvauksesta. Painopisteenä työssä oli erityisesti tulevien tietotekniikan opiskelijoiden ymmärryksen lisääminen testaamisesta ja koodaamisen oppimisprosessin vahvistaminen opintojen alkuvaiheessa. Opinnäytetyön työosassa ei laadittu kokonaan uusia automaattitestejä, vaan pyrittiin selkiyttämään mm. tehtävänantoa, jotta tehtävät toimisivat opiskelijoita aktivoivina sekä oivallusta ja motivaatiota lisäävinä. Jokaisessa tehtävänannossa painotetaan ohjelmoinnin luonnetta ja koodin kirjoitusmuotoja. Testejä muokattiin toimimaan Windows ja Linux -käyttöjärjestelmillä ja testien toimivuutta selkeytettiin ohjeistuksessa, varmistaen kuitenkin opiskelija aktiivista roolia ohjeiden soveltamisessa eri testiskenaarioissa. Mahdollisesti pistokokeet monivalintakysymyksinä voivat toimia opiskelijoiden osaamisen arvioinnin välineenä. Jatkossa opiskelijatyytyväisyyden ja opiskelutulosten mittaaminen Ohjelmoinnin perusteet-kurssilla toisi lisätietoa tässä opinnäytetyössä ehdotetun muutoksen vaikutuksista.

Avainsanat
Ohjelmistotestaus, automaattitestit, ohjelmointi, opiskelu

SISÄLTÖ

1. JOHDANTO

Ohjelmistojen toimivuus on ensiarvoisen tärkeää loppukäyttäjälle. Toimivien ohjelmistojen tekeminen vaatii koodaajalta oman koodin ja työn kriittistä arviota. Bugit ja toimintahäiriöt vaikuttavat olennaisesti käyttäjätyytyväisyyteen ja ovat haitallisia esim. yrityksen imagolle. Niitä voi syntyä ohjelmointiprosessin missä vaiheessa tahansa; suunnittelu-, kehitys-, ja testausvaiheessa tai myös loppukäyttäjän sovelluskäytössä. Ohjelmistotestaus on keskeinen osa laadunvarmistusprosessia. (Rehn, E 2023.)

Ohjelmistotestauksen avulla yleisesti vähennetään ohjelmistovirheitä, arvioidaan niiden riskiä sekä arvioidaan ja parannetaan ohjelmiston laatua. Ohjelmistotestauksella myös etsitään ohjelmiston vikoja ja laukaistaan niitä, arvioidaan koodia sekä tarkistetaan, että ohjelmisto täyttää sille asetetut vaatimukset. (Ohjelmistotestauksen perusteet 2019.)

Ohjelmistotestausta voidaan suorittaa mm. manuaalisesti tai automatisoidusti (Rehn 2023). Automaattitestaus on aiemman käsin tehtävän testauksen lisänä käytettävä testausmuoto, jonka avulla toistuvat testitapaukset suoritetaan automaattisesti (Rana 2019). Automaattitestaus on yleistynyt yrityksissä, kun työskentelymenetelmät kehittyvät ja projektit ovat laajempia. Toimiva automaattitestaus varmistaa ohjelmiston laadun ylläpitoa automaattisesti ja vapauttaa työntekijöiden resursseja. Testit voidaan myös luoda niin, että niitä voidaan käyttää uudelleen. Testien huolellinen suunnittelu on tärkeää ja siihen tarvitaan hyvä kokonaiskuva tehtävistä. (Kasurinen, J 2013.)

Yksikkötestit ovat pieniä ohjelmia suuren kokonaisuuden sisällä. Yksikkötestauksessa ohjelmiston komponentteja tarkistetaan yksittäin. Yksikkötestaus suoritetaan siten, että jokainen ohjelmiston komponentti eristetään, jonka jälkeen sille suoritetaan sarja testejä ja varmistetaan siten ohjelmiston toimivuus. Tarkoitus on havaita virheet aikaisessa vaiheessa, etteivät ne kertaudu ja toisaalta ne on helpompi löytää ja hoitaa alkuvaiheessa kehitystyötä. (Kasurinen, J 2013.)

Tietotekniikan AMK insinööriopintojen Ohjelmoinnin perusteet -kurssi on ensimmäinen osa laajaa ohjelmointikurssien kokonaisuutta. Kurssin osaamistavoitteina on mm. oppia suunnittelemaan ja toteuttamaan ohjelmistoja C#-kielellä, joka sisältää muuttujien, merkkijonojen taulukoiden ja niiden käsittelyfunktioiden hallinnan. Opiskelija oppii tekemään omia funktioita ja käyttämään C#:n omia funktioita. Tavoitteena on myös oppia luomaan ohjelmallisesti hakemistoja ja tiedostoja, kirjoittaa dataa tiedostoihin ja lukea tietoa tiedostoista. (Savonia AMK.)

Ohjelmoinnin perusteet -kurssilla opiskelijoiden tekemien ohjelmien tarkastaminen tapahtuu nykyään automaattitestien avulla. Opiskelijan ymmärrys automaattitestauksesta on opintojen tässä vaiheessa vielä vähäistä. Ohjelmistotestaus on kuitenkin tärkeä osa opiskelijan oppimisprosessissa. Kun mahdollinen koodin virhe tunnistetaan testin avulla nopeasti, se on helpompi ymmärtää kuin myöhemmässä koodausprosessin vaiheessa. Opiskelija voi myös varmistua oman logiikan toimivuudesta automaattitestauksen avulla ja tämä vahvistaa opiskelijan oppimista kurssin aikana.

Oppiminen ja korkeakouluopinnoissa pärjääminen on monen tekijän summa (Mälkki, K ja Mansikka-Aho, A 2020). Tässä tietotekniikan AMK-insinööriopintojen lopputyössä korostetaan erityisesti opiskelijoiden ohjaamista testien ymmärtämiseen. Painotus Ohjelmoinnin perusteet -kurssilla on erityisesti ohjelmointitehtävien tekemisessä ja harjoittelemisessa. Opiskelijoiden tekemien ohjelmien testaamisessa testien virheilmoitukset ja näiden selkeyttäminen opiskelijalle ohjaavat ja valmistavat opiskelijaa seuraavia kursseja sekä työelämää varten. Oman koodin ja työn virheiden tunnistaminen ja ymmärtäminen on kriittistä tietoa, joka tulee antamaan hyvän pohjan opintojen etenemiselle ja työelämälle.

Työssä tuodaan esille automaatiotestauksen tärkeys ohjelmistojen luomisessa. Ohjelmistotestit auttavat myös ulkopuolista ymmärtämään ohjelmistoa ja erityisesti opintojen kannalta on tärkeää huomata ohjelmistovirheet ja saada niistä palautetta mahdollisimman helposti.

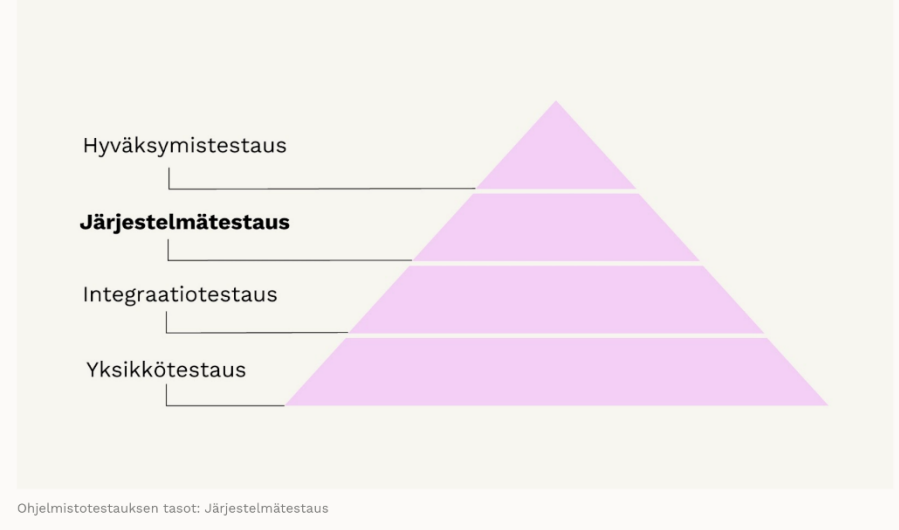
Toimeksiantaja on Savonia-ammattikorkeakoulu. Työ on toteutettu etänä GitHubin kautta ja raportti toteutetaan päiväkirjamuotoisena. Kyseessä on kehittämistyö Ohjelmoinnin perusteet -kursseille automaatiotestien selkeyttämisessä opiskelijalle ja edelleen oppimisprosessin vahvistamisessa ohjelmoinnin opiskelussa.

2. OHJELMISTOTESTAUS

2.1. Ohjelmistotestauksesta yleisesti

Ohjelmistotestauksella varmistetaan, että ohjelmisto toimii suunnitellusti ja täyttää sille asetetut tehtävät ja vaatimukset. Ohjelmistotestauksen avulla havaitaan ohjelmiston kehitysvaiheessa tulleet virheet ja puutteet. Yritysmaailmassa ohjelmistotestauksella vähennetään ohjelmistovirheiden aiheuttamia kustannuksia ja parannetaan ohjelmiston laatua ja loppukäyttäjän tyytyväisyyttä. (Vertics 2019.)

Ohjelmistotestauksessa on erilaisia vaiheita. Tärkeää on, että testaus suunnitellaan huolellisesti ja toteutetaan aina systemaattisesti. Erilaisia ohjelmistotestaustyyppejä ovat yksikkötestaus, integraatiotestaus, järjestelmätestaus sekä hyväksymistestaus (Vertics 2019).



Kuva 1. Ohjelmistotestauksen vaiheet (Vala group, 2023).

Yllä olevassa kuvassa (Kuva 1) on kuvattu ohjelmistotestausta pyramidimallilla seuraavasti:

1. Yksikkötestauksessa (Unit testing) testataan yksittäisiä toimintoja koodausvirheiden varalta.

2. Integraatiotestauksessa (Integration testing) testataan eri ohjelmistojen, ohjelmistomoduulien tai sovellusten toimivuutta yhdessä (integraatioita) ja pyritään varmistamaan esim. datan liikkuminen eri ohjelmistojen välillä tehokkaasti.

3. Järjestelmätestauksessa testataan koko järjestelmän toimivuutta, että kaikki järjestelmän osat toimivat suunnitellusti yhdessä ja täyttävät määritellyt käyttäjävaatimukset.

4. Hyväksymistestaus, (Acceptance Testing, UAT, User Acceptance Testing) on viimeinen validointi, jossa testataan, että ohjelmisto toimii alusta loppuun asti. Se tehdään mahdollisimman valmiissa toimintaympäristössä ja oikealla datalla (Vertics 2019).

Ohjelmistotestaus on myös osa laajempaa laadunvarmistuksen prosessia, jossa on myös muita elementtejä. Laadunvarmistus takaa, että ohjelmistojen kehitys on yrityksessä tehokasta ja valmiit

ohjelmistot täyttävät niille asetetut vaatimukset ja ovat loppukäyttäjälle mahdollisimman käytettäviä. (Haltu 2023, Homes, B 2012.)

SDLC (Software Development Life Cycle) on ohjelmistokehityksen elinkaari. Se voidaan ymmärtää prosessina, jonka tarkoitus on tuottaa laadukkaita ja kustannuksiltaan edullisia ohjelmistoja mahdollisimman lyhyessä ajassa. SDLC tarjoaa hyvin jäsennellyn vaiheiden kulun, jonka avulla tuotetaan nopeasti laadukkaita ja hyvin testattuja ohjelmistoja (Alexandra 2024, Tutorialspoint 2024).

SDLC-malleja on useita erilaisia, joista yleisimpinä vesiputousmalli (Waterfall Model) ja ketterä malli (Agile Model). Vesiputousmalli on lineaarinen ja järjestelmällinen lähestymistapa, jossa jokainen vaihe suoritetaan loppuun ennen seuraavan aloittamista. Se on perinteinen ja perustuu ennakoivaan lähestymistapaan ja täsmällisisiin tehtäviin ja ominaisuuksien yksityiskohtaiseen suunnitteluun ja työn alun vaatimusanalyysiin. (Tutorialspoint 2024.)

Ketterässä lähestymistavassa (Ketterä SDLC-malli, Agile Software Development) tuotetta taas testataan hyvin usein, mikä vaikuttaa lopputulokseen. Se on menetelmä, jossa ohjelmistotuote on mukautuva ja prosessoitava ja se kehittyy yhteistyössä toimittajan (koodaajan) ja asiakkaan kanssa. Ketterät kehitysmetodit ovat ohjelmistokeskeisiä, nopeaan muutokseen, reagointiin ja kommunikaatioon perustuvia kehitysmalleja, joissa pyritään tekemään iteraatioita ohjelmistosta nopealla syklillä. Ketterässä ohjelmistokehityksessä edetään: suunnittelu->vaatimusten analysointi->suunnittelu->koodaus->yksikkötestaus->hyväksymistestaus. Ketterässä mallissa kehitystyötä tehdään reaaliajassa ja ohjelmistoa voidaan mukauttaa jatkuvasti muuttuvien vaatimusten mukaan. (Tutorialspoint 2024.)

Vuonna 2002 on perustettu kansainvälinen testausalan järjestö International Software Testing Qualifications Board (ISTQB, 2016 ja 2023), joka keskittyy ohjelmistotestaajien sertifiointiin ja koulutukseen. Sen tehtävänä on parantaa ohjelmistotestauksen laatua ja ammatillista tasoa maailmanlaajuisesti ja varmentaa yhtenäistä ja korkeatasoista osaamista, että testaajat eri puolilla maailmaa saavuttavat samanlaisen osaamistason. ISTQB on määritellyt vaatimukset testaajille ja paikannut ohjelmistotestauksen koulutuspuutteita. Asiantuntijaryhmä on koonnut testaajien koulutussuunnitelman ja siten on määritelty kansainvälisen sertifiointi peruskoulutus ja sertifikaatti (ISTQB 2024).

2.2. Yksikkötestit

Yksikkötesteillä testataan yksittäisiä ohjelmistokomponentteja tai -ryhmiä yksinkertaisilla testitapauksilla (Iivonen, 2024.) Yksikkötesti ajetaan nopeasti muutoksen jälkeen, ettei uusi koodi tai lisätty moduuli ole aiheuttanut virhettä aiemmissa moduuleissa. Yksikkötestaus on matalan tason testaamista ja sitä suoritetaan usein. Testaus suoritetaan suoraan ohjelmakoodia vasten, ennen kuin

sitä on yhdistetty osaksi järjestelmää ja näin voidaan varmistua yksittäisen logiikan toimivuudesta. Yksikkötestaus ei varmista koko ohjelmiston toimimista oikein mutta on tärkeä osa testausprosessia, koska myöhemmin pienten ja yksittäisten virheiden korjaaminen on hankalaa. Yksittäinen virhe logiikassa voi aiheuttaa myöhemmin ongelmia suuremmassa järjestelmässä ja siksi yksikkötestejä suoritetaan usein. (Hamilton, T 2024.)

Yksiköiden arvoalueet voivat kuitenkin olla niin suuria, ettei niitä pysty kokonaan testaamaan. Silloin korostuu etenkin suunnittelu ja dokumentointi. Esimerkkinä tästä on testisyötteiden rajaaminen ekvivalenssiosituksen tai raja-arvoanalyysin avulla. Rajaaminen mahdollistaa yksikön toiminnan testaamisen hallittavalla määrällä testitapauksia. Testitapausten muodostaminen aloitetaan jakamalla syöte ns. ekvivalenssiluokkiin. Luokat valitaan siten, että testattava kohdekoodi käsittelee kaikki luokkaan kuuluvat alkiot samalla tavalla ja koodin testaaminen yhdellä luokan edustajalla edustaa koodin toimintaa koko luokalla. Jako ekvivalenssiluokkiin tehdään tunnistamalla jokaiselle syötealkiolle mahdollinen arvojen alue ja jaetaan se tyypillisesti perusjaolla sallittuihin ja kiellettyihin arvoihin. Kun ekvivalenssiluokkajako on saatu tarpeeksi tarkaksi, jokaisesta luokasta valitaan edustaja ja kirjoitetaan testitapaukset niitä käyttäen. (Myers, GJ 2011.)

Kun käytetään useampaa ekvivalenssiluokkaa, virheen aiheuttaja koodissa on tyypillisesti syöte tai yksikkö, joka sijaitsee luokan reunalla. Raja-arvoanalyysissa keskitytään testaamaan näitä reuna-alkioita. Esimerkiksi, jos syöte kattaa arvot 10-99, on perusteltua testata arvoilla 9, 10, 99 ja 100.

Testivetoinen kehitys (Test Driven Development, TDD) on ohjelmistokehityskäytäntö, jossa keskitytään yksikkötestien luomiseen ennen varsinaisen koodin kehittämistä (Unadkat, J 2023). Se on monitekijäinen lähestymistapa, jossa yhdistyvät ohjelmointi, yksikkötestien tekeminen ja refaktorointi. TDD:n avulla voidaan luoda laadukkaita ohjelmistoja nopeasti. Testausprosessi ohjaa ohjelmistokehitystä. Se on strukturoitu prosessi, jolla ohjelmistokehittäjät ja testaajat saavat optimoitua koodia. Työ etenee testistä toiseen ja samalla rakentuu lisää toiminnallisuutta ohjelmistoon. TDD:ssä luodaan pieniä testitapauksia jokaiselle ominaisuudelle ja koodia muutetaan tai kirjoitetaan uudelleen vain, jos testit epäonnistuvat. Tämä vähentää testiskriptien päällekkäisyyttä.

Esimerkkinä perustyyppinen laskin -sovellus, jossa on laskutoiminnot yhteenlasku, vähennyslasku, jakolasku ja kertolasku. Kun rakennetaan laskutoimintoa, TDD-lähestymistapa käsittää testitapauksen kirjoittamisen ”yhteenlasku”-funktiolle ja sen jälkeen prosessin koodin kirjoittamisen, jotta testi läpäistään. Kun ”yhteenlasku”-toiminto toimii oikein, kirjoitetaan lisää testitapauksia muille toiminnoille, kuten ”vähennyslasku”, ”jakolasku” ja ”kertolasku”.

Testaaminen ei ole erillinen vaihe ohjelmiston kehityksessä, vaan se on kiinteä osa ohjelmiston kehitystyötä ja suunnitteluprosessia. Ohjelmoinnin aikana syntyneet virheet pyritään saamaan heti kiinni ja koodin yksittäiset osat tarkistetaan heti valmistumisen jälkeen. Ohjelmien toiminnallisuus korostuu jo suunnitteluvaiheessa ja koodaaja pystyy testaamaan omaa koodiaan

suunnitteluprosessissa. (Hamilton, T 2024.) Tämä on myös taloudellista, koska ohjelmiston yksikkötestaus voidaan tehdä ohjelmoinnin kanssa samanaikaisesti. Ohjelmoinnin jälkeen erillisenä projektina tehty testaus olisi hitaampi tapa.

Toisaalta liian aikainen testaus voi johtaa myös turhaan testaustyöhön, jos ohjelma on vielä kehitysvaiheessa. Hyvä testauksen suunnittelu ja ajoittaminen on tärkeää ja siinä korostuukin ohjelmistosuunnittelijan ammattitaito.

2.3. Automaattitestit

Automaattitestauksessa toistetaan yksinkertaisia esim. yksikkötestejä eli testitapauksia automaattisesti osana ohjelmiston kehitystä. Kun komponentti toteutetaan, sille tehdään sitä testaavat testit samanaikaisesti. Yksikkötestauksessa testit ajetaan sekä toteuttajan omalla koneella paikallisesti ja niitä voidaan tehdä myös vielä automaattisesti osana jatkuvaa integraatiota. Samoin kun tehtäviä muutetaan, automaattitestit voidaan ajaa uudelleen virheiden havaitsemiseksi. (IEEE 1990.)

Nykyisin on käytössä useita erilaisia testausmalleja, jotka mahdollistavat erilaisten testikehysten rakentamisen eri käyttötarkoituksiin. Automaattisten testien avulla voidaan varmistaa ohjelmiston toimivuus muutosten jälkeen (regressiotestaus) sekä tehdä käyttöliittymätestausta. (Lewis, WE 2004.)

Regressiotestaus on uudelleentestaamista. Sitä käytetään silloin, kun toimivan järjestelmän osaa muutetaan ja koko järjestelmän toimivuus varmistetaan sen jälkeen. Järjestelmissä ilmenevät virheet kohdistuvat usein juuri uusiin komponentteihin tai niitä hyödyntäviin toimintoihin. Regressiotestauksella varmistetaan järjestelmien toimivuus uusien ominaisuuksien lisäämisen jälkeen. Järjestelmää testataan osien lisäämisen jälkeen ikään kuin se olisi kokonaan uudistettu. Regressiotestausta voidaan myös käyttää versionhallinnassa tehtyjen kehityshaarojen yhdistämisen jälkeen, ettei vanhassa koodissa ole uusien moduuleiden yhdistämisen jälkeenkään ilmennyt virheitä. (Kasurinen, J 2014.)

Korjaamisen apuna voidaan käyttää myös vianhallintaprosessia (JIRA, 2024).
Jira on työkalu esim. virheiden hallintaan ja ratkaisemiseen määritetyn ongelmanseurantajärjestelmän kautta, jossa on erilaisia tarkasti määritettyjä vaiheita ja toimintoja.

Robot Framework on avoimen lähdekoodin automaatiokehys mm. testien automatisointiin. (Robot Framework, 2024). Sitä voi käyttää apuna testien automatisoinnissa. Robot Framework on rakennettu Python-kielelle ja se on helppo integroida muihin Python-kirjastoihin ja -kehyksiin. Sen ymmärrettävä ja monipuolinen syntaksi käyttää avainsanoja ja laajentuu Python-, Java- ja muiden

kielten kirjastojen avulla. Testit ja sovellukset rakennetaan avainsanojen perusteella eikä pitkien koodirivien perusteella, ne voidaan luoda nopeasti ja ylläpito ja päivittäminen on myös helppoa. (Eficode, 2024.)

Automaattitestikehysten ymmärtäminen ja suunnittelu on tärkeää, jotta ne toimivat luotettavasti ja tehokkaasti ja tukevat tavoitteiden saavuttamista.

2.4. Automaattitestien luominen käytännössä

Refaktorointi on koodin rakenteen muuttamista siten, ettei sen toiminnallisuus muutu. Ei-toiminnallisen laadun parantuessa koko ohjelmiston tekninen velka pienenee. Teknisellä velalla kuvataan esimerkiksi vanhentunutta järjestelmää, johon on haastavaa tai mahdotonta päivittää uusia ominaisuuksia. Refaktoroinnin avulla voidaan helpommin huomata ja ratkaista piilossa olevia ongelmia ja ohjelmiston toiminnallisuus kehittyy ja lisääntyy. Ohjelmiston sisäinen rakenne voi kuitenkin muuttua. Refaktorointi vähentää teknistä velkaa, koska sen kautta koodin teknistä laatua saadaan parannettua ennen kuin ohjelmiston sisäinen rakenne muuttuu hallitsemattomasti.

Testejä kirjoittaessa on hyvä keskittyä tekemään lyhyitä selkeitä testifunktioita ja niissä yksinkertaisia tarkistuksia. Se pitää dokumentoida, testata ja ylläpitää ihan niin kuin toiminnallinenkin koodi. Myös testikoodi voi olla virheellistä. (Vertics, 2019.)

Testitapauksia keksiessä hyvä ajattelun lähtökohta on miettiä, mikä on metodille kaikkein tärkeintä ja testata aina kaikkein yleisimmät tapaukset. Kannattaa myös pyrkiä olemaan mahdollisimman luova, koska virheet löytyvät useimmin selkeiden suorituspolkujen reunamilta. Testaamisessa on hyvä keskittyä komponenttien rajapintoihin, jotka auttavat toiminnallisuuden kehittämisessä. Testeissä ei kannata hakea liian mutkikkaita ratkaisuja vaan pitää testit mahdollisimman yksinkertaisina. Testaamisessa on myös hyvä käyttää testikehystä. Pythonissa tällainen on pytest. (Vertics, 2019.)

2.5. Miten automaattitestit sopivat yksikkötestaukseen opiskeluympäristössä

Automaattitestillä voidaan testata yksittäisiä eristettyjä koodien osia tai sitten moduuleita, jotka sisältävät toiminnallisuuksia. Näin pystytään varmistamaan, että jokainen tehtävän osa toimii itsenäisesti oikein. Automaattitestejä voidaan hyvin käyttää opiskelijoiden tehtävien tarkastamiseen ja niiden avulla yksittäisen koodin yksiköiden toimivuus voidaan varmistaa.

Automaattitestit ovat suoraviivaisia ja ennustettavia tietyillä syötteillä. Automaattiset testit tarkastavat tehtävät ennalta määrätyillä kriteereillä tai parametreillä, joka tekee testeistä luotettavia.

Testit ovat nopeita ja toistettavia. Parametrejä voidaan muuttaa mutta itse testejä ei tarvitse muuttaa. Koodia voidaan testata eri syötteillä, esim. välivaiheilla ja eri lopputuloksilla. (Lewis, 2004.)

Automaattitestit tehtävien tarkastuksessa hyödyntävät samoja periaatteita kuin yksikkötestaus, mikä tekee niistä tehokkaita ja luotettavia työkaluja niin tuotekehityksessä kuin opetuksessa. Automaattitestit antavat opiskelijalle nopeaa palautetta virheistä, joten virheet tehtävissä voi korjata nopeasti ja tämä palaute auttaa opiskelijaa myös syventämään oppimistaan koodaamisessa.

Testauspohjainen kehitys (TDD), soveltuu hyvin automaattitestien käyttöön tehtävien tarkastuksessa. TDD:n mukaisesti testit kirjoitetaan ennen varsinaista koodia, mikä varmistaa, että tehtävien vaatimukset ovat selkeät ja kattavat jo ennen tehtävien ratkaisujen kehittämistä. Kun opetus on linjakasta eli vaatimukset, tavoitteet ja niihin pääseminen sekä oppimisen arviointi ovat selkeästi opiskelijalla tiedossa, opiskelijalla on paras mahdollisuus motivoitua ja oppia. Hän ymmärtää mihin opetus tähtää ja mitä tavoitteita opetuksella on ja miten oppimista ja opetuksen tavoitteiden saavuttamista mitataan. (Nevgi ja Lidblom-Ylänne, 2011.)

2.6. Ohjelmistotestaus ohjelmointikurssilla

Oppiminen ja korkeakouluopinnoissa pärjääminen vaativat motivaatiota ja myös innostavaa ja monimuotoista opetusta. Oppiminen on monen tekijän summa. Kun opetusmenetelmät ovat aktivoivia, ne tukevat opiskelijaa itsenäiseen tiedon löytämiseen ja tiedon syventämiseen ja siten edesauttavat oppimistavoitteeseen pääsemistä. Opiskelija pystyy liikkumaan oppimisen eri tasoilla aiemmin opitun ja uuden tiedon välillä ja pystyy myös syventämään oppimistaan oivaltamalla itse asioita eikä vain passiivisesti ottamalla tietoa vastaan. (Mälkki ja Mansikka-Aho, 2020.)

Tässä tietotekniikan AMK-insinööriopintojen lopputyössä korostetaan opiskelijoiden ohjaamista testien ymmärtämiseen. Painotus Ohjelmoinnin perusteet-kurssilla on erityisesti ohjelmointitehtävien tekemisessä ja harjoittelemisessa. Opiskelijoiden tekemien ohjelmien testaamisessa testien virheilmoitukset ja näiden selkeyttäminen opiskelijalle ohjaa opiskelijaa ohjelmoinnin ymmärtämisessä ja toisaalta omien virheiden tunnistamisessa. Opiskelijoiden tekemien ohjelmistojen testaaminen on merkittävässä roolissa oppimisprosessissa. Se syventää opittua tietoa, edistää oppimista ja vahvistaa opiskelijan itsenäistä roolia opintojen edetessä. Oman koodin ja työn virheiden tunnistaminen ja ymmärtäminen on kriittistä tietoa, joka tulee antamaan hyvän pohjan opintojen etenemiselle ja työelämälle. Opiskelijat oppivat myös korjaamaan ja tunnistamaan virheitään ja edelleen analysoimaan ja arvioimaan koodiaan kriittisesti jo sen laatimisvaiheessa. Kun testaus havaitsee virheen ohjelmistossa, opiskelija oppii ajattelemaan kriittisesti ja ongelmanratkaisutaidot vahvistuvat. Näin myös vahvistetaan opiskeluvalmiuksia seuraavia kursseja varten.

Opettajan palaute ja testien ymmärtäminen kehittävät myös opiskelijoiden kykyä antaa ja vastaanottaa rakentavaa palautetta. Automaattitestien vuoksi näiden tuottamien virheilmoitusten lukeminen on erittäin suuressa roolissa oppimisen tukemisessa ja toimii myös aktivoivana opiskelumenetelmänä.

Kriittinen ajattelu, ongelmanratkaisutaidot ja koodin virheistä oppiminen vahvistavat taitoja työelämään, koska ohjelmistoja parannetaan jatkuvasti palaute- ja testauskierrosten avulla. Opiskelija oppii ymmärtämään ohjelmiston laadun ja luotettavuuden tärkeyden sekä koko ohjelmistokehityksen prosessin. Ohjelmistotestauksen opettaminen ja sen avulla opiskelijoiden tekemien ohjelmointien tarkistaminen auttaa opiskelijoita osaltaan ymmärtämään laadunvarmistuksen merkityksen ohjelmistokehityksessä. On hyvä oppia, että testaus on aina suunnitelmallista, dokumentoitua ja tähtää laadunvarmistukseen. Tämä ymmärrys tukee myös valmistautumista kohti työelämää.

3. PÄIVÄKIRJA

3.1. Viikko 18

Hain tietoa päiväkirjamuotoisista opinnäytetöistä ja suunnittelin, miten tulisin tekemään omaa työtäni. Aluksi kirjoitin insinöörityöanalyysin päiväkirjamuotoisesta opinnäytetyöstä. Laadin työsuunnitelman omaa opinnäytetyötäni varten. Sitä varten luin "Autoetnografia - päiväkirjaan perustuva tutkimus" (Kukkurainen, ML 2019.) ja sain tästä hyvin taustatietoa opinnäytetyöni kirjoittamiseen. Samalla viikolla perehdyin kahteen uuteen opinnäytetyöhön, jotka ovat myös päiväkirjamuotoisia (Vaskivuo, P 2018, Porkka, J 2019). Työsuunnitelman lisäksi tein aikataulun opinnäytetyön tekemistä varten. Perehdyin opinnäytetyöni kirjallisuuskatsausta varten yksikkötestauksen kirjallisuuteen ja etsin lähteitä aiheesta. Haastattelin myös ystävääni, joka tekee päivätöidensä ohessa yksikkötestausta Python-kielellä. Keskustelimme laajasti hyödyllisistä lisäohjelmista ja menetelmistä, joita minun kannattaisi käyttää työssäni. Taustatyötä tehdessäni sain myös ohjaajaltani apua ja vinkin mm. Robot Frameworkista, jonka ominaisuuksiin ja toiminnallisuuksiin perehdyin. Robot Framework on avoimen lähdekoodin automaatiokehys, jota käytetään laajasti yksikkötestauksessa sekä robotiikan prosessiautomaatiossa (RPA) (Robot Framework 2024). Sen avulla testien kirjoittaminen ja suorittaminen on helppoa ja tehokasta. Robot Framework tarjoaa monipuolisia kirjastoja erilaisten testien toteuttamiseen ja integroituu saumattomasti muihin työkaluihin ja teknologioihin. Harkitsen tämän palvelun käyttöä yhtenä vaihtoehtona yksikkötestauksessa saatuani tarpeeksi taustatietoa.

3.2. Viikko 19

Aloittelin viikkoni tilaamalla ilmaisen kuukauden "GitHub Copilot":ista (GitHub Copilot, 2024) koodini tueksi. Tämän jälkeen avasin "Visual Studio Coden" ja alustin sen toimimaan C# sekä Python-kielillä. Asensin GitHub Copilotin. Kun sain ladattua oikeat SDK-paketit, virkistin muistiani harjoittelemalla ja perehtymällä testien tekemiseen. Sen jälkeen aloitin tehtävät alusta ja testasin niitä eri variaatioilla. Ensimmäiset pari tehtävää menivät läpi ilman ongelmia C#-testien kanssa. Python-testejä en ajanut vielä tänään. Seuraavana päivänä tutkiessani testejä ymmärsin, että yksikkötestauksessa pitää keskittyä vain yhteen tulokseen huomioimatta variaatioita. Tästä syntyi haaste tai ongelma, kuinka opiskelijan virheilmoitukset tehdään ja miten ne kohdennetaan. Yksikkötestauksen toimivuuteen vaikuttaa moni muuttuja. Pohdin tässä tilanteessa jonkin testiautomaatiokehyksen käyttöönottoa, joka identifioisi virheen paremmin. Palasin takaisin opintojeni alkuvaiheisiin ja huomasin, kuinka nykyään helpolta kuulostavat virheilmoitukset eivät todellakaan olleet sitä, kun aloitin opinnot. Tällä hetkellä suurimpana haasteena projektissani on virheilmoituksen lukemisen selkeyttäminen opiskelijalle, siten että opiskelija ymmärtää sen sisällön. Yritän ratkaista tämän projektini aikana. Loppukäyttäjän kannalta virheilmoituksen kohdentaminen olisi merkittävä apu ohjelmoinnin opetteluun, kuten myös virheiden havaitsemiseen jatkossa. Yksikkötestaus onkin ruohonjuuritason testausta ja varsinkin tässä kontekstissa on tärkeää ymmärtää opiskelijan taitotasot ja havainnointikyvyt. C#-testejä läpikäydessäni huomaan, että yhdessä tehtävässä on oikeastaan vain merkintätapa, ks. esimerkki alla:

```
Esittele muuttuja pii, jolle annat alkuarvoksi piin likiarvon 6 desimaalin tarkkuudella.
Lue käyttäjältä ympyrän halkaisija ja tulosta ympyrän piiri ja pinta-ala kolmen desimaal
tarkkuudella seuraavasti (halkaisijaksi annettu 2,5) :

PIIRI     : 7,854
PINTA-ALA : 4,909
```

Kuva 2. Tehtävänanto opiskelijoille.

Tässä kuvassa, (Kuva 2) jos ”PIIRI” sanan jälkeen ei ole tehty samaa väliä, testi ei päästä läpi. Harkitsenkin tässä kohtaa kirjoittavani tehtävänantoihin aina tuon syötteen merkitsijän tarkasti. Opiskelijalle pitää kertoa, että hänen pitää kopioida tämä osa ennen omaa syötettä, jolloin testi päästäisi läpi. Laitoinkin lopulta moneen muuhunkin tehtävään ohjaavan lauseen, jotta testit menisivät läpi ilman vastaavanlaisia ongelmia. Kirjoitin kaikkiin tehtäviin seuraavasti: ”Muista tarkistaa, että syötteesi on täsmälleen kuten yllä annetussa esimerkissä”. Tämä on ensimmäinen variaatio ohjauslauseesta ja tätä muokkaan tehtävien eri vaativuustasojen mukaan. Uskon että tämä auttaa opiskelijoita huomattavasti. Itse muistan, että usein ongelmat testien kanssa johtuivat siitä, että en lukenut tehtävänantoa tai esimerkkivastauksia tarpeeksi tarkasti. Mietin myös, miten voisin testeihin itseensä laittaa joitain apuja, jos testi ei mene läpi. Tämän viikon aikana aloin myös etsiä lisää tietoa laajemmin ja kiinnostuin myös oppimisesta ja opiskelusta yliopistossa. Perehdyin hiukan siihen, miten korkeakouluopintojen sujuvuutta voisi kohentaa. Pohdin myös näitä asioita suhteessa omiin oppimiskokemuksiini. Opiskelijaa kannattaa ohjata oivaltamaan ja oppimaan esimerkiksi näiden testien avulla virheistä. Tehtävien on tarkoitus aktivoida ja ylläpitää opiskelumotivaatiota. Selkeät tehtävänannot, oppimistavoitteet ja linjassa oleva arviointi auttavat, opiskelija ymmärtää mitä häneltä vaaditaan ja miten tavoitteeseen voi päästä.

3.3. Viikko 20

Jatkoin aputekstien kirjoittamista tehtävänantoihin, sekä aloitin myös selkeyttämään tehtävänantoja. Mietin myös, kuinka saisin opiskelijat lukemaan tehtävänannot huolellisesti. Ymmärrän nyt tehtävänantojen merkityksen tehtävien tekemisessä, onnistumisessa ja oppimisessa yleensä. Huomaan nyt, kuinka monta tuntia olisin itse säästänyt opiskeluaikana, jos olisin lukenut tekstit tarkasti. Koitan tässä taas mennä samaan mielentilaan, jossa olen ollut aloitellessani ohjelmointia. Tarkkojen tehtävänantojen kirjoittaminen onkin haastavampaa, kuin sen alun perin oletin olevan. Tässä korjaillessani tehtävänantoja huomaan myös, kuinka esimerkiksi kurssin opiskelijan oma kanta asiaan olisi hyvin merkittävä ohjeiden rakentamisessa. Olen tehnyt tehtäviä tänä aikana ja erityisesti fokusoinut tehtävänantoihin. Kuitenkin, koska olen jo ohjelmoinut jonkun aikaa, en varmasti pääse siihen samaan mielentilaan tai epätoivon tilaan, missä opiskelija on lukiessaan tehtävänantoa. Koitankin tehdä tehtävänantoihin todella laajoja kuvauksia siitä, mitä oikeasti tehtävässä haetaan. Pyrin antamaan opiskelijalle tehtävän vaatimukset selkeästi tiedoksi. Olen työstänyt Teema 2:ta ja tähän mennessä tehtävät ovat olleet melko suoraviivaisia. Niissä ei ole ollut liikaa korjattavia asioita omasta mielestäni, mutta olen koittanut ainakin mainita syötteen virheettömyydestä mahdollisimman tarkasti. Tämä on mielestäni tärkein asia, koska vaikka koodi olisi oikein mutta

syöte on muotoiltu väärin, testi antaa tässä kohtaa virheilmoituksen ja aiheuttaa harmaita hiuksia. Tämä voi olla todella haastavaa aloittelevalle ohjelmoijalle. Loppuviikosta tarkastelin koodia ja aloin miettimään, miten saisin ymmärrettävien tehtävänantojen lisäksi optimoitua opiskelijoiden ajankäyttöä. Ajattelin lisätä nykyiseen READ.me tiedostoon hieman ohjeita testituloksen analysointiin. Alla olevassa kuvassa näkyy testattu koodi, jossa testattiin tyhjää ohjelmaa ilman ohjelmakoodia.

```
  Failed _01_tests.UnitTest1.CheckOutput(data: [10, 5, -1], expected: "7.5") [1 ms]
  Error Message:
   Assert.Contains() Failure
Not found: 7,5
In value:
  Stack Trace:
     at _01_tests.UnitTest1.CheckOutput(IEnumerable`1 data, String expected) in S:\Kouluhommat\c-shar

  Failed _01_tests.UnitTest1.CheckOutput(data: [7, 5, 8, -1], expected: "6.7") [< 1 ms]
  Error Message:
   Assert.Contains() Failure
Not found: 6,7
In value:
  Stack Trace:
     at _01_tests.UnitTest1.CheckOutput(IEnumerable`1 data, String expected) in S:\Kouluhommat\c-shar

  Failed _01_tests.UnitTest1.CheckOutput(data: [-1], expected: "Et antanut yhtään lukua") [< 1 ms]
  Error Message:
   Assert.Contains() Failure
Not found: Et antanut yhtään lukua
In value:
  Stack Trace:
     at _01_tests.UnitTest1.CheckOutput(IEnumerable`1 data, String expected) in S:\Kouluhommat\c-shar
```

Kuva 3. Testien virheilmoitukset.

Analysoidaan yllä olevaa kuvaa (Kuva 3) hieman. Koitan antaa apuja opiskelijalle testituloksen analysointiin, ”Failed _01_tests. UnitTest1.CheckOutPut(data: [10, 5, -1] ”, mitä numeroita koodille syötetään hakasulkujen sisällä ja tämän jälkeen ”expected: ”7.5” ” joka tarkoittaa mitä koodin odotettiin tulostavan. Ja viimeinen kohta ” [1 ms] ” tarkoittaa vain, kuinka kauan testin suorittaminen kesti. Kyseisessä virheilmoituksessa käytetään tiettyä ulkoista syötettä. Olen tehnyt esimerkin tässä kurssissa käytetyn xUnit- testien ja valmiin syötteen mukaisesti.

Tämän alla nähdään kuvassa (Kuva 3) virheviesti eli ”Error Message:” Tässä ”Assert.Contains()” osa koodia ei mennyt läpi, joka nimenomaan tarkistaa, mitä koodi antoi tulokseksi. Tässä tilanteessa koodi ei antanut mitään syötettä. Tämän alapuolellahan selitetäänkin hyvin ”Not found: 7.5” eli tulosta 7.5 ei löytynyt syötteestä, jota tarkasteltiin, joten testi palautti hylätyn arvosanan. Tämän alapuolelta löytyy vielä ”In value:” jossa olisi koodin palauttama syöte, mikäli se olisi palauttanut jotain. Viimeiset kaksi riviä: ”Stacktrace:” -rivi sekä alempi rivi ovat vain opaste testikoodin kohtaan, jossa virhe tapahtui eli tässä tapauksessa:

” Stack Trace: at _01_tests. UnitTest1.CheckOutput(IEnumerable`1 data, String expected) in S:|Kouluhommat|c-sharp|dotnet|teema 3|base|exercises|tests|01_tests|UnitTest1.cs: line 30.”

Yllä oleva koodin pätkä viittaa suoraa kansioon, johon koodi on tallennettu lokaalisti sekä testitiedoston tietylle riville, jossa ei ole saatu oikeaa lopputulosta. Tämän pyrin selittämään opiskelijoille mahdollisimman tarkasti, jotta heidän ajankäyttönsä tehostuisi ja he ymmärtäisivät tarkemmin missä kohtaa koodissa ongelma on. Viikon lopuksi kysyin vielä luokkatovereiltani missä osa-alueissa he olisivat halunneet lisäohjeistusta aloittaessaan koulun ja koodaamisen. Hyvin vahvasti tuli esille, että tehtävänantojen täytyy olla helposti ymmärrettäviä ja näissä ei saisi olla mahdollisuutta ymmärtää tehtävänantoa väärin. Näitä asioita pohtiessani ja samalla tehtävänantoja korjatessani mietin, kuinka voisin vielä parantaa tehtävänantoja paljastamatta liikaa tehtävästä. Mietin, kuinka pystyisin tehostamaan opiskelijoiden ymmärrystä tehtävänannosta ja optimoimaan heidän ajankäyttöään vain tehtävän tekemiseen, joka vähentää turhautumista yleisesti. Vahvasti tuli myös esille, että jos tehtäviä testattaisiin siten, että ”return” komento olisi jo koodissa ja välivaiheet testataan, saadaan ”return” -käsky tuottamaan oikea tulos. Tämän en valitettavasti usko olevan mahdollista tai ainakin se vaatisi suuria muutoksia nykytesteihin ja tehtäviin. En näe sen edes olevan hyödyllistä tässä projektissa. Koitan kuitenkin ilmaista tehtävänannoissa tarkalleen ja selkeästi, mitä lopputulosta tehtävässä haetaan sekä myös menettelytapoja, joilla lopputulos olisi saatava näkyville. Painotan edelleen hyvin vahvasti opiskelijoita tarkistamaan syötettä ja välimerkkejä. Ohjelmointia opiskellessa on kuitenkin hyvä ymmärtää jo heti alussa, että kyseessä on pilkuntarkka ammattitaito ja sinnepäin oleva koodi ei ole riittävä. Täsmälleen oikea vastaus on ainoa oikea tapa käsitellä ongelmia koodissa.

3.4. Viikko 21

Tänään pidimme väliaikakatsauksen työstä, ja tämän pohjalta ymmärrän taas tarkemmin oman työni tarkoitusta. Työssäni keskitytään enimmäkseen opiskelijan ohjaamiseen, jotta saadaan poistettua turhia välivaiheita ja harmaita hiuksia kooditehtäviä tehtäessä. Itse testit toimivat hyvin ja niihin on vaikeaa saada suoria kommentteja opiskelijan tekemistä virheistä, koska virheitä on monia erilaisia. Tämä tuottaa haasteita virheiden ehkäisyssä. Tältä pohjalta alan tekemään enemmän tekstipohjaista ohjeistusta jo tehtävänantoon. Päädyin tekemään READ.me tiedoston kurssille, jossa käsittelen jokaisen testikategorian eri virheilmoitukset ja pyrin antaman mahdollisimman hyvät virheenlukuohjeet opiskelijalle. Sovimme ohjaajan kanssa, että teen yhden tehtävän uusiksi. Teen myös uudet testit kyseiseen tehtävään, koska nykyinen tehtävä tuottaa ongelmia testauksessa. Tähän minun pitää paneutua huolellisesti. Tehtävän on muistutettava edellisiä tehtäviä kyseisestä teemasta, mutta se pitää olla silti helppo testata, niin Pythonilla kuin C#-kielillä. Python-testeihin en ole vielä paneutunut tarkemmin vaan jatkan vielä C#-testeillä. Sain kyllä hyviä ohjeita ja vinkkejä Python-testien tekemiseen, ja pyrin ne hyödyntämään. Sain tällä viikolla vastauksen ohjaajaltani kyseisen tehtävän muokkaamista varten. Tehtävä oli seuraavanlainen:

” Lue käyttäjältä merkkejä, kunnes käyttäjä painaa enteriä. Isot kirjaimet tulostuvat pienenä ja pienet kirjaimet tulostuvat isona. Muut merkit eivät tulostu. Riittää, että toimii kirjaimilla väliltä a-z ja A-Z.”

Olin tehnyt ehdotuksen kokonaan uudesta tehtävästä, mutta se muistutti liikaa toista tehtävää kyseisessä teemassa. Päädyin tekemään koodin käyttäen ”Console.Read()” -metodia ”Console.Readkey()” -metodin sijaan, koska ”Console.ReadKey()” -metodia oli hankala testata. Tein kyseisestä tehtävästä tänään mallivastauksen ja pyysin ohjaajaltani mielipiteen, ettei tehtävä ole liian haastava tai monimutkainen opiskelijoille. Käytin koodissani ”stringbuilder” -metodia, joka käy läpi käyttäjän syötettä keräten siitä erilaisia merkkejä. Mielestäni paras tapa testata koodia on käyttää ”Console.Read()”-metodia, koska merkkien erottelu on tehty niin hyvin tällä kyseisellä metodilla. Jään odottamaan vastausta ohjaajaltani ja teen sitten tarvittavat muutokset. Sain ohjaajaltani vielä samalla viikolla viestin muutosehdotuksia tehtävään. Olin käyttänyt taulukoita sekä ”stringbuilder”-metodia, joita ei ole vielä ollut opetettu opiskelijalle. Tein näiden muutosten perusteella uuden tehtävän, jossa käytin vain tyhjää ”string”-muuttujaa, jota tutkin ”for-loopissa” ”char”-merkistön avulla. Tätä tehdessäni huomasin taas, että oma ajattelutapani toimii jo taulukoiden ja erilaisten metodien avulla. Koulutuksen alussa olevalle opiskelijalle on tärkeää ymmärtää nimenomaan perinteisiä ohjelmointitapoja, jotka luovat pohjaa myöhemmin opittaville helpottaville ohjelmointitavoille. Uusi tieto rakentuu vanhan päälle ja tiedon perusta pitää olla kunnossa. Ajattelin tehtävää tehdessä suoraan taulukoita ja helpottavia metodeita, mutta tässä kohtaa pitää fokusoida opiskelijan ohjelmointipohjan ja perustiedon luomisen. Opiskelija ymmärtää myöhemmin uusia menetelmiä helpommin ja yhdistää niitä uusiin suurempiin kokonaisuuksiin. Tein tänään vain C#-testin ja teen Python-testin myöhemmin samasta aiheesta, kunhan etenen siihen kurssia läpikäydessäni.

3.5. Viikko 22

Tällä viikolla tein viimeisiä muutoksia työhöni. Tein Python-testin myös kyseiselle tehtävälle toimivaksi. Käytin samaa mallivastausta, kuin C#-tehtävien puolella. Tämä ei tuottanut hirveästi vaivaa, sillä testi toimi käytännössä jo valmiiksi oikein tässä tehtävässä. Ainoastaan oma tapani ajaa testejä oli aluksi virheellinen. Menin testikansioon ajamaan testejä “Python-m unittest tests.py” -komennolla. Ohjelma herjasi erikoisia virheilmoituksia paketeista, jotka pitäisi latautua automaattisesti testin avautuessa. Juutuin tähän kohtaan pidemmäksi aikaa, kunnes ymmärsin mennä katsomaan kurssilta videon, kuinka testejä ajetaan Python-ympäristössä. Tunnistin jälleen itseni ja oman tapani toimia. Sama tapahtui samalla kurssilla kolme vuotta aiemmin. Jos vain lukisin ohjeet ennen tehtävän tekemistä, olisi kaikki aina helpompaa. Ehkä vielä joskus opin. Ymmärsin, että testit tulee ajaa suoraa tehtäväkansiosta ja käyttäen komentoa “python test.py”. Tämän jälkeen lisäilin vielä loppuhinkin tehtäviin ohjaavia lauseita tehtävien tekemiseen. Sama tieto opiskelijoille eritavoin muotoiltuna: ”Lue tarkasti ohjeet niin säästyt turhalta harmilta ja harmailta hiuksilta”. Tämän jälkeen pohdin, miten voisin avata testien virheilmoituksia opiskelijoille. Pyrin luomaan yksinkertaisen koosteen jostain tietystä virheilmoituksesta ja annan opiskelijalle apua, että hän pääsee alkuun testien lukemisessa. Ohjaajani kanssa sovimme, että tekisin jokaisesta eri testityypistä ohjeet. Se ei kuitenkaan mielestäni ole tarpeen. Näkemykseni mukaan, jos annan opiskelijalle vain ensimmäisen tason testin virheestä ohjeet, hän oppii varmasti hyödyntämään sitä myös muunlaisissa tehtävissä. Tämä on mielestäni paras tapa opettaa opiskelijaa soveltamaan

saamaansa tietoa. Kirjoittelin parilla lauseella testejä, mitä tavoitteita niille on ja pyrin mahdollisimman yksinkertaisen ohjeeseen. Näin ei jäisi liikaa kysyttävää ja ohjeen soveltaminen jatkossa olisi helppoa. En myöskään halua antaa suoraan avaimia käteen, opiskelijan pitää pystyä soveltamaan tehtäviä tehdessään. Kun olen lukenut opettamisesta yliopistossa ja oppimisteorioista, se on vahvistanut omien kokemustenkin kautta, että opiskelijan aktivoiminen ja oma-aloitteellisuuden tukeminen tehostaa oppimista. Projektini on valmis. Ohjaajan kanssa tarkastelen ja varmennan, että ohjeeni ja versionhallintaan viemäni muutokset ovat valideja ja että ne tukevat opiskelijoiden oppimista. Tavoitteena projektillani oli helpottaa uusien opiskelijoiden koodaustaitojen opettelua, rakentaa tietotaitoa hyvälle perustalle ja painottaa harjoittelun ja tarkkuuden merkitystä koodaustyössä.

4. YHTEENVETO

Työnkuva oli minulle alussa hieman epäselvä ja pohdin pitkään toteutustapaa. Toisaalta perehtyminen kirjallisuuteen ja ajatusten pyörittely auttoi työstämään lopputyötä siten, että sen kirjoittaminen oli lopulta melko selkeä prosessi. Jaottelin kirjallisuuskatsauksen selkeäksi, aloitin ohjelmistotestauksesta ja sen merkityksestä ohjelmistosuunnittelussa. Etenin yksikkötesteihin ja edelleen automaattitesteihin ja pyrin käsittelemään niitä suhteessa päättötyöni aiheeseen. Perehdyin myös yliopistossa opettamiseen ja oppimisteorioihin.

Varsinaisessa työosassa olin valmistautunut tekemään automaattitestit kokonaan uusiksi molempiin kursseihin, mutta työnkuva muotoutuikin enemmän tulevien opiskelijoiden ymmärryksen lisäämiseen testaamisesta ja myös kurssin suorittamisen helpottamiseksi. Erityisesti korostin opiskelijoiden ymmärryksen vahvistamista testaamisesta. Projektin alussa mietin omakohtaista kokemusta ensimmäisestä ohjelmoinninkurssista ja tämän pohjalta yritin linjata ja selkeyttää tehtävänantoa ja vahvistaa sitä, että tehtävät toimisivat opiskelijoita aktivoivina ja oivallusta lisäävinä vahvistaen opiskelijoiden oppimista ja motivaatiota. Tein jokaiseen tehtävänantoon uuden kappaleen, jossa pyrin painottamaan ohjelmoinnin "pilkun tarkkaa” luonnetta ja myös kirjoitusmuotoja millä koodia kirjoitetaan. Tämä oivallus syntyi omien kokemusten perusteella, koska itselläni varsinkin alussa suurin osa koodin ongelmista tuli kirjoitusvirheistä. Tämä toteutustapa kuitenkin lisää luettavan tekstin määrää tehtävänannoissa, joka voi vaikuttaa myös siihen luetaanko tehtävänanto huolellisesti. Mutta tehtävänantoon voi tietysti palata, mikäli koodi ei mene läpi ja siten se voi vahvistaa oppimista kuitenkin. Suuressa osassa tehtävistä on kirjattu lähes sama virke, joka mielestäni myös korostaa sanoman tärkeyttä ja painottaa, että ongelma jossain koodin osassa ei välttämättä ainakaan toistu.

Uuden tehtävän suunnitteluun lähdin pohdinnan jälkeen opiskelijalähtöisyys edellä. Neuvoteltuani ohjaajani kanssa päädyin mielestäni hyvään ja järkevään ratkaisuun, joka poisti alkuperäisen ongelman, jossa testit itsessään saattoivat aiheuttaa ongelmia tai haasteita. Testien muokkaamisen jälkeen sain lopulta testit toimimaan Windows ja Linux-käyttöjärjestelmillä, edelleen vein muutokset GitHubiin. Laadin n. sivun mittaiset ohjeet testien virheiden lukemisesta. Pyrin yksinkertaistamaan ja selkeyttämään opiskelijalle, miten testit toimivat ja miten niistä luetaan tiedot. Tein näitä ohjeita vain yhden C#-testeille ja yhden Python-testeille, koska näkemykseni mukaan opiskelijan pitää itse osata soveltaa ohjeita erilaisissa testiskenaarioissa. Testit vaativat kuitenkin arvoja ja kertovat mitä ne odottavat ja näyttävät mitä ne ovat saaneet, joten yhdet ohjeet molemmille kielille riittävät.

Mielestäni lopputyöprojektini onnistui hyvin. Perehdyin aiheeseen laajasti ja pyrin kiteyttämään kirjallisuuskatsaukseen olennaisimmat asiat ja laatimaan automaattitestit ja niiden ohjeistukset opiskelijoille selkeiksi ja oppimista tukeviksi ja siten myös aktivoiviksi ja motivoiviksi. Valitettavasti mitään systemaattista dataa ei ole kerätty edellisiltä vuosilta siitä, kuinka turhautuneita opiskelijat ovat olleet koodia tehdessään. Jää nähtäväksi, auttavatko tekemäni muutokset huomattavasti opiskelijan oppimista koodaamisessa tulevaisuudessa. Edelleen pidän tärkeänä, että kursseilla pidettäisiin kokeita ja opiskelijoiden osaamista testataan. Nykypäivänä turhautunut opiskelija voi helposti vaikkapa ChatGPT:n avulla tehdä kurssitehtäviä. Mahdollisesti pistokokeet, joissa on

esimerkiksi puoli minuuttia aikaa valita monivalintakysymyksistä kysymyskentässä olevan koodin lopputulos, saattaisi olla hyvä tapa testata opiskelijan kooditaitoja. Tietysti opetetut asiat pitää olla linjassa arviointimenetelmien kanssa, jotta opiskelija tietää mitä häneltä odotetaan ja miten oppimistavoitteeseen pääsee parhaiten, sekä miten opitun tasoa arvioidaan. Selkeys ja linjakkuus luo motivaatiota korkeakouluopinnoissa.

Mielestäni onnistuin lopputyöprojektissani, vaikkakaan sen tuloksia ei mitata. Uskon, että projektini hyödyttää käytäntöjä Ohjelmoinnin perusteet-kurssilla.

5. LÄHTEET

Alexandra 2024. What is SDLC? Understand the Software Development Life Cyckle. WWW-blogi. https://stackify.com/what-is-sdlc/. Viitattu 31.5.2024.

ChatGPT 2024. OpenAI. GPT-3.5. Käytetty kielentarkistukseen, toukokuu 2024. https://chat.openai.com/

Eficode 2024. Robot Framework: Past, Present and Future. WWW-blogi. https://www.eficode.com/blog/en/blog/robot-framework Viitattu 31.5.2024

GitHub Copilot 2024, Github. Käytetty Koodin oikolukemiseen, sekä apuna ongelmissa, toukokuu 2024. https://github.com/features/copilot

Haltu 2023. Ohjelmistotestaus - Laadunvarmistuksen rooli ohjelmistokehityksessä. Haltu.fi. WWW-blogi. https://www.haltu.fi/blogi/ohjelmistotestaus. Viitattu 31.5.2024

Hamilton, T 2024. What is Integration Testing? WWW-blogi. https://guru99.com/integration-testing.html

Homes, B 2012. Testers and Code of Ethics 27-35 TeoksessaFundamentals of software testing. Hoboken, New Jersey: ISTE/Wiley.com. https://download.e-bookshelf.de/download/0000/7533/44/L-G-0000753344-0002285971.pdf

ISTQB 2016. Certified Tester Advanced Level Syllabus. Test Automation Engineer. International Software Testing Qualifications Board. Oppimateriaali ISTQB www-sivulla. https://istqb-main-web prod.s3.amazonaws.com/media/documents/ISTQB-CTTAE_Syllabus_v1.0_2016.pdf Viitattu 31.5.2024

ISTQB. 2023. Certified Tester Foundation Level Syllabus v4.0. International Software Testing Qualifications Board. Oppimateriaali ISTQB www-sivustolla. https://istqb-main webprod.s3.amazonaws.com/media/documents/ISTQB_CTFL_Syllabus-v4.0.pdf Viitattu 31.5.2024

ISTQB 2024. https://www.istqb.org/ Viitattu 31.5.2024

IEEE Standard Glossary of Software Engineering Terminology (610.12-1990) 1990. New York, USA.

Iivonen, J. Testaussanasto - ohjelmistotestauksen tärkeimmät termit selitettynä. WWW-blogi. Viitattu 31.5.2024. https://projecttop.com/testaussanasto/

JIRA Bug Life Cycle. N.d. Javatpoint.com. WWW-sivusto. https://www.javatpoint.com/jira-bug-life-cycle. Viitattu 31.5.2024.

Kasurinen, J 2013. Ohjelmistotestauksen käsikirja, Docendo.

Kukkurainen, ML 2019. Autoetnografia - päiväkirjaan perustuva tutkimus. LAMK Pro. [08.04.2024]. Saatavissa: http://www.lamkpub.fi/2019/01/04/autoetnografia---paivakirjaan-perustuva-tutkimus/

Lewis, WE. Software testing and continuous quality improvement. 2nd ed. CRC Press; 2004. https://tienhuong.wordpress.com/wp-content/uploads/2009/08/software-testing-and-continuous-quality-improvement-second-edition.pdf Viitattu 31.5.2024

Myers, G. J., Sandler, C., & Badgett, T 2011. Test-Case Design 41-85. Teoksessa The Art of Software Testing (3rd ed.), Wiley. https://malenezi.github.io/malenezi/SE401/Books/114-the-art-of-software-testing-3-edition.pdf Viitattu 31.5.2024

Mälkki, K. ja Mansikka-Aho, A 2020. Kasvatustieteilijän Taskutuutori - Raketti ajattelun avaruuteen. Kasvatustieteiden ja kulttuurin tiedekunta, Tampereen yliopisto.

Nevgi, A ja Lindblom-Ylänne, S 2009. Linjakasta opetusta ja oppimista yliopistossa 138-194, Oppimisen teoriaa ja käytäntöä 194-320. Teoksessa Yliopisto-opettajan käsikirja, WSOY, Helsinki 2009.

Porkka, J 2019. Koodarin päiväkirja. WWW-sivusto. https://www.theseus.fi/handle/10024/171931

Rana, K 2023. What is automation testing? ArtOfTesting. WWW-blogi. https://artoftesting.com/automation-testing.

Rehn, E. 2023. Mitä on testaus? Ohjelmistotestaus laadunvarmistajana. Gofore.com. WWW-blogi. Viitattu 31.5.2024. https://gofore.com/mita-on-testaus-ohjelmistotestaus-laadunvarmistajana/

Robot Framework 2024. N.d. WWW-sivusto. https://robotframework.org/ Viitattu 31.5.2024

Savonia AMK (https://www.savonia.fi/opiskele-tutkinto/tutkinnot-ja-hakeminen/amk-ja-yamk-tutkinnot-tarjonta/insinoori-amk-tietotekniikka-paivatoteutus/) viitatttu 31.5.2024.

StringBuilder Class. WWW-sivusto. https://learn.microsoft.com/en-us/dotnet/api/system.text.stringbuilder?view=net-8.0

Tutorialspoint.com 2024. WWW-sivusto. SDLC - Agile Model. N.d. https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm. Viitattu 31.5.2024

Tutorialspoint.com 2024. WWW-sivusto. SDLC - Waterfall Model. N.d. https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm Viitattu 31.5.2024

Vala group 2023. Järjestelmätestaus eli systeemitestaus: Mitä se on ja miksi se on tärkeää? WWW-blogi. https://www.valagroup.com/fi/blogi/jarjestelmatestaus/ Viitattu 31.5.2024

Vaskivuo, P. 2018. Päiväkirjamuotoinen opinnäytetyö: laskutusjärjestelmän käyttöönotto. Oulun ammatikorkeakoulu. WWW-sivusto. https://www.theseus.fi/handle/10024/142543

Vertics 2019. Ohjelmistotestauksen perusteet. Vertics.co. WWW-blogi. Viitattu 31.5.2024. https://vertics.co/blogi/ohjelmistotestauksen-perusteet.

Unadkat, J 2023. What is Test Driven Development (TDD)? Community Contributor - June 14, 2023. Rowserstack.com/guide.