Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Andrei Misukka

Suunnittelumallien käyttö peleissä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

1.6.2019

Tekijä Otsikko Sivumäärä Aika	Andrei Misukka Suunnittelumallien käyttö peleissä 43 sivua + 1 liite 1.6.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaaja	Lehtori Miikka Mäki-Uuro

Insinöörityössä tutkittiin yleensä ohjelmistotuotannossa käytettävien suunnittelumallien käyttömahdollisuuksia peleissä. Tarkoituksena oli tutkia suunnittelumalleja yleisesti, mutta myös erityisesti pelien näkökulmasta.

Suunnittelumallit ovat erityisesti ohjelmistotuotannossa käytettäviä, yleisesti tiedettyihin ohjelmakoodin suunnitteluun liittyviin ongelmiin kehitettyjä, abstrakteja malleja.

Tutkimuksessa oli tarkoituksena selvittää, voiko suunnittelumalleilla saavuttaa merkittävää hyötyä pelien kehityksessä. Työssä käytettiin Unity-pelimoottorilla tuotettua avoimen lähdekoodin peliä, jonka koodirakennetta paikoin muokattiin käyttäen eri suunnittelumalleja. Tuloksia arvioitiin seuraavasti: koodin luettavuus ja uudelleen muokkaamisen helppous sekä muistinhallinnan ja suorituskyvyn parantuminen. Saatuja tuloksia analysoitiin kriittisesti ja arvioitiin sen mukaan, kuinka paljon suunnittelumalli käytännössä ongelmaa helpotti.

Insinöörityön tuloksena apuna käytettyyn peliin saatiin lopulta toteutettua kolme eri suunnittelumallia. Käytetyt mallit olivat Komento, Strategia ja Oliovarasto. Kaikki kolme mallia ratkaisevat hyvin erilaisia ongelmia. Kaksi käytetyistä malleista luokitellaan käyttäytymismalleihin ja kolmas erityisesti peleissä käytettäväksi.

Projektin tulokset olivat yleisesti odotettuja. Verrattuna alkuperäisen pelin ohjelmakoodiin voitiin todeta käytettyjen mallien lisäävän koodin luettavuutta ja helpottavan uusien ominaisuuksien lisäämistä. Suorituskykyä ja muistinhallintaa pystyttiin parantamaan merkittävästi vain vähemmän tehokkailla tietokoneilla.

Projektista saadut kokemukset näyttivät suunnittelumallien käytön sopivan myös peleihin. Olemassa olevan koodin muokkaus vastaamaan mallin rakennetta ei kuitenkaan aina ilmene merkittävänä hyötynä, vaan saattaa aiheuttaa tarpeetonta monimutkaisuutta. Projektin kulku osoitti, että aluksi tulisi perusteellisesti selvittää suunnittelumallilla saavutettavaa etua.

Avainsanat	suunnittelumallit, Unity, ohjelmistoarkkitehtuuri

Author Title Number of Pages Date	Andrei Misukka Utilization of design patterns in games 43 pages + 1 appendix 1 June 2019
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Game Applications
Instructor	Miikka Mäki-Uuro, Senior Lecturer

The purpose of this final year project was to research and collect data about the feasibility of design patterns in game developing. The goals were to determine how the design patterns are used in games and game engines, in addition with how the design patterns could help the building of the code architecture of a game.

The design patterns are described as abstract blueprints for solving commonly occurring design problems inside the code. The usage of design patterns is focused on software development, but the research was done with focus on games. The researching consisted of studying the Unity game engine and an open source game produced by it. Furthermore, the game's own code was refactored for implementation of three different design patterns. The effects and consequences of the refactoring were critically evaluated by inspecting the changes for the code's readability, reusability, memory management and performance.

As a result, Unity was concluded to have considerable amount of design patterns used all around the engine. Many of them were found to be related to games specifically. The three implemented patterns yielded results that were mostly expected: both readability and reusability increased. However, the change in memory management and performance remained lower than first estimated.

In conclusion, the design patterns are widely used in games and some patterns even focus on increasing the performance. Moreover, they can have positive impact on the code architecture of a game, but for the usage there needs to be a careful measure of the advantages and disadvantages.

Keywords	design patterns, Unity, software architecture

Sisällys

1 Johdanto 1

2 Oliopohjaisen ohjelmakoodin suunnittelu 2

2.1 Ohjelmoinnin haasteet 2
2.2 Hyvä ohjelmakoodi ja refaktorointi 3
2.3 Refaktorointi käytännössä 4
2.4 Sovelluksen arkkitehtuurin suunnittelu 7
2.5 Suunnittelumallit 12
2.6 Suunnittelumallit peleissä 15
2.7 Suunnittelumallien kritiikki 19

3 Suunnittelumallien tutkiminen ja toteuttaminen 20

3.1 Suunnittelumallit Unityssa 20
3.2 Suunnittelumallien toteutus 25
3.2.1 Komento-malli 26
3.2.2 Oliovarasto-malli 29
3.2.3 Strategia-malli 33
3.3 Tutkimusten ja oman työn arviointi 37

4 Yhteenveto 39

Lähteet 41

Liitteet

Liite 1. Vuokaavio skriptin elinkaaresta Unityssa

1 Johdanto

Tietotekniikan alan yritykset käyttävät vuosittain huomattavan osan vuosibudjetistaan tuotekehitykseen ja vanhojen tuotteidensa ylläpitoon. Jatkuvasti muuttuvassa maailmassa ohjelmistot tarvitsevat hyvin useasti ylläpitämistä julkaisunsa jälkeen. Tekniikka kehittyy ja kilpailu ajaa yritykset etsimään tehokkaampia keinoja maksimoida tuotto ja minimoida virheet. Tämä luo IT-yrityksille painetta suunnitella tuotteensa mahdollisimman hyvin, jotta vältyttäisiin aikaa ja rahaa vaativilta jälkikorjauksilta. Motivaatio hyvään suunnitteluun on siis taloudellinen. Jos ohjelmistot suunnitellaan huonosti, täytyy yrityksen käyttää myös enemmän resursseja, kun lisätään uusia ominaisuuksia tai parannetaan vanhoja. Mahdollisimman tehokas ohjelmistojen tuottaminen ja ylläpitäminen ovat siis jokaisen tietotekniikan alan yrityksen elinehto.

Insinöörityössä tutkittiin ohjelmiston koodirakenteen suunnittelussa hyvin yleisesti käytettyä tapaa ratkaista ongelmat niin, että tuloksena oleva koodi on mahdollisimman yleiskäyttöistä ja vaihtuviin tilanteisiin mukautuvaa. Suunnittelumallit kuvaavat ohjelmakoodin tuotantoympäristössä aina uudelleen ilmeneviä ongelmia. Ne pyrkivät vastaamaan kysymyksiin, mikä ongelma on kyseessä, mikä ratkaisu ongelmaan sopii ja mitkä ovat tämän ratkaisun vaikutukset ja seuraukset ohjelmakoodille.

Projektissa tutkittiin, miten näitä malleja voisi hyödyntää myös pelituotannossa. Lisäksi tutkittiin hyvän ohjelmakoodin suunnittelun periaatteita. Projektin aikana tutustuttiin eri suunnittelumallien käyttötarkoituksiin ja tutkittiin käytännössä, miten Unity-pelimoottori ja sillä tuotettu Arenagame-videopeli hyödynsivät suunnittelumalleja. Tarkoituksena oli löytää peleille tyypillisiä ongelmakohtia, joiden ratkaisemiseen voisi käyttää suunnittelumallia. Pelialan yritysten tavoitteet hyvin suunnitellun ohjelmakoodin saavuttamiseksi ovat luonnollisesti samat kuin kaikilla IT-alan yrityksillä.

Hyvän koodirakenteen suunnittelu käyttäen oliopohjaista ohjelmointikieltä on aina vaikeaa. Ohjelmoijan tavoite on löytää ongelmaan yksinkertainen ratkaisu, jotta pystytään välttämään tai vähintään minimoimaan tulevaisuudessa ilmenevät haasteet. Ainoastaan kokemuksen tuoma taito auttaa luomaan ohjelmakoodista hyvän, mutta

suunnittelumalleja hyödyntämällä myös vähemmän kokeneet ohjelmoijat saavat uusia näkökulmia lähestyä haasteita, ja näin he oppivat suunnittelemaan ohjelmansa paremmin.

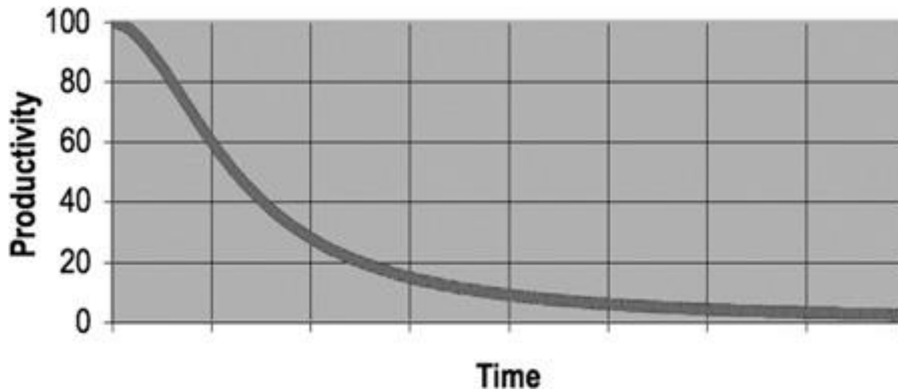
Tämän insinöörityön toisessa luvussa keskitytään hyvän ohjelmakoodin periaatteisiin teoreettisesta näkökulmasta: mitä tarkoittaa koodin refaktoroiminen, minkälaisia asioita kokeneemmat ohjelmoijat pitävät hyvän koodin mittapuuna ja mitä ovat suunnittelumallit. Kolmannessa luvussa esitellään käytännön vaiheen aikana tehdyt tutkimukset suunnittelumallien käytöstä Unity-pelimoottorissa sekä tulokset kolmen suunnittelumallin käyttämisestä Unitylla tuotettuun peliin.

2 Oliopohjaisen ohjelmakoodin suunnittelu

Hyvän ohjelmakoodin rakentamista verrataan usein talon rakentamiseen. Talo ei tarvitse kuin seinät, lattian ja katon. Huonosti tehty suunnittelu ja väärin rakennetut perustukset altistavat talon sortumiselle. Samaa periaatetta noudattaen ohjelmoijat pyrkivät välttämään mahdollisen uudelleenrakentamisen suunnittelemalla ohjelmansa kestämään. Kestävän talon voi oppia rakentamaan rakentamalla sortuneen talon aina uudelleen, mutta ohjelmoijan kannattaa tukeutua jo valmiiksi yleisesti hyväksyttyihin suunnittelua helpottaviin periaatteisiin. Tässä luvussa käsitellään näitä hyvän ohjelmankoodin rakentamista ja suunnittelua helpottavia elementtejä.

2.1 Ohjelmoinnin haasteet

Ohjelmoijien haasteet eivät rajoitu yhteen työpaikkaan, kaupunkiin tai valtioon. Kaikki ammatikseen ohjelmointia tekevät kohtaavat opiskellessaan tai työssään ongelmia, jotka ovat tuttuja kaikille mitä hyvänsä ohjelmointia tekeville henkilöille. Toisinaan haasteena voi olla esimerkiksi kollegan kirjoittaman koodin ymmärtäminen. Jo valmiiksi huonoa ohjelmakoodia on hankalaa lukea. Jokainen korjausyritys voi vain rikkoa toiminnallisuutta toisessa osassa koodia. Ongelmat kasaantuvat, ja työtahti hidastuu. Lopulta epäjärjestys koodissa kasvaa liian suureksi siivota, ja edistys pysähtyy lähes täysin, kuten kuva 1 havainnollistaa. [1, s. 4.]



Kuva 1. Robert Martinin esimerkki tuottavuuden laskemisesta ajan funktiona, kun ohjelmakoodin epäjärjestys kasvaa. Kriittisen rajan ylittyessä tuottavuus on lähes nolla. [1, s. 4.]

Tuotannon täydellinen pysähtyminen ohjelmakoodin siivottomuuden vuoksi on luonnollisesti epätoivottu lopputulos. Järjestelmällisesti ja hyvin ylläpidetty ohjelmakoodi lisää tuotetun ohjelman elinkaarta huomattavasti. Kokeneet ohjelmoijat määrittelevät hyvin kirjoitetun koodin ”elegantiksi”, ”helppolukuiseksi” ja ”yksiselitteiseksi”. Helppolukuisen ja elegantin lähdekoodin saavuttaminen ei ole varsinkaan aloittelevalle ohjelmoijalle yksiselitteistä. Oliopohjaisen ohjelmointikielen kirjoittamiseen on olemassa yhtä monta eri tapaa kuin on ohjelmoijia. [1, s. 7–13.]

Ohjelmoijat eivät kuitenkaan ensisijaisesti kirjoita koodia hyvän näköiseksi ja hyvällä tavalla, koska näin täytyisi tehdä. Ohjelmointikielen tulkki ei välitä siitä, millä tavalla koodi on kirjoitettu, jos siinä ei ole sääntöjen vastaisia virheitä. Ohjelma voidaan suorittaa ilman mitään ongelmia. Vain ohjelman kirjoittaneet ja sen ohjelmakoodia lukevat ihmiset välittävät siitä, millä tavalla koodi esitetään. Vasta siinä vaiheessa, kun ohjelmakoodiin halutaan tehdä muutoksia, ihminen joutuu tulkitsijan rooliin. Martin Fowler [2] esittää asian mielestäni hyvin: ”Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

2.2 Hyvä ohjelmakoodi ja refaktorointi

Jotta aloittelevat ohjelmoijat välttyisivät toistamasta samoja virheitä kuin heitä monta kertaa kokeneemmat ammattilaiset, elegantin lähdekoodin saavuttamiseksi täytyy sekä nähdä vaivaa että käyttää aikaa. Hyvin suunniteltu lähdekoodi valmistuu noudattamalla

tiettyjä hyviksi havaittuja standardeja, joita noudattamalla ohjelmoija vähentää paitsi tarvetta ajatella toteutusta uudelleen, myös virheiden ilmenemisen mahdollisuutta.

Olemassa olevan ohjelmakoodin muuttamisesta parempaan muotoon samalla kuitenkin säilyttäen sen toiminnallisuuden käytetään nimitystä refaktorointi. Ideaali ohjelmakoodin suunnittelu lähtee ajatuksesta, että refaktorointia ei tarvitsisi tehdä. Paraskaan suunnittelu ei voi kuitenkaan aina ennakoida tulevaisuutta. Refaktorointi muuttaa alkuperäistä ohjelmakoodia ja muuttaa samalla alkuperäistä suunnitelmaa. Refaktorointia ei kuitenkaan tule ajatella epäonnistumisena ohjelmakoodin suunnittelussa. Suunnitelmat voivat muuttua nopeasti, ja jotkin suunnitelman osa-alueet eivät ole selviä, kunnes tuotannossa on edistytty riittävän pitkälle. [2, s. 46.]

Äärimmäisyyksiin vietäessä refaktoroinnilla korvataan suunnittelu täysin. Toisinaan ohjelmoijat eivät aina itse tiedä, kuinka tietyt ongelmat ratkeavat. Tällöin voidaan nopeasti rakentaa useampia ratkaisuvaihtoehtoja miettimättä liikaa suunnittelua ja ohjelmakoodin ulkonäköä. Kun ohjelma saadaan toimimaan ilman ongelmia, voidaan nopeasti tehty koodin osuus refaktoroida parempaan muotoon. Vaikka tämä malli toimii, se ei ole pitkällä aikavälillä kannattava. Jopa vain nopeasti rakennettu koodi tarvitsee hieman suunnittelua. Mutta jälkeenpäin tehty refaktorointi on aina kalliimpaa ajassa mitattuna kuin koodin rakentamisen aikana tehty. [2, s. 56–57.]

2.3 Refaktorointi käytännössä

Refaktorointia tulisi tehdä oikeastaan koko ajan, jos mahdollista. Hyvänä periaatteena Martin Fowler [2, s. 49] määrittelee kolmen kohdan säännön:

1. Kun teet jotain ensimmäisen kerran, tee se.
2. Toisella kerralla, jos teet saman asian, mieti hetki toistuvaa koodia, mutta lisää se silti.
3. Jos teet saman asian kolmannen kerran, refaktoroi.

Vaikka tämä sääntö on jo itsessään hyvä, Fowlerin mukaan refaktorointia tulisi toteuttaa myös silloin, kun ohjelmaan lisätään toiminnallisuutta aliohjelmilla tai korjataan ohjelmointivirheitä. [2, s. 50.]

Koska refaktorointi pyrkii ehkäisemään sellaiset tilanteet, joissa ohjelmoija joutuu käyttämään paljon aikaa vaikean ohjelmakoodin muokkaamiseen, voi ohjelmoija ajatella tekevänsä kahta työtä. Kent Beck määrittelee nämä työt refaktoroinnille ”kahden hatun periaatteella”. Kun ohjelmoija lisää lähdekoodiinsa lisää toiminnallisuutta, hän tekee sen alkuperäisen suunnitelman mukaan. Kun lisätty osuus todetaan toimivaksi, muuttuu työn luonne tämän periaatteen mukaan erilaiseksi. Tässä vaiheessa aloitetaan lisätyn toiminnallisuuden refaktorointi, ja tarpeen mukaan voidaan lähdekoodia järjestää uudelleen. Näiden kahden roolin ajoittainen vaihtaminen takaa ohjelmakoodin toimivan paremmin, kun rakennetta ylläpidetään mahdollisimman usein toiminnallisuuden lisäämisen välillä. [2, s. 47.]

Fowler [2, s. 63] määrittelee refaktorointia tarvitsevan koodin ”haisevan” (engl. smell). Aloitteleva ohjelmoija ei pysty tunnistamaan suoraan haisevaa ohjelmakoodia, mutta tiettyjä merkkejä seuraamalla niiden huomaaminen helpottuu. Siinä missä Fowler puhuu jo olemassa olevan koodin refaktoroinnista, ovat Robert Martinin [1, s. 285] määrittelyt hyville koodin esitystavoille hyvin samankaltaista, mutta tarkoituksena on suunnitella koodi jo valmiiksi hyvin. Refaktorointi tehdään kuitenkin vasta toiminnallisuuden lisäämisen jälkeen, joten Martin argumentoi hyvän suunnittelun tapahtuvan jo ohjelmointivaiheessa. Mutta hän lisää, että oikein ohjelmoiminen heti ensimmäisellä kerralla on lähes myytti. Sen sijaan tulisi yhtenä päivänä keskittyä ohjelmoimaan sen hetken suunnitelman mukaan, refaktoroida, ja jatkaa taas huomenna. Jotta ohjelmoija ymmärtäisi paremmin, miksi suunnittelua tarvitaan, esittelen muutamia sekä Fowlerin että Martinin esittelemiä periaatteita siihen, missä kohdissa ohjelmakoodin voidaan ajatella ”haisevan”. [1, s. 158; 2, s. 63.]

Merkitykselliset nimet

Olio-ohjelmoinnissa mm. muuttujat ja metodit tarvitsevat nimen, jotta niitä voidaan referoida ja käyttää muualla lähdekoodissa. Hyvä nimeämistapa lähtee ajatuksesta, että oliot tulisi nimetä sen tehtävän mukaan, johon ne on luotu. Nimeäminen tapahtuu ajattelemalla, että lukijana on toinen ihminen eikä kone. Pahimmassa tapauksessa oliot on nimetty aivan päinvastaisesti, kuin niiden käyttötapa todellisuudessa on. Parhaassa tapauksessa hyvin nimetty metodi ei tarvitse ohjelmoijan tekemiä kommentteja laisinkaan, kun metodin käyttötapa on selkeä jo nimen perusteella. Lyhyesti sanottuna:

nimeämisessä tulee välttää antamasta väärää informaatiota lukijalle. Toinen tärkeä asia nimeämisessä on tehdä selkeä ero nimettyjen olioiden välille. Hyvin samalla tapaa nimetyt oliot eivät lisää informaatiota, vaikka ne olisi nimetty merkityksellisesti. [1, s. 17; 2, s. 52.]

Metodien suunnittelu

Metodeiksi kutsutaan ohjelmakoodissa esiintyviä aliohjelmia, jotka ovat itsenäisiä osia ja sisältävät toiminnallisuutta, joilla koko ohjelma toimii. Paitsi että metodit kuuluu nimetä merkityksellisesti, on ohjelmoijan osattava rakentaa metodinsa toimimaan järkevästi. Yksi sanonta nousee erityisesti ylitse muiden metodien hallinnassa: pieni on kaunista. Lukija voi helposti kadottaa satoja rivejä pitkän metodin tarkoituksen jo kauan ennen sen loppua. Myös virheiden löytäminen voi olla hankalaa. Pienempiä metodeja on helpompi refaktoroida kuin vain yhtä isoa. Fowlerin mukaan kaikkein pisimpään elävät sellaiset olio-ohjelmat, jotka käyttävät pieniä metodeja. Sata riviä pitkä metodi kannattaa jo jakaa pienempiin, helpommin ymmärrettäviin kokonaisuuksiin. Martinin mukaan metodin ei hyvin usein tarvitse olla kuin enintään 20 riviä pitkä. [1, s. 31; 2, s. 64.]

Toistuvat koodirakenteet

Jos ohjelmoija huomaa kahdessa kohdassa lähdekoodia täysin samalla tavalla esitettyä toiminnallisuutta, on hyvin mahdollista, että ohjelma toimii paremmin, jos nämä yhdistetään. Kahteen kertaan kirjoitetulle ohjelmakoodille ei tulisi olla mitään perustetta. Dave Thomas ja Andy Hunt [3, s. 27] esittävät, että jokaiselle kohdalle ohjelmassa tulisi olla yksi ja vain yksi tarkoitus. Heidän mukaansa DRY (Don’t repeat yourself) -periaate on ainoa tapa varmistaa sovelluksen luotettava toimivuus ja se, että tuotantovaihe ei kärsi jälkikäteen tehtävästä refaktoroinnista. Fowler määritteli kolmen kohdan säännössään toistuvan informaation olevan selvä merkki refaktoroinnin tarpeesta. Toisinaan ohjelmoijat ovat pakotettuja lisäämään samanlaista informaatiota sisältäviä elementtejä ohjelmaansa, koska yritysten osakkaat vaativat nähdä tuloksia tai projektin aikataulut ovat liian tiukkoja. Fowlerin sääntö lisättynä DRY-periaatteeseen osoittaa kuitenkin toistuvan koodin lisäävän ongelmia useammin kuin poistavan niitä. [1, s. 173; 2, s. 63.]

Dokumentointi

Ohjelmoijia ohjeistetaan aina dokumentoimaan ohjelmakoodiaan myöhempää käyttöä varten. Dokumentoinnin tarkoituksena on välittää lukijalle informaatiota, joka ei suoraan ole nähtävissä ohjelmakoodista. Dokumentaatiosta voi olla työskentelyssä suurta apua, mutta DRY-periaatteen mukaan dokumentaatio tulee aina pitää korkealla tasolla ja jättää matalan tason informaatio ohjelmakoodin välitettäväksi. Käytännössä tämä tarkoittaa esimeriksi oikein nimettyjä olioita ja hyvin suunniteltuja metodeja. Muuten lisätään toistuvaa informaatiota koodiin, joka ei sitä tarvitse. Milloin tahansa dokumentaatiota kirjoittaessa tulee ennen kaikkea miettiä, voiko tiedon välittää muulla keinolla. ”Haisevan” ohjelmakoodin pystyy tunnistamaan helposti, jos asian selittämiseksi tarvitaan suuri määrä dokumentaatiota. Väliaikainen dokumentaatio voi olla tarpeen, kun ohjelman rakenteen suunnittelu ei ole vielä valmis. [1, s. 53; 2, s. 71.]

Toistuvat muutokset samassa paikassa

Ohjelmakoodi pyritään aina suunnittelemaan niin, että jo kirjoitettua toiminnallisuutta ei tarvitse muokata erilaiseksi myöhemmin. Muutokset ovat mahdollisia hyvästä suunnittelusta huolimatta. On tärkeää kuitenkin huomata ero tavallisen ohjelmakoodin muokkaamisen ja refaktoroinnin välille. Refaktoroinnin ideana ei ole muuttaa ohjelmakoodin toiminnallisuutta, mutta jos ohjelmoija huomaa muokkaavansa ohjelman tiettyä toiminnallisuutta jokaisen muualle lisätyn koodiin jälkeen, on suunnittelussa tapahtunut virhe. Vastaavasti, jos lähdekoodiin lisätään toiminnallisuutta, joka aiheuttaa tarvetta korjata pieni osa monesta paikasta, on refaktoroinnille tarvetta. [2, s. 66; 4.]

2.4 Sovelluksen arkkitehtuurin suunnittelu

Kokeneet, olio-ohjelmointia työkseen tekevät henkilöt ovat yleensä parhaita sovelluksen ohjelmistokoodin suunnittelijoita. Toisaalta vähemmän kokeneet ohjelmoijat saattavat pelästyä lukuisten eri vaihtoehtojen määrää ja valita näin huonoimman suunnittelupolun. Hyväksi suunnittelijaksi noviisi voi oppia vain kokemuksen tuoman taidon kautta. Vähemmän kokeneen ohjelmoijan erottaa kokeneesta erityisesti tapa, jolla kokeneemmat lähestyvät ongelmia ohjelmoidessaan. Jos ongelmaan on jo olemassa hyväksi havaittu tapa toimia, on se varmasti käyttökelpoinen myös tulevaisuudessa. Pyörän keksiminen

uudelleen on väärä tapa lähestyä uuden ohjelmiston suunnittelua. DRY-periaate esimerkiksi antaa neuvon siihen, minkälaista ohjelmakoodia tulisi kirjoittaa. Seuraamalla erityisesti suunnittelua varten kehitettyjä periaatteita voidaan ymmärtää paremmin myös suunnittelumalleja. Tässä luvussa esitellään aluksi viisi suosittua oliopohjaisen ohjelmoinnin periaatetta, jotka yhdessä muodostavat SOLID-periaatekokoelman. Robert Martinin [5] ensimmäisenä esittelemä kokoelma on kerännyt inspiraationsa usealta eri henkilöiltä. SOLID-periaatteiden jälkeen esitellään kaksi Eric Gamman ym. [6, s. 17–20] esittämää periaatetta olioiden ja rajapintojen hyödyntämisestä koodin arkkitehtuurissa.

Single Responsibility -periaate

Vähentämällä ohjelman luokille ja aliohjelmille asetettua vastuuta ohjelman suorittamisesta voidaan vähentää myös tarvetta suurille muutoksille. Single Responsibility -periaate esittää, että kaikilla ohjelmakoodin eri alueilla, kuten luokilla ja aliohjelmilla, tulisi olla vain yksi määritelty vastuu. Enemmän vastuuta tarkoittaa potentiaalisempaa uhkaa muutoksille, jotka puolestaan voivat aiheuttaa virheitä. Huolehtimalla vastuun tasaisesta jakaantumisesta ohjelmoija tekee koodista sietokykyisemmän potentiaalisille muutoksille. [7; 8.]

Open–closed-periaate

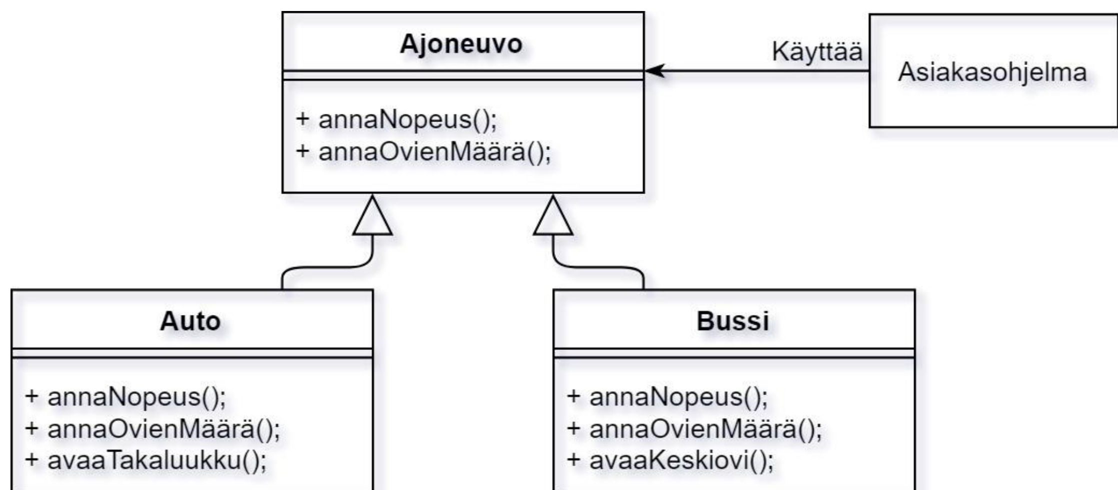
Robert C. Martin [5, s. 4] kuvaa Open-closed-periaatteen olevan tärkein ohjelmakoodin suunnittelussa muistettava ohjenuora. Perimmäinen ajatus siinä on, että ohjelmistokoodin luokkien täytyisi olla avoinna toiminnallisuuden lisäämiselle, mutta suljettuna muutokselle. Vaikka periaate kuulostaa ristiriitaiselta, on sille olemassa selitys. Olemalla avoinna toiminnallisuuden lisäämiselle tarkoittaa, että luokkaan voidaan vapaasti lisätä koodia. Vastaavasti olemalla suljettuna tarkoittaa sitä, että luokan jo olemassa olevaan toiminnallisuuteen ei kosketa. Ristiriidoilta vältytään käyttämällä hyväksi olio-ohjelmoinnissa tuttuja tekniikoita, kuten dynaamista polymorfismia ja template-malleja. [8.]

The Liskov Substitution -periaate

Barbara Liskov [5] on määritellyt The Liskov Substitution -periaatteen seuraavalla tavalla:

> Let $\Phi(x)$ be a property provable about objects *x* of type T. Then $\Phi(y)$ should be true for objects *y* of type S where S is a subtype of T,

jossa Φ on metodi, joka käyttää tyypin T muuttujaa x. Jos S on T:n alityyppi, on Φ:n oltava mahdollista käyttää myös tyypin S muuttujaa y. Toisin sanoen tämä periaate tarkoittaa sitä, että jokaisen T-luokan olion tulee olla korvattavissa luokan S olioilla. Vaikeasti esitetty käsite voidaan yksinkertaistaa ajatukseen, että ohjelman tulee toimia myös tilanteessa, jossa yliluokan olio korvataan aliluokan oliolla. Ohjelman suunnittelu The Liskov Substitution -periaatetta noudattaen vaatii aliluokan syötteen y olevan vähintään samanlaista kuin yliluokalle annettu syöte x. Tämä varmistaa ohjelman toimivuuden myös tilanteessa, jossa asiakasohjelma tarvitsee informaatiota aliluokalta. Kuva 2 on esimerkki tämän periaatteen toiminnasta käytännössä. Periaatteen mukaan asiakasohjelman tulee toimia, kun Ajoneuvo-luokan olio on asetettu olemaan Auto- tai Bussi-olio. Sekä Auto- että Bussi-luokka määrittelevät tarvittavat metodit Ajoneuvon määrittämiseksi tässä yhteydessä, joten periaate toteutuu. [5, s. 8; 8.]



Kuva 2. Luokkakaavio The Liskov Subsitution -periaatteesta.

Interface Segregation -periaate

Suosittu ohjelmisto on aina toivottava asia menestykseen pyrkivälle IT-yritykselle. Suositun ohjelmiston käyttäjät ovat myös usein halukkaita toivomaan ohjelmistoon lisää ominaisuuksia. Ohjelmoijan näkökulmasta tämä voi olla ongelmallista, jos halutun ominaisuuden toteuttamiseksi on muokattava ohjelmakoodia runsaasti. Open-Closed-periaatteen mukaan muutoksia on pyrittävä välttämään, jotta olemassa olevat ominaisuudet

eivät hajoaisi. Eräs tapa välttää muutokset on lisätä toiminnallisuus ohjelman käyttämään rajapintaan. Interface Segregation -periaatteen mukaan tämä johtaa ongelmiin, koska kaikki rajapintaa käyttävät osa-alueet eivät välttämättä tarvitse kyseistä toiminnallisuutta. Jakamalla toiminnallisuudet useampaan rajapintaan vältytään täyttämästä vain yhtä rajapintaa epäolennaisilla ominaisuuksilla. [5, s. 14; 8.]

Dependency Inversion -periaate

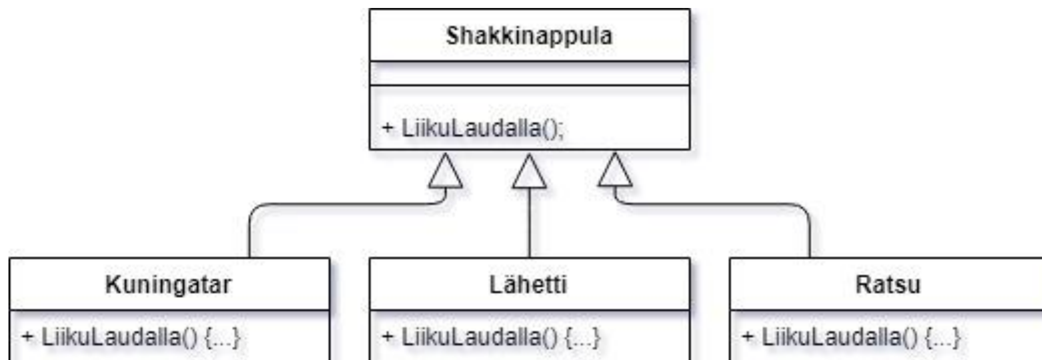
Dependency Inversion -periaatteen mukaan ohjelmiston tulisi luottaa enemmän abstrakteihin kuin konkreettisiin olioihin. Ohjelmakoodin ylemmällä tasolla määritelty abstraktio luo enemmän joustavuutta alemmalla tasolla tarkemmin määriteltyyn toiminnallisuuteen. Päämääränä on, että sekä ylemmän että alemman tason ohjelmakoodi luottaa toiminnassaan abstraktioon. Ylemmän tason oliot eivät saa olla riippuvaisia alemman tason konkreettisesta toteutuksesta. Kääntämällä riippuvaisuussuhteen luottamaan abstraktioon vältytään luomasta liian konkreettista toteutusta, joka voi tulevaisuudessa muuttua suunnittelun edetessä. [5, s. 12; 8.]

Rajapinnan hyödyntäminen

Olio-ohjelmoinnissa usein hyödynnettävä perintä auttaa määrittämään samankaltaisia olioita. Aliluokat perivät yliluokassa määritellyn toteutuksen. Vaikka perityt toiminnallisuudet määriteltäisiin abstraktisti, voi massiiviseksi paisuva periminen tehdä ohjelmasta vaikean hallita. Toiminnallisuuden lisääminen yliluokkaan tarkoittaa myös muutoksia aliluokkaan, ja jos usea aliluokka perii saman yliluokan, voi ohjelmoija joutua kirjoittamaan moneen kertaan saman toteutuksen. Kun kapseloidaan epäolennainen toiminnallisuus yliluokasta rajapintaan erilliseen toteutukseen, luodaan parempi suhde ominaisuutta käyttävien olioiden välille. Freemanin ym. [9] ja Gamman ym. [6] mukaan tämä voidaan esittää kahtena periaatteena: **suosi rajapintaa konkreettisen implementoinnin sijaan** ja **suosi olioiden yhdistelmää perinnän sijaan**.

Kuvassa 3 on esitetty shakkipelin luokkahierarkia. Aliluokat Kuningatar, Lähetti ja Ratsu perivät yliluokassa määritellyn abstraktin liikkumisen ja määrittelevät omassa luokassaan konkreettisen liikkumisen säännöt. Shakkipeli kuitenkin tarvitsee toimiakseen paljon enemmän nappuloita, mikä tarkoittaa ohjelman laajentamista ja pahimmassa

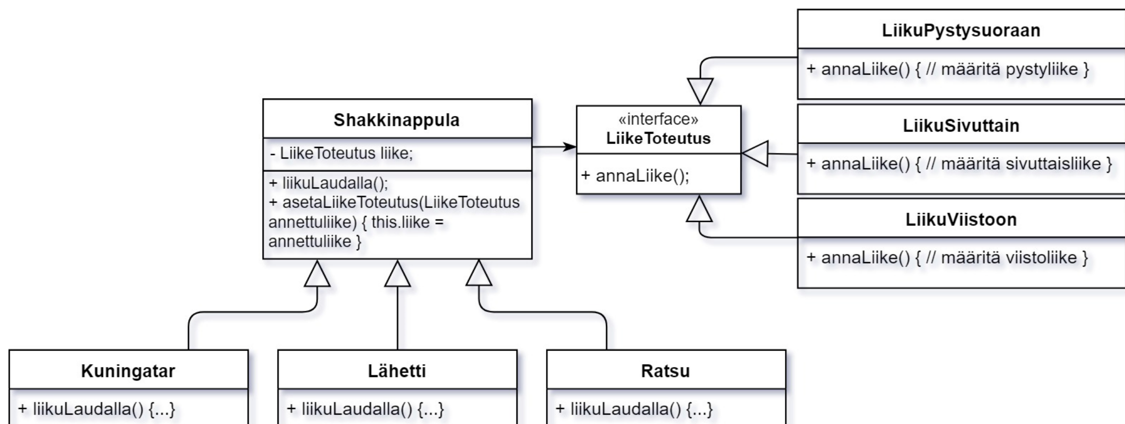
tapauksessa saman koodin lisäämistä useaan kertaan. Voidaan huomata, että liikkeen toteutus vaihtelee eri luokkien välillä. Nappulan ei kuitenkaan tarvitse itse määritellä liikettään, vaan se voidaan erottaa omaan toteutukseen, mikä vähentää konkreettisten nappuloiden vastuuta.



Kuva 3. Luokkakaavio, joka esittää shakkipelin nappuloiden hierarkiaa. Kuningatar-, Lähetti- ja Ratsu-luokat perivät Shakkinappula-luokan metodin liikkua, mutta joutuvat implementoimaan omat toteutukset liikkumiselle. Liikkeen toteutus on määritelty IS-A-suhteella.

Kuvassa 4 on esitetty uusi luokkahierarkia shakkinappuloille. Nappulat saavat rajapinnan kautta LiikeToteutus-olion eivätkä ole tietoisia sen konkreettisesta implementaatiosta. Vastuun jakaminen erilliselle LiikeToteutus-oliolle ja sen yhdistäminen Shakkinappula-luokkaan tekee ohjelmasta dynaamisemman ja helpomman muokata, jos esimerkiksi pelin säännöt muuttuvat. Kun kapseloidaan liikkeen toteutus ulos, vähennetään nappula-olion vastuuta ja varsinainen tehtävä tarkentuu. Vaikka olioiden ja luokkien määrä kasvaa, pysyvät yksittäiset luokat pienempinä. Mielestäni tämä malli luo ideaalin tilanteen työskennellä ohjelman parissa. Vaikka perinnällä voidaan saavuttaa toiminnallisuuden uudelleenkäyttöä, on järkevämpää jakaa toiminnallisuus pienempiin kokonaisuuksiin, jotka voidaan puolestaan uudelleen käyttää perinnällä. Freeman ym. [9] käyttävät olioiden yhdistämisestä termejä IS-A ja HAS-A. IS-A-suhteella tarkoitetaan sitä toiminnallisuutta, jonka olio määrittelee itse ja joka ei tule ulkopuolelta. HAS-A-suhde on olion yhdistämistä toiseen ajatellen sitä, mitä toiminnallisuutta olio tarvitsee. Shakkinappulalla on LiikeToteutus.

Metropolia



Kuva 4. UML-kaavio, jossa nappuloiden liikkeen määrittely on eristetty muusta toteutuksesta. Tarvittaessa voidaan määrittää useampi Liiketoteutus tai vaihtoehtoisesti nappuloille voidaan asettaa enemmän kuin yksi liike. Liikkeen toteutus on nyt määritelty HAS-A-suhteella.

Abstraktiot ja rajapinnat ovat oliopohjaisessa ohjelmoinnissa hyvin kriittisessä asemassa, kun ohjelmasta halutaan suunnitella mahdollisimman hyvin muutoksia kestävä. Oikeanlaisen rajapinnan tai abstraktion määrittely on kuitenkin hankalaa. Suunnittelumallit auttavat tunnistamaan abstraktion avainasemassa olevien elementtien välille ja kapseloimaan ne.

2.5 Suunnittelumallit

Alkujaan suunnittelumallit perustuvat Christopher Alexanderin ajatukseen siitä, että talon rakentamisessa usein ilmeneviä ongelmia pystytään ratkaisemaan samanlaisella mallilla aina uudelleen, ilman, että käytetty malli toteutuu täysin samalla tavalla [10]. Eric Gamma ym. [6, s. 2] eli Gang of Four, joka olkoon tästä eteenpäin GoF, tutustuivat tähän ajatukseen ja sovelsivat sitä ohjelmistotekniikkaan. Ohjelmistotekniikan suunnittelumallit kuvaavat heidän mukaansa olio-ohjelmoinnissa ja ohjelman tuottamisessa yleisesti esiintyviä ongelmia, ja esittävät abstraktin ”kaavan” ongelman ratkaisuun. On ohjelmoijan vastuulla tunnistaa suunnittelussa ilmaantuvat ongelmat, koska suunnittelumallit eivät kuitenkaan anna valmista ratkaisua.

Suunnittelumallit on luotu siis välittämään kokeneempien ohjelmoijien kokemuksen pohjalta tietoa esitettynä tietynlaisena mallina. Ne toimivat parhaimmillaan vähemmän

kokeneen ohjelmoijan apuna tukemassa ohjelman suunnitteluratkaisujen päättämistä. Koskimies ja Mikkonen [11, s.117] esittävät suunnittelumallien seuraavan suuntausta, jossa teknologian kehityksessä tapahtuvat toistuvat ilmiöt pyritään konkretisoimaan nimeämällä ja ilmaisemalla ne jollain esitystavalla.

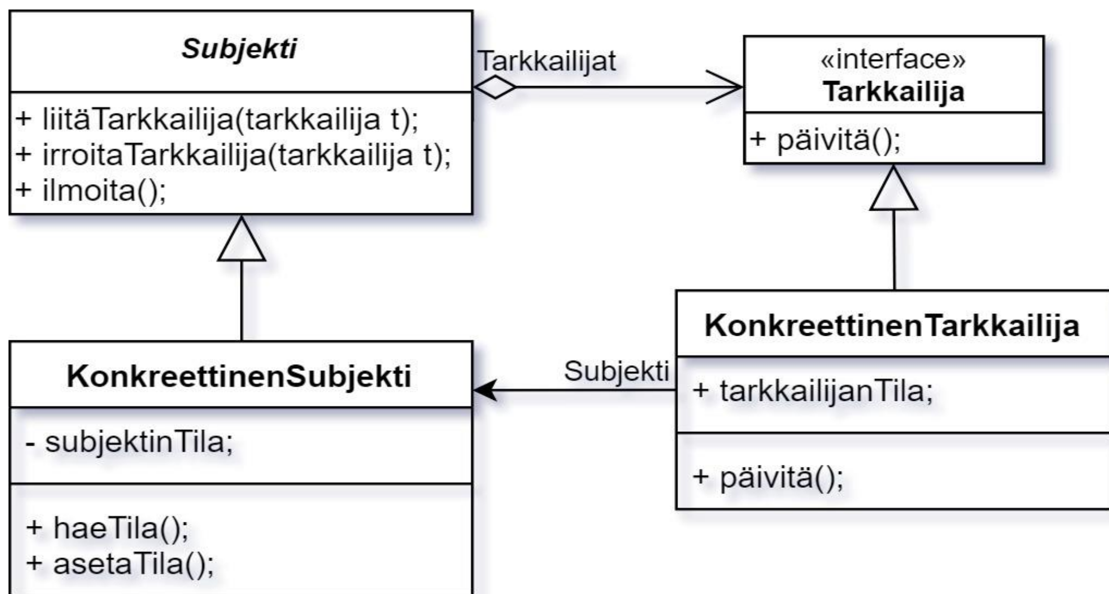
Yksittäisestä suunnittelumallista voidaan erottaa neljä kuvaavaa tekijää. Ensinnäkin mallin nimi kuvaa kaikkea sitä, mitä mallista voidaan lyhyesti tuoda esiin. Se sallii mallin käsittelyn abstraktimmalla tasolla menemättä liikaa yksityiskohtiin. Pitämällä yllä sanastoa suunnittelumalleista voidaan kommunikoida helposti eri ongelmiin soveltuvista suunnittelumalleista, sillä mallien nimet kuvaavat paitsi ongelmaa ja ratkaisua, myös mahdollisia seurauksia ohjelmalle. [6, s. 3; 11, s. 105.]

Toinen elementti on se ongelma, johon mallia sovelletaan. Ongelman kuvaukseen liittyy vahvasti myös konteksti, jossa se esiintyy. Ongelma voi olla vahvasti suunnitteluun tai esimerkiksi arkkitehtuuriin liittyvä. Ongelma pyritään kuitenkin kuvaamaan mahdollisimman selkeästi, jotta suunnittelumallin soveltuvuus ratkaisuksi on yksiselitteinen. Ongelmaan liittyen on tavallista esittää tiettyjä vaatimuksia, joiden on täytyttävä, ennen kuin voidaan tehdä päätös mallin soveltuvuudesta ongelman ratkaisuksi. [6, s. 3; 11, s. 105.]

Kolmas tärkeä elementti on varsinainen ratkaisu. Ratkaisussa esitetään kaikki ohjelmoinnin elementit kuten, luokat sekä niiden väliset suhteet ja velvollisuudet. Ratkaisua tulee käyttää vain suuntaa antavana kaavana kyseessä olevaan ongelmaan, sillä sen tarkoitus ei ole kuvata erityisesti yksittäistä toteutusta. Käytössä olevan ohjelmointikielen ei tulisi vaikuttaa suoraan ratkaisun soveltuvuuteen, mutta esimerkiksi perinnän ja polymorfismin puuttuminen voi tehdä mallin toteuttamisesta hankalaa. Suunnittelumallien suosion vuoksi nykyiset ohjelmointikielet tukevat monia malleja suoraan ohjelmointirajapinnoissaan. [6, s. 3; 11, s. 105.]

Neljäntenä elementtinä kuvataan niitä seurauksia, joita suunnittelumallin soveltamisesta seuraa. Yleensä mallin käytöstä aiheutuu sekä hyviä ja huonoja puolia arkkitehtuurille. Koskimiehen ja Mikkosen [11, s. 104] mukaan suunnittelumallit eivät tarjoa optimaalista ratkaisua ongelmalle, vaan kunkin mallin kohdalla on tietty vaihtokauppa. Esimerkiksi joustavuuden lisääminen voi vastaavasti heikentää suorituskykyä. Mielestäni tiettyjen mallien suhteen erityisesti peleissä nämä osien heikennykset ovat kuitenkin minimaalisia

verrattuna saavutettavaan hyötyyn, joten ne eivät oikeastaan ole merkityksellisiä. Seurausten vaikutusten tulee kuitenkin olla listattuna tarkasti, jotta ohjelmoija voi arvioida tehokkaasti mallin soveltuvuutta. Usein seurausten kuvaukset tarjoavat lisäksi vaihtoehtoista mallia, joka voi toimia tietyissä tilanteissa jopa paremmin. Näiden elementtien lisäksi suunnittelumallin käytöstä voidaan antaa esimerkkejä pseudokoodia hyväksi käyttäen tai kuvata mallia oikean maailman ilmentymänä. Hyvin yleinen tapa kuvata suunnittelumallia abstraktisti on esittää se UML (engl. Unified Modeling Language) -mallinnuksessa hyödynnettävällä luokkakaaviona, kuten kuvassa 5, jossa on esitetty Tarkkailija-mallin (engl. Observer) rakenne.



Kuva 5. Tarkkailija-suunnittelumalli esitettynä UML-luokkakaaviona. Abstrakti esitys auttaa ymmärtämään nopeasti mahdolliset ohjelmakoodin tarpeet.

Suunnittelumalleja luokitellaan lisäksi kolmeen kategoriaan liittyen niiden tarkoitukseen tai tarkoitusperään. Luontimallit keskittyvät Gamman ym. [6, s. 10] mukaan olioiden luomiseen ja auttavat tekemään ohjelmasta riippumattoman varsinaisesta luomisesta. Rakennemallit huolehtivat siitä, miten luokat ja oliot yhdistyvät suuremmaksi kokonaisuudeksi. Käyttäytymismallit liittyvät olioiden välisten vastuiden, suhteiden ja vuorovaikutusten kuvaamiseen. Sen lisäksi jaottelua tehdään kohteen mukaan: onko kohde luokka vai olio.

Useimpien GoF-suunnittelumallien ideana on tehdä ohjelmakoodista helpommin muokattava ja uudelleenkäytettävä. Muita mahdollisia laatuominaisuuksia on Koskimiehen ja Mikkosen [11, s. 104] mukaan myös ylläpidettävyys, siirrettävyys ja suorituskyky. Mielestäni suorituskyvyn kanssa tulisi ajatella myös muistinhallintaa laatuominaisuutena. Sen vuoksi suorituskyky ja muisti tulisi ajatella yhtenä käsitteenä – optimaalisuutena. Optimaalisuuden kasvattaminen varsinkin pelejä ajatellen on tärkeää erityisesti loppukäyttäjän näkökulmasta.

Robert Nystromin [12] mukaan kysymys on oikean tasapainon löytämisestä hyvän ohjelma-arkkitehtuurin ja optimaalisuuden välillä. Toisaalta tavoitteena on tehdä ohjelmasta esimerkiksi helposti muokattava ja ylläpidettävä, mutta toisaalta saavuttaa nopea suorituskyky, ja kaikki tämä hyödyntäen uusinta teknologiaa. Nyströmin argumentti kuitenkin on, että on helpompaa tehdä pelistä hauska nopeasti kuin tehdä nopeasta pelistä hauska. Kompromissina hän esittää, että ensin kannattaa suosia ohjelman rakentamista joustavaksi, kunnes suunnitteluvaihe on saatu päätökseen, ja vasta tämän jälkeen poistaa ohjelmaa hidastavia elementtejä.

2.6 Suunnittelumallit peleissä

Peliteollisuus on kasvanut nopeasti viimeisten vuosikymmenten aikana, ja tulevaisuuden näkymät ennustavat suunnan jatkuvan nousevasti. Vuoteen 2020 mennessä videopelimarkkinoiden uskotaan ylittävän 90 miljardin dollarin arvon [13]. Tähän pisteeseen pääseminen ei ole tapahtunut sattumalta, vaan peliteollisuus on menestyäkseen hyvin useasti lainannut paljon muilta viihdeteollisuuden alueilta, kuten filmiteollisuudelta, mutta myös ohjelmistotekniikan alueilta. Yhtenä näistä lainatuista asioista voidaan pitää suunnittelumalleja, jotka alun perin kehitettiin ohjelmistoille eikä niinkään peleille. [14.]

Nykyään peliteollisuus jättää taakseen sekä elokuva- että musiikkiteollisuuden, jos verrataan suosiota ja taloudellista arvoa. Paine peliyhtiöiden puolella on toisaalta kasvanut kuluttajien kasvavien vaatimusten myötä. Yleisö odottaa korkeaa laatua suuremmilta peliyhtiöiltä. Laadun takaamiseksi tulee käyttää aikaa, vaikka kiusaus nopeampaan aikatauluun on suuri peliteollisuuden kovan kilpailun vuoksi. Grand Theft Auto 5 -pelin [15] julkaisua jouduttiin siirtämään noin neljä kuukautta, koska peli yksinkertaisesti tarvitsi enemmän viimeistelyaikaa [16]. Luonnollisesti julkaisuajankohdan ylittäminen ei ole

erityisen hyvää julkisuutta, vaikkakin tässä tapauksessa pelin julkaisun siirtyminen myöhemmäksi saattoi toimia jopa peliyhtiön eduksi, kun se toi mainosta ja kiinnostusta peliä kohtaan ja teki odottavista kuluttajista entistä kiinnostuneempia. Kuluttajat usein myös ymmärtävät julkaisun venymisen, koska valmiin pelin halutaan olevan niin näyttävä kuin mahdollista, unohtamatta mahdollisimman sujuvaa pelattavuutta.

Mahdollisimman sujuvan pelattavuuden puolestaan mahdollistaa toimiva ohjelma, joka ei sisällä pelin immersiota rikkovia ohjelmointivirheitä. Ohjelmointivirheiden korjaaminen puolestaan tehdään muuttamalla olemassa olevaa koodia, mikä on aina riski. Huolellisella suunnittelulla saadaan aikaan vähemmän virheitä sisältävä peli, joka vaatii samalla vähemmän ylläpitoa. Suunnittelumallit voivat auttaa pelin kehityksessä tunnistamaan tiettyjä elementtejä, joihin mallin käyttö sopii parhaiten. Pelien kehittäjät ovat myös tunnistaneet suunnittelumallien voiman ja kehittäneet lukuisia omia mallejaan, joiden motiivina on pelinkehityksen edesauttaminen tunnistamalla pelinkehityksessä yleisiä ongelmia. Yksi näistä malleista on lähes jokaisesta pelistä löytyvä Pelisilmukka-malli (engl. The Game Loop).

Pelisilmukka-malli

Pelisilmukkaa verrataan usein koko pelin sykkivään sydämeen, koska se mahdollistaa kaiken interaktiivisen toiminnan pelissä. Silmukka toistuu jokaisen kehyksen (engl. frame) aikana ja luo saumattoman näköisen kulun iteroimalla kaiken pelattavuuteen tarvittavat resurssit. Jos pelin kuvataajuus eli kehysnopeus on vaikkapa 60 fps, tekee silmukka 60 iteraatiota sekunnissa. Vaikka Pelisilmukka-mallista on olemassa useita eri variaatioita, on sen perusrakenne yleensä hyvin samanlainen kaikissa peleissä. On myös huomattava, että hyvin harva muu kuin peliohjelma käyttää Pelisilmukka-mallia toiminnassaan, vaikka samannäköisellä periaatteella toimivia ohjelmia onkin olemassa. [12.]

Motivaatio Pelisilmukka-mallin käyttöön kasvoi Nystromin [12] mukaan sen jälkeen, kun peleistä haluttiin saada interaktiivisempia. Vanhempi tapa tehdä pelejä ei mahdollistanut kovin nopeaa ohjelman testausta, sillä datan saaminen kesti kauan. Ensimmäiset pelit, jotka tätä hyödynsivät, olivat tekstiseikkailupelejä. Peli odotti käyttäjän antamaa syötettä ja reagoi sen mukaan. Tällä samalla periaatteella toimivat mm. tekstinkäsittelyohjelmat.

Ohjelma odottaa, mitä näppäimistön nappulaa käyttäjä painaa, ja piirtää kyseisen merkin. Muulloin ohjelma odottaa seuraavaa komentoa. Nykyajan pelit eivät tietenkään pysähdy odottamaan komentoa, vaan pelin on kuljettava riippumatta pelaajan syötteestä.

Pelisilmukan kulku voidaan jakaa karkeasti kolmeen eri osaan, jotka on eritelty esimerkkikoodissa 1. Ensiksi mallissa käsitellään saatavilla oleva syöte. Tähän kuuluvat luonnollisesti pelaajan antamat esimerkiksi näppäimistön, hiiren tai peliohjaimen komennot. Sanjay Madhav [17] lisää, että tämä ei kuitenkaan ole kaikki syöte, vaan tässä vaiheessa käsitellään myös ulkopuolelta tuleva syöte. Esimerkkinä ulkopuolelta tulevasta syötteestä voidaan mainita kaikki internetin välityksellä tuleva informaatio. Moninpeleissä tämä informaatio on tärkeää, jotta peli piirtyy oikein kaikille pelaajille.

```
while(true)
{
    käsitteleSyöte();
    päivitäResurssit();
    renderöi();
}
```

Esimerkkikoodi 1. Esimerkki Pelisilmukka-mallin perustasta.

Seuraavaksi peli käy päivittämässä kaikki ne pelin aktiiviset oliot, jotka tarvitsevat päivitystä. Määrä voi helposti nousta satoihin tai tuhansiin olioihin. Olioiden määrä voi vaikuttaa negatiivisesti kuvataajuuteen, jos Pelisilmukan täytyy päivittää massoittain olioita. Päivityksessä tapahtuvat yleensä myös tekoälyn ja fysiikan laskut, jotka voivat osaltaan kuormittaa suoritinta. [17.]

Viimeinen askel on varsinaisen pelin piirtäminen ruudulle, jotta pelaaja näkee, mitä pelissä tapahtuu. Nyt päivitetyt oliot saavat uuden päivitetyn sijainnin pelimaailmassa. Tässä vaiheessa Madhavin [17] mukaan kuvan lisäksi toinen yhtä tärkeä annettava syöte on äänet, jotka myös päivitetään tässä. Pelkästään pelaajalle itselleen näkyviin tulevat kuvat eivät ole kuitenkaan ainoa syöte. Moninpelissä muiden pelaajien tarvitsema data lähetetään tässä vaiheessa.

Esimerkkikoodissa 1 esitettyä Pelisilmukkaa prosessoidaan niin nopeasti kuin suoritin pystyy. Luvun alussa määriteltiin yhden kierroksen pituudeksi yksi kehys. Tämä tarkoittaa myös sitä, että jokainen kehys vie pelimaailman aikaa yhden pykälän eteenpäin. Ongelmaksi tämän tekee se, ettei ohjelmoija pysty hallitsemaan nopeutta mitenkään.

Hitaammalla koneella silmukka olisi aivan liian hidas. Vanhemmissa videopeleissä sekunnin määritys oli helppoa, koska suorittimen teho tiedettiin tarkasti. Ohjelmakoodi räätälöitiin sopimaan juuri tiettyä konsolia varten, mutta jos tätä peliä yritettiin pelata esimerkiksi huonommalla laitteella, peli toimi vastaavasti hitaammin. Nykyään ohjelmoijat eivät hyvin todennäköisesti tiedä kaikkien kuluttajien laitteiston tasoa, joten Pelisilmukka tarvitsee mukautumista. Tämä on oikeastaan Pelisilmukan päätehtävä: se pitää peliä käynnissä yhtenäisellä nopeudella riippumatta koneen laitteiston tasosta. [12; 18.]

Yksinkertainen lisäys esimerkkikoodi 1:n tilanteeseen voidaan nähdä esimerkkikoodissa 2. Koska jokaisen silmukan läpi käyminen vie tietyn verran oikean maailman aikaa, täytyy ajan kuluminen kiinnittää tiettyyn aikaan ohjelmassa. Nämä ajat ovat usein hyvin lyhyitä, ja ne esitetään millisekunteina. On yleinen tapa kiinnittää pelin kuvataajuus 60 kehykseen sekunnissa. Tämän saavuttamiseksi silmukan minimiajaksi määritellään siis 16 ms. [12; 18.]

```
double viimeAika = saaNykyinenAika();
double viive = 0.0;
while(true)
{
    double nykyinenAika = saaNykyinenAika();
    double kulunutAika = nykyinenAika - viimeAika;
    viimeAika = nykyinenAika;
    viive += kulunutAika;

    käsitteleSyöte();

    while(viive >= MILLISEKUNTIA_PER_PÄIVITYS)
    {
        päivitäResurssit();
        viive -= MILLISEKUNTIA_PER_PÄIVITYS;
    }

    renderöi();
}
```

Esimerkkikoodi 2. Esimerkki Pelisilmukka-mallista, jossa laitteiston teho on otettu huomioon.

Nyström selittää muutoksen mahdollistavan resurssien päivityksen täsmällisesti määriteltyjen aikojen välein. Tällöin päivitysten määrä ei ole sidottu suorittimen suorituskykyyn, vaan päivitystä tehdään tarpeen vaatiessa. Jos edellinen kehys vei kauan, täytyy päivitystä tehdä enemmän, jotta peli saa oikean ajan kiinni, ennen kuin voidaan piirtää. Vastaavasti, jos kehys suoriutui nopeammin, ei päivitystä tarvitse tehdä. Tällä menetelmällä kuvataajuus voi olla jopa suurempi kuin 60 fps. Varsinainen hyöty on kuitenkin se, että aikaa vievä renderöinti on saatu eristettyä päivittämisestä. [12.]

Muita suunnittelumalleja

Pelisilmukka-malli on ehdottomasti tärkeimpiä malleja, joita peleissä käytetään. Se on jossakin muodossa lähes kaikissa tunnetuimmissa pelimoottoreissa. Toisaalta, oman Pelisilmukan kirjoittamisella saavutetaan täysi kontrolli sen käytöstä, ja se voidaan suunnitella optimaalisesti omaan käyttöön. Pelisilmukka ei kuitenkaan ole ainoa varsinaisesti pelejä varten kehitetty suunnittelumalli. Päivitysmetodi (engl. Update Method) ja Komponentti (engl. Component) ovat malleja, jotka Nystromin [12] mukaan muodostavat yhdessä Pelisilmukan kanssa ytimen pelimoottorin toimintalogiikalle. Päivitysmetodi-malli oikeastaan toimii käytännössä Pelisilmukka-mallin osana, ja Komponentti-malli tukemassa toiminnallisuuden lisäämistä pelin olioihin komponentteina. Nämä ja muut Nystromin määrittelemät peleihin suuntautuneet suunnittelumallit jaetaan niiden tarkoituksen mukaan, kuten GoF-mallit mm. käyttäytymis- tai optimointimalleihin.

2.7 Suunnittelumallien kritiikki

Vaikka suunnittelumallien käyttö ohjelman arkkitehtuurin suunnittelussa on yleisesti todettu hyväksi, liittyy niiden käyttöön myös ongelmia. Erityisesti GoF-mallien kohdalla täytyy ohjelmoijan tarkasti miettiä, onko suunnittelumallin käyttö ongelman ratkaisemiseksi paras mahdollinen ratkaisu. Koskimiehen ja Mikkosen [11, s. 117] mukaan GoF-suunnittelumallit lisäävät yleisesti ohjelmakoodin arkkitehtuurin kompleksisuutta esimerkiksi lisäämällä luokkia ja rajapintoja. Jos ohjelma ei hyödy mallin käytöstä merkittävästi, on kenties parempi vaihtoehto jättää mallin käyttö pois. Liian pienen ongelman ratkaiseminen suunnittelumallilla vaikuttaa jo itsessään hyvin epäjohdonmukaiselta. Toinen suunnittelumalleja vastaan osoitettu kritiikki liittyy niiden maineeseen ja niiden ympärillä käytyyn keskusteluun.

Jos ohjelman suunnittelu on vain suunnittelumallipohjaista, tekee ohjelmoija itselleen karhunpalveluksen. ”If all you have is a hammer, everything looks like a nail” [20]. Joissain piireissä suunnittelumalleja ja GoF-kirjaa on ylennetty lähes jumalalliseen asemaan, mikä on tehnyt joistain ohjelmoijista sokeita mallien aiheuttamille ongelmille. Kenties kirjan julkaisuaikaan ohjelmointiyhteisö oli suunnittelumallien tarpeessa, mutta nykypäivänä niiden käyttöä halutaan suitsia enemmän.

3 Suunnittelumallien tutkiminen ja toteuttaminen

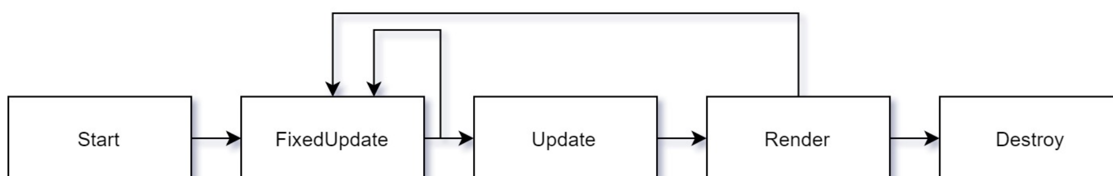
Insinöörityössä tutkittiin, kuinka suunnittelumallien käyttö ilmenee Unity-pelimoottorissa, ja sen lisäksi implementoitiin kolme erilaista suunnittelumallia, joiden käytön oletettiin parantavan kohteena olevan pelin arkkitehtuuria ja optimaalisuutta. Tässä luvussa käsitellään käytännön toteutusta.

3.1 Suunnittelumallit Unityssa

Unity-pelimoottori on tällä hetkellä maailman mittakaavassa johtavassa asemassa kolmannen osapuolen pelinkehityssovellusten markkinoilla. Sen asema perustuu mm. ilmaiseen lisenssiin niin kauan kuin kehitetyn pelin tuottamat tulot eivät ylitä 100 000 euron rajaa [20]. Unity on myös tehty mahdollisimman helposti lähestyttäväksi aloittelijan näkökulmasta, sillä pelimoottori tarjoaa monia ominaisuuksia ohjelmoinnin helpottamiseksi. Insinöörityössä tarkasteltiin lähemmin näitä ominaisuuksia ja tutkittiin suunnittelumallien ilmentymistä niissä.

Vaikka Unity on rakennettu käyttäen C++:aa, se tarjoaa ohjelmoijan käyttöön .NET-ohjelmointirajapinnan, joka on kiedottu Unityn ympärille. Tämä tarjoaa mahdollisuuden ohjelmoida pelin toiminnallisuus skripteinä käyttäen C#-ohjelmointikieltä. Jotta Unityn tarjoamia ominaisuuksia pystytään hyödyntämään, tulee jokaisen skriptin olla johdettu Unityn kantaluokasta, MonoBehaviourista. [21.]

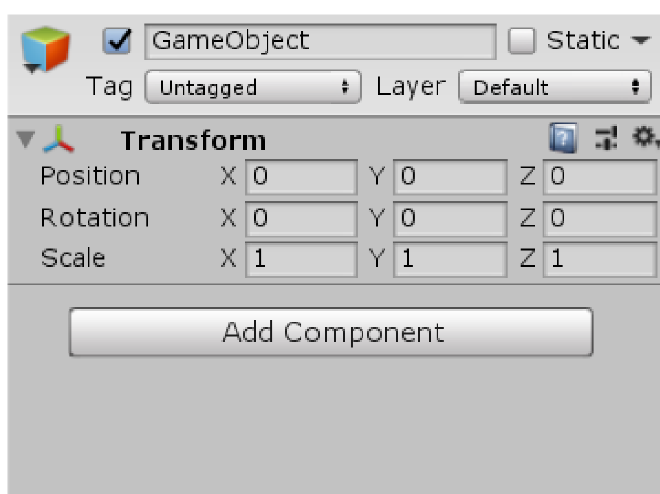
Kun tarkastellaan kuvassa 6 Unityn skriptien suoritusjärjestystä, voidaan huomata selvä yhteneväisuus Pelisilmukka-mallin kanssa. Skriptin alustuksen jälkeen seuraa resurssien päivitys, jonka jälkeen renderöidään pelin grafiikka. Peruslogiikka kuitenkin yhä pätee, kun rakennetta verrataan esimerkkikoodissa 1 esitettyyn logiikkaan.



Kuva 6. Liitteen 1 vuokaaviosta yksinkertaistettu malli Unityn skriptin elinkaaresta oliossa.

Kuvasta 6 havaittavat FixedUpdate- ja Update-vaiheet toteuttavat Päivitysmetodi-mallia. Päivitysmetodi-mallin tarkoitus on Nystromin mukaan pitää yllä pelimaailman jokaisen olion päivittäminen. Unity jakaa siis resurssien päivityksen kahteen eri päämetodiin. FixedUpdate on Unityn fysiikkamoottorin laskujen päivityssilmukka ja riippumaton pelin kuvataajuudesta. Sen tarkoitus on päivittää resurssit määritellyin aikavälein, kuten esimerkkikoodissa 2 olevan esimerkin mukaan. Unityssa tämä aikaväli on noin 0,02 sekuntia, mikä johtaa noin 50 kutsuun sekunnissa. Update-metodi päivitetään kerran yhden kehyksen aikana. Jos kuvataajuus on 60 fps, kutsutaan Update-metodia 60 kertaa sekunnissa. Ero FixedUpdaten ja Updaten välillä syntyy siis siitä, että Updaten päivittäminen riippuu pelin kuvataajuudesta, mutta FixedUpdaten päivitys on rajattu vain aikaväliin. [22.]

Jokainen olio Unityssa on GameObject-olio, ja nämä oliot voivat edustaa mitä hyvänsä valosta kameroihin. Samaan aikaan nämä oliot ovat pelin kannalta turhia ilman minkäänlaista toiminnallisuutta. Unity hyödyntää toiminnallisuuden lisäämiseksi Komponenttimallia. Ohjelmoijalla on vapaus lisätä haluamansa toiminnallisuudet Komponentteina suoraan olioon, mistä on esimerkki kuvassa 7. Komponentti-mallin ideana on siis määrittää erilaiset oliot eli GameObjectit toiminnallisuudella, joka on eroteltu Komponentteihin. Tämä suunnittelumalli toteuttaa periaatetta olioiden yhdistämisen suosimisesta perinnän sijaan. Komponentit erotetaan muusta ohjelmasta samalla periaatteella kuin kuvassa 3 olevat liiketoteutukset. [12; 23.]

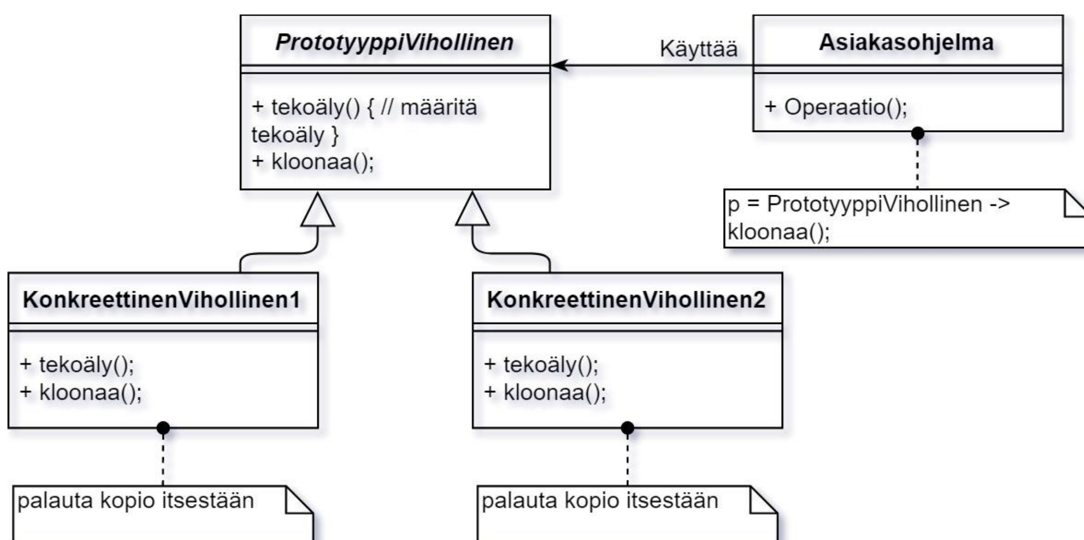


Kuva 7. Esimerkki tyhjästä GameObject-oliosta, johon lisätään Komponentteja ”Add Component”-napista.

Metropolia

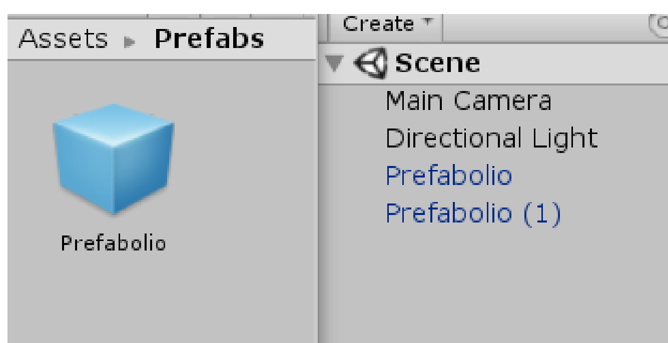
Tutkimalla Unityn GameObjectien käyttöä voidaan havaita muitakin suunnittelumalleja. Olioita voidaan periaatteessa ajatella vain datana ohjelmiston sisällä. Olio edustaa itsessään tietynlaiseen muotoon rakennettua dataa. Pelin tuotantovaiheessa voi kuitenkin tulla tilanne, että samasta oliosta tarvitaan useita kopioita. Tämä on hyvin usein tilanne varsinkin peleissä, joissa kopiot ovat olennaisessa osassa pelin sisältöä, olivat ne sitten luonto-olioita, kuten kiviä, tai vihollisia, kuten lohikäärmeitä.

Prototyyppirajapinta-malli (engl. Prototype) on luontimalli, joka auttaa luomaan olioista klooneja vaivattomasti. Jokainen kloonattava olio käsitellään prototyyppinä niille olioille, joita halutaan tehdä. Tekemällä lohikäärmeoliosta prototyyppi voidaan luoda useita saman datan sisältäviä, pelimaailmassa konkreettisesti eläviä lohikäärmeklooneja. Kloonit eivät kuitenkaan ole sidottuja prototyypin dataan, vaan ohjelmoija voi myös muokata kloonien toteutusta erilaiseksi rikkomatta prototyyppiä. Kuvassa 8 on esitetty yksinkertainen malli Prototyyppirajapinnasta. Asiakasohjelma luo klooneja kutsumalla kloonattavan olion kloonaa-metodia. Kloonattu olio saa täsmälleen samat ominaisuudet ja toiminnallisuudet kuin prototyyppi. Kuvassa 8 olevan esimerkin mukaan voidaan luoda halutun tekoälytoteutuksen omaavia vihollisia klooneina. Prototyyppirajapinta sallii myös sekä kloonien luomisen että poistamisen, kun ohjelma on käynnissä. Suurin ongelma mallissa on se, että jokainen aliluokka joutuu määrittelemään kloonausoperaation, mikä ei ole aina helppoa. [6, s. 117; 24.]



Kuva 8. Esimerkki Prototyyppirajapinnan käytöstä. Asiakasohjelma luo klooneja kutsumalla Konkreettisen vihollisen kloonaa-metodia. Nämä kloonit ovat itsenäisiä, ja niiden dataa pystytään muokkaamaan.

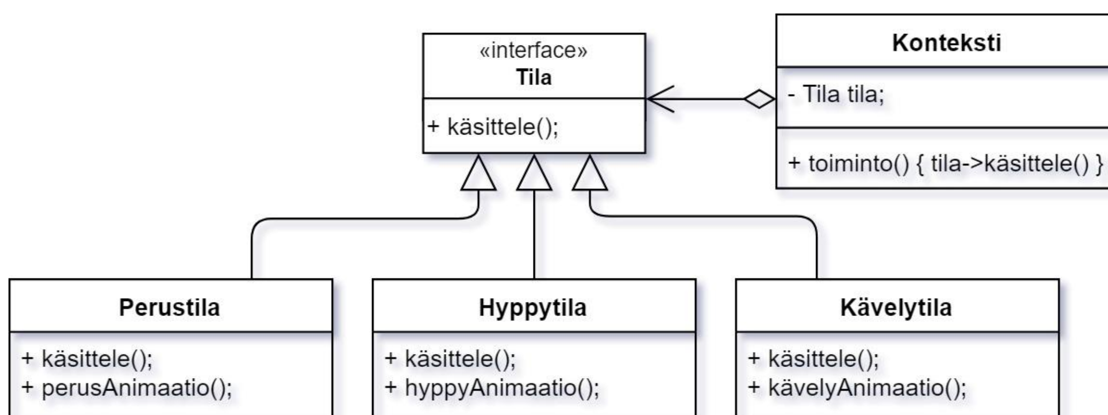
Prototyyppirajapinnan hyödyistä huolimatta on vaikeaa löytää hyödyllisiä käyttökohteita, joissa malli olisi paras ratkaisu, sillä ongelmat pystytään usein ratkaisemaan paremmin muilla tavoilla, väittää Nystrom [12]. Tutkittuani Unityn Prefab-järjestelmää olen Nystromin kanssa hiukan eri mieltä. Unity toteuttaa Prototyyppirajapinta-mallia pelin resursseihin tallennettujen Prefab-elementtien kautta. Nämä elementit edustavat prototyyppiä siitä oliosta, joka elementtiin tallennetaan, ja ovat tallennuksen jälkeen tarvittaessa ohjelmoijan saatavilla. Kuvassa 9 Prefab-olio toimii prototyyppinä kahdelle kloonille. Kuten kuvan 8 konkreettiset luokat, myös Prefabin kloonit ovat konkreettisia ilmentymiä. Prefabin kautta syntyneet kloonit ovat aina täydellisiä klooneja, ja Unity mahdollistaa myös yksittäisten kloonien datan muuttamisen, jos ohjelmoija niin tahtoo. [24.]



Kuva 9. Esimerkki Unityn resursseihin tallennetusta Prefabista ja siitä peliin luoduista klooneista.

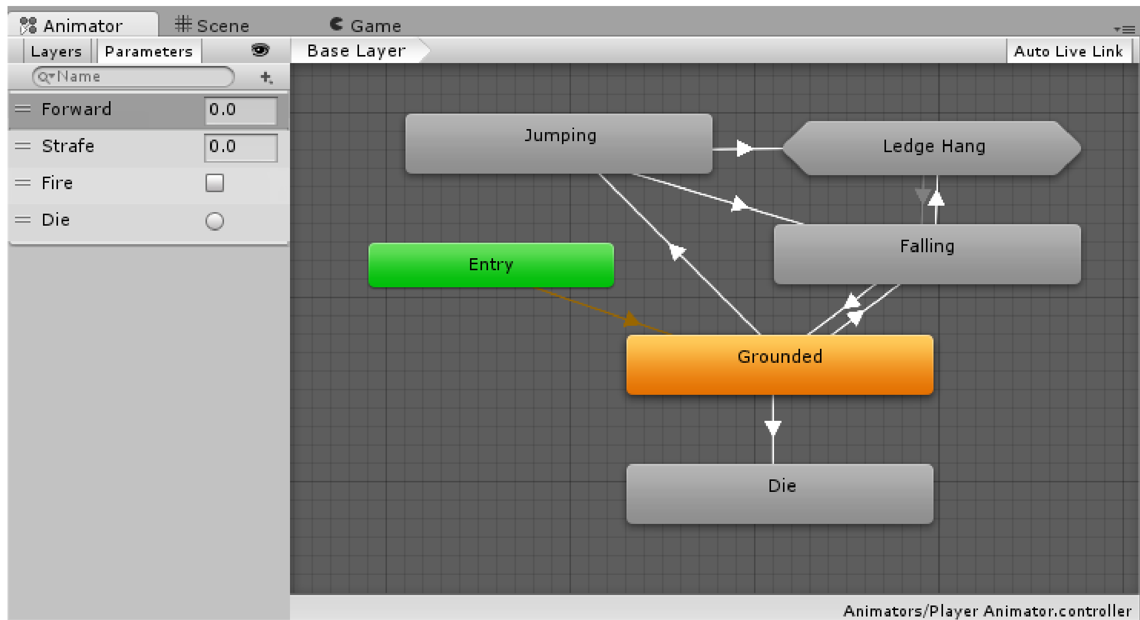
Ohjelmoija käyttää Prefabeja pääasiassa pelin sisällön tuottamiseen editoimisen aikana, mutta Prefabeja hyödynnetään myös skriptien kautta pelin käynnissä olon aikana. Yksi yleisin käyttötapa uusien kloonien nopealle luomiselle pelin aikana on ammusten luominen ampumisen yhteydessä. Yhdestä ammusprototyypistä luodaan tarvittavalla nopeudella aina uusi klooni. Toinen käyttökohde, jossa Prefabeja hyödynnetään kloonien luomisessa, on Unityn maanmuokkaustyökalu. Editori ottaa käyttöönsä halutun Prefabin ja luo pelin maastoon esimerkiksi puita ja kiviä. Suurimpana haittapuolena voidaan oikeastaan pitää mahdollisuutta luoda kaaos, kun ohjelmoija täyttää kansionsa lähes identtisillä, mutta pienillä muutoksilla varustetuilla Prefabeilla. Toisaalta, Prefabit käytännössä poistavat ohjelmoijien tarpeen luoda Prototyyppirajapinta itse, ja näin ollen vältytään mahdollisuudelta ohjelmointivirheisiin kloonausmetodin implementoimisessa. Jos Prefabit vaatisivat oman kloonaamismetodin toteuttamisen jokaiselle prototyypille, olisin enemmän Nystromin mielipiteen kannalla. [24.]

Unityn animaattorieditori on vielä yksi melko selvä esimerkki suunnittelumalleista. Animaatiot ja tekoäly jaetaan usein tiloihin, joissa pelaaja tai vihollinen sillä hetkellä on. Animaatioissa tilat voivat olla esimerkiksi perustila, juoksutila, kävelytila ja hyppytila. Animaattorieditori käyttää Tila-mallia (engl. State) hyväkseen esittämällä nämä eri animaatiotilat ja niiden väliset suhteet. Tila-malli on käyttäytymismalli, joka sallii olion muuttavan käyttäytymistään, kun sen sisäinen olotila muuttuu. Erottamalla erilaiset animaatiotilat omiksi luokikseen saadaan aikaan rakenne, jossa voidaan käsitellä yhtä tilaa kerrallaan, eikä tilojen tarvitse tietää toisistaan. Näin malli toteuttaa Single Responsibility -periaatetta. Kuvassa 10 oleva Konteksti-luokka voi olla se luokka, joka hallinnoin Pelaajan tilaa, ja Tila-rajapinnan toteuttavat luokat puolestaan niitä konkreettisia luokkia, jotka määrittelevät itse animaatiotilan. [6; 25.]



Kuva 10. Tila-mallin rakenne, jossa animaation näyttäminen tapahtuu Konteksti-luokan kautta delegoimalla käsittele-metodin sisällön konkreettisille tiloille.

Unityn animaattorieditori luo eri animaatioista tiloja samalla periaatteella. Tilat toimivat rakennuspalikoina, kun ne lopulta yhdistetään erilaisten siirtymien kautta äärelliseksi automaatiksi eli tilakoneeksi. Tilojen välinen siirtyminen tapahtuu skriptin kautta ohjelmoijan määrittelemällä tavalla. Kuvassa 11 on esimerkki animaattorieditorin näkymästä, jossa on esitettynä tilakone. Erilaiset tilat kuvataan laatikoina, joista jokainen edustaa animaatioleikettä. Nuolet edustavat siirtymiä ja osoittavat suunnan, jossa tilakoneessa edetään tilasta toiseen. [25.]



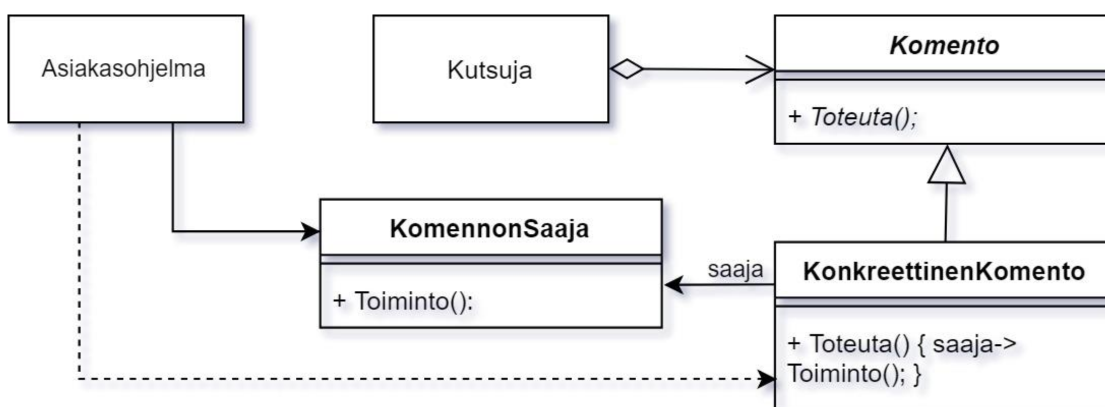
Kuva 11. Unityn animaattorieditori, jossa on määritelty tilakone. Tilojen väliset siirtymät kuvataan nuolella. [26.]

3.2 Suunnittelumallien toteutus

Suunnittelumallien oma toteutus peliin tehtiin tässä insinöörityössä valitsemalla avoimen lähdekoodin peli, jonka ohjelmakoodia analysoimalla oli tarkoitus löytää suunnittelumallin hyödyntämiselle soveltuvia kohteita. Analysoinnin aikana tutkittiin monia suunnittelumalleja ja verrattiin niiden tuomia hyviä ja huonoja puolia. Ylisuunnittelun välttämiseksi valittiin käyttökohteiksi koodista hyvin selkeät alueet, joiden refaktorointi oli tarpeen ja joissa huomattiin potentiaalisia muutoksen kohteita. Suunnittelumallien implementoinnin lisäksi peliin lisättiin toiminnallisuutta, joka mahdollistui suunnittelumallin kautta. Kohdepeliksi valikoitui Arenagame [27], jonka puolesta puhui valmis ulkoasu, erinomainen dokumentointi sekä vapaa ohjelmiston MIT-lisenssi. Arenagame on ensimmäisen persoonan ammuntapeli (engl. First-person shooter), jossa pelaaja kamppailee muita pelaajia vastaan suljetulla, areenan muotoisella alueella. Koska pelin tarkoitus oli toimia monen pelaajan pelinä, oli suunnittelumallien implementoiminen tehtävä säilyttäen alkuperäinen toiminnallisuus. Jokainen suunnittelumalli esitellään seuraavassa kolmessa luvussa aluksi yleisesti, ja UML-kaavioita käyttämällä annetaan esimerkkejä mallin hyödyistä ja motiivista sen käytölle pelissä.

3.2.1 Komento-malli

Komento-malli (engl. Command) on käyttäytymismalli, jonka tavoitteena on erilaisten ohjelman sisäisten kutsujen muuttaminen abstraktimpaan muotoon. Käytännössä Komento-malli kapseloi kutsut olioiksi, jotka sisältävät tiedon kutsusta. Tämä mahdollistaa myös kutsujen viivästyttämisen, säilömisen tai kumoamisen. Kapseloituja oliota kutsutaan komennoiksi suunnittelumallin mukaan. Klassinen esimerkki Komento-mallin käytöstä on tekstinmuokkausohjelman kopioi-liitä -komennot sekä mahdollisuus näiden kumoamiseen. Kuvassa 12 on esimerkki Komento-mallin rakenteesta. Konkreettiset komennot sisältävät varsinaisen komennon sisällön, ja ne kaikki toteuttavat abstraktin Komento-luokan. Saaja-luokka on komennon kohde, ja saajana voi olla minkälainen luokka hyvänsä. Oleellista on se saajan toiminto, joka halutaan komennon toteuttavan. Asiakasohjelma määrittelee ja luo halutut komennot sekä asettaa komentojen saajat. Kutsuja on se olio, jonka vastuulla on komennon Toteuta-metodin kutsuminen. Kutsujan rooli kaaviossa vaihtelee, sillä se voi olla esimerkiksi erillinen luokka, asiakasohjelma tai toinen komento. [6; 12.]



Kuva 12. Komento-mallin esimerkki UML-kaaviona. Asiakasohjelma luo konkreettiset komento-oliot ja määrittelee komennon saajan. Kutsuja kutsuu Komennon Toteuta-metodia. Kutsuja voi olla myös asiakasohjelma, mutta ei aina.

Komentoja pystytään laajentamaan tarpeen mukaan, ja yksi komento voi kutsua useaa eri komentoa. Näitä komentoja kutsutaan Gamman ym. [6] mukaan makrokomennoiksi. Yhdistelemällä eri komentoja voidaan luoda monimutkaisiakin toiminnallisuuksia tarpeen vaatiessa. Makrokomennot huolehtivat alikomentojen lisäämisestä ja poistamisesta

omasta toteutuksestaan, mutta eivät tarvitse tietoa saajasta, koska saaja määritellään suoraan alikomentoon. [6.]

Mallin käyttö pelissä

Komento-mallille voidaan keksiä monia käyttökohteita pelissä. Komentojen linkitys sarjaksi voi olla tärkeä ominaisuus strategiapelissä, jossa yksiköt liikkuvat komentojen mukaan. Tässäkin tapauksessa liikkumiset tapahtuvat pelaajan painamien nappien mukaan. Mallin toteutuksessa hyödynnettiin selvää kohdetta Komento-mallin käytölle ja refaktoroitiin Arenagamen liikkuminen sekä kaiken muun pelaajan antaman syötteen tapahtuminen komentoina.

Arenagame lukee syötteen tavalla, joka ei ole Unityn kannalta kovinkaan tehokasta tai selkeää. Liikkumisen ja hyppäämisen tarkistus tapahtui eri osassa ohjelmakoodia omassa Update-päivitysmetodissaan kuin esimerkiksi ampumisen. Turhaa Update-metodin käyttöä tulisi välttää, sillä liian moni turha kutsu kuormittaa suoritinta [28]. Hajautettu koodi tarkoitti myös sitä, että tällä mallilla uuden toiminnallisuuden lisääminen olisi hyvin vaikeaa. Jos pelin pelaajat haluaisivat käyttää pelaamiseen näppäimistön ja hiiren sijaan peliohjainta, pitäisi jokaisen syötteen tarkistamisen lisäksi tarkistaa myös peliohjaimen syöte.

Ohjelmaan esiteltiin ratkaisuksi Komento-mallia, joka kapseloisi hyppäämisen ja ampumisen komennoiksi. Syötteen tarkistamisessa valitun syötteen mukainen komento toteutettaisiin. Kaikki syötteen tarkistamiset pystyttäisiin siirtämään pois saaja-luokista ja keskittämään ne sille omistettuun luokkaan. Uusien ohjaimien lisääminen mahdollistuisi helpommin, ja pelaaja pystyisi muokkaamaan nappulan toiminnallisuutta pelin ollessa käynnissä; esimerkiksi hyppääminen tapahtuisi näppäimistöllä välilyönnin sijaan x-napista.

Toteutus aloitettiin määrittelemällä tarvittavat komennot ja abstrakti Command-luokka. Esimerkkikoodissa 3 nähdään sekä abstrakti komento että konkreettinen hyppäämisen toteuttava komento. Tässä tapauksessa CharacterControls-luokka toimii saajan roolissa. Saaja voidaan myös vaihtaa toiseen, jos samaa komentoa halutaan käyttää sekä pelaajan että vihollisen hyppykomennon määrittelyyn. Yhteensä konkreettisia komento-luokkia määriteltiin viisi.

```
abstract class Command
{
    public abstract void execute();
};
class JumpCommand : Command
{
    CharacterControls chara;

    public JumpCommand(CharacterControls ch)
    {
        chara = ch;
    }
    public override void execute()
    {
        if(chara.isGrounded())
            chara.charaJump();
    }
};
```

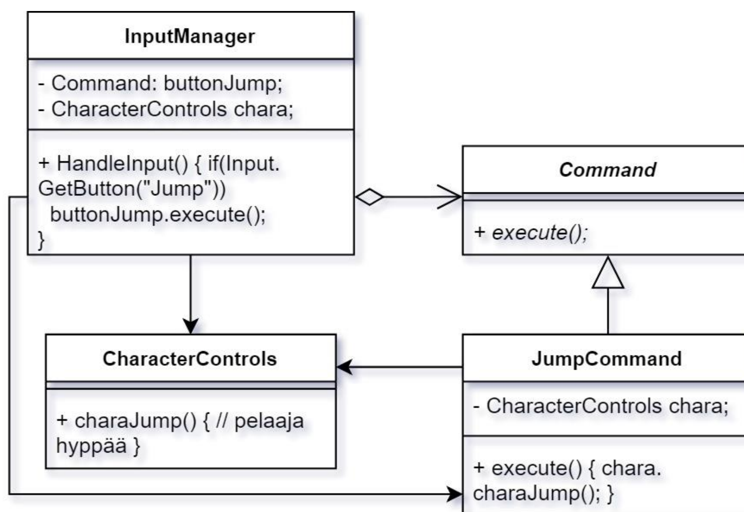
Esimerkkikoodi 3. Abstraktin Command-luokan ja yhden komennon määrittely.

Aikaisemmat syötteen tarkistamiset poistettiin Update-metodeista, ja koko ohjelman tarvitsemien syötteiden lukemista varten luotiin oma InputManager-luokka. InputManager on lisäksi vastuussa komento-luokkien luomisesta ja varsinaisesta komennon toteuttamisen kutsumisesta. Kuva 13 havainnollistaa UML-kaaviolla Arenagameen implementoitua Komento-mallia, mutta kaikkia viittä komentoa ei kuvata tilan säästämiseksi. Vertaamalla kuvan 12 ja kuvan 13 toteutuksia Komento-mallissa voidaan huomata muutamia eroja. Insinöörityön toteutuksessa varsinainen asiakasohjelma ja kutsuja ovat yhdistettynä InputManager-luokkaan. Jos komennot luotaisiin muualla, olisi siinä tapauksessa InputManager kutsuja. Komentojen vähäisen määrän vuoksi Kutsuja-olio ei olisi tuonut lisää parannusta muunneltavuuteen.

Mallin seuraukset ohjelmalle

Komento-mallin soveltaminen toi ohjelmistoon peräti seitsemän uutta luokkaa, joista viisi oli kapseloituja komentoja. On selvää, että arkkitehtuuri kasvoi, mutta samalla syötteen lukeminen eriytettiin ulos luokista, joille se vastuu ei kuulunut. Kasvattamalla hierarkiaa ohjelmasta tuli helpommin muunneltava ja ylläpidettävä. Jos pelin kehityksessä nousee tarve lisätä luettavan syötteen määrää, voidaan komento-luokkien määrää lisätä helposti. Vanhemmassa hierarkiassa ohjelmoija olisi joutunut tekemään muutoksia useampaan luokkaan, jos esimerkiksi peliohjaimen tukeminen oli haluttu ominaisuus. Toisaalta voidaan argumentoida, että ohjelman suuri refaktorointi lisäämällä luokkien määrää ei

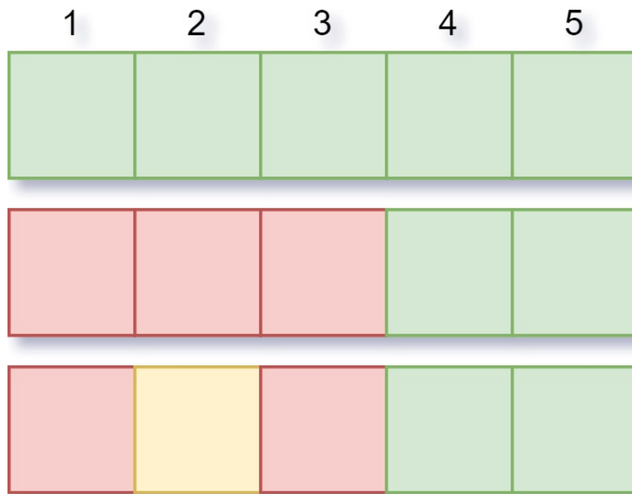
ollut tehokasta ajankäyttöä, jos lopputuloksena saavutettiin vain vähän uutta toiminnallisuutta.



Kuva 13. Keskeisimmät luokat Komento-mallin toteutuksesta Arenagameen.

3.2.2 Oliovarasto-malli

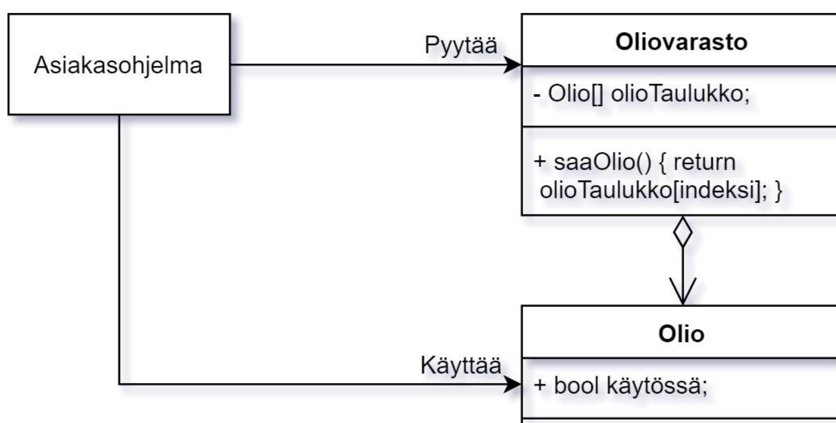
Oliovarasto-malli (engl. Object pool) on pelien optimointiin keskittynyt suunnittelumalli, jonka tavoitteena on kierrättää olioita pelin sisällä, jotta niitä voidaan käyttää tarvittaessa uudelleen. Nimensä mukaisesti Oliovarasto-malli luo kierrätettävistä olioista joukon, josta otetaan tarvittaessa yksi elementti kerrallaan käyttöön ja palautetaan takaisin käytön jälkeen. Motivaatio Oliovaraston käytölle löytyy muistinhallinnan ongelmista. Kuva 14 esittää muistin pirstoutumisesta aiheutuvaa ongelmaa. Vihreät neliöt esittävät yhtä vapaata lohkoa tietokoneen muistia. Uuden olion luonnin yhteydessä muistista otetaan tarpeeksi tilaa olion käyttöä varten. Punaiset neliöt ovat nyt vain olion käytössä olevaa tilaa, joka vie kolme lohkoa. Jos kuvan esimerkin tapauksessa vapautuu tilakuutio kohdasta 2, jää jäljelle vielä kaksi täytettyä lohkoa. Valitettavasti uuden samanlaisen olion luominen ei tällöin ole mahdollista, koska vaikka vapaata tilaa on kolmen lohkon verran, on tila nyt pirstoutunut eri puolille muistia eikä, uutta oliota näin ollen voida luoda. [12.]



Kuva 14. Esimerkki dynaamisen muistinhallinnan vuoksi tapahtuvasta muistin pirstoutumisesta. Ylhäältä alaspäin on kuvattu kolme eri vaihetta.

Yleensä pelit luottavat automaattiseen roskienkeräykseen (engl. Garbage collection) muistinhallinnan ratkaisemiseksi. Tällä tarkoitetaan pirstoutuneiden muistilohkojen palauttamista takaisin ohjelman käyttöön. Tätä tekniikkaa hyödynnetään myös Unityssa [29]. Jotkin ohjelmointikielet, kuten C++, luottavat manuaaliseen muistinhallintaan, jolloin olioiden oikea luominen ja tuhoaminen on ohjelmoijan vastuulla. Tämä voi tuoda ongelmia peleissä, jossa olioita täytyy luoda ja tuhota runsaasti lyhyellä aikavälillä. Pirstoutumisesta voi pahimmillaan olla seurauksena pelin kaatuminen, kun tietokoneen muisti täyttyy pirstoutuneista muistin osista. Oliovarasto-mallin toiminta perustuu tarvittavan muistin varaamiseen jo pelin käynnistymisen aikana. Varattuun muistiin luodaan tarvittava määrä sellaisia olioita, joita tavallisesti luodaan ja tuhotaan paljon, esimerkiksi ammuksia. [12.]

Kuva 15 esittää Oliovarasto-mallin rakenteen. Oliovarasto tallentaa ohjelman alussa ohjelmoijan arvioiman määrän Olio-luokan olioita taulukkoon. Asiakasohjelman pyytäessä yhtä oliota käyttöönsä antaa Oliovarasto-luokka taulukostaan ensimmäisen ei-käytössä olevan olion ja merkitsee sen käytössä olevaksi. Oliovarasto voi antaa taulukostaan vain sellaisia olioita, jotka eivät ole jo käytössä. Muussa tapauksessa oliovarasto saattaisi palauttaa olion, joka oli jo toisen asiakasohjelman käytössä, ja näin luoda mahdollisia ongelmia. Kun Asiakasohjelma ei enää käytä oliota, se merkitään takaisin ei-käytössä olevien olioiden joukkoon, josta se voidaan tarvittaessa taas palauttaa käyttöön. [12.]



Kuva 15. UML-kaavio Oliovarasto-mallista. Asiakasohjelma pyytää Oliovarastolta käyttöönsä Olion-luokka olion, jonka varasto palauttaa listaltaan, jos kyseinen olio ei ole jo käytössä.

Mallin käyttö pelissä

Oliovarasto-mallilla voidaan peleissä saavuttaa huomattavia parannuksia suorituskykyyn ja muistinhallintaan. Oliovarasto-mallin avulla varastoitava olio voi käytännössä olla minkälainen olio tahansa, joten suunnittelun kannalta valinnan varaa on paljon. Oliovarasto-mallin käyttö onkin lähinnä ohjelmiston arkkitehtuuriin liittyvä kysymys. Ohjelmoijan on tunnistettava ne tilanteet, joissa oliovaraston käyttö on parempi vaihtoehto kuin roskienkeräykseen luottaminen. On myös arvioitava oikein tarvittavien olioiden määrä, jotta varasto tuottaa maksimaalisen hyödyn pelille. Runsas määrä käyttämättömiä olioita varastossa vie turhaan tilaa.

Arenagamen alkuperäinen ampumista hallinnoiva koodi luotti perinteiseen ammusten luomiseen laukaisun yhteydessä, minkä jälkeen ammukset tuhottiin osuman sattuessa tai viimeistään muutaman sekunnin päästä. Malli refaktoroitiin Arenagameen korvaamaan sen vanha toteutustapa, koska teoriassa varaston edut ylittävät sen haitat. Tavoitteena oli luoda peliin Oliovarasto, johon luodaan tarvittava määrä ammuksia pelaajien käyttöön. Moninpelinä Arenagame hyötyy kaikkien pelaajien käytössä olevasta varastosta, koska luontiin ja tuhoamiseen liittyvää tietoa ei myöskään tarvitse lähettää verkon kautta muille pelaajille.

Mallin toteutus Arenagameen alkoi oliovaraston määrittelyllä. Varastoa varten tuli arvioida peliin liittyviä faktoja. Koska pelialue itsessään ei ollut kovin suuri, ei varastosta

kannattanut tehdä merkittävän suurta. Ammukset ehtivät osua seinään tai viholliseen muutamassa sekunnissa vaikka alueen toiselta laidalta. Toisaalta suuremman varaston puolesta puhui ajatus useasta pelaajasta ampumassa samaan aikaan. Ei käytössä olevien ammusten loppuminen kesken johtaisi pelaajan ampuvan tyhjää, kunnes varastossa olisi käytössä taas ammus. Runsaasti ammuksia käyttävät sarjatuliaseet söisivät puolestaan pienen varaston tyhjäksi hetkessä. Liian suuri varasto on Oliovarasto-mallia implementoitaessa turvallisempi vaihtoehto, kun tarvittava määrä olioita ei vielä ole tiedossa.

Esimerkkikoodissa 4 on esitetty ObjectPool-luokan rakenne. Pelin alussa Start-funktio alustaa luokan listaolion, johon tallennetaan haluttu määrä olioita. Arenagamen tapauksessa tallennetut oliot olivat ammuksia. getObject-metodia kutsumalla palautetaan ensimmäinen Unityn hierarkiassa ei-aktiivisena oleva ammusolio. Jos kaikki oliot ovat käytössä, palautetaan tyhjä olio, jotta peli ei kaatuisi. Käytännössä tyhjän olion palauttamista halutaan välttää, mutta jos peli ei jostain syystä toimisi oikein, tyhjä olio mahdollistaa pelin toiminnan ongelmasta huolimatta.

```
public GameObject ammo;
public int pooledAmount = 50;
private List<GameObject> objectsInPool;
void Start ()
{
    objectsInPool = new List<GameObject>();
    for (int i = 0; i < pooledAmount; i++)
    {
        GameObject obj = Instantiate(ammo, gameObject.transform);
        obj.SetActive(false);
        objectsInPool.Add(obj);
    }
}
public GameObject getObject()
{
    for (int i = 0; i < objectsInPool.Count; i++)
    {
        if (!objectsInPool[i].activeInHierarchy)
            return objectsInPool[i];
    }
    return null;
}
```

Esimerkkikoodi 4. ObjectPool-luokka. Alussa luodaan tarvittava määrä olioita, ja getObject-metodi palauttaa ei-aktiivisen olion listalta.

Toimiva Oliovarasto ei tarvitse muuta kuin esimerkkikoodissa 4 esitellyt asiat, mutta varastoitaviin olioihin on myös tehtävä muutoksia. Arenagamen BaseProjectile-luokkaan

lisättiin OnDisable- ja OnActivate-funktioita hyödyntämällä tieto, onko ammusolio aktiivinen vai ei. Tällä tavalla ammus pystyttiin palauttamaan ei-aktiiviseksi.

Mallin seuraukset ohjelmalle

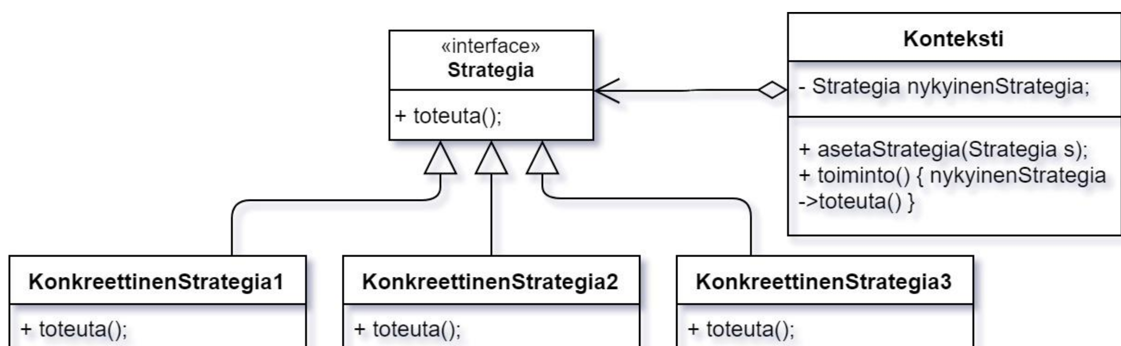
Oliovaraston käyttö ei juurikaan monimutkaistanut alkuperäistä ohjelmakoodia. Mallia varten luotiin vain yksi uusi luokka ja refaktoroitiin olemassa olevaa koodia. Oliovaraston todellinen hyöty tuli selvittää tutkimalla muistia ja seuraamalla suorituskykyä. Tulokset eivät tukeneet Oliovaraston-mallin käyttöä ainakaan suoraan. Kuormittamalla peliä suurella määrällä kutsuja ei saatu merkittävää eroa alkuperäisen ammunnanhallinnan ja Oliovarasto-malliin perustuvan järjestelmän välille, vain 2 fps. Toisaalta mahdollista monen pelaajan tilannetta oli hyvin vaikeaa simuloida oikein, mutta lukuisista testeistä huolimatta tämä oli tulos. Tulos osoittaa, että Arenagame ei juurikaan hyötynyt kevyiden olioiden kuten ammusten varastoimisesta, vaan tietokone pystyi hallitsemaan nopean luomisen ja poistamisen aivan yhtä tehokkaasti. Samankaltainen tilanne oli myös muistinhallinnan puolella. Vaikka ohjelmisto hyötyi siitä, että pirstoutumista ei tapahtunut, ei muistin käytön eroja huomannut, koska roskienkeräys toimi sulavasti.

3.2.3 Strategia-malli

Strategia-malli (engl. Strategy) on käyttäytymismalli samaan tapaan kuin Komento-malli. Molempien tavoitteena on tehdä ohjelman rakenteesta abstraktimpaa, mutta eri tavoilla. Komento-malli luo halutusta operaatiosta olion, mutta Strategia-malli luo tietynlaisia olioksi muutettuja suunnitelmia, joiden mukaan kohteena oleva olio toimii. Komennot voidaan määritellä vapaammin olemaan periaatteessa minkälaisia hyvänsä, mutta strategiat yleensä kuvaavat enemmän erilaisia algoritmeja, joita käytetään saavuttamaan jokin haluttu lopputulos. Rakenteeltaan Strategia-malli muistuttaa jossain määrin Komponentti-mallia, mutta vielä enemmän Tila-mallia. Nystromin mukaan Komponenttien ja Strategioiden välinen ero voi hämärtyä, kun konkreettinen komponentti ei tiedä tilastaan mitään. Yleensä Komponentit tarvitsevat toimiakseen enemmän informaatiota, joten ne luovat omanlaisensa identiteetin ohjelmassa. [6; 12.]

Tarkastellaan seuraavaksi kuvan 16 kaaviota. Vertaamalla Strategia-mallin esitystä kuvan 10 Tila-mallin kaavioon, huomataan aiemmin mainitut yhteneväisyydet. Molemmat

mallit sisältävät Kontekstin ja rajapinnan, jota kautta konkreettiset luokat määritellään. Strategioita on kuitenkin ajateltava ensisijaisesti algoritmeina, joilla sisäiset muuttujat sisältävät tietoa algoritmin tilasta. Loppujen lopuksi näillä muuttujilla on merkitystä vain algoritmin käynnissä olon aikana. Strategiat ovat lisäksi tietämättömiä kohteena olevasta Kontekstista, mutta Tila-mallin konkreettiset tilat yleensä sisältävät tästä referenssin. Toisaalta Tila-mallin konkreettiset tilat luodaan reagoimaan tietynlaiseen tilaan pelissä ja toimimaan sen mukaan, kun taas konkreettiset strategiat ovat erilaisia tapoja suorittaa tietty tehtävä. Tästä on esimerkki kuvassa 4, jossa shakkinappulan liikkumisen toteuttava liikeToteutus-strategia on kapseloitu erilaisiksi strategioiksi. Ne suorittavat tietyn tehtävän, mutta omalla tavallaan.



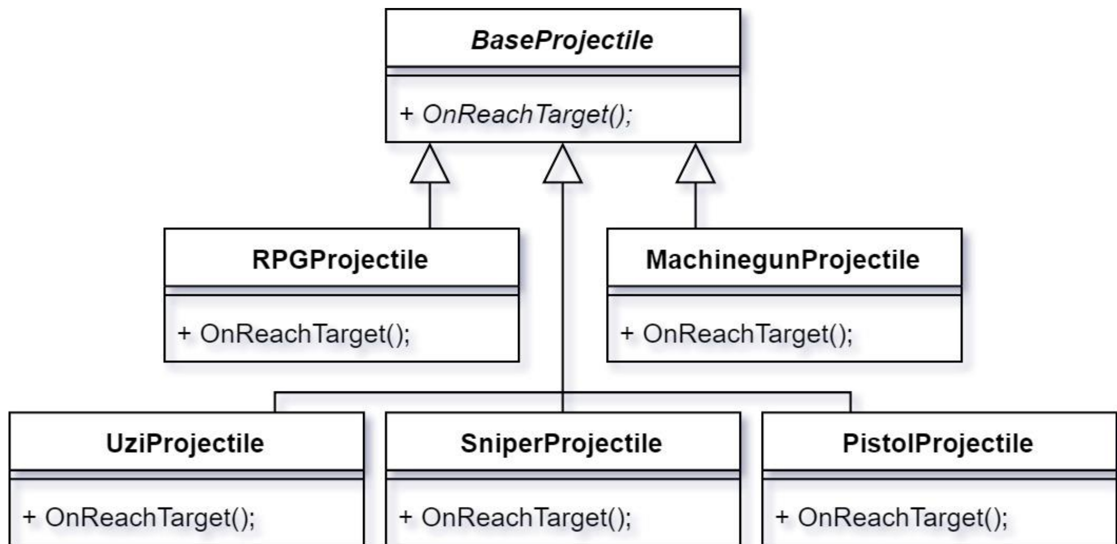
Kuva 16. Strategia-malli esitettynä UML-kaaviona. Rakenne on hyvin samankaltainen kuin Tila-mallin rakenne.

Mallin käyttö pelissä

Shakkipelin tapauksessa liikkumisen toteutus ei ole ainoa asia, joka voidaan kapseloida strategiaksi. Esimerkiksi tietokonevastustajan vaikeustasoa voitaisiin säätää antamalla tekoälylle erilaisia strategioita parhaan mahdollisen siirron laskemiseksi. Joka tapauksessa Strategia-malli on hyödyllinen vain silloin, kun monia erilaisia algoritmeja tarvitaan samankaltaiseen tehtävään. Jos shakkipeli käyttää vain yhdenlaista ja pelikohtaista tekoälyä, ei Strategia-malli tuo ohjelmalle mitään arvoa.

Arenagamen tapauksessa halu käyttää Strategia-mallia lähti pelin tavasta käsitellä ammusten ohjelmakoodin logiikkaa. Alun perin pelin tekijät olivat poistaneet ammusten näkymisen pelissä kokonaan lukuun ottamatta ontelokranaattia. Pelin sivuilla kehotetaan ottamaan myös muut ammukset haluttaessa käyttöön ottamalla olemassa oleva

RPGProjectile-luokka esimerkiksi. Käytännössä asia onnistuu kopioimalla jokaiseen pelin viiteen aseen ammukseen sama koodi. Tämä toimii, ja se varmasti riittäisi, huolimatta täysin samasta koodista useassa eri luokassa, jos peliä ei kehitettäisi enempää. Tässä muodossa kuvan 17 ammusten luokkahierarkia muistuttaa kuvan 3 tapausta. Konkreettiset aliluokat perivät abstraktin BaseProjectile-luokan, jonka metodi OnReachTarget huolehtii mm. ammuksen törmäystarkistuksesta.



Kuva 17. Arenagamen ammusten luokkahierarkia UML-kaaviona ennen Strategia-mallin implementoimista.

Ratkaisuksi esiteltiin Strategia-mallia, joka kapseloisi OnReachTarget-metodin algoritmiksi luokkien ulkopuolelle omaksi strategiakseen. Prosessi alkoi määrittelemällä DamageBehaviour-rajapinta, joka sisältää ammuksen alustamisen ja varsinaisen vahingon laskemisen metodit. Konkreettinen strategialuokka kapseloi OnReachTarget-metodin sisällön. BaseProjectile-luokan tehtäväksi määriteltiin strategian asettamisesta huolehtiminen ja sen käytöstä huolehtiminen. Luokasta tuli siis strategian konteksti. Kun ammusta käytetään, toimii nyt törmäystarkistus ja vahingon laskeminen sillä hetkellä aktiivisena olevan strategian mukaan. Esimerkkikoodissa 5 on eritelty tärkeimmät osat Strategia-mallin toteutuksesta Arenagamessa. Rajapinnan tarjoama Init-metodi vastaa DamageBasic-strategian alustuksesta, ja varsinainen strategia on TakeDamage-metodi. BaseProjectile-luokka asettaa strategian luomisensa alussa, ja peli käyttää tätä strategiaa oletuksena.

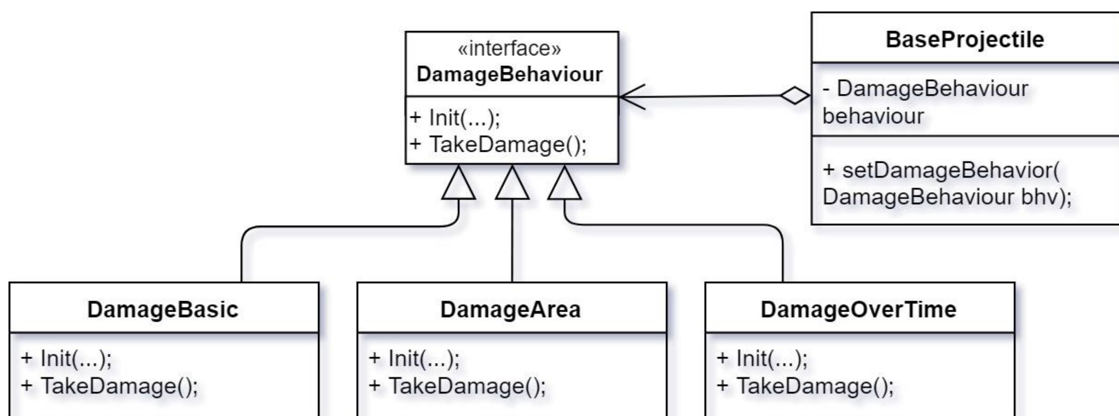
```
DamageBehaviour.cs
public interface DamageBehaviour
{
    void Init(int damage, string attackername, Vector3 target, Vector3 hitNor-
mal);
    void TakeDamage();
}
public void Init(int damage, string attackername, Vector3 target, Vector3 hit-
Normal)
    {
        _damage = damage;
        _attackerName = attackername;
        _target = target;
        _hitNormal = hitNormal;
    }
public void TakeDamage()
{
    if (NetworkManager.Instance.IsServer)
    {
        Collider[] hitColliders = Physics.OverlapSphere(transform.position,
radius);
        foreach (Collider hitCollider in hitColliders)
        {
            HealthSystem hp = hitCollider.GetComponent<HealthSystem>();
            if (hp)
            {
                float actualDamage = _damage - (Vector3.Distance(hp.trans-
form.position, transform.position) * 10);

                hp.TakeDamage((int)actualDamage, _attackerName, _target, _hit-
Normal);
            }
        }
    }
}
}
BaseProjectile.cs
public virtual void Start()
{
    DamageBasic behaviour = gameObject.AddComponent<DamageBasic>();
    behaviour.Init(damage, attackerName, target, hitNormal);
    setDamageBehavior(behaviour);
}
```

Esimerkkikoodi 5. Strategia-mallin koodi esitettynä pääpiirteissään. Toteutus on jaettu kahteen eri tiedostoon: DamageBehaviour.cs ja BaseProjectile.cs.

Mallin seuraukset ohjelmalle

Strategia-mallin käyttö ratkaisi saman koodin käytön useassa luokassa. Toisaalta saman lopputuloksen olisi voinut saada toisellakin tapaa. Siirtämällä OnReachTarget-metodin konkreettinen toteutus BaseProjectile-luokkaan ja muuttamalla luokan normaaliksi abstraktin sijaan olisi konkreettiset ammusluokat tehty tarpeettomiksi yhtä tehokkaasti. Ongelmallinen tämä menettely olisi ollut siinä vaiheessa, jos ammusten käyttäytymistä olisi pitänyt muuttaa. Jos ammuksen tapoja aiheuttaa vahinkoa täytyisi lisätä, saattaisi

BaseProjectile-luokka kasvaa ajan myötä hallitsemattoman suureksi. Fantasiapeleistä hyvin tuttu ”vahinko ajan myötä” tai ”aluevahinko” kasvattaisivat kokoa. Lisäksi, jos peliin lisättäisiin erikoisempia aseita, olisi alkuperäinen vahingon laskun toteutus myös muutettava tukemaan niitä. Käyttämällä Strategia-mallia varmistetaan tässä tapauksessa liikkumavara tulevaisuutta ajatellen. Uusien vahinkotyyppien lisääminen peliin on hyvin helppoa: luodaan uusi strategia vahingon laskemista. Toisaalta uuden strategian lisääminen tulee mahdollisesti kasvattamaan muutoksen tarvetta algoritmin ulkopuolella, jos kyse on erikoisemmasta strategiasta. Loppujen lopuksi peliin lisättiin kaksi uutta strategiaa. Strategiaa voidaan muuttaa pelaajan valinnan mukaan, ja ammus käyttää valittua strategiaa vahingon laskemiseksi. Kuva 18 esittää kaavion lopputuloksesta. DamageBasic-luokka toteuttaa alkuperäisen algoritmin, kun taas DamageArea- ja DamageOverTime-luokat toteuttavat lopputuloksen eri vaikutuksilla.



Kuva 18. UML-kaavio peliin toteutetun Strategia-mallin lopputuloksesta.

3.3 Tutkimusten ja oman työn arviointi

Insinöörityön käytännön vaiheen tuloksena valmistunut tutkimus suunnittelumallien käytöstä pelimoottorissa toi tietoa pelimoottorien toiminnasta ja siitä kuinka laajasti malleja hyödynnetään myös peliteollisuuden piireissä. Pelien ja pelimoottorien arkkitehtuuriin keskittyneet suunnittelumallit ovat keskeisessä osassa pelien toiminnallisten algoritmien suunnittelua. Vaikka kaikki pelit eivät välttämättä käytä esimerkiksi Komponentti-mallia, on tietous algoritmin olemassaolosta tärkeää, jotta mahdollisia seurauksia ohjelmalle voidaan arvioida mahdollisimman tarkasti.

Unity-pelimoottorin suunnittelumallien käyttöä arvioitaessa oli kuitenkin luotettava pääasiassa ulkopuolisiin lähteisiin ja verrattava työkaluja suunnittelumallin dokumentaatioon niiden toiminnasta ohjelmassa. Prefab-työkalu hyödyntää todennäköisesti Prototyyppirajapinta-mallia, mutta koska vain C#-koodin osuus Unity-moottorista on julkisesti saatavilla, on asian sataprosenttinen vahvistaminen vaikeaa.

Työssä valmistui lisäksi kokeellinen kolmen erilaisen suunnittelumallin refaktoroiminen avoimen lähdekoodin peliin. Kaikki mallit toimivat halutulla tavalla, ja niiden vaikutukset pelille olivat paikoin yllättäviä, mutta pääosin odotettuja. Ehkä suurin yllätys oli Oliovarasto-mallin vähäinen vaikutus suorituskykyyn ja muistin käsittelyyn. Vaikutus olisi todennäköisesti suurempi, jos varastoitava olio olisi monimutkaisempi, jolloin tavallinen luominen ja poistaminen olisi selkeästi kalliimpi ratkaisu. Oliovaraston hyvä puoli on lisäksi sen joustavuus varastoitavan olion suhteen: periaatteessa mikä hyvänsä olio voidaan varastoida. Varastoitavien olioiden määrän on oltava riittävän suuri ja olion rakenteen riittävän monimutkainen suorittimelle, jotta varaston hyödyt tulisivat parhaiten esille.

Muiden mallien kohdalla tilanne ei ollut yhtä jyrkkä. Komento- ja Strategia-mallit ovat molemmat GoF-malleja, ja kuten jo luvussa 2 Koskimies ja Mikkonen toteavat, GoF-mallien tyypillinen tavoite on tehdä ohjelmasta helposti muokattava ja uudelleenkäytettävä, mutta usein suorituskyvyn kustannuksella. Mielestäni nämä laatuominaisuudet toteutuivat, mutta suorituskyvyn laadussa ei ollut eroa alkuperäiseen toteutukseen verrattuna. Komento-mallin kohdalla poistettiin suorituskykyä heikentäviä elementtejä, kuten turhia Update-metodin kutsuja, mutta niitä myös lisättiin, sillä esimerkiksi Strategia-malli luottaa toiminnassaan myöhäiseen sidontaan, mikä mahdollistaa oikean strategian käytön pelin ollessa käynnissä, mutta on myös samalla Koskimiehen ja Mikkosen mukaan hidastava tekijä. Toisaalta he toteavat vaikutusten olevan yleensä minimaalisia, mikä oli mielestäni totta. [11.]

On kuitenkin huomattava, että jos kyseessä olisi mobiilialustalle julkaistava peli, pitäisi mallien suorituskyvyn ja muistinhallinnan vaikutusta arvioida tarkasti uudelleen. Mobiilipohjaisten laitteiden komponentit eivät pysty samanlaiseen suorituskykyyn kuin esimerkiksi tietokoneet, joten suunnittelumallien käytöstä ohjelman arkkitehtuurissa on tehtävä tarkempaa esityötä, jotta vaikutukset pystytään kartoittamaan oikein.

Suurimman haasteen suunnittelumallien käytölle Arenagamessa loi juuri vaikutusten tarkka arvioiminen. Jotta mallin käyttäminen olisi mielekästä, olisi pelin ohjelmakoodista löydyttävä kohta tai ongelma, jonka ratkaisuun olisi järkevää käyttää suunnittelumallia. Luvussa 3 mainittua ylisuunnittelua haluttiin välttää riittävän harkinnan avulla. Tästä huolimatta kaikkia vaikutuksia ei osattu ennustaa, kuten Oliovaraston tapauksessa. Harkintaa vaikeutti lisäksi arvioitavien mallien suuri määrä: pelkästään GoF-malleja on 23, ja vaikka mallien tavoitteet menevät osittain päällekkäin, on yksinkertaisesti vaikeaa osata arvioida mallien toimivuutta ilman perusteellista analyysiä. Kun kohteen ratkaisuun löytyi oikeanlainen malli, oli soveltaminen yllättävän helppoa, vaikka Unity-ympäristön käyttö toi oman haasteen.

4 Yhteenveto

Insinöörityössä tutkittiin hyvän oliopohjaisen ohjelmakoodin erilaisia yleisesti hyväksyttyjä ohjeita ja kaavoja, sekä ohjelman arkkitehtuurin suunnittelussa apuna käytettävien suunnittelumallien käyttöä peliohjelmoinnissa. Tuloksena syntyi kattava läpileikkaus keskeisimpiin ohjelmointia koskeviin normeihin ja periaatteisiin, kuten refaktorointiin tai hyvään muuttujan nimeämistyyliin. Nämä periaatteet auttavat ohjelmoijaa tuottamaan koodia, joka on helposti luettavaa, mutta myös vähemmän altista virheille.

Työssä tutustuttiin suunnittelumalleihin ja niiden käyttöön ohjelmistoissa, peleissä ja pelimoottoreissa. Mallit välittävät sitä tietoa, jonka kokeneemmat ohjelmoijat ovat keränneet vuosien aikana, ja esittävät sen suunnittelumallin muodossa. Parhaimmillaan suunnittelumallit toimivat ohjelman suunnittelun apuvälineenä, auttavat ohjelmoijaa ratkaisemaan ongelman tehokkaasti ja tekevät koodista kestävämmän ja joustavamman. Huonoimmillaan suunnittelumallit saattavat olla ylitehokas ratkaisu, mutta niitä käytetään silti, vaikka ohjelma toimisi paremmin ilman. Tulos myös osoitti, että suunnittelumallit ovat vahvassa roolissa pelien algoritmeissa, kuten Pelisilmukka-malli, jonka lähes jokainen peli jossain muodossa tarvitsee.

Työssä tutkittiin myös suunnittelumallien käyttöä Unity-pelimoottorin toiminnoissa ja työkaluissa ja toteutettiin kolme suunnittelumallia avoimen lähdekoodin pelissä Arenagame. Unityn todettiin käyttävän hyväkseen monia suunnittelumalleja, vaikka asiaa ei voi täysin

vahvistaa. Suunnittelumallien elementit ovat kuitenkin työkaluissa selvästi esillä. Laajempi tutkimus Unityn C#-pohjaisesta lähdekoodista luultavasti paljastaisi mallien laajempaa käyttöä ottaen huomioon viiden todetun mallin todennäköisen hyödyntämisen.

Peliin toteutettiin kolme suunnittelumallia sillä olettamuksella, että ne toisivat ohjelmakoodiin enemmän joustavuutta tukemaan mahdollisia muutoksia ja parantamaan mm. suorituskykyä ja luettavuutta. Kaikkien mallien kohdalla odotukset täyttyivät pääpiirteissään, mutta odottamatonta oli heikko vaikutus suorituskykyyn myös erityisesti sitä parantamaan tarkoitetun suunnittelumallin toteutuksen jälkeen. Suunnittelumallien toteuttaminen käytännössä opetti paljon pelin ohjelmakoodin suunnittelusta ja erityisesti erilaisten lähestymistapojen vaikutusten arviointia. Tulevaisuudessa ongelmien ratkaisuksi soveltuvien suunnittelumallien käyttäminen on lisääntyneen kokemuksen myötä helpompaa, ja uusien mallien käyttöön ottamisen kynnys on nyt matalampi kuin aikaisemmin.

Lähteet

1 Martin, Robert C. 2008. Clean code. New Jersey: Prentice Hall.

2 Fowler, Martin. 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

3 Hunt, Andrew & Thomas, David. 2000. Pragmatic Programmer: from journeyman to master. Massachusetts: Addison-Wesley.

4 Shvets, Alexander. 2019. Refactoring. Verkkoaineisto. <https://refactoring.guru/refactoring/>. Luettu 15.1.2019.

5 Martin, Robert C. 2000. Design Principles and Design Patterns. Verkkoaineisto. <https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf>. Luettu 21.1.2019.

6 Gamma, Erich; Vlissides, John; Johnson, Ralph & Helm, Richard. 1994. Design Patterns. Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley.

7 Martin, Robert C. 2014. The Single Responsibility Principle. Verkkoaineisto. <https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html>. Luettu 20.1.2019.

8 Janssen, Thorben. 2018. SOLID Design Principles Explained. <https://stackify.com/solid-design-principles/>. Luettu 21.1.2019.

9 Freeman, Eric; Robson, Elisabeth; Sierra, Kathy & Bates, Bert. 2004. Head First Design Patterns. Sebastopol: O'Reilly.

10 Alexander, Christopher; Ishikawa, Sara; Silverstein, Murray; Jacobson, Max; Fiksdahl-King, Ingrid & Angel, Shlomo. 1977. A Pattern Language. New York: Oxford University Press.

11 Koskimies, Kai & Mikkonen, Tommi. 2005. Ohjelmistoarkkitehtuuri. Verkkoaineisto. <http://www.cs.tut.fi/~ohar/OHAR2012_13-KIRJA-KoskimiesMikkonen.pdf>. Luettu 13.2.2019.

12 Nystrom, Robert. 2014. Game Programming Patterns. Verkkoaineisto. <http://gameprogrammingpatterns.com/>. Luettu 26.1.2019.

13 2019 Video Game Industry Statistics, Trends & Data. 2019. Verkkoaineisto. WePc. <https://www.wepc.com/news/video-game-statistics/>. Päivitetty 1.4.2019. Luettu 2.4.2019.

14 Björk, Staffan & Holopainen Jussi. 2003. Game Design Patterns. Digital Games Research Conference 2003, s. 1–2. University of Utrecht.

15 Grand Theft Auto V. 2013. Verkkoaineisto. Wikipedia. <https://fi.wikipedia.org/wiki/Grand_Theft_Auto_V>. Luettu 16.3.2019.

16 Karmali Luke. 2013. Grand Theft Auto V Gets A September Release Date. Verkkoaineisto. IGN. <https://www.ign.com/articles/2013/01/31/grand-theft-auto-v-gets-a-september-release-date>. 31.1.2013. Luettu 5.4.2019.

17 Madhav, Sadjay. 2013. Game Programming Algorithms and Techniques. Indiana: Addison-Wesley.

18 Witter, Koen. 2009. The Game Loop. Verkkoaineisto. <https://dewitters.com/dewitters-gameloop/>. Luettu 17.3.2019.

19 Maslow, Abraham. 1966. The Psychology of Science: A Reconnaissance. Gateway Editions.

20 Unity Game Engine Review. 2018. Verkkoaineisto. Unity3d Course. <http://www.unity3dcourse.com/2018/11/05/unity-game-engine-review/>. Luettu 20.1.2019.

21 Creating and Using Scripts. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/CreatingAndUsingScripts.html>. Luettu 25.2.2019.

22 MonoBehaviour.FixedUpdate(). 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html>. Luettu 25.2.2019.

23 GameObject. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/class-GameObject.html>. Luettu 25.2.2019.

24 Prefabs. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/Prefabs.html>. Luettu 25.2.2019.

25 Animator Component. 2019. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/class-Animator.html>. Luettu 26.2.2019.

26 Animator Controller. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Manual/class-AnimatorController.html>. Luettu 5.3.2019.

27 Arena Game. 2017. Verkkoaineisto. Footstomp Studios. <https://www.youtube.com/watch?v=BF15LJP7clc&feature=youtu.be>. 4.11.2017. Luettu 20.11.2018.

28 Optimizing scripts in Unity games. Verkkoaineisto. Unity Technologies. <https://unity3d.com/learn/tutorials/topics/performance-optimization/optimizing-scripts-unity-games>. Luettu 3.3.2019.

29 Optimizing garbage collection in Unity games. Verkkoaineisto. Unity Technologies. <https://unity3d.com/learn/tutorials/topics/performance-optimization/optimizing-garbage-collection-unity-games>. Luettu 3.3.2019.

Vuokaavio skriptin elinkaaresta Unityssa