Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Simo-Pekka Salonen

Ohjelmistokehykset peliohjelmoijan näkökulmasta

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

11.5.2020

Tekijä Otsikko Sivumäärä	Simo-Pekka Salonen Ohjelmistokehykset peliohjelmoijan näkökulmasta 30 sivua 11.5.2020
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Pelisovellukset
Ohjaajat	Lehtori Antti Laiho CEO Roni Jokinen

Insinöörityössäni perehdyttiin ohjelmistokehyksien käyttöönotettavuuden Unity-pelikehitysympäristössä ja vertailin UniRx-, PureMVC- ja Unity-ECS -kehyksiä toisiinsa. Niiden omaksumista ja projektin tekemistä niillä vertailtiin, ja niiden käyttöönottoa Unity-ympäristössä analysoitiin.

UniRx on kevyt reaktiivisen ohjelmointimallin kehys, joka lähenee enemmän kirjastoa kuin kehystä. Se seuraa tarkkailija-suunnittelumallia ja käyttää datavirtoja ohjelmointilogiikassa. UniRx helpottaa käyttöliittymän tietojen päivittämistä ja käyttöliittymän yleistä käyttöä, mutta se ei tuo koodikantaan kehyksestä haluttua järjestystä.

PureMVC on MVC-arkkitehtuurin pohjalta tehty vuodesta 2008 ominaisuuslukossa ollut kehys, joka on kehitetty monelle eri kielelle ja ympäristölle. Unity-dokumentaatiota on vähän PureMVC:lle, ja siksi kehyksellä ei saatu työssä tehtyä projektia, mutta sen käytöstä luotiin analyysi.

Unity ECS on Unityn kehittämä ECS-kehys. Unity ECS pyrkii tuomaan tehokkuuden oletuksena pelikehitykseen ja on kehitetty Unity-pelikehitysympäristökeskeisesti. Unity ECS ratkoo monet ongelmat, kuten riippuvuudet ja epätehokkaan koodin, ja vahvistaa Unityn pelikehitysympäristöä. Unity ECS oli työn innostavin kehys ja suositeltava testattavaksi kaikille Unity-ehittäjille.

Insinöörityön tuloksena saatiin vahva suositus Unity ECS -kehyksen testaukseen ja huomattiin, kuinka tärkeä hyvä dokumentaatio ja esimerkkien runsaus on uuden kehyksen opettelussa.

Avainsanat	ohjelmistokehys, arkkitehtuurimalli, MVC, ECS, tarkkailijamalli

Author Title Number of Pages Date	Simo-Pekka Salonen Frameworks from game programmer's point of view 30 pages 8 April 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Professional Major	Game Applications
Instructors	Antti Laiho, Senior Lecturer Roni Jokinen, CEO

The purpose of this study was to compare three different software frameworks, UniRx, PureMVC and Unity ECS, and analyse their usability in Unity game engine environment.

UniRx is a light-weight reactive programming paradigm framework, which is closer to a library than a full framework. UniRx follows observer design pattern and uses data streams for programming logic. UniRx makes user interface data updating and overall user interface usage easier, but it does not bring the wanted framing to codebase.

PureMVC is based on MVC-architecture and has been feature-locked from 2008. It was originally developed for multiple programming languages and -environments. Usage and testing of PureMVC did not produce a working project, but its use is still analysed.

Unity ECS is an ECS-framework developed by Unity. Unity ECS strives to bring performance as a default to game development and it was developed to be Unity centric. Unity ECS resolves many of the problems such as dependencies and unoptimized code and it reinforces Unity game development environment. Unity ECS turned out to be the most promising framework and it is recommended to be tested by all Unity developers.

This final year report also analyses Design Patterns' and Architectural Patterns' background and their relation to frameworks.

The results of this study strongly suggest to further test Unity ECS. The results also confirmed the need for good documentation and a lot of examples to learn a new framework.

Keywords	Design Pattern Architectural MVC ECS Observer

Sisällys

Lyhenteet ja termit

MVC	Model-View-Controller-ohjelmistoarkkitehtuurimalli
ECS	Entity-Component-System-ohjelmistoarkkitehtuurimalli
MVVM	Model-View-ViewModel-ohjelmistoarkkitehtuurimalli
Boilerplate-koodi	Ohjelmistokehyksen tai ohjelmointitavan mukana tuleva koodi, jota on toistettava usein. Tuo koodiin rakennetta, mutta tuntuu turhauttavalta toistolta.
Ohjelmistokehys	Englanniksi: Software Framework, Framework.
Kehys	Tämän insinöörityöraportin kontekstissa yleensä tarkoittaa ohjelmistokehystä.
Unity	Pelimoottori, joka käyttää pääohjelmointikielenä C#:a.
Monobehaviour	Unityn ohjelmointirajapinta, jonka kautta pääsee käsiksi muun muassa päivityssykliin (Update()) ja Unityn fysiikka-komponentteihin).
DOTS	Data-Oriented Technology Stack.
Unity Job System	Unityn kirjasto yksinkertaiselle ja turvalliselle koodin moni-säikeistämiselle.

1 Johdanto

Ohjelmointiala on nuori ja teknologiat kehittyvät nopeasti. Videopelien tekoprosessi on aikaa vievää ja iterointiprosessi on raskas. Tuotteita myös jatkokehitetään tai osia niistä käytetään tulevissa peleissä. Nämä ja monet muut syyt tarkoittavat, että pelikehitysprosessin täytyy olla hyvin suunniteltu ja kehystetty, jotta prosessi voi edetä mahdollisimman helposti.

Insinöörityössä tutkitaan pelientekoprosessin ohjelmointipuolta ja siihen liittyviä ohjelmistokehyksiä, jotka voisivat auttaa niin aloittelevaa kuin ammattipeliohjelmoijaa nostamaan ohjelmoinnin tasoa. Tavoitteena on selvittää, minkälainen ohjelmistokehys olisi helppo opetella, ottaa käyttöön ja käyttää, mutta myös tarpeeksi vahva ja ominaisuuksiltaan monipuolinen ammattikäyttöön. Peliteollisuudessa on huomattu, kuinka voimakas työkalu ohjelmistokehys voi olla, ja tässä työssä pyritään myös saamaan suositus, mitä kehystä aloittelevan peliohjelmoijan kannattaa lähteä opettelemaan.

Työn toimeksiantaja on Lunarbyte, joka pyrkii ”raketoimaan koodisi”, eli se haluaa peliohjelmoijat korkealle tasolle mahdollisimman pian. Työssä käsitellään myös suunnittelu- ja arkkitehtuurimalleja ohjelmoinnissa, koska ne ja niiden käsitteet ovat vahvasti sidottuina ohjelmistokehyksiin. Työn lopuksi pyritään tunnistamaan ohjelmistokehysten hyvät ja huonot puolet ja suosittelemaan kehystä tai kehyksen ominaisuuksia, joilla voi kirjoittaa hyvää, selkeää ja uudelleenkäytettävää koodia.

Työn rakenne ja lähtökohdat

Työssä ohjelmistokehyksiä valitessa päädyttiin kolmeen erilaiseen kehykseen: PureMVC, UniRx ja Unity-ECS. Ohjelmointi tehdään C#-kielellä ja kehitysympäristönä käytetään Unity-pelimoottoria. Yhden kehyksen opetteluun käytetään noin viikko, ja sillä tehdään pieni peliprojekti, ja tämä toistetaan kaikilla kolmella kehyksellä. Työssä keskitytään muun muassa opittavuuteen, tiedostomäärään, toistetun koodin määrään, modulaarisuuteen, koodirivien määrään ja Unity-pelimoottorin kanssa yhteensopivuuteen.

Näen itseni hyvänä koekaniinina, näin aloittelevana Unity-ohjelmoijana, enkä ole ohjelmoinut yhdelläkään edellä mainituista kehyksistä. Lyhyt kehyksen oppimisjakso kärjistää

opittavuuden esille tulon ja tuo kehyksen käyttämisen selkeyden tärkeämpään rooliin. En kuitenkaan halua ehdottaa vain helposti opeteltavaa kehystä, jossa ei olisi tarpeeksi ominaisuuksia, vaan haluan nostaa ohjelmoinnin ammattitasolle.

2 Ohjelmistokehykset

Ohjelmistokehykset ovat työkaluja, joilla ohjelmistoa kehitetään valmiiksi ratkaistulla tavalla. Kehyksiä kuitenkin käytetään myös ketterässä kehityksessä, koska ohjelmistokehyksen ei tarvitse määritellä lopputuotetta mitenkään, vaan prosessia, jolla lopputuote luodaan. Kehyksiä on tiukempisääntöisiä ja avoimempia, ja tämän takia on tärkeää käyttää oikeaa työkalua oikeassa paikassa. Kehyksen päälle voi lähteä tekemään omaa ohjelmaa, ja kehys vain antaa raamit ja työkalut yleisien ominaisuuksien implementointiin koodikirjastoina. Tämän takia kehykset lähtivät web-sovelluksista, joiden luominen oli samanlaista. Tässä kuitenkin on riskinä, että ohjelmointiprosessi eroaa kehyksen toiminnallisuudesta ja silti sitä aletaan käyttää (17).

ElegantCoding-blogissa (5) käydään läpi uudelleenkäytettävyyttä ja ohjelmistokehyksiä. Siinä viitataan useisiin mielenkiintoisiin resursseihin, jotka käyvät syvemmin läpi erilaisia informaatioteorioita ja sitä, miten peruskonseptit ohjelmistokehyksien ja ohjelmistorajapintojen suunnittelussa pitää ottaa huomioon. Mutta niin kuin hyvä rajapinta, hyvä ohjelmistokehys myös on

- helppo oppia
- helppo käyttää, jopa ilman dokumentaatiota
- vaikea käyttää väärin
- helppo lukea ja ylläpitää koodia
- tarpeeksi vahva tarpeisiin
- helppo laajentaa
- oikea käyttötarkoitukseen.

Ohjelmistokehykset usein sekoitetaan koodikirjastoihin (1), mutta kirjastot ovat vain osa kehystä. Ohjelmistokehykseen kuuluu yleensä koodikirjasto, mallipohja ja säännöt, joita seurata (5). Säännöt ovat usein arkkitehtuurimalli, joka taas koostuu yksittäisistä suunnittelumalleista, ja joskus kehysten säännöt ovat vain yksittäinen suunnittelumalli. Raja

suunnittelumallin ja arkkitehtuurimallin välillä on häilyvä, mutta se käydään läpi myöhemmin omissa luvuissaan. Ohjelmistokehykset voivat olla selvästi rakennettu arkkitehtuurimallin päälle, kuten PureMVC ja Unity-ECS, mutta yleensä ne on rakennettu yksittäisistä suunnittelumalleista tai arkkitehtuurimalleja on viety niin pitkälle, ettei niitä voi enää selvästi kategorisoida esimerkiksi MVC- tai ECS-kehykseksi.

Vaikka ohjelmistokehykset ovat vahvoja työkaluja, jokainen projekti ei kuitenkaan niitä tarvitse. Kehyksissä on hyviä ja huonoja puolia, mutta hyvät puolet pitkällä tähtäimellä aina voittavat, kunhan kehys on prosessiin sopiva. Yksi ainoista tapauksista, jolloin kehykset eivät luonnistu, ovat lyhyen ajan projektit uudella tiimillä, joka ei osaa yhteistä kehystä, tai kun itse luontiprosessi on iteroinnin alla. Tällöinkin uuden kehyksen luominen prosessia varten voisi olla edullista.

Aloittelevalle ohjelmoijalle uudet ohjelmistokehykset voivat olla haastavia. Opettelu, uudet kirjastot ja usein myös täysin uusi ajattelumalli voivat olla usein haastavia uudelle tai myös kokeneelle ohjelmoijalle, mutta kuten muidenkin tärkeiden työkalujen opettelu, niiden opettelu kannattaa, varsinkin jos näkee itsensä toistamassa prosessia usein. Ohjelmistokehykset tuovat tiimeille yhtenäisen tavan ohjelmoida ja helpottavat koodin jaettavuutta. Kehys ratkaisee yksinkertaiset ongelmat ja antaa ohjelmoijalle mallipohjan, johon implementoida toteutus (17).

2.1 Ohjelmistokehysten historia ja matka nykypäivään

Rakennus- ja ohjelmistoarkkitehtuuria vertailtiin toisiinsa jo 1960-luvulla (2), mutta ”ohjelmistoarkkitehtuuri”-termi yleistyi vasta 1990-luvulla (3). Ohjelmat alkoivat paisua yhä isommiksi, ja tarvittiin selkoa näihin ohjelmiin. Ohjelmistoarkkitehtuuri lähti datarakenteista ja algoritmeista, mutta jo 1990-luvulla alettiin puhua suunnittelumalleista, ideoista ja konsepteista, jotka voisivat auttaa ajattelemaan ohjelmointia yhteisellä tavalla ja joiden kautta suunnitella ja ohjelmoida (4).

Nykyään, ei pelkästään joka ohjelmistoalalla, vaan melkein myös joka ohjelmistoyrityksellä on oma kehyksensä. Moneen eri ohjelmistonluontiprosessiin on luotu oma kehys ja luontiprosessia on nopeutettu ja yhdennetty huomattavasti. Web-sovellusalaa pidetään

ohjelmistokehysten lähteenä, ja ohjelmistokehyksistä puhuttaessa webkehitys-ala otetaan usein puheeksi. ModelView-arkkitehtuurimalliperhe on yksi tämän hetken yleisimmistä arkkitehtuureista, varsinkin web-kehityksessä (18), ja sen päälle kehyksiä rakennetaan. Nämä korkeamman tason arkkitehtuurimallien rajat häilyvät, kun kehyksiä yhdistellään ja jatketaan niin pitkälle, että kehyksen taustalla olevan mallin kategorisointi ei vastaa vakiintunutta arkkitehtuurimallia.

2.2 Peliala ja ohjelmistokehykset

Pelimoottorit ovat yleensä isoin raami tai kehys, mitä pelientekoprosessissa käytetään, mutta ne harvoin tuovat mukanaan kehystä itse ohjelmointiin. Syntaksi, miten saada asia liikkumaan tai reagoimaan ruudulla, on annettu pelimoottorin puolelta, mutta ei sitä, miten kooditiedostot ja ohjelman datankulku on järjestetty. Tähän tehtävään tarvitaan toinen työkalu.

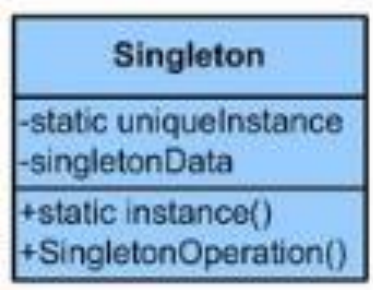
Pelien teko on iteroiva prosessi, ja yhä suurempaan suosioon nouseva ”Games as a service” -malli tarkoittaa, että peliä pitäisi päivittää sen julkaisun jälkeen ja jatkaa sitä palvelun tarjoamisen tavalla (19). Tähän tarvitaan modulaarista, uudelleenkäytettävää ja dynaamista koodia, joka on usein ohjelmistokehysten päätavoitteena. Myös monen yksittäisen pelin julkaiseminen tai pelisarjan jatkaminen hyötyy uudelleenkäytettävyydestä ja muista kehysten keskeisistä ominaisuuksista. Mitä pidempään yritys on toiminut, sitä enemmän resursseja se ovat keränneet näiden yksittäisten ominaisuuksien kautta, joita voi käyttää uusissa peleissä ja parannella ilman, että iso kokonaisuus kärsii (15).

2.3 Suunnittelumallit

Ohjelmoinnin suunnittelumallit ovat parhaiden ohjelmointitapojen kiteytys ajatus- ja ohjelmointimalleiksi. Suunnittelumallit voivat nopeuttaa ohjelman suunnittelu- ja ohjelmointiprosessia, mutta ne voivat myös väärin käytettäessä tuoda suorituskykyongelmia, niin luontiprosessissa kuin ohjelmiston suoritusaikana. Hyviä suunnittelumalleja on monia, ja yhden käyttäminen harvoin estää toisen käytön, joten niistä voidaan kerätä kokonaisuuksia ja nämä kokonaisuudet ovat ohjelmistokehyksiä.

Ohjelmointimallit lähtivät liikkeelle jo vuonna 1966 Christopher Alexanderin arkkitehtuurikonseptina (6), ja niiden ohjelmointipuolen kehitys alkoi vuonna 1987, kun Kent Beck ja Ward Cunningham alkoivat testailla ideaa ohjelmoinnissa (7; 8). Suunnittelumalleja on käytetty jo kauan, mutta niiden määritteleminen ja formalisointi on roikkunut ilmassa monia vuosia. Nykyään suunnittelumallit ovat melkein jokaisen ohjelmistokehittäjän tiedossa, mutta niiden soveltaminen ja yhdistely on oma haasteensa. Ohjelmistosuunnittelumallit on myös jaettu kolmeen pääkategoriaan: luontimallit, rakennemallit ja käyttäytymismallit.

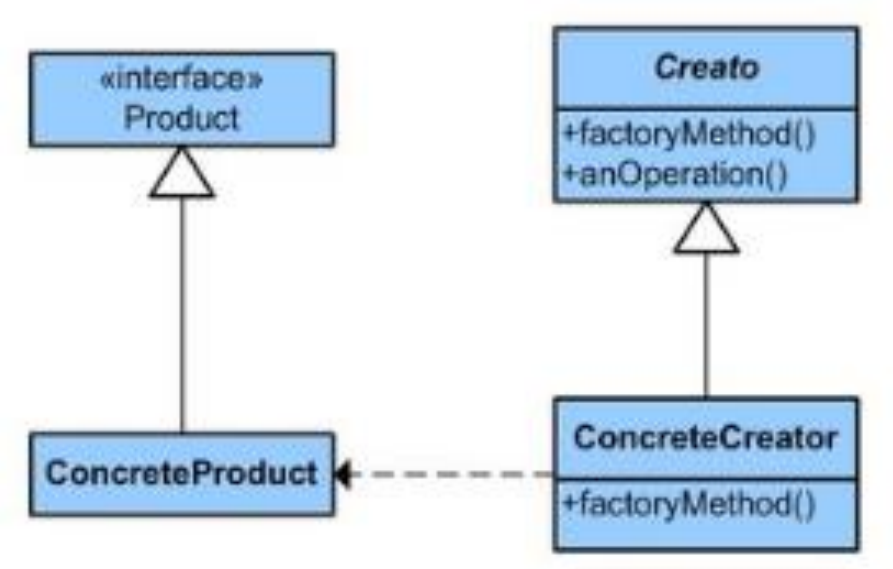
Luontimallit yleisesti käsittävät tiettyjä olioiden ja objektien luomistapoja. Niihin kuuluvat muun muassa riippuvuusinjektio-, singleton- ja tehdasmetodimallit. Riippuvuusinjektio on ohjelmiston osa, joka pitää huolta muun muassa siitä, että konstruktorit, jotka tarvitsevat tietyntyyppisenparametrin, saavat tietyn parametrin ilman sen referenssin erikseen luokalle antamista. Singleton-mallissa taas luodaan vain yksi instanssin luokasta ja luodaan siihen yleinen rajapinta, jota kautta muu ohjelma pääsee siihen käsiksi. Kuvassa 1 näkee singleton-mallin rakenteen. Singletonilla on yleinen instanssi, jonka kautta siihen pääsee käsiksi, ja sillä on myös yleinen operaatio, jota voi kutsua. Kaikki insinööriraportin suunnittelumallien visualisoinnit ovat osa Celinion käyttäjän blogia (11).



Kuva 1. Singleton-mallin visualisointi (11).

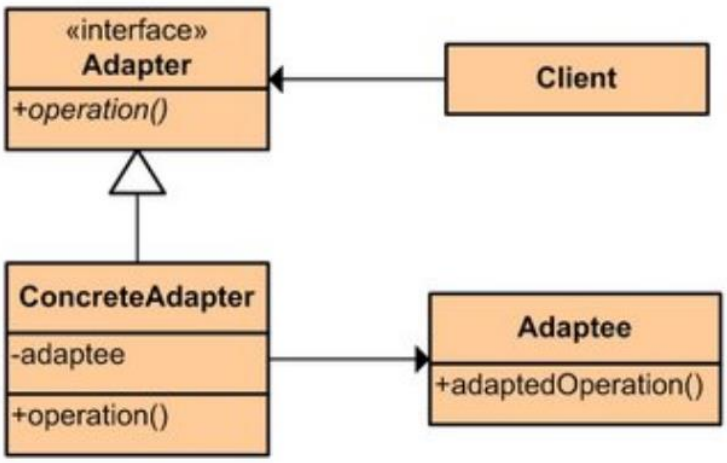
Vaikka kuvan 1 ”singletonData” on vain singletonille itselle nähtävissä olevaa tietoa, voi yleisten operaatioiden kautta saada tämän tiedon haettua ja jopa muokattua. Tämä on vain singleton-mallin perusta.

Kuvan 2 tehdasmetodimallin visualisoinnista voi huomata, kuinka vahvasti eri toiminnot on eritelty. Tehdasmalli yleensä palauttaa factoryMethod-metodista tehtaan luoman olion.



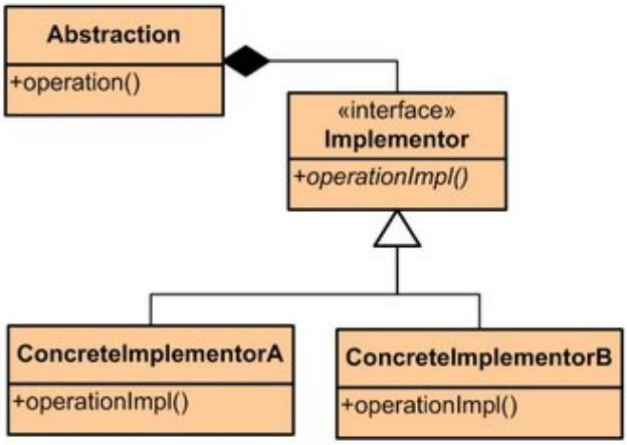
Kuva 2. Tehdasmetodimallin (Factory Method) visualisointi (11).

Rakennemallit käsittävät ohjelmiston rakenteellisen arkkitehtuurin, kuten välikappaleet eri osien välillä. Rakenteellisia malleja ovat muun muassa silta-, proxy- ja adapterimallit. Adapterirakenne toimii ”kääntäjänä” kahden luokan välillä, jotka eivät muuten voisi kommunikoida. Proxy toimii väliaikaisena objektina, jonka kautta toiseen objektiin voidaan päästä käsiksi. Siltamalli on irtikytkemistoteutus (decoupling) kahden luokan välillä, jotteivät ne ole toisistaan riippuvaisia. Kuvassa 3 voi huomata eritellyt tiedostot, jotka parantavat irtikytkentää ja luovat vaihdettavan adapterin kahden ohjelman osan välille.



Kuva 3. Adaptersuunnittelumallin visualisointi (11).

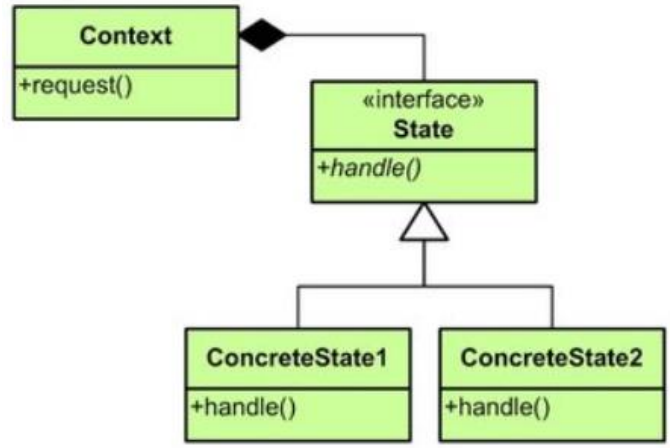
Kuvassa 4 oleva Implementor toimii ”siltana” eri toteutuksille, ettei sen kutsumista tarvitse muuttaa.



Kuva 4. Siltamallin (Bridge) visualisointi (11).

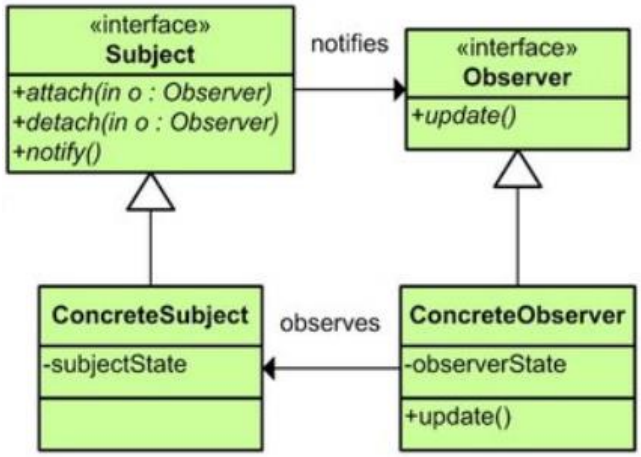
Käyttäytymismallit ovat ohjelman ja sen objektien ja olioiden käyttäytymistä kuvaavia malleja. Esimerkiksi tilamallin mukaan ohjelman objektilla voi olla tila, jonka muuttuessa sen käyttäytyminen muuttuu. Tarkkailijamallissa objektit eivät lähetä omaa tietoaan koko ajan, vaan niitä tarkkailevat objektit saavat tiedon vain, kun tarkkailtava arvo muuttuu.

Kuvan 5 tilamallissa eri tiloilla on sama rajapinta, jotta niitä voi käyttää samalla tavoin, mutta niissä voi olla eri toteutus.



Kuva 5. Tilamallin (State) visualisointi (11).

Kuvan 6 tarkkailijamallissa voi nähdä tarkkailijasuunnittelumallin luoman huomautusten ja tarkkailujen suhteen.



Kuva 6. Tarkkailijamallin (Observer) visualisointi (11).

Suunnittelumallit ovat siis teoreettisia malleja, joiden perusteella voidaan tehdä monia erilaisia ohjelmistototeutuksia, mutta tietyt kehitysmallit ovat saaneet seurauksen, että niistä on tullut suunnittelumalleja.

2.4 Arkkitehtuurimallit

Arkkitehtuurimallit, tai arkkitehtuurityylit, ovat laajempi konsepti kuin yksittäiset suunnittelumallit. Arkkitehtuurimallit kattavat kokonaisuuksia, jotka on usein luotu useista suunnittelumalleista. Suurin ero, mikä erottaa arkkitehtuurimallit ja suunnittelumallit, on,että suunnittelumallit auttavat koodin implementoimisessa ja arkkitehtuurimallit korkean tason infrastruktuurin luonnissa. Arkkitehtuurimallien nimityksetkin ovat usein jopa ohjelmistoalojen nimiä, koska tietylle alalle on jo luotu oma arkkitehtuuripohja. Jotkut ohjelmistoarkkitehtuurimallit voi myös jakaa isompien alojen mukaan, kuten data-arkkitehtuuri, analytiikka, tekoäly ja datan mallinnus (9).

Monet arkkitehtuurimallit lähentelevät ohjelmointikehyksiä, koska tietyllä alalla voi olla standardikirjastoja ja muita yleisiä työkaluja, jotka jo kuuluvat osaan alan arkkitehtuuria. Ohjelmointikehykset on kuitenkin usein tehty arkkitehtuurimallien perusteella, kuten tämän työn vertailuun valitut PureMVC ja Unity-ECS.

Pelialalla pelimoottorit tuovat suurimman osan toiminnallisuudesta, ja ohjelmistokehyksien suurin työ olisi saada järjestystä tieto- ja tiedostorakenteisiin ja tiedon kulkuun. Tämän takia insinöörityöhön on valittu kaksi ohjelmistokehystä, joiden taustalla on suurimmaksi osaksi rakenteellinen arkkitehtuurimalli, ja kolmanneksi kehys, joka seuraa vain yksittäistä suunnittelumallia, reaktiivisen ohjelmoinnin ajatusmallia.

Model-View-Controller- eli malli-näkymä-ohjainarkkitehtuuri luo heti nimensä perusteella selkeän rakenteellisen jakauman. Mallin puolella säilytetään kaikki data. Näkymä on käyttäjärajapinta, joka visualisoi käyttäjälle tarpeellisia tietoja ja toiminnallisuuksia. Ohjain toimii toimintalogiikan ytimenä, joka käyttää ja muokkaa Mallin dataa näyttääkseen asioita näkymässä, josta ohjain myös ottaa syötteitä vastaan. Malli- ja ohjaintasot myös usein luodaan itsenäisiksi näkymästä, jotta ohjelmaa voidaan suorittaa ”päättömässä” tilassa, joka mahdollistaa nopean toiminnallisuuden testaamisen ja helpon näkymän vaihtamisen, esimerkiksi vaihdettaessa sovellusalustasta toiseen. MVC-arkkitehtuuri on erittäin suosittu web-sovelluksissa. (13; 18.)

Entity-Component-System- eli entiteetti-komponentti-systeemiarkkitehtuurin mukaan entiteetit ovat olio-ohjelmoinnista tuttuja olioita, joilla on komponentteja. Systeemit käyvät läpi komponentteja ja suorittavat logiikkaa niiden arvoilla tai niiden arvoille. Esimerkiksi pelaajaentiteetillä voisi olla elämäpiste- ja paikkakomponentit, ja pelin elämäpistesysteemi pitää huolen, että pelaajalla on oikeat elämäpisteet. Liikkumiskomponentti pitää kirjaa pelaajan paikasta ja päivittää sitä pelaajasyötesysteemin mukaisesti. ECS on käytetty esimerkiksi Overwatch-pelin verkkokoodissa, ja työn lähteenä käytetyssä Overwatch-ohjelmoijan GDC-puheessa kerrotaan hyvin, miten ECS:ä käytännössä hyödynnetään isossa projektissa. (16; 20.)

Reaktiivinen ohjelmointi ei tuo paljoa rakenteellista apua projektiin, vaan se toimii käyttäytymissuunnittelumallin mukaan. Arkkitehtuurimallit antavat ohjelmalle rakennetta, mutta UniRx-kehys seuraa vain reaktiivista ohjelmointimallia, joka on tehty Unity-pelimoottoria varten. Kehys on .Net Reactive Extensions-jatke, joka perustuu tarkkailija- ja iterointisuunnittelumalleihin ja asynkronisiin datavirtoihin (14).

3 Ohjelmistokehyksien vertailun alustus

3.1 Kehysten vertailun lähtökohdat ja olettamukset

Insinöörityössä vertailtiin kahta rakenteen tuovaa ohjelmistokehystä, PureMVC ja Unity-ECS, ja kolmantena UniRx, joka toimii enemmän ajatusmallina ja kirjastona reaktiivisen ohjelmistokoodin toteuttamiseen Unityllä. Työssä ohjelmoitiin C#-kielellä ja Unity-pelimoottorilla ja pyrittiin toteuttamaan seuraavat ominaisuudet pienessä matopeliprojektissa:

- päävalikko
- pelinäkymä
- dynaaminen sisältö piste-ennätysluettelon muodossa
- pisteiden lasku ja sen näyttäminen ja tallentaminen
- pelikentän rajat, joihin koskiessa häviää pelin
- häviö-pop-up-ruutu

- tiedon liikkuminen näkymän ja dataobjektin välillä.

3.2 Tavoitteet kehyksien vertailussa

Tavoitteena oli saada yksi kehys, jota suositella ensimmäisenä pelikehitykseen Unityllä, varsinkin aloittelevalle pelinkehittäjälle, ja tiedostaa eri tilanteet, joissa käyttää kaikkia kolmea valittua kehystä. Työssä pyrittiin tutkimaan nykyistä ohjelmistokehysten tilaa ja eri arkkitehtuureihin perustuvien kehysten ominaisuuksia.

3.3 Vertailun hypoteesi ja ennusteet

Kehysten valitsemisen ja pinnallisesti niihin perehtymisen jälkeen ennuste oli seuraava:

PureMVC-kehys on raskain ja laajin, ja sen mukana tulee paljon tiedostohierarkiaa, mutta ajattelumalli vaikuttaa perus-olio-ohjelmointimallilta. Mallin, näkymän ja ohjaimen erottelu ja tiedonkulku niiden välillä tulee olemaan isoin haaste opetella yhtenäiseksi. PureMVC on monen ohjelmointikielen kanssa yhteen sopiva, myös C#-kielen kanssa, joten siinä ei tule olemaan mitään Unity-spesifiä ja se voi olla vaikea integroida Unityn kanssa.

UniRx vaikuttaa kevyimmältä, ja se on erikoistunut erityisesti Unityyn. UniRx:n reaktiivinen ajattelumalli voi olla vaikea opetella, mutta tiedostohierarkiaa ja tiedonkulkua ei ole mitenkään määritelty, eli UniRx:n voisi jopa yhdistää PureMVC:n kanssa. Ensimmäisenä UniRx-esittelystä jäi käteen Monobehaviour coroutinejen käsittelyn helpottuminen ja erikoinen ajattelumalli.

Unity-ECS vie luokkien irti kytkemisen seuraavalle tasolle. Samanlaisia luokkia ja tiedostoja tulee paljon, eli alikategorisointi tulee olemaan tärkeää Entity-, Component- ja System-kansioiden alle. Projekti ei ole iso ohjelmointikokonaisuus, mutta monisäikeinen suorittaminen voi olla tehokasta. Tämä kehys on innostavin ajatusmalliltaan ja tehokkuudeltaan.

4 Kehysten vertailun tulokset

4.1 UniRx-kehys

UniRx on vähiten ohjelmistokehys kolmesta valitusta kehyksestä, mutta kehys-termi o laaja käsite ja mukaan mielletään ohjelmointiajattelumallit. UniRx noudattaa reaktiivisen ohjelmoinnin paradigmaa eli ajattelumallia, eli se ei tuo tiedostorakenteellisia ohjeita projektiin. Reaktiivinen ohjelmointi tarkoittaa koodaamista tavalla, jossa yritetään päästä eroon logiikkapuista eli if-lausekkeista ja kirjoittaa deklaratiivista- eli toteavaa koodia, jossa koodista tulee helpommin luettavaa ja syysuhde näytetään samassa lausekkeessa (14). Esimerkkikoodissa 1 olevassa esimerkissä näkyy datavirran luominen ja tapahtumaan funktion sitominen.

```
Observable.EveryUpdate().Where(_ => Input.GetKeyDown(KeyCode.UpArrow)).Subscribe(_ =>
event1.Invoke());
```

Esimerkkikoodi 1. Kerran kutsuttava datavirran luova käyttäjäsyötettä tarkkaileva tarkkailijatoteutus.

Tässä koodirivissä ei tarvitse laittaa if-lauseketta Update-funktion sisään, vaan voi kertoa kerran alussa, että haluaa joka päivityssyklillä katsoa, onko nuolinäppäin ylös alhaalla, ja jos on, niin ”event1” tapahtuu.

Kehyksen oppiminen

UniRx kehyksen oppiminen oli helppoa. Unity-esimerkkejä ja dokumentaatiota riitti, ja niitä oli helppo seurata. Koko koodirakennetta ei tarvitse tehdä yhtenäiseksi toimivuuden kannalta, ja monissa paikoissa perus- Unity Monobehaviour -koodin voi muuttaa reaktiiviseksi jälkeenpäin. Ajatusmallin omaksumisessa menee oma aikansa, mutta jälkikäteen koodin lukeminen on helppoa, kun voi seurata koodilausekkeita kuin englannin kielen lauseita. Projektista ja myös koodikannasta tuli yksinkertaista, joten projektissa ei päästy näkemään suorituskyvyllisiä etuja eikä käyttämään coroutineja, joten niidenkin käsittely jäi projektin ulkopuoliseksi testailuksi, mutta testaamattakin jo voi kuvitella, että coroutinejen virhekäsittelyn paraneminen auttaa monissa toiminnallisuuksissa.

Kehyksellä tekeminen

Tapahtumat ja Update-silmukat jäävät melkein kokonaan pois koodista, koska UniRx-tietovirroilla pystyy tekemään kaiken toiminnallisuuden yhdessä lausekkeessa. Esimerkiksi esimerkkikoodissa 2 lisätään häviäminen, kun pelaaja lähtee pelialueelta. Luodaan datavirta, jossa katsellaan pelaajan paikkaa, jossa (Where) jokin arvoista ylittää tai alittaa toisen arvon. Where korvaa tässä tilanteessa If-lausekkeen. Sitten tilataan, eli halutaan suoritettavaksi, lambda-funktioon määritellyt toiminnallisuudet. Datavirrat voi myös lisätä CompositeDisposable-tyyppiseen objektiin, jotta virran muistinkäyttö optimoituu, kunhan muistaa kutsua disposables.Dispose() käytön jälkeen.

```
disposables = new CompositeDisposable();
var stream = playerObject.transform.UpdateAsObservable().Where(_ =>
        playerObject.transform.position.x < -MovementScript.mapWidth
      || playerObject.transform.position.x > MovementScript.mapWidth
      || playerObject.transform.position.z > MovementScript.mapHeight
      || playerObject.transform.position.z < -MovementScript.mapHeight).Subscribe(x => {
MenuController.LoseEvent.Invoke();
         disposables.Dispose();
         }).AddTo(disposables);
```

Esimerkkikoodi 2. Pelirajojen ja pelaajan paikan seuraaminen suoritettu yhdellä UniRx-lausekkeella.

Tehdyssä projektissa on myös lokalisaatiojärjestelmä, jolla pystyy vaihtelemaan suomen- ja englanninkielisen käyttöliittymän välillä. Esimerkkikoodirivillä 3 näkyy, kuinka etsitään lokalisaatioteksti listasta tekstin numeromuotoon muunnettu enum-muuttuja indeksinä ja laitetaan se käyttöliittymän tekstikenttään. UniRx SubscribeToText on luotu juuri tällaista varten, ja se automaattisesti laittaa löydetyn tekstin annetun komponentin tekstikenttään.

```
LocalizationTexts[(int)UIText.StartGame].SubscribeToText(Start.GetComponentInChil-
dren<Text>());
```

Esimerkkikoodi 3. Lokalisaatiotekstilistasta haetun tekstin liittäminen tekstikomponenttiin SubscribeToText-metodilla.

Esimerkkikoodissa 4 on samanlainen koodi, jossa tehdään normaali ”tilaus” ja asetetaan saatu teksti manuaalisesti komponentin tekstikenttään.

```
LocalizationTexts[(int)UIText.Restart].Subscribe(x=>loseScreen.RestartButton.GetCompo-
nentInChildren<Text>().text = x);
```

Esimerkkikoodi 4. Lokalisaatiotekstilistasta haetun tekstin liittäminen tekstikomponenttiin normaali-Subscribe-metodilla.

Yleisesti voi huomata tarkkailtavien arvojen ja käyttöliittymän päivittämisen olevan helppoa niin opetella kuin käyttää. Nappuloihinkaan ei tarvitse lisätä normaaleita kuuntelijoita ”AddListener”-komennolla, vaan UniRx mahdollistaa kaikkien tapahtumien luomisen datavirroiksi esimerkkikoodin 5 tapaan.

```
StartButton.onClick.AsObservable().Subscribe(_ =>
      {
         UIController.ChangeUIState.Invoke();
         GameController.StartGame();
      });
```

Esimerkkikoodi 5. Luodaan nappulan tapahtumaa tarkasteleva datavirta, joka suorittaa lambdaan määritellyn funktion nappia painettaessa.

Monissa kohdissa koodia on käytetty normaaleja tapahtumia (Event), ja tämä on myös yksi käytettävyyden ominaisuuksista. Koko koodikannan ei tarvitse olla UniRx-tyyppisesti ohjelmoitu.

Esimerkkikoodissa 6 näkyy, kuinka käyttäjän syötteen voi lukea. Annetaan sisään jokainen tapahtuma, joka halutaan suorittaa, ja kysytään tarkkailtavalta päivitysvirralta, milloin nappi on alhaalla.

```
public void InputScriptInit(UnityEvent event1, UnityEvent event2, UnityEvent event3, UnityEvent
event4)
   {
      Observable.EveryUpdate().Where(_ => Input.GetKeyDown(KeyCode.UpArrow)).Sub-
scribe(_ => event1.Invoke());
      Observable.EveryUpdate().Where(_ => Input.GetKeyDown(KeyCode.DownArrow)).Sub-
scribe(_ => event2.Invoke());
      Observable.EveryUpdate().Where(_ => Input.GetKeyDown(KeyCode.RightAr-row)).Sub-
scribe(_ => event3.Invoke());
      Observable.EveryUpdate().Where(_ => Input.GetKeyDown(KeyCode.LeftAr-row)).Sub-
scribe(_ => event4.Invoke());
   }
```

Esimerkkikoodi 6. Tapahtumien liittäminen datavirroilla pelaajan syötteisiin.

Kehyksen hyvät ja huonot puolet

Hyvät puolet

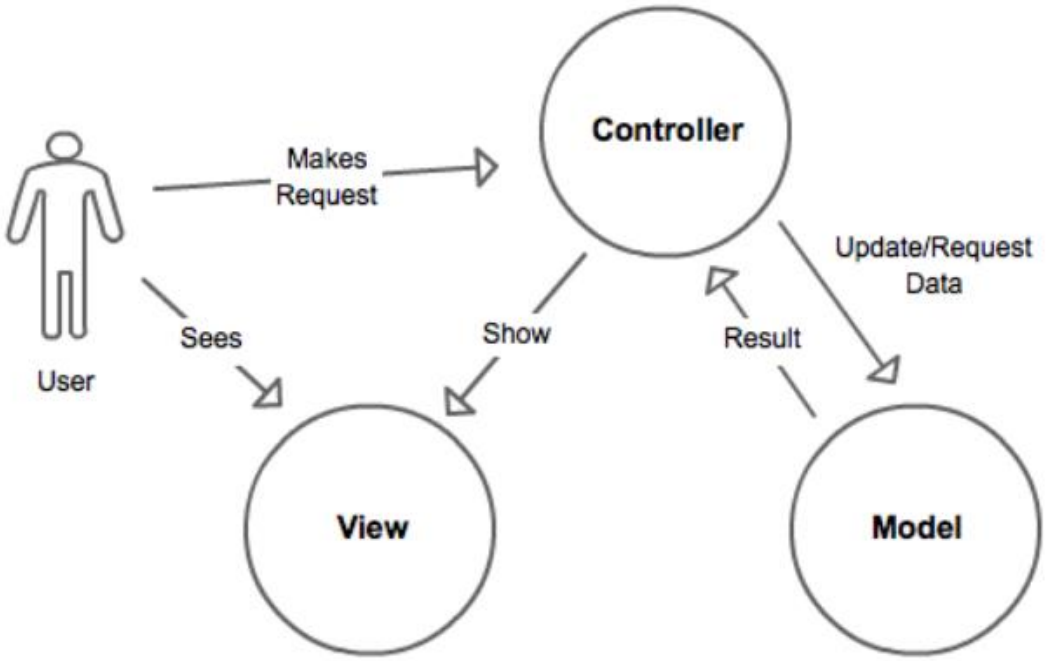
UniRx:n hyvä puoli on sen helppo opittavuus ja se, että sen käyttö tuo selkeyttä koodiin ja parantaa luettavuutta. If-lausekkeet muuttuivat Linq-tyyppisiksi kyselyiksi ja logiikka oli helpommin suunniteltavissa, kirjoitettavissa ja luettavissa. Koodirivien määrä vähentyi if-lausekkeiden kadotessa, ja paljon rivejä tuottavat logiikkapuut muuttuivat yksittäisiksi, mutta joskus myös pitkiksi, lausekkeiksi. Kehys ei vaadi, että kaikki tai edes yksittäinen kokonainen ominaisuus tehdään reaktiivisesti, se vain mahdollistaa reaktiivisen koodin luomisen, ja aloittaessa pystyy helposti muuttamaan perus- Unity Monobehaviour -koodin reaktiiviseksi jälkeenpäin.

Huonot puolet

Kehys on kevyt eikä anna tiedostoihin rakenteellista opastusta tai tiedon liikkumisen kannalta minkäänlaista opastusta tai mallia, jota seurata. UniRx on yleisesti kirjastomainen, eli pelkkä työkalupakki, ja uskon, että se voisi olla enemmän kirjasto kuin kokonainen kehys. Suosittelisin myös käyttämään UniRx:n kanssa jotain toista kehystä.

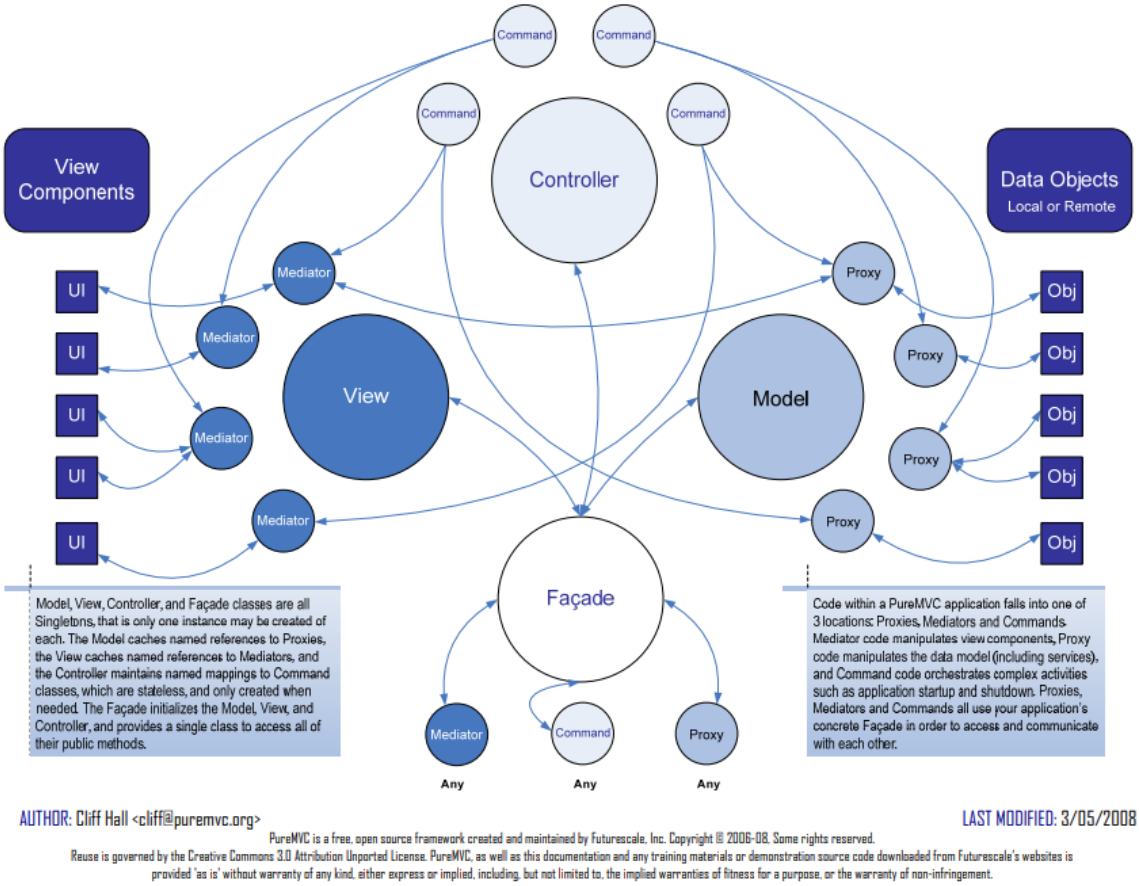
4.2 PureMVC-kehys

PureMVC on model-view-controller-arkkitehtuurimallin pohjalta luotu kehys monille eri ohjelmointikielille, ja se mainostaa itseään yksinkertaisena ja luotettavana kehyksenä, jota ei iteroida tai kehitetä eteenpäin. Kuvassa 7 näkee MVC:n perustoimintamallin, jossa käyttäjä näkee View-osan ohjelmasta, Controller hoitaa syötteiden läpikäynnin ja Model-tason tiedon kanssa toimimisen (13).



Kuva 7. MVC-mallin toiminta visualisoituna (12).

Vuoden 2008 jälkeen jäissä ollut toiminnallisuus on edelleen toimiva. Kuvassa 8 esitetään tiedonkulkua PureMVC-ohjelmistossa, eli Model tasolla on Proxyt, jotka hoitavat kommunikoinnin dataobjektien ja muun ohjelmiston välillä. Controller-luokalla on komentoja, joita voi kutsua Controllerin kautta, ja se muokkaa tietoa ja suorittaa kaiken ohjelmiston logiikan. View-komponenteilla on Mediator-välikappaleet, jotka päivittävät käyttöliittymää ja lähettävät tapahtumia syötteistä. Kuva 8 on PureMVC:n toiminnan perustasta ja tiedon ja logiikan kulun visuaalisesta suhteesta.



Kuva 8. PureMVC:n konseptidiagrammi 13).

Kehyksen oppiminen

PureMVC:n käyttöön ei löytynyt paljoa C#-dokumentointia ja Unity-esimerkit olivat suppeita. Tämän kehyksen kohdalla tuntui aluksi, että pitäisi vaihtaa toiseen, mutta se silti yritettiin saada toimimaan ja testattua Unity-ympäristön ulkopuolelta tulevaa kehystä. Kehystä testatessa oli opiskeltu ja testailtu kehystä eikä saatu mitään tehtyä, asetettu aikamäärä tuli täyteen ja jouduttiin lopettamaan kehyksen osalta. Dokumentaation vähyys, ajattelumallin vaikeus ja sen selityksen puute ja kehyksen vanhuus olivat rajoittavia tekijöitä. Kehys haluttiin vaihtaa toiseen kehykseen model-view-arkkitehtuuriperheestä, mutta huomattiin, että epäonnistuminen on hyvä havainto työn aiheen perusteella.

Kehyksellä tekeminen

Se, mitä PureMVC-kehyksestä ymmärsin, vaikutti mielenkiintoiselta. Yksinkertainentoimintakehys vaikuttaa hyvältä ympäristöltä koodata. Toinen model-view-arkkitehtuurimalliperheen kehys, jossa on tarpeeksi Unity C#-dokumentaatiota ja esimerkkejä, toimii hyvin aloittelevalle koodarille tuomaan järjestystä kaoottiseen koodikantaan. Niitä löytyy esimerkiksi Unity Asset Storesta.

Kehyksen hyvät ja huonot puolet

Hyvät puolet

PureMVC:n hyvät puolet olivat sen yksinkertainen ja selvä idea.

Huonot puolet

PureMVC:n huonot puolet olivat vähäinen Unity C#-dokumentaatio ja se, että se oli vaikea integroida Unity Monobehaviour -ympäristöön.

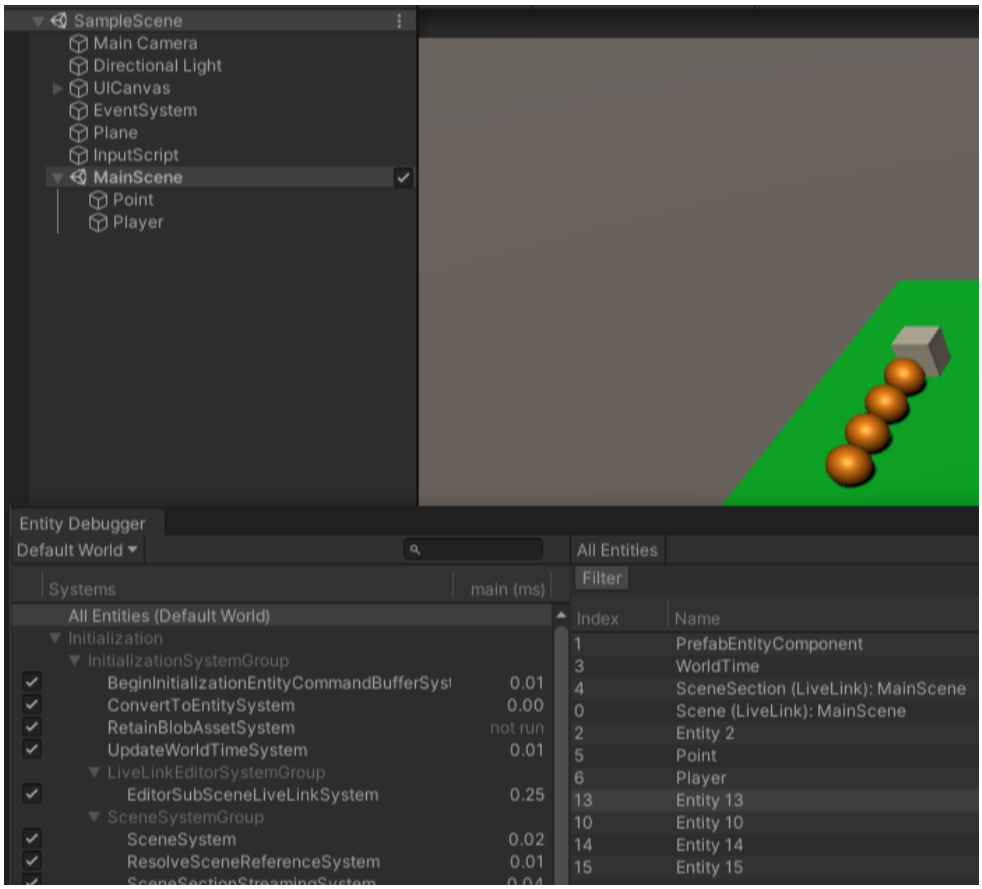
4.3 Unity-ECS-kehys

Unity-ECS on Unityn oma kehys, joka on vasta testivaiheessa. Kehyksen iskulause on ”tehokkuus itseisarvona”, ja Unityn näyttämät presentaatiot vakuuttavat. Kehys seuraa ECS- eli entity-component-system-arkkitehtuurimallia, eli Unityn GameObjectit vaihtuvat entiteeteiksi, joilla on komponentteja. Systeemit sisältävät kaiken toimintalogiikan ja löytävät globaalista entiteettikannasta tyypitetyt komponenttireferenssit ja voivat tehdä funktioita komponenttien arvoille ja arvoilla (16).

ECS-kehykset ovat usein vaikeita oppia, koska koko ohjelma pitää tehdä niiden kautta, mutta Unityllä on Hybrid-ECS-toiminnallisuus, joka on osa opettelun, käyttämisen ja itse kehyksen kehityksen helpottamista. Useat natiivikomponentit, kuten transform ja renderer, toimivat ECS-entiteettien ja -systeemien kanssa. ECS:n mukana tulee ConvertToEntity-skripti, jonka voi lisätä GameObjectiin, ja se suoritetaan automaattisesti. ECS Con-

vertToEntity pyrkii muuttamaan komponentit ECS-komponenteiksi, kadottamaan GameObjectin hierarkiasta ja käsittelemään alkuperäistä kokonaisuutta entiteettinä. ECS antaa hyvät virheviestit, jos komponentin kääntäminen ei onnistu, ja dokumentaation perusteella muutokset on helppo tehdä pikkuhiljaa. Uusien komponenttien käsitteleminen on erilaista: esimerkiksi vanha transform-komponentti vaihtuu translation-komponentiksi, jolla on vain yksi float3-arvo, ja rotation ei ole osa transform-komponenttia, vaan on oma quaternion-komponenttinsa.

Kuvassa 9 on näkymä ECS Debugger-ikkunasta, Unity Gameobject -hierarkiasta ja pelinäkymästä. Projektin matopelissä pisteitä kerätessä pelaajan häntä pitenee ja kuvassa 9 näkyy, kuinka piste ja pelaaja on sijoitettu ”SubScenen” alle GameObjecteina. Subscenen alla olevat GameObjectit automaattisesti konvertoituvat entiteeteiksi, ilman erikseen ConvertToEntity-skriptin lisäämistä. Entiteetit näkyvät Entity Debugger -ikkunassa alareunassa. Koodissa luodut entiteetit 10, 13, 14, 15, eivät näy GameObject-hierarkiassa, mutta se ei tarkoita, ettei niitä näkyisi pelissä.



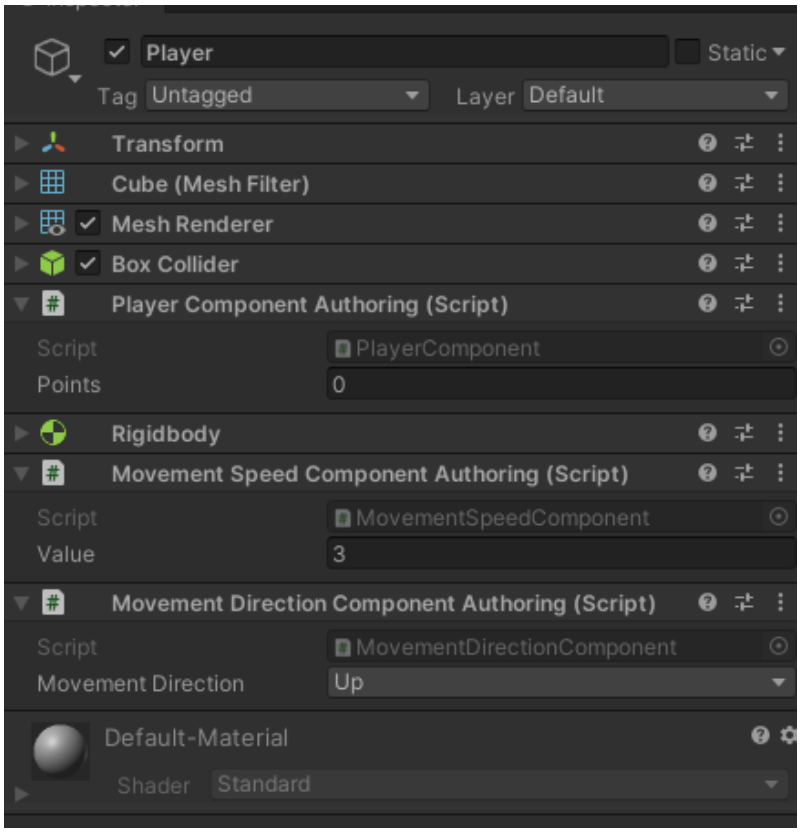
Kuva 9. Unityn GameObject-hierarkia, ECS Entity Debugger ja pelinäkymä.

Kuvassa 10 on esimerkki prefabistä instantioidusta entiteetistä. Entiteetti ja sen komponentit on GameObject prefabin perusteella konvertoitu entiteetiksi. Jokainen aivan vasemmasta laidasta alkava nimi on eri komponentti.

Kuva 10. Kuvan 9 Entity 13, häntäentiteetti.

Kuvassa 11 on esimerkki Hybrid-ECS-entiteetistä. Pelaajaentiteetti-Gameobject on hierarkiassa Subscenen alla, ja sillä on normaaleja natiivikomponentteja, mutta sen

löytää Entitymanagerin kautta entiteettinä. Pelaajaentiteetin komponentit voi myös löytää muiden entiteettien komponenttien joukosta.



Kuva 11. Pelaaja-GameObject konvertoituna hybridientiteetiksi.

Koko projektia ei tarvitse tehdä täysin ECS-mallilla, vaan voi käyttää hybridtoiminnallisuutta, mutta projektin kokonaan tekeminen täysin ECS:n mukaan tuo sen vahvuuden esille. Burst Compiler (10) on syy, miksi ECS on niin tehokas. Burst on Unityn kehittämän matematiikkakeskeinen koodinkääntäjä, joka luo optimoitua konekieltä kohdealustan perusteella. ECS-ohjelmoijan ei tarvitse osata koodata tehokasta kieltä, kuten C++, tai osata konekieltä, vaan Burst tekee sen automaattisesti. Koodin optimointi tulee automaattisesti ECS:n käyttämisen sivutuotteena.

Kehyksen oppiminen

ECS tuo täysin uuden ajattelutavan Unity-ohjelmointiin, ja on vaikeaa päästä mukaan kehyksen ajattelumalliin, mutta oppimisen helpotukseksi ja Unity-lähtöisen kehityksen

takia Unity Hybrid-ECS on olemassa. Vanha GameObject-tyyppinen ohjelmoiminen on mahdollista ECS-projektissa, ja samoin kuin UniRx:ssä, pystyy käyttämään normaalia Unity Monobehaviour -olio-ohjelmointia ECS-ohjelmoinnin kanssa.

ECS ei auta käyttöliittymän ohjelmoinnissa, joten projektista jäi suurin osa käyttöliittymästä pois ja testattiin vain, miten käyttöliittymän tapahtumia pystyy liittämään ECS-koodiin. Se onnistuu helposti. Asynkronisen koodin suoritukseen tapahtumia pitää käsitellä vähän eri tavalla, mutta Unity Jobs -systeemin kanssa toimimiseen löytyy paljon esimerkkejä, dokumentaatiota ja apuvideoita.

Kun ajatusmallin on omaksunut, pelilogiikan laatiminen on helppoa, kun linkkejä ei tarvitse luoda ja systeemit löytävät entiteetit automaattisesti. Vaikeat konseptit ovat, miten löytää itse entiteetti, ei pelkkä komponenttia, ja Transform-komponentin muuttuminen Translation-komponentiksi, kun vanhan Vector3-tyypin vaihtuminen float3-tyyppiin kestää hetken ymmärtää.

Kehyksellä tekeminen

UniRx-projektissa keskityttiin paljon käyttöliittymään. Sen jälkeen lähdettiin tekemään ECS-projektia käyttöliittymäjohtoisesti mutta hetki sitä tehtyä, ymmärrettiin, ettei ECS ole käyttöliittymän ohjelmointiin tarkoitettu, ainakaan vielä. Pelilogiikan tekeminen oli mukavaa ja selkeää. Ensin siinä tehdään komponentti-struct esimerkkikoodirivin 7 mukaan.

```
[GenerateAuthoringComponent]
public struct MovementSpeedComponent : IComponentData
{
    public float Value;
}
```

Esimerkkikoodi 7. Esimerkki komponenttikooditiedostosta.

Class vaihtuu Structiin ja Monobehaviour vaihtuu IComponentDataan. Lisätessä GenerateAuthoringComponent-attribuutin structille, pystyy skriptin lisäämään GameObjectiin kiinni, ja se on osa kehyksen Hybrid-ECS-toiminnallisuutta. Muuten entiteetteihin komponenttien lisääminen tapahtuu entiteettiä luodessa esimerkkikoodirivin 8 tavalla.

```
var entity = eManager.Instantiate(PrefabEntityComponent.prefabEntity);
      eManager.AddComponentData(entity, new TailComponent { entity = entity, ID = e,
 Target = e-1, TargetTransform = new float3(0, 0, 0)});
      eManager.AddComponentData(entity, new Translation { Value = new float3(0, 0, 0) });
```

Esimerkkikoodi 8. Entiteetti luodaan prefabin perusteella ja siihen lisätään komponentteja ja komponentteihin aloitusarvoja.

Edellisessä esimerkissä käytetään Entitymanageria ja käytetään Monobehaviourista tuttua Instantiate-kutsua luomaan olio pelimaailmaan. Prefab-objektissa voivat olla jo transform- ja TailComponent-komponentti kiinni, ja tässä tapauksessa niiden arvot vain muutetaan kyseisiin arvoihin. eManager on aikaisemmin haettu ECS-maailman EntityManager (esimerkkikoodi 9), jonka löytää mistä vain ja joka on keskeinen pala ECS-arkkitehtuuria. Entitymanagerin kautta voi löytää tai luoda uuden systeemin. Entitymanagerilla pääsee käsiksi komponentteihin, entiteettiarkkityyppien luontiin ja entiteettikyselyiden muodostamiseen.

```
World.DefaultGameObjectInjectionWorld.EntityManager;
```

Esimerkkikoodi 9. Entiteettimaailmasta haettu entiteettimanageri.

Arkkityypit ovat Unity ECS -kehyksessä komponenttikokonaisuuksia. Esimerkiksi vihollisella voi olla elämäpisteet-, paikka- ja reppukomponentit, ja tällöin näistä voi muodostaa arkkityypin, ettei niitä tarvitse aina listata uutta vihollista luodessa uudestaan. Esimerkissä 10 näkyy, miten löydetään entiteettimaailman kautta systeemi ja sen julkinen muuttuja.

```
entityWorld.GetOrCreateSystem<TailFollowSystem>().OnLoseTail += LoseGame;
```

Esimerkkikoodi 10. TailFollowSystem-systeemillä on event EventHandler OnLoseTail-muuttuja, johon voi +=-operaattorilla lisätä metodin, joka suoritetaan tapahtuman tapahtuessa. LoseGame on rivin omaavassa tiedostossa oleva metodi, joka suorittaa pelin häviämisen logiikan.

Update-silmukoista ei silti päästä eroon, koska jokaisella systeemillä on oma OnUpdate-silmukassa, johon logiikan voi tehdä. Systeemit pääsevät käsiksi kaikkiin tietyn tyypin komponentteihin koodiesimerkin 11 tavoin.

```
Entities.ForEach((ref PlayerComponent player, ref Translation playerTransform) => {…}
```

Esimerkkikoodi 11. Systeemin sisällä päästään käsiksi entiteettikantaan, ja sieltä voi hakea komponentteja tai komponenttiyhdistelmiä. Tämä koodirivi etsii kaikki entiteetit, joilla on PlayerComponent, ja Translation-komponentti.

Nämä foreach-silmukat ovat kuitenkin tehokkaita, koska ne etsivät ja käsittelevät muistiin peräkkäin laitettuja komponentteja, ja tämä on osa ECS-arkkitehtuurin vahvuudesta. Edellisessä koodissa systeemi löytää kaikki entiteetit, joilla on PlayerComponent- ja Translation-komponentti, ja aaltosulun jälkeen voi lambdalla käsitellä niitä. Translation-komponentti on Monobehaviourin Transform-komponentti muutettuna entiteettimaailmaan ja sisältää float3:n eikä vector3:a.

Ajatusmallin sisäistyksen jälkeen pelilogiikka oli helppo implementoida, eikä tarvitse miettiä, mihin tiedostoon laittaa mitäkin, kun yksittäinen systeemi on helppo tehdä ja tietojen linkittäminen luokasta toiseen, tai tässä tapauksessa systeemistä toiseen, on helppoa globaalin entiteettimaailman kautta. Käyttöliittymän integrointi tuntui ensin hölmöltä, kun ECS:ä ei voinut siihen käyttää, vaan käyttöliittymä on kokonaan erillinen, mutta helposti linkitettävissä tapahtumien kautta.

Kehyksen hyvät ja huonot puolet

Hyvät puolet

ECS-dokumentaatiota ja aiheesta keskustelua riittää paljon, ja yleisesti halutut resurssit löytyivät nopeasti. ECS on tehokas ja suositeltava työkalu, jos pelissä on paljon rinnakkaista suorittamista tai paljon yksittäisiä olioita tai entiteettejä. Tiedostohierarkiaa ei tarvitse miettiä, ja jokaisen koodinpätkän voi laittaa eri tiedostoon ilman suurta määrää koodin toistamista. Linkittäminen ja referenssien etsiminen on helppoa globaalin entiteettimaalilman kautta. Unityn DOTS-kirjasto, eli Data-Oriented Technology Stack, tulee olemaan mielessä jatkoprojekteja ajatellen varmasti.

Huonot puolet

Kaiken hyödyn irti saaminen vaatii koko DOTS-kirjaston käyttämistä, ja Hybrid-ECS on enemmän sekava kuin auttava. Heti ensimmäisen testiprojektin jälkeen kehotettaisiin

siirtymään käyttämään kokonaan ECS:iin, paitsi käyttöliittymän kanssa. Koska kirjasto on testivaiheessa, joitain ominaisuuksia puuttuu, esimerkiksi vuoden 2020 alussa ei ollut vielä animaatioita. Ajatusmallin omaksuminen voi viedä hetken, ja parhaat ohjelmointikäytännöt eivät ole vielä muodostuneet tai ne eivät ole ilmiselviä.

5 Testien päätelmät

Luvussa 3.3 esitetty hypoteesi osoittautui oikeaksi. Projektin ulkopuoliset tekijät haittasivat hieman, eikä projektissa välttämättä saatu tehtyä haluttuja lisätunteja kehysten kanssa, joista oltiin innoissaan, mutta päästiin aikamääreisiin, jotka oli asetettu. Koska olin töissä ohjelmoijana, oli vaikea asennoitua iltapäivisin täysin eri ajatusmalliin, mutta samaan Unity-ympäristöön, joten suurin osa työstä tehtiin vapaapäivinä ja viikonloppuisin. Tavoite uuden työkalun löytämiseen toteutui Unity ECS -muodossa, ja toivottavasti joku muukin huomaa ohjelmistokehysten tehokkuuden. Valitut kehykset ovat erilaisia, ja aloittelevana ohjelmoijana on vaikea vaihtaa kehyksestä toiseen, varsinkin työn ohella.

UniRx-kehys on kevyt, sitä voi käyttää toisen kehyksen rinnalla, ja sillä käyttöliittymän ohjelmointi oli mukavaa ja helppoa. Myös käyttöliittymän tietokenttien ja luokkien muuttujien välille sai tehtyä helpon synkronointivirran, joka helpotti muuttuvien tietojen näyttämistä ruudulla. Harjoittelujen MVVM-kehyksen jälkeen olin valmiina tiukkaan tiedostorakennekuriin, jota kehys olisi voinut tuoda, ja kun tein ensimmäisenä UniRx-projektin, se tuntui löysältä ja kevyeltä, mutta sain kehyksestä irti mitä halusin ja odotin.

PureMVC:n kanssa kävi huonosti: Heti alkuun pelättiin, ettei dokumentaatio riittäisi tai Unityyn liittäminen olisi vaikeaa, mutta oltiin optimistisia. Aikarajoitteiden ohi mennessä, päätetiin ottaa kokemuksesta kaikki irti ja analysoitiin, mitä meni vikaan ja miksi.

Unity ECS oli viimeisenä, mutta ennakkosuosikkina. Sen syntaksi oli luettavaa ja helppoa. Ennen käytetyn riippuvuusinjektion hoitamat referenssit ja linkitykset hoituivat automaattisesti entiteettimaailman kautta, ja entiteettimaailma helpotti paljon työskentelyä. Boilerplate oli vähäistä, ja uusia tiedostoja pystyi luomaan ilman, että tarvitsi miettiä, ”onko tämä liian pieni asia olemaan yksin tiedostossa”, vaan kaikki oli helppoa laittaa omiin tiedostoihinsa.

5.1 Vertailtujen kehysten erot

UniRx on vähiten kehys ja toimii enemmän lisätyökaluna kuin ohjelmistokehyksenä. Se tuo asynkronista toiminnallisuutta helposti, ja sitä voisi jopa käyttää käyttöliittymän ohjelmointiin pienemmissä projekteissa tai toisen kehyksen kanssa isommissakin.

PureMVC ja muut model-view-perheen kehykset tuovat järjestystä tiedostokantaan ja voivat antaa tiukat raamit, joita seurata. Model-view tuo isompiinkin projekteihin toiminnallisuuden löydettävyyttä koodikannasta ja tiedostohierarkian selkeyttä.

Unity ECS on Unityn tulevaisuus ja Unity itsekin kehittää ECS-kehystä vanhan olio-ohjelmoinnin päälle. Tiedostojen paljous tekee tiedostohierarkiasta tärkeän ja nimeämiskäytännön pitää olla tarkka ja johdonmukainen, jotta tiedostojen paljous ei haittaa. ECS-kehys on suositeltava kehys kokeilla, mutta kannattaa ottaa huomioon puuttuvat ominaisuudet mielessä.

5.2 Unity-pelimoottorin ja kehysten yhteensopivuus

UniRx oli suunniteltu Unitylle, ja sen huomasi. Monia Unityn komponenttien käsittelyfunktioita löytyi kehyksen kirjastosta. Ainoat ongelmat ovat TextMeshPro-kirjaston kanssa, ja ne ovat marginaalisia ja niistä pääsee ohi muuttamalla kyseistä toiminnallisuutta toteutuksessa.

PureMVC-Unity-dokumentaatiota löytyi yksi kappale, jossa ei selitetty kunnolla, miten kehystä pitäisi käyttää. Monet keskustelufoorumit tarjosivat ison vaivan takana olevia ratkaisuja, jotka toimivat kyseenalaisella vakaudella.

Unity ECS on Unityn tekemä kehys, ja se on täysin integroitu Unity-ympäristöön. Vaikka kehys on vasta kehitysvaiheessa, on siinä vahvoja työkaluja Unity-ohjelmoijalle, niin vanhan GameObject-osaajalle kuin täysin uudelle ohjelmoijalle.

5.3 Kehysten hyödyt aloittelevalle Unity-ohjelmoijalle

Eri projektit tarvitsevat aina eri työkalut, ja insinöörityössä käsitellyt kolme työkalua ovat aivan eri tilanteisiin. Usein aloittelevan Unity-ohjelmoijan kannattaa pysyä perusteissa, käyttää Unity-pelimoottorin tuomaa kehystä ja toimintatapaa ja harjoittaa hyviä perusohjelmointitapoja koodikannassaan. Niin kuin kaiken opiskelussa, seuraava askel olisi luoda projektikokonaisuus, ja tällöin kehykset tuovat rakennetta ja järjestystä projektiin. Muut projektissa mukana olevat voivat lukea ja ymmärtää koodia helposti, olettaen että hekin ovat opetelleet kehystä, ja suunnittelun tarve vähenee, kun on jo ennalta määritelty tapa ohjelmoida.

Unity ECS

ECS tuo tehokkuutta vaativiin laskutoimenpiteisiin ja vilkkaaseen pelimaailmaan. Jos pelisi koko on suuri tai tarvitaan monisäikeistä suorittamista, ECS:n käyttö on suotavaa ja se ratkoo ongelmia, joita massiiviset pelit tuottavat. Ohjelmoinnin suunnittelu ja suorittaminen on helppoa, kun ymmärtää kehyksen tuoman ajatusmallin. Turhakoodi, eli boilerplate, on minimaalista ja hyvin korostettu, joten se ei sekoita aloittelijan päätä.

UniRx

UniRx on hyvä käyttöliittymän, WWW-kutsujen ja monisäikeiseen ohjelmointiin. Se ei vaadi koko koodikannalta täyttä muutosta. Se on myös kevyt ja helppo tapa löytää uusi työkalu ohjelmointiin. Syntaksi auttaa ymmärtämään toiminnallisuutta, ja reaktiivinen ohjelmointi on yksi niistä asioista, joka voi olla ”se” juttu ohjelmoijalle.

PureMVC

PureMVC voi olla hyvä, jos siihen löytää opetusresurssin tai opettajan, mutta sitä ei voi suositella aloittelijalle ilman mentoria. Toinen Model-View-arkkitehtuurimalliperheen alle kuuluva kehys on hyvä tuomaan järjestystä aloittelevan ohjelmoijan projekteihin, ja vahvasti suosittelen tällaisen kehyksen testaamista. Boilerplate voi olla usein korkea, mutta se on sen arvoista. Edeltä määritellyt toimintatavat aina auttavat, niin ettei niitä tarvitse itse suunnitella, kun vain haluaa opetella ohjelmoimaan.

6 Yhteenveto

Ohjelmointiala on nuori ala ja kehittyy nopeasti. Teoreettisen tiedon opettaminen ja ymmärtäminen nostaa aloittelevat ohjelmoijat nopeasti ammattitasolle. Ohjelmoinnin konseptit, kuten suunnittelumallit, arkkitehtuurimallit, ohjelmistokehykset ja monet muut, ovat tulevaisuuden avain tehokkaaseen ohjelmointiin. Itse ohjelmointitavan suunnittelemisen vähentyessä voi ohjelmiston luovalle kehitykselle jäädä enemmän tilaa.

Vaikeinta valmiiden työkalujen käyttöönotossa on projektin sopivuuden tunnistaminen. Tämän takia työkalun taustalla olevien ominaisuuksien tietäminen on tärkeää. Koko työkalun taustatoteutuksen ymmärtäminen ei ole tarpeen, vaan konseptien, joiden kautta se on luotu.

Ohjelmointikehykset ovat vahva työkalu, kun niitä käytätetään oikein ja oikeissa paikoissa. Vahvoissa kehyksissä on usein pitkä oppimisaika, mutta yhtenäinen pitkäaikainen ohjelmistokehitys tarvitsee jonkinlaiset raamit, ja ohjelmistokehykset tuovat sellaisen. Ohjelmistokehyksistä ECS vaikuttaa Unitylle sopivimmalta, ja se vaikuttaa vahvimmalta työkalulta, ottaen huomioon pelien ominaisuudet ohjelmistona. Siinä on komponentteja, joihin monet systeemit ovat kytköksissä ja paljon samanaikaista suorittamista. Itse innostuin ECS-kehyksestä paljon, ja jatkossa aion käyttää Unity-ECS:ä peliprojekteissani.

Lähteet

1 What is Framework in Software Engineering? 2019. Verkkoaineisto. Gbksoft. <https://gbksoft.com/blog/what-is-framework/>. Päivitetty 20.12.2019. Luettu 27.1.2020.

2 Naur, P. & Randell, B. 1969. Software Engineering. Verkkoaineisto. <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>. Luettu 27.1.2020.

3 Kruchten, P; Obbink, H. & Stafford J. 2006. The past, present and future of software architecture. IEEE Software, vol. 23, s. 22–30.

4 IEEE Transactions on Software Engineering. 1995. Introduction to the Special Issue on Software Architecture, vol. 21, s. 269–274.

5 Software Frameworks: Resistance isn't futile. 2011. Verkkoaineisto. Elegantcoding. <http://www.elegantcoding.com/2011/07/software-frameworks-resistance-isnt-futile.html>. Luettu 27.1.2020.

6 Alexander, Christopher. 1966. The Pattern of Streets. Journal of the AIP, vol. 32, s. 273–278.

7 Smith, Reid. 1987. Panel on design methodology. ACM SIGPLAN Notices.

8 Beck, Kent & Cunningham, Ward. 1987. Using Pattern Languages for Object-Oriented Program. OOPSLA 87.

9 Sharma, Anubha; Kumar, Manoj & Agarwal, Sonali. 2015. A Complete Survey on Software Architectural Styles and Patterns. Elsevier.

10 Amat, Charles. 2018. Unity Burst Compiler: Performance Optimization Made Easy. Verkkoaineisto. <http://infalliblecode.com/unity-burst-compiler/>. Luettu 4.4.2020.

11 Useful posters of the GoF patterns. 2009. Verkkoaineisto. Celinio. <http://www.celinio.net/techblog/?p=65>. Luettu 8.4.2020.

12 Hiraash, Thawfeek. MVC Explanation (Concept Diagram). Verkkoaineisto. <https://creately.com/diagram/example/h6p18fex4/MVC+Expaination?utm_source=pinterest&utm_medium=social&utm_campaign=pincmaps>. Luettu 8.4.2020.

13 The PureMVC Framework. Verkkoaineisto. PureMVC. <https://puremvc.org/>. Luettu 13.4.2020.

14 Kawai, Yoshifumi (neuecc). UniRx – Reactive Extensions for Unity. Verkkoaineisto. <https://github.com/neuecc/UniRx>. Luettu 13.4.2020.

15 Framework. 2013. Verkkoaineisto. TechTerms. <https://techterms.com/definition/framework>. Päivitetty 7.3.2013. Luettu 13.4.2020.

16 Entity Component System. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/index.html>. Luettu 13.4.2020.

17 Architectural frameworks, models, and views. Verkkoaineisto. Mitre. <https://www.mitre.org/publications/systems-engineering-guide/se-lifecycle-building-blocks/system-architecture/architectural-frameworks-models-and-views>. Luettu 13.4.2020.

18 Goel, Aman. 2020. 10 Best Web Development Frameworks. Verkkoaineisto. Hackr.io. <https://hackr.io/blog/top-10-web-development-frameworks-in-2020>. Päivitetty 9.4.2020. Luettu 13.4.2020.

19 Why companies use games as a service? 2017. Verkkoaineisto. Staff Zco. <https://www.zco.com/blog/why-companies-use-games-as-a-service/>. Luettu 13.4.2020.

20 Ford, Timothy. 2017. Overwatch Gameplay Architecture and Netcode. Verkkoaineisto. <https://www.youtube.com/watch?v=W3aieHjyNvw>. Katsottu 13.4.2020.