Jyri Tero

KRIISIOSAAJA-OPPIMISPELIN YLLÄPI-DETTÄVYYDEN PARANTAMINEN

Opinnäytetyö

Tekniikan ammattikorkeakoulututkinto

Peliohjelmoinnin koulutus

2024

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tutkintonimike	Insinööri (AMK)
Tekijä	Jyri Tero
Työn nimi	Kriisiosaaja-oppimispelin ylläpidettävyyden parantaminen
Toimeksiantaja	Kaakkois-Suomen ammattikorkeakoulu
Vuosi	2024
Sivut	42 sivua
Työn ohjaaja	Marko Oras

TIIVISTELMÄ

Opinnäytetyössä paneudutaan ylläpidettävyyteen ja hyviin ohjelmointikäytänteisiin tavoitteena parantaa Kriisiosaaja-oppimispelin koodipohjaa ja siten ylläpidettävyyttä. Työn teoria pohjaa ylläpidettävyyteen, hyviin ohjelmointikäytänteisiin ja koodihajujen välttämiseen. Työssä tarjotaan ehdotuksia Kriisiosaaja-projektin koodipohjan ylläpidettävyyden parantamiseksi.

Työ on kehittämistutkimus, jossa selvitetään, mitä on ylläpidettävyys, miten ylläpidettävyyttä parannetaan ja mitkä osa-alueet Kriisiosaaja-pelin koodista kaipaavat parantamista. Kysymyksiin vastataan ohjelmointialaan ja erityisesti ylläpidettävyyteen pohjaavan kirjallisuuden kautta muodostuneen teoriapohjan avulla. Kriisiosaajan ongelmat löydettiin projektin koodia tutkimalla ja projektiaikaista materiaalia, kuten dokumentaatiota ja muistiinpanoja, lukemalla.

Kriisiosaaja on Kaakkois-Suomen ammattikorkeakoulun terveysalan ja ensihoidon koulutusyksikön Kriisi- ja poikkeusolojen osaaja -opintojaksoa varten toteutettu oppimispeli, jossa pelaajan on tarkoitus selvitä kotonaan 72 tuntia kriisitilanteessa, kuten pitkittyneen sähkökatkon aikana.

Työn tuotoksena syntyi ohjeita ja kohdennettua teoriaa Kriisiosaaja-projektin käyttöön. Työssä olevat ohjeet rakentuvat työn ylläpidettävyyteen pohjaavan teoreettisen viitekehyksen päälle. Ohjeet on esitelty niin, että niiden soveltaminen projektiin laajemmin on mahdollisimman helppoa. Teoria ratkaisuiden taustalla esitellään kattavasti samasta syystä. Ratkaisuiden on tarkoitus auttaa poistamaan projektin koodipohjasta sen yleisimmät ja suurimmat ongelmat sekä antaa työkaluja ja ajattelumalleja tulevien ongelmien välttämiseksi.

Asiasanat: ohjelmointi, ylläpidettävyys, hyvät käytänteet, suunnittelumallit

South-Eastern Finland University of Applied Sciences

Degree title	Bachelor of Engineering
Author	Jyri Tero
Thesis title	Improving the maintainability of the Kriisiosaaja learning game
Commissioned by	South-Eastern Finland University of Applied Sciences
Time	2024
Pages	42 pages
Supervisor	Marko Oras

ABSTRACT

This focus on about maintainability and good programming practices with the goal of improving the codebase of the Kriisiosaaja learning game. The theoretical framework of the thesis was rooted in coding literature addressing maintainability, best practices and techniques for avoiding code smells.

The goal of improving the Kriisiosaaja codebase required understanding what maintainability is, how it can be improved and what parts of the Kriisiosaaja codebase would benefit from such improvements. The answers to these questions were found in and formed with the help of literature written about programming and maintainability in particular. The parts of the Kriisiosaaja project in need of improvement were found by exploring the project as well as by reading project related documentation and notes made during the development of the project.

The thesis resulted in tips and guidelines for improving the Kriisiosaaja codebase. The tips and guidelines presented within the thesis were formed with the theory found in the literature read. The theory behind the tips was also presented and explained. This helps the future developer of the project to get into the mindset of writing maintainable code and help clear up issues not presented in the thesis.

Keywords: programming, maintainability, best practices, design patterns

SISÄLLYS

1 JOHDANTO

Ohjelmistoprojektien tilaa on toisinaan hyvä tarkastella, sillä näin projektin koodipohjasta voidaan siistiä pois epäselvät osa-alueet, nopeasti toteutetut väliaikaisratkaisut ja huonosti muokattavat tai laajennettavat systeemit. Muutosten jälkeen tarkoitus on, että projektin eteenpäin vieminen on helpompaa. Tämän työn tarkoituksena on tutkia ja tarjota parannusehdotuksia Kriisiosaaja-oppimispelin koodipohjaan, jotta sen jatkokehittäminen ja ylläpitäminen helpottuu.

Kriisiosaaja on oppimispeli, jossa pelaajan on tarkoitus selviytyä kotonaan 72 tuntia kriisitilanteessa, kuten pitkittyneen sähkökatkon aikana, pitämällä hahmon tarpeet täytettyinä. Projektia työstää useampi tiimi ja ensimmäinen tiimi on saanut pelin ensimmäisen prototyypin valmiiksi ja siten päättänyt oman osuutensa projektin parissa. Tämä on hyvä tilaisuus tutkia projektin koodipohjaa ja tarjota korjaus ja parannusehdotuksia, jotta tulevien tiimien on helpompi rakentaa vanhan päälle.

Opinnäytetyön tarkoitus on tutustua Kriisiosaajaan ja arvioida projektin koodipohjaa. Työn tuotoksena syntyy teoriaan pohjaavia neuvoja ja ohjeita pelin ylläpidettävyyden parantamiseksi. Lisäksi teoria ohjeiden taustalla selitetään, jotta ajatusmalli ylläpidettävän koodin takana on helpompi soveltaa projektin edetessä. Projekti on rakennettu Unity-pelimoottorin versiolla 2022.3.20f1 ja kirjoitettu C#-kielellä.

Työ toteutetaan Kaakkois-Suomen ammattikorkeakoulun terveysalan ja ensihoidon koulutusyksikölle osana Kriisiosaaja-projektia. Peliä on tarkoitus käyttää osana Kriisi- ja poikkeusolojen osaaja -opintojaksoa. Valmis opinnäytetyö helpottaa projektin kehitystä.

2 TUTKIMUSASETELMA

Luvussa esitellään tutkimusongelma, -kysymykset ja -ote sekä esitellään, kuinka työssä käytettävä teoreettinen tieto kerätään. Lisäksi selvennetään työn luotettavuuden varmistamista.

2.1 Tutkimusongelma ja -kysymykset

Ohjelmistoprojektit pyritään lähes aina rakentamaan niin, että niiden arkkitehtuuri hyödyntää tunnettuja suunnittelumalleja ja noudattaa hyvän ohjelmoinnin käytänteitä tuottaen helposti ymmärrettävää ja muokattavaa koodia. Kriisiosaajan kohdalla projektin koko ja aikataulurealiteetit suhteutettuna tiimin kokoon estivät tämän, sillä kaikki aika meni projektin työstämiseen, jotta edes prototyyppi saatiin toimintaan ajoissa. Tästä syystä pelin arkkitehtuurin varmistamiseen ja teorian tarkistamiseen ei jäänyt pintatason tutkimusta enempää aikaa. Tässä työssä on tarkoitus tutkia projektin koodipohjaa ja tarjota korjaus- ja parannusehdotuksia, joissa pohjana on teoreettinen viitekehys, joka rakentuu suunnittelumalleista ja muista projektin arkkitehtuurin ja ylläpidettävän koodin kannalta tärkeistä lähteistä.

Tutkimusongelma on projektin ylläpidettävyys ja kuinka sitä voi teoriaa hyödyntäen siistiä ja selventää, jotta lopputuloksena olisi siistimpi, ylläpidettävämpi ja sitä myöten jatkokehitettävämpi projekti. Tästä voi johtaa tutkimuskysymykset:

1. Mitä tarkoittaa ylläpidettävyys ja mitä se pitää sisällään?
2. Miten ylläpidettävyyttä voidaan parantaa?
3. Mitkä osat pelin koodipohjasta kaipaavat hiomista ja miksi?

Työ on tyypiltään kehittämistutkimus, sillä työssä on tarkoitus tuottaa jotain uutta, kuten ohjeita ja neuvoja, projektin parantamiseksi. Kehittämistutkimus on yksi yhdistelmätutkimuksen muodoista, eli siinä on piirteitä sekä laadullisesta että määrällisestä tutkimuksesta. Laadullinen tutkimus pyrkii ymmärtämään ilmiöitä ja työssä tämä näkyy pyrkimyksenä ymmärtää, mitä on ylläpidettävä koodi ja kuinka sellaista tuotetaan. Tähän sisältyy myös ohjelmoinnin hyvien käytänteiden ymmärtäminen. Määrällisen tutkimuksen piirteitä työssä on vähän, sillä niissä pyritään ymmärtämään ja käsittelemään lukuja ja taulukoita. Puhdas laadullinen tutkimus työ ei kuitenkaan ole, sillä laadullinen tutkimus ei pyri muutokseen, vaan ymmärtämään ilmiötä. Tässä työssä on tarkoitus aiheuttaa muutos projektissa parantamalla sen ylläpidettävyyttä tuottamalla neuvoja ja ohjeita projektin koodipohjan parantamiseksi. (Kananen 2015, 39–40.)

2.2 Kehittämistutkimus

Kehittämistutkimukseen kuuluu muutoksen iterointi eli kehittämistutkimuksen sykli. Täysimittaiseen sykliin kuuluu useita vaiheita, jotka kuvataan seuraavaksi. (Kananen 2015, 41.)

Nykytilan kartoitus ja ongelmien havaitseminen: Selvitetään kehitettävän asian nykytila, ja kartoitetaan ongelmat ja parannuskohdat. (Kananen 2015, 41.) Tässä työssä ongelmien havaitseminen on projektin koodipohjan tutkimista ja ongelmakohtien etsimistä. Koodin rakenteesta löydetyt ongelmat ovat myös niitä ongelmia, joita työssä pyritään ratkomaan. Mahdollisia ongelmia ovat esimerkiksi vajavaiset suunnittelumallit tai epäselvät koodiratkaisut.

Suunnittelu: Kun tiedetään, mikä on ongelmana, voidaan pohtia, kuinka ongelma ratkaistaan. Tähän sisältyy eri ratkaisuvaihtoehtojen selvittäminen ja arviointi. (Kananen 2015, 41.) Kehittämistutkimuksen käytännön toteutus tapahtuu kehittämissykleissä, ja suunnittelu on kehittämissyklin ensimmäinen vaihe, sillä varsinainen kehitystyö alkaa tästä (Pernaa 2013, 17). Tässä työssä suunnitteluvaihe on löydettyjen ongelmien eri ratkaisuiden arviointi ja sopivan ratkaisun ehdottaminen.

Toiminta: Suunnittelun tuotoksena syntyy toimintamalli, jonka on tarkoitus viedä kohti tavoitetta. (Kananen 2015, 41.) Tässä työssä aiemman vaiheen ehdotuksia ei toteuteta työn koon ja aikataulun takia, sillä ehdotusten implementointi ja testaaminen vaatii paljon aikaa.

Havainnointi: Suunnitelman toteuduttua muutoksia havainnoidaan. Vaihe tunnetaan myös testaamisena. Tässä vaiheessa varmistetaan tai vähintäänkin kirjataan ylös tehtyjen muutosten vaikutuksia. (Pernaa 2013, 17.) Tässä työssä vaiheeseen kuuluu ehdotettujen muutosten vaikutusten arviointi.

Seuranta: Vaiheessa seurataan ja arvioidaan tehtyjä muutoksia ja vaikutuksia. Vaihe tunnetaan myös arviointina. (Pernaa 2013, 17.) Tässä työssä seuranta ja havainnointivaihe ovat lähekkäin, sillä toiminta vaihe jää työssä väliin.

Tämän työn kehittämissykli koostuu pääasiassa kartoitus-, suunnittelu-, havainnointi- ja seurantavaiheista. Toteutusvaiheen pois jättäminen jättää prosessiin aukon, joka vaikuttaa koko työhön, mutta erityisesti havainnointi- ja seurantavaiheisiin. Työssä havainnointi- ja seurantavaihe yhdistyvät yhdeksi työvaiheeksi, jossa arvioidaan ehdotusten vaikutusta, kuten paljonko työtä ehdotuksen toteuttaminen vaatisi. Aukko ja sen mahdolliset vaikutukset on otettava koko työn ajan huomioon esimerkiksi panostamalla työn luotettavuuteen muilla osa-alueilla.

2.3 Tiedon tarve ja hankinta

Jotta voidaan parantaa projektin ylläpidettävyyttä, tulee tietää ja ymmärtää, mitä tarkoittaa ylläpidettävä ja kuinka projektista saadaan sellainen. Pohjaoletuksena on, että ohjelmoinnin hyvät käytänteet liittyvät ylläpidettävyyteen, joten niitä tulee tutkia ja selvittää. Lisäksi tarkoitus on tutustua suunnittelumalleihin, jotta projektin rakenteesta on helpompi puhua ja sen ymmärtäminen helpottuu.

Tarvittava teoreettinen tieto käsittelee ohjelmistoprojektien ylläpidettävyyttä ja ohjelmointikäytäntöjä ja -tapoja, jotka edesauttavat hyvää ja luettavaa koodia. Tarvittava aineisto, ja aineisto laadullisessa tutkimuksessa yleisestikin, voidaan jakaa sekundääri- ja primääriaineistoon (Kananen 2017, 44).

Sekundääriaineistoa on kaikki, mitä aiheesta on tuotettu ennen tutkimusta, kuten kirjat, artikkelit, videot ja raportit (Kananen 2017, 44). Tämän työn sekundääriaineisto koostuu pääosin ohjelmoinnin kirjallisuudesta. Kirjallisuus on kerätty epätieteellisten keskusteluiden, kuten nettikeskusteluiden, lähteitä ja lukusuosituksia seuraamalla. Myös kirjallisuudessa itsessään on usein suosituksia lisälukemiseksi. Internetlähteistä hyödynnetään luotettavien tahojen ohjelmointidokumentaatiota ja -neuvoja. Luotettavia tahoja ovat esimerkiksi Microsoftin ja Unityn dokumentaatio. Lisäksi aiheita puidaan paljon erinäisillä keskustelufoorumeilla ja -alustoilla, mutta ne eivät ole tieteelliseen tekstiin sopivia lähteitä.

Primääriaineisto on aineistoa, joka on tuotettu työtä varten tai sen ohessa, kuten havainnot, haastattelut, kyselyt ja kokeilut (Kananen 2017, 44). Tämän

työn primääriaineisto on projektin kehityksen aikana tuotettu materiaali, kuten muistiinpanot, työpäiväkirjat, tuntikirjanpidot, raportit ja projektin dokumentaatio.

2.4 Luotettavuusarvio

Tieteelliseen työhön kuuluu luotettavuuden arviointi. Kehittämistutkimuksen luotettavuuden arviointiin ei ole suoria työkaluja, sillä kyseessä on yhdistelmätutkimuksen muoto. Käytettävät luotettavuuden mittarit on valittava sen mukaan, mitä menetelmiä tutkimuksessa käytetään. (Kananen 2015, 111.)

Tavanomaisen tieteellisen tutkimuksen luotettavuutta arvioidaan validiteetin ja reliabiliteetin kautta. Validiteetti tarkoittaa pätevyyttä, eli että tutkimus kohdistuu siihen, mitä haluttiinkin tutkia, ja reliabiliteetti tarkoittaa tulosten luotettavuutta ja toistettavuutta. Käsitteet ovat kehittyneet määrällisen tutkimuksen maailmassa, eivätkä siten ole suoraan sopivia tähän työhön, sillä työssä on merkittävästi enemmän laadullisen tutkimuksen piirteitä. (Pernaa 2013, 18.) Työn reliabiliteettia on tarkoitus arvioida, jotta voidaan pohtia työn luotettavuutta kokonaisuudessaan. Validiteetin arviointiin liittyy pohdinta, olisiko samaan lopputulokseen päästy ilman työn projektiin tuomaa muutosta (Kananen 2017, 70).

Työn luotettavuuden päämittarina toimii arvioitavuus, joka tunnetaan myös dokumentaationa. Se tarkoittaa työn aineistojen ja johtopäätösten dokumentointia eli perusteluja. Näin lukija voi halutessaan seurata johtopäätösten polkua ja varmistua työn tieteellisyydestä. (Kananen 2015, 115.) Tässä työssä arvioitavuus näkyy tehtyjen päätösten taustasyiden selventämisenä ja auki selittämisenä, jotta mahdolliset virheet tai vaihtoehtoiset polut ovat seuraavalle tutkijalle helpompi löytää. Tämä vaikuttaa projektin reliabiliteetin arviointiin, sillä dokumentoitu työ on helpompi toistaa tarvittaessa. Koska työssä ei käydä läpi kehittämissyklin kaikkia vaiheita, on arvioitavuus erityisen tärkeä mittari seurata ja arvioida, jotta työn aikana tehdyt havainnot ja arviot pysyvät luotettavina.

3 TEOREETTINEN VIITEKEHYS

Työn teoreettinen viitekehys pohjaa ohjelmointiprojektien ylläpidon ja selvyyden parantamiseen keskittyvien kirjojen ympärille. Avaintermeinä ovat ylläpidettävyys ja coupling eli kytkennät. Lisäksi selvitetään, mitä arkkitehtuuri tarkoittaa ohjelmistomaailmassa.

3.1 Arkkitehtuuri

Arkkitehtuuri viittaa työssä ohjelmistoarkkitehtuuriin, joka on joukko rakenteita, joiden avulla järjestelmää voidaan kuvata ja siitä voidaan puhua. Rakenne tarkoittaa järjestelmän eri osia, näiden välisiä yhteyksiä ja osien sekä yhteyksien ominaisuuksia. (Clements ym. 2011, 1.) Ohjelman arkkitehtuuritason suunnittelu auttaa tuottamaan ylläpidettävää koodia (Visser 2016, 9).

Kriisiosaaja on rakennettu Unity-pelimoottorin päälle, joten kehittäjän ei tarvitse murehtia arkkitehtuurin syvällisistä osista, kuten pelaajan syötteen vastaanottamisesta tai grafiikan piirtämisestä. Näin aika ja energia voidaan keskittää pelin omaan arkkitehtuuriin, kuten toimintoihin ja niitä ympäröivään rakenteeseen. Työ keskittyy näihin arkkitehtuurin osiin, eikä tarkoitus ole analysoida Unity-pelimoottoria. Projektin tärkeimmät arkkitehtuurin osat esitellään luvussa 4.

3.2 Coupling eli kytkennät

Coupling eli kytkentä kuvaa metodien, luokkien tai suurempien koodipohjien välisien yhteyksien määrä ja laatua (ISO-24765:2017). Luokka on rakenne, jossa on muuttujia, eli dataa, ja metodeja eli tapoja käsitellä ja tehdä asioita tällä datalla tai tarjota muuta toiminnallisuutta (Codecademy 2023a). Decoupling tarkoittaa koodiyksiköiden välisien yhteyksien vähentämistä. Kaikki tämän työn lähteinä käytetyt ohjelmoinnin kirjat ohjaavat vähentämään kytkentöjä.

Jos koodi A on tiukasti kytköksissä osaan B ja koodi A kaipaa muutoksia, täytyy ohjelmoijan tutkia ja ymmärtää kumpikin osa-alue, jotta hän osaa tehdä muokkauksia osaan A ilman, että osa B hajoaa seurauksena. Isomman koodimäärän lukeminen ja sisäistäminen vaatii kehittäjältä enemmän, joten muokkausten tekeminen on hidasta, eikä kokemattomampi ohjelmoija koe voivansa

lähteä muokkaamaan monimutkaista järjestelmää. A tietää liikaa B:n koodista ja käyttää sitä omanaan ja mahdollisesti toisinpäin. Löyhässä kytkennässä A ja B voivat yhä keskustella keskenään, tämä on usein välttämätöntä, mutta A tietää B:stä vain sen, mitä se toimintaansa vaatii, muu osa on piilotettu näkyvistä. Löyhä kytkentä vaatii usein arkkitehtuuritason ratkaisuja, jossa koodin eri osat on rakennettu näyttämään ulos vain tarpeelliset osat ja hyödyntämään muista vain oleelliset osat. (Gamma ym. 1995, 24–25.)

3.3 Ylläpidettävyys

Jotta projektia voidaan jatkokehittää, tulee sitä voida ylläpitää, sillä uusien ominaisuuksien lisääminen johtaa väistämättä vanhan koodin muokkaamiseen (Visser 2016, 54). Ylläpidettävyys on ohjelmiston laadun mittari, jolla seurataan, kuinka helppoa ohjelmiston koodia on tulkita ja tarvittaessa muokata. (Visser 2016, 2). Jos ohjelman koodiin on helppo tehdä muokkauksia, kestää yksittäisen ominaisuuden lisäämisessä tai virheen korjaamisessa vähemmän aikaa, jolloin kehittäjälle jää aikaa useammalle korjaukselle tai ominaisuudelle. Ylläpidettävyyteen liittyy useita ohjelmoinnin osa-alueita, kuten eri koodilohkojen välisten kytkentöjen vähentäminen. Ohjelmoinnin hyvät käytänteet ovat myös oleellisia ylläpidettävyyden kannalta (Visser 2016, 127).

Building Maintainable Software (Visser 2016) esittelee 10 ohjenuoraa ylläpidettävän koodin kirjoittamiseen. Ohjenuorat rakentuvat kolmen periaatteen päälle:

1. Ylläpidettävyys hyötyy parhaiten yksinkertaisista ohjeista, se ei parane itsestään uudella hienolla teknologialla.
2. Ylläpidettävyys ei ole jälkikäteen tehtävä lisäys, vaan projektitason elämäntapa, jossa jokainen muokkaus ja lisäys on tärkeä osa kokonaisuuden ylläpidettävyyttä.
3. Kaikki ylläpidettävyysrikkeet eivät ole saman arvoisia. On eroa, onko yksittäinen luokka vai yksittäinen metodi vaikeasti ylläpidettävä. (Visser 2016, 4–5.)

Ohjenuorat ovat siinä järjestyksessä, missä kirja ne esittelee:

1. Kirjoita pieniä koodiyksiköitä. Koodiyksikkö (Unit of code) tarkoittaa metodeja ja konstruktoreita. Niiden tulisi olla korkeintaan 15 riviä, jotta niiden analysointi, testaaminen ja uudelleenkäyttäminen helpottuu (Visser 2016, 11). Yksinkertaistetusti konstruktori on erikoismetodi, jota kutsutaan, kun luokasta luodaan uusi instanssi (Microsoft 2023a).

2. Kirjoita yksinkertaisia koodiyksiköitä. Yksinkertainen koodi on helpompi ymmärtää. Yksinkertaisessa koodissa on vain muutama haarautumiskohta, Visser suosittelee maksimissaan neljä haaraa per yksikkö (2016, 29). Haarautumiskohtia ovat mm. if, case ja erinäiset loopit (silmukka).
3. Kirjoita koodi kerran. Koodia ei tule leikkaa/liittää ympäri projektia, sillä tällöin mahdolliset muutokset tulee tehdä erikseen jokaiseen kopioituun paikkaan. Kirjota uudelleenkäytettävää koodia ja uudelleen käytä olemassa olevaa metodia.
4. Pidä yksikön parametrimäärä pienenä, maksimissaan 4 per yksikkö (Visser 2016, 57). Pitkät parametrilistat kielivät ongelmasta yksikön sisällä, kuten monta eri asiaa tekevästä metodista. Parametrimäärää voi vähentää pakkaamalla usein yhdessä käytetty data yhteen luokkaan. Parametri on metodille annettu data, kuten muuttuja, jota voidaan käyttää kyseisen metodin sisällä (C# Method Parameters s.a.).
5. Jaa vastuu luokkien välillä. Näin vältetään suuria ja tiukasti toisiinsa sidottuja luokkia.
6. Pidä arkkitehtuuritason sidokset vähäisinä. Näin ohjelmaan muodostuu jo korkeimmalla tasolla erillään olevia yksiköitä, joiden ylläpito helpottuu.
7. Pidä arkkitehtuuritason yksiköt keskenään tasapainossa sekä koossa että vastuussa, jottei yksittäinen yksikkö kasva muita suuremmaksi. Tämä helpottaa ylläpitoa, sillä eri osia voi työstää samanaikaisesti ja jokaisella osalla on omat tehtävänsä.
8. Pidä koodipohja pienenä. Virheen juurisyy on vaikeampi löytää isosta koodipohjasta. Iso koodipohja on myös vaikeampi ylläpitää, sillä jokainen muutos vaatii suuremman koodipohjain ymmärtämistä, muokkaamista ja testaamista.
9. Automatisoi testausta. Automaattiset testit antavat lähes välitöntä palautetta muutoksen toimivuudesta. Käsin tehty testaus skaalautuu huonosti suuremmissa projekteissa ja suurien muutosten yhteydessä.
10. Kirjoita puhdasta koodia. Muun muassa vältä harhaanjohtavia kommentteja, siivoa kommentoitu koodi pois ja käytä kuvaavia nimiä sekä muuttujille, metodeille että luokille. Kommentit ovat kooditiedostoissa olevaa tekstiä, jota ei suoriteta. Kommenteissa voidaan muun muassa selittää koodia tai ottaa osa koodista pois käytöstä esimerkiksi nopeaa testausta varten. (C# Comments s.a.)

Näitä ohjenuoria (Visser 2016, 9) avataan tarkemmin sitä mukaan, kun ne tulevat projektia tutkiessa esille. Kaikkia ohjenuoria ei käydä tämän työn lomassa läpi, vaan tarkoitus on keskittyä projektin kannalta oleellisiin ohjeisiin.

Vaikka ohjenuorat ovatkin vain yhdestä kirjasta, antavat ne hyvän pohjan projektin ylläpidettävyyden ja jatkokehitettävyyden arvioinnille, sillä Robert Martin käy kirjassaan Clean Code: A Handbook of Agile Software Craftsmanship (2009) hyvin pitkälti samoja asioita. Kirjan näkökulmana on ylläpidettävyyden alaotsikko, hyvä koodi, joten asioihin paneudutaan syvemmin. Kumpikin kirja antaa neuvoja hyviin kommentointikäytänteisiin, muuttujien nimeämiseen ja

metodien suositeltuihin pituuksiin. Asiat, joista Visser (2016) puhuu osana kymmenettä ohjenuoraa, on avattu paljon syvemmin Martinin (2009) kirjassa.

3.4 Hyvät käytänteet

Ohjelmoinnin hyvät käytänteet tarkoittavat ohjelmointityylejä ja -tapoja, joiden avulla ja joita noudattamalla tuotetaan hyvää koodia. Hyvä koodi tarkoittaa luettavaa ja ymmärrettävää koodia, joka etenee helposti seurattavalla logiikalla. (Martin 2009 7–12.) Ohjelmoinnin saralla on yleisesti hyväksyttyjä käytänteitä, joista on kirjoitettu useita kirjoja, kuten Building Maintainable Software (Visser 2016) ja Clean Code (Martin 2009). Näiden lisäksi Martin Folwer, Kent Beck, John Brant, William Opdyke ja Don Roberts esittelevät kirjassaan Refactoring: Improving the Design of Existing Code (2002) tapoja koodin parantamiseksi. Näiden lisäksi käytänteitä puidaan myös paljon erinäisillä keskustelualustoilla, sillä jokaisella on hieman oma käsityksensä, mikä on hyvä käytäntö. Nämä keskustelut eivät sovellu lähteiksi, ja tämän työn hyvät käytänteet ovat ohjelmoinnin kirjoista.

Hyvästä koodista ja olemassa olevan koodin parantamisesta puhuttaessa puhutaan usein termistä code smells eli ”koodihajuista”. Termi ja ajatus pohjaa konseptiin, jossa kokenut ohjelmoija koodia silmäillessään huomaa jotain, mikä ei ole hyvästä koodin ylläpidettävyyden kannalta. Usein havaittu asia itsessään ei ole pahan alku taikka juuri, vaan kielii syvemmästä ongelmasta. Esimerkiksi large class smell (Suuren luokan haju) kuvaa luokkaa A, joka on päässyt kasvamaan turhan suureksi (Fowler ym. 2002, 63). Luokka A saattaa olla suuri, sillä se tekee toisistaan riippumattomia asioita, kuten esimerkiksi hiiren klikkaamisen seuraamuksia ja pelaajahahmon tarpeiden säätämistä. Esimerkissä hiiren klikkaamisen logiikka ja tarpeiden säätämisen logiikka eivät liity mitenkään toisiinsa, mutta luokka A on vahvasti kytkettynä kumpaankin ja kytkösten on jo todettu olevan pahasta luvussa 3.2. Suuri luokka ei itsessään ole ongelma, mutta harva luokka kasvaa kovinkaan suureksi, mikäli esimerkiksi Visserin (2016) ohjeita ylläpidettävään ohjelmistosuunnitteluun noudatetaan.

Työssä ei ole tarkoitus koota listaa hyvistä ohjelmointikäytänteistä, vaan tutkia, onko projektin koodi hyvien käytänteiden mukaista. Merkittävimmät käytänteet on tarkoitus esitellä ja merkittävyys määräytyy esimerkiksi sen perusteella, kuinka paljon se näkyy koodissa tai kuinka paljon projektin koodia joutuisi muuttamaan, jotta se vastaisi käytännettä.

3.5 Suunnittelumallit

Design pattern eli suunnittelumalli on tapa ratkoa ongelma yleistetyllä tavalla, joka sopii useampaan tilanteeseen ja jonka kommunikointi on helpompaa (Gamma ym. 1995, 2–3). Suunnittelumalleja voi verrata kirjailijoiden juoniarkkityyppeihin, kuten kolmiodraamaan. Kolmiodraama kertoo käsitteenä, että tarinassa kaksi ihmistä rakastaa kumpikin kolmatta (Cambridge Dictionary: Love triangle s.a.). Samalla tavalla singleton-suunnittelumalli kertoo, että on luokka, joka instantioidaan eli luodaan ohjelman ollessa käynnissä vain kerran ja että kaikilla ohjelmassa on pääsy luokkaan (Gamma ym. 1995, 127).

Tarkoitus ei ole pakottaa projektia tai sen osa-aluetta mihinkään malliin, vaan tutustua vaihtoehtoihin ja pitää silmät auki, jos projektin jossain osa-alueessa voi hyödyntää tai on hyödynnetty suunnittelumalleja. Suunnittelumalleista kertovan kirjallisuuden pohja ohjaa usein Erich Gamman, Richard Helmin, Ralph Johnsonin ja John Vlissidesin kirjoittamaan Design Patterns: Elements of Reusable Object-Oriented Software -kirjaan (1994), jossa esitellään 23 eri ohjelmointi- ja suunnittelumallia sekä ajatus kunkin mallin taustalla, miten malli toteutetaan, mihin sitä yleensä käytetään ja mitä malleja käytetään usein yhdessä. Kirja on verrattain vanha, mutta sen opit ovat yhä oleellisia. Useampi tätä työtä varten luettu kirja suosittelee mainitun kirjan lukemista.

3.6 Unity ScriptableObject

Unity ScriptableObject (”SkriptattavaEsine”), työssä usein SO-esine, on luokka, josta periyttämällä kehittäjä voi luoda luokkia, jotka eivät ole sidottu mihinkään pelimaailman esineeseen tai Unity sceneen, eli kenttään. SO-esineeseen ja sen sisältöön pääsee käsiksi lähes mistä tahansa projektin osasta. (Unity Technologies 2024.) Universaaliuden takia SO-esineitä käytetään usein datan säilyttämiseen. Esimerkkinä peli, jossa vihollisen tiedot ja kyvyt tallenne-

taan SO-esineeseen. Nyt jokainen pelin vihollinen voi tarkistaa arvonsa yhdestä luokasta, sen sijaan että jokaisella vihollisella olisi oma kopio näistä tiedoista. Tarvittavan muistin määrä on vähentynyt. (Unity Technologies 2018.)

Kriisiosaajassa SO-luokan johdannaisia käytetään lähinnä skenaariodatan säilyttämiseen ja osana event channel -arkkitehtuuria. Event channel -arkkitehtuuri esitellään luvussa 4.1, skenaariodatan säilyttämistä ei tässä työssä käsitellä. Arkkitehtuuri on toteutettu SO-esineillä, sillä ne ovat yksittäisistä sceneistä tai skripteistä irrallaan ja niitä voi käsitellä helposti Unity Editorissa. Unity Editor on pelimoottorin muokkaus ja kehityspuoli.

4 KRIISIOSAAJAN ARKKITEHTUURIN YDINPILARIT

Luvussa on tarkoitus tutustua projektin arkkitehtuuriin ja esitellä peliä kannattelevat ydinpilarit ja arvioida niiden laatua. Pilareita tutkiessa kiinnitetään huomiota piirteen nykyiseen rakenteeseen ja sitä verrataan mahdollisiin suunnittelumalleihin tai muihin yleisiin käytänteisiin. Mikäli pilari noudattaa mallia tai muuta yleistä käytännettä, avataan mallin tarkoitusta. Jos mallia ei noudateta täysin, avataan nykyisen toteutuksen puutteita. Muu yleinen käytänne tarkoittaa tässä esimerkiksi koodihajuja ja ohjelmoinnin hyviä käytänteitä. Käytänteitä hyödynnetään myös, kun analysoidaan koodin selkeyttä.

4.1 Event channel -arkkitehtuuri

Event channel -arkkitehtuuri on tapa toteuttaa luokkien välistä kommunikaatiota kytköksiä välttäen. Projektissa kanavia käytetään melko paljon, mutta niiden käytössä vaikuttaisi olevan ongelma, joka esitellään ja johon tarjotaan ratkaisua.

4.1.1 Event channnel -arkkitehtuurin teoria

Projektissa laajalti käytetty event channel -arkkitehtuuri (EC, ”Tapahtumakanava-arkkitehtuuri”) on observer-suunnittelumallin (”Tarkkailija”) ilmenemismuoto. Mallissa luokka voi aiheuttaa tapahtuman (Raise event), jota kyseisestä tapahtumasta kiinnostuneet luokat voivat seurata (Gamma ym. 1995, 293.) Kun tapahtuma aiheutetaan, saavat kaikki tapahtumaa seuraavat siitä ilmoituksen, ja ne suorittavat tapahtumaan sidotut toimintonsa. Nystrom

(2014) käyttää esimerkkinä peliä, johon lisätään saavutuksia, joista yksi on sillalta putoaminen. Pelin fysiikkamoottoriin voidaan lisätä pätkä koodia, joka aiheuttaa tapahtuman, kun jotain putoaa sillaksi luokitellulta tasolta. Saavutuksista vastaava järjestelmä seuraa tätä tapahtumaa ja tarkistaa, mitä sillalta putosi, ja jos kyseessä on pelaajahahmo, saa pelaaja saavutuksen putoamisesta. Työssä event channel -arkkitehtuurista puhutaan suomen kielen termein kanava, tapahtumakanava ja tapahtumakanava-arkkitehtuuri. Termit eivät ole virallisia, ja lisätietoja löytää englanninkielisellä termillä event channel architecture.

Suunnittelumallissa tapahtuman aiheuttajaa eli kohdetta (Subject) ei kiinnosta kuinka moni tarkkailee (Observe) tapahtumaa, eikä tarkkailijoita kiinnosta, kuka tapahtuman aiheuttaa, vaan vain se, että tapahtuma aiheutuu. Arkkitehtuurin etuna on luokkien välisten sidosten vähentäminen ja tapahtumien ilmoittaminen usealle luokalle kerralla (Gamma ym. 1995, 293). Sen sijaan, että asiasta kiinnostuneet tarkkailijat kyselevät säännöllisesti, kukin erikseen, kohteelta sen tilaa, kohde ilmoittaa kaikille yhtä aikaa, mikäli sen tila muuttuu. Tarkkailijat itsenäisesti päättävät, mitä ne tilamuutoksella tekevät.

Unity EC-arkkitehtuurissa luodaan kanavaesine (Channel Object), joka periytyy Unityn ScriptableObject-luokasta. Unity Editorissa kanavaesine raahataan osaksi koodikomponenttia, joka aiheuttaa tapahtuman, sekä kaikkia kyseisestä tapahtumasta kiinnostuneita koodikomponentteja. (Unity 2017.) Koditasolla kanavaesineessä on metodi, jota kohde kutsuu. Kuvassa 1 näkyy metodin kutsu, jossa kanava kuljettaa mukanaan pelaajan nälkätarpeen arvon. Kuvassa 2 näkyy kanavaesineessä oleva metodi, jossa tarkistetaan, että kanavalla on tarkkailijoita. Tämä on yksi kanava-arkkitehtuurin ongelmista Kriisiosaajassa, siitä lisää luvussa 4.1.3. Lopuksi kutsutaan tarkkailijassa kanavaesineeseen sidottua metodia. Sitominen näkyy kuvassa 3. Yhteen kanavaan voi sitoa useita metodeja. Seurauksena koodiosaset eivät keskustele toisilleen missään vaiheessa, vaan tapahtuma aiheutetaan kanavaesineessä, joka puolestaan kutsuu niitä metodeja, jotka tarkkailijat esineeseen sitoi. (Unity 2017.)

```
onHungerChangeEC.RaiseEvent(playerHunger);
```

Kuva 1. Koodirivi, jossa aiheutetaan tapahtuma, kun pelaajan nälkätarve muuttuu. Suluissa tapahtuman mukana annetaan pelaajan uusi nälkätarpeen arvo.

```
31 references
public void RaiseEvent(T parameter)
{

    if (OnEventRaised == null)
    {
        Debug.LogWarning("Event channel error, OnEventRaised is null for channel with parameter value" + parameter.ToString());
        return;
    }

    OnEventRaised.Invoke(parameter);

}
```

Kuva 2. Metodi, joka kanavan sisällä kutsutaan tapahtumaa aiheuttaessa. OnEventRaised.Invoke(parameter) kutsuu kanavaan sidotut metodit, eli suorittaa tapahtuman tarkkailijoissa metodin.

```
onHungerChangeEC.OnEventRaised += OnHungerChange;
```

Kuva 3. Koodirivi, jossa metodi OnHungerChange sidotaan onHungerChangeEC-kanavaan.

Projektissa EC-arkkitehtuuria käytetään paljon, sillä projektin tavoitteena oli suunnitella järjestelmä, jossa koodikytkökset pyritään pitämään löyhinä. Pääasiassa projekti saavutti tavoitteensa, sillä pelissä on useampi alue, jossa kanavat aidosti vähentävät kytköksiä ja sallivat toisistaan erillään olevien järjestelmien vaikuttaa toisiinsa ilman merkittäviä kytköksiä.

4.1.2 Ongelma Event Channel -arkkitehtuurin implementaatiossa

Kanavia käytetään projektissa toisinaan epäideaalilla tavalla. Sillä projektin aikataulu oli melko tiukka suhteutettuna tavoitteisiin ja tiimin kokoon, ja projektin aikana tapahtui paljon oppimista, ei myöhemmin opittua ollut aikaa implementoida osaksi aiemmin tehtyä koodia. Tämä näkyy erityisesti EC-arkkitehtuurissa. Projektin loppupuolella kanavia käytettiin esimerkiksi toteuttamaan pisteytys, jonka ajatus vastaa pitkälti Nystromin (2014) Observer-mallissa esiteltyä esimerkkiä saavutuksista. Halutussa kohtaa aiheutetaan tapahtuma, jota pisteytys seuraa ja toimii tilanteen vaatimalla tavalla. Pisteytysluokka ja pisteytyksen aiheuttaja eivät ole sidoksissa toisiinsa.

Kehityksen varhaisessa vaiheessa toteutettiin järjestelmä, jossa sekä kaikki pelaajan tarpeet että maailman tilaa kuvaavat arvot noudetaan ja muokataan kanavien kautta. Arvoja ovat esimerkiksi pelaajan nälkätarve tai maailman ulkolämpötila. Rakenteen tavoite oli, että näihin luokkiin pääsee suoraan käsiksi vain harva ja valtaosa yhteyksistä tapahtuisi kanavien kautta eli siis että luokat

olisivat löyhästi kytkettyinä. Tavoitteeseen päästiin, eikä luokissa ole juurikaan ulkoisia yhteyksiä. Ongelma on kyseisten kanavien turhuus, kun pelin rakennetta katsoo isomassa mittakaavassa. Projektissa on InteractionManager (”Toimintomanageri”), joka hoitaa toimintojen ulkoiset yhteydet. Pähkinänkuoressa toiminnot, kuten mene kävelylle, tarkastavat ja muokkaavat pelaajan tarpeita ja maailman tilaa InteractionManagerin kautta, jottei jokaisen toiminnon tarvitse erikseen toteuttaa omaa tarkastusmetodiaan. Sillä InteractionManagerilla on jo yhteydet muokata ja lukea pelaajan tarpeita, hyödyntää moni pelin järjestelmä sitä. Käytännössä rakenteessa on tiukka kytkös, jota kanavilla pyrittiin välttämään. Sen sijaan, että kaikki esimerkiksi pelaajan tarpeista kiinnostuneet ovat yhteydessä pelaajan ja säätävät arvoja suoraan sitä kautta, ovat ne nyt yhteydessä InteractionManageriin.

Lisäksi projektin kanavat eivät tätä kirjoittaessa toimi, mikäli kanavalla ei ole tarkkailijaa eli mikäli kanavaesineessä olevaa metodia ei ole sidottu yhteenkään metodiin. Kanavat eivät siis täysin noudata observer-suunnittelumallia, sillä malliin kuuluu välinpitämättömyys kohteen ja tarkkailijan välillä. Tapahtumia aiheutetaan ja niitä odotetaan välittämättä siitä, kuunteleeko kukaan tai aiheutuuko tapahtuma koskaan. (Nystrom 2014.) Malli on vain osittain käytössä, sillä kanavien debugaaminen eli virhekorjaaminen oli helpompaa, kun näki heti, jos kanavaa ei ole sidottu osaksi tarkkailijaa. Projektissa ei nykyisellään ole tilanteita, joissa aiheutuisi tapahtuma kanavalle, jolla ei ole tarkkailijaa. Kommunikaation ja selvyyden nimissä on suositeltavaa muuttaa rakennetta niin, että kanavat toimivat ilman tarkkailijoita, koska silloin projektin voidaan sanoa hyödyntävän observer-ohjelmointimallia. Tämä helpottaa projektin koodin kuvailua ja sisäistämistä. Kanavan, jossa aiheutetaan tapahtuma ilman tarkkailijoita, kuuluisi olla aiheuttamatta virhettä. Viesti lokiin riittää kertomaan kehittäjälle, että kanavalla ei ole tarkkailijoita, jolloin kehittäjä voi tarkistaa, pitäisikö kanavalla olla tarkkailijoita vai ei.

4.1.3 InteractionManager-ongelma

Ongelma sijaitseekin EC-arkkitehtuurin sijaan InteractionManagerissa ja usean luokan kytköksessä manageriin. Sillä kyseistä manageria käyttää tarpeiden lukemiseen ja kirjoittamiseen toimintojen lisäksi esimerkiksi pisteytys-

järjestelmä, on InteractionManager tiukasti kytköksissä useaan luokkaan. InteractionManagerilla on toinenkin merkittävä ongelma, sillä luokassa on metodeja useisiin toisistaan riippumattomiin tehtäviin. Toisistaan riippumattomuus tarkoittaa tässä sitä, että metodien kutsut tulevat pelin eri osa-alueilta ja ne vaikuttavat eri osa-alueisiin. Osa-alueet, joihin InteractionManagerilla on metodi:

- Tarpeiden ja maailman tilaa kuvaavien arvojen lukeminen ja kirjoittaminen. Tästä on jo tässä luvussa puhuttu.
- Toimintojen valitsemiseen liittyvää logiikkaa, kuten toimintovalikon näyttäminen ja valitun toiminnon aloittaminen. Tähän osa-alueeseen liittyy useampi metodi, joista osa kutsutaan hiiren klikkausta tarkkailevasta luokasta ja osa useasta toiminnosta.
- Toiminnon lopettamiseen liittyvä metodi, jota kaikki toiminnot kutsuvat.
- Käyttöliittymän Inventory-valikon (Tavaraluettelo) tekstin päivittäminen. Metodikutsu tulee muutamasta toiminnosta.
- Ilmoitusten näyttämiseen liittyvä metodi, joka kutsutaan useista luokista, kuten joistakin toiminnoista ja toisesta managerista. InteractionManager ei toteuta logiikkaa asiaan liittyen, vaan kutsuu sitä luokkaa, joka hoitaa ilmoitusten näyttämiseen liittyvän logiikan.

Listasta huomaa, että luokka tekee paljon erilaisia tehtäviä, joita kutsutaan eri puolilta koodipohjaa. InteractionManager on siis tiukasti kytköksissä useisiin luokkiin. Kytköksen laatuun vaikuttaa sekä laajuus että vahvuus (ISO-24765:2017). Kytkökset InteractionManageriin ovat lähtökohtaisesti tiukkoja, sillä se on yhteydessä useaan luokkaan ja managerilla on paljon julkisia metodeja ja muuttujia. Julkinen (public) on luokan, metodin tai muuttujan edessä oleva avainsana, joka määrittää, kuinka muut luokat tai metodit pääsevät kyseiseen esimerkiksi muuttujaan käsiksi. Julkista muuttujaa voi lukea ja kirjoittaa mikä luokka tahansa, kunhan niillä on tieto luokasta, jossa metodi on. (Microsoft 2024b.) Tämän seurauksena projektissa on useita luokkia, jotka pääsevät käsiksi tietoihin, joihin niillä ei ole mitään asiaa. Esimerkiksi pelaajan nälkätarvetta täyttävän toiminnon ei kuuluisi päästä käsiksi toiminnon valitsemiseen liittyvään logiikkaan. Yksi koodihajuista eli huonon koodin tunnusmerkeistä on liiallinen tiedonjako, jossa luokka antaa liikaa metodeja ja muuttujia ulkoiseen käyttöön. Koodihaju tunnetaan nimellä Too much information, eli liikaa tietoa. (Martin 2009, 291–292.)

4.1.4 Ongelman ratkaisuehdotus

EC-arkkitehtuurissa ja InteractionManagerissa on useita ongelmia, alla tiivistys niistä:

- EC-arkkitehtuuria käytetään noutamaan ja kirjoittamaan usein tarvittuja arvoja. Kanavien lisääminen arvoista kiinnostuneille on hidasta, joten kanavat sidottiin luokkaan, joka vastaa tiedon noudosta ja kirjoittamisesta useille aiheesta kiinnostuneille (InteractionManager).
- Projektin edetessä arvoista kiinnostui yhä useampi luokka, kanavien lisäämisen hitauden ja aikataulukiireiden vuoksi kiinnostuneet ohjattiin olemassa olevaan luokkaan. Lisäksi luokan vastuut kasvoivat projektin edetessä tuoden luokalle muitakin tehtäviä.
- Lopputuloksena on luokka, johon on kytkettynä paljon eri luokkia, jotka tekevät luokan avulla toisistaan riippumattomia asioita. Samalla ne näkevät kaiken muunkin.

Kanavia itsessään ei tarvitse eikä kannata projektista tai edellä kuvatusta järjestelmästä poistaa. Niistä on oikein käytettynä hyötyä, ja iso osa käyttökohteista on hyödyllisiä. Projektin pisteytysjärjestelmä on esimerkki kanavia tehokkaasti hyödyntävästä järjestelmästä. Lisäksi esimerkiksi tarpeiden lukemisen ja kirjoittamisen muuttaminen käyttämään jotain muuta järjestelmää vaatisi paljon koodin uudelleensuunnittelua ja -kirjoittamista ympäri projektia. Tähän kuluisi paljon aikaa, sillä uusi järjestelmä pitäisi suunnitella, toteuttaa ja testata.

InteractionManagerilla on useita tehtäviä, joten tehokas tapa vähentää luokan kytköksiä on jakaa tehtävät eri luokkiin. Näin esimerkiksi toimintojenvalintalogiikka ja tarpeiden lukemislogiikka eivät ole samassa luokassa, eikä toisesta kiinnostunut voi vahingossa vaikuttaa myös toiseen. Visser (2016, 67) ohjeistaa välttämään suuria luokkia, tavoitteena löyhä kytkentä niiden välillä. Yksi mahdollinen korjaustapa on vastuiden jakaminen eri luokkiin (Visser 2016, 73). Ehdotus sopii InteractionManagerin ongelmaan, sillä luokassa on selvästi eri osa-alueiden tehtäviä. Alla ehdotus luokista, joihin nykyisen InteractionManagerin voisi jakaa. Lopullinen päätös on aina tehtävä projektin sen hetkisen tilan, tarpeiden ja aikataulun pohjalta.

Pelaajahahmon tarpeiden ja maailman tilaa kuvaavien arvojen lukeminen ja kirjoittaminen siirretään omaan luokkaansa. Kanava-arkkitehtuuri pysyy sellaisenaan ja arvoista kiinnostuneet voivat hyödyntää tätä uutta luokkaa. Tämä

uusi luokka olisi kytkettynä arvoista kiinnostuneisiin luokkiin, mutta nyt kiinnostuneilla luokilla ei olisi pääsyä niille kuulumattomiin osiin, sillä ne sijaitsevat muissa luokissa.

Toimintojen valitsemiseen ja aloittamiseen liittyvä logiikka siirretään hiiren klikkaamista tarkkailevaan luokkaan, sillä se on ainut paikka, josta metodeja kutsutaan ja johon ne liittyvät. Kyseiset metodit eivät vaadi InteractionManagerista mitään, joten niiden ei tarvitse olla siellä. Näin vältetään kytkös hiiren klikkaamista vastaavan luokan ja InteractionManagerin välillä.

Käytännössä InteractionManageriin jää metodit, joiden kautta toiminnot kommunikoivat muiden luokkien kanssa. Esimerkkinä InteractionManagerissa nyt oleva käyttöliittymän tavaraluettelon päivittäminen, jota muutama toiminto kutsuu. Kyseisen kaltaiset metodit ovat omassa luokassaan, jottei jokaisen toiminnon, joka vaatii tavaraluettelon päivittämistä, tarvitse implementoida omaa versiotaan samasta asiasta. Kyseessä olisi koodin toistoa, ja se on pahasta, sillä se on tehokas tapa huomaamatta lisätä ja levittää virheitä koodipohjaan (Martin 2009, 48). Luokka tulisi nimetä uudelleen, sillä se ei manageroi toimintoja, vaan toimii niiden yhteytenä muuhun peliin. Esimerkiksi InteractionOutsideConnections (”ToimintoUlkoinenYhteys”) kuvaa luokan tehtävää. Luokkien ja metodien työtä kuvaavat nimet ovat yksi tapa parantaa koodipohjaa ja sen käytettävyyttä (Martin 2009, 18.)

4.2 Perusluokat ja perintä

Perusluokat ja perintä on C#-ohjelmoinnin peruskonsepti. Projektissa on toteutettu sen avulla melko paljon oleellisia rakenteita, joiden toimivuutta tarkastellaan ja ongelmia ratkotaan tässä luvussa.

4.2.1 Perinnän teoria

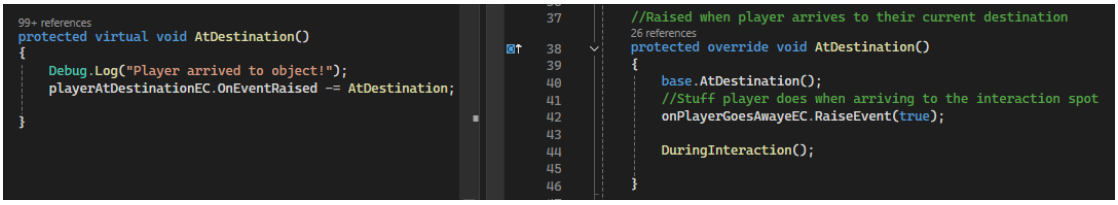
Ohjelmoinnissa perintä tarkoittaa tilannetta, jossa uusi luokka luodaan laajentamalla jo olemassa olevaa luokkaa. Tästä syntyvää luokkaa kutsutaan termillä derived class, eli johdettu luokka, ja pohjalla olevaa luokkaa termillä base class eli perusluokka. Johdettu luokka on normaali luokka siinä missä perusluokkakin, merkittävä ero on johdannaisen pääsy kaikkiin perusluokan julkisiin ja suojattuihin (Protected) muuttujiin ja metodeihin (Microsoft 2022b). Suojatut

metodit ja muuttujat näkyvät vain luokalle itselleen sekä sen perillisille (Microsoft 2024b).

Johdettuja luokkia voidaan käsitellä kuin ne olisivat perusluokan jäseniä (Microsoft 2023b). Tähän liittyy ohjelmointikonsepti tyyppi (Type), joka kuvaa minkälainen kyseinen muuttuja tai asia on ja määrittelee, mitä sillä voi, tai pitäisi voida, tehdä koodissa. Tyyppejä on esimerkiksi int, eli kokonaisluvut, ja string, eli merkkijono, sekä jokainen luokka on tyyppi. Kun odotetaan tiettyä luokkaa oleva muuttuja, odotetaan siis tiettyä tyyppiä. Tyyppejä ei lähtökohtaisesti voi C#-kielessä sekoittaa keskenään. (Microsoft 2024c.) Esimerkiksi kohtaan, johon kaivataan merkkijono string ei voi syöttää kokonaislukua int, jollei sitä erikseen muuteta muotoon string. Polymorfismi, eli mahdollisuus käsitellä johdannaisluokkia kuin ne olisivat perusluokkaa, ja johdannaisluokkien kyky ohittaa perusluokan metodeja auttaa kiertämään edellä kuvattua tyyppiongelmaa. Kriisiosaajassa kaikki toiminnot periytyvät BaseInteraction-luokasta. Nukkumistoimintoa voidaan siis käsitellä kuin se olisi tyyppiä Sleep_Interaction, (”Nuku_Toiminto”) eli johdannaisluokka, tai BaseInteraction, eli perusluokka.

Johdetut luokat voivat ”ohittaa” (Override) metodeja perusluokassa. Tiivistetysti sekä johdetussa luokassa että perusluokassa on oltava samannimiset metodit. Ohitettava metodi on perusluokassa merkittävä avainsanalla abstract (”abstrakti”) tai virtual (”virtuaalinen”) ja johdetussa luokassa avainsanalla override. Nyt johdetulla luokalla on metodi samalla nimellä, kuin perusluokassa ja siihen voi lisätä toiminnallisuutta. Asiat, mitä kaikki johdetut luokat tekevät, voidaan laittaa perusluokkaan ja johdetut luokat voivat kutsua ja suorittaa logiikan. (Microsoft 2023b.)

Kuvassa neljä näkyy vasemmalla AtDestination-metodin toteutus perusluokassa. Metodin nimen edessä on avainsanojen joukossa virtual, eli metodin voi ohittaa. Vasemmalla näkyy johdetun luokan AtDestination-toteutus. Avainsanojen joukossa on override, eli metodi ohittaa perusluokan toteutuksen. Metodin ensimmäisellä rivillä kutsutaan base.AtDestination, eli perusluokan metoditoteutus kutsutaan ennen johdetun luokan omaa logiikkaa. Näin perusluokkaan saadaan laitettua logiikka, jota kaikki johdannaiset tarvitsevat.



Kuva 4. Kuva, jossa näkyy sama metodi BaseInteraction ja Sleep_Interaction luokissa.

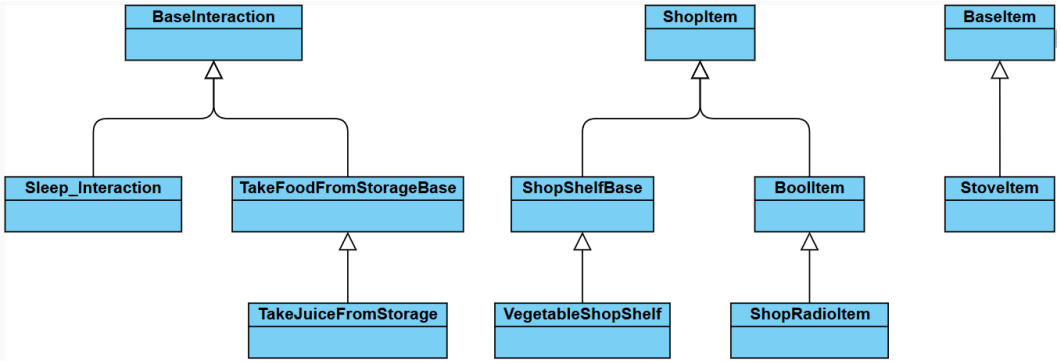
Kun pelaaja Kriisiosaajassa valitsee toiminnon, klikkaa hän pelimaailmassa olevaa huonekalua. Taustalla peli hakee kaikki esineessä olevat toiminnot ja kerää ne listaan, josta pelaaja tekee lopullisen valinnan. Sillä toiminnot periytyvät samasta perusluokasta, voidaan niitä käsitellä yhdessä listassa, joka on tyyppiä BaseInteraction. Ilman perintää toimintoja ei voitaisi laittaa samaan listaan, sillä ne ovat kaikki eri tyyppiä. Kun valittua toimintoa klikataan listassa, suoritetaan tämän toiminnon OnInteractionBegin-metodi (”ToiminnonAlussa”). Sillä jokainen esineessä oleva toiminto on johdannainen ja ohittaa kyseisen metodin, suoriutuu tämä ohitettu johdannaismetodi ja toiminto alkaa.

4.2.2 Perintä Kriisiosaajassa

Kriisiosaajassa perintää ja polymorfismia hyödynnetään melko paljon:

- Jokainen pelin esine, jossa on valittavia toimintoja, periytyy BaseItem-luokasta (”PerusEsine”). Kuvassa 5 esineiden perintä näkyy oikealla.
- Jokainen pelin toiminto periytyy BaseInteraction-luokasta (”PerusToiminto”). Kuvassa 5 toimintojen perintä näkyy vasemmalla.
- Muutamat erityisen samanlaiset toiminnot periytyvät yhteisestä perusluokasta. Esimerkiksi kaikki toiminnot, joissa ruokaesine otetaan esille kaapista, periytyvät TakeFoodFromStorage-luokasta (”OtaRuokaVarastosta”), joka puolestaan periytyy BaseItem-luokasta. Kuvassa 5 toimintojen perintä näkyy vasemmalla.
- Kauppanäkymässä kaikki ostettavat esineet perivät ShopItem-luokan. Luokassa on metodi esineen ostamiseen, jonka johdannaisluokat ohittavat. Perusluokan implementaatiotakin kutsutaan, sillä se vastaa rahojen vähentämisestä. Kuvassa 5 kauppaesineiden perintä näkyy keskellä.

Kuvassa 5 on UML-luokkakaavio. UML tulee englannin kielen sanoista Universal Modeling Language eli universaali mallinnuskieli. Kyseisen tyyppisiä kaavioita käytetään ohjelmointimaailmassa usein luokkien ja niiden välisten yhteyksien suunnitteluun ja kuvaamiseen. Kaaviossa nuoli lähtee johdannaisluokasta ja osoittaa kohti perusluokkaa. (Visual Paradigm s.a.) Kuvassa näkyy yllä selitetty perintä. Kuvassa nuoli osoittaa kohti perusluokkaa, joten tässä kuvassa perusluokka on aina johdannaista ylempänä.



Kuva 5. Kaavio perinnästä Kriisiosaajassa. Vasemmalla toimintojen perintä, keskellä kauppaesineet ja oikealla esineet. Nuoli osoittaa perusluokkaan.

Lisäksi projektissa on useita luokkia, jotka perivät ScriptableObject-luokan. Näistä esimerkkinä luvussa 4.1 käsitellyt event channel -esineet. Lisäksi valtaosa pelin luokista perii Unityn MonoBehaviour-luokan, joka tarjoaa Unityn toimintaan liittyviä metodeja käytettäväksi.

4.2.3 BaseItem

BaseItem on kaikkien pelissä kotona olevien esineiden, joissa on toimintoja, perusluokka. BaseItem-luokka ei toteuta juuri mitään, se lähinnä säilöö muutaman tärkeän muuttujan, joita lähes kaikki esineet tarvitsevat. BaseItem-luokassa on muutama ohitettava metodi, kuten luokan alustamiseen liittyvä Awake-metodi (”Herää”), ja ajanpitämiseen liittyvä TimeBeat-metodi (”AikaTahti”).

BaseItem-luokassa tai sen käytössä ei ole havaittavissa vakavia virheitä. Tämä johtunee osittain yksinkertaisesti siitä, ettei perusluokka tee juuri mitään muutaman metodin ja muuttujan tarjoamisen ulkopuolella. Pieniä parannusehdotuksia on kuitenkin.

Joissain toiminnoissa suoritetaan logiikkaa, joka vaikuttaa esineeseen, esimerkkinä uuni ja ruoanlaittamistoiminto. Kun pelaaja laittaa uunin avulla ruokaa, aukeaa ja sulkeutuu uunin ovi sekä toiminnon alussa että lopussa. Nyt oven avaamislogiikka on toiminnossa itsessään. Pelissä on vain tämä yksi toiminto, joka avaa uunin oven, joten logiikan laittaminen toimintoon itseensä käy järkeen. Logiikka oven avaamiseen kannattaa kuitenkin siirtää esineeseen,

jotta jos useampi esineen toiminto tarvitsee ominaisuutta, on se siellä saatavilla.

Toinen yleisempi parannusehdotus on BaseItem-perusluokan muuttaminen abstraktiksi. Avainsana abstract lisääminen luokan yhteyteen kertoo, että luokka, tai metodi, johon sana liittyy, ei ole tarkoitettu käytettävän sellaisenaan ja että toinen luokka jatkaa tai ohittaa kyseisen asian. Abstraktia luokkaa ei voida instantioida eli luokan tyyppiä olevaa esinettä ei voi luoda sellaisenaan. (Microsoft 2024a.) Luokat, joiden ei ole tarkoitus olla sellaisenaan missään ja jotka toimivat perusluokkana muille, voidaan merkitä abstrakteiksi, jotta projektin muut ohjelmoijat tietävät, mikä on luokan tarkoitus, eivätkä yritä instantioida tai muulla tavalla käyttää abstraktiksi merkittyä luokkaa sellaisenaan.

4.2.4 Base Interaction

BaseInteraction-luokan ajatus on verrattavissa BaseItem-luokkaan. Kyseessä on perusluokka kaikille pelin toiminnoille. Toiminto on asia, jonka pelaaja voi valita ja jonka hahmo suorittaa, kun pelaaja klikkaa esinettä ja valitsee valikosta yhden esineen toiminnoista. Toisin kuin esineillä, toiminnot hyödyntävät perintää ja polymorphismia laajalti.

BaseInteraction-perusluokassa on muutama johdettujen luokkien käytössä oleva muuttuja sekä useampi virtual-avainsanalla merkattu metodi. Virtual-avainsana tarkoittaa, että metodin voi ohittaa johdetussa luokassa. Virutal-avainsanalla varustetussa perusluokan metodissa voi olla koodia ja johdannaisluokat voivat halutessaan kutsua sitä. (Microsoft 2024d.) Ohitetut metodit liittyvät joko toiminnon alustamiseen, jolloin ne suoritettaan, kun toiminto ladataan, tai toiminnon suorittamiseen ja lopettamiseen. Toiminto on perusluokassa jaettu kolmeen osaan:

- AtInteractionBeign (”ToiminnonAlussa”) on metodi, jota kutsutaan, kun toiminto valitaan suoritettavaksi. Perusluokka ei tee metodissa mitään, johdetut luokat ohjaavat pelaajan valitun toiminnon omistajaesineen toimintokohtaan (InteractionSpot).
- AtDestination (”Määränpäässä”) kutsutaan, kun pelaajahahmo saapuu valitun toiminnon omistajaesineen toimintokohtaan. Perusluokassa metodissa irrottaudutaan kanavasta, joka ilmoittaa, kun pelaaja pääsee toimintokohtaan. Näin kyseinen toiminto ei aktivoidu, kun pelaaja siirtyy tekemään seuraavaa toimintoa.

- EndInteraction (”PäätäToiminto”) kutsutaan kun toiminto päättyy. Perusluokan implementaatio kutsuu interactionManagerissa olevaa toiminnonlopettamismetodia.

Yllä kuvatut metodit saattaisivat yksinäänkin riittää toiminnon toteuttamiseen ja suorittamiseen, mutta projektissa lähes kaikissa johdetuissa luokissa on AtDestinationin jälkeen DuringInteraction-metodi (”ToiminnonAikana”), jossa varsinainen toiminto tapahtuu. Nukkumistoiminnossa energian saanti tapahtuu tässä metodissa. Metodi ei kuitenkaan ole osana perusluokkaa, vaan johdetut luokat lisäävät sen itse, kukin tahollaan. Tämä on seurausta työmallista, jossa uutta toimintoa varten vanhan runko kopioitin ja muokattiin sopivaksi. Ajatus taustalla oli ajan säästäminen, sillä vaihtoehtona oli saman rakenteen käsin uudelleen kirjoittaminen, sillä kopioitava osuus keskittyi siihen, kuinka johdannaisluokka ohittaa perusluokassa olevia metodeja. Ohitettavien metodien lisäksi kopioitava osuus sisälsi myös muuta, kuten DuringInteraction-metodin ja sen, että jokainen johdettuluokka asettaa pelaajahahmolle määränpään täysin samassa paikassa, sen sijaan että käsky tapahtuisi perusluokassa ”automaattisesti”. Rutiininomaisen kopioinnin seurauksena kopioiduista osista tuli oleellinen osa toiminnon luomiseen tarkoitettua ajattelumallia.

Koodin kopiointi on pahasta, tämä todettiin jo luvussa 4.1.3. Edellä kuvattu kopiointi on esimerkki tilanteesta, jossa ei ole tarkoitus korvata suoritettavaa koodia, vaan rakenne. Ongelmia ilmaantui, kun ensimmäinen kopiointi tehtiin huolimattomasti tai ilman suunnitelmaa, jolloin kopioon, ja kopion kopioon, päätyi suoritettavaa koodia. Lopputilanne on sama kuin jos olisi kopioitu metodi: muutokset on tehtävä jokaiseen kopioituun paikkaan.

BaseInteractionin ja sen johdannaisten suurin ongelma on epämääräisyys, jossa vain osa johdannaisissa olevista, toiminnon kannalta oleellisista, metodeista on ohitettu perusluokasta ja loput on lisätty suoraan johdannaisluokkaan osana kopioitua runkoa. Toimintojen rakenne pohjaa täysin dokumentaatiossa olevaan ja suullisesti ilmaistuun ajatukseen eri metodeista ja niiden rooleista toiminnon edetessä. Kokemattomampi ohjelmoija saattaa epähuomiossa toteuttaa toiminnon, joka ei vastaa rakennetta. Projektin kannalta on paljon kestävämpää, jos rakennetta ylläpitää ulkoistan käytänteiden ja dokumen-

taation sijaan perusluokka, jossa on metodit, joita tulee jatkaa ja käyttää. Kyseessä on Structure over convention -koodihaju, eli rakenne käytänteiden yli (Martin 2009, 301). Ratkaisu ongelmaan on siirtää DuringInteraction ja muut toiminon rakenteen kannalta oleelliset metodit osaksi BaseInteraction-luokkaa ja tehdä itse luokasta sekä toiminnon kannalta oleellisista metodeista abstrakteja. Näin olemassa olevien metodien käyttäminen on pakollisempaa, joten toimintojen rakenne pysyy yhtenäisenä, vaikka niitä tekisi eri ihmiset eri aikoina. Lisäksi johdannaisluokista tulee siirtää kaikille toiminnoille yhteinen toiminallisuus osaksi perusluokkaa. Esimerkki tällaisesta toiminnallisuudesta on esineen toimintokohtaan siirtyminen.

Ehdotuksen suurin haitta on sen aiheuttama työmäärä. Jokainen projektissa jo oleva toiminto kaipaa muokkauksia, sillä olemassa oleva DuringInteraction-metodi on muokattava olemaan ohitus ja siirrettävä perusluokkaan. Samoin perusluokkaan siirretty toiminallisuus on poistettava johdannaisluokista ongelmien välttämiseksi. Työ ei ole varsinaisen haastava tai aiheuta suurta virheriskiä, mutta siihen kuluu aikaa.

4.2.5 Samankaltaiset toiminnot

Kriisiosaajassa on muutama toimintoryhmä, jotka ovat ryhmänsisäisesti keskenään niin samanlaisia, että niitä varten on oma perusluokkansa. Esimerkkinä ruoan esille ottaminen, jolle on työtasossa kuusi ja jääkaapissa kaksi eri toimintoa, sillä kaapista voi ottaa useita ruokatyyppejä. Näillä toiminnoilla on yhteinen perusluokka TakeFoodFromStorage (”OtaRuokaVarastosta”), joka puolestaan perii BaseInteraction-luokan. Yksittäiselle toimintoluokalle, kuten TakeJuiceFromStorage (”OtaMehuVarastosta”), jää tehtäväksi toivotun ruokaesineen luominen ja oikeaan paikkaan asettaminen sekä joitakin pohjustustehtäviä, kun peli alkaa. Muu toimintoon liittyvä, kuten tarkistus, mahtuuko tason päälle lisää ruokaa, sijaitsee perusluokassa. Näin kaikki ruokaa esille ottavat toiminnot voivat hyödyntää samaa metodia ja riittää, että ruokakohtaiset muutokset tehdään yhteen paikkaan.

Toimintoryhmissä ja niiden toteutuksessa ei Kriisiosaajassa vaikuta olevan mitään merkittävästi vialla, joten ainut parannusehdotus tähän kohtaan on edellisistä kappaleista tuttu ehdotus tehdä perusluokasta, jonka tyyppisiä asioita

pelimaailmassa ei ole tarkoitus olla, abstrakteja abstract-avainsanalla. Näin vältetään mahdollinen riski, että joku koittaa tehdä TakeFoodFromStorage-tyyppisen asian peliin tai muokata luokasta kokonaista.

4.2.6 Kauppaesineet

Pelissä on kauppa, josta pelaaja ostaa hahmolleen ruokaa, juotavaa sekä kriisitilanteessa hyödyllisiä esineitä, kuten paristokäyttöinen radio ja retkikeitin. Kaupassa esineet ostetaan klikkaamalla haluttua esinettä ja valitsemalla osta. Jos kyseessä on ruoka tai juoma, voi niitä ostaa useamman, niin monta kuin varastoa riittää.

Teknisesti ostettavat esineet jaetaan kahteen:

- Hyllyt, joista saa ruokaa ja juomaa. Näitä voi ostaa niin monta kertaa, kun hyllyssä riittää varastoa
- Bool-esineet, joita voi ostaa vain kerran per esine. Nimi juontuu siitä, että esineen voi ostaa vain kerran ja siitä pidetään kirjaa bool-tyyppisellä muuttujalla. Bool-muuttujan arvo voi olla joko tosi tai epätosi (true tai false) (Microsoft 2022a).

Kaikki kaupassa myytävät esineet periytyvät ShopItem-luokasta (”KauppaEsine”). Luokassa on kaikille kaupassa myytäville esineille yhteiset ominaisuudet, kuten esineen nimi, hinta sekä metodi, jossa pelaajan rahoista vähennetään esineen hinta. Lähes kaikki muu tapahtuu johdetuissa luokissa. ShopItem-luokasta periytyy kaksi luokkaa yllä kuvatun jaon mukaisesti: ShopShelfBase (”KauppaHyllyPerus”) on kaikkien hyllyjen perusluokka ja BoolItem (”BoolEsine”) on kaikkien bool-esineiden perusluokka.

ShopShelfBase-luokasta johdetaan jokainen yksittäinen hylly, kuten VegetableShopShelf (”VihannesKauppaHylly”). Perusluokka vastaa hyllyn varaston tilan seurannasta ja päivittämisestä. Yksittäinen hylly muuttaa pelaajan tavaraluettelossa olevaa lukua, VegetableShopShelf lisää pelaajan vihannesten määrää yhdellä oston yhteydessä.

BoolItem-perusluokka säilöö esineen ostamiseen liittyvä isBought-boolean arvon (”onOstettu”). Yksittäinen esine, kuten ShopRadioItem (”KauppaRadioEsine”), muokkaa pelaajan tavaraluetteloa ostamisen yhteydessä, jotta peli tietää pelaajan ostaneen kyseisen esineen.

Kauppaesineiden perinnässä ei ole havaittavia ongelmia perusluokkien abstract -avainsanan puuttumista lukuun ottamatta. Luokat ShopItem sekä BoolItem ja ShopShelf ovat perusluokkia, jonka tyyppisiä esineitä pelimaailmassa ei ole tarkoitus olla. Niistä kannattaa siis tehdä luokkina abstrakteja, jottei niitä voi vahingossakaan luoda pelimaailmaan. Metodeista ei tässä tilanteessa voi tehdä abstrakteja, sillä perusluokan versio BuyItem-metodista tekee ostamisen kannalta oleellisia asioita, eikä abstrakti metodi voi sisältää logiikkaa (Microsoft 2024a).

5 PROJEKTIN TILA YLEISESTI

Luvussa käydään projektin koodipohjaa yleisemmin läpi tutustuen koodipohjan suurimpiin koodihajuihin ja niiden syihin. Löydettyihin ongelmiin tarjotaan ratkaisuja.

5.1 Metodit

Metodit sisältävät ohjelman logiikan, joten niiden laatuun on hyvä kiinnittää huomiota. Tässä luvussa käydään läpi metodeihin liittyvät suurimmat koodihajut ja niihin tarjotaan yleistettäviä ratkaisuja.

5.1.1 Suuret metodit

Projektin koodihajuja ja niihin ratkaisua pohtiessa käy ilmi, että iso osa projektin hajuista johtuu liian montaa asiaa tekevistä metodeista. Metodin tulisi tehdä vain yksi asia (Martin 2009, 35–36.) Esimerkkinä luokka, joka vastaa pelaajan tarpeiden tilasta. Luokassa on yksi suuri metodi, joka vastaa tarpeiden tilan passiivisesta laskusta ja tarpeiden tilan seurauksista. Jo otsikkotasolla näkee, että metodi tekee ainakin kaksi asiaa: Se laskee tarpeita ja aiheuttaa tarpeiden tilasta seuraamuksia. Esimerkkinä seuraamuksesta on matalasta nälkätarpeesta johtuva terveyden aleneminen. Todellisuudessa metodi tekee siis paljon enemmän kuin vain yksi tai kaksi asiaa, sillä jokainen laskeva tarve ja tarpeen tilan seuraamus on asia, jonka metodi tekee.

Sekä Martin (2009, 31), Fowler ym. (2002, 64) ja Visser (2016, 11) ovat yhtä mieltä siitä, että metodien tulee olla lyhyitä, jotta niiden ymmärtäminen on helpompaa ja sitä myöten muokkausten tekeminen nopeampaa. Tehokas tapa pienentää metodia on pilkkoa se pienempiin osiin, Fowler ym. (2002) käyttävät tästä termiä extract method eli poimi metodi. (Tai poimintatapa, sillä method tarkoittaa englanniksi myös työtapaa ja kyseessä voi olla sanaleikki.) Näin jokainen tarpeen alenemiskohta ja tarpeen tilan seuraamus voi olla oma metodinsa, jota päämetodi kutsuu. Tuloksena päämetodi on lyhyempi ja luettavampi kun se koostuu kuvaavasti nimetyistä metodeista. Lisäksi jokainen tarve on helpompi löytää ja muokata erikseen.

Projektissa on paljon paikkoja, jossa metodin poimintaa voi hyödyntää koodin luettavuuden parantamiseksi. Edellä annettu esimerkki on varsin yksinkertainen tilanne, sillä metodit voidaan helposti siirtää erilleen päämetodista, koska ne toimivat itsenäisinä kokonaisuuksina. Ongelmia voi muodostua tilanteissa, joissa koodipätkä, joka haluttaisiin poimia irralleen, käyttää metodin sisällä olevia muuttujia. Kyseiset muuttujat voisi teoriassa siirtää metodista luokkaan, jolloin kaikki luokassa olevat metodit voivat sitä hyödyntää. Käytännössä muuttujien scopen eli laajuuden muuttaminen saattaa vaatia isojakin muutoksia luokan sisällä, jos esimerkiksi vastaavan nimistä muuttujaa käytetään jo samassa luokassa toisen metodin sisällä. Laajuus tarkoittaa esimerkiksi metodin näkyvyyttä ja käytettävyyttä ohjelman sisällä (Codecademy 2023b). On paljon parempi hoitaa metodin poiminta niin, ettei muuttujan laajuus muutu. Metodin sisällä olevat muuttujat voidaan antaa uudelle poimitulle metodille parametreinä. Parametrin kautta metodi saa tarvitsemansa arvot ilman, että niiden laajuutta joudutaan muuttamaan. (Fowler ym. 2002)

5.1.2 Kuollut koodi

Dead code eli kuollut koodi on koodia, jota ei koskaan suoriteta. Koodi voi olla mahdottomien if-lauseiden takana tai metodi, jota ei koskaan kutsuta. (Martin 2009, 302.) Myös koodi, jota kutsutaan mutta jonka lopputulosta ei käytetä, voidaan pitää kuolleena koodina (Visser 2016, 131). Kuolleen koodin suurin

ongelma on se, ettei sitä ylläpidetä, joten sen design ja logiikan kulku on vanhaa eikä noudata ohjelman uusia normeja. Kuollut koodi kannattaa poistaa, sillä sitä ei selvästikään tarvita. (Martin 2009, 302.)

Kriisiosaaja-projektissa on jonkin verran kuollutta koodia, joka olisi hyvä siivota pois. Esimerkiksi ConsumableBase-luokassa on paljon yksityisiä muuttujia, joita ei käytetä luokassa mihinkään. Yksityinen (private) on avainsana, joka kertoo, että kyseinen muuttuja tai metodi on vain sen luokan käytössä, jonka sisällä se on (Microsoft 2024b). Nyt ne ovat viemässä ruutu- ja aivokapasiteettia koodia tulkitsevalta. Projektissa on myös joitakin oletettavasti testaustarkoituksessa luotuja luokkia, joita ei käytetä ja ne tulisi siivota pois projektin tiedostorakennetta täyttämästä.

5.1.3 Flag argument

Flag argument eli lippuargumentti tai lippuparametri tarkoittaa, että metodi ottaa parametrikseen boolean-muuttujan. Kyseessä on selvä merkki siitä, että kyseinen muuttuja tekee useamman kuin yhden asian, sillä metodi tekee jotain muuttujan ollessa tosi, ja jotain muuta sen ollessa epätosi. (Martin 2009, 41.)

Projektissa on karkeasti kahdenlaisia boolean-parametrillä varustettuja metodeja. Pelaajaluokalla on booleanit HasWater ja HasElectrcity (”OnVettä” ja ”OnSähköä”) ja näiden muokkaamiseen liittyvät metodit ottavat booleanin parametriksi. Tällöin metodi ei tee kahta asiaa parametrillä, vaan tallentaa sen muuttujaan. Toinen ryhmä sen sijaan tekee kaksi asiaa booleanin arvosta riippuen.

Esimerkiksi pisteytysluokassa on metodi OnWindowSeal(bool isSealed) (”IkkunaaTiivistäessä(bool onTiivistetty)”). Suluissa oleva isSealed on boolean parametri, joka annetaan metodille. Metodi näkyy kuvassa 6. Muuttujan arvo tarkistetaan ja jos se on tosi, suoritetaan lisätarkistuksia ja annetaan mahdollisesti pisteitä. Jos muuttuja on epätosi, muutetaan koodiluokassa olevan muuttujan arvo.

```
2 references
private void OnWindowSeal(bool isSealed)
{
    if (isSealed)
    {
        //if window seal was done with lethal air, give score
        if (interactionManager.GetWorldAirQ() == GlobalValues.Quality.Dangerous)
        {
            //accumulativeScore += properPreparationScore;
            lethalAirInRoom = false;
            lethalAirGraceEnded = false;

            scoringStrings.Add(properWindowPrepDescription);
            scoringScores.Add(properWindowPreparationScore);
        }
        //else prep was done with non lethal air, penalty
        else
        {
            scoringStrings.Add(inproperWindowPrepDescription);
            scoringScores.Add(inproperWindowPreparationScore);
        }
    }
    else
    {
        if (interactionManager.GetWorldAirQ() == GlobalValues.Quality.Dangerous)
        {
            lethalAirInRoom = true;
        }
        else
        {
        }
    }
}
```

Kuva 6. OnWindowSeal-metodi

Yksinkertainen ratkaisu on jakaa metodi kahteen esimerkiksi OnWindowSeal ja OnWindowUnseal. Ehdotetut nimet eivät ole parhaat mahdolliset, sillä ne muistuttavat toisiaan liikaa, varsinkin tekstiä nopeasti silmäillessä. Muutosta suunnitellessa on otettava huomioon, että metodia kutsutaan kanavan kautta, joten metodin jakaminen kahteen vaatisi kaksi kanavaa. Lopputuloksena on luettavampi koodipätkä. Kuvan 6 metodi on hyvä esimerkki projektin koodihajujen määrästä ja laadusta. Kyseessä on pitkä metodi, jossa on paljon haarautumiskohtia, joten se tulisi joka tapauksessa pilkkoa pienempiin osiin luettavuuden ja ylläpidettävyyden parantamiseksi.

5.2 Nimeäminen

Nimeäminen on ymmärrettävän koodin ytimessä, sillä iso osa koodin luettavuudesta nojaa oikein valittuihin ja tarkoituksenmukaisiin nimiin. Nimeäminen tarkoittaa tässä kaikkea ohjelmointiin liittyvää nimeämistä, kuten muuttujien, metodien ja luokkien nimiä. Martinin (2009, 309) mukaan nimet ovat 90 % koodin luettavuutta parantavista tekijöistä ja neuvoo käyttämään aikaan nimen

valintaan, sekä säännöllisesti pohtimaan, onko aiemmin valittu nimi yhä ajankohtainen. Oikein valittu ja tarkoituksen mukainen nimi kuvaa, mikä nimettävä asia on tai mitä se tekee (Martin 2009, 312).

Suuri projektin nimiin liittyvä koodihaju on metodien nimissä. Projektissa metodit on usein nimetty sen mukaan, milloin niitä kutsutaan. Tällaiset nimet ovat kuvaavia, kun luetaan koodia, josta metodia kutsutaan. Kun pisteytys luokkaa silmäilee ja vastaan tulee metodi nimeltä OnUnpoweredBathroomUse (”VirratontaVessaaKäytettäessä”), on helppo tehdä päätelmä, että metodia kutsutaan, kun käytetään vessaa, jossa ei ole virtaa. Metodi näkyy kuvassa 7. Nimi ei kuitenkaan avaa juuri yhtään, mitä käytännössä tapahtuu. Kehittäjän on siis käytettävä aikaa koodin tulkintaan tai koodia ympäröivien kommenttien lukemiseen sen sijaan, että metodi osaisi itse kertoa itsestään eli olisi nimetty kuvaavasti.

```
2 references
private void OnUnpoweredBathroomUse(bool unpoweredCapasityFull)
{
    scoringScores.Add(unpoweredBathroomUseScorePenalty);
    if (firstUnpoweredBathroomUse)
    {
        scoringStrings.Add(firstUnpoweredBathroomUseDescription);
        firstUnpoweredBathroomUse = false;
    }
    else
        scoringStrings.Add(laterUnpoweredBathroomUseDescription);

    if (unpoweredCapasityFull)
    {
        if (firstUnpoweredBathroomFull)
        {
            scoringStrings.Add(firstFullUnpoweredBathroomUseDescription);
            firstUnpoweredBathroomFull = false;
        }
        else
        {
            scoringStrings.Add(laterFullUnpoweredBathroomUseDescription);
        }
        scoringScores.Add(fullUnpoweredBahtroomScorePenalty);
    }
}
```

Kuva 7. OnUnpoweredBathroomUse-metodi

Yksi ratkaisu ongelmaan on suurista metodeista tuttu metodin poiminta. Pisteytysluokkaan voisi lisätä metodin, AddScore (”LisääPisteitä”), jonka ainut

tehtävä on lisätä listaan luku. Metodi ei tee paljoa, mutta se lisää muiden metodien luettavuutta, kun ScoringStrings-listaan (”PisteytysStrings”) lisäämisen sijaan näkyy kuvaavasti nimetty metodikutsu AddScore. Lisäksi kuvan 7 metodin koko ensimmäinen if/else -osuus, eli käytännössä koko metodin ensimmäinen puolisko, voidaan siirtää helposti omaan metodiinsa, sillä se ei tarvitse nykyisen metodin sisältä mitään, eikä se tuota arvoja, joita nykyinen metodi tarvitsee.

Aiemmin esitellyssä BaseItem-luokssa on paljon niin sanotusti kutsutilanteen mukaan nimettyjä metodeja. Toiminnon runko ja sen metodien nimet, kuten OnInteractionBegin ja AtDestination, on nimetty sen mukaan, milloin niitä kutsutaan, sillä kyseessä on runko, jota kehittäjän tulisi seurata. Nimet eivät kuvaa yhtään, mitä metodin sisällä tehdään, vaan kehittäjän on tulkittava koodia rivi riviltä. Lähes jokaisesta toiminnosta voi poimia asioita rungon metodeista omiin metodeihinsa, nimetä ne kuvaavasti ja kutsua näitä uusia metodeja rungossa. Näin rungon kutsuajankohtaa kuvaava nimi käy järkeen ja metodi on lyhyt ja selkeälukuinen, sillä toteuttava logiikka on siirretty kuvaavasti nimettyjen metodien taakse.

5.3 Kommentit

Kommentit ovat ymmärrettävän koodin kannalta sekä hyödyksi että haitaksi. Hyvä ja hyödyllinen kommentti avaa koodin lukijalle hämmentävää metodia, avaa tarkoitusta koodin taustalla tai varoittaa vaaroista. Haitallinen kommentti sisältää vanhentunutta, turhaa tai jopa väärää tietoa. Ideaalitilanteessa koodi on rakennettu ja nimetty niin, ettei kommentteja tarvita, sillä koodi puhuu puolestaan. (Martin 2009, 53–54.)

Kriisiosaajassa kommentointi ei ole erityisen hyvää. Projektissa on paljon turhia kommentteja, poiskommentoitua koodia ja jonkin verran harhaan johtavia tai hämmentäviä kommentteja. Valtaosa on turhia kommentteja, jotka eivät kerro mitään uutta vaan toistavat saman kuin kommentin läheisyydessä oleva koodipätkä. Toisinaan koodipätkä on hieman monimutkaisempi, jolloin kommentti on oikeutetumpi.

Esimerkiksi ItemManager-luokassa (”EsineManageri”) on metodi OnWater-Loss (”VedenMenetyksessä”), jonka yläpuolella oleva kommentti kertoo, että metodia kutsutaan, kun vesi menetetään. Metodi näkyy kuvassa 8. Veden menetys tarkoittaa tässä kontekstissa pelaajan käyttövedentulon loppumista, mutta ilman kontekstiakin metodin nimi ja kommentti kertovat saman asian, eli kommenttia voidaan pitää turhana. Tämän tyyliset kommentit ovat projektin kommenteista yleisimpiä.

```
//Called when the water is lost
2 references
private void OnWaterLoss()
{
    foreach (IRequiresWater requiresWater in requiresWaterItems)
    {
        requiresWater.OnWaterOuttageBegin();
    }
}
```

Kuva 8. OnWaterLoss-metodi, jonka yläpuolella on turha kommentti.

Projektissa on epämäärisiä kommentteja, kuten kommentti, jossa kerrotaan, että sen alapuolella olevaa metodia ”voisi ja pitäisi ehkä hienosäätää”. Kommentti ja metodi näkyvät kuvassa 9. Kommentti on sellaisenaan ontuva ja epämääräinen, siinä ei avata yhtään, miten metodia tulisi muokata eikä metodissa näy suoraan mitään muokattavaa. On siis myös mahdollista, että kyseessä on vanha kommentti, joka kirjoitettiin, ennen kuin metodia muokattiin, ja kun metodi päivitettiin nykyiseen muotoonsa, ei kommenttia jostain syystä poistettu.

```
//Used to check if at destination, could and should probably be finetuned
1 reference
private IEnumerator IsAtDestination()
{
    yield return new WaitUntil(() => Vector3.Distance(transform.position, currentDestination)
    //When at destination

    interactionManager.InteractionCancelButtonGO.SetActive(false);
    playerAtDestinationEC.RaiseEvent();
    yield return null;

}
```

Kuva 9. IsAtDestination-metodi, jonka yläpuolella on sekä turha että hämmentävä kommentti.

Projektissa on myös paljon pois kommentoitua koodia. Osa koodista on kommentoitu pois, sillä se ei toiminut, eikä tiimillä ollut aikaa poistaa kutsuja kommentoituun koodiin. Pelissä pelaajalla oli tarkoitus olla käytössä puhelin, jossa

on useita toimintoja, kuten klassinen matopeli. Puhelimen kaikkia ominaisuuksia ei saatu toimimaan ajoissa, joten esimerkiksi matopeliin liittyvä koodi kommentoitiin pois, sillä sen kutsuminen hajotti Kriisiosaajan. Koodia ei haluttu poistaa, sillä tarkoitus on jatkaa projektia ja saada esimerkiksi matopeli toimimaan oikein.

Osa kommentoidusta koodista on poiskommentoitu, sillä sitä on käytetty osana testaamista, mutta lopulliseen toteutukseen on haluttu toisenlainen implementaatio. Esimerkiksi pelaajan tarpeiden laskusta vastaavassa luokassa pelaajan vessatarvetta laskevassa kohdassa on käytössä koodirivi, jossa tarvetta lasketaan yhden verran. Rivin alapuolella on kommentoituna ”DEBUG”, joka tarkoittaa virheenkorjaamista, ja sen alapuolella poiskommentoitu koodirivi, jossa vessatarvetta lasketaan viidellä. Vessatoiminallisuutta kehittäessä tarpeen oli hyvä laskea nopeammin, joten tehtiin poiskommentoitu rivi. Rivi jätettiin koodiin, sillä jos on tarve testata vessatarvetta tai siihen liittyviä asioita, voidaan tarvetta laskea nopeammin helposti poistamalla rivin kommentointi ja kommentoimalla nyt käytössä oleva rivi pois.

```
interactionManager.AdjustPlayerBathroom(-1);
//DEBUG
//interactionManager.AdjustPlayerBathroom(-5);
```

Kuva 10. Pelaajan vessatarpeen laskusta vastaava koodikohta, jossa on poiskommentoitua koodia.

Edellä kuvattu poiskommentoidut koodirivit tulisi teoriassa poistaa, sillä kommentoitu rivi on yleensä haitaksi. Käytännössä nykyiset ratkaisut toimivat väliaikaisesti, mutta niistä tulisi hankkiutua eroon mahdollisimman pian. Puhelimen poiskommentoitu toiminnallisuus tulisi korjata, jotta koodia voi käyttää sellaisenaan, ja tarpeen testaamiseen liittyvä koodirivi tulisi poistaa ja testaamista varten tehdä oma toteutuksensa. Luokkaan voi esimerkiksi lisätä debug-booleanin, jonka kehittäjä voi laittaa päälle, kun tarvetta testataan. Näin tarve voi laskea normaalisti yhden verran ja tarvittaessa viidellä.

6 TULOKSET

6.1 Mitä tarkoittaa ylläpidettävyys ja mitä siihen kuuluu?

Luvussa 3.3 ylläpidettävyys määritellään olevan ohjelmiston laadun mittari, jolla seurataan, kuinka helppoa ohjelmiston koodia on tulkita ja tarvittaessa muokata. Ylläpidettävyyteen liittyy useita osa-alueita, joita esitellään samassa luvussa kymmenenä ohjenuorana ylläpidettävän koodin kirjoittamiseen.

6.2 Miten ylläpidettävyyttä voidaan parantaa?

Ylläpidettävyyttä voidaan parantaa noudattamalla luvussa 3.3 esiteltyä kymmentä ohjenuoraa. Lisäksi koodihajuja tulisi välttää, sillä ne johtuvat usein ylläpidettävyysongelmista. Koodihajuja käsitellään luvussa 3.5 osana hyviä käytänteitä.

6.3 Mitkä osat pelin koodipohjasta kaipaavat parantamista ja miksi?

Pelin koodipohjassa lähes jokainen osa-alue kaipaa parantamista, sillä ne eivät ole nykyisellään ylläpidettäviä siten, miten se teoreettisessa viitekehyksessä määritellään. Projektin koodissa sekä nimeäminen että kommentointi voidaan toteuttaa paremmin, jotta projektista tulee ylläpidettävämpi, näitä käsitellään luvuissa 5.2 ja 5.3. Koodipohjan eniten parantamista kaipaava osa-alue on metodit, sillä ne ovat usein liian pitkiä tai tekevät useampaa asiaa. Metodien parantamisesta kerrotaan luvuissa 4.2 ja 5.1. Myös luvussa 4.1 käsitelty ongelma ratkaistaan parantamalla ongelman ytimenä olevaa metodia. Ylläpidettävyys on projektille erityisen tärkeää, sillä sitä on tarkoitus jatkokehittää toisen tiimin toimesta.

7 JOHTOPÄÄTÖKSET

Työn tavoitteena oli tutkia ylläpidettävyyttä ja selvittää, kuinka Kriisiosaajan ylläpidettävyyttä voidaan teorian avulla parantaa, jotta projektista tulisi ylläpidettävämpi ja siten jatkokehitettävämpi. Vastauksena ongelmaan syntyi luvut 4 ja 5, joissa aiemmin tutkittua teoriaa hyödynnetään projektin koodipohjasta löytyvien ongelmien ratkaisuun. Luvuissa esitellään projektin yleisimmät jatkokehitettävyysongelmat ja tarjotaan niihin yleistettäviä ratkaisuja. Tämän työn ulkopuolelle jääneiden ongelmien korjaaminen helpottuu työn avulla, sillä teoria

työssä esiteltyjen ratkaisuiden taustalla selitetään, jotta sitä voidaan soveltaa työssä yleisesti.

8 POHDINNAT

Pidän työtä onnistuneena, sillä tutkimusongelmaan ja -kysymyksiin saatiin vastaukset ja projektia varten paljon korjaus- ja parannusehdotuksia. Työ muuttui sen edetessä melkoisesti, sillä aluksi tarkoitus oli suunnitella ja toteuttaa projektin arkkitehtuuri, mutta projektin aikataulukiireiden vuoksi teorian tutkimiseen ja käyttöönottoon ei kehityksen aikana ollut aikaa. Tämä näkyy projektin nykytilassa sen ongelmina, joihin tässä työssä tarjotaan ratkaisuja. Tarjotut ratkaisut pohjaavat teoriaan ja tarkoitus teorian taustalla selitetään. Lisäksi opin itse työn aikana valtavasti ohjelmoinnista, hyvästä koodista ja ylläpidettävyydestä. Pidän sitä yhtenä työn onnistumisen merkkinä.

Työn teoreettinen viitekehys on kattava ja pohjaa ohjelmointialan kirjallisuuteen. Käytetyt kirjat ovat laadukkaita ja usein muussa alan kirjallisuudessa viitattua. Paljon nykyisissä lähteissä viitattua kirjallisuutta jäi myös lukematta. Osa jäi lukematta aikataulukiireiden takia ja iso osa oli työn aiheen ulkopuolella.

Luvussa 4 oli tarkoitus käsitellä nykyisten asioiden lisäksi projektissa olevia jatkokehitystä varten luotuja työkaluja. Käytännössä jatkokehitettävyyttä varten luotujen työkalujen arviointi on mahdotonta ilman vähintäänkin auttavaa tietämystä esimerkiksi käytettävyydestä. Työn koko olisi helposti räjähtänyt käsiin, mikäli teoreettiseen viitekehykseen olisi tuotu käytettävyys myös mukaan. Osuuden pois jättäminen auttoi kohdentamaan työn otsikkoa ylläpidettävyyteen aiemman jatkokehitettävyyden sijaan, jonka osa-alue ylläpidettävyys on.

Työn luotettavuuden ytimessä on dokumentaatio ja tarjottujen ratkaisuiden pohjaaminen teoriaan. Projektin esiteltyihin ongelmiin tarjottiin yleensä vain yksi ratkaisu, sillä teoria, johon ratkaisu pohjaa, oli hyvin usein yksioikoinen, mutta ratkaisun tarkoitus pyrittiin avaamaan, jotta esiteltyjen ongelmien ratkaisija voi halutessaan tai tarvittaessa soveltaa tai etsiä vaihtoehtoisia ratkaisuja niin, että ajatus alkuperäisen ratkaisuehdotuksen taustalla on tiedossa.

Työn luotettavuus kärsii tarjottujen ehdotusten toteutuksen ja testauksen puutteesta. Vaiheet jätettiin työstä pois aikataulun ja työn koon takia. Esiteltyjen ratkaisujen toteuttaminen, testaaminen ja tarvittaessa muokkaaminen veisi kuukausia. Toinen vaihtoehto olisi ollut keskittyä yhteen tai kahteen osa-alueeseen ja suunnitella, toteuttaa ja testata ratkaisuehdotus siihen. Valitsin nykyisen lähestymistavan, sillä koin siitä olevan eniten hyötyä sekä projektille että itselleni ja työlle. Testauksen puutteesta syntyvää luotettavuuspulaa pyrittiin täyttämään tehtyjen päätösten selittämisellä ja tarjotun ratkaisun tarkoituksen avaamisella. Näin, mikäli tarjottu ratkaisu koetaan olevan väärä tai muute projektiin epäsopiva, voidaan tarkoitusta ja teoriaa ratkaisun taustalla hyödyntää toista ratkaisua pohtiessa.

Teoreettisessa viitekehyksessä tarjottu teoria toimi hyvin käytäntöön laitettaessa. Työn teorian pohjana olevat materiaalit on tarkoitettu ylläpidettävyyden parantamiseen, joten luonnollisesti ne toimivat hyvin ylläpidettävyyttä parannettaessa. En koe, että työn aikana oli missään vaiheessa epäilystä siitä, saako projektia parannettua teoriaa hyödyntäen, kunhan aiheeseen liittyvän teorian löytää. Sen löytäminen oli suhteellisen mutkaton prosessi, sillä ylläpidettävyyttä on pohdittu ohjelmointimaailmassa jo ohjelmoinnin alkumetreiltä.

Tämän työn jatkokehittäminen on verrattain helppoa, sillä työssä esitellyt ratkaisut ovat toteuttamatta. Ehdotusten toteuttaminen ja testaaminen sekä mahdollinen hiominen on myös Kriisiosaaja-projektin eduksi, sillä sitä kautta projektista saadaan kiistatta ylläpidettävämpi. Toinen mahdollinen jatkokehitysväylä on tämän työn ulkopuolelle jääneiden ongelmien esiin tuonti ja korjaaminen tai korjausehdotusten tarjoaminen. Tällaisia osa-alueita voivat olla esimerkiksi aiemmin mainittujen jatkokehitystyökalujen testaus, arviointi ja parantaminen.

LÄHTEET

Cambridge Dictionary: Love triangle s.a. Cambridgen yliopisto. WWW-dokumentti. Saatavissa: https://dictionary.cambridge.org/dictionary/english/love-triangle [viitattu 7.11.2024].

Codecademy. 2023a. Classes. WWW-dokumentti. Saatavissa: https://www.codecademy.com/resources/docs/c-sharp/classes [viitattu 22.11.2024].

Codecademy. 2023b. Scope. WWW-dokumentti. Saatavissa: https://www.codecademy.com/resources/docs/general/scope [viitattu 18.11.2024].

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R. & Stafford, J. 2011. Documenting Software Architectures: Views and Beyond. 2. painos. Boston: Addison-Wesley.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, D. 2002. Refactoring Improving the Design of Existing Code. Boston: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. 27. painos. Boston: Addison-Wesley.

ISO-24765:2017. 2017. Systems and software engineering. Vocabulary.

Kananen, J. 2015. Kehittämistutkimuksen kirjoittamisen käytännön opas: Miten kirjoitan kehittämistutkimuksen vaihe vaiheelta. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Kananen, J. 2017. Kehittämistutkimus interventiotutkimuksen muotona: opas opinnäytetyön ja pro gradun kirjoittajalle. Jyväskylä: Jyväskylän ammattikorkeakoulu.

Martin, R. 2009. Clean Code: A Handbook of Agile Software Craftmanship. New Jersey: Prentice Hall.

Microsoft. 2022a. bool (C# reference). WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/bool [viitattu 12.11.2024].

Microsoft. 2022b. Inheritance - derive types to create more specialized behavior. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/inheritance [viitattu 4.11.2024].

Microsoft. 2023a. Constructors (C# programming guide). WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/constructors [viitattu 20.11.2024].

Microsoft. 2023b. Polymorphism. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/object-oriented/polymorphism [viitattu 8.11.2024].

Microsoft. 2024a. abstract (C# Reference). WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract [viitattu 12.11.2024].

Microsoft. 2024b. Access Modifiers (C# Programming Guide). WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/access-modifiers [viitattu: 30.10.2024].

Microsoft. 2024c. The C# type system. WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/ [viitattu 8.11.2024].

Microsoft. 2024d. virtual (C# Reference). WWW-dokumentti. Saatavissa: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual [viitattu 12.11.2024].

Nystrom, R. 2014. Observer. WWW-dokumentti. Saatavissa: https://gameprogrammingpatterns.com/observer.html [viitattu 20.11.2024].

Pernaa, J. 2013. Kehittämistutkimus opetusalalla. Jyväskylä: PS-kustannus.

Unity Technologies. 2018. ScriptableObject. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2022.3/Documentation/Manual/class-ScriptableObject.html [viitattu 4.11.2024].

Unity Technologies. 2024. ScriptableObject. WWW-dokumentti. Saatavissa: https://docs.unity3d.com/2022.3/Documentation/ScriptReference/ScriptableObject.html [viitattu 4.11.2024].

Unity. 2017. Unite Austin 2017 - Game Architecture with Scriptable Objects. Youtube. Videoleike. Saatavissa: https://www.youtube.com/watch?v=raQ3iHhE_Kk [viitattu 30.10.2024].

Visser, J. 2016. Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code. Sebastopol: O'Reilly Media.

Visual Paradigm. s.a. UML Class Diagram Tutorial. WWW-dokumentti. Saatavissa: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/ [viitattu 21.11.2024].

C# Comments. s.a. W3Schools. WWW-dokumentti. Saatavissa: https://www.w3schools.com/cs/cs_comments.php [viitattu 21.11.2024].

C# Method Parameters. s.a. W3Schools. WWW-dokumentti. Saatavissa: https://www.w3schools.com/cs/cs_method_parameters.php [viitattu 18.11.2024].