Ohjelmistorobotiikan testauksen hyödylliset menetelmät

Diplomityö
Turun yliopisto
Tulevaisuuden teknologioiden laitos
Ohjelmistotekniikka
2019
Tatu Heinonen

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos

TATU HEINONEN: Ohjelmistorobotiikan testauksen hyödylliset menetelmät

Diplomityö, 76 s., 0 liites.
Ohjelmistotekniikka
Marraskuu 2019

Ohjelmistorobotiikka on muutaman vuoden aikana kehittynyt huomattavasti ja se on otettu osaksi monen yrityksen toimintaa. Ohjelmistorobotiikalla tarkoitetaan ohjelmaa, joka käyttää toista ohjelmaa sen käyttöliittymän kautta ja tekee toimenpiteitä oikean ihmisen tavoin. Ohjelmistoroboteilla saadaan nopeasti ja tehokkaasti parannettua vanhojen legacy-ohjelmistojen välistä käyttöä rajapintojen puutteesta huolimatta.

Testaus on aina oleellinen osa kehitystyötä ja ohjelmistokehityksessä sitä on tutkittu paljon. Testauksella saadaan parannettua ohjelmiston laatua ja luotettavuutta, joka lisää tuotteen arvoa. Ohjelmistokehityksessä testaukseen on kehitetty erilaisia tapoja, joiden avulla ohjelmistoa saadaan testattua eri näkökulmista ja koko ohjelma tulee kattavasti ja ankarasti testattua. Nykyään ohjelmistoa on käytössä jokaisessa elektronisessa laitteessa ympäri maailman, joten testaustakin tehdään laajasti.

Diplomityön tavoitteena on tutkia, miten ohjelmistorobotiikkaa saadaan testattua tehokkaasti. Työssä esitellään ohjelmistorobotiikan toiminta yleisesti ja keskitytään UiPath-robotiikkaohjelmiston käyttöön. Tämän jälkeen perehdytään ohjelmistokehityksen testausmenetelmiin ja tutkimuksessa vertaillaan ohjelmistorobotiikan testausta ohjelmistokehityksen testaukseen.

Työn tuloksena selvitettiin mitkä ohjelmistokehityksen testauksen menetelmistä ovat käyttökelpoisia ja hyödyllisiä ohjelmistorobotiikan testauksessa. Asiantuntijahaastatteluista saatiin selville, että kehitysympäristön puuttuminen, ohjelmistorobotin määritys ja automatisoitavan ohjelman tuomat haasteet ovat avainasemassa testauksen kannalta. Suoraa mallia ohjelmistorobotiikan testaukseen ei ole mahdollista tehdä, koska eri ohjelmistorobotiikkaprojektit voivat erota toisistaan paljon, mutta samoista ohjelmistokehityksen testausmenetelmistä on paljon hyötyä myös ohjelmistorobotiikan testauksessa.

Asiasanat: RPA, ohjelmistorobotiikka, testaus

Sisältö

Luku 1

Johdanto

Ohjelmistorobotiikka on kasvussa oleva teknologia, jolla pystytään tehostamaan nykyisiä prosesseja kustannustehokkaasti. Ohjelmistorobotiikalla tarkoitetaan tietokoneohjelmia, jotka tekevät toistuvia manuaalisia toimenpiteitä, joihin ihmisellä menisi huomattavan pitkä aika ja virhemarginaali olisi korkea. Ohjelmistorobotti pystyy toimimaan muiden järjestelmien välillä ilman erillistä rajapintaa toimien täsmälleen samalla tavalla kuin ihminenki toimisi näiden järjestelmien kanssa. Koska järjestelmien väliin ei tarvitse tehdä erillisiä rajapintoja, kehitystyö on nopeaa ja tehokasta.

Perinteisessä ohjelmistokehityksessä testausta voidaan tehdä yksinkertaisesti monella tasolla. Yksikkötestauksessa (engl. unit testing) pieniä komponentteja testataan yksinään, integraatiotestauksessa (engl. integration testing) näitä pieniä komponentteja testataan yhdessä ja näiden jälkeen järjestelmätestauksessa (engl. system testing) koko järjestelmää testataan kokonaisuutena. Testausta voidaan tehdä staattisesti (engl. static testing) oikolukemalla koodia ja varmistamalla, ettei ohjelmassa ole virheitä tai dynaamisesti (engl. dynamic testing) ajamalla ohjelmaa testimateriaalilla. Testaukseen voidaan ottaa myös erillinen testiryhmä testaamaan ohjelman eri ominaisuuksia joko valkoisen laatikon testauksella (engl. white box testing), jolloin testiryhmä tuntee ohjelman lähdekoodin ja funktiot tai mustan laatikon testauksella (engl. black box testing), jolloin testiryhmä käyttää ohjelmaa tietämättä sen sisällöstä mitään.

Ohjelmistorobotiikan testaamiseen tarvitaan runsaasti testimateriaalia, mutta erilaisia käyttötapauksia voi olla työlästä ajaa hallitusti läpi. Usein ohjelmistorobotit testataan tuotannossa todellisilla käyttötapauksilla, mutta virheiden ilmetessä työ joudutaan tekemään manuaalisesti erikseen, jolloin työskentely hidastuu.

Tämän työn tarkoituksena on löytää tehokas tapa testata ohjelmistorobotteja. Työssä tarkastellaan ensin ohjelmistorobotiikkaa, robottien luomista ja niiden hallinnointia, jotta voidaan muodostaa hyvä kuva robottiohjelmistoista ja voidaan määrittää mitkä osat ovat tärkeitä testauksen kannalta. Tämän jälkeen tarkastellaan ohjelmistokehityksen testausta sekä ohjelmistokehityksen testaukseen käytettyjä valmiita metodeja ja tapoja. Näiden pohjalta vertaillaan ohjelmistorobotiikan ja ohjelmistokehityksen testausta ja etsitään ohjelmistorobotiikalle käyttökelpoiset ja hyödylliset testausmenetelmät. Ohjelmistorobotiikan testausta varten tehtiin myös asiantuntijahaastatteluja, joiden avulla tunnistettiin erilaisia ongelmakohtia, joihin olisi hyvä testausta varten kiinnittää huomiota.

Luku 2

Ohjelmistorobotiikka

Tässä luvussa kerrotaan ohjelmistorobotiikan taustatietoa ohjelmistorobottien luomisesta ja niiden käyttämisestä. Lisäksi luvussa käsitellään pintapuolisesti ohjelmia robottien luomiseksi.

2.1 Mikä on RPA

Robottisella prosessien automatisoinnilla (engl. robotic process automation, RPA) tarkoitetaan ohjelmistorobotiikkaa. Ohjelmistorobotiikalla automatisoidaan datan käsittelyyn tarkoitettuja toimenpiteitä, joihin ihmisellä menisi huomattava määrä aikaa ja jotka ovat pääasiassa saman asian toistamista. Ohjelmistorobotit eivät ole käveleviä robotteja, jotka puhuvat ja tekevät ihmismäisesti työtä. Ne ovat tietokoneohjelmia, jotka käsittelevät dataa tekemällä sille erilaisia toimenpiteitä ja tallentavat ne muihin järjestelmiin tai ymmärrettävämpään muotoon. Ohjelmistorobotit eivät kykene itsenäisiin päätöksiin eivätkä ne osaa käsitellä kuvia tai papereita ja ne tekevät täsmälleen sen, mitä niille ohjelmoidaan ja käsittelevät vain sähköisessä muodossa olevaa dataa. Tällöin datan, jota ohjelmistorobotti käsittelee, tulee olla jo kerran digitalisoitua.

2.1.1 Erilaisia automaatiotekniikoita

Automaatiota voidaan rakentaa monessa muodossa. Perinteinen sovellus tai ohjelma pystyy tekemään toistuvia tehtäviä nopeasti ja tehokkaasti, vaikka sellaista harva suoraan laskee robotiksi. Ohjelmien sarjatuotannot (engl. batch) ovat olleet pitkään käytössä ja niiden avulla voidaan suorittaa ketjutettuna useampi toimenpide. [1]

Esimerkki 1. Monet kuvankäsittelyohjelmat kykenevät käsit-telemään useita kuvia nopeasti sarjatuotannolla. Sarjatuotannolle voidaan määritellä tarkat askeleet jokaisen kuvan käsittelyyn. Näitä askeleita voivat olla esimerkiksi kuvan kääntäminen 90 astetta, väritasapai-non automaattinen käsittely ja kuvan pienentäminen tiettyyn pikselikokoon. Sarjatuotantoon voidaan laittaa esimerkiksi 100 kuvaa, jolloin kuvankäsittelyohjelma tekee kaikille 100 kuvalle samat toimenpiteet nopeuttaen kuvankäsittelyprosessia yksittäiseen, manuaalisen käsittelyyn verrattuna.

Ohjelmistorobotti on virtuaalista työvoimaa, joka käyttää järjestelmiä samalla tavalla kuin ihminen. Ohjelmistorobotti ei vaadi erillistä integraatiota eri järjestelmien välillä, sillä robotin saa ohjelmoitua siirtymään datan kanssa järjestelmästä toiseen käyttäen useaa eri järjestelmää samaan aikaan. Eri selainautomaatiosovelluksilla (engl. browser automation) voidaan lukea nettisivuilta tekstiä, täyttää lomakekenttiä tai muuttaa koko sivun ulkonäköä. Vastaavasti työpöytäautomaatiolla (engl. desktop automation) pystyy tietoja käsittelemään sovellusten välillä. [1]

Esimerkki 2. Työpöytäautomaatiolla voidaan avata sovellus, tehdä sillä toimenpiteitä ja tallentaa tuloksia jatkokäsittelyä varten. Laskin-sovellus voidaan avata käyttäen absoluuttista polkua sovelluksen exe-tiedostoon, laskee yhteenlaskun 3 + 5 klikkaamalla nappeja 3, + ja 5, kopioi summan leikepöydälle ja palauttaa sen toista toimenpidettä varten.

Käytännössä ohjelmistorobotti poimii tiedon yhdestä järjestelmästä ja täyttää sen toiseen järjestelmään. Välissä voidaan tehdä muotoilua tai muuta datan muovaamista. Saman robotin pystyy ohjelmoimaan siten, että se osaa tunnistaa saamastaan datasta järjestelmät,

joihin dataa halutaan tallentaa. Tällöin saman robotin saa tekemään toimenpiteitä useisiin eri järjestelmiin sujuvasti, vaikka sen saama data muuttuu. Kuvassa 2.1 on esitetty eri automaatiot jaoteltuina yksinkertaisimmista, sovellusten sisällä yksittäisiä toimenpiteitä helpottavista tekniikoista, monimutkaisempiin tekniikoihin. Toisessa portaassa tekniikat ovat sovellusten välisiä automaatiotekniikoita, kuten selain- ja työpöytäautomaatio sekä tietokantojen välisiä rajapinta-automaatioita. Kolmannen portaan tekniikoita ovat avustava robotiikka (engl. assisted RPA) ja avustamaton robotiikka (engl. unattended RPA). Näillä tarkoitetaan ihmisen käsin ohjaamaa robottia tai itsenäisesti toimivaa robottia, joka kuuntelee ja odottaa tehtäviä suoritettavaksi. Viimeisessä portaassa on tekoälyllinen robotti, joka oppii tekemään itsenäisesti valintoja ja pystyy ennakoimaan asioita. Tässä työssä keskitytään työpöytäautomaatioon sekä vartioimattomaan robotiikkaan. [1]

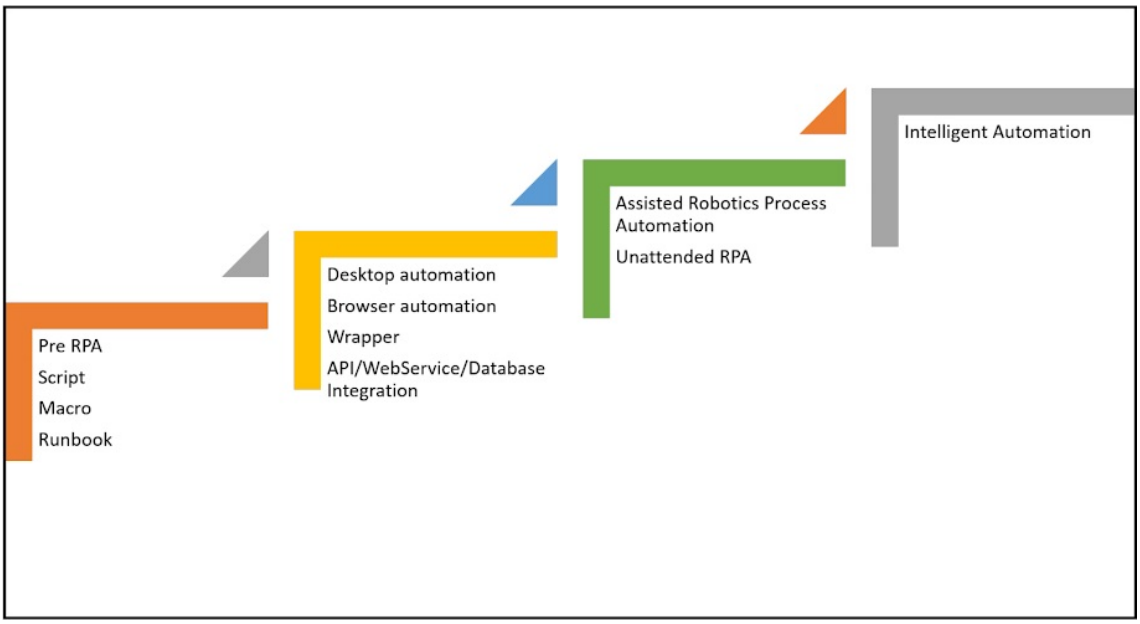
2.1.2 Makrot ja komentosarjat

Makrolla tarkoitetaan pientä tietokoneohjelmaa, joka on ohjelmoitu tapahtumaan näppäinyhdistelmällä. Makro on kuin ohjelmistorobotti, mutta huomattavasti pienemmässä mittakaavassa ja usein niitä luodaan muiden tietokoneohjelmien sisällä. Komentosarjat (engl. script) ovat kuin makrot, mutta usein ne ajetaan suoraan kirjoittamisen jälkeen. Komentosarjalla voidaan ajaa muita makroja samalla.

Makroilla ja komentosarjoilla on sama käyttötarkoitus kuin ohjelmistoroboteilla. Niitä luodaan nopeuttamaan ja helpottamaan datan käsittelyä, muotoilua ja tallentamista. Toisin kuin ohjelmistorobotit ne toimivat kuitenkin vain yhden tietokoneohjelman sisällä.

2.2 Mitä kannattaa automatisoida?

On eri asia mitä voidaan automatisoida ja mitä pitäisi automatisoida. Automatisoinnilla usein pyritään helpottamaan ihmisten manuaalisesti tekemää työtä, joten seuraavia asioita pitäisi automatisoida:



Kuva 2.1: Automaation eri tasot. [1]

- Toistuvat askeleet
- Aikaa vievät askeleet
- Toimenpiteet, joilla on korkea virhemarginaali
- Toimenpiteet, joiden tuotos on matalalaatuista
- Toimenpiteet, joissa on mukana useita ihmisiä ja paljon askeleita

Näin yksinkertaisilla edellä kuvatuilla ohjeilla automatisointi voi kuitenkin olla vaikeaa ja tästä syystä automatisoitavilla asioilla tulee olla tiettyjä ominaisuuksia:

- Hyvin määritellyt ja säännönmukaiset askeleet
- Loogiset askeleet
- Mahdollisuus kääntää yksittäiset tehtävät ohjelmistoon
- Automatisoinnin hyödyt ovat korkeammat kuin kustannukset [1]

RPA:n pohjimmainen käyttötarkoitus on nopeuttaa yksinkertaisia toimenpiteitä ja vähentää virhemarginaalia. Se parantaa organisaatioiden tehokkuutta ja helpottaa työntekijöiden perehdytystä, kun robotti pystyy tekemään työtä usean järjestelmän kautta ja työntekijän täytyisi hallita vain yksi järjestelmä.

Esimerkki 3. Uuden luottotilin avaaminen vanhalle asiakkaalle pankkiin. Asiakas soittaa puhelinpalveluun, jossa asiakaspalvelija ottaa asiakkaan tiedot ylös. Asiakaspalvelija tallentaa tiedot ja lähettää työn eteenpäin. Tästä työ siirtyy myyntiavustajalle, jolla menee noin 20 minuuttia uuden tilin avaamiseen. Jos myyntiavustajalla on paljon töitä rästissä, tämän käsittely saattaa siirtyä parin arkipäivän päähän. Myyntiavustaja avaa työpyynnön ja käyttää puhelinasiakaspalvelijan täyttämiä tietoja ja hakee asiakastietojärjestelmästä asiakkaan tiedot. Tämän jälkeen myyntiavustaja kirjautuu uuden tilin avaamista varten omaan järjestelmään, johon hän kopioi tiedot näistä kahdesta muusta järjestelmästä. Myyntiavustaja avaa vielä neljännen järjestelmän, joka on nettisivu, tarkistaakseen luoton takauksen verotusarvon ja täyttää sinne asiakkaan tiedot. Tästä järjestelmästä myyntiavustaja kopioi taas tietoja uuden tilin avaamista varten. Tähän operaatioon on mennyt kahdelta työntekijältä noin 30 minuuttia työaikaa, muutama arkipäivä ja työhön on käytetty neljää eri järjestelmää. Kaikki nämä yksittäiset kohdat tietojen keräämisen jälkeen voidaan automatisoida siten, että ohjelmistorobotti kirjautuu eri järjestelmiin, kerää tiedot, täyttää ne uuden tilin avaamista varten ja lähettää tiedot asiakkaalle vahvistusta varten. Tämä kaikki voisi tapahtua heti, kun puhelinasiakaspalvelija on kerännyt asiakkaalta tiedot ja lähettänyt työpyynnön, jolloin operaation kokonaiskesto jää alle minuuttiin. [2]

Ohjelmistorobotiikalla korvataan manuaalista tietojenkäsittelytyötä, kuten lomakkeiden täyttöä, datan kopiointia ja tietojen muokkaamista, jota ollaan tehty käsin jo vuosikymmeniä. Kun näitä samoja toimenpiteitä jatketaan useita tunteja, päiviä tai tehdään täyspäiväisesti jatkuvasti, työntekijän motivaatio laskee, työtahti hidastuu ja virhemarginaali kasvaa. Virhemarginaali kasvaa myös silloin, kun samaa toimenpidettä tehdään vain harvoin eikä aivan muisteta, miten toimenpide pitää suorittaa.

Esimerkin 3 tapauksesta voitiin tunnistaa monta vaihetta, jossa tietoa kopioitiin kentästä toiseen. Joka kerta, kun tietoja kopioidaan manuaalisesti ja työtä tehdään pitkiä aikoja, virhemarginaali on huomattavan iso. Luottotiliesimerkissä virhemarginaali oli noin 10 prosenttia manuaalisesti tehtynä, mutta ohjelmistorobotilla tällaista ei ole lainkaan. [2]

2.3 Ohjelmistot

Ohjelmistorobotteja voidaan luoda monilla eri ohjelmilla. Pääpiirteittäin ohjelmat ovat samanlaisia, mutta eroja tulee esimerkiksi ohjelmoinnin tarpeesta verrattuna valmiiden kaavioiden käyttämiseen. Sovelluksista löytyy usein samankaltaiset komponentit robottien luomista, ajamista ja hallintaa varten. [1]

Nauhoittimen (engl. recorder) avulla robotteja voidaan määrittää hiiren ja näppäimistön avulla. Nauhoitin laitetaan päälle, hiirellä ja näppäimistöllä tehdään halutut toimenpiteet ja nauhoitin nauhoittaa tehdyt askeleet. Tämän jälkeen nämä samat askeleet pystytään toistamaan nopeasti useita kertoja. [1]

Kehitysstudiolla (engl. development studio) robotteja voidaan määrittää tarkemmin. Osassa sovelluksissa määrityksiä voidaan tehdä kaavioilla (engl. flowchart), mutta määritykset voidaan tehdä myös ohjelmoimalla. Tällöin perinteiset toimenpiteet ja funktiot, kuten silmukat, jos-lauseet ja muuttujat ovat tärkeitä ja näitä käytetäänkin paljon robottien määrityksessä. [1]

Robottia voidaan hallinnoida ja sen toimintaa seurata valvontapaneelista. Robotti voidaan kytkeä päälle tai pois, niille voidaan tehdä aikatauluja ja niiden tehtäviä voidaan vaihtaa tai ylläpitää. Lisäksi lukuisilla lisäosilla sovelluksen toimintaa voidaan parantaa tai laajentaa. [1]

Luku 3

Ohjelmistot robottien luomiseksi

Tässä luvussa esitellään tarkemmin UiPath-ohjelmistoa robottien luomiseksi. UiPath on suosittu ja aloittelijaystävällinen ohjelmisto, josta löytyy ilmainen yhteisöversio. Luvun lopussa esitellään myös muita ohjelmistorobotiikkasovelluksia, kuten BluePrismiä, Automation Anywhereä ja SikuliXia. Näistä Blue Prism ja Automation Anywhere ovat samankaltaisia, kokonaisia robotiikkaohjelmistoja kuin UiPath ja SikuliX on avoimen lähdekoodin nauhoitintyökalu.

3.1 UiPath

UiPath on nykyään yksi käytetyimmistä RPA-ohjelmistoista. UiPathissä on kolme erillistä sovellusta, joilla jokaisella on oma tarkoituksensa robottien luomiseksi ja hallitsemiseksi. [3]

3.1.1 UiPath Studio

UiPath Studiolla luodaan prosessit ja toimenpiteet, joita robottien halutaan suorittavan. Ohjelmistosta löytyy visuaalinen käyttöliittymä, joten robottien luominen ei vaadi käyttäjältä mitään ohjelmointitaitoja. UiPath Studiossa olevalla nauhoittimella halutut toimenpiteet pystyy tallentamaan helposti ja robotin saa toimimaan täsmälleen halutulla tavalla.

Nauhoitettuun ohjelmaan voi tämän jälkeen lisätä muita operaatioita valmiista kirjastosta. [4]

3.1.2 UiPath Robot

UiPath Robotilla käynnistetään aiemmat Studiolla tehdyt prosessit käytäntöön. Itse robotti luodaan siis UiPath Robotilla. Robotit voivat olla joko avustavia robotteja (engl. attended robot) tai avustamattomia robotteja (engl. unattended robot). Avustavalla robotilla tarkoitetaan sitä, että robotille annetaan erilinen käsky tehtävän suorittamiseksi ja se suorittaa toimenpiteitä hallitusti. Avustamaton robotti odottaa tietyn laukaisimen aktivoivan sen ja tekee sitten toimenpiteensä itsenäisesti. [4]

3.1.3 UiPath Orchestrator

UiPath Orchestrator on hallintasovellus Robotin luomille ohjelmistoroboteille. Yksittäisen robotin hallitsemiseksi Orchestratoria ei tarvita, mutta kun robottien määrä kasvaa ja käytön tarve muuttuu, Orchestrator auttaa työnkulun sujuvana pitämisessä. Orchestrator ryhmittelee robotit jonoihin ja niiden toimintaa voidaan priorisoida lennosta. Orchestrator tallentaa robottien toiminnasta dataa, jotta robottien työskentelyä voidaan seurata. [4]

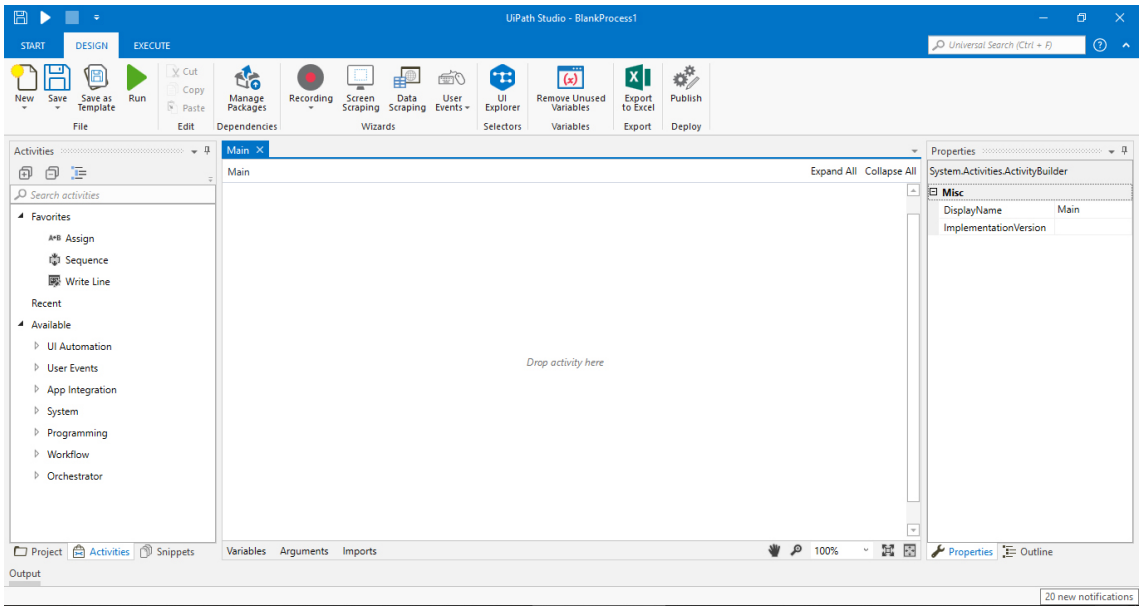
3.2 Robottien luominen UiPathilla

UiPathissa on neljä eri prosessityyppiä: sarja (engl. sequence), prosessikaavio (engl. flowchart), avustaja (engl. assistant) ja tilakone (engl. state machine). Prosessityypeistä sarja on pienin ja se on hyvä yksinkertaisille tehtäville. Sarjassa voi olla useita yksittäisiä aktiviteetteja, joita on helppo seurata ja joiden avulla voidaan etsiä virheitä (engl. debug). Sarjassa aktiviteetit tapahtuvat lineaarisessa järjestyksessä yksi toisensa jälkeen. Prosessikaaviolla voidaan luoda monimutkaisempia projekteja ja siihen voidaan yhdistää aktiviteetteja. Avustajaa käytetään, kun on tarkoitus luoda avustavia robotteja auttamaan työnteossa.

Tilakonetta käytetään suuriin projekteihin, joissa käytetään useita eri tiloja, jotka laukeavat tietyn ehdon täyttyessä. [1]

UiPathissä uuden projektin luomisen voi aloittaa joko tyhjästä (engl. blank project) tai käyttämällä valmiita pohjia (engl. template). Valmista pohjaa käyttämällä voidaan valita joko liiketoiminnallinen prosessi ja aloittaa luomalla prosessikaavio, luoda avustaja reagoimaan hiiren tai näppäimistön painallukseen tai lähteä rakentamaan suurta yritysjärjestelmää. Yksinkertaisinta on kuitenkin valita tyhjä pohja ja alkaa rakentaa siihen uutta kokonaisuutta pienistä osista. [1]

UiPathin perusnäkymä on selkeä ja helppo ymmärtää. Sivun yläreunassa olevassa nauhassa on perustyökalut kuten uusi projekti, tallennus ja ajo, mutta näiden lisäksi myös nauhoitus ja tietojen kerääminen (engl. data scraping). Vasemmasta reunasta löytyy aktiviteetti-ikkuna (engl. activities panel). Aktiviteetteja löytyy satoja erilaisia ja niiden käyttäminen on helpointa etsimällä aktiviteetin nimellä käyttäen hakukenttää. Aktiviteetti vedetään aktiviteetti-ikkunasta keskelle suunnitteluikkunaan (engl. designer panel). Suunnitteluikkuna on kuin työpöytä, jossa eri aktiviteetteja, työnkulkua ja prosesseja saa hallittua. Oikeassa reunassa on ominaisuusikkuna (engl. properties panel), josta saa aktiviteettien ja muiden komponenttien ominaisuuksia muutettua. Alareunassa vasemmalla olevista välilehdistä aktiviteetti-ikkunan tilalle saa näkymään projektin kokonaiskuvan tai listan pienistä, valmiista ohjelman osista (engl. snippet). Keskellä olevista välilehdistä saa näkymiin työnkulun muuttujat (engl. variable), argumentit (engl. argument) ja ulkopuolelta tuodut komponentit (engl. import). Oikean reunan välilehdistä saa näkyviin projektin pääpiirteet (engl. outline). (kuva 3.1) [1]



Kuva 3.1: UiPathin perusnäkymä. Yläreunassa perustyökalut, vasemmassa reunassa aktiviteetti-ikkuna ja oikeassa ominaisuusikkuna.

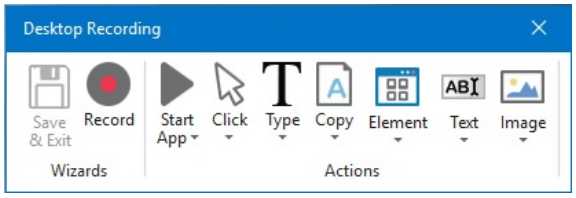
3.2.1 Nauhoittimen käyttö

Robotteja voi luoda UiPath Studiolla kahdella tavalla: nauhoittamalla tai vetämällä aktiviteetteja yksitellen suunnitteluikkunaan. Usein nauhoittamalla ei saa suoraan toimivaa ohjelmaa, koska nauhoitin ei välttämättä tunnista kaikkia hiiren tai näppäimistön komentoja, vaan näitä aktiviteetteja joutuu jälkikäteen lisäämään ohjelmaan. [1]

UiPathissä on viisi erilaista nauhoitinta: yksinkertainen (engl. basic), työpöytä, web, kuva ja Citrix. Yksinkertaisella nauhoittimella on tarkoitus nauhoittaa yksittäisiä aktiviteetteja työpöydällä. Nämä aktiviteetit eivät ole erillisen säiliön sisällä, vaan omina komponentteinaan. Työpöytänauhoittimella nauhoitetaan myös työpöydän aktiviteetteja, mutta sillä voidaan nauhoittaa suurempia kokonaisuuksia ja jokainen näistä aktiviteeteista laitetaan erillisen attach window -komponentin sisälle. Web-nauhoitin luo myös erillisen säiliön aktiviteetteja varten ja käyttää oletuksena simulate type/click -tekstinsyöttömetodia, joka on nopeampi kuin normaali oletus- tai SendWindowMessages-tekstinsyöttömetodit. Näiden erona on miten tekstin syöttö suoritetaan taustalla ja miten se tukee näppäimistön

pikanäppäinten käyttöä toimenpiteiden suorittamiseksi. Kuvanauhoitinta käytetään virtuaaliympäristöjen, kuten VNC ja Citrix kanssa. Kuvanauhoitin hyväksyy vain kuva-, teksti- tai näppäimistöautomaa-tion. Citrix -nauhoitin on vastaava kuin työpöytänauhoitin, mutta sitä joudutaan käyttä-mään etähallintatyökalujen kanssa, koska yksinkertainen ja työpöytänauhoitin eivät toimi näiden kanssa. RDP-ympäristöissä (remote desktop protocol, suom. etätyöprotokolla) data lähetetään kuvina, joita yksinkertainen ja työpöytänauhoitin eivät tunnista. Citrix-nauhoittimessa on click text- ja click image -aktiviteetit tätä varten. [1, 5]

Esimerkki 4. Yksinkertaisen ohjelman luominen työpöytänauhoittimen avulla. Robotin tarkoituksena on laskea yksinkertainen laskutoimitus laskin-sovelluksella. UiPathin perustyökaluista valitaan Recording ja pudotusvalikosta Desktop. UiPath pienentää Studion ja avaa pienen Desktop Recording -ikkunan (kuva 3.2). Myös muut nauhoitinvaihtoehdot avaavat vastaavanlaisen ikkunan, jossa on samat työkalut. Poikkeuksena tähän on kuvanauhoitin, jossa on erilaisia kuvan valitsemiseen ja sisällöntunnis-tamiseen tarkoitettuja työkaluja. Myös Citrix -nauhoitin on hieman erilainen Citrix-työka-lujen osalta. Kuvan 3.2 mukaan klikataan Record-napista, joka aloittaa nauhoituksen.

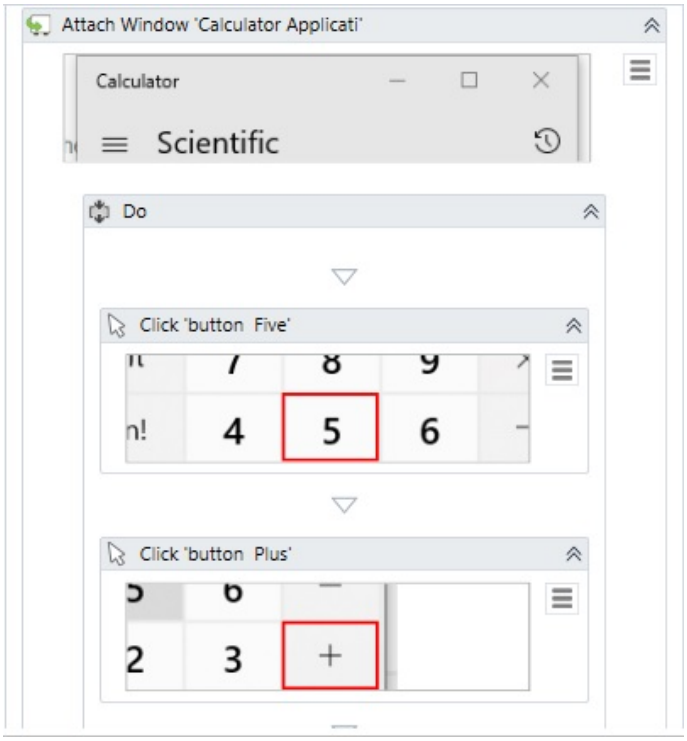


Kuva 3.2: Työpöytänauhoitin. Työpöytänauhoittimesta löytyy erilaisia työkaluja, joilla operaatioita voi nauhoittaa.

Esimerkkiä varten kiinnitin Laskimen Windowsin alareunassa olevaan tehtäväpalkkiin, jotta Laskin-sovelluksen avaaminen on helppoa. Laskimesta painetaan nappeja 5, +, 3 ja =. Esc-nappia näppäimis-töstä painamalla saa nauhoitinikkunan auki, jotta siitä voidaan valita Copy ja screen scraping summan kopioimiseksi. Klikataan laskimesta summan kohdalle ja ruutuun avautuu uusi ikkuna tietojen kaapimiseksi. Valitaan kaavintatavaksi teks-

titunnistus, eli OCR (engl. optical character recognition) ja valitaan Finish. Tämä toimenpide kerää ruudulta dataa, jonka voi myöhemmin asettaa aktiviteetin ulostulokohtaan (engl. output) datan käyttöä varten. Tämän jälkeen nauhoitus on valmis ja valitaan Save & Exit, jotta UiPath näyttää suunnitteluikkunan.

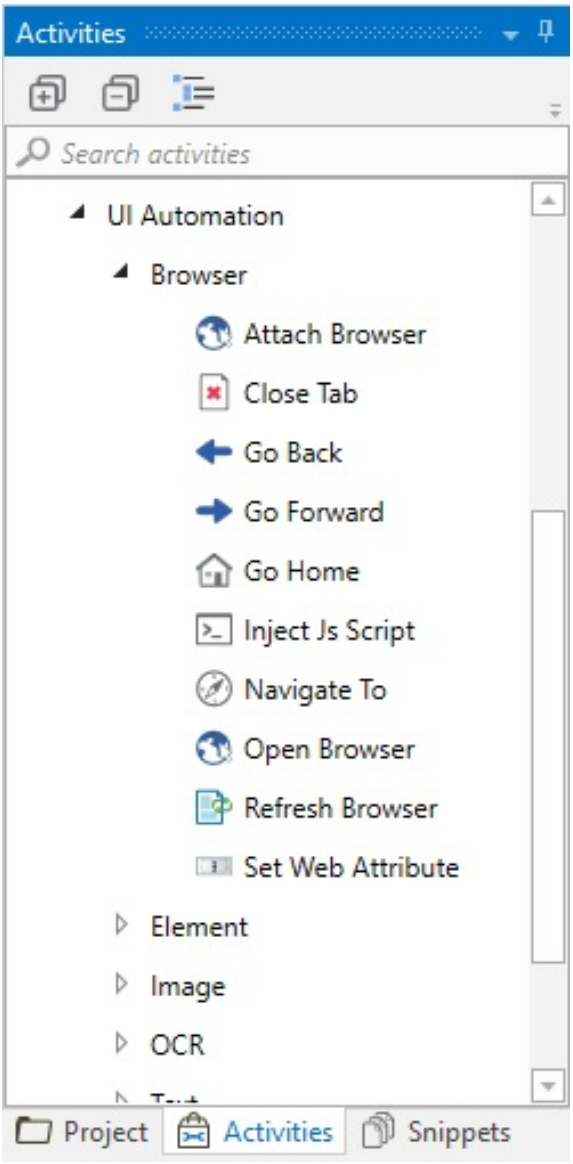
Suunnitteluikkunassa nähdään kuinka nauhoitin on luonut säiliön Desktop, jonka sisällä on attach window -elementti. Tämän sisällä on kaikki ne operaatiot, jotka nauhoituksen aikana tehtiin yksitellen (kuva 3.3). UiPathin yläreunasta painetaan Run-painiketta, jolloin ohjelma suoritetaan ja ruudulla voi nähdä, kuinka robotti tekee jokaisen operaation erikseen. Pelkällä laskutoimituksen suorittamisella ei välttämättä ole suurta hyötyarvoa, mutta laskimen käyttöä robotin avulla voidaan hyödyntää esimerkiksi laskutuksessa eri tuotteiden summan laskemiseksi ja laskun lähettämiseksi automaattisesti.



Kuva 3.3: UiPathin suunnitteluikkuna. UiPath on luonut säiliön ja sen sisällä kaikki operaatiot erikseen.

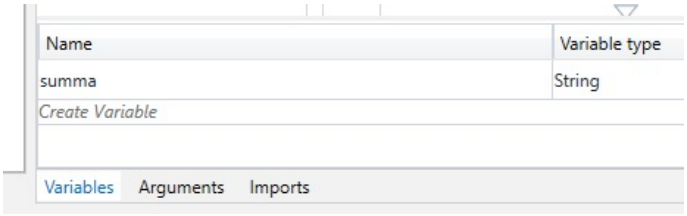
3.2.2 Aktiviteetti-ikkunan käyttö

Usein ohjelmista ei tule suoraan valmiita käyttäen pelkästään nauhoitinta, koska nauhoitin ei tunnista erilaisia painalluksia tai näppäimistökomentoja. Tästä syystä aktiviteetteja voi vetää myös suoraan aktiviteetti-ikkunasta suunnitteluikkunaan. Nauhoittimella ei myöskään pysty tekemään kaikkia operaatioita kerätyllä datalla. Aktiviteetti-ikkunasta löytyy erilaisia toimenpiteitä, joita esitellään kappaleesta 3.3 eteenpäin. (kuva 3.4) [1]



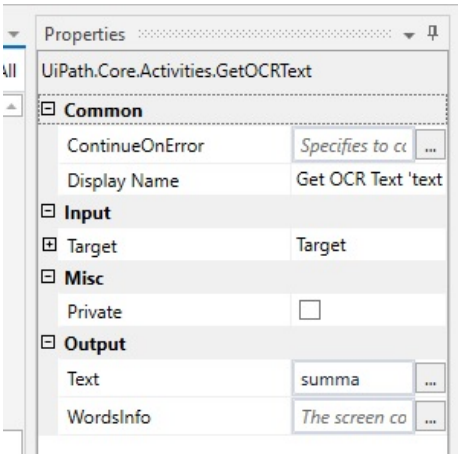
Kuva 3.4: Aktiviteetti-ikkuna. Aktiviteetti-ikkunassa löytyy erilaisia aktiviteetteja laidasta laitaan.

Esimerkki 5. Esimerkissä 4 tehtiin vain yksinkertainen laskutoimitus, mutta tätä dataa ei käytetty tai tallennettu mihinkään. Jatketaan tätä esimerkkiä käyttämällä sen keräämää dataa. Esimerkissä 4 summa kopioitiin Screen scraping -operaatiolla. Luodaan muuttujaikkunaan uusi muuttuja nimeltä `summa` (kuva 3.5).

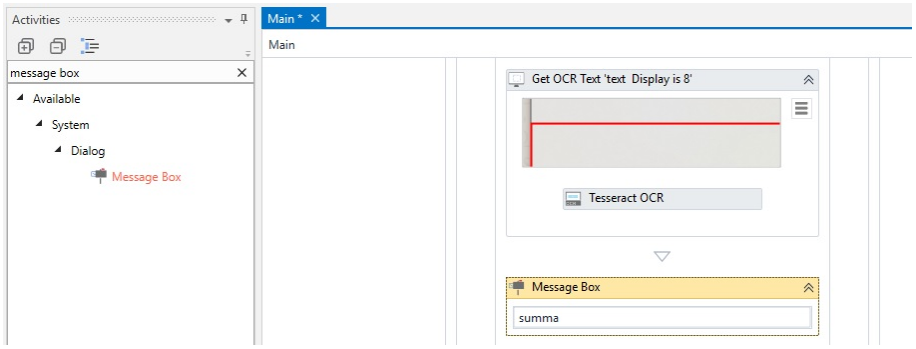


Kuva 3.5: Muuttujan luominen. Aluksi luodaan uusi muuttuja `summa` muuttujaikkunassa.

Asetetaan Screen scraping -operaation tuotokseksi (engl. output) tämä muuttuja ominaisuusikkunassa (kuva 3.6). Tämän jälkeen etsitään aktiviteetti-ikkunasta viestilaatikko (engl. message box) ja vedetään se viimeiseksi operaatioksi säiliön sisälle. Viestilaatikon sisällöksi kirjoitetaan muuttuja `summa`, jolloin robotti viimeiseksi tulostaa laskutoimituksen summan ruudulle viestilaatikkoon.



Kuva 3.6: Ominaisuuksien muuttaminen. Ominaisuusikkunasta saa muutettua tuotoksen kohteen.



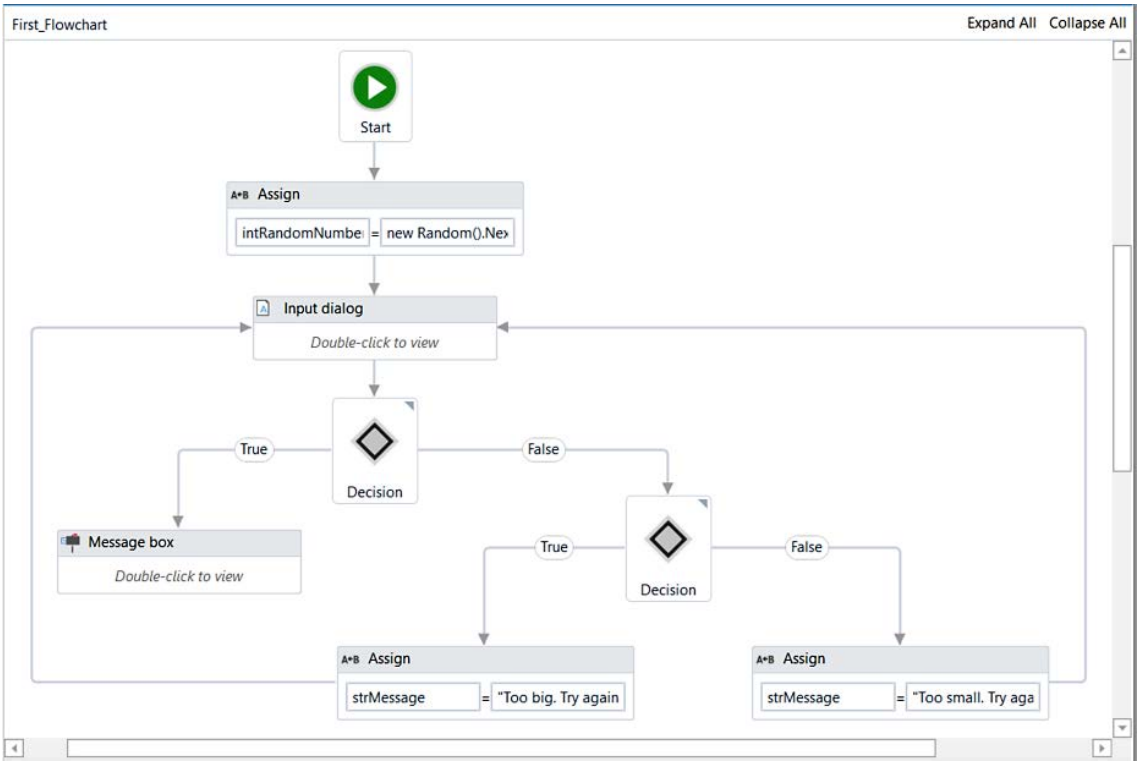
Kuva 3.7: Viestilaatikon käyttäminen. Aktiviteetti-ikkunasta on otettu viestilaatikko.

3.3 Prosessikaaviot ja prosessivirtauksen hallinta

Prosessikaavio on säiliö, jonka sisään voi asettaa aktiviteetteja. Prosessikaavion pääidea on komponenttien modulaarisuus, jotta niitä voidaan käyttää myöhemmin myös muissa paikoissa samassa projektissa tai kokonaan muissa projekteissa. Prosessikaavioita voidaan käyttää myös sarjojen sisällä. Prosessikaavio voidaan viedä tiedostoon, jotta se voidaan tuoda helposti toisiin projekteihin. Prosessivirtausta (engl. control flow) voi hallita monin tavoin prosessikaavion sisällä. Prosessikaavion sisälle voi tehdä erilaisia loogisia operaatioita, joiden avulla robotin toiminta haarautuu tekemään jotain muuta. [1]

Esimerkki 6. Kuvassa 3.8 prosessikaavion sisälle on määritetty alkuun aloitussolmu (engl. start node), joka on yhdistetty *assign*-aktiviteettiin. Tässä *assign*-aktiviteetissa määritellään muuttuja `intRandomNumber` ja sille arvoksi satunainen kokonaisluku. *Input dialog* -aktiviteetilla voidaan käyttäjältä kysyä syötettä ja tässä esimerkissä käyttäjää pyydetään arvaamaan satunainen luku. Ensimmäinen *decision*-aktiviteetti tarkistaa oliko luku oikein ja antaa oikeasta arvauksesta tekstilaatikon. Väärästä vastauksesta prosessi jatkuu toiseen *decision*-aktiviteettiin, jossa tarkistetaan oliko annettu syöte suurempi vai pienempi kuin satunainen luku ja määritetään muuttujan `strMessage` arvoksi viesti joko liian suuresta tai pienestä arvauksesta. Tämän jälkeen prosessikulku ohjaa takaisin syötteen an-

tamiseen ja käyttäjä saa arvata satunaislukua uudelleen. Ohjelman suoritus loppuu vasta oikeaan arvaukseen. [1]



Kuva 3.8: Prosessikaavio, jonka sisällä on erilaisia aktiviteetteja. [1]

Prosessivirtauksen hallintaan on olemassa myös useita muita aktiviteetteja. Prosessivirtauksen hallinnalla tarkoitetaan prosessin tapahtumien etenemisen, järjestyksen tai ajoituksen hallitsemista. Nämä ovat ohjelmistokehityksestä tuttuja toimenpiteitä, joilla esimerkiksi toistetaan jotain ohjelman osaa useita kertoja. [1]

UiPathissa näitä pystyy muuttamaan seuraavilla aktiviteeteilla:

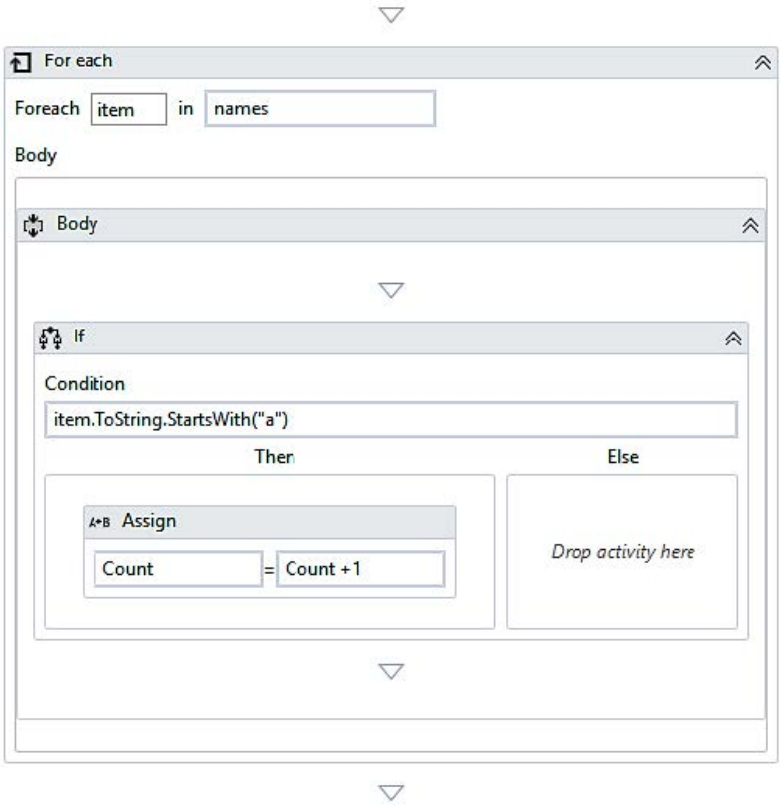
- Assign-aktiviteetti
- Delay-aktiviteetti
- Break-aktiviteetti
- While-aktiviteetti

- Do While -aktiviteetti
- For each -aktiviteetti
- If-aktiviteetti
- Switch-aktiviteetti [1]

Assign-aktiviteetillä pystytään määrittämään muuttujan arvo. Tämä on hyödyllinen aktiviteetti prosessivirtauksen hallinnan kannalta, koska sillä pystytään ohjaamaan esimerkiksi silmukan toimintaa. *Delay*-aktiviteetilla robotin toimintaa voidaan hidastaa tai viivyttää jotain toista toimenpidettä, esimerkiksi raskaan sovelluksen avaamista varten. *Break*-aktiviteetti pysäyttää For each -silmukan toistamisen, jotta ohjelma voi jatkaa seuraavaan aktiviteettiin heti. *While*-aktiviteetti tarkistaa sille määritetyn ehdon ja jos ehto on tosi, se toistaa silmukan sisällä olevia aktiviteetteja niin kauan, kunnes ehto ei enää täyty. *Do While* -aktiviteetti on samanlainen kuin while-aktiviteetti, mutta siinä ehto tarkistetaan vasta silmukan kierroksen jälkeen. Jos ehto täyttyy, silmukka suoritetaan uudelleen ja suoritus loppuu, kun kierroksen jälkeen ehto ei enää täyty. *For each* -aktiviteetti on myös silmukka, mutta se iteroi jokaisen listan elementin kerran läpi yksi kerrallaan ja tekee jokaisella kierroksella samat toimenpiteet, jotka silmukan sisälle on määritetty. If-aktiviteetilla voidaan tarkistaa toteutuuko jokin ehto vai ei. Jos ehto toteutuu, tehdään ensimmäiset toimenpiteet ja jos ehto on epätosi, tehdään toiset toimenpiteet. Tämä on vastaava kuin decision-aktiviteetti, mutta siinä prosessivirtaus ei siirry eri reitille, vaan if-aktiviteetin sisällä suoritetaan pelkästään eri toimenpiteet. *Switch*-aktiviteetti on vastaava kuin if-aktiviteetti sillä erolla, että siinä eri vaihtoehtoja voi lisätä haluamansa määrän. Sillä voidaan tehdä eri skenaarioita ja jokaiselle skenaariolle omat toimenpiteensä. [1]

Esimerkki 7. Listasta halutaan käydä kaikki nimet läpi ja laskea, kuinka moni niistä alkaa A-kirjaimella. Robotin ohjelmointi aloitetaan *For each* -aktiviteetilla. *For each* -aktiviteetin sisällä on ensin kaksi kenttää, joista ensimmäinen on muuttuja, johon jälkimmäisen kentän muuttuja (lista) halutaan jakaa. Tässä tilanteessa lista `names` jaetaan `item-`

muuttujiksi ja jokainen `item` käydään erikseen läpi. Näiden kenttien alla on aktiviteetin `body`-osa, jossa toimenpiteet näille läpikäytäville muuttujille tehdään. `Body`-osan sisälle on laitettu if-aktiviteetti, jonka ehtona on `item`-muuttujan, joka muutetaan String-tyyppiseksi muuttujaksi, ensimmäinen kirjain on ”a”. Jos tämä toteutuu, muuttujan `count` arvoon lisätään yksi. (kuva 3.9) Tämä ohjelma käy koko listan `names` läpi ja joka kerta, kun muuttujan `item` ensimmäinen kirjain on A, muuttujaa `count` kasvatetaan yhdellä. Ohjelman jälkeen muuttujassa `count` on a-kirjaimella alkavien nimien määrä ja lista `names` pysyy muuten ennallaan.



Kuva 3.9: *For each* -aktiviteetin käyttö. [1]

3.4 Erilaiset muuttujat ja datan hallinta

UiPathilla muuttujia on useaa eri tyyppiä. Muuttujiin tallennetaan väliaikaisesti tietoa, jota voidaan käyttää joko ohjelman ohjaamiseen tai datan käsittelyyn ja tallentamiseen tietyn säiliön sisällä. Muuttujia on esimerkiksi seuraavia tietotyyppejä:

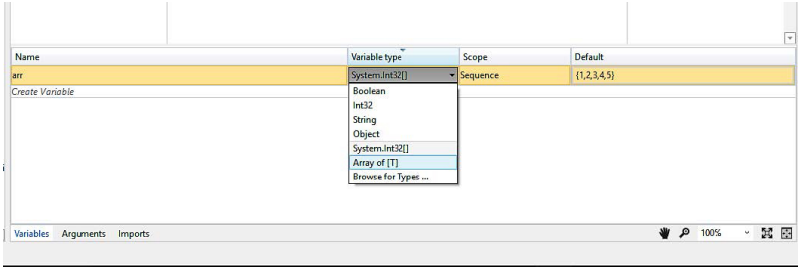
- Integer, johon tallennetaan kokonaislukuja
- String, johon tallennetaan tekstiä missä muodossa vain
- Boolean, johon tallennetaan totuusarvo tosi tai epätosi
- Generic, johon voi tallentaa mitä vain [1]

Muuttujat voidaan nimetä halutulla tavalla. Muuttujan oikeanlainen nimeäminen on kuitenkin todella tärkeää, jotta muuttujan käsittely on yksiselitteistä ja kuka vain, joka lukee ohjelmaa, ymmärtää muuttujan käyttötarkoituksen. Hyviä muuttujan nimiä ovat esimerkiksi `daysDateRange`, `flightNumber` tai `carColour`, kun taas huonoja ovat vain `days`, `temp` tai `data`, koska ne ovat ympäripyöreitä eivätkä kuvaa käyttötarkoitusta tai kohdetta. [1]

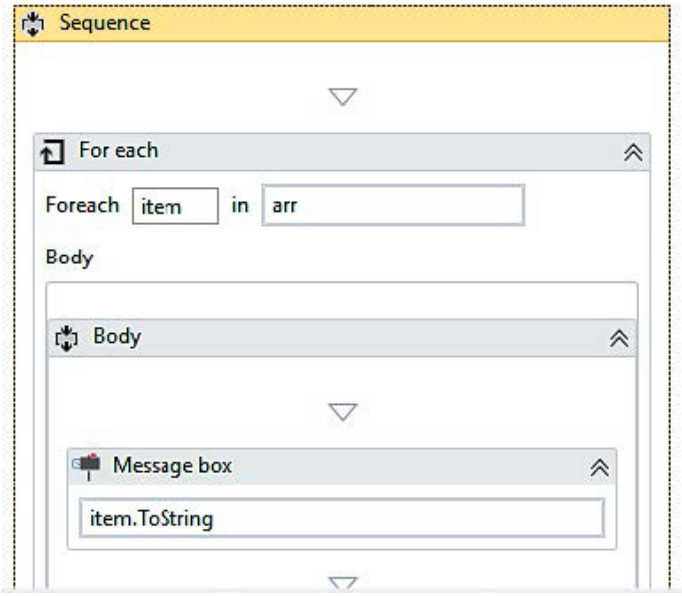
Muuttujista voidaan myös tehdä kokoelmia. Kokoelmiin voidaan tallentaa useita arvoja, mutta näiden arvojen tulee olla samaa tietotyyppiä. Muuttujat voidaan jakaa kolmeen kategoriaan:

- Scalar, joihin tallennetaan yksittäinen arvo, esimerkiksi Character, Integer, Double tms.
- Collection, joihin tallennetaan yksi tai useampi saman tietotyypin arvo, esimerkiksi array, list tms.
- Table, joihin tallennetaan taulukkoon riveille ja sarakkeille tietoa [1]

Esimerkki 8. Array-kokoelmaan, joka on tietotyyppiä Integer, pystytään tallentamaan lista kokonaisluvuista. Kuvassa 3.10 näytetään, kuinka muuttujaan `arr` tallennetaan arvot `{1, 2, 3, 4, 5}`. Kuvassa 3.11 for each -aktiviteetin sisällä jaetaan muuttujan `arr` arvot `item`-muuttujiksi ja käydään ne kaikki läpi. For each -aktiviteetin `body`-osassa jokainen `item` muutetaan String-tyypiksi ja tulostetaan tekstilaatikkoon. Ohjelmaa ajettaessa ruudulle tulostuu peräkkäin viisi kertaa tekstilaatikko, joissa on numerot 1-5. [1]



Kuva 3.10: Muuttujaan `arr` tallennetaan array-kokoelma kokonaislukuja. [1]

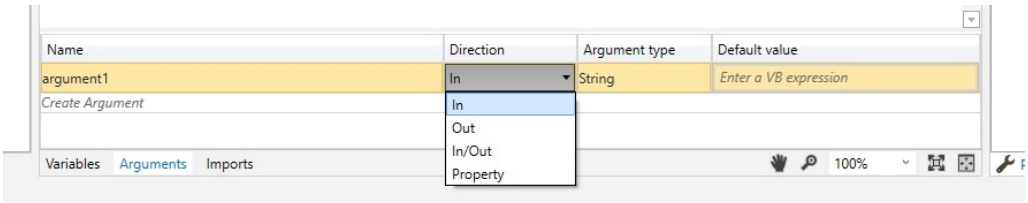


Kuva 3.11: Array-kokoelman läpikäyminen ja arvojen tulostaminen for each -aktiviteetilla. [1]

3.4.1 Argumentit ylittävät työnkulun rajat

Argumentit ovat myös muuttujia, mutta niiden käyttötarkoitus on laajempi. Muuttujia käytetään vain tietyn työnkulun (engl. workflow) sisällä, esimerkiksi prosessikaaviossa, joka on viety erilliseen tiedostoon ja tuodaan isompaan projektiin. Argumentteja käytetään näiden eri työnkulkujen välillä, jotta pieniltä, modulaarisilta komponenteilta saadaan dataa välitettyä eteenpäin muille komponenteille. Argumenteille määritetään suuntaominaisuus, joka kertoo mihin suuntaan argumentilla on tarkoitus käyttää dataa. (Kuva 3.12) Argumentin suuntaominaisuuksia ovat seuraavat:

- In, jolla määritetään tiedon tulevan toiselta työnkululta sisään
- Out, jolla määritetään tämän työnkulun lähettävän tietoa eteenpäin
- In/Out, jolla määritetään tiedon voivan kulkea kumpaankin suuntaan
- Property, jolla määritetään, ettei kyseistä argumenttia käytetä kyseisellä hetkellä [1]



Kuva 3.12: Argumentti-ikkunasta voi luoda uusia argumentteja ja valita argumentin suunnan.

3.5 Robotin ohjaaminen ruudulla

Ohjelmistorobotiikan ehkä oleellisin osa on muiden ohjelmien käyttö ja tiedonkerääminen graafisesta käyttöliittymästä (engl. user interface, UI). Jos tietoa ei saada suoraan kerättyä RPA-työkaluilla, käytetään tekstintunnistusta (engl. optical character recognition, OCR).

Aiemmin esimerkissä 4 käytettiin OCR:ää laskin-sovelluksesta summan kopioimiseen. Datan keräämiseksi käyttöliittymän elementti pitää kuitenkin kiinnittää ensin. [1]

3.5.1 Elementin kiinnittäminen kohteeksi

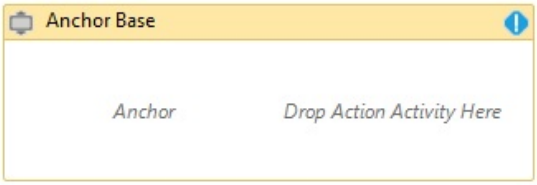
Esimerkissä 4 esitettiin, miten desktop-nauhoitin luo attach window -elementin ja asettaa muut aktiviteetit sen sisälle. Attach window -aktiviteetti kiinnittää jo auki olevan ikkunan käsittelyn kohteeksi. UiPath tunnistaa tiedoston, sovelluksen, tyypin, otsikon ja luokan ja tallentaa nämä muistiin tunnistamista varten. Näitä kutsutaan valitsimiksi (engl. selector). Valitsimia on kokonaisia (engl. full selector) ja osittaisia (engl. partial selector). Yksinkertainen nauhoitin luo kokonaisia valitsimia ja työpöytänauhoitin osittaisia. Kokonaiset valitsimet sisältävät tiedon myös elementin ylätason ikkunasta, kun taas osittaiset eivät ja tästä syystä osittaiset valitsimet laitetaan säiliön, esimerkiksi attach window, sisään. Kokonaisia valitsimia suositellaan, jos ohjelmassa vaihdetaan paljon eri ikkunoiden välillä ja osittaisia, jos tehdään useita toimenpiteitä samassa ikkunassa. Jos attach window -aktiviteetilla kiinnittää nimettömän muistion kohteeksi ja tallentaa tämän jälkeen muistion nimellä, seuraavalla suorituskerralla attach window ei enää tunnista tätä samaa muistiota, koska sen nimi on muuttunut. UiPathistä tällaisia tilanteita varten löytyy jokerimerkkejä (engl. wildcard). Kysymysmerkki (`?`) korvaa yhden merkin ja asteriski ($\star$) korvaa useita merkkejä. Näitä kannattaa käyttää sellaisten tunnisteiden kanssa, jotka muuttuvat usein. [1, 6]

Käyttöliittymän hallintaan käytettävien ohjaimien löytämiseksi UiPathissä on useita aktiviteetteja. Nämä ovat nimenomaan käyttöliittymästä ohjaimien etsimiseen. Tällasia aktiviteetteja ovat:

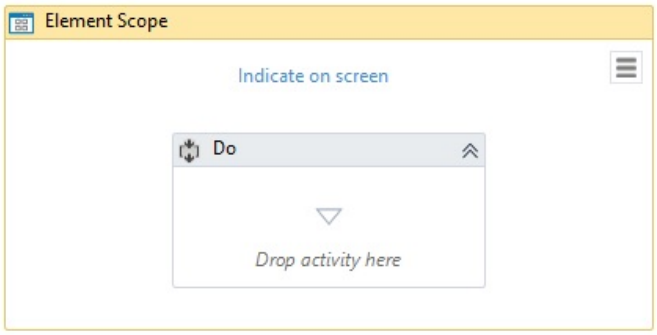
- Anchor base
- Element exists
- Element scope

- Find children
- Find element
- Find relative element
- Get ancestor
- Indicate on screen [1]

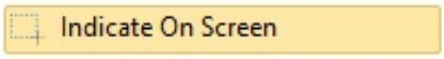
Anchor base -aktiviteetilla voidaan etsiä elementin käyttöliittymästä käyttämällä avuksi sen viereistä elemettiä. Tätä käytetään silloin, kun ei ole luotettavaa valintaa haluttuun elementtiin. Anchor base -aktiviteetissa on kaksi osaa: ankkuripiste ja sille tehtävä aktiviteetti. (kuva 3.13) *Element exists* palauttaa totuusarvon tosi tai epätosi, kun se on tarkistanut löytyykö kyseistä elementtiä halutusta sovelluksesta. Tätä on hyvä käyttää, jos ei ole varmaa onko kyseinen elementti käytettävissä tai jos käyttöliittymä muuttuu paljon. *Element scope* kiinnittyy käyttöliittymän elementtiin ja sille voidaan tehdä useita toimenpiteitä. Tästä aktiviteetista löytyy kaksi kohtaa: käyttöliittymän elementin valinta ja laatikko kaikille toimenpiteille, joita elementille halutaan tehdä. (kuva 3.14) Nämä toimenpiteet tehdään sarjana peräkkäin. *Find children* etsii valitun elementin lapsielementit (engl. child) ja palauttaa ne kokoelmana. Tämä kokoelma voidaan tallentaa `IEnumerable <UIElements>` -tyyppiseen muuttujaan. *Find element* odottaa halutun elementin ilmestymistä ja palauttaa sen. *Find relative element* toimii samalla tavalla kuin find element, mutta käyttää viereistä elementtiä halutun elementin löytämiseksi. *Get ancestor* hakee elementin esi-isäelementin ja (engl. ancestor) palauttaa sen. *Indicate on screen* valitsee käyttöliittymäelementin ohjelman ajon aikana. Tämä on eri toiminto kuin muiden aktiviteettien sisällä olevat indicate on screen -napit, esimerkiksi kuvassa 3.14 element scope -aktiviteetissa, joilla pystyy valitsemaan halutun elementin kohteeksi. (kuva 3.15) [1]



Kuva 3.13: Anchor base -aktiviteetissa kiinnitetään viereinen elementti ankkuripisteeksi ja määritetään tehtävät toimenpiteet viereiseen osaan.



Kuva 3.14: Element scope -aktiviteetissa ruudulta kohdistetaan elementti ja sille voidaan tehdä useita toimenpiteitä sarjassa.



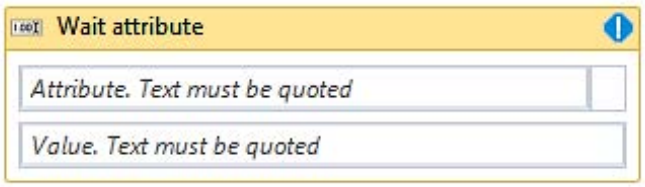
Kuva 3.15: Indicate on screen eroaa muitten elementtien saman nimisestä napista.

3.5.2 Ohjauselementin odottaminen

Ohjelman kulun aikana voi joskus joutua odottamaan jonkun toimenpiteen tapahtumista. Tätä varten UiPathistä löytyy kolme aktiviteettia:

- Wait element vanish
- Wait image vanish
- Wait attribute [1]

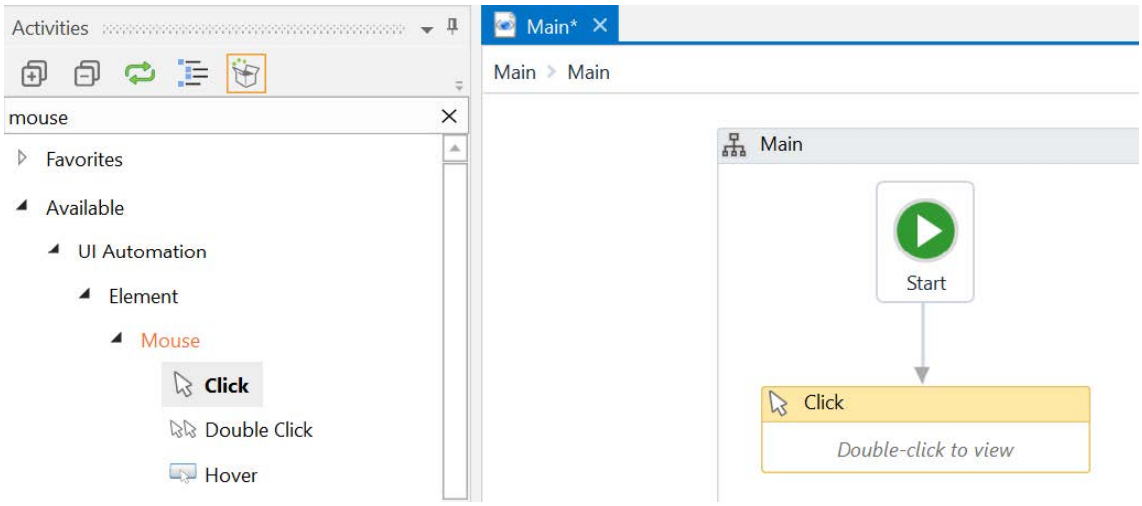
Wait element vanish odottaa elementin häviävän ruudulta. *Wait image vanish* on vastaava aktiviteetti kuin wait element vanish, mutta siinä kuvan odotetaan häviävän käyttöliittymästä. Nämä ovat hyödyllisiä toimenpiteitä, jos käytettävä sovellus tai nettisivu toimii hitaasti ja kohteen pitää sulkeutua ennen ohjelman jatkamista. *Wait attribute* -aktiviteetti odottaa elementin ominaisuuden olevan annettu merkkijono. (kuva 3.16) [1]



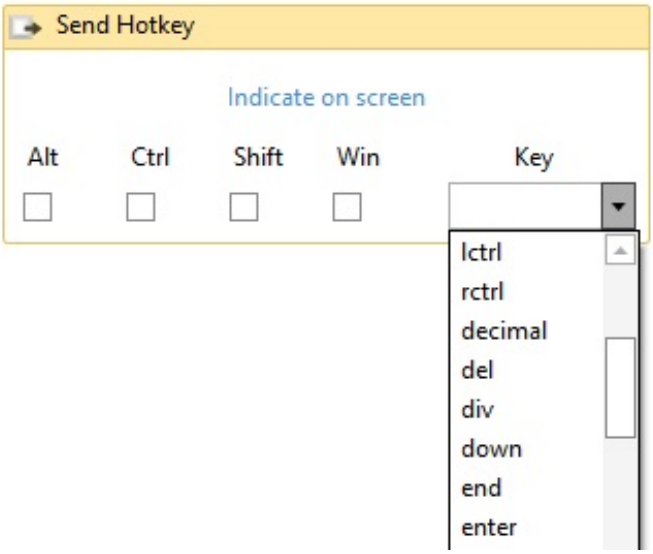
Kuva 3.16: Wait attribute -aktiviteetissa määritetään minkä ominaisuuden halutaan olevan tietty määritetty arvo. [1]

3.5.3 Ohjauselementin käyttäminen hiirellä ja näppäimistöllä

Ohjauselementin kohteen valinnan jälkeen sitä käsitellään hiiren tai näppäimistön painalluksilla. Hiirellä voidaan joko klikata (Click), tuplaklikata (Double click) tai leijua (Hover) elementin päällä (kuva 3.17) ja näppäimistöllä käyttää pikanäppäinyhdistelmiä (Send hotkey), kirjoittaa tekstiä normaalisti (Type into) tai suojatusti (Type secure text). Hiiren klikkaukselle voi määrittää tapahtuuko se hiiren vasemmalla, oikealla vai keskimmäisellä näppäimellä. Näppäimistön pikanäppäinyhdistelmästä voi valita painetaanko samalla Alt, Ctrl, Shift tai Windows-näppäintä kun painetaan jotain muuta näppäimistön painiketta. Edellä mainitut neljä valitaan valintaruudusta ja muut painikkeet voidaan valita pudotusvalikosta tai kirjoittamalla kenttään. Yksittäiset kirjaimet pitää kirjoittaa kenttään lainausmerkeissä. (kuva 3.18) Normaalin tai suojatun tekstin voi syöttää tekstilaatikkoon lainausmerkkien sisälle. Suojattua tekstinsyöttöä on hyvä käyttää esimerkiksi salasanojen syöttämiseen. [1]



Kuva 3.17: Hiiren aktiviteetit vedetään samalla tavalla aktiviteetti-ikkunasta. [1]



Kuva 3.18: Näppämistön pikanäppäinyhdistelmässä voi valita useamman näppäimen samanaikaisen painalluksen.

3.5.4 Tekstinkaavinta ja OCR

Roboteilla käsitellään usein tekstiä ja materiaalia pitää kerätä jostain käyttöliittymän elementistä. Elementti voi olla dokumentti, nettisivu, PDF-tiedosto tai kuva. UiPathissä on kolme tapaa kerätä tekstiä:

- Full text
- Native
- OCR [1]

Full text -aktiviteetilla pystytään lukemaan tekstiä dokumenteista ja nettisivuilta. Sen tarkkuus tekstin keräämisessä on 100 % ja se on esitellyistä kolmesta tavasta nopein. Sitä voidaan käyttää ohjelman ajon taustalla ja sen avulla voidaan kaapia jopa piilotettua tekstiä, mutta Citrix-ympäristöissä se ei kuitenkaan toimi. [1]

Native vastaa *full text* -aktiviteettia, mutta se on hitaampi, eikä sillä voida tunnistaa mistä kohtaa dokumenttia teksti on kerätty. Se on myös 100 % tarkka, mutta sitä ei voi käyttää ajon taustalla. Native ei myöskään toimi Citrix-ympäristöissä. [1]

OCR on aktiviteetti, jota käytetään viimeisenä vaihtoehtona. Se on huomattavasti hitaampi kuin kaksi edellistä tapaa, mutta sitä voidaan käyttää kuvien tekstin tunnistuksessa ja näin ollen sitä voidaan käyttää myös Citrix-ympäristöissä. OCR:n tarkkuus on vain 98 % eikä sitä ei pysty käyttämään ajon taustalla. UiPath Studiossa on käytössä kaksi OCR-moottoria: *Microsoft OCR* ja *Google OCR*. Microsoft OCR:llä on suora tuki usealle eri kielelle ja sillä voidaan kerätä tekstiä suurelta alueelta. Google OCR:lle taas on mahdollista hankkia useita eri kieliä, sillä voi hyvin kerätä tekstiä pieneltä alueelta, se tunnistaa vastavärit ja sillä pystyy suodattamaan tekstistä halutut kirjaimet. Muitakin OCR-moottoreita on saataville UiPathille lisäosina ja niillä on monia eri käyttötarkoituksia, kuten käsinkirjoitetun tekstin tunnistamiseen. [1]

3.6 Robotin aktivoinnin laukaiseminen

Aikaisemmin kappaleessa 3.1.2 mainittiin kahdenlaisia robotteja: osallistuvia ja hoitamattomia robotteja. Näistä osallistuvat robotit toimivat avustajina jokapäiväisessä työskentelyssä ja ne aktivoidaan usein hiiren tai näppäimistön painalluksella. UiPathissä on myös muita tapoja laukaista robotin toiminta. Laukaisimet (engl. trigger) voidaan jakaa kolmeen kategoriaan:

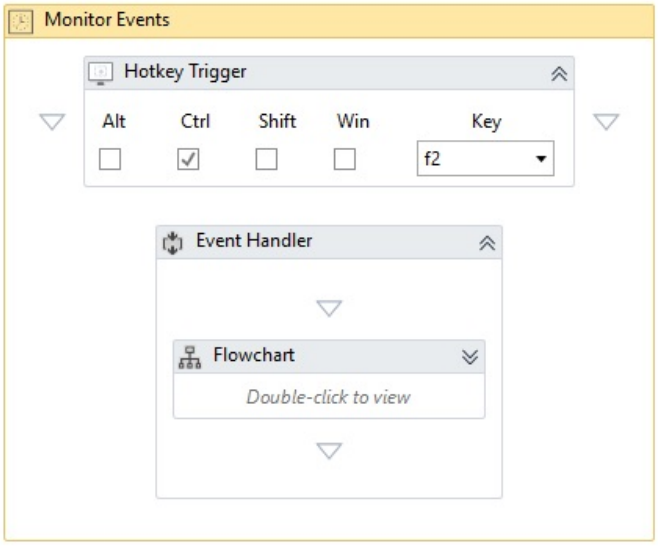
- System trigger
- Image trigger
- Element trigger [1]

System trigger tarkoittaa yleisesti järjestelmän toimien seuraamista ja niitten aiheuttamaa laukaisua. System triggerin alta löytyy *Hotkey trigger*-, *Mouse trigger*- ja *System trigger* -aktiviteetit. Hotkey triggerissä voidaan pikanäppäinyhdistelmällä aktivoida työnkulku. Tätä aktiviteettia käytetään samalla tavalla kuin Send hotkey -aktiviteettia (kuva 3.18). Ohjelman suoritus voidaan esimerkiksi asettaa aktivoitumaan painamalla näppäimistöltä Ctrl- ja F2-näppäimiä samaan aikaan. *Mouse trigger* aktivoi työnkulun hiiren painalluksella ja siihen voidaan laittaa hiiren painikkeen lisäksi erikoisnäppäin näppäimistöltä. Esimerkiksi aloitetaan ohjelman suoritus painettaessa Alt-näppäintä ja hiiren keskinappulaa. *System trigger* -aktiviteetilla saadaan ohjattua ohjelman aktivointia näppäimistön, hiiren tai molempien nappuloiden tai näppäinten painalluksesta. *System triggeriin* voidaan myös määrittää toimenpide, joka määrittelee mitä laukaisimesta tapahtuu. Näitä ovat joko ohjelman jatkaminen tai tapahtuman estäminen. [1]

Image triggerin alla on ainoastaan *Click image trigger*. Tämä laukaisee työnkulun aktivoinnin, kun määritettyä kuvaa klikataan hiirellä. *Element trigger* on vastaava kuin *image trigger*, mutta siinä kohteena voi olla mikä vain elementti. *Element triggerin* alta löytyy *Click trigger* ja *Key press trigger*. *Click trigger* toimii vastaavalla tavalla kuin aiemmat hiirenklikkausaktiviteetit, mutta siinä kohteena on määritetty elementti. *Key press*

trigger taas kuuntelee näppäimistön näppäilyä määritetyssä elementissä ja aktivoi ohjelman työnkulun, jos määritettyjä näppäimiä painetaan. [1]

Kaikki laukaisimet asetetaan *Monitor events* -säiliön sisälle, jossa on kaksi osaa: kuunneltava laukaisin ja laukaisun jälkeen tapahtuvat toimenpiteet. Laukaisimen kohdalle määritetään joku yllä mainituista aktiviteeteista ja kohtaan *Event handler* määritetään halutut toimenpiteet (kuva 3.19). [1]



Kuva 3.19: Monitor eventsin sisään voi laittaa esimerkiksi Hotkey triggerin, joka aktivoi prosessikaavion, kun näppäimistöltä painetaan Ctrl + F2.

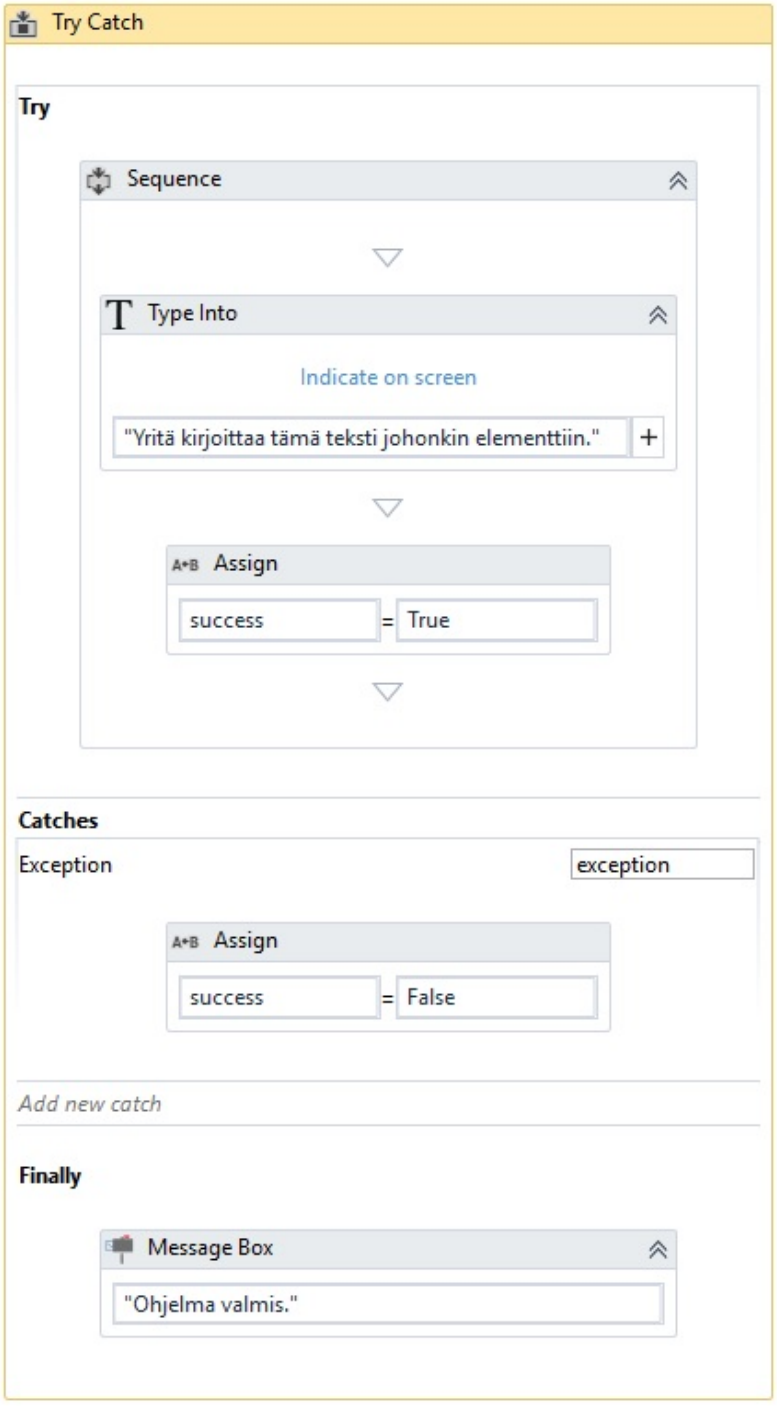
3.7 Poikkeustilanteet ja niiden hoitaminen

Roboteilla ohjataan usein muita ohjelmia ja poikkeustilanteita näiden käyttämisessä syntyy aina. Poikkeuksella tarkoitetaan jotain tapahtumaa, joka on tavallisesta poikkeavaa tai erikoista. Koska robotti tekee täsmälleen sen, mitä sille on ohjelmoitu, robotti ei osaa itsenäisesti toimia näissä tilanteissa. Tästä syystä poikkeustilanteisiin pitää tarttua ja kertoa robotille erikseen, mitä näissä tilanteissa tulee tehdä. [1]

UiPathistä löytyy poikkeustilanteiden käsittelyyn *Try catch* -aktiviteetti. Tässä aktiviteetissa on kolme osaa: try, catch ja finally. Try-osuudessa määritellään ne aktiviteetit, joi-

ta yritetään tehdä. Jos nämä toimenpiteet suoritetaan onnistuneesti eivätkä aiheuta virheilmoituksia, ohjelma jatkaa eteenpäin. Tällaisessa tilanteessa suoritus ei eroa juurikaan lainkaan siitä, että nämä toimenpiteet olisivat olleet ohjelmassa ilman try catch -aktiviteettia. Jos kuitenkin poikkeustilanne tapahtuu ja ohjelmaan tulee virhe, try catch -aktiviteetti siirtyy catch-osioon. Catch-osiossa poikkeus (engl. exception) voidaan antaa ilmoituksena ruudulle tai poikkeustilanteeseen voidaan tehdä vaihtoehtoinen ratkaisu. Poikkeusilmoituksen saa tekstimuotoisena lisäämällä add expection -vaihtoehdon. Näitä vaihtoehtoja voi valita erityyppisiä, kuten System.ArgumentException, System.NullReferenceException tai iha vain System.Exception. Useimmiten System.Excep-tion on hyvä valinta poikkeukseksi, koska se ei määrittele poikkeuksen tyyppiä liian tarkasti. Finally-osuus tehdään joka tapauksessa jokaisen try catch-aktiviteetin lopussa riippumatta siitä, tuliko virhe- tai poikkeustilanne vai ei. [1]

Esimerkki 9. Kuvassa 3.20 on havainnollistettu kuinka *try*-osuuden sisällä on sarja, jonka sisällä ensin kirjoitetaan tiettyyn elementtiin tekstiä. Tämän jälkeen määritetään muuttujan `success` arvoksi `True`. Jos ohjelma ei onnistu tekemään näitä toimenpiteitä, suoritus siirtyy *catch*-osioon ja asetetaan muuttujan `success` arvoksi `False`. Onnistui ohjelma toimenpiteissään tai ei, lopuksi ruudulle tulee tekstilaatikko, joka kertoo ohjelman suorituksen olevan valmis.



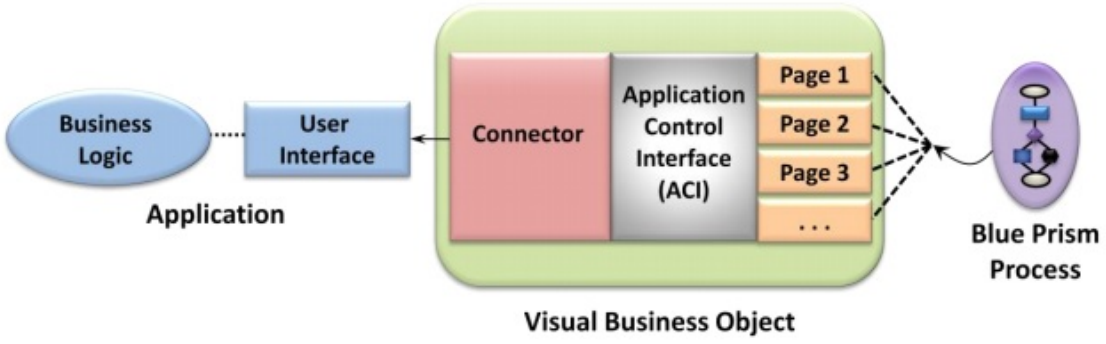
Kuva 3.20: *Try catch* -aktiviteetin rakenne.

3.8 Muita ohjelmistorobotiikkasovelluksia

Muita paljon käytettyjä ohjelmistorobotiikkasovelluksia ovat muun muassa Blue Prism ja Automation Anywhere. Nämä ovat kokonaisia robotiikkaohjelmistoja, jotka ovat tarkoitettu keskisuurten ja suurten yritysten käyttöön. Pienten yritysten ei välttämättä kannata käyttää kokonaista robotiikkaohjelmistoa, vaan ohjelmistorobotin voi luoda myös käyttäen ilmaisia, avoimen lähdekoodin työkaluja, kuten SikuliXiä. [7, 8]

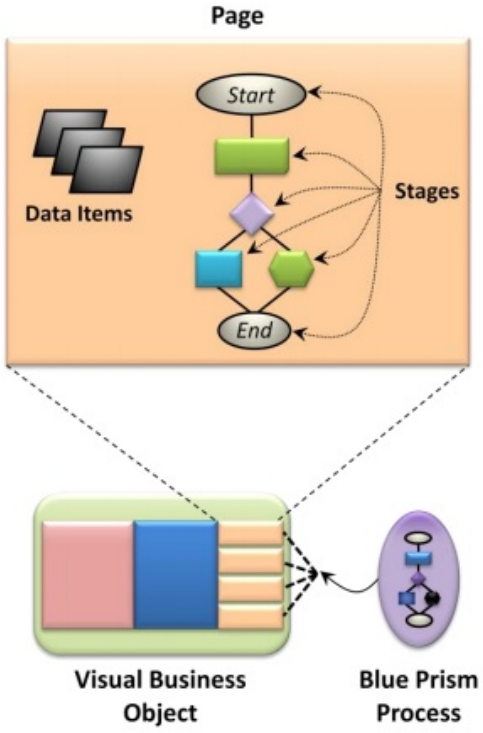
3.8.1 Blue Prism

Blue Prism on UiPathia vastaava ohjelmisto, jonka käyttämisessä ohjelmistokokemuksesta on kuitenkin huomattavasti enemmän hyötyä. Sitä voidaan käyttää millä alustalla ja ohjelmalla tahansa ja sen ohjelmien suorittaminen on nopeaa. Robottien luominen Blue Prismillä eroaa kuitenkin UiPathista käytännön tasolla. Blue Prismissä on kaksi osaa robottien luomiseen: Object Studio ja Process Studio. Object Studiota käytetään Visual Business Objectien (VBO), eli automatisoitavan ohjelman ja Blue Prismin prosessien välisten sovittimien luomiseen (kuva 3.21). VBO rakentuu kolmesta osasta, joista ensimmäinen on liitin (engl. connector), joka käsittelee automatisoitavan ohjelman käyttöliittymää. Liittimiä ovat esimerkiksi HTML-liitin nettisivuja varten ja Windows-liitin Windows-käyttöjärjestelmää varten. [7, 9]



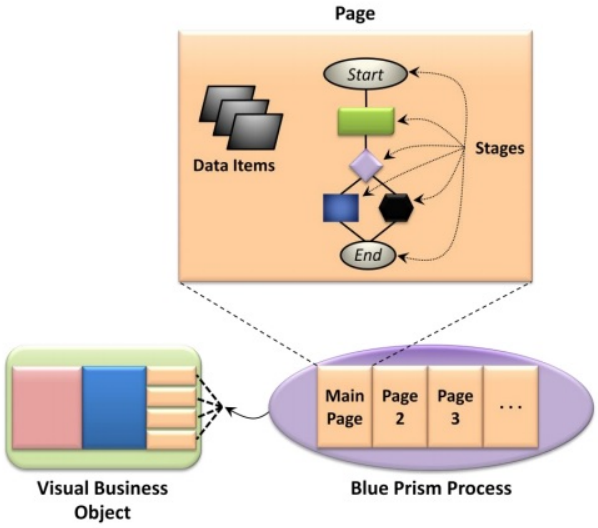
Kuva 3.21: Visual Business Objectin kolme osaa. [9]

Application Control Interface käsittelee liittimen avulla käyttöliittymän elementtejä. Kolmas osa VBO:ssa on sivut (engl. page) (kuva 3.22). Sivut ovat automatisoitavan ohjelman käyttöliit-tymän yksittäisiä toimenpiteitä varten. Toimenpiteistä käytetään termiä `stage` ja sivujen sisällä on myös erillisiä dataelementtejä, jotka toimivat muuttujina sivun sisällä. [9]



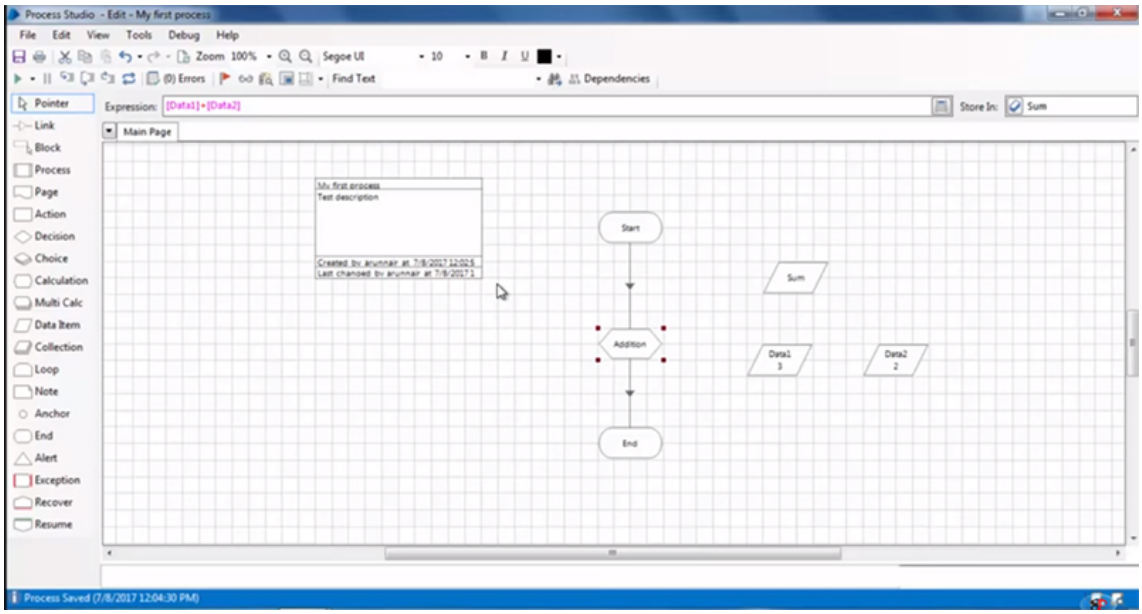
Kuva 3.22: VBO:n sivun rakenne. [9]

VBO:t toimivat yksittäisten operaatioiden toteuttamisessa, kuten ohjelman sisäänkirjautumissa. Kokonaisten robottien luomiseen käytetään Process Studiota, jolla kootaan VBO:eja yhteen prosesseiksi. Prosessit ovat samantyylisiä kuin VBO:t, mutta niissä on pieniä eroja. Prosesseissa on samalla tavalla sivuja, mutta nämä sivut toteutetaan aina samassa järjestyksessä pääsivusta (engl. main page) lähtien (kuva 3.23). Näissä prosessien sivuissa voidaan pääasiassa käyttää samoja stageja kuin VBO:ssa, mutta esimerkiksi Read-, Write- ja Navigate-toimenpiteitä ei voi prosesseissa käyttää, koska ne ovat tarkoitettu suoraan automatisoitavan ohjelman käyttöliittymällä käytettäväksi.



Kuva 3.23: Blue Prism -prosessin rakenne. [9]

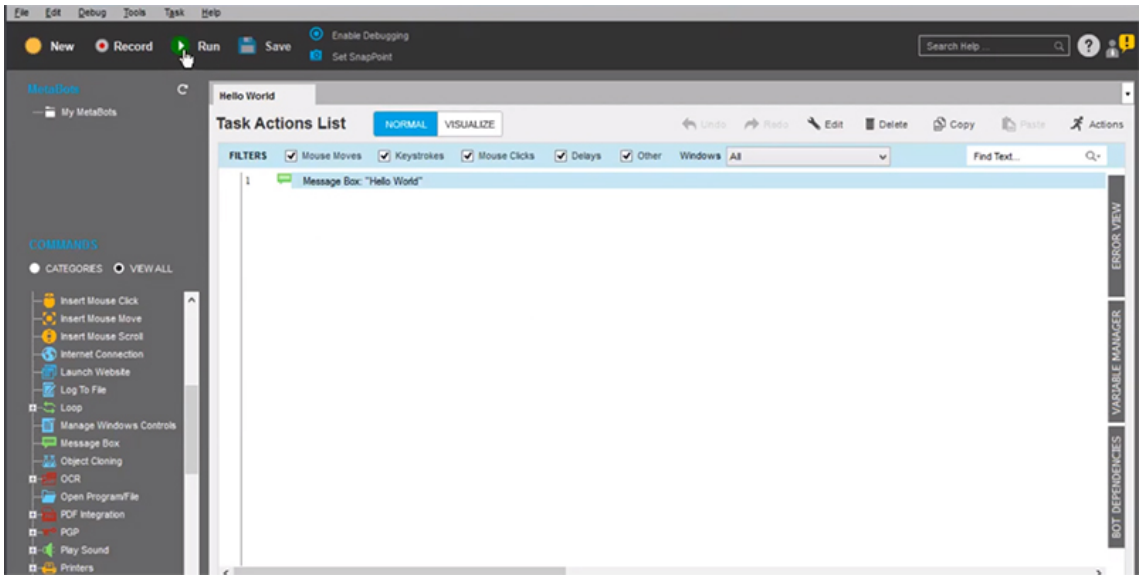
Robottien luominen Blue Prismillä tapahtuu kuvan 3.24 mukaisella käyttöliittymällä. Käyttöliittymän keskellä on tyhjä pohja, jossa on aluksi vain Start- ja End-staget. Vasemmasta reunasta muita stageja voi vetää keskelle ja yhdistää niitä Link-stagella. Muuttujat tallennetaan omiin Data Item -stageihin. Kuvassa 3.24 Calculation-stagella summataan `Data1` ja `Data2` ja tallennetaan summa `Sum`-muuttujaan. [10]



Kuva 3.24: Blue Prismin käyttöliittymä. [10]

3.8.2 Automation Anywhere

Automation Anywhere toimii eri tavalla kuin UiPath tai Blue Prism. UiPathissä ja Blue Prismissä ohjelmat rakennettiin vetämällä elementtejä kaavioon ja yhdistämällä eri elementtejä toisiinsa. Automation Anywheressä tehtäviä vedetään listalle, josta ne suoritetaan järjestyksessä (kuva 3.25). Tällainen tapa ei vaadi käyttäjältä ohjelmointitaustaa, kun kaikki toimenpiteet tehdään visuaalisella käyttöliittymällä. [7, 11]



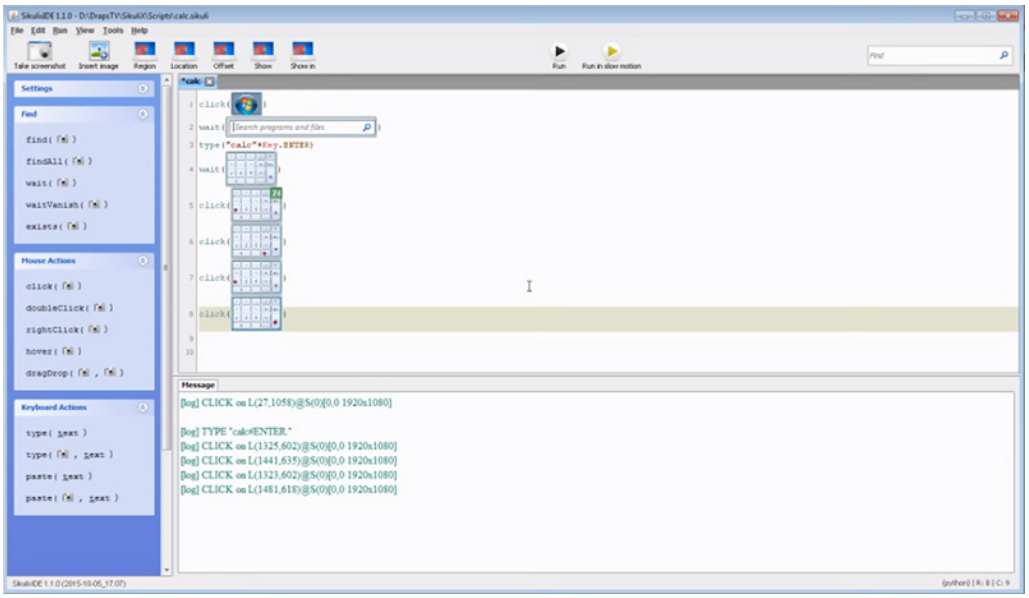
Kuva 3.25: Automation Anywheren käyttöliittymä. [11]

3.8.3 SikuliX

SikuliX on avoimen lähdekoodin ohjelma, jolla voidaan automatisoida tietokoneen ruudulla tapahtuvia asioita. Se käyttää OpenCV:n kuvantunnistusta ja automatisointi tapahtuu vain kuvien avulla, sillä SikuliX ei pääse käsiksi esimerkiksi nettisivujen lähdekoodiin. SikuliX on Java-pohjainen ohjelma, mutta sillä voidaan käyttää Jythonia, eli Pythonia Java-alustalla. [8]

SikuliXin käyttöliittymä muistuttaa enemmän tavallista tekstieditoria kuin UiPathin tai Blue Prismin perusnäkymän työpöytä (kuva 3.26). Elementtejä ei vedetä hiirellä työpöydälle ja niille ei voi erikseen tehdä määrityksiä. Kaikki toimenpiteet tulevat tekstinä,

kuten tekstieditorilla ohjelmoidessa. Käyttöliittymän vasemmassa reunassa on lista mahdollisista SikuliXin tarjoamista automatisoitavan ohjelman käyttöliittymän hallintaan liittyvistä työkaluista. Klikkaamalla niistä SikuliX pienentää itsensä ja antaa mahdollisuuden valita haluttu kohta automatisoitavasta ohjelmasta. [12]



Kuva 3.26: SikuliXin käyttöliittymä. [12]

SikuliXissa on muutamia erilaisia ominaisuuksia kuvantunnistukseen. Ohjelmalla voidaan säätää sitä, kuinka samankaltaista kuvaa ruudulta etsitään ja mistä kohtaa kuvaa esimerkiksi halutaan klikattavan. Samankaltaisuuden voi määrittää prosentteina, jolloin ohjelma yrittää hakea ruudulta kuvaa, joka muistuttaa tämän prosentin verran ennalta määritettyä kuvaa. Ennalta määritetystä kuvasta voidaan valita pikseleinä siirros, mistä kohtaa kuvaa halutaan hallinnoitavan. Kkuvassa 3.26 on Windows-käyttöjärjestelmällä ensin valittu Windows-painike tehtäväpalkista, jonka jälkeen odotetaan hakupalkin löytymistä ruudulta. Tämän jälkeen kirjoitetaan näppäimistöllä `calc` ja painetaan Enter-näppäintä näppäimistöltä. Kun laskimen numerovalikko löytyy ruudulta, samasta kuvasta valitaan merkkien 1, +, 1 ja = kohdilta. Tätä dataa voidaan käsitellä normaaleilla Pythonin komennoilla ja suorittaa poikkeuksen käsittelyä tai silmukoita, kuten muilla Python-ohjelmilla. [12]

Luku 4

Perinteinen ohjelmistotestaus

Ohjelmistotestauksella tarkoitetaan ohjelmointiprojekteissa ohjelman testausta, jonka kautta mahdolliset koodivirheet löytyvät ja ne saadaan korjattua. Nykyään lähes kaikissa laitteissa tai työkaluissa on jotain ohjelmistoa, joten testauksenkin rooli on korostunut. Tämä luo painetta, koska ohjelmistot ovat niin laajasti saatavilla ympäri maailman. Toisaalta nykyään ohjelmistotestaus on helpompaa kuin aiemmin, koska teknologia on kehittynyt huomattavasti tuoden mukanaan erilaisia, valmiiksi testattuja kirjastoja, joita voi käyttää omassa projektissaan suoraan. [13]

Tässä luvussa käsitellään ohjelmistokehityksen testausta. Aluksi käydään läpi testauksen psykologinen puoli ja miten testaukseen tulisi suhtautua. Testauksessa on perusperiaatteita, jonka mukaan testausta kannattaa toteuttaa. Eri testauksen tapoihin pureudutaan kappaleessa 4.2 ja siinä käydään konkreettisesti läpi testauksen vaiheita. Tämän jälkeen käsitellään ohjelmiston tarkastelu kooditasolla ryhmässä, jotta ajatusvirheet ja väärinymmärrykset ohjelman toiminnasta saadaan korjattua. Esimerkkitapaukset ovat seuraava aihe, sillä ne ovat erittäin hyvä tapa käydä ohjelman syötteitä ja tuloksia läpi.

Ohjelmistokehityksen testauksen automatisointia ei käsitellä tässä työssä, sillä samoja menetelmiä on hankala käyttää ohjelmistorobotiikan testaukseen eikä tällöin hyödytä tämän tutkimuksen tarkoitusta.

4.1 Testaus psykologisella tasolla

Yleinen perusajatus ohjelmistotestauksen taustalla on ohjelmien virheettömyyden ja toiminnan oikeellisuuden tarkastaminen. Todellisuudessa ohjelmistoa pitäisi testata täysin päinvastaisista syistä ja testaus tulisi tehdä nimenomaan virheiden löytämiseksi. Jos tavoitteena on todistaa, ettei ohjelmassa ole virheitä, testaukset tehdään syötteillä, jotka todennäköisesti eivät aiheuta virheitä. Virheiden löytämiseksi ohjelma yritetään rikkoa tarkoituksellisesti ja tämä voi tuntua vaikealta. Ohjelmistokehitys on rakentavaa ja sen tarkoituksena on kehittää toimivaa ja tehokasta ohjelmaa. Ohjelmistotestauksen idea on yrittää rikkoa ohjelma ja päätyä tilanteeseen, jossa ohjelma ei anna haluttua tulosta, ei tee mitä sen on tarkoitus tai tekee jotain, mitä sen ei ole tarkoitus tehdä. [13]

Testaus on onnistunut, jos sen avulla löydetään virheitä. Ohjelma ei ole koskaan täydellinen ja virheiden löytyminen voidaan nähdä positiivisena asiana. Virheet kuitenkin usein nähdään negatiivisena ongelmana ja sovelluskehittäjän ajatellaan tuottavan virheellistä koodia. Tästä syystä ohjelmiston testaukseen onkin suhtauduttava siten, että vaikka tarkoitus on löytää virheitä, sen tehtävä on parantaa työn laatua ja arvoa. [13]

Esimerkki 10. Verrataan ohjelmistotestausta lääketieteellisen diagnoosin tekemiseen. Jos huonovointinen henkilö menee lääkäriin, lääkäri alkaa tutkia potilasta, jotta voisi diagnosoida missä vika on. Lääkäri voi määrätä erilaisia laboratoriotestejä, joiden tarkoituksena on löytää virheitä ihmisestä. Jos laboratoriotulosten perusteella potilas on terve, ei näitä testejä pidetä onnistuneina. Testit olivat epäonnistuneita, koska potilaan kannalta laboratoriokokeiden tulokset eivät olleet odotetut ja aiheuttivat ylimääräisiä kustannuksia ilman tuloksellista diagnoosia. Jos testeillä tulee tuloksia, joista lääkäri voi tehdä diagnoosin, hän voi aloittaa lääkityksen ja potilaan yleiskunto saadaan paranemaan. [13]

Ohjelmistotestauksessa suurinta osaa toimenpiteistä pidetään itsestäänselvyytenä ja usein unohdetaan kokonaan. Tästä syystä ohessa oleva lista on hyvä pitää mielessä testausta tehdessä, jotta jokainen aspekti tulee huomioitua. [13]

1. Tärkeä osa testausta on määrittää odotettu tulos.
2. Ohjelmistokehittäjän pitää välttää itse testaamasta omaa ohjelmaansa.
3. Ohjelmisto-organisaation pitää välttää itse testaamasta omaa ohjelmaansa.
4. Jokaisen testauskerran pitää sisältää tulosten käsittely.
5. Esimerkkitapausten pitää sisältää syötteitä, jotka ovat odotettuja ja valideja, mutta myös syötteitä, jotka eivät ole odotettuja ja valideja.
6. Ohjelmaa pitää tarkastella, eikö se tee mitä sen pitäisi tehdä, mutta myös niin, tekeekö se mitä sen ei pitäisi tehdä.
7. Kertakäyttöisiä testejä pitää välttää, jos ohjelma ei ole kertakäyttöinen.
8. Ohjelmistotestausta ei pidä suunnitella sillä ajatuksella, että virheitä ei löydy.
9. Ohjelman osasta jo löytyneiden virheiden määrä on suoraan verrannollinen todennäköisyyteen löytää samasta osasta lisää virheitä.
10. Testaus on luovaa ja älykkyyttä haastavaa työtä. [13]

Tärkeä osa testausta on määrittää odotettu tulos. Tällä tarkoitetaan sitä, että ennen testiä tulee määrittää tarkkaan syötteet ja niistä odotetut tulokset. Muuten uskottavan näköinen, mutta virheellinen tulos saatetaan laskea oikeaksi tulokseksi, koska tuloksen halutaan olevan oikea. [13]

Ohjelmistokehittäjän pitää välttää itse testaamasta ohjelmaansa ja *Ohjelmisto-organisaation pitää välttää itse testaamasta ohjelmaansa*. Nämä kaksi koskevat samaa asiaa, mutta eri tasoilla. Yleisesti oman tekstin tai työn oikolukemista pidetään huonona ideana,

koska oman työnsä virheille tulee sokeaksi. Jos sama kehittäjä on suunnitellut ja rakentanut ohjelman, on todella vaikeaa yrittää rikkoa sitä tarkoituksella. Ongelmaksi nousee myös, jos kehittäjä on alunperin ymmärtänyt väärin ohjelman tarkoituksen tai oikean lopputuloksen. Tällöin testaaminen johtaa kehittäjän mielestä oikeaan tulokseen, joka on jo lähtökohtaisesti väärin. Ohjelmisto-organisaation tasolla ongelmaksi saattaa muodostua tilanne aikataulun tai budjetin kanssa, koska nämä ovat helposti mitattavia yksiköitä. Testausta pidetään ongelmana näitä kahta kohtaan, koska se saattaa hidastaa aikataulua ja nostaa kuluja. [13]

Jokaisen testauskerran pitää sisältää tulosten käsittely. Tämä on itsestäänselvä kohta, mutta joka usein katsotaan läpi sormien. Jokaisen testin tulokset tulisi käydä läpi, jotta virheet huomataan mahdollisimman aikaisessa vaiheessa eikä vasta myöhemmillä testikerroilla. [13]

Esimerkkitapausten pitää sisältää syötteitä, jotka ovat odotettuja ja valideja, mutta myös syötteitä, jotka eivät ole odotettuja ja valideja. Esimerkkitapauksessa tarkoitetaan testitapausta, jossa sisältö on keksitty ja vastaa todellisen käyttötarpeen mukaisia arvoja. Testauksessa käytettyjen syötteiden tulee sisältää myös sellasia arvoja, jotka ovat tarkoituksella väärin ja antavat virheen. Jos ohjelma ei anna virheilmoitusta tällaisissa tilanteissa, ohjelma on virheellinen. Esimerkiksi jos ohjelmaan pitäisi syöttää henkilön pituus ja syötteeksi antaa negatiivisen luvun, ohjelman pitäisi antaa virheilmoitus. [13]

Ohjelmaa pitää tarkastella niin, että eikö se tee mitä sen pitäisi tehdä, mutta myös niin, että tekeekö se mitä sen ei pitäisi tehdä. Yleisesti ohjelmaa rakennettaessa sille määritetään tarkka määränpää, mitä ohjelman tulee tehdä. Testausvaiheessa usein keskitytään myös samaan asiaan ja tarkastellaan tekeekö ohjelma mitä halutaan. Yhtä tärkeää on tarkastella myös sitä, tekeekö ohjelma jotain muuta ylimääräistä tai haitallista. [13]

Kertakäyttöisiä testejä pitää välttää, jos ohjelma ei ole kertakäyttöinen. Tällä tarkoitetaan lennosta keksittyjä testitapauksia, joita ei dokumentoida mitenkään. Nämä testitulokset katoavat heti ja jos ohjelmaa joudutaan testaamaan uudelleen, pitää testitapauksetkin kehittää joka kerta uudelleen. Koska testien suunnittelu vie aikaa, uudelleen tehdyt testit ovat usein kevyempiä eivätkä lainkaan niin tarkkoja kuin alkuperäiset. Testitapausten dokumentointia ja samojen testien käyttämistä komponenttien muuttamisen jälkeen kutsutaan regressiotestaukseksi. [13]

Ohjelmistotestausta ei pidä suunnitella ajatuksella, että virheitä ei löydy. Täysin virheetöntä ohjelmaa todennäköisesti ei ole, joten testauksen tulisi kohdistua virheiden löytämiseen eikä niiden välttämiseen. Jos haluttaisiin varmistaa, ettei ohjelmassa ole yhtään virhettä, pitäisi kaikki mahdolliset yhdistelmät kaikista mahdollisista toimenpiteistä ja syötteistä käydä läpi. Tällaisen tyhjentävän testauksen, joka testaa kaiken, ei ole mahdollista suorittaa. Esimerkiksi pienen laskimen kaiken testaamiseen pitäisi kokeilla kaikki mahdolliset syötteet ja kaikilla mahdollisilla laitteilla ja alustoilla, mikä tarkoittaisi lähes ääretöntä määrä kokeiluja. [13, 14]

Ohjelman osasta jo löytyneiden virheiden määrä on suoraan verrannollinen todennäköisyyteen löytää siitä osasta lisää virheitä. Jos ohjelman osasta löytyy virheitä, on hyvin todennäköistä, että samasta osasta löytyy lisää virheitä. Jos verrataan kahtaa ohjelman osaa, josta ensimmäisestä löytyy 10 virhettä ja toisesta 1, todennäköisyys on suurempi löytää ensimmäisestä osasta lisää virheitä kuin toisesta. [13, 14]

Testaus on todella luovaa ja älykkyyttä haastavaa työtä. Ohjelman testauksessa on useita huomioonotettavia asioita ja se on ohjelman laadun kannalta tärkeää. Testaus vaatii aikaa ja on kuluttavaa, joten suurien ohjelmistojen testausta kannattaa tehdä pienissä osissa, etteivät yksittäiset testauskerrat veny liian pitkiksi ja keskittyminen laskee. [13]

4.2 Ohjelmistotestaus eri tasoilla

Kuten edellisessä kappaleessa kerrottiin, testauksen tarkoituksena on etsiä virheitä ohjelmistosta. Virheiden etsiminen satunaisella tarkastelulla ja ilman suunnitelmaa on työlästä ja todennäköisesti suuri osa virheistä jäisi löytämättä. Tästä syystä testausta on hyvä tehdä eri tasoilla ja käydä tällä tavalla jo ohjelman kehitysvaiheessa mahdollisia virhetilanteita läpi. Kokonaisuutta ei tarvitse haravoida kerralla ja löytyneet virheet voidaan kohdistaa tarkemmin oikeaan paikkaan korjaamista varten.

4.2.1 Yksikkötestaus

Yksikkötestauksella (engl. unit testing) tarkoitetaan yhden aliohjelman, moduulin tai luokan testausta. Yksittäisen ohjelman osan testauksella on monia hyötyjä. Yksikkötestauksessa ohjelmaa on helpompi hallita ja testaus on helpompaa, koska testattava osuus on pieni. Tämä helpottaa virheiden etsintää, koska virheen tiedetään löytyvän tietystä osasta ohjelmaa. Yksikkötestausta voidaan tehdä samanaikaisesti monelle eri ohjelman osalle. [13]

Esimerkkitapausten suunnittelu yksikkötestausta varten vaatii tarkan määrittelyn moduulista ja ohjelman lähdekoodin. Yksikkötestaus on parasta tehdä white box -testauksena, jotta ohjelman funktionaalisuus tulee testattua hyvin mahdollisia rakenteellisia virheitä varten. White box -testausta on huomattavasti vaikeampi toteuttaa isolle kokonaisuudelle ja sen tehokkuus on parhaimmillaan yksikkötestauksessa, jossa eri päätökset ja ohjelmankulku saadaan kattavasti testattua. [13]

4.2.2 Integraatiotestaus

Kun yksittäinen osa ohjelmaa on testattu ja se toimii yksinään oikein, nämä moduulit yhdistetään toisiinsa kokonaisuudeksi. Integraatiotestauksella tarkoitetaan testauksessa sitä vaihetta, jossa eri moduuleja aletaan yhdistää kokonaisuutta varten. Tätä integraatiota

pitää testata moduuli kerrallaan, jotta osien yhdistäminen ei aiheuta virheitä muihin osiin. Suurissa ohjelmaprojekteissa saattaa olla kymmeniä tai satoja ohjelmoijia ja eri moduulit saattavat olla kokonaan eri tiimien luomia. Eri tiimit saattavat työskennellä kokonaan eri paikoissa ja tämä voi aiheuttaa väärinymmärryksiä dokumentaatioissa johtaen uusiin virheisiin. [15]

Ohjelman osan testaaminen yksikkötestausvaiheessa voi olla hankalaa, koska toimiakseen oikein moduuli vaatii useampia osia. Jos yksikkötestauksessa on jouduttu käyttämään ylimääräisiä testimoduuleja osoittamaan ohjelman toimivuus, tällaisen moduulin integroiminen oikeiden moduulien kanssa voi aiheuttaa ennalta-arvaamattomia virheitä. Osa moduuleista saattaa olla itsessään monimutkaisempia toteuttaa ja olla taipuvaisia virheisiin. Tällaisen moduulin yhdistäminen muihin voi tuoda haasteita. [15]

4.2.3 Järjestelmätestaus

Kun kaikki moduulit saadaan integroitua yhteen kokonaiseksi järjestelmäksi, tulee ohjelma testata kokonaisuutena. Tätä vaihetta kutsutaan järjestelmätestaukseksi. Järjestelmätestauksen tarkoituksena on tarkistaa vastaako implementaatio asiakkaan asettamia määrityksiä. Koska kokonainen ohjelma on todella suuritöinen testata kokonaan, testaus jaetaan pienempiin osiin, joissa keskitytään eri aspekteihin. Järjestelmätestauksen eri luokat on esitelty taulukossa 4.1. [15]

Perustestaus	ohjelma voidaan asentaa, konfiguroida ja saada käyttökuntoon
Funktionaalisuustestaus	ohjelma tekee pääpiirteittäin halutut toimenpiteet määrityksien mukaan
Palautuvuustestaus	ohjelma palautuu virhetilanteesta jälleen operatiiviseen tilaan
Yhteentoimivuustestaus	ohjelma pystyy toimimaan ulkopuolisten ohjelmien kanssa
Suorituskykytestaus	mitataan ohjelman perussuorituskykyä, esimerkiksi vasteaikoja
Skaalautuvuustestaus	testataan miten hyvin ohjelma sopeutuu käyttäjien, ympäristön tai resurssien skaalaukseen
Stressitestaus	lisätään ohjelman stressitasoa rajojen ja virhetilanteiden lisääntymisen mittaamista varten
Kuormitus- ja vakaustestaus	pidetään ohjelmaa käynnissä täydellä kuormituksella pitkiä aikoja ja mitataan miten hyvin ohjelma kestää kuormituksen
Luotettavuustestaus	käytetään ohjelmaa pitkiä aikoja ja mitataan miten pitkään ohjelma pystyy toimimaan ilman virhetilanteita
Regressiotestaus	ohjelma toimii ilman virheitä, vaikka uusia aliohjelmia tai huoltotoimenpiteitä tehdään ohjelmalle
Dokumentaatiotestaus	varmistetaan, että ohjelman dokumentaatio pitää paikkansa ja vastaa ohjelman sisältöä
Regulaatiotestaus	ohjelma täyttää kaikki maakohtaiset vaatimukset ja toimii lain puitteissa [15]

Taulukko 4.1: Järjestelmätestauksen luokat [15]

4.2.4 Hyväksyttämistestaus

Kun ohjelma on saatu valmiiksi ja kaikki yksittäiset osat on testattu, integroitu yhteen ja kokonaisuutta on testattu uudelleen eri osa-alueilla, ohjelma annetaan asiakkaan tai ostajan testattavaksi, jotta kaikki halutut ominaisuudet löytyvät, ohjelma tekee mitä sen halutaan tekevän ja kriteerit täyttyvät. Tätä testausvaihetta kutsutaan hyväksyttämistestaukseksi. Sillä pyritään varmistamaan, että kaikki asiakkaan vaatimat kriteerit täyttyvät ennen ohjelman käyttöönottoa. [15]

Hyväksyttämistestausta varten luodaan hyväksymiskriteerit. Näistä sovitaan ennen hyväksymistestausta ja testauksessa tarkistetaan näiden kriteerien toiminta. Nämä kriteerit pohjautuvat pääasiassa ohjelman alkuperäisiin määrityksiin siitä, millaisia ominaisuuksia ohjelmassa halutaan olevan ja mitä ohjelman halutaan tekevän. Hyväksyttämistestausta on kahdenlaista: *käyttäjän hyväksyttämistestaus* (engl. user acceptance testing) ja *liiketoiminnan hyväksymistestaus* (engl. business acceptance testing). Käyttäjän hyväksyttämistestauksen tekee asiakas ja sen tarkoituksena on tarkistaa, että ohjelma vastaa alkuperäisiä määrityksiä. Käyttäjän hyväksyttämistestauksen voi tehdä myös asiakkaan puolesta kolmas osapuoli, joka tekee testauksen hyväksymiskriteerien pohjalta. Liiketoiminnan hyväksyttämistestauksen tekee ohjelman toimittaja, eli ohjelman luonut yritys. Tällä on tarkoitus varmistaa yrityksen sisällä, että ohjelma tulee läpäisemään asiakkaan testit. [15]

4.3 Black box- ja white box -testaus

Laatikkotestauksilla tarkoitetaan tapaa käsitellä testausta. Tässä kappaleessa käsitellään white box- ja black box -testausta, mutta näiden väliin voitaisiin vielä laskea myös vähemmän käytetty gray box -testaus, joka käyttää periaatteita molemmista näistä.

4.3.1 Black box

Black box -testauksella tarkoitetaan funktionaalista syöte-tulos-lähtöistä testausta. Siinä testattavaa ohjelmaa tai sen osaa käsitellään mustana laatikkona, jolloin ohjelman sisälle ei nähdä lainkaan. Tässä tilanteessa ohjelman testausta voidaan suorittaa vain määrittelemällä sille eri syötteitä ja oletettuja tulosteita ohjelman alkuperäisen tarkoituksen mukaan. Syötteisiin tulee valita määrityksen mukaisia, valideja arvoja, joiden oletetaan toimivan suoraan oikein, mutta myös tarkoituksella invalideja arvoja, jotka aiheuttavat virheen. Odotettuja tuloksia verrataan todellisiin tulosteisiin testin jälkeen. [15, 16]

Black box -testausta voidaan käyttää missä tahansa vaiheessa ohjelman testausta ja minkä tahansa suuruiselle ohjelman osalle. Black box -testausta varten ei tarvitse olla ohjelmoija tai tietää testattavasta ohjelmasta mitään, kunhan ohjelman alkuperäiset määrittelyt ovat hyvät. [15, 16]

Ekvivalenssiluokkatestaus

Ekvivalenssiluokkatestauksessa testattavat syötteet jaetaan luokkiin, jotta testattavia syötteitä saadaan vähennettyä. Luokkien ei tarvitse olla saman kokoisia, mutta luokkien syötteiden pitää ohjata ohjelman kulkua samaan suuntaan.

Esimerkki 11. Jaetaan henkilöt ikäluokittain niin, että alle 16-vuotiaita ei palkata, 16-18-vuotiaat voidaan palkata osa-aikaisesti, 18-55-vuotiaat voidaan palkata kokopäiväisesti ja yli 55 vuotiaita ei palkata. (taulukko 4.2) [16]

0-16	ei palkata
16-18	palkataan osa-aikaisesti
18-55	palkataan kokopäiväisesti
55-99	ei palkata

Taulukko 4.2: Palkattavat henkilöt [16]

Olisi liian työlästä testata ohjelmaa jokaisella iällä erikseen välillä 0-99. Testausta hel-

pottaa huomattavasti, jos henkilöt jaetaan luokkiin alle 16, 16-18, 18-55 ja yli 55, jolloin testiä ei tarvitse ajaa kuin yhdellä syötteellä jokaista luokkaa kohti. Näitä luokkia kutsutaan ekvivalenssiluokiksi, sillä niistä jokainen on testausmielessä samanarvoinen. Jos luokan yksi syöte aiheuttaa virheen tai poikkeuksen, todennäköisesti jokainen luokan syöte tekee saman. Vastaavasti jos yksi luokan syöte ei aiheuta virheitä, todennäköisesti mikään niistä ei tee niin. [16]

Raja-arvotestaus

Raja-arvotestauksessa keskitytään luokkien raja-arvoihin. Raja-arvoihin jää usein virheitä, jotka saattavat olla jo alkuperäisessä ohjelman määrityksessä. Esimerkissä 11 ikä 16 on sekä ei palkattavien että osa-aikaisesti palkattavien luokassa. Tässä tilanteessa ikäluokat pitää rajata uudelleen niin, ettei sekaannuksia voi syntyä (taulukko 4.3). [16]

0-15	ei palkata
16-17	palkataan osa-aikaisesti
18-54	palkataan kokopäiväisesti
55-99	ei palkata

Taulukko 4.3: Palkattavat henkilöt uusilla rajoilla [16]

Raja-arvotestauksessa valitaan testisyötteiksi ekvivalenssiluokan rajoilta alapuolelta, rajalta ja yläpuolelta yksi arvo. Taulukon 4.3 mukaisesti raja-arvosyötteet testataan ekvivalenssiluokkien rajoilta {-1, 0, 1}, {15, 16, 17}, {17, 18, 19}, {54, 55, 56} ja {98, 99, 100}. [16]

Päätöstaulutestaus

Päätöstaulut ovat tapa kuvata ohjelman määrityksiä ja sääntöjä. Päätöstauluun kirjataan syötteet eri ehtoina vaakariveille, säännöt ehtojen toteutumiselle pystyriveille ja ehtojen alle vaakariveille kirjataan tehtävät. Kun eri syötteiden ehtoja verrataan sääntöihin, niistä

saadaan selville mitkä tehtävät toteutetaan missäkin tilanteessa. Tällä tekniikalla pystytään helpottamaan ohjelman määrityksiä ja eri tehtävien sääntöjen toteuttamista. [16]

Esimerkki 12. Taulukossa 4.4 on yksinkertainen esimerkki ostotapahtuman määrittelystä. Syötteinä ovat varakkuus ja varastotila ja tehtävinä ostotapahtuman toteutuminen. Jos ostoon ei ole varaa tai tuotteita ei ole tarpeeksi varastossa, ostotapahtumaa ei tule.

	Sääntö 1	**Sääntö 2**	**Sääntö 3**	**Sääntö 4**
Syötteet				
Varaa ostaa?	ei	kyllä	ei	kyllä
Tuotteita varastossa?	ei	ei	kyllä	kyllä
Tehtävät				
Ostotapahtuma	ei	ei	ei	kyllä

Taulukko 4.4: Tuotteen ostotapahtuma, sen syötteet ja säännöt.

4.3.2 White box

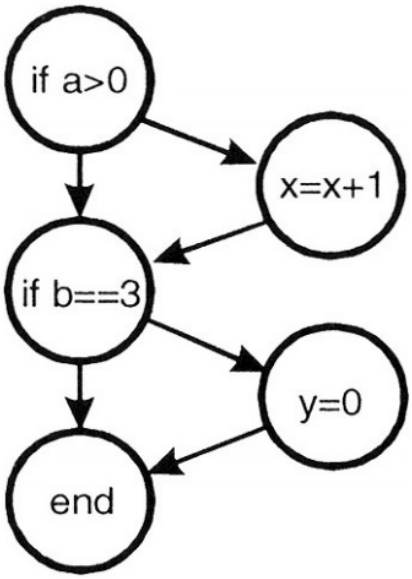
White box -testauksessa on tarkoitus tutkia ohjelman lähdekoodia ja rakennetta ja löytää niistä prosessivirtauksen eri polut. Ohjelman kaikkia mahdollisia polkuja voi olla työlästä tai mahdotonta testata, mutta jokainen solmu- tai päätöskohta ohjelmassa tulisi käsitellä vähintään kerran. [15, 16]

White box -testausta voidaan käyttää missä tahansa vaiheessa ohjelman testausta, mutta mitä suurempi osa ohjelmasta on testattavana, sitä vaikeampaa kaikkien mahdollisten polkujen tai solmujen testaaminen on. Tästä syystä White box -testausta on parempi käyttää yksittäisen ohjelman osien testauksessa yksikkö- tai integraatiotestauksessa. [16]

Prosessivirtaustestaus

Prosessivirtaustestauksessa keskitytään eri polkujen testaamiseen. Prosessivirtauksen osalta ohjelmat rakentuvat prosessikappaleista (engl. process block), joissa ohjelman funktioi-

ta suoritetaan sarjassa alusta loppuun. Prosessikappaleissa on vain yksi tie sisään ja yksi tie ulos. Näiden prosessikappaleiden välillä on päätöspisteitä (engl. decision point). Ne ohjaavat ohjelman suorittamista eri suuntiin joko kaksijakoisesti ehtolauseilla tai moneen eri polkuun määriteltyjen tapausten (engl. case) kautta. Liitoskohdat (engl. junction point) yhdistävät monta eri polkua jälleen yhteen prosessikappaleeseen. Näillä komponenteilla prosessivirtaustestauksessa ohjelman osasta luodaan kuvaaja, jota käytetään esimerkkitapausten luomiseen eri polkujen testaamiseksi (kuva 4.1). [16]



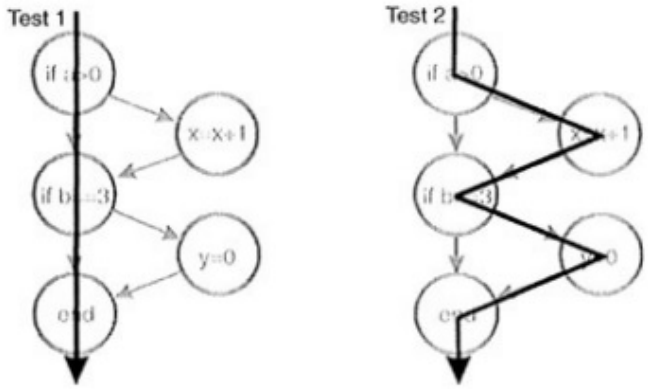
Kuva 4.1: Esimerkki yksinkertaisesta prosessivirtauskuvaajasta. [16]

Prosessivirtaustestaus voidaan jakaa eri kattavuuden tasoihin. Kattavuudella tarkoitetaan kuinka paljon ohjelman osan eri poluista testataan. Yleisesti kaikki tasot sisältävät tavan testata 100 % jollain tavalla ohjelmaa, mutta silti osassa yrityksissä käytetään tapaa ”testataan mitä halutaan ja annetaan käyttäjän kokeilla loput”. Alin testauksen taso on testata 100 % prosessikappaleiden lausekkeista (engl. statement). Tällöin tavoitteena on luoda esimerkkitapauksia tarpeeksi, jotta jokainen lauseke testataan vähintään kerran. Ongelmana on, ettei kaikkia mahdollisia polkuvaihtoehtoja testata. Esimerkiksi kuvan 4.1 kuvaaja voidaan kirjoittaa kahdella koodirivillä:

```
if (a>0) {x=x+1;}
if (b==3) {y=0;}
```

Jos halutaan testata kaikki lausekkeet näistä kahdesta koodirivistä, valitaan arvoiksi vaikkapa `a=6` ja `b=3`. Tällöin kaikki lausekkeet käydään läpi, vaikka useita polkuvaihtoehtoja jää käymättä läpi. [16]

Seuraava prosessivirtaustestauksen taso on testata 100 % päätöksistä. Tällöin testataan kaikki `TOSI`- ja `EPÄTOSI` -valintoja sisältävät solmukohdat molemmilla arvoilla. Tällöin ei tule välttämättä testattua kaikkia polkuja, mutta vähintään kaikki lausekkeet tulee. (kuva 4.2) [16]



Kuva 4.2: Toisella tasolla testataan jokaisen ehtolauseen molemmat päätökset. [16]

Kolmannella tasolla testataan 100 % ehdoista. Tämä on periaatteessa sama kuin päätöksien testaus, mutta jos yksittäisissä päätöksissä on mukana useampia ehtoja kuin vain yksi, kolmannella tasolla testataan niitä. Esimerkkinä koodissa

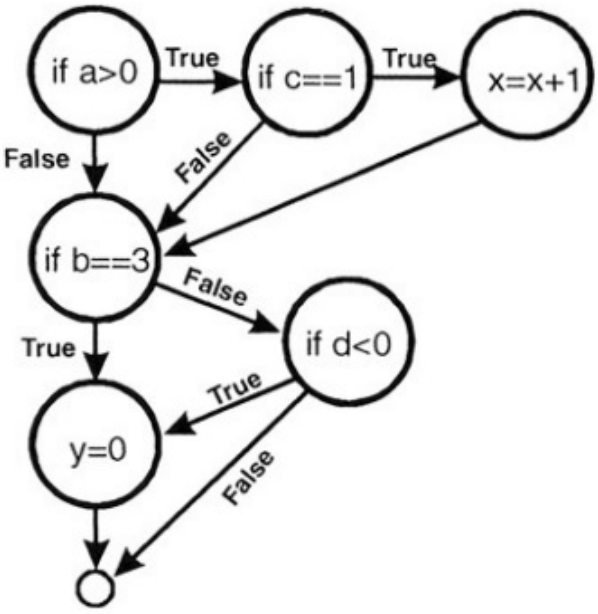
```
if (a>0 && c==1) {x=x+1;}
if (b==3 || d<0) {y=0;}
```

voidaan kaikki yksittäiset ehdot testata erikseen arvoilla $(a > 0, c = 1, b = 3, d < 0)$ ja $(a \leq 0, c \neq 1, b \neq 3, d \geq 0)$. Neljäs taso on muuten sama kuin kolmas, mutta siinä testataan 100 % ehdoista ja päätöksistä. [16]

Viidennellä tasolla testaan myös 100 % ehdoista, mutta mukaan otetaan useampia ehtoja vaikuttamaan päätöksen tekoon. Koodiesimerkki

```
if (a>0 && c==1) {x=x+1;}
if (b==3 || d<0) {y=0;}
```

voidaan esittää kuvaajana kuvan 4.3 mukaisesti. [16]



Kuva 4.3: Esimerkki monen ehdon kattavasta kuvaajasta. [16]

Kuudennella ja seitsemännellä tasolla mukaan otetaan silmukat. Kuudennella tasolla silmukoiden määrää voidaan tarkoituksella vähentää niin, että joko ei suoriteta silmukkaa yhtään kertaa, suoritetaan se tasan yhden kerran, suoritetaan silmukka `n`-määrä tai suoritetaan silmukka sen maksimimäärä. Seitsemännellä tasolla on tarkoitus testata 100 % poluista eli kaikki mahdolliset lausekkeet, ehdot ja päätökset käydään kattavasti läpi. Ilman silmukoita tällaisen määrän esimerkkitapauksia kehittäminen on mahdollista, mutta silmukoiden kanssa se voi osoittautua melkein mahdottomaksi. [16]

Datavirtaustestaus

Datavirtaustestaus keskittyy muuttujien käsittelyyn. Datavirtaustestausta on kahdenlaista: staattista ja dynaamista. *Staattisessa datavirtatestauksessa* ohjelmaa ei ajeta lainkaan, vaan koodia tarkastellaan ja sieltä etsitään määrittely-käyttö-tuhoaminen-rakennetta. Tähän

käytetään avuksi prosessivirtaustestauksessa käytettyä kuvaajaa (kuva 4.1) muuttujien operaatioita varten. *Dynaamisessa datavirtaustestauksessa* ohjelmaa joko ajetaan suoraan tai käydään koodissa ohjelman polut läpi ja etsitään jokaisen muuttujan määrittely-käyttöparit ja luodaan esimerkkitapauksia näille. [17]

Jokainen muuttuja tulisi alustaa ennen sen käyttöä ja muuttujat usein tuhoutuvat automaattisesti, kun poistutaan sen määritte-lyalueen ulkopuolelle. Perusoperaatiot muuttujilla ovat määrittely (engl. define), käyttä-minen (engl. use) ja tuhoaminen (engl. kill). Joissain kielissä muuttujan määrittely tapahtuu automaattisesti, mutta monissa kielissä (esim. C++, Java) muuttujat pitää määritellä manuaalisesti. Esimerkiksi koodirivit

```
int x;
string y;
```

alustavat muuttujan `x` kokonaisluvuksi ja muuttujan `y` merkkijonoksi. Datavirtaustestauksen tarkoitus on löytää virheet muuttujien käytöstä ajamatta ohjelmaa. Esimerkiksi tilanteessa, jossa muuttujaa käytetään ennen sen määrittelyä tai määritellään muuttuja, muttei käytetä sitä missään kohtaa. [16]

4.4 Visuaalisen käyttöliittymän testaus

Monilla ohjelmilla on visuaalinen käyttöliittymä (engl. graphical user interface, GUI), jonka kautta ohjelmaa käytetään. Käyttöliittymä voi olla esimerkiksi nettisivu, ohjelman oma käyttöliittymä tai komentorivi. Käyttöliittymä on hyvä testata, jotta jokainen käyttäjän toimenpide laukaisee oikeat toiminnot, data näytetään käyttäjälle oikein ja käyttöliittymän tila muuttuu oikein. [18]

Käyttöliittymän testauksen voi karkeasti jakaa kahteen luokkaan: käytettävyystestaukseen ja komponenttitestaukseen. *Käytettävyystestauksessa* on tarkoitus tutkia onko ohjelma helppokäyttöinen, helppo oppia, auttaako se käyttäjää olemaan tehokas ja aikaansaava ja onko ohjelmaa mukava käyttää. *Komponenttitestauksessa* on tarkoitus testata tekevätkö

kaikki käyttöliittymän komponentit mitä niiden on tarkoitus tehdä, toimivatko syötekentät oikein ja voiko niihin syöttää vain oikeanlaista dataa ja siirtyykö ohjelma oikeaan tilaan, kun tietyt toimenpiteet tehdään. [18, 19]

Käyttöliittymän testaukseen on kolme tekniikkaa: käsikirjoitettu testaus, (engl. scripted testing), tutkiva testaus (engl. exploratory testing) ja käyttäjäkokemustestaus (engl. user experience testing). Näistä käsikirjoitettu ja tutkiva testaus ovat pääasiassa komponenttitestausta, jolla selvitetään tekeekö ohjelma haluttuja toimenpiteitä oikein ja toteuttaako se halutut määritykset. *Käyttäjäkokemustestauksella* testataan ohjelman käytettävyyttä eli onko ohjelmaa helppo käyttää ja täyttääkö se käyttäjän toiveet. Käyttäjäkokemustestaus ja aiemmin kappaleessa 4.2.4 mainittu hyväksyttämistestaus ovat täysin eri asioita. Hyväksyttämistestauksessa asiakas tai asiakkaan edustaja selvittävät täyttyvätkö määritykset ja tekeekö ohjelma mitä sen halutaan tekevän, mutta käyttäjäkokemustestauksessa täysin ulkopuoliset käyttäjät testaavat löytyykö kaikki tarpeellinen loogisista paikoista, onko ohjelma johdonmukainen ja helposti ymmärrettävä. [20]

Käsikirjoitetulla testauksella tarkoitetaan etukäteen tarkasti kirjoitettujen ohjeiden mukaan tehtyä testausta. Käsikirjoituksessa kerrotaan mitä käyttäjän tulee tehdä jokaisessa testin vaiheessa ja mikä on oletettu toimenpiteen tuotos. Saadut testitulokset kirjataan ylös ja niiden avulla on helppo osoittaa, että testaus on ollut tarkkaa ja täsmällistä ja niiden avulla voidaan aikaisessa vaiheessa tunnistaa, puuttuuko ohjelmasta määrityksien mukaisia ominaisuuksia. Huono puoli käsikirjoitetussa testauksessa on se, että se vaatii ylimääräistä aikaa suunnitella tarkat käsikirjoitukset testeille ja huomaamatta saattaa jäädä joitain virheitä, jotka tutkivalla testauksella olisi löytynyt. [20]

Tutkivassa testauksessa testaaja käyttää tietotaitoaan ja kokemuksiaan testien suunnitteluun ja toteuttaa niitä suoraan heti. Nämä testit saattavat tuoda esille muita testattavia kohteita, uusia testejä kehitetään lennosta ja testaaja antaa näistä suoraan palautetta kehittäjille. Tutkivassa testauksessa testaajan tulee olla kokenut ohjelmistokehityksen testaajana, jotta pystyy kehittämään hyviä testejä ohjelmalle ja koko ohjelman kattava testaa-

minen saattaa olla hankalaa. Asiakkaalle voi olla myös vaikea todistaa, että koko ohjelma on kattavasti ja tarkasti testattu käyttäen vain tutkivaa testausta ja koska kaikkia testejä ei välttämättä kirjata ylös, testien uudelleen tekeminen voi olla vaikeaa ohjelman muuttuessa. Testien uudelleen tekemistä ohjelman muuttuessa kutsutaan regressiotestaukseksi. [20]

4.5 Ohjelmiston tarkastelu, kertaus ja arvostelu

Ohjelmiston tarkastelulla (engl. code review) tarkoitetaan ohjelman läpikäymistä testitiimin kanssa. Tällaisia koodinlukutilaisuuksia on muutamia erilaisia, joissa idea on sama: koodi käydään läpi ja ohjelmasta yritetään löytää virheitä. Osassa tällaisissa tilaisuuksissa osallistujien tulee tutustua ohjelmaan etukäteen ja itse tarkastelutilanteessa keskitytään virheiden löytämiseen eikä niiden korjaamiseen. Tarkastelutilanteessa 3-4 hengen ryhmässä käydään ohjelman koodia läpi. Mukana ryhmässä voi olla testiryhmän moderaattori, joka järjestää tilaisuuden, ohjelman kehittäjä, joka esittelee ohjelman koodia, ohjelman suunnittelija ja testausasiantuntija. Muita ryhmäläisiä voivat olla junioriohjelmistokehittäjä, joka tuo tarkasteluun uusia näkemyksiä, ohjelman ylläpitäjä tai ohjelmistokehittäjä toisesta projektista. [13]

Koodin tarkastelun myötä virheet löytyvät tarkasti. Kuten kappaleessa 4.1 kerrottiin, kehittäjän ei pidä tehdä oman ohjelmansa testausta, koska omia virheitä on usein hankala havaita. Tarkastelussa muut henkilöt näkevät ohjelman eri tavalla ja löytävät kehityskohteet paremmin ja tarkat korjauskohdat löytyvät heti. Tällä tavalla myös ajatusvirheet ja väärinymmärrykset ohjelman tarkoituksesta tulevat varhaisessa vaiheessa esille. [13]

4.6 Esimerkkitapaustestaus

Esimerkkitapaustestauksella (engl. test case testing) tarkoitetaan ohjelman normaalista toiminnasta esimerkkitapausten tekemistä, joilla testataan täyttääkö ohjelma annetut mää-

ritykset. Pääasiassa esimerkkitapaukset voi jakaa kolmeen luokkaan: määrityksien mukaan tehtyihin (engl. specification-based), rakenteen mukaan tehtyihin (engl. stucture-based) ja kokemuksien mukaan tehtyihin (engl. experience-based) testitapauksiin. [21]

Määrityksien mukaan tehdyt esimerkkitapaukset ovat perustuvat alkuperäisiin ohjelman määrityksiin mitä ohjelman halutaan tekevän. Tällöin käytetään black box -testauksen toimenpiteitä (kappale 4.3.1) ja testataan lähinnä syöte-tuotos-pohjaisesti ohjelmaa. *Rakenteen mukaan tehtyihin esimerkkitapauksiin* käytetään white box -testauksen tapoja (kappale 4.3.2) ja testataan ohjelman eri polkuja ja näiden tuotoksia. *Kokemuksien mukaan tehdyt testitapaukset* perustuvat testaajan omiin kokemuksiin testauksesta ja ymmärrykseen ohjelman oleellisista testattavasti kohteista. Testaustavoista tutkiva testaus (kappale 4.4) lukeutuu tähän luokkaan. [21]

Testitapaukset kannattaa yleisesti tehdä mahdollisimman yksinkertaiseksi. Hyvän tapauksen rakentamiseen kannattaa käyttää muistilistaa:

1. **Niiden tulee olla yksinkertaisia.**

   Käytettävän kielen tulee olla yksiselitteistä, jotta testi on helppo toteuttaa. Elementeistä tulee puhua absoluuttisilla tunnisteilla, ettei sekaannuksen mahdollisuutta tule. Tapauksessa tulee olla vain tärkeät askeleet eikä mitään ylimääräistä.

2. **Niiden tulee olla määritysten mukaisia.**

   Esimerkkitapausten käytön tarkoitus on testata, että ohjelma täyttää määritykset ja sitä on helppo käyttää. Testitapauksen tulee testata kohdistetusti tiettyä määritystä eikä sen tekemisessä saa olettaa mitään ohjelman toiminnasta.

3. **Niiden tulee kattaa koko ohjelma.**

   Ohjelmasta tulisi testata 100 % sen toiminnasta. Esimerkkitapaukset kannattaa jaotella edellä mainittujen luokkien mukaan ja testata ohjelman eri polut, syötteet ja mahdolliset muut yleiset ongelmakohdat. Esimerkkitapausten tulee kattaa kaikki positiiviset ja negatiiviset tapaukset, jotta koko ohjelma tulee varmasti testattua.

4. **Testien tulee olla toistettavia.**

   Jokainen esimerkkitapaus pitää olla toistettavissa ja palauttaa aina samat tuotokset ohjelmasta riippumatta siitä kuka testin teki. Parhaassa tapauksessa samaa testiä voidaan myös uudelleenkäyttää muissa tilanteissa.

5. **Esimerkkitapaukset tulee dokumentoida kunnolla.**

   Jokainen esimerkkitapaus tulee dokumentoida ennen testin ajamista ja testin tulokset pitää ottaa talteen korjauksia varten. Hyödyllisiä tietoja tapauksen dokumentointia varten ovat esimerkiksi testitapaus-ID, testin kuvaus, testin askeleet, käytettävät syötteet, oletettu tuotos, todellinen tuotos, testin status (läpäisikö testi vaatimukset vai ei) ja testin päivämäärä. [22, 23]

Luku 5

Ohjelmistorobotiikan testaaminen

Tässä luvussa käsitellään ohjelmistorobotiikan testausta. Luvun alussa on käyty läpi käytännön haasteita, minkä takia ohjelmistorobottien testaukseen kannattaa kiinnittää huomiota ja panostaa. Tätä varten haastattelin eri yrityksistä viittä henkilöä, jotka tekevät ohjelmistorobotteja työkseen. Tämän jälkeen käsitellään miten luoda hyviä esimerkkitapauksia, joilla ohjelmistorobotteja voidaan testata tehokkaasti. Pääpainona on verrata perinteisen ohjelmistokehityksen testaustapoja ohjelmistorobottien testaukseen ja yrittää löytää hyviä tapoja sitä kautta.

Testausta tehdään usein jatkuvasti kehitystyön ohessa. Yksittäisiä komponentteja testataan yksikkötestauksena sitä mukaa, kun komponentteja kehitetään ja saadaan valmiiksi. Tässä diplomityössä keskitytään valmiin robotin testaukseen, eli järjestelmä- ja hyväksyttämistestausvaiheisiin.

Ohjelmistorobottien kehitystyö on tarkoitus olla nopeaa. Ohjelmistorobotin määrittelyihin asiakkaan puolelta voi mennä pitkäkin aika, mutta robotin kehitystyön tulisi kestää korkeintaan kuukaudesta kahteen. Tämä aikataulu nousi esille useassa haastattelussa. Ohjelmistorobotin kehitystiimin pitäisi pystyä keskittämään pelkästään ohjelmistorobottien luomiseen ja tällöin aikaa ei saisi kulua paljoa. Ohjelmistorobotin määrittelyihin sisältyy toimintaym-päristön ymmärtäminen, prosessien valitseminen jatkotutkintaan, prosessien tarkat kuvaukset, päätös mitkä prosessit päätyvät robotiikka-analyysiin ja yksityiskohtai-

nen suunnittelu robotisoinnille. Määritysten jälkeen työ siirtyy kehitykseen, johon sisältää robotin luominen, testaus, käyttöönotto ja ylläpito. Koska kehitystyön tulee olla määritysten tekemisen jälkeen nopeaa, testaukseen ei voida käyttää paljoa aikaa.

5.1 Käytännön haasteet työelämässä

Työelämässä kehitystyön haasteet konkretisoituvat. Haastattelin viittä ohjelmistorobotiikan parissa työskentelevää henkilöä työelämän haasteista ja kokosin niiden pohjalta suurimmat haasteet yhteen.

5.1.1 Asiantuntijahaastattelu tutkimusmetodina

Haastattelin diplomityötäni varten Sampsa Vuorelaa, Miika Länsi-Seppästä, Pyry Vanamoa, Mikko Hälistä ja Hilkka Hangasmäkeä. Jokainen heistä tekee työkseen ohjelmistorobotiikkaa, joten heiltä sain tietoa eri yritysten käyttämistä tavoista robotiikan kehityksessä.

Sampsa Vuorela on tehnyt runsaan vuoden töitä RPA:n parissa. Hän on ollut mukana robottien koko kehitysyklissä: määrittelyssä, toteutuksessa, hiomisessa ja ylläpidossa. Ennen RPA-töihin siirtymistä hän on työskennellyt kaksi vuotta ohjelmistokehityksessä. Robotiikassa pääpaino on ollut Pythonin käytössä, mutta muutamissa projekteissa on käytetty myös muita kieliä ja ohjelmistoja.

Miika Länsi-Seppäsellä on työkokemusta IT-alalta noin kolme vuotta ja tästä on puolet ohjelmistorobotiikassa. Koko työura on kuitenkin keskittynyt ohjelmistoautomaatioon. Ohjelmistorobotiikassa hän on toiminut lähinnä pienissä projekteissa, joissa on kehitetty jatkuvasti käytössä olevissa robotteja. Miika on työssään käyttänyt Pythonia, mutta on tehnyt robotiikkaa myös muilla ohjelmilla, kuten UiPathillä.

Pyry Vanamo on työskennellyt IT-alalla kolme vuotta ja tästä viimeiset puoli vuotta ohjelmistorobotiikan parissa. Hän ei ole ollut ohjelmistorobottien kehitystyössä paljoa

mukana, vaan hän on hoitanut kehitys-, testi ja tuotantoympäristöjä kuntoon kehittäjiä varten. Hän käsittelee ohjelmistorobotiikkaa korkeammalta tasolta, eli suunnittelee robottien kehitystyötä liikearvon kannalta ja jakaa tehtäviä prioriteetin mukaan eteenpäin kehittäjille. Vanamo on tehnyt ohjelmistorobotiikkaa vain UiPathin kanssa.

Mikko Hälinen on ollut RPA-kehittäjänä melkein kolme vuotta. Hän on työssään käyttänyt pääasiassa vTask-, BluePrism- sekä NICE-ohjelmistoja. Kolmen vuoden aikana hän on osallistunut moniin eri kokoisiin projekteihin, joiden tuotoksena tehtyjä robotteja käytetään osaa useita kertoja ja osaa kerran päivässä.

Hilkka Hangasmäki on tehnyt ohjelmistorobotiikkaa kaksi vuotta. Ohjelmistoista hän on käyttänyt vTaskiä ja Blue Prismiä ja näillä hän on tehnyt sekä monen kuukauden kestäviä että lyhyempiä projekteja. Pääasiassa Hangasmäki on toiminut robottien kehittäjänä, mutta hän on myös vastannut kysymyksiin määritysvaiheen analyyseistä.

5.1.2 Kehitysympäristön puuttuminen

Perinteisessä ohjelmistotestauksessa ohjelman kehittäminen ja testaus tapahtuu usein erillisessä kehitysympäristössä. Kehitysympäristö voi olla täysin erillinen ympäristö tuotannosta, jossa ohjelmaa voidaan kokeilla ilman mitään rajoituksia. Se voi myös olla täysin samanlainen kuin tuotantoympäristö samoilla rajoituksilla, mutta ilman oikeaa tietokannan dataa tai riskejä tuottaa ongelmia varsinaiselle käytössäolevalle tuotantoympäristölle. Kun ohjelmaa on testattu kehitysympäristössä ja se on todettu toimivaksi, ohjelma voidaan siirtää tuotantoympäristöön käsittelemään oikeaa tietokannan dataa. [24]

Ohjelmistorobotiikassa on usein ongelmana kehitysympäristön puuttuminen. Ohjelmistorobotti kehitetään käyttämään tuotannossa olevaa ohjelmaa, jossa on kaikki data jatkuvasti saatavilla. Virhetilanteissa, jossa ohjelmistorobotti tekee väärän toimenpiteen, järjestelmästä saattaa pahimmassa tapauksessa poistua dataa. Tästä syystä robotin testaus tuotantoympäristössä voi aiheuttaa ongelmia ja vakavia rahallisia seurauksia.

Esimerkki 13. Järjestelmään on tallennettu asiakastietoja ja ohjelmistorobotin tarkoituksena on avata asiakkaan tiedot, muuttaa henkilötietoja ja sulkea asiakkuus. Tietyssä vaiheessa ohjelmistorobotin suoritusta robotti on virheellisesti ohjelmoitu valitsemaan väärästä kohdasta eikä päivitäkään asiakkaan tietoja, vaan poistaa ne järjestelmästä. Asiakkaan kaikki palvelut ja tiedot poistetaan järjestelmästä ja robotti ilmoittaa ohjelman suorituksen olevan valmis. Jos kyseessä olisi ollut kehitysympäristö ja malliasiakas, virhe ohjelmassa ei olisi aiheuttanut ongelmia ja virhe olisi korjattu. Tuotantoympäristössä todellisen asiakkaan tiedot poistuisivat järjestelmästä eikä niitä välttämättä pystyttäisi palauttamaan takaisin.

Kehitysympäristökään ei aina kuitenkaan ratkaise ongelmaa. Tarkalleen legacy-ohjelmiston tuotantoympäristön kaltaisen kehitysympäristön luominen voi olla mahdotonta ja todella kallista, jolloin kaikki kehitysympäristössä testatut ohjelmistorobotit eivät käyttäydy tuotantoympäristössä samalla tavalla. Kehitysympäristössä ei välttämättä myöskään ole tuotantoympäristön kaltaista dataa, jolla ohjelmistorobottia testata tehokkaasti. Jos tällaisessa tilanteessa on pakko testata ohjelmistorobottia suoraan tuotantoympäristössä ja oikealla datalla, sopivan oikean tapauksen kehittäminen tai ilmaantuminen voi olla vaikeaa.

5.1.3 Ohjelmistorobotin määritys

Kuten luvun 5 alussa esiteltiin, ohjelmistorobotin määrityksiin sisältyy toimintaympäristön ymmärtäminen, prosessien valitseminen jatkotutkintaan, prosessien tarkat kuvaukset, päätös mitkä prosessit päätyvät robotiikka-analyysiin ja yksityiskohtainen suunnnittelu robotisoinnille. Ohjelmistorobotin määrityksien lisäksi on tärkeää suunnitella myös robotin käytön määritykset: mitä raja-arvoja robotin toiminnalla on, mitä sen ei haluta tekevän ja milloin robottia halutaan käyttää.

Toimintaympäristön ymmärtämisellä tarkoitetaan automatisoitavan ohjelmiston opettelua ja tuntemista. Toimintaympäristön tuomista haasteistä kerrotaan lisää kappaleessa

5.1.4. Prosessien valitseminen jatkotutkintaan tarkoittaa niiden prosessien, jotka kappaleen 2.2 mukaan kannattaa automatisoida, valitsemista tarkempaan analysointiin. Vain yksinkertaisia ja toistuvia askeleita kannattaa automatisoida ja jos prosessi on liian monimutkainen, siinä tapahtuu liikaa poikkeuksia ja todennäköisesti ohjelmistorobotti tulisi toimimaan huonosti, ohjelmistorobotin kehitykseen ei kannata lähteä. Prosessien tarkat kuvaukset pitää sisällään tarkan esimerkkitapauksen prosessista. Esimerkkitapauksessa käydään kohta kohdalta läpi minkälainen operaatio kyseinen prosessi on ja mitä toimenpiteitä siinä robotin tulisi tehdä. Tämän pohjalta tehdään päätös mitkä prosessit päätyvät robotiikka-anylyysiin, jossa tehdään yksityiskohtainen suunnittelu robotisoinnille. Tässä kohdassa suunnitellaan ohjelmistorobotin arkkitehtuuri ja miten se rakennetaan. Tämän suunnitelman pohjalta robotti siirtyy kehitysvaiheeseen, jossa robotti luodaan.

5.1.4 Automatisoitavan ohjelman tuomat haasteet

Usein automatisoitavat ohjelmistot ovat legacy-ohjelmia, jotka ovat olleet todella pitkään käytössä ja ovat sen verran vanhoja, ettei niiden integroiminen uusien ohjelmistojen kanssa onnistu mitenkään rajapintojen puutteiden vuoksi. Legacy-ohjelmissa saattaa olla ongelmia myös yhteensopivuuden kanssa käyttöjärjestelmien, selaimien ja verkkorakenteiden kanssa. Niitä käytetään edelleen, koska ne tekevät tehtävänsä ja niiden päivittäminen maksaisi todella paljon. Legacy-ohjelman tunteminen automatisointia varten on todella tärkeää, jotta robotti voidaan suunnitella toimimaan optimaalisesti ja mahdollisiin yllättäviin virhetilanteisiin voidaan varautua. Asiakkaan täytyy osata kertoa kaikista mahdollisista tilanteista määrittelyn yhteydessä, jotta robotti voidaan suunnitella niin, että se osaa varautua kaikkiin poikkeustilanteisiin. [25]

Poikkeustilanteiden käsittelyä tulee tehdä mahdollisimman paljon. Legacy-ohjelmaa käytettäessä yksinkertaiset valinnat ja päätökset saattavat tuntua itsestäänselviltä, mutta ohjelmistorobotille nämä kaikki eri yhdistelmät pitää ohjelmoida erikseen. Koska valintoja ja eri polkuja toimenpiteiden tekemiselle on paljon, poikkeustilanteitakin saattaa

tulla usein. Legacy-ohjelmisto saattaa myös toimia välillä hitaammin kuin ohjelmistorobotti ja tässä tilanteessa ohjelmistorobotin tulee odottaa tarpeeksi kauan automatisoitavan ohjelmiston reagointia. Tavallisen tekstikentän täyttämisessäkin kannattaa tehdä erikseen tarkistus sisältääkö tekstikenttä nyt varmasti siihen kirjoitetun tekstin ennen ohjelman jatkamista, jotta myöhemmissä toimenpiteissä ei tapahdu mitään odottamatonta.

Esimerkki 14. Legacy-ohjelmisto halutaan hakevan syötetyllä ID-numerolla asiakas ja muuttavan tämän tietoja. Asiakkaan puhelinnumeroa ja sähköpostiosoitetta ei ole vahvistettu, joten legacy-ohjelmista antaa yhtäkkiä pop up -ikkunan ruudulle ja pyytää vahvistamaan puhelinnumeron ja sähköpostiosoitteen. Ohjelmistorobotin tilannut asiakas oli unohtanut mainita, että legacy-ohjelmisto saattaa antaa pop up -ikkunoita tietyissä tilanteissa ja ohjelmistorobottia ei ole suunniteltu käsittelemään poikkeustapauksia tässä kohtaa ohjelmaa. Ohjelmistorobotti menee virhetilaan eikä pysty jatkamaan suoritusta. Tässä tilanteessa ohjelmistorobotin tulisi osata laittaa työ sivuun, antaa ilmoitus ohjelmistorobotin käsittelijälle ja ottaa jonosta uusi työ käsittelyyn. Jos useampi peräkkäinen työ antaa saman virheen, jonon suorittaminen tulee lopettaa ja käsittelijälle antaa ilmoitus tästäkin.

5.2 Ohjelmistorobotiikan testauksen vertailu ohjelmistokehityksen testaukseen

Testauksen psykologinen puoli on sama ohjelmistokehityksen testauksessa ja ohjelmistorobotiikan testauksessa. Ohjelmistorobottejakin tulisi testata virheiden löytämiseksi, jotta mahdollisilta ongelmatilanteilta vältyttäisiin. Ohjelmistorobotiikan testauksessa ajattelumalli virheiden löytämisestä täytyy kuitenkin kohdistaa ohjelmistorobotin toimintaan eikä automatisoitavan ohjelman virheiden huomioimiseen. Tarkoitus on löytää robotin virheet ja robotin tulee osata mukautua automatisoitavan ohjelman mahdollisiin virheisiin.

Kappaleen 4.1 lista ohjelmistotestauksen periaatteista pätee myös ohjelmistorobotii-

kan testaukseen. Ohjelmistorobotin testaus tulee suunnitella etukäteen ja suunnitelmaan tulee määrittää odotettu tulos. Robotin kehittäjän ja parhaassa tilanteessa kehittävän organisaation tulisi välttää testaamasta omaa ohjelmaansa. Ohjelmistorobotin testauksen tulisi sisältää erilaisia syötteitä, joita käsitellään lisää kappaleessa 5.3. Vaikka voi olla vaikea asennoitua robotin rikkomiseen, tulee silti testauksessa keskittyä virheiden löytämiseen. Tämä tekee ohjelmistorobotiikan testauksesta yhtä vaikeaa kuin ohjelmistokehityksen testaus.

5.2.1 Järjestelmä- ja hyväksyttämistestaus ohjelmistorobotiikassa

Järjestelmätestauksessa suuren ohjelmiston testauksessa on monta kohtaa ja testauksen tulee olla kattavaa ja tehokasta. Ohjelmiston testauksessa on taulukon 4.1 mukaisesti useita testattavia aspekteja, joista kaikkea ei kuitenkaan ohjelmistorobotiikan testauksessa ole tarpeellista suorittaa.

Perus- ja funktionaalisuustestaus ovat samat ohjelmistorobotiikassa kuin ohjelmistokehityksessä. Ohjelmistorobotti voidaan ottaa käyttöön ja se tekee halutut toimenpiteet. *Palautuvuustestausta* kannattaa tehdä ohjelmistorobotille. Ohjelmistorobotti tulee aina ohjelmoida käsittelemään poikkeustilanteet automatisoitavan ohjelman virhetilanteissa, miten robotti antaa ilmoituksen virhetilanteesta ja miten se jatkaa suoritusta virhetilanteen jälkeen. Ohjelmistorobotti ei saa jäädä jumiin virhetilanteeseen, vaan parhaassa tilanteessa samaa toimenpidettä yritetään uudelleen ja jos muutaman kerran jälkeen työtä ei voida jatkaa, virhetilanteeseen ajautunut työ asetetaan sivuun, asiasta annetaan ilmoitus ohjelmistorobotin käsittelijälle ja ohjelmistorobotti jatkaa seuraavaan jonossa olevaan työhön. Jos jonossa olevista töistä pari seuraavaa päätyy samaan virhetilanteeseen eikä toimenpidettä saada suoritettua loppuun, ohjelmistorobotin olisi hyvä huomata tämä, lopettaa jonossa olevien töiden tekeminen ja antaa ohjelmistorobotin käsittelijälle ilmoitus. Ohjelmistorobotin ei ole hyödyllistä suorittaa koko jonoa tyhjäksi samoilla virheilmoituksilla, vaan jonossa olevat työt on hyvä jättää odottamaan robotin korjausta. *Yhteen-*

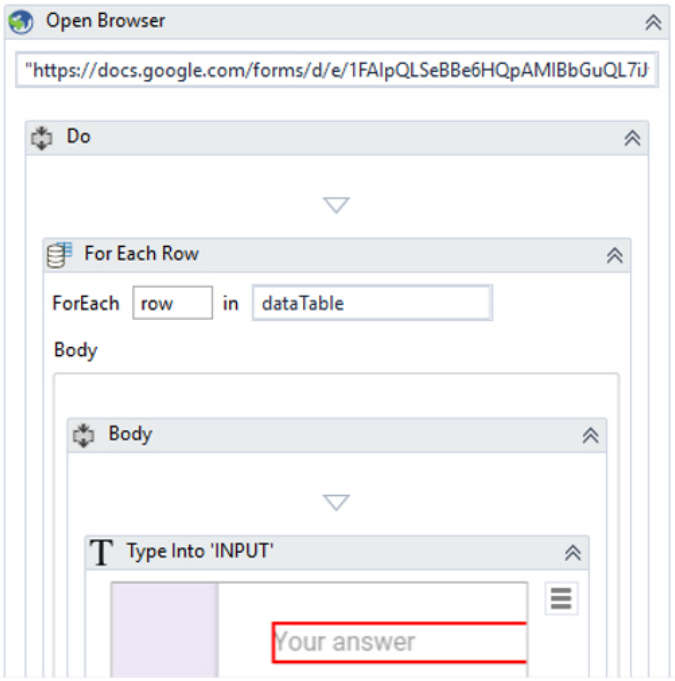
sopivuustestausta ei tarvitse tehdä erikseen, sillä ohjelmistorobotin tarkoitus on toimia muiden ohjelmien kanssa. *Suorituskykytestaus* voidaan tehdä ohjelmistorobotille, mutta siinä tulee testata ohjelmistorobotin suorituskykyä sen toimenpiteissä eikä mitata ohjelmoitavan ohjelman suoritusta, kuten vasteaikoja. *Skaalautuvuustestaus* on ohjelmistorobotin testauksen kannalta epäoleellinen, koska robotin ei ole tarkoitus skaalautua suurempaa käyttöä varten, vaan suurempaa käyttöä varten ohjelmistorobottien määrää voidaan kasvattaa. *Stressitestaus*, *kuormitus- ja vakaustestaus* ja *luotettavuustestaus* ovat myös ohjelmistorobotiikan kannalta epäoleellisia, sillä ohjelmistorobotin ei tulisi missään tilanteessa joutua kovan stressin alle yleensäkään, koska se on luotu toteuttamaan vain tiettyä tehtävää. *Regressiotestaus* on ohjelmistorobotiikassa yhtä tärkeää kuin ohjelmistokehityksessäkin. Vaikka ohjelmistorobotit luodaan toteuttamaan tiettyä tehtävää, niitä voidaan hyötykäyttää myös uusien robottien luomisessa tai niiden kehitystyötä voidaan jatkaa lisäämällä samalle ohjelmistorobotille uusia aliohjelmia. *Dokumentaatiotestaus* ja *regulaatiotestaus* ovat yleisiä testauksia, joilla varmistetaan ohjelman riittävä dokumentaatio ja sen laillisuus, joten ne pätevät myös ohjelmistorobotiikkaan.

Hyväksyttämistestaus toimii ohjelmistorobotiikassa aivan kuten ohjelmistokehityksessäkin. Ohjelmistorobotti tulee hyväksyttää asiakkaalla, kun kehitystyö on saatu valmiiksi. Vaikka ohjelmistorobotin määritysvaihe voi kestää kauemmin kuin kehitysvaihe, ohjelmistorobotin hyväksyttämiskriteerien luominen alkuperäisten ohjelman määritysten pohjalta ei välttämättä vie paljoa aikaa. Ohjelmistorobotti luodaan tekemään tiettyä tehtävää, joten sen hyväksymiskriteerit eivät välttämättä ole kovin laajat. Tällöin hyväksyttämistestauskin on kuin ohjelmistorobotin esittelytilaisuus eikä niinkään vaativa testaustilanne. Kuten kappaleessa 5 kerrottiin, ohjelmistorobotiikan kehityksen tulisi olla nopeaa, joten kolmannen osapuolen käyttäminen hyväksyttämistestauksessa ei ole tarpeellista.

5.2.2 Ohjelmistorobotin tarkastelu

Ohjelmistorobotin tarkastelua voidaan tehdä samalla tavalla kuin ohjelmistokehityksessäkin. Vastaavasti kuin ohjelmistokehityksessä paikalla olisi hyvä olla kehittäjiä toisista projekteista ohjelmistorobotin kehittäjän lisäksi. Jos ohjelmistorobotin kehitystapa on graafinen käyttöliittymä ja aktiviteetit vedetään hiirellä paikalleen, koodinlukutilaisuudessa paikalla voi olla myös kehitystiimin ulkopuolisia henkilöitä, jotka osaavat käyttää automatisoitavaa ohjelmistoa hyvin. Tällöin väärinymmärrykset ohjelman määrityksestä saadaan hyvin korjattua, mutta myös automatisoitavan ohjelman käyttötapoihin voidaan kiinnittää huomiota ja robotin toimintaa voidaan saada parannettua.

Esimerkki 15. Käydään ohjelmistorobotin tarkastelussa läpi robotti, jonka tarkoitus on poimia csv-tiedostosta dataa ja kirjoittaa se Google Forms -lomakkeeseen. Kuvan 5.1 mukaan luodaan ohjelmistorobotti, joka avaa ensin halutun lomakkeen selaimella, käy kaikki `dataTable`-muuttujan rivit läpi ja syöttää ne yksitellen lomakkeen kenttiin. Tällaisen robotin läpikäyminen kohta kohdalta on yksinkertaista ja ilman ohjelmistorobotiikkakokemustakin oleva henkilö ymmärtää mitä missäkin kohdassa tapahtuu. Tässä tilanteessa kehitystiimin ulkopuolisella henkilöllä voi olla tärkeää tietoa lomakkeen käyttötarkoituksesta tai csv-tiedostossa olevan datan muotoilusta, jotka vaikuttavat ohjelmistorobotin kehitystyöhön. Esimerkiksi lomakkeen kautta syötetyistä tiedoista täytyy korvata kirjaimet ä, ö ja å kirjaimilla a, o ja a tai nimet täytyy muotoilla isolla alkukirjaimella ja muut kirjaimet pienellä. [26]



Kuva 5.1: Ohjelmistorobotin tarkasteluesimerkki. [26]

5.3 Ohjelmistorobotiikan esimerkkitapaukset

Järjestelmätestaus ohjelmistorobotiikassa tehdään kokonaan esimerkkitapausten avulla. Esimerkkitapauksiin tulee suunnitella sellaisia syötteitä, jotka käyvät kaikki mahdolliset syöteluokat (black box -testaus) ja polut (white box -testaus) läpi. Kappaleessa 4.6 esitetyt asiat pätevät myös ohjelmistorobotiikan esimerkkitapausten suunnitteluun. Jokaisen esimerkkitapaus tulee kohdistaa tiettyyn ohjelmistorobotin osaan ja niiden tulee olla yksinkertaisia. Koko ohjelmistorobotti tulee testata läpi ja näistä tulee kirjoittaa hyvät dokumentaatiot. Dokumentaatiot voidaan näyttää asiakkaalle hyväksyttämistestauksen yhteydessä, jotta voidaan todentaa ohjelmistorobotin testauksen olleen kattava.

5.3.1 Black box -menetelmät

Black box -testauksessa keskitytään syöte-tulos-pohjaiseen testaukseen ja tämä on hyvä tapa testata osaa ohjelmistoroboteista. Ohjelmistorobotiikassa black box -testauksen tekeminen voi olla hyvin erilaista perinteiseen ohjelmistokehitykseen verrattuna, sillä ohjelmistorobotti ei välttämättä saa lainkaan syötteitä. Ohjelmistorobotti saattaa kuitenkin kerätä dataa tiedostoista, nettisivuilta tai lukea sitä ruudulta kaavintatyökalulla ja tätä voidaan pitää syötteenä ohjelmalle. Tämä data käsitellään ja tallennetaan tai syötetään toiseen paikkaan, jolloin saadaan ohjelmasta tulos. Ohjelmistorobottia voidaan testata muuttamalla tätä luettavaa dataa ja näin voidaan myös hyödyntää black box -testauksen ekvivalenssiluokka- ja raja-arvotestausta.

Ekvivalenssiluokkia voi olla vaikea luoda ohjelmistorobotille. Syötettävät tai kerättävät syötteet ovat harvoin luokiteltavissa erityisen tarkkaan kategorioihin, mutta jos niille tehdään eri toimenpiteitä eli ne menevät ohjelmassa eri polkuja, ekvivalenssiluokat voidaan yhdistää white box -testauksen prosessivirtaustestaukseen (kappale 4.3.2). Tästä kerrotaan lisää kappaleessa 5.3.2. Kuitenkin validien ja epävalidien syötteiden jaottelu onnistuu ekvivalenssiluokkiin ja niiden avulla voidaan toteuttaa myös raja-arvotestaus. Tällöin valitaan syötteisiin rajan alapuolelta, rajalta ja yläpuolelta arvoja ja testataan ohjelmistorobottia niillä. Raja-arvotestauksessa voidaan käyttää myös esimerkiksi merkkimäärältään eri pituisia syötteitä salasanassa tai kuinka monta syötettä ohjelmistorobotti voi saada.

5.3.2 White box -menetelmät

White box -testauksessa käydään ohjelman polut läpi. Yleisesti white box -testausta kannattaa käyttää yksikkö- tai integraatiotestauksessa, koska suuren ohjelman kaikkien polkujen läpikäyminen on todella työlästä. Ohjelmistorobotiikassa tarkoitus on kuitenkin suunnitella robotti tekemään vain tiettyä tehtävää, joten polkuja tai silmukoita ei ohjelmassa pitäisi olla kovin montaa. Tästä syystä white box -testausta voidaan käyttää ohjelmistorobotiikassa myös järjestelmätestauksessa.

Ohjelmistorobotiikassa prosessivirtaustestauksesta on hyvä käyttää vähintään tasoa viisi. Tällöin kaikki ehdot ja päätökset testataan läpi ja ohjelma on testattu melkein kokonaan. Yleensä ohjelmistorobotit ovat toiminnassaan niin suoraviivaisia, ettei solmukohdissa ole useampia ehtoja tai päätöksiä. Joskus kuitenkin robotti voi tehdä päätökset useamman syötteen tai eri paikoista kerätyn datan mukaan, jolloin kaikki nämä vaihtoehdot olisi hyvä käydä läpi. Tasoja kuusi ja seitsemän kannattaa käyttää, jos ohjelmistorobotissa on silmukoita.

Datavirtaustestaus on ohjelmistorobotiikassa erittäin tärkeää. Muuttujien käsittely on yksi ohjelmistorobotiikan avainasioita, sillä lähes jokainen ohjelmistorobotti kerää dataa ja käyttää sitä jotenkin. Ohjelmistorobotiikassa jokainen muuttuja tulee alustaa erikseen, jotta niitä voidaan käyttää datan tallentamiseen ja myöhemmin siirtämiseen toiseen paikkaan. Muuttujat normaalisti tuhoutuvat automaattisesti niiden alustaman luokan tai säiliön suorittamisen päättymisessä, mutta kuten kappaleessa 3.4.1 kerrotaan, ohjelmistorobotiikasta löytyy myös muuttujia, jotka ylittävät eri työnkulun rajat. Ohjelmistorobotiikassa voidaan käyttää staattista ja dynaamista datavirtaustestausta.

Luku 6

Päätelmät

Ohjelmistorobotiikan testauksessa voidaan käyttää samoja menetelmiä kuin ohjelmistokehityksessä. Koska ohjelmistorobotin tarkoitus on tehdä tiettyä tehtävää eikä ohjelmistorobotti ole yhtä laaja kuin kokonanainen ohjelmisto, kaikkia ohjelmistokehityksen testausmenetelmiä ei ole tarpeellista käyttää. Ohjelmistorobotit ovat myös projektista ja automatisoitavasta ohjelmasta riippuen erilaisia, joten testaus voi muuttua huomattavan paljon projektien välillä.

Ohjelmistorobotin määrityksiä ei voi korostaa liikaa. Jos ohjelmistorobottia kehitetään ulkopuoliselle asiakkaalle tai yrityksen sisäiseen käyttöön ohjelmistolle, jota kehittäjä ei ole käyttänyt aiemmin, ohjelmistorobotin määritys on oltava tarkkuudeltaan sataprosenttinen. Jokainen toimenpide, joka ohjelmistorobotin on tehtävä, on oltava kirjoitettuna määrityksiin niin selvästi ja yksinkertaisesti, ettei väärinkäsityksiä voi tapahtua. Määrityksien on oltava absoluuttisia.

Testausympäristö on helpottaa ohjelmistorobotin testausta huomattavasti. Testausympäristöllä tässä tilanteessa tarkoitetaan tuotantoympäristön kopiota, jossa on samat rajoitukset ja parhaassa tapauksessa jopa oikeaa dataa käytettävissä. Tällöin ohjelmistorobotin testejä saadaan suoritettua ympäristössä, joka vastaa täydellisesti tuotantoympäristöä ja ei voi tulla tilannetta, että ohjelmistorobotti toimii testauksessa, mutta ei tuotannossa.

Ohjelmistorobotteja täytyy testata jokaisella testauksen tasolla, mutta koska yksikkö-

ja integraatiotestaus tapahtuu kehitystyön aikana, tässä diplomityössä keskitytään järjestelmätestaukseen. Järjestelmätestauksen eri testausnäkökulmat esitellään kappaleessa 4.2.3 ja ohjelmistorobotiikan osalta niitä käsitellään kappaleessa 5.2.1. Nämä näkökulmat ovat testauksen kohteet, mitä testataan ja black box- ja white box -menetelmät ovat ne, millä näitä kohteita testataan. Käytännössä koko testaus tapahtuu luomalla oikeanlaisia esimerkkitapauksia, jotka kohdistetaan järjestelmätestauksen näkökulmiin ja käytetään tähän black box- ja white box -menetelmiä.

Ajattelumalli ohjelmistorobotin ja ohjelmistokehityksen testauksessa on sama: ohjelma pitää rikkoa. Ei ole koskaan hyvä, että kehittäjä itse testaa omaa ohjelmaansa. Mitä enemmän silmäpareja työtä katsoo, sitä paremmin virheet löytyvät ja tulee parempi lopputulos. Ohjelmistorobotin tarkastelutilaisuudet ovat helppo ja nopea tapa käydä ohjelma läpi, koska aikaa kuuluu vain muutama tunti ja hyödyt ovat suuret.

Ohjelmistorobotille toimintavarmuus on tärkeämpää kuin suorituksen nopeus. Jos ohjelmistorobotin suorittamisessa tehdään ylimääräisiä tarkistuksia ja varmistetaan, että syötetty data on varmasti mennyt oikein oikeaan tekstikenttään, suorittamiseen ei mene paljoa ylimääräistä aikaa. Jos ohjelmistorobotti kiirehtii legacy-ohjelmiston edelle eikä legacy-ohjelmisto pysy mukana, syötetty data voi olla virheellistä ja pahimmassa tapauksessa suorittamisesta koituu suuriakin ongelmia, jos ohjelmistorobotti on otettu käyttöön jo tuotantoympäristössä. Tästä syystä ylimääräisten tarkistuspisteiden ja poikkeuskäsittelyn lisääminen ohjelmistorobotin toimenpiteisiin on hyvä tapa lisätä toimintavarmuutta. Ohjelmistorobotti joutuu kuitenkin toimimaan legacy-ohjelmiston suoritusnopeuden mukaan ja todennäköisesti on silti ihmistä nopeampi suorituksessaan.

Tämän diplomityön tuloksena löydettiin ohjelmistorobotin testauksen ongelmakohdat ja erilaisia, hyödyllisiä testausmenetelmiä robotin kattavaan testaukseen. Ongelmakohdat on tärkeä tiedostaa missä vain ohjelmistorobotin kehitystyössä ja niiden ratkaisemiseen kannattaa panostaa, sillä se helpottaa ja parantaa testausta. Ohjelmistokehityksen testauksen vertailulla löydetyt menetelmät testaukseen on hyvä ottaa käyttöön, mutta

koska jokainen ohjelmistorobottiprojekti on erilainen, ei kaikkia ole tarpeellista jokaisessa tilanteessa käyttää. Tässä työssä käytettiin pohjana vain UiPathia ja lyhyesti esiteltiin BluePrism ja Automation Anywhere, joten löydetyt tulokset eivät välttämättä täsmää suoraan muihin ohjelmistorobotiikkaohjelmistoihin. Haastetteluissa oli vain neljän eri yrityksen edustajia, joten myös muita työskentelymenetelmiä ohjelmistorobotiikan kehitykseen ja testaukseen mahdollisesti löytyy.

Lähdeluettelo

[1] Alok Mani Tripahti. *Learning Robotic Process Automation: Create Software robots and automate business processes with the leading RPA tool - UiPath*. Packt Publishing Ltd., 2018.

[2] IBM Cloud. Robotic Process Automation (RPA) bots in action, 2018. URL www.youtube.com/watch?v=loOR-nz9DGY, Viitattu 9.12.2018.

[3] myTectra. Top 10 most popular RPA Tools of 2019, 2019. URL www.mytectra.com/blog/Top-10-most-popular-RPA-Tools-of-2019/, Viitattu 23.10.2019.

[4] UiPath. UiPath website, 2018. URL www.uipath.com/, Viitattu 8.1.2019.

[5] UiPath Studio Guide. Recording Types, 2019. URL studio.uipath.com/docs/about-recording-types, Viitattu 23.4.2019.

[6] UiPath Studio Guide. Full versus Partial Selectors, 2019. URL studio.uipath.com/docs/full-versus-partial-selectors, Viitattu 17.5.2019.

[7] Software Testing Help. 10 Most Popular Robotic Process Automation RPA Tools In 2019, 2019. URL www.softwaretestinghelp.com/robotic-process-automation-tools, Viitattu 29.7.2019.

[8] RaiMan. SikuliX by RaiMan, 2019. URL sikulix.com, Viitattu 29.7.2019.

[9] David Chappell. Introducing Blue Prism : Automating Business Processes with Presentation Integration, 2010. URL www.davidchappell.com/writing/white_papers/Introducing_Blue_Prism_v1.0-Chappell.pdf, Viitattu 29.7.2019.

[10] Busy Ping. Blue Prism Video Tutorial — 011 — Calc, Data item and multi calc stages, 2017. URL www.youtube.com/watch?v=690vk1P3m8g, Viitattu 30.7.2019.

[11] Asha24. Automation Anywhere Tutorial - 02 Hello World, 2018. URL www.youtube.com/watch?v=g0IzTbWdocQ, Viitattu 30.7.2019.

[12] DrapsTV. SikuliX Tutorial #2 - The Basics, 2015. URL www.youtube.com/watch?v=I-UYoezac4Q, Viitattu 31.7.2019.

[13] Glenford J. Myers, Corey Sandler ja Tom Badgett. *The Art of Software Testing*. John Wiley & Sons, 2012.

[14] Bernard Homès. *Fundamentals of Software Testing*. John Wiley & Sons, 2012.

[15] Kshirasagar Naik ja Priyadarshi Tripathy. *Software Testing and Quality Assurance: Theory and Practice*. John Wiley & Sons, 2008.

[16] Lee Copeland. *A Practitioner's Guide to Software Test Design*. Artech House, 2004.

[17] ProfessionalQA.com. Data Flow Testing, 2019. URL www.professionalqa.com/data-flow-testing, Viitattu 6.7.2019.

[18] Martin Fowler. The Practical Test Pyramid, 2019. URL martinfowler.com/articles/practical-test- pyramid.html, Viitattu 15.7.2019.

[19] Jeffrey Rubin, Dana Chisnell ja Jared Spool. *Handbook of usability testing : How to Plan, Design, and Conduct Effective Tests*. John Wiley & Sons, 2008.

[20] Ranorex. GUI Testing : The Beginner's Guide for User Interface (UI) Testing, 2019. URL www.ranorex.com/resources/testing-wiki/gui-testing/, Viitattu 24.7.2019.

[21] ReQtest. Test Case Design Techniques to Ensure High-Quality Software, 2019. URL reqtest.com/testing-blog/test-case-design-techniques/, Viitattu 24.7.2019.

[22] Software Testing Fundamentals. Test Case, 2019. URL softwaretesting fundamentals.com/test-case/, Viitattu 28.7.2019.

[23] Guru99. How to Write Test Cases: Sample Template with Examples, 2019. URL www.guru99.com/test-case.html, Viitattu 28.7.2019.

[24] Techopedia. Development Environment, 2019. URL www.techopedia.com/definition/16376/development-environment, Viitattu 7.8.2019.

[25] TechTarget. Legacy application, 2019. URL searchitoperations.techtarget.com/definition/legacy-application, Viitattu 7.8.2019.

[26] ebureka! 5 UiPath Automation Examples That You Can Practice, 2019. URL www.edureka.co/blog/uipath-automation-examples, Viitattu 15.8.2019.