Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Karoliina Webster

Yksikkö- ja integraatiotestaus Unity-pelimoottorissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

6.11.2018

Tekijä Otsikko Sivumäärä Aika	Karoliina Webster Yksikkö- ja integraatiotestaus Unity-pelimoottorissa 36 sivua + 1 liitettä 6.11.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Miikka Mäki-Uuro

Insinöörityön tarkoitus oli tehdä joukko automatisoituja yksikkö- ja integraatiotestejä käyttäen Unity-pelimoottoria osana ammattikorkeakoulun opiskelijoiden peliprojektia. Tavoitteena oli luoda automatisoituja testejä pelin keskeiset ominaisuudet toteuttavalle ohjelmakoodille, korjata löydetyt virheet ja parantaa ohjelmakoodin laatua refaktoroimalla sitä. Testejä oli tarkoitus käyttää projektin aikana ja tulevaisuudessa ohjelmakoodin toiminnan varmistamiseen sen lisäämisen tai muokkaamisen jälkeen.

Projektin aikana tutustuttiin siihen, miten yksikkö- ja integraatiotestejä voidaan toteuttaa Unity-pelimoottorissa. Testauksen kohteet suunniteltiin laatimalla testaussuunitelma, jota seurattiin ja päivitettiin projektin aikana. Pelin ohjelmakoodia refaktoroitiin siten, että sitä oli mahdollista testata. Testien avulla löydetyt virheet eli bugit kirjattiin ja korjattiin.

Projektin aikana pelille kirjoitettiin 50 erilaista testiä. Osa testeistä testasi keskeisien pelimekaniikoiden ohjelmakoodia ja osa pelin ja sen käyttämän verkkopalvelun välistä toimintaa. Testit voidaan suorittaa Unityn Test Runner -työkalulla, kaikki kerralla tai erikseen. Projektia tehdessä todettiin, että Unityn toimintaa tulee ymmärtää riittävästi, jotta sen vaikutus testien rakenteeseen ja suorittamiseen voidaan ottaa huomioon testejä tehdessä.

Projekti opetti, minkälaiset ratkaisut ohjelmakoodissa hankaloittavat tai helpottavat testaamista, ja se opetti erilaisia refaktorointitekniikoita olemassa olevan ohjelmakoodin testattavuuden parantamiseksi. Projektin ansiosta pelin ohjelmakoodista tuli helpommin testattavaa, selkeämpää ja toimintavarmempaa.

Avainsanat	ohjelmistotestaus, yksikkötestaus, Unity, pelikehitys

Author Title Number of Pages Date	Karoliina Webster Unit and integration testing in Unity game engine 36 pages + 1 appendices 6 November 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Game Applications
Instructor	Miikka Mäki-Uuro, Senior Lecturer

The goal of the thesis was to create unit and integration tests for a game developed in Unity by students. The objectives were to create automated tests for the code of the central game mechanics, fix any bugs found, and increase the quality of the code by refactoring it. The tests were intended to use to verify the functionality of the code after additions or modifications, during the project, as well as in the future.

Over the project, knowledge about developing unit and integration tests in Unity was acquired. The test objectives were planned and documented in a test plan, which was followed and updated during the project. The code was refactored to make it more testable. Any bugs found were written down and fixed.

50 different tests were written during the project. Some tested the central game mechanics, and some the interaction between the game client and the backend service. The tests could be run with Unity's Test Runner tool, either one at a time, or all at once. It was discovered, that having enough knowledge about how Unity functions is important in understanding how it affects the structuring and running of the tests.

The final year project taught what kind of architectural solutions in the code make it more testable, and different refactoring techniques to improve the testability of the existing code. As a result, the code became easier to test, clearer and more reliable.

Keywords	software testing, unit testing, Unity, game development

Sisällys

Lyhenteet

1 Johdanto 1

2 Testaus osana pelikehitysprojektia 1

2.1 Testauksen tarve ja määrä 2
2.2 Testaussuunnitelma 3
2.3 Testivetoinen kehitys 3
2.4 Virheiden raportointi ja korjaus 4

3 Testityypit ja testien rakenne 5

3.1 Yleisiä testityyppejä 5
3.2 Yksikkötestit 7
3.3 AAA-malli 8
3.4 Jäljitelmäoliot 9
3.5 Integraatiotestit 11

4 Ohjelmakoodin testattavuus ja rekfaktorointi 11

5 Automatisoitu testaus Unityssä 13

5.1 Test Runner-työkalu 13
5.2 Attribuutit 17
5.3 Testidatan syöttäminen testimetodeille 18
5.4 MonoBehaviour-olioiden testaaminen 19

6 Laser Gridin testaaminen 19

6.1 Testaussuunnitelman laatiminen 21
6.2 Keskeisien pelimekaniikoiden testaaminen 22
6.3 Pelin ja PlayFab-verkkopalvelun välisen toiminnan testaaminen 28

7 Yhteenveto 34

Lähteet 37

Liitteet

Liite 1. Testaussuunnitelma, versio 6

Lyhenteet

API	Application Programming Interface. Ohjelmointirajapinta mahdollistaa ohjelman kommunikoinnin (datan noutamisen ja/tai tallentamisen) toisen ohjelman kanssa.
JSON	JavaScript Object Notation. Avoimen standardin tiedostomuoto, jota käytetään erilaisien tietorakenteiden ilmaisemiseen tekstimuodossa.
TDD	Test Driven Development. Testivetoinen kehitys on kehitysmenetelmä, jossa ensin kirjoitetaan testejä ja sitten ohjelmakoodi, jonka tulee läpäistä testit.
XP	Experience points. Pelaaja voi ansaita pelissä kokemuspisteitä tekemällä pelissä tiettyjä asioita.

1 Johdanto

Ohjelmiston testaaminen on keskeinen menetelmä, jonka avulla pyritään parantamaan ohjelmiston laatua. Se on tärkeä osa jokaista ohjelmistotuotantoprojektia, koska ohjelmistoissa esiintyvät virheet, eli bugit, ovat erittäin yleisiä. Jotta ohjelmassa olevat virheet voidaan korjata, ne pitää ensin paikallistaa. Ohjelmiston testaaminen on yksi työkalu, jota voidaan käyttää apuna ohjelman debuggaamiseen eli virheiden jäljittämiseen ja korjaamiseen.

Insinöörityössä kehitettiin yksikkö- ja integraatiotestejä osana Metropolia Ammattikorkeakoulun opiskelijoiden Laser Grid -moninpeliprojektia. Tavoitteena oli luoda ohjelmakoodille automatisoituja testejä, joilla voitaisiin löytää virhetilanteita ja varmistua sen toiminnasta muokkaamisen ja uusien ominaisuuksien lisäämisen jälkeen. Testien automatisointi tekee testeistä helposti suoritettavia, jolloin ohjelmaa voidaan testata jatkuvasti sen kehityksen aikana. Tarkoitus oli luoda testejä keskeisien pelimekaniikoiden ohjelmakoodille ja testata peliohjelman ja palvelimen välistä vuorovaikutusta.

Tässä insinöörityöraportissa käsitellään lisäksi testaamista pelikehitysprojektin osana, erilaisia testityyppejä ja sitä, miten automatisoitua testausta voidaan tehdä Unity-pelimoottorissa.

2 Testaus osana pelikehitysprojektia

Ohjelmistovirheitä aiheuttavat sekä ihmisten tekemät virheet ohjelman luonnin aikana että ohjelman ulkopuoliset syyt, kuten laitteisto-ongelmat. Testaamisella pyritään löytämään ohjelmistossa esiintyviä virhetilanteita, joita ohjelmistossa olevat virheet aiheuttavat. Testaamisella voi olla myös muita tavoitteita kuin virhetilanteiden etsintä, esimerkiksi pyrkiä varmistamaan, että ohjelma täyttää sille asetetut vaatimukset käytettävyyden ja luotettavuuden osalta tai sen suorituskyvyn mittaus. [1, 2.1 Terms and Motivation, 2.1.1 Error, Defect and Bug Terminology.]

Yleinen testaustapa on testata peliä pelaamalla sitä, ja suurissa peliyrityksissä onkin usein oma laadunvalvontaosasto ja pelitestaajat, jotka tekevät testausta tällä tavalla.

Tätä kutsutaan myös manuaaliseksi testaamiseksi, ja se on usein hidas ja työläs testaamisen muoto, vaikkakin välttämätön pelin kokonaisvaltaisen toimivuuden varmistamiseksi [2]. Insinöörityössä keskityttiin pelin ohjelmakoodin automatisoituun testaamiseen, jolla pyritään löytämään ohjelmakoodissa olevien virheiden aiheuttamia virhetilanteita.

2.1 Testauksen tarve ja määrä

Jokaiselle projektille pitää määrittää testauksen toteutustavat, tarve ja määrä yksilöllisesti ottaen huomioon, miten suuria riskejä testaamatta jättäminen ja mahdolliset virhetilanteet aiheuttaisivat. Näihin riskeihin kuuluvat mahdolliset taloudelliset ja oikeudelliset riskit sekä riskit yrityksen maineelle. Testaamisella ei kuitenkaan voida täysin varmistaa, että ohjelmistossa ei olisi virheitä. Jotta tämä olisi mahdollista, pitäisi ohjelma pystyä testaamaan kaikissa mahdollisissa ohjelman tiloissa ja kaikilla mahdollisilla syötteillä. Käytännössä tämä ei ole toteuttamiskelpoista, koska jo pienessä ohjelmassa, joka sisältää vain muutaman ehtolauseen ja silmukan, eri ohjelman tiloja voi olla ajon aikana erittäin suuria määriä. [1, 2.1.4 Test effort.]

Yksi esimerkki vakavasta bugista videopelissä on CPP Games -pelikehittäjän vuonna 2007 julkaiseman Eve Online: Trinity -pelin sisältöpäivityksen asennusohjelmassa esiintynyt bugi. Bugi aiheutti sen, että asennusohjelma poisti boot.ini-käynnistystiedoston Windows-käyttöjärjestelmän C-asemalta, vaikka sen oli tarkoitus poistaa boot.ini-niminen tiedosto asennuskansiosta. Tämä johti pahimmassa tapauksessa siihen, että tietokone ei enää käynnistynyt käyttöjärjestelmään.

Eve Online -pelin virallisessa blogissa Eve Online Software Groupin johtaja kertoo bugin olleen asennusohjelmaa varten kirjoitetussa skriptissä. Skriptiä kirjoittaessa oletettiin, että boot.ini-tiedoston poistava Delete-komento ajettaisiin asennuskansiossa, joten tiedoston osoittamiseen ei käytetty koko tiedostopolkua, vaan pelkästään tiedostonimeä. Pelin ja sisältöpäivityksen julkaisun jälkeen paljastui kuitenkin nopeasti, että pelkkää tiedostonimeä käytettäessä asennusohjelma poisti tiedoston C-asemalta. Bugin takia työntekijöitä jouduttiin kutsumaan keskellä yötä takaisin töihin tutkimaan ja korjaamaan sitä.

Syy siihen, miksi näin vakava bugi pääsi julkaistuun ohjelmistoon asti, oli kehittäjän mukaan se, että testaamiseen oli käytettävissä liian vähän aikaa. Sisältöpäivitys oli kiire julkaista samaan aikaan itse pelin kanssa. Lisäksi jälkikäteen todettiin, että testauksen aikana käytetyt tietokoneet käyttöjärjestelmineen eivät olleet kokoonpanoiltaan ja konfiguraatioiltaan tarpeeksi monipuolisia, jotta bugi olisi tullut esiin testauksen aikana.

Ainakin 215 pelaajaa kärsi bugin aiheuttamista ongelmista ja otti sen vuoksi yhteyttä pelin asiakastukeen. Pelaajien auttamiseksi jouduttiin käyttämään myös ulkopuolista, maksullista teknistä tukipalvelua. Tapaus johti testausprosessien uudelleenarviointiin CPP Gamesissa. [3.]

2.2 Testaussuunnitelma

Ohjelmistotuotantoprojektin alkuvaiheessa on hyvä laatia testaussuunnitelma, johon kirjataan testauksen tavoitteet ja testaukseen käytettävät resurssit, esimerkiksi ketkä työntekijöistä tekevät testausta, milloin sitä tehdään ja mitä testauslaitteistoa he tarvitsevat. Testaussuunnitelman tavoitteena on priorisoida testauksen kohteet ottaen huomioon riskienhallinnan ja käytettävissä olevat resurssit. Lisäksi kannattaa sopia, miten ja minne löydetyt virheet kirjataan.

Projektin aikana pidetään kirjaa siitä, onko testaussuunnitelmassa pysytty ja suunnitelmaa päivitetään tarvittaessa. Jos testaussuunnitelmaa ei pidetä ajan tasalla, voidaan päätyä testaamaan turhaan jo poistettuja ominaisuuksia tai raportoimaan virheellisesti bugeja. Testaussuunnitelmaan on hyvä kirjata myös suunnitelman versio, jotta eri tuotannon vaiheista olevat testaussuunnitelmat ovat helposti erotettavissa toisistaan. [1, 2.2.1 Test Planning and Control; 4, 15.3 Test Plans.]

2.3 Testivetoinen kehitys

Kehitystyön menetelmänä voidaan käyttää testivetoista kehitystä (engl. Test Driven Development). Se on tekniikka, jossa pienissä erissä ensin kirjoitetaan testejä ja vasta sitten kirjoitetaan ohjelmakoodi, joka läpäisee testit. Näin päädytään luomaan joukko suoritettavia vaatimuksia, jotka ohjelmakoodin täytyy läpäistä. Niin kauan, kuin

ohjelmakoodi ei läpäise testejä, se ei täytä ohjelmistolle asetettuja vaatimuksia. Yksikkötestien kirjoittaminen on keskeinen osa testivetoista kehitystä. [5, s. 7–8.]

2.4 Virheiden raportointi ja korjaus

Kun virhe eli bugi löytyy testauksen aikana, se tulisi kirjata muistiin. Tähän tarkoitukseen on tarjolla useita valmiita ohjelmistoja ja palveluita. Jotta bugi voidaan korjata mahdollisimman helposti, siitä pitää kirjata riittävästi tietoa. Bugista kannattaa kirjata ainakin seuraavat tiedot:

- Bugin tila, joka kertoo, missä tilassa se on. Tiloja voivat olla esimerkiksi Löydetty, Korjattu tai Suljettu.
- Bugin kategoria. Kategoria kertoo, mihin pelin osa-alueeseen bugi liittyy (esimerkiksi käyttöliittymään, pelimekaniikoihin tai tekoälyn toimintaan).
- Lyhyt kuvaus. Lyhyt kuvaus bugista helpottaa bugien organisointia ja läpikäyntiä.
- Pidempi kuvaus bugista. Lyhyen kuvauksen lisäksi tarvitaan tarkempi selitys, jossa käydään yksityiskohtaisesti läpi, millä tavalla ohjelma toimii virheellisesti eli bugi ilmenee.
- Vaiheet tai tapahtumat, joilla bugin saa toistettua (mikäli mahdollista).
- Bugin vakavuusaste.
- Bugin prioriteetti.

Bugin vakavuusasteita voivat olla esimerkiksi

- Crash bug eli kaatumisen aiheuttava bugi. Bugi on niin vakava, että se voi esimerkiksi kaataa peliohjelman tai estää pelaajaa etenemässä pelissä.
- Critical bug eli kriittinen bugi. Bugi aiheuttavaa huomattavaa haittaa pelissä, mutta ei estä sen pelaamista.
- Minor bug eli vähäinen bugi. Bugi ei aiheuta merkittävää haittaa pelikokemukselle.

Vakavuusasteen lisäksi bugi kannattaa priorisoida. Saman vakavuusasteen bugeja voi olla useita, mutta riippuen siitä, missä kohtaa peliä ne esiintyvät, niiden aiheuttama haitta voi olla suurempi. Enemmän haittaa aiheuttavat bugit tulisi priorisoida korkeammalle, jotta ne tulevat korjatuksi nopeammin. [4, 15.4 Testing Pipelines; 3, 15.5 Testing Cycle.]

Korjauksen jälkeen bugin poistuminen tulisi varmistaa, ennen kuin se asetetaan Suljettu-tilaan. Korjaus ei välttämättä onnistunut kuten ajateltiin, tai se saattoi rikkoa jonkin muun koodin osan toiminnan. Tässä tilanteessa tulee erityisesti esiin testien automatisoinnin hyöty. Jos ne on kirjoitettu valmiiksi, ne voidaan nyt suorittaa napin painalluksella tai jopa ilman käyttäjän toimenpiteitä.

3 Testityypit ja testien rakenne

3.1 Yleisiä testityyppejä

Ohjelmistotuotannossa käytettävät testit voidaan jakaa eri tyyppeihin sen mukaan, mitkä niiden kohteet ja tavoitteet ovat. Yleisiä testityyppejä ovat seuraavat:

- yksikkötestit
- integraatiotestit
- järjestelmätestit
- hyväksyttämistestit.

Yksikkö- ja integraatiotesteillä testataan, miten ohjelmisto vastaa sille asetettuja teknisiä vaatimuksia. Yksikkötesteillä (Unit tests) testataan ohjelmakoodin yksittäisiä osia, mikä olio-ohjelmoinnissa tarkoittaa yleensä luokkien testaamista [1, 3.2.1 Explanation of Terms]. Integraatiotesteillä (Integration tests) testataan, miten ohjelman eri osat (kuten luokat tai asiakasohjelma ja ohjelmistorajapinta eli API) toimivat keskenään.

Siinä missä yksikkö- ja integraatiotestit keskittyvät ohjelman osien teknisen toiminnallisuuden varmistamiseen, järjestelmätestit (System tests) testaavat ohjelman toimintaa kokonaisuutena ja niiden tavoite on varmistaa, että ohjelma on käyttökelpoinen kokonaisuus ja että kaikki suunnitellut ominaisuudet on muistettu implementoida [1, 3.4.1 Explanation of Terms; 1, 3.4.3 Test Objectives.].

Hyväksyttämistestausta (Acceptance testing) tehdessä testataan, täyttääkö ohjelma sille asetetut vaatimukset käyttökelpoisuuden osalta. Testaukseen voivat osallistua myös loppukäyttäjät. Pelituotannossa erityisesti käytettäviä hyväksymistestauksen muotoja ovat

alfa- ja beetaversioiden testaus ja yhteensopivuus- ja vaatimustenmukaisuustestaus. Pelin alfa- ja beetaversio ovat nimityksiä pelille sen tietyissä kehitysvaiheissa. Alfaversiossa pelin keskeiset mekaniikat on implementoitu, ja osa asseteista, eli pelissä käytettävistä digitaalisista resursseista, on luotu. Beetaversiossa pelin ohjelmakoodi ja assetit ovat valmiita ja keskitytään debuggaamiseen. Alfatestaus suoritetaan yleensä tuottajan sisäisesti, kun taas beetatestaukseen voivat osallistua myös käyttäjät. [1, 3.5 Acceptance Test; 4, 10.3 Define Milestones and Deliverables.]

Yhteensopivuustestauksessa (Compatibility testing) testataan, että peli toimii eri laitteilla tai laitteistokokoonpanoilla. Etenkin PC-pelien kohdalla erilaisia laitteistokokoonpanoja, joilla peliä voidaan pelata, on suuria määriä. Yhteensopivuustestauksen voi tarvittaessa suorittaa kolmas osapuoli, jolla on riittävä osaaminen ja laitteisto testaamiseen, kuten testilaboratorio. [4, 15.6 External testing.]

Eri julkaisualustoilla (esim. pelikonsolit tai mobiilikäyttöjärjestelmät) on omat vaatimuksensa sille, minkälaista sisältöä niillä voidaan julkaista. Ennen kuin peli julkaistaan julkaisualustalle, peli käy läpi julkaisualustasta riippuvan arvioinnin tai testauksen, jossa katsotaan, että se täyttää julkaisualustan sisältövaatimukset. Pelin julkaisu alustalla voidaan hylätä keskeneräisyyden, bugien tai viimeistelemättömän ulkoasun takia [6].

Jotta voidaan varmista, että peli täyttää erityiset julkaisualustaan liittyvät vaatimukset, pelille voidaan suorittaa vaatimustenmukaisuustestaus (Compliance testing). Vaatimuksenmukaisuustestauksen voi toteuttaa kehittäjä, julkaisualustan haltija tai testilaboratorio. [4, 15.5 Testing Cycle].

Kuvassa 1 nähdään Applen sovelluskauppa App Storen yleisemmät syyt sovelluksen julkaisun hylkäämiseen viikon mittaisella tarkastelujaksolla huhtikuussa 2018. Yleisin syy sovelluksen hylkäämiseen on ollut sen keskeneräisyys, johon kuuluvat myös ohjelmistossa esiintyvät bugit [7].

Top 10 reasons for app rejections during the 7-day period ending April 3, 2018.

42% Guideline 2.1 – Performance: App Completeness
10% Guideline 4.3 – Design: Spam
8% Guideline 5.1.1 - Privacy: Data Collection and Storage
8% Guideline 2.3.3 – Performance: Accurate Metadata
5% Guideline 5.1.5 - Privacy: Location Services
5% Guideline 3.1.1 – Business: In-App Purchase
3% Guideline 2.3.7 – Performance: Accurate Metadata
3% Guideline 4.2 – Design: Minimum Functionality
2% Guideline 3.2.1 – Business: Other Business Model Issues
2% Guideline 2.4.1 – Performance: Hardware Compatibility

Total Percent of App Rejections

88% Top 10 Reasons
12% Other Reasons (<2% each)

Kuva 1. Apple App Storen yleisimmät syyt sovelluksien julkaisun hylkäämiseen [6].

3.2 Yksikkötestit

Yksikkötesti on ohjelmakoodia (yleensä metodi), joka testaa luokkaan kuuluvan julkisen metodin toimintaa. Yksikkötestejä kirjoitetaan luokan julkisille metodeille, koska tavoitteena on testata luokan toimintaa riippumatta siitä, miten se on sisäisesti toteutettu. Kaikki metodin sisäiset suorituspolut tulisi pyrkiä testaamaan, joten yhdelle metodille voi olla useita yksikkötestejä, jotka testaavat metodin toimintaa eri tilanteissa. Yksikkötestien yhteisiä piirteitä ovat seuraavat:

- Ne on eristetty muusta ohjelmakoodista.
- Niillä on selkeä kohde.
- Ne ovat toistettavissa.
- Niiden tulos on ennakoitavissa. [5, s. 20.]

Yksikkötestaus tulisi aloittaa mahdollisimman varhaisessa vaiheessa projektia. Tuloksena on ohjelman kasvaessa jatkuvasti kehittyvä testikanta, josta saadaan heti palautetta, jos jonkin ominaisuuden muuttaminen tai lisääminen rikkookin jonkin osan koodista.

Esimerkkikoodissa 1 on yksikkötesti, jonka kohteena on Calculator-luokan Add-metodi. Yksikkötesti tarkistaa, että Add-metodi palauttaa arvon 3, kun sille syötetään argumentteina luvut 2 ja 1.

```
[Test]
public void Add_1To2_Returns3()
{
    var calculator = new Calculator();
    var result = calculator.Add(2, 1);
    Assert.That(result, Is.EqualTo(3));
}
```

Esimerkkikoodi 1. Esimerkki yksikkötestistä.

3.3 AAA-malli

AAA-malli (Arrange, Act, Assert) on yleinen suunnittelumalli testeille. Mallin mukaan suunnitellussa testissä on kolme vaihetta:

- valmistelu (Arrange)
- suorittaminen (Act)
- varmennus (Assert).

Valmisteluvaiheessa testin aikana tarvittavat oliot ja data luodaan tai alustetaan käyttövalmiiksi halutuilla testiarvoilla. Suorittamisvaiheessa testauksen kohteena oleva metodi ajetaan. Varmennusvaiheessa varmennetaan, että testauksen kohteena oleva metodi teki sen, mitä pitääkin. Tämä toteutetaan tilanteeseen sopivalla varmennuksella (Assertion). Varmennus on metodi, joka tarkistaa, onko jokin ehto tosi tai epätosi. Varmennusmetodille syötetään yleensä kaksi parametria: varsinainen ja odotettu arvo, joita verrataan keskenään tietyn ehdon mukaisesti. Jos epätosi varmennus on osana yksikkötestiä, koko testi raportoidaan epäonnistuneeksi. Selvyyden vuoksi jokaisella yksikkötestillä tulisi olla yksi selkeä asia, jonka se varmistaa. [8.]

NUnit Framework on yksikkötestaukseen tarkoitettu ohjelmistokehys .NET-ohjelmointikielille. Esimerkkikoodin 2 varmennus tarkistaa, onko muuttujan myString arvona merkkijono ”Hello” [9]. Varmennuksessa käytetään varsinaisena arvona muuttujaa myString ja odotettuna arvona Is.EqualTo-tyyppistä Constraint-oliota, jolle annetaan arvoksi merkkijono ”Hello”.

```
Assert.That(myString, Is.EqualTo("Hello"));
```

Esimerkkikoodi 2. Esimerkki varmennusmetodin käytöstä [9].

NUnit Frameworkin versiosta 3.0 lähtien varmennukset tehdään pääasiassa Assert.That-metodilla, jossa vertailuarvona käytetään Constraint-oliota. Joitakin muita hyödyllisiä Constraint-olioita ovat seuraavat:

- Is.True
- Is.False
- Is.Null
- Is.NotNull
- Is.GreaterThan
- Is.LesserThan
- Is.GreaterOrEqual
- Is.LesserOrEqual
- Is.InRange
- Has.Exactly.

Esimerkkikoodissa 3 on esimerkki Has.Exactly-olion käytöstä. Se tarkistaa, että array-taulukossa esiintyy numero 3 vain yhden kerran.

```
int[] array = new int[] { 1, 2, 3 };
Assert.That(array, Has.Exactly(1).EqualTo(3));
```

Esimerkkikoodi 3. Has.Exactly-olion käyttö osana varmennusta [10].

Varmennuksia voidaan käyttää varsinaisten testien lisäksi suoraan tavallisen ohjelmakoodin seassa virhetilanteiden havaitsemiseen. Unity-pelimoottorissa on tätä varten oma Assertions-kirjasto. Varmennuksia ei oletuksena sisällytetä valmiiseen ohjelmaan, vaan kääntäjä poistaa ne automaattisesti ohjelmaa käännettäessä. [11.]

3.4 Jäljitelmäoliot

Yksikkötestin kohteena oleva metodi voi tarvita käyttöönsä joitakin resursseja, kun se suoritetaan yksikkötestin aikana. Näitä ulkoisia resursseja kutsutaan metodin

riippuvuuksiksi. Riippuvuuksia voivat olla esimerkiksi metodin argumentteina syötettävät oliot, toiset metodit, joita metodi kutsuu, tai tietokanta, johon metodi tekee muutoksia. [12.]

Tilanteeseen liittyy useita ongelmia: Nämä ulkoiset resurssit voivat itsessään sisältää virheitä ja johtaa yksikkötestin epäonnistumiseen, vaikka sen kohdemetodissa ei todellisuudessa olisi vikaa. Lisäksi ulkoiset resurssit voivat olla työläitä valmistella käyttöön ja hidastaa testien suorittamista, mikä voi johtaa pahimmillaan siihen, että testejä jätetään suorittamatta [5, s. 29].

Näistä riippuvuuksista aiheutuvat ongelmatilanteet voidaan poistaa käyttämällä jäljitelmäolioita (Mock objects). Kohdemetodille jäljitelmäolio vaikuttaa olevan oikea olio, vaikka se vain esittää oikeaa oliota. [13.] Jäljitelmäolioiden tekoon voidaan käyttää esimerkiksi NSubstitute-kirjastoa. Jäljitelmäolio tulisi luoda perustuen rajapintaan. Esimerkkikoodin 4 esimerkissä luodaan jäljitelmäolio laskinluokalle tarkoitetusta ICalculator-rajapinnasta ja tallennetaan se _calculator-muuttujaan.

```
_calculator = Substitute.For<ICalculator>();
```

Esimerkkikoodi 4. Jäljitelmäolion luonti ICalculator-rajapinnasta käyttäen NSubstitute-kirjastoa [14].

Käyttämällä Returns-laajennusmetodia, jäljitelmäolion metodit voidaan asettaa palauttamaan tiettyjä paluuarvoja. Esimerkkikoodissa 5 asetetaan _calculator-jäljitelmäolion Add-metodi palauttamaan arvo 3, kun sille syötetään arvot 1 ja 2.

```
_calculator.Add(1, 2).Returns(3);
```

Esimerkkikoodi 5. Returns-laajennusmetodin käyttö [14].

Void-metodeja, eli metodeja, jotka eivät palauta mitään arvoa kutsujalle, testatessa voidaan käyttää Received-laajennusmetodia. Se tarkistaa, kutsuttiinko siihen liitettyä metodia testin aikana määritellyillä argumenteilla (Esimerkkikoodi 6).

```
_calculator.Received().Add(1, 2);
```

Esimerkkikoodi 6. Received-laajennusmetodi tarkistaa, kutsuttiinko Add-metodia testin aikana argumenteilla 1 ja 2 [14].

Received-laajennusmetodin vastakohta on DidNotReceive-laajennusmetodi, joka varmistaa, että metodia ei kutsuttu testin aikana (Esimerkkikoodi 7).

```
_calculator.DidNotReceive().Add(5, 7);
```

Esimerkkikoodi 7. DidNotReceive-laajennusmetodi tarkistaa, että Add-metodia ei kutsuttu testin aikana argumenteilla 5 ja 7 [14].

Received- ja DidNotReceive-laajennusmetodit korvaavat varmennusmetodin kutsumisen kyseisessä testissä. Niitä voi käyttää testissä vain kerran.

3.5 Integraatiotestit

Integraatiotesteillä pyritään varmistamaan, että ohjelman osien, esimerkiksi luokkien, välinen vuorovaikutus toimii oikein. Integraatiotestejä voidaan tehdä samoin periaattein kuin yksikkötestejä eli käyttämällä AAA-mallia ja varmennuksia. Integraatiotestejä tehdessä tulisi kuitenkin käyttää oikeita resursseja jäljitelmäolioiden sijaan, koska tarkoitus on testata näiden eri resurssien keskinäistä toimintaa.

4 Ohjelmakoodin testattavuus ja rekfaktorointi

Jo ohjelmakoodia kirjoittaessa kannattaa miettiä, miten siitä voitaisiin tehdä mahdollisimman helposti testattavaa. Ohjelmakoodia kirjoittaessa on hyvä noudattaa viittä SOLID-periaatetta [5, s. 50–52], jotka toimivat ohjenuorina testattavissa olevan ja helposti ylläpidettävän koodin luomiseen. SOLID-periaatteet ja niiden selitykset ovat seuraavat:

- Yhden vastuualueen periaate. Jokaisella metodilla ja luokalla on yksi selkeä vastuualue.
- Open/Close-periaate. Ohjelmakoodin tulisi olla avoin laajentamiselle, mutta ei muokattavissa. Jo olemassa oleviin luokkiin tulisi tehdä mahdollisimman vähän muutoksia, mutta niiden toiminnallisuutta voidaan

tarvittaessa laajentaa ja muokata aliluokin tai laajennusmetodein. Olemassa olevien luokkien, metodien ja muuttujien suora muokkaaminen voisi aiheuttaa ongelmia niiden nykyisille käyttäjille, mikä voidaan ehkäistä tällä tavalla.

- Liskovin korvaavuusperiaate. Yliluokan olio tulee voida korvata aliluokan oliolla rikkomatta ohjelman toimintaa. Aliluokka tulee implementoida siten, että koodi, jossa käytetään yliluokkaa, toimii samalla tavalla myös aliluokan kanssa.
- Rajapintojen erotteluperiaate. Rajapintojen tulee olla luotuja siten, että rajapinnan käyttäjä pääsee käsiksi vain sille tarpeellisiin metodeihin. Pääsy tarpeettomiin metodeihin voi aiheuttaa väärinkäsityksiä ja virhetilanteita.
- Riippuvuuden kääntöperiaate. Koodin tulee olla riippuvainen abstraktioista, ei toteutuksista. Käytännössä siis luokat voivat olla riippuvaisia muista luokista, mutta eivät niiden tarkasta toteutuksesta.

SOLID-periaatteiden mukaisesti tulisi siis pyrkiä rajapintojen käyttöön luokkien välisessä kommunikaatiossa, ja rajapintojen tulisi sisältää vain niiden käyttäjille tarpeellisia toimintoja. Näistä rajapinnoista voidaan sitten luoda jäljitelmäolioita, joilla yksikkötesteistä saadaan aidosti eristettyjä.

Koodista tekee helpommin testattavaa myös riippuvuuksien vähentäminen. Sen sijaan, että olio luodaan luokan sisällä new-avainsanalla, voidaan olio (tai olion korvaava jäljitelmäolio) syöttää luokalle luokan ulkopuolelta konstruktorissa, metodin argumenttina tai property-jäsenen avulla. Tätä kutsutaan riippuvuusinjektioksi (dependency injection). [5, s. 35; 15.]

Refaktoroinnilla tarkoitetaan olemassa olevan koodin parantelua siten, että siitä tulee helpommin luettavaa, helpommin ylläpidettävää ja tehokkaampaa ilman, että se, mitä koodi tekee, muuttuu. Yksikkötestit mahdollistavat koodin huolettoman refaktoroinnin: koodia voidaan parannella, ja jos se ei enää teekään mitä pitäisi, se huomataan todennäköisesti nopeasti, kun yksikkötesti epäonnistuu. Koodia refaktoroitaessa tulisi miettiä seuraavia seikkoja ja muuttaa koodia siten, että ne toteutuvat:

- Onko koodi helposti luettavaa? Sellaisellekin henkilölle, joka ei ole ennen nähnyt sitä?
- Noudattaako koodi SOLID-periaatteita?
- Käyttääkö koodi abstraktioita (rajapintoja, yliluokkia, var-tyyppiä muuttujissa)?
- Käyttääkö koodi riippuvuusinjektiota?

Koodin luettavuutta voidaan parantaa metodien ja muuttujien selkeällä nimeämisellä. Nimien pitäisi olla kirjoitettu ymmärrettävällä kielellä. Pitkiä nimiä on turha kaihtaa: pidempi, selkeä nimi on aina parempi kuin lyhyt, joka ei ole riittävän kuvaava. [5, s. 67, s. 165–166.]

Metodin ekstraktio (method extraction) on refaktorointimenetelmä, jossa osa olemassa olevan metodin sisällöstä siirretään omaksi metodikseen, jotta SOLID-periaatteiden yhden vastuualueen periaate täyttyisi. Lyhyemmät metodit ovat myös helpompia yksikkötestauksen kohteita.

5 Automatisoitu testaus Unityssä

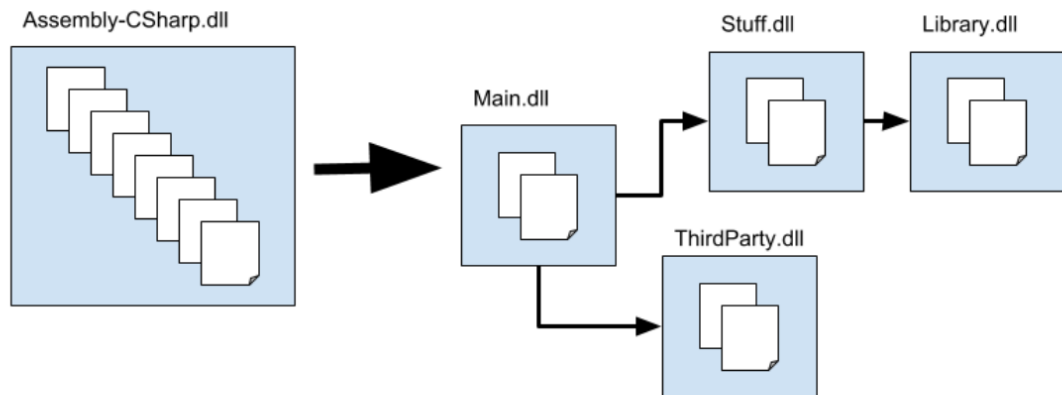
Unity-pelimoottorissa on testien kirjoittamista varten avoimen lähdekoodin NUnit Framework -ohjelmistokehys, jota voidaan käyttää yksikkö- ja integraatiotestien tekemiseen C#-ohjelmointikielellä kirjoitetuille skripteille. Testit voidaan suorittaa Unity-editorissa Test Runner -työkalulla.

5.1 Test Runner-työkalu

Test Runner on Unityn työkalu, jolla testit voidaan suorittaa Unity-ikkunassa. Unityn versiossa 2018.2.8f1 Test Runner on Window-valikon General-alavalikossa. Test Runner tunnistaa testeiksi merkityt metodit, kerää ne listaksi ja suorittaa ne, kun käyttäjä klikkaa Run All- tai Run Selected -nappeja.

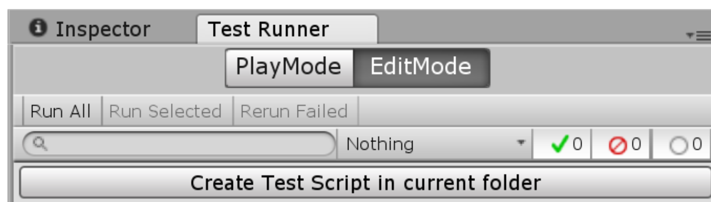
Koottaessa C#-ohjelmaa lähdekoodista ohjelmaksi ohjelman sisältämät skriptit kootaan yhdeksi tai useammaksi assembly-tiedostoksi, joiden tiedostopääte on .dll [16]. Oletuksena Unity tekee tämän automaattisesti, mutta skriptejä voidaan jakaa myös manuaalisesti pienemmiksi assembly-tiedostoiksi. Unity-projektissa olevan kansion sisältö määritellään kuulumaan tiettyyn assembly-tiedostoon luomalla kansioon Assembly Definition -tiedosto (tiedostopääte .asmdef). Kun skriptejä jaetaan pienempiin assembly-tiedostoihin, koodin kääntämiseen menee vähemmän aikaa sen jälkeen, kun johonkin skriptiin tehdään muutoksia.

Kuvassa 2 nähdään esimerkki, miten Assembly-CSharp.dll-tiedosto jaetaan pienemmiksi assembly-tiedostoiksi Main.dll, Stuff.dll, Library.dll ja ThirdParty.dll. Jos esimerkiksi Stuff.dll-tiedoston sisältämiin skripteihin tehdään muutoksia, vain se ja Main.dll-tiedosto joudutaan kääntämään uudelleen. [17.]



Kuva 2. Yhden ison assembly-tiedoston (Assembly-CSharp.dll vasemmalla) sijaan Unity-projektin skriptit voidaan jakaa useampaan pienempään assembly-tiedostoon (oikealla). Tämä lyhentää koodin kääntämiseen kuluvaa aikaa muutoksia tehtäessä. [17.]

Unityssä testejä on kahta eri tyyppiä: Edit-tilan testit ja Play-tilan testit. Edit-tilan testit voi suorittaa Unity-editorissa Edit-tilassa menemättä olenkaan Play-tilaan. Play-tilan testit voidaan suorittaa Unity-editorissa Play-tilassa tai käytettäessä koottua, ajettavaa peliohjelmaa eli buildia. Edit- tai Play-tilan testejä ei oletusarvoisesti sisällytetä buildiin, vaan ne ovat käytettävissä vain ajettaessa testejä Test Runnerin kautta. Play-tilan testit on kuitenkin mahdollista sisällyttää buildiin Test Runnerin asetuksista. Test Runnerin yläosassa ovat PlayMode- ja EditMode-napit, joilla voidaan vaihtaa Play-tilan ja Edit-tilan testien välillä (Kuva 3). [18.]



Kuva 3. Test Runner-työkalussa on välilehdet Play-tilan ja Edit-tilan testeille.

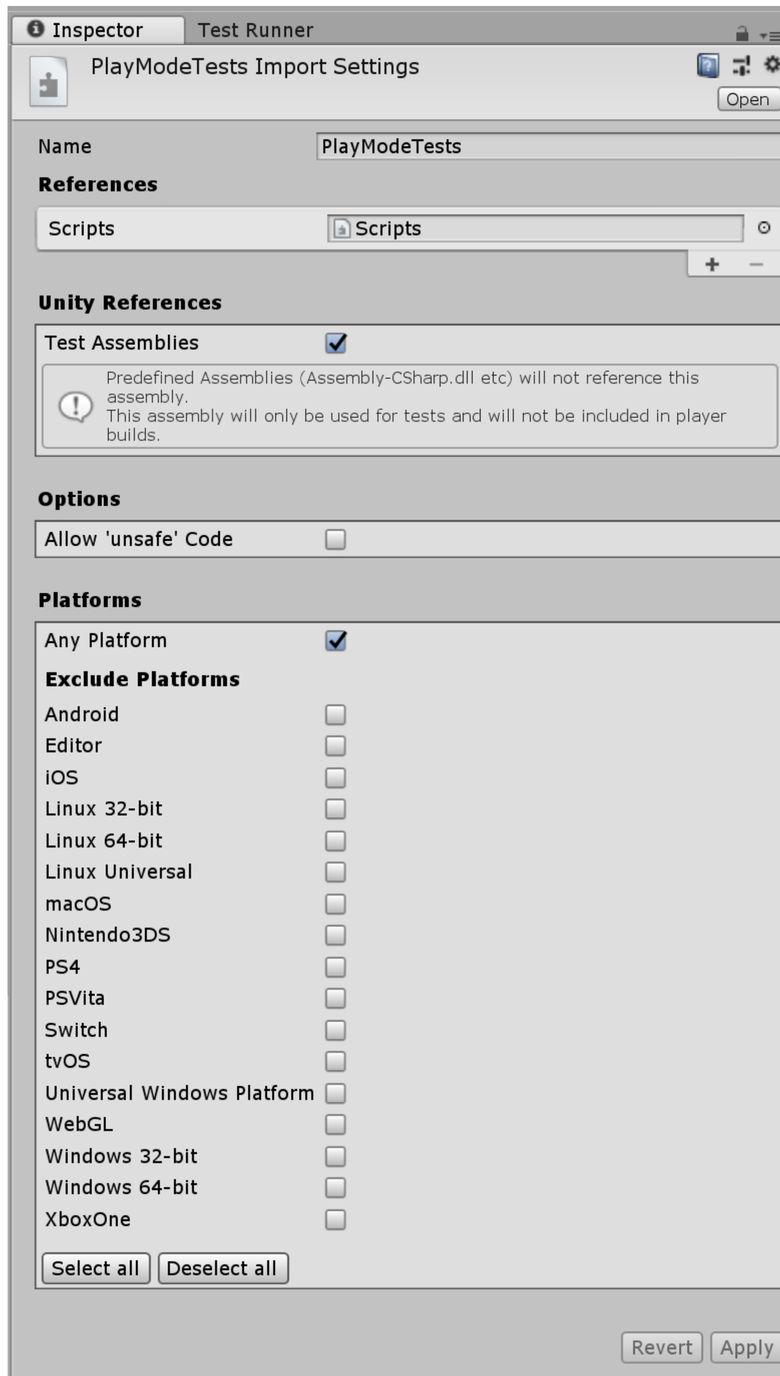
Kun Play-tilan testejä aletaan suorittaa, Unity siirtyy Play-tilaan ja luo tyhjän testiskenen. Skene (Scene) on Unityssä tiedosto, joka sisältää pelin objektit, ympäristön ja valikot. Jos Play-tilan testit halutaan suorittaa testiskenen sijaan pelin omassa skenessä, täytyy skene ladata erikseen testiskriptissä.

Testit määritellään Edit-tilan testeiksi niiden tallennuspaikan mukaan. Kelpaavia tallennuspaikkoja ovat

- Editor-kansio
- kansio, joka on määritelty kuulumaan assembly-tiedostoon, joka on merkitty Test Assembly -tiedostoksi ja jonka kohdealustana on ”Editor”
- esikäännetty assembly-tiedosto, joka sijaitsee Editor-kansiossa.

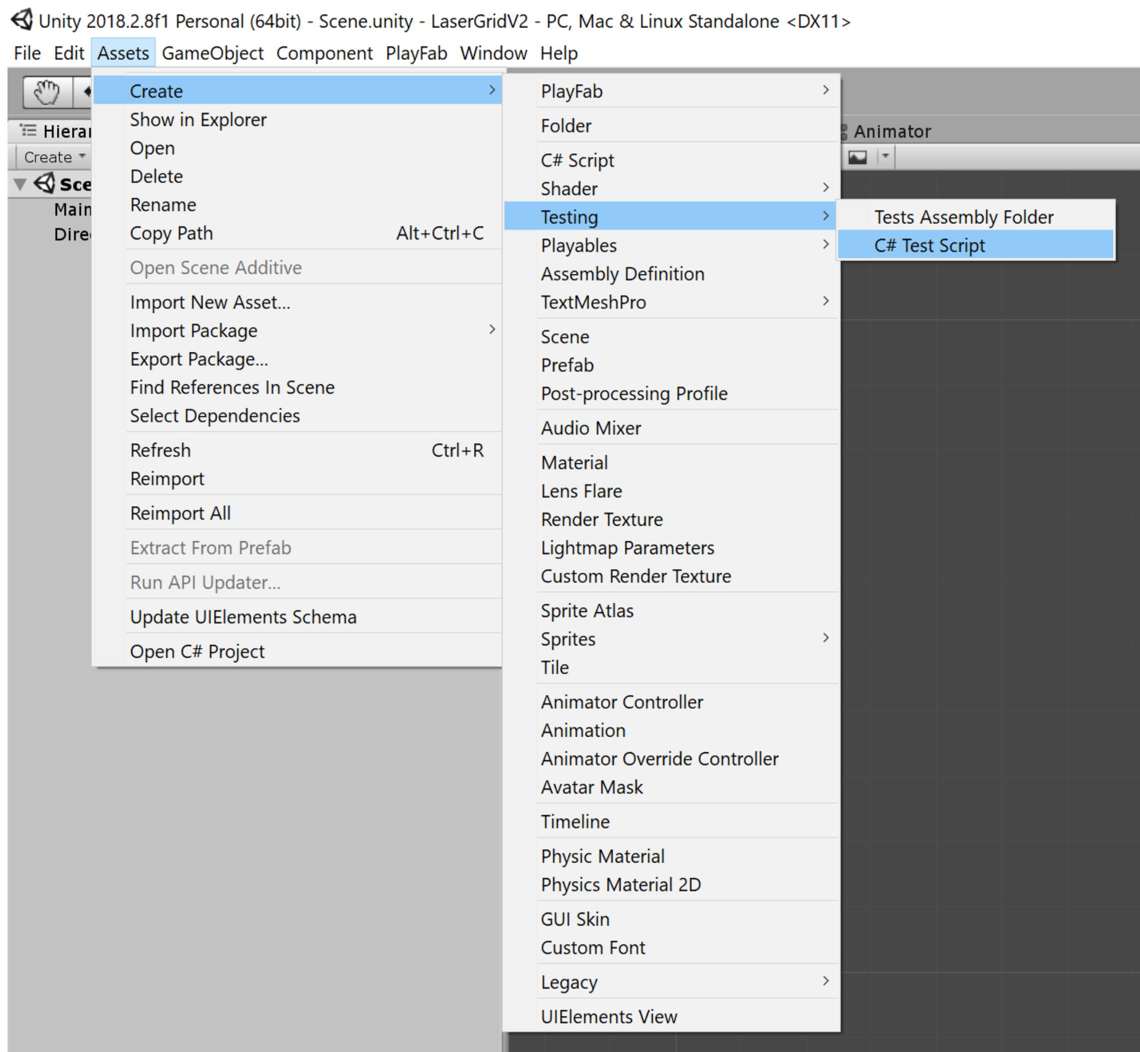
Play-tilan testeille ei ole määritelty mitään oletuskansiota, mutta niiden täytyy olla osana assembly-tiedostoa. Play-tilan testien Assembly Definition -tiedostoon pitää lisätä viittaus kaikkiin niihin assembly-tiedostoihin, jotka sisältävät testien kohteena olevia luokkia, tai skriptejä, joista kohdeluokat ovat riippuvaisia.

Kuvassa 4 näkyvät PlayModeTests-nimisen Assembly Definition -tiedoston asetukset. Asetuksista nähdään, että References-kohtaan on lisätty viittaus Scripts-nimiseen Assembly Definition -tiedostoon, jonka määrittelemä assembly-tiedosto sisältää testien kohteena olevan luokan. Jos viittausta ei lisätä, testiskripti ei tunnista kohdeluokkaa. Kuvassa näkyy myös, että Test Assemblies -kohta on valittuna ja se määrittelee, että assembly-tiedoston sisältää testejä, jolloin sitä ei sisällytetä valmiiseen ohjelmaan eli buildiin. Platforms-kategorian valinnoilla voidaan määrittää mille alustoille assembly-tiedosto on tarkoitettu. Allow ’unsafe’ code -kohta pitää olla valittuna, jos assembly-tiedostoon kuuluvassa skriptissä käytetään C#-ohjelmointikielen unsafe-avainsanaa. Avainsanaa täytyy käyttää, jos ohjelmakoodissa käytetään osoittimia (pointer).



Kuva 4. Esimerkki Assembly Definition -tiedoston asetuksista konfiguroituna testiskriptejä varten.

Uusi testiskripti luodaan Unityssä Assets-valikossa (Kuva 5). Testit vaativat UnityEngine.TestTools- ja NUnit.Framework-nimiavaruudet käyttöönsä. Testiskriptiin on lisätty ne valmiiksi, ja se sisältää myös esimerkkejä testimetodeista.



Kuva 5. Uuden testiskriptin voi luoda Unityn Assets-valikon kautta.

5.2 Attribuutit

Metodi merkitään testimetodiksi attribuutilla. Niillä voidaan hallita myös testin ajamisen ajankohtaa. Attribuutit ovat hakasulkujen sisään kirjoitettuja avainsanoja, jotka kirjoitetaan metodin yläpuoliselle riville. Metodi merkitään testimetodiksi joko Test-attribuutilla (NUnit Frameworkin tapa) tai UnityTest-attribuutilla. UnityTest-attribuutilla merkityn

metodin pitää palauttaa IEnumerator-tyypin olio. Tämä mahdollistaa odottamisen yield-käskyllä, jolloin testin ajo voidaan pysäyttää esimerkiksi odottaessa joidenkin taustalla tapahtuvien prosessien valmistumista. Yield-käsky toimii sekä Edit- että Play-tilan testeillä, mutta Edit-tilassa on mahdollista käyttää ainoastaan seuraavaa ruutupäivitystä odottavaa yield return null -lausetta. Play-tilassa voidaan käyttää muitakin paluuarvoja yield return -lauseessa, esimerkiksi yield return WaitForSeconds, joka odottaa tietyn sekuntimäärän, ennen kuin koodin ajoa jatketaan. UnityPlatform-attribuutilla voidaan määrittää, millä alustoilla testi ajetaan. [18; 19.]

Mikäli ennen testin tai testien ajoa halutaan tehdä joitakin ennakkotoimenpiteitä, voidaan käyttää SetUp- ja OneTimeSetUp-attribuutteja metodien edessä. SetUp-attribuutilla merkitään metodi, joka halutaan ajettavan ennen jokaista testiä. OnTimeSetUp-attribuutilla merkitään metodi, joka ajetaan ainoastaan kerran, ennen kuin kaikkien testien ajo aloitetaan. Näiden valmisteluattribuuttien vastakohdat ovat TearDown- ja OnTimeTearDown-attribuutit, jotka ajetaan joko jokaisen testin suorittamisen jälkeen tai vain kerran, kun kaikki testit on suoritettu. [20.]

5.3 Testidatan syöttäminen testimetodeille

NUnit Frameworkissa on kaksi tapaa syöttää testidataa testimetodeille: TestCase-attribuutti ja TestCaseData-luokan käyttäminen. TestCase-attribuutilla voidaan merkitä metodi testimetodiksi ja syöttää sille argumentteja, joilla sitä halutaan testata ajon aikana. Esimerkkikoodissa 8 nähdään, miten Add-metodille syötetään eri numeroita summattavaksi ja määritellään odotetut tulokset TestCase-attribuuteilla.

```
[TestCase(1, 2, ExpectedResult = 3)]
[TestCase(-8, 2, ExpectedResult = -6)]
[TestCase(12, 12, ExpectedResult = 24)]
public int Add(int a, int b) {
	return a + b;
}
```

Esimerkkikoodi 8. Summattavien numeroiden syöttäminen Add-metodille ja tuloksien tarkistaminen TestCase-attribuutteja käyttäen.

Jos testiargumentteja on paljon, ne voidaan sijoittaa omaan luokkaansa. Luokka, jossa testiargumentit ovat, osoitetaan testimetodille TestCaseSource-attribuutilla. [21.]

5.4 MonoBehaviour-olioiden testaaminen

Unityssä kaikkien peliobjekteihin (game object) liitettävien skriptien täytyy periä MonoBehaviour-luokka. MonoBehaviour-luokan aliluokat voivat toteuttaa tiettyjä tapahtumafunktioita (event functions), joita kutsutaan tietyssä vaiheessa skriptin ajoa tai tiettyjen tapahtumien tapahtuessa pelimoottorissa. Usein käytettyjä MonoBehaviour-luokan tapahtumafunktioita kutsuajankohtineen ovat seuraavat:

- Awake-funktiota kutsutaan, kun skripti ladataan.
- Start-funktiota kutsutaan ennen ensimmäistä ruutupäivitystä.
- OnEnable-funktiota kutsutaan, kun skripti asetetaan Enabled-tilaan.
- Update-funktiota kutsutaan kerran ruutupäivitystä kohti. [22.]

Tapahtumafunktioita ei kutsuta Unity editorin ollessa Edit-tilassa. Jos näiden funktioiden sisältämää koodia halutaan testata, voidaan testit toteuttaa Play-tilan testeinä.

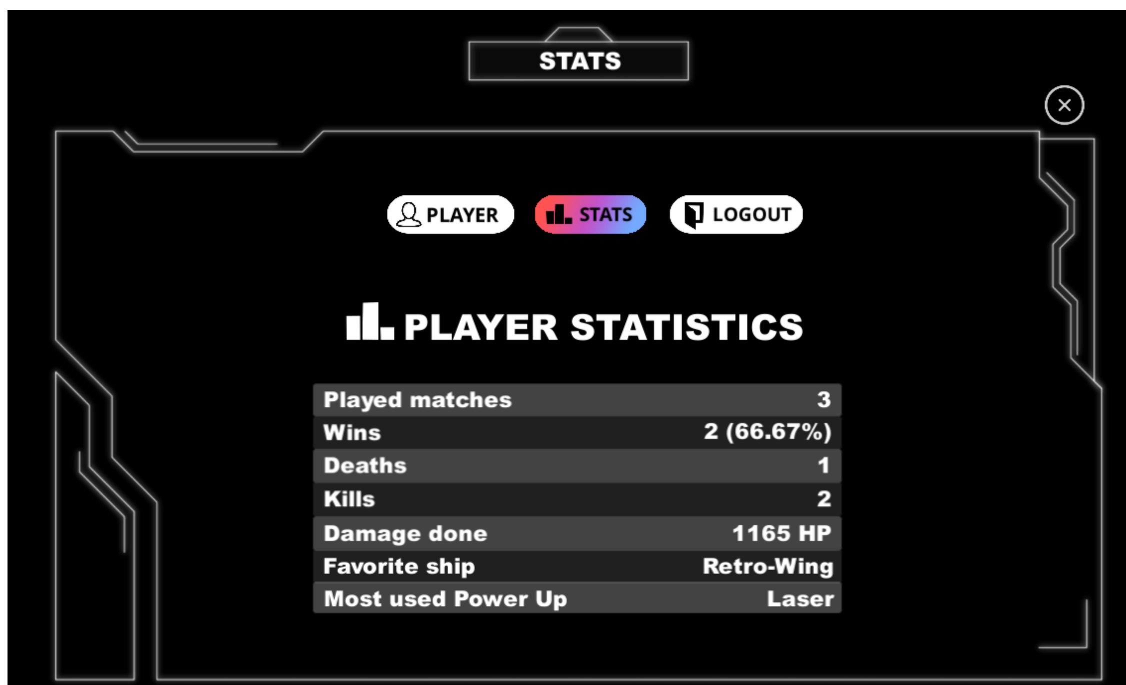
6 Laser Gridin testaaminen

Insinöörityön käytännön projektina oli testata Metropolia Ammattikorkeakoulun opiskelijoiden keskeneräistä Laser Grid -peliprojektia yksikkö- ja integraatiotestein. Laser Grid on verkkomoninpeli, jossa 2–4 pelaajaa taistelee otteluita toisiaan vastaan erilaisin aluksin. Pelaajan tavoite on tuhota muut pelaajat, ja viimeisenä elossa oleva pelaaja voittaa ottelun. Pelaajat voivat ampua toistensa aluksia luodein, ja lisäksi pelikentälle ilmestyy satunnaisin väliajoin Power Upeja, eli poimittavia, kulutettavia höydykkeitä, joita pelaaja voi käyttää. Kun pelaaja käyttää poimimansa Power Upin, hän saa jonkin väliaikaisen edun: esimerkiksi Flamethrower Power Up antaa pelaajan käyttöön väliaikaisesti liekinheitin-erikoisaseen, Shield Power Up antaa pelaajalle väliaikaisen lisäsuojauksen ja Health Packin palauttaa pelaajalle menetettyjä elämäpisteitä tietyn määrän. Kuvassa 6 nähdään pelaaja käyttämässä Flamethrower Power Upia. Pelaajat saavat tekemistään tapoista ja pelaamistaan otteluista kokemuspisteitä. Kun pelaaja on kerännyt tarpeeksi kokemuspisteitä, hänen tasonsa nousee.

Kuva 6. Pelikuvaa Laser Grid -pelistä, jossa pelaaja (valkoinen alus keskellä kuvaa) käyttää liekinheitin- Power Upia.

Peli käyttää kahta eri verkkopalvelua: Photon-moninpelipalvelua [23] verkossa pelattavan moninpelin toteuttamiseen ja PlayFab-nimistä backend-as-service-palvelua [24] käyttäjän autentikointiin ja datan tallentamiseen. Pelaaja voi luoda peliin käyttäjätunnuksen, jolloin pelaajan keräämät kokemuspisteet ja taso voidaan tallentaa PlayFab-palveluun. Lisäksi kerätään tilastoja, kuten pelattujen ja voitettujen otteluiden määrät, pelaajan tekemät tapot, pelaajan kuolemat ja pelaajan käyttämät Power Upit (kuva 7). Pelaajakohtaisen datan lisäksi PlayFabiin tallennetaan pelin yleisiä tilastoja, kuten kaikki pelissä käytetyt Power Upit.

Peliä oli ohjelmoinut tähän mennessä kuusi eri opiskelijaa. Peliä kehitettiin edelleen aktiivisesti Metropolian opiskelijoiden toimesta, ja se oli tarkoitus tulevaisuudessa julkaista, joten pelille haluttiin luoda yksikkötestejä, joilla voitaisiin nopeasti varmistua, että koodin lisääminen ja muokkaaminen ei hajottanut jo olemassa olleen koodin toimintaa. Lisäksi integraatiotesteillä voitaisiin nopeasti varmistua, että yhteys ja tiedonsiirto PlayFab-palvelun ja asiakasohjelman välillä toimii kuten pitää.



Kuva 7. Pelaajan henkilökohtaiset tilastot Laser Grid -pelissä.

6.1 Testaussuunnitelman laatiminen

Projekti aloitettiin miettimällä, mitkä pelin osa-alueet ovat toimivuudeltaan keskeisimpiä pelikokemukselle ja niiden puutteellinen toimivuus olisi isoin riski pelin kannalta. Asiassa tultiin siihen tulokseen, että tärkeimpiä osa-alueita pelikokemuksen kannalta olivat kilpailemiseen ja sillä saavutettuihin asioihin liittyvät toiminnot:

- pelaajaan liittyvät pelimekaniikat, kuten ampuminen ja vahingoittuminen
- Power Upien poimiminen ja käyttäminen
- kokemuspisteiden keräys ja pelaajan tason nousu
- pelaajan tunnistautuminen PlayFab-verkkopalveluun
- pelaajakohtaisen ja yleisen datan tallentaminen verkkopalvelimelle ja niiden noutaminen sieltä.

Koska pelin teemana on kilpailu muita pelaajia vastaan, mainituissa toiminnoissa esiintyvät ongelmat voisivat karkottaa pelaajat nopeasti pelin parista. Tämän pohjalta laadittiin testaussuunnitelma (liite 1). Testaussuunnitelmaan kirjattiin kuvaus jokaisesta toiminnosta, joka projektin aikana pyrittäisiin testaamaan, testauksen tila (ei vielä tehty,

onnistui, ohitettu tai epäonnistui) ja kuka testauksen tekisi. Yhtä testattavaa toimintoa kohden tultaisiin projektin aikana kirjoittamaan yksi tai useampia yksikkö- tai integraatiotestejä, jotta testauksen kohteena olevan koodin kaikki eri suorituspolut, eli sisäiset tilat, tulisivat mahdollisimman kattavasti testatuksi.

Jokaiselle yksittäiselle testaussuunnitelman kohdalle annettiin oma järjestysnumero eli id, jota voitaisiin tarvittaessa käyttää viittaamaan kyseiseen kohtaan. Testaussuunnitelmaan merkittiin myös suunnitelman versio, jota päivitettäisiin muutoksia tehdessä.

Testaussuunnitelmaan syntyi kolme kategoriaa: Pelaaja, XP ja Level ja Backend. Pelaaja-kategoriaan kirjattiin pelaajaan liittyviä testauskohteita, kuten ”Pelaaja voi ampua luoteja” tai ”Pelaaja voi poimia Power Upin”. XP ja Level -kategoriaan kirjatut testauskohteet liittyivät pelaajan kokemuspisteisiin ja tasoon, esimerkiksi ”Pelaajan taso nousee, kun pelaaja saavuttaa tietyn määrän kokemuspisteitä.” (Kuva 8.) Backend-kategoriassa olevat testauksen kohteet liittyivät Laser Grid -asiakasohjelman ja PlayFab-verkkopalvelun väliseen toiminnallisuuteen, esimerkiksi ”Pelaajakohtaista dataa (Player Data) voidaan noutaa verkkopalvelusta.”

5. Level			
ID	**Kuvaus**	**Tila**	**Tekijä**
501	Pelaajan taso nousee, kun pelaaja saavuttaa tietyn määrän kokemuspisteitä.	Ei tehty	Karoliina

Kuva 8. Esimerkki suunnitellusta testistä.

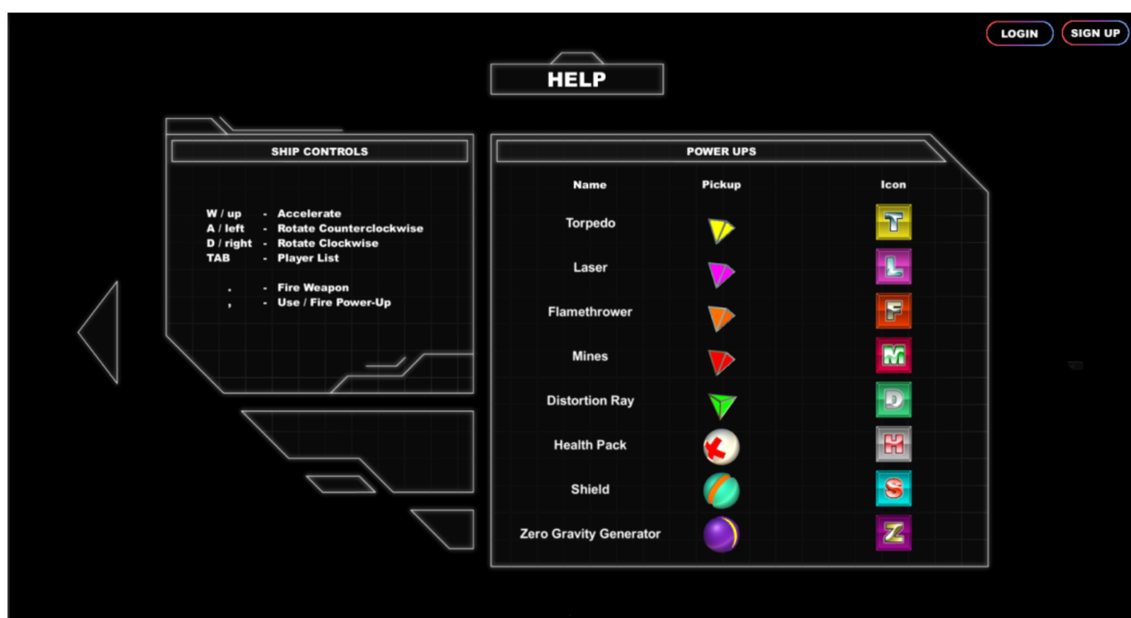
Bugien kirjaamista varten otettiin käyttöön ilmainen, selainpohjainen ohjelmistotuotantoon suunnattu projektinhallintatyökalu Pivotal Tracker [25], joka on ilmainen enintään kolmen käyttäjän tiimeille. Pivotal Tracker valittiin sen ilmaisuuden ja kätevän selainkäyttöliittymän takia.

6.2 Keskeisien pelimekaniikoiden testaaminen

Heti projektin alussa huomattiin, että pelin ohjelmakoodi vaatisi paljon refaktorointia, jotta siitä saataisiin helposti testattavaa. Koodissa esiintyi runsaasti riippuvuuksia, useat metodit olivat pitkiä eivätkä noudattaneet yhden vastuualueen periaatetta, rajapintoja ei

käytetty, ja osa metodeista ja muuttujista oli epäselvästi nimettyjä ja hankalasti luettavia. Lisäksi verkkotoiminnallisuuteen liittyvää koodia oli runsaasti sekaisin muuta pelin logiikkaa toteuttavan koodin seassa.

Power Upien toimintaa testattaessa kohteena oli kaksi eri luokkaa: Power Up -luokka, joka määrittelee yksittäisen Power Upin ominaisuudet ja PowerUpHandler-luokka, joka liitetään pelaajan aluksen peliobjektiin. Se hoitaa Power Upin poimimisen, kun pelaaja lentää Power Upin Pick Up -objektiin, ja kutsuu poimitun Power Upin Use-metodia, kun pelaaja painaa pilkkunäppäintä. Kuvassa 9 näkyvät pelissä olevien eri Power Upien nimet, Pick Up -objektit ja kuvakkeet.



Kuva 9. Pelin Help-valikko, jossa on lista eri Power Upien Pick Up -objekteista ja kuvakkeista.

Ensimmäisenä testauksen kohteeksi otettiin PowerUpHandler-luokka. Testejä varten luotiin PowerUpHandlerTests-niminen skripti. PowerUpHandler-luokkaa refaktoroitiin käyttäen metodin ekstraktiota, jossa pitkiä metodeita jaettiin useammaksi lyhyemmäksi metodiksi, joilla kullakin oli yksi tehtävä. Tällä tavalla Photon-moninpelipalvelinyhteyteen liittyvä koodi saatiin eriytettyä omiin metodeihinsa, jolloin yksikkötestaus onnistui ilman, että aktiivinen yhteys Photon-moninpelipalvelimeen oli päällä tai että peli asetettaisiin erikseen offline-tilaan.

PowerUp-luokalle tehtiin rajapinta IPowerUp (kuva 10), ja PowerUpHandler-luokka refaktoroitiin käyttämään PowerUp-tyyppisten muuttujien sijaan IPowerUp-rajapintaa. Tämä mahdollisti PowerUpHandler-luokan testaamisen IPowerUp-jäljitelmäolioilla.

```
/// <summary>
/// Interface for PowerUp class.
/// </summary>
public interface IPowerUp {
    float Units { get; set; }
    float MaxUnits { get; set; }
    bool IsAlreadyCounted { get; set; }
    GameObject Icon { get; set; }
    PowerUpStackMode StackMode { get; set; }
    PowerUpType Type { get; set; }
    void Use();
    void Stop();
}
```

Kuva 10. PowerUp-luokalle tehty IPowerUp-rajapinta, joka mahdollistaa jäljitelmäolioiden luonnin.

Usean eri luokan testit tarvitsivat käyttöönsä samoja asioita tekeviä metodeja. Testiluokkien jakamia metodeja varten tehtiin TestUtils-luokka. Sinne tehtiin muun muassa CreateMockPowerUp-metodi IPowerUp-jäljitelmäolioiden luontia varten (kuva 11).

```
/// <summary>
/// Helper method that creates a mock Power Up.
/// </summary>
public IPowerUp CreateMockPowerUp(PowerUpType type) {
    var powerUp = Substitute.For<IPowerUp>();
    powerUp.Type = type;
    return powerUp;
}
```

Kuva 11. Metodi, joka luo IPowerUp-jäljitelmäolion käyttäen NSubstitute-kirjaston Substitute-luokkaa.

PowerUpHandler-luokka perii MonoBehaviour-luokan, joten testit tehtiin Play-tilan testeinä, jotta luokan Awake- ja Start-tapahtumafunktiot tulisivat kutsutuksi. Ennen kuin PowerUpHandler-luokalle voitiin suorittaa testejä, täytyi luoda uusi peliobjekti ja liittää siihen PowerUpHandler-skripti. Tämä tehtiin OneTimeSetUp-attribuutilla merkityssä metodissa, joka ajettiin kerran ennen testien alkua. Aluksi ongelmaksi muodostui, että Test

Runner alkoi ajaa ensimmäistä testiä, ennen kuin PowerUpHandler-luokan Awake- ja Start-tapahtumafunktioita oli ehditty kutsua, ja testi epäonnistui.

Ongelma ratkaistiin muuttamalla OneTimeSetUp-metodi UnitySetup-metodiksi, ja sen paluutyyppi void-tyypistä IENumerator-tyypiksi, jolloin metodin sisällä voitiin jäädä odottamaan seuraavaa ruutupäivitystä yield return null -lauseella. Odotuksen aikana pelimoottori ehti kutsua PowerUpHandler-luokan Awake- ja Start-tapahtumafunktioita, ja testin ajo onnistui normaalisti.

Ship-luokka sisältää pelaajan alukseen liittyvää toiminnallisuutta. Sitä testatessa keskityttiin erityisesti TakeDamage-metodiin, jonka parametrit ovat pelaajan alukselle aiheutuvan vahingon määrä, vahingon lähteenä oleva peliobjekti ja vahingon tyyppi (aiheutuuko se normaalista aseesta, ympäristöstä, räjähdyksestä vai Power Upista). TakeDamage-metodille kirjoitettiin useita yksikkötestejä, jotka testasivat sitä eri vahinkolähteillä ja -tyypeillä. Kuvassa 12 on yksi näistä yksikkötesteistä. Sen aikana pelaajan alus vahingoittuu viiden elämäpisteen verran, minkä jälkeen varmistetaan, että pelaajan elämäpisteet vähenivät saman verran.

```
[Test] //Merkitsee metodin testimetodiksi
public void TakeDamage_FromNormalWeapon_HealthDecreases()
{
    //Arrange
    var damage = 5.0f;
    var damageSource = _enemyShip.gameObject;
    var expectedHealth = _playerShip.Health - damage;

    //Act
    _playerShip.TakeDamage(damage, damageSource, DamageType.NormalWeapon);

    //Assert
    //Vasemmalla todellinen arvo, oikealla odotettu arvo
    Assert.That(_playerShip.Health, Is.EqualTo(expectedHealth));
}
```

Kuva 12. Yksikkötesti, joka testaa, että pelaajan alus vahingoittuu toisen pelaajan aseesta.

XpTools-niminen luokka laskee pelaajan ansaitsemien kokemuspisteiden määrän sitä mukaa, kuin pelaaja pelaa otteluita. Luokan tehtävä on laskea myös, paljonko kokemuspisteitä tarvitaan kutakin tasoa kohti. Yksikkötestauksen aikana luokan AwardMostKillsBonus-metodista löytyi bugi. Metodin tehtävä on antaa ottelussa eniten tappoja tehneelle

pelaajalle ylimääräisiä kokemuspisteitä bonuksena. Virheellinen toiminta esiintyi sellaisessa tilanteessa, jossa kukaan pelaajista ei ollut tehnyt tappoja ottelun aikana. Koska metodi ei tarkistanut, onko suurin tappomäärä yli nollan, se antoi virheellisesti kokemuspistebonuksen kaikille pelaajille. Tilanne on epätodennäköinen pelin aikana, mutta periaatteessa mahdollinen, jos kaikki pelaajat kuolisivat ympäristön aiheuttamasta vahingosta. Bugi kirjattiin Pivotal Trackeriin (Kuva 13), ja se korjattiin lisäämällä metodiin tarkistus sille, että suurin tappomäärä on suurempi kuin nolla ennen bonuksen antamista. Korjauksen jälkeen yksikkötesti meni onnistuneesti läpi.

Kuva 13. Projektin aikana löydetty bugi kirjattuna Pivotal Tracker -projektinhallintatyökaluun.

XpTools-luokan GetNextLevelXp-metodille kirjoitettiin yksikkötesti käyttäen TestCase-attribuuttia (kuva 14). Metodi palauttaa seuraavaan tasoon tarvittavan kokemuspistemäärän, kun sille annetaan argumenttina pelaajan nykyinen taso. TestCase-attribuutille annettiin argumenteiksi nykyinen taso ja odotettu paluuarvo eli seuraavaan tasoon vaadittu kokemuspistemäärä.

```
[TestCase(1, ExpectedResult = 400)]
[TestCase(4, ExpectedResult = 3200)]
[TestCase(6, ExpectedResult = 5879)]
[TestCase(8, ExpectedResult = 9051)]
[TestCase(9, ExpectedResult = 10800)]
[TestCase(99, ExpectedResult = 394015)]
public int GetNextLevelXp(int currentLevel)
{
    return XpTools.GetNextLevelXp(currentLevel);
}
```

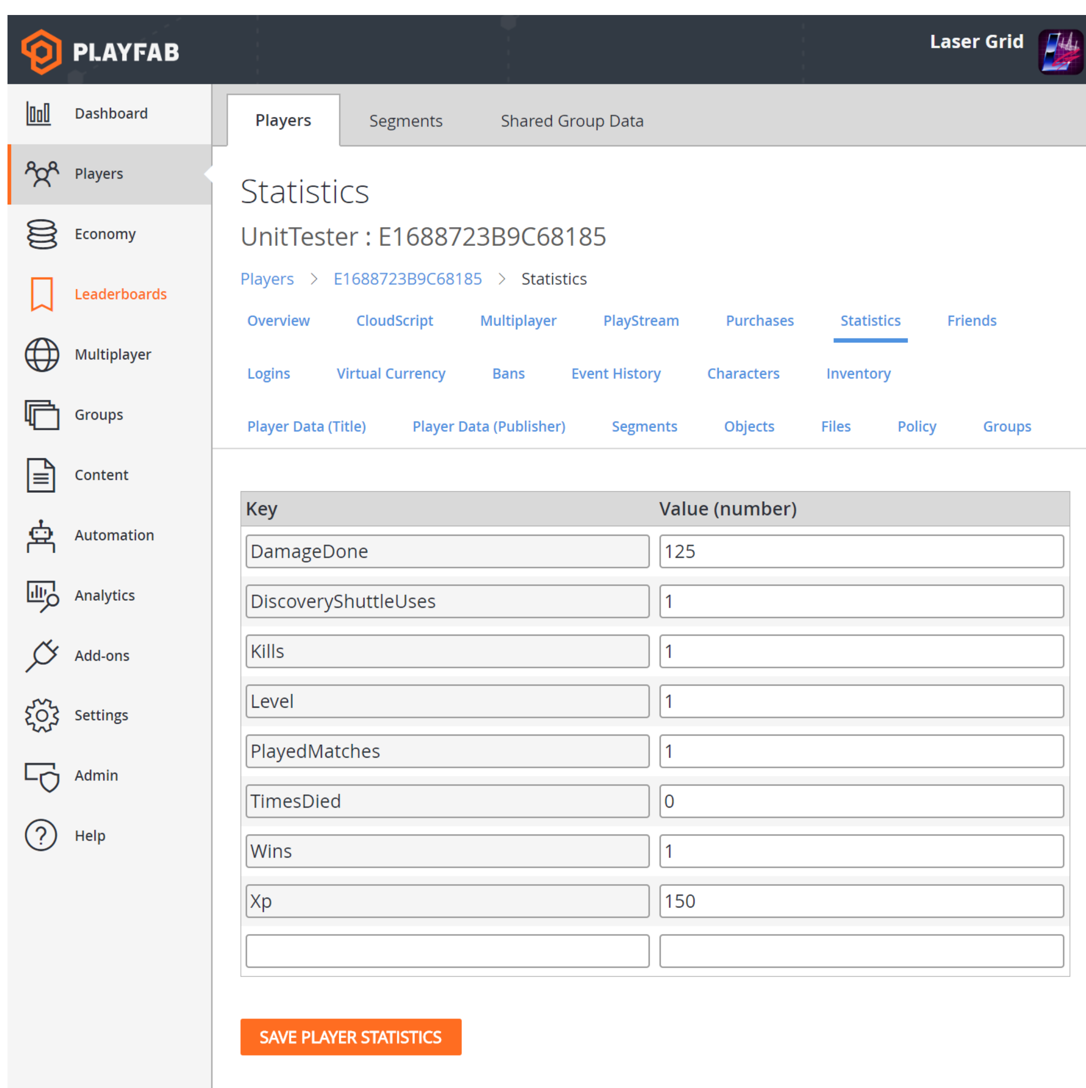
Kuva 14. XpTools-skriptin GetNextLevelXp-metodia testattiin TestCase-attribuuteilla.

Projektin aikana todettiin, että peliobjektiin liitettävien skriptien korvaaminen koodissa rajapinnoin ei aina ole vaivatonta tai tarkoituksenmukaista Unityssä, koska Unity käyttää komponenttiarkkitehtuuria. Kaikki peliobjekteihin liitettävät skriptit perivät Unityssä Component-luokan, jossa ovat jäseninä muun muassa gameObject- ja transform-ominaisuudet, joilla päästään suoraan käsiksi peliobjektiin, johon skripti on liitetty, ja sen Transform-komponenttiin. Rajapinnan kautta ei päästä suoraan käsiksi näihin paljon käytettyihin ominaisuuksiin, joten rajapintojen ehdoton käyttäminen monimutkaistaisi koodia tilanteissa, joissa peliobjektin ominaisuuksia käytetään paljon.

Pelin koodissa oli käytetty muutamassa luokassa staattisia muuttujia ja Singleton-mallia, jotka tekevät testaamisesta hankalaa. Rajapinnoissa ei voi olla staattisia muuttujia, joten staattisia muuttujia sisältävistä luokista ei voida tehdä jäljitelmäolioita. Singleton-mallissa luokasta sallitaan luoda vain yksi instanssi, johon pääsee käsiksi staattisella muuttujalla. Mikäli mahdollista, staattiset muuttujat ja Singleton-mallin voi refaktoroida koodista pois, mutta se saattaa olla hankalaa ja työlästä, jos niitä on käytetty laajalti. Yksittäisistä metodeista staattisien muuttujien suoran käytön voi ainakin refaktoroida pois riippuvuusinjektiota käyttäen. Näin toimittiin myös projektin aikana.

6.3 Pelin ja PlayFab-verkkopalvelun välisen toiminnan testaaminen

PlayFab-verkkopalvelulla on oma Client API, jolle asiakasohjelma voi tehdä pyyntöjä. API on ohjelmointirajapinta, jonka kautta kaksi ohjelmaa voivat kommunikoida keskenään; esimerkiksi asiakasohjelma voi kommunikoida verkkopalvelun kanssa. API:n kautta verkkopalvelulle voidaan lähettää dataa tallennettavaksi, tai sitä voidaan pyytää lähettämään dataa takaisin asiakasohjelmalle. PlayFabin Client API:a käytettäessä asiakasohjelman täytyy tunnistautua, joten testausta varten luotiin testikäyttäjätunnukset. Testikäyttäjän tietoihin tallennettu testidata (kuva 15) olisi tarvittaessa helppo palauttaa alkuarvoihinsa PlayFabin hallintaportaalin kautta.



Kuva 15. Testikäyttäjän Statistics-välilehti PlayFab-palvelun hallintaportaalissa.

Lähes kaikki Laser Gridissä käytetyt PlayFabin Client API:lle tehtävät pyynnöt vaativat kaksi Action-tyyppistä delegaattia argumentteina: delegaatin, jonka osoittamaa metodia kutsutaan, jos API-pyyntöön tuli vastaus onnistuneesti, ja delegaatin, jonka osoittamaa metodia kutsutaan, jos API-pyyntö epäonnistui. C#-ohjelmointikielessä delegaatit ovat metodiosoittimia eli olioita, jotka osaavat kutsua niihin liitettyjä metodeja. Delegaateilla voidaan välittää metodeita argumentteina toisille metodeille [26; 27].

Testien aikana täytyi odottaa, että API vastasi sille lähetettyyn pyyntöön ja kutsui jompaakumpaa delegaattia, ennen kuin testin tuloksesta voitiin varmistua. Testimetodit merkittiin UnityTest-attribuutein, ja ne asetetiin palauttamaan IENumerator-olio. Tällöin testimetodin ajo pystyttiin pysäyttämään yield return null -käskyllä, kunnes API:lle tehty pyyntö palautui joko onnistuneena tai epäonnistuneena.

Kuvassa 16 on yksi projektin aikana kirjoitettu integraatiotesti, joka testaa asiakasohjelman ja PlayFabin palvelimen välistä toimintaa. Testi tekee API-pyynnön, joka pyytää pelaajan tilastot palvelimelta.

```
/// <summary>
/// Tests fetching Player Statistics from PlayFab server.
/// </summary>
[UnityTest]
public IEnumerator GetPlayerStatistics_StatisticsReceived()
{
    //Pyydetään API:lta pelaajan statistiikat.
    Dictionary<string, int> stats = null;
    _target.GetPlayerStatistics(
        //Anonyymi funktio, joka käsittelee onnistuneena palautuneen pyynnön.
        (result) => {
            stats = result;
            OnSuccessfulRequest();
        },
        //Metodi, joka käsittelee epäonnistuneen pyynnön.
        OnUnsuccessfulRequest
    );

    //Odotetaan, että API-pyyntö palautuu.
    while (IsWaitingForResponse)
    {
        yield return null;
    }

    //Tarkistetaan, että data vastaanotettiin onnistuneesti.
    Assert.That(WasRequestSuccessful, Is.True, "Error: " + ErrorMessage);
    Assert.That(stats, Is.Not.Null);
    Assert.That(stats.Count, Is.GreaterThan(0));
}
```

Kuva 16. PlayFabConnector-skriptin GetPlayerStatistics-metodille kirjoitettu integraatiotesti, joka noutaa pelaajan tilastot palvelimelta.

Niin kauan kuin bool-tyyppisen IsWaitingForResponse-muuttujan arvo on tosi, testimetodi palauttaa null-arvon, eli odottaa yhden ruutupäivityksen verran. Kun IsWaitingForResponse-muuttujan arvo vaihtuu epätodeksi, testin ajoa jatketaan ajamalla while-silmukan jälkeiset varmistukset.

Projektin aikana API-pyyntöjä tekeviä metodeita jouduttiin muokkaamaan, jotta niitä voitiin testata. Alkuperäisessä toteutuksessa API-pyynnöille oli määritetty suoraan metodin sisäiseen koodiin, mitä delegaatteja tulee kutsua, kun pyyntö palautuu (kuva 17). Tässä tilanteessa API-pyynnön tulosta ei voitu ohjata takaisin testikoodille.

```
/// <summary>
/// Login with a CustomId.
/// </summary>
public void LoginWithCustomId(string customId)
{
    LoginState = State.LoginInProcess;

    //Kutsutaan API:n metodia LoginWithCustomID.
    PlayFabClientAPI.LoginWithCustomID(
        //Tehdään uusi pyyntö.
        new LoginWithCustomIDRequest()
        {
            //Pyynnön sisältö.
            TitleId = _titleId,
            CustomId = customId,
            CreateAccount = true,
            InfoRequestParameters = _infoRequestParams
        },
        //Metodi, jota kutsutaan jos pyyntö onnistui.
        OnLoginSuccess,
        //Metodi, jota kutsutaan jos pyyntö epäonnistui.
        OnPlayFabError);
}
```

Kuva 17. Alkuperäinen tilanne: LoginWithCustomID API-pyyntö kutsuu onnistuessaan OnLoginSuccess-metodia ja epäonnistuessaan OnPlayFabError-metodia.

Alkuperäisiä API-pyyntöjä refaktoroitiin käyttämään riippuvuusinjektiota. API-pyynnön palautuessa kutsuttavat delegaatit syötetään metodille parametreina, jolloin kutsuttavia metodeita voidaan helposti vaihtaa ja API-pyynnön vastaus suunnata testimetodille (kuva 18).

```
/// <summary>
/// Login with a CustomId.
/// </summary>
public void LoginWithCustomId(string customId, Action<LoginResult> resultCallback, Action<PlayFabError> errorCallback)
{
    LoginState = State.LoginInProcess;

    //Kutsutaan API:n metodia LoginWithCustomID.
    PlayFabClientAPI.LoginWithCustomID(
        //Tehdään uusi pyyntö.
        new LoginWithCustomIDRequest()
        {
            //Pyynnön sisältö.
            TitleId = _titleId,
            CustomId = customId,
            CreateAccount = true,
            InfoRequestParameters = _infoRequestParams
        },
        (result) => {
            //Metodi, jota kutsutaan jos pyyntö onnistui.
            resultCallback(result);
        },
        (error) => {
            //Metodi, jota kutsutaan jos pyyntö epäonnistui.
            errorCallback(error);
        }
    );
}
```

Kuva 18. Tilanne refaktoroinnin jälkeen: metodi saa Action-delegaatit parametreina riippuvuusinjektiota käyttäen.

Käyttäjän rekisteröimisen testaaminen oli monivaiheinen prosessi, joka toteutettiin Edit-tilan testinä. Aluksi testikäyttäjä piti rekisteröidä ja sitten poistaa juuri luotu käyttäjätunnus, jottei tietokantaan kertyisi turhia testikäyttäjiä, joita ei oikeasti käytetä. Tilanteessa muodostui ongelmaksi se, että käyttäjää yritettiin poistaa vielä, kun rekisteröitymiseen liittyvät prosessit olivat palvelimen puolella kesken. Tämä aiheutti palvelimella virheilmoituksen konfliktista, koska liian monta API-pyyntöä yritti muokata samaa resurssia yhtä aikaa.

Testi saatiin toimimaan ilman virheilmoituksia muuttamalla se Play-tilan testiksi, jolloin käyttäjän luonnin jälkeen voitiin pysähtyä odottamaan muutamaksi sekunniksi, minkä jälkeen käyttäjä voitiin poistaa ilman palvelinvirhettä.

Laser Gridiin oli kehitteillä ominaisuus, jossa pelaaja voi saada alukseensa erilaisia päivitysosia, kuten uusia perus- ja erikoisaseita, moottoreita ja kilpiä. Lista eri päivitysosista oli tallennettuna PlayFabin palvelimelle JSON-tiedostomuodossa. Asiakasohjelmassa oli ShipUpgrades-niminen skripti, joka haki JSON-tiedoston palvelimelta ja loi siinä listattujen päivityksien pohjalta ShipUpgrade-tyyppisiä C#-olioita.

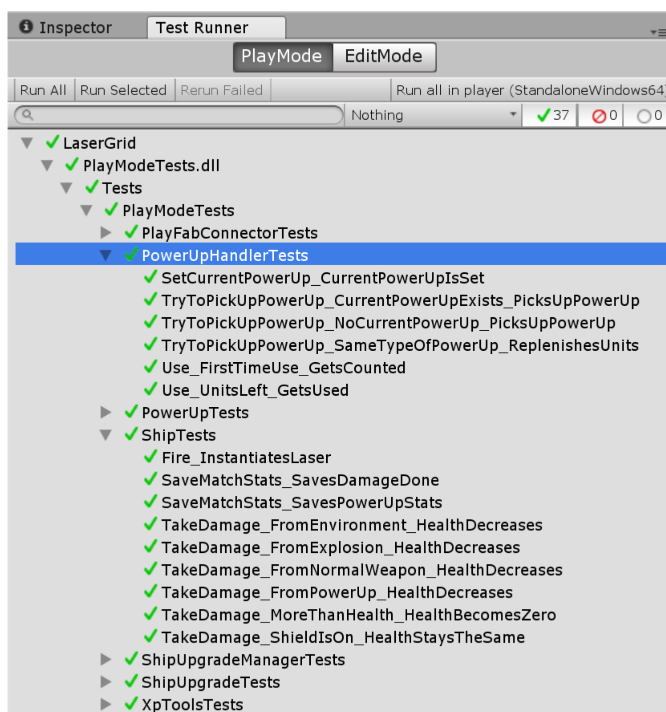
Testauksen aikana todettiin, ettei JSON-merkkijonon muuntaminen ShipUpgrade-olioiksi mennyt kuten piti. ShipUpgrade-luokalla oli jäsenmuuttuja tyyppiä ShipUpgradeType, joka oli enum-tietotyyppi. Enum on C#-ohjelmointikielessä tietotyyppi, jonka jäseninä on joukko vakioarvoisia muuttujia. ShipUpgradeType-tyyppinen muuttuja voi saada arvon Shield, Thruster, Weapon tai SpecialWeapon päivitysosan tyypin mukaisesti. Testauksen aikana kuitenkin havaittiin, että kaikki luodut ShipUpgrade-oliot saivat virheellisesti saman arvon Shield. Bugi kirjattiin Pivotal Tracker -sovellukseen (kuva 19).

Kuva 19. ShipUpgrades-skriptistä löytynyt bugi kirjattuna Pivotal Tracker -ohjelmaan.

Bugia tutkiessa havaittiin ongelman johtuvan siitä, ettei käytössä ollut JSON-serialisointikirjasto osannut muuntaa JSON-merkkijonon avain-arvoparia avainta vastaavan luokan jäsenmuuttujan arvoksi oikein, koska kyseessä oli enum-tyyppinen muuttuja. Kun muutos epäonnistui, kaikki C#-oliot saivat ShipUpgradeType-tyyppisen jäsenmuuttujan arvoksi tyypin ensimmäisen jäsenen, joka oli Shield.

JSON-serialisointikirjasto ei kuitenkaan antanut mitään virheilmoitusta muuntamisen epäonnistumisesta, joten virhe oli jäänyt aiemmin huomaamatta. Muuntamiseen kokeiltiin myös toista JSON-serialisointikirjastoa, joka puolestaan antoi virheilmoituksen muuntamisen epäonnistumisesta. Ongelma ratkaistiin lopulta muuttamalla ShipUpgrade-luokan ShipUpgradeType-tyyppinen jäsenmuuttuja string-tyyppiseksi. Muutos oli helppo tehdä, eikä sillä ollut merkittävää vaikutusta muun koodin toimintaan. Korjauksen jälkeen testi ajettiin onnistuneesti läpi.

Projektin aikana kirjoitettiin 50 testiä, joista osa oli yksikkö- ja osa integraatiotestejä. Testeistä 13 oli Edit-tilassa suoritettavia testejä ja 37 Play-tilassa suoritettavia testejä (kuva 20).



Kuva 20. Projektin aikana tehtyjä Play-tilan testejä Test Runner -työkalussa. Vihreä merkki kertoo, että testit on ajettu onnistuneesti läpi.

7 Yhteenveto

Ohjelmistotestaus on ohjelmistokehityksen tärkeä osa alue, jonka merkitystä ei voi vähätellä. Ammattimaisissa projekteissa korjaamattomilla virheillä eli bugeilla voi olla vakavia seuraamuksia. Yksikkötestit ovat nopea ja kustannustehokas tapa tehdä ohjelmistotestausta, ja niitä pitäisi suosia kehityksen aikana, unohtamatta kuitenkaan muita testejä, jotka voivat löytää virhetilanteita, jotka eivät tule esiin yksikkötestauksen aikana. Unityssä on perustyökalut AAA-suunnittelumallilla toteutettavien automatisoitujen testien tekemiseen, ja testit voi suorittaa kätevästi napin painalluksella.

Insinöörityöprojektin aikana seurattiin ja ylläpidettiin testaussuunnitelmaa ja pidettiin yllä erillistä tehtävälistaa. Testaussuunnitelmaa laajennettiin hieman projektin aikana kattamaan toiminnallisuutta, joka liittyi välillisesti jo suunniteltuihin testeihin mutta jota ei ollut kirjattu ensimmäiseen versioon. Testaussuunnitelmassa pysyttiin hyvin: kaikki suunnitellut testauskohteet saatiin testattua.

Projektin aikana kirjoitettiin 50 eri suoritettavaa testiä, ja sen lisäksi koodin refaktorointi oli huomattava osa projektia. Projekti kehitti ymmärrystä siitä, mikä tekee koodista helposti testattavaa, harjaannutti näkemään mahdollisuuksia refaktoroinnille ja auttoi ymmärtämään syitä, miksi hyväksi havaittuja toimintatapoja, kuten riippuvuusinjektiota, kannattaa käyttää. Metodin ekstraktiolla pitkiä metodeita, jotka aluksi näyttivät mahdottomilta testata, voitiin jakaa pienemmiksi, testattavissa oleviksi metodeiksi, joilla oli SOLID-periaatteiden mukaisesti yksi vastuualue.

Kirjoitettaessa testejä olemassa olevalle ohjelmakoodille, jota ei ole kirjoitettu testausta silmällä pitäen, kannattaa harkita, mitä koodia testataan ja refaktoroidaan, ottaen huomioon työhön käytettävissä olevat resurssit, kuten ajan. Runsaasti riippuvuuksia sisältävän koodin refaktoroiminen vie enemmän aikaa.

Löydetyt bugit kirjattiin selainpohjaiseen Pivotal Tracker -työkaluun. Aina suurien refaktorointien jälkeen peli koottiin ohjelmaksi, ja sille tehtiin testaus pelaamalla, jotta varmistuttiin, että yksikkötestien ulkopuolella oleva koodi toimi edelleen normaalisti.

Testin aikana käytettävien resurssien huolellinen valmistelu ja purkaminen on tärkeää, jotta niistä aiheutuvat virhetilanteet voidaan välttää. Ennen jokaista yksikkötestiä kannattaakin luoda uudet instanssit tarvittavista resursseista SetUp-attribuutilla merkityssä metodissa, jotta edellisen testin aikana muokatun olion tila ei aiheuttaisi virhettä seuraavassa testissä. Mikäli mahdollista, yksikkötestien aikana tulee käyttää jäljitelmäolioita mahdollisien virhetilanteiden minimoimiseksi.

Unity-pelimoottorin arkkitehtuuri ja toimintatapa tuovat omat erityispiirteensä testaukseen ja refaktorointiin. Unityn tapahtumafunktioiden kutsuajankohtien ymmärtäminen on tärkeää, jotta niissä alustettavia tai muokattavia resursseja ei yritetä käyttää väärään aikaan ja tästä aiheutuvat testien epäonnistumiset voidaan välttää. Jos testin aikana joudutaan odottamaan, esimerkiksi yield return WaitForSeconds -lauseella, on hyvä tulostaa konsoliin, että testi odottaa ajan kulumista, jotta sen ei luulla jumiutuneen.

Kaikkien luokkien korvaaminen koodissa rajapinnoin ei välttämättä ole suoraviivaista ja voi tehdä koodista sekavampaa, koska rajapintojen kautta ei päästä suoraan käsiksi siihen peliobjektiin, johon rajapinnan toteuttava skripti on liitetty. Tilanteessa kannattaa pyrkiä löytämään kultainen keskitie, jossa pyritään ennen kaikkea selkeään koodiin, ja käyttämään rajapintoja siellä, missä se on tarkoituksenmukaista.

Kokonaisuutena projekti onnistui hyvin. Testaussuunnitelmassa määritellyt tavoitteet saavutettiin ja löydetyt bugit korjattiin. Projekti laajensi ymmärrystä ohjelmistotestauksesta tuotantoprosessin osana ja opetti konkreettisia taitoja testaamisen tekemiseen. Se kehitti taitoa, miten kirjoittaa testattavissa olevaa, vähemmän riippuvuuksia sisältävää ja selkeämpää koodia. Pelin ohjelmakoodista tuli projektin seurauksena selkeämpää ja toimintavarmempaa, etenkin harvinaisissa tilanteissa. Ohjelmakoodi sisältää vähemmän riippuvuuksia ja enemmän käsittelyä poikkeustilanteille.

Pelin testausprosessia voitaisiin tulevaisuudessa kehittää siten, että testaussuunnitelmaa alettaisiin käyttää aktiivisesti ja sitä päivitettäisiin tarvittaessa. Peliprojektissa mukana olevat ohjelmoijat voisivat kirjoittaa yksikkötestejä tekemälleen ohjelmakoodille jo sitä kirjoittaessa. Ohjelmakoodin refaktorointia voitaisiin jatkaa SOLID-periaatteiden mukaisesti. Yksikkö- ja integraatiotestejä voitaisiin kirjoittaa myös harvemmin käytettävälle

ohjelmakoodille, koska projektin aikana kirjoitetut testit keskittyvät yleisimmin käytettyihin luokkiin ja niiden metodeihin.

Lähteet

1 Spillner, Andreas; Linz, Tilo & Schaefer, Hans. 2014. Software Testing Foundations. 4th Edition. E-kirja. Rocky Nook.

2 Hamedani, Mosh. 2018. What Is Automated Testing? Verkkoaineisto. Udemy. <https://www.udemy.com/unit-testing-csharp/learn/v4/t/lecture/8997840?start=0>. Katsottu 24.9.2018. (Maksullinen verkkokurssimateriaali.)

3 Thorsteinsson, Erlendur. 2007. About the boot.ini issue. Verkkoaineisto. Eve Online. <https://www.eveonline.com/article/about-the-boot.ini-issue/>. 12.12.2007. Luettu 22.10.2018.

4 Chandler, Heather. 2013. The Game Production Handbook. 3rd Edition. E-kirja. Jones & Barlett Learning.

5 Bender, James & McWherterm, Jeff. 2011. Professional Test-Driven Development with C#: Developing Real World Applications with TDD. E-kirja. Wiley Publishing.

6 Common App Rejections. Verkkoaineisto. Apple. <https://developer.apple.com/app-store/review/rejections/>. Luettu 25.9.2018.

7 App Store Review Guidelines. Verkkoaineisto. Apple. <https://developer.apple.com/app-store/review/guidelines/#app-completeness>. Luettu 4.10.2018.

8 Unit test basics. 2018. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2017>. 4.8.2018. Luettu 17.9.2018.

9 Constrain Model. Verkkoaineisto. Github. <https://github.com/nunit/docs/wiki/Constraint-Model>. Luettu 26.9.2018.

10 Assertions. Verkkoaineisto. Github. <https://github.com/nunit/docs/wiki/Assertions>. Luettu 10.10.2018.

11 Paszek, Tomek. 2015. The Unity Assertion Library. Verkkoaineisto. Unity. <https://blogs.unity3d.com/2015/08/25/the-unity-assertion-library/>. 25.8.2015. Luettu 17.9.2018.

12 Mindra, Dmitriy. 2014. Unit testing at the speed of light with Unity Test Tools. Verkkoaineisto. Unity. <https://blogs.unity3d.com/es/2014/07/28/unit-testing-at-the-speed-of-light-with-unity-test-tools/>. 28.7.2014. Luettu 24.9.2018.

13 Creating a substitute. Verkkoaineisto. Github. <http://nsubstitute.github.io/help/creating-a-substitute/>. Luettu 26.9.2018.

14 NSubstitute. Verkkoaineisto. Github. <https://github.com/nsubstitute/NSubstitute>. Luettu 5.10.2018.

15 Mindra, Dmitriy. 2014. Dependency injection and abstractions. Verkkoaineisto. Unity. <https://blogs.unity3d.com/2014/05/07/dependency-injection-and-abstractions/>. 7.5.2014. Luettu 1.10.2018.

16 Hamedani, Mosh. 2018. Architecture of .NET Applications. Verkkoaineisto. Udemy. <https://www.udemy.com/csharp-tutorial-for-beginners/?start=0>. Katsottu 24.9.2018. (Maksullinen verkkokurssimateriaali.)

17 Script compilation and assembly definition files. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html>. Luettu 19.9.2018.

18 Unity Test Runner. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/testing-editortestsrunner.html>. Luettu 19.9.2018.

19 Writing and executing tests in Unity Test Runner. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/PlaymodeTestFramework.html>. Versio 2018.2. Luettu 19.9.2018.

20 Attributes. Verkkoaineisto. Github. <https://github.com/nunit/docs/wiki/Attributes>. Luettu 26.9.2018.

21 TestCaseData. Verkkoaineisto. Github. <https://github.com/nunit/docs/wiki/TestCaseData>. Luettu 5.10.2018.

22 Execution Order of Event Functions. Verkkoaineisto. Unity. <https://docs.unity3d.com/Manual/ExecutionOrder.html>. Versio 2018.2. Luettu 4.10.2018.

23 Photon PUN-kotisivu. Verkkoaineisto. Photon Engine. <https://www.photonengine.com/en-US/PUN>. Luettu 10.10.2018.

24 PlayFab-kotisivu. Verkkoaineisto. PlayFab. <https://playfab.com/>. Luettu 10.10.2018.

25 Pivotal Tracker-kotisivu. Verkkoaineisto. Pivotal Tracker. <https://www.pivotaltracker.com/>. Luettu 19.10.2018.

26 Hamedani, Mosh. 2018. Delegates. Verkkoaineisto. Udemy. <https://www.udemy.com/csharp-advanced/learn/v4/t/lecture/1988794?start=0>. Katsottu 2.10.2018. (Maksullinen verkkokurssimateriaali.)

27 Delegates (C# Programming Guide). 2015. Verkkoaineisto. Microsoft. <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/index>. Luettu 2.10.2018.

Testaussuunnitelma, versio 6

Peli: Laser Grid

Pelaaja

1. Ampuminen			
ID	**Kuvaus**	**Tila**	**Tekijä**
101	Pelaaja voi ampua luoteja.	Onnistui	Karoliina

2. Pelaajan vahingoittuminen			
ID	**Kuvaus**	**Tila**	**Tekijä**
201	Pelaaja ottaa vahinkoa toisen pelaajan luodeista.	Onnistui	Karoliina
202	Pelaaja ottaa vahinkoa ympäristöstä.	Onnistui	Karoliina
203	Pelaaja ottaa vahinkoa räjähdyksistä (miinat, torpedot).	Onnistui	Karoliina
204	Pelaaja ottaa vahinkoa Power Upeista.	Onnistui	Karoliina

3. Power Upien käyttö			
ID	**Kuvaus**	**Tila**	**Tekijä**
301	Pelaaja voi poimia Power Upin.	Onnistui	Karoliina
302	Pelaaja voi käyttää jatkuvan Power Upin.	Onnistui	Karoliina
303	Pelaaja voi käyttää ammuttavan Power Upin.	Onnistui	Karoliina
304	Pelaaja voi käyttää pelikentälle ilmestyvän Power Upin.	Onnistui	Karoliina
305	Pelaajan health lisääntyy, kun pelaaja käyttää Health Packin.	Onnistui	Karoliina
306	Pelaajan shield aktivoituu, kun pelaaja käyttää Shield Power Upin.	Onnistui	Karoliina

XP ja Level

4. XP eli kokemuspisteet			
ID	**Kuvaus**	**Tila**	**Tekijä**
401	Pelaaja saa XP:tä otteluista.	Onnistui	Karoliina
402	Pelaaja saa XP:tä taposta.	Onnistui	Karoliina
403	Pelaaja saa XP bonuksen, jos pelaaja tekee eniten tappoja matsissa.	Onnistui	Karoliina

5. Level eli taso			
ID	**Kuvaus**	**Tila**	**Tekijä**
501	Pelaajan taso nousee, kun pelaaja saavuttaa tietyn kokemuspistemäärän.	Onnistui	Karoliina

Backend

6. Autentikointi			
ID	**Kuvaus**	**Tila**	**Tekijä**
601	Asiakasohjelma voi kirjautua vieraana.	Onnistui	Karoliina
602	Pelaaja voi kirjautua käyttäjätunnuksella ja salasanalla.	Onnistui	Karoliina
603	Pelaaja voidaan rekisteröidä.	Onnistui	Karoliina
604	Pelaaja voi kirjautua ulos.	Onnistui	Karoliina

7. Datan tallentaminen ja noutaminen			
ID	**Kuvaus**	**Tila**	**Tekijä**
701	Pelaajan statistiikkaa (Player Statistics) voi tallentaa.	Onnistui	Karoliina
702	Pelaajakohtaista dataa (Player Data) voi tallentaa.	Onnistui	Karoliina
703	Pelaajan statistiikkaa (Player Statistics) voi noutaa.	Onnistui	Karoliina
704	Pelaajakohtaista dataa (Player Data) voi noutaa.	Onnistui	Karoliina
705	Pelikohtaista dataa (Title Data) voi noutaa.	Onnistui	Karoliina
706	Pelaajan profiili voidaan noutaa.	Onnistui	Karoliina
707	Pelaajan lempinimi voidaan vaihtaa.	Onnistui	Karoliina

8. Ottelun datan tallentaminen			
ID	**Kuvaus**	**Tila**	**Tekijä**
801	Ottelun aikaisia tapahtumia voidaan lähettää palvelimelle.	Onnistui	Karoliina
802	Ottelun aikana kerätty data tallennetaan palvelimella pelaajan tietoihin oikein.	Onnistui	Karoliina

9. Alusten päivitysten listaus			
ID	**Kuvaus**	**Tila**	**Tekijä**
901	Alusten päivitysten lista voidaan noutaa palvelimelta.	Onnistui	Karoliina
902	Alusten päivitys voidaan hakea tasovaatimuksen mukaan.	Onnistui	Karoliina
903	Alusten päivitys voidaan hakea Id:n perusteella.	Onnistui	Karoliina
904	Päivitysolio voidaan luoda palvelimelta haettujen tietojen perusteella.	Onnistui	Karoliina

10. Cloud Script (palvelimella ajettava JavaScript-koodi)			
ID	**Kuvaus**	**Tila**	**Tekijä**
1001	Palvelinkoodia voidaan suorittaa asiakasohjelmasta käsin.	Onnistui	Karoliina