Saska Sinkkonen

Alustavia havaintoja Unity-spesifistä mutaatiotestauksesta

Tietotekniikan Pro gradu -tutkielma

5. toukokuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Saska Sinkkonen

Yhteystiedot: saansink@student.jyu.fi

Ohjaaja: Antti Valmari

Työn nimi: Alustavia havaintoja Unity-spesifistä mutaatiotestauksesta

Title in English: Preliminary findings on Unity-spesific mutation testing

Työ: Pro gradu -tutkielma

Opintosuunta: Ohjelmisto- ja tietoliikennetekniikka

Sivumäärä: 69+0

Tiivistelmä: Mutaatiotestaus on keino parantaa ohjelmistotestauksessa käytettyjä testijoukkoja hyödyntäen ohjelmistoon automaattisesti generoituja virheitä. Sitä pidetään parhaimpana – mutta myös resurssivaatimuksiltaan raskaimpana – tapana tavoitella vahvinta mahdollista testijoukkoa. Mutaatiotestauksesta pelikehityksen kontekstissa on hyvin vähän tutkimusta ja tämän tutkielman tavoitteena on antaa alustavia tuloksia Unity-pelimoottorille spesifisti suunnitellun mutaatiotestauksen tehokkuudesta.

Tutkielmassa suunniteltiin Unityn ohjelmointirakenteisiin pohjautuva joukko mutaatio-operaattoreita ja kehitettiin työkalu, joka mahdollisti mutaatiotestauksen niitä käyttäen. Työkalulla saatuja tuloksia verrattiin perinteiseen mutaatiotyökaluun. Unity-spesifi mutaatiotestaus todettiin huomattavasti nopeammaksi ja vähemmän manuaalista työtä vaativaksi generoitujen mutanttien vähäisemmän määrän vuoksi. Toisaalta perinteisellä mutaatiotestauksella tuotettu testijoukko vaikutti kokeen tulosten perusteella kattavammalta. Kumpikaan testijoukko ei yksinään ollut kuitenkaan hyvin kattava. Siksi tärkeimpänä tuloksena pidettiin, että Unity-pelien tehokasta testausta varten Unity-spesifiä ja perinteistä mutaatiotestausta on käytettävä yhdessä. Tutkimuksen empiirinen osa toteutettiin kuitenkin hyvin suppealla aineistolla ja siksi esiteltiin myös huomattava määrä jatkotutkimusaiheita, joihin pitäisi perehtyä ennen tarkempien päätelmien tekemistä.

Avainsanat: Mutaatiotestaus, pelitestaus, Unity

Abstract: Mutation testing is a technique used to improve software test sets by automatically generating errors into a program's source code. It is considered the best – but also most resource-intensive – way to achieve the strongest possible test set. There is very little research done on mutation testing in the context of game development and the aim of this thesis is to provide preliminary results on the effectiveness of mutation testing specifically designed for the Unity game engine.

A set of mutation operators based on Unity programming structures was designed and a tool which enabled mutation testing using them was developed. The results obtained with the tool were compared with ones of a traditional mutation tool. Unity-specific mutation was found to be significantly faster and needing less manual work due to the lower number of mutants generated. On the other hand, based on the results of the experiment traditional mutation seemed to produce a stronger test set. However, neither test set alone was very comprehensive. Therefore the main conclusion was that for effective testing of Unity-games, Unity-spesific mutation should be used in conjunction with traditional mutation. However, the empirical part of the study was conducted with a very limited data set and because of that multiple topics for further research were presented. These topics should be explored before drawing more detailed conclusions.

Keywords: Mutation testing, game testing, Unity

Termiluettelo

CREAM	C#-ohjelmia varten kehitetty mutaatiotyökalu, joka sisältää sekä perinteisiä että oliopohjaisia mutaatio-operaattoreita.
Ekvivalentti mutantti	Mutantti, jonka toiminta ei eroa mitenkään alkuperäisestä ohjelmasta.
Komponentti	Unity-pelissä olevan peliobjektin osa.
Mutantin tappaminen	Testijoukon ajosta saatu tulos eroaa mutantilla ja alkuperäisellä ohjelmalla.
Mutantti	Ohjelma, jonka koodiin on generoitu muutos tai muutoksia. Muutokset johtavat usein virheisiin ohjelman toiminnassa.
Mutaatio-operaattori	Sääntö, jonka perusteella ohjelmakoodiin generoidaan muutoksia.
Mutaatiopisteytys	Tapettujen mutanttien osuus kaikista generoiduista mutanteista, jotka eivät ole ekvivalentteja.
Mutaatiotestaus	Prosessi, jossa mutantteja hyödynnetään ohjelmistotestauksessa käytettävien testijoukkojen arviointiin ja parantamiseen.
Pelimoottori	Ohjelmistokehys, jonka pohjalta on mahdollistaa kehittää eri tyyppisiä pelejä.
Peliobjekti	Kappale, joista Unity-skenet koostuvat.
Skripti	Unity-komponentti, joka on peliobjektin käyttäytymisen määrittelyyn käytettävä C#-luokka.
Skene	Unity-pelin pelikenttä, peliobjekteja sisältävä 3D-avaruus.
Unity	Unity Technologiesin kehittämä pelimoottori.
Unity Mutator	Tätä tutkielmaa varten kehitetty työkalu, joka mahdollistaa Unity-spesifin mutanttien generoinnin ja testauksen.
Virheenlöytökyky	Testijoukon ominaisuus, joka kuvaa kuinka kattavasti se löytää ohjelmistossa olevia ohjelmointivirheitä.
Ylimääräinen mutantti	Mutantti, joka tulee aina tapetuksi jonkin tietyn toisen mutantin yhteydessä.

Kuviot

Kuvio 1. Moderni mutaatiotestausprosessi 12
Kuvio 2. Pelimoottorien uudelleenkäytettävyyden kirjo 17
Kuvio 3. Unity-editorin oletusnäkymä 19
Kuvio 4. CREAM:lla generoitujen mutanttien jakauma mutaatio-operaattoreittain 42
Kuvio 5. Unity Mutatorilla generoitujen mutanttien jakauma mutaatio-operaattoreittain .. 44

Taulukot

Taulukko 1. Viisi perinteistä mutaatio-operaattoria 5
Taulukko 2. Esimerkki mutaatiomatriisista 9
Taulukko 3. Unity-spesifit mutaatio-operaattorit 26
Taulukko 4. Lopullisesta versiosta poistetut mutaatio-operaattorit 26
Taulukko 5. Testijoukko 1 40
Taulukko 6. Testijoukkoon 2 lisätyt testitapaukset 45
Taulukko 7. Testijoukkoon 3 lisätyt testitapaukset 46
Taulukko 8. Testijoukkojen tulosvertailu 47

Sisältö

1 JOHDANTO 1

2 MUTAATIOTESTAUS 4
2.1 Mitä on mutaatiotestaus? 4
2.2 Mutaatiotestauksen ongelmat 7
2.3 Mutaatiotestauksen vaiheet 12

3 UNITY-SPESIFI MUTAATIOTESTAUS 16
3.1 Unity 16
3.2 Unityn tärkeimmät luokat 20
3.3 Mutaatio-operaattoreiden valinta 23
3.4 Unity Mutator 27

4 TUTKIMUSASETELMA 30
4.1 Peliaineisto 30
4.2 Testaustyökalut 31
4.2.1 CREAM 32
4.2.2 Unity Test Framework 33
4.3 Mutaatiotekniikoiden vertailuperusteet 34
4.4 Testijoukkojen vaatimukset 36
4.4.1 Rivikattavuus 36
4.4.2 Mutaatiopisteytys 38

5 MUTAATIOTESTAUSPROSESSIN SUORITUS 39
5.1 Yleistä 39
5.2 Testijoukko 1 40
5.3 Mutanttien generointi 41
5.4 Testijoukko 2 44
5.5 Testijoukko 3 45

6 TULOKSET JA POHDINTA 47
6.1 Mutaatiotekniikoiden vertailu 47
6.2 Tutkimuksen puutteet ja jatkotutkimus 51

7 YHTEENVETO 54

LÄHTEET 56

1 Johdanto

Ohjelmistot ovat nykypäivänä kaikkialla. Koko maailma pyörii erinäisten ohjelmistojen päällä ja jos ne eivät toimi virheettömästi, voivat seuraukset olla katastrofaaliset. Esimerkki tällaisesta tapauksesta on sädehoitolaite Therac-25, joka vuosina 1985–1987 aiheutti kuusi vakaviin vammoihin tai kuolemaan johtanutta onnettomuutta. Onnettomuudet johtuivat huomaamatta jääneistä virheistä laitteen ohjelmakoodissa (Leveson ja Turner 1993).

Useimmiten ohjelmiston virheet kuitenkin johtavat onneksi vain taloudellisiin seurauksiin, mutta nekin voivat olla hyvin merkittäviä. Esimerkkejä useista historian saatossa sattuneista suuria menetyksiä aiheuttaneista ohjelmistovirheistä on esitelty muun muassa kirjassa *Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science* (Huckle ja Neckel 2019). Yhteistä kaikilla näillä on se, että virheet ovat jääneet kokonaan huomaamatta ohjelmistojen kehitysprosessin aikana ja tulleet esille vasta käyttöönoton jälkeen.

Vaikka virheiden aiheuttamia todellisia kustannuksia onkin vaikea tarkkaan arvioida, ovat ne huomattavasti pienemmät, mitä aikaisemmassa vaiheessa kehitysprosessia virhe löydetään. Ammann ja Offutt (2017) esittävät kuvaajan, jonka mukaan integraatiotestauksessa löydetty virhe tulee viisi kertaa niin kalliiksi korjata kuin yksikkötestauksessa löydetty. Järjestelmätestausvaiheessa löydetyn virheen hinta on kymmenkertainen ja käyttöönoton jälkeen jopa 50-kertainen verrattuna yksikkötestausvaiheeseen.

Ohjelmistotestaus on siis välttämätön osa ohjelmistokehitysprosessia, mutta täydellinen testaus on kuitenkin mahdotonta. ”Ohjelmistotestauksella voidaan osoittaa virheiden olemassaolo, mutta ei koskaan niiden poissaoloa”, kuuluu Edsger Dijkstran klassinen sanonta. Mahdollisten käyttäjän antamien erilaisten syötteiden määrä on niin valtava, että kaikkia ei voida mitenkään testata, ellei ohjelma ole täysin triviaali. Siksi voimmekin vain yrittää päästä mahdollisimman lähelle testausta, joka todistaisi ohjelman olevan virheetön.

Mutaatiotestaus on tasaisesti suosiota kasvattava aihe sekä tietotekniikan alan tutkimuksessa että teollisuudessa (Papadakis ym. 2019). Mutaatiotestauksessa ohjelmakoodiin generoidaan muutoksia, jotka usein johtavat virheisiin ohjelman toiminnassa. Tämän jälkeen muokatul-

le ohjelmalle ajetaan ohjelmistotestit. Näin saadaan selville jäävätkö keinotekoiset virheet kiinni testeillä ja voidaan tehdä päätelmiä testauksen kattavuudesta. Sitä pidetään tällä hetkellä yleisesti parhaimpana tapana arvioida ja kehittää testijoukkojen tehokkuutta (Petrovic ym. 2021b) ja näin tavoitella täydellistä testausta. Se on kuitenkin kallista ja aikaa vievää toteuttaa, mikä onkin ollut merkittävä este mutaatiotestauksen laajemmalle käytölle alalla (Papadakis ym. 2019). Teknologian ja menetelmien kehittyessä mahdollisuudet mutaatiotestaukseen kuitenkin paranevat koko ajan.

Etenkin pelialaan kohdistuva mutaatiotestauksen tutkimus on edelleen hyvin vähäistä eikä erityisesti siihen suunniteltuja työkaluja löydy lainkaan. Niinpä tämän tutkielman tavoitteena onkin selvittää alustavia havaintoja liittyen mutaatiotestaukseen pelikehityksen ja vielä tarkemmin Unity-pelimoottorilla toteutettujen pelien kontekstissa.

Ongelmat joihin tutkielmassa etsitään vastauksia, on muotoiltu seuraaviksi kahdeksi tutkimuskysymykseksi:

1. Miten Unity-spesifiä mutaatiotestausta olisi mahdollista toteuttaa?
2. Olisiko Unity-spesifillä mutaatiotestauksella potentiaalia saavuttaa hyötyjä verrattuna perinteiseen mutaatiotestaukseen?

Ensimmäistä kysymystä varten tutkielmassa perehdytään Unity-peleissä usein käytettyihin pelimoottorille spesifeihin ominaisuuksiin. Niiden perusteella suunnitellaan tapa mutatoida Unity-peleille tyypillisiä ohjelmointirakenteita ja luodaan työkalu, joka mahdollistaa koko mutaatiotestausprosessin suorittamisen. Toiseen kysymykseen etsitään vastausta tutkielman kokeellisessa osassa vertailemalla erinäisillä kriteereillä tällä uudella työkalulla saavutettuja tuloksia perinteiseen mutaatiotestaustyökaluun.

Tutkielman toisessa luvussa perehdytään mutaatiotestaukseen yleisesti sekä sen hyviin ja huonoihin puoliin. Lisäksi käydään lävitse tarpeellisia termejä sekä perinteinen mutaatiotestausprosessi, jota myös tutkielman kokeellisessa osassa suurimmilta osin seurataan. Kolmannessa luvussa esitellään ja tarkastellaan Unity-pelimoottorin ominaisuuksia. Tämän jälkeen esitetään niiden pohjalta päätetty joukko Unity-spesifejä mutaatio-operaattoreita sekä toteutetaan työkalu, johon implementoidaan nämä operaattorit ja jonka avulla mutaatiotestaus käytännössä suoritetaan. Luvussa neljä esitellään tutkimusasetelma kokeellista osiota varten.

Siihen kuuluvat aineistona käytetty peli, tärkeimmät prosessin apuna käytetyt työkalut sekä arviointikriteerit joiden perusteella eri mutaatiotekniikoita verrataan toisiinsa. Viidennessä luvussa suoritetaan mutaatiotestausprosessi käyttäen sekä perinteistä että Unity-spesifiä mutaatiota. Kuudennessa luvussa esitellään eri tekniikoilla saadut tulokset ja pohditaan niiden merkitystä. Sen lisäksi huomioidaan tutkimuksen puutteet ja esitetään tutkielman perusteella esille nousseita jatkotutkimusaiheita. Viimeisessä luvussa kerrataan vielä tiiviissä muodossa koko tutkimuksen kulku ja merkittävimmät tulokset.

2 Mutaatiotestaus

Tässä luvussa esitellään mutaatiotestausta yleisesti ilman pelikehityksen kontekstia. Tärkeimmät mutaatiotestauksen termit määritellään ja lisäksi kerrotaan sen toimintaperiaatteista, historiasta ja käyttökohteista. Toisessa alaluvussa käydään lävitse mutaatiotestauksen ongelmia, jotka ovat hidastaneet sen käyttöönottoa alan teollisuudessa ja kolmannessa kuvataan käytännön mutaatiotestausprosessi jaettuna kymmeneen vaiheeseen.

2.1 Mitä on mutaatiotestaus?

Koska ohjelmistojen täydellinen testaaminen on mahdotonta, yksi ohjelmistotestauksen suurimmista ongelmista on tietää, milloin ohjelmistoa on testattu tarpeeksi (Papadakis ym. 2019). Testauksen riittävyyttä voidaan mitata erilaisilla tekniikoilla, joista koodikattavuus lienee yhä käytetyin. Koodikattavuutta on käytetty jo vuosikymmenten ajan, mutta sen korrelaation vahvuudesta testijoukon tehokkuuden kanssa on yhä eriäviä tutkimustuloksia. Muun muassa Inozemtseva ja Holmes (2014) tutkivat laajaa aineistoa aiheeseen liittyen ja totesivat korrelaation olevan vain matala tai kohtalainen.

Koodikattavuutta parempana – ja yleisesti ottaen parhaimpana tapana tavoitella täydellistä ohjelmistotestausta – pidetäänkin usein mutaatiotestausta. Oikein hyödynnettynä mutaatiotestauksen avulla yleensä kokonaan tai lähes kokonaan täytetään kaikki muut kattavuuskriteerit (Ammann ja Offutt 2017). Testauksen kattavuuden arvioimisen lisäksi mutaatiotestauksella saadaan konkreettista tietoa siitä, miten testijoukkoa voisi parantaa.

Mutaatioanalyysillä tarkoitetaan prosessia, jossa ohjelman lähdekoodia muokataan automaatiota apuna käyttäen ja näin ohjelmasta pyritään luomaan eri tavalla käyttäytyviä versioita – *mutantteja*. *Mutaatiotestauksella* taas tarkoitetaan mutaatioanalyysin hyödyntämistä ohjelmistotestauksessa käytettävien testijoukkojen arviointiin ja parantamiseen (Papadakis ym. 2019). Mikäli generoidun mutantin toiminta eroaa alkuperäisestä ohjelmasta, täytyy myös testijoukkojen tulosten erota niiden välillä. Jos testitulos eroaa, sanotaan, että mutantti on tapettu (engl. *killed*) ja testijoukko voidaan todeta tältä osin päteväksi. Jos taas testauksen tulos ei eroa alkuperäisen ja mutantin välillä – eli mutantti jää eloon (engl. *live*) – täytyy

Nimi	**Kuvaus**	**Esimerkki**
ABS (absolute value insertion)	Matemaattisten lausekkeiden arvojen etumerkkiä vaihdetaan tai arvo muutetaan nollaksi	`-x*y` **-->** `x*y`
AOR (arithmetic operator replacement)	Aritmeettinen operaattori korvataan toisella	`x*y` **-->** `x/y`
LCR (logical connector replacement)	Looginen operaattori korvataan toisella	`x && y` **-->** `x || y`
ROR (relational operator replacement)	Vertailuoperaattori korvataan toisella	`x > y` **-->** `x < y`
UOI (unary operator insertion)	Arvon eteen lisätään unaarioperaattori	`x` **-->** `++x`

Taulukko 1. Viisi mutaatio-operaattoria, jotka mahdollistavat tehokkaan mutaatiotestauksen (Offutt ym. 1996).

testaajan analysoida, johtuuko se testijoukon puutteista vai muista syistä. Jos syynä on puutteellinen testijoukko, on tavoitteena parantaa sitä niin, että mutantti saadaan tapettua. Mutaatiotestauksen avulla olemassa olevasta testijoukosta siis voidaan löytää puutteita ja näin kehittää sitä. Mutaatiotestausta voidaankin ajatella ohjelmiston testauksen sijaan testauksen testauksena.

Mutatointiprosessia varten määritellään *mutaatio-operaattorit*. Ne ovat syntaktisia sääntöjä, joiden mukaan ohjelmakoodiin generoidut muutokset tehdään (Papadakis ym. 2019). Taulukossa 1 on esimerkin vuoksi esitetty joitakin tärkeimmistä perinteisesti käytetyistä mutaatio-operaattoreista. Offutt ym. (1996) tutkivat, että jo näiden viiden operaattorin avulla on mahdollista tehokkaasti toteuttaa mutaatiotestausta.

Mutaatiotestauksen tyyppejä voidaan luokitella monilla tavoilla, esimerkiksi mutantin tappamisen määritelmän tai generoitujen mutanttien rakenteen mukaan. Yksi tällainen olennainen luokittelukriteeri on mutaatiotestauksen vahvuus. Mutaatiotestaus voi olla joko heikkoa tai

vahvaa ja ne eroavat siinä, mitä mutantin tappamiselta vaaditaan. Heikossa mutatoinnissa tappamiseen riittää, että mutatoidun ohjelman tila eroaa alkuperäisestä välittömästi mutatoidun kohdan suorituksen jälkeen. Vahvassa mutatoinnissa taas vaaditaan, että alkuperäisen ohjelman ja mutatoidun version tulosteessa on jokin havaittava ero (Papadakis ym. 2019). Heikko mutaatiotestaus tavallisesti tuottaa virheenlöytökyvyltään vahvaa mutaatiotestausta huonompia testijoukkoja (Chekam ym. 2017) ja sitä voikin ehkä pitää lähempänä tyypillisiä koodikattavuuskriteereitä. Tässä tutkielmassa mutaatiotestauksesta puhuttaessa käsitellään oletuksena vahvaa mutatointia ja myös kokeellisessa osassa toteutetussa prosessissa käytetään vahvaa mutatointia.

Mutaatiotestauksen juuret pohjautuvat 1970-luvun tutkimukseen, josta etenkin *Hints on test data selection: help for the practicing programmer* (DeMillo, Lipton ja Sayward 1978) mainitaan usein keskeisenä julkaisuna. 1980-luvulta lähtien kiinnostus mutaatiotestaukseen on kasvanut ja myös siihen liittyvien tieteellisten artikkelien määrä on noussut tasaisesti (Jia ja Harman 2011; Papadakis ym. 2019). Siltikin kymmenien vuosien tutkimuksesta ja todetusta tehokkuudesta huolimatta se ei yhäkään ole teollisesti laajassa käytössä alalla (Petrovic ym. 2018). Syitä tähän on monia, sillä mutaatiotestauksella on myös selkeät ongelmansa. Niihin perehdytään tarkemmin luvussa 2.2.

DeMillo, Lipton ja Sayward (1978) esittivät mutaatiotestauksen perustaksi kaksi olettamusta: osaavan ohjelmoijan ja yhdistelmävaikutuksen (engl. *coupling effect*). Osaava ohjelmoija kirjoittaa ohjelmakoodia, joka on lähes oikein – pieniä virheitä lukuun ottamatta. Yhdistelmävaikutus taas on testijoukon ominaisuus, jonka mukaan testijoukko, joka löytää yksinkertaiset virheet, löytää todennäköisesti myös monimutkaiset virheet. Näiden olettamusten voimassa ollessa voidaan todeta mutaatiotestaus tehokkaaksi ja generoitujen mutanttien löytämisen korreloivan todellisten virheiden löytämisen kanssa.

DeMillo, Lipton ja Sayward (1978) toteavat artikkelissaan, ettei ole keinoa todistaa yhdistelmävaikutuksen olemassaoloa ja myöntävät sen olevan vain empiirisiin havaintoihin perustuva periaate. Myöhemmissä tutkimuksissa on kuitenkin onnistuttu osoittamaan, että yksinkertaiset mutantit tappava testijoukko tappaa tehokkaasti myös monimutkaiset mutantit (Offutt 1992). Lisäksi Just ym. (2014) osoittivat, että yhdistelmävaikutus pätee myös suurelta osin yksinkertaisten mutanttien ja tosielämän ohjelmointivirheiden välillä. Mutaatiotestauk-

sen avulla siis voidaan kehittää testejä löytämään myös todellisia virheitä, vaikka ne eroaisivat mutantteihin generoiduista virheistä. Geist, Offutt ja Harris (1992) tiivistävätkin mutaatiotestauksen toiminnan periaatteen: ”jos ohjelmisto sisältää virheen, on luultavasti olemassa mutantti, joka voidaan tappaa vain testillä, joka löytää myös kyseisen virheen.”

Testijoukon kehityskohteiden löytämisen lisäksi mutaatioanalyysillä on muitakin käyttötarkoituksia. Tutkimuksissa sitä käytetään usein arviointitarkoituksessa, vaikkei tutkimus itsessään varsinaisesti koskisikaan mutaatioanalyysiä tai -testausta (Papadakis ym. 2019). Erilaisia testaustekniikoita voidaan vertailla tai arvioida *mutaatiopisteytyksen* avulla, joka kuvaa testijoukolla tapettujen mutanttien osuutta kaikista mutanteista. Mutaatioanalyysin avulla voidaan myös täysin automaattisesti toteuttaa testitapauksia (Offutt 1988) tai parantaa valmiiksi olemassa olevia testejä (Baudry ym. 2002).

Lisäksi mutaatioanalyysiä voidaan käyttää testitapausten priorisointiin. Etenkin regressiotestauksessa ajetaan usein suuri määrä testejä ja saadaan hyötyjä, kun tärkeimmät – siis eniten virheitä löytävät – testitapaukset ajetaan ensimmäisenä. Mutaatioanalyysin avulla priorisointia voidaan automatisoida ja näin nopeuttaa virheiden löytämistä testiajon aikana (Do ja Rothermel 2006). Samaan tapaan sitä voidaan käyttää apuna testijoukon minimointiin (Offutt, Pan ja Voas 1995). Silloin tarkoituksena on resurssien säästämiseksi ajaa vain osa testijoukosta säilyttäen testien tehokkuus kuitenkin mahdollisimman korkeana.

2.2 Mutaatiotestauksen ongelmat

”Mutaatiotestaus on tehokas, mutta laskennallisesti raskas tekniikka ohjelmistojen yksikkötestaukseen”, kuvasivat Offutt ja Untch (2001). Kaksikymmentä vuotta myöhemmin sama ongelma mainitaan yhä Googlen mutaatiotestauksen käyttöä tarkastelleessa tutkimuksessa (Petrovic ym. 2021b).

Mutaatiotestauksen perimmäinen ongelma on generoitavien mutanttien suuri määrä. Jo yhtä koodiriviä saatetaan mutatoida usealla eri tavalla, joten suhteellisen yksinkertaisellekin ohjelmalle mutantteja generoituu helposti tuhansia. Tämä johtaa huonoon skaalautuvuuteen, minkä Papadakis ym. (2019) sanovatkin olevan avainsyy, joka estää mutaatiotestauksen laajemman käyttöönoton alalla.

Mutaatiotestauksen laskennallinen raskaus johtuu nimenomaan huonosta skaalautuvuudesta ja siksi vaikka laitteiden laskentateho kasvaakin, ei mutaatiotestauksen tehokkuus välttämättä parane, sillä myös testattavat ohjelmistot ovat aiempaa laajempia. Kaikki mutantit täytyy kääntää ajettaviksi ohjelmiksi ja testijoukko täytyy ajaa jokaisella niistä. On ymmärrettävää, että esimerkiksi Googlen ohjelmistojen tapauksessa tämä vaatii huomattavan määrän laskentatehoa sekä aikaa.

Laskennallisen raskauden lisäksi toinen osa skaalautuvuusongelmaa on vaadittu manuaalisen työn määrä. Kääntämisen ja ajamisen lisäksi mutantteja täytyy analysoida sekä tappaa testitapauksia kehittämällä ja tämä tarkoittaa yhä usein manuaalista työtä (Chekam ym. 2020). Tosin molempiin tehtäviin on myös kehitetty erinäisiä automatisoituja apukeinoja ja tulevaisuudessa manuaalisen työn määrää varmasti pystytään vähentämään.

Yksi juurisyy mutanttien suureen määrään ja näin ollen huonoon skaalautuvuuteen ovat tarpeettomat mutantit. Tarpeettomat mutantit ovat mutantteja, jotka eivät auta mitenkään testijoukon kehityksessä, vaan lopputulos olisi sama myös ilman niitä. Näin ollen ne turhaan vain vievät aikaa mutaatiotestausprosessissa.

Suurin ongelma ovat *ekvivalentit* (engl. *equivalent*) ja *ylimääräiset* (engl. *redundant*) mutantit. Ekvivalentit mutantit ovat mutantteja, jotka käyttäytyvät identtisesti alkuperäisen ohjelman kanssa ja näin ollen niiden tappaminen on mahdotonta (Papadakis ym. 2016). Ylimääräiset mutantit taas ovat mutantteja, jotka tulevat tapetuksi aina jonkin toisen tietyn mutantin yhteydessä.

Useimmiten ylimääräiset mutantit ovat niin sanotusti *oheiskuolevia* (engl. *subsumed*) mutantteja. Mutantti A määritellään oheiskuolevaksi, jos on olemassa toinen mutantti B, jonka tappaa mutantti A:n tappavien testitapausten osajoukko (Papadakis ym. 2016). Oletuksena tässä määritelmässä on, että ainakin yksi testitapaus tappaa mutantin B. Siis oheiskuoleva mutantti tapetaan myös aina toisen mutantin yhteydessä. Toisin kuin ekvivalentin mutantin tapauksessa, tämä ominaisuus on riippuvainen myös muista mutanteista sekä käytetystä testijoukosta.

Käytännön esimerkin vuoksi taulukossa 2 on kuvattuna mutaatiomatriisi, johon on merkitty jokaisen testitapauksen tappamat mutantit. Esimerkkitapauksessa mutantti M2 on ylimääräi-

	M1	**M2**	**M3**
T1	x	x	
T2		x	
T3			
T4			

Taulukko 2. Esimerkki mutaatiomatriisista

nen, sillä se kuolee aina, kun mutantti M1 kuolee. Mutanttia M3 taas ei tapeta ollenkaan, mikä tarkoittaa, että joko testijoukossa on puutteita tai kyseinen mutantti on ekvivalentti.

Toinen alaluokka ylimääräisistä mutanteista ovat kaksoismutantit (engl. *duplicate*) eli mutantit, jotka ovat ekvivalentteja toisen mutantin kanssa, mutta eivät alkuperäisen ohjelman (Papadakis ym. 2016). Tällöin ominaisuus ei riipu käytetystä testijoukosta. Ei myöskään ole mahdollista sanoa, kumpi mutanteista on varsinaisesti ylimääräinen, vaan sen voi ajatella olevan kumpi tahansa, mutta ei kuitenkaan molemmat.

Mutaatioprosessin tehokkuuden kannalta olisi tärkeää generoida mahdollisimman vähän tarpeettomia mutantteja tai poistaa ne jo ennen testijoukon ajamista. Tarpeettomien mutanttien tunnistus on kuitenkin ratkeamaton ongelma (Papadakis ym. 2019; Budd ja Angluin 1982). Se tarkoittaa, että tunnistusta ei voida kokonaan automatisoida, vaan se vaatii testaajalta manuaalista työtä. Erityisesti ekvivalentit mutantit täytyy analysoida, sillä ne näyttäytyvät testauksessa eloonjääneinä mutantteina.

Schuler ja Zeller (2013) analysoivat tutkimuksessaan Java-ohjelmiin generoituja eloonjääneitä mutantteja ja heillä yhden mutantin analysointi vei keskimäärin jopa 15 minuuttia ja 45 % niistä todettiin lopulta ekvivalenteiksi. Prosenttiosuus tietysti kasvaa testijoukon laadun mukaisesti, sillä parempi testijoukko tappaa suuremman osan epäekvivalenteista mutanteista. Analyysiin kuluva manuaalisen työn määrä on siis huomattava, kun otetaan huomioon, kuinka paljon mutantteja usein generoidaan.

Tarkan mutaatiopisteytyksen laskeminen vaatii kuitenkin kaikkien ekvivalenttien mutanttien tunnistamisen, sillä se määritellään tapettujen mutanttien osuudeksi kaikista tapettavissa olevista mutanteista (Offutt ja Untch 2001). Joskus mutaatiopisteytyksessä saatetaan laskea

joukkoon myös ekvivalentit mutantit, mutta tällöin se antaa vääristyneen kuvan todellisuudesta. Testaajan on siis joko käytettävä huomattava määrä aikaa eloon jääneiden mutanttien analysointiin tai tarkan mutaatiopisteytyksen sijaan siitä on tehtävä vain arvio. Vaikka jälkimmäinen vaihtoehto on nopeampi, laskee se myös testauksen luotettavuutta (Papadakis, Delamaro ja Traon 2014). Testauksen lopetuskriteeriksi ei voida myöskään asettaa 100 %:n mutaatiopisteytystä, sillä se voi olla mahdotonta saavuttaa ekvivalenttien mutanttien vuoksi. Näin ollen kun valitaan pienempi vaatimus mutaatiopisteytykselle, voi myös todellisia tapettavissa olevia mutantteja jäädä huomioimatta.

Tarpeettomien mutanttien automaattinen tunnistus onkin ajankohtainen tutkimuskohde ja siihen on jo kehitetty erinäisiä apukeinoja, joiden avulla manuaalista työtä voidaan vähentää. Näistä mainittakoon esimerkkinä suhteellisen helposti implementoitava *Trivial Compiler Equivalence* eli TCE, joka kattavassa empiirisessä tutkimuksessa tunnisti jopa 30 % kaikista ekvivalenteista mutanteista (Papadakis ym. 2015). TCE:ssä ideana on käyttää hyväksi ohjelmointikielien kääntäjien tekemää optimointia. Vaikka alkuperäisen ohjelman ja mutantin lähdekoodi olisivat erilaisia, voivat ne kääntäjän suorittaman optimoinnin jälkeen olla konekieleltään identtisiä. TCE:ssä tällaiset mutantit tunnistetaan ekvivalenteiksi.

Myös ylimääräisten mutanttien tunnistukseen on esitetty kirjallisuudessa eri tekniikoita. Sekin on ymmärrettävästi hankalaa etenkin ennen testijoukon ajamista mutanteilla, sillä kuten aiemmin mainittiin mutantin ylimääräisyys riippuu myös ajetuista testitapauksista. Esimerkiksi kaksoismutanttien tunnistukseen voidaan soveltaa kuitenkin samaa TCE-tekniikkaa kuin ekvivalentteihin mutantteihin. Ylimääräisten mutanttien määrää voidaan myös melko tehokkaasti vähentää jättämällä pois tiettyjä mutaatio-operaattoreita. Tällöin on kuitenkin vaara, että myös todellisuudessa hyödyllisiä mutantteja jätetään generoimatta (Lindström ja Márki 2019).

Sen lisäksi, että tarkan mutaatiopisteytyksen laskeminen on vaikeaa ekvivalenteista mutanteista johtuen, on todettu, että se ei ole muutenkaan järin toimiva tapa mittaamaan testijoukon laatua (Kurtz ym. 2016). Mutaatiopisteytys nousee vähilläkin testitapauksilla korkealle, mutta sen jälkeen nousu hidastuu huomattavasti suhteessa vaadittavien testitapausten määrään. Tämä johtuu muun muassa ylimääräisten mutanttien aiheuttamasta vääristymästä. Ylimääräiset mutantit näyttäytyvät tapettuina, vaikka suuri osa niistä on käytännössä triviaaleja

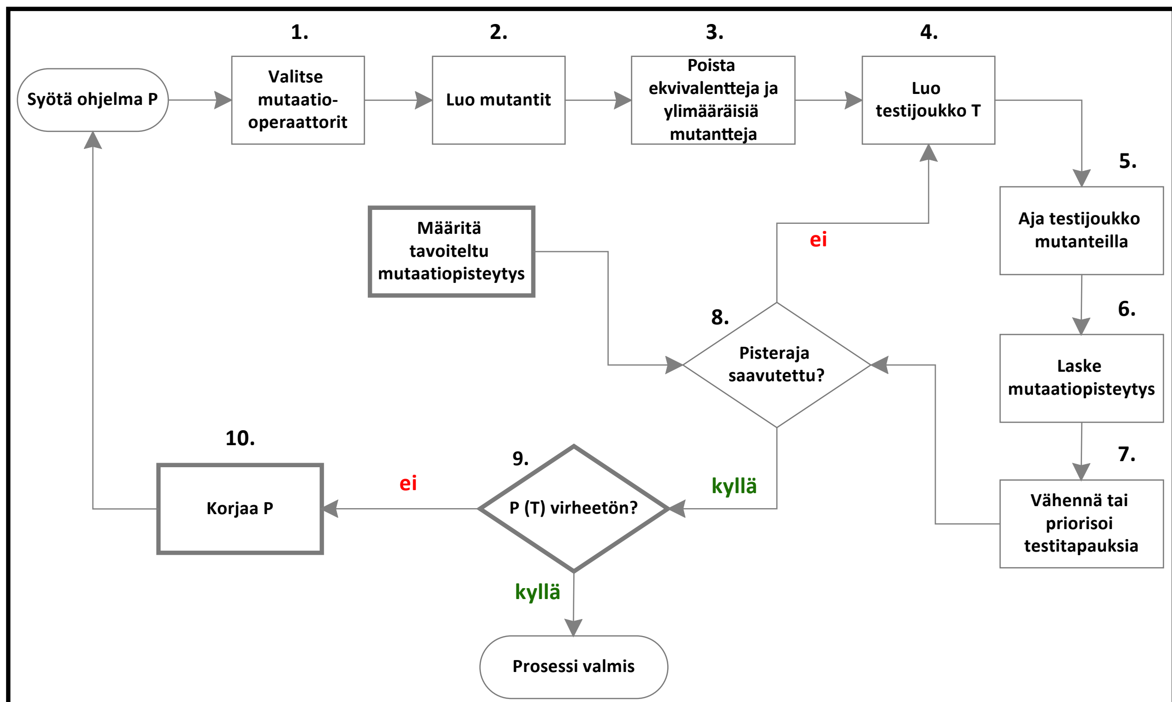
tappaa ja siihen pystyisi lähes mikä tahansa testitapaus.

Perinteisen mutaatiopisteytyksen tilalle onkin esitetty erilaisia menetelmiä, jotka paremmin kuvaavat todellista testauksen tilannetta. Näitä ovat esimerkiksi *subsuming mutation score* (Papadakis ym. 2016) sekä *dominator mutation score* (Kurtz ym. 2016). Molemmissa näistä jätetään tarpeettomat mutantit huomiotta ja pisteytys lasketaan vain niin kutsuttujen hallitsevien mutanttien (engl. *subsuming mutant* tai *dominator mutant*) perusteella. Tällöin mutaatiopisteytys nousee lineaarisesti testijoukon valmiuden mukaisesti ja sen avulla voidaan paremmin arvioida, kuinka paljon uusia testitapauksia – ja näin ollen työtä – vaaditaan haluttuun mutaatiopisteytykseen.

Joskus mutaatiotestauksen ongelmaksi esitetään myös se, että generoidut mutantit eivät välttämättä vastaa todellisia ohjelmointivirheitä (Nguyen ja Madeyski 2014). Koska generoidut mutaatiot ovat vain yksinkertaisia syntaktisia muutoksia ei niiden tappaminen testijoukolla välttämättä tarkoita sitä, että testijoukko löytäisi myös oikeita virheitä. Yhdistelmävaikutuksen ansiosta tämä ei kuitenkaan ole niin suuri ongelma, kuin miltä se saattaa vaikuttaa.

Just ym. (2014) käyttivät artikkelissaan *Are Mutants a Valid Substitute for Real Faults in Software Testing?* aineistona viittä laajaa Java-ohjelmistoa, jotka sisälsivät 357 todellista ohjelmointivirhettä ja joista generoitiin yli 200 000 mutanttia. He saivat tulokseksi, että 73 % todellisista ohjelmointivirheistä löydettiin yleisesti käytettyjen mutaatio-operaattoreiden avulla ja tätä osuutta olisi mahdollista vielä kasvattaa käyttämällä vahvempia mutaatio-operaattoreita. Tutkimuksessa todettiin, että 17:ää prosenttia todellisista virheistä ei pystytty simuloimaan millään mutanteilla.

Mutaatiotestauksen oleellisin ongelma eli sen huono skaalautuvuus ei ole tämän tutkielman puitteissa este, sillä kokeellista osaa varten aineistoksi valitaan yksinkertainen peli, jonka lähdekoodin määrä on suhteellisen vähäinen. Mutaatiopisteytykseen liittyvät ongelmat sen sijaan täytyy ottaa huomioon, kun vertailua suoritetaan eri mutaatiotekniikoiden välillä. Arviointia ja vertailua käsitellään tarkemmin luvussa 4.3.



Kuvio 1. Moderni mutaatiotestausprosessi (Papadakis ym. 2019).

2.3 Mutaatiotestauksen vaiheet

Offutt ja Untch (2001) esittivät kaavion, joka kuvaa perinteisen mutaatiotestausprosessin ja Papadakis ym. (2019) uuteen tietoon perustuvan ja päivitetyn version kyseisestä kaaviosta. Tämä tutkielma toteutetaan päivitettyä prosessia mukaillen, joka on esitetty kuviossa 1 ja jonka vaiheet käydään tarkemmin läpi tässä luvussa.

Kuviossa vaiheet, jotka välttämättömästi vaativat manuaalista työtä, on merkitty tummennetuilla laatikoilla. Tämän tutkielman puitteissa ei kuitenkaan ole järkevää alkaa automatisoimaan kaikkia muita vaiheita, sillä se veisi todennäköisesti enemmän aikaa kuin niiden suorittaminen manuaalisesti. Monien vaiheiden automatisoiminen on muutenkin vasta lähinnä tutkimuksen tasolla ja tosielämän mutaatioprosesseissa ne suoritetaan yhä manuaalisesti. Niinpä kuvion mukainen vaiheiden luokittelu ei vastaa tässä tutkielmassa automatisoitavia ja manuaalisesti suoritettavia vaiheita.

Mutaatiotestauksen ensimmäinen vaihe on mutaatio-operaattoreiden valinta. Tässä vaiheessa siis päätetään millaisia mutaatioita ohjelmakoodin halutaan tehdä. Kuten taulukossa 1 esitel-

lyistä esimerkkioperaattoreista käy ilmi, ovat mutaatiot tavallisesti hyvin pieniä ja yksinkertaisia muutoksia ohjelmakoodissa. Tätä tutkielmaa varten määriteltävät mutaatio-operaattorit poikkeavat hieman tästä periaatteesta, sillä mutaatioita generoidaan vain Unity-spesifiin syntaksiin ja niinpä ne eivät voi kaikissa tapauksissa olla aivan yhtä yksinkertaisia. Unity-spesifien mutaatio-operaattoreiden valintaprosessi esitellään luvussa 3.3.

Toinen vaihe mutaatiotestausprosessissa on mutanttien generointi. Määriteltyjen mutaatio-operaattoreiden mukaiset muutokset täytyy tehdä systemaattisesti ohjelmakoodiin ja kääntää niistä ajettavat ohjelmat. Perinteinen tapa generoida mutantit on luoda jokaiselle mutaatiolle uusi lähdekooditiedosto ja näin muodostaa suuri määrä ohjelmia, joista jokaisen lähdekoodissa on yksi muutos alkuperäiseen verrattuna (Papadakis ym. 2019). Vaihtoehtoisiakin tapoja tälle tekniikalle on tutkittu. Muun muassa (Harman, Jia ja Langdon 2010) ovat esittäneet artikkelissaan *A Manifesto for Higher Order Mutation Testing*, että korkeamman asteen mutaatiotestauksella – eli useiden mutaatioiden luomisella yhteen mutanttiin – voidaan paremmin simuloida todellisia ohjelmointivirheitä. Tällöin kuitenkin tarvitaan jonkinlaisia algoritmeja määrittämään, miten usean mutaation yhdistelmät luodaan, sillä kaikkien yhdistelmien läpikäymisessä mutanttien määrä todella kasvaisi täysin hallitsemattomaksi. Niinpä tässäkin tutkielmassa käytetään vain perinteistä ensimmäisen asteen mutaatiota. Myös silloin mutanttien määrä voi kasvaa hyvin suureksi, mutta tämä otetaan huomioon käytetyn aineiston valinnassa.

Kolmas vaihe on tarpeettomien mutanttien poistaminen. Kuten luvussa 2.2 mainittiin, tarpeettomat mutantit ovat suurimpia mutaatiotestauksen ongelmia ja niistä olisi hyvä poistaa mahdollisimman suuri osa ennen kuin testejä aletaan ajaa. Poistamalla tarpeettomat mutantit säästetään testijoukon ajossa aikaa ja saadaan mutaatiopisteytyksestä todenmukaisempi käsitys testauksen tasosta (Papadakis ym. 2019). Tämän vaiheen suoritus ennen testiajoja on kuitenkin ehkä järkevintä vain silloin, kun tarpeettomien mutanttien poistoon on käytettävissä automatisoituja keinoja. Tutkielman kokeellisessa osassa mutanttien analyysi suoritetaan manuaalisesti ja siksi tarpeettomat mutantit poistetaan joukosta vasta testiajon jälkeen. Näin säästetään aikaa analyysivaiheessa, kun esimerkiksi kuolleet mutantit voidaan jättää analysoimatta kokonaan. Toisaalta testiajot vievät silloin enemmän aikaa, mutta ne eivät kuitenkaan vaadi manuaalista työtä ja siksi sitä ei pidetä niin merkittävänä ongelmana. Tutkielman

puitteissa täysin kattavaan analyysiin ei ole resursseja, joten tarpeettomina poistetaan vain suurimpana ongelmana pidetty joukko eli ekvivalentit mutantit.

Neljännessä vaiheessa luodaan testijoukko, jota mutaatiotestauksen avulla pyritään kehittämään paremmaksi. Usein testijoukko on jo olemassa, kun mutaatiotestausprosessia aloitetaan. Käytännössä tämän testijoukon on oltava automaattisesti ajettava mutanttien suuren määrän vuoksi. Mutaatiotestauksen avulla testitapauksia on myös mahdollista generoida (Offutt 1988) ja parantaa (Baudry ym. 2002) automaattisesti. Tässä tutkielmassa sekä alkuperäiset testitapaukset että mutaatiotestauksen avulla selvitettävät parannukset testeihin toteutetaan kuitenkin manuaalisesti, sillä aineistona käytetty ohjelma on suhteellisen pieni ja tarvittavien testien toteuttaminen ilman automaatiota ei ole ongelma, vaan oletettavasti nopeampaa kuin automaation käyttöönotto.

Viides vaihe on varsinainen testien ajaminen aiemmin luoduilla mutanteilla. Tämä on ehkä laskennallisesti raskain osuus mutaatiotestausprosessista (Papadakis ym. 2019), sillä perinteisesti jokainen testijoukon testitapaus ajetaan jokaisella generoidulla mutantilla. Näin ohjelman ajojen määrä ja sitä myöten testaukseen kuluva aika kasvaa äkkiä testien määrän ja lähdekoodin laajuuden kasvaessa. Tähänkin ongelmaan on kuitenkin myös optimoidumpia ratkaisuja. Koska mutaatiopisteytyksen laskemiseen tarvitaan jokaisesta mutantista vain tieto, jäikö se eloon vai kuoliko se testeissä, voidaan testijoukon suoritus lopettaa kesken, mikäli jokin testi tappaa mutantin. Kun tämä tekniikka yhdistetään priorisoituun testijoukkoon, jossa eniten mutantteja tappavat testit ajetaan ensimmäisenä, voidaan säästää suuria määriä aikaa.

Kun testijoukot on ajettu jokaisella mutantilla, saadaan tieto eloon jääneistä mutanteista ja näin voidaan edetä kuudenteen vaiheeseen eli mutaatiopisteytyksen laskemiseen. Mikäli halutaan laskea todellinen mutaatiopisteytys, täytyy kaikki eloon jääneet mutantit myös analysoida ekvivalenttien mutanttien poistamiseksi laskusta. Koska tässä tutkielmassa mutanttien määrä pysyy inhimillisenä sopivasti valitun aineiston ansiosta, voidaan tämä analysointi toteuttaa.

Seitsemänneksi vaiheeksi Papadakis ym. (2019) esittävät testien vähentämisen tai priorisoinnin. Testejä voidaan poistaa turhina, jos ne eivät mitenkään vaikuta mutaatiopisteytykseen ja

testien ajojärjestystä voidaan priorisoida sen mukaan, mitkä testit tappavat eniten mutantteja. Näin testijoukon ajo voidaan suorittaa tehokkaammin. Tässä tutkielmassa tämä vaihe jätetään huomiotta, sillä testausprosessin optimointi ei ole tärkeää tutkielman näkökulmasta.

Kun mutaatiopisteytys on laskettu, arvioidaan kahdeksannessa vaiheessa testijoukon pätevyyttä sen perusteella. Mikäli kaikki tapettavissa olevat mutantit on tapettu – eli mutaatiopisteytys on 100 % – voidaan testijoukko todeta täysin päteväksi ja mutaatiotestausprosessi viedä eteenpäin. Kriteerinä voidaan pitää myös jotain muuta kuin täydellistä mutaatiopisteytystä, sillä sen saavuttaminen voi olla hyvin vaikeaa. Etenkin laajempien ohjelmistojen tapauksessa se tulee todella kalliiksi eikä sitä ole järkevää tavoitella (Petrovic ym. 2018). On kuitenkin tärkeää huomioida, että testijoukon kyky löytää virheitä paranee huomattavasti vasta korkeissa mutaatiopisteytyksissä (Papadakis ym. 2018). Mikäli mutaatiopisteytys ei ylitä päätettyä rajaa, siirrytään takaisin vaiheeseen 4, eli testitapausten luomiseen ja parantamiseen. Tavoitteena on kehittää testijoukkoa niin, että eloon jääneet mutantit saadaan tapettua. Tätä kiertoa jatketaan, kunnes haluttu mutaatiopisteytys saavutetaan.

Yhdeksännessä vaiheessa testijoukko on paranneltu mutaatiotestauksen avulla ja on aika ajaa alkuperäinen ohjelma tällä uudella testijoukolla. Jos alkuperäisessä testijoukossa oli puutteita, on todennäköistä, että uudella testijoukolla löydetään todellisia virheitä alkuperäisestä ohjelmasta. Mikäli näin tapahtuu, mennään vielä eteenpäin vaiheeseen 10, jossa korjataan alkuperäisestä ohjelmasta löydetyt virheet. Etenkin jos virheitä löytyy paljon ja niiden korjaaminen johtaa suuriin muutoksiin ohjelmakoodissa, on syytä aloittaa mutaatiotestausprosessi alusta, sillä uusi lähdekoodi johtaa tietenkin myös uusiin mutantteihin. Tämän tutkielman tarkoitukseen virheiden löytäminen aineistona käytetystä ohjelmasta ei ole kuitenkaan olennaista ja siksi mutaatiotestausprosessi suoritetaan vain kerran.

3 Unity-spesifi mutaatiotestaus

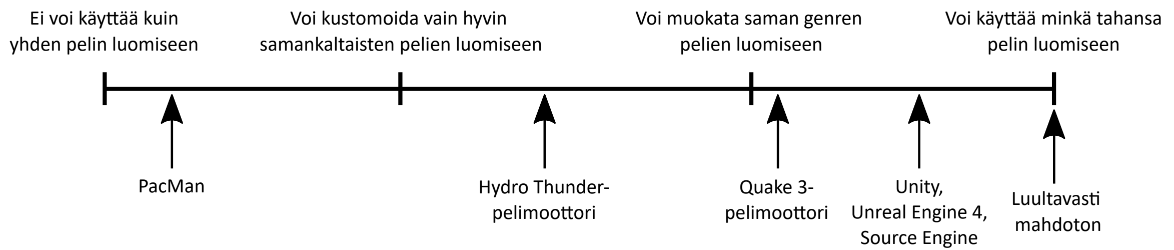
Tässä luvussa esitellään Unity-pelimoottori sekä sillä kehitettyjen pelien yleinen rakenne. Mutaatio-operaattoreiden valintaa varten käydään lävitse Unityn dokumentaatiossa esitetyt tärkeimmät luokat, minkä jälkeen operaattorit valitaan näiden sekä aiemman tutkimuksen perusteella. Sen jälkeen toteutetaan työkalu, johon implementoidaan valitut mutaatio-operaattorit ja joka mahdollistaa koko Unity-spesifin mutaatiotestausprosessin käytännössä.

3.1 Unity

Pelimoottori on termi, joka tuli käyttöön 1990-luvulla etenkin tunnetun ammuntapelin *Doomin* kautta (Gregory 2018). Doomissa pelin arkkitehtuuri oli luotu erottelemaan alemman tason komponentit kuten grafiikan renderöinnistä huolehtivat osat varsinaisesta pelin sisällöstä – esimerkiksi grafiikoista itsestään tai pelilogiikasta. Tämä mahdollisti pelin helpon muokattavuuden ja uusien pelien luomisen sen pohjalta, kun muutoksia voitiin tehdä vain pelin datan sisältäviin komponentteihin ja muut voitiin käyttää lähes sellaisenaan uudelleen. Nykyään näistä valmiista pohjakomponenteista koostuvaa ohjelmistokehystä nimitetään pelimoottoriksi.

Nämä pohjakomponentit luovat koko pelin perustan ja sen vuoksi niiltä vaaditaan hyvin tehokasta toteutusta. Siksi ei olekaan millään muotoa järkevää luoda niitä alusta alkaen uudestaan jokaista peliä varten (Bishop ym. 1998). Niinpä pelikehittäjät nykyään joko käyttävät ilmaiseksi saatavilla olevia pelimoottoreita tai ostavat lisenssin sellaiseen, mikä mahdollistaa pelikehityksen aloituksen suoraan pelin sisällöstä nopeuttaen prosessia huomattavasti. Monilla suurilla pelifirmoilla on myös omat suljetussa käytössä olevat pelimoottorinsa.

Pelimoottorien historia on hyvä ymmärtää hahmottaakseen, että pelimoottori ei varsinaisesti ole yksiselitteinen käsite ja etenkin vanhempien pelien kuten mainitun Doomin tapauksessa raja moottorin ja pelin itsensä välillä voi olla häilyvä. Tämän vuoksi Gregory (2018) esittääkin pelimoottorien uudelleenkäytettävyyden kirjon, joka on kuvattu kuviossa 2. PacMan esimerkiksi on kirjon alkupäässä ja niinpä sitä voi ennemminkin pitää pelkkänä pelinä pelimoottorin sijaan, vaikka siitäkin joitain uudelleen käytettäviä komponentteja voi löytyä.



Kuvio 2. Pelimoottorien uudelleenkäytettävyyden kirjo (Gregory 2018).

Spektrin toisessa päässä taas on moderneja nimenomaan pelimoottoriksi suunniteltuja ohjelmistokehyksiä, joiden avulla on mahdollista luoda hyvin monen tyyppisiä pelejä.

Yksi näistä moderneista pelimoottoreista on Unity ja tässä tutkielmassa pyritäänkin selvittämään mutaatiotestauksen mahdollisuuksia Unitylla luoduissa peleissä. Unity on etenkin pienempien pelikehittäjien keskuudessa suosituimpia pelimoottoreita tällä hetkellä ja on tieteellisessä kirjallisuudessakin todettu ominaisuuksiltaan erinomaiseksi monien pelimoottorien vertailussa (Christopoulou ja Xinogalos 2017) (Pattrasitidecha 2014). Se on myös julkaistujen pelien määrässä mitattuna käytetyimpiä pelimoottoreita, mikä käy ilmi esimerkiksi tarkasteltaessa Unity-pelien määrää Steam- ja itch.io-kaupoissa (Toftedahl ja Engström 2019). Unityn lisäksi vain Unreal Engine oli molemmissa palveluissa kymmenen käytetyimmän moottorin joukossa. Toftedahlin ja Engströmin tuloksien mukaan Itch.io:ssa Unity oli selvästi suosituin pelimoottori 47,3 %:n osuudella kaikista julkaistuista peleistä. Steamissa taas Unreal Engine oli eniten käytetty pelimoottori 25,6 %:n osuudella Unityn ollessa toisena 13,2 %:lla. Tämäkin kertoo Unityn suosiosta etenkin pienempien indie-kehittäjien keskuudessa, sillä itch.io mainostaa sivustoaan ”helppona tapana löytää ja jakaa indie-pelejä” (itch.io 2022), kun taas Steamissa on saatavilla myös suurempien yritysten tuotteita.

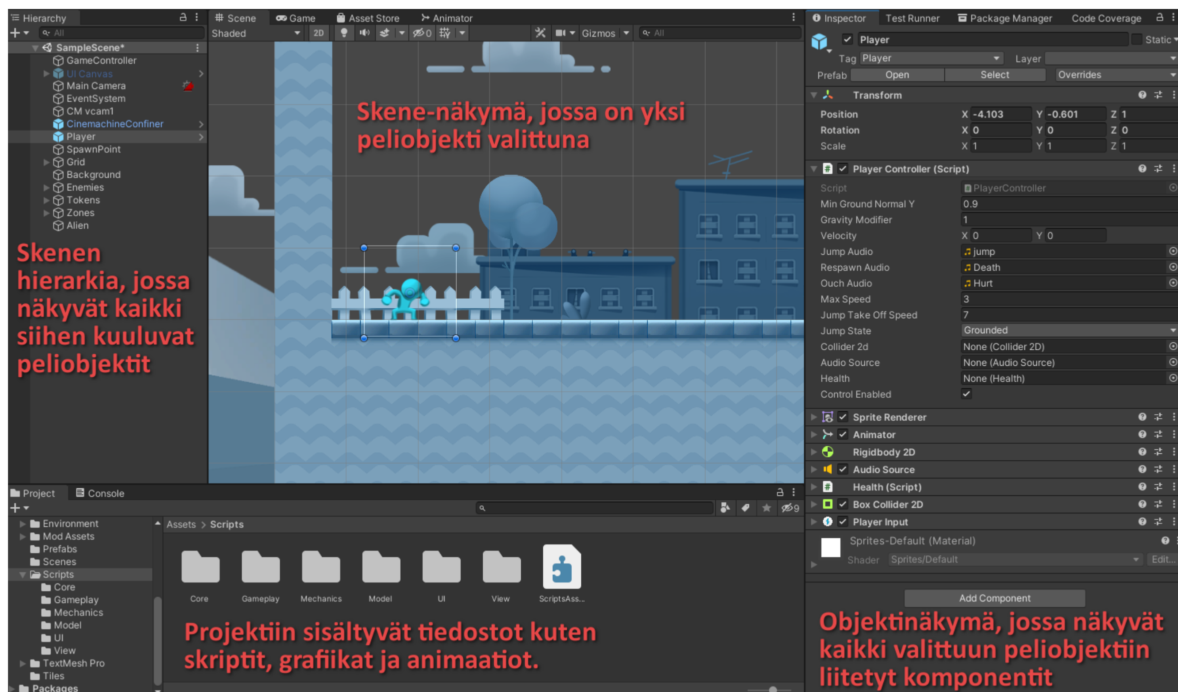
Unity mahdollistaa sekä 2D- että 3D-pelien kehityksen ja julkaisun lähes kaikille käytetyimmille pelialustoille (Unity 2022b). Unity tukee myös VR- ja AR-pelien kehitystä. Pelimoottorin lisäksi Unity tarjoaa muita relevantteja palveluita, kuten moninpelipalvelimia ylläpitävän *Multiplayn* ja oman *Asset Storen*, jossa käyttäjät voivat myydä tai ilmaiseksi tarjota esimerkiksi pelikomponentteja kuten grafiikkaa tai koodikirjastoja.

Unitya on mahdollista käyttää myös muihin tarkoituksiin kuin perinteisten viihdepelien luomiseen ja sitä onkin tutkittu akateemisesti useissa konteksteissa. Unityn avulla on luotu simulaatioita helpottamaan esimerkiksi baseballin pelaajien harjoittelua (Tsai ym. 2019) ja sairaalahenkilöstön koulutusta (Muangpoon ym. 2020). VR:n avulla voidaan myös tehokkaasti visualisoida esimerkiksi molekyylirakenteita ja siihen onkin kehitetty useita työkalu- ja Unitylla. Näitä esitellään erityisesti koronavirus SARS-CoV-2:n molekyylirakennetta tarkastelleessa artikkelissa *An immersive journey to the molecular structure of SARS-CoV-2: Virtual reality in COVID-19* (Calvelo Souto, Piñeiro ja Garcia-Fandino 2020).

Tässä tutkielmassa kohteena ovat kuitenkin vain viihdepelit, vaikka mutaatiotestauksesta voisi olla hyötyä muissakin käyttötarkoituksissa – etenkin kriittisten sovellusten kuten lentosimulaattoreiden tapauksessa, kun halutaan varmistaa ohjelman mahdollisimman oikea toiminta. Sovellusten käyttötarkoituksella saattaisi kuitenkin olla vaikutusta myös mutaatiotestauksen toteutukseen ja niin sanottujen vakavien sovellusten tapauksessa esimerkiksi paras mahdollinen mutaatio-operaattoreiden joukko voisi erota viihdepeleistä.

Unity-pelit koostuvat *skeneistä*, jotka ovat kappaleita sisältäviä 3D-avaruuksia – siis yleiskielellä pelikenttiä. Skenejen sisältämät kappaleet ovat Unityssa *peliobjekteja*, jotka taas koostuvat useista *komponenteista*. Kuviossa 3 on havainnollistettu visuaalisesti, miten nämä pelin eri osat näkyvät Unityn graafisessa editorissa. Erinäisiä komponentteja on Unityssa valmiina lukemattomia ja niitä voi myös itse luoda lisää. Kaikilla peliobjekteilla on esimerkiksi *Transform*-komponentti, joka sisältää tiedon objektin sijainnista ja asennosta skenen sisällä. Muita yleisiä komponentteja ovat muun muassa äänilähteet, animaattorit, 3D-renderöijät ja törmäyksentunnistajat.

Tämän tutkielman kannalta tärkeä komponentti on *skripti*. Unityn – ja tämän tutkielman – kontekstissa skriptit ovat kustomoituja komponentteja, jotka mahdollistavat muun muassa peliobjektien monipuolisen muokkaamisen, pelitapahtumien laukaisun ja käyttäjän syötteiden hallitsemisen pelin aikana (Unity 2022a). Käytännössä ne ovat C#-kielellä kirjoitettuja luokkia, jotka periytyvät Unityn *MonoBehaviour*-luokasta. MonoBehaviour-luokasta periytyminen mahdollistaa juurikin skriptien liittämisen peliobjekteihin ja Unityn tapahtumafunktioiden käytön, joiden avulla pelitapahtumiin kuten ruudun päivitykseen tai objektin törmäykseen voidaan reagoida. Vaihtoehtona MonoBehaviourista periytyville luokille ovat *Sc-*



Kuvio 3. Ruutukaappaus Unity-editorin oletusnäkymästä, johon on lisätty selittävät tekstit.

riptableObjectista periytyvät luokat, jotka on tarkoitettu erilaisen datan tallentamiseen ja joita ei liitetä peliobjekteihin.

Skriptien avulla muodostetaan tavallisesti pelin toimintalogiikka ja ne mahdollistavat peliobjektien välisen monipuolisen vuorovaikutuksen. Skriptit siis määrittävät usein koko pelin säännöt ja ovat välttämätön osa interaktiivista peliä. Siksi tutkielman kokeellisessa osassa mutaatiot luodaankin niihin.

Vaihtoehtokin Unityssa tosin on laajalti skriptipohjaisille peleille. Unity tarjoaa *Bolt*-työkalun visuaaliseen skriptaukseen, jossa ohjelmakoodin sijaan toimintalogiikka rakennetaan erilaisten kaavioiden avulla (Unity 2022c). Vastaavanlainen ja ehkä enemmän käytetty ominaisuus löytyy Unreal Engine -pelimoottorista termillä *Blueprints*. Bolt on kuitenkin suhteellisen uusi ominaisuus ja suurin osa Unitylla kehitetyistä peleistä ei sitä varmastikaan vielä käytä.

3.2 Unityn tärkeimmät luokat

Unity-spesifien mutaatio-operaattoreiden valitsemiksi on tiedettävä Unity-peleille uniikkeja ja usein käytettyjä ohjelmointirakenteita. Niinpä tässä luvussa esitellään Unityn tärkeimmät luokat, kuten Unityn dokumentaatiossa (Unity 2022g) on kerrottu. Siellä tärkeimmiksi luokiksi mainitaan:

- MonoBehaviour
- GameObject
- Object
- Transform
- Vectors
- Quaternion
- ScriptableObject
- Time
- Mathf
- Random
- Debug
- Gizmos/Handles

Kaksi viimeistä luokkaa liittyvät lähinnä pelikehitystä helpottaviin ominaisuuksiin. Esimerkiksi Debug-luokka mahdollistaa viestien kirjoittamiseen lokiin sekä yksinkertaisten viivojen piirtämisen. *Gizmot* taas mahdollistavat hieman monimutkaisempien muotojen piirtämisen ja *Handlet* interaktiivisten apukomponenttien lisäämisen Unity-editoriin. Koska nämä eivät ole tarkoitettu käytettäväksi lopullisessa pelissä käyttäjälle näkyviin ominaisuuksiin, ei niiden mutatointia pidetä hyödyllisenä tässä tutkielmassa.

Kuten aiemmassakin luvussa mainittiin, MonoBehaviour on Unityn yläluokka, josta muut skriptit usein peritään. Se mahdollistaa skriptin liittämisen peliobjektiin ja peliobjektin hallitsemisen Unityn tarjoamien tapahtumafunktioiden avulla. Yleisimpiä tapahtumafunktioita ovat esimerkiksi *Start*, joka kutsutaan vain kerran välittömästi skriptin aktivoiduttua sekä *Update*, joka kutsutaan jokaisen ruudunpäivityksen aikaan skriptin ollessa aktiivisena. Nämä metodit generoidaankin oletuksena automaattisesti uusiin skripteihin Unityssa. Se ei kui-

tenkaan tarkoita, että niiden käyttäminen olisi pakollista, mutta kertoo ehkä niiden tärkeydestä. Lisäksi MonoBehaviour-luokassa on joitakin yleisesti käytettyjä staattisia metodeita, olennaisimpina ehkä *Destroy* sekä *Instantiate*, joita käytetään peliobjektien poistamiseen ja lisäämiseen pelin aikana.

GameObject on yläluokka, jota kaikki skenessä olevat peliobjektit edustavat. Luokassa on useita paljon käytettyjä metodeita peliobjektien käsittelyyn ja monia niistä voidaankin käyttää myös mutaatio-operaattoreiden perustana. *Object* taas on vielä GameObjectia ylempi luokka, josta myös se on peritty. Object-luokassa ei kuitenkaan varsinaisesti ole mitään metodeita, joita ei jo otettaisi huomioon GameObjectin puolesta ja sitä ei olekaan tarkoitettu laajalti käytettävän skriptien kautta (Unity 2022e).

Transform on GameObjectin lisäksi toinen paljon skripteissä esiintyvä luokka. Transform pitää sisällään tietoa skenessä olevan peliobjektin sijainnista, asennosta ja koosta. Se myös sisältää paljon metodeita, joilla näihin ominaisuuksiin voidaan vaikuttaa ja esimerkiksi peliobjektien liikuttaminen toteutetaan tavallisesti joko Transformin tai fyysisten voimien avulla. Transform-luokka on myös hyvä mutatoinnin kohde, sillä objektien sijainteja, asentoja ja kokoja täytyy manipuloida lähes kaiken tyyppisissä peleissä.

Vector-luokkiin kuuluvat Vector2-, Vector3- ja Vector4-luokat. Vector2- ja Vector3-luokkia käytetään 2D- ja 3D-pelien tapauksessa kuvaamaan sijainteja pelimaailmassa. Vector4 taas on vähemmän käytetty luokka, josta on hyötyä tavallisimmin varjostimien kanssa toimiessa, kun tarvitaan RGBA-mallin (*red green blue alpha*) mukaisia neljän parametrin rakenteita. Vector-luokat antavat työkaluja vektorimatematiikkaan kuten peliobjektien välisten etäisyyksien ja suuntien laskemiseen. Esimerkiksi *MoveTowards* ja *Lerp* ovat paljon käytettyjä Vector-luokan metodeja, joiden avulla objekteja voidaan liikuttaa pehmeästi haluttuihin suuntiin. Myös Vector-luokat ovat toimivia kohteita mutaatioon, sillä niitä käytetään usein läheisesti Transformien kanssa toimittaessa.

Quaternion-luokan avulla esitetään objektien asentoa eri akseleiden suhteen. Quaternionin käsittelyn käytänteet ovat tärkeitä ottaa huomioon, sillä Unityn dokumentaatio kertoo niiden olevan vaikeita ymmärtää intuitiivisesti ja painottaa, ettei Quaternionin ominaisuuksia kannata suoraan muokata. Toisin kuin vaikka Transformin sijainnin muuttamista sen x-, y-

ja z-komponenttien avulla, Quaternionin käsittelyä tällä tavalla ei suositella (Unity 2022f). Sen sijaan Quaternion-luokkaa käytetään joskus metodien kautta, joilla voidaan esimerkiksi kääntää jokin objektin akseli toisen objektin suuntaan.

ScriptableObject on Unityn tarjoama ratkaisu datan säilyttämiseen. Tavallisesti skriptit – ja niiden sisältämien muuttujien data – liitetään skenessä olevaan peliobjektiin. Esimerkiksi jokainen vihollinen voi pitää sisällään skriptin, jossa on tieto sen liikkumisnopeudesta. Jos vihollisia on pelissä suuri määrä, on tämä kuitenkin turhaa resurssien kulutusta ja liikkumisnopeus voidaan tallettaa sen sijaan yhteen ScriptableObjectiin, josta jokainen vihollinen ottaa tämän tiedon. ScriptableObjectin käyttötarkoituksen voi usein korvata myös yksinkertaisesti tavallisella – siis ei MonoBehaviourista periytyvällä – C#-luokalla, mutta ScriptableObject tarjoaa paremman integroinnin Unityn editorin kanssa. Siitä esimerkiksi voidaan luoda helposti uusia instansseja sekä sen arvoja muuttaa editorissa. ScriptableObjectiin ei kuitenkaan liity sellaisia ominaisuuksia, joita olisi hyödyllistä mutatoida, joten se jätetään tässä mutaatio-operaattoreiden valinnassa huomiotta.

Time-luokka on tärkeä, kun halutaan saada tietoa ruudunpäivitysnopeudesta pelin aikana. Koska Unityssa Update-metodia kutsutaan jokaisen ruudunpäivityksen kohdalla, on siinä tehtävissä toiminnoissa otettava huomioon ruudunpäivitysnopeus. Pelissä olevaa objektia ei siis esimerkiksi liikuteta metriä jokaisessa Update-kutsussa vaan objektia liikutetaan metri * *Time.deltaTime*. Time.deltaTime kertoo kuinka kauan viimeisimmästä ruudun päivityksestä on kulunut aikaa. Näin objekti liikkuu yhtä paljon riippumatta ruudun päivityksen – ja sitä myötä Update-metodin kutsujen – nopeudesta. Tämä on kenties tavallisin käyttötarkoitus Time-luokalle, mutta sen avulla voidaan tehdä muutakin, kuten esimerkiksi muuttaa pelitapahtumien nopeutta tai kokonaan pysäyttää peli.

Mathf on Unityn tarjoama luokka matemaattisiin funktioihin, joita pelikehityksessä yleisesti tarvitaan. Toiminnallisuudeltaan se vastaa pitkälti C#:ssa yleisesti käytettävää *Math*-luokkaa. Suurimpana – ja monessa tapauksessa ainoana – erona on, että Mathf-luokan metodit palauttavat tuloksen float-tyypin muuttujana, kun taas Math-luokan metodit käyttävät double-muuttujia. Monet Mathf-luokan metodit ovatkin kuoren alla vain Math-luokan kutsuja, joissa tulos muunnetaan float-tyyppiseksi. Tämän vuoksi Mathf-luokka ei varsinaisesti tarjoa mitään uutta Unity-spesifiä toiminnallisuutta ja siksi se jätetään huomioimatta mutaa-

tio-operaattoreiden valinnassa.

Random on toinen Unityn tarjoama luokka, josta on myös olemassa geneerisempi versio C#-käytössä, tässä tapauksessa vielä täysin samalla nimellä. Unityn Random-luokka tarjoaa kuitenkin valmiita metodeja, joilla voi esimerkiksi suoraan hakea satunnaisia objektin asentoja tai pelimaailman pisteitä halutun alueen sisältä. Tämän vuoksi Random-luokkaa voidaan ajatella enemmän Unity-spesifinä ominaisuutena kuin Mathf-luokkaa ja siten myös sen mutatointi voisi olla järkevämpää.

3.3 Mutaatio-operaattoreiden valinta

Mutaatiotestaukseen vaaditaan joukko mutaatio-operaattoreita, joihin perustuen mutaatiot generoidaan (Papadakis ym. 2019). Niinpä Unity-spesifiin mutaatiotestaukseen tarvitaan mutaatio-operaattorit, jotka kohdistuvat Unity-peleissä ilmeneviin rakenteisiin.

Unity-spesifillä mutaatiotestauksella tarkoitetaan tässä tutkielmassa mutaatiotestausta, jossa mutaatiot luodaan ohjelmakoodin kohtiin, joissa käytetään Unityn sisäänrakennettuja luokkia. Näin mutaatiot simuloivat nimenomaan ja ainoastaan Unity-projekteissa esiintyviä ohjelmointivirheitä. Unity-spesifiä mutaatiota voitaisiin toteuttaa kuitenkin monilla muillakin tavoilla, mitkä voisivatkin olla mielenkiintoisia tutkimusaiheita tulevaisuudessa.

Unity-pelien komponenttipohjaisen rakenteen vuoksi mutaatiotestausta olisi mahdollista suorittaa kokonaan muuttamatta varsinaista ohjelmakoodia. Sen sijaan mutaatioita voitaisiin luoda projektissa oleviin tiedostoihin, joissa on dataa pelin sisältämistä objekteista. Esimerkiksi peliobjektien painoa tai törmäyksentunnistusta voitaisiin muokata luomalla mutaatioita objektien *Prefab*-tiedostoihin. Tällöin mutaatiot simuloisivat pikemminkin pelikehittäjän tekemiä virheitä Unity-editorin puolella eikä ohjelmakoodissa. Myös Ghayyur (2018) antaa Android-alustaisiin peleihin keskittyvässä tutkielmassaan esimerkin mutaatiosta, joka ei perustu ohjelmakoodin muutokseen. Siinä pelin manifestitiedostoon, jossa on tieto muun muassa käyttäjän mobiililaitteessaan sallimista käyttöoikeuksista, luodaan mutaatioita. Tälläkin tavalla voidaan hyvin helposti aiheuttaa ongelmatilanteita pelin toimintaan.

Tässä tutkielmassa halutaan kuitenkin tarkastella nimenomaan ohjelmakoodiin perustuvaa

mutaatiotestausta ja siksi mutaatioita generoidaan vain Unity-projektissa oleviin C#-skripteihin.

Erilaisiin tarkoituksiin suunniteltuja operaattoreiden joukkoja on esitetty tieteellisessä kirjallisuudessa paljon. Operaattorijoukot ovat usein erikoistuneita tietyn tyyppisille ohjelmistoille tai tietyille ohjelmointikielille tai niiden kategorioille (Papadakis ym. 2019). Esimerkiksi Android-ohjelmistojen mutaatiota on tutkittu paljonkin viime vuosina ja sitä varten on kirjallisuudessa ehdotettu useita mutaatio-operaattoreiden joukkoja (Deng ym. 2017; Vásquez ym. 2017).

Tällaista mutaatio-operaattoreiden tutkimusta ei kuitenkaan löytynyt lähes lainkaan liittyen Unityyn tai pelikehitykseen yleisemminkään. Mirshokraie, Mesbah ja Pattabiraman (2013) sovelsivat JavaScriptiin keskittyneessä tutkimuksessaan luomaansa operaattorijoukkoaan JavaScript-pohjaiselle pelille, mutta operaattorit itsessään eivät liittyneet peliohjelmointiin.

Ainoa asiaan liittyvä tutkimus vaikuttaisikin olevan yksi Unity-spesifiä mutaatiota tarkastellut pro gradu -tutkielma (*master's thesis*) (Ghayyur 2018). Siinä Ghayyur perehtyy Unity-pelien mutaatiotestaukseen ja esittelee myös mutaatio-operaattoreita tehtävää varten. Tutkielma kuitenkin tarkastelee asiaa vain Android-pohjaisten Unity-pelien kautta ja siksi kaikki tulokset eivät sovellu tämän tutkielman tarkoituksiin, jossa näkökulmana on kaikki Unitylla luodut pelit.

Koska Ghayyurin tutkielman lisäksi muuta aiheeseen liittyvää tutkimusta ei ollut tehty, päätettiin mutaatio-operaattoreiden joukko luoda etenkin aiemmassa luvussa esiteltyjen Unityn tärkeimpien luokkien (Unity 2022g) pohjalta. Tämä tehtiin kuitenkin ottaen huomioon myös Ghayyurin esittämät operaattorit siltä osin kuin ne tähän tutkielmaan soveltuivat ja niiden valinta tuntui perustellulta. Suuri osa Ghayyurin operaattoreista päätyikin mukaan kokonaan tai lähes samanlaisina myös Unityn tärkeimpien luokkien perusteella. Osaa taas ei pidetty sopivina niihin liittyvien ohjelmointirakenteiden arvioidun harvinaisuuden vuoksi. Tällaisia olivat pelikameran tyyppiin ja *PlayerPrefs*-tiedostoon liittyvät mutaatio-operaattorit. Lisäksi tutkielman näkökulman vuoksi se sisälsi vain Android-alustaisille peleille suunniteltuja mutaatio-operaattoreita, jotka myös jätettiin pois tästä tutkielmasta.

Lopullinen joukko koostui 17:stä mutaatio-operaattorista, joista 15 liittyi Unityn esittelemien

tärkeimpien luokkien metodien ja ominaisuuksien mutatointiin. Kaksi operaattoria taas valikoitui mukaan Ghayyurin tutkielman perusteella, vaikkeivat ne mutatoineetkaan juuri näitä tärkeimpiä luokkia. Operaattoreiden valinta tehtiin sen mukaan, kuinka yleisesti käytetyiksi ja geneerisiksi ohjelmointirakenteet arvioitiin. Esimerkiksi MonoBehaviour-luokan *Instantiate*-metodiin liittyvä mutaatio-operaattori valittiin joukkoon, koska suuressa osassa peleistä peliobjekteja täytyy luoda pelin ajon aikana tätä metodia käyttäen. Toisaalta samasta luokasta ei mutatoitu vaikkapa metodeja *OnFailedToConnect* tai *OnMasterServerEvent*, sillä niitä käytetään vain verkkomoninpeleissä.

On syytä huomata, että Unityn ohjelmointirakenteiden yleisyyteen liittyen ei löytynyt aiempaa tutkimusta tai muutakaan dataa. Niinpä arviointi on tehty vain tutkijan oman asiantuntemuksen pohjalta eikä välttämättä vastaa tarkasti todellisuutta. Tulevissa tutkimuksissa olisikin hyvä pohtia perusteita mutaatio-operaattoreiden valinnalle ja mahdollisesti parempaa operaattoreiden joukkoa.

Valittujen mutaatio-operaattoreiden joukko on esitelty taulukossa 3 jaettuna kategorioihin, jotka vastaavat aiemmassa luvussa esiteltyjä Unityn tärkeimpiä luokkia. Osa mutaatio-operaattoreista saattaa aiheuttaa mutaatioita usean luokan metodeihin, sillä monet metodeista periytyvät useisiin luokkiin. Esimerkiksi *CompareTag*-metodi on olemassa sekä MonoBehaviour-, GameObject- että Transform-luokissa. Tällöin metodi on luokiteltu siihen luokkaan, josta sitä yleisimmin arvioitiin käytettävän. Pelkästään Ghayyurin mutaatio-operaattorijoukon pohjalta tähän tutkielmaan valitut operaattorit on listattu taulukon Muut-kategoriassa.

Osa alkuperäisesti valituista mutaatio-operaattoreista poistettiin käytöstä alustavien testien perusteella. Nämä sisälsivät Ghayyurin esittämän *Awake*- ja *Start*- metodien mutatoinnin keskenään sekä vastaavan vaihdoksen *Update*- ja *FixedUpdate*-metodeille. Myös *deltaTime*- ja *fixedDeltaTime*-muuttujia oli tarkoitus mutatoida keskenään alkuperäisessä suunnitelmassa. Näiden kaikkien operaattoreiden todettiin kuitenkin luovan huomattavan määrän ekvivalentteja ja vaikeasti analysoitavissa olevia mutantteja eikä juurikaan todellisesti hyödyllisiä mutantteja. Niinpä ne poistettiin mutaatio-operaattoreiden joukon lopullisesta versiosta. Nämä käyttämättä jätetyt mutaatio-operaattorit ovat nähtävissä taulukossa 4.

Kategoria	Lyhenne	Nimi
MonoBehaviour	DPR	Destroy-method parameter replacement
	INPR	Instantiate-method parameter replacement
	IVPR	Invoke-method parameter replacement
	SCPR	StartCoroutine-method parameter replacement
GameObject	FPR	Find-method parameter replacement
	FTPR	FindWithTag-method parameter replacement
	CTPR	CompareTag-method parameter replacement
	GCPR	GetChild-method parameter replacement
	SAPR	SetActive-method parameter replacement
	ASR	Activity state replacement
Transform	TDR	Transform direction replacement
	TPR	Transform parent replacement
Vector	VDR	Vector direction replacement
	VAR	Vector axis replacement
	VMR	Vector magnitude replacement
Muut	LSPR	LoadScene-method parameter replacement
	ALMR	AddListener method replacement

Taulukko 3. Unity-spesifit mutaatio-operaattorit

Kategoria	Lyhenne	Nimi
Unity-tapahtumafunktiot	IER	Initialization event replacement
	UER	Update event replacement
Time	DTR	Delta time replacement

Taulukko 4. Lopullisesta versiosta poistetut mutaatio-operaattorit

3.4 Unity Mutator

Tutkielmaa varten täytyi kehittää uusi työkalu, sillä mitään Unitylle suunniteltua mutaatiotyökalua ei ollut saatavilla. Työkalun vaatimuksina oli Unity-projektin skriptien mutatointi käyttäen määriteltyjä Unity-spesifejä mutaatio-operaattoreita. Lisäksi työkalulta vaadittiin kyky ajaa automaattisesti Unityyn luotu testijoukko näillä kaikilla mutanteilla ja raportoida tämän ajon tulokset helposti tarkasteltavassa muodossa. Kokeellista osaa varten tarvittiin myös työkalu, joka pystyy helposti automatisoimaan toisella, perinteisellä mutaatiotyökalulla generoitujen mutanttien testiajon Unityssa.

Näihin vaatimuksiin kehitettiin *Unity Mutator*, joka koostuu kolmesta osasta: pääohjelmasta, *ResultsParserista* sekä *CreamUnityTestRunnerista*. Pääohjelma vastaa mutaatioiden luomisesta ja Unitylle luodun testijoukon ajamisesta mutanteilla. ResultsParser kokoaa HTML-sivun, josta mutaatioajon tulokset ovat ymmärrettävässä muodossa tarkasteltavissa ja analysoitavissa. CreamUnityTestRunner taas on oikeastaan erillinen työkalu, joka käyttää sekä pääohjelman että ResultsParserin osia ja niiden avulla ajaa Unityn testit ja generoi tulosraportin *CREAM*-ohjelmalla luoduista mutanteista. CREAM:a käytetään kokeellisessa osassa vertailukohtana ja se esitellään tarkemmin luvussa 4.2.1. CREAM generoi mutantit eri logiikalla kuin Unity Mutator ja siksi testien ajaminen ja raportin generointi täytyi toteuttaa niille omalla tavallaan.

Unity Mutatorin toimintaperiaate on yksinkertainen. Se käy rivi riviltä lävitse syötetyn Unity-projektin jokaisen *Scripts*-kansiossa olevan C#-tiedoston ja luo tiedostosta mutatoidun version, mikäli rivillä esiintyy mutaatio-operaattoreita vastaavaa syntaksia. Jos tiedostosta luotiin mutaatio, korvataan projektin alkuperäinen tiedosto sillä, jonka jälkeen ajetaan projektille Unityssa luotu testijoukko ja tallennetaan sen perusteella generoituva tulosraportti. Kun jokaisen C#-tiedoston jokainen rivi on käyty läpi, luo ResultsParser koosteraportin kaikkien mutaatioajojen tuloksista. Tämä raportti on HTML-sivu, josta nähdään sekä mutaatioajon tulos kokonaisuudessaan että jokaisen mutantit tiedot yksitellen. Näin testaaja pystyy helposti analysoimaan mutantteja esimerkiksi ekvivalenttiuden kannalta.

CREAM-työkalun toiminta eroaa hieman Unity Mutatorista. Se generoi jokaista mutanttia varten kokonaan uuden kansion, joka sisältää koko Unity-projektin. Tämä on huomattavasti

aikaa vievempää kuin Unity Mutatorin periaate, jossa samaan projektiin vain vaihdetaan mutatoitu tiedosto. Toisaalta lopputuloksena CREAM:sta saadaan mutatoidut Unity-projektit, joita voidaan tarvittaessa käsitellä myöhemmin uudestaan, suorittamatta koko mutaatioiden generointiprosessia uudestaan. Tutkielman tarkoituksia varten tätä ei kuitenkaan pidetty tärkeänä Unity Mutatorissa, vaan nopeamman generointiprosessin mahdollistava tekniikka valittiin käytettäväksi.

CREAM:n toiminnan eroavaisuuksien vuoksi myös ResultsParserin generoima tulosraportti eroaa CREAM-mutaatioiden ja Unity Mutator -mutaatioiden välillä. CREAM:lla generoiduista mutanteista ei saada yhtä paljon tietoa tulosraporttiin, koska CreamUnityTestRunner ei saa dataa sille annettujen mutatoitujen ohjelmien lähdekoodista verrattuna alkuperäiseen. Näin ollen tulossivulla ei voida näyttää esimerkiksi sitä, miten alkuperäistä koodia on muutettu kussakin mutaatiossa. Tämä tieto on kuitenkin saatavilla CREAM:sta ja tulossivulla on mutanteille annettu CREAM:n kanssa täsmäävät nimet. Näin tulosten analyysin voi tehdä käyttäen generoidun raportin apuna CREAM:n tarjoamaa tietoa mutanteista. Tämä tekee analyysiprosessista hieman vaikeamman kuin Unity Mutatorin suorittaman mutatoinnin tapauksessa. Arvioitiin kuitenkin, että ohjelman kehittämiseen kuluva aika olisi suurempi vaiva kuin pieni lisätyö analyysivaiheessa, joten se hyväksyttiin.

Unity Mutatorin lähdekoodi on saatavilla osoitteessa `https://github.com/Vahv1/UnityMutator`. On kuitenkin syytä huomata, että Unity Mutator on vain tämän tutkielman tavoitteita varten toteutettu, eikä sitä ole tehty muuta käyttöä huomioon ottaen. Se muun muassa sisältää kovakoodattuja muuttujia liittyen kansiorakenteisiin, joita käyttäjän tulisi muokata ennen käyttöä. Se ei myöskään ole ominaisuuksiltaan yleiseen käyttöön valmis ja sisältää vakavuudeltaan vähäisiksi arvioituja puutteita, jotka eivät ole este työkalun käytölle tämän tutkielman puitteissa. Mitään takuita ohjelmiston toimivuudesta muussa käytössä ei kuitenkaan ole ja lähdekoodi on asetettu nähtäville pikemminkin ohjelman toiminnan esittämiseksi kuin jaettavaksi yleiseen käyttöön.

Tutkielman tarkoituksen kannalta merkittävin ongelma liittyy Unity Mutatorin tapaan iteroida ohjelmakoodia riveittäin eteenpäin, mikä johtaa siihen, että yhdelle riville voidaan suorittaa vain yksi mutaatio. Jos yhdellä koodirivillä on monta mutatoitavaa kohtaa, niiden tyypistä riippuen joko yhteen mutanttiin tehdään usea mutaatio tai jokin kohta jää mutatoimatta. Tä-

mä ei kuitenkaan ole liian suuri ongelma, sillä tällaiset tapaukset ovat suhteellisen harvinaisia. Lisäksi yksittäisten mutanttien puuttumisella ei ole merkittävää eroa tuotetun testijoukon virheenlöytökykyyn ainakaan perinteisten mutaatio-operaattoreiden tapauksessa (Papadakis ja Malevris 2010). Unity-spesifeillä operaattoreilla mutantteja generoidaan vähemmän, joten yhden mutantin vaikutus on oletettavasti suurempi, mutta idean voidaan ajatella yhä pätevän.

Muita mainittavia, mutta tutkielman tulosten kannalta merkityksettömiä ongelmia ovat Unity Mutatorin heikko mukautuvuus ja virheidensietokyky. Mukautuvuuden kannalta vaadittaisiin yleiseen käyttöön julkaistussa versiossa mahdollisuus valita, mitä skriptejä Unity-projektista mutatoidaan. Nyt Unity Mutator mutatoi kaikki skriptit, jotka projektin Scripts-kansiossa ovat, mikä johtaa myös siihen, että projektilta vaaditaan täsmällinen kansiorakenne. Mukautuvuuteen liittyen ohjelmassa olisi toivottavaa olla myös mahdollista valita, mitä mutaatio-operaattoreista käytetään. Tällainen ominaisuus on olemassa useimmissa mutaatiotyökaluissa.

Virheiden sietoon ja niistä toipumiseen liittyviä ongelmia on, että ohjelman kaatuessa tai keskeytyessä se ei generoi siihenastisia testauksen tuloksia ResultsParserin avulla lainkaan. Jos taas Unity kaatuu tai keskeytetään manuaalisesti kesken testien ajon, voi Unity Mutator myös kaatua. Suurin virheensieto-ongelma prosessissa johtuu kuitenkin Unitysta itsestään ja vaatisi tarkempaa selvitystä, onko sitä mahdollista estää Unity Mutatorin kautta. Jos ohjelmakoodi päätyy ikuiseen silmukkaan tai muuten saa pelin jumiin kesken Unityn automaattisten testien, ei Unity toivu siitä mitenkään tai esimerkiksi aikakatkaise testiä. Mutaatiotestauksen kannalta tämä on iso ongelma, sillä mutatoitu koodi voi helposti johtaa tällaiseen tilanteeseen. Silloin testaajan pitää manuaalisesti keskeyttää ja aloittaa prosessi uudestaan.

4 Tutkimusasetelma

Tutkielman kokeellisessa osassa perinteisen ja Unity-spesifin mutaatiotestauksen tuloksia pyrittiin vertaamaan keskenään. Tässä luvussa kuvataan kuinka tämä vertailu tehtiin, mitä työkaluja sen toteutukseen tarvittiin ja millaista aineistoa siihen käytettiin.

4.1 Peliaineisto

Mutaatiotekniikoiden vertailua varten tarvittiin Unity-peli, jolle mutaatiotestausta suoritettiin. Aineistona käytettäväksi peliksi etsittiin yksinkertaista ja ohjelmakoodin määrältään suhteellisen suppeaa avoimen lähdekoodin peliä. Vaatimus johtui mutaatiotestauksen vaatimista resursseista ja niiden nopeasta kasvusta ohjelman koon kasvaessa, minkä vuoksi tämän tutkielman puitteissa ei ollut mahdollista tutkia laajempaa kokonaisuutta. Peliltä myös toivottiin valmista Unity Test Frameworkin avulla toteutettua automatisoitua testijoukkoa, joka olisi joko sellaisenaan tai pienin muutoksin toiminut luvussa 4.3 määriteltynä testijoukkona 1. Tällaista molempia vaatimusta täyttävää aineistoa ei kuitenkaan löytynyt ja lopulta testijoukko 1 päädyttiin luomaan alusta asti tutkielmaa varten, sillä ensimmäistä vaatimusta pidettiin tärkeämpänä.

Aineistona käytettäväksi peliksi valittiin Unityn tarjoama *Platformer Microgame*-peli, joka oikeastaan on vain pohja, josta laajempaa peliä on tarkoitus alkaa rakentaa (Unity 2022d). Se toimii kuitenkin myös valmiina pelattavana pelinä, jolla on selkeät ominaisuudet, joita testata. Pelin koodipohja koostuu vain 36 skriptistä ja noin 2000 rivistä koodia, joten se sopii hyvin tämän tutkielman aineistoksi ilman, että mutanttien määrä kasvaa hallitsemattomaksi. Platformer Microgame on perinteinen 2D-tasohyppely, jossa pelaajan tavoitteena on edetä maaliin keräten kentästä löytyviä merkkejä ja väistellen tai tuhoten vihollisia. Seuraavassa listassa on tarkemmin lueteltuna pelissä havaitut ja testattavaksi päätetyt ominaisuudet.

- Liikkuminen
- Hyppy
- Kuolema
- Vihollisen tuhoaminen

- Merkin kerääminen
- Pelin voittaminen
- Valikon avaaminen

Pelin alkuperäiseen lähdekoodiin täytyi tehdä joitain muutoksia, jotta se saatiin toimimaan testityökalujen kanssa. Testauksessa käytettävä luokka *InputTestFixture*, jonka avulla simuloidaan pelaajan näppäinpainalluksia, vaatii Unityn uuden *Input Systemin* käyttöä. Platformer Microgame taas on toteutettu vanhaa *UnityEngine.Inputia* käyttäen. Tästä johtuen kaikki näppäinsyötteiden käsittely täytyi refaktoroida koodissa käyttämään InputTestFixturen kanssa yhteensopivia tapoja.

Testitapausten luomisen helpottamiseksi joillekin objekteille myös lisättiin tunnisteita tai niiden nimiä vaihdettiin. Lisäksi pelistä muutettiin yksi ominaisuus, joka arvioitiin virheeksi ja jonka muuttaminen teki testitapauksista loogisempia. Alkuperäisessä ohjelmassa pelaajahahmo syntyi kuoleman jälkeen hieman eri paikkaan kuin pelin alussa. Tätä muokattiin niin, että pelaaja syntyy aina samaan kohtaan pelikenttää. Tätä korjausta lukuun ottamatta mikään muutoksista ei vaikuttanut pelin toimintaan, vaan se säilyi identtisenä alkuperäisen ohjelman kanssa.

Lisäksi lähdekoodiin täytyi tehdä muutoksia, jotta C#-mutaatioon käytettävä CREAM-työkalu saatiin toimimaan ilman häiriöitä. CREAM ei osannut käsitellä joitain rakenteita pelin koodissa johtuen ehkä ominaisuuksista, joita C#-kieleen on kehitetty viimeisimmän CREAM-julkaisun jälkeen. Ongelman syihin ei kuitenkaan tarkemmin perehdytty sillä muutokset olivat helppoja tehdä ja sisälsivät vain koodin refaktorointia ja tiettyjen rakenteiden muuttamista toiseen muotoon. Myöskään nämä muutokset eivät vaikuttaneet pelin toimintaan lainkaan.

4.2 Testaustyökalut

Luvussa 3.4 esitellyn ja tätä tutkielmaa varten kehitetyn Unity Mutatorin lisäksi kokeellisessa osassa apuna käytettiin myös valmiita työkaluja ja ohjelmistokehyksiä. Tässä luvussa esitellään niistä tärkeimmät: perinteiseen mutaatioon käytetty *CREAM* sekä Unity-projektin automaattisten testien luomiseen ja ajamiseen käytetty *Unity Test Framework*. Muitakin työkaluja tutkimuksessa toki hyödynnettiin – esimerkiksi Unityn *Code Coveragea* laskemaan ja

raportoimaan luotujen testijoukkojen koodikattavuutta – mutta ne eivät olleet niin olennaisessa roolissa kuin kaksi aiemmin mainittua.

4.2.1 CREAM

Vertailtavan työkalun valintaa varten haettiin tieteellisiä julkaisuja C#-kielelle suunniteltuista mutaatiotyökaluista. Vertailtavaksi haluttiin työkalu, joka ottaa huomioon nimenomaan olio-ohjelmoinnin rakenteet ja C#-kielen mutaatioiden luonnissa, sillä on syytä uskoa, että perinteiset mutaatio-operaattorit yksinään eivät ole riittäviä simuloimaan olio-ohjelmistojen virheitä (Derezinska ja Kowalski 2011). Ajatusta mukaillen päästään myös tämänkin tutkielman kysymyksiin. Voisiko mutaatiotestausta yhä kehittää käyttämällä erikoistuneempia mutaatio-operaattoreita – kuten juuri Unity-peleille suunniteltuja?

Lopulta työkaluksi valittiin CREAM. CREAM on avoimen lähdekoodin mutaatiotyökalu, joka on kehitetty C#-ohjelmien mutaatiotestausta varten. Se sisältää perinteisten mutaatio-operaattoreiden lisäksi operaattoreita, jotka ottavat huomioon olio-ohjelmoinnin ominaisuuksia ja simuloivat niihin perustuvia virheitä. CREAM esiteltiin ensi kertaa vuonna 2007 julkaistussa pro gradu -tutkielmassa (*master's thesis*) (Szustek 2007) ja sitä ovat siitä lähtien kehittäneet oliopohjaiseen mutaatioon perehtyneet tutkijat. Tässä tutkielmassa käytettiin versiota CREAM 3.0.

Muitakin oliopohjaisia mutaatiotyökaluja on olemassa ja muun muassa Uzunbayir ja Kurtel (2019) ovat vertailleet niiden ominaisuuksia. He käsittelivät kuutta eri C#-mutaatiotyökalua mukaan lukien CREAM:a. Näistä kolmessa – CREAM:ssa, *NinjaTurtlesissa* sekä *Visual Mutatorissa* on mahdollisuus käyttää oliopohjaisia mutaatio-operaattoreita. Tarkemman ohjelmiin perehtymisen jälkeen CREAM valittiin sen akateemisen taustan sekä etenkin mutanttien generointitavan vuoksi. CREAM luo jokaiselle mutantille kokonaan uuden projektikansion ja näin on helppo automaattisesti ajaa niille kaikille Unity-testit. Esimerkiksi Visual Mutator taas on Visual Studioon integroitu lisäosa, jonka käyttäminen Unityn kanssa yhdessä olisi ollut vaikeampi toteuttaa.

CREAM sisältää kahdeksan perinteistä ja 18 oliopohjaista mutaatio-operaattoria, joista testaajan on mahdollista valita, mitä haluaa käyttää. Oliopohjaiset operaattorit luovat mutaa-

tioita olio- ja C#-ohjelmille spesifeihin rakenteisiin ohjelmakoodissa. Esimerkiksi EOC-operaattori vaihtaa ==-vertailuoperaattorin C#-metodiin *Equals* tai päinvastoin (Derezinska ja Szustek 2008). Näin olion viittauksen vertaaminen vaihtuukin olion sisällön vertaamiseen, joka saattaa aiheuttaa häiriöitä ohjelman toiminnassa.

Myös tiedostot joihin mutaatioita halutaan luotavan, on mahdollista valita. Tämän voi tehdä joko manuaalisesti tai määrittämällä mutatoitavat kohdat koodikattavuuden perusteella. CREAM tukee muun muassa Visual Studion ja NCover:n luomia koodikattavuustiedostoja. CREAM tukee myös luotujen mutanttien helppoa testaamista NUnit-työkalun avulla ja luo ymmärrettävässä muodossa olevan tulosraportin. Tässä tutkielmassa testaus kuitenkin suoritetaan Unity-editorin kautta, joten näitä ominaisuuksia ei käytetä, vaan mutanttien automaattista testausta ja tulosten raportointia varten kehitettiin oma ratkaisu. CREAM:a käytetään ainoastaan generoimaan mutatoituja versioita aineistona käytettävästä pelistä.

4.2.2 Unity Test Framework

Testausautomaation toteutus tehdään Unityn tarjoaman *Unity Test Frameworkin* (UTF) avulla. Paketista käytetään tutkielman aikana saatavilla olevaa uusinta versiota 2.0.1. Tämä versio ei ole vielä Unityn tarjoama virallinen paketti, vaan ennakkojulkaistu versio, jossa saattaa vielä ilmetä tuntemattomia ongelmia. Ennakkojulkaisu tarjoaa kuitenkin Unityn muutoslokin mukaan muun muassa informatiivisemman käyttöliittymän ja parannuksia testiajon nopeuteen, joista etenkin jälkimmäinen on tutkielman kannalta hyödyllinen ominaisuus ja siksi virallisesti tuetun version sijaan käytetään ennakkojulkaisua.

UTF:ssä testit jaetaan kahteen eri tyyppiin, *Edit Modeen* ja *Play Modeen*. Edit Mode -testit ovat huomattavasti nopeampia ajaa, sillä ne ajetaan vain Unity Editorissa eikä varsinaista peliä tarvitse käynnistää ollenkaan. Edit Mode -testeillä voidaan tehokkaasti suorittaa esimerkiksi perinteisiä yksikkötestejä, joilla varmistetaan yksittäisten metodien toimivuus. Niillä ei kuitenkaan ole mahdollista suorittaa kattavampaa integraatio- tai järjestelmätestausta. Sitä varten on Play Mode -testit, joita myös kaikki tässä tutkielmassa toteutetut testitapaukset ovat. Play Mode -testeissä varsinainen peli käynnistetään ja näin voidaan käyttää hyväksi Unityn tapahtumafunktioita, kuten *Awake*, *Start*, ja *Update*. Tämä mahdollistaa korkeamman

tason testauksen ja todellisen pelaamisen simuloimisen antamalla simuloituja näppäinsyötteitä ja tarkkailemalla pelin tilaa tämän jälkeen. Koska Play Mode -testit ovat huomattavasti hitaampia ajaa, soveltuvat ne Edit Mode -testejä huonommin mutaatiotestaukseen, jossa testiajojen määrät voivat kasvaa hyvin suuriksi. Unityn tapahtumafunktiot ovat kuitenkin niin olennainen osa pelin toimintaa, että kattavaa testausta ei voida toteuttaa ilman niitä ja siksi Play Mode -testejä on käytettävä.

UTF käyttää pohjana *NUnit*-kirjastoa ja testitapausten toteutus muistuttaakin NUnit-testejä hyvin läheisesti. Play Mode -tyypin testit koostuvat tavallisesti simuloiduista syötteistä, joiden jälkeen pelin tilaa verifioidaan *Assert*-lauseiden avulla. Tähän voi käyttää joko NUnitissakin tavallisesti käytettyjä Assert-metodeita tai Unityn lisäämiä UnityAssert-metodeita, jotka ovat monelta osin hyvin samanlaisia. Testitapaukset yleensä myös sisältävät jonkinlaiset *Setup*- ja *Teardown*-osiot, joista Setup suoritetaan ennen jokaista testiä ja Teardown testin jälkeen. Näin voidaan esimerkiksi helposti ladata haluttu skene jokaisen testitapauksen alussa.

4.3 Mutaatiotekniikoiden vertailuperusteet

Mutaatiotekniikoita verrataan niiden generoimien mutanttijoukkojen sekä niiden pohjalta toteutettujen testijoukkojen kautta. Aineistona käytettävää peliä varten toteutettiin kolme testijoukkoa. Testijoukot on esitelty idealtaan seuraavassa listassa ja niiden vaatimuksia käsitellään vielä tarkemmin luvussa 4.4.

1. **Testijoukko 1 – rivikattavuus**

Ensimmäinen testijoukko toteutettiin täyttämään haluttu rivikattavuuden kriteeri niin, että kaikkia pelin ominaisuuksia testattiin vähintään yhdellä testitapauksella. Tätä testijoukkoa käytettiin myös pohjana muissa testijoukoissa, kun mutaatiotestausprosessi aloitettiin.

2. **Testijoukko 2 – perinteinen mutaatio**

Perinteisessä mutaatiossa käytettiin lähes kaikkia mutaatio-operaattoreita, jotka CREAM-työkalussa on valittavissa. Tämä sisälsi kahdeksan standardioperaattoria sekä 16 olio-operaattoria, jotka on kohdennettu simuloimaan virheitä oliopohjaisissa ohjelmointirakenteissa. Testijouk-

ko luotiin täyttämään haluttu mutaatiopisteytyksen kriteeri CREAM:lla generoiduilla mutanteilla.

3. **Testijoukko 3 – Unity-spesifi mutaatio**

Unity-spesifissä mutaatiossa käytettiin luvussa 3.3 esiteltyä joukkoa mutaatio-operaattoreita ja mutatointiprosessi suoritettiin tutkielmaa varten tehdyllä Unity Mutator -työkalulla. Testijoukko luotiin täyttämään haluttu mutaatiopisteytyksen kriteeri Unity Mutatorilla generoiduilla mutanteilla.

Näin saatiin aikaiseksi kaksi testijoukkoa, jotka on toteutettu eri mutaatiotekniikoiden pohjalta sekä alkuperäinen testijoukko, joka on mutaatiopisteytyksen sijaan toteutettu täyttämään rivikattavuuden kriteeri. Tämä mahdollisti vertailun testijoukkojen ja siten mutaatiotekniikoiden välillä. Luvussa 6 testijoukoilla saavutettuja mutaatiopisteytyksiä verrataan ristiin tarkastelemalla kuinka suuren osan perinteisistä mutanteista testijoukko 3 tappaa ja kuinka suuren osan Unity-spesifeistä mutanteista testijoukko 2 tappaa. Myös testitapausten määrä otetaan huomioon joukkojen vertailussa. Tämän vertailun avulla tehdään päätelmiä eri työkalujen ja mutaatio-operaattoreiden avulla parannettujen testijoukkojen kattavuudesta ja näin mutaatiotekniikoiden tehokkuudesta testijoukkojen parantamiseen.

Tuotettujen testijoukkojen laatu yksinään ei ole kuitenkaan riittävä mittari mutaatiotyyppien vertailuun. On syytä muistaa koko mutaatiotestauksen suurin ongelma eli sen huono skaalautuvuus ja tämän pohjasyy eli generoitujen mutanttien suuri määrä. Niinpä kehitettyjen testijoukkojen lisäksi verrataan myös työkaluilla generoituja mutanttijoukkoja. Mutanttien kokonaismäärä ja etenkin ekvivalenttien mutanttien määrä on tärkeää ottaa huomioon, sillä kuten luvussa 2.2 mainittiin, vaatii ekvivalenttien mutanttien analysointi huomattavan määrän manuaalista työtä.

Ekvivalentit mutantit määriteltiin aiemmin mutanteiksi, joiden toiminta on identtistä alkuperäisen ohjelman kanssa, eikä niitä näin ollen voida tappaa testaamalla. Tämän tutkielman kokeellisessa osassa ekvivalenteiksi määritellään myös mutantteja, jotka eivät tätä määritelmää kokonaan täytä. Osa mutanteista saattaa erota toiminnaltaan alkuperäisestä ohjelmasta, mutta niiden tappaminen testaamalla on silti mahdotonta. Tällaisia ovat esimerkiksi jotkin pelin ääniin liittyvät mutaatiot, sillä äänen soittotavasta riippuen voi olla mahdotonta saada

Unitysta tietoa, soitetaanko jollain hetkellä ääntä vai ei.

Etenkin tässä tutkielmassa käytetyn aineiston kohdalla on myös syytä huomata, että vaikka jokin mutaatio näyttäisi tekevän selvän muutoksen ohjelmakoodiin ja ohjelman toimintaan, ei näin välttämättä todellisuudessa ole. Aineistona käytetyn pelin lähdekoodissa on joitakin metodeja, joita ei kutsuta missään tilanteessa ja näin ollen niiden kattaminen tutkielmassa suoritetulla järjestelmätestauksella on mahdotonta. Kaikki tällaisiin kohtiin tehdyt mutaatiot ovat ekvivalentteja.

Näin mutantti- ja testijoukkoja arvioimalla saadaan käsitys Unity-spesifillä mutaatiotestauksella saavutettavasta testauksen kattavuudesta, sekä siihen vaaditusta työn määrästä verrattuna perinteiseen mutaatiotestaukseen. Tämän perusteella voidaan vastata toiseen tutkimuskysymykseen.

4.4 Testijoukkojen vaatimukset

Koska kokeellisessa osassa luotiin testijoukkoja, on myös välttämätöntä määrittää käytetyt kattavuuskriteerit. Koska täydellinen testaus todellakin on mahdotonta, täytyy jonkin säännön mukaan päättää milloin testaus on riittävää. Kattavuuskriteerit tarjoavat tämän säännön (Ammann ja Offutt 2017). Tässä luvussa käydään lävitse ja perustellaan kokeellisessa osassa toteutettuja testijoukkoja varten valitut kattavuuskriteerit. Testijoukko 1 hyväksyttiin rivikattavuuden perusteella ja testijoukot 2 ja 3 taas mutaatiopisteytyksen.

4.4.1 Rivikattavuus

Koodikattavuus on jo pitkään ohjelmistotestauksessa käytössä ollut tekniikka testauksen riittävyyden arvioimiseksi. Jo Miller ja Maloney (1963) totesivat, että ”tarkastuksen aikana ohjelman jokaista osaa täytyy käyttää, jotta sen oikeellisuus voidaan varmistaa.” Tämä on koodikattavuuden perimmäinen ajatus. Nykyään koodikattavuus on vakiintunut tapa testijoukon riittävyyden arviointiin (Ivankovic ym. 2019). Tähän on vaikuttanut muun muassa se, että koodikattavuutta on helppo visualisoida, se on kevyt laskea ja sitä varten on olemassa paljon kaupallisia työkaluja (Petrovic ym. 2021a).

Koodikattavuutta voidaan mitata useallakin eri tavalla, joista tässä tutkielmassa käytetään vain *rivikattavuutta*. Rivikattavuus on yksinkertainen ja suhteellisen helposti saavutettava kattavuuskriteeri. Siinä mitataan testiajon aikana suoritettujen ohjelmakoodin rivien osuutta kaikista riveistä. 100 %:n rivikattavuus siis tarkoittaisi, että ohjelmakoodin jokainen rivi suoritettiin testiajossa. Käytännössä 100 %:n kattavuus on vaikea saavuttaa eikä sitä yleensä edes tavoitella.

Rivikattavuutta läheisesti muistuttava kriteeri on lausekattavuus, jossa rivien suorituksen sijaan tarkastellaan lauseiden suoritusta. Nämä tekniikat tuottavat yleisten tapojen mukaan kirjoitetussa ohjelmakoodissa usein lähes saman tuloksen (Ivankovic ym. 2019) eivätkä monet työkalutkaan erottele niitä toisistaan (Yang, Li ja Weiss 2009). Rivikattavuus ja lausekattavuus ovat yleisesti käytettyjä kattavuuskriteerejä ja Chekam ym. (2017) kuvaavatkin lausekattavuutta laajalti käytettynä minimivaatimuksena testauksessa. Esimerkiksi Googlella taas koodikattavuutta mitataan rivikattavuudella (Ivankovic ym. 2019). Koska Unityn tarjoama Code Coverage -työkalu näyttää nimenomaan testiajon rivikattavuuden, päätettiin sitä käyttää apuna tässä tutkielmassa. Rivikattavuutta käytettiin kriteerinä mutaatiotestauksen pohjana käytetyn testijoukon 1 arvioinnissa.

Koska täydellinen rivikattavuus on vaikea saavuttaa ja sen tuomat hyödyt eivät usein ole lisätyön arvoisia, pidetään todellisissa ohjelmistoprojekteissa kattavuuskriteerinä yleensä jotain pienempää lukua kuin 100 %. Chekam ym. (2017) tutkivat virheenlöytökyvyn ja rivikattavuuden yhteyttä ja tulivat tulokseen, että kun rivikattavuus ylittää 80 %:n rajan, tuo sen nostaminen vain pientä parannusta testijoukon virheenlöytökykyyn. Ivankovic ym. (2019) esittävät graafin Googlen projektien mediaanikattavuudesta, joka on tutkimuksen julkaisuhetkellä ollut noin 85 %. Myös Williams ym. (2001) sanovat useiden kehitysryhmien vaativan 85 %:n kattavuutta. Niinpä tutkielman kokeelliseen osaan päätettiin rivikattavuuskriteeriksi valita 85 %. Näin testijoukko 1 – joka siis luotiin ennen mutaatiotestauksen suoritusta – täytti aineistona käytetylle pelille 85 %:n rivikattavuuden ja tilanteen voitiin ajatella olevan lähellä todellista ohjelmistoprojektia, jossa testauksen apuna ei ole käytetty mutaatiotestausta.

Vaikka koodikattavuutta käytetään yleisesti, on sen hyödyistä testauksen kehittämiseen yhä kiistaa (Ivankovic ym. 2019). Tässäkin tutkimuksessa lähtöoletuksena oli, että rivikattavuuden kriteerin täyttävä testijoukko 1 vaatii huomattavaa parannusta täyttääkseen mutaatiopis-

teytyksen kriteereitä kummallakaan mutaatiotyypeistä.

4.4.2 Mutaatiopisteytys

Testijoukolle 2 kriteerinä toimi sen saavuttama mutaatiopisteytys CREAM:lla generoiduilla mutanteilla ja testijoukolle 3 kriteerinä oli vastaavasti sen saavuttama mutaatiopisteytys Unity Mutatorilla generoiduilla mutanteilla. Mutaatiopisteytyksen kriteeriksi valittiin 100 %. Kuten aiemmin on perusteltu, todellisten ohjelmistoprojektien tapauksessa tämä on harvoin järkevä tavoite tai edes mahdollista saavuttaa, mutta tätä tutkielmaa varten se todettiin parhaaksi ratkaisuksi. Tutkielmassa käytetty aineisto eli pelin lähdekoodi on suhteellisen suppea ja näin manuaalisen työn määrä ei kasva sietämättömän suureksi vaikka testijoukon vaatimuksena onkin kaikkien mutanttien tappaminen. Tämä myös johtaa tasapuolisimpaan vertailuun, sillä muissa tapauksissa mutaatiopisteytykset eivät välttämättä olisi samat testijoukoilla 2 ja 3. Koska yhden uuden testin luominen usein tappaa monta mutanttia, ei kuvion 1 mukainen iteratiivinen testausprosessikaan, jossa jokaisen uuden testitapauksen luomisen jälkeen mutaatiopisteytys lasketaan uudestaan, johda yleensä täsmälleen vaadittuun pisterajaan. Kun rajaksi asetetaan 100 %, molemmat testijoukot ovat siltä osin täysin tasavertaisia.

Korkealle kriteerille mutaatiopisteytyksen suhteen on tukea myös aiempien tutkimuksien mukaan. Chekam ym. (2017) selvittivät eroja eri kattavuustyyppien välillä ja huomasivat, että nimenomaan vahvan mutaatiotestauksen tapauksessa vielä korkeissakin arvoissa mutaatiopisteytyksen nostaminen parantaa huomattavasti testijoukon virheenlöytökykyä. Lausekattavuuden, haarakattavuuden ja heikon mutaatiotestauksen tapauksessa vaikutus taas oli selvästi pienempi. Myös (Papadakis ym. 2018) huomasivat testijoukkojen virheenlöytökyvyssä selkeitä parannuksia juuri korkeimmissa mutaatiopisteytyksissä.

On syytä huomata, että valitun 100 %:n rajan vuoksi testijoukon 1 vertailu testijoukkojen 2 ja 3 kanssa ei ole mielekästä, sillä testijoukon 1 rivikattavuuskriteeri oli vain 85 %. Tutkielman tarkoituksena ei kuitenkaan olekaan tehdä päätelmiä koodikattavuuden avulla luotujen testijoukkojen laadusta verrattuna mutaatiotestauksen avulla luotuihin, vaan vertailla testijoukkoja vain Unity-spesifin ja perinteisen mutatoinnin välillä.

5 Mutaatiotestausprosessin suoritus

Tässä luvussa esitellään yksityiskohtaisesti, millaiset mutanttijoukot CREAM:lla ja Unity Mutatorilla generoitiin sekä millaiset testijoukot niiden avulla tuotettiin. Mutanttijoukot esitetään mutaatio-operaattoreiden ja ekvivalenttien mutanttien suhteen kuvioissa ja testijoukot testitapauskohtaisesti taulukoissa.

5.1 Yleistä

Huomionarvoista on, että ohjelmistokehitysprosessin vaiheet toteutuivat tutkielmassa nurinkurisesti tutkimusasetelman vuoksi. Perinteisessä ohjelmistokehitysprosessissa kaikki alkaa vaatimusmäärittelystä, jonka pohjalta toteutetaan muun muassa ohjelmiston koodaus ja testaus (Royce 1987). Koska tutkielman tilanteessa pelin toteutus on jo valmiina, täytyy testijoukolla verifioitavat vaatimukset johtaakin toteutuksen mukaan. Testijoukot siis luotiin olettaen, että sen hetkinen ohjelma toimi virheettömästi ja halutulla tavalla.

Tutkimuksen toistettavuuden kannalta on myös syytä huomioida, että testijoukkojen luonti ja parannus mutaatiotestauksen avulla ei ole eksakti prosessi. Kuten jo DeMillo, Lipton ja Sayward (1978) artikkelin otsikossaan sanoivat, sen tarkoitus on oikeastaan vain tarjota vinkkejä testidatan valintaan. Mutaatiotestauksen avulla tuotetut testijoukot siis riippuvat vahvasti testauksen suorittavasta henkilöstä. Eri testaajat voivat eloonjääneitä mutantteja analysoimalla päätyä hieman erilaisiin testitapauksiin ja usein testaaja saa myös ideoita, jotka eivät välttämättä liity juuri kyseisen mutantin tappamiseen.

Tämän tutkimuksen tapauksessa mutatoinnin avulla luodut testitapaukset pyrittiin – hieman intuition vastaisesti – jättämään suppeiksi niin, että niiden avulla tapetaan vain eloonjäänyt mutantti, jonka analysointi johti kyseiseen testitapaukseen. Näin testitapausten luonti säilyi johdonmukaisena eikä riippuvaisena siitä, kuinka paljon testitapauksia testaaja milläkin hetkellä pystyi keksimään yhden eloonjääneen mutantin perusteella.

Ominaisuus	Testitapauksen nimi	Testitapauksen kuvaus
Liikkuminen	Move	Pelaajahahmo liikkuu molempiin suuntiin
Hyppy	JumpFullHeight	Pelaajahahmo hyppää odotetulle korkeudelle
	InterruptJump	Jos hyppynäppäimen päästää irti kesken hypyn, hyppy keskeytyy
	NoDoubleJump	Hyppyä ei voi aloittaa ennen kuin pelaajahahmo on maassa
Kuolema	KilledByEnemy	Pelaaja joutuu aloittamaan alusta jos osuu viholliseen sivusuunnasta
	KilledByMapBounds	Pelaaja joutuu aloittamaan alusta jos tippuu ulos kentästä
Vihollisen tuhoaminen	KillEnemy	Vihollinen poistetaan kentästä jos pelaaja tippuu sen päälle ylhäältä
Merkin kerääminen	CollectToken	Merkki poistetaan kentästä pelaajan osuttua siihen
Pelin voittaminen	ReachGoal	Pelaajahahmoa ei voi liikuttaa enää pelaajan koskettua maaliin
Valikon avaaminen	PauseGame	Peli pysähtyy kun pelaaja avaa valikon
	OpenMainMenu	Valikkoikkuna tulee näkyviin kun pelaaja avaa valikon

Taulukko 5. Testijoukko 1

5.2 Testijoukko 1

Testijoukko 1 luotiin täyttämään määritelty rivikattavuuden kriteeri aiemmin lueteltujen pelin ominaisuuksien mukaisesti. Ensin luotiin yksi testitapaus jokaista ominaisuutta kohden, minkä jälkeen testijoukon rivikattavuuden perusteella testitapauksia luotiin niin, että koodirivejä katettaisiin mahdollisimman paljon lisää. Jokaisen uuden testitapauksen jälkeen rivikattavuus mitattiin uudestaan ja lopulta rivikattavuudeksi saatiin 11:llä testitapauksella 87,7 %. Testijoukko 1 on esitelty kokonaisuudessaan taulukossa 5.

Tosiasiassa koko pelin lähdekoodiin tällä testijoukolla saavutetaan vain 80,1 %:n rivikatta-

vuus. Laskusta kuitenkin jätettiin pois kaksi skriptiä: Fuzzy.cs ja AnimatedTile.cs. Fuzzy tarjoaa erinäisiä metodeja lukujen satunnaistamiseen, mutta näitä metodeja ei käytetä mistään kohtaa muuta ohjelmakoodia. AnimatedTile taas liittyy Unityn editorissa tapahtuvaan kentän luontiin, eikä sen metodeja kutsuta pelin aikana lainkaan. Näin ollen näitä skriptejä oli mahdoton kattaa pelissä tapahtuvalla testauksella eikä niitä otettu huomioon rivikattavuuden laskussa.

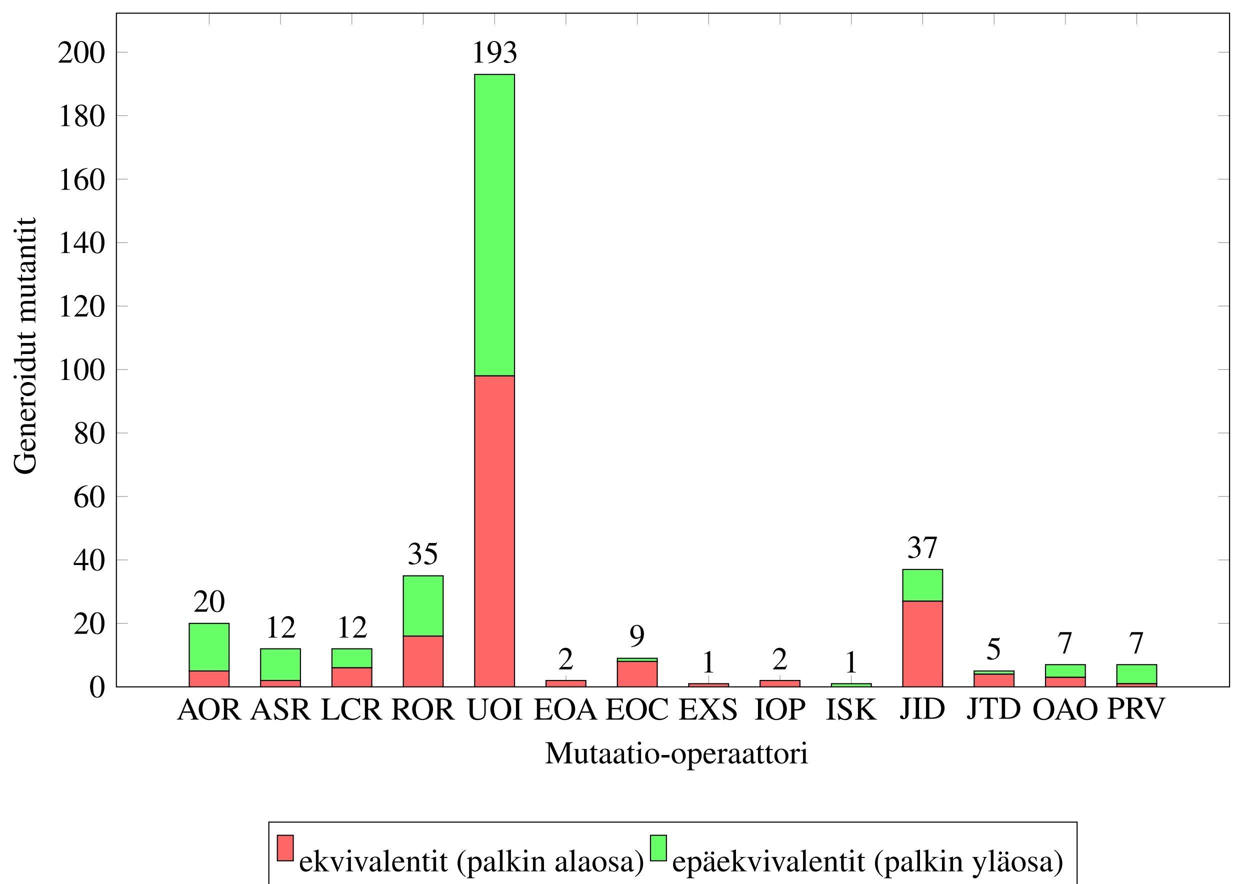
5.3 Mutanttien generointi

CREAM:sta valittiin mutanttien generointia varten käyttöön kaikki perinteiset mutaatio-operaattorit sekä oliopohjaisista operaattoreista kaikki paitsi IOD ja IOK. Kyseiset operaattorit jätettiin pois käytöstä niiden aiheuttamien teknisten ongelmien vuoksi, sillä CREAM kaatui aina näitä operaattoreita käytettäessä. Arvioitiin, että IOD- ja IOK-operaattorit eivät olisi generoineet lähdekoodiin huomattavaa määrää mutantteja. Valituilla operaattoreilla mutantteja luotiin lopulta 343 kappaletta. Manuaalisen analyysin jälkeen näistä 175 todettiin ekvivalenteiksi ottaen huomioon tämän tutkielman kokeellista osaa varten luvussa 4.3 täsmennettyä määritelmää. Useat mutantit olivat myös valmiiksi kuolleita (*engl. stillborn*) eli ne eivät edes kääntyneet ajettavaksi ohjelmaksi. Näistä suurin osa oli UOI-operaattorilla luotuja mutantteja, joissa liukulukuja yritettiin mutatoida bittioperaattoreilla.

Kuviossa 4 on nähtävillä generoituneiden mutanttien määrät kullakin mutaatio-operaattorilla. Operaattorit, joilla mutantteja ei generoitunut lainkaan on jätetty kuviosta pois.

Unity Mutatorista käytössä oli kaikki taulukossa 3 esitellyt mutaatio-operaattorit. Mutantteja generoitui huomattavasti vähemmän kuin CREAM:lla, kokonaisuudessaan vain 42 kappaletta. Näistä lähes kaikki generoituivat Vector-luokkiin kohdentuvista mutaatio-operaattoreista ja suurin osa muista operaattoreista eivät generoineet ainoatakaan mutanttia.

Tälle voidaan pohtia monia syitä. Aineistona käytetty peli oli hyvin yksinkertainen ja sillä oli suppea lähdekoodi. Siitä johtuen on selvää, että eri luokkia ja ohjelmointirakenteita ei voi olla käytetty niin paljon kuin laajemmassa pelissä olisi mahdollista. Suuri osa ohjelmakoodista liittyi hahmojen liikkumiseen ja siksi Vector-luokkia mutatoivat operaattorit olivat suurimmassa roolissa. Monet Unityn ominaisuudet, joihin valitut mutaatio-operaattoritkin keskit-



Kuvio 4. CREAM:lla generoitujen mutanttien jakauma mutaatio-operaattoreittain

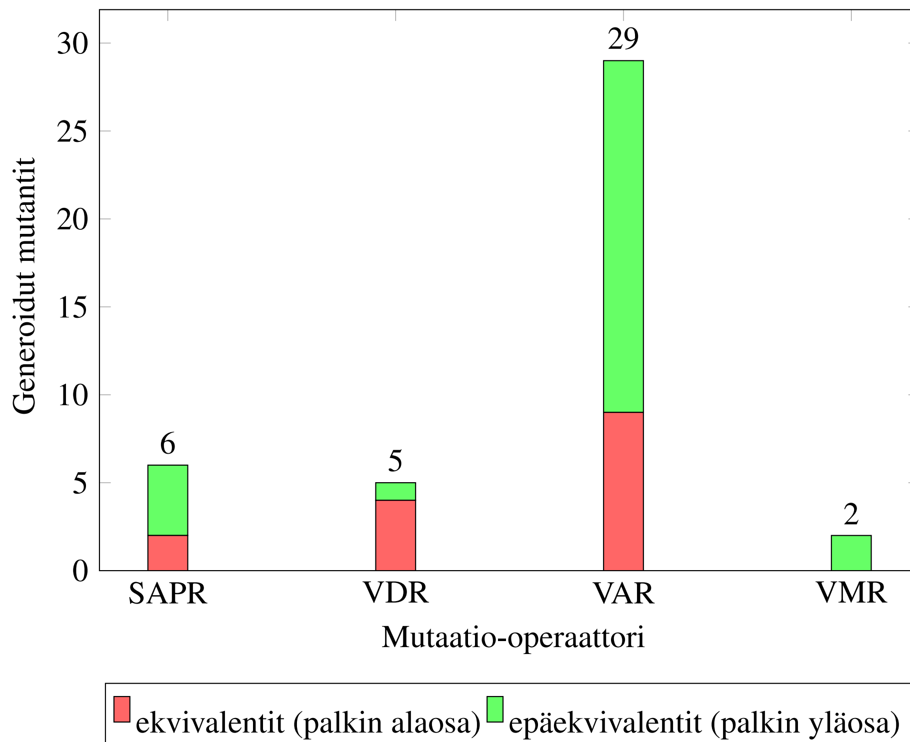
tyvät, ovat myös vähemmän tarpeellisia pienissä peleissä. Esimerkiksi LoadScene-metodia, jota LSPR-operaattori mutatoi, ei tässä pelissä käytetty lainkaan, sillä koko peli koostui vain yhdestä skenestä.

Myös pelin arkkitehtuuriratkaisu vaikuttaa hyödyllisiin operaattoreihin. Esimerkiksi AddListener- ja Invoke-metodeja mutatoivat ALMR- ja IVPR-operaattorit olisivat huomattavasti tärkeämpiä, jos pelin toiminta ja skriptien välinen kommunikointi olisi toteutettu tapahtumapohjaisesti. Näin yksinkertaisessa pelissä se olisi kuitenkin ehkä tarpeettoman kompleksinen ratkaisu. Toinen projektissa tehtävä ratkaisu, joka vaikuttaisi moniin valituista operaattoreista, on Unityn editorin käytön ja ohjelmakoodin kirjoittamisen välinen tasapaino. Useat asiat Unityssa voi tehdä joko editorin kautta kirjoittamatta lainkaan koodia tai skriptien kautta. Tällainen on vaikkapa skriptien sisältämät viittaukset peliobjekteihin. Viittaukset voi asettaa joko editorista tai koodista esimerkiksi Find- tai GetChild-metodeja käyttäen. Näitä metodeja mutatoivat esitellyt operaattorit FPR ja GCPR. Tässä pelissä metodeja ei kuitenkaan käytetty vaan kaikki skriptien data oli asetettu editorin kautta. Isommissa projekteissa se ei yleensä ole mahdollista vaan viittauksia täytyy muuttaa ajon aikana skripteistä.

On myös mahdollista, että mutaatio-operaattoreiden joukko on suunniteltu huonosti. Se saattaa sisältää lähes turhia operaattoreita, jotka vain harvoin generoivat mutantteja ja siitä voi puuttua hyödyllisiä operaattoreita, joiden avulla niitä generoitaisiin useammin.

Näiden syiden vuoksi tutkimus olisikin syytä toistaa myös useammilla ja laajemmilla peleillä sekä erilaisilla mutaatio-operaattoreiden joukoilla. Samankaltaista operaattoreiden käytön epätasaisuutta on tosin havaittavissa myös CREAM:lla generoituja mutantteja tarkastellessa. Käytössä oli 16 oliopohjaista operaattoria ja 8 perinteistä operaattoria. Silti vain viidesosa generoiduista mutanteista tuli oliopohjaisista operaattoreista ja seitsemällä oliopohjaisella operaattorilla ei mutantteja generoitunut lainkaan. Kenties onkin odotettavissa, että mitä spesifimpejä mutaatio-operaattoreita käytetään, sitä suuremmalla todennäköisyydellä joillakin niistä mutantteja ei generoidu lainkaan.

Kuviossa 5 on esitetty Unity Mutatorilla generoituneiden mutanttien määrät kullakin operaattorilla. Operaattorit, joilla mutantteja ei generoitunut yhtään kappaletta on jätetty kuviosta pois.



Kuvio 5. Unity Mutatorilla generoitujen mutanttien jakauma mutaatio-operaattoreittain

5.4 Testijoukko 2

Testijoukko 2 luotiin täyttämään päätetty 100 %:n mutaatiopisteytys CREAM:lla generoituun mutanttijoukkoon. Alkuperäinen testijoukko 1 tappoi kaikista generoiduista 343:sta mutantista 119, jolloin epäekvivalenteista mutanteista eloon jäi 49. Tästä laskettuna testijoukko 1 saavutti 70,8 %:n mutaatiopisteytyksen CREAM:lla generoituun mutanttijoukkoon. 100 %:n pisteytyksen saavuttamiseksi luotiin 11 uutta testitapausta ja korjattiin yhtä testijoukon 1 tapausta, joka huomattiin puutteelliseksi.

CREAM-mutatoinnin avulla luodut uudet testitapaukset on esitelty taulukossa 6. Näistä sekä perustana käytetystä testijoukko 1:stä muodostui testijoukko 2. Tällä testijoukolla saavutettiin 100 %:n mutaatiopisteytys ja joukko todettiin valmiiksi.

Testitapauksen nimi	Testitapauksen kuvaus
SpawnWithHealth	Pelaajahahmolla on täydet elämäpisteet syntymisen jälkeen
PlayerZeroHealthAfterDeath	Pelaajahahmolla on 0 elämäpistettä kuolemisen jälkeen
PlayerRunAudio	Pelaajan juostessa soitetaan ääni oikein
IdleEnemyStaysStill	*Idle*-tilaan määritellyt vihollishahmot eivät liiku
EnemyVisitsPatrolPathEnds	Liikkuva vihollishahmo käy polkunsa molemmissa päissä
FlipPlayerSprite	Pelaajahahmon sprite kääntyy liikkumissuunnan mukaan
FlipEnemySprite	Vihollishahmon sprite kääntyy liikkumissuunnan mukaan
CollectMultipleTokens	Pelaaja voi kerätä monta merkkiä hyvin lyhyen ajan sisään
TokenCollectAnimation	Merkin keräämisanimaatio näytetään kokonaan
TokenAnimationCorrectSpeed	Merkeissä käytetty animaatio näytetään oikealla nopeudella
OpenMenuTabs	Valikkoikkunassa on mahdollista vaihtaa välilehtiä

Taulukko 6. Testijoukkoon 2 lisätyt testitapaukset

5.5 Testijoukko 3

Testijoukolta 3 vaadittiin 100 %:n mutaatiopisteytys Unity Mutatorilla generoituun mutanttijoukkoon. Testijoukko 1 saavutti siihen vain 48,1 %:n mutaatiopisteytyksen, tappaen kaikista 42:sta mutantista 13 ja jättäen eloon 14 epäekvivalenttia mutanttia. 100 %:n mutaatiopisteytys saavutettiin luomalla 9 uutta testitapausta ja korjaamalla yhtä alkuperäisistä testitapauksista – samaa jota korjattiin testijoukon 2 luonnissa. Myös kokonaan uusista testitapauksista kolme oli samoja kuin testijoukossa 2. Mutaatioiden seuraukset pelin toimintaan olivat lopulta siis hyvinkin samanlaisia joissain tapauksissa, vaikka mutantteihin tehdyt koodinmuutokset itsessään erosivat.

Uudet testitapaukset on esitelty taulukossa 7. Nämä uudet tapaukset yhdistettynä testijoukon 1 testitapauksiin koostivat testijoukon 3. Tämä testijoukko täytti 100 %:n mutaatiopisteytyksen ja se todettiin valmiiksi.

Näin luotiin kolme testijoukkoa kolmella eri tekniikalla – koodikattavuuden, CREAM-mutatoinnin ja Unity Mutator -mutatoinnin avulla. Koodikattavuudella luotu testijoukko 1 on selvästi vähiten kattavin, sillä se toimi vain pohjana testijoukoille 2 ja 3. Lisäksi se toteutet-

Testitapauksen nimi	Testitapauksen kuvaus
PlayerRunAnimation	Pelaajan juostessa näytetään oikea animaatio
PlayerStopRunWhenColliding	Pelaajan vaakaliike pysähtyy törmätessä johonkin
PlayerStopJumpWhenColliding	Pelaajan pystyliike pysähtyy törmätessä johonkin
PlayerBounceAfterEnemyKill	Pelaaja pompahtaa ylöspäin tappaessaan vihollisen
PatrollingEnemyNoJump	Vihollinen ei hypi jos se on liikkeessä
EnemyPatrolSpeed	Vihollinen liikkuu oikealla nopeudella
FlipPlayerSprite	Pelaajahahmon sprite kääntyy liikkumissuunnan mukaan
FlipEnemySprite	Vihollishahmon sprite kääntyy liikkumissuunnan mukaan
OpenMenuTabs	Valikkoikkunassa on mahdollista vaihtaa välilehtiä

Taulukko 7. Testijoukkoon 3 lisätyt testitapaukset

tiin kevyemmällä kriteerillä, sillä rivikattavuudeksi vaadittiin vain 85 %, kun taas mutaatiotestauksen avulla kehitetyille testijoukoille vaatimus oli 100 %:n mutaatiopisteytys. Niinpä tekniikkojen vertailussa keskityttiin etenkin testijoukkojen 2 ja 3 välisiin tuloksiin.

6 Tulokset ja pohdinta

Tässä luvussa esitetään eri mutaatiotekniikoilla tuotettujen testijoukkojen tulokset kullakin kattavuuskriteerillä. Tekniikoita verrataan tämän ja aiemmassa luvussa esitettyjen mutantti- ja testijoukkojen perusteella ja pohditaan, mistä erot tuloksissa johtuvat ja mitä päätelmiä niistä voi tehdä. Lisäksi tärkeänä osana esitetään jatkotutkimusaiheita, sillä täytyy pitää mielessä, että tutkielman tulokset todellakin ovat vain alustavia havaintoja.

6.1 Mutaatiotekniikoiden vertailu

Testijoukkojen – ja siten Unity-spesifin ja perinteisen mutatoinnin – vertailu toteutettiin ajamalla testijoukot ristiin mutanttijoukkojen kanssa. Siis testijoukko 2 ajettiin Unity Mutatorilla generoiduilla mutanteilla, kun taas testijoukko 3 ajettiin CREAM:lla generoiduilla mutanteilla. Vertailun tulokset on esitelty taulukossa 8. Testijoukko 2 saavutti 85,2 %:n mutaatiopisteytyksen Unity Mutatorin mutanteilla. Testijoukon 3 pisteytys CREAM-mutanteilla oli 77,4 %. Testijoukko 2 myös sisälsi kaksi testitapausta enemmän. Näyttäisi siis, että tässä kokeessa testijoukko 3 suoriutui paremmin kuin testijoukko 2.

Yleistettyjä johtopäätelmiä perinteisen mutaatiotestauksen – tai edes testijoukon 3 – paremmuudesta voi kuitenkin tehdä vain hyvin varovasti näiden tulosten perusteella. Ristiin vertailu ei ole luotettavin metodi testijoukkojen arvioimiseen ja mutanttijoukkojen suuri kokoero vaikeuttaa myös tulosten tulkitsemista. Tulevissa tutkimuksissa vertailun voisi tehdä luotettavammaksi esimerkiksi käyttämällä kolmatta mutantti- tai virhejoukkoa, johon molempia testijoukkoja verrattaisiin. Tämä kolmas joukko voisi olla todelliset ohjelmointivirheet, joita aineistona käytettyyn peliin on joskus tehty.

Testijoukko	Rivikattavuus	CREAM	Unity Mutator
Testijoukko 1	87,7 %	70,8 %	48,2 %
Testijoukko 2	87,7 %	100 %	85,2 %
Testijoukko 3	87,7 %	77,4 %	100 %

Taulukko 8. Testijoukkojen tulosvertailu

Mielenkiintoista on huomata, että testijoukko 1 saavutti selvästi huonomman tuloksen Unity Mutatorin mutanttijoukolla kuin CREAM:n. Tämän olisi voinut ajatella tarkoittavan, että Unity Mutator luo vaikeammin tapettavia mutantteja ja siten sen avulla luotu testijoukko olisi kattavampi. Se ei kuitenkaan pitänyt paikkaansa, vaan prosenttilukujen perusteella tehty oletus osoittautuukin vääräksi generoitujen mutanttien määrän takia. Vaikka testijoukko 1 tappoikin prosentuaalisesti suuremman osan CREAM-mutantteja jäi niitä absoluuttisena määränä kuitenkin huomattavasti enemmän eloon – 49 kappaletta Unity Mutatorin 14:n sijaan.

Toisena sivuhuomiona mainittakoon rivikattavuuden pysyvyys. On erikoista, että molemmilla selvästi testijoukkoa 1 kattavammilla joukoilla rivikattavuus pysyi siltikin samana. Suurin syy tähän on luultavimminkin se, että ohjelmakoodi on yhä sisältänyt rivejä, joita oli mahdoton kattaa pelin sisäisellä järjestelmätestauksella. Vaikka rivikattavuuden laskusta jätettiinkin luvun 5.2 mukaisesti pois kaksi kokonaista skriptiä, joita pelissä ei todellisuudessa käytetty ollenkaan, muutkin skriptit sisälsivät tällaisia osia. Tällaisiin kohtiin generoidut mutaatiot on myös analysoitu ekvivalenteiksi. Niinpä todellinen rivikattavuus on jo testijoukossa 1 ollut luultavasti lähelle 100 %, kun laskusta jätetään pois kaikki rivit, joita toteutetun tyyppisellä testauksella oli mahdoton kattaa.

Kun testijoukkojen tappamia ja eloon jättämiä mutantteja tarkastellaan, huomataan helposti puutteita Unity-spesifiin mutaatioon käytettävässä mutaatio-operaattoreiden joukossa. Testijoukko 3 jätti CREAM-mutanteista eloon 38 kappaletta, joista 11 liittyi hahmojen animaatioihin ja 15 ääniin. Nämä ovat Unityn ominaisuuksia, joita ei mutatoitu käytetyillä operaattoreilla ollenkaan. Niinpä tämän tuloksen perusteella voidaankin ehdottaa kahta lisäystä mutaatio-operaattoreiden joukkoon: ATR (*Animation trigger replacement*) ja APR (*Audio parameter replacement*). Näidenkään käyttö ei tosin olisi auttanut tappamaan kaikkia animaatioihin ja ääniin liittyviä mutantteja, sillä animaatioita ja ääniä voi toteuttaa myös ilman Unityn tarjoamia luokkia, kuten aineistona käytetyssä pelissä osittain oli tehty. Tällöin geneerisemmät mutaatio-operaattorit ovat ainoa tapa kattaa näiden ominaisuuksien mutatointi.

Loput CREAM:n generoimista mutanteista, jotka jäivät testijoukolla 3 eloon, liittyivät pelaajan elämäpisteiden vähennykseen ja lisäykseen kuolemisen ja syntymisen yhteydessä. Elämäpisteiden toiminta oli toteutettu ilman Unity-spesifejä rakenteita eikä niitä esimerkik-

si näytetty peliruudulla mitenkään, joten on vaikea suunnitella mutaatio-operaattoria, jonka avulla vastaavanlaisia mutaatioita generoitaisiin myös Unity Mutatorilla. Kuitenkin esimerkiksi yleinen GetComponent-metodia mutatoiva operaattori – jonka lisäämistä tähänkin tutkielmaan harkittiin – olisi saattanut auttaa tappamaan nämä mutaatiot. Niinpä voidaan ehdottaa kolmatta uutta mutaatio-operaattoria GCOR (*GetComponent object replacement*). Tämä operaattori vaihtaisi objektin, jolle GetComponent-metodia kutsutaan johonkin toiseen objektiin, jolla on myös olemassa vastaava komponentti. Tällaisen operaattorin tekninen toteutus olisi kuitenkin huomattavasti monimutkaisempi kuin minkään muun Unity Mutatorissa toteutetun operaattorin ja siksi se jätettiin lopulta pois.

Unity Mutatorin generoimat mutantit, jotka jäivät testijoukolla 2 eloon liittyivät pitkälti pelihahmojen liikkeeseen. Liikkumisen toteutuksessa käytettiin paljon Vector-luokkien ominaisuuksia ja siksi Unity Mutator generoi siihen liittyen suuren määrän mutantteja. Perinteisillä operaattoreilla ei taas päästy niin hyvin kiinni samoihin ohjelmakoodin osiin etenkään, koska monet peliobjektien kentistä olivat projektissa määritelty Unity-editorin kautta.

Kuten luvussa 4.3 mainittiin, pelkkä saavutettujen testijoukkojen kattavuus tai virheenlöytökyky ei ole järkevä mittari mutaatiotekniikoiden vertailuun. Koska mutaatiotestaus on tasapainottelua parhaimman mahdollisen testijoukon ja nopeimman mahdollisen prosessin välillä, myös vaaditut resurssit – etenkin manuaalisen työn määrä – on otettava huomioon. CREAM loi mutantteja kaikkiaan 343 kappaletta, joista jopa 51 % oli ekvivalentteja. Unity Mutator taas loi mutantteja vain 42 kappaletta ja niistä ekvivalentteja oli 36 %. CREAM:lla suoritetussa mutaatiotestausprosessissa pelkkä ekvivalenttien mutanttien tunnistaminen vaati siis yli kymmenkertaisen määrän manuaalista työtä, jos oletetaan, että mutanttikohtainen työn määrä ei eroa eri tekniikoilla.

Monet CREAM:n generoimista ekvivalenteista mutanteista johtuivat nimenomaan Unity-projektille tyypillisistä seikoista, mikä tukee ajatusta siitä, että Unity-spesifin työkalun ja mutaatio-operaattoreiden käyttö voisi olla perinteisiä tekniikoita tehokkaampaa. Selvin esimerkki tästä ovat ekvivalentit mutantit, jotka johtuvat aineistoprojektissa käytetystä tavasta alustaa kentät – siis suoraan luokan sisällä esitellyt muuttujat. Kentille oli asetettu alkuarvo sekä lähdekoodista että Unityn editorista, mikä onkin melko usein käytetty tapa Unity-projekteissa. Tällöin editorista asetetut arvot ylikirjoittavat lähdekoodissa asetetut, mutta mi-

käli kentät esimerkiksi vaihdetaan ohjelmakoodista yksityiseksi – mikä estää niiden asettamisen editorista – siirrytään käyttämään koodissa alustettua arvoa. Tämä tapa aiheuttaa kuitenkin paljon ekvivalentteja mutantteja perinteisillä mutaatio-operaattoreilla, kun mutaatioissa alustettujen kenttien etumerkkejä vaihdetaan ohjelmakoodissa tai niiden alustus poistetaan kokonaan. Kuudesosa kaikista CREAM:n generoimista ekvivalenteista mutanteista johtui pelkästään tällaisista mutaatioista.

Toisaalta suuri osa CREAM:n generoimista ekvivalenteista mutanteista ei liittynyt mitenkään siihen, että mutaatiotestausta suoritettiin nimenomaan Unity-projektille ja työkalua kehittämällä määrää voitaisiin laskea melko helpostikin myös ilman Unitylle tyypillisten ominaisuuksien huomioon ottamista. Esimerkiksi JID-operaattori luo mutaatioita poistamalla kentälle alustetun arvon ohjelmakoodista. Se tekee näin, vaikka arvo olisi alustettu samaksi kuin kentän tyypin oletusarvo. Tämä johtaa aina ekvivalenttiin mutanttiin, joka olisi helppo poistaa joukosta automaattisesti, tarkastamalla onko kentälle asetettu tyypin oletusarvo ja jättämällä mutaatio tekemättä jos näin on. On siis johtopäätöksiä tehdessä tärkeää huomata, että tutkielmassa käytetyt perinteisen mutatoinnin työkalutkaan eivät ole täydellisiä. Vaikka tavoitteena onkin verrata perinteisen mutaatiotestauksen ja Unity-spesifin mutaatiotestauksen eroja, todellisuudessa verrataan vain CREAM:n ja Unity Mutatorin välisiä eroja.

Nopeamman analyysivaiheen lisäksi mutanttien generointiprosessi vei myös huomattavasti vähemmän aikaa Unity Mutatorilla – myös kun otetaan huomioon mutanttien vähäisempi määrä. Tämä johtuu osittain luvussa 3.4 kuvailluista eroista CREAM:n ja Unity Mutatorin toimintaperiaatteissa. Unity Mutator korvaa jokaisen mutantin kohdalla vain mutatoidun tiedoston ja ajaa sen jälkeen testijoukon tällä. CREAM taas kopioi jokaiseen mutanttiin koko Unity-projektin, joka saattaa olla kooltaan sadoista megatavuista useisiin gigatavuihin. Unity Mutatorin tavassa kuitenkin joudutaan tinkimään muista ominaisuuksista. Sitä ei muutenkaan voi sinällään pitää Unity-spesifin mutaatiotestauksen parempana puolena, sillä samaa tekniikkaa voitaisiin käyttää myös perinteisessä mutatoinnissa.

Joka tapauksessa tämä nopeampi tapa on mahdollista vain koska työkalussa on otettu huomioon Unity-testauksen työnkulku ja siihen käytettävät tekniikat. Siinä mielessä se tukee ajatusta ainakin Unityn erityisesti huomioon ottavasta työkalusta – olivat mutaatio-operaattorit mitä hyvänsä.

Tulosten perusteella on vaikea yksiselitteisesti sanoa, kumpi mutaatiotyypeistä suoriutui tässä tutkimuksessa paremmin. Unity-spesifi mutaatiotestaus oli mutanttien vähäisen määrän vuoksi selvästi kevyempi ja nopeampi prosessi. Testijoukkojen vertailun perusteella voisi kuitenkin varovasti arvioida, että perinteisen mutaatiotestauksen avulla saavutettu testijoukko oli kattavampi. Projektin prioriteeteista riippuen joko nopea, mutta epätarkka mutaatioprosessi tai hidas mutta perusteellisempi mutaatioprosessi voi olla parempi valinta.

Kumpikaan testijoukko ei kuitenkaan yltänyt hyvin korkeaan mutaatiopisteytykseen toisella mutanttijoukolla. Tämä tarkoittaa, että myös Unity-spesifillä mutatoinnilla luotiin mutantteja, joita vastaavia ei luotu perinteisellä tekniikalla. Toki perinteisellä mutatoinnilla luotiin vielä enemmän mutantteja, joita vastaavia Unity-spesifeillä operaattoreilla taas ei generoitunut.

Siten tärkein tulos tutkimuksesta ehkä onkin, että selvästi paras testijoukko Unity-peleille saadaan yhdistämällä perinteinen ja Unity-spesifi mutaatiotestaus. Tässä mielessä toiseen tutkimuskysymykseen voidaan vastata, että Unity-spesifillä mutaatiotestauksella on potentiaalia saavuttaa hyötyjä verrattuna perinteiseen mutaatiotestaukseen. Tämä vastaus ei kuitenkaan tarkoita, että Unity-spesifillä mutatoinnilla voitaisiin korvata perinteinen, vaan että sillä voidaan täydentää sitä parhaan tuloksen saavuttamiseksi.

6.2 Tutkimuksen puutteet ja jatkotutkimus

Tutkielman toteuttamisessa oli paljon työtä, koska aiempaa tutkimusta aiheeseen liittyen ei ollut lähes ollenkaan. Käytännön toteutus kuten työkalujen kehittäminen ja alkuperäisen testijoukon tekeminen veivät paljon aikaa kuten myös – täysin odotetusti – varsinainen mutaatiotestausprosessi. Kuten tutkielman otsikossa sanotaan, kyseessä olikin vain alustava katsaus Unity-spesifiin mutaatiotestaukseen. Laajempaa empiiristä koetta ei ollut tutkielman puitteissa mahdollista toteuttaa, vaikka se olisikin yleistettävämpien tulosten kannalta ollut tärkeää.

Aineistona käytetty peli oli hyvin yksinkertainen ja jatkossa olisi syytä tutkia muuttuvatko tulokset, jos aineistona käytetään laajempia ja tosielämän projekteihin paremmin vertautuvia pelejä pienten tutoriaalipelien sijaan. Vertailua olisi myös hyvä tehdä useampien eri tyyppis-

ten pelien välillä. Ohjelmointirakenteet riippuvat huomattavasti pelin tyypistä ja kehittäjästä ja nytkin aineistona käytetyssä pelissä monilla mutaatio-operaattoreilla – sekä Unity-spesifeillä että perinteisillä – generoitiin hyvin vähän tai ei lainkaan mutantteja.

Mutaatio-operaattorit ovatkin mahdollinen kehityskohde, jota pohtimalla myös Unity-spesifillä mutaatiotestauksella saavutetut tulokset voisivat olla paremmat. Tutkielmassa käytettiin lopulta vain 17:ää Unity-spesifiä mutaatio-operaattoria, kun taas vertailtavassa työkalussa CREAM:ssa oli käytössä 24 operaattoria. Lisäksi jo tämän tutkielman tuloksissa löydettiin hyvin selkeitä puutteita käytetyistä Unity-spesifeistä mutaatio-operaattoreista. Unity Mutator myös generoi huomattavasti vähemmän mutantteja kuin CREAM, joten siinäkin mielessä mutaatio-operaattoreita voisi vielä lisätä.

Olisi siis syytä tarkemmin tutkia, millaisia operaattoreita Unity-spesifiin mutaatioon voisi lisätä ja mitkä Unity Mutatoriin toteutetuista taas ovat mahdollisesti tarpeettomia. Toimivimman ratkaisun selvittämiseksi jatkotutkimuksissa eri operaattorijoukoilla saatuja tuloksia tulisi vertailla keskenään. Koska tämänkin tutkimuksen tulosten perusteella on syytä uskoa, että lopullisessa ratkaisussa tehokkainta voisi olla käyttää monentyyppisiä operaattoreita yhdessä, pitäisi vertailtavien operaattorijoukkojen sisältää niin Unity-spesifejä kuin perinteisiäkin operaattoreita.

Lisäksi koska Unity-spesifi mutaatioprosessi oli huomattavasti nopeampi, mutta sen avulla tuotettu testijoukko myös suppeampi, olisi mielenkiintoista tietää, miten minimoitu versio perinteisestä mutanttijoukosta vertautuisi Unity-spesifiin mutanttijoukkoon. Mutanttien määrän vähentämiseen on olemassa erinäisiä tekniikoita, joita perinteiseen mutanttijoukkoon voisi soveltaa ja tarkastella, tuottaisiko se edelleen kattavamman testijoukon.

Tutkielmaa varten kehitetty Unity Mutator jätettiin hyvin alkeelliseksi työkaluksi rajallisten resurssien vuoksi. Työkalu vaatisi huomattavia parannuksia ennen sen suosittelua yleiseen käyttöön. Se kuitenkin täytti tehtävänsä tämän tutkielman tarpeisiin ja mahdollisti alustavien tulosten hankkimisen Unity-spesifistä mutaatiosta. Jatkossa olisi mahdollista luoda täsmällisemmin toimivia ja myös nopeammin tehtävästä suoriutuvia ohjelmia.

Jatkotutkimusta ajatellen seuraava askel olisikin kehittää Unity Mutatoria kehittyneempi työkalu Unity-spesifiin mutaatiotestaukseen, joka olisi mahdollista laittaa yleisesti saataville.

Tämä helpottaisi ja nopeuttaisi empiiristen kokeiden järjestämistä ja työkalun saattaminen myös muuhun kuin akateemiseen käyttöön johtaisi varmasti hyödylliseen dataan. Valitettavasti mutaatiotestaus peliteollisuudessa on edelleen hyvin harvinaista ainakin sitä käsittelevien tieteellisten artikkelien määrän perusteella.

Jatkotutkimuksissa vertailussa pitäisi myös käyttää useampia työkaluja pelkän CREAM:n sijasta, jotta johtopäätöksiä voisi luotettavammin tehdä perinteisestä mutaatiotestauksesta kokonaisuudessaan. CREAM:n heikko suoriutuminen jossain asiassa ei välttämättä tarkoita, että vika olisi pohjimmiltaan perinteisessä mutaatiotestauksessa itsessään, vaan ongelma voi olla pelkästään työkalussa, kuten edellisessä luvussa jo huomattiinkin.

7 Yhteenveto

Tutkielmassa etsittiin vastausta kahteen tutkimuskysymykseen

1. Miten Unity-spesifiä mutaatiotestausta olisi mahdollista toteuttaa?
2. Olisiko Unity-spesifillä mutaatiotestauksella potentiaalia saavuttaa hyötyjä verrattuna perinteiseen mutaatiotestaukseen mutaatiotestaukseen?

Ensimmäiseen kysymykseen paneuduttiin määrittelemällä ensin Unity-peleille spesifi joukko mutaatio-operaattoreita Unityn tärkeimpien luokkien ja aiemman tutkimuksen perusteella. Tätä joukkoa myös testattiin alustavasti ja joitakin operaattoreita poistettiin jättäen lopulta 17:n mutaatio-operaattorin joukon. Sen jälkeen luotiin Unity Mutator – prototyyppi työkalusta, joka mahdollisti helpon Unity-projektien mutaatiotestauksen näitä operaattoreja käyttäen.

Toista kysymystä varten luotiin aineistona käytetylle valmiille pelille yksinkertainen testijoukko, josta tehtiin parannetut versiot sekä CREAM-työkalulla että Unity Mutatorilla suoritetun mutaatiotestauksen avulla. Näiden testijoukkojen kykyä tappaa toisella mutaatiotyökalulla generoidut mutantit verrattiin keskenään ja havaittiin, että CREAM:n – siis perinteisen mutaatiotestauksen – avulla luotu testijoukko saavutti korkeamman mutaatiopisteytyksen ja se sisälsi enemmän testitapauksia. Niinpä tämän testijoukon arvioitiin olevan kattavampi pitäen kuitenkin mielessä, että kokeessa käytetyillä metodeilla tätä päätelmää ei voi täysin luotettavasti tehdä. Mutaatiotekniikoita verrattiin myös niiden vaatiman manuaalisen työn – lähinnä generoitujen ekvivalenttien mutanttien määrän – perusteella ja todettiin, että Unity-spesifi mutaatiotestaus vaati huomattavasti vähemmän työtä. Unity-spesifi mutaatioprosessi siis oli nopeampi ja se myös loi mutantteja, joita ei saatu tapetuksi perinteisen mutaatiotestauksen avulla. Siksi voidaan todeta, että Unity-spesifiä mutatointia hyödyntäen olisi potentiaalia suorittaa Unity-pelien mutaatiotestausta tehokkaammin kuin pelkästään perinteisen mutatoinnin ja siihen luotujen työkalujen avulla. Tämä kuitenkin tarkoittaa vain sitä, että Unity-spesifillä mutaatiotestauksella voidaan täydentää perinteistä mutaatiotestausta – ei korvata sitä.

Kaiken kaikkiaan tutkielma oli kuitenkin vain alustava katsaus Unity-spesifiin mutaatiotes-

taukseen, sillä aiempaa tutkimusta aiheeseen liittyen ei ollut lähes ollenkaan. Sama pätee mutaatiotestauksen soveltamiseen pelikehityksessä yleensäkin. Kokeellinen osuus eli eri mutaatiotekniikoiden vertailu suoritettiin käyttäen aineistona vain yhtä peliä ja siksikin tuloksien yleistyksen kanssa on syytä olla varovainen ennen jatkotutkimusta aiheesta.

Lähteet

Ammann, Paul, ja Jeff Offutt. 2017. *Introduction to Software Testing.* 2. painos. Cambridge University Press. ISBN: 978-1-107-17201-2.

Baudry, Benoit, Franck Fleurey, Jean-Marc Jézéquel ja Yves Le Traon. 2002. "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment". Teoksessa *13th International Symposium on Software Reliability Engineering, ISSRE 2002, 12-15 November 2002, Annapolis, MD, USA,* 195–206. IEEE Computer Society. doi:10.1109/ISSRE.2002.1173246.

Bishop, Lars, Dave Eberly, Turner Whitted, Mark Finch ja Michael Shantz. 1998. "Designing a PC Game Engine". *IEEE Computer Graphics and Applications* 18 (1): 46–53. doi:10.1109/38.637270.

Budd, Timothy A., ja Dana Angluin. 1982. "Two Notions of Correctness and Their Relation to Testing". *Acta Informatica* 18:31–45. doi:10.1007/BF00625279.

Calvelo Souto, Martin, Ángel Piñeiro ja Rebeca Garcia-Fandino. 2020. "An immersive journey to the molecular structure of SARS-CoV-2: Virtual reality in COVID-19". *Computational and structural biotechnology journal* 18:2621–2628. doi:10.1016/j.csbj.2020.09.018.

Chekam, Thierry Titcheu, Mike Papadakis, Tegawendé F. Bissyandé, Yves Le Traon ja Koushik Sen. 2020. "Selecting fault revealing mutants". *Empirical Software Engineering* 25 (1): 434–487. doi:10.1007/s10664-019-09778-7.

Chekam, Thierry Titcheu, Mike Papadakis, Yves Le Traon ja Mark Harman. 2017. "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption". Teoksessa *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017,* toimittanut Sebastián Uchitel, Alessandro Orso ja Martin P. Robillard, 597–608. IEEE. doi:10.1109/ICSE.2017.61.

Christopoulou, Eleftheria, ja Stelios Xinogalos. 2017. “Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices”. *International Journal of Serious Games* 4 (4). doi:10.17083/ijsg.v4i4.194.

DeMillo, Richard A., Richard J. Lipton ja Frederick G. Sayward. 1978. “Hints on Test Data Selection: Help for the Practicing Programmer”. *IEEE Computer* 11 (4): 34–41. doi:10.1109/C-M.1978.218136.

Deng, Lin, Jeff Offutt, Paul Ammann ja Nariman Mirzaei. 2017. “Mutation operators for testing Android apps”. *Information and Software Technology* 81:154–168. doi:10.1016/j.infsof.2016.04.012.

Derezinska, Anna, ja Karol Kowalski. 2011. “Object-Oriented Mutation Applied in Common Intermediate Language Programs Originated from C#”. Teoksessa *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings,* 342–350. IEEE Computer Society. doi:10.1109/ICSTW.2011.54.

Derezinska, Anna, ja Anna Szustek. 2008. “Tool-Supported Advanced Mutation Approach for Verification of C# Programs”. Teoksessa *Third International Conference on Dependability of Computer Systems, DepCoS-RELCOMEX 2008, June 26-28, 2008, Szklarska Poreba, Poland,* toimittanut Wojciech Zamojski, Jacek Mazurkiewicz, Jaroslaw Sugier ja Tomasz Walkowiak, 261–268. IEEE Computer Society. doi:10.1109/DepCoS-RELCOMEX.2008.51.

Do, Hyunsook, ja Gregg Rothermel. 2006. “On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques”. *IEEE Transactions on Software Engineering* 32 (9): 733–752. doi:10.1109/TSE.2006.92.

Geist, Robert, Jeff Offutt ja Frederick Harris Jr. 1992. “Estimation and Enhancement of Real-Time Software Reliability Through Mutation Analysis”. *IEEE Transactions on Computers* 41 (5): 550–558. doi:10.1109/12.142681.

Ghayyur, Omaid. 2018. “Mutation Testing for Unity 3D”. Tutkielma, Capital University of Science & Technology Islamabad.

Gregory, Jason. 2018. *Game Engine Architecture.* 3. painos. A K Peters/CRC Press. ISBN: 9781315267845. doi:10.1201/9781315267845.

Harman, Mark, Yue Jia ja William B. Langdon. 2010. "A Manifesto for Higher Order Mutation Testing". Teoksessa *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings,* 80–89. IEEE Computer Society. doi:10.1109/ICSTW.2010.13.

Huckle, Thomas, ja Tobias Neckel. 2019. *Bits and bugs: a scientific and historical review of software failures in computational science.* 1. painos. Society for Industrial & Applied Mathematics. ISBN: 978-1-61197-556-7. doi:10.1137/1.9781611975567.

Inozemtseva, Laura, ja Reid Holmes. 2014. "Coverage is not strongly correlated with test suite effectiveness". Teoksessa *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014,* toimittanut Pankaj Jalote, Lionel C. Briand ja André van der Hoek, 435–445. ACM. doi:10.1145/2568225.2568271.

itch.io. 2022. "itch.io - Kotisivu". Viitattu 20. huhtikuuta. https://itch.io.

Ivankovic, Marko, Goran Petrovic, René Just ja Gordon Fraser. 2019. "Code coverage at Google". Teoksessa *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019,* toimittanut Marlon Dumas, Dietmar Pfahl, Sven Apel ja Alessandra Russo, 955–963. ACM. doi:10.1145/3338906.3340459.

Jia, Yue, ja Mark Harman. 2011. "An Analysis and Survey of the Development of Mutation Testing". *IEEE Transactions on Software Engineering* 37 (5): 649–678. doi:10.1109/TSE.2010.62.

Just, René, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes ja Gordon Fraser. 2014. "Are mutants a valid substitute for real faults in software testing?" Teoksessa *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014,* toimittanut Shing-Chi Cheung, Alessandro Orso ja Margaret-Anne D. Storey, 654–665. ACM. doi:10.1145/2635868.2635929.

Kurtz, Bob, Paul Ammann, Jeff Offutt ja Mariet Kurtz. 2016. “Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness”. Teoksessa *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016,* 142–151. IEEE Computer Society. doi:10.1109/ICSTW.2016.41.

Leveson, N.G., ja C.S. Turner. 1993. “An investigation of the Therac-25 accidents”. *Computer* 26 (7): 18–41. doi:10.1109/MC.1993.274940.

Lindström, Birgitta, ja András Márki. 2019. “On strong mutation and the theory of subsuming logic-based mutants”. *Software Testing, Verification and Reliability* 29 (1-2). doi:10.1002/stvr.1667.

Miller, Joan C., ja Clifford J. Maloney. 1963. “Systematic mistake analysis of digital computer programs”. *Communications of the ACM* 6 (2): 58–63. doi:10.1145/366246.366248.

Mirshokraie, Shabnam, Ali Mesbah ja Karthik Pattabiraman. 2013. “Efficient JavaScript Mutation Testing”. Teoksessa *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013,* 74–83. IEEE Computer Society. doi:10.1109/ICST.2013.23.

Muangpoon, Theerapat, Reza Haghighi Osgouei, David Escobar-Castillejos, Christos Kontovounisios ja Fernando Bello. 2020. “Augmented Reality System for Digital Rectal Examination Training and Assessment: System Validation”. *Journal of medical Internet research* 22 (8): 1–13. doi:10.2196/18637.

Nguyen, Quang Vu, ja Lech Madeyski. 2014. “Problems of Mutation Testing and Higher Order Mutation Testing”. Teoksessa *Advanced Computational Methods for Knowledge Engineering - Proceedings of the 2nd International Conference on Computer Science, Applied Mathematics and Applications, ICCSAMA 2014, 8-9 May, 2014, Budapest, Hungary,* toimittanut Tien Van Do, Hoai An Le Thi ja Ngoc Thanh Nguyen, 282:157–172. Advances in Intelligent Systems and Computing. Springer. doi:10.1007/978-3-319-06569-4_12.

Offutt, Jeff. 1988. “Automatic Test Data Generation”. Tohtorinväitöskirja, Georgia Institute of Technology.

Offutt, Jeff. 1992. “Investigations of the Software Testing Coupling Effect”. *ACM Transactions on Software Engineering and Methodology* 1 (1): 5–20. doi:10.1145/125489.125473.

Offutt, Jeff, Ammei Lee, Gregg Rothermel, Roland H. Untch ja Christian Zapf. 1996. “An Experimental Determination of Sufficient Mutant Operators”. *ACM Transactions on Software Engineering and Methodology* 5 (2): 99–118. doi:10.1145/227607.227610.

Offutt, Jeff, Jie Pan ja Jeffrey M. Voas. 1995. “Procedures for Reducing the Size of Coverage-based Test Sets”. Teoksessa *Proceedings of the 12th International Conference on Testing Computer Software,* 111–123. ACM.

Offutt, Jeff, ja Roland H. Untch. 2001. “Mutation 2000: Uniting the Orthogonal”. Teoksessa *Mutation Testing for the New Century,* 34–44. Boston, MA: Springer US. doi:10.1007/978-1-4757-5939-6_7.

Papadakis, Mike, Márcio Eduardo Delamaro ja Yves Le Traon. 2014. “Mitigating the effects of equivalent mutants with mutant classification strategies”. *Science of Computer Programming* 95:298–319. doi:10.1016/j.scico.2014.05.012.

Papadakis, Mike, Christopher Henard, Mark Harman, Yue Jia ja Yves Le Traon. 2016. “Threats to the validity of mutation-based test assessment”. Teoksessa *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016,* toimittanut Andreas Zeller ja Abhik Roychoudhury, 354–365. ACM. doi:10.1145/2931037.2931040.

Papadakis, Mike, Yue Jia, Mark Harman ja Yves Le Traon. 2015. “Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique”. Teoksessa *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1,* toimittanut Antonia Bertolino, Gerardo Canfora ja Sebastian G. Elbaum, 936–946. IEEE Computer Society. doi:10.1109/ICSE.2015.103.

Papadakis, Mike, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon ja Mark Harman. 2019. “Chapter Six - Mutation Testing Advances: An Analysis and Survey”. *Advances in Computers* 112:275–378. doi:10.1016/bs.adcom.2018.03.015.

Papadakis, Mike, ja Nicos Malevris. 2010. "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies". Teoksessa *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings,* 90–99. IEEE Computer Society. doi:10.1109/ICSTW.2010.50.

Papadakis, Mike, Donghwan Shin, Shin Yoo ja Doo-Hwan Bae. 2018. "Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults". Teoksessa *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018,* toimittanut Michel Chaudron, Ivica Crnkovic, Marsha Chechik ja Mark Harman, 537–548. ACM. doi:10.1145/3180155.3180183.

Pattrasitidecha, Akekarat. 2014. "Comparison and evaluation of 3D mobile game engines". Tutkielma, Chalmers University of Technology, University of Gothenburg.

Petrovic, Goran, Marko Ivankovic, Gordon Fraser ja René Just. 2021a. "Does mutation testing improve testing practices?" Teoksessa *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021,* 910–921. IEEE. doi:10.1109/ICSE43902.2021.00087.

———. 2021b. "Practical Mutation Testing at Scale: A view from Google". *IEEE Transactions on Software Engineering.* doi:10.1109/TSE.2021.3107634.

Petrovic, Goran, Marko Ivankovic, Bob Kurtz, Paul Ammann ja René Just. 2018. "An Industrial Application of Mutation Testing: Lessons, Challenges, and Research Directions". Teoksessa *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018,* 47–53. IEEE Computer Society. doi:10.1109/ICSTW.2018.00027.

Royce, W. W. 1987. "Managing the Development of Large Software Systems: Concepts and Techniques". Teoksessa *Proceedings, 9th International Conference on Software Engineering, Monterey, California, USA, March 30 - April 2, 1987,* toimittanut William E. Riddle, Robert M. Balzer ja Kouichi Kishida, 328–339. ACM Press. http://dl.acm.org/citation.cfm?id=41801.

Schuler, David, ja Andreas Zeller. 2013. “Covering and Uncovering Equivalent Mutants”. *Software Testing, Verification and Reliability* 23 (5): 353–374. doi:10.1002/stvr.1473.

Szustek, Anna. 2007. “CREAM - An Automated Object Mutation System for C#”. Tutkielma, Warsaw University of Technology, Institute of Computer Science.

Toftedahl, Marcus, ja Henrik Engström. 2019. “A Taxonomy of Game Engines and the Tools that Drive the Industry”. Teoksessa *Proceedings of the 2019 DiGRA International Conference: Game, Play and the Emerging Ludo-Mix, DiGRA 2019, Kyoto, Japan, August 6-10, 2019.* Digital Games Research Association. http://www.digra.org/digital-library/publications/a-taxonomy-of-game-engines-and-the-tools-that-drive-the-industry/.

Tsai, Yu-Tza, Wei-Yi Jhu, Chia-Chun Chen, Chien-Hao Kao ja Cheng-Yi Chen. 2019. “Unity game engine: interactive software design using digital glove for virtual reality baseball pitch training”. *Microsystem Technologies* 27:1401–1417. doi:10.1007/s00542-019-04302-9.

Unity. 2022a. “Unity - Creating and Using Scripts”. Viitattu 20. huhtikuuta. https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html.

———. 2022b. “Unity - Multiplatform”. Viitattu 20. huhtikuuta. https://unity.com/solutions/multiplatform.

———. 2022c. “Unity - Visual Scripting”. Viitattu 20. huhtikuuta. https://unity.com/features/unity-visual-scripting.

———. 2022d. “Unity – Platformer Microgame”. Viitattu 12. huhtikuuta. https://learn.unity.com/project/2d-platformer-template.

———. 2022e. “Unity Script Reference - Object”. Viitattu 21. huhtikuuta. https://docs.unity3d.com/ScriptReference/Object.html.

———. 2022f. “Unity Script Reference – Quaternion”. Viitattu 1. helmikuuta. https://docs.unity3d.com/ScriptReference/Quaternion.html.

———. 2022g. “Unity Scripting – Important Classes”. Viitattu 1. helmikuuta. https://docs.unity3d.com/Manual/ScriptingImportantClasses.html.

Uzunbayir, Serhat, ja Kaan Kurtel. 2019. “An Analysis on Mutation Testing Tools For C# Programming Language”. Teoksessa *2019 4th International Conference on Computer Science and Engineering (UBMK),* 439–443. IEEE. doi:10.1109/UBMK.2019.8907222.

Vásquez, Mario Linares, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas ja Denys Poshyvanyk. 2017. “Enabling mutation testing for Android apps”. Teoksessa *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017,* toimittanut Eric Bodden, Wilhelm Schäfer, Arie van Deursen ja Andrea Zisman, 233–244. ACM. doi:10.1145/3106237.3106275.

Williams, T.W., M.R. Mercer, J.P. Mucha ja R. Kapur. 2001. “Code coverage, what does it mean in terms of quality?” Teoksessa *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179),* 420–424. IEEE. doi:10.1109/RAMS.2001.902502.

Yang, Qian, J. Jenny Li ja David M. Weiss. 2009. “A Survey of Coverage-Based Testing Tools”. *Computer Journal* 52 (5): 589–597. doi:10.1093/comjnl/bxm021.