Janita Korhonen

Koodin laadun parantaminen pelikehityksessä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

18.4.2024

Tiivistelmä

Tekijä:	Janita Korhonen
Otsikko:	Koodin laadun parantaminen pelikehityksessä
Sivumäärä:	52 sivua
Aika:	18.4.2024
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaajat:	Lehtori Antti Laiho

Insinöörityössä oli tarkoituksena tutustua erilaisiin ohjelmointikäytäntöihin, suunnitteluperiaatteisiin ja suunnittelumalleihin pelikehityksessä sekä niiden soveltamiseen käytännössä projektin avulla. Projektin tekemiseen käytettiin Unity-pelimoottoria.

Projektissa suunnittelumalleista singleton-malli ja objektivarasto osoittautuivat hyödyllisiksi resurssien hallinnassa ja dynaamisessa objektien luomisessa. Singleton-malli auttoi yhden ilmentymän varmistamisessa ja tarjosi tehokkaan tavan käyttää ja hallita yksittäisiä resursseja. Objektivarasto puolestaan tarjosi joustavan tavan luoda ja hallita objekteja tarpeen mukaan.

Toisaalta suunnittelumalleista tehdasmalli, havaintomalli ja tilamalli eivät olleet tarpeellisia tässä projektissa. Tehdasmallin ja havaintomallin monimutkaisuudet eivät tarjonneet lisäarvoa, ja myös tilamalli oli ylimitoitettu ja monimutkaistettu projektin tarpeisiin nähden.

Insinöörityön lopputuloksena muodostui koodillisesti laadukas peliprojekti, joka hyödynsi ohjelmointikäytäntöjä ja valittuja suunnitteluperiaatteita ja -malleja tehokkaasti. Insinöörityön edetessä opittiin arvioimaan erilaisten suunnitteluperiaatteiden ja -mallien soveltuvuutta eri käyttötarkoituksia varten. Jokaisen suunnitteluperiaatteen ja -mallin etuja ja haittoja on tärkeä arvioida projektin kontekstissa, jotta voidaan tehdä tietoisia ja perusteltuja valintoja ohjelmistosuunnittelussa.

Avainsanat: ohjelmointikäytäntö, suunnitteluperiaate, suunnittelumalli

---

Abstract

Author:	Janita Korhonen
Title:	Improving Code Quality in Game Development
Number of Pages:	52 pages
Date:	18 April 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Game Applications
Supervisors:	Antti Laiho, Senior Lecturer

The purpose of this final year project was to explore various code conventions, design principles, and design patterns in game development and their practical application through a project. The Unity game engine was used for the project.

In the project, the Singleton pattern and Object Pool proved to be useful in resource management and dynamic object creation. The Singleton pattern helped ensure a single instance and provided an efficient way to use and manage individual resources. The Object Pool, on the other hand, offered a flexible way to create and manage objects as needed.

However, the Factory pattern, Observer patter, and State pattern were not necessary for this project. The complexities of the Factory pattern and Observer pattern did not provide additional value, and the State pattern was also overcomplicated considering the project's needs.

The result of the final year project was a high-quality game project that effectively utilized code conventions and selected design principles and patterns. Throughout the project, the suitability of various design principles and patterns for different purposes was evaluated. It is important to evaluate the advantages and disadvantages of each design principle and pattern in the context of the project to make informed and justified decisions in software design.

Keywords: code convention, design principle, design pattern

Sisällys

Lyhenteet

DIP: *Dependency Inversion Principle*. SOLID-periaate, jossa keskitytään riippuvuuksien hallintaan sovelluksen komponenttien välillä.

DRY: *Don't repeat yourself*. Yleinen suunnitteluperiaate, joka kehottaa välttämään koodin toistoa.

ISP: *Interface Segregation Principle*. SOLID-periaate, jossa tärkeänä osana rajapinnat ja niiden alttius rajapintoja käyttäville komponenteille.

KISS: *Keep it simple, stupid*. Yleinen suunnitteluperiaate, joka korostaa yksinkertaisuutta ohjelmien toteutuksessa.

LSP: *Liskov Substitution Principle*. SOLID-periaate, jossa olennaista aliluokat ja periytyminen.

MVP: *Model-View-Presenter*. Suunnittelumalli, jota käytetään etenkin käyttöliittymissä.

OCP: *Open-Closed Principle*. SOLID-periaate, jossa olennaista luokkien suunnittelu ja ominaisuuksien lisääminen.

OOP: *Object-oriented programming*. Olio-ohjelmointi. Ohjelmointiparadigma, joka käyttää objekteja sovelluksien ja tietokoneohjelmien suunnittelussa.

SOLID: Lyhenne viidestä suunnitteluperiaatteesta, joita käytetään usein OOP:n yhteydessä.

SRP: *Single Responsibility Principle*. SOLID-periaate, joka korostaa yksinkertaisia ja pieniä luokkia.

1 Johdanto

Tämän insinöörityön tarkoituksena on tutkia pelikehityksessä käytettäviä keinoja, kuten ohjelmointikäytäntöjä, suunnitteluperiaatteita ja suunnittelumalleja, koodin laadun parantamiseksi, ja näiden keinojen soveltamista Unity-pelimoottorilla toteutettavaan peliprojektiin. Insinöörityössä käsitellään erilaisia ohjelmointikäytäntöjä, suunnitteluperiaatteita ja suunnittelumalleja, joiden avulla voidaan parantaa pelin kehitysprosessia ja lopputuloksen laatua.

Ohjelmointikäytännöt ovat keskeinen osa laadukkaan ja ylläpidettävän koodin tuottamista (1). Luvussa 2 perehdytään erilaisten ohjelmointikäytäntöjen merkityksiin ja hyötyihin pelikehityksessä sekä käsitellään ohjelmointikielen ulkoasua koskevia käytäntöjä, kuten muuttujien nimeämistä, koodin selkeyttämistä ja kommentointia. Luvussa 3 esitellään olio-ohjelmointiin liittyviä keskeisiä suunnitteluperiaatteita (KISS, DRY ja SOLID –periaatteet), jotka auttavat suunnittelemaan joustavaa ja helposti laajennettavaa koodia. Luvun lopuksi vielä pohditaan suunnitteluperiaatteiden toteuttamista abstraktioiden ja rajapintojen näkökulmasta. Neljännessä luvussa käsitellään erilaisia pelikehityksessä käytettäviä suunnittelumalleja: tehdasmallia, objektivarastoa, singleton-mallia, komentomallia, tilamallia, havaintomallia ja MVP-mallia. Luvussa kerrotaan, mitä kyseiset suunnittelumallit ovat, mitkä ovat niiden hyödyt ja haitat, millaisissa tilanteissa niitä käytetään ja miten niitä toteutetaan sekä mikä niiden hyödyllisyys on projektin näkökulmasta.

Keskeisenä osana insinöörityötä on peliprojekti, joka toteutetaan Unity-pelimoottorilla. Peliprojekti toimii käytännön esimerkkinä siitä, miten erilaisia ohjelmointikäytäntöjä, suunnitteluperiaatteita ja -malleja voidaan soveltaa pelin kehityksessä. Projektin avulla esitellään muun muassa oikeaoppista koodin rakennetta, suunnitteluperiaatteiden noudattamista ja suunnittelumallien käyttöä pelin toiminnallisuuksien toteuttamisessa. Peliprojektina on endless runner -peli, jossa on tarkoituksena toteuttaa eri suunnittelumalleja pelin toiminnallisuuksissa

ohjelmointikäytäntöjä noudattaen. Pelin toiminnallisuuksia ovat muun muassa juokseminen, ampuminen, seinähyppely, kiipeäminen ja pelaajan jahtaus.

Insinöörityön lopuksi kootaan käsitellyt asiat ja niiden vaikutus peliprojektiin. Tavoitteena on antaa lukijalle kokonaisvaltainen käsitys siitä, miten ohjelmointikäytännöt vaikuttavat pelin kehitysprosessiin ja koodin laatuun.

Tämän insinöörityön avulla pyritään tarjoamaan hyödyllistä tietoa ja käytännön vinkkejä henkilöille, jotka ovat kiinnostuneita pelien kehittämisestä Unity-ympäristössä ja laadukkaan koodin tuottamisesta peliprojekteissa.

2 Yleiset ohjelmointikäytännöt

Ohjelmointikäytännöt ovat suosituksia ja vakiintuneita ohjeita, joita ohjelmoijat seuraavat koodia kirjoittaessaan (2). Ohjelmointikäytännöt parantavat koodin luettavuutta ja ymmärrettävyyttä ja vähentävät virheiden riskiä. Lisäksi ne tekevät koodin ylläpidosta ja laajentamisesta helpompaa verrattuna siihen, jos ohjelmointikäytäntöjä ei käytettäisi (1.) Erityisesti koodin yksinkertaisuus, kuvaavat nimet, selkeät määritelmät, johdonmukaisuus, siisti muotoilu ja kommentit edistävät koodin ymmärrettävyyttä (3, luku 1.7).

Ohjelmointikäytäntöjen noudattaminen on erityisen tärkeää, kun sovelluksessa on paljon koodia tai kun useat henkilöt osallistuvat koodin kirjoittamiseen. Jos kaikki projektissa mukana olevat jäsenet seuraavat ohjelmointikäytäntöjä, koodi on helposti luettavaa, ja sen ymmärtäminen on helppoa ja yhteistyö projektin jäsenten välillä helpottuu (2).

Hyvän tyylin omaksuminen on keskeistä ohjelmoinnissa. Koodia kirjoitettaessa on suositeltavaa ajatella koodin tyyliä ja käyttää aikaa sen tarkistamiseen ja parantamiseen. Kun koodin parantamisen tavoittelusta tulee automaattista, alitajunta hoitaa monia yksityiskohtia, ja laadukas koodi muodostuu kuin itsestään. Huonosti kirjoitettu koodi ei ole vain vaikeasti luettavaa; se on usein lisäksi virheellistä ja sen ymmärtäminen vie aikaa. (3, luku 1.7.)

Ohjelmointityylin periaatteet perustuvat kokemuksen ohjaamaan maalaisjärkeen, eivätkä ne ole kiveen hakattuja tai mielivaltaisia sääntöjä (3, luku 1.7). Ohjeistukset kehittyvät ajan myötä, ja jokaisella ohjelmointikielellä on omat tyylisääntönsä (1). Tässä insinöörityössä keskitytään C#:n ohjelmointikäytäntöihin ja tyylisääntöihin.

2.1 Muuttujien nimeäminen

Nimeämiskäytännöt tekevät koodista ymmärrettävää projektin jäsenille. Muuttujan tai funktion nimen tulisi kertoa sen tarkoituksesta, ja lisäksi nimen pitää olla ytimekäs, helposti muistettava ja lausuttava. Samaan asiaan liittyville asioille pitäisi antaa samankaltaiset nimet, jotka näyttävät asioiden suhteen ja korostavat niiden eroa. On olemassa erilaisia nimeämiskäytäntöjä, mutta on osittain mielipideasia, mikä käytäntö on paras. Tärkeintä on olla johdonmukainen ja pysyä tietyssä tyylissä. Mitä suurempi ohjelma on, sitä tärkeämpää on valita hyviä, kuvaavia ja johdonmukaisia nimiä. (3, luku 1.1.)

Muuttujan näkyvyysalue vaikuttaa muuttujan nimeämiseen. Mitä laajempi näkyvyys muuttujalla on, sitä enemmän tietoa sen nimestä tulisi välittyä. Globaaleilla muuttujilla, funktiolla, luokilla ja rakenteilla tulisi olla kuvaavat nimet, jotka kertovat niiden roolista ohjelmistossa, ja on suositeltavaa lisätä lyhyt kommentti niiden julistuksiin. Lyhyet nimet riittävät paikallisille muuttujille. Esimerkkikoodissa 1 näytetään, miten muuttuja on nimetty Player-luokassa. Jos kyseessä on paikallinen muuttuja, muuttujan nimen ei tarvitse olla ”PlayerScore”, vaan ”Score” riittää kertomaan muuttujan tarkoituksesta. (3, luku 1.1.)

```
public class Player : MonoBehavior
{
     private int _score;
     public int Score { get => _score; set => _score = value; }

     private bool _isDead;
     public bool IsDead => _isDead;
}
```

Esimerkkikoodi 1. Yksityisen ja julkisen muuttujan nimeäminen.

PascalCase ja camelCase ovat C#:ssa yleisesti käytettyjä nimikäytäntöjä (4). PascalCase on keino kirjoittaa ilman välilyöntejä fraaseja, joissa ensimmäinen kirjain jokaisesta sanasta on isolla kirjaimella. CamelCase on samanlainen kuin PascalCase, paitsi ensimmäinen kirjain koko fraasista on pienellä kirjaimella. (5.)

Microsoft on tehnyt omat suosituksensa C#:n nimeämiskäytäntöihin. Niissä kerrotaan, että luokkien ja metodien nimet, vakiot ja kentät pitäisi kirjoittaa PascalCase-tyylillä. Lisäksi rajapintojen nimet alkavat isolla I:llä. Yksityiset kentät alkavat alaviivalla ja metodien parametrit ja lokaalit muuttujat kirjoitetaan camelCase-tyylillä. Esimerkkikoodi 1 havainnollistaa myös oikeanlaista PascalCasen ja camelCasen käyttöä yksityisen ja julkisen muuttujan nimessä. Lisäksi suositellaan käyttämään tarkoituksellisia ja kuvaavia nimiä ja välttämään yhden kirjaimen nimiä paitsi yksinkertaisissa silmukoissa. (4.)

Funktiot suorittavat toimintaa, joten niiden nimen pitäisi alkaa verbillä (6, luku 2). Boolean-arvon palauttavat funktiot tulisi nimetä niin, että ei ole epäselvää, mitä ne palauttavat (3, luku 1.1). Esimerkiksi ”tarkistaSaatavuus(esine)” on huono nimi funktiolla, koska on epäselvää minkä arvon se palauttaa. Jos sen sijaan käytetään nimenä ”onkoSaatavilla(esine)”, se kertoo heti, että funktiosta palautuu tosi, jos esine on saatavilla, ja epätosi, jos esinettä ei ole saatavilla. Esimerkkikoodissa 1 on myös esimerkki boolean-arvon nimeämisestä.

2.2 Määritelmät ja lausekkeet

Nimeämisellä ohjataan lukijan ymmärrystä. Määritelmät ja lausekkeet puolestaan tekevät nimeämisen tarkoituksesta mahdollisimman selkeän. Hyvällä, johdonmukaisella muotoilulla on positiivinen vaikutus koodin ymmärtämiseen (3, luku 1.2).

Välilyönnit ja sisennykset vaikuttavat koodin luettavuuteen. Vaikka ne saattavat vaikuttaa vähäpätöisiltä, ne ovat kuitenkin arvokkaita yksityiskohtia etsittävien asioiden löytämisessä ja rakenteen näyttämisessä. Välilyönnit operaattorien

välissä auttavat erottamaan ryhmittymiä, ja välilyönneillä voidaan vähentää myös koodin tiheyttä. Lisäksi välilyönnit auttavat erottamaan parametrit toisistaan funktioissa. (3, luku 1.2.) Pystyvälillä voidaan erotella muuttujia toisistaan ja yhdistää tiettyyn asiaan liittyvät muuttujat toisiinsa. On lisäksi suositeltavaa laittaa vain yksi muuttuja jokaista riviä kohti luettavuuden lisäämiseksi (6, s. 15). Säännöllisyys sisennyksissä on keino tehdä ohjelman rakenne itsestään selväksi. Ylimääräinen tyhjä tila antaa visuaalista erottuvuutta rivien eri osiin, mutta myös tyhjän tilan poistaminen voi auttaa lukijaa hahmottamaan koodin rakennetta (3, luku 1.2).

Ohjelmoinnissa on erilaisia tyylejä käyttää aaltosulkuja. Yksi yleisimmistä tyyleistä on Allman-tyyli, jossa aaltosulut lisätään aina uudelle riville seuraavan lausekelohkon jälkeen. Se tarkoittaa sitä, että aaltosulut alkavat omalta riviltään ja ovat visuaalisesti erillään muusta koodista. Toisessa suositussa tyylissä, K&R-tyylissä, aaltosulut alkavat samalta riviltä kuin edellinen lauseke. (6, s. 25.) Esimerkkikoodissa 2 havainnollistetaan ensin Allman-tyyliä ja sen jälkeen K&R-tyyliä.

```
if(Allmanstyle)
{
     Function();
}
else
{
     KRStyle = true;
}

if(KRStyle){

     Function2();
}
else{

     Allmanstyle = true;
}
```

Esimerkkikoodi 2. Allman- ja K&R-tyylit aaltosulkujen käytössä.

On lisäksi muita tyylejä, joissa aaltosulkujen käyttö vaihtelee. Joissain tyyleissä aaltosulkuja ei käytetä lainkaan, jos aaltosulut voidaan jättää pois ilman, että koodin rakenne kärsii. Kuten monissa ohjelmointikäytännöissä, myös

aaltosulkujen käytössä ei ole ehdotonta oikeaa tapaa, kunhan kaikki projektin jäsenet noudattavat samaa tyyliä. (6, s. 25.)

Microsoftin ohjeiden mukaan kuitenkin suositellaan käyttämään aaltosulkuja aina jokaisen lausekelohkon jälkeen ja sijoittamaan aaltosulut seuraavalle riville (Allman-tyyli). Allman-tyyli edistää koodin luettavuutta ja ylläpidettävyyttä, sillä lausekelohkon alku ja loppu ovat selkeästi erillään muusta koodista. (7.)

Sulut auttavat ratkaisemaan monitulkinnallisuutta, osoittavat ryhmittymiä ja voivat selventää tarkoitusta myös silloin, kun niitä ei välttämättä tarvita, mutta niistä ei ole haittaa. Esimerkkikoodissa 3 ensimmäinen rivi osoittaa, miten looginen lauseke on rakennettu ilman ylimääräisiä sulkuja, ja toinen rivi näyttää, miten sulkeita voidaan käyttää selkeyttämään ryhmittymiä ja helpottamaan lausekkeen tulkintaa (3, luku 1.2).

```
leapYear = y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
leapYear = ((y%4==0) && (y%100 != 0)) || (y%400 == 0);
```

Esimerkkikoodi 3. LeapYear on boolean-arvo, joka palauttaa tosi- tai epätosiarvot riippuen siitä, onko y-muuttuja karkausvuosi.

Lukemisen helpottamiseksi kannattaa pitkät lausekkeet rivittää lyhyiksi (6, s. 30). Koodia kirjoittaessa on tärkeää olla selkeä, eikä lyhyt versio aina ole parempi kuin pitkä versio, sillä lyhyttä versiota voi olla vaikea ymmärtää. Pidemmät lausekkeet ovat usein selkeämpiä kuin lyhyet. If-lauseissa ja for-silmukoissa määritelmien luonnollinen muoto tekee koodin lukemisesta sujuvaa. (3, luku 1.2.)

On suositeltavaa käyttää lausekepohjaisia ominaisuuksia yksirivisille lukuominaisuuksille silloin, kun tarvitsee palauttaa vain ominaisuuden arvo (6, s. 22). Esimerkiksi ominaisuuden MaxHealth määrittelyssä on uusi tapa käyttää lausekepohjaista syntaksia (”=>”), joka havainnollistetaan esimerkkikoodissa 4.

```
public int MaxHealth => _maxHealth;
```

Esimerkkikoodi 4. MaxHealth -nimisen integer-arvon määrittely lausekepohjaisella ominaisuudella.

MaxHealth-ominaisuus palauttaa arvon _maxHealth-taustakentästä. Tämä syntaksi on yksirivinen ja helppolukuinen. Kun täytyy palauttaa muutakin kuin vain ominaisuuden arvo, tulisi käyttää vanhaa tapaa, perinteistä syntaksia ({ get; private set; }) (6, s. 22), josta on esimerkki esimerkkikoodissa 5.

```
public int MaxHealth { get; private set ;}
```

Esimerkkikoodi 5. MaxHealth -nimisen interger-arvon määrittely perinteisellä syntaksilla.

Perinteinen syntaksi luo automaattisesti taustakentän ja tarjoaa julkisen (get) ja yksityisen (private set) käyttöoikeuden MaxHealth-ominaisuudelle.

Lausekepohjainen syntaksi on lyhyempi ja yksinkertaisempi tapa määritellä yksinkertaisia ominaisuuksia kuin perinteinen syntaksi, kun taas perinteinen syntaksi tarjoaa enemmän joustavuutta ja toiminnallisuutta kuin lausekepohjainen syntaksi, kuten mahdollisuuden määrittää käyttöoikeudet (get ja set). On suositeltavaa, että ohjelmoija käyttää jompaakumpaa syntaksia riippuen siitä, onko tarvetta päästä set-ominaisuuteen käsiksi.

2.3 Yhdenmukaisuus ja idiomit

Yhdenmukaisuus ja idiomit ohjelmoinnissa ovat tärkeitä tekijöitä, jotka vaikuttavat ohjelman selkeyteen ja ymmärrettävyyteen. Kun ohjelmointityyli vaihtelee odottamattomasti tai käytetään erilaisia käytäntöjä saman tyyppisissä rakenteissa, koodin lukeminen ja ymmärtäminen vaikeutuvat. Esimerkiksi jos silmukat on toteutettu eri tavoin eri osissa ohjelmaa tai käytetään erilaisia kopioimismenetelmiä eri tilanteissa, se voi hämmentää koodin lukijaa ja tehdä koodin tarkoituksen epäselväksi. (3, luku 1.3.)

Yhdenmukainen sisennys ja sulkutyylit auttavat hahmottamaan ohjelman rakennetta. Sisennykset osoittavat loogisia lohkoja ja niiden sisäkkäisyyttä, kun taas sulkujen käyttö auttaa selventämään, mitkä osat kuuluvat yhteen lohkoon. (3, luku 1.3.)

Sulkutyylin valinta voi vaihdella eri ohjelmointikäytäntöjen ja kielen mukaan. Tärkeintä on kuitenkin pysyä samassa tyylissä koko ohjelmassa, jotta koodi on yhdenmukaista ja helposti luettavaa. (6, s. 25.)

Käyttämällä yhdenmukaista tyyliä ja idiomien noudattamista, koodissa varmistetaan, että koodin rakenne on johdonmukainen ja helppo ymmärtää. Esimerkiksi monivalintapäätöksissä suositellaan käytettäväksi else if -rakennetta ja asettamaan else-lauseet samalle riville kuin ylin if-lause, jotta koodi säilyy selkeänä ja helposti luettavana. Sarja if-lauseita peräkkäin voi viitata huonoon koodirakenteeseen, joten on kannattavaa harkita vaihtoehtoisia tapoja tehdä ohjelma selkeäksi ja ylläpidettäväksi. (3, luku 1.3.)

2.4 Numeroiden nimeäminen

Numeroiden nimeäminen on tärkeää ohjelmoinnissa, koska numerot voivat esiintyä monissa eri yhteyksissä koodissa, kuten vakioina, taulukoiden kokoina, merkkien sijainteina ja muuntokertoimina. Kun nimetään numeroita, kannattaa pitää mielessä seuraavat asiat (3, luku 1.5.):

- Kuvaava nimi: Jos numero ei ole vain 0 tai 1, sille tulisi antaa kuvaava nimi, joka kertoo sen tarkoituksesta ja auttaa ymmärtämään, mitä numero edustaa.
- Selkeys ja ymmärrettävyys: Nimeämällä numerot asianmukaisesti koodin lukijat saavat paremman käsityksen siitä, mitä laskelmat tai koodin osat tekevät.
- Ohjelman ymmärrettävyys: Numeroiden nimeäminen vähentää koodin mysteerisyyttä ja tekee siitä helposti ymmärrettävää.

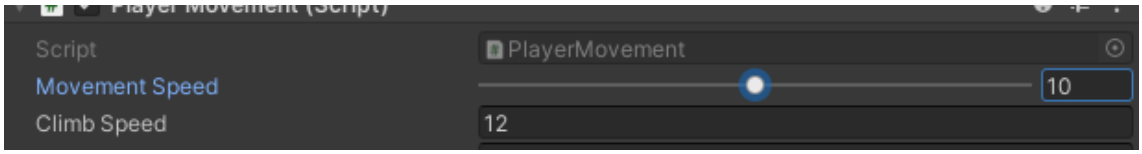
Numeroiden nimeäminen auttaa siis parantamaan koodin luettavuutta ja ymmärrettävyyttä, mikä on erityisen tärkeää suurissa tai monimutkaisissa

ohjelmissa. Selkeästi nimetyt numerot tekevät koodista helposti seurattavaa ja auttavat muita ohjelmoijia ymmärtämään ilman syvempää perehtymistä, mitä koodi tekee. (3, luku 1.5.) Numeroiden nimeämistä havainnollistaa PlayerMovement-luokan liikkeeseen liittyvien arvojen nimeäminen esimerkkikoodissa 6. Kun numerot ovat selkeästi nimetty, lukijalle ei jää epäselväksi, mihin toimintoon numerot liittyvät.

```
public class PlayerMovement : MonoBehavior
{
     [Range(5, 15), SerializeField] private float _movementSpeed;

     [SerializeField] private float _climbSpeed;

     [Tooltip("How high player jumps.",)SerializeField] private float
     _jumpForceUp;

     [SerializeField] private float _jumpForceForward;
     [SerializeField] private float _jumpDuration;
     private float _jumpTimer = 0f;
}
```

Esimerkkikoodi 6. Numeroiden nimeäminen PlayerMovement-luokassa.

Range-attribuutti Unityssa on hyvä esimerkki siitä, miten numeroiden hallintaa ja käyttöä voidaan ohjata ja rajoittaa ohjelmointikäytännöillä, erityisesti kun kyseessä ovat käyttöliittymässä näkyvät arvot. Range-attribuutti auttaa parantamaan ohjelman ymmärrettävyyttä ja käytettävyyttä. Range-attribuutilla varustettu kenttä näkyy liukusäätimenä Inspectorissa. Sille voidaan määritellä minimi- ja maksimiarvot ja rajoittaa, mitä arvoja käyttäjä voi asettaa numerokentille. (6, s. 24.) Esimerkkikoodissa 6 on lisättynä Range-attribuutti _movementSpeed-muuttujaan. Se näkyy Inspectorissa kuvan 1 näyttämällä tavalla.



Kuva 1. MovementSpeed-muuttuja Range-attribuutilla, jolle on asetettu minimi- ja maksimiarvot.

2.5 Kommentointi

Kommenttien tarkoitus on auttaa ohjelman lukijaa ymmärtämään koodia. Hyvät kommentit selkeyttävät ohjelman toimintaa osoittamalla keskeisiä yksityiskohtia tai antamalla laajemman kuvan toiminnoista kuin kommentoimaton koodi (3, luku 1.6). Kommentoitaessa olisi hyvä kertoa, miksi tehdään tiettyjä päätöksiä sen sijaan, että keskityttäisiin vain kertomaan, mitä päätökset ovat. Tämä auttaa tuomaan esiin tietoja, jotka eivät ole ilmeisiä pelkästä koodista.

Cory Housen mukaan ”Koodi on kuin huumori. Jos sitä täytyy selittää, se on huonoa.” (6, s. 43). Tämä tarkoittaa, että hyvin suunniteltu koodi on ymmärrettävää ilman liiallista selittelyä. Kommenttien ei tulisi pyrkiä korvaamaan huonoa koodia, vaan täydentämään hyvää koodia tarpeen tullen.

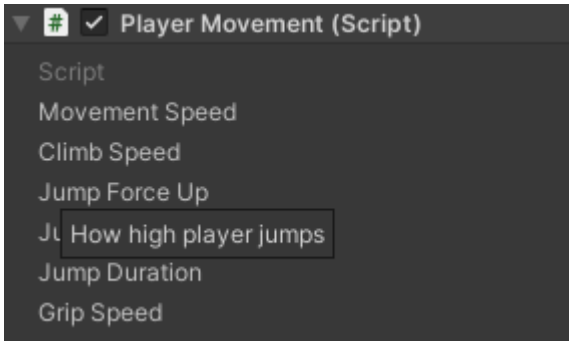
Hyvin sijoitetut kommentit parantavat koodin luettavuutta, mutta liialliset tai pinnalliset kommentit voivat aiheuttaa päinvastaisen vaikutuksen. KISS-periaate korostaa yksinkertaisuutta ohjelmien toteutuksessa, ja periaatetta noudattaessa suurin osa koodista ei tarvitse kommentteja, sillä se on jaettu helposti ymmärrettäviin pieniin osiin. Hyvin nimetty koodi selittää itsensä ilman ylimääräisiä kommentteja. Kommenttien ei tulisi myöskään raportoida itsestään selvää tietoa, kuten yksinkertaisia ohjelmointikonsepteja (esimerkiksi i++ kasvattaa i:n arvoa). (3, luku 1.6.)

Kommentoinnissa on suositeltavaa käyttää yhden rivin kommentteja, jotta ne voidaan sijoittaa lähelle selitettävää koodia. Koodin lähelle sijoitetut kommentit auttavat säilyttämään koodin selkeyden. Kommentit tulisi sijoittaa erillisille riveille eikä rivin loppuun, ja niiden tulisi olla yksinkertaisia ja informatiivisia ilman erikoismerkkejä tai koristeita. Yhden rivin kommenttien aloittamiseksi on suositeltavaa käyttää kaksinkertaista vinoviivaa, ja kommenttiviivojen ja tekstin väliin tulisi laittaa yksi välilyönti. (6, s. 44.) Esimerkki hyvän kommentin rakenteesta:

```
// Tämä on informatiivinen kommentti.
```

Serialisointi on prosessi, jossa muutetaan datastruktuureja tai objektien tiloja muotoon, jota Unity voi tallentaa ja rakentaa uudelleen myöhemmin (6, s. 23). Serialisoinnin yhteydessä on suositeltavaa käyttää SerializeField-attribuuttia, jotta kenttä näkyy Inspectorissa (7). SerializeField-attribuutin käyttö lisäksi tiivistää dataa paremmin kuin muuttujan tekeminen julkiseksi. Data kannattaa ryhmitellä serialisoitaviin luokkiin ja rakenteisiin Inspectorin selkeyttämiseksi (6, s. 23).

Esimerkkikoodissa 6 JumpForceUp-muuttujaan on lisätty myös Tooltip-ominaisuus. Kun Inspectorissa laitetaan kursori kyseisen muuttujan päälle, tulee kyseisessä Tooltip-ominaisuudessa määritelty teksti näkyviin kuvan 2 osoittamalla tavalla. Tooltip-ominaisuutta voidaan hyödyntää serialisoitujen kenttien selittämiseen Inspectorissa kommentin korvikkeena. (6, s. 45.)



Kuva 2. Tooltip-ominaisuuden näkyminen Inspectorissa.

Yhtenäisen tyylin noudattaminen kommenteissa on tärkeää projektin jäsenten yhteistyön kannalta. Lisäksi vanhat ja tarpeettomat kommentit ja koodit tulisi poistaa, ja olisi hyvä pitää TODO-kommentit ajan tasalla, jotta ne eivät hämmennä sovelluskehittäjiä. Lailliset vastuuvapautuslausekkeet on myös hyvä lisätä tarvittaessa kommentteihin. (6, s. 46.)

Ottaen huomioon edellä mainitut ohjelmointikäytännöt, voidaan todeta, että kun kaikki projektin parissa työskentelevät henkilöt käyttävät näitä yhteisiä ohjelmointikäytäntöjä, saadaan aikaan hyvää ja laadukasta koodia, jota on helppo lukea.

3 Suunnitteluperiaatteet olio-ohjelmoinnissa

Suunnitteluperiaatteet (Design Principles) ovat keskeinen osa ohjelmointikäytäntöjä. Tässä luvussa käydään läpi suunnitteluperiaatteita, jotka ovat yleisiä olio-ohjelmoinnissa (OOP). Ne tarjoavat suuntaviivoja ja periaatteita, joiden avulla voidaan suunnitella ja toteuttaa tehokkaita, laadukkaita ja helposti ylläpidettäviä ohjelmistoja. Ohjelmointiperiaatteita seuraamalla voidaan välttää yleisimpiä koodin rikkojia, kuten liian tiivistä riippuvuutta, monimutkaisuutta ja koodin turhaa kopiointia –mikä tekee koodista helposti luettavaa ja laajennettavaa. (8.)

Suunnitteluperiaatteet perustuvat vuosikymmenten kokemukseen ohjelmistotuotannosta ja auttavat kehittäjiä välttämään huonoa koodisuunnittelua (9, luku 7). Vaikka olio-ohjelmoinnin yksinkertaiset periaatteet tarjoavat työkalut ongelmien ratkaisemiseen, ne eivät yksinään takaa kestävää ja helposti ylläpidettävää sovellusta. Sovelluksen kestävyys, ylläpidettävyys ja joustavuus riippuvat suuresti siitä, miten sovellus suunnitellaan, miten sen komponentit yhdistetään ja miten olio-ohjelmoinnin periaatteita käytetään. (10, luku 11.)

3.1 KISS-periaate

KISS-periaate (Keep It Simple, Stupid) on suunnitteluperiaate, joka korostaa järjestelmien toimivuutta ja tehokkuutta pitämällä ne mahdollisimman yksinkertaisina. Periaate on peräisin Yhdysvaltain armeijasta, ja sitä sovelletaan laajasti eri aloilla, kuten käyttöliittymäsuunnittelussa, tuotesuunnittelussa ja ohjelmistokehityksessä. (11).

KISS-periaatteen mukaan systeemit toimivat parhaiten, kun ne pidetään yksinkertaisina eikä niihin lisätä tarpeetonta monimutkaisuutta, mikä auttaa parantamaan systeemin ymmärrettävyyttä, ylläpidettävyyttä ja käytettävyyttä. Suunnittelussa tulisi siis pyrkiä yksinkertaisiin ja selkeisiin ratkaisuihin, ja lisätä monimutkaisuutta vain tarvittaessa ja hyödyllisellä tavalla. KISS-periaate auttaa

varmistamaan, että lopputuloksena on toimiva, helppokäyttöinen ja tehokas ratkaisu. (12, s. 6.)

3.2 DRY-periaate

DRY-periaate (Don’t Repeat Yourself) on keskeinen ohjelmointiperiaate, joka pyrkii vähentämään koodin toistoa ohjelmistokehityksessä. Sen tarkoituksena on ylläpitää koodin tiiviyttä, parantaa ylläpidettävyyttä ja vähentää virheiden riskiä (6, s. 41). DRY-periaatteen noudattaminen edistää lisäksi ohjelman tehokkuutta ja hallittavuutta (11).

Kun DRY-periaatetta sovelletaan, saman logiikan tai toiminnallisuuden toistamista pyritään välttämään koodin useissa paikoissa. On tärkeää huomata, että DRY-periaatetta ei tarvitse tulkita kirjaimellisesti jokaisen koodirivin ainutlaatuisena vaatimuksena. (6, s. 41.) Sen keskeinen idea on kannustaa abstraktioiden käyttöön ja tarpeettomien toistojen välttämiseen koodissa (13), mikä tarkoittaa, että saman logiikan tai toiminnallisuuden tulisi olla keskitetysti yhdessä paikassa ja hyödynnettävissä tarpeen mukaan, eikä koodia tarvitse monistaa kirjaimellisesti jokaisessa käyttötapauksessa. Tällainen lähestymistapa vähentää virheiden riskiä, koska toiminnallisuutta muutetaan ja ylläpidetään vain yhdessä paikassa. (6, s. 41.)

DRY-periaate on läheisessä yhteydessä yhden vastuun periaatteeseen (Single Responsibility Principle eli SRP). SRP:n mukaan jokaisella luokalla tai komponentilla tulisi olla vain yksi vastuualue tai syy muuttua (9, luku 8). Noudattamalla SRP:tä koodin toistoa vältetään ja vastuut jaetaan selvästi eri komponenteille tai luokille, mikä tukee DRY-periaatteen tavoitetta vähentää koodin toistoa ja parantaa koodin laadukkuutta. (6, s. 41.)

3.3 SOLID-periaatteet

SOLID-periaatteet ovat olleet keskeisiä ohjelmistoarkkitehtuurissa lähes kahden vuosikymmenen ajan, erityisesti suurten ja skaalautuvien sovellusten

kehittämisessä. SOLID-periaatteet mainitaan usein olioperusteiden ohjelmoinnin yhteydessä, ja ne tarjoavat ohjenuoria puhtaan, ylläpidettävän ja laajennettavan koodin kirjoittamiseen. (12, s. 34.)

SOLID on lyhenne, joka muodostuu viidestä suunnitteluperiaatteesta, jotka ovat (10, luku 11 s. 1):

- Single Responsibility Principle (yhden vastuun periaate), joka korostaa yksinkertaisia ja pieniä luokkia.
- Open–Closed Principle (avoin-suljettu-periaate), joka keskittyy luokkien suunnitteluun ja ominaisuuksien lisäämiseen.
- Liskov Substitution Principle (Liskovin korvausperiaate), joka käsittelee aliluokkia ja periytymistä.
- Interface Segregation Principle (rajapinnan erotteluperiaate), joka käsittelee rajapintoja ja niiden alttiutta rajapintoja käyttäville komponenteille.
- Dependency Inversion Principle (riippuvuuden kääntämisen periaate), joka keskittyy riippuvuuksien hallintaan sovelluksen komponenttien välillä.

Näiden periaatteiden noudattaminen auttaa kirjoittamaan koodia, joka on kestävää, joustavaa ja ylläpidettävää (10, luku 11 s. 1). Ne parantavat ohjelman ymmärrettävyyttä ja vähentävät luokkien välistä riippuvuutta, mikä tekee koodista ylläpidettävää ja helposti testattavaa (12, s. 8). Niiden käyttäminen auttaa tunnistamaan sovelluksen potentiaaliset riskitilanteet (10, luku 11 s. 1).

SOLID-periaatteiden noudattaminen on yleensä ajan mittaan kannattavaa, koska ne auttavat vähentämään koodin haavoittuvuuksia ja tekevät ohjelmistosta helposti ylläpidettävän ja laajennettavan (10, luku 11 s. 1). Vaikka SOLID-periaatteiden soveltaminen saattaa aluksi vaatia lisätyötä, hyödyt tulevat esiin myöhemmin.

On kuitenkin tärkeää soveltaa näitä periaatteita joustavasti projekteissa. Periaatteiden tulee palvella ohjelmiston tavoitetta eikä toimia tiukkoina sääntöinä, jotka on pakko toteuttaa kaikissa tilanteissa. Jos ollaan epävarmoja, milloin ja miten SOLID-periaatteita tulisi käyttää, kannattaa ottaa huomioon KISS-

periaate. Se auttaa välttämään liiallista abstraktiota, joka voi johtaa tarpeettomaan koodin monimutkaisuuteen. (12, s. 34.)

Yhden vastuun periaate

Yhden vastuun periaate (Single Responsibility Principle eli SRP) on yksi keskeinen SOLID-periaate. Se auttaa pitämään koodin selkeänä, ylläpidettävänä ja laajennettavana (12, s. 8). SRP korostaa, että jokaisella funktiolla, luokalla tai moduulilla tulisi olla vain yksi tehtävä tai vastuualue, mikä auttaa välttämään monimutkaisia ja vaikeasti ylläpidettäviä rakenteita, joissa yksi komponentti vastaa liian monesta eri tehtävästä. (9, luku 8.) Käytän luvussa SRP:stä esimerkkejä omasta projektistani.

SRP kannustaa lyhyisiin ja selkeisiin luokkiin ja metodeihin, joiden nimi kuvaa selvästi niiden toimintaa. Lyhyet koodit ovat helpompia lukea ja ymmärtää kuin pitkät koodit, koska ei ole tarvetta selata kauan koodia koodin lukemiseksi. Useat kehittäjät pitävät lyhyen koodin rajana 200–300 riviä; jos koodia on enemmän kuin 300 riviä tai luokka kattaa useita vastuualueita, se kannattaa jakaa pienempiin osiin. (12, s. 11.)

Kun moduulit tai luokat ovat pieniä ja keskittyvät yhteen vastuuseen, niitä on helpompi käyttää uudelleen eri osissa ohjelmistoa, mikä parantaa koodin laajennettavuutta ja ylläpidettävyyttä (12, s. 11). Perintä on lisäksi helpompaa pienistä luokista kuin suurista luokista, ja pieniä luokkia voidaan muokata tai korvata ilman pelkoa, että ne menevät rikki.

Lisäksi metodien tulisi tehdä vain yhtä asiaa. Niissä tulisi välttää liiallista argumenttien määrää, sillä ne lisäävät koodin monimutkaisuutta. Kun argumenttien määrää vähennetään, metodia on helppo lukea ja testata. Lisäksi metodin ylikuormitusta tulisi välttää; tällöin valitaan vain muutama tapa kutsua metodia ja kehitetään ne erillisiksi, jos metodi tarvitsee erilaisia toimintoja. (6, s. 39.)

Tässä projektissa on tehty pelaajahahmolle erilliset ohjelmakoodit, kuten PlayerMovement, PlayerInput ja PlayerHealth, joista jokainen luokka on vastuussa yhdestä pelaajaan liittyvästä asiasta. Nämä kaikki luokat voitaisiin toteuttaa yhden Player-luokan sisällä, mutta se monimutkaistaisi suunnittelumallia sekoittamalla eri vastuualueita. Kuvassa 3 on projektin pelaajahahmo.

Kuva 3. Projektin pelaajahahmo.

Vaikka yhden vastuun periaatetta voi olla haastavaa toteuttaa täydellisesti, sen noudattaminen auttaa luomaan laadukkaan ja kestävän ohjelmiston (9, luku 8). Hyvin jaoteltu rakenne auttaa pitämään kaiken paikoillaan ja helpottaa uusien ominaisuuksien lisäämistä tai testaamista.

Avoin-suljettu-periaate

Avoin-suljettu-periaate (Open–Closed Principle eli OCP) on yksi SOLID-periaatteista, joka korostaa, että ohjelmistoyksiköiden, kuten luokkien, moduulien ja funktioiden, pitää olla avoinna laajentamiselle, mutta suljettuna muokkaamiselle (9, luku 9).

Tämä tarkoittaa sitä, että luokan tulee olla avoin uusien toimintojen ja ominaisuuksien lisäämiselle, mutta samalla suljettu niin, että alkuperäistä koodia ei tarvitse muokata, mikä auttaa säilyttämään ohjelmiston yhteensopivuuden ja

eheyden (9, luku 9). Jos uusissa laajennuksissa on ongelmia, alkuperäistä koodia ei tarvitse katsoa, vaan tiedetään, että vika on kyseisissä laajennuksissa (12, s. 14).

OCP voi käytännössä tarkoittaa abstraktin kantaluokan perimistä toteuttavalle luokalla. Peritty luokka voi aina lisätä metodeja ja ominaisuuksia, ja siten kantaluokka on avoin laajentamiselle. Samalla kantaluokka on suljettu, koska sitä ei voi muuttaa. Abstraktin kantaluokan lisäksi OCP voi myös tarkoittaa rajapintoja. Hyvin määritellyt rajapinnat ovat suljettuja, mutta luokan toimintaa voidaan laajentaa uusilla toiminnoilla tai lisäämällä siihen rajapinta. (9, luku 9.)

OCP on keskeinen OOP-suunnittelun periaate, joka tarjoaa joustavuutta, uudelleenkäytettävyyttä ja ylläpidettävyyttä ohjelmistokehityksessä. Se kannustaa abstraktioiden käyttöön ohjelman niissä osissa, jotka todennäköisimmin muuttuvat, mutta samalla vastustaa liian aikaista abstrahointia, joka voi johtaa tarpeettomaan monimutkaisuuteen. (9, luku 9.)

Liskovin korvausperiaate

Liskovin korvausperiaatteen (Liskov Substitution Principle eli LSP) keskeinen idea on varmistaa, että johdettu luokka voidaan korvata sen kantaluokalla ilman, että ohjelman toiminta muuttuu tai rikkoutuu (9, luku 10). Se tarkoittaa sitä, että kaikkialla, missä voidaan käyttää kantaluokkaa, tulisi voida käyttää myös sen johdettuja luokkia ilman odottamattomia sivuvaikutuksia. LSP mahdollistaa joustavan ja laajennettavan ohjelmiston suunnittelun, sillä se auttaa välttämään hauraita hierarkioita ja varmistaa luokkien oikeanlaisen toiminnan ja korvattavuuden eri tilanteissa. (12, s. 18–21.)

LSP edellyttää, että johdettua luokkaa voidaan käyttää kantaluokan sijasta ilman, että ohjelma tarvitsee muutoksia toimiakseen oikein, mikä varmistaa ohjelman vakauden ja ennakoitavuuden. Johdetun luokan tulee tarjota samat julkiset jäsenet ja käyttäytyminen kuin kantaluokan, jotta luokat voidaan vaihtaa keskenään ongelmitta. On suositeltavaa välttää liian monimutkaisia abstraktioita kantaluokassa, sillä ne voivat vaikeuttaa korvaamista ja johtaa LSP:n

rikkoutumiseen. (12, s. 18–21.) Kantaluokan tulisi keskittyä perusominaisuuksiin ja delegoida erikoistunut logiikka johdetuille luokille, mikä auttaa säilyttämään LSP:n noudattamisen ja tekee aliluokista kestäviä ja joustavia. (12, s. 15.)

Esimerkkinä LSP:n soveltamisesta peliprojektissa on Monster-luokka, josta voidaan johtaa luokat Bear ja Ghost. Jos kaikkia näitä voidaan käsitellä yleisesti Monster-luokan kautta ilman, että ohjelma toimii eri tavalla, se osoittaa Liskovin korvausperiaatteen noudattamisen. Kuvassa 4 on Ghost-luokan hahmo, joka voidaan korvata LSP:n mukaisesti Monster-luokan hahmolla.

Kuva 4. Ghost-luokan hahmo projektissa.

Kun suunnitellaan luokkahierarkioita, on hyvä varmistaa, että perintää käytetään harkiten ja kaikki luokat hierarkiassa täyttävät LSP:n. On hyvä varmistaa myös, että johdettujen luokkien toiminnot eivät poikkea odottamattomasti kantaluokan toiminnoista. LSP:n mukaan rajapintojen tai luokkien yhdistelmiä kannattaa suosia perinnän sijaan, mikä tarkoittaa, että toiminnan välittäminen kannattaa

toteuttaa mieluummin rajapintojen tai erillisten luokkien kuin perinnän avulla. Tällä tavalla pystytään rakentamaan yhdistelmiä erilaisista halutuista toiminnoista. (12, s. 18–21.)

Liskovin korvausperiaate on yksi keskeisistä SOLID-periaatteista, joka mahdollistaa avoin-suljettu-periaatteen; korvattavissa olevat aliluokat sallivat kantaluokan olevan laajennettavissa ilman muokkaamista. (9, luku 10.)

Rajapinnan erotteluperiaate

Rajapinnan erotteluperiaate (Interface Segregation Principle eli ISP) korostaa useiden selkeästi määriteltyjen rajapintojen luomisen tärkeyttä yhden suuren yleisen rajapinnan sijaan (12, s. 21). Periaatteen tavoitteena on varmistaa, että luokka ei ole pakotettu riippumaan niistä metodeista, joita se ei käytä (10, luku 11 s. 5).

Käytännössä ISP tarkoittaa suurten rajapintojen jakamista pieniin, eri tarkoituksiin eroteltuihin rajapintoihin (12, s. 22). Jokainen rajapinta määrittelee vain ne toiminnot, joita tietyt luokat tarvitsevat.

ISP tukee rajapintojen yhdistelmiä perinnän sijaan, mikä on samansuuntainen lähestymistapa kuin Liskovin korvausperiaatteessa. Noudattamalla ISP:tä järjestelmät tulevat modulaarisiksi ja minimoivat riippuvuuksia luokkien välillä, mikä tekee koodista itsenäistä ja helposti ylläpidettävää (9, luku 12). ISP parantaa luokkien joustavuutta ja uudelleenkäytettävyyttä sekä estää ei-toivottujen vaikutusten syntymisen rajapintojen muutosten yhteydessä (12, s. 21).

ISP vastustaa suuria yleisiä rajapintoja, sillä ne sisältävät tarpeettomia metodeja, ja kun yksi luokka tekee muutoksia isoon rajapintaan, muutokset vaikuttavat myös kaikkiin muihin luokkiin (9, luku 12). Koska luokka voi toteuttaa useamman rajapinnan, on tärkeää toteuttaa vain ne, jotka vastaavat luokan tarpeita. Näiden rajapintojen tulisi sisältää vain ne metodit, jotka ovat olennaisia luokan toiminnallisuuden kannalta, mikä antaa maksimaalisen joustavuuden

luokkien toteuttamisessa ja samalla pitää rajapinnat kompakteina ja keskittyneinä. (12, s. 21.)

Sekä yhden vastuun periaate että rajapinnan erotteluperiaate edistävät ohjelmointikomponenttien yksinkertaistamista ja yhtenäisyyttä. SRP keskittyy koko komponenttiin, kun taas ISP keskittyy erityisesti julkisten rajapintojen yksinkertaistamiseen, varmistaen, että luokat sisältävät vain tarvitsemiansa metodeja. (10, luku 11 s. 5.)

Riippuvuuden kääntämisen periaate

Riippuvuuden kääntämisen periaate (Dependency Inversion Principle eli DIP) tarkoittaa sitä, että korkean tason moduulien (high-level module) ei tulisi riippua matalan tason moduulien toiminnasta. Sen sijaan molempien tulisi riippua abstraktioista (9, luku 11). Riippuvuuksien kääntäminen abstraktioihin auttaa vähentämään korkeita riippuvuuksia ja tekemään koodista helposti ylläpidettävää ja muokattavaa. Ideaalisesti pyritään mahdollisimman vähään riippuvuuteen luokkien välillä. (12, s. 24.)

Ohjelmistoa luodessa luokkien välille saattaa syntyä vahvoja riippuvuuksia, jos korkean tason luokka tietää liikaa matalan tason luokan toteutuksesta. Jokainen riippuvuus sisältää riskin ja voi johtaa herkästi rikkoutuvaan koodiin, jossa yhden osan muuttaminen vaikuttaa useisiin muihin osiin. Riippuvuuden kääntämisen periaate pyrkii estämään tätä ongelmaa varmistamalla, että abstraktiot määrittävät luokkien välisen rajapinnan yksityiskohtien sijaan. (12, s. 24.)

DIP:n mukaisesti abstraktioiden tulisi olla riippumattomia yksityiskohdista, mutta yksityiskohtien tulisi riippua abstraktioista, mikä tarkoittaa sitä, että korkean tason luokat käyttävät matalan tason luokkia abstraktioiden kautta ilman, että niiden tarvitsee tietää yksityiskohtia matalan tason luokkien toteutuksesta. Käytännössä luokka sisältää rajapintatyyppisiä ominaisuuksia sen sijaan, että perisi suoraan matalan tason luokan. (9, luku 11).

DIP toteutetaan käyttämällä rajapintoja ja abstraktioita, joiden avulla luokat voivat kommunikoida keskenään. Korkean tason luokka ei suoraan periydy matalan tason luokasta, vaan se käyttää matalan tason luokan toteuttamaa rajapintaa, mikä mahdollistaa joustavan ja helposti laajennettavan ohjelmakoodin. (9, luku 11).

Riippuvuuden kääntämisen periaate on keskeinen osa SOLID-periaatteita ja auttaa luomaan modulaarisia ja uudelleenkäytettäviä ohjelmistoja. Sen avulla ohjelmisto säilyy joustavana ja helposti muokattavana, mikä on tärkeää pitkäaikaisen ylläpidon kannalta. DIP auttaa arvioimaan luokkien välisiä suhteita ja laajentamaan projektia pienillä riippuvuussuhteilla. (12, s. 24–24.)

3.4 Abstraktio ja rajapinta suunnitteluperiaatteissa

Monet SOLID-periaatteet suosivat rajapintoja perinnän sijaan, mutta silti monia suunnitteluperiaatteita ja -malleja voidaan noudattaa myös abstrakteilla luokilla (12, s. 30). Abstraktien luokkien etuna on, että ne voivat sisältää kenttiä, vakioita ja staattisia jäseniä, ja niillä voi olla enemmän rajoituksia kuin rajapinnoilla, kuten suojattu ja yksityinen pääsy. Abstraktit luokat mahdollistavat logiikan määrittämisen, mikä voidaan jakaa konkreettisten luokkien kesken, jotka edustavat ydintoiminnallisuuksia. Perintä toimii hyvin, kunnes tarvitaan johdettu luokka, joka sisältää piirteitä kahdesta kantaluokasta. C#-kielessä luokka voi periä vain yhden kantaluokan. (14.)

Rajapinnat puolestaan antavat enemmän joustavuutta; jos jokin ei sovi perintään, luokkien välisiä suhteita on helpompi hallita käyttäen rajapintoja. Rajapinnat sisältävät kuitenkin vain jäsenten määrittelyjä, ja toteuttavan luokan on tarjottava logiikka näille rajapinnan jäsenille. (14.)

Abstrakteja luokkia käytetään ydintoiminnallisuuksien määrittelyyn ja koodin jakamiseen. Rajapinnat määrittävät rajat, joissa joustavuutta tarvitaan. Sekä abstraktit luokat että rajapinnat ovat pätevä keino saavuttaa abstraktioita C#-kielellä. Valinta niiden välillä riippuu tilanteen erityisistä tarpeista. (12, s. 30–32.)

Suunnitteluperiaatteet toimivat pohjana suunnittelumalleille, joita käsitellään seuraavassa osiossa. Suunnittelumallit varmistavat, että koodi on joustavaa, laajennettavaa ja helposti muokattavaa.

4 Suunnittelumallit pelikehityksessä

Suunnittelumallit ovat ohjelmistosuunnittelussa keskeisiä käytäntöjä, jotka tarjoavat testattuja ratkaisuja yleisiin ohjelmointiongelmiin. Ne ovat päteviä lähestymistapoja ongelmien ratkaisemiseksi ja uudelleenkäytettäviä erilaisissa sovelluskehityksen tilanteissa. Suunnittelumallit tarjoavat abstraktin pohjan, jota voidaan muokata omien tarpeiden mukaisesti. Ne eivät siis tarjoa tarkkoja vastauksia, vaan yleispäteviä ratkaisuja. Niiden avulla voidaan välttää toistoa koodissa ja keskittyä enemmän laatuun. (15, luku 2.; 16, luku 2.)

Kun suunnittelumalleja ymmärretään, niihin perustuvien ratkaisujen avulla voidaan käsitellä monimutkaisia suunnitteluperiaatteita ja soveltaa tuttuja ratkaisuja päivittäisiin ohjelmointiongelmiin. Vaikka kaksi ohjelmaa noudattaisi samaa suunnittelumallia, niiden toteutukset voivat olla hyvin erilaisia. (12, s. 5.)

Ohjelmoijan ei tarvitse osata suunnittelumalleja ulkoa, mutta niiden oppiminen auttaa kehittymään paremmaksi ohjelmoijaksi. On tärkeää oppia tunnistamaan sopivia suunnittelumalleja, koska kaikki suunnittelumallit eivät sovi kaikkiin tilanteisiin ja jokaisella suunnittelumallilla on omat vahvuutensa ja heikkoutensa. (12, s. 36–38.).

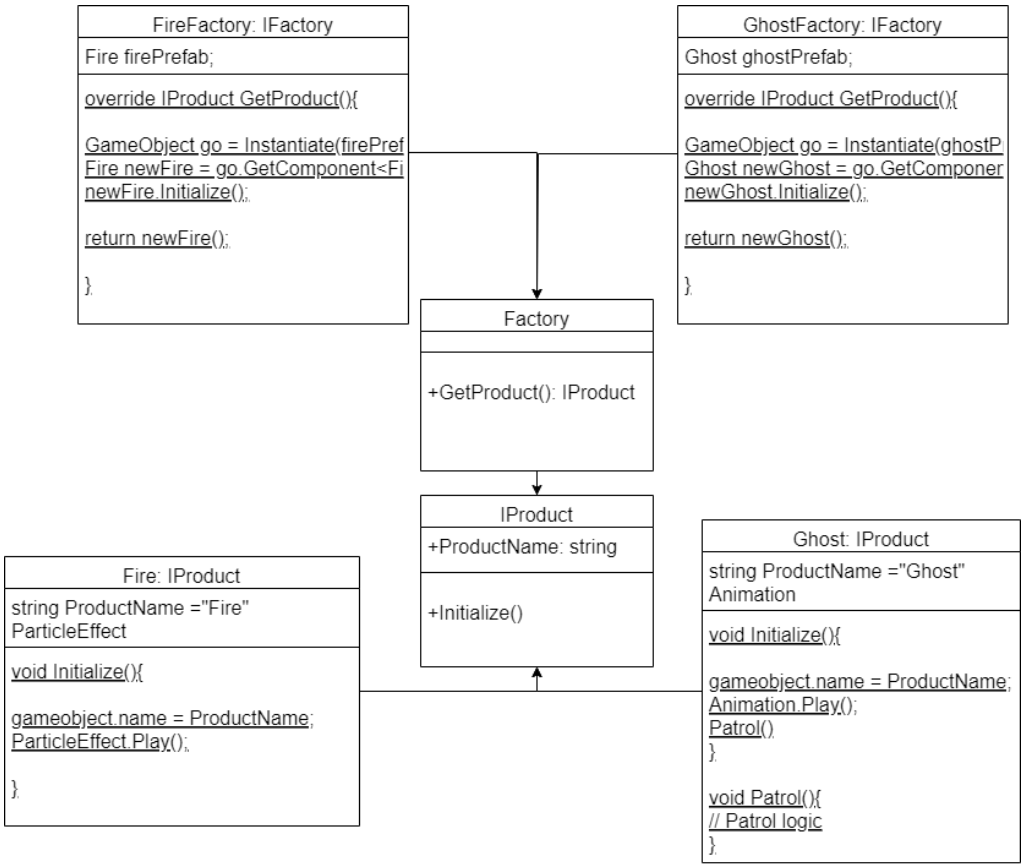
Suunnittelumalleja voidaan ajatella työkaluina, jotka parantavat koodin luettavuutta ja auttavat rakentamaan suuria ja skaalautuvia sovelluksia oikein käytettynä. Ne tekevät koodin uudelleenkäytettäväksi, laajennettavaksi ja ylläpidettäväksi toteuttaen samalla hyviä suunnittelutekniikoita ohjelmoinnissa (15, luku 2). Suunnittelumallit lisäksi parantavat koodin luettavuutta ja tekevät koodipohjasta siistin (12, s. 5).

4.1 Tehdasmalli

Monet pelit luovat erilaisia objekteja, kuten vihollisia tai esteitä, pelin aikana. Tehdasmalli (Factory Pattern) on suunnittelumalli, joka tarjoaa tähän tarkoitukseen erityisen tehtaaksi kutsuttavan objektin, joka luo toisia objekteja (tuotteita) (17). Tehdasmalli kiteyttää yhteen objektiin monimutkaiset yksityiskohdat, jotka liittyvät objektien luomiseen, ja auttaa pitämään koodin siistinä (18).

Tehdasmallin avulla voidaan luoda uusia objekteja noudattamalla perusrajapintaa tai kantaluokkaa (18). Jos tehdasmallin tuotteet seuraavat jompaakumpaa näistä, tuotteiden sisälle voidaan laittaa omaa rakennuslogiikkaa, joka piilotetaan tehtaalta itseltään, vähentäen objektien luomiseen liittyviä riippuvuuksia (18). Uusien tuotteiden luominen tulee siten laajennettavaksi.

Tehdasmallia voidaan laajentaa luomalla aliluokkia tehtaasta, jotta erilaiset tehtaat voivat tuottaa tiettyjä tuotteita tarpeen mukaan (17). Tehdasmallin laajentaminen auttaa eriyttämään objektien käyttäytymisen ja rakenteen tehtaan luomisesta. Kuva 5 esittää, miten tehdasmalli voidaan toteuttaa käyttämällä IProduct-rajapintaa ja abstraktia Factory-luokkaa ja Factory-luokan aliluokkia eri tyyppisten tuotteiden luomisessa.



Kuva 5. Tehdasmallin esittäminen kaavion avulla.

Esimerkissä on kaksi eri tuotetta, Fire ja Ghost, jotka molemmat toteuttavat IProduct-rajapinnan. Rajapinnassa määritellään tuotteen nimi ja Initialize-funktio, mutta tuotteet voivat itse määritellä nämä ominaisuudet omalla tavallaan.

Factory-kantaluokalla on metodi GetProduct, joka on abstrakti ja palauttaa IProduct-rajapinnan, mikä tarkoittaa sitä, että jokaisen tehtaan täytyy toteuttaa GetProduct-metodi omalla tavallaan, jotta ne voivat palauttaa halutun tyyppisen tuotteen.

Rajapinnan (IProduct) käyttö tehtaan yhteydessä on tärkeää, koska se määrittelee objektien toiminnallisuuden erottaen sen konkreettisesta toteutuksesta. Tällä tavoin tehtaan käyttämä rajapinta tarjoaa yleisen käyttöliittymän, jonka avulla eri tuotteita voidaan luoda ja käsitellä samalla tavalla, riippumatta niiden tarkasta toteutuksesta. Esimerkissä FireFactory toteuttaa GetProduct-metodinsa niin,

että se luo uuden Fire-tyyppisen tuotteen, ja GhostFactory vastaavasti luo Ghost-tyyppisen tuotteen. Näin tehdasmalli mahdollistaa erilaisten tuotteiden luomisen yhtenäisellä ja laajennettavalla tavalla.

Tehdasmallin kehittämiseen on useita tapoja, kuten sanakirja-tyyppisen muuttujan käyttäminen objektien luomiseen, staattisen tehdasluokan käyttäminen yksinkertaisemman käyttöliittymän saavuttamiseksi tai tehdasmallin yhdistäminen objektivarastoon objektien luomisen ja hallinnan helpottamiseksi. Lisäksi tehdasmallia voidaan soveltaa myös muissa ohjelmistoissa eikä pelkästään peleissä. (12, s. 43–44.)

Tehdasmallissa uusien objektien luominen on helppoa ja laajennettavaa ilman olemassa olevan koodin muuttamista, mikä on hyödyllistä silloin, kun tarvitaan monia samantyyppisiä objekteja ja niiden luominen vaatii abstraktiota, ylläpidettävyyttä ja laajennettavuutta (18). Lisäksi tehtaan koodi pysyy suhteellisen lyhyenä ja siistinä, koska jokaisen tuotteen sisäinen logiikka on eriytetty omiksi luokikseen.

Tehdasmalli on kuitenkin monimutkainen toteuttaa, ja siihen saattaa liittyä paljon luokkien ja aliluokkien luomista (12, s. 43). Sen takia sitä ei kannata käyttää liiallisesti, jos sovelluksessa ei ole tarvetta monimutkaiselle objektien luontilogiikalle tai niiden luonti ei vaadi suurta abstraktiota. Silloin tehdasmalli voi aiheuttaa suuren määrän turhaa monimutkaisuutta ohjelmaan. (16, luku 9 s. 10.)

4.2 Objektivarasto

Objektivarasto (Object Pool) on optimointitekniikka, joka parantaa ohjelmiston suorituskykyä ja resurssien hallintaa säilyttämällä ennalta luotuja ja alustettuja objekteja uudelleenkäytettävässä varastossa. Se vähentää objektien luomisesta ja tuhoamisesta aiheutuvaa kuormitusta ja auttaa välttämään resurssien ehtymistä erityisesti tilanteissa, joissa objektien luominen ja tuhoaminen on kallista tai vaivalloista. (19.)

Unity 2021:ssä ja sitä uudemmissa versioissa Unity sisältää valmiin objektivarastosysteemin, joten sitä ei tarvitse rakentaa alusta asti. UnityEngine.Pool-ohjelmointirajapinta antaa pääsyn Unityn omaan objektivarastosysteemiin ja IObjectPool-rajapintaan. (12, s. 50.) Tässä insinöörityössä käytetään Unityn objektivarastoa ammusten ja esteiden luomiseen. Esimerkkikoodissa 7 on ObjectPoolManager-luokka, jossa on esimerkki ammusten objektivaraston toiminnasta.

```
public class ObjectPoolManager : MonoBehavior
{
     [SerializeField] private CakeProjectile _cakeProjectile;

     private IObjectPool<CakeProjectile> _cakePool;
     public IObjectPool<CakeProjectile> CakePool { get => _cakePool; }

     //Throw an exception if trying to return an existing item, already
     //in the pool.
    [SerializeField] private bool _cakeCollectionCheck = true;
    //Extra options to control the pool capacity and maximum size.
    [SerializeField] private int _cakeDefaultCapacity = 20;
    [SerializeField] private int _cakePoolMaxSize = 100;}


     private void Awake()
     {
          _cakePool = new ObjectPool<CakeProjectile>(CreateProjectile,
               OnGetFromPool, OnReleaseToPool, OnDestroyPooledObject,
               _cakeCollectionCheck, _cakeDefaultCapacity,
               _cakePoolMaxSize);
     }

     private CakeProjectile CreateProjectile()
     {
        CakeProjectile cakeProjectile = Instantiate(_cakeProjectile);
        cakeProjectile.CakePool = _cakePool;
        return cakeProjectile;
     }

    private void OnReleaseToPool(CakeProjectile cakeObject)
    {
        cakeObject.gameObject.SetActive(false);
    }

    private void OnGetFromPool(CakeProjectile cakeObject)
    {
        cakeObject.gameObject.SetActive(true);
    }

     private void OnDestroyPooledObject(CakeProjectile cakeObject)
    {
        Destroy(cakeObject.gameObject);
    }

}
```

Esimerkkikoodi 7. Ammusten objektivarasto ja siihen liittyvät toiminnallisuudet.

Koodin alussa määritellään CakeProjectile-tyyppinen muuttuja, jota halutaan käyttää ammusten luomisessa. Yksityinen ja julkinen objektivarasto -muuttujat varmistavat, että objektivarastoa ei pystytä muokkaamaan luokan ulkopuolelta. Loput muuttujat määrittävät objektivaraston asetuksia. Awake-metodissa luodaan uusi objektivarasto _cakePool-muuttujan arvoksi. Objektivarastoa

luodessa objektivarasto ottaa parametreihin arvoksi metodit objektin luomiseen, varastosta ottamiseen, takaisin laittamiseen ja tuhoamiseen sekä aikaisemmin mainittujen muuttujien asetukset. Kapasiteetin ja maksimikoon lisäksi parametreissä on CollectionCheck-boolean, joka tarkoittaa käytännössä tarkastusta, jossa katsotaan, onko varastoon palautettava objekti jo ennestään varastossa.

Jokainen este ja ammus on objektivarasto-objekti, ja objektivarastoissa on määritelty maksimikoko resurssien liikakäytön estämiseksi. Esimerkkikoodissa 8 näytetään ammuksen CakeProjectile-luokka, jossa annetaan viite ammusten objektivarastoon.

```
public class CakeProjectile : MonoBehavior
{
     private IObjectPool<CakeProjectile> _cakePool;

    //Give the projectile a reference to its ObjectPool.
     public IObjectPool<CakeProjectile> CakePool { set => _cakePool =
     value; }
}
```

Esimerkkikoodi 8. CakeProjectile-luokka objektivarastoa varten.

Objektien palautus objektivarastoon tapahtuu törmäystarkistuksen avulla; kun este tai ammus osuu johonkin, se palautetaan takaisin objektivarastoon seuraavaa käyttöä varten. Kuvassa 6 näytetään, miltä ampuminen näyttää pelin sisällä.

Kuva 6. Pelaaja ampuu kakun muotoisia ammuksia. Yksi ammuksista on osunut Ghost-hahmoon, joka on tuhoutunut iskun osuessa.

Objektivarastoon tehtiin parannuksia lisäämällä siihen singleton-malli helpottamaan objektivarastoon pääsyä eri puolilta koodia, ja objektivarastossa käytettiin lisäksi sanakirja-tyyppistä muuttujaa, jotta voidaan hallinnoida useampaa objektivarastoa projektissa käytettäville eri esteille.

Objektivarastot voivat vähentää viivettä, joka liittyy roskankeräyksen väliaikaiseen kuormittumiseen, joka usein ilmenee, kun muistia varataan ja vapautetaan usean objektin luomisen ja tuhoamisen yhteydessä. Objektien luomisen ja tuhoamisen sijaan objektivarastoa käytettäessä objekteja vain kytketään päälle ja pois päältä. (12, s.46–49.) Kuva 7 havainnollistaa kyseistä toimintaa Unityn hierarkiassa.

Kuva 7. Kakun muotoisten ammusten objektivarasto.

Objektivaraston käyttö on hyödyllistä etenkin silloin, kun on tarve luoda ja tuhota useita objekteja kerralla. Objektivarasto luo objekteja asetusten mukaiseen määrään asti, jonka jälkeen niiden näkyvyystilaa vain muutetaan.

4.3 Singleton-malli

Singleton-malli (Singleton Pattern) on suunnittelumalli, jonka tarkoituksena on varmistaa, että jokaisesta luokasta on olemassa vain yksi ilmentymä ja tarjota globaali pääsy tähän ilmentymään (20). Singleton-mallia käytetään yleisesti tilanteissa, joissa tarvitaan vain yksi objekti koordinoimaan toimintaa koko sovelluksen laajuudella, kuten pelin hallintaan tai muiden hallintatason objektien käytössä (12, s. 54–56).

Tärkeä ero singletonin ja staattisen luokan välillä liittyy niiden ilmentymän luontitapaan ja elinkaareen. Ensinnäkin singleton voidaan luoda vasta, kun sitä tarvitaan ensimmäistä kertaa, mikä tarkoittaa sitä, että se ei vie resursseja ennen kuin sitä tarvitaan, kun taas staattinen luokka luodaan automaattisesti sovelluksen käynnistäessä (20). Staattisilla luokilla on aina olemassa ilmentymä, mikä voi johtaa resurssien käyttöön jopa silloin, kuin luokkaa ei vielä tarvita.

Singletonit ovat hyödyllisiä silloin, kun tarvitaan vain yksi ilmentymä koordinoimaan sovelluksen toimintaa läpi sen elinkaaren. Niiden avulla voidaan hallita sovelluksen tilaa ja resursseja keskitetysti (20). Staattiset luokat taas soveltuvat tilanteisiin, joissa tarvitaan toistuvaa käyttöä samanlaisille apumetodeille tai vakioille ilman, että tarvitaan ilmentymän luontia tai hallintaa.

Singleton-mallia käytettiin projektissa hallintatason objektien, kuten GameManager ja AudioManager, kanssa, koska se mahdollistaa yksinkertaisen käytön kaikkialla projektissa, koska singletonista on saatavilla aina vain yksi ilmentymä. Esimerkkikoodissa 9 kutsutaan kyseisiä luokkia Ghost-luokan funktiossa. Singleton-mallin käyttäminen tekee projektin tilan hallinnasta helppoa. Lisäksi sitä käytetään projektin objektivarastossa, jotta varastoituihin objekteihin on helppo pääsy kaikkialta koodista.

```
AudioManager.Instance.PlaySound(“Bloody punch”, true);
GameManager.Instance.IncreaseScore(_scoreValue);
```

Esimerkkikoodi 9. AudioManager ja GameManager -luokkien kutsuminen koodissa.

Singleton-mallilla on omat etunsa erityisesti pienissä projekteissa, kuten peleissä, joissa tarvitaan yksinkertaista ja nopeaa tapaa hallita keskitetysti sovelluksen tilaa ja resursseja. Singletonit ovat suorituskykyisiä ja niiden käytön oppiminen on nopeaa. Lisäksi ne tarjoavat helppokäyttöisen tavan käsitellä globaaleja tiloja. (12, s. 59–60.)

Singleton-malli on kuitenkin pahamaineinen mallin helppokäyttöisyyden vuoksi, mikä altistaa singletonin väärinkäytöksille. Yleensä singletonin käyttö viittaa

siihen, että ohjelman osat ovat tiukasti sidoksissa toisiinsa tai että logiikka on hajautettu eri puolille koodia, mikä vaikeuttaa testaamista ja ylläpitoa. (16, luku 9. s. 4.) Erityisesti suurissa projekteissa singletonin käyttöä tulisi harkita tarkkaan, sillä se voi johtaa tarpeettomiin globaaleihin tiloihin ja riippuvuuksiin (12, s. 54–56).

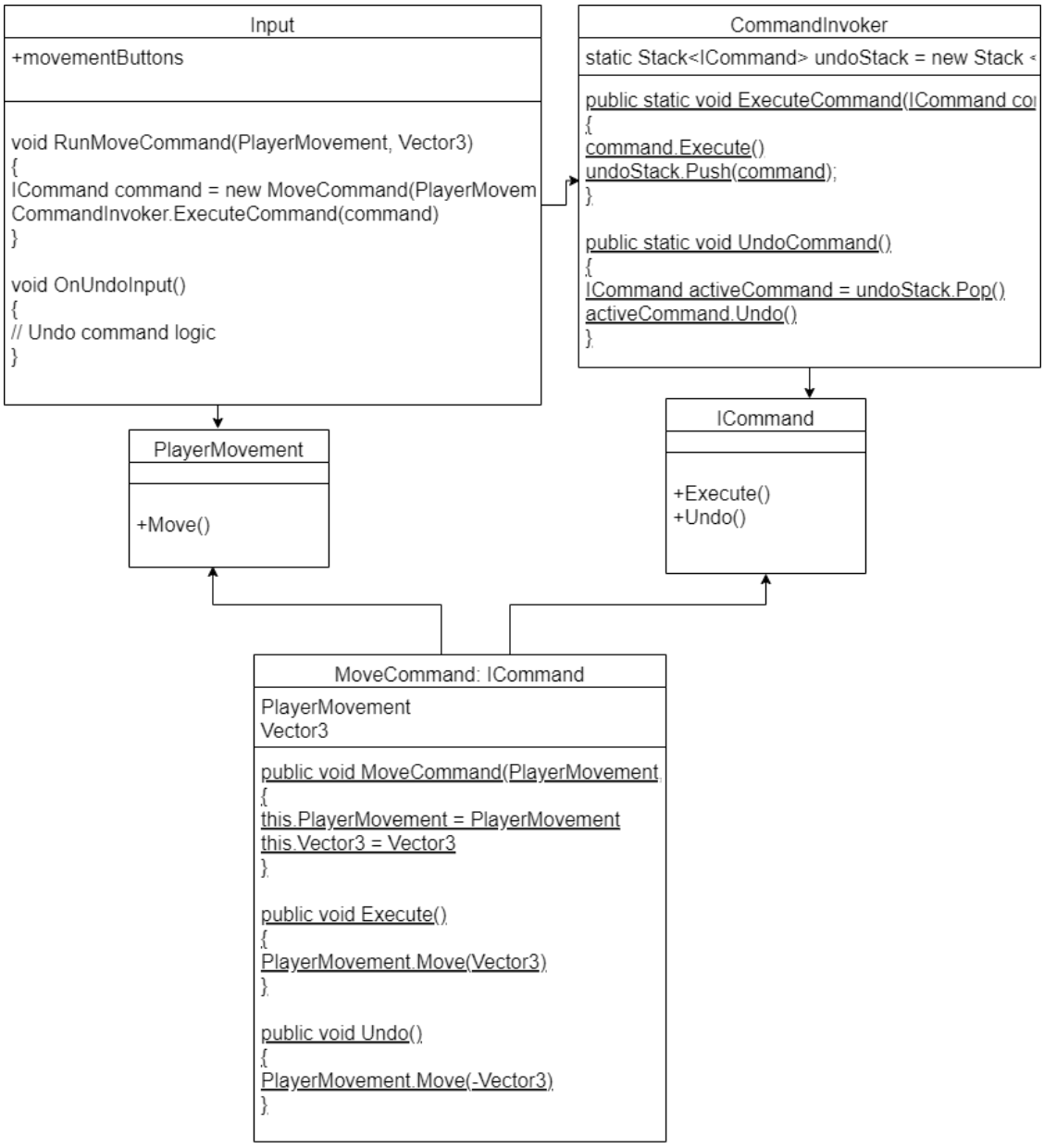
Singletonit rikkovat useita suunnitteluperiaatteita, joten niiden käyttöä tulisi rajoittaa, ja niitä tulisi käyttää vain sellaisiin luokkiin, jotka todella hyötyvät globaalista pääsystä aiheuttamatta liikaa riippuvuuksia tai monimutkaisuutta (20).

4.4 Komentomalli

Komentomalli (Command Pattern) on suunnittelumalli, jota käytetään tilanteissa, joissa halutaan seurata ja hallita tiettyjen toimintojen sarjaa (16, luku 9. s. 8). Esimerkiksi strategiapelissä, jossa pelaaja suunnittelee useita vuoroja ennen niiden toteuttamista, komentomalli voi olla hyödyllinen.

Komentomallissa pyritään tiivistämään metodikutsut, pyynnöt ja operaatiot yhteen komento-objektiin, joka tallennetaan kokoelmaan, esimerkiksi jonoon tai pinoon (16, luku 9. s. 8). Komento-objekti mahdollistaa toimintojen ajoituksen hallinnan ja viivästyttämisen sekä toimintojen peruuttamisen myöhempää toistoa varten (21).

Komentomalli toimii ikään kuin pienenä puskurina, ja sen avulla voidaan helpottaa monimutkaisten toimintojen hallintaa ja niiden toistamista tietyssä järjestyksessä tai takaperin (12, 62). Esimerkiksi pelaajan liikesarjojen tallentaminen ja toistaminen taistelupelissä voi olla yksi käyttötarkoitus. Projektissa ei ollut tarvetta komentomallin käytölle, mutta se olisi voitu toteuttaa, esimerkiksi pelaajan liikkumisen seurannassa ja toistamisessa, kuvassa 8 esitetyllä tavalla.



Kuva 8. Kaavio komentomallin käyttämisestä pelaajan liikkumisessa.

Komentomallin toteuttamiseen tarvitaan komento-objekti, joka sisältää tietyn toiminnon ja mahdollisuuden peruuttaa se (21). Esimerkissä se on MoveCommand-objekti, joka toteuttaa ICommand-rajapinnan, jossa on Execute- ja Undo-metodit toimintojen suorittamiseen ja perumiseen. Jokainen komento-objekti määrittää näihin metodeihin liittyvät omat toiminnallisuutensa.

Komentojen hallintaan tarvitaan myös CommandInvoker-luokka, joka sisältää ExecuteCommand- ja UndoCommand-metodit komento-objektin suorittamiseen ja peruuttamiseen sekä undo stack -pinon komento-objektien säilyttämiseen (21).

Kun pelaaja painaa näppäintä liikkumiseen, Input-luokka luo uuden komennon (MoveCommand) liikkumiseen ja lähettää sen CommandInvoker-luokalle suoritettavaksi. CommandInvoker puolestaan käsittelee komennon ja lisää sen undo stack-pinoon tarvittaessa peruutusta varten, minkä jälkeen liiketoimintalogiikka (PlayerMovement) toteuttaa liikkeen. Koska komennot säilötään pinoon, niitä voidaan tarvittaessa käydä läpi järjestyksessä ja peruuttaa yksi kerrallaan, jolloin pelaajan sijainti voidaan palauttaa haluttuun kohtaan.

Vaikka komentomalli tarjoaa etuja metodikutsujen hallinnassa, se samalla lisää rakennetta ja luokkia ohjelmaan. Siksi on tärkeää harkita tarkkaan, missä tilanteissa komento-objektien käyttö on tarpeellista ja hyödyllistä, jotta vältytään ylimääräisen monimutkaisuuden lisäämiseltä koodiin. (12, s. 66.)

Komentomallin käyttö voi vähentää suoraa riippuvuutta kutsuvan ja vastaanottavan osan välillä, mikä tekee ohjelmasta modulaarisen ja helpottaa osien muokkaamista ja uudelleenkäyttöä erillisinä komponentteina (16, luku 9. s. 8). Komentomallin oppimisen myötä voidaan hallita paremmin toimintojen ajoitusta, toistoa ja peruuttamista, ja se tarjoaa joustavuutta erilaisten komentojen toteuttamiseen ja niiden uudelleenjärjestelyyn tarvittaessa (12, s. 66).

4.5 Tilamalli

Tilamalli (State Pattern) on suunnittelumalli, jota käytetään tilanteissa, joissa halutaan hallita objektin sisäistä tilaa ja sen käyttäytymistä eri tilojen perusteella (22). Yhdessä tilassa oleva objekti suorittaa tiettyä toiminnallisuutta, ja tila voi siirtyä toiseen tilaan riippuen tietyistä ehdoista (12, s. 72).

Tilamalli auttaa organisoimaan logiikkaa selkeäksi ja ratkaisee kaksi yleistä objektin tilojen hallintaan liittyvää ongelmaa. Ensinnäkin objektin pitäisi vaihtaa käyttäytymistä objektin sisäisen tilan muuttuessa, ja toisekseen tilaan liittyvän käytöksen on oltava määritelty itsenäisesti, eikä uusien tilojen lisäämisen tulisi vaikuttaa olemassa oleviin tiloihin. (12, s. 72.)

Perinteinen switch- tai if-lauseisiin perustuva tilojen hallinta on hankala ja altistaa virheille erityisesti tilojen määrän kasvaessa (12, s. 70). Aluksi projektissa käytettiin tällaista perinteistä keinoa tilojen hallinnassa Monster-luokassa, joka sisälsi Idle-, Chase- ja ResetChase-tilat. Esimerkkikoodi 10 havainnollistaa kyseisen logiikan. Projektin pienen laajuuden vuoksi perinteinen tilojen hallinta oli ratkaisuna hyvä, mutta jos Monster-luokka tarvitsee enemmän tiloja ja monimutkaisuutta, koodin logiikan muokkaamisesta tulee vaivalloista, sillä tilojen vaihto tapahtuu eri puolilla Monster-luokan koodia.

```
switch (_myState)
{

     case MonsterState.Wait:
          WaitForPlayer();
          break;

     case MonsterState.Chase:
          ChasePlayer();
          break;

     case MonsterState.ResetChase:
          ResetPlayerChase();
          break;

     default:
          break;
}
```

Esimerkkikoodi 10. Yksinkertainen tilojen hallintaan liittyvä switch-lause.

Sen sijaan tilamallissa jokainen tila toteuttaa rajapinnan, jossa määritellään tilan elinkaaren eri vaiheet: aloitus, päivitys ja lopetus. Esimerkkikoodi 11 on tilamallin tilan rajapinnasta. Tilan aloitus suoritetaan, kun tila aktivoituu ensimmäisen kerran. Päivitys puolestaan suoritetaan jokaisella peliruudun päivityksellä ja tässä vaiheessa tarkistetaan lisäksi mahdolliset ehdot tilan vaihtamiseen

seuraavaan tilaan. Lopetus suoritetaan ennen kuin tila vaihdetaan seuraavaan tilaan. (12, s. 73.)

```
public interface IState
{
     public void Enter();

     public void Update();

     public void Exit();
}
```

Esimerkkikoodi 11. IState-rajapinta, joka toteutetaan jokaisessa tilamallin tilassa.

Tilamallia käytettäessä jokainen tila on omassa luokassaan, joka toteuttaa rajapinnan (12, s. 73). Projektissa muutettiin Monster-luokan tilojen hallinta noudattamaan tilamallia siirtämällä tilalogiikat uusiin tilaluokkiin. Yksi uusista tilaluokista havainnollistetaan esimerkkikoodissa 12, joka sisältää Chase-tilan logiikan.

```
public class ChaseState : IState
{

     private MonsterController monster;

     public ChaseState(Monster monster)
     {
          this.monster = monster;
     }

     public void Enter()
     {
          Debug.Log(“Entered Chase State”);

     }

     public void Update()
     {
          float step = monster.MovementSpeed * Time.deltaTime;
          monster.transform.position = Vector3.MoveTowards(mon-
ster.transform.position, monster.EndPoint.position, step);

           if (Vector3.Distance(monster.transform.position, mon-
          ster.EndPoint.position) < 0.001f)
         {
             monster.MyStateMachine.TransitionTo(monster.MyStateMa-
chine.resetChaseState);
         }
     }

     public void Exit()
     {
          monster.Anim.SetBool("Run Forward", false);
          monster.Anim.SetBool("Stunned Loop", true);

     }
}
```

Esimerkkikoodi 12. ChaseState-luokka, joka on tehty tilamallin mukaisesti.

Chase-tilan rakentaja määrittää MonsterController-muuttujan arvon, jonka kautta päästään kyseisen luokan muuttujiin. Enter-metodi ilmoittaa viestillä, että Chase-tila on käynnistetty. Update-metodissa liikutetaan objektia ja tarkastetaan sen etäisyyttä kohteesta. Kuvassa 9 näkyy MonsterController-luokkaa käyttävä objekti Chase-tilassa.

Kuva 9. MonsterController-luokkaa käyttävä karhuobjekti Chase-tilassa jahtaamassa pelaajahahmoa.

Kun objekti on saavuttanut kohteensa, MonsterController-luokan StateMachinelle kerrotaan, että on aika siirtyä seuraavaan tilaan. Ennen kuin tila muuttuu, Exit-metodissa vielä päivitetään MonsterControllerin animaatioiden arvoja. Muut tilat toteutetaan samalla tavalla kuin ChaseState-luokka, ja niiden hallintaan käytetään erillistä StateMachine-luokkaa, jonka toimintaa kuvaa esimerkkikoodi 13.

```
[Serializable]
public class StateMachine
{
     public IState CurrentState { get; private set; }

     public WaitState waitState;
     public ChaseState chaseState;
     public ResetChaseState resetChaseState;

     public StateMachine(MonsterController monster)
    {
        this.waitState = new WaitState(monster);
        this.chaseState = new ChaseState(monster);
        this.resetChaseState = new ResetChaseState(monster);
    }

     public void Initialize(IState startingState)
     {
          CurrentState = startingState;
          startingState.Enter();
     }

     public void TransitionTo(IState nextState)
     {
          CurrentState.Exit();
          CurrentState = nextState;
          nextState.Enter();
     }

     public void Update()
     {
          if (CurrentState != null)
          {
               CurrentState.Update();
          }
     }
};
```

Esimerkkikoodi 13. Esimerkki StateMachine-luokasta, jossa on MonsterController-luokan tilat.

StateMachine-luokka vastaa tilojen hallinnasta ja siirtymisistä. MonsterController-luokassa luodaan uusi StateMachine-instanssi ja annetaan sille viite kyseiseen MonsterController-luokkaan, jonka jälkeen kutsutaan StateMachinen Initialize-metodia. MonsterControllerin Update-metodi kutsuu StateMachinen Update-metodia.

Tilamallin etuna on selkeä rakenne, jossa jokaisella tilalla on oma vastuunsa ja toimintalogiikkansa (22). Siinä noudatetaan avoin-suljettu-periaatetta, sillä uusien tilojen lisääminen ei vaikuta olemassa oleviin itsenäisiin tiloihin, ja koodi

pysyy modulaarisena ja helposti ylläpidettävänä. Lisäksi tilat toimivat itsenäisesti toisistaan riippumatta, mikä helpottaa tilojen testaamista erikseen. (12, s. 76.)

On kuitenkin hyvä harkita tilamallin käyttöä vain silloin, kun odotetaan tilojen monimutkaisuuden kasvavan. Pienissä projekteissa tai yksinkertaisissa tilanteissa tilamallin käyttö voi olla liioiteltua ja lisää projektiin ylimääräistä rakennetta. Malli on hyödyllinen erityisesti silloin, kun tarvitaan selkeää ja joustavaa tapaa hallita monimutkaista tilakonetta. (12, s. 76.)

4.6 Havaintomalli

Havaintomalli (Observer Pattern) on suunnittelumalli, joka mahdollistaa väljän riippuvuuden kahden objektin välille. Siinä on yksi objekti eli subjekti tai julkaisija, joka ylläpitää listaa objekteista eli havaitsijat tai tilaajat, jotka riippuvat subjektista (23). Subjekti ilmoittaa automaattisesti havaitsijoille muutoksista, mikä on hyödyllistä tilanteissa, joissa halutaan, että objektit kommunikoivat keskenään ilman tiivistä riippuvuutta (16, luku 9. s. 5). Väljä riippuvuus objektien välillä vähentää koodin monimutkaisuutta.

Havaintomalli mahdollistaa monen komponentin reagoinnin yhteen subjektin tapahtumaan, mikä vähentää suoraa riippuvuutta ja luo yksi-moneen-riippuvuussuhteen objektien välille sekä mahdollistaa joustavan ja modulaarisen koodin. (24.)

Havaintomalli on laajalti käytetty suunnittelumalli, ja se on integroitu C#-kieleen. C#:n tapahtumat tarjoavat helpon tavan toteuttaa havaintomallia ilman tarvetta rakentaa omaa mekanismia sitä varten. C#:n tapahtuma on ilmoitus, joka kertoo, että jokin tietty tapahtuma on tapahtunut tai tila on muuttunut (24). Tapahtumat soveltuvat erinomaisesti monenlaisiin ohjelmointitehtäviin, kuten pelinkehitykseen Unity-pelimoottorissa.

Subjekti luo tapahtuman, toiminnan, jonka subjekti toteuttaa pelin aikana. Jokaisella havaitsijalla on metodi nimeltä tapahtumankäsittelijä, joka kuuntelee subjektin tapahtumaa. Kun subjekti toteuttaa tapahtuman, kaikki havaitsijat toteuttavat tapahtumaan liittyvän oman metodinsa. Subjektin tapahtumaa voi kuunnella kuinka monta havaitsijaa tahansa. Subjekti ei ole tietoinen havaitsijoista eivätkä havaitsijat toisistaan. (24.) Havaitsija voi myös poistaa itsensä tapahtumasta, jos halutaan, että se ei enää saa ilmoituksia siitä (16, luku 9 s. 5).

Projektiin tehtiin yksinkertainen tapahtuma havainnollistamaan tapahtuman toimintaa. Tapahtumassa on kaksi pääluokkaa: PlayerHealth ja UIManager. PlayerHealth-luokka toimii tapahtuman subjektina. PlayerHealth-luokassa on UpdateLives-tapahtuma, jota muut luokat voivat kuunnella. Kuvassa 10 havainnollistetaan, miltä elämän menetys näyttää projektin käyttöliittymässä.

Kuva 10. Pelaajan elämien määrän esitys käyttöliittymässä.

Kun TakeDamage-metodia kutsutaan ja pelaajan elämät vähenevät, kutsutaan UpdateLives-tapahtumaa, joka ilmoittaa kaikille havaitsijoille, että pelaajan elämien määrä on muuttunut. PlayerHealth-luokan toimintaa esitetään esimerkkikoodissa 14.

```
public class PlayerHealth : MonoBehavior
{
     public event Action UpdateLives;
     private int _health = 3;

     public void TakeDamage()
     {
          _health--;
          UpdateLives?.Invoke();
     }
}
```

Esimerkkikoodi 14. PlayerHealth-luokka havaintomallin subjektina.

UIManager-luokka puolestaan toimii tapahtuman havaitsijana, joka esitetään esimerkkikoodissa 15. Se ilmoittautuu PlayerHealth-luokan UpdateLives-tapahtuman kuuntelijaksi Start-metodissa ja peruuttaa kuuntelun OnDestroy-metodissa. Kun UpdateLives-tapahtuma laukaistaan PlayerHealth-luokassa, OnUpdateLives-metodi suoritetaan UIManager-luokassa päivittämään elämien määrä käyttöliittymässä.

```
public class UIManager : MonoBehavior
{

     [SerializeField] private PlayerHealth _health;


     private void Start()
     {
          if (_health != null)
          {
               _health.UpdateLives += OnUpdateLives;
          }
     }

     private void OnUpdateLives()
     {
          // Functionality to update health UI.
     }

     private void OnDestroy()
     {
          if (_health != null)
          {
               _health.UpdateLives -= OnUpdateLives;
          }
     }

}
```

Esimerkkikoodi 15. UIManager-luokka havaitsijamallin havaitsijana.

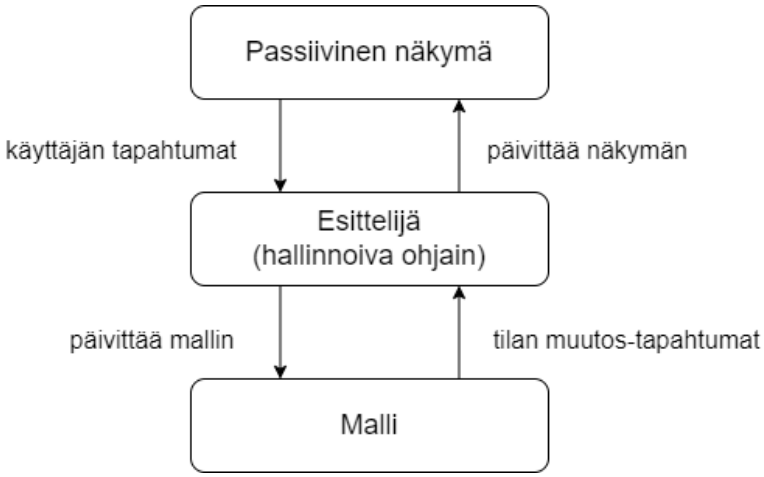
On tärkeää muistaa, että havaitsijat tarvitsevat viittauksen siihen luokkaan, joka määrittelee tapahtuman. Havaintomallin käyttö helpottaa testaamista ja virheiden etsintää, mutta se voi lisätä suorituskykyongelmia suurissa pelimaisemissa. Havaintomalli soveltuu erityisen hyvin käyttöliittymiin, koska se mahdollistaa pelilogiikan erottamisen käyttöliittymästä (23). Se on myös vahvasti liitoksissa MPV-malliin, jota käsitellään seuraavassa luvussa (12, s. 87).

4.7 Malli-näkymä-esittelijä

Malli-näkymä-esittelijä (Model-View-Presenter eli MVP) on suunnittelumalli, joka soveltuu erityisesti käyttöliittymien rakentamiseen ja auttaa erottamaan sovelluksen logiikan, datan ja käyttöliittymän toisistaan. MVP-mallissa sovellus jaetaan kolmeen tasoon: malli, näkymä ja esittelijä (25).

Malli vastaa datan hallinnasta, ja se sisältää tiedon sovelluksen tilasta. Malli ei suorita graafisia toimintoja tai pelilogiikkaa, vaan keskittyy pelkästään datan tallentamiseen ja muokkaamiseen. (25.) Näkymä näyttää datan graafisesti käyttäjälle. Unity-pelimoottorissa käytetään usein valmiita käyttöliittymäelementtejä, joten näkymä voi olla esimerkiksi ruudulla näkyvä käyttöliittymä. MVP-mallissa näkymä ei suoraan tarkkaile mallin muutoksia. (12, s. 88.) Esittelijä huolehtii kommunikaatiosta mallin ja näkymän välillä. Se hakee datan mallista ja muotoilee sen näkymään sopivaksi sekä käsittelee käyttäjän syötteet ja päivittää tarvittaessa mallia. Kun mallissa tapahtuu muutoksia, esittelijä saa tiedon tästä ja päivittää tarvittaessa näkymän vastaamaan uutta tilaa. (25.)

MPV-mallissa tapahtumat soveltuvat hyvin siihen, että käyttäjä tekee syötteen, josta näkymä on vastuussa. Kuva 11 esittää kaavion MVP-mallin toiminnasta. Siinä käyttäjä painaa nappia käyttöliittymässä, näkymä lähettää syötteen esittelijälle käyttöliittymätapahtuman kautta ja esittelijä vuorostaan muokkaa mallia. Tilan muutos mallissa kertoo esittelijälle, että dataa on muokattu. Esittelijä välittää muokatun datan näkymään, joka päivittää käyttöliittymän.



Kuva 11. MVP-mallin toiminta (26).

MVP-malli vähentää koodissa olevia riippuvuuksia ja lisää koodin luettavuutta, testattavuutta ja ylläpidettävyyttä. Se lisäksi helpottaa tiimityöskentelyä, koska käyttöliittymän kehittäjät voivat työskennellä lähes itsenäisesti muusta koodikannasta. Lisäksi se mahdollistaa yksikkötestauksen ilman pelitilan käynnistämistä, mikä säästää aikaa kehitysvaiheessa. (16, luku 10 s. 2.)

MVP-malli soveltuu parhaiten isoihin sovelluksiin ja projekteihin, joissa on monimutkainen arkkitehtuuri ja pitkä kehitysaika. MVP-mallin toteuttamiseksi luokat on jaettava vastuun mukaan, mikä vaatii paljon organisoitua työskentelyä ja etukäteistyötä. (12, s. 93)

On tärkeää huomioida, että kaikki Unity-projektin osat eivät välttämättä sovi suoraan MVP-malliin. Yksinkertaiset koodit tai komponentit, jotka eivät käsittele monimutkaista dataa tai logiikkaa, eivät välttämättä hyödy MVP-mallin käytöstä. (12 s. 94)

5 Työn tulokset

Projektista muodostui endless runner -peli nimeltä Cake Coated Cat, jossa juostaan kissana erilaisten maisemien läpi ja heitetään kummituksia kakuilla.

Pelin päävalikko näkyy kuvassa 12. Pelin asetuksissa pystytään muuttamaan äänenvoimakkuutta ja pelaajanimeä, joka näkyy tulostaulussa.

Kuva 12. Projektin päävalikko.

Juoksemisen ja kakkujen heittelyn lisäksi pelissä nähdään kolme erikoisominaisuutta. Ensimmäisenä ominaisuutena on seinähyppely, jossa pelaajan tarkoituksena on katsoa liikkumisen oikea järjestys seinien tukipalkeista, ja painaa sen mukaisesti joko Q- tai E-näppäintä. Seinähyppely havainnollistetaan kuvassa 13. Loiste indikoi seuraavan oikean näppäimen sijaintia.

Kuva 13. Pelaaja hyppimässä seinissä olevia tukipalkkeja pitkin.

Jokainen tukipalkki seinässä on Leap-luokan objekti, johon on määritelty siihen liittyvä näppäin ja funktio loisteen kytkemiseen päälle ja pois. LeapContainer-luokka sisältää tiedon kaikista seinissä olevista Leap-objekteista ja määrittää oikean näppäinjärjestyksen Leap-objektien sijaintien perusteella. Kun pelaaja saapuu alueelle, jossa voidaan hyppiä seiniä pitkin, aloitetaan hyppyjärjestys. Pelaajan täytyy painaa oikeaa näppäintä tarpeeksi nopeasti siirtyäkseen seuraavaan Leap-objektiin. Jos pelaaja painaa väärää näppäintä tai on liian hidas, katsotaan pelaajan epäonnistuneen hyppelyssä ja pelaaja putoaa.

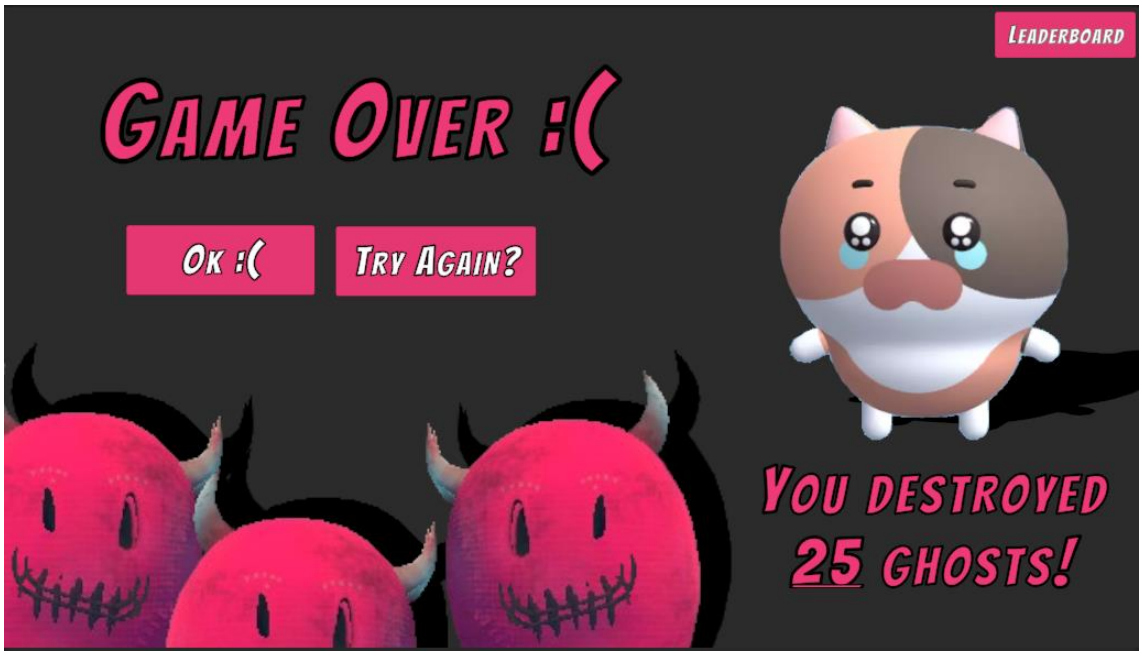
Toisena projektin ominaisuutena on pelaajan jahtaus. Kun pelaaja juoksee pelimaailmassa olevan karhun lähelle, karhun Chase-tila aktivoituu. Karhun toiminnallisuutta käsitellään tilamallista kertovassa luvussa (4.5). Pelaajalle lähetetään tieto karhun Chase-tilan aktivoitumisesta, mikä muuttaa pelaajan liikesuuntaa automaattisesti ja pelaaja lähtee juoksemaan karhua pakoon takaisin suuntaan, josta on alun perin tullut. Kun Chase-tila loppuu, tieto tilan

muutoksesta välitetään pelaajalle, ja pelaajan liikesuunta muuttuu takaisin normaaliksi.

Kolmantena ominaisuutena on kiipeily, jossa pelaajan liikesuunta muuttuu vertikaaliseksi. Tätä havainnollistetaan kuvassa 14. Kiipeilyn laukaisemiseksi käytetään törmäystarkistusta, joka kertoo pelaajalle, että suunta vaihdetaan vertikaaliseksi. Seinän huipulta tiputetaan pelaajalle objektivaraston objekteja esteiksi, jotka putoavat painovoiman avulla suoraan alaspäin. ObstacleDropper-luokka määrittää kyseisille objekteille sijainnin x-akselilla, minkä jälkeen se hakee objektivarastosta esteen. Hetken odotuksen jälkeen uusi este pudotetaan.

Kuva 14. Pelaaja kiipeämässä seinää pitkin.

Lopputuloksena pelistä muodostui pätevä kokonaisuus. Pelissä on monia erilaisia ominaisuuksia, jotka tuovat vaihtelua peliin, ja pistetaulukko on mukava lisä, jonka avulla päästään kisaamaan muita pelaajia vastaan. Kuvassa 15 näytetään vielä pelin loppunäkymä.



Kuva 15. Pelin loppunäkymä.

Halutessaan pelaaja voi yrittää peliä heti uudelleen tai tarkastella pistetaulukkoa. Pistetaulukon tietokannassa käytettiin Microsoftin Azure PlayFab-palvelua.

6 Yhteenveto

Monet asiat vaikuttavat koodin laatuun. Hyvät ohjelmointikäytännöt, suunnitteluperiaatteet ja suunnittelumallit ovat keskeisiä tekijöitä laadukkaan ja ylläpidettävän ohjelmiston kehittämisessä. Kuvaavien muuttujanimien käyttö, selkeä koodin rakenne ja yhdenmukainen tyyli parantavat koodin luettavuutta ja ylläpidettävyyttä (6). Suunnitteluperiaatteet, kuten KISS ja DRY, ohjaavat yksinkertaisiin ja tehokkaisiin ratkaisuihin. SOLID-periaatteet tarjoavat arvokkaita ohjeita koodin suunnitteluun ja toteutukseen (12).

Suunnittelumallit ovat muokattavia pohjakaavoja, jotka auttavat organisoimaan ja eriyttämään koodia eri vastuualueihin. Erityisesti havaintomalli ja MVP ovat hyödyllisiä käyttöliittymäpohjaisissa sovelluksissa, joissa tarvitaan selkeää erottelua datan, esityksen ja sovelluslogiikan välillä.

Näiden suunnitteluperiaatteiden ja -mallien yhteinen tavoite on parantaa ohjelmiston luettavuutta, ylläpidettävyyttä, joustavuutta ja laadukkuutta. Niiden noudattaminen auttaa kehittäjiä rakentamaan tehokkaita ja kestäviä ohjelmistoratkaisuja erilaisiin tarpeisiin ja haasteisin.

Projektin aikana kokeiltiin erilaisia suunnittelumalleja, kuten singleton-mallia, objektivarastoa, havaintomallia ja tilamallia, mutta huomattiin, että niiden kaikkien käyttö ei ollut tarpeen projektille, joka oli melko pienimuotoinen. Singleton- ja objektivarasto-mallit olivat hyödyllisiä tietyissä tilanteissa. Singleton-malli auttoi hallitsemaan yksittäisiä resursseja tehokkaasti ja antamaan niihin globaalin pääsyn. Objektivarasto puolestaan tarjosi hyödyllisen tavan luoda ja hallita objekteja, mikä olisi ollut myös helposti laajennettavissa tarpeen tullen. Kun singleton-malli yhdistettiin objektivarastoon, saatiin lisäksi helppo pääsy objektivaraston objekteihin.

Sen sijaan havaintomalli ja tilamalli eivät osoittautuneet tarpeellisiksi projektin kontekstissa. Havaintomallin käyttö lisäsi turhaa monimutkaisuutta ilman merkittävää lisäarvoa, sillä projekti ei vaatinut monen komponentin reagoimista yhteen tapahtumaan. Tilamallin tarjoamat tilan hallintakeinot olivat myös liioiteltuja tarpeisiin nähden, kun yksinkertainen lähestymistapa oli riittävä.

Lisäksi tehdasmallille, komentomallille ja MVP-mallille ei ollut käyttöä projektissa. Tehdasmallin olisi voinut yhdistää objektivarastoon, mutta sen monimutkaisen toteutustavan ja vähäisen objektimäärän takia yhdistäminen ei ollut tarpeellista.

Näiden havaintojen perusteella opittiin arvioimaan tarkemmin, milloin ja miten erilaisia suunnittelumalleja tulisi soveltaa. On tärkeää valita suunnittelumalli sen perusteella, miten hyvin se vastaa projektin tarpeita ja haasteita, ja välttää tarpeetonta monimutkaisuutta. Jokainen suunnittelumalli tarjoaa erilaisia etuja ja haittoja, ja suunnittelumallin valinnassa on tärkeää arvioida näitä seikkoja projektin vaatimusten näkökulmasta.

Ennen kuin valitsee suunnitteluperiaatteen tai -mallin, on tärkeää ymmärtää ongelma perusteellisesti. Kaikki ongelmat eivät välttämättä tarvitse suunnitteluperiaatteita tai -malleja, ja siksi vaatimusten, rajoitusten ja tilanteeseen liittyvien asioiden perusteellinen analysointi auttaa valitsemaan parhaan ratkaisun. Lisäksi on tärkeää olla tietoinen valitun mallin eduista ja haitoista sekä niiden vaikutuksesta suorituskykyyn, koodin luettavuuteen ja testattavuuteen (8).

Kun suunnittelumallit ja -periaatteet valitaan oikein ja niitä käytetään sopivissa tilanteissa, ne voivat nopeuttaa työnkulkua ja tarjota elegantteja ratkaisuja, jotka parantavat ohjelmiston kestävyyttä ja laadukkuutta. Tällöin ei tarvitse keksiä pyörää uudestaan, vaan voidaan hyödyntää jo olemassa olevia parhaita käytäntöjä ja ratkaisumalleja.

Lähteet

1 Common C# code conventions. Verkkoaineisto. Microsoft. <learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>. Luettu 7.4.2024.

2 Understanding Coding Conventions in Software Engineering. Verkko-aineisto. Institute of Data. <https://www.institutedata.com/blog/software-engineering-coding-conventions/>. Luettu 8.4.2024.

3 Kernighan, Brian W. & Pike, Rob. 1999. The Practice of Programming. E-kirja. Addison-Wesley Professional.

4 C# identifier naming rules and conventions. Verkkoaineisto. Microsoft. <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names?source=recommendations>. Luettu 7.4.2024.

5 Snake Case VS Camel Case VS Pascal Case VS Kebab Case – What's the Difference Between Casings? Verkkoaineisto. Free Code Camp. <https://www.freecodecamp.org/news/snake-case-vs-camel-case-vs-pascal-case-vs-kebab-case-whats-the-difference/>. Luettu 9.4.2024.

6 Unity. 2022. Create a C# style guide: Writing Cleaner Code That Scales. E-kirja. Unity.com.

7 Coding guidelines. Verkkoaineisto. Microsoft. <https://learn.microsoft.com/en-us/mixed-reality/world-locking-tools/documentation/howtos/codingconventions?source=recommendations>. Luettu 7.4.2024.

8 How can software developers use design patterns and principles effectively? Verkkoaineisto. LinkedIn. <https://www.linkedin.com/advice/0/how-can-software-developers-use-design-patterns-principles-jmxwc>. Luettu 8.4.2024.

9 Martin, Robert C. & Martin, Micah. 2007. Agile principles, patterns and practises in C#. E-kirja. Prentice Hall.

10 Chiarelli, Andrea. 2016. Mastering JavaScript object-oriented programming: unleash the true power of JavaScript by mastering Object-Oriented programming principles and patterns. E-kirja. Packt Publishing 2016.

11 Keep It Simple, Stupid (KISS). Verkkoaineisto. Interaction Design Foundation. <https://www.interaction-design.org/literature/topics/keep-it-simple-stupid>. Luettu 6.4.2024.

12 Unity. 2021. Level up your programming with game programming patterns. E-kirja. Unity.com.

13 What is DRY Development? Verkkoaineisto. Digital Ocean. <https://www.digitalocean.com/community/tutorials/what-is-dry-development>. Luettu 7.4.2024.

14 Difference between Abstract Class and Interface in C#. Verkkoaineisto. <https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-c-sharp/>. Luettu 14.4.2024.

15 Holzner, Steve. 2006. Design Patterns for Dummies. E-kirja. Wiley.

16 Osmani, Addy; Romano, Robert & Demarest, Rebecca. 2012. Learning Java-Script design patterns. E-kirja. O'Reilly.

17 C# Factory Method Design Pattern. Verkkoaineisto. <https://www.dofactory.com/net/factory-method-design-pattern>. Luettu. 14.4.2024.

18 Understanding the Factory Pattern in C# - With Examples. Verkkoaineisto. <https://hackernoon.com/understanding-the-factory-pattern-in-c-with-examples>. Luettu 12.4.2024.

19 What is object pooling in C#? Verkkoaineisto. <https://www.educative.io/answers/what-is-object-pooling-in-c-sharp>. Luettu 12.4.2024

20 Eduard Ghergu. 27.06.2023. Singleton Design Pattern. Verkkoaineisto. <https://www.pentalog.com/blog/design-patterns/singleton-design-pattern/>. Luettu 11.4.2024.

21 Command Pattern in C#: From Basics to Advanced. Verkkoaineisto. <https://medium.com/@lexitrainerph/command-pattern-in-c-from-basics-to-advanced-29d954cafb92#:~:text=The%20Command%20Pattern%20is%20a,it%2C%20and%20support%20undoable%20operations.>. Luettu 9.4.2024.

22 Jignesh Trivedi. 2013. State Design Pattern. Verkkoaineisto. <https://www.c-sharpcorner.com/UploadFile/ff2f08/state-design-pattern/>. 06.08.2013. Luettu 10.4.2024

23 Observer design pattern. 2023. Verkkoaineisto. < https://learn.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern/>. 25.5.2023. Luettu 14.4.2024.

24 Handle and raise events. 2022. Verkkoaineisto. < https://learn.microsoft.com/en-us/dotnet/standard/events/>. 4.10.2022. Luettu 14.4.2024.

25 MVP (Model View Presenter) Architecture Pattern in Android with Example. 2020. Verkkoaineisto. <https://www.geeksforgeeks.org/mvp-model-view-presenter-architecture-pattern-in-android-with-example/>. 29.10.2020. Luettu 15.4.2024.

26 MVP-mallin kuva. Verkkoaineisto. Wikipedia. <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter#/media/File:Model_View_Presenter_GUI_Design_Pattern.png>. Luettu 12.4.2024.