OAMK OULUN AMMATTIKORKEAKOULU

SOLID-PERIAATTEET JA SUUNNITTELUMALLIT 2D-PELIN KEHITYKSESSÄ

Kalle Määttä
Opinnäytetyö AMK
Kevät 2025
Tietotekniikan tutkinto-ohjelma
Ohjelmistokehitys
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma
Ohjelmistokehitys

Tekijä: Kalle Määttä
Opinnäytetyön otsikko: SOLID-periaatteet ja suunnittelumallit 2D-pelin kehityksessä
Työn ohjaaja: Janne Kumpuoja
Työn valmistumislukukausi ja -vuosi: Kevät 2025
Sivumäärä: 39 + 1 liite

Tietokonepelien kehittäminen on pitkä prosessi, jossa työstettävä koodikokonaisuus kasvaa suureksi. Kehittämisen aikana on tärkeää kirjoittaa koodia, joka kestää koko työprosessin ajan. SOLID-periaatteet ohjaavat puhtaan ja skaalautuvan koodin kirjoittamiseen. Peleissä on usein monia erilaisia monimutkaisia toimintoja, joiden tulee toimia yhdessä moitteettomasti. Suunnittelumallit tarjoavat valmiiksi ratkaisuja ja testattuja tapoja toteuttaa näitä toimintoja.

Opinnäytetyön tavoitteena oli perehtyä SOLID-periaatteisiin ja valittuihin suunnittelumalleihin, ja tutkia, miten ne ovat toteutuneet kehitetyssä pelissä. Vertailun kohteena oleva 2D-toimintatasohyppelypeli oli kehitetty Unity-pelimoottorilla.

Periaatteisiin ja suunnittelumalleihin ei ollut tutustuttu syvemmin eikä niitä ollut tarkoituksellisesti seurattu kehityksen aikana. Työn teoriaosuudessa periaatteet ja suunnittelumallit käydään läpi yleisen ohjelmistokehityksen kannalta. Kehitetystä pelistä kuvataan yleisesti valmiina olevia toimintoja. SOLID-periaatteiden ja suunnittelumallien toteutumiset pelissä avataan kaavioiden ja kuvien avulla. Lopuksi tutkitaan, miten toteutuneet periaatteet ja suunnittelumallit vaikuttivat pelin toiminnallisuuksiin ja itse koodiin. Vaikutuksien pohjalta mietitään pelin toiminnallisuuksien parannuskohteita.

Opinnäytetyön tulosten perusteella voidaan todeta SOLID-periaatteiden ja suunnittelumallien tuovan monia positiivisia vaikutuksia pelin koodirakenteeseen. SOLID-periaatteiden toteutuminen toivat selkeyttä koodiin ja mahdollistivat pelin nopean skaalautumisen. Suunnittelumallit tarjosivat kestäviä ratkaisuja pelin monimutkaisempien toimintojen toteutuksiin. Syvällisempi ymmärrys siitä, miten periaatteiden ja suunnittelumallien teoria toteutuu käytännössä, mahdollistaa pelin jatkokehityksen parantamisen harkituilla ratkaisuilla.

ABSTRACT

Oulu University of Applied Sciences
Degree Program in Information Technology
Option of Software Development

Author: Kalle Määttä
Title of thesis: SOLID-principles and Design Patterns in 2D-game Development
Supervisor: Janne Kumpuoja
Term and year when the thesis was submitted: Spring 2025
Number of pages: 39 + 1 appendix

Developing computer games is hard and long process that requires developers to make and maintain a large codebase. This task can be made easier by following SOLID principles and design patterns. SOLID principles are guidelines that guide to more clean and scalable code. Design patterns give instructions and ideas to solve game developments more complicated functional problems.

The goal of this thesis was to research the SOLID principles and chosen design patterns and compare them to a 2D-game. During the initial game development, the main goal was not to follow SOLID principles and design patterns. The game was built on Unity engine. This study describes the games ready functionalities shortly. The SOLID principles and design patterns usage in the game is described and reviewed in the compare section.

Based on the results of the thesis, it can be stated that SOLID principles and design patterns bring many different benefits to the game. The main benefits of SOLID principles are improved code readability and scalability. Design patterns give answers to more complicated coding problems, that can be reused and adapted if understood well. A better understanding of the SOLID principles and design patterns makes it possible to continue game development with well-structured and maintainable code.

SISÄLLYS

TIIVISTELMÄ 2
ABSTRACT 3
SISÄLLYS 4
1 JOHDANTO 5
2 SOLID-PERIAATTEET 7
2.1 Yhden vastuun periaate 7
2.2 Avoin-suljettu periaate 8
2.3 Liskovin korvausperiaate 10
2.4 Rajapinnan erotteluperiaate 12
2.5 Riippuvuuden käänteisyyden periaate 12
3 SUUNNITTELUMALLIT 14
3.1 Tila-malli ja tilakone 14
3.2 Ainokainen-malli 16
3.3 Oliovarasto-malli 17
3.4 Tarkkailija-malli 18
4 2D-TOIMINTAPELI UNITYSSÄ 20
4.1 Pelaajahahmo 20
4.2 Kentät ja ympäristöt 21
4.3 Viholliset 23
4.4 Taistelusysteemi 24
5 SOLID-PERIAATTEIDEN JA SUUNNITTELUMALLIEN TOTEUTUMINEN 2D-PELISSÄ 26
5.1 SOLID-periaatteiden esiintyminen pelissä 26
5.2 Tila-mallin ja tilakoneen esiintyminen pelissä 27
5.3 Ainokainen-mallin esiintyminen pelissä 28
5.4 Oliovarasto-mallin esiintyminen pelissä 30
5.5 Tarkkailija-mallin esiintyminen pelissä 30
6 JOHTOPÄÄTÖKSET 32
7 POHDINTA 36
LÄHTEET 38

1 JOHDANTO

Pelit ovat suuria ohjelmistokokonaisuuksia, joissa lukuisat komponentit ja moduulit toimivat yhdessä. Ohjelmistorakenne on tärkeää pitää selkeänä, jotta jatkuva pelin kehittäminen ei hidastu. Sekava koodi tai ohjelmistorakenne hankaloittaa jatkokehityksen kulkua, kun pitää tutkia, mitä aikaisemmin on tehty. Joissain tapauksissa huono koodi voi jopa poissulkea jatkokehityksen mahdollisuuksia. Sotkuinen koodi myös hidastaa pelin suorituskykyä, mikä vastaavasti huonontaa pelaajakokemusta. Pahimmassa tapauksessa kehnot koodiratkaisut voivat johtaa pelin kaatumiseen, joka kokonaan estää pelin pelaamisen.

Tämän opinnäytetyön aiheena on tutkia, miten SOLID-periaatteet ja suunnittelumallit vaikuttivat kehitettyyn 2D-peliin. Peliä kehitettiin ennen opinnäytetyön alkua, jolloin SOLID-periaatteisiin ja suunnittelumalleihin ei ollut syvemmin tutustuttu. Työn tavoitteena on perehtyä tarkemmin periaatteiden ja suunnittelumallien teoriaan sekä tunnistaa, mitkä pelin toiminnot toteuttavat niitä. Pelin toimintojen vertailulla teoriaan on tarkoitus tuoda esille, miten SOLID-periaatteista ja suunnittelumallit vaikuttivat käytännössä pelinkehityksessä. Tavoitteena on myös tunnistaa, miten periaatteita ja suunnittelumalleja voitaisiin seurata pelissä paremmin.

Luvussa 2 käydään läpi SOLID-periaatteiden teoria, ja havainnollistetaan periaatteita UML-kaavioiden avulla. Eri suunnittelumallien teoria käsitellään luvussa 3. Suunnittelumalleja on lukuisa, joten käsiteltävien suunnittelumallien määrä rajattiin neljään. Esiintyvät suunnittelumallit valikoitiin niiden merkityksellisyyden perusteella kehitettävän pelin kannalta. Suunnittelumalleista valittiin tila-, ainokainen-, oliovarasto- ja tarkkailija-mallit. Kehitetyn 2D-pelin tyylilaji on toimintatasohyppely, mikä on kehitetty Unity-pelimoottorilla. Pelin päätoiminnot esitellään luvussa 4. Päätoiminnoista käydään läpi pelaajahahmo, kentät ja ympäristöt, viholliset ja taistelusysteemi. Peliä kehitettiin yksin, eikä sillä ollut rahallisia tavoitteita. Kehityksen tavoitteena oli saada syvempi ymmärrys pelien toiminnasta ja parantaa tekijän ohjelmointitaitoja pelinkehityksen kannalta. Luvussa 5 kerrotaan, miten SOLID-periaatteet ja suunnittelumallit ovat toteutuneet pelissä sekä miten

niiden toteutus näkyy koodirakenteessa. Lopuksi luvussa 6 mietitään toteutumien pohjalta johtopäätöksiä.

2 SOLID-PERIAATTEET

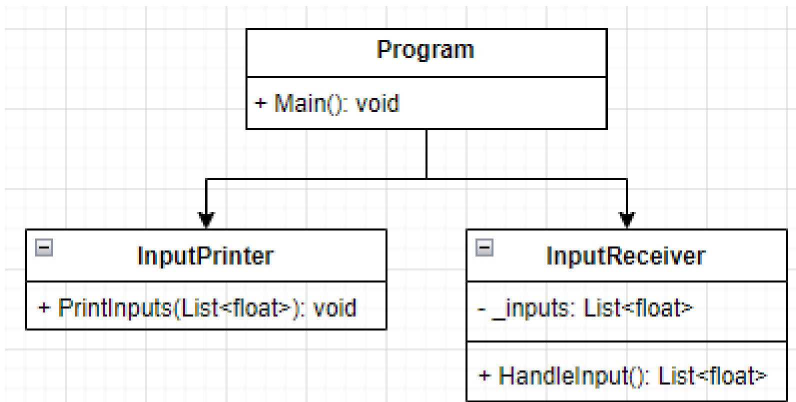
SOLID-lyhenne tulee oliopohjaisen ohjelmistokehityksen viidestä eri periaatteesta: yhden vastuun periaate (single responsibility), avoin-suljettu periaate (open-closed), Liskovin korvausperiaate (Liskov substitution), rajapinnan erotteluperiaate (interface segregation) ja riippuvuuden käänteisyyden periaate (dependency inversion). Nämä periaatteet ovat joukko sääntöjä ja kehotuksia, joita seuraamalla kehitettävästä ohjelmasta saadaan helppolukuinen, testattava ja skaalautuva. (Yiğit 2020.)

2.1 Yhden vastuun periaate

Yhden vastuun periaate määrää, että ohjelman menetelmällä, luokalla tai funktiolla on vain yksi vastuu. Tätä luokkaa ei tarvitse muokata, jos ohjelmaan lisätään lisää toiminnallisuuksia. Pienemmät luokat hoitavat yhden tehtävän ja kommunikoivat keskenään muodostaen suuremman kokonaisuuden. (Lin 2021, 24.)

Yhden vastuun periaatteen noudattaminen tarjoaa monia hyötyjä. Pienempiä luokkia on helpompi testata, jolloin ohjelman viat löytyvät nopeammin. Kirjoitettua koodia on sujuvampaa lukea, kun luokat ovat kompakteja. Ohjelmaan voi vaivattomasti lisätä toiminnallisuuksia tekemällä uusia luokkia. Yhdistämisristiriidat vähentyvät, kun aikaisempaa koodia ei tarvitse muokata. Projektin versionhallinta on helpompaa ja nopeampaa. (Yiğit 2020.)

Yksinkertaisena esimerkkinä konsoliohjelma, joka vastaanottaa liukulukuja ja tulostaa ne konsoliin. Ohjelman toiminnallisuus on jaettu InputReceiver- ja InputPrinter-luokille (kuva 1). Kummallakin luokalla on vain yksi tehtävä, josta ne ovat vastuussa ohjelman suoritusaikana.



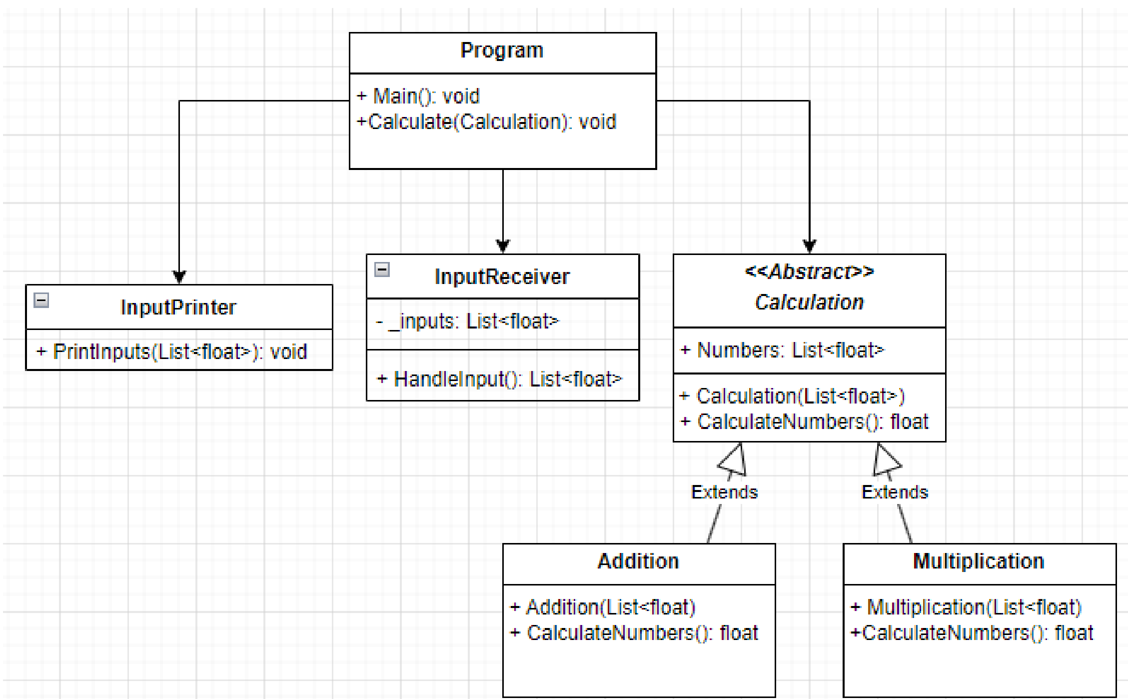
KUVA 1. Konsoliohjelman InputPrinter- ja InputReceiver luokat

2.2 Avoin-suljettu periaate

Avoin-suljettu periaate määrää, että luokkien pitää olla avoimia laajennuksille ja suljettuja muutoksille. Luokkia tulee suunnitella siten, että uusia toiminnallisuuksia pystytään lisäämään, ilman aikaisemmin kirjoitetun koodin muokkaamista. Aikaisemman koodin muokkaaminen mahdollistaa uusien virheiden syntymisen. (Yiğit 2020.)

Avoin-suljettu periaate voidaan saavuttaa käyttämällä abstraktiota. C# kielessä abstraktion tarkoituksena on piilottaa ohjelman sisäiset yksityiskohdat ja näyttää vain toiminnallisuudet käyttäjälle. Tämä voidaan toteuttaa abstrakteilla luokilla ja metodeilla. Abstraktista luokasta ei luoda suoraan oliota, vaan se toimii pohjapiirustuksena siitä periville alaluokille. Abstrakti luokan metodeilla ei ole omia toteutuksia. Perivien alaluokkien tulee toteuttaa abstraktit metodit omalla tavallaan. (GeeksForGeeks 2023.)

Abstrakti luokka Calculation määrittelee abstraktin metodin CalculateNumbers, jolla ei ole toteutusta. Addition ja Multiplication alaluokat toteuttavat CalculateNumbers metodin. (Kuva 2.) Alaluokat suorittavat omat laskutoimitukset ja palauttavat tuloksen.



KUVA 2. Avoin-suljettu periaate saavutetaan abstraktilla luokalla

Program-luokka käyttää hyväkseen dynaamista polymorfismia, joka edentää avoin-suljettua periaatetta (kuva 3). Calculate-metodi mahdollistaa eri olioiden käyttöä yhden rajapinnan kautta. Metodin parametrina voi olla mikä tahansa Calculation-luokan alaluokka. Kutsuttava metodi määritetään ohjelman ajonaikana riippuen siitä, minkä olion Calculate-metodi vastaanottaa. Uusien laskutoimituksien lisääminen ohjelmaan vaatii vain oman luokan toteutuksen ja yhden rivin Program-luokkaan.

```
class Program
{
    0 references
    static void Main()
    {
        InputReceiver inputReceiver = new();
        var inputs = inputReceiver.HandleInput();

        InputPrinter inputPrinter = new();
        inputPrinter.PrintInputs(inputs);

        Calculate(new Addition(inputs));
        Calculate(new Multiplication(inputs));
    }
    2 references
    static void Calculate(Calculation calculation)
    {
        var result = calculation.CalculateNumbers();
        Console.WriteLine($"Result of the {calculation.GetType().Name}: {result}");
    }
}
```

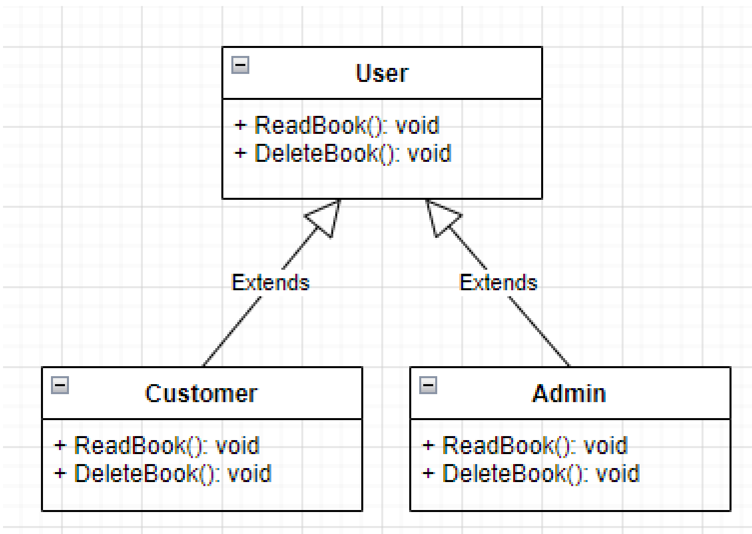
KUVA 3. Calculate-metodi vastaanottaa parametrina Calculation-luokan

2.3 Liskovin korvausperiaate

Liskovin korvausperiaate määrää, että kantaluokan pitää pystyä korvaamaan alaluokalla. Metodi, joka odottaa parametrina kantaluokkaa, voi myös vastaanottaa alaluokan olion ilman, että ohjelma rikkoutuu. (Yiğit 2020.)

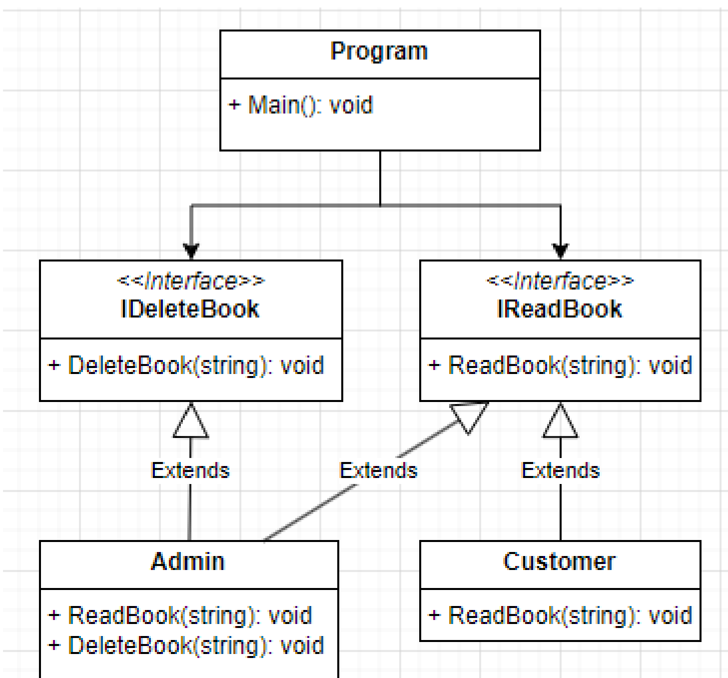
Aikaisempi Calculate-metodi antaa hyvän mallin, miten Liskovin korvausperiaate toimii käytännössä (kuva 3). Aliluokat voivat korvata Calculation-luokan ilman, että ohjelman toiminnallisuudessa olisi ongelmia. On tärkeää, että aliluokat toteuttavat kantaluokkien julkiset metodit, jotta säästytään odottamattomilta tilanteilta. Perittyjä metodeja ei myöskään voi jättää tyhjiksi, sillä aliluokkien tulee käyttäytyä samalla tavalla kuin kantaluokka (Lin 2021, 18).

Kaaviossa kuvataan ohjelmaa, jossa Customer- ja Admin-luokat perivät User luokan (kuva 4). Ohjelman toiminnallisuuksissa sanotaan, että asiakas ei pysty poistamaan kirjoja. DeleteBook-metodia ei täten voi toteuttaa Customer luokassa.



KUVA 4. Liskovin korvausperiaatetta rikkova luokkakaavio

Liskovin korvausperiaatetta voidaan seurata myös käyttämällä rajapintoja. Rajapinnasta ei voi luoda oliota, ja sen metodeilla ei voi olla toteutusta. Rajapinnan sisällä olevat metodit ovat samankaltaisia kuin abstraktin luokan abstraktit metodit. Ne on pakko toteuttaa rajapinnan toteuttavan luokan sisällä. Rajapintojen vahvuus on, että luokat voivat toteuttaa useita eri rajapintoja. (JavatPoint s.a.)



KUVA 5. Liskovin korvausperiaate saavutettu rajapintoja käyttäen

Ohjelmassa kaksi eri rajapintaa IDeleteBook ja IReadBook (kuva 5). Customer luokka toteuttaa IReadBook rajapinnan (kuva 5). Admin luokka toteuttaa

IReadBook ja IDeleteBook rajapinnat, joten se voi sekä lukea että poistaa kirjoja (kuva 5). Program luokassa kutsutaan metodit käyttämällä rajapintoja (kuva 6).

```
class Program
{
    1 reference
    static void Main()
    {
        IReadBook read = new Admin();
        read.ReadBook("Faust");

        IDeleteBook delete = new Admin();
        delete.DeleteBook("Moby Dick");

        read = new Customer();
        read.ReadBook("The Great Gatsby");
    }
}
```

KUVA 6. Kuvassa kutsutaan metodeja rajapintojen kautta

2.4 Rajapinnan erotteluperiaate

Rajapinnan erotteluperiaatteen mukaan asiakkaiden ei saa olla riippuvaisia rajapinnoista, joita ne eivät käytä. Rajapintojen tulee olla yksiselitteisiä eikä liian suuria. On parempi tehdä monta eri rajapintaa, jotka kommunikoivat tiettyjen asiakkaiden kanssa, kuin yksi suuri monikäyttöinen rajapinta. (Lin 2021, 21.)

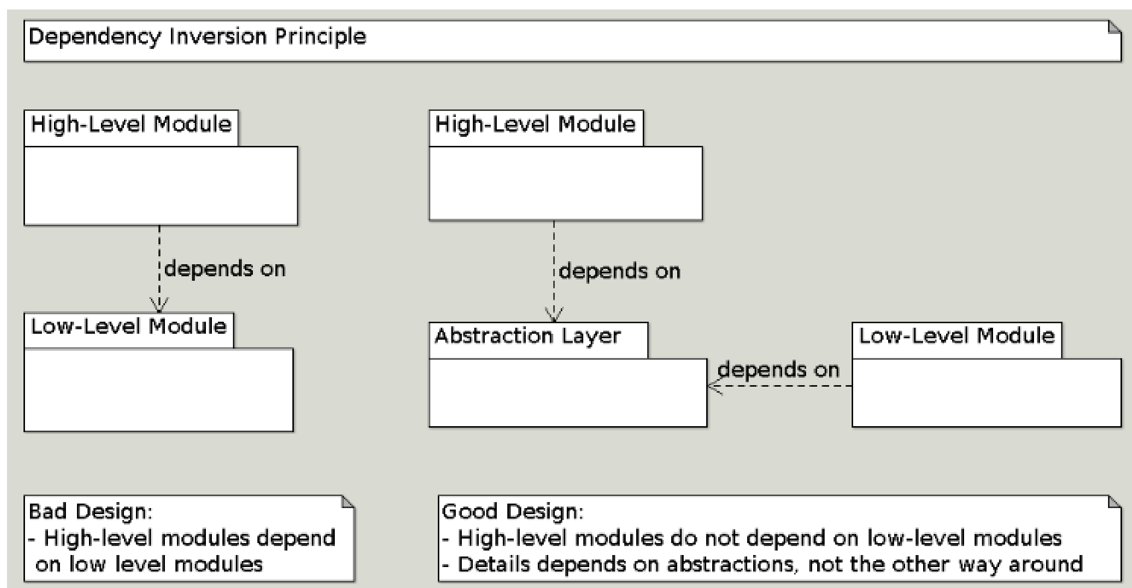
Tämä periaate ajaa tietyllä tapaa samaa asiaa kuin yksittäisen vastuun periaate. Rajapinnoilla on yksi vastuu, jonka ne hoitavat hyvin. Aikaisempi kaavio seuraa rajapinnan erotteluperiaatetta (kuva 5). Rajapinnoissa ei ole monia eri metodeja, jotta niitä voidaan toteuttaa tarvittaessa monessa eri luokassa.

2.5 Riippuvuuden käänteisyyden periaate

Riippuvuuden käänteisyyden periaatteen mukaan korkean tason moduulien ei tulisi olla riippuvaisia alemman tason moduuleista. Toimijoiden välinen suhde tulisi riippua abstraktiosta. Tämä vähentää moduulien välisiä kytkentöjä, joka johtaa helpommin skaalattavaan koodiin. (OODesign.com s.a.)

Ohjelman korkean tason moduuleilla tai luokilla, tarkoitetaan luokkia, jotka sisältävät monimutkaisemman logiikan. Alemman tason luokat toteuttavat perustoimintoja. Ilman SOLID- periaatteita korkean tason luokka on riippuvainen alemman tason luokasta suorittaakseen haluttuja tehtäviä (kuva 7). Tämä tuottaa ongelmia, jos alemman tason luokka pitää korvata. (Lin 2021, 24; OOdesign.com s.a.)

Käytännössä tämä tarkoittaa, että luokkien välillä on abstraktitaso, jota käytetään funktio kutsuissa (kuva 7). Abstraktiotaso tuo monia eri hyötyjä. Komponentteja voi vapaasti muokata, ilman että se vaikuttaa koko ohjelmaan. Uusia ominaisuuksia voidaan lisätä nopeasti abstraktiotason kautta. Tätä kautta ohjelman testattavuus myös paranee. Testiluokkien- ja objektien tekeminen suoraviivaistuu, jolloin koko ohjelman laatu paranee. (Karropoulos 2023.)



KUVA 7. Kuvaus riippuvuuden käänteisyyden periaatteesta (OODesign.com s.a.)

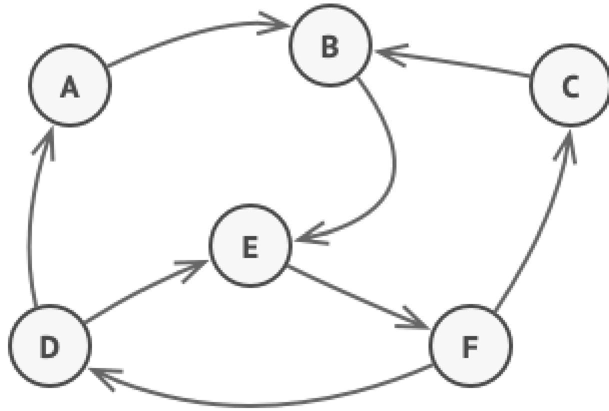
3 SUUNNITTELUMALLIT

Suunnittelumallit ovat yleisiä ratkaisuja ohjelmoinnin yleisesti esiintyviin ongelmiin. Mallit eivät anna suoraa ratkaisua koodiongelmiin, mutta antavat korkeamman tason ohjeistusta, miten ongelmaa tulisi lähestyä. Nämä auttavat käyttäjää kirjoittamaan tehokkaampaa ja ylläpidettävää koodia. Suunnittelumallit voidaan jakaa kolmeen eri kategoriaan. Luontimallit abstraktoivat olioiden ilmentämisen, ja tekevät järjestelmästä riippumattoman siitä, miten oliot luodaan. Rakennemallit antavat ratkaisuja, miten oliot ja luokat kannattaa luoda, jotta ne muodostavat järkeviä suurempia kokonaisuuksia. Käyttäytymismallit huolehtivat olioiden välisistä tehtävistä ja kommunikoimisesta. (Gamma, Helm, Johnson, & Vlissides 1994, kappale 3, 4, 5.)

Tutkimus rajattiin neljään suunnittelumalliin, jotka valittiin niiden esiintymisen sekä arvioidun merkityksen perusteella kehitetyssä pelissä. Käsiteltäviksi suunnittelumalleiksi valittiin käyttäytymismallit tila ja tarkkailija sekä luontimallit ainokainen ja oliovarasto.

3.1 Tila-malli ja tilakone

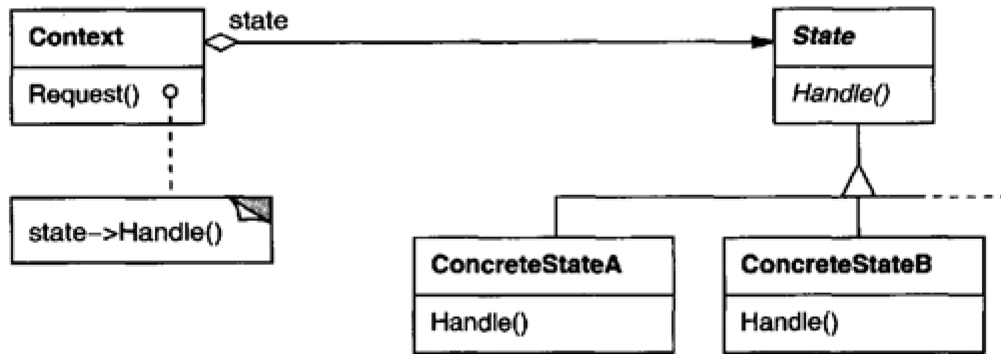
Tilakoneessa on ohjelman ajoaikana rajallinen määrä tiloja, joissa ohjelma voi olla (kuva 8). Ohjelma voidaan vaihtaa tilasta toiseen välittömästi, jossa ohjelma toimii eri tavalla. Tilojen välisiä vaihdoksia kutsutaan siirtymiksi, ja ne määräävät mistä tietystä tilasta voidaan vaihtaa toiseen tilaan. (Shvets 2023, 354.)



KUVA 8. Tilakoneen tilat ja siirtymät kuvattuna (Shvets 2023, 354.)

Tila-malli mahdollistaa olion muuttaa käyttäytymistään, kun sen sisäinen tila vaihtuu. Mallin mukaan kaikille olion mahdollisille tiloille tehdään oma luokka. Luokkien sisälle toteutetaan tilakohtaiset käyttäytymiset. Alkuperäinen olio tallentaa viitteen luokkaan, joka kuvastaa olion tämänhetkistä tilaa. Tämä luokka on vastuussa olion tietyn tilan vaatimista tehtävistä. (Shvets 2023, 357.)

Alkuperäistä oliota kuvataan sanalla konteksti (kuva 9). Konteksti määrittelee rajapinnan asiakkaille ja ylläpitää viittausta konkreettiseen aliluokkaan. Tila komponentti määrittelee rajapinnan kontekstille, mikä sisältää metodit, jotka aliluokat toteuttavat tilakohtaisesti. Asiakkaat voivat määritellä kontekstin tilan konkreettisilla olioilla, ja tehdä pyyntöjä kontekstin kautta. Kun tilajärjestelmään saapuu asiakkaalta pyyntö, välittää konteksti sen eteenpäin aliluokalle rajapinnan kautta. Tilojen vaihdon logiikka voidaan määritellä kontekstissa tai aliluokissa. (Gamma ym. 1994, kappale 5.)

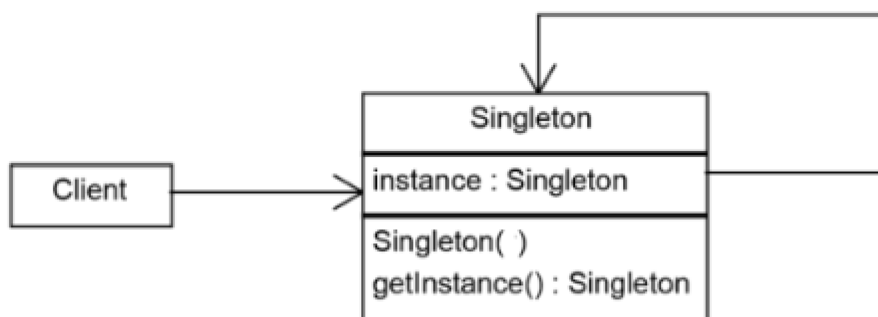


KUVA 9. Kaavio tila-mallin peruskomponenteista (Gamma ym. 1994, kappale 5.)

3.2 Ainokainen-malli

Ainokainen-malli pitää huolen, että luokalla on vain yksi instanssi. Luokka, joka on suunniteltu ainokaiseksi, on itse vastuussa ainoan instanssinsa hallinnasta. Ainokainen varmistaa, ettei luokasta voida luoda lisää instansseja. Se tarjoaa globaalin pääsykohdan luokan ainoalle instanssille, jota voidaan käyttää mistä tahansa ohjelman osasta. (Shvets 2023, 139–140.)

Ainokais-mallia toteutettaessa on tärkeää tehdä oletuskonstruktorista yksityinen, jotta luokkaan ei voida tehdä kutsuja konstruktorin kautta. Luokan konstruktoria imitoi staattinen metodi, joka luo olion kutsumalla yksityistä konstruktoria luokan sisällä. Olio tallennetaan staattiseen kenttään. Asiakkaat voivat kutsuvat staattista metodia, joka palauttaa aina saman tallennetun olion. (Shvets 2023, 140–141.) Kuvassa 10 nähdään Ainokainen-malli toimimassa asiakkaan kanssa. Asiakas kutsuu getInstance-staattista metodia, joka tarvittaessa luo olion ja palauttaa sen asiakkaan käyttöön.



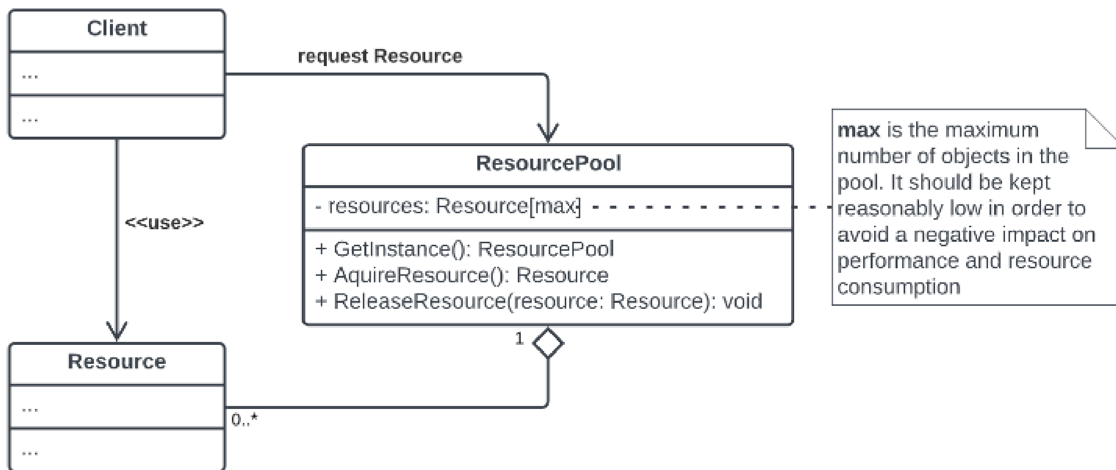
KUVA 10. Ainokainen-mallin toteutus (GeeksForGeeks 2024.)

Ainokainen-mallin käyttö on kiistanalainen aihe ohjelmoijien parissa. Nystromin mukaan mallin väärinkäyttäminen voi johtaa ongelmiin koodikannassa pidemmällä aikavälillä. Ainokainen-malli tekee koodista vaikeammin luettavaa, koska niitä voidaan kutsua mistä tahansa. Tämä voi tehdä bugien löytämisen vaikeammaksi kuin on tarpeen, varsinkin kun työstetään isompaa projektia, johon osallistuu useita koodareita. Mallin käyttö luo kytköksiä komponenttien väleille, joiden olisi parempi olla erillään. (Nystrom 2014, kappale 6.)

3.3 Oliovarasto-malli

Oliovarasto-malli on optimointitekniikka, jonka tarkoituksena on helpottaa järjestelmän kuormitusta, kun luodaan ja poistetaan monia olioita. Mallin mukaan ohjelmassa käytetään valmiiksi luotuja ja alustettuja olioita odotustilassa, kunnes niitä tarvitaan. Luodut oliot odottavat deaktiivisena varastossa, josta niitä voidaan tarvittaessa kutsua. Ohjelma ei luo uusia instansseja olioista, vaan aktivoi ne. Tarvittavan käytön jälkeen olion tila vaihdetaan takaisin deaktiiviseksi ja palautetaan varastoon. (Lin 2021, 46.)

Kuvassa 11 kuvataan Oliovarasto-mallin toimintaa asiakkaan kanssa. Ghergun mukaan ohjelman ajon aikana oliovarasto ensimmäisenä etsii vapaata Resource oliota, ja palauttaa sen asiakkaalle. Jos vapaata oliota ei löydy, ResourcePool luokka yrittää luoda uuden olion, joka palautetaan asiakkaalle. Oliovaraston olioiden maksimimäärän saavutettua varasto jää odottamaan, kunnes käytettävä olio on vapautettu. Asiakkaat eivät tiedä, että he jakavat olioita. Asiakkaan näkökulmasta se omistavaa uuden olion, joka tuli oliovarastosta. Asiakas on vastuussa Resource olion statuksen vaihtamisen vapaaksi, jotta toinen asiakas voi käyttää oliota uudelleen. (Ghergu 2023.)

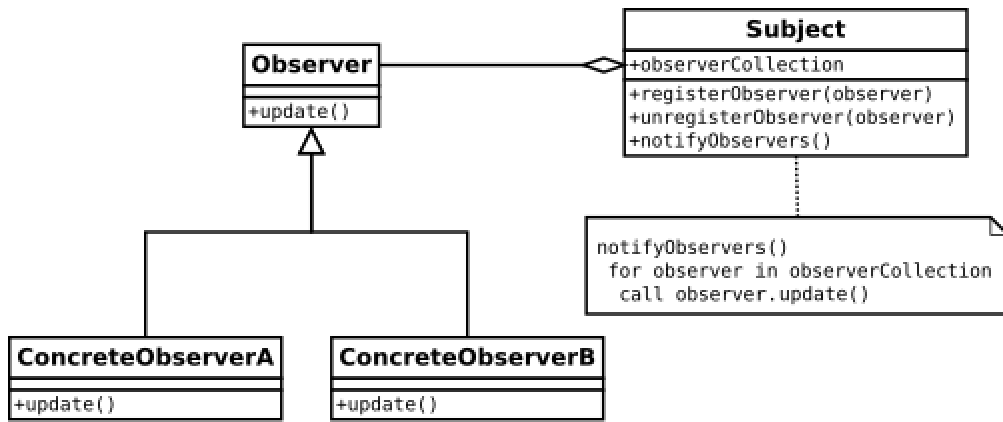


KUVA 11. Oliovarasto-mallin peruskaavio (Ghergu 2023.)

3.4 Tarkkailija-malli

Tarkkailija-malli määrittelee yksi-moneen-suhteen olioiden välille, jossa yhden olion muutoksista ilmoitetaan olioille, joilla on siihen suhde. Suhteessa yhtä oliota kutsutaan subjektiksi ja monia olioita tarkkailijoiksi. Tarkkailijoita voi olla mikä määrä tahansa. Kaikki subjektia kuuntelevat tarkkailijat saavat ilmoituksen, kun subjektin tila muuttuu. Tarkkailijat hakevat tarvittavat tiedot subjektista samalla päivittäen oman tilan vastaamaan subjektin tilaa ja suorittavat määritetyt tehtävät. (Gamma ym. 1994, kappale 5.)

Tarkkailija-mallin toteutuksessa subjekti-luokka sisältää array-kentän, joka pitää sisällään viittauksia tarkkailijoista. Tarvitaan myös useampi julkinen metodi, joiden kautta voidaan lisätä ja poistaa tarkkailijoita listasta. Tapahtumasta ilmoitettaessa subjekti-luokka käy läpi sen tarkkailijat ja kutsuu niiden objektien määritettyjä metodeja. Tarkkailija-mallissa tärkeässä roolissa on rajapinta, jonka kaikki tarkkailijat perivät. Rajapinta määrittelee ilmoituksen metodin ja parametrit, joiden avulla voidaan lähettää tarvittavaa dataa tarkkailijoille ilmoituksen mukana. Subjekti kommunikoi tarkkailijoiden kanssa vain rajapinnan kautta, jotta vältytään turhilta kytköksiltä. Kytköksiä on tärkeää välttää, sillä tarkkailijat voivat vaihtua ohjelman suoritusaikana. Kuvassa nähdään diagrammi tarkkailija-mallista, jossa kuvattuna subjekti, rajapinta ja tarkkailijat (kuva 12). (Shvets 2023, 339–340.)



KUVA 12. Tarkkailija-mallin tarkkailijat ja subjekti UML-kaaviossa (Wikipedia 2024.)

4 2D-TOIMINTAPELI UNITYSSÄ

Kehityksessä oleva peli on Metroidvania-tyylinen 2D-peli, joka on kehitetty Unity-pelimoottorilla. Pelin tavoitteena on luoda kiinnostava maailma, joka houkuttelee pelaajaa tutkimaan, sekä luoda yksinkertainen mutta palkitseva taistelujärjestelmä. Peli on saanut inspiraatiota muista menestyneistä 2D- peleistä, kuten Hollow Knightista ja Blasphemousista.

Metroidvania-lajityyppi tulee vanhoista klassikko peleistä Metroid ja Castlevania. Näistä peleistä löytyvistä elementeistä on kehittynyt oma tyylilaji, jota monet menestyneet pelit ovat seuranneet. Metroidvania-tyyliset pelit ovat 2D-tasohyppelyn ja toimintaseikkailun yhdistäviä pelejä. Muita tunnusmerkkejä ovat pelikenttien tutkiminen, hahmon kehittäminen ja ei-lineaarinen pelin eteneminen. Tyylilajin peleissä ei usein keskitytä tarinan kertomiseen suoraan pelaajalle, vaan pelaaja voi selvittää pelimaailman tapahtumat tutkimalla ympäristöjä. (Nutt 2015.)

4.1 Pelaajahahmo

Kehitettävän pelin kulku on nopeatempoinen, ja usein vaatii tarkkoja syöttökomentoja. Peli vaatii tarkkaa hyppelyä alustoilla ja vihollisten hyökkäyksien väistämistä. Pelattava hahmo suunniteltiin näkymään ruudussa suhteellisen pienenä, jotta pelaaja näkisi mahdollisimman paljon informaatiota ympäristön eri kappaleista (kuva 13).

KUVA 13. Pelaajahahmo laskeutumassa laatalle

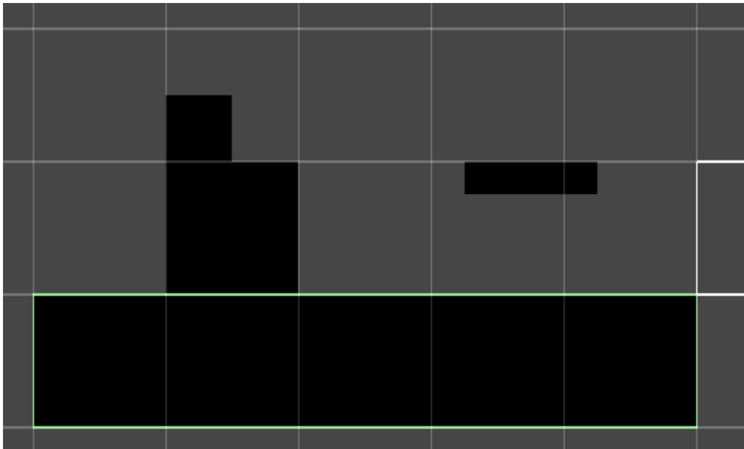
Kehittäessä pelattavaa hahmoa oli tärkeää, että se reagoi nopeasti komentoihin. Pelattavan hahmon eri toimintojen piti vaihtua sulavasti ja nopeasti. Pelaajahahmon liikkumiseen liittyvät erityiskyvyt ovat seiniin tarttuminen ja niistä hyppääminen ja nopea syöksyminen väistääkseen vihollisten hyökkäyksiä. Miekan iskujen lisäksi taisteluun liittyviä kykyjä on vihollisten hyökkäysten torjuminen. Kun torjuminen ajoitetaan oikein, voi pelaaja tehdä erikoisiskun, joka aiheuttaa paljon vahinkoa viholliseen. Pelaaja voi palauttaa terveyspisteitä tai taikaenergiaa käyttämällä verta, jota kerääntyy vahingoittamalla vihollisia.

Hahmon animaatiot piirrettiin avoimen lähdekoodin kuvankäsittelyohjelmalla Kritalla. Animaatiot tuotiin Unityyn, jossa niitä vaihdellaan käyttämällä Unityn Animator-komponenttia (liite 1). Eri animaatio tiloja on yli 25, ja yksittäisten avainkehysten määrä nousi kolminumeroiseksi.

4.2 Kentät ja ympäristöt

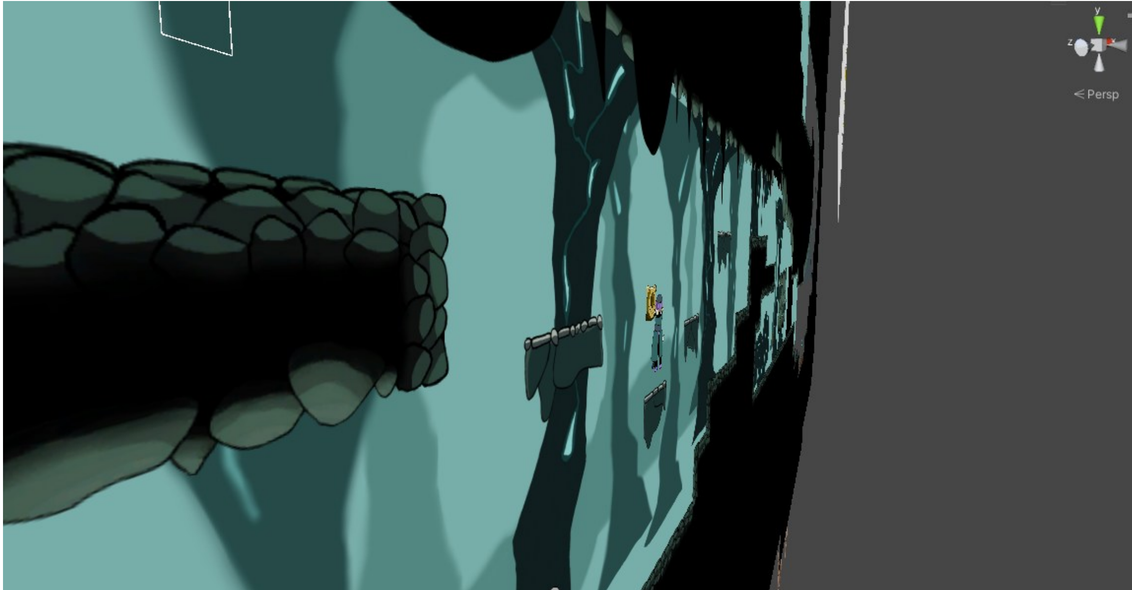
Pelin kentät on rakennettu käyttäen Unityn Tilemap-komponenttia. Pelimaailma on jaettu kolmeen erikokoiseen ruudukkoon, joita voidaan täytellä laatoilla. Laattojen koot ovat 32 x 32, 64 x 64 ja 128 x 128 pikseliä (kuva 14). Suurimpia laattoja käytetään isoimmin kenttien lattioiden ja kattojen rakentamiseen. Pienemmillä

laatoilla on helpompi muokata ympäristöä halutun mukaiseksi. Laattoja voidaan lisätä ja poistaa hiiren klikkauksella. Pelaajan törmäys laattoihin havaitaan Tilemap Collider 2D- ja Rigidbody 2D-komponenttien avulla. Laattojen käyttö mahdollistaa kenttien nopean rakentamisen ja muokkaamisen. Suunniteltuja kenttiä on vaivatonta testata, esimerkiksi tarkistaessa, ovatko laattojen välit liian pienet tai suuret pelaajan hypyn kantamalle.



KUVA 14. Erikokoiset ruudukon laatat

Pelin graafisessa näkymässä pyritään sulauttamaan mustat laatat osaksi luonnollista ympäristöä. Piirretyt resurssit, jotka ovat kosketuksessa laattojen kanssa, on varjostettu siten, että laatat näyttäisivät varjolta. Pelin näkymää tehostaa myös perspektiivinen kamera. Resursseja voidaan sijoittaa eri z-akselin arvoille, jolloin saavutetaan syvyysvaikutelma, jossa taka-alalla olevat resurssit liikkuvat hitaammin kuin etualalla olevat (kuva 15).



KUVA 15. Pelikappaleet eri z-akselin arvoilla

4.3 Viholliset

Pelissä on erilaisia vihollisia, jotka käyttäytyvät eri tavoin (kuva 16). Vihollisten käyttäytymiseen haluttiin luoda useita eri tiloja, jotta ne tuntuisivat interaktiivisilta pelin aikana. Eri tilojen piirtäminen käsin osoittautui liian suureksi työmääräksi, joten vihollisten liikkeet animoitiin käyttäen Spine-animaatio-ohjelmistoa.

KUVA 16. Luuranko viholliset rivissä

Vihollisten tekoälyn luomiseen otettiin käyttöön Opsiven Behaviour Designer työkalu. Behaviour Designerillä rakennettiin käyttäytymispuut, joiden mukaan viholliset tekevät päätöksiä ja reagoivat tilanteisiin (kuva 17). Käyttäytymispuuhun

voidaan lisätä mukautettuja tehtäviä ja ehtotarkastuksia, joiden mukaan puussa liikutaan.

KUVA 17. Puukkoa kantavan luurangon käyttäytymispuu

Behaviour Designerin käyttäytymispuun toiminta voidaan pilkkoa neljään eri tehtävään: toiminto, ehto, yhdistelmä ja koristelija. Toimintatehtävät muuttavat pelin tilaa jollain tavalla, kuten liikuttamalla vihollista. Ehtotehtävät tarkistavat jotain pelin ominaisuutta ja määräävät aktiivisen toimintatehtävän. Yhdistelmätehtävät toimivat vanhempitehtävinä, jotka sisältävät listan alitehtävistä. Koristelijatehtävä on myös vanhempitehtävä, jolla voi olla vain yksi alitehtävä. Tehtävällä muokataan alitehtävän käyttäytymistä, kuten keskeyttämään sen suorittaminen ennenaikaisesti. (Opsive s.a.)

4.4 Taistelusysteemi

Pelattava hahmo käyttää miekkaa iskeäkseen vihollisia ja muita objekteja. Kun pelaaja koskettaa maata, hän voi painaa hyökkäyspainiketta kolme kertaa peräkkäin suorittaakseen kolmen iskun yhdistelmän. Yhdistelmän jälkimmäiset iskut tekevät enemmän vahinkoa. Pelaajan täytyy punnita, suorittaako hän

yhdistelmän loppuun ja aiheuttaa enemmän vahinkoa, vai perääntyykö hän tulevien vihollisten iskujen varalta. Pelattava hahmo voi iskeä kerran ylös, jotta se voi taistella lentäviä vihollisia vastaan. Kun pelattava hahmo on ilmassa voi se iskeä alas. Vihollisten ja tiettyjen esineiden iskeminen yläpuolelta puskee pelaajahahmon takaisin ilmaan. Tämän tekniikan avulla voidaan rakentaa mielenkiintoisia esteitä pelaajalle, jossa esimerkiksi pelaajan täytyy käyttää vihollisia hyväkseen, jotta hän saavuttaa korkealla olevan reunan.

Pelaajan lyöntejä tehostaa partikkeli- ja ääniefektit. Vihollisista lähtevät veritipat (kuva 18) luotiin käyttäen Unityn Particle System-komponenttia. Ääniefektejä muokattiin sopiviksi Audacity ohjelmalla, jonka jälkeen ne tuotiin Unityyn. Pelaajahahmon lyönteihin tuotiin lisää painoa lisäämällä pientä näytön tärinää iskun osuessa viholliseen. Efekti antaa pelaajalle välittömästi palautteen oliko lyönti onnistunut, millä pyritään luomaan taistelua tyydyttävämmäksi.

KUVA 18. Vihollinen päihitetty yhdistelmän viimeisellä iskulla

5 SOLID-PERIAATTEIDEN JA SUUNNITTELUMALLIEN TOTEUTUMINEN 2D-PELISSÄ

Pelin kehitysvaiheessa priorisoitiin selkeyttä ja nopeutta, jotta peli saatiin nopeasti pelattavaksi perustoiminnoilla. Eri toiminnallisuuksien toteutuksia varten suunniteltiin itsenäisesti, etsittiin tietoja foorumeilta ja katsottiin opetusvideoita. Vaikka kehitysvaiheessa ei määränpäänä ollut toteuttaa SOLID-periaatteita ja suunnittelumalleja, osa toteutuksista vastaavat niitä.

5.1 SOLID-periaatteiden esiintyminen pelissä

Yhden vastuun periaate näkyy pelin koodikannan eri tasoilla. Pelin toiminnallisuudet pilkottiin osiin ja eroteltiin luokkiin. Hyvänä esimerkkinä luokkien erottelusta nähdään pelattavan hahmon ja vihollisten monet eri tilat. Mahdollisille liikkeille tehtiin niitä vastaavat luokat, jotka ovat vain vastuussa yhdestä toimenpiteestä.

Avoin-suljettu periaatteen mukaista abstraktiota nähdään pelaajan PlayerState-luokassa, joka määrittelee abstraktit metodit LogicUpdate ja PhysicsUpdate. Näin varmistetaan, että kaikki perivät luokat tekevät tarvittavat tarkastukset LogicUpdate-metodissa ja toteuttavat fysiikkaan liittyvät toiminnot PhysicsUpdate-metodissa. Projektissa käytettiin myös periaatteen mukaisia skriptattavia objekteja. Skriptattaviin objekteihin voidaan tallentaa dataa, joka ei ole kytköksissä koodiin. Luotuja objekteja pystyttiin hiirellä vetämään ja pudottamaan peliobjekteihin Unityn Editor-näkymässä. Toimintoa hyödynnettiin äänitehosteiden ja näytön tärinän datan tallentamiseen.

Peliin luotiin erilaisia objekteja, joiden kanssa pelaaja voi olla vuorovaikutuksessa. Liskovin korvausperiaate toteutuu näiden objektien luokkien perimissä rajapinnoissa. Esimerkiksi objektit, joita lyömällä pelaaja voi hypätä korkeammalle, toteuttavat IBouncable-rajapinnan. Tehdyt rajapinnat seuraavat rajapinnan erotteluperiaatetta. Muita rajapintoja IBouncablen lisäksi ovat muun muassa

lentävissä ammuksissa käytetty IDeflectable ja vahingoitettavissa objekteissa käytetty IDamageable.

Riippuvuuden käänteisyyden periaate esiintyy pelaajan vuorovaikutuksessa muiden peliobjektien kanssa. Sitä käytetiin muun muassa törmäyksien tietojen välityksissä. Pelaajan lyönnin OnTriggerEnter2D-metodissa tarkistetaan, periikö osutun törmäyskoneen objektin sisäinen luokka IDamageable-rajapinnan (kuva 19). Vahinkotapahtuman toimenpiteet voidaan kutsua rajapinnan kautta ilman kytköksiä.

```
Unity Message | 0 references
private void OnTriggerEnter2D(Collider2D collision)
{

    IDamageable damageable = collision.GetComponent<IDamageable>();
    if (damageable != null)
    {
        damageable.Damage(damage, transform.position, knockBack);

        playerBlood.RestoreBloodWithHit(bloodAmountCollected);

        CameraShakeManager.instance.ScreenShakeFromProfile(screenShakeProfile, impulseSource);
    }
```

KUVA 19. Pelaajahahmon iskun tunnistus

5.2 Tila-mallin ja tilakoneen esiintyminen pelissä

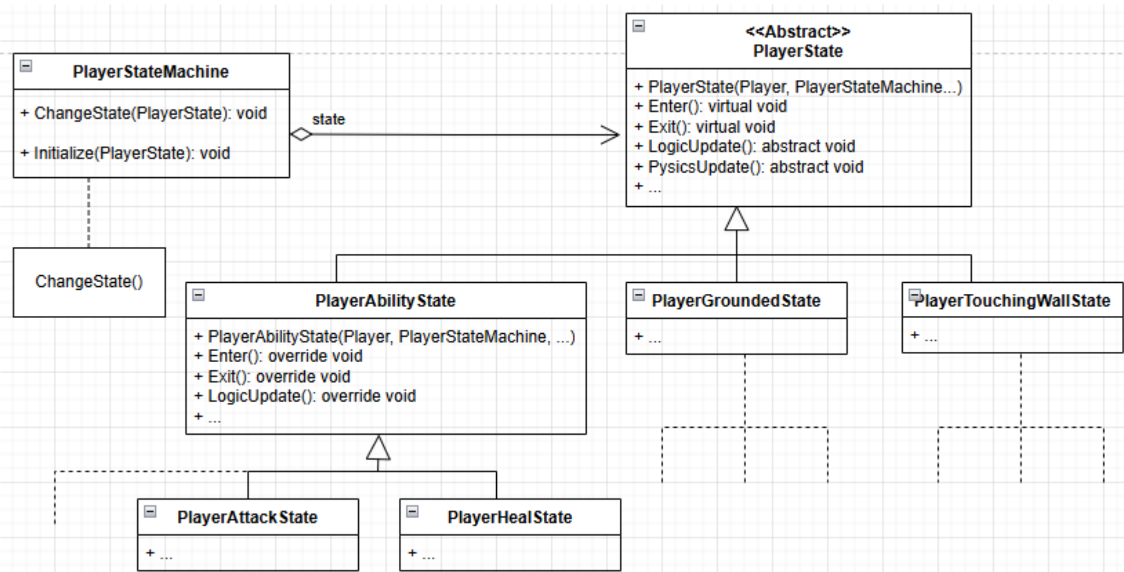
Pelaajahahmon eri toimintojen toteutuksiin tehtiin tilakone, joka toteuttaa tila-mallin piirteitä (kuva 20). Tilakoneen päärakenne koostuu PlayerStateMachine- ja PlayerState-luokista.

PlayerStateMachine-luokka vastaa tila-mallin kontekstiluokkaa (kuva 9). Luokka tarjoaa rajapinnan tilan alustamiseen ja vaihtamiseen asiakkaille, ja pitää viittausta nykyiseen tilaan. Tilakoneen ulkopuolelta tilan vaihtaminen uuteen tapahtuu kutsumalla PlayerStateMachine-luokan ChangeState-metodia. Metodi ottaa parametrina PlayerState-olion, joka vastaa uutta tilaa. Tilakone alustetaan kerran Initialize-metodilla pelaajahahmon pääluokasta pelin alkaessa.

PlayerState-luokka määrittää pelaajahahmon tilojen metodit, jotka konkreettiset luokat toteuttavat. Se on myös vastuussa oikeiden animaatioiden toistamisista. Luokan konstruktori vastaanottaa merkkijonon, mikä vastaa Unityn Animator-

komponentissa asetettuja animaatiosiirtymiä. Animaatiot aloitetaan luokan Enter-metodissa ja lopetetaan Exit-metodissa.

Pelaajahahmon eri konkreettiset tilat muodostuvat kolmesta eri yläluokasta; PlayerAbilityState, PlayerGroundedState ja PlayerTouchingWallState. PlayerAbilityState-luokan aliluokkiin kuuluvat kaikki pelaajahahmon erikoistoiminnot, kuten hyökkääminen, parantaminen ja syöksyminen. Toiminnot, jotka voidaan toteuttaa vain pelaajahahmon koskettaessa maata, kuuluvat Player-GroundedState-luokan aliluokkiin. Näitä toimintoja ovat esimerkiksi paikallaan seisominen, liikkuminen ja maahan laskeutuminen. PlayerTouchingWallState-yläluokan perivät aliluokat toteuttavat toimintoja, kun pelaajahahmo on tarrautunut seinään kiinni. Näitä toimintoja ovat seinää pitkin liukuminen ja seinästä hyppääminen.



KUVA 20. Pelaajahahmon tilakone

5.3 Ainokainen-mallin esiintyminen pelissä

Kehitetyssä pelissä käytetään ainokainen-mallia muutamassa eri komponentissa. Koska pelin luokat perivät Unityn MonoBehaviour-luokan, mallin toteuttaminen eroaa perinteisestä ainokainen-mallista. Alkuperäisessä mallissa hallitaan instanssin luominen ja tarkastaminen getInstance-metodilla (kuva 10). Unityssä ainokainen luodaan luokan Awake-metodissa, joka kutsutaan aina kerran ennen objektien käyttämistä. Näin luokan julkisia metodeja voidaan kutsua suoraan

ainokaisen instanssin kautta, eikä sitä tarvita hallinnoida sen oman luokan ulkopuolelta.

SoundFXManager-luokka on vastuussa pelin äänitehosteiden hallinnasta ja toistamisesta. Luokan Awake-metodissa määritellään ainokainen (kuva 21). Instanssien määrä varmistetaan if-lauseella, jossa tarvittaessa poistetaan peliobjekti, johon luokka on kiinnitetty.

```
public static SoundFXManager instance;
Unity Message | 0 references
private void Awake()
{
    if(instance == null)
    {
        instance = this;
    }
    else
    {
        Destroy(gameObject);
    }
}
```

KUVA 21. SoundFXManager-luokan Awake-metodi

Ainokaisen julkisia metodeja kutsutaan eri peliobjekteista, jotka voivat tuottaa äänitehosteita. Esimerkiksi Player-luokka kutsuu ainokaisen PlaySoundFXClip-metodia, kun pelaajahahmon lyönti osuu eri kohteisiin. Kuvassa parametreina annetaan oikea äänileike, sijainti tiedot ja äänenvoimakkuuden arvo (kuva 22).

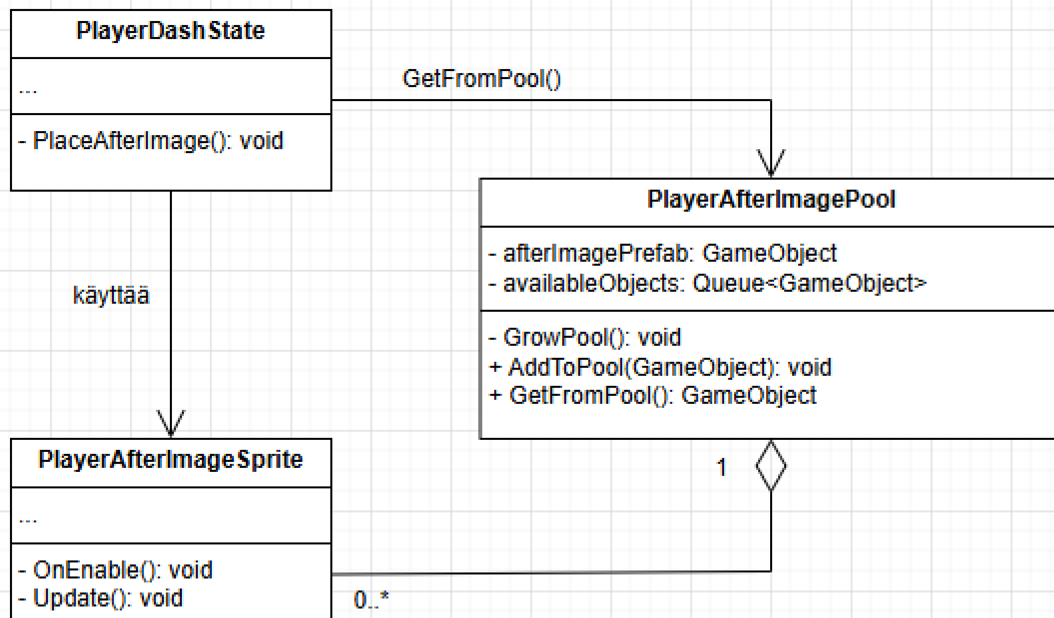
```
SoundFXManager.instance.PlaySoundFXClip(playerData.playerAttackSounds[0], transform, 0.7f);
```

Kuva 22. Ainokaisen julkisen metodin kutsu Player-luokassa

Ainokainen-mallia käytetään myös hyväksi pelin kameraa hallitsevissa luokissa. CameraManager-luokka vastaa kameran sujuvasta pelaajahahmon seuraamisesta, ja muista tarvittavista kameran liikkeistä. CameraShakeManager-luokka toteuttaa kameran pienet heilautukset eri taistelu tapahtumien aikana. Pelin aikana uuteen kenttään edetessä ainokainen-luokka SwitchScenesManager tekee tarvittavat toimenpiteet.

5.4 Oliovarasto-mallin esiintyminen pelissä

Pelihahmon syöksyessä jätetään hahmon taakse jälkikuvia tehostamaan efektiä. Jälkikuvien toteutukseen tehtiin oliovarasto-mallin mukainen PlayerAfterImagePool-luokka (kuva 23). Luokan yksityisenä kenttänä on availableObjects-jono, jota käytetään afterImagePrefab-peliobjektien säilyttämiseen. GrowPool-metodi vastaa olioiden luomisesta, kun varasto on tyhjä. Oliot deaktivoidaan ja asetetaan availableObjects-jonoon kutsumalla AddToPool-metodia. Olioita haetaan käyttöön GetFromPool-metodilla, joka myös ensin tarkistaa onko olioita käytettävissä jonossa. Tarvittaessa se kutsuu GrowPool-metodia luomaan lisää olioita. Saatavilla olevat peliobjektit asetetaan aktiivisiksi pelaajahahmon käyttöä varten. PlayerAfterImageSprite-luokka on asetettu varastoituihin peliobjekteihin. Kun peliobjekti otetaan oliovarastosta ja asetetaan aktiiviseksi, alkaa luokan Update-metodissa ajastin. Ajastimen kuluessa loppuun peliobjekti palautetaan takaisin varastoon kutsumalla AddToPool-metodia.



KUVA 23. Jälkikuvien kierrättäminen pelin aikana

5.5 Tarkkailija-mallin esiintyminen pelissä

Unityllä työskennellessä ei tarvita luoda omaa tarkkailija-mallin toteutusta. Malli on niin yleinen, että se on sisäänrakennettu C# kieleen käyttäen event-

rakennetta. Event-rakenne toimii samalla periaatteella kuin tarkkailija malli, mutta sen käytännön toteutus on vähän erilainen. Rajapinnan sijasta event-rakenne käyttää delegaatteja. (Lin 2021, 80–81.)

Delegaatti on tyyppi, jolla pidetään viittaus metodiin ja määritetään sen palautustyypin ja parametrien tyypit. Delegaatti voi viitata vain niihin metodeihin, jotka vastaavat määritettyä signatuuria. Delegaatti toimii tapahtuman lähteen ja tapahtuman saavuttajan välisenä siltana. Yleinen event-rakenteen delegaatti on EventHandler, jolla ei ole palautusarvoa ja ottaa parametrina kaksi objektia. Ensimmäinen objekti edustaa tapahtuman lähdettä, ja toinen sisältää tapahtuman tiedot. (Microsoft 2022.)

EventHandler-delegaatti-luokkaa käytettiin hallitsemaan pelaajahahmon kuolemaan liittyvät tapahtumat. PlayerHealth-luokka toimii subjektina ja GameManager-luokka tarkkailijana (kuva 24). Tapahtuma OnPlayerDeath laukaistaan, kun pelaajahahmo kuolee. GameManager-luokka on rekisteröity tarkkailemaan tapahtumaa Start-metodissa. DeathListener-metodissa reagoidaan tapahtumaan ja tehdään tarvittavat toimenpiteet.

PlayerHealth-luokassa.

```
public event EventHandler<PlayerDeathEventArgs> OnPlayerDeath;
```

Elämänpisteiden loppuessa.

```
OnPlayerDeath?.Invoke(this, new PlayerDeathEventArgs { deathPosition = this.transform.position});
```

GameManager-luokassa.

```
playerHealth.OnPlayerDeath += DeathListener;
```

```
1 reference
private void DeathListener(object sender, PlayerHealth.PlayerDeathEventArgs e)
{
    StartCoroutine(FadeCoroutine());
    StartCoroutine(DeathCoroutine(e.deathPosition));
}
```

KUVA 24. Subjektin tapahtuma ja tarkkailijan reagointi

6 JOHTOPÄÄTÖKSET

Pelin kehityksen aloittaessa en ollut syvemmin tutustunut SOLID-periaatteisiin tai suunnittelumalleihin. Olin tietoinen yleisistä ohjelmistokehityksen hyvistä käytännöistä. Kehityksen aikana toteutuneet periaatteet ja suunnittelumallit vaikuttivat positiivisesti pelin koodirakenteeseen ja toiminnallisuuksiin. Syvemmän teoriakatsauksen jälkeen huomasin toteutuksissa kohtia, joihin periaatteita ja suunnittelumalleja voitaisiin käyttää tai soveltaa paremmin.

SOLID-periaatteet

Yhden vastuun periaatteen toteutuminen pelissä toi selkeyttä projektin rakenteeseen, jolloin sen parissa oli helpompi työskennellä. Periaatetta toteuttavat oman toiminnallisuuden mukaan nimetyt luokat ja metodit oli nopea tunnistaa. Näistä luokista ja metodeista sai yhdellä katsauksella käsityksen, mistä toiminnallisuudesta ne ovat vastuussa.

Tärkeässä osassa oleva Player-luokka rikkoo yhden vastuun periaatetta. Luokka on vastuussa pelaajahahmon tilojen alustamisesta, Rigidbody2D-komponentin liikuttamisesta, maan ja seinien tarkastuksesta ja pelaajahahmon äänien toistamisesta. Näille toiminnoille voitaisiin luoda omat luokat, jolloin koodista tulisi selkeämpää.

Avoin-suljettu periaatteen mukaiset toiminnot vaikuttivat positiivisesti kehityksen nopeuteen. Onnistuneet toteutukset eivät vaatineet aikaisemman koodin muokkaamista projektin laajentuessa, mikä teki uusien toimintojen virheenkorjauksesta selkeämpää.

Liskovin korvausperiaatteen hyödyt näkyivät rajapintoja toteuttavissa peliobjekteissa, joiden kautta pelaajahahmo oli niiden kanssa vuorovaikutuksessa. Koska rajapinnat määrittelivät tarvittavat metodit, peliobjektien toiminnallisuudet saatiin jaettua ryhmiin. Esimerkiksi IDamageable-rajapinnan toteuttavat tynnyri ja arkku olivat rikottavia esineitä, jolloin ne voitiin pelaajahahmon näkökulmasta käsitellä samalla tavalla. Uusia rikottavia esineitä pystyttiin täten lisäämään nopeasti ilman aikaisemman koodin muutoksia.

Rajapinnan erotteluperiaatetta toteuttavat rajapinnat toivat selkeyttä itse rajapintoihin. Pienet rajapinnat mahdollistivat Liskovin korvausperiaatteen toteutumisen, sillä rajapinnat ja niitä toteuttavat peliobjektit voitiin jakaa omiin kategorioihin.

Riippuvuuden käänteisyyden periaatteen toteutukset edistivät koodin laajennettavuutta ja uudelleenkäyttöä. Pelaajahahmon vuorovaikutukset muiden peliobjektien kanssa toteutuvat rajapintojen kautta, joten uusia vuorovaikutteisia peliobjekteja voitiin lisätä helposti. Itse pelaajahahmon koodia ei tarvinnut muuttaa ollenkaan, kun tehtiin uusia päihitettäviä vihollisia tai lentäviä ammuksia.

Tila-malli ja tilakone

Pelaajahahmoa kontrolloiva tilakone nopeutti toiminnallisuuksien kehitystä. Tilakoneen tuoma suurin hyöty oli pelaajahahmon tilojen selkeä erottelu ja vaihtaminen. Erotetuissa tiloissa pelaajahahmoa pystyttiin hallinnoimaan täsmällisesti tilan tarpeiden mukaan. Tiloihin voitiin määrittää omat halutut arvot, joita käytettiin toimintojen hallintaan. Esimerkiksi pelaajaobjektin Rigidbody2D-komponentin nopeusarvoja säädettiin tiloissa vastaamaan toiminnon vaatimuksia. Pelaajan painaessa hyppäämispainiketta hyppäämistila asettaa Rigidbody2D-komponentin y-arvoa suuremmaksi. Kun päästetään hyppäämispainikkeesta irti ilmatilassa, säädetään y-arvoa hieman negatiiviseksi, jotta painovoiman vaikutus vahvistuisi. Tilakoneen avulla saatiin pelaajahahmo reagoimaan välittömästi ympäristöönsä. Tilaa voitiin vaihtaa yhdellä funktiokutsulla ilman muita tarkistuksia, mikä teki pelaajahahmon tilojen vaihtamisesta nopeaa ja yksinkertaista. Tila-mallin toteutuminen teki myös uusien toimintojen kehityksestä suoraviivaisempaa. Yläluokat tarjosivat pohjan uusille tilaluokille, jolloin toistuvaa koodia ei tarvinnut kirjoittaa. Voitiin keskittyä vain olennaisiin tehtäviin, kuten peliobjektien hallinnoimiseen tai tilojen vaihtamisen logiikkaan.

Pelaajahahmon animaatioiden toistaminen Unityn Animator-komponentilla hyötyi tilakoneen toteutuksesta. Kaikki animaatiot voitiin suorittaa suoraan koodissa, jolloin Animator-komponentin animaatiosiirtymiä käytettiin vain Entry- ja Exit-tiloihin (liite 1). Ilman tilakonetta jokainen mahdollinen animaatiotilan vaihto olisi pitänyt yhdistellä toisiinsa Animator-komponentin käyttöliittymässä. Pelin

animaatiotilojen suuri määrä olisi johtanut sekavaan näkymään ja hankalasti hallittavaan logiikkaan.

Ainokainen-malli

Ainokainen-mallin käyttö vaikutti positiivisesti kehityksen nopeuteen. Manager-tyyppisiä ainokaisluokkia voitiin kutsua ilman viittauksia koko projektin laajuisesti. Toteutus varmisti, että luokista on olemassa vain yksi instanssi olemassa pelin elinkaaren aikana. Äänientoistossa ja kameran hallitsemisessa tämä varmistaa, että halutut toiminnot toistuvat vain kerran.

Ainokainen-mallin toteutuksessa on myös negatiivisia puolia ja mahdollisia riskitilanteita. Tällä hetkellä peliobjektit ovat suoraan kytkettynä Manager-tyyppisiin luokkiin. Esimerkiksi SoundFXManager-luokkaan on suoraan kytkettynä monia peliobjekteja, mikä voi johtaa ongelmiin, jos pelin äänentoistojärjestelmää halutaan tulevaisuudessa muokata tai laajentaa. Äänentoiston mahdollisten muutoksien jälkeen pitäisi päivittää jokainen siitä suoraan riippuva luokka. Testien tekeminen äänijärjestelmään on hankalaa, koska se on suoraan kytketty peliobjekteihin. Ainokaisen instanssin metodia kutsutaan suoraan koodista, jolloin sille ei voida välittää äänijärjestelmän testiversiota.

Äänentoistojärjestelmä voitaisiin vaihtaa käyttämään rajapintoja ja riippuvuuden injektointia, joka ratkaisisi ainokainen-mallin tuomat ongelmat. Äänentoistoluokkaa voitaisiin muokata ja laajentaa haluttuun tapaan, kun peliobjektit riippuvat rajapinnasta eivätkä suoraan SoundFXManager-luokasta. Testitapauksiin olisi mahdollista luoda MockFXManager-luokka, jolla testataan äänentoistojärjestelmää.

Oliovarasto-malli

Oliovarasto-mallin käyttö mahdollisti sujuvan pelaajahahmon syöksytoiminnon jälkikuvien näyttämisen. Toteutus oli hyödyllinen pelin suorituskyvyn kannalta, koska sillä vältettiin turhien peliobjektien luontia ja poistoa. Jälkikuvien luominen ja poistaminen manuaalisesti ei itsessään veisi paljon resursseja, mutta syöksy toimintoa voidaan toistaa todella useasti pelin suoritusaikana, jolloin pelin suorituskyky voi heikentyä. Oliovarasto-malli teki myös jälkikuvista enemmän

ennustettavan. Kun tarvittavat peliobjektit luotiin Unityn Awake-metodissa, jälkikuvien toteutuksesta muodostui itsenäinen komponentti, jonka toiminnasta ei tarvinnut huolehtia pelin suoritusajan aikana.

Pelistä löytyy muita toimintoja, joissa voitaisiin käyttää oliovarasto-mallia suorituskyvyn parantamiseksi. Pelissä kerättävät kolikot luodaan, kun pelaaja avaa arkun tai päihittää vihollisen. Kun pelaajahahmo koskettaa kolikkoa, se poistetaan. Jos pelissä päihitetään useita vihollisia nopealla aikavälillä, joutuu pelimoottori luomaan ja tuhoamaan monia kolikoita, mikä voi heikentää suorituskykyä. Oliovarasto-malli voitaisiin toteuttaa samalla tavalla kuin jälkikuvien toteutuksessa, mutta kierrätettävien peliobjektien yläraja pitäisi olla korkeampi, sillä kolikoita voidaan tarvita useampia pelin suoritusaikana.

Tarkkailija-malli

Pelin tarkkailija-mallin mukainen toteutus vähensi kytköksiä pelaajahahmon ja pelinhallintaluokan välillä. Event-rakenteen käyttö voitaisiin nähdä tällä hetkellä jopa turhana, sillä toteutuksessa on vain yksi tarkkailija. Se kuitenkin tekee toteutuksesta avoimen laajennuksille, jotka olivat jo suunnitteilla kehitysvaiheessa. Tarkkailijoiksi oli aikomuksena myös asettaa muun muassa pelin viholliset, jotka pelaajahahmon kuollessa lopettaisivat sen takaa-ajon.

Jatkokehityksen kannalta tarkkailija-mallin toteutukseen voitaisiin tehdä arkkitehtuurisia muutoksia, jotka parantaisivat projektin laajennettavuutta entisestään. Pelaajahahmon kuoleman tapahtuman subjektiksi voitaisiin luoda kokonaan uusi luokka, jota pelaajahahmo kutsuu rajapinnan kautta kuollessaan. Uudessa luokassa tehtäisiin vikoja estäviä tarkistuksia ennen tapahtuman laukaisua. Esimerkiksi tarkistetaan, onko peli keskeytetty, jolloin odotetaan pelin jatkumista ennen tapahtuman laukaisua.

7 POHDINTA

Opinnäytetyön tavoitteena oli perehtyä SOLID-periaatteisiin ja suunnittelumalleihin sekä vertailla, miten ne ovat toteutuneet Unityllä kehitetyssä 2D-toimintapelissä. Työn tähtäimenä oli paremman tietotaidon pohjalta tunnistaa hyvin onnistuneet toteutukset, joita voidaan jatkossakin käyttää. Toteutuksista mietittiin myös periaatteiden ja suunnittelumallien näkökulmasta kehityskohtia, jotka parantaisivat pelin koodikantaa.

Työn tuloksena saatiin parempi käsitys pelin toiminnoista. Syvemmän teoriatutkimuksen ja vertailun jälkeen saatiin selville, miten SOLID-periaatteet ja suunnittelumallit vaikuttavat tai voisivat vaikuttaa käytännössä pelin kehitykseen. SOLID-periaatteiden toteutuminen teki koodin rakenteesta selkeää ja skaalautuvaa, mikä nopeutti kehitysprosessia. Suunnittelumallien toteutuminen oli mahdollistanut pelin monimutkaisempien toimintojen onnistumisen. Mallien toteutuminen ei aina vastannut teoriassa täysin suunnittelumalliaan, mikä voidaan nähdä positiivisena asiana. Tällöin vertailussa huomattiin, mitä suunnittelumalleilla pyritään ratkaisemaan käytännön kannalta. Kehitetyn pelin jatkokehityksen kannalta saatiin johtopäätöksissä eriteltyä periaatteiden ja suunnittelumallien tuomat hyödyt. Tutkittujen hyötyjen pohjalta voitiin pohtia, mitä muita pelin komponentteja voitaisiin parantaa käyttämällä periaatteita tai suunnittelumalleja.

Työn teoriapohjaksi löytyi hyvin materiaalia artikkeleista ja E-kirjoista. Tietoa pyrittiin keräämään mahdollisimman monipuolisesti eri lähteistä. Opinnäytetyössä käsiteltiin kaikki SOLID-periaatteet, mutta rajattiin suunnittelumallit vain niihin, jotka nähtiin kehitetyn pelin kannalta oleellisiksi. Suunnittelumallien valinta onnistui mielestäni hyvin, sillä kehitetystä pelistä löytyi niitä vastaavia toteutuksia, jotka olivat pelin kokonaiskuvassa tärkeitä. Rajattujen mallien sopiva määrä mahdollisti laajan kokonaisuuden muodostamiseen, jota pystyttiin tutkimaan perusteellisesti.

Kehittyneen tietotaidon pohjalta pystyn jatkossa tunnistamaa, missä kohdin SOLID-periaatteita tai suunnittelumalleja kannattaisi käyttää hyödykseni jatkokehityksessä. Erityisen hyödyllisenä pidän syvempää ymmärrystä käsitellyistä suunnittelumalleista, sillä tämä mahdollistaa niiden soveltamisen muihin toimintoihin.

Jatkokehitystä nopeuttaa johtopäätöksistä saadut tulokset, jotka pystyn toteuttamaan heti. Tiedostan myös, että SOLID-periaatteet ja suunnittelumallit eivät ole ehdottomia sääntöjä, vaan suuntaviivoja. Erityisesti pelinkehityksessä kehityksen nopeus ja joustavuus ovat usein etusijalla, eikä esimerkiksi yhden vastuun periaate aina sovi kaikkiin tilanteisiin. Vaikka opinnäytetyön aiheena oli tutkia periaatteita ja malleja pelinkehityksen kannalta, voi käsiteltyä aihetta hyödyntää mihin tahansa ohjelmistokehitykseen.

LÄHTEET

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. 2009. Design Patterns. Elements of Reusable Object-Oriented Software. 37. uudistettu painos. Addison-Wesley. Westford. E-kirja. Luettu: 03.11.2024.

GeeksForGeeks. 2024. Singleton Method Design Pattern. Luettavissa: https://www.geeksforgeeks.org/singleton-design-pattern/. Luettu: 04.11.2024.

GeeksForGeeks. 2023. C# | Abstract Classes. Luettavissa: https://www.geeksforgeeks.org/c-sharp-abstract-classes/. Luettu: 4.10.2024.

Ghergu, E. 2023. Object Pool Design Pattern – Sharing is Caring. PentaBlog. Luettavissa: https://www.pentalog.com/blog/design-patterns/object-pool-design-pattern/. Luettu: 04.11.2024.

JavatPoint. s.a. C# Interface. Luettavissa: https://www.javatpoint.com/c-sharp-interface. Luettu: 5.10.2024.

Karropoulos, T. 2023. The Power Of Dependecy Inversion Principle (DIP) in Software Development. DEV. Luettavissa: https://dev.to/tkarropoulos/the-power-of-dependency-inversion-principle-dip-in-software-development-4klk. Luettu: 29.10.2024.

Lin, W. 2021. Level up your code with game programming patterns. Unity Technologies. E-kirja. Luettu: 21.09.2024.

Microsoft. 2022. Handle and raise events. Luettavissa: https://learn.microsoft.com/en-us/dotnet/standard/events/. Luettu: 11.12.2024.

Nutt, C. 2015. The Undying Allure of The Metroidvania. Game Developer. Luettavissa: https://www.gamedeveloper.com/design/the-undying-allure-of-the-metroidvania. Luettu 19.11.2024.

Nystrom, R. 2014. Game Programming Patterns. Luettavissa: https://gameprogrammingpatterns.com/. Luettu: 26.09.2024.

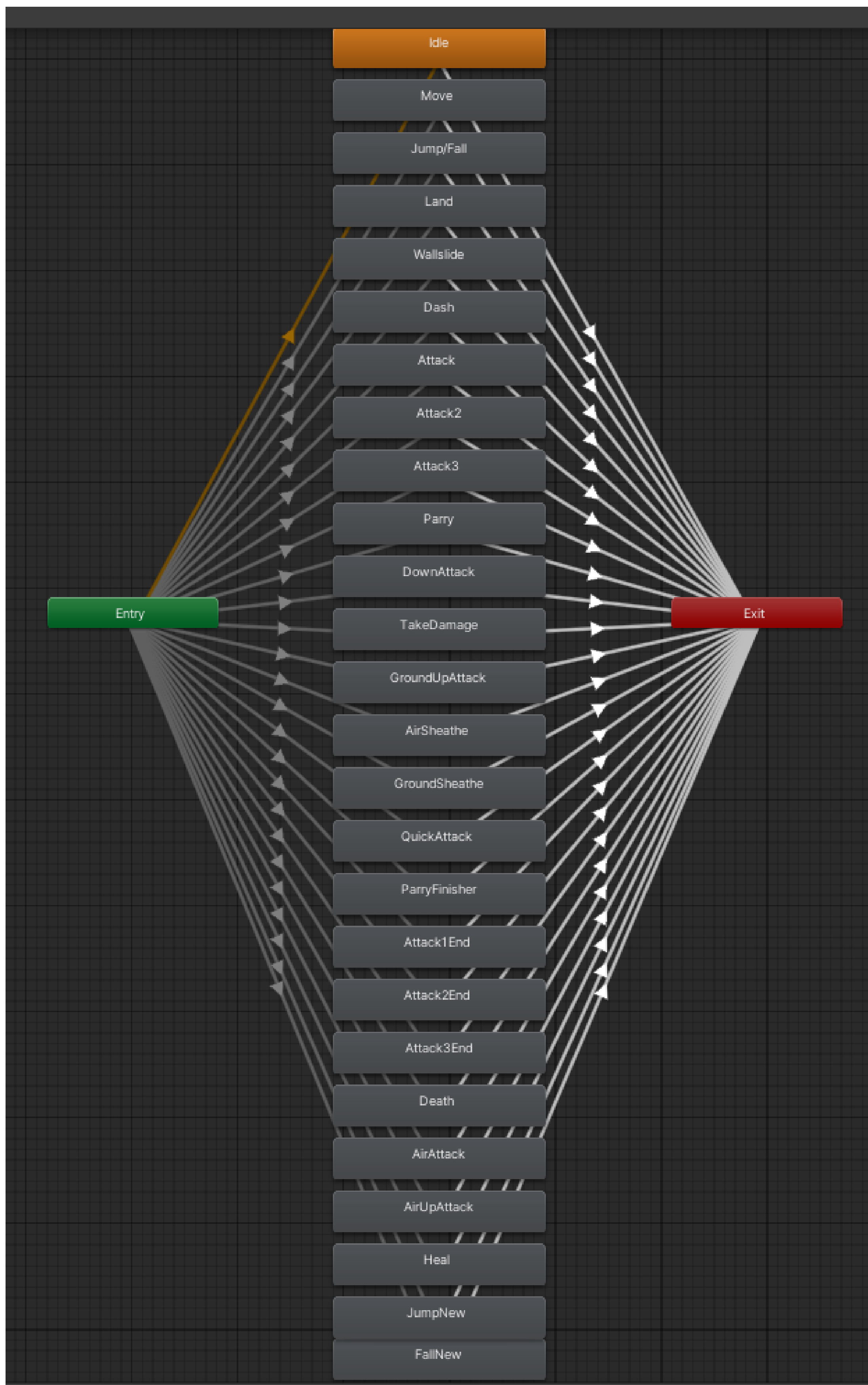
OODesign.com. s.a. Design Patterns, Luettavissa: https://www.oodesign.com/. Luettu: 15.10.2024.

Opsive. s.a. What is a Behavior Tree. Luettavissa: https://opsive.com/support/documentation/behavior-designer/what-is-a-behavior-tree/. Luettu 21.11.2024.

Shvets, A. 2023. Dive Into Design Patterns. E-kirja. Luettu: 01.11.2024.

Wikipedia. 2024. Valokuva. Observer pattern. Luettavissa: https://en.wikipedia.org/wiki/Observer_pattern. Luettu: 11.12.2024.

Yiğit, K. E. 2020. The SOLID Principles of Object-Oriented Programming Explained in Plain English. freeCodeCamp. Luettavissa: https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/. Luettu: 20.09.2024.


Idle
Move
Jump/Fall
Land
Wallslide
Dash
Attack
Attack2
Attack3
Parry
DownAttack
Entry
TakeDamage
Exit
GroundUpAttack
AirSheathe
GroundSheathe
QuickAttack
ParryFinisher
Attack1End
Attack2End
Attack3End
Death
AirAttack
AirUpAttack
Heal
JumpNew
FallNew