Teemu Salminen

Ylläpidettävyys Unity3D-projekteissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

5.5.2014

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Teemu Salminen Ylläpidettävyys Unity3D-projekteissa 48 sivua + 1 liite 5.5.2014
Tutkinto	insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Tekninen arkkitehti Otso Alho Lehtori Simo Silander

Unity3D on ensisijaisesti pelien kehittämiseen tehty työkalu, jota voidaan käyttää niin pienissä kuin suurissakin ohjelmistoprojekteissa. Unity3D soveltuu niin kokeneille ohjelmistoalan ammattilaisille kuin harrastelijakoodareille, ja sitä voidaan käyttää 3D-pelien lisäksi myös 2D-pelien sekä monenlaisten interaktiivisten ja graafisten sovellusten tekemiseen.

Tämä insinöörityö sisältää pintapuolisen kuvauksen kehitysympäristöstä, jossa käytetään Unity3D:tä menetelmin, jotka parantavat ylläpidettävyyttä. Kehitysympäristö on Intelle Innovations Ltd:lle rakennettu kehitysympäristö, jonka on tarkoitus tukea Intellen reaaliaikaisten 3D-tuotteiden kehitystä. Kehitysympäristöön kuuluu Unity3D, versionhallintajärjestelmä, käännösautomaatio ja muita Unityyn kytkettäviä työkaluja.

Toisaalta työssä käsitellään myös ohjelmakoodin laatuun vaikuttavia tekijöitä. Työssä käydään läpi siistin koodin elementtejä sekä koodin rakennetta ja arkkitehtuuria parantavia tekijöitä.

Insinöörityö pohjautuu Intelle Innovations Ltd:n tekemän tuotteen kehitykseen, mutta vaikka tuotetta kehitetään Unity3D-sovelluksen avulla, ovat monet työssä käsiteltävät asiat yleispäteviä muissakin ohjelmistokehitysympäristöissä, joissa ei käytetä Unity3D:tä.

Avainsanat	Unity3D, Ylläpidettävyys, Clean Code, C#

Author(s) Title Number of Pages Date	Teemu Salminen Maintainability in Unity3D projects 48 pages + 1 appendix 5 May 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Otso Alho, Technical Architect Simo Silander, Senior Lecturer

Unity3D is a tool developed primarily for game development purposes and it can be used in both small and large software development projects. Unity3D suits experienced programming professionals as well as amateur coders, and in addition to 3D games, it can be used to develop 2D games and various kinds of interactive and graphical applications.

This study includes a superficial description of a development environment, where Unity3D is used with methods that improves maintainability. The development environment is built for Intelle Innovations Ltd, and is intended to support the development of Intelle's real-time 3D applications. The development environment includes Unity3D, version control tools, continuous integration system and some plugins for Unity.

This study also covers some factors that has an impact on code quality. Elements and structure of clean code will be dealt with among with factors that enhance code's architecture.

This thesis is based on development of Intelle Innovations Ltd's product. Even though Unity3D is broadly used on the software development process in Intelle, many of the issues and cases on this thesis are valid in software development in general, even if Unity3D is not involved.

Keywords	Unity3D, Maintainability, Clean Code, C#

Sisällys

Lyhenteet

TDD	Test Driven Development. Ohjelmistokehitysmalli, jossa ohjelmointi tapahtuu kirjoittamalla ensin testit ja vasta sitten itse ohjelmakoodi.
REST	Representational State Transfer. HTTP-protokollaan perustuva ohjelmointirajapintamalli.
HTTP	Hypertext Transfer Protocol. Web-selainten ja palvelinten käyttämä tiedonsiirtoprotokolla.
LINQ	Language Integrated Query. .NET-kehykseen kuuluva komponentti, jolla voidaan tehdä datakyselyjä.
MVC	Model-View-Controller. Arkkitehtuurimalli, jossa tilaa sisältävä logiikka, käyttöliittymä ja käskyjä vastaanottava kontrolleri on eroteltu toisistaan.
SRP	Single Responsibility Principle. Olio-ohjelmoinnissa käytetty periaate, jonka mukaan jokaisella luokalla on vain yksi vastuualue.
URL	Uniform Resource Locator. Merkkijono, jolla määritellään mm. verkko-osoitteita, joiden avulla verkkosivut haetaan.
DTO	Data Transfer Object. Olio, jonka avulla siirretään dataa. DTO ei sisällä yleensä logiikkaa.
IDE	Integrated Development Environment. Ohjelmointiympäristö, jolla tarkoitetaan sovellusta ja sen liitännäisiä, jotka mahdollistavat ohjelmoinnin.
GUI	Grafical User Interface. Graafinen käyttöliittymä.

1 Johdanto

Tämä opinnäytetyö keskittyy ohjelmiston ylläpidettävyyteen Unity3D:llä tehtävässä projektissa, joka vaatii hyvää ylläpidettävyyden tasoa. Ylläpidettävyyttä pyritään parantamaan tunnetuilla ketterän ohjelmistokehityksen työkaluilla ja tekniikoilla sekä Unity3D:lle ominaisilla teknologioilla. Projekti on Intelle Innovations Ltd:n tuotekehitysprojekti, jonka tuloksena syntyy tuote, jonka kehityksen ennustetaan jatkuvan usean vuoden ajan. Projekti on vielä alkuvaiheissa, joten nyt on oikea aika tutkia ja kehittää parhaita käytäntöjä tuotteen kehittämiseen, jotta kehitystiimi olisi nyt ja tulevaisuudessa mahdollisimman tuottava. Tuotteesta kerrotaan salassapitosopimusten vuoksi abstraktilla tasolla.

Tämä raportti on jaettu pääpirtein neljään osaan. Ensimmäisessä osassa, joka rajoittuu lukuun 2, motivoidaan lukijaa aiheeseen ja kerrotaan, millaisissa tilanteissa hyvä ylläpidettävyys nousee merkittävään rooliin. Tämän jälkeen luvuissa 3 - 5 käsitellään ohjelmoinnin yleisiä ohjenuoria liittyen sovelluksen arkkitehtuuriin ja koodin muotoiluun, joita voidaan soveltaa kaikessa ohjelmoinnissa riippumatta siitä, minkälaista ohjelmistokehystä tai kehitysympäristöä käytetään. Kolmannessa osassa luvuissa 6 - 8 syvennytään Unity3D-projekteissa käytettäviin konkreettisiin työkaluihin ja tekniikoihin, jotka parantavat ylläpidettävyyttä nimenomaan Unityn kanssa työskenneltäessä. Lopuksi luvuissa 9 - 10 arvioidaan ja kootaan yhteen tutkimuksessa syntyneitä tuloksia.

Työ on suunnattu ohjelmistoalan ammattilaisille, jotka ovat jo työskennelleet jonkin aikaa ohjelmoijana osana tiimiä, mahdollisesti juuri Unity3D:n parissa. Ilman tällaista kokemusta aiheet eivät välttämättä vaikuta merkityksellisiltä, ja käsitteet saattavat tuntua vierailta. Tämä työ voi kuitenkin toimia myös oppaana tuoreelle ohjelmoijalle, joka on vasta aloittamassa työskentelyä osana kokenutta tiimiä eikä halua sekoittaa ja hidastaa muun tiimin työtä. Aloittavaa ohjelmoijaa kannustaisin lisäksi erityiseen valppauteen ja itsensä aktiiviseen kehittämiseen, sillä vaikka nämä tekniikat ja käytännöt ovat Intellellä hyväksi havaittuja, on kuitenkin muistettava, että niin hienoja tekniikoita ei ole keksitykään, etteikö olisi mahdollista kehittää vielä hienompia.

2 Ylläpidettävyyden merkitys

Ohjelmistot suorittavat toimenpiteitä, jotka on kirjoitettu ohjelmakoodiin. Jos ohjelmiston tekemät toimenpiteet ovat ohjelmistokehittäjän tarkoituksenmukaisia kaikissa tilanteissa, voidaan ajatella, että ohjelmisto toimii halutulla tavalla. Tämä ei kuitenkaan kerro koko totuutta ohjelmiston laadusta ja ylläpidettävyydestä. Miten ohjelmisto toimii, jos käyttäjä käyttää ohjelmistoa tavalla, jota ei ole suunniteltu? Mitä pitää tehdä, jos PC:llä toimivaa ohjelmistoa halutaankin käyttää Android-laitteella? Miten edetään, jos ohjelmistoon halutaan lisää ominaisuuksia?

Ohjelmiston ylläpidettävyys nousee hyvin tärkeään rooliin, kun ohjelmiston toiminnallisuutta pitää muuttaa tai siihen pitää tehdä uusia ominaisuuksia. Ohjelmiston toimittaja on saattanut vaihtua, ja jos koodi on rakenteeltaan sekavaa, voi uudella toimittajalla kulua lukemattomia työtunteja pienenkin ominaisuuden lisäämiseen. Jos esimerkiksi koodin jokaisesta osasta on lukuisia riippuvuuksia ohjelmiston muihin osiin, voi pienikin muutos koodissa aiheuttaa koko ohjelmiston toimimattomuuden. Tämä tekee pientenkin virheiden korjaamisesta tuskallisen ja vaativan tehtävän. [Martin 2009: 3.]

Ongelma nousee esiin erityisesti pelialalla ja 3D-järjestelmissä. Syitä on monia, mutta ongelma saattaa liittyä osittain siihen, että pelialalla hyviä käytäntöjä on kehitetty huomattavasti vähemmän aikaa kuin muilla ohjelmistoaloilla. Lisäksi reaaliaikaiset 3D-järjestelmät ovat hyvin monimutkaisia kokonaisuuksia, ja niihin on huomattavasti haastavampaa soveltaa monia muissa kehitysympäristöissä käytettäviä hyväksi havaittuja ohjelmointikäytäntöjä. Lisäksi tuotteen elinkaaren ollessa hyvin pitkä joudutaan yhä useammin palaamaan vanhaan koodiin, joten koodin on oltava siistiä, jotta vanhakin koodi olisi nopeasti muutettavissa.

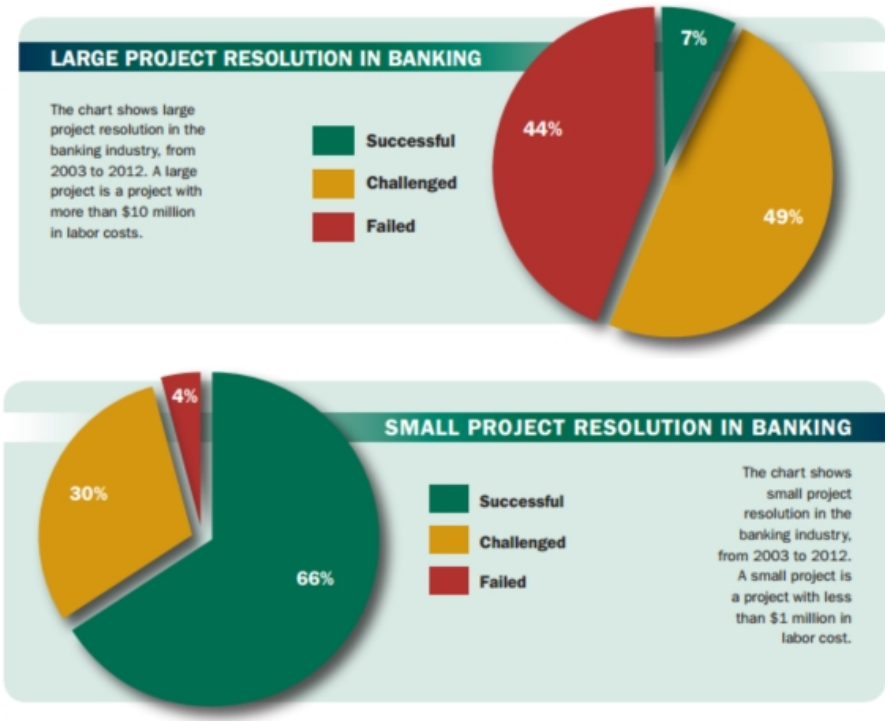
Olen tätä kirjoittaessani työskennellyt Intellellä n. 10 kuukautta, jonka aikana olen oppinut keskeisimpiä asioita ohjelmointityöstä ammattimaisessa ympäristössä. Olen nähnyt, miten huolimattomalla ohjelmistokehityksellä on mahdollista saada koko tuotantotiimi polvilleen niin, että tuotteen kehitystä ei yksinkertaisesti voida jatkaa. Tällaisessa projektissa loppuvaiheissa yhden bugin korjaaminen aiheuttaa usein muutaman uuden bugin, ja uuden ominaisuuden lisäämiseksi täytyy muuttaa useita eri skriptejä niin, että uuden ominaisuuden aiheuttamat bugit voivat löytyä sovelluksen useista eri osista. Lopulta kehittäjät pelkäävät jokaisen koodirivin kirjoittamista, sillä he

pelkäävät, että heidän kirjoittamansa koodi rikkoo jälleen sovelluksen, eikä tuotetta voida enää kehittää ilman laajamittaisia refaktorointeja. Olen myös ollut mukana tekemässä tällaista refaktorointia ja valitettavasti joudun myöntämään, että me emme onnistuneet saamaan tuotetta oikeille raiteille edes laajalla, koko tiimin työllistävällä viikkojen refaktoroinnilla. Kun olimme jatkaneet refaktorointia n. 3 viikkoa, olimme saaneet refaktoroinnilla vain pieneen osaan koodikannasta parannuksia. Huomasimme, että refaktorointi ei tulisi maksamaan itseään takaisin. Joskus voi olla tehokkaampaa kirjoittaa koko sovellus uudelleen paremman arkkitehtuurin päälle kuin refaktoroida vanhaa sotkuista koodia askel askeleelta parempaa arkkitehtuuria kohti.

Jokainen ymmärtää, että on hyvin epäedullista joutua kirjottamaan koko sovellus uudelleen. Tällaisessa toimenpiteessä ei saada yhtäkään uutta ominaisuutta, mutta miestyötunteja kuluu usein enemmänkin kuin alkuperäisen sovelluksen kehittämiseen, sillä arkkitehtuurin kehittämiseen kuluu oma aikansa. Sen takia minä olen asettanut suurimmaksi pelokseni sen, että omalla koodilla aiheuttaisin tiimini lähtevän pikkuhiljaa väärille raiteille. Tämän pelon ajamana pyrin jatkuvasti kehittämään näkemystäni selkeästä arkkitehtuurista ja hyvistä ohjelmointikäytännöistä ja pyrin tarkastelemaan oman koodini laatua eri näkökulmista aina kirjoittaessani sitä. Palaan myös vanhoihin koodeihini vähän väliä ja peilaan usein tänään tehtyjä ratkaisuja eiliseen koodiini. Jos koodistani löytyy jotain pientä parannettavaa, siivoan aina ilmeisimmät sotkut. Jos paranneltavaa on paljon, niin järjestän sprint backlogiin refaktorointitehtävän koskien sotkuista koodia, jotta voin paneutua huolella koodin laadun parantamiseen. Näin koodin laatu pysyy juuri niin hyvänä, kuin mihin kehitystiimin ymmärrys hyvästä koodista yltää.

Kun koodikantaa ei päästetä vanhentumaan ja koodin laatu pidetään aina ajan tasalla ja kun tiimi kehittää vielä jatkuvasti osaamistaan ja koettaa löytää uusia parempia tapoja tehdä asioita, päästään tilanteeseen, missä tuotteen laatu paranee jatkuvasti kehityksen myötä. Tämä ei ole itsestään selvää ohjelmistoalalla, sillä monissa onnistuneissakin projekteissa tuotteesta voi tulla buginen ja huonolaatuinen, jos korkeaa tuottavuutta tavoitellen pyritään vain saamaan paljon uusia ominaisuuksia tehtyä kiinnittämättä huomiota kokonaiskuvaan. Kuvassa 1 olevista kaavioista ilmenee selkeä ero laajojen ja pienten projektien onnistumisissa. Kaavion perusteella isoissa yli 10 miljoonan dollarin projekteissa onnistumisprosentti on 7 %, kun taas pienissä alle miljoonan dollarin projekteissa onnistumisprosentti on 66 %. Epäonnistumisia isoissa

projekteissa on 44 % ja pienissä 4 %. Tuloksista voidaan päätellä, että vaikeusaste kasvaa nopeasti projektin laajentuessa.



Kuva 1. Onnistumissuhteet isoissa ja pienissä pankkialan ohjelmistoprojekteissa vuosina 2003 - 2012

Hyvin ylläpidettävä ohjelmistoarkkitehtuuri on ohjelmistoyrityksessä ainoa tapa säilyä kilpailukykyisenä. Tämän takia ylläpidettävyys on erittäin mielenkiintoinen ja tärkeä aihe, ja siitä voitaisiin kirjoittaa useita kirjoja. Täydellisyyttä tavoitellessa on kuitenkin hyvä muistaa myös, että pelkkien työkalujen ja prosessien kehittäminen ei välttämättä kehitä itse tuotetta eteenpäin. On löydettävä kultainen keskitie tuotteen ja tuotantoympäristön kehittämisen väliltä, sillä ne tukevat toinen toisiaan. Tuotetta kehittäessä syntyy tarpeita paremmille käytännöille, uusille automaatioille ja uudelleenkäytettäville moduuleille, ja tuotantoympäristön kehittäminen luonnollisesti nopeuttaa tuotteen kehitysprosessia.

3 Koodin luettavuus

Tässä luvussa keskitytään siihen, millaisessa muodossa koodi käytännössä tuotetaan. Käyn läpi, kuinka abstraktiotasoilla voidaan rajata koodilohkojen kontekstia, jotta

koodin lukija ymmärtäisi koodia paremmin. Esitän, kuinka koodia voidaan jakaa erilaisiin osiin, jotta koodikannasta saadaan niin korkealla kuin matalallakin tasolla järjestelmällisempi ja rakenteellisempi. Kerron myös, kuinka koodin luettavuutta voidaan parantaa sovituilla muotoilukäytännöillä sekä kuinka nimeämisellä voi vaikuttaa dramaattisesti koodin selkeyteen ja ymmärrettävyyteen.

Koodia voi kirjoittaa monella tavalla. Koko ohjelman toiminta voidaan sisällyttää yhteen Main-funktioon (vaikkakaan Unity3D-ympäristössä ei kirjoiteta Main-funktioita, sillä ohjelmointi tapahtuu skriptien kautta, joita ajetaan UnityEngine-ohjelmistokehyksessä), joka sisältää kaikki ohjelman muuttujat, logiikan, sivuvaikutukset jne. Ohjelman kasvaessa muutamaa riviä pidemmäksi tällainen arkkitehtuuri (tai sen puute) muodostuu äärimmäisen vaikeaselkoiseksi. Toisaalta koodin rakenteeseen voidaan kiinnittää enemmän huomiota ja rakentaa ohjelmasta kaunista arkkitehtuuria noudattava kokonaisuus, jossa jokaisen funktion toiminta on helposti ymmärrettävä, jokaisen luokan tehtävä on selkeä ja jokaisella muuttujalla on luonnollinen paikkansa. Tällaista ohjelmaa voi olla hyvin miellyttävää jatkokehittää, sillä sen jokaista osaa tutkiessaan ohjelmoija ymmärtää, mitä siinä tehdään ja miksi.

Ammattiympäristössä samaa ohjelmakoodia lukee ja kehittää useita ohjelmoijia, joten on tärkeää, että koodi on helposti ymmärrettävää ja läpinäkyvää, sillä epäselvän koodin kehittäminen on työlästä ja hidasta. Jos koodin laatuun ei kiinnitetä huomiota, muodostuu ohjelmaan enemmän virheitä. Näiden korjaamiseen kuluu pitkiä aikoja, joka johtaa tuottavuuden ja tätä myöten kilpailukyvyn heikkenemiseen. Muutosten tekeminen johonkin toiminnallisuuteen on hidasta, jos toiminnallisuuden aikaansaavan ohjelmakoodin pelkkään ymmärtämiseen kuluu enemmän aikaa kuin mitä sen kirjoittamiseen alun perin on kulunut.

Siisti koodi (eng. clean code) on melko abstrakti käsite, ja jokaisella ohjelmoijalla on oma mielikuvansa siitä, millaista siisti koodi on. Itse ajattelen koodin siisteyttä samaan tapaan kuin kodin siisteyttä. Kotini on siisti, kun roskat on viety pois haisemasta ja astiat ovat puhtaina kaapeissaan. Siistissä kodissa ei ole täysin tarpeettomia tavaroita ympäriinsä, eikä vaatteet loju mytyssä lattioilla, tuoleilla, pöydillä tai missään muuallakaan niille kuulumattomassa paikassa, vaan ovat siististi viikattuna kaapeissa. Samoin kaikki muutkin esineet täyttävät jonkin tehtävän ja ovat siistissä järjestyksessä niille suunnitelluilla paikoilla. Tietokone ja työpöytä ovat työhuoneessa, ja sänky ja yöpöydät ovat makuuhuoneessa.

Siisti koodi sen sijaan ei sisällä pois kommentoituja vanhoja lohkoja, eikä tarpeettomia funktioita, joita ei kutsuta missään. Siistissä koodissa funktiot ja luokat ovat selkeitä kokonaisuuksia, jotka täyttävät jonkin tehtävän ja toteuttavat yhden vastuun periaatetta (eng. Single Responsibility Principle, SRP). Ohjelman osat on ryhmitelty osakokonaisuuksiksi ja ne on jaoteltu abstraktiotasoihin, eikä koodi hypi rivi riviltä eri abstraktiotasoilla.

Helposti luettavan koodin kriteerit pätevät niin Unity3D-ympäristössä kuin sen ulkopuolellakin, joten kyseessä ei ole ainoastaan Unity-projekteja koskeva aihe.

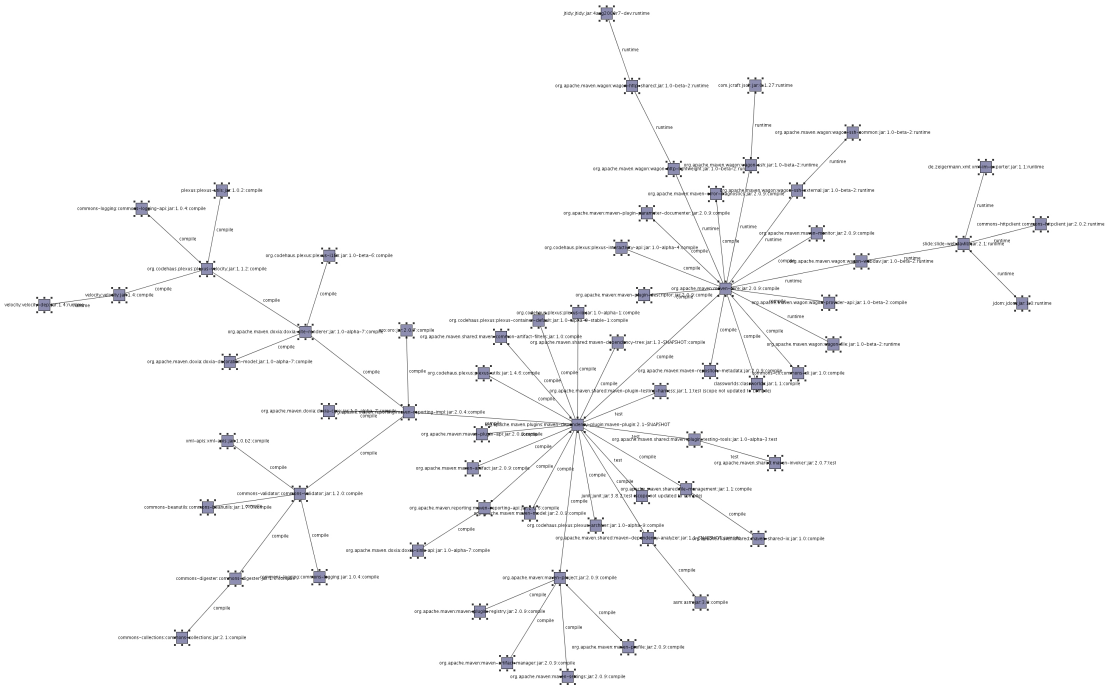
3.1 Abstraktiot

Abstraktioiden ymmärtäminen on ollut minulle suurimpia ahaa-elämyksiä ohjelmoinnissa. Abstraktioiden avulla on mahdollista hallita äärettömän suuria ohjelmistokokonaisuuksia, ja ilman jatkuvasti kehittyviä abstraktioita ei kehitystä tapahtuisi koko ohjelmistoalalla. Jos tietokoneen toimintaa ajatellaan konkreettisella tasolla, liipaisee muistissa (kovalevyllä, BIOSin ROM, RAM, CD-levy tms.) oleva jälki tietyn suuruisia jännitteitä, jotka johdetaan prosessorin loogisiin portteihin (AND, OR, NAND, NOR, NOT, XOR, XNOR), jotka ovat käytännössä erilailla järjestettyjä transistoreja. Jotta eri suuruisista jännitteistä ja transistorien vuotovirroista yms. detaljeista päästäisiin eroon, on keksitty alimman tason abstraktio, binääridata. Kun loogisiin portteihin tulevat jännitteet ovat joko 1 tai 0, voidaan jo paljon helpommin suunnitella erilaisia digitaalisia järjestelmiä ja 1940-luvulla tietokoneita ohjelmoitiinkin tällä ns. konekielellä [Konekieli 2013]. Seuraavalla abstraktiotasolla on symboliset konekielet, joilla konekieltä voitiin ohjelmoida jo binäärilukujen sijaan lyhenteisiin ja sanoihin perustuvilla komennoilla. Unity3D taas on nykyajan mittapuulla hyvin korkean tason ohjelmistokehys. Sillä voi muutamalla komennolla luoda mallinnetun 3D-objektin pelimaailmaan, asettaa sille massa ja siihen vaikuttava painovoima, antaa sille 3D-koordinaatistossa voimaimpulssin ja asettaa objektille haluamansa pyörimisnopeuden. Olen tavannut ihmisiä, jotka ovat ylpeinä kertoneet ohjelmoineensa paljonkin binäärikomennoilla, mutta rohkenen epäillä, että he eivät ole ikinä saaneet ohjelmoitua binäärikomennoilla kovin hienoja 3D-sovelluksia.

Abstraktiot ovat siis keskeisessä roolissa ohjelmoinnissa. Ohjelmoijan täytyy päättää uusia tietorakenteita muodostaessaan, minkälainen kokonaisuus muodostaisi sopivan

abstraktion. Jos koodikannassa on useita samankaltaisia tietorakenteita, indikoi tämä usein siitä, että näille tietorakenteille voisi olla ylemmän tason abstraktio. Karkeana esimerkkinä voi kertoa, että jos ohjelmassa on oliot “mies”, “nainen”, “poika” ja “tyttö”, joita kaikkia yhdistää etunimi, sukunimi, syntymäpäivä, paino, pituus jne, niin näiden tietorakenteiden ylemmän tason abstraktio voisi olla “ihminen”. Kun näille olioille halutaan tehdä ihmiselle ominaisia funktioita, kuten “lue kirjaa”, voidaan funktion toteutuksessa käyttää ihmistä sen sijaan, että jokaista oliota varten tehtäisiin erikseen “lue kirjaa” -funktio. Tämä auttaa myös koodin ymmärtämisessä, sillä koodin lukijan ei tarvi ihmetellä, miten kirjan lukeminen liittyy miehisyyteen tms. Sen sijaan lukija ymmärtää heti, että kyky lukea kirjoja on nimenomaan ihmisille ominainen taito, eikä tässä ole kyse minkäänlaisesta sukupuoleen tai ikään liittyvistä erityisominaisuudesta. Myös ohjelmointikielet tukevat abstraktioita erinomaisesti, sillä mm. Unityn käyttämässä C#:ssa ja JavaScriptissä voi tehdä esim. abstrakteja luokkia, joilla voi olla muuttujia ja funktioita, jotka ovat käytössä perivissä luokissa.

Kun sovelluksen arkkitehtuuri on selkeä ja hyvin suunniteltu, sen riippuvuustaulu muodostuu pienemmistä ja isommista kokonaisuuksista, kuten kuvan 2 riippuvuustaulussa. Sotkuisessa koodikannassa jokainen luokka sisältää riippuvuuksia joka puolelle sovellusta, ja riippuvuustaulu näyttää yhdeltä sotkuiselta lankakerältä, jossa riippuvuudet muodostavat yhden ison sekasotkun, josta ei pysty erottelemaan ylemmän ja alemman tason komponentteja.



Kuva 2. Siisti riippuvuustaulu.

Jotta koodi pysyisi ymmärrettävänä ja helppona sisäistää, täytyy abstraktiotasojen välillä liikkumista rajoittaa. Jos esimerkiksi REST-rajapintaolion tehtävänä on lähettää GET- ja POST HTTP -kyselyjä, ei tällaisen olion palveluiden käyttäjä halua tietää kyselyissä käytettävistä data streameistä, streamien lukijoista, byte-taulukoista tai muista REST-rajapintaolion käyttämistä yksityiskohtaisista tiedonsiirtomenetelmistä. Palvelujen käyttäjä haluaa vain kertoa oliolle, mihin osoitteeseen kysely pitää lähettää ja mitä dataa kysely sisältää. Kuvitellaan, että sovelluksessa on void UploadUserInfo -funktio, joka ensin hakee vanhat käyttäjätiedot, päivittää ne ja lähettää takaisin palvelimelle. Jos tähän funktioon olisi määritelty yksityiskohtaisesti matalan abstraktiotason toimenpiteitä kyselyn lähettämiseksi, olisi funktiota muuttavalla ohjelmoijalla suuri vaiva päästä selville, mitä kohtaa hänen pitäisi muuttaa tai mitä funktiossa tarkalleen oikeastaan tapahtuu. Nyrkkisääntönä onkin, että funktion tulee pidättäytyä yhdessä abstraktiotasossa, ja siitä tulee kutsua funktioita, jotka ovat enintään yhden abstraktiotason korkeammalla tai matalammalla. Tämä mahdollistaa ohjelmoijan keskittymisen kapeaan ja helposti ymmärrettävään näkyvyysalueeseen kerrallaan.

3.2 Sovitut muotoilukäytännöt

Koodin muotoilu on tärkeä osa koodin ymmärrettävyyttä. Jos koodin muotoilu on siistiä ja järjestelmällistä, rohkaisee se samanlaiseen järjestelmällisyyteen toiminnallisuuksien muuttuessa ja koodin kasvaessa. Tässä mielessä koodin siisteys on kestävämpää kuin itse toiminnallisuudet, sillä toiminnallisuuksia voidaan joutua muuttaa useaan kertaan ensimmäisen toteutuksen jälkeen, mutta koodin siisteys voi kantaa yllättävänkin pitkälle olettaen toki, että kaikki koodiin koskevat ohjelmoijat ovat ammattilaisia ja ymmärtävät koodin laadun tärkeyden.

Kun tiimi sopii yhteiset ohjelmointikäytännöt, joita noudatetaan, oppivat tiimin jäsenet erottamaan koodin erilaisia rakenteita nopeammin, mikä säästää ohjelmoijien kallisarvoista työaikaa. Sovittavia asioita voivat olla esimerkiksi seuraavat:

- muuttujien, metodien, luokkien yms. nimeämiskäytännöt

- metodien ja luokkien koot
- rivien maksimipituudet
- kommentointikäytännöt
- sisennykset ja aaltosulut
- suojaustasot.

Dokumentti Intellellä käytetyistä ohjelmointikäytännöistä on liitteenä. Käyttämämme käytännöt ovat melko universaaleja ja teknologioista riippumattomia, mutta tällainen dokumentti voi sisältää myös hyvin ympäristöspesifejä käytäntöjä. Intellen ohjelmointikäytännöistä löytyvä IEnumeratorien suojaustasoa koskeva ohje on esimerkki ympäristöspesifistä käytännöstä, sillä IEnumeratoria kutsuva StartCoroutine-funktio on UnityEnginen toiminnallisuus.

3.3 Nimeämiset

Hyvän nimeämisen tärkeyttä havainnollistaa mielestäni hyvin esimerkiksi Malbolge-ohjelmointikieli. Lähdekoodissa 1 on esimerkki Malbolgella ohjelmoidusta “Hello World!” -ohjelmasta:

```
('&%:9]!~}|z2Vxwv-,POqponl$Hjig%eB@@>}=<M:9wv6WsU2T|nm-,jcL(I&%$#"
`CB]V?Tx<uVtT`Rpo3NlF.Jh++FdbCBA@?]!~|4XzyTT43Qsqq(Lnmkj"Fhg${z@>
```

Lähdekoodi 1. Malbolgella ohjelmoitu “Hello World” -sovellus.

Ohjelma yksinkertaisesti tulostaa näytölle sanat “Hello World!”. Tutustumatta sen tarkemmin Malbolgen syntaksiin, minulla ei ole mitään mahdollisuutta selvittää, mitä ohjelma oikeastaan tekee lukemalla ohjelmakoodia. Tämä johtuu siitä, että syntaksissa ei esiinny mitään ihmisten ymmärrettäviä selkokielisiä sanoja tai lyhenteitä, eikä siitä näin voi mitenkään päätellä, mitä se tekee.

Sama pätee myös Intellellä käytettävään C#-ohjelmointikieleen, jos nimeämiset on tehty huonosti, sillä ohjelma koostuu kehittäjien määrittelemistä muuttujista, metodeista ja luokista, joille kehittäjien täytyy myös keksiä nimet. Nimien ollessa arvoituksellisia joutuu koodin lukija käymään koodia läpi rivi riviltä yrittäen pitää mielessä kaikkien muuttujien ja funktioiden tarkoitukset, mikä tekee jatkokehityksestä ja virheiden korjauksesta tuskastuttavan hidasta.

Nimien pituutta ei pidä pelätä. Yksittäisestä nimestä ei kuitenkaan kannata tehdä myöskään koko rivin täyttävää tarinaa muuttujan alkuperästä, sillä tällaisten nimien lukeminen on työlästä ja alkuperän selventäminen turhaa, sillä nimen tuleekin ilmaista vain muuttujan näkyvyysalueella vaadittavaa selitystä muuttujan tarkoituksesta. Jos nimeä joutuu venyttämään kovin pitkäksi, voi olla paikallaan miettiä, onko funktion tai luokan näkyvyysalue liian suuri. Esimerkiksi jos muuttujan nimi on “locationsInGetResponseOfOldAssetBundleManifestFromAssetBundleBuilder”, niin voidaan melko varmasti sanoa, että joko nimi kertoo paljon asioita muuttujan näkyvyysalueen ulkopuolelta, tai sitten muuttuja on määritelty luokassa tai funktiossa, joka on kasvanut aivan liian suureksi. Joka tapauksessa tällaisen muuttujan nimen tulisi aiheuttaa toimenpiteitä. Siistissä koodissa tulisi noudatta SRP:tä, joten funktioiden ja luokkien pitäisi olla sen verran lyhyitä, ettei tällaisia muuttujia tarvitse ikinä määritellä.

Lyhenteiden käyttö voi olla vaarallista nimeämisissä. On tapauksia, joissa lyhenteet ovat enemmän kuin hyväksyttyjä, kuten esimerkiksi luokassa nimeltä AssetBundleUrlsDto. Luokan nimi kertoo selkeästi, että luokassa ei ole logiikkaa, vaan se on tiedonsiirtoon tarkoitettu luokka, joka sisältää pelkkiä osoitteita AssetBundle-tiedostoihin, sillä nimessä käytetyt lyhenteet URL ja DTO ovat ohjelmoijille tuttuja käsitteitä. Jos nimi olisi kirjoitettu auki AssetBundleUniformResourceLocatorDataTransferObject, sekoittaisi tämä monen ohjelmoijan silmät, ja luokka pitäisi avata ja tutkia tarkemmin, jotta lukija pääsisi selville luokan tarkoituksesta. Sen sijaan jos muuttujan nimi on esimerkiksi PCGO, jotka tulisivat vaikka sanoista PlayerControllerGameObject, täytyisi lukijan jälleen selvittää tarkemmin muuttujan tyyppi ja sisältö, jotta pääsisi selville muuttujan tarkoituksesta. GO on valitettavasti iskostunut monien Unity-kehittäjien nimeämiskäytäntöihin, sillä GameObject-olio on Unityssa ehkä yleisimmin käytetty olio. Minulle tämä teki muuttujien nimistä arvoituksellisia, kun aloitin Intellellä työskentelyn. Esimerkiksi PlayerGO-muuttujasta minulla tuli ensimmäisenä mieleen, että tämä muuttuja panee pelaajan menemään johonkin. Haluan kuitenkin taistella tällaisia käytäntöjä vastaan,

sillä muuttujan tyypin ilmaiseminen sen nimessä on nykyaikaisissa IDEissä täysin turhaa, sillä tyypin saa yleensä halutessaan selville esimerkiksi viemällä hiiren muuttujan päälle. Sen sijaan tyypin kertominen nimessä vain tuo lisää luettavaa, mikä yksinkertaisesti tekee koodin lukemisesta ja ymmärtämisestä hieman hitaampaa.

Metodien nimien tulisi olla verbejä. [Martin 2009: 25.] Tämä on ymmärrettävää, sillä metodit aina tekevät jonkun asian, joten hyvin nimetty tekeminen kuvastaa metodin olemusta parhaiten. Jos metodilla yksinkertaisesti haetaan tai asetetaan jotakin tietoa, voidaan metodit nimetä esimerkiksi GetManifest() tai SetManifest(IManifest newManifest). Jos metodilla on jotakin sivuvaikutuksia, tulee niiden niin ikään ilmetä sen nimestä. Esimerkiksi metodi, joka piirtää ruudulle virheilmoituksen, voitaisiin nimetä DisplayError(IError error). Jos metodin nimessä on Or- tai And-sanoja, haiskahtaa se siltä, että metodi rikkoo SRP:tä. Jos esimerkiksi metodin nimi on PauseGameAndAutosaveStatus(), niin metodilla selkeästi näyttäisi olevan useampia tehtäviä. Luokkien ja muuttujien nimien ei sen sijaan pitäisi olla verbejä, sillä muuttujat ovat aina jonkin asian instansseja, ei niinkään tehtäviä. Vaikka luokat saattavat sisältää pelkkiä metodeja eikä yhtään luokkamuuttujaa, mikä tekee luokasta puhtaasti loogisen, niin luokka itsessään on silti kokoelma näistä metodeista, eikä sinänsä kuvasta tekemistä.

3.4 Funktioiden käyttö

Ohjelmakoodi koostuu useista eri osista. C#:ssa on namespaceja, joilla voidaan rajata ja jaotella moduuleita osiin niin, ettei ohjelman kaikki luokat ole automaattisesti näkyvissä toisten namespacejen luokissa. Unityssa kehittäjät kirjottavat koodinsa skripteihin, jotka voivat sisältää useita luokkia. Luokat taas sisältävät luokkamuuttujia ja funktioita ja funktiot paikallisia muuttujia ja viittauksia muihin funktioihin ja luokkamuuttujiin. Näihin jokaiseen ohjelman osaan voisi perehtyä ja analysoida loputtomiin, mutta minulle tärkeimpään asemaan ovat nousseet funktiot. Namespaceja on helppo määritellä jälkikäteen. Skriptien sisältämiä luokkia voi siirrellä nopeasti leikkaa ja liitä -menetelmällä. Jos luokka kasvaa liian isoksi, niin on verrattain helppo jakaa luokan vastuualueita usealle eri luokille, tai refaktoroida luokan pääosat korkeammalle abstraktiotasolle ja eristää matalamman abstraktiotason toimenpiteet

muihin luokkiin. Mutta jos funktio on rakenteeltaan sekava, liian pitkä, useita abstraktiotasoja ja vastuualueita leikkaava ja vielä huonosti nimetty, niin koodin läpikäynti ja refaktorointi voivat muodostua ylitsepääsemättömäksi tehtäväksi. Ongelmakohtia funktioista niin kuin muistakin koodin osista voi etsiä staattisen analyysin työkaluilla, joita on saatavilla liitännäisinä nykyaikaisiin IDEihin.

Tärkeimpiä ohjenuoria funktioita kirjoittaessa on, että sen pitäisi olla lyhyt ja että sen tulisi tehdä vain yksi asia. [Martin 2009: 35.] Duplikaatteja pitäisi myös välttää funktioissa sekä suhteessa muihin funktioihin että funktioiden sisällä. Intellellä tunnetaan kirjoittamattomana sääntönä, että hyvän funktion tulisi useimmiten olla pituudeltaan alle 10 riviä pitkä, ja hyvin harvoin on perusteltua kirjoittaa yli 20 riviä pitkiä funktioita. Jos funktioon muodostuu lohkoja, joiden toimintoja mahdollisesti on vielä selitetty kommenteilla, kielii tämä yleensä siitä, että funktiossa tehdään useita eri asioita. Avainasemassa yhden asian tekemisessä on se, että funktion sisältämä koodi pysyy tasan yhden abstraktiotason alempana, kuin millä abstraktiotasolla funktion nimi on. On esimerkiksi selvää, että lähdekoodin 2 koodirivien ei pitäisi olla saman funktion sisällä:

```
string[] errorMsgs = new StreamReader("Path/ErrorLog.txt").codeString.Split(new char[]{'\n'};
DisplayConnectionError(errorMsg.Join("! ", errorMsgs);
```

Lähdekoodi 2. Eri abstraktiotasoja yhden funktion sisällä.

Tämä keksitty esimerkki yrittää havainnollistaa sitä, että funktion, jossa kutsutaan “DisplayConnectionError(string message)” -nimistä funktiota, ei pitäisi missään nimessä sisältää tietoa lokitiedostosta, josta se haetaan, StreamReaderistä, jolla tiedostoa luetaan, eikä erottimista, joilla tiedoston rivit erotellaan taulukkoon. Funktion parametriksi ei myöskään pitäisi antaa string-taulukkoa, joka yhdistetään yhdeksi merkkijonoksi, jonka on tarkoitus toimia virheilmoituksena. DisplayConnectionError -funktio on paljon korkeamman abstraktiotason funktio, jonka olettaisi esiintyvän esimerkiksi toisen korkeamman tason operaation, kuten “GetErrorMessage()”:n kanssa.

Funktiolle annettavien parametrien määrää tulee myös hillitä. Jos funktiolle joudutaan antamaan enemmän kuin kolme parametriä, haiskahtaa tämä siltä, että parametreista olisi syytä tehdä abstraktio tai että funktiossa tehdään useita asioita yhden sijaan. Useiden parametrien antaminen on harvoin välttämätöntä, mutta jos parametrit ovat yhden arvon järjestettyjä komponentteja, on käytäntö täysin sallittu. Esimerkki

tällaisesta voisi olla AddForce(int x, int y, int z), sillä kaikki parametrit ovat yhden voimavektorin komponentteja, jotka ovat järjestettyjä. Sen sijaan vaikka AddFurniture(var material, var color, var model) ei ole järkevästi parametrisoitu funktio, sillä nämä parametrit ovat erillisiä komponentteja vailla järjestystä, ja niistä muodostuu mahdollisesti huonekalu vasta tämän funktion sivuvaikutuksena. Jos huonekaluja halutaan lisätä, niin parametriksi pitäisi sitten antaa huonekaluja, kuten funktiossa AddFurniture(var furniture).

Metodien ei pitäisi koskaan sisältää sivuvaikutuksia. Jos kehittäjä noudattaa orjallisesti SRP:tä, ei hän joudu koskaan tällaiseen tilanteeseen, mutta tämä on niin tärkeä ohje, että siihen on hyvä kiinnittää erityistä huomiota. Jos funktion nimi on esimerkiksi PauseGame(), mutta todellisuudessa funktio pysäyttää pelin ja tekee automaattitallennuksen pelistä, on funktiolla sivuvaikutus joka jää niiltä kehittäjiltä huomaamatta, jotka eivät lue funktion sisältöä rivi riviltä tarkasti läpi. Tämä saattaa johtaa tilanteeseen, missä esimerkiksi AutoSave()-funktio aiheuttaa virheitä ohjelmassa, mutta ne ilmenevät vain PauseGame()-funktiota kutsuessa. Kun virhettä korjaava kehittäjä koettaa päästä ongelmasta selville, alkaa hän käymään läpi kaikkia niitä asioita, joita pysäytyksen aikana tehdään, kunnes vihdoin jäljet johtavat AutoSave()-funktioon. Kehittäjältä on kulunut turhaan kallisarvoista ohjelmointiaikaa vain sen takia, että edellinen kehittäjä on jättänyt funktioon sivuvaikutuksen, joka ei ilmene funktion nimestä.

4 Refaktorointi

Refaktorointi on koodin rakenteen parantamista ilman, että koodin toiminnallisuus muuttuu ulkopuolelta katsoen mitenkään. Refaktoroinnin tarkoituksena on tehdä koodista selkeämpää, ymmärrettävämpää ja helpompaa muuttaa. Esimerkiksi suunnittellumalleista tuttu mantra “ohjelmoi rajapintoja älä toteutuksia vasten” pyrkii tällaiseen helposti muokattavissa olevaan ohjelmistoarkkitehtuuriin. Järjestelmän ylläpidettävyyden kannalta koodia on välttämätöntä refaktoroida jatkuvasti, sillä vaikka rakenne olisi suunniteltu kuinka huolella ennen itse koodausta, voi hienokin arkkitehtuuri vanhentua siinä vaiheessa, kun sovellukseen täytyy ohjelmoida jotakin, mitä suunnitteluvaiheessa ei ole otettu huomioon [Fowler ym. 2002: 9]. Nämä uudet toiminnallisuudet eivät välttämättä sovi yksi yhteen suunniteltuun arkkitehtuuriin, joten

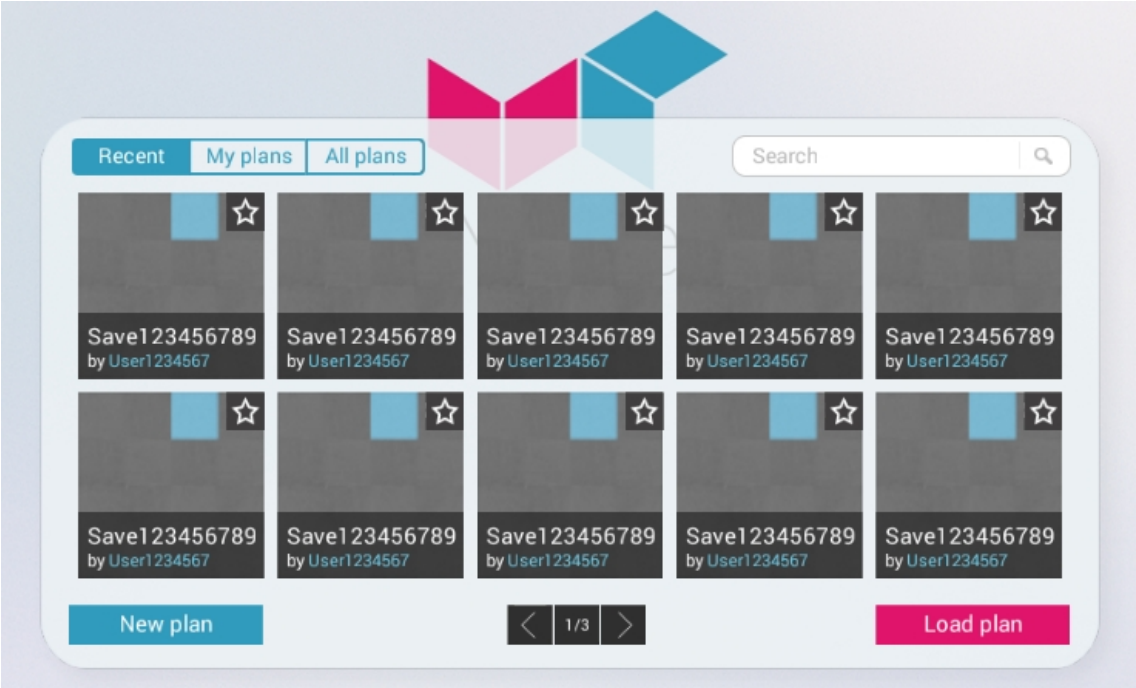
jotta koodikanta pysyisi siistinä, on järjestelmän rakennetta muutettava. Tämä tehdään refaktoroimalla olemassa olevaa koodia.

Refaktorointi on tärkeimpiä asioita, joita ohjelmoijan tulee osata. Jatkuvalla refaktoroinnilla sovelluksen laatu paranee jatkuvasti, mutta jos refaktorointia ei tehdä lainkaan, koodi vanhenee ja muuttuu ajan mittaan käyttökelvottomaksi. Tämän takia refaktoroinnilla on kriittinen vaikutus ohjelmiston ylläpidettävyyteen.

Vesiputousmallissa refaktoroinnin välttämättömyyttä on yritetty kiertää massiivisilla kaiken kattavilla suunnitteluvaiheilla, joiden jälkeen vasta koodataan itse toteutus. Käytäntö on kuitenkin osoittanut, että kaikkia järjestelmän osia ei voida ottaa huomioon ennen toteutusta, ja suunnitelma elää vielä koodauksen alettuakin. Tämän takia ketterät menetelmät ovatkin niin suosittuja ja niitä noudattaen päästään usein parempaan lopputulokseen. Refaktoroinnissa voidaan käyttää erilaisia menetelmiä, ja nykyiset IDEt tekevät refaktoroinnista paljon helpompaa ja nopeampaa. Esimerkiksi muuttujien uudelleennimeäminen on monissa IDEissä yhden napinpainalluksen takana niin, että nimi muuttuu kaikissa paikoissa, joissa kyseistä muuttujaa käytetään, eikä jokaista kohtaa tarvitse käydä läpi itse. Myös “Extract methdod” -työkalu, jonka avulla metodin sisältä voi eristää jonkin osion omaan metodiinsa, löytyy monista IDEistä valmiina.

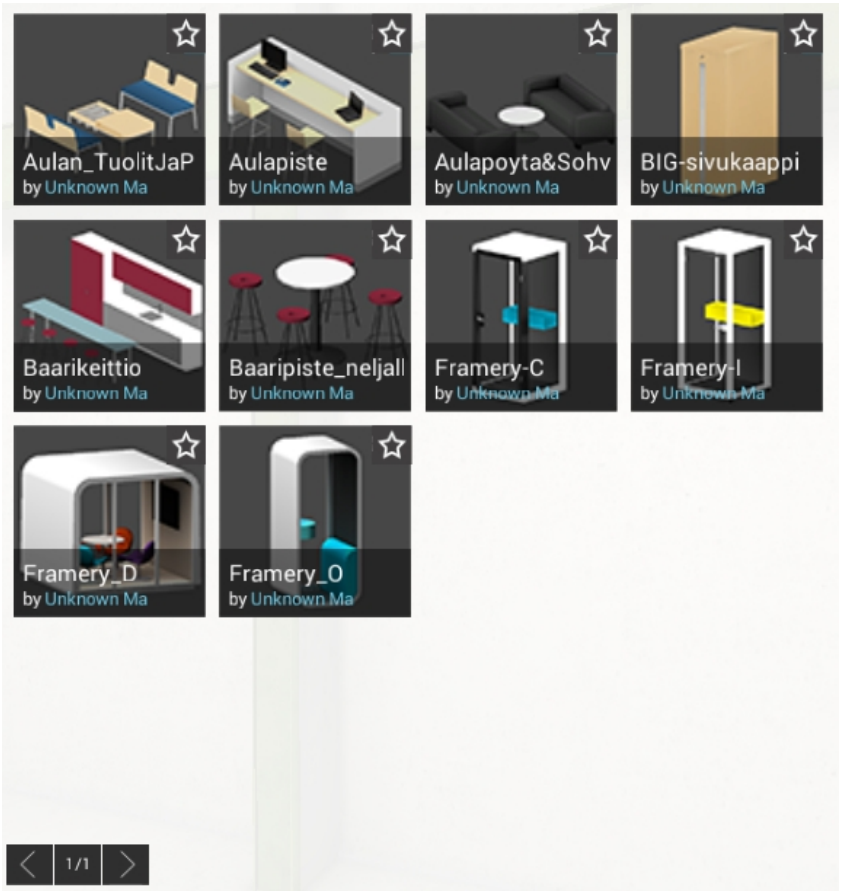
Koska pidän refaktorointia erittäin tärkeänä, mutta usein haastavana toimenpiteenä, esittelen refaktoroinnista konkreettisen esimerkin. Esimerkki havainnollistaa paremmin refaktoroinnissa käytettäviä toimenpiteitä ja tuo esille refaktoroinnin todellisia hyötyjä. Tässä luvussa käsitellään muutaman päivän kestänyttä refaktorointitehtävää, jonka sain Intellellä tehtäväkseni. Refaktoroinnin tuloksena syntyi luokka nimeltä CatalogController, joka kontrolloi katalogityylistä luetteloa alkioista, jotka on jaoteltu erillisille sivuille. Sivuja voi selata joko raahaamalla sivua hiirellä vasemmalle tai oikealle, tai vaihtoehtoisesti painamalla sivunvaihtonäppäimiä. Lukijan on hyvä huomata, että CatalogController pohjautuu UnityEnginen lisäksi vahvasti NGUI-liitännäisen käyttöön, joten näiden teknologioiden ymmärtäminen helpottaa huomattavasti refaktorointitoimenpiteiden ymmärtämistä.

Alkutilanteessa sovelluksessa oli kaksi osaa, jotka noudattivat samankaltaista rakennetta. Toinen oli kuvassa 3 esitetty tallenne-katalogi, josta käyttäjä voi selata aiemmin tehtyjä tallenteita.



Kuva 3. Tallenne-katalogi

Toinen oli kuvassa 4 esitetty huonekalu-katalogi, millä käyttäjä voi selata huonekaluja, joilla hän voisi sisustaa pelikenttää.



Kuva 4. Huonekalu-katalogi

Näiden katalogien muodostamiseen käytettiin yhteensä kuutta skriptiä, joiden yhteenlaskettu pituus oli n. 750 koodiriviä. Refaktorointiin kului kaikkiaan n. 20 tuntia, ja siihen kuului monia pieniä työvaiheita, joten en käsittele tässä kaikkia detaljitason refaktorointeja, sillä kaikkiaan alkuperäisten skriptien yhteispituus on n. 25-30 sivua Kingsoft Office Writer -tekstinkäsittelyohjelmassa.

Koska skriptien toiminnallisuus on suhteellisen yksinkertainen ja koska kaikki skriptit sisälsivät useita viittauksia toisiinsa, tulin siihen tulokseen, että halutut toiminnot pitäisi tehdä keskitetysti. Ensimmäinen askel refaktoroinnissa olikin yhdistää nämä skriptit yhdeksi luokaksi. Koska molempia katalogeja varten oli omat 3 skriptiä, keskityin aluksi täysin toisen katalogin käyttämien skriptien yhtenäistämiseen. Kun ensimmäinen katalogi saataisiin toimimaan yhdessä skriptissä, voitaisiin se parametrisoida niin, että sitä voidaan käyttää molempien katalogien kanssa.

Aluksi halusin päästä eroon LoadWidgetAlignListLeft-skriptistä, sillä se oli lähes kokonaan kopioitu NGUI-kirjastosta, joka on GUIn tekemiseen kehitetty Unity-liitännäinen. LoadWidgetAlignListLeft pohjautuu NGUIn UICenterOnChild-luokkaan, jonka tehtävä on muuttaa sisar-GameObject-olioiden sijainteja siten, että niiden sijainnit toisiinsa nähden säilyvät, mutta lähimpänä parent-GameObject-oliota oleva lapsi keskitetään automaattisesti parent-GameObject-olion sijaintiin. Tämä mahdollistaa katalogin sivujen selaamisen siten, että kun on raahannut keskellä olevan sivun vasempaan reunaan, keskittää UICenterOnChild katalogin automaattisesti oikealla olevan sivun keskelle. Se, että luokasta oli tehty oma muunnos, johtui siitä, että katalogeissa halutaan pitää kirjaa joka hetki sivun numerosta ja asettaa kauempia sivuja ei-aktiiviseen tilaan suorituskyvyn parantamiseksi.

UICenterOnChild tarjoaa kyllä mahdollisuuden asettaa delegaatti “OnFinished”, joka suoritetaan silloin, kun keskittäminen on tehty loppuun. Ongelma tämän delegaatin käyttämisessä sivun vaihtamisen sivuvaikutuksiin on se, että OnFinished kutsutaan vasta, kun sivun keskittäminen on suoritettu kokonaan loppuun, joten jos nopeasti selaa sivuja läpi antamatta sivujen pysähtyä, suoritetaan OnFinished-delegaatti huomattavalla viiveellä, eikä ohjelma toimi sulavasti. Ongelma saatiin ratkaistua manipuloimalla UICenterOnChildin käyttämää UIScrollView-luokan onDragFinished-delegaattia. Koska UICenterOnChild käyttää itsekin tätä delegaattia ja asettaa sen arvon ensimmäisen kerran Recenter-funktiossa, päädyin ratkaisuun, missä CatalogController kutsuu ensin UICenterOnChildin Recenter-funktiota, ja lisää tämän jälkeen UIScrollView:n onDragFinished-delegaattiin oman OnDragFinished-funktion. Tämä funktio tarkistaa, onko UICenterOnChildin keskittämä GameObject-olio sama GameObject-olio kuin edellisellä kerralla keskitetty sivu. Jos sivu on vaihtunut, muutetaan sivunumerot ja osoittimet oikeiksi. Lähdekoodissa 3 esitetään oma OnDragFinished-funktio ja kuinka se asetetaan UIScrollView’n onDragFinished-funktion jatkoksi sen jälkeen kun UICenterOnChildin Recenter-funktiota on kutsuttu ensimmäisen kerran.

```
private void Start() {
    BindDragFinished();
}

//Initializes ScrollView's dragFinish. Order is important
private void BindDragFinished() {
    _centerOnChild.Recenter();
    _scrollView.onDragFinished += OnDragFinished;
}

public void OnDragFinished() { StartCoroutine(DragFinished()); }

private IEnumerator DragFinished() {
    yield return new WaitForEndOfFrame();
    if (_centerOnChild.centeredObject != _currentPage)
        ChangePage(GetChildIndex(_centerOnChild.centeredObject) - GetChildIndex(_currentPage));
}
```

Lähdekoodi 3. CatalogControllerin lohko, jonka avulla UICenterOnChild-luokkaa käytetään.

Seuraavaksi halusin poistaa koodissa olevat duplikaatit. Alkuperäisessä LoadPaginationController-luokassa oli kaksi funktiota, joita kutsumalla katalogi voitiin keskittää edelliseen tai seuraavaan sivuun. Lähdekoodista 4 huomataan, että alkuperäisten funktioiden rakenne on lähes identtinen. Ainoana erona on se, että toisessa funktiossa vähennetään nykysivun indeksiä ja sijaintia, kun toisessa taas kasvatetaan.

```
public void IncreasePage() {
    if (CurrentPage < NumberOfPages) {
        CurrentPage += 1;
        NumberLabel.text = CurrentPage + "/" + NumberOfPages;

        ShowOrHidePagesOnIncrease();

        var springpanel = ScrollView.gameObject.GetComponent<SpringPanel>();
        springpanel.target.x -= 750;
        springpanel.enabled = true;
    }
}

public void DecreasePage() {
    if (CurrentPage > 1) {
        CurrentPage -= 1;
        NumberLabel.text = CurrentPage + "/" + NumberOfPages;

        ShowOrHidePagesOnDecrease();

        var springpanel = ScrollView.gameObject.GetComponent<SpringPanel>();
        springpanel.target.x += 750;
        springpanel.enabled = true;
    }
}
```

Lähdekoodi 4. Alkuperäisen LoadPaginationControllerin sivunvaihtofunktiot.

Refaktoroidussa versiossa, joka esitetään lähdekoodissa 5, päätoiminnallisuus on eristetty MovePage- ja ChangePage-funktioihin.

```
public void IncreasePage() { if (_itemGrid.transform.childCount > 0 &&
    GetChildIndex(_currentPage) < _itemGrid.transform.childCount - 1) MovePage(1); }

public void DecreasePage() { if (_itemGrid.transform.childCount > 0 &&
    GetChildIndex(_currentPage) > 0) MovePage(-1); }

private void MovePage(int pageDelta) {
    ChangePage(pageDelta);
    SpringPanel springpanel = _scrollView.gameObject.GetComponent<SpringPanel>();
    springpanel.target.x -= _itemsPerRow*_itemCellWidth*pageDelta;
    springpanel.enabled = true;
}

private void ChangePage(int pageDelta) {
    _currentPage = _itemGrid.transform.GetChild(GetChildIndex(_currentPage) + pageDelta).gameObject;
    _pageNumber.text = _currentPage.name + "/" + _itemGrid.transform.childCount;
    ShowOrHidePagesOnChange(pageDelta);
}
```

Lähdekoodi 5. Refaktoroidut sivunvaihtofunktiot

Eristäminen mahdollistaa IncreasePage- ja DecreasePage -funktioiden yksinkertaisen rakenteen ja poistaa duplikaatit. ChangePage-funktio vaihtaa ainoastaan sivunumerot ja asettaa oikeat sivut aktiivisiksi, ja tämä on eristetty omaksi funktiokseen siksi, että sitä voidaan käyttää OnDragFinished-funktiossa, missä NGUI hoitaa sivun sijainnin siirtämisen. MovePage-funktio kutsuu ChangePage-funktiota, mutta myös liikuttaa sivuja halutusti. On myös huomionarvoista, että alkuperäisessä luokassa esiintyvä kovakoodattu luku 750 on vaihdettu _itemsPerRow'n ja _itemCellWidthin tuloon, jolloin sivujen siirto on suhteutettu sivujen kokoon. Myös luokkamuuttuja int CurrentPage on muutettu viittaukseksi scene-tiedostossa olevaan GameObject-olioon, jolloin luokka hyödyntää scene-tiedostossa olevaa tilaa sen sijaan, että ylläpitäisi ylimääräistä tilaa, joka jäljittelee ja duplikoi jo olemassa olevaa tilaa.

Kun kaikki kolme skriptiä oli yhdistetty toisesta katalogista yhdeksi skriptiksi, oli seuraavaksi aika yleistää skripti yleiskäyttöiseksi niin, että sitä voitaisiin käyttää molemmissa katalogeissa sekä mahdollisissa tulevissa katalogeissa. Tämän toteuttamiseksi on parametrisoitava joitain asioita. Halusin, että katalogi voidaan luoda mistä tahansa skriptistä ja että se voidaan asettaa mihin tahansa scene-tiedoston hierarkiassa. Saavuttaakseni tämän tein skriptiin FactoryMethodin, joka on staattinen metodi, joka palauttaa uuden instanssin CatalogControllerista. Lähdekoodissa 6

esitetty CatalogFactory-metodi ei ole suurin ylpeydenaihe tässä refaktoroinnissa, sillä se tarvitsee jopa 6 parametria, mutta päädyin tähän ratkaisuun yrittäessäni tehdä CatalogControllerista mahdollisimman joustavan ja yleiskäyttöisen.

```
public static CatalogController CatalogFactory(Transform parent, Vector3 position,
UILabel pageNumber, UIAtlas itemAtlas, int itemsPerRow, int itemsPerColumn) {
    GameObject catalog = GameObject.Instantiate(Resources.Load("Catalog")) as GameObject;
    SetTransform(catalog.transform, parent, position);
    catalog.GetComponent<SpringPanel>().target = position;
    CatalogController controller = catalog.GetComponent<CatalogController>();
    controller._itemAtlas = itemAtlas;
    controller._itemGrid = catalog.GetComponentInChildren<UIGrid>();
    controller._pageNumber = pageNumber;
    controller._scrollView = catalog.GetComponent<UIScrollView>();
    controller._centerOnChild = catalog.GetComponentInChildren<UICenterOnChild>();
    controller._itemsPerRow = itemsPerRow;
    controller._itemsPerColumn = itemsPerColumn;
    controller._itemGrid.cellWidth = controller._itemCellWidth*itemsPerRow;
    controller._itemGrid.cellHeight = controller._itemCellHeight*itemsPerColumn;
    catalog.GetComponent<UIPanel>().clipRange = new Vector4(0, 0,
    controller._itemCellWidth*itemsPerRow, controller._itemCellHeight*itemsPerColumn);
    return controller;
}
```

Lähdekoodi 6. CatalogControllerin CatalogFactory-funktio

CatalogFactoryssa muodostetaan Catalog-prefabista aluksi GameObject-olio, jota lähdetään alustamaan parametrien mukaan. Aluksi GameObject-oliolle osoitetaan parent-GameObject-olio, jonka alle katalogi asetetaan scene-tiedoston hierarkiassa. GameObject-oliolle annetaan myös sijainti, sillä katalogin sijaintia näytöllä voidaan haluta säätää tilanteen mukaan. Factorylle annetaan myös välttämätön UILabel, joka on jokin scene-tiedostossa oleva tekstikenttä, jossa pidetään yllä sivunumeroa. Tämä UILabel oltaisiin voitu liittää myös osaksi Catalog-prefabia, mutta tämä olisi tehnyt sivunumero-kentän muokkaamisesta paljon hankalampaa. Lisäksi factorylle annetaan UIAtlas, joka kertoo, mistä kuva-atlaksesta katalogiin luotavat kuvat löytyvät. Viimeisenä, vaikkakin ehkä oleellisimpana, factorylle annetaan katalogin rivien ja sarakkeiden mitat. Näiden mittojen avulla katalogin sivut voidaan suhteuttaa oikean kokoisiksi ja factorylla voidaan tuottaa kaiken mittaisia katalogeja.

Refaktoroinnissa tehdyt muutokset tekivät CatalogControllerista yleiskäyttöisen ja mahdollistaa myös toiseen katalogiin käytettyjen skriptien korvaamisen uudella refaktoroidulla CatalogControllerilla. Lopulta alkuperäinen, n. 750 rivin pituinen, usean skriptin koodi saatiin refaktoroitua yhteen 147:n rivin pituiseen skriptiin, jota voidaan käyttää uudelleen tulevaisuudessa useammassakin kohdassa. Jos katalogien

toiminnallisuutta halutaan muuttaa jossain vaiheessa, ei muutoksia tarvitse tehdä kuuteen tai useampaan skriptiin, vaan CatalogControllerin muokkaaminen riittää. Sen lisäksi, että refaktorointi poisti huomattavan määrän duplikaattia, selkeytti koodin rakennetta ja paransi käytettävyyttä, sen avulla saatiin karsittua myös scene-tiedostosta joitakin hierarkisia rakenteita niin, että rakenne muodostetaan ajonaikaisesti. Tämä selkeyttää scene-tiedoston hierarkiaa ja parhaassa tapauksessa vähentää myös muistin kuormitusta, sillä tarpeettomia GameObject-olioita ei pidetä muistissa koko sovelluksen elinkaaren ajan, vaan niitä luodaan ajonaikaisesti tarpeen mukaan.

5 Suunnittelumallit ja antimallit

Tässä luvussa käsittelen ohjelmistoalalla suuressa arvossa pidettyjä suunnittelumalleja (eng. design patterns) ja niiden vastakohtia antimalleja (eng. anti patterns). Aihetta käsitellään melko pintapuolisesti ja kriittisesti, sillä Intellellä olemme kokeneet, että suunnittelumalleista saadut hyödyt eivät ole kovin suuret Intellen tuotekehitysprojektissa.

Suunnittelumallit ovat 1980-luvun lopulla kehitetty konsepti, joka on alkujaan rakennusarkkitehtuurista. Suunnittelumalleilla on tarkoitus tarjota ratkaisuja joihinkin yleisiin ohjelmoinnissa toistuviin ongelmiin. Suunnittelumallit perustuvat Erich Gamman, Richard Helmin, Ralph Johnsonin ja John Vlissidesin kirjaan “Design Patterns: Elements of Reusable Object-Oriented Software” vuodelta 1995.

Suunnittelumalleja kohtaan on kuitenkin esitetty myös jonkin verran kritiikkiä. Kritiikin mukaan nykyisillä ohjelmointikielillä voidaan ratkaista suurin osa suunnittelumallien ratkaisemista ongelmista [Norvig 1996]. Nykyaikaiset ja kehittyneet ohjelmointikielet, kuten Java 8 ja C# 5, sisältävät paljon funktionaalisen ohjelmoinnin ominaisuuksia, joilla voidaan korvata useita perinteisen olio-ohjelmoinnin suunnittelumalleja. Jokaisesta 23:sta alkuperäisestä suunnittelumallista on tehty korvaava toteutus funktionaalisten ohjelmointikielien ominaisuuksilla. [Yang 2010] Osa korvaavista toteutuksista on huomattavasti helpompia ja kätevämpiä käyttää, mutta jotkut suunnittelumalleista on vaikeammin korvattavissa. Lisäksi suunnittelumallien liiallinen käyttö voi johtaa sovellukseen ylimääräiseen monimutkaisuuteen, kuten esimerkiksi singletonia käsittelevässä luvussa käy ilmi.

Intellellä me käytämme suunnittelumalleja melko vähän sellaisena, kuin ne oppikirjassa esitetään. Esimerkiksi suunnittelumalleista tuttua mantraa “ohjelmoi rajapintaa vasten, älä toteutusta vasten” ei ole noudatettu sanatarkasti. Sen sijaan, että ohjelmoisimme ainoastaan rajapintoja vasten, ohjelmoimme itse asiassa useimmiten toteutusta vasten, mutta vain jos samankaltaisia toteutuksia on vain yksi. Jos sovellukseen täytyy ohjelmoida jokin luokka, joka on samankaltainen kuin jokin toinen luokka, on tietysti aihetta harkita rajapinnan käyttöä.

Suunnittelumallien merkittävä hyöty on kuitenkin se, että ne luovat terminologiaa erilaisille ohjelmistoissa oleville abstrakteille rakenteille. Suunnittelumallit luovat ohjelmoijille yhteistä kielipohjaa, jotta heidän ei tarvitse rivi riviltä esitellä sovelluksen toimintaa tai piirtää luokkakaavioita jonkin yksinkertaisen asian selittämiseksi. Suunnittelumallit ovat siis mielestäni hyvä apuväline ohjelmointityössä, mutta kehittäjän pitää myös varoa käyttämästä niitä väärin.

Suunnittelumallien ohella myös antimallien ymmärtäminen auttaa ohjelmistokehityksessä. Antimallit ovat varoittavia esimerkkejä toimintatavoista ja ohjelmistokäytännöistä, joita ei pidä noudattaa. Vaikka suunnittelumallien noudattaminen ei aina onnistu tai ole edes tarpeen, antimallien noudattamatta jättäminen on aina hyväksi. Antimalleihin on kerätty etenkin sellaisia huonoja toimintamalleja, joihin ohjelmistokehittäjät usein saattavat sortua, joten ne ovat usein helposti ymmärrettäviäkin. Koen itse antimalleista olevan jopa enemmän hyötyä kuin suunnittelumalleista, sillä ilman suunnittelumallejakin on mahdollista ohjelmoida loistava sovellus hyvälle arkkitehtuuriselle pohjalle, mutta antimalleja noudattaen arkkitehtuurissa on parannettavaa.

Esimerkkejä antimalleista on mm. God Object, Magic Numbers, Spaghetti Code ja Analysis Paralysis. God Objectilla tarkoitetaan luokkaa, joka hallitsee aivan liian monia asioita eikä toteuta SRP:tä. Magic Numbersilla tarkoitetaan jossakin algoritmissa esiintyvää numeroa, jota ei selitetä mitenkään, mutta taianomaisesti se saa algoritmin toimimaan. Spaghetti Code on kaikille koodareille tuttu käsite, jolla tarkoitetaan yleisesti sekavaa koodia, jossa hypitään holtittomasti metodien ja luokkien välillä. Loistava tapa kokata spagettikoodia on käyttää goto-lausekkeita. Analysis Paralysis on taas ohjelmointiprosessissa esiintyvä ilmiö, missä kehittäjä koettaa ratkaista jotakin ongelmaa mahdollisimman hienolla tavalla, mutta ongelman ollessa liian

monimutkainen kehittäjä jumittuu analysoimaan ongelmaa eikä saa minkäänlaista ratkaisua tehtyä.

6 Kehitystä tukevat järjestelmät

Ohjelmoijat ovat kehittäneet useita työkaluja, jotka helpottavat heidän jokapäiväistä työskentelyään. Työskentelyyn kuuluu monia proseduraalisia tehtäviä, jotka voidaan automatisoida, ja automatisointiin tulisi pyrkiä aina, kun se on mahdollista. Tämä vapauttaa ohjelmoijien kallisarvoista aikaa, jonka he voivat käyttää muiden tehtävien hoitamiseen. Automatisointi pienentää myös inhimillisten virheiden syntymisen mahdollisuutta, joten pitkälle automatisoidussa ympäristössä voidaan myös tehdä virheettömämpiä sovelluksia. Hyvillä työkaluilla on mahdollista tehdä järjestelmän ylläpidosta ja jatkokehityksestä huomattavasti kevyempää, joten niiden tutkimiselle ja käyttöönotolle on syytä antaa arvoa.

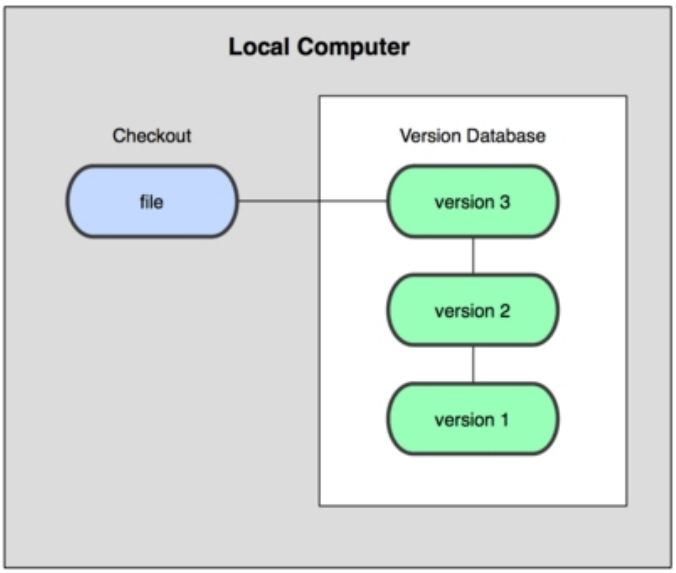
Tässä luvussa käsitellään kriittisimpiä järjestelmiä, jotka nopeuttavat ohjelmoijan jokapäiväistä työtä sekä tekevät koodin laadunvalvonnasta tehokkaampaa. Nämä järjestelmät ovat myös välttämättömiä työkaluja tehokkaan tiimityöskentelyn ja sovelluksen jatkokehityksen kannalta.

6.1 Versionhallintajärjestelmä

Versionhallintajärjestelmää tarvitaan jokaisessa ohjelmistoprojektissa, johon osallistuu useita ohjelmistokehittäjiä tai jotka ovat laajuudeltaan isompia, kuin mitä yksi kehittäjä voi muutamassa päivässä tehdä. Tämä käsittää siis jokseenkin kaikki ammattimaiset ohjelmistoprojektit. Versionhallintajärjestelmä tallentaa käytännössä kaikki muutokset, joita projektiin on tehty siitä lähtien, kun projekti on viety versionhallintaan [About Version Control]. Tämä tarkoittaa, että kaikki vanhatkin versiot kuvista, 3D-malleista, skripteistä ja muista tiedostoista pysyvät tallessa, ja ne voidaan palauttaa projektiin, mikäli huomataan esimerkiksi, että uusi versio on viallinen eikä vikaa voida korjata. Tiedostojen versioiden palauttamisen lisäksi versionhallinnasta on myös se selkeä hyöty, että useampikin kehittäjä voi työskennellä saman projektin ja samojen tiedostojen parissa. Jos tiedostoja tallennetaan versionhallintajärjestelmään tarpeeksi usein, tietävät muutkin kehittäjät aina, miltä uusin versio mistäkin tiedostosta näyttää,

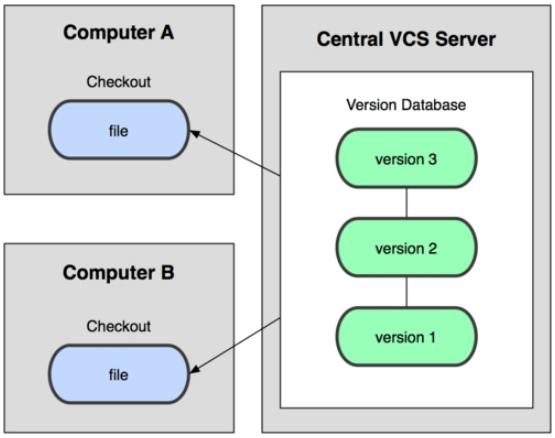
eivätkä he tee päällekkäisiä muutoksia projektiin. Versionhallintajärjestelmien mukana saa yleensä työkalut mm. kooditiedostojen yhdistämiseen. Niiden avulla eri kehittäjien samaan tiedostoon tehdyt muutokset voidaan yhdistää, eikä kenenkään työ mene hukkaan.

Versionhallintajärjestelmät voidaan jakaa kolmeen luokkaan. On paikallisia versionhallintajärjestelmiä, joiden toimintaperiaate esitetään kuvassa 6.



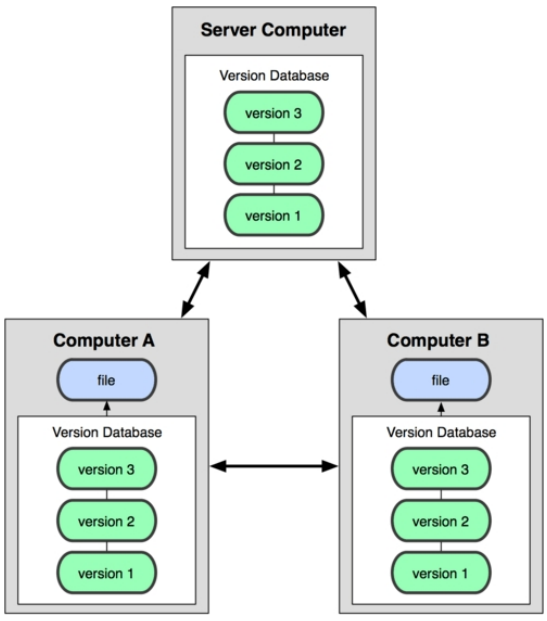
Kuva 5. Paikallisen versionhallintajärjestelmän toimintaperiaate.

Paikallisessa versionhallintajärjestelmässä kaikki versiot tiedostoista säilytetään tietokoneen paikallisessa muistissa. Tämä soveltuu hyvin yhden ohjelmoijan projekteihin, sillä muutoksia ei ole tarvetta jakaa koneen ulkopuolelle. Kuvassa 7 esitetty keskitetty versionhallintajärjestelmä on kehitetty, kun ohjelmoijien on täytynyt jakaa tiedostoja keskenään ja kehittää samanaikaisesti näitä tiedostoja.



Kuva 6. Keskitetyn versionhallintajärjestelmän toimintaperiaate.

Keskitetyssä versionhallintajärjestelmässä tiedostot sijaitsevat keskitetyllä palvelimella, josta tiedostoja voidaan hakea muokattavaksi. Näin kaikkien kehittäjien versiot järjestelmästä pysyvät synkronoituna uusimpaan versioon, ja päällekkäisten tai ristiriitaisten muutosten aiheuttamat ongelmat saadaan minimoitua. Kolmas ja kehittynein muoto versionhallintajärjestelmästä on kuvassa 8 esitetty hajautettu versionhallintajärjestelmä.



Kuva 7. Hajautetun versionhallintajärjestelmän toimintaperiaate.

Hajautetussa versionhallintajärjestelmässä koko versiotietokanta on tallennettu jokaiselle koneelle, johon versionhallinnassa oleva projekti on haettu. Tämä tarkoittaa

sitä, että jokaisen kehittäjän koneella on koko versiohistoria sellaisena, kuin se on palvelimellakin, siinä missä keskitetyssä versionhallintajärjestelmässä versiohistoria sijaitsee ainoastaan palvelimella. Tästä on se etu, että jos esimerkiksi keskuspalvelin kaatuu, niin kehittäjät voivat edelleen tallentaa tekemiään muutoksia paikalliseen versionhallintaansa eikä palvelinta välttämättä tarvita muutoksien jakamiseksi muille. Keskitetyssä versionhallintajärjestelmässä versiotietokannan palauttaminen on varmuuskopioiden varassa, mikäli tietokanta tuhoutuu palvelimelta, eivätkä kehittäjät voi käyttää versionhallintaa niin kauan, kuin palvelin on alhaalla.

Käytimme Intellellä aluksi hajautettua Git-versionhallintajärjestelmää, joka on hyvin suosittu ohjelmistokehittäjien parissa. Muutaman kuukauden tuotekehityksen aikana versionhallintaan tallennettiin kuitenkin jatkuvasti suuria binääritiedostoja, mikä aiheutti ongelman Gitin kanssa. Pienen tutkimustyön tuloksena selvisi, että Git ei käsittele binääritiedostoja tehokkaasti, eikä ongelmaan ole vielä kehitetty ratkaisua. Meillä ongelma näkyi pitkinä vasteaikoina haettaessa Gitistä uusia tai muuttuneita binääritiedostoja. Koko versiohistorian hakemiseen kului lopulta useita tunteja, jolloin päätimme tutkia muita vaihtoehtoja versionhallintaan.

Toisena ehdokkaana oli keskitetty versionhallintajärjestelmä Perforce. Päädyimme ottamaan Perforcen testattavaksi, sillä löysimme Juicebox Games Inc:n artikkelin, missä he esittelivät Unity3D:n ympärille rakentamaansa kehitysympäristöään, ja tämä vaikutti erittäin uskottavalta ja toimivalta toteutukselta. Juicebox Gamesin kehitysympäristön versionhallintajärjestelmänä toimii artikkelin mukaan Perforce, mikä sai meidät kiinnostumaan Perforcesta. Tutkimustyön tuloksena selvisi, että Perforce ratkaisee isojen binääritiedostojen aiheuttaman ongelman, sillä Perforce varastoi ja jakaa isojakin tiedostoja hyvin tehokkaasti [Comparison: Perforce and Git 2012: 13]. Otimme siis Perforcen käyttöön ja huomasimme koko versiokannan lataamiseen kuluvan pari minuuttia, siinä missä Gitillä siihen kului pari tuntia. Tämä oli huima askel kohti sulavampaa ja tehokkaampaa ohjelmistokehitystä.

Käytettyämme Perforcea muutaman viikon ajan aloimme huomata joitakin ongelmia sen käytössä. Työnkulku Perforcella ei aina tuntunut loogiselta Gitin jälkeen, ja Perforceen oli keksitty aivan uusia termejä monien vanhemmista versionhallintajärjestelmistä tuttujen termien tilalle. P4Ignore-ominaisuus, jolla rajataan tiedostoja ja kansioita pois versionhallinasta, oli myös toiminnaltaan epävakaata, sillä sen pitäisi estää käyttäjän määrittelemien tiedostojen viemisen

versionhallintajärjestelmään, mutta nämä määritykset täytyy tehdä kunkin käyttäjän koneella, eivätkä määritykset kaikissa tilanteissa säilyneet. Tästä syystä mm. jotkin versionhallintaan kuulumattomat asetustiedostot joutuivat sinne, mikä sekoitti monien käyttäjien henkilökohtaiset asetukset mm. Visual Studiossa ja Unityssa. Perforcessa oli muitakin pieniä käyttömukavuuteen liittyviä ongelmakohtia, joten päätimme evaluoida vielä muita vaihtoehtoja.

Kolmantena versionhallintajärjestelmänä otimme käyttöön Plastic SCM:n, joka Perforcen tavoin on myös keskitetty versionhallintajärjestelmä. Plasticia käytetään samaan tyyliin kuin Perforcea, mutta ilman Perforcen edellä mainittuja ongelmakohtia. Plastic on yhtä tehokas myös isojen binääritiedostojen siirrossa kuin Perforce, ja sen graafinen käyttöliittymä on helppokäyttöinen ja toimiva. Plasticia suunnitellessa on otettu vahvasti huomioon pelinkehittäjien tarpeet, minkä takia se sopii erinomaisesti myös Unity3D-projektien versionhallintajärjestelmäksi.

6.2 Käännösautomaatio

Unity3D on niin sanottu multiplatform-kehitysympäristö, eli sillä kehitettyjä järjestelmiä voidaan kääntää usealle eri alustalle. Näihin alustoihin lukeutuvat mm. iOS, Android, Windows 8, OSX, web-selaimet ja monet muut ympäristöt. Unity3D:ssä on myös oma Editor-alustansa, joka käytännössä kääntää ohjelmakoodin aina, kun työskennellään Unity-ympäristössä ja koodiin on tullut muutoksia. Valitettavasti eri alustoilla on kuitenkin joitain eroavaisuuksia ja rajoitteita. Tämän takia on äärimmäisen tärkeää, että nämä mahdolliset ongelmat tulevat vastaan siinä vaiheessa, kun ongelmia aiheuttavaa ohjelmakoodia kirjoitetaan. Projektin alkuvaiheissa sattui tilanteita, joissa jokin ominaisuus on tehty ja testattu useita viikkoja sitten, mutta testit on tehty ainoastaan Unityn Editorissa, missä ne ovat toimineet hienosti. Kun ohjelmaa on koetettu sitten kääntää iPadille päivää ennen deadlinea, on huomattu, että ohjelma ei käänny lainkaan, eikä virheilmoituksista edes ilmene selkeästi, missä vika on. Virheistä oppineena olemme ymmärtäneet, että käännökset pitäisi tehdä kaikille halutuille laitteille mahdollisimman usein, jotta mahdolliset virheet ilmenisivät aikaisessa vaiheessa.

Tätä tarkoitusta palvelemaan on kehitetty käännösautomaatio-ohjelmistoja, jotka tekevät käännöksen automaattisesti käyttäjän määrittelemien sääntöjen mukaisesti. [Per Asset Versioning with Unity Asset Bundles 2013] Juicebox Games käyttää

käännösautomaatioon Jenkinsiä, joka on myös Intellen kokeneiden ohjelmistoarkkitehtien mukaan erinomainen työkalu automaattisten käännösten tekemiseen. Lisäksi Jenkins tarjoaa myös mahdollisuuden ajaa halutut testit aina käännösten yhteydessä, mikä tukee erinomaisesti hyvän ylläpidettävyyden päämäärää. [What is Jenkins? 2013] Testejä ajamalla voidaan käännösaikaisten virheiden lisäksi jäljittää myös sovelluksen ajonaikaisia toimintahäiriöitä.

Jenkins tutkii määrätyin väliajoin versionhallintaa, ja mikäli versionhallintaan on tullut uusia versioita joistakin tiedostoista, hakee Jenkins nämä tiedostot ja suorittaa komentorivikäskyillä käännöksen halutuille alustoille. Mikäli käännöksessä tulee virhe, se näkyy lokista, ja Jenkins ilmoittaa virheestä määritetyllä tavalla. Intellellä on asetettu kaikkien näkyville monitori, jossa on merkkivalo jokaista Jenkinsin käännöstä varten. Jos viimeisin käännös on onnistunut, palaa tämä valo vihreänä, mutta jos käännöksessä on tullut jokin virhe, muuttuu valo punaiseksi. Tällöin tiedetään, että versionhallinnassa on versio, joka ei käänny kaikille alustoille ja ongelmaan voidaan paneutua heti. [Cockburn 2000: 76.] Tällaista monitoria kutsutaan Information Radiatoriksi ja sitä käytetään ketterissä menetelmissä tärkeimpien tietojen esittämiseen. Jenkins ajaa myös projektiin tehdyt testit jokaisella alustalla, ja samalla periaatteella yhdenkin testin epäonnistuessa valo vaihtuu punaiseksi. Periaatteena on, että kaikkien merkkivalojen pitäisi aina olla vihreinä.

6.3 Testikehys

Kattava testaus on Unity3D:llä haastavaa. Tämä johtuu siitä, että Unityn sovelluskehykseen kuuluu niin paljon reaaliaikaisesti muuttuvia osia, että on mahdotonta ottaa huomioon kaikkia muuttuvia tekijöitä. Unity pyörittää taustalla reaaliaikaista grafiikka- ja fysiikkamoottoria, hallinnoi käyttäjän syötteitä ja sovelluksessa pyöritettäviä ääniä. Lisäksi kaikki tapahtuvat 3D-maailmassa, joka kirjaimellisesti tuo uuden ulottuvuuden verrattuna perinteisiin 2D-sovelluksiin.

Reaaliaikaisuus on testien kannalta haastavaa sen takia, että testien pitäisi pyöriä nopeasti. Jos pelaaja laitetaan testissä kävelemään pelikenttää tunniksi ympäriinsä ja tarkastellaan, tippuuko hän seinien läpi pois alueelta, ei testi todennäköisesti kata kaikkia mahdollisia tilanteita, eikä se ole tarpeeksi nopea, jotta sitä voitaisiin ajaa useita kertoja päivässä kaikkien muiden testien kanssa. Monet niistä asioista, joita haluttaisiin

testata, osoittautuivat lähes mahdottomiksi tai ainakin niin haastaviksi testata, että kehitystiimin tuottavuus romahtaisi tällaisia testejä suunniteltaessa. Tämä ei kuitenkaan ole pätevä syy luopua testauksesta kokonaan, mutta varsinaisia kapseloituja yksikkötestejä ei ole hyödyllistä tehdä, ellei kirjoitettu ohjelmakoodi ole eristetty Unityn ohjelmakehyksestä. Sen sijaan tilanteesta riippuen voi olla hyvinkin hyödyllistä tehdä integraatiotestauksia sellaisille ominaisuuksille, joiden käyttö aiheuttaa tilan muutosta jossakin tunnetussa moduulissa tai oliossa. Näin voidaan testata esimerkiksi, että pelaajan kävellessä ovea päin ovi avautuu.

Testejä varten Unitylle on tehty oma testikehyksensä uUnit, joka on alun perin jUnit-kehyksestä Unitylle sopivaksi käännetty versio. uUnitilla järjestelmän tila voidaan alustaa jokaista testiä varten sellaiseen tilaan, mitä kussakin testissä jäljitellään, ja testin lopussa voidaan tutkia kehittäjän määräämiä muuttujia. Mikäli muuttujat eivät ole odotettuja, testi epäonnistuu, ja epäonnistumisesta kirjoitetaan lokiin tieto. Testejä voi näin käydä läpi testatessa uutta ohjelmakoodia. Jos koodi ei toimi halutulla tavalla, nähdään se testilokista testien ajettua läpi sen sijaan, että järjestelmä pitäisi käynnistää alusta ja testata ominaisuus manuaalisesti. Ehkä jopa suurempi etu testeistä on se, että kattavalla testipedillä havaitaan heti, jos kehittäjän tekemät muutokset ovat aikaansaaneet odottamattomia virheitä muualla järjestelmässä. Näitä virheitä ei välttämättä löydettäisi manuaalisellakaan testauksella, vaan vika saattaisi jäädä piiloon, kunnes tuote on julkaistu ja käyttäjät raportoivat virheestä. uUnit antaa testitulokset jUnitista tutussa muodossa, joten myös Jenkins osaa tulkita testituloksia. Tämä mahdollistaa testien läpikäymisen käännösautomaation ohessa, jolloin kehittäjät pakotetaan reagoimaan aiheuttamiinsa virheisiin.

Lähdekoodissa 7 on esimerkki uUnitilla tehdystä testistä. Esimerkistä näkee, millaista syntaksia testit käyttävät. Testillä testataan, että AssetBundleLoader todella lataa joitakin objekteja.

```
[TestFixture]
public class LoadAssetBundleTest {

    [Test]
    public IEnumerator CanLoadSomeItems() {
        AssetBundleLoader loader = AssetBundleLoader.Instance;
        float timeout = 50f;
        float timer = 0f;
        int loadedCount = 0;
        Events.Instance.AddListener<ItemLoaded>((loadedEvent) => {
            loadedCount = loadedEvent.FurnitureItems.Count;
        });
        while (!(loadedCount > 0 ) && (timer < timeout)) {
            yield return new WaitForSeconds(1);
            timer += 1f;
        }
        Assert.True(loadedCount > 0);
    }
}
```

Lähdekoodi 7. uUnitilla tehty testi.

7 Työskentely Unity3D:llä

Projektin pääasiallinen kehitystyökalu on Unity3D, johon kootaan kaikki 3D-mallit, tekstuurit, kooditiedostot ja äänitiedostot ja näistä rakennetaan halutunlainen järjestelmä. On siis tärkeää, että työskentely Unityllä on mahdollisimman tehokasta. Unityllä työskenneltäessä kehitettävää sovellusta tulisi pyrkiä optimoimaan mahdollisimman kevyeksi niin, että monimutkaistakin Unityllä tehtyä järjestelmää voisi käyttää sujuvasti vanhemmilla ja halvemmilla tietokoneilla tai vähemmän tehokkailla mobiililaitteilla. Näin tuotteen potentiaalinen käyttäjäkunta on mahdollisimman laaja, vaikka laadussa ei olisikaan tingitty. Tätä varten täytyy määrittää joitakin sääntöjä kehittäjille, sillä Unityn ohjelmakehys ei suoranaisesti ohjaa optimoidun ohjelmiston kehittämiseen. Tässä luvussa käydään läpi joitakin arkkitehtuurisia päätöksiä, joita Intellellä on tehty modulaarisuuden, tehokkuuden ja joustavan kehityksen parantamiseksi. Nämä kaikki asiat yhdessä tekevät sovelluksesta jatkokehityskelpoisemman ja ylläpidettävämmän.

7.1 Scene-tiedostojen käyttö

Scene on Unityn oma tiedostotyyppi. Scene-tiedosto voi sisältää mitä hyvänsä objekteja, mitä pelissä tai järjestelmässä halutaan käyttää. Monien tutoriaalien, opetusvideoiden ja dokumenttien perusteella [Creating Scenes 2013] scene-tiedostoja käytetään usein yksilöllisen kentän tekemiseen. Scene-tiedoston hierarkiarakenteeseen voisi näin kuulua kaikki kentässä olevat talot, puut, pelaajat, huonekalut ja kaikki muut kentän sisältämät objektit. Vaihtoehtoisesti scene-tiedosto voi sisältää valikon, joka avaa pelaajan valintojen jälkeen seuraavan scene-tiedoston, esimerkiksi ensimmäisen pelikentän.

Intellen tuotteessa oli alun perin kolme scene-tiedostoa. Nämä olivat nimeltään “Login”, “MainMenu” ja “Main”, ja kuten nimestä voidaan päätellä, ensimmäisessä scene-tiedostossa käyttäjä kirjautui sisään, seuraavassa scene-tiedostossa avautuu päävalikko, joka ohjaa käyttäjän pää-scene-tiedostoon, jossa järjestelmää pääsääntöisesti käytetään. Tämä järjestely toimi siinä mielessä hyvin, että jokainen scene-tiedosto sisälsi vain scene-tiedoston kontekstissa tarvittavat objektit, eikä muiden scene-tiedostojen muutoksista tai sisällöstä tarvinnut välittää. Hyöty useasta scene-tiedostosta muuttui kuitenkin kyseenalaiseksi, kun joidenkin objektien täytyikin säilyä scene-tiedostojen välillä. Esimerkiksi kirjautumisessa saatavan istuntotunnuksen säilyttävän web-rajapintaolion täytyi säilyä Login-scene-tiedostosta aina Main-scene-tiedostoon asti, sillä istuntotunnuksia käytetään Main-scene-tiedostossa tehtävissä HTTP-kutsuissa. Vastaavasti MainMenu-scene-tiedostossa avattavien tallenteiden avaimia täytyi taltioida rekisteriin, josta ne voitiin Main-scene-tiedostossa hakea ja tämän jälkeen avata tallenteita avaimien perusteella. Koska scene-tiedostot ovat loppujen lopuksi vain säiliöitä olioille ja komponenteille, tulimme lopulta siihen tulokseen, että yhdistämme kaikkien scene-tiedostojen objektit yhteen scene-tiedostoon, johon haluttuja olioita voidaan ladata dynaamisesti järjestelmän ollessa käynnissä. Tämä ratkaisee sen ongelman, että tietoa tai objekteja tarvitsisi kuljettaa scene-tiedostojen välillä. Ratkaisu myös nopeuttaa järjestelmän toimintaa, sillä uuden scene-tiedoston lataamisessa on jonkin verran viivettä.

7.2 Prefabien käyttö

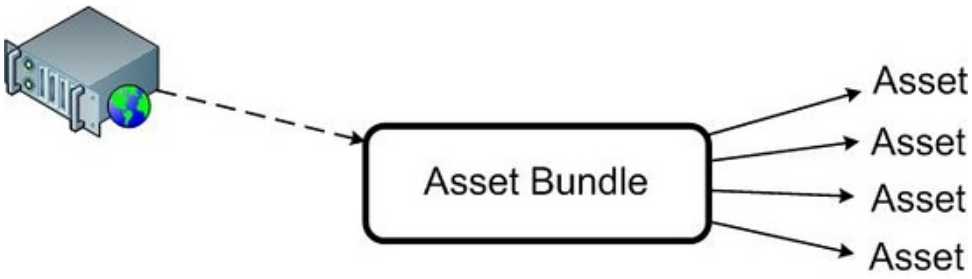
Kun scene-tiedostoa kehitetään, sinne voidaan luoda uniikkeja objekteja, jotka ovat olemassa vain kyseisen scene-tiedoston hierarkiassa. Objekteja voi kyllä kopioida ja siirrellä, mutta jos näihin kaikkiin kopioihin halutaan tehdä jokin muutos, täytyy muutos tehdä jokaiseen yksitellen. Scene-tiedostossa olevissa uniikeissa objekteissa on myös toinen hankala puoli. Kaikki tällaisiin objekteihin tehdyt muutokset tallennetaan siihen scene-tiedostoon, joka nämä objektit sisältää, ja scene-tiedosto tallennetaan yhtenä binääritiedostona tietokoneelle. Versionhallintajärjestelmissä on hyvät työkalut kooditiedostojen yhdistämistä varten, mutta binääritiedostoja versionhallintajärjestelmät eivät osaa yhdistää. Tämä johtaa siihen, että jos kaksi kehittäjää tekee samanaikaisesti muutoksia samassa scene-tiedostossa oleviin objekteihin, voidaan vain toisen tekemät muutokset tallentaa versionhallintajärjestelmään.

Näihin ongelmiin ratkaisuna Unity tarjoaa prefab-tiedostotyypin. Prefab voi olla mistä tahansa scene-tiedostossa olevasta objektista tehty tiedosto, joka sisältää objektin kaikki tiedot, kuten mitä materiaaleja ja tekstuureja objekti käyttää, mitä skriptejä objektiin on liitetty, mikä objektin sijainti on 3D-koordinaatistossa jne. [Prefabs 2013] Kun prefabia halutaan käyttää scene-tiedostossa, luodaan siitä scene-tiedostoon instanssi, joka sisältää linkin prefabiin. Kun prefabia muutetaan, muuttuvat prefabista luodut instanssit samalla, eikä jokaista instanssia tarvitse muuttaa erikseen. Tämä pienentää virheiden syntymisen riskiä, ja syntyneet virheet löytyvät paljon helpommin, kun ei tarvitse tutkia jokaista objektia erikseen. Prefabin ja instanssin välisen linkin voi halutessa myös poistaa, jolloin instansseista tulee itsenäisiä objekteja.

Toisena etuna prefabeista on se, että ne ovat itsenäisiä binääritiedostoja. Scene-tiedostoa ei tarvitse muuttaa vaikka prefabia muutettaisiin, joten tämä mahdollistaa scene-tiedoston sisältöön tapahtuvat samanaikaiset muutokset ilman, että itse scene-tiedostoa tarvitsee muuttaa. Kun objektit on pilkottu tarpeeksi pieniin palasiin ja scene-tiedosto muodostuu lukuisista pienistä prefabeista, on melko epätodennäköistä, että kahden kehittäjän täytyisi tehdä muutoksia samaan prefabiin samanaikaisesti [50 Tips for Working with Unity 2012].

7.3 AssetBundlejen käyttö

AssetBundlet ovat Unityssä pakattuja kokoelmia tiedostoista. [AssetBundles 2013] Tiedostot voivat olla kuvia, tekstitiedostoja, 3D-malleja tai vaikka Unityssä tehtyjä prefabeja. AssetBundleja voidaan ladata internetistä ajonaikaisesti ja niissä olevia tiedostoja voidaan käyttää sovelluksessa samalla tavoin kuin käännöksen mukana sisällytettyjä tiedostoja. Kuvassa 9 havainnollistetaan AssetBundlejen toimintaperiaatetta. Asiakasohjelma voi ladata palvelimelta yhden datapaketin, joka voi sisältää useita eri Unityn käyttämiä tiedostoja eli asseteja.

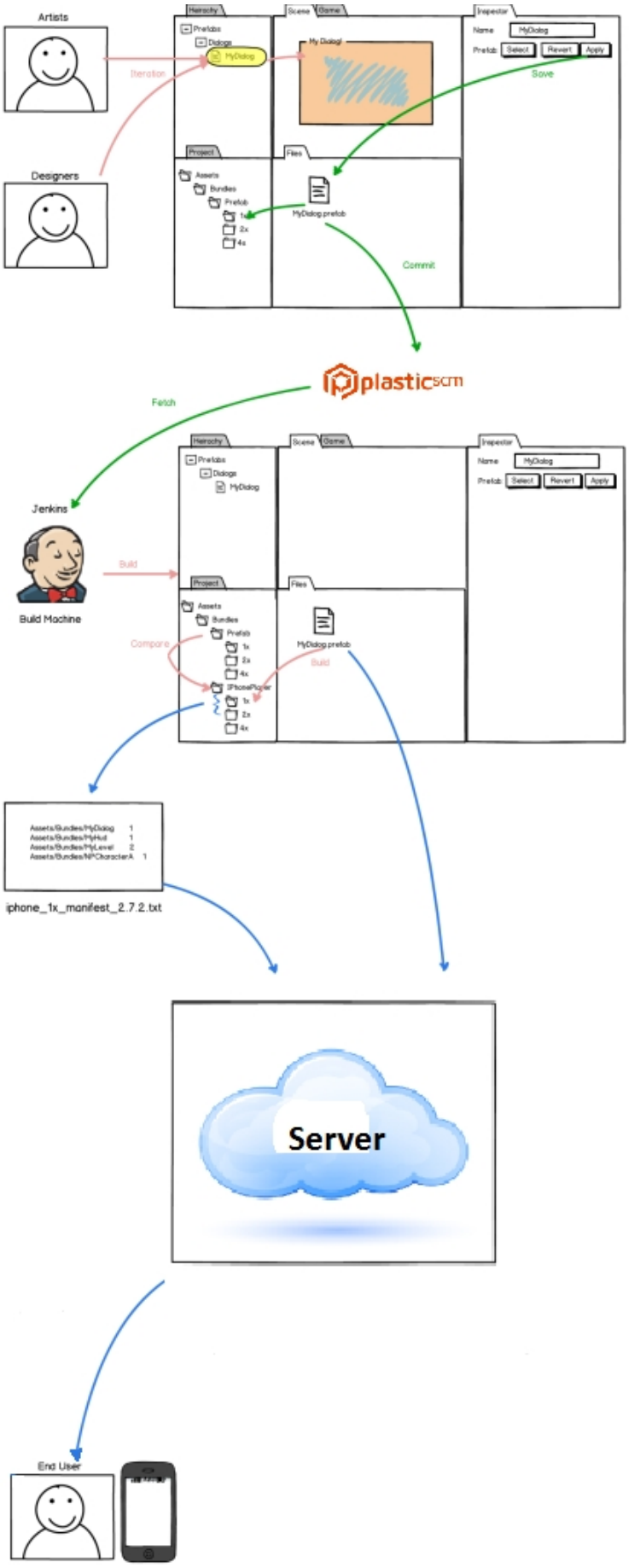


Kuva 8. AssetBundlen toimintaperiaate.

Unity tallentaa myös AssetBundleja tiettyihin kokorajoitteisiin asti päätelaitteen muistiin, jotta bundleja ei tarvitse ladata internetistä joka kerta, kun niitä halutaan käyttää. Tämä ajatusmalli sopii meidän tuotteeseemme erinomaisesti, sillä AssetBundlet mahdollistavat sen, että pidämme sovelluksen sisältöä palvelimella. Palvelimella sisältöä voidaan korjata ja muuttaa, tiedostoista voidaan tehdä kehittyneempiä versioita ja sovellus lataa uudet parannellut tiedostot ajonaikaisesti ilman, että käyttäjä huomaa mitään. Jos kaikki sisältö koetettaisiin sisällyttää käännökseen valmiiksi, jouduttaisiin koko sovellus päivittämään kaikille päätelaitteille, mikäli halutaan muuttaa tai lisätä sisältöä sovellukseen. Lisäksi tiedostojen normaali lataaminen muistiin tapahtuu synkronisesti, eli sovellus pysähtyy odottamaan tiedoston latausta. AssetBundlejen laataaminen tapahtuu asynkronisesti, joten isojakin tiedostoja voidaan ladata muistiin ilman, että sovelluksen suoritus keskeytyy [Per Asset Versioning with Unity Asset Bundles 2013].

AssetBundlejen kääntäminen ja hallinta voi kuitenkin olla työlästä. Tätä varten Intellellä on päätetty toteuttaa AssetBundlejen hallinta samaan tapaan kuin Juicebox Gamesilla. Tämä toteutus toimii erinomaisesti Intellen prefab-käytäntöjen ja Jenkins-

käännösautomaation kanssa. Käytännössä projektin hakemistorakenne sisältää yhden kansion, jonne graafikot ja ohjelmoijat tallentavat prefabeja, jotka sisältävät uutta sisältöä sovellukseen. Nämä prefabit on tarkoitettu käännettäväksi AssetBundleiksi, joten kun kansioon lisätty prefab viedään Plastic SCM -versionhallintaan, ajaa Jenkins Unityssa automaattisesti skriptin, joka tekee päivitetyistä tai uusista prefabeista uudet AssetBundlet. AssetBundlet päivitetään palvelimelle, mistä ne ovat ladattavissa asiakasohjelmalle. Käännetyistä AssetBundleista pidetään myös yllä listaa, joka sijaitsee määritetyssä osoitteessa palvelimella, josta lista haetaan aina sovelluksen käynnistyessä. Näin kehittäjien ei tarvitse tehdä muuta kuin päivittää versionhallinnassa olevia prefabeja, ja automaatio hoitaa loput toimenpiteet niin, että päivitetyt tiedostot latautuvat lopulta käyttäjien päätelaitteille. Tämä käytäntö mahdollistaa myös erilaisen sisällön eri käyttäjille. Listaus voi sisältää tiedon siitä, mille käyttäjille mikäkin AssetBundle on saatavilla. Tällä tavoin käyttäjien ryhmittelyllä ja personoinnilla käyttäjille voidaan tuottaa kohdennettua sisältöä. Tätä työnkulkua on hahmoteltu kuvassa 10.


Artists
Designers
Iteration
Save
Commit
plasticscm
Fetch
Jenkins
Build
Build Machine
Compare
iphone_1x_manifest_2.7.2.txt
Server
End User

Kuva 9. AssetBundlejen elinkaari.

Lähdekoodissa 8 on esitetty AssetBundleja kääntävän AssetBundleBuilder-skriptin olennaisimmat funktiot. AssetBundleBuilder on se skripti, jonka Jenkins ajaa muutoksien saavuttua versionhallintajärjestelmään.

```
private static BundlePrefabNode BuildBundle(GameObject nodeObject, params string[] flags) {
    var rootNode = MakeNode(nodeObject, flags);
    var assets = AddDescendantAssetsToList(nodeObject, new List<string>());
    foreach (var asset in assets)
        rootNode.BundleDependencies.Add(asset);
    BuildBundleWithDependencies(rootNode, BuildOptions);
    return rootNode;
}

private static BundlePrefabNode MakeNode(Object nodeObject, params string[] flags) {
    var node = new BundlePrefabNode {
        GUID = GetAssetGuid(nodeObject),
        BundleDependencies = new List<string>(),
        Name = nodeObject.name,
        Flags = flags.ToList()
    };
    node.AssetBundle = node.GUID;
    return node;
}
private static IEnumerable<string> AddDescendantAssetsToList(GameObject nodeGameObject, List<string> assets) {
    GetChildComponents(nodeGameObject, assets);
    var meshFilter = nodeGameObject.GetComponent<MeshFilter>();
    if (meshFilter != null && meshFilter.sharedMesh != null) {
        string meshGuid = GetAssetGuid(meshFilter.sharedMesh);
        if (!assets.Exists(x => x == meshGuid) && !String.IsNullOrEmpty(meshGuid)) assets.Add(meshGuid);
    }
    var meshRenderer = nodeGameObject.GetComponent<MeshRenderer>();
    if (meshRenderer != null && meshRenderer.sharedMaterials != null) {
        foreach (Material material in meshRenderer.sharedMaterials) {
            string materialGuid = GetAssetGuid(material);
            if (!assets.Exists(x => x == materialGuid)) {
                assets.Add(materialGuid);
                IEnumerable<Object> textures = EditorUtility.CollectDependencies(new Object[] {material})
                    .OfType<Texture2D>().Cast<Object>();
                foreach (Object texture in textures)
                    if (!assets.Exists(x => x == GetAssetGuid(texture))) assets.Add(GetAssetGuid(texture));
            }
        }
    }
    return assets;
}

private static void GetChildComponents(GameObject nodeGameObject, List<string> assets) {
    List<GameObject> childGameObjects = nodeGameObject.transform.Cast<Transform>()
        .Select<Transform, GameObject>(t => t.gameObject).ToList();
    foreach (var childGameObject in childGameObjects) {
        AddDescendantAssetsToList(childGameObject, assets);
    }
}
```

Lähdekoodi 8. AssetBundleBuilder-skriptin sisältämät tärkeimmät funktiot.

BuildBundle funktio rakentaa GameObject-oliosta riippuvuuspuun, jonka perusteella AssetBundlet tehdään asseteista, joista GameObject-olio on riippuvainen, sekä itse GameObject-oliosta. AddDescendantAssetsToList-funktiossa palautetaan lista kaikista mesheistä, materialeista ja tekstuureista, joista GameObject-olio on riippuvainen.

7.4 MVC:n soveltaminen Unity-ympäristössä

MVC- eli Model-View-Controller -malli on ohjelmistotekniikassa tunnettu rakenne, minkä perusajatuksena on se, että ohjelmalogiikka on eroteltu ohjelman graafisesta ulkoasusta ja käyttäjäsyötteitä vastaanottavasta kontrollerista. Tämä mahdollistaa sen, että esimerkiksi kontrolleri voidaan vaihtaa hiireen pohjautuvasta ratkaisusta peliohjaimeen pohjautuvaan ratkaisuun ilman, että ohjelmalogiikkaan ja visuaaliseen toteutukseen tarvitsee koskea. Vastaavasti logiikkaa tai graafista ulkoasua voidaan muuttaa koskematta muihin osiin. MVC-arkkitehtuuri myös parhaimmillaan selkeyttää ohjelman rakennetta ja koodia huomattavasti, sillä esimerkiksi ohjelmalogiikassa olevaa pinta-alan laskenta-algoritmia lukiessa ei tarvitse miettiä, mitä visuaalisia vaikutuksia algoritmin käytöllä on, tai mikä syöte liipaisee algoritmin. Näin kehittäjä voi keskittyä yhteen luontevasti rajattuun osaan kerrallaan.

Unity-ympäristössä MVC:n toteuttaminen ei onnistu aivan samalla tavalla kuin esimerkiksi web-sovelluksissa. Unityssa graafinen ulkoasu perustuu pitkälti siihen, mitä pyöritettävässä scene-tiedostossa on sisältöä. Tämä sisältö on omalla tavalla taas osa ohjelman logiikkaa, sillä sisältöön voi kuulua GameObject-olioita, joilla on fyysisiä ominaisuuksia kuten massa, nopeus, kiihtyvyys, pyörimisnopeus jne. Käyttäjäsyötteitä varten Unityssa on Input-luokka, jonka kautta voidaan tarkastella esimerkiksi näppäimistön ja hiiren tilaa, mutta tämä luokka ei yksin toteuta kovin vankkaa kontrollirakennetta laajan 3d-sovelluksen ohjelmalogiikan hallitsemiseksi.

Intellellä olemme kuitenkin halunneet tehdä selkeän rakenteen ohjelmalogiikan erottamiseksi, joten olemme päätyneet arkkitehtuuriseen ratkaisuun, jossa kontrolleriskripteiksi on eroteltu osiot, jotka manipuloivat jollain tavalla scene-tiedostoa, mutta jossa varsinainen logiikka on hyvin kevyttä, tai sitä ei ole. Kontrolleriskripteissä ei käytännössä tarvitse olla mitään tilaa, sillä ohjelman kaikki tila hallinnoidaan joko scene-tiedostossa UnityEnginen toimesta tai logiikkaskripteissä. Tilaa ovat käytännössä kaikki luokkamuuttujat, ja jos luokkamuuttujia ei ole, toimii kontrolleri itsenäisesti siten, että vain logiikkaskriptien ja scene-tiedoston tilalla on merkitystä kontrollerin funktioiden vaikutuksiin. Kontrolleriskriptit siis toimivat rajapintana scene-tiedoston ja ohjelmoijien tekemien logiikkaskriptien välillä. Esimerkki kontrolleriskriptistä voisi olla skripti, joka luo parametrisoidun määrän vihollisia pelikentälle. Logiikkaskripti voisi sen sijaan vaikka pitää kirjaa jäljellä olevista vihollisista, ja kun vihollisia on jäljellä

alle 10, pyytäisi logiikkaskripti kontrolleriskriptiä luomaan pelikentälle pelaajan tason mukaisen määrän vihollisia.

Logiikkaskriptien täytyy tuntea käyttämänsä kontrolleriskriptit, joten monessa tilanteessa on myös luonnollista, että logiikaskripti luo kontrolleriskriptistä instanssin, jota käyttää. Näin logiikkaskriptit saavat välitettyä halutut vaikutukset kontrollereiden kautta scene-tiedostoon. Jos järjestys halutaan olevan päinvastainen, eli scene-tiedostossa tapahtuu jotakin, joka liipaisee toiminnallisuutta logiikkaskripteissä, toteutetaan se event-olioiden avulla. Event-olioista kerrotaan lisää seuraavassa luvussa.

8 Unity3D:llä ohjelmontia helpottavia luokkia

Ohjelmointi Unity3D:llä perustuu vahvasti GameObject-olioiden ja niiden sisältämien komponenttien manipulointiin. Tästä arkkitehtuurisesta rakenteesta johtuen monet GameObject-olioihin liittyvät toimenpiteet on mahdollista vain luokissa, jotka perivät UnityEnginen MonoBehaviour-luokan. MonoBehaviour-luokkia ei toisaalta voida luoda normaalilla new-syntaksilla, vaan ne täytyy liittää johonkin GameObject-olioon AddComponent-syntaksilla. Tämä rakenne johtaa siihen, että moniin yleisiin ohjelmointiongelmiin on jouduttu tekemään Unitya varten omat toteutuksensa. Toisaalta tämä rakenne myös aiheuttaa ongelmia, joita ei normaalisti jouduttaisi ratkoa useimmista kehitysympäristöissä.

Tässä luvussa käsitellään tilakonetta, events-moduulia sekä singleton-luokkaa, joilla on ratkaistu joitakin kohtaamiamme ongelmia. Tämän kaltaisille apuluokille syntyy todennäköisesti lisää tarpeita sovelluksen kehittyessä pitemmälle, ja pidemmässä juoksussa uskon näiden luokkien muodostavan Intellen omia ohjelmointimalleja, jotka ovat verrattavissa yleisiin suunnittelumalleihin.

8.1 Tilakone

Ohjelmistossa erilaisia tiloja on lukemattomia. Uusi 32-bittinen integer-muuttuja tuo ohjelmistoon teoreettisesti 4.294.967.296 eli reilu 4 miljardia uutta mahdollista tilaa. Jos ohjelman eri tiloja manipuloidaan ja tarkastellaan hallitsemattomasti useista eri

skripteistä, niin syntyy tilanne, missä tilaa manipuloivassa koodissa oleva virhe voi asettaa sovelluksen väärään tilaan ja rikkoa useiden muiden ominaisuuksien toiminnallisuuden. Ongelmallista tämä on etenkin sen takia, että ongelma saattaa esiintyä ensimmäisen kerran jossain aivan toisessa ominaisuudessa kuin siinä, jossa virhe oikeasti on. Kun ongelma on jäljitetty väärässä tilassa olevaan komponenttiin, joudutaan virhettä vielä etsimään kaikista tilaa manipuloivista komponenteista, joten loppujen lopuksi pelkkä virheen löytäminen voi kestää monta kertaa kauemmin kuin itse vian korjaaminen. Tämän takia useiden toimintojen ollessa riippuvaisia jonkin komponentin tilasta, pitäisi tätä tilaa hallita keskitetysti, jotta tilanhallinnassa oleva virhe löydetään heti. Tähän ongelmaan ratkaisuna on käytetty tilakoneita.

Konkreettisempana esimerkkinä tuotteessamme otetaan käyttäjän syötteitä vastaan, mutta se miten niihin reagoidaan riippuu siitä, missä tilassa ohjelma on. Sovelluksella voidaan lisätä, poistaa ja siirrellä tavaroita pelikenttään, joten jos sovellus on poisto-tilassa ja käyttäjä klikkaa hiirtä tavaran päällä, niin tavara poistetaan. Jos sovellus on taas katselutilassa ja käyttäjä klikkaa hiirtä tavaran päällä, tulee tavara valituksi ja sovellus menee katselutilasta muokkaustilaan. Muokkaustilassa käyttäjän painaessa ja pitäessä hiirtä pohjassa tavaran päällä, hän voi siirrellä ja pyörittää tavaroita raahaamalla hiirtä. Katselutilassa sama käyttäjäsyöte taas yksinkertaisesti liikuttaisi kameraa raahauksen mukaisesti.

Tilalla tarkoitetaan järjestelmän tilaa, missä järjestelmä odottaa seuraavan transition suorittamista. Transitiolla taas tarkoitetaan toimenpiteiden (eng. action) joukkoa, jotka suoritetaan jonkin event-olion syntyessä tai ehtojen toteuduttua. Intellellä käyttämässämme tilakoneessa on ns. running action-, entry action- ja exit action -toimenpiteet. Running action suoritetaan event-olion syntyessä, mutta tilaa ei muuteta. Entry action sen sijaan suoritetaan, kun jonkin event-olion syntymän seurauksena mennään johonkin tilaan. Exit action taas suoritetaan tilasta poistuttaessa.

Tilakonetta ohjataan event-olioiden (suom. tapahtuma) avulla. Event-olio kuvaa ohjelmoijan määrittelemää tapahtumaa, joka liipaistaan halutussa paikassa. Esimerkiksi käyttäjän klikatessa tavaraa voitaisiin liipaista event-olio nimeltä “MouseClickOnItem”. Tilakone voi vastaanottaa tämän event-olion ja toimia event-olion pohjalta tilan vaatimalla tavalla. Käyttämässämme tilakoneessa eri tiloille voidaan määrittää toimenpiteitä, mitä suoritetaan tilaan mentäessä, tilasta lähdettäessä ja tilassa ollessa. Lisäksi voidaan määritellä, miten eri event-olioihin reagoidaan eri

tiloissa. Esimerkiksi meidän tapauksessamme oltaessa katselutilassa, kun tilakone vastaanottaa “MouseClickOnItem” event-olion, tilakone päättelisi, että sen pitäisi siirtyä muokkaustilaan ja että muokkaustilaan mentäessä hiiren alla oleva tavara pitää tulla valituksi.

Lähdekoodissa 9 esitellään tilakoneen käyttöä. Ensimmäisenä tilakoneelle on määritelty IEvent-luokkia, joita tilakone voi kuunnella. Seuraavaksi määritellään tilakoneen eri tilat, ja että mitä Entry, Exit ja Running actioneja milläkin tilalla on. Itse tilasiirtymät määritellään seuraavaksi luontevalla syntaksilla niin, että aluksi ilmoitetaan missä tilassa ollaan ja sitten kerrotaan, minkä event-olion syntyminen aiheuttaa siirtymisen mihinkä tilaan. Esimerkiksi ThirdPersonViewState-tilassa ollessa SwitchFirstPersonViewState event-olion syntyessä tilakone siirtyy FirstPersonViewState-tilaan. Lopuksi tilakoneen konstruktorissa määritellään event-oliot, joita tilakoneen tulee kuunnella, sekä tila, jossa tilakone on ensimmäisenä.

```
// Events
// ======
public class SwitchFirstPersonViewState : IEvent {}
public class SwitchThirdPersonViewState : IEvent {}
public class SwitchTopDownViewState : IEvent {}
public class SwitchTopOrthographicViewState : IEvent {}

// States
// ======
private static readonly State FirstPersonViewState =
new State("FirstPersonViewState", CameraModeChangeFunction(CameraModeController.CameraModes.FirstPerson));
private static readonly State ThirdPersonViewState =
new State("ThirdPersonViewState", CameraModeChangeFunction(CameraModeController.CameraModes.ThirdPerson));
private static readonly State TopDownViewState =
new State("TopDownViewState", CameraModeChangeFunction(CameraModeController.CameraModes.TopDown));
private static readonly State TopOrthographicViewState =
new State("TopOrthographicViewState", CameraModeChangeFunction(CameraModeController.CameraModes.TopOrthographic));

static CameraModeStateMachine() {
    FirstPersonViewState
        .When<SwitchThirdPersonViewState>(ThirdPersonViewState)
        .When<SwitchTopDownViewState>(TopDownViewState)
        .When<SwitchTopOrthographicViewState>(TopOrthographicViewState);
    ThirdPersonViewState
        .When<SwitchFirstPersonViewState>(FirstPersonViewState)
        .When<SwitchTopDownViewState>(TopDownViewState)
        .When<SwitchTopOrthographicViewState>(TopOrthographicViewState);
    TopDownViewState
        .When<SwitchFirstPersonViewState>(FirstPersonViewState)
        .When<SwitchThirdPersonViewState>(ThirdPersonViewState)
        .When<SwitchTopOrthographicViewState>(TopOrthographicViewState);
    TopOrthographicViewState
        .When<SwitchFirstPersonViewState>(FirstPersonViewState)
        .When<SwitchThirdPersonViewState>(ThirdPersonViewState)
        .When<SwitchTopDownViewState>(TopDownViewState);
}

public CameraModeStateMachine(CameraModeController cameraModeController): base(FirstPersonViewState)
{
    _cameraModeController = cameraModeController;
    Events.Instance.AddListener<IEvent>(EventHappens);
}
```

Lähdekoodi 9. Esimerkki tilakoneen käytöstä.

8.2 Events-moduuli

Tuotetta tehdessämme käytimme aluksi melko paljon UnityEnginen GameObject.Find()- ja GameObject.FindGameObjectWithTag()-funktioita. Näiden funktioiden avulla voidaan hakea scene-tiedostosta tietyn nimisiä GameObject-olioita ja GameObject-olioita, joilla on tietty kehittäjän määrittelemä tagi. Alkuvaiheissa tämä lähestymistapa ei vaikuttanut ongelmalliselta, vaan se tuntui oikeastaan todella helpolta ja nopealta tavalta löytää haluamansa GameObject-olio, jotta päästiin käyttämään sen sisältämän skriptin palveluita. Tämä mahdollisti sen, ettei scene-tiedostossa tarvinnut pitää yllä hyvin organisoitua arkkitehtuurista rakennetta, vaan jos haluttiin tehdä joku uusi pieni toiminto, riitti, kun teki scene-tiedostoon uuden GameObject-olion, keksi sille mahdollisimman kuvaavan tagin ja laittoi uuden skriptin siihen GameObject-olioon, jolloin sekin olisi käytettävissä kaikkialta koodista. Jälkikäteen ajateltuna oli melko naiivia ajatella, että tällainen lähestymistapa olisi hyvä ja toimisi pitemmän päälle.

Ensimmäiset oireilut huonosta arkkitehtuurista olivat virheet, joita syntyi kirjotusvirheistä ja GameObject-olioiden ja tagien nimenmuutoksista. Koodista GameObject-olioita etsittäessä tagin tai nimen perusteella funktiolle annetaan parametriksi vain kovakoodattu string-muuttuja. Jos joku muuttaa jotakin tagia tai GameObject-olion nimeä, ei funktio löydäkään mitään GameObject-oliota ja ohjelma antaa NullReferenceException-virheen yrittäessä löytää haluamaansa skriptiä GameObject-oliosta. Tämä oli kevyt lieveilmiö huonosta arkkitehtuurista, sillä ongelman sai korjattua etsimällä haluamansa GameObject-olion scene-tiedostosta ja tarkastamalla sen nimi ja tagi. Seuraava hieman hankalampi lieveilmiö syntyi, jos jollekin GameObject-oliolle annettiin väärä tagi tai sellainen nimi, joka on jo olemassa, vaikka nimen pitäisi olla uniikki. Tällaisessa tapauksessa GameObject-oliota käyttävä skripti löytää väärän GameObject-olion ja heittää NullReferenceException-virheen yrittäessään hakea GameObject-olioon liitettyä skriptiä, jota väärässä GameObject-oliossa ei olekaan. Tämä johtaa pahimmillaan siihen, että kehittäjän täytyy käydä kaikki scene-tiedoston GameObject-oliot läpi ja tarkistaa niiden nimi tai tagi vain löytääkseen virheen alkuperän.

Päästäksemme Find()- ja FindGameObjectWithTag()-funktioiden käytöstä eroon, otimme käyttöön events-moduulin, joka mahdollistaa toimintojen liipaisemisen toisissa skripteissä ilman, että skriptin tarvitsee hakea ensin GameObject-olio, johon toinen skripti on kytketty. Events-moduuli toimii käytännössä siten, että kehittäjä voi määritellä

IEvent-rajapinnan perivän luokan, josta tehdään instanssi silloin, kun jokin muita skriptejä kiinnostava asia tapahtuu. Esimerkiksi hiiren liikuttaminen voi kiinnostaa useitakin komponentteja ohjelmassa. Se saattaa mahdollistaa tavaroiden raahaamisen peliympäristössä, kameran liikuttelun tai mitä tahansa muuta. Jotta hiiren tilaa ei tarvitsisi tarkastella joka paikassa erikseen, voidaan tehdä kontrolleri, joka tarkastelee hiiren sijaintia, ja mikäli sijainti on eri kuin edellisellä framella, voidaan luoda MouseMove event-olio, joka ottaa konstruktorissaan parametrina sijaintinsa xy-koordinaatistossa. Event-oliota voidaan näin käyttää muissa skripteissä siten, että asettaa skriptin Start()-funktiossa kyseiselle event-oliolle kuuntelijan, johon on määritelty mitä skriptin pitäisi tehdä event-olion syntymisen yhteydessä ja mitä mahdollisilla event-olion sisältämillä tiedoilla tehdään. Events-moduuli poistaa näin ongelman, joka syntyi kovakoodatuista tagien ja GameObject-olioiden nimistä.

Event-olioiden luomisesta esitellään esimerkki lähdekoodissa 10. Esimerkissä event-olio luodaan OnClick-funktiossa, joka tapahtuu, kun tämän skriptin sisältämää GameObject-oliota klikataan. Kyseinen GameObject-olio on tässä tapauksessa nappi, josta vaihdetaan ensimmäisen persoonan näkymään.

```
public void OnClick()
{
    Events.Instance.Raise(new CameraModeStateMachine.SwitchFirstPersonViewState());
}
```

Lähdekoodi 10. Esimerkki event-olion luomisesta.

Tässä yhteydessä on kuitenkin hyvä huomata, että kovakoodattujen nimien ja tagien aiheuttamat ongelmat olivat vain oireita paljon suuremmasta ongelmasta, mikä tuotantotiimiämme koetteli. Koska emme kiinnittäneet tarpeeksi huomiota koodin arkkitehtuuriin, vaan kehitimme sovellusta yksittäisten toimintojen vetämänä, ongelmat alkoivat kasaantua, ja sovellus alkoi levitä käsiin. Meiltä puuttui ylemmän tason abstraktioita, jotka sitovat kokonaisuuksia sieviksi paketeiksi, jotka on helppo ymmärtää ja jotka tuovat sovellukseen selkeyttävää hierarkiaa. Meidän kaikki toiminnallisuudet liikkuivat hyvin matalalla abstraktiotasolla, jonka takia johonkin yksittäiseen toiminnallisuuteen perehtyminen kesti hyvin pitkään, sillä kehittäjän täytyi selvittää toiminnan ympäriltä hyvin paljon asioita, jotta hän sai jonkinlaisen isomman kuvan toiminnon kantavasta ajatuksesta. Lisäksi scene-tiedostoon ja tag manageriin alkoi kerääntyä GameObject-oliota ja tageja, joita ei käytetty koko sovelluksessa. Kukaan ei

kuitenkaan uskaltanut siivota näitä roskia, sillä kehittäjät alkoivat pelätä, että he aiheuttavat virheitä, jos poistavat tageja tai GameObject-olioita. Vaikka kehittäjä olisi itse aiemmin luonut jonkin tagin, ei hän myöhemmin välttämättä enää uskaltanut poistaa sitä, sillä joku muu kehittäjä on voinut tehdä skriptejä, jotka käyttävät kyseistä tagia.

Tilannetta voisi havainnollistaa armeijalla, joka on hyvin hierarkkinen järjestelmä. Kuvitellaan, että kaikki sotilaat liikkuisivat samalla matalalla abstraktiotasolla, eli olisivat sotamiehiä. Jos Haminaan kohdistuisi suurhyökkäys ja rintaman rakennetta pitäisi muuttaa siten, että Haminan alueelle pitäisi tuoda muualta Suomesta lisää sotilaita, miten se tehtäisiin? Yksittäisille sotilaille pitäisi ympäri Suomea antaa käskyjä siirtyä tiettyihin asemiin Haminan lähellä. Tällainen ei olisi mitenkään mahdollista yhdeltä ihmiseltä järkevässä ajassa. Selkeän hierarkian avulla viesti voi kuitenkin kulkea armeijassa parhaimmillaan hyvin nopeastikin ja suuria sotilasmassoja pystytään keskittämään tarkkaa harkittujen päätösten perusteella siten, että koko armeija toimii yhteisen tavoitteen saavuttamiseksi mahdollisimman tehokkaasti. Jokaisella on selkeä tehtävä ja selkeä vastuu, ja näin saadaan paras yhteissuoritus niin taisteluparilta, ryhmältä, joukkueelta, komppanialta, pataljoonalta, rykmentiltä ja lopulta koko rintamalta. Vähän mielikuvitusta käyttäen tätä esimerkkiä voidaan soveltaa myös ohjelmistotekniikkaan, missä ohjelmiston kasvaessa selkeä rakenteellisuus on ehdoton edellytys koodin ymmärtämiselle ja jatkokehitykselle.

8.3 Singleton

Singleton on suunnittelumalli, jota käytetään paljon ohjelmoinnissa. Perusajatus singletonissa on, että halutaan pitää huoli, että jostakin tietystä luokasta on mahdollista tehdä vain yksi ja ainoa instanssi sovelluksessa. Helppo esimerkki käyttökohteesta on ajasta huolehtiva komponentti, jolta voidaan kysyä jonkin tapahtuman ajankohtaa ja esimerkiksi päätellä tämän avulla kilpailussa nopeimmin pärjännyt kilpailija. Tällaiseen tarkoitukseen on luontevaa, että olemassa on vain yksi komponentti joka antaa ajan, sillä näin estetään useiden komponenttejen joutumasta asynkroniseen käsitykseen ajasta, jolloin kilpailun tulokset voisivat vääristyä. Meidän tuotteessamme hyvä käyttökohde on esimerkiksi RESTClient -komponentille, joka tekee http-kyselyjä ja ottaa vastaan vastaukset. RESTClient tulee olla käytettävissä ohjelman koko elinkaaren ajan, ja kun siitä on olemassa vain yksi instanssi, voidaan palvelimen

osoitetta esimerkiksi muuttaa ajonaikaisesti ilman, että osoitetta täytyisi muuttaa jokaiseen yksittäiseen instanssiin. Unity-ympäristössä singletonista on toinenkin etu. Koska singletonia voidaan kutsua globaalisti mistä tahansa, ei singletonia käyttävien skriptien tarvitse hakea sitä Find()- ja FindGameObjectWithTag()-funktioilla, vaan he voivat viitata suoraan singletionin staattisen getterin palauttamaan olioon.

Lähdekoodeissa 11 ja 12 esitetään, miten RestClientista on tehty singleton ja miten RestClientin palveluja voidaan tämän jälkeen käyttää.

```
public class RestClient : Singleton<RestClient> {
    ...
}
```

Lähdekoodi 11. RestClientin muuttaminen singletoniksi.

```
void Authenticate(Credentials cred, bool rememberMe)
{
    RestClient.Instance.Authenticate(cred, () => LoginSuccess(), () => LoginFailed(), rememberMe);
    _loginText.text = "Authenticating...";
}
```

Lähdekoodi 12. RestClientin palveluiden käyttö singletonin avulla.

Luokasta tehdään Singleton perimällä Singleton-luokka ja antamalla sille tyyppiparametriksi luokka, josta halutaan tehdä Singleton. Singletonia toteuttavaa luokkaa voidaan käyttää helposti Instance-funktion kautta. Instance-funktio palauttaa ainoan olemassaolevan RestClientin ilmentymän. Jos olemassaolevaa ilmentymää ei ole, se luodaan.

Singleton on kuitenkin erittäin kiistelty suunnitelumalli [Why Singletons Are Controversial 2010], sillä singleton tuo sovellukseen globaalia tilaa, joka ei sovelluksen testauksen kannalta ole haluttua, kuten tilakoneita käsittelevästä luvusta käy ilmi. Singleton tekee tiukan kytköksen sitä käyttäviin luokkiin tarkoittaen sitä, että luokan toiminta riippuu singletonista. Jos singletonia käyttävää luokkaa haluttaisiin esimerkiksi testata, ei sitä voitaisi tehdä eristetysti, kuten yleensä halutaan, vaan singleton pitäisi aina ottaa myös testiin mukaan. Meidän tuotteessamme singletonin liikakäyttöä on esiintynyt esimerkiksi tilanteessa, missä jokin käyttäjän syötteitä hallinnoiva skripti on tarvinnut kameraa hallinnoivalta skriptiltä tietoja, jolloin kameraskripti on muutettu singletoniksi ja siihen on ollut mahdollista päästä käsiksi syöteskriptistä. Tämä ohittaa

kaiken hierarkian ja rakenteen. Pitkälle vietynä singleton voi synnyttää tilanteen, missä kaikki skriptit ovat täynnä tiukkoja kytköksiä toisiinsa, eikä mitään voida testata eristettynä, ja yksittäisen virheen etsiminen voi tarkoittaa koko koodikannan debuggaamista.

9 Tulosten arviointia ja yhteenveto

Kehittämämme tuote elää tällä hetkellä melko varhaisessa vaiheessa elinkaartaan, minkä takia monet ohjelmointikäytännöt, arkkitehtuuri, työkalut ja automaatiot ovat vielä alkutekijöissään. Kliseeksi muodostunut hokema “asenne ratkaisee” pitää tässäkin kohdassa paikkaansa, sillä prosessit ja työkalut eivät tule ikinä olemaan täydellisiä, mutta sovelluksen kasvaessa niitä pitää orjallisesti ja aktiivisesti kehittää, jotta kehitystiimi pysyisi mahdollisimman tuottavana.

Kun vertaa Intellen ohjelmointitiimin nykytilannetta siihen, mitä se oli 10 kuukautta sitten, voin todeta että kehitys on ollut erinomainen. Noin kuukausi sitten saimme tuotteemme kehitettyä uudella pohjalla siihen pisteeseen, missä se oli noin 5 kuukautta sitten ennen uudelleensuunnittelua ja -kirjoitusta. Nyt sovelluksen jokaisen moduulin ja luokan lähdekoodista on helppo ymmärtää, mitä kyseinen koodi tekee, ja jatkokehitys tuntuu aiempaan verrattuna uskomattoman nopealta. Siinä missä aiemmin pienen ominaisuuden tekemiseen saattoi pahimmillaan kulua viikkoja, niin nyt samanlaisia ja suurempia ominaisuuksia valmistuu päivittäin. Jos pystymme pitämään koodikantamme yhtä siistinä ja jäsenneltynä kuin mitä se tällä hetkellä on, ja kehittää uusia hyviä automaatioita, aputyökaluja ja menetelmiä ohjelmoinnin tueksi, uskon, että meillä on erinomaiset mahdollisuudet voittaa kilpailijamme niin laadun kuin tuottavuudenkin mittareilla, vaikka kilpailijallamme olisikin enemmän resursseja käytettävissä. Tämän takia uskon, että meillä on kaikki mahdollisuudet tehdä Intellestä huippumenestystarina.

Tässä työssä on käsitelty joitakin tärkeimpiä tekniikoita ja työkaluja, joita Intellellä käytetään Unity3D-projektien kanssa. Olen käynyt läpi korkean tason periaatteita koodin luettavuudesta ja ymmärrettävyydestä, joita voidaan soveltaa ohjelmoinnissa yleisesti teknologiasta riippumatta. Olen tuonut esille, kuinka vanhan koodin jatkuva siistiminen ja kehittäminen on äärimmäisen tärkeää, ja esittänyt esimerkin avulla, kuinka refaktorointia voidaan harjoittaa. Työssä myös syvennyttiin Unity3D:lle

ominaisiin tekniikoihin ja ongelmiin ja esitettiin näihin ongelmiin joitakin ratkaisuja. Vaikka menetelmät ovat auttaneet meitä paljon eteenpäin, on tämä kuitenkin vasta pintaraapaisua ylläpidettävyyden parantamisesta.

Lähteet

Martin, R. 2009. Clean Code: A Handbook of Agile Software Craftsmanship. Boston: Pearson Education, Inc.

About Version Control. Git. <http://git-scm.com/book/en/Getting-Started-About-Version-Control>. Luettu 25.11.2013.

Comparison: Perforce and Git. 2012. Perforce. <http://www.perforce.com/sites/default/files/pdf/perforce-git-comparison.pdf>. 2012. Luettu 25.11.2013.

Per Asset Versioning with Unity Asset Bundles. 2013. Jason McGuirk. <http://blog.juiceboxmobile.com/2013/06/19/per-asset-versioning-with-unity-asset-bundles/>. Päivitetty 19.6.2013. Luettu 25.11.2013.

What is Jenkins? 2013. Kohsuke Kawaguchi. <https://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins>. Päivitetty 7.1.2013. Luettu 23.2.2014.

Creating Scenes. 2013. Unity Technologies. <http://docs.unity3d.com/Documentation/Manual/CreatingScenes.html>. Päivitetty 31.7.2013. Luettu 25.11.2013.

Prefabs. 2013. Unity Techonologies. <http://docs.unity3d.com/Documentation/Manual/Prefabs.html>. Päivitetty 16.7.2013. Luettu 25.11.2013.

50 Tips for Working with Unity. 2012. Herman Tulleken. <http://devmag.org.za/2012/07/12/50-tips-for-working-with-unity-best-practices/>. Päivitetty 12.7.2012. Luettu 25.11.2013.

AssetBundles. 2013. Unity Technologies. <http://docs.unity3d.com/Documentation/Manual/AssetBundlesIntro.html>. Päivitetty 23.4.2013. Luettu 25.11.2013.

Cockburn, A. 2000. Agile Software Development. Verkkodokumentti. IT University of Copenhagen. <http://www.itu.dk/~oladjones/semester2/Project2/materials/newmaterials/Agile%20Software%20development.pdf>.

Why Singletons Are Controversial. 2010. Verkkodokumentti. Google Project Hosting. <https://code.google.com/p/google-singleton-detector/wiki/WhySingletonsAreControversial>.

Konekieli. 2013. Verkkodokumentti. Wikipedia Foundation. <http://fi.wikipedia.org/wiki/Konekieli>.

Fowler, M. Beck, K. Brant, J. Opdyke, W. Roberts, D. 2002. Refactoring: Improving the Design of Existing Code. Omakustanne.

Norvig, P. 1996. Design Patterns in Dynamic Programming. Verkkodokumentti. <http://www.norvig.com/design-patterns/design-patterns.pdf>.

Yang, E. 2010. Design Patterns in Haskell. Verkkodokumentti. <http://blog.ezyang.com/2010/05/design-patterns-in-haskel/>.

I2 ohjelmointikäytännöt

- Muuttujien, metodien, luokkien yms. nimet tulevat olla merkityksellisiä, jotta niistä selviää mikä kyseisen elementin tarkoitus on
- Metodit tulee pilkkoa lyhyisiin selkeisiin kokonaisuuksiin, jotka suorittavat yleisesti yhden nimettävissä olevan toiminnon
- Vältä copy-pastea, sillä toistuvat rakenteet voidaan yleensä toteuttaa yhdellä geneerisemmällä ratkaisulla, joka vähentää tarvittavien koodirivien määrää ja helpottaa koodin ylläpidettävyyttä
- Käytä LINQ lausekkeita aina kun mahdollista
- Käytä isoja alkukirjaimia luokissa, skripteissä, propertyissä, public fieldeissä, metodeissa
- Aloita privaatit fieldit alaviivalla (esim. ”private int _number”)
- Järjestele luokkasi seuraavaan järjestykseen ylimmästä lähtien: 1. public fieldit, 2. private fieldit, 3. public metodit, 4. private metodit, 5. Aliluokat
- Käytä fieldeissä ja metodeissa aina korkeinta mahdollista suojaustasoa, eli ensisijaisesti private (ja mahdollisuuksien mukaan readonly), mutta tarpeen tullen protected tai public
- Vältä turhaa kommentointia. koodin toiminnallisuus pyritään pitää selkeänä hyvillä nimeämiskäytännöillä
- Käytä sisennyksissä neljää (4) välilyöntiä
- Pidä oletusarvoisesti aina IEnumerator luokat privaatteina, jotta StartCoroutinea kutsutaan vain samassa luokassa ja coroutinen elinkaari pysyy hallinnassa