TAMPEREEN
AMMATTIKORKEAKOULU

SQLite-tietokantaratkaisu C#-pohjaisessa pelissä

Eero Leppäniemi

Opinnäytetyö
Elokuu 2016
Tietojenkäsittely
Pelituotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma
Pelituotanto

LEPPÄNIEMI, EERO:
SQLite-tietokantaratkaisu C#-pohjaisessa pelissä

Opinnäytetyö 40 sivua, joista liitteitä 2 sivua
Elokuu 2016

Työn toimeksiantaja Dreamloop Games Oy tarvitsi peliinsä Challengers of Khalea tietokantaratkaisun, joka olisi kevyt eikä tarvitsisi ulkoista palvelinta toimiakseen. Peli tarvitsi tämän järjestelmän, jotta pelaaja voisi tallentaa ja ladata pelin tilanteen. Challengers of Khalea on Unityn pohjalle toteutettu peli, joka on ohjelmoitu C#-kielellä.

Ratkaisun tietokantajärjestelmäksi valittiin SQLite, joka vastasi haetun järjestelmän vaatimuksia. SQLiten toimivuuden helpottamiseksi ohjelmistoon ladattiin myös C#-kirjasto Dapper.

Ohjelmisto, johon järjestelmä toteutettiin, käytti arkkitehtuuriratkaisuna StrangeIoC-viitekehystä. StrangeIoC käyttää MVC-rakenteen (model-view-controller) muokattua versiota nimeltä MVCS (model-view-controller-service). Opinnäytetyön kaikki C#-koodiesimerkit noudattavat tätä rakennetta.

Työssä luotiin rakenne tietokantajärjestelmälle ja se toteutettiin toimivana kokonaisuutena toimeksiantajan peliin. Toteutukseen kuului tietokannan luominen, tietokantaan tiedon syöttäminen, tiedon hakeminen tietokannasta, komentojen yhdistäminen helposti käytettäviin luokkiin ja komento, joka toteuttaa koko pelin tallentamisen.

Lopputuloksena järjestelmä saatiin toteutettua ja se toimi odotetulla tavalla. SQLite-syntaksin ohjelmoinnin todettiin olevan melko kankeaa, mutta tietokanta toimi SQLite-viitekehyksen valmistuttua helposti. Myös Dapper vähensi vaivaa SQLiten ohjelmoinnissa. StrangeIoC:n todettiin myös parantavan ohjelmiston käyttökokemusta.

---

tietokanta sqlite strangeioc dapper unity mvcs

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Business information systems
Game Development

LEPPÄNIEMI, EERO
SQLite Database Solution in a C# Based Game

Bachelor's thesis 40 pages, appendices 2 pages
August 2016

Dreamloop Games inc, commissioner of this thesis, needed a database solution for their game, Challengers of Khalea. The database had to be light and it should independent of external servers. The game needed this solution for saving and loading the progress of the game. Challengers of Khalea is a game that was developed in the Unity framework and it is coded in the C# language.

SQLite was chosen as the database for the solution, as it corresponded with the desired goals. To make the SQLite coding easier, the C# library Dapper was included in the development tools.

The System where the solution was created used StrangeIoC framework as an architectural solution. StrangeIoC uses a model-view-controller-service structure, which is a modified version of the original model-view-controller-structure. All code examples in this thesis follow this architecture.

Database structure was created for the game, as well as the whole database system. The System included creation of the database, getting the data from the database, setting the data into the database, combining the methods into easily usable classes and one command, which saves the whole progress of the game.

On the whole, the system was successfully created and it worked as intended. Coding in the SQL-syntax seemed to be quite hard, but working with the database itself was easy after the framework was completed. Dapper also made working with the SQLite much easier. StrangeIoC was also seen as contributing to making the codebase more usable.

---

database sqlite strangeioc dapper unity mvcs

SISÄLLYS

ERITYISSANASTO

Unity	Pelinkehitysympäristö
Skripti	Engl. script, Unityn yhteyteen luotu kooditiedosto, joka sisältää yhden tai useamman ohjelmistoluokan
SQLite	Tietokantakirjasto
C#	C-kantainen ohjelmointikieli
Tietue	Muuttuja joka sisältää tietoa
Modulaarinen	Itsenäisistä osista koostuva kokonaisuus. Esimerkki: modulaarisen ohjelmiston osat eivät ole riippuvaisia toisistaan.
Kysely	Engl. query, SQL-komento joka palauttaa pyydettyä tietoa
Malli	Datamalli, engl. Datamodel, model. Tiedon tilan säilytykseen tarkoitettu luokka.
Ilmentymä	Engl. instance. Yksittäinen erikseen luotu versio luokasta.
Soturi	Engl. warrior, Challengers of Khalea -pelin gladiaattori

1 JOHDANTO

Peliala on kasvanut merkittävästi lähivuosien aikana ja se on kasvanut nopeammin kuin mikään muu viihdeteollisuuden haara. Vuonna 2015 pelimyynnin arvoksi arvioitiin maailmanlaajuisesti noin 92 miljardia dollaria. Peliteollisuudesta on myös tullut 2000-luvun aikana yksi Suomen hallitsevista kulttuurivientiteollisuudesta. Suomen pelitoimialan liikevaihdon arvioidaan olleen vuonna 2015 noin 2,4 miljardia euroa. (neogames.fi, 2016.) Dreamloop Games Oy on yksi tämän teollisuuden aloittelevista yrityksistä. Heidän luomaansa projektiin Challengers of Khalea tarvittiin tietokantaratkaisu, jotta pelaaja voi tallentaa ja ladata pelin tilanteen. Tämän opinnäytetyön tarkoitus on käsitellä Dreamloop Games Oy:n projektiin luotavaa tietokantaratkaisua.

Peli on toteutettu Unityllä, joka on maailman käytetyin usean alustan pelimoottori. Unity Technologies on useassa eri maassa toimiva yritys, joka kehittää tätä pelinkehitysalustaa. Pelimoottorilla tarkoitetaan pelinkehitysympäristöä, johon sisältyvät kehitystyökalut, kääntäjät (engl. compiler), ohjelmistokirjastot ja mahdolliset tukiohjelmat. Unityllä ohjelmointiin käytetään joko C#-, tai JavaScrip-kieltä, joista tähän projektiin on valittu C#. Unityn yleistymisen vuoksi myös C# on yleistynyt pelien ohjelmoinnin välineenä. (unity3d.com, 2016.)

Kaikki opinnäytetyön koodiesimerkit on ohjelmoitu C#-kielellä, joista jotkin sisältävät myös SQL-syntaksia. Unityn toimintaan ei pääasiallisesti keskitytä tässä opinnäytetyössä. Vaikka peli toimii Unityn pohjalta, kaikki tallentamiseen ja lataamiseen liittyvät ohjelmistot käsitellään Visual Studion avulla omassa projektissaan.

Tietokantamenetelmäksi työhön valittiin SQLite, joka on ilman palvelinta toimiva kevyt tietokanta. SQLiten toimivuuden yksinkertaistamiseksi projektiin implementoitiin myös C#-kirjasto Dapper.

Projektiin luotuun viitekehykseen toteutettiin tietokannan ja taulukoiden luonnin lisäksi toiminnallisuus tiedon tallentamiseen, lataamiseen ja muokkaamiseen. Tämän lisäksi toteutettiin myös komennot, joilla peli voidaan kokonaisuudessaan tallentaa ja ladata. Opinnäytetyö keskittyy selittämään ja tutkimaan luotuja ohjelmistoratkaisuja, niiden toimivuutta ja käytännön toteutusta.

2 TOIMEKSIANTO

2.1 Dreamloop Games

Dreamloop Games Oy on vuonna 2015 perustettu Tamperelainen startup-peliyritys. Yrityksen perustivat nykyiset toimitusjohtaja Joni Lappalainen, markkinointipäällikkö Steve Stewart ja tekniikkapäällikkö Hannes Väisänen. Yritys kehittää tällä hetkellä peliä Challengers of Khalea. Dreamloop Games fuusioitui vuonna 2016 Vasara-peliyrityksen kanssa. Fuusion seurauksena yrityksellä on nykyisin omistuksessaan myös pc-peli Stardust Galaxy Warriors, joka julkaistiin vuoden 2015 lopulla. (Dreamloop Games, 2016)

2.2 Challengers of Khalea

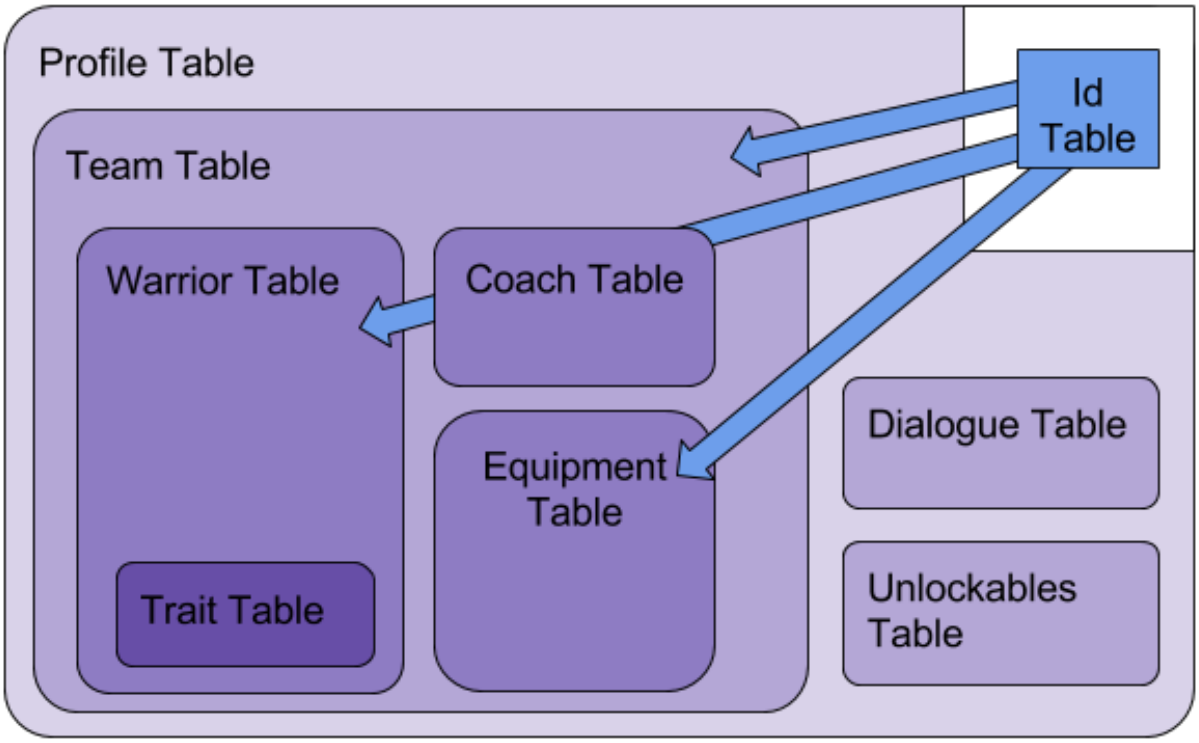
Challengers of Khalea on Dreamloop Games Oy:n kehitteillä oleva peli. Peli on toteutettu käyttäen pelinkehitysympäristönä Unityä ja se on ohjelmoitu C#-kielellä.

Challengers of Khalea on yhdistelmä shakkia muistuttavaa vuoropohjaista taktiikkapeliä, tarinavetoista dialogirikasta roolipeliä ja strategista oman joukkueen manageröintia. Pelaaja luo pelin alkaessa oman gladiaattorijoukkueensa jonka etenemistä pelissä seurataan. Pelin kuluessa joukkue matkustaa ympäri maailmaa ja taistelee muita joukkueita vastaan. Näitä joukkueita ohjaa tietokoneen tekoäly ja ne kehittyvät samalla tahdilla kuin pelaajan oma joukkue. Pelaaja myös ostaa joukkueeseen uusia sotureita (engl. warrior) ja näille varusteita lukuisista eri kauppapaikoista pelin sisäisellä valuutalla. Pelin kulkua tahdittaa jatkuvana kulkeva juoni ja siihen liittyvä dialogi pelin hahmojen välillä.

Challengers of Khaleaan tarvittiin menetelmä pelaajan tallennusten säilyttämiseen. Tietokanta valittiin tietojen tallentamisen menetelmäksi, jotta se olisi erillinen tiedosto ohjelmiston ulkopuolella. Tietokantajärjestelmän tuli myös olla kevyt, eikä se saanut vaatia ulkoista palvelinta ja ylläpitoa toimiakseen. Tämä tarkoittaa käytännössä sitä, ettei pelinkehittäjän tarvitse ylläpitää tietokantaa, vaan jokainen peli luo pelattaessa tietokoneen levylle oman itsestään päivittyvän tietokannan. Opinnäytetyö keskittyy luomaan peliin toimivan järjestelmän, joka tallentaa pelin tiedot ja tapahtumat SQLite-taulukkoon josta ne voi myöhemmin ladata pelattavaksi.

2.3 Tietokannan Rakenne

Challengers of Khaleassa muuttuvia asioita olivat muun muassa yksittäiset soturit, joukkueet, varusteet ja pelissä käydyt dialogit. Kuviossa 1 esitellään kaikki projektiin suunnitellut taulukot. Opinnäytetyössä keskitytään esittelemään WarriorTable, sen sisältö ja sen muokkaaminen.



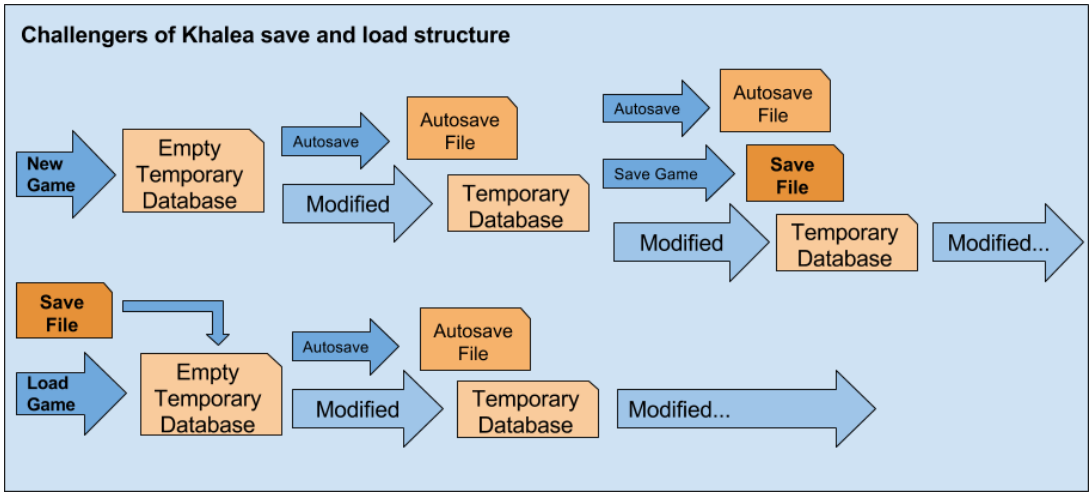
Kuvio 1 tietokannan rakenne havainnollistettuna kaaviolla

Taulukoissa viitataan toisiin taulukoihin relaatiotietokannan periaatteiden mukaan, eli antamalla jokaiseen taulukkoon yhdeksi arvoksi lukuarvo, joka viittaa toisen taulukon avainlukuun. Esimerkiksi WarriorTableen luodaan TeamId joka viittaa TeamTablen availukuun, jolloin jokaisella soturilla on tieto siitä, mihin joukkueeseen se kuuluu. Tietokannan toiminnasta kerrotaan lisää luvussa 3.

2.4 Tallennettavat tietokannat

Kun yksittäinen tietokanta on kerran luotu, se voidaan luoda useita kertoja helposti uudelleen. Challengers of Khalean pelin kulun tallennus tapahtuu luomalla pelin alkaessa

väliaikainen tietokanta, jonne tietoa talletetaan ja josta sitä palautetaan käyttöön. Kun pelaaja haluaa tallentaa nämä pelatessa muodostuneet tiedot, luodaan pelaajan nimeämä tallennustietokanta (engl. save file) ja kopioidaan kaikki tiedot väliaikaisesta tietokannasta (engl. temporary database) tähän pysyvämpään tietokantaan. Tietokanta myös varmuuskopioidaan pelin kuluessa automaattiseen tallennustiedostoon (engl. autosave file). Myöhemmin kun pelaaja tahtoo ladata pelin, tallennetun tietokannan tiedot kopioidaan sellaisenaan väliaikaiseen tietokantaan. Kuviossa 2 havainnollistetaan tämän järjestelmän toimintaa.



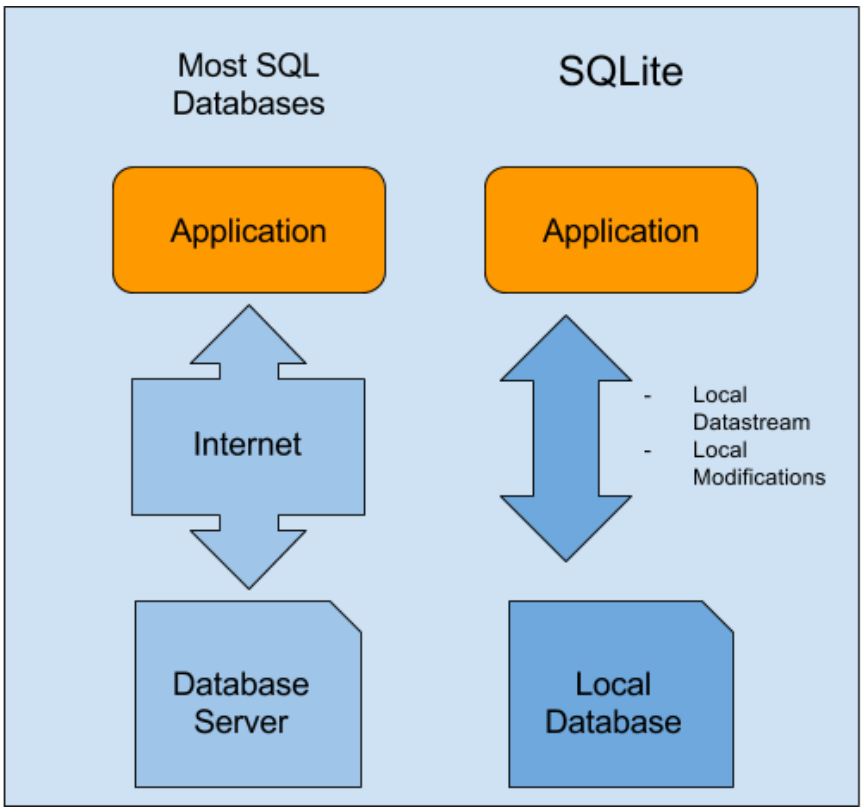
Kuvio 2 tallennusjärjestelmän toiminta kaaviona

3 SQLITE

SQLite on relaatiotietokanta. Tämä tarkoittaa joukkoa tietueita, jotka on järjestetty taulukon muotoon, jotka voidaan tietokannan sisäisesti liittää toisiinsa ja jotka voivat viitata muihin taulukoihin. Tietokannan etuna on se, että sitä voidaan käsitellä ohjelmiston ulkopuolella ja se on oma modulaarinen osansa. Tietokanta vähentää myös yksittäisten tietojen hakemisen vaivaa luomalla tallennetulle tiedolle järjestyksen. Relaatiotietokannat ovat nykyaikana pääasiallinen tiedon tallettamisen muoto. (Connolly T. & Begg C. 2005. Database Systems.)

SQLite on tietokantamalli, joka esimerkiksi MySql:stä poiketen toimii lähes täysin itsenäisesti vaatien vain vähän muiden ohjelmistojen tukea. Se on kehitteillä oleva kirjasto, joka toteuttaa itsenäisen, ilman palvelinta toimivan, transaktionaalisen SQL-tietokantamoottorin. SQLiteä käyttävät muun muassa Google, Facebook, Apple, Adobe sekä monet muut suuret yritykset. Se on maailman käytetyin tietokanta ja se on käytössä esimerkiksi jokaisessa Android- ja iOS-laitteessa sekä Skypessä, iTunesissa ja käytetyimmissä selaimissa. SQLite on keveytensä vuoksi pääasiassa käytössä mobiilialustoilla. (sqlite.org, luettu 2016.)

SQLite ei varastoi tietoa palvelimelle, vaan tallentaa tiedot suoraan kovalevylle erillisiin tiedostoihin. Tämän ansiosta SQLite-tietokantoja ei tarvitse ylläpitää erikseen, vaan jokainen ohjelmisto, jossa tietokantaa käytetään, voi luoda ja käyttää omaa tietokantaansa ilman ulkoista ylläpitoa. Tämä on havainnollistettu kuviossa 3. Tällä tavoin toteutettu tietokanta on myös hyvin suojattu mahdollisilta ohjelmistojen virheiltä, eikä tieto katoa yhtä helposti koneen kaatuessa. Käyttäjän ei tarvitse erikseen asentaa SQLiteä, vaan se on käyttövalmis heti kun kirjastot on siirretty käyttäjälle. SQLite käyttää karsittua versiota SQL-syntaksista. SQL-syntaksista esimerkiksi komennot RIGHT OUTER JOIN ja ALTER TABLE DROP COLUMN eivät ole käytössä. (sqlite.org, luettu 2016.)



Kuvio 3 tietokannat palvelimella ja ilman havainnollistettuna

SQLiten asennetaan Unityllä toteutettuun peliin lataamalla SQLiten omalta sivustolta paketit, jotka sisältävät tarvittavat kirjastot. Tämän jälkeen ne tulee siirtää vastaaviin Unityn kansioihin. Jokaiseen luokkaan joka toimii SQLiten kanssa, tulee ladata kirjasto Mono.Data.SQLite.

3.1 Datatyypit

Datatyyppi kertoo, millaisessa muodossa tallennettava tieto on. Esimerkiksi onko tieto luku, teksti, tai esimerkiksi lista useita lukuja. SQLite antaa tallentaa tyhjiä (engl. null), kokonaislukuja (engl. integer), liukulukuja (engl. float, SQL-syntaksissa real), tekstiä (engl. text) ja suuria binääridatatyyppejä (engl. blob). (sqlite.org, luettu 2016.)

Tallennettaessa tietoa SQLite-tietokantaan C#-kieltä käyttäen, luodaan malli vastaamaan jokaista taulukkoa tietokannassa. Tällä tavoin esimerkiksi Dapper osaa siirtää pelissä käsiteltävät tiedot oikeassa muodossa taulukkoon.

Opinnäytetyön tietokantoihin tallennettavista tiedoista suurin osa on kokonaislukuja tai tekstiä, mutta osa tiedoista on muodossa, jota ei voi suoraan tallentaa SQLite-tietokantaan. Näitä varten tallennettaviin malleihin on luotu metodit tietotyypin muuntamiseen. Esimerkiksi Enumeraatio-tyyppiset tietueet tulee muuntaa SQLiteen sopivaan muotoon, joka on tässä tapauksessa kokonaisluku.

```
wm.RoleInt = (int) wm.Role;
wm.TribeInt = (int) wm.Tribe;
wm.GenderInt = (int) wm.Gender;
```

Koodiesimerkki 4 enumeraatioiden muuntaminen kokonaisluvuiksi

Koodiesimerkissä 4 syötetään mallissa wm sijaitseviin kokonaislukutietueisiin vastaavat enumeraatiot muunnettuna kokonaisluvuiksi. Nämä kokonaisluvut voidaan nyt siirtää SQLiteen, ja tietoa haettaessa ne tulee jälleen muuntaa Enumeraatioiksi koodiesimerkin 5 mukaisesti.

```
Tribe = (Enumerations.Tribes) TribeInt;
Gender = (Enumerations.Genders) GenderInt;
Role = (Enumerations.Roles) RoleInt;
```

Koodiesimerkki 5 kokonaislukujen muuntaminen enumeraatioiksi

3.2 SQLite-viitekehyksen alustaminen

Ensimmäinen vaihe tietokantaviitekehyksen luomisessa on sen perusasetusten ja viittausten määrittely. MVCS-mallin mukaan SQLite-viitekehys on malliltaan palvelu (engl. service), sillä se on vuorovaikutuksessa ohjelmiston ulkopuolisten osa-alueiden kanssa. MVCS-mallista kerrotaan luvussa 4.

Ohessa koodiesimerkissä 6 esitellään SQLiteService-viitekehyksen alustaminen. Aluksi luodaan skripti sopivaan kansioon ja nimetään se “SQLiteService” nimeämiskäytännön mukaisesti. SQLiteService perii myös rajapinnan ISQLiteService. Seuraavaksi skriptiin tulee lisätä tietokantaviitekehykseen tarvittavat kirjastot, jotka kirjoitetaan skriptin alkuun käyttämällä using-avainsanaa. Tarvittavat kirjastot ovat Dapper ja Mono.Data.Sqlite.

Viitekehyksen alustamiselle oleelliset metodit ovat ”DbFile”, ”SetDataFile” ja ”DbConnection”. SQLite tallentaa halutut tiedot projektista erilliseen tiedostoon. Metodi DbFile

hakee kutsuttaessa tietokannan tallennussijainnista, joka tässä tapauksessa on assets-kansion alta streamingAssets/gamedata.

Muuttuja _datafile sisältää nimen, jolla tietokanta halutaan hakea. _dataFillelle määritellään oletusarvo, joka on pelin aikana väliaikaisesti käytettävän tietokannan nimi. Tätä tietokantaa tullaan käyttämään pelin aikana tietojen siirtämiseen ja manipuloimiseen, eikä pelaaja voi nimetä sitä itse. Alustavasti _dataFilen oletusarvoksi on annettu ”WarriorDatabase”.

```
    public class SQliteService : ISQliteService
    {

        private string _datafile = "WarriorDatabase";

        public void SetDataFile(string temp)
        {
            _datafile = temp;
        }

        private string DbFile
        {
            get { return Application.dataPath + "/StreamingAssets/GameData/" + _datafile +
".sqlite"; }
        }

        private SqliteConnection DbConnection()
        {
            return new SqliteConnection("Data Source=" + DbFile);
        }
```

Koodiesimerkki 6 SQLiteServicen alustamiseen tarvittavat metodit

Kutsumalla SetDataFile -metodia määritetään muuttuja _datafile. Tietokannan nimeäminen erillisen metodin avulla toteutetaan siksi, että pelaaja voi pelatessaan tallentaa useamman eri tallennustiedoston eri nimillä ja myöhemmin ladata ne nimen mukaisesti. Esimerkki: mikäli halutaan käyttää tietokantaa nimeltä Tietokanta, käyttäjä kutsuu metodia SetDataFile ja antaa sen parametriksi string-muuttujan “Tietokanta”, jolloin ohjelma osaa hakea vastaavan nimisen tietokannan yllä olevasta sijainnista.

Metodi “DbConnection” palauttaa SQLite-yhteyden yllä määriteltyyn tietokantaan aina kun sitä kutsutaan. Metodia voidaan myöhemmin kutsua kaikissa tapauksissa, kun halutaan muodostaa yhteys tietokantaan.

3.3 Dapper

Dapper C#-kirjasto, jonka tarkoitus on helpottaa C#:n ja SQLiten välistä vuorovaikutusta. Se vähentää kutsuttavien komentojen määrää ja yksinkertaistaa ohjelmiston rakennetta. Mikäli SQLite-tietokantaa käyttäisi ilman Dapperia, ohjelmistossa joutuisi toistamaan monia asioita useasti, jotka Dapper tiivistää muutamaan riviin. (Maskalik S. luettu 2016.) Alaluvussa tutkitaan SQLite-toimintojen rakennetta ilman Dapperia ja sen kanssa.

3.3.1 Tiedon hallinta ilman Dapperia

Luodaan TestModel-luokka, kuten koodiesimerkissä 7, joka sisältää lukutietueen Id sekä tekstitietueen Test. Nämä tietueet kuvastavat tietoja, joita halutaan tallentaa tietokantaan. Koodiesimerkissä 8 tietokantaan luodaan taulukko ThesisTable, johon on asetettu tietueiksi vastaavat Id ja Test.

```
public class TestModel
{
    public int Id { get; set; }
    public string Test { get; set; }
}
```

Koodiesimerkki 7 TestModel-malli

```
    public void CreateThesisTable()
    {
        using (var cnn = DbConnection())
        {
            var sql = "CREATE TABLE IF NOT EXISTS ThesisTable('Id'
         integer primary key, 'Test' string)";
            cnn.Open();
            var command = new SqliteCommand(sql, cnn);
            command.ExecuteNonQuery();
            cnn.Close();
        }
    }
```

Koodiesimerkki 8 tietokannan luominen

```
public void SetTestData(TestModel test)
{
    using (var cnn = DbConnection())
    {
        var sql =
        "INSERT INTO ThesisTable
        (Id, Test)
        VALUES(" + test.Id + ", '" + test.Test + "')";
        cnn.Open();
        var command = new SqliteCommand(sql, cnn);
        command.ExecuteNonQuery();
        cnn.Close();
    }
}
```

Koodiesimerkki 9 SetTestData-metodi, tiedon syöttäminen tietokantaan

Koodiesimerkin 9 metodi SetTestData asettaa kutsuttaessa tietokannassa sijaitsevaan ThesisTableen arvoja, jotka metodin kutsumiseen vaadittavaan TestModeliin on syötetty. Ensimmäiseksi metodiin luodaan tekstimuuttuja, joka sisältää SQL-syntaksilla kirjoitetun koodin. Tähän muuttujaan sijoitetaan ne tiedot metodiin syötetystä TestModelista, jotka taulukkoon halutaan syöttää. SQL-syntaksi erotellaan muusta koodista luettavuuden helpottamiseksi. Tämän jälkeen avataan tietokanta ja luodaan uusi SQLite-komento, jolle annetaan parametreiksi juuri luotu SQL-tekstimuuttuja ja tietokantayhteys cnn. Komennon luomisen jälkeen se lähetetään komennolla ExecuteNonQuery. Lopuksi suljetaan tietokantayhteys.

```
public TestModel GetTestData(int Id)
{
    using (var cnn = DbConnection())
    {
        var result = new TestModel();
        var sql = "SELECT Test FROM ThesisTable WHERE Id = " + Id + "";
        cnn.Open();
        var command = new SqliteCommand(sql, cnn);
        result.Test = (string) command.ExecuteScalar();
        cnn.Close();
        return result;
    }
}
```

Koodiesimerkki 10 GetTestData-metodi, tiedon palauttaminen tietokannasta

Koodiesimerkin 10 metodi GetTestData palauttaa syötetyn id:n mukaisen TestModelin ThesisTablesta. Luodaan palautettavan TestModel-mallin ilmentymä result. Tämän jälkeen kirjoitetaan jälleen tekstimuuttujaan SQL-syntaksia noudattava koodi, johon sijoitetaan se tietue, jonka mukaan tieto haetaan. Seuraavaksi avataan tietokantayhteys ja luodaan SQLite-komento. Palautettavaan TestModeliin halutaan hakea kaikki tietokantaan tallennetut tiedot Id:n mukaiselta riviltä. ThesisTable sisältää esimerkissä vain kaksi tietuetta, joten ainoana sijoitettava muuttujana on tekstitietue Test. Suoritettavassa SQLite-komennossa valitaan ThesisTablesta syötettyä Id-tietuetta vastaava Test-tietue. Kun

SQLite-komento suoritetaan, se siirretään suoraan TestModelin Test-tietueeseen muuntamalla saatu ExecuteScalar-tulos String-formaattiin.

Mikäli tietueita olisi useampi, tätä metodia käyttäen ne pitäisi kaikki siirtää palautettavaan malliin yksi kerrallaan, jokainen omalla SQLite-kyselyllään kuten koodiesimerkistä 11 ilmenee. Challengers of Khalea sisältää paljon hyvin laajoja malleja, joten tästä syystä projektiin implementoidaan Dapper yksinkertaistamaan tulosten hakemista.

```
cnn.Open();
var result = new TestModel();
var sql = "SELECT Test FROM ThesisTable WHERE Id = " + Id + "";
var command = new SqliteCommand(sql, cnn);
result.Test = (string) command.ExecuteScalar();
sql = "SELECT Test1 FROM ThesisTable WHERE Id = " + Id + "";
command = new SqliteCommand(sql, cnn);
result.Test1 = (string)command.ExecuteScalar();
sql = "SELECT Test2 FROM ThesisTable WHERE Id = " + Id + "";
command = new SqliteCommand(sql, cnn);
result.Test2 = (string)command.ExecuteScalar();

        . . .
```

Koodiesimerkki 11 esimerkki laajasta tietokannasta tietoa hakevasta metodista

3.3.2 Dapper-implementointi

Dapper toimii tietoa syöttäessä hyvin vastaavasti kuin puhtaasti C#-SQLite-implementaatio. Tietokantayhteyden avaamisen jälkeen suoritetaan Dapper-komento Execute, joka vastaa SQLite-kirjaston ExecuteNonQuery –komentoa. Koodiesimerkissä 12 myös SQL-syntaksin sisältänyt tekstimuuttuja on tiivistetty Execute-komennon sisään, sillä komentoja on kokonaisuudessaan vähemmän.

```
public void SetTestData(TestModel test)
{
    using (var cnn = DbConnection())
    {
        cnn.Open();
        cnn.Execute(
            @"INSERT INTO ThesisTable
        (Id,
        Test) VALUES
        (@Id,
        @Test);"
            , test);
        cnn.Close();
    }
}
```

Koodiesimerkki 12 tiedon syöttö Dapperin avulla

Kuten koodiesimerkistä 12 ilmenee, sen sijaan että tekstimuuttujan sisään sijoitettaisiin muuttujina syötettävät arvot, ne merkitään @-merkillä, jolloin Dapper osaa hakea syötettävästä mallista vastaavat muuttujat ja sijoittaa ne SQL-syntaksiin.

Tietoa haettaessa Dapper toimii lyhyemmin ja yksinkertaisemmin kuin puhdas C#-SQLite-syntaksi. Palautettava TestModel luodaan suoraan sisältämään SQLite-kyselyllä haetut tiedot. Dapper osaa hakea kyselyn mukaiset tiedot ja sijoittaa ne malliin, mikäli tietueet ovat molemmissa identtisesti nimetyt.

```
public TestModel GetTestData(int id)
{

    using (var cnn = DbConnection())
    {
        cnn.Open();
        var result = cnn.Query<TestModel>(
            @"SELECT Id,
                Test
            FROM ThesisTable
            WHERE id = @Id", new {id}).FirstOrDefault();
        cnn.Close();
        return result;
    }
}
```

Koodiesimerkki 13 tiedon hakeminen Dapperin avulla

Select-komennolla haetaan tiedot samalla tavalla id:n mukaisesti kuin aiemmassa esimerkissä, mutta se osaa syöttää tiedot suoraan annettuun malliin. Koodiesimerkissä 13 kyselylle annetaan parametriksi malli, johon tiedot syötetään, tässä tapauksessa TestModel. Lopuksi SQL-syntaksin jälkeen valitaan ensimmäinen sopiva tulos hakutuloksista, joka syötetään palautettavaan malliin.

Laajempia malleja käyttäessä Dapper ei vaadi useampien kyselyjen suorittamista eikä useampia SQL-syntaksin tekstimuuttujia ja tietojen muuttuessa tai lisääntyessä ne voidaan lisätä aiemman tekstin perään. Tämä malli vähentää siis komentojen ja tekstirivien määrää koodiesimerkin 14 tapaisesti.

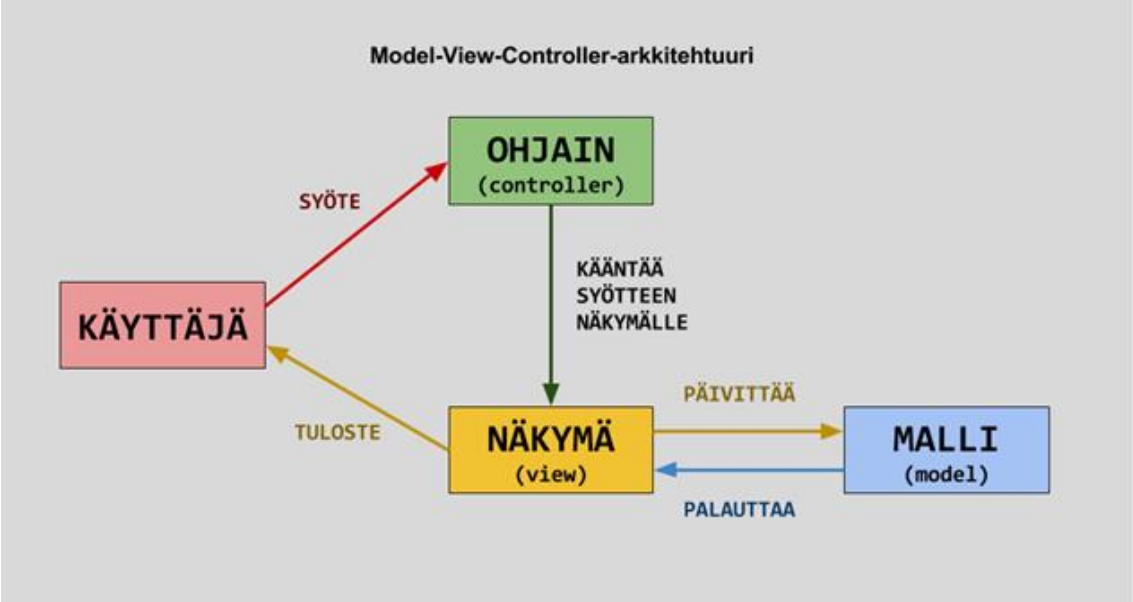
```
cnn.Open();
var result = cnn.Query<TestModel>(
    @"SELECT Id,
        Test,
        Test1,
        Test2
    FROM ThesisTable
    WHERE id = @Id", new {id}).FirstOrDefault();
cnn.Close();
return result;
```

Koodiesimerkki 14 esimerkki laajasta tietokannasta tietoa hakevasta metodista Dapperin avulla

4 OHJELMISTON ARKKITEHTUURI

4.1 MVC

MVC-arkkitehtuuri on norjalaisen Trygve Reenskaugin luoma arkkitehtuurirakenne, jossa noudatetaan kolmen eri osa-alueen vuorovaikutusta, jotka ovat malli (engl. model), näkymä (engl. view), ohjain (engl. controller). Kuviossa 15 havainnollistetaan Reenskaugin alkuperäisen MVC-arkkitehtuurin toimintaa. Malli sisältää ohjelmiston tiedot, josta näkymä hakee tarvittavat tiedot käyttäjän nähtäväksi ja muokkaa mallia käyttäjän syötteen mukaan. Käyttäjä havaitsee tulosteen, jonka näkymä tarjoaa, ja ohjaa sovellusta sen mukaan syöttämällä komennot ohjaimelle, joka puolestaan muokkaa syötteestä näkymälle käyttökelpoisia komentoja. (Reenskaug T. luettu 2016.)



Kuvio 15 Trygve Reenskaugin mukainen malli MVC-arkkitehtuurista

MVC-arkkitehtuuria sovelletaan käyttötarkoituksista riippuen eri tavoilla. Monet kehittäjät ovat poikenneet Reenskaugin luomasta mallista ja määrittäneet erilaiset suhteet MVC-komponenteille. Esimerkiksi joissain lähestymistavoissa ohjain on vastuussa mallin ja näkymän välisestä tiedonsiirrosta, jolloin ne eivät ole kytköksissä toisiinsa, kuten Reenskaug on kuvaillut. On myös tapauksia, joissa käyttäjä ei annan syötteitä ohjaimelle vaan suoraan näkymään. Ohjain kuuntelee näkymää ja kääntää käyttäjän syötteet mallille, joka taas palauttaa tietoa takaisin ohjaimelle.

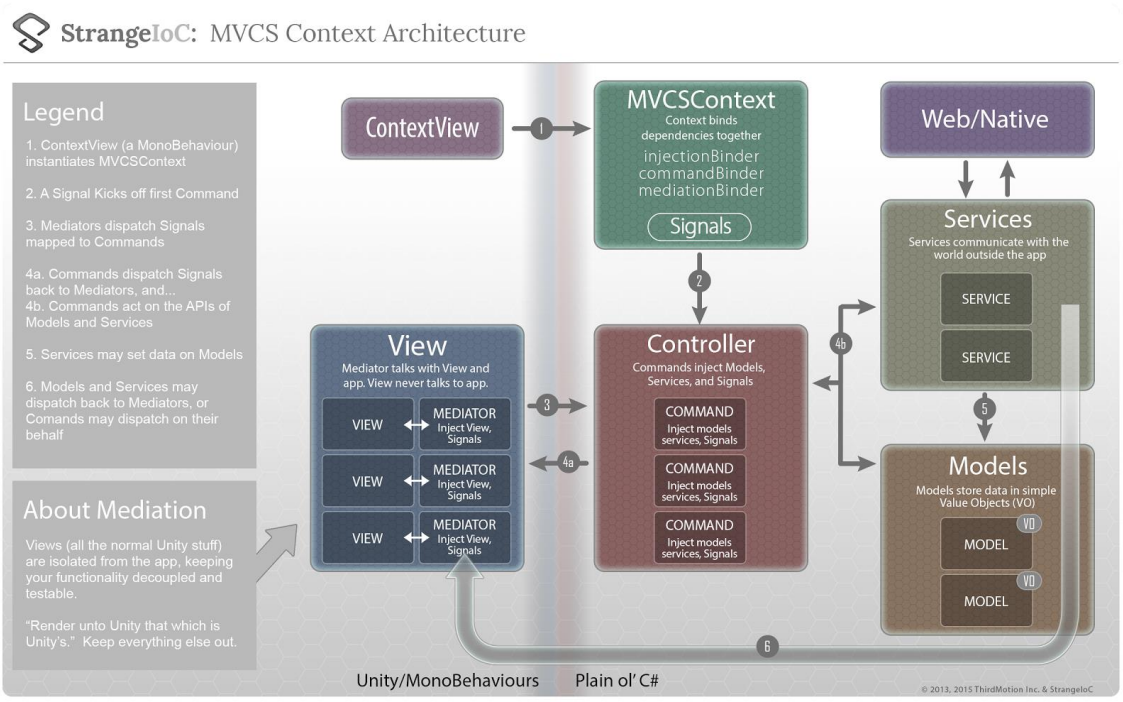
4.2 Ioc

Hallinnan muutossuunnan vaihtaminen (engl. Inversion of Control, IoC) on suunnitteluperiaate, jonka avulla saadaan vähennettyä ohjelmiston luokkien välisiä riippuvuuksia. Tähän periaatteeseen viitataan usein leikkimielisesti ”Hollywood”-periaatteella: ”Älä soita meille, me soitamme sinulle”. (Fowler, M. 2005) StrangeIoC sisältää hallinnan muutossuunnan vaihtamista varten riippuvuusinjektoinnin (engl. dependency injection). Riippuvuusinjektiolla oliolle annetaan (injektoidaan) sen ilmentymän (engl. instance) muuttujat sen sijaan, että olio määrittäisi ne itse. Kaikki edellä mainitut osa-alueet ovat modulaarisia, eli ne eivät ole riippuvaisia toisistaan. Strange Ioc:n tarkoitus onkin jakaa projekti itsenäisiin osa-alueisiin, joita voidaan työstää samanaikaisesti vaikuttamatta muihin osa-alueisiin. Tällä tavoin toteutettuja luokkia on myös helppo käyttää uudelleen muissa projekteissa.

4.3 StrangeIoC

Challengers of Khalea -projektin kasvaessa suuremmaksi, StrangeIoc valittiin projektin arkkitehtuurimalliksi. StrangeIoC on hallinnan muutossuunnan vaihtamiseen tarkoitettu viitekehys, joka on suunnattu C#-ohjelmointikielelle ja Unity-pelikehitysalustalle. Sen kehittäjä on ThirdMotion, Inc. ja se perustuu pitkälti ActionScript 3.0:lle kehitettyyn avoimen lähdekoodin Robotlegs-viitekehykseen. StrangeIoC ei kuitenkaan ole sovitus Robotlegsistä, mutta sen piirteitä on käytetty runsaasti StrangeIoC:ssä. StrangeIoC sisältää myös valinnaisen MVCS (Model-View-Controller-Service) arkkitehtuurirakenteen. StrangeIoC:n käyttäjä voi valita sen osa-alueista kaikki tai osan käytettäväksi projektissa. (ThirdMotion, Inc & StrangeIoC 2016.)

StrangeIoC:n käyttämä MVCS-malli toimii muutoin samalla tavalla kuin tavallinen MVC, mutta siihen on lisätty palvelu (engl. service) -osa-alue. Palveluiden tarkoitus on toimia ohjelmiston ulkopuolisten lähteiden kanssa. StrangeIoC:n MVCS-mallissa on myös erotettu näkymät välittäjillä (engl. mediator) ja tieto eri osien välillä kuljetetaan signaalien avulla. (ThirdMotion, Inc & StrangeIoC 2016.) Kuviossa 16 kuvataan StrangeIoc:n toimintaa. Tämän opinnäytetyön kannalta oleellisin luokka on SQLiteService, joka kuuluu kaaviossa havainnollistettuun palvelut (engl. services) -osioon, sillä se toimii muun ohjelmiston ulkopuolisen tiedon välittäjänä.



Kuvio 16 StrangeIoC:n toiminta kaaviona. Kuva suurennettuna liitteessä 1

StrangeIoc perustuu luokkien sitomiseen kontekstin avulla (engl. binding), niiden kutsumiseen muissa luokissa ”injektoimalla” ja näiden avulla kaikkien projektin ohjelmiston osa-alueiden riippuvuuksien vähentämiseen. Kun luokka sidotaan toiseen, jälkimmäistä injektoitaessa MVCS-konteksti palauttaa ilmentymän sidotusta luokasta. Yleisimmin StrangeIoC:ta käytettäessä sidotaan malleja rajapintoihin, näkymiä välittäjiin tai komentoja signaaleihin. (ThirdMotion, Inc & StrangeIoC 2016.)

Tässä opinnäytetyössä luotava SQLiteService sidotaan StrangeIoC:n kontekstin avulla rajapintaan ISQLiteService jota kutsutaan muissa luokissa injektoimalla. Tällä tavoin SQLiteServiceä voidaan käyttää ilman että käyttäjän tarvitsee tietää sen toiminnasta. Nämä ominaisuudet on havainnollistettu koodiesimerkissä 17.

```
injectionBinder.Bind<ISQliteService>().To<SQliteService>();

[Inject]
public ISQliteService SQlite { get; set; }
```

Koodiesimerkki 17 SQLiteService sidotaan rajapintaan ISQLiteService ja esimerkki sen injektoinnista

5 ENSIMMÄISET TAULUKOT

5.1 StrangeWarriorModel

Pelissä oleellisin tallennettava yksikkö on nimeltään soturi (engl. warrior). Ensimmäisenä luodaan taulukko, joka tallentaa sotureita myöhempää latausta varten. Jokainen soturi saa oman ilmentymänsä mallin "StrangeWarriorModel" mukaan. StrangeWarriorModel-malli sisältää jokaisen soturin kaikki yksilölliset tiedot omissa muuttujissaan. Tallennettavia tietoja ovat esimerkiksi soturin tunnistamiseen tarkoitettu Id-numero, soturin nimi, soturin attribuutit, esimerkiksi Strength ja Agility sekä soturin nykyinen sijainti ruudukossa. StrangeWarriorModelin sisäiset tiedot on myös jaettu kahteen sisäiseen malliin: PersistentWarriorModel ja TemporaryWarriorModel.

PersistentWarriorModel sisältää kaikki sellaiset tiedot, jotka pysyvät sotureilla läpi pelin, kun taas TemporaryWarriorModel sisältää yksittäisille taisteluille olennaiset tiedot, esimerkiksi soturin sijainnin ruudukossa tai soturin sen hetkiset elämäpisteet. StrangeWarriorModel on jaettu näihin osiin, jotta SQLite voi lukea aina vain haluttavat tiedot ja ohittaa ne tiedot joita ei tarvita. SQLiten malleja palauttavat komennot eivät myöskään osaa toimia sellaisten mallien kanssa jotka sisältävät Constructoreita, joten tällainen jako antoi enemmän vapautta StrangeWarriorModelin suunnittelun. Ensimmäisenä luotava taulukko tulee tallentamaan StrangeWarriorModelin PersistentWarriorModel-osuuden. Koodiesimerkissä 18 esitellään PersistentWarriorModelin 12 ensimmäistä tietuetta.

```
public class PersistentWarriorModel : IPersistentWarriorModel
{

    public int Id { get; set; }
    public int AuctionHouseId { get; set; }
    public int TeamId { get; set; }

    public string Name { get; set; }
    public string Nickname { get; set; }
    public byte Age { get; set; }

    public Enumerations.Genders Gender { get; set; }
    public int GenderInt { get; set; }
    public Enumerations.Tribes Tribe { get; set; }
    public int TribeInt { get; set; }
    public Enumerations.Roles Role { get; set; }
    public int RoleInt { get; set; }
```

Koodiesimerkki 18 PersistentWarriorModel-luokka

5.2 Taulukon luominen

Taulukon muodostamiseksi luodaan metodi CreateWarriorRepo. Kun metodia kutsutaan, se luo tietokantatiedostoon taulukon nimeltä Warriortable, johon voidaan syöttää ilmentymiä StrangeWarriorModel-mallista.

Ensimmäiseksi taulukon luomisessa tarkistetaan, onko annetun nimistä taulukkoa olemassa etukäteen, tässä tapauksessa nimeltään WarriorTable. Mikäli taulukko on olemassa, se poistetaan. Tämän jälkeen Taulukko luodaan komennolla Create Table.

Create Table -komentoon tulee syöttää jokainen tietue, joka halutaan tallentaa tietokantaan. Koodiesimerkissä 19 näkyvät 7 ensimmäistä tietuetta, mitkä luodaan warriortable-taulukkoon. Jokaisella rivillä annetaan ensimmäiseksi tietueelle nimi ja toiseksi tallennettava datatyyppi. Muita näkyviä parametrejä ovat primary key, joka kertoo tietueen olevan yksilöllinen avainarvo, jonka mukaan tallennetut soturit järjestetään taulukossa ja not null, joka estää tyhjän arvon antamisen tietueelle.

```
public void CreateWarriorRepo()
{
    using (var cnn = DbConnection())
    {
        cnn.Open();
        cnn.Execute(
        @"drop table if exists WarriorTable", null);
        cnn.Execute(
            @"create table WarriorTable

                  (
                     'Id' integer primary key,
                     'Name' string not null,
                     'TeamId' integer,
                     'Age' integer,
                     'GenderInt' integer,
                     'RoleInt' integer
                  )
              ", null);
        cnn.Close();
    }
}
```

Koodiesimerkki 19 WarriorTable-taulukon luominen

C#-kielessä jokaiselle SQL-komennolle tulee antaa parametriksi datamalli, josta tiedot siirretään tietokantaan. Taulukon luonnissa mallia ei tarvita, sillä tietoja ei vielä luonnin aikana siirretä. Annetaan komennolle “Execute” toiseksi parametriksi null. Lopuksi suljetaan tietokantayhteys, ettei se vie muistia turhaan.

5.3 Taulukon Käyttö

Taulukon luomisen lisäksi luodaan metodit, joilla taulukkoa muokataan ja luetaan. Seuraavia metodeja kutsumalla vaikutetaan taulukon sisältöön tai luetaan taulukon sisältämää tietoa. Syötettävänä ja palautettavana mallina toimii StrangeWarriorModel.

5.3.1 Tallentaminen

Luodaan metodi SetWarriorSQlite, jolla taulukkoon “WarriorTable” luodaan uusi soturi. Kun metodia kutsutaan, siihen syötetään parametriksi StrangeWarriorModel, josta metodi lukee PersistentWarriorModel-osuuden, sillä se sisältää kaikki ne tiedot, jotka halutaan tallentaa taulukkoon. Tämän jälkeen metodi syöttää kaikki luetut tiedot WarriorTable-taulukkoon.

StrangeWarriorModel sisältää tietueita, joita ei voi siirtää suoraan SQLite-tietokantaan. Näihin kuuluvat enumeraatiot Role, Tribe ja Gender, sekä erilliset mallit Ability1, Ability2, Ability3, Ability4, CaptainAbility ja InjuryModel. Koodiesimerkissä 20 enumeraatiot muunnetaan kokonaisluvuiksi ja siirretään niille varattuihin tietueisiin, kun taas jokaisen mallin kohdalla tarkistetaan niiden olemassaolo, valitaan mallin Id ja siirretään saatu Id sille varattuun kokonaislukutietueeseen.

```
public void SetWarriorSQlite(StrangeWarriorModel warru)
        {
            var wm = warru.Persistent;
            wm.RoleInt = (int) wm.Role;
            wm.TribeInt = (int) wm.Tribe;
            wm.GenderInt = (int) wm.Gender;
            if (wm.Ability1 != null)
            {
                wm.Ability1Id = wm.Ability1.AbilityId;
            }
            if (wm.Ability2 != null)
            {
                wm.Ability2Id = wm.Ability2.AbilityId;
            }
            if (wm.Ability3 != null)
            {
                wm.Ability3Id = wm.Ability3.AbilityId;
            }
            if (wm.Ability4 != null)
            {
                wm.Ability4Id = wm.Ability4.AbilityId;
            }
            if (wm.CaptainAbility != null)
            {
                wm.CaptainAbilityId = wm.CaptainAbility.AbilityId;
            }
            if (warru.InjuryModel != null)
            {
                wm.InjuryId = warru.InjuryModel.ID;
                wm.InjuryCurrentLength = warru.InjuryModel.CurrentLength;
            }
```

Koodiesimerkki 20 StrangeWarriorModelin muuttujien tarvittavat datatyyppien muunnokset

Kun kaikki muuttujat ovat oikeassa muodossaan, metodiin syötetty malli voidaan siirtää SQLite-taulukkoon. Annetaan Execute-komennolle SQL-komennoksi insert into ja valitaan aiemmin luotu WarriorTable taulukoksi, johon tiedot syötetään. Koodiesimerkissä 21 näytetään 6:n ensimmäisen tietueen syöttäminen. Dapperin mukaisesti tietueet luetellaan samassa järjestyksessä Insert into-, sekä values –osioissa. Values-osion tietueet on merkattu @-merkillä, sillä ne ovat mallista tuotavat muuttujat. Execute-komennolle annetaan parametriksi syötettävä malli, joka on tässä tapauksessa aiemmin määritetty PersistentWarriorModel wm.

```
        using (var cnn = DbConnection())
        {
            cnn.Open();
            cnn.Execute(
                @"INSERT INTO WarriorTable
                (   Id,
                    Name,
                    TeamId,
                    Age,
                    GenderInt,
                    RoleInt
                        )
                VALUES(
                    @Id,
                    @Name,
                    @TeamId,
                    @Age,
                    @GenderInt,
                    @RoleInt
                    );
                ", wm);
            cnn.Close();
        }
    }
```

Koodiesimerkki 21 tiedon syöttäminen WarriorTable-taulukkoon

5.3.2 Lataaminen

Metodi GetWarriorData palauttaa kutsuttaessa syötetyn Id:n mukaisen StrangeWarrior-Modelin SQLite-tietokannan WarriorTable-taulukosta. Ennen SQL-komentoja tarkastetaan tietokannan olemassaolo ja keskeytetään metodi, mikäli tietokantaa ei löydy. Tämän jälkeen avataan tietokantayhteys ja suoritetaan SQL-kysely. Kyselyn suorittamiseksi käytetään Query-komentoa, jolle annetaan parametriksi palautettava PersistentWarriorModel.

Kyselyn alle luodaan SQL-syntaksin mukainen select-komento, jolla haetaan WarriorTablesta kaikki StrangeWarriorModelin aiemmin tallennetut tiedot. Koodiesimerkissä 22 näytetään taulukon 6 ensimmäistä tietuetta. Tiedot haetaan metodiin syötetyn Id:n mukaan ja kysely palauttaa PersistentWarriorModel-muodossa ensimmäisen annettua Id:tä vastaavan rivin taulukosta.

```
public StrangeWarriorModel GetWarriorData(int id)
{

    if (!File.Exists(DbFile)) return null;

    using (var cnn = DbConnection())
    {
        cnn.Open();
        var result = cnn.Query<PersistentWarriorModel>(
            @"SELECT Id,
                Name,
                TeamId,
                Age,
                GenderInt,
                RoleInt

            FROM WarriorTable
            WHERE id = @Id", new { id }).FirstOrDefault();
        cnn.Close();
        return new StrangeWarriorModel {Persistent = result};
    }
}
```

Koodiesimerkki 22 tiedon hakeminen WarriorTable-taulukosta

Yksittäisen warriorin lataamisen lisäksi käyttäjä saattaa haluta ladata myös kaikki taulukosta löytyvät soturit, joten luodaan koodiesimerkin 23 tapaan metodi GetAllWarriors. Metodi toimii lähes samalla tavalla kuin GetWarriorData. Eroavaisuutena on, että se palauttaa listan yksittäisen mallin sijaan ja GetWarriorDatan FirstOrDefault on korvattu ToList-parametrillä. Palautetut soturit käydään läpi ForEach-komennolla, jotta ne voidaan muuntaa StrangeWarriorModel-muotoon.

```
public List<StrangeWarriorModel> GetAllWarriors()
{
            if (!File.Exists(DbFile)) return null;

            using (var cnn = DbConnection())
            {

            cnn.Open();
            var list = cnn.Query<PersistentWarriorModel>(
                  @"SELECT Id,
                       Name,
                       TeamId,
                       Age,
                       GenderInt,
                       RoleInt,

                   FROM WarriorTable"
                   , null).ToList();
                        var result = new List<StrangeWarriorModel>();
                        foreach (var warru in list)
                        {
                           var wm = new StrangeWarriorModel {Persistent = warru};
                           result.Add(wm);
                        }
                        cnn.Close();
                        return result;
            }
}
```

Koodiesimerkki 23 kaikkien warriorien hakeminen WarriorTable-taulukosta

Näiden metodien avulla pelin jokainen soturi voidaan tallentaa ja ladata metodeja kutsuttaessa. Liitteessä 1 näkyvät kaikki talletettavat tietueet.

5.4 Muut taulukot

Tietokannan rakennetaulukossa esitellyt taulukot luodaan samalla tavoin kuin WarriorTable ja niille luodaan vastaavanlaiset muokkausmetodit. Näistä tärkeimpiä ovat TeamTable, EquipmentTable ja ProfileTable.

5.5 Esimerkki tietokannasta ja sen toiminnasta

Kuviossa 24 näytetään esimerkki WarriorTablesta, johon on syötetty sisältöä. Kuvan näkymä on DbBrowser-tietokantaohjelman ikkuna.

	Id	Name	TeamId	Age	GenderInt	RoleInt
	Filter	Filter	Filter	Filter	Filter	Filter
1	31	Fishy Fish	1	21	4	1
2	32	Tuna Eel	1	18	4	2
3	33	Salmon Snake	1	20	4	4
4	34	Piranha Bob	1	22	4	8
5	35	John	0	24	4	64
6	36	Pig	0	21	2	2
7	37	Pork	0	18	2	1
8	38	Captain Bacon	1	23	2	4

Kuvio 24 esimerkki täytetyn tietokannan sisällöstä

Kuvassa näkyvät GenderInt ja RoleInt ovat lukuarvoja jotka on saatu muuntamalla enumeraatiot kokonaisluvuiksi. Kun yksittäistä soturia haetaan, metodiin GetWarriorData syötetään kokonaisluku Id, joka vastaa taulukossa näkyvää Id-arvoa. Palautettavassa StrangeWarriorModelissa kaikki arvot ovat edelleen pelkkiä SQLite-yhteensopivia datamuotoja, joten esimerkiksi kaikki enumeraatiot ja mallit tulee asettaa malliin SQLite-kyselyn ulkopuolella.

6 KOKONAISEN PELIN TALLENTAMINEN JA LATAAMINEN

Pelin koko progression tallentamiseen tarvitaan ensimmäiseksi palvelut, jotka tekevät kutsuttaessa kaikki tarvittavat datatyyppien muutokset tallennettavaan ja ladattavaan peliin. Tämän lisäksi SQLiteServiceen tulee luoda metodit kokonaisuuden tallettamista varten, kun nykyiset metodit tallentavat tietoa vain yksittäisiä tietoja valittuun tietokantaan. Lopuksi luodaan palvelu, jonka avulla voidaan suorittaa kutsuttaessa kaikki nämä komennot, luoda kokonaisia tallennustiedostoja, ja ladata tallennetut tiedostot käyttöön myöhemmin.

6.1 Wrapperit

Ensimmäiseksi luodaan wrapperit, eli luokat jotka tekevät kaikki datatyyppien muutokset ja palauttavat käyttäjälle käyttökelpoisia malleja niitä kutsuttaessa. Kun käyttäjä tahtoo luoda toiminnallisuutta mihin tarvitsee ladata esimerkiksi sotureita tai joukkueita, hänen ei tarvitse tietää muuta kuin nämä luokat ja kutsua niiden metodeja tarpeen tullen.

Kutsumalla WarriorLoaderServicen GetWrappedWarrior-metodia ja syöttämällä parametriksi halutun soturin id:n palvelu palauttaa halutun soturin muunnettuna varastoitavasta muodosta käytettävään muotoon. WarriorLoaderService on havainnollistettu koodiesimerkissä 25.

```
public class WarriorLoaderService : IWarriorLoaderService
    {
        [Inject]
        public ISQliteService SQlite { get; set; }

        [Inject]
        public IResourceProviderModel ResourceProvider { get; set; }

        [Inject]
        public IEquipmentStorageService EquipmentStorage { get; set; }

        public IWarriorModel GetWrappedWarrior(int id)
        {
            IWarriorModel warrior = SQlite.GetWarriorData(id);
            var tempTraits = SQlite.GetTraitStorage(id);
            warrior.Traits = ConvertTraits(tempTraits);
            warrior.IntIntoEnum();
            warrior.Ability1 = ResourceProvider.GetAbilityById(warrior.Ability1Id);
            warrior.Ability2 = ResourceProvider.GetAbilityById(warrior.Ability2Id);
            warrior.Ability3 = ResourceProvider.GetAbilityById(warrior.Ability3Id);
            warrior.Ability4 = ResourceProvider.GetAbilityById(warrior.Ability4Id);
```

Koodiesimerkki 25 WarriorLoaderService-luokka

Luokan toimintaan tarvittavat rajapinnat ovat SQLiteService, ResourceProviderModel ja EquipmentStorageService. SQLiteService on lähde, josta soturit haetaan ja siihen viitataan, kun ensimmäiseksi haetaan pyydetty soturi ja palautetaan se varastoidussa muodossa. ResourceProviderModel hakee Protobuf-netin avulla ulkoisista tiedoista soturille sellaiset tiedot, jotka eivät muutu vaan pysyvät samanlaisina läpi kaikkien pelikertojen. EquipmentStorageServicestä puolestaan haetaan tiedot soturin sen hetkisistä varusteista, jotta ne voidaan palauttaa pyydetyn soturin yhteydessä. Nämä rajapinnat injektoidaan StrangeIoC:n avulla WarriorLoaderServiceen, jotta niiden ilmentymiä voidaan käyttää.

TeamLoaderService ja EquipmentStorageService toimivat myös samalla tavoin. TeamLoaderService palauttaa annetun Id:n mukaisen joukkueen ja lataa siihen valmiiksi kaikki siihen kuuluvat soturit. EquipmentStorageService puolestaan tallentaa ja lataa varusteet oikeassa muodossaan. Palveluun syötetään ase, kilpi tai haarniska ja se tallennetaan yleisessä StrangeEquipmentModel-muodossa. Varustetta (engl. equipment) pyydettäessä palvelu hakee varusteen yleisessä muodossa ja osaa tämän jälkeen palauttaa oikean tyyppisen mallin, StrangeWeaponModel, StrangeShieldModel tai ArmorModel.

6.2 SQLite-toiminnot

SQLiteServiceen luodaan metodit SaveIntoFile (koodiesimerkki 26) ja LoadFromFile (koodiesimerkki 27). Näiden metodien tarkoitus on siirtää kaikki tiedot nykyisestä väliaikaisesta tietokannasta tallennettavaan tietokantaan ja palauttaa ne peliä ladatessa väliaikaiseen tietokantaan. Metodeihin syötetään parametriksi sen tietokantatiedoston nimi, johon tiedot halutaan tallentaa tai josta ne halutaan ladata.

```
public void SaveIntoFile(string fileName)
        {
            using (var cnn = DbConnection())
            {
                string code = Application.dataPath + "/StreamingAssets/GameData/"
+fileName+".sqlite";

                cnn.Open();
                cnn.Execute(@"ATTACH DATABASE '"+code+"' AS 'save';", null);
                cnn.Execute(@"
                            INSERT INTO save.WarriorTable
                            SELECT * FROM WarriorTable"
                     , null);
                cnn.Execute(@"
                            INSERT INTO save.EquipmentTable
                            SELECT * FROM EquipmentTable"
                    , null);
                cnn.Execute(@"
                            INSERT INTO save.TeamTable
                            SELECT * FROM TeamTable"
                    , null);
```

Koodiesimerkki 26 SaveIntoFile-metodi

```
        public void LoadFromFile(string fileName)
        {
            using (var cnn = DbConnection())
            {
                 string code = Application.dataPath +
             "/StreamingAssets/GameData/WarriorDatabase.sqlite";

                SetDataFile(fileName);
                cnn.Open();
                cnn.Execute(@"ATTACH DATABASE '"+code+"' AS 'load';", null);
                cnn.Execute(@"
                            INSERT INTO load.WarriorTable
                            SELECT * FROM WarriorTable"
                     , null);
                cnn.Execute(@"
                            INSERT INTO load.EquipmentTable
                            SELECT * FROM EquipmentTable"
                    , null);
                cnn.Execute(@"
                            INSERT INTO load.TeamTable
                            SELECT * FROM TeamTable"
                    , null);
```

Koodiesimerkki 27 LoadFromFile-metodi

6.3 Tallennusmetodit

Kaikkien SQLite-komentojen kokoamiseksi yhteen luodaan koodiesimerkin 28 mukainen luokka SaveAndLoadService. Se sisältää metodit, joita kutsumalla voidaan hoitaa kokonaisen pelitiedoston tallennus, eikä käyttäjän tarvitse kutsua muita luokkia sen toteuttamiseksi. Tarvittavat metodit ovat NewGame, SaveGame, LoadGame ja DeleteGame. SaveAndLoadService Injektoi ISQLiteServicen ja suorittaa metodeita kutsuttaessa kaikki tarvittavat SQLite-komennot ilman että käyttäjän tarvitsee tietää niiden olemassaolosta.

```
public class SaveAndLoadService : ISaveAndLoadService
    {
        [Inject]
        public ISQliteService SQlite { get; set; }

        public void NewGame(string fileName)
        {
            SQlite.SetDataFile(fileName);
            SQlite.CreateProfileRepo();
            SQlite.CreateTeamRepo();
            SQlite.CreateWarriorRepo();
            SQlite.CreateEquipmentStorage();
            SQlite.CreateIdStorage();
            SQlite.CreateStoryFlagTable();
            SQlite.CreateTraitTable();
            SQlite.CreateCoachRepo();
            SQlite.CreateStaffTable();
        }
```

Koodiesimerkki 28 SaveAndLoadService-palvelun ensimmäiset rivit ja NewGame-metodi

NewGame-metodiin syötetään parametriksi tallennustiedostolle haluttu nimi. Tämän jälkeen metodi luo kaikki tietokannan tarvitsemat taulukot. Itse tietokantaa ei tarvitse luoda erikseen, sillä SQLite osaa luoda puuttuvan tietokannan automaattisesti sitä kutsuttaessa.

```
public void SaveGame(string fileName)
{
    SQlite.SetDataFile("WarriorDataBase");
    SQlite.SaveIntoFile(fileName);
}
```

Koodiesimerkki 29 SaveGame-metodi

Koodiesimerkin 29 SaveGame-metodiin syötetään parametriksi aiemmin tallennetun tiedoston nimi, jonka jälkeen se kopioi pelin nykyisen tilanteen mukaisen väliaikaisen tietokannan (WarriorDataBase) tiedot syötettyyn tietokantaan, korvaten vanhat tiedot.

```
public void LoadGame(string fileName)
        {

            SQlite.SetDataFile("WarriorDatabase");
            SQlite.CreateProfileRepo();
            SQlite.CreateTeamRepo();
            SQlite.CreateWarriorRepo();
            SQlite.CreateEquipmentStorage();
            SQlite.CreateIdStorage();
            SQlite.CreateStoryFlagTable();
            SQlite.CreateTraitTable();
            SQlite.CreateCoachRepo();
            SQlite.CreateStaffTable();
            SQlite.SetDataFile(fileName);
            SQlite.LoadFromFile(fileName);
            SQlite.SetDataFile("WarriorDatabase");
        }
```

Koodiesimerkki 30 LoadGame-metodi

Koodiesimerkin 30 LoadGame-metodiin syötetään jälleen aiemmin luodun tallennustietokannan nimi ja luodaan väliaikainen tietokanta samalla tavoin kuin tallennustietokanta luotiin NewGame-Metodissa. Tämän jälkeen otetaan tiedot syötetystä tallennustietokannasta ja siirretään ne väliaikaiseen tietokantaan.

```
public void DeleteGame(string fileName)
{
    File.Delete("Assets/StreamingAssets/GameData/"+fileName+".sqlite");
}
```

Koodiesimerkki 31 DeleteGame-metodi

Koodiesimerkin 31 DeleteGame-Metodiin syötetään poistettavan tietokannan nimi. Tämän jälkeen metodi poistaa halutun tallennustiedoston tallennussijainnista.

6.4 Automaattinen tallennuskomento

Kaikki aiemmin luodut palvelut saadaan käyttöön luomalla koodiesimerkissä 32 esitelty AutoSaveCommand-komento, jota voidaan kutsua aina kun pelin tilanne tahdotaan tallentaa. Komentoon injektoidaan palvelut SaveAndLoadService ja SQLiteService sekä mallit ManagementClientModel ja StrangeResourceProviderModel. Injektoitaessa käytetään jälleen luokkien rajapintoja, esimerkiksi ISaveAndLoadService. Komennon Execute-metodia kutsutaan sillä hetkellä, kun komento ajetaan. Ensimmäiseksi AutoSaveCommand asettaa SQLiteServicen käsiteltäväksi tallennusmallin mukaisesti väliaikaisen tietokannan (WarriorDatabase), jotta siitä voidaan hakea tiedot tallennettavaksi. Tämän jälkeen haetaan SQLiteServicestä kaikki soturit ja niiden tarvittavat tiedot ResourceProviderModelista tallennusta varten, sillä kaikkia tietoja ei voida hakea suoraan SQLitestä datatyyppien vuoksi. Kun kaikki tiedot on haettu, jokainen soturi tallennetaan vielä kerran väliaikaiseen tietokantaan. Lopuksi peli tallennetaan korvaamalla profiiliin tallennetun pelitiedoston mukainen tietokantatiedosto väliaikaisella tietokannalla.

```
public class AutoSaveCommand : Command
    {
        [Inject]
        public ISaveAndLoadService SaveAndLoad { get; set; }

        [Inject]
        public ISQliteService SQlite { get; set; }

        [Inject]
        public IManagementClientModel ManagementClientModel { get; set; }

        [Inject]
        public IResourceProviderModel ResourceProvider { get; set; }

        public override void Execute()
        {
            SQlite.SetDataFile("WarriorDataBase");
             foreach (var warrior in
             ManagementClientModel.ProfileModel.PlayerTeam.GetAllWarriors())
            {
                warrior.IntIntoEnum();
                warrior.Ability1 = ResourceProvider.GetAbilityById(warrior.Ability1Id);
                warrior.Ability2 = ResourceProvider.GetAbilityById(warrior.Ability2Id);
                warrior.Ability3 = ResourceProvider.GetAbilityById(warrior.Ability3Id);
                warrior.Ability4 = ResourceProvider.GetAbilityById(warrior.Ability4Id);
                 warrior.CaptainAbility =
             ResourceProvider.GetAbilityById(warrior.CaptainAbilityId);
                if (warrior.EquippedWeapon != null)
                {
                    warrior.EquippedWeapon.WarriorId = warrior.Id;
                    warrior.EquippedWeapon.TeamId = warrior.TeamId;
                    IEquipmentModel castedModel = warrior.EquippedWeapon;
                    SQlite.ModifyEquipmentSQlite((StrangeEquipmentModel)castedModel);
                }
                SQlite.ModifyWarriorSQlite((StrangeWarriorModel)warrior);
            }
            SaveAndLoad.SaveGame(SQlite.GetProfileData(1).SaveName);
        }
    }
```

Koodiesimerkki 32 AutoSaveCommand-luokka

Komennon jälkeen aiemmin NewGame-komennolla luotu tietokantatiedosto on korvattu uusimmalla pelin tilanteen sisältävällä tietokannalla. Tämän tiedoston voi myöhemmin ladata käyttöä varten kutsumalla SaveAndLoadService-palvelun LoadGame-metodia ja syöttämällä parametriksi tallennetun tiedoston nimi.

7 POHDINTA

Kun vertailtiin SQLiten käyttöä ilman Dapperia ja sen kanssa, lopputulos osoitti, että Dapper vähentää koodin pituutta huomattavasti, jos kyseessä on vähänkin suurempi taulukko.

Modulaarisuuden lisääminen on helpottanut projektityöskentelyä. Projektin aiemmassa versiossa olemassa olevan ohjelmiston muuttaminen saattoi rikkoa monia muita projektin luokkia, joihin ei muuten olisi tarvinnut koskea. StrangeIoc:n implementoiminen lisäsi tarvittavan modulaarisuuden, joka vähensi riippuvuuksia huomattavasti. MVCS-mallin toiminta on aluksi vaikea ymmärtää, mutta jos kaikki projektin työntekijät omaksuvat toimintamallin, työnteko helpottuu huomattavasti. Jälkeenpäin ajateltuna myös tässä opinnäytetyössä esitellyistä luokista moni olisi voitu luokitella eri tavoin, esimerkiksi Warrior-, ja TeamLoaderService-palvelut olisivat toimineet arkkitehtuurin mukaisesti, mikäli niistä olisi tehty komentoluokkia.

SQLite toimii hyvin yksin pelattavan pelin tietokantana, kun siihen tarvittavat palvelut on luotu. C#-ohjelmistoon SQL-syntaksin kirjoittaminen on tosin vaativaa, sillä Visual Studio ei osaa automaattisesti tarkastaa siitä löytyviä virheitä. Tietokantojen ylläpito on näin myös ohjelmiston rakenteen muuttumisen aikana vaativa prosessi, mutta kun tietokantojen rakennetta ei enää muuteta, niiden muokkaus ja hallinta onnistuvat helposti.

LÄHTEET

SQLite–tietokantakielen viralliset kotisivut, luettu 6.3.2016
https://www.sqlite.org/

Unity Technologies -kotisivut, luettu 6.3.2016
https://unity3d.com/

Maskalik, S. 2012. Dapper–viitekehys, luettu 6.3.2016 http://blog.maskalik.com/asp-net/sqlite-simple-database-with-dapper/

StrangeIoC:n viralliset kotisivustot, luettu 10.8.2016
http://strangeioc.github.io/strangeioc/

Reenskaug, T. MVC-lähde, luettu 24.8.2016
http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

Stackoverflow–kysymyspalsta, luettu 6.3.2016
http://stackoverflow.com/

Neogames Finland ry:n kotisivut, tietoa pelialasta, luettu 26.8.2016
http://www.neogames.fi/

Fowler, M. 2005. Inversion of Control, luettu 26.8.2016
http://martinfowler.com/bliki/InversionOfControl.html

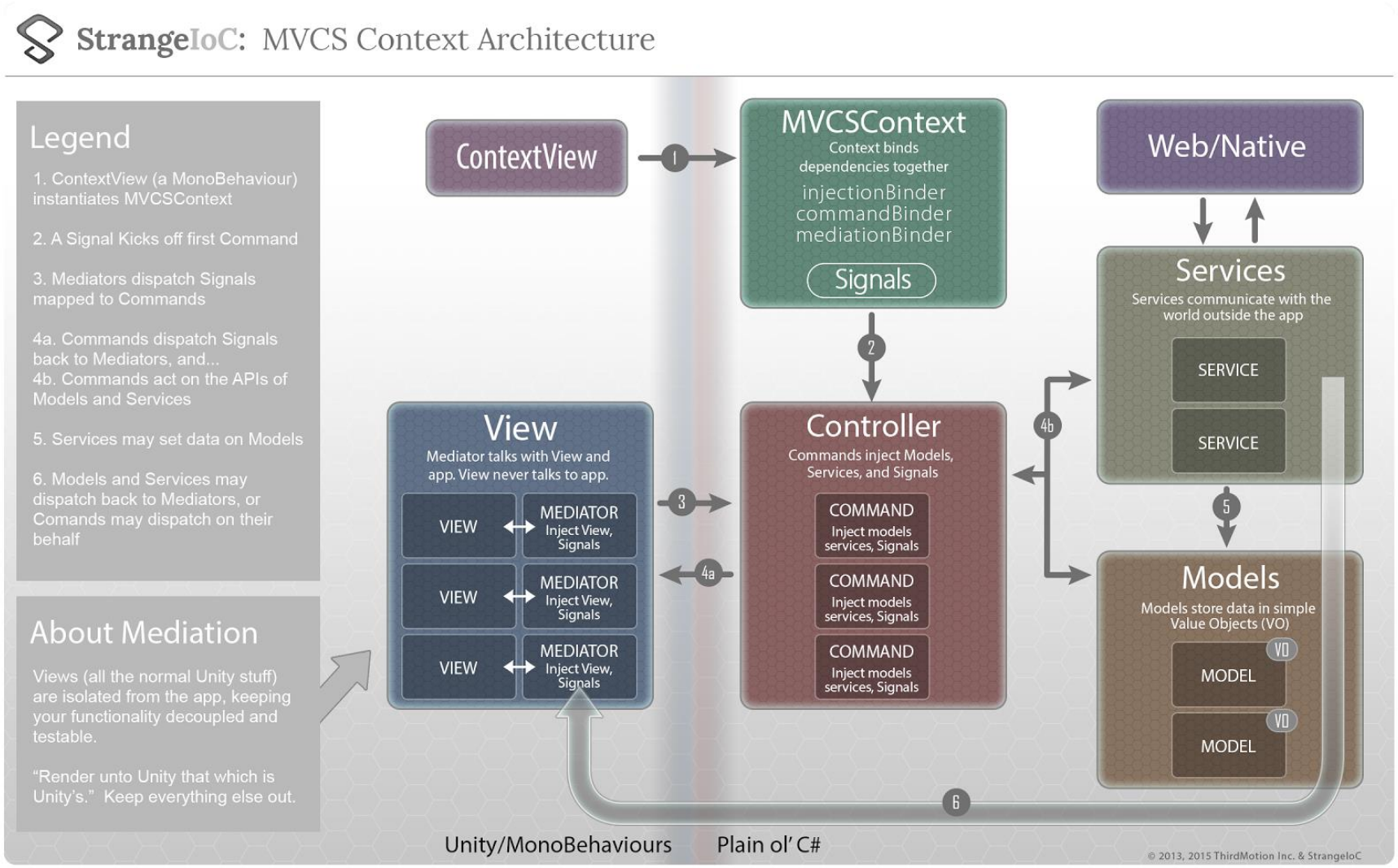
Dreamloop Games Oy. Kotisivut. Luettu 29.8.2016.
http://www.dreamloop.net/

Connolly, T. & Begg, C. 2005. Database Systems

LIITTEET

Liite 1. StrangeIoC:n toiminta kaaviona

Lähde: https://docs.google.com/document/d/1OV6vOZB6Od9vKmMlJaKkDUXtWsU8dwVSwIxaa5fQwd8/edit



Liite 2. Kaikki warriortablen tietueet lueteltuna erikseen

```
'Id' integer primary key,
'Name' string not null,
'TeamId' integer,
'Age' integer,
'GenderInt' integer,
'RoleInt' integer,

'Condition' integer,
'Mood' integer,
'Fame' integer,
'Infamy' integer,
'TribeInt' integer,

'Constitution' integer,
'Strength' integer,
'Agility' integer,
'Endurance' integer,
'Intelligence' integer,
'Reaction' integer,
'Stamina' integer,
'Mobility' integer,
'Marksmanship' integer,
'Experience' integer,

'SwordSkill' integer,
'AxeSkill' integer,
'BluntSkill' integer,
'PolearmSkill' integer,
'ShieldSkill' integer,
'BowSkill' integer,
'CrossbowSkill' integer,
'MagicSkill' integer,

'Ability1Id' integer,
'Ability2Id' integer,
'Ability3Id' integer,
'Ability4Id' integer,
'CaptainAbilityId' integer,

'UniqueWeaponId' integer,
'UniqueShieldId' integer,
'UniqueArmorId' integer,

'AuctionHouseId' integer,
'HasContract' bool,
'InActiveTeam' bool,
'InjuryId' integer,
'InjuryCurrentLength' integer
```