Daniel Heugenhauser

ScriptableObjectien hyödyntäminen tehtäväjärjestelmässä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

27.2.2026

Tiivistelmä

Tekijä:	Daniel Heugenhauser
Otsikko:	ScriptableObjectien hyödyntäminen tehtäväjärjestelmässä
Sivumäärä:	55 sivua
Aika:	27.2.2026

Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Pelisovellukset
Ohjaaja:	Lehtori Heini Puuska

Insinöörityö toteutettiin yhteistyössä startup-yrityksen kanssa, jonka tavoitteena oli kehittää nopeatempoinen, rampage-tyylinen mobiilipeli. Pelin rakenne perustuu lyhytkestoisiin mutta intensiivisiin tasoihin, joita tarvitaan suuri määrä. Kehityksen aikana tasojen määrä, logiikka ja niihin liittyvä data muuttuvat jatkuvasti, joten tarvittiin joustava ja helposti muokattava pohja tehtäväjärjestelmälle. Ratkaisun tuli myös kestää asiakkaan myöhempiä vaatimusmuutoksia ilman merkittäviä ohjelmistoteknisiä muutoksia.

Tätä tarkoitusta varten valittiin Unity-pelimoottorin ScriptableObject-komponentti. ScriptableObject toimii keskitettynä tietovarastona kaikelle tehtäviin liittyvälle datalle ja mahdollistaa uusien tehtävien lisäämisen peliin ilman ylimääräistä ohjelmointia. Järjestelmä on modularisoitu siten, että uusi tehtävä käsitellään automaattisesti, ja kehittäjän vastuulle jää vain serialisoitujen kenttien täyttäminen.

Järjestelmän rajapintana toimi pelin Service Locator-arkkitehtuuriin liitetty Manager-komponentti, joka hallinnoi kaikkia Quest-tyyppisiä ScriptableObjecteja joka päivittää niiden dataa pelin ajon aikana. Päivityksen yhteydessä käytettiin paikallista JSON-pohjaista tallennusta, jonka lataaminen ja kirjoittaminen tapahtui taustalla ilman, että pelaaja huomaa sitä.

Jokainen Unity-scene oli yhteydessä vastaavaan ScriptableObjectiin, mikä varmistaa pelin toimivuuden ja datan eheyden. Kehittäjän vastuulle jäi ainoastaan määritellyn kohtaus-nimeämisprotokollan noudattaminen.

Olen käyttänyt OpenAI:n ChatGPT:n versiota 5 tutkimusasetelman ideoinnissa ja työn jäsentelyssä ja otsikoinnissa. Olen käyttänyt samaa ohjelmaa myös tekstini kieliasun viimeistelyyn ja lähdeviitteiden muotoiluun. Opinnäytetyön tekijänä olen vastuussa kaikesta opinnäytteeni sisällöstä.

Avainsanat: Unity, quest system, ScriptableObject

Tämän opinnäytetyön alkuperä on tarkastettu Turnitin Originality Check -ohjelmalla.

Abstract

Author:	Daniel Heugenhauser
Title:	Utilizing ScriptableObjects in the Quest System
Number of Pages:	55 pages
Date:	27 February 2026
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Game Applications
Supervisor:	Senior Lecturer Heini Puuska

This final year project was carried out as part of a new start-up project developing a fast-paced, rampage-style mobile game where each level is short and action-intensive. Due to rapid gameplay, many levels are required. Level logic and data are expected to evolve throughout development, which creates the need for a flexible quest system that can be easily modified without major code changes as the client's requirements shift.

To meet these demands, Unity's ScriptableObject system was selected as the foundation of the quest architecture. ScriptableObjects serve as lightweight data containers that define all quest-related information, and new quests can be added to the game without additional programming. The system is modular: when a new quest asset is created, the architecture automatically recognizes and processes it, requiring the developer only to fill in the serialized fields.

A Quest Manager component, integrated into the project's Service Locator architecture, functions as the central interface for managing all quest ScriptableObjects. It updates quest data at runtime and handles loading and saving through local JSON-based storage, ensuring that player progress is preserved seamlessly.

Each Unity scene was linked to its corresponding quest ScriptableObject through a naming convention. This ensures that the gameplay logic remains synchronized with the correct quest data, leaving the developer responsible only for following the established scene-naming protocol.

Carrying out the project, I used OpenAI's ChatGPT (version 5) to support the ideation of the research design and the structuring and titling of this thesis. I also used the tool to assist in refining the language of the text and formatting the references. As the author of this thesis, I am fully responsible for all of its content.

Keywords:
Unity, quest system, ScriptableObject

Sisällys

Lyhenteet

1 Johdanto 1

2 Tehtäväjärjestelmän toteutustavat 2

2.1 ScriptableObject-pohjainen toteutus 2
2.2 Unity-instanssin rakenne 6
2.3 Staattiset Manager-luokat 8
2.4 Singleton Manager -luokat 9
2.5 Service Locator -arkkitehtuuri 11

3 Käytettävyys 15

3.1 Kehittäjän näkökulma 15
3.2 Pelaajan näkökulma 17
3.3 Testattavuus ja virheensieto 17

4 Skaalautuvuus 24

4.1 Tehtäväjärjestelmän skaalautuvuus 24
4.2 Modulaarisuus: tehtävien lisääminen ja poistaminen 25

5 ScriptableObjectit ja käyttöliittymä 28

5.1 Datan siirto ScriptableObjectista QuestManageriin 28
5.2 Esimerkki: Questin tilan päivitys käyttöliittymässä 34

6 ScriptableObjectit ja tallennusjärjestelmät 40

6.1 Kovakoodattu 41
6.2 JSON- ja XML-serialisointiformaatit 42
6.3 Tietokannat ja palvelimet 46

7 Yhteenveto 51

Lähteet 53

Lyhenteet

API: *Application Programming Interface*. Ohjelmointirajapinta, jonka kautta ohjelmistot voivat kommunikoida keskenään.

BaaS: *Backend as a Service.* Palvelu, jossa taustajärjestelmät ovat palveluntarjoajan ylläpitämiä.

DBMS: *Database Management System.* Tietokannan hallintajärjestelmä.

JSON: *JavaScript Object Notation.* Kevytrakenteisen datan esitysmuoto, joka perustuu avain-arvopareihin.

Scene: Unity-pelimoottorissa pelitasoa edustava rakennuspalikka.

SO: *ScriptableObject.* Unityn serialisoitava olioformaatti datan tallentamiseen ja jakamiseen.

SQL: *Structured Query Language.* Tietokantojen hallintaan ja datan käsittelyyn käytettävä kieli.

XML: *Extensible Markup Language.* Merkintäkieli hierarkkiseen datan kuvaamiseen.

1 Johdanto

Ohjelmistokehityksessä modulaarisuus ja skaalautuvuus ovat keskeisiä periaatteita. Modulaarisuus tarkoittaa ohjelmiston muokattavuutta ilman, että muut osat rikkoutuvat, ja skaalautuvuus sitä, että ohjelmisto säilyy hallittavana sen laajentuessa. [1, s. 242–243]. Unity-pelimoottori tarjoaa ratkaisuja datan organisointiin, eri komponenttien erotteluun ja pelilogiikan jäsentelyyn. Yksi näistä ratkaisuista on ScriptableObject-luokkarakenne, jonka avulla voidaan jakaa dataa tehokkaasti pelin eri osa-alueiden välillä. [2.]

Opinnäytetyö toteutettiin yhteistyössä aloittavan startup-yrityksen kanssa. Projektin tavoitteena oli kehittää ylhäältä kuvattu, Grand Theft Auto 1 -tyylinen rampage-peli. Pelin teemana on koiranulkoilutus. Pelaaja ulkoiluttaa tehtävänantajien koiria ja ansaitsee siitä pisteitä. Pisteitä voi ansaita myös aiheuttamalla mahdollisimman paljon tuhoa pelimaailmassa.

Jokaisessa tehtävänantajan tarjoamassa tehtävässä on kolme tasoa, jotka pelaajan tulee suorittaa yhtäjaksoisesti. Kun kaikki kolme tasoa on suoritettu, avautuu seuraava tehtävänantaja. Pelin vaikeustaso kasvaa progressiivisesti pelin edetessä. Pelin etenemismalli muistuttaa Angry Birds -pelin etenemismallia.

Tässä opinnäytetyössä tarkastellaan ScriptableObjectien hyödyntämistä asiakasprojektissa toteutetussa pelissä. ScriptableObjecteja on käytetty erityisesti tehtäväjärjestelmän datan ja logiikan hallinnassa. Ne toimivat rajapintana tallennusjärjestelmän ja visuaalisten komponenttien välillä, mikä mahdollistaa tehtävien helpon hallinnan, lisäämisen ja poistamisen.

ScriptableObject-pohjainen ratkaisu valittiin, koska peliin suunniteltiin alun perin vähintään yhdeksääkymmentä tehtävää. Näiden tehtävien sisältämä data kommunikoi useiden pelin osa-alueiden kanssa, kuten käyttöliittymäelementtien, pelinäkymän, eri pelikohtausten (scenejen) sekä valikkorakenteiden kanssa. Jär-

jestelmän rakentamisen tavoitteena oli mahdollistaa tehtävien lisääminen ja poistaminen ilman ohjelmointia tai ylimääräisiä toimenpiteitä Unity-editorissa. ScriptableObjectien arvoja muuttamalla peliohjelmisto integroi datan automaattisesti kaikkiin pelin osa-alueisiin. Näin tehtävien hallinta pysyy yksinkertaisena ja tehokkaana myös laajassa kokonaisuudessa.

Työssä tarkastellaan lisäksi vaihtoehtoisia tapoja tehtäväjärjestelmän toteuttamiseen, kuten staattisia hallintaluokkia ja C#-olioita. [2]. Näitä verrataan ScriptableObject -pohjaiseen ratkaisuun, ja arvioidaan, miksi juuri se soveltuu parhaiten kyseiseen peliprojektiin. Työssä käsitellään myös erilaisia tallennusratkaisuja, joista tässä toteutuksessa käytettyä JSON-formaattia verrataan vaihtoehtoisiin menetelmiin. [3]. Lisäksi työssä arvioidaan järjestelmän käytettävyyttä sekä kehittäjän että pelaajan näkökulmasta ja pohditaan testauksen ja virheensiedon merkitystä.

2 Tehtäväjärjestelmän toteutustavat

2.1 ScriptableObject-pohjainen toteutus

ScriptableObject on Unityn tarjoama tieto-olio, joka toimii tehokkaana datavarastona ja on irrallaan pelin muista C#-komentosarjainstansseista. Se on serialisoitava Unity-luokka, jota voidaan käyttää suurten tietomäärien tallentamiseen erillään pelin varsinaisesta logiikasta. Serialisointi on tiedon automaattista tallentamista ja lataamista Unity-editorin ja ajon aikaisen pelin välillä, ja instanssi on yksittäinen olio, joka on luotu luokan pohjalta. Toisin kuin useimmat Unityn komponentit, ScriptableObjectit eivät peri MonoBehaviour-luokkaa, minkä vuoksi ne eivät ole pelimoottorin GameObject-olioita. [4.]

Merkittävä ero GameObjecteihin verrattuna on se, että ScriptableObjectiin tehdyt muutokset säilyvät myös Unity-instanssien ja Scene-vaihdosten välillä. Tämä tekee niistä erityisen hyödyllisiä pelin pysyvän datan, kuten tehtävien, asetusten ja konfiguraatioiden, hallintaan. [4.]

ScriptableObjecteja voidaan hyödyntää ensisijaisesti datan varastointiin, mutta niitä voidaan käyttää myös pelilogiikan jäsentämisen tukena. Ne perivät Unity-pelimoottorin Object-luokan, minkä ansiosta niitä voidaan tarkastella ja muokata suoraan Unityn Inspector-näkymässä. [4.]

ScriptableObjectien luominen tapahtuu luomalla Unityyn uusi C#-skripti. MonoBehaviour perinnän sijaan luokka perii ScriptableObject-luokan ja sille määritellään CreateAssetMenu-attribuutti, jotta uusia asset-instansseja voidaan luoda helposti Unityn editorissa. Esimerkkikoodissa 1 esitetään yksinkertaisen QuestScriptableObject-luokan toteutus sekä CreateAssetMenu-attribuutin käyttö ScriptableObject-assetin luomiseksi Unityn editorissa.

```
using UnityEngine;

namespace Data
{
     [CreateAssetMenu]
     public class QuestScriptableObject : ScriptableObject
     {
          public int questID;
          public QuestState questState;

          public void CompleteQuest()
          {
               questState = QuestState.Completed;
          }
     }
}
```

Esimerkkikoodi 1. Koodiesimerkki ScriptableObject-luokan luomisesta.

Esimerkkikoodi 1 havainnollistaa C#-kielisen ScriptableObject-luokan toteutusta, joka hyödyntää UnityEngine-kirjastoa. ScriptableObject-luokka sijoitetaan pelin datakerrokseen, ja sen pohjalta luodaan uusi julkinen luokka QuestScriptableObject, joka perii Unityn ScriptableObject-luokan. Luokassa on määritelty kaksi muuttujaa: kokonaislukutyyppinen questID sekä QuestState-enumeraatio, jota käytetään kuvaamaan tehtävän tilaa. QuestState on enumeraatio, eli rajatun arvojoukon määrittelevä tietotyyppi, jonka avulla tehtävän tila voidaan esittää selkeässä ja tyypiturvallisessa muodossa. Lisäksi luokkaan on toteutettu CompleteQuest-metodi, joka asettaa tehtävän tilan arvoksi Completed. Crea-

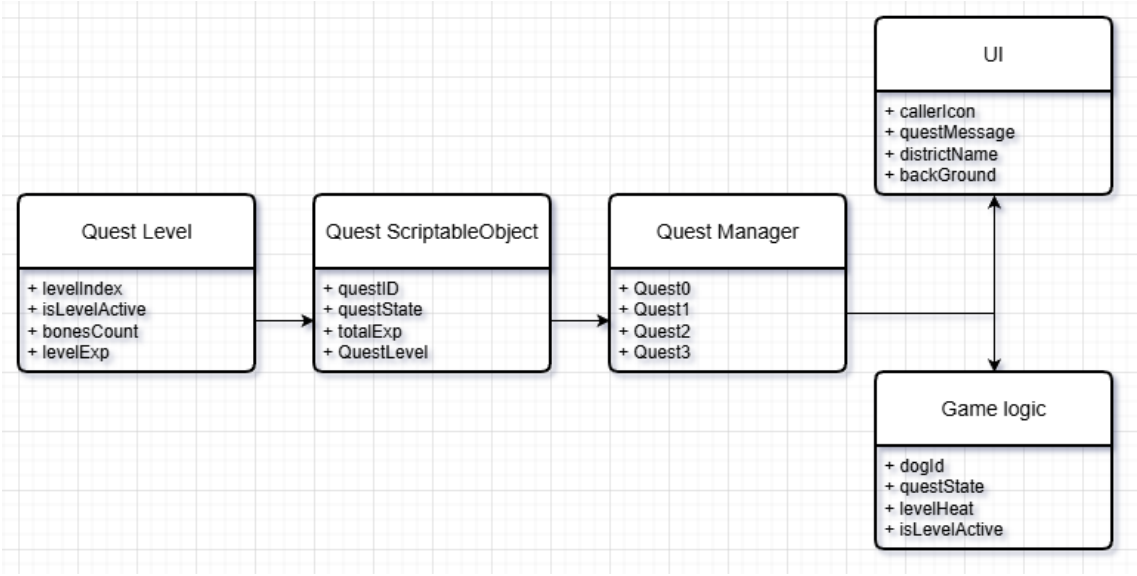
teAssetMenu-attribuutti mahdollistaa uusien QuestScriptableObject-instanssien luomisen suoraan Unityn editorissa.

Peliin toteutettiin ratkaisu, jossa hyödynnettiin ScriptableObjecteja. Ratkaisu rakennettiin siten, että jokainen quest-tyyppinen ScriptableObject sisältää joukon tasoa edustavia C#-olioita. Pelin lähtökohtana oli luoda kolme tasoa jokaiseen tehtävään, mutta järjestelmä on toteutettu skaalautuvasti siten, että tasoja voidaan lisätä listaan rajattomasti.

Jokainen taso sisältää tason tilaan liittyviä muuttujia sekä muutamia metodeja, jotka vastaavat esimerkiksi tason suorittamisesta ja pistearvon (heat-arvon) lisäämisestä. Varsinainen ScriptableObject säilyttää tehtävään liittyvän datan, kuten käyttöliittymäelementit (UI), sprite-grafiikan sekä muun tehtävädataan liittyvän informaation.

Itse QuestManager-luokka pitää sisällään listan kaikista QuestScriptableObject-olioista, ja suurin osa kaikesta tehtäviin liittyvästä logiikasta sijaitsee juuri QuestManagerissa, joka on kytköksessä pelin muun arkkitehtuurin kanssa.

Kuvassa 1 esitetään vuokaavio, joka havainnollistaa tehtäväjärjestelmän tiedonkulkua. Kaaviossa kuvataan, miten data siirtyy tehtävän tasokohtaisesta rakenteesta ScriptableObjectiin ja siitä edelleen QuestManager-luokkaan. QuestManager vastaa ScriptableObjectien hallinnasta, kun taas ScriptableObjectit ylläpitävät QuestLevel-olioita. Lisäksi QuestManager huolehtii tehtäviin liittyvästä logiikasta sekä käyttöliittymäelementtien päivityksestä.



Kuva 1. Havainnollistava kaavio siitä, miten tehtäväjärjestelmän datarakenne on organisoitu.

QuestManager-luokan konstruktorissa (eli metodissa, joka suoritetaan, kun QuestManager-instanssi luodaan) alustetaan koko tehtäväjärjestelmä. Konstruktorissa kutsutaan LoadQuests()-metodia, joka lataa kaikki QuestScriptableObject-oliot listaan hyödyntäen UnityEngine-kirjaston Resources.LoadAll-metodia. Samalla kaikkien listan olioiden arvot nollataan. Tämän jälkeen kaikki quest-oliot alustetaan tarkistamalla, löytyykö pelin tallennuskansiosta JSON-muodossa oleva tiedosto, johon tehtävädata on aiemmin tallennettu. Jos tallennustiedostoa ei ole saatavilla, kaikki tehtävät asetetaan lukittuun tilaan, ja ensimmäinen taso avataan. Toisin sanoen peli palautetaan lähtöasetuksiinsa.

QuestManager vastaa siitä, että pelin data jakautuu oikeisiin komponentteihin riippumatta siitä, ovatko ne visuaalisia vai pelilogiikkaan liittyviä. Esimerkiksi tässä projektissa, jossa pelin ideana oli toteuttaa GTA1-tyylinen ylhäältä kuvattu rampage-peli koirien ulkoiluttamisesta, koirien määrä ja tyyppi määritellään suoraan ScriptableObjecteista. Samalla nämä objektit varastoivat myös pelin käyttöliittymään liittyvää dataa, kuten tehtävänantajan kuvan tai tason taustagrafiikan.

2.2 Unity-instanssin rakenne

Ohjelmistokehityksessä oliolla tarkoitetaan yleensä olio-ohjelmoinnin käsitettä, joka sisältää sekä dataa että siihen liittyvää logiikkaa. Esimerkiksi voidaan luoda uusi C#-luokka nimeltä Auto, joka tallentaa auton värin ja kilometrilukeman, kuten esimerkkikoodissa 2 esitetään. Auto-luokassa voisi olla myös metodi, joka kasvattaa auton kilometrilukemaa. Tämä on yksinkertainen esimerkki siitä, mitä olio-ohjelmointikielissä tarkoitetaan oliolla. Tällaiset oliot ovat olemassa puhtaasti ohjelman sisäisen logiikan ja datarakenteiden hallintaan, eivätkä ne ole sidoksissa Unity-pelimoottoriin. Esimerkkikoodi 2 havainnollistaa edellä kuvattua olion käsitettä. [5.]

```
class Car
{
     public string color;
     private int milage;
     public void AddMilage(int amount)
     {
          milage += amount;
     }
}

class Program
{
     static void Main(string[] args)
     {
          Car RedCar = new Car();
          RedCar.color = "Red";
          RedCar.AddMilage(10000);

          Car BlueCar = new Car();
          BlueCar.color = "Blue";
          BlueCar.AddMilage(1000);
     }
}
```

Esimerkkikoodi 2. Yksinkertainen esimerkki C#-luokan olioista. Koodissa luodaan kaksi erillistä Car-oliota, joille asetetaan eri värit ja kilometrilukemat.

Uusista olioista voidaan luoda monta eri instanssia käyttämällä new-komentoa. Tämä tarkoittaa sitä, että luodaan monta rinnakkaista auto-oliota, jossa on erilaisia värejä ja malleja. Nämä kaikki oliot jakavat kuitenkin saman logiikan eli metodit. [6.]

Unity-pelimoottorissa oliot ovat integroituna moottorin komponenttipohjaiseen arkkitehtuuriin. Unityssä uudet skriptit perivät yleensä MonoBehaviour-luokan, joka tarjoaa valmiita elinkaareen liittyviä funktioita, kuten Awake(), Start(), Update() ja OnDestroy(). Esimerkkikoodin 3 mukaisesti näiden elinkaarimetodien avulla skripti saa tiedon siitä, missä vaiheessa Unityn päivityssykli on, ja voi toimia sen mukaisesti. [7.]

```
public class Car : MonoBehaviour
{

     private void Awake()
     {

     }

     void Start()
     {

     }

     void Update()
     {

     }

     private void OnDestroy()
     {

     }
}
```

Esimerkkikoodi 3. Car-niminen Unity-komponentti, joka perii MonoBehaviour-luokan. Komponenttiin sisältyvät Unityn elinkaaren metodit Awake, Start, Update ja OnDestroy, jotka Unity kutsuu automaattisesti komponentin elinkaaren eri vaiheissa.

MonoBehaviour-luokan perivät oliot elävät GameObjectien komponentteina, mikä tarkoittaa, että ne voidaan liittää suoraan peliobjekteihin Unityn editorissa. Toisin kuin tavalliset C#-oliot, MonoBehaviour-komponentit eivät ole pelkkää dataa ja logiikkaa, vaan niillä on suora yhteys pelimaailmassa olevaan objektien elinkaareen ja käyttäytymiseen. [7.]

MonoBehaviour-luokille ei voi luoda omia konstruktoreita new-avainsanalla, vaan niiden alustaminen tapahtuu Unityn elinkaarimetodien, kuten Awake(), Start() ja OnDestroy(), kautta. Unity-instanssilla tarkoitetaan tällaista MonoBe-

haviour-komponentin tai GameObjectin konkreettista esiintymää pelimaailmassa, jonka elinkaarta Unity hallitsee. Peliohjelmiston elinkaaren loputtua kaikki tallentamaton data lakkaa olemasta.

2.3 Staattiset Manager-luokat

Staattinen luokka C#-ohjelmointikielessä on luokka, jota ei voi instansioida eli siitä ei voi luoda olioita new-avainsanalla. Koska staattinen luokka on olemassa ohjelmassa vain yhtenä versiona, siitä ei voi olla useita rinnakkaisia kopioita. Staattinen luokka eroaa tavallisesta luokasta siten, että kaikki sen jäsenet (muuttujat ja metodit) ovat myös staattisia, ja niitä käytetään suoraan luokan nimen kautta ilman, että oliota luodaan. Tämä tekee siitä kätevän esimerkiksi globaalien apumetodien tai yksittäisen tilan ylläpitoon pelissä. Esimerkkikoodissa 4 havainnollistetaan staattisen SaveManager-luokan käyttöä pelin tehtävätietojen tallentamiseen. [8.]

```
public static class SaveManager
{
      public const string QUEST_SAVE_FILE_PATH = "quest_data.json";

      public static void SaveQuestData(QuestScriptableObject[] questList)
      {
            List<QuestData> questDataList = questList
                  .Select(quest => new QuestData
                  {
                        questID = quest.questID,
                        questState = quest.questState,
                        totalExp = quest.totalEXP,
                        newMessageIndicator = quest.newMessageIndicator,

                        levels = quest.questLevels
                              .Select(level => new QuestData.LevelData
                              {
                                    levelIndex = level.levelIndex,
                                    levelNumber = level.levelNumber,
                                    isLevelCompleted = level.isLevelCompleted,
                                    isLevelActive = level.isLevelActive,
                                    bonesCount = level.bonesCount,
                                    levelHeat = level.levelHeat,
                                    levelExp = level.levelEXP
                              })
                              .ToList()
                  })
                  .ToList();

                  string jsonData = JsonUtility.ToJson(
                        new QuestDataList { quests = questDataList },
                        true
                        );

                  string path = Path.Combine(
                        Application.persistentDataPath,
                        QUEST_SAVE_FILE_PATH
                        );

                  File.WriteAllText(path, jsonData);
                  Debug.Log("Quest data saved successfully.");
      }
}
```

Esimerkkikoodi 4. Staattinen SaveManager-luokka sekä sen staattinen SaveQuestData-metodi. Metodi saa parametrina listan kaikista pelissä käytettävistä QuestScriptableObject-olioista.

2.4 Singleton Manager -luokat

Singleton-luokka muistuttaa staattista luokkaa siinä mielessä, että se tarjoaa globaalin pääsyn yhteen ja samaan instanssiin koko ohjelman ajaksi. Toisin kuin staattinen luokka, Singleton on kuitenkin tavallinen luokka, josta luodaan yksi ainoa instanssi. Unityssa tämä instanssi voidaan säilyttää kohtausvaihdosten yli käyttämällä DontDestroyOnLoad()-metodia. Näin varmistetaan, että pelissä on aina olemassa vain yksi Singleton-instanssi kerrallaan. [9;10.] Esimerk-

kikoodissa 5 havainnollistetaan ObjectPooler-luokan toteutus Singleton-rakenteella sekä sen käyttö objektivaraston hallintaan.

```
public class ObjectPooler : MonoBehaviour
{
     [System.Serializable]
     public class Pool
     {
          public string tag;
          public GameObject prefab;
          public int size;

     }
     public List<Pool> pools;
     public Dictionary<string, Queue<GameObject>> poolDictionary;

     #region Singleton
     public static ObjectPooler instance;

     void Awake()
     {
          if(instance == null)
          {
               instance = this;
          }
          else if(instance != this)
          {
               Destroy(this);
          }

          DontDestroyOnLoad(this);
     }
     #endregion
}
```

Esimerkkikoodi 5. Esimerkki singleton rakenteesta, jossa on objektivarasto, jonka tehtävä on instansioida lista objekteja awake komennon suorittaessa. [10].

Asiakasprojektiin toteutettu tallennusjärjestelmä oli staattinen Manager-luokka ilman Singleton-rakennetta. Tähän ratkaisuun päädyttiin, koska järjestelmän ei tarvinnut säilyttää dataa scenejen välillä, jolloin Singletonin käyttö olisi ollut tarpeetonta. Tieto siirretään tallennusjärjestelmään suoraan ScriptableObjecteista.

Staattiset Manager-luokat ovat yksi tapa hallita pelin dataa ja logiikkaa. Luokkien staattisen luonteen ansiosta niihin voidaan viitata helposti mistä tahansa ohjelmakoodin osasta ilman instanssien luomista. Yleisiä esimerkkejä ovat SaveManager- ja GameManager-luokat, jotka sisältävät keskeistä pelidataa, kuten

pelaajan tilan pelinäkymien vaihdosten yli, ja toimivat keskitettyinä tiedon lähteinä pelin eri osa-alueille.

Staattisten Manager-luokkien etuna on yksinkertaisuus ja helppokäyttöisyys. Koska niiden metodeja ja muuttujia voidaan kutsua suoraan luokan kautta, riippuvuudet vähenevät ja käyttö on suoraviivaista erityisesti pienissä projekteissa. [8.]

Niihin liittyy kuitenkin myös rajoituksia. Koska staattiset luokat eivät ole olioita, niitä ei voi käsitellä Unityn Inspector-näkymässä, eikä niiden sisältämää dataa voi sarjoittaa tallennusmuotoon ilman erillistä logiikkaa. Lisäksi staattisia luokkia on vaikeampi testata verrattuna ScriptableObjecteihin. Tämä johtuu siitä, että staattinen luokka on globaali ja tiukasti sidottu koodiin, jolloin sen tilaa on hankala eristää yksikkötesteissä. ScriptableObjectit puolestaan voidaan luoda erillisinä assetteina ja syöttää testattavaan koodiin, minkä lisäksi niiden arvoja on helppo muokata Unityn Inspector-näkymässä. Asset tarkoittaa Unity-projektissa resurssia tai sisältöä, joka on tallennettu projektin tiedostoihin ja jota pelimoottori käyttää pelin suorituksen aikana. Suurissa projekteissa staattiset luokat voivat kasvaa liian suuriksi ja muodostua niin sanotuiksi God Class -rakenteiksi, joissa yksi luokka kantaa liikaa vastuuta pelin tilasta.

Tehtäväjärjestelmän toteutuksessa lähtökohtana oli luoda skaalautuva ja helposti muokattava järjestelmä, jossa uusien tehtävien lisääminen onnistuisi ilman ylimääräistä ohjelmointia. Tämä ei olisi ollut mahdollista, jos toteutus olisi perustunut pelkästään Singleton- tai staattisiin Manager-luokkiin.

2.5 Service Locator -arkkitehtuuri

Service Locator -arkkitehtuurin tarkoituksena on luoda abstrakti kerros, johon voidaan rekisteröidä ja kapseloida erilaisia palveluita tai prosesseja. Mallin ydinajatuksena on ylläpitää yksi keskitetty singleton-luokka, joka toimii palvelujen hakupisteenä koko sovelluksessa. Tämä keskitetty luokka vastaa rekisteröityjen palveluiden hallinnasta ja tarjoaa yhtenäisen tavan hakea niitä muualta ohjel-

mistosta. Tämä on hyödyllistä silloin, kun projekti skaalautuu, sillä samalla komponenttien väliset riippuvuudet kasvavat.

Ongelmia syntyy erityisesti siitä, että Unityn peliobjektien skriptit perivät aina MonoBehaviour-luokan. Tämä rajoitus estää omien konstruktorien käytön näissä luokissa. Koska konstruktoreja ei voi hyödyntää riippuvuuksien injektointiin, jokaisen MonoBehaviour-luokan täytyy hakea tarvitsemansa riippuvuudet itse joko ohjelmallisesti tai Unityn Inspector-näkymän kautta. Tämän seurauksena riippuvuuksien hallinta alkaa muuttua vaikeaksi ohjelmiston skaalautuessa, ellei niiden ylläpitoon ole selkeää ja keskitettyä rakennetta. [11.]

Service Locator -mallia voidaan ajatella eräänlaisena varastona, johon on keskitetty erilaisia palveluita. Kun jokin komponentti tarvitsee palvelua, se pyytää sitä Service Locatorilta, joka palauttaa aiemmin rekisteröidyn riippuvuuden. Näin yksittäisten luokkien ei tarvitse itse huolehtia riippuvuuksiensa etsimisestä.

Peliin toteutetussa Service Locator -mallissa jokaisessa Unity-scenessä käytettiin yhtä Context-nimistä malliobjektia (prefab), joka sisälsi ServiceContext-nimisen C#-skriptin. ServiceContext vastasi kaikkien palveluiden rekisteröinnistä. Pelissä kaikki logiikkakerrokseen kuuluvat manager-luokat rekisteröitiin tämän arkkitehtuurin kautta.

Kun ServiceContext suoritetaan Unityn pääsäikeen käynnistyessä, se rekisteröi kaikki palvelut ServiceLocatoriin. Koska jokaisessa scenessä on Context-komponentti, kaikki siihen rekisteröidyt palvelut ovat haettavissa mistä tahansa koodin osasta ServiceLocatorin kautta. Esimerkkikoodi 6 havainnollistaa peliin toteutetun ServiceContext komponentin RegisterServices-metodin, jossa rekisteröidään kaikki peliin toteutetut manager-luokat.

```
private void RegisterServices()
{
     ServiceLocator.RegisterService<IPlayerManager, PlayerManager>(
          new PlayerManager(_initialPlayerState));

     ServiceLocator.RegisterService<IGameManager, GameManager>(
          new GameManager(_initialGameState));

     ServiceLocator.RegisterService<IOptionsManager, OptionsManager>(
          new OptionsManager(_initialOptionsState));

     ServiceLocator.RegisterService<IDogManager, DogManager>(
          new DogManager(_initialDogState));

     ServiceLocator.RegisterService<ISceneManager, SceneManager>(
          new SceneManager());

     ServiceLocator.RegisterService<IQuestManager, QuestManager>(
          new QuestManager());
}
```

Esimerkkikoodi 6. ServiceContext-peliobjektin RegisterServices-metodi, jossa rekisteröidään kaikki pelin logiikkakerroksessa sijaitsevat manager-luokat Service Locator -arkkitehtuuriin.

Esimerkkikoodin 7 mukaisesti ServiceLocator-luokka sisältää sanakirjan, jossa ylläpidetään palvelun tyyppiä ja tunnistetta. ServiceLocator pitää sisällään metodit, joilla voidaan hakea ja rekisteröidä palveluja. Itse palvelut haetaan rekisteröidyn luokan rajapintaa käyttämällä (Interface).

Ohjelmoinnissa rajapinta (Interface) määrittelee joukon metodeja ja ominaisuuksia, jotka rekisteröidyn luokan pitää toteuttaa, mutta ei tee toteutusta. Metodin kuuluu olla lisättynä rajapintaan, jotta sitä voidaan käyttää pelin ServiceLocatorissa.

```
public static void RegisterService<TInterface, TImplementation>(
     TImplementation instance,
     int id = 0)
     where TInterface : class
     where TImplementation : notnull, TInterface
{
     _serviceMap[(typeof(TInterface), id)] = instance;
}

public static TInterface GetService<TInterface>(int id = 0)
     where TInterface : class
{
     return _serviceMap.TryGetValue(
          (typeof(TInterface), id),
          out object instance)
          ? instance as TInterface
          : null;
}
```

Esimerkkikoodi 7. ServiceLocator-luokan RegisterService- ja GetService-metodit, jotka vastaavat palveluiden rekisteröinnistä ja hakemisesta.

Peliin toteutettu QuestManager-komponentti on liitetty ServiceLocator-arkkitehtuuriin. Jos pelin visuaalisen kerroksen komponentit tarvitsevat tehtävään liittyvää tietoa, ne hakevat sen ServiceLocatorin kautta esimerkkikoodin 8 mukaisesti MonoBehaviour-luokasta peritystä skriptistä. Tämän jälkeen luokassa on käytössä kaikki QuestManagerin metodit.

```
using UnityEngine;

public class ExampleUnityObject : MonoBehaviour
{
     private IQuestManager _questManager;

     private void Start()
     {
          _questManager = ServiceLocator.GetService<IQuestManager>();
     }
}
```

Esimerkkikoodi 8. ServiceLocatorin kautta haettavan palvelun käyttö MonoBehaviour-luokassa.

3 Käytettävyys

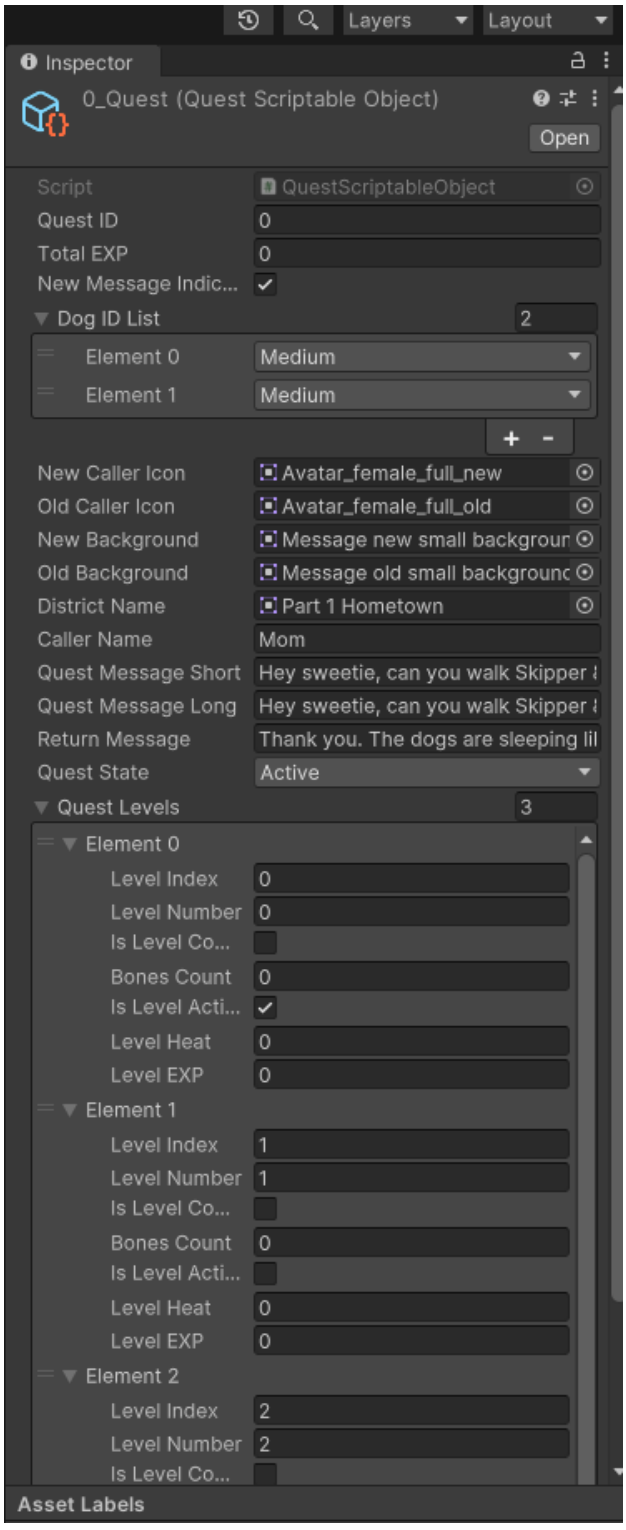
3.1 Kehittäjän näkökulma

Kehittäjän näkökulmasta ScriptableObject-pohjainen ratkaisu on helppo ja selkeä tapa lisätä uusia quest-olioita peliin. Koska objektit on eriytetty omiksi komponenteikseen, pelilogiikka ja data pysyvät erillään, mikä tekee koodista selkeämpää ja ylläpidettävämpää.

Quest-olioiden muokkaaminen onnistuu suoraan Unityn Inspector-näkymässä ilman koodimuutosten tarvetta. Uuden tehtävän lisääminen tapahtuu luomalla uusi quest-malli Unityn kautta, jolloin järjestelmä generoi valmiin Scriptable Object -instanssin. Kehittäjän tehtäväksi jää täyttää serialisoidut kentät, jotta pelin komponentit voivat hyödyntää dataa ja logiikka toimii oikein. [12.]

Järjestelmä sisältää lisäksi itse toteutetun editorityökalun, jonka avulla tehtäväelementeille voidaan generoida yksilölliset tunnisteet (ID:t). Tämä varmistaa, että pelimoottori osaa tunnistaa, mikä tehtävä on käytössä missäkin tilanteessa.

Kuvassa 7 nähdään Quest-objektiin sisältyvä QuestState-niminen enumeraattori, jonka avulla tehtävän tila määritetään suoraan koodissa. Enum sisältää seuraavat tilat: Locked (lukittu), Unlocked (avattu), Completed (suoritettu) ja Active (aktiivinen). Järjestelmä tarkistaa aina ensimmäisen tehtävän tilan, joka on oletusarvoisesti Locked. Tämän vuoksi on tärkeää, että kaikki uudet quest-ScriptableObjectit asetetaan aluksi Locked-tilaan, jotta järjestelmä toimii oikein.



Kuva 2. Pelissä käytetty Quest tyyppinen ScriptableObject. Muuttujien lisäksi objekti sisältää QuestLevel-nimisiä C#-olioita, jotka edustavat kunkin tehtävän eri tasoja.

Tehtävän tila tallennetaan automaattisesti staattisen SaveManager-luokan kautta. QuestManager-luokassa on metodi, joka jokaisen tehtävän vaihtuessa kutsuu funktiota SaveManager.SaveQuestData, ja tallentaa pelin tilan ScriptableObjecteista tiedostoon tietokoneen tallennuskansioon (Application.persistentDataPath). Uutta peli-instanssia ladattaessa järjestelmä hakee tiedot pelaajan kovalevylle tallennetusta JSON-tiedostosta.

3.2 Pelaajan näkökulma

Pelaajan näkökulmasta ScriptableObject-pohjainen järjestelmä tekee etenemisestä sujuvaa. Jokainen Quest-komponentti on kapseloitu omaan lokeroonsa ja päivittää pelin tilaa huomaamattomasti. Pelaaja ei tule tietoiseksi siitä, että taustalla toimii kyseinen järjestelmä, mutta huomaa sen, että pelissä ei ole erillistä tallennusvalintaa. Kokemuspisteet ja avatut tasot päivittyvät aina automaattisesti pelaajan edetessä pelissä.

Pelin progressio ja tallennusjärjestelmä suunniteltiin jäljittelemään Angry Birds -pelin etenemismallia, jossa suoritetut tasot pysyvät pysyvästi avattuina pelin edetessä. Jokainen quest-objekti sisältää kolme tasoa, ja pelaajan yhteispisteet lasketaan yhteen jokaisen läpäistyn tason jälkeen. Pelaaja näkee kokonaispistemäärän tason päätyttyä, minkä perusteella hän voi valita seuraavan pelattavan tason. Kun kaikki kolme tasoa on suoritettu, peli avaa automaattisesti uuden quest-objektin, jolloin pelaajalle avautuu mahdollisuus jatkaa seuraavaan tehtävään. Lisäksi peli sisältää arvostelujärjestelmän, jossa jokainen taso arvioidaan asteikolla nollasta kolmeen. Arvostelu määräytyy pelaajan keräämien kokemuspisteiden perusteella.

3.3 Testattavuus ja virheensieto

Unity-pelimoottorissa on käytettävissä automaattisia testaustyökaluja, joista keskeisin on Test Runner. Test Runnerin avulla voidaan suorittaa yksikkö- ja integraatiotestejä Edit Mode ja Play Mode -tilassa. Kehittäjä voi kirjoittaa testimetodeja, joilla varmistetaan pelin logiikan oikea toiminta ilman, että peliä täytyy

ajaa manuaalisesti läpi. [13, s. 8–9.] Unityn intergroitu Test runner on Unity-pelimoottoriin integroitu NUnit-ohjelmistokehys. NUnit on avoimen lähdekoodin testauskirjasto .NET-kielille.

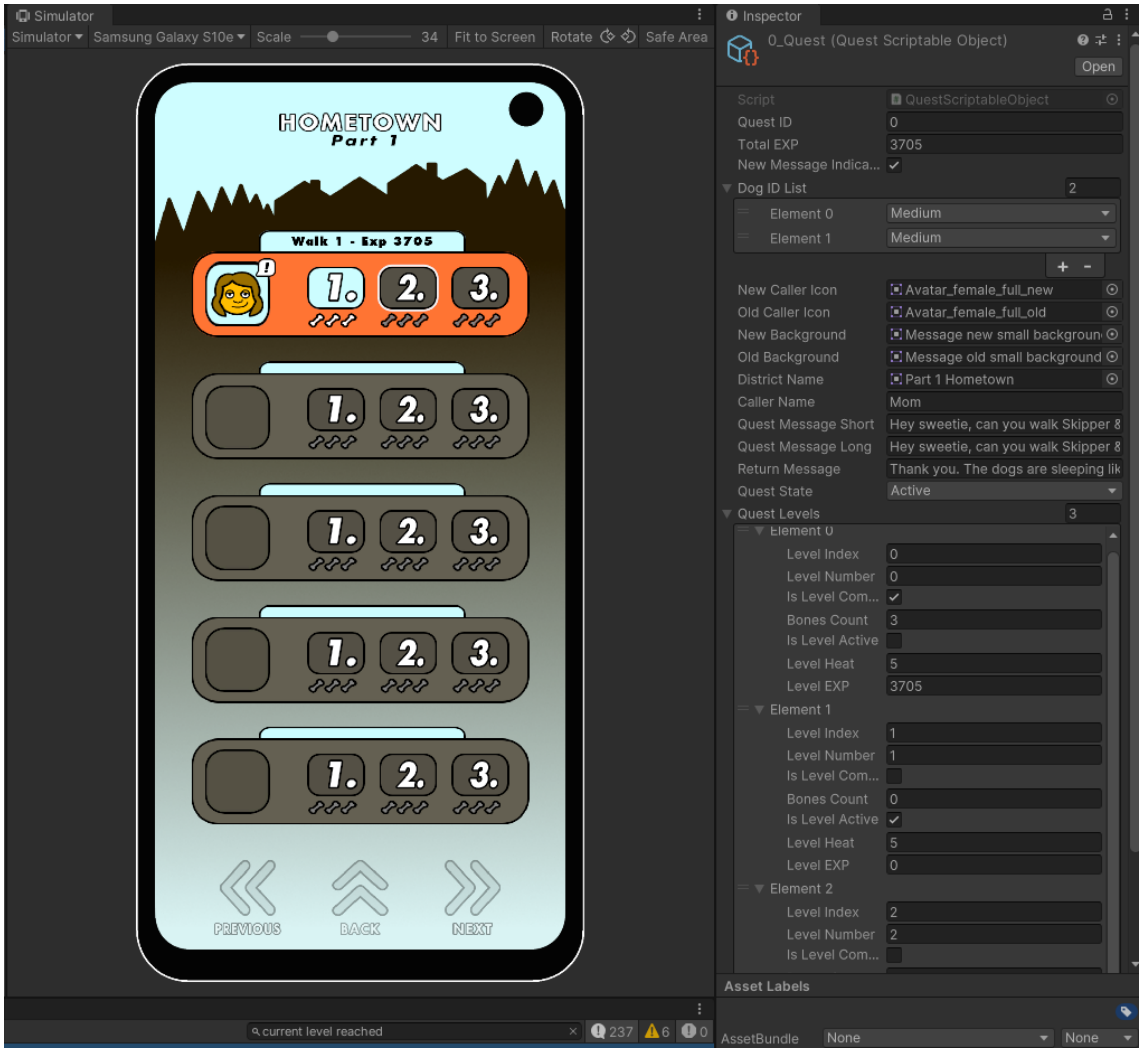
Tehtäväjärjestelmässä käytettiin sekä integroitua testaustyökalua että manuaalista testausta. Integroituna työkaluna hyödynnettiin Unityn Test Runner -komponenttia, kun taas manuaalinen testaus toteutettiin sekä ohjelmointikoodin että Unityn Inspector-näkymän kautta. ScriptableObjectien integrointi Unityn Inspectoriin helpottaa testausta, koska arvojen muutoksia voidaan seurata suoraan Play Modessa. Tämä tekee virheiden havaitsemisesta ja korjaamisesta huomattavasti helpompaa.

Tehtäväjärjestelmän lähtökohtana oli modularisoida komponentti, eli erottaa se omaksi itsenäiseksi kokonaisuudekseen muusta pelin logiikasta ja datasta. Kuten Insinööri Mikael Korpinen on todennut, testitapaukset kannattaa aina rajata ohjelmiston sisäisiin rajapintoihin, jotta testien korjaaminen ja ylläpito on tulevaisuudessa helpompaa. [13, s. 19]. Tehtäväjärjestelmän tapauksessa rakennettiin hallintaluokka, joka kontrolloi kaikkia tehtäviin liittyviä ScriptableObject-olioita ja kommunikoi muun pelilogiikan kanssa. Itse QuestManager-niminen komponentti, joka liittyy pelin Service Locator -arkkitehtuuriin, toimi testitapauksissa rajapintana muun logiikan kanssa.

Kun peliä pelataan Unityn Play Modessa, QuestManager-hallintaluokka päivittää aktiivisen quest-komponentin tiedot siihen liittyvään ScriptableObjectiin. Kaikki käyttöliittymäkomponentteihin päivitettävät tiedot tulevat suoraan QuestManager-luokasta, joten virheiden etsiminen rajoittuu pääasiassa kyseiseen luokkaan.

Quest-tyyppinen ScriptableObject sisältää kaikki aktiiviseen quest-komponenttiin liittyvät tiedot riippumatta siitä, liittyvätkö ne käyttöliittymäelementteihin vai pelin sisäisiin komponentteihin. Kuvassa 8 havainnollistetaan tilannetta, jossa ensimmäinen taso on avattu ja on Active-tilassa. Tällöin quest-ScriptableObjec-

tin ensimmäinen taso on merkitty Active-tilaan, ja se näkyy käyttöliittymässä aktiivisena tasona.



Kuva 3. Quest SciptableObject -olioista päivitetään dataa suoraan UI-elementteihin ajon aikana.

Kun pelaaja suorittaa tason, siihen liittyvä arviointiasteikko (1–3 luuta) päivittyy vastaaviin käyttöliittymäelementteihin. Samalla myös yhteispisteet päivittyvät reaaliaikaisesti UI-näkymään. Vaaleansininen numero osoittaa läpäistyn tason, kun taas harmaalla ja valkoisella reunuksella merkitty numero kuvaa parhaillaan aktiivista tasoa.

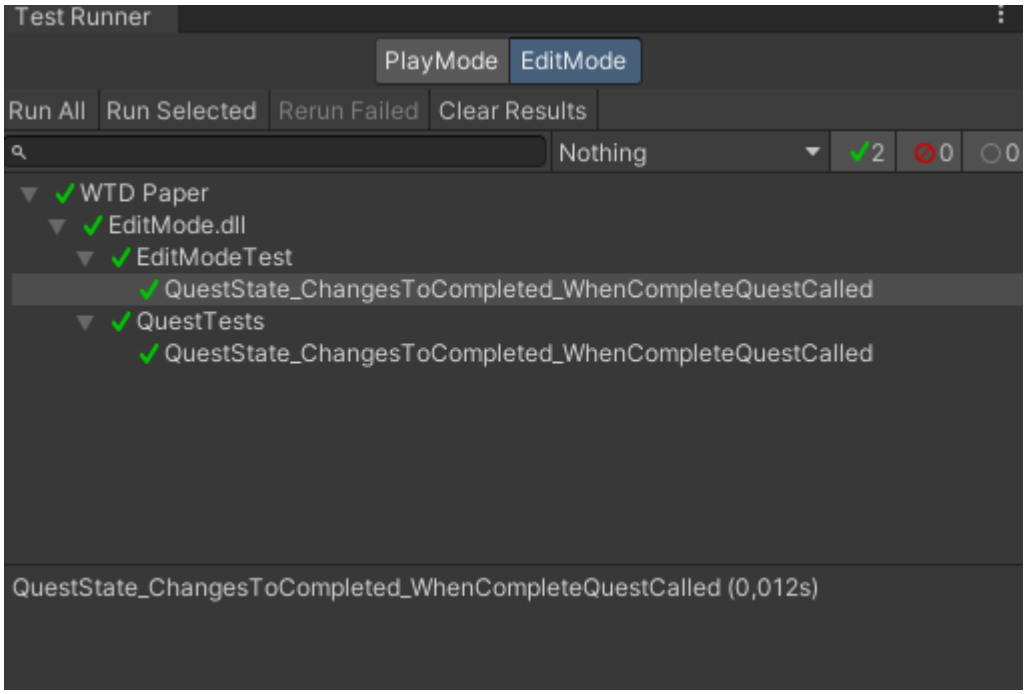
Esimerkkikoodissa 9 havainnollisestetaan, miten Unityn Test Runner -työkalu käyttää NUnit.Framework-kirjastoa Quest-olion CompleteQuest-metodin toiminnan testaamiseen. Jos CompleteQuest-metodi toimii odotetulla tavalla, testi läpäistään.

```
using UnityEngine;
using NUnit.Framework;
using Data;

public class QuestTests
{
	[Test]
	public void QuestState_ChangesToCompleted_WhenCompleteQuestCalled()
	{
		var quest = ScriptableObject.CreateInstance<QuestScriptableObject>();
		quest.questID = 1;
		quest.questState = QuestState.Active;
		quest.CompleteQuest();
		Assert.AreEqual(QuestState.Completed, quest.questState);
	}
}
```

Esimerkkikoodi 9. Esimerkki Test Runnerin Edit Mode -testin C#-koodista.

Test Runner -työkalua käytettäessä on hyvän ohjelmistokehityskäytännön mukaista luoda testitapauksia jo pelikehityksen alkuvaiheessa, vaikka se saattaisi aluksi tuntua työläältä. Peliprojektin kasvaessa ja monimutkaistuessa manuaalinen virheiden korjaaminen muuttuu yhä työläämmäksi. Jos integroidut testitapaukset on toteutettu huolellisesti, ne helpottavat virheiden jäljittämistä ja nopeuttavat ohjelmistokehityksen kokonaisprosessia. [14.]

Kuvassa 4 havainnollistetaan, kuinka kaikki testitapaukset (tässä tapauksessa yksi) on suoritettu ja läpäisty. Testit ajettiin Test Runner -ikkunan yläkulmassa olevasta Run All -komennosta. Kyseiset testit suoritettiin Edit Mode -tilassa, eli ympäristössä, jossa testattavat pelin komponentit eivät vaadi pelin ajamista, eivätkä siten periydy Unityn MonoBehaviour-luokasta. [14.]



Kuva 4. Test Runner -editorityökalu Unityn pelimoottorinäkymässä, jossa kaikki testit ovat läpäisty Edit Modessa.

Kuten esimerkkikoodi 10 havainnollistaa, Play Mode on rakenteeltaan lähes samanlainen kuin Edit Mode, mutta Play Modessa voidaan hyödyntää Unityn korutiinitoiminnallisuutta. Korutiini ei pysäytä ohjelman pääsäiettä, vaan sen suoritus keskeytyy hallitusti ja jatkuu myöhemmin, esimerkiksi tietyn ajan kuluttua. Tätä ominaisuutta voidaan hyödyntää muun muassa tasojen välisissä siirtymissä, joissa kamera voidaan asteittain häivyttää mustaksi ennen Scene-vaihtoa. Play Moden ja Edit Moden eroavaisuuksia on koottu taulukkoon 1. [15.]

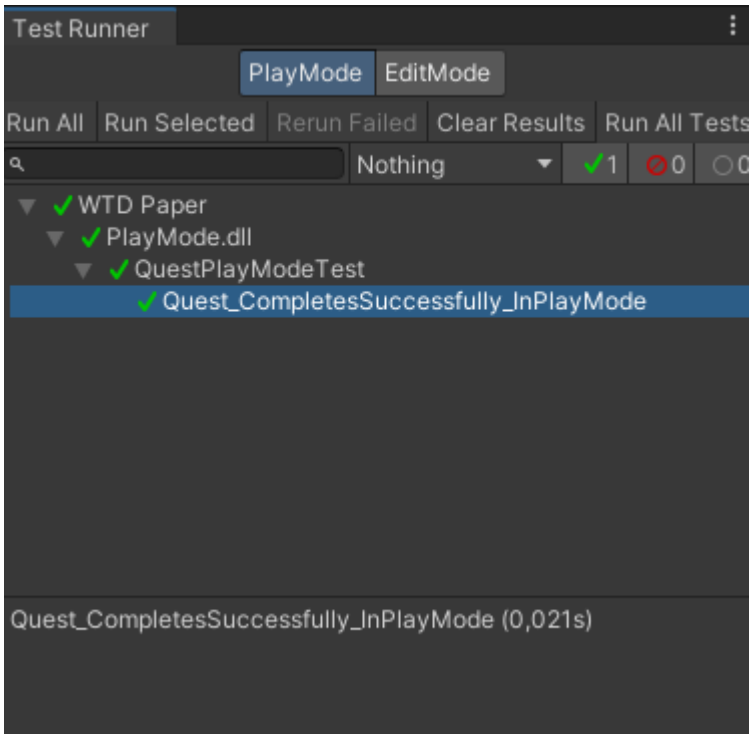
```
using UnityEngine;
using NUnit.Framework;
using Data;

public class QuestPlayModeTest
{
	[UnityTest]
	public IEnumerator Quest_CompleteSuccessfully_inPlayMode()
	{
		var quest = ScriptableObject.CreateInstance<QuestScriptableObject>();
		quest.questID = 1;
		quest.questState = QuestState.Active
		yield return null;
		quest.CompleteQuest();
		Assert.AreEqual(QuestState.Completed, quest.questState);
	}
}
```

Esimerkkikoodi 10. Esimerkkikoodi Test Runnerin Play Mode -koodista. Play Mode on vastaavanlainen kuin Edit Mode, mutta ohjelmisto ajetaan korutiinin sisällä.

Korutiini on metodi, jonka suoritus on jaettu usealle ruudulle siten, että ohjelman muu toiminta jatkuu normaalisti. Yksi ruutu tarkoittaa yksittäistä kuvaa, joka piirretään näytölle pelin päivityssilmukan aikana. Korutiinit ovat hyödyllisiä tilanteissa, joissa halutaan jonkin toiminnon kestävän usean ruudun ajan ilman, että pelin suoritus pysähtyy. Tällaisia käyttökohteita ovat esimerkiksi visuaaliset häivytykset tai käyttöliittymän elementtien, kuten pistetaulukon, asteittainen päivittyminen. [15.]

yield return -komento määrittää kohdan, jossa korutiinin suoritus keskeytetään tilapäisesti. Esimerkkikoodi 10 esimerkin mukaisesti yield return null tarkoittaa, että korutiinin suoritus pysähtyy kuluvan ruudun jälkeen ja jatkuu seuraavan ruudun alussa. [15.]

Kuvassa 5 Play Modessa peli ajetaan normaalisti, ja siinä testataan pelin sisällä tapahtuvia toimintoja. Esimerkiksi voidaan tarkistaa, että MonoBehaviour-luokan perinyt pelaajaobjekti suorittaa hyppymekaniikan oikein, vaikka toiminto kestää tietyn ajan. Tämä käyttäytyminen tapahtuu korutiinin yield return -rakenteiden avulla, joissa voidaan käyttää erilaisia viivästyskomentoja tietyn toiminnon suorittamiseen. [14.]



Kuva 5. Test Runner -editorityökalu Unityn pelimoottorinäkymässä, jossa kaikki testit ovat läpäisty Play Modessa.

Kaikki Test Runneriin liitettävät C#-skriptit tarvitsevat oman kokoonpanomäärittelmän (Assembly Definition). Unity luo oletuksena yhden yhteisen kokoonpanon nimeltä Assembly-CSharp, johon kaikki projektin skriptit sisällytetään. Luomalla omia kokoonpanomääritelmiä kehittäjä voi organisoida ja luokitella koodipohjan selkeämmin. Projektin kasvaessa tällainen rakenne helpottaa ohjelmiston hallintaa ja tekee sen toiminnallisuuksien hahmottamisesta huomattavasti selkeämpää. [14.]

Taulukko 1. Taulukossa on esitetty Test Runner -komponentin [Test]- ja [UnityTest]-attribuuttien keskeiset eroavaisuudet.

Ominaisuus	**[Test]**	**[UnityTest]**
Testin tyyppi	Yksikkötesti	Korutiinipohjainen testi
Suorittamisympäristö	Edit Mode (ei pelin ajoa)	Play Mode (Peli ajetaan)
Sallii yield return	Ei	Kyllä
Käyttökohde	Logiikan ja datan testaukseen	Pelimaailman ja aikaperusteisten toimintojen testaukseen

4 Skaalautuvuus

4.1 Tehtäväjärjestelmän skaalautuvuus

Skaalautuvuus on keskeinen tavoite suurten ja monimutkaisten ohjelmistojen kehityksessä. Ohjelmistoa rakennettaessa on jo projektin alkuvaiheessa otettava huomioon, miten eri ohjelmiston komponentit tulevat skaalautumaan tulevaisuudessa. Kehittäjän on tärkeää muistaa, että vaikka suoraviivainen ja yksinkertainen ratkaisu saattaa tuntua nopealta projektin alkuvaiheessa, se ei välttämättä enää toimi yhtä hyvin silloin, kun ohjelmiston koodipohja kasvaa ja monimutkaistuu. [1, s. 57–59.]

Ohjelmiston skaalautuvuus tarkoittaa sitä, että ohjelmiston suorituskykyä voidaan kasvattaa tai pienentää suhteessa käsiteltävän työn määrään muutoksiin. [16.]

Skaalautuvuus kuvaa ohjelmiston kykyä käsitellä vaihtelevaa kuormitusta siten, että ohjelmiston odotetaan pysyvän vakaana, ja säilyttävän suorituskykynsä silloinkin, kun sen kuormitus kasvaa jyrkästi. Esimerkkejä kuormituksesta voisi olla vaikka käyttäjämäärän kasvu, samanaikaisten pyyntöjen lisääntyminen tai datan määrän kasvu. [16.]

Tehtäväjärjestelmän lähtökohtana oli luoda ratkaisu, jossa kasvavan logiikan ja datan hallinta sekä uusien ominaisuuksien lisääminen eivät hidastaisi kehitystä tai tekisi järjestelmän laajentamisesta tarpeettoman monimutkaista. On tärkeää huomioida, että jos pelin koko kasvaa merkittävästi ja kaikki pelilogiikka keskittyy yhteen quest-tyyppiseen ScriptableObjectiin, on olemassa riski niin sanotun God Class -rakenteen muodostumisesta. Tämä tarkoittaa tilannetta, jossa yksi luokka sisältää liikaa vastuuta ja hallitsee useita pelin osa-alueita, mikä heikentää ohjelmiston ylläpidettävyyttä ja skaalautuvuutta. [17.]

Tällä hetkellä pelissä on yksi ScriptableObject, joka hallinnoi kaikkea tehtäväjärjestelmään liittyvää tietoa. Yksittäiset quest-oliot sisältävät sekä käyttöliittymän (UI) näkymiin liittyvää dataa, että pelin sisäistä logiikka- ja tiladataa. Näiden osa-alueiden määrä ja monimutkaisuus tulevat todennäköisesti kasvamaan pelin kehityksen edetessä, mikä saattaa edellyttää järjestelmän jakamista pienempiin, selkeämmin rajattuihin kokonaisuuksiin.

Tehtävän tilan jakamista voisi kehittää luomalla erillisiä C#-dataolioita, joita QuestScriptableObject hallinnoi. Tällä hetkellä QuestLevel-olio sisältää kaiken tietyn tason datan, mutta muu tieto on suoraan sidottu ScriptableObjectiin. Rakennetta voisi parantaa jakamalla dataa pienempiin kokonaisuuksiin, kuten QuestUI- ja QuestGameplay-olioihin, jotka vastaisivat omista osa-alueistaan ja rajaisivat tietosisällön selkeämmin omiin komponentteihinsa.

4.2 Modulaarisuus: tehtävien lisääminen ja poistaminen

Modulaarisuus on ensimmäinen prioriteetti tehtäväjärjestelmän suunnittelussa, sillä asiakkaan vaatimuksena oli toteuttaa Angry Birds -mallin mukainen peli, jossa on suuri määrä tasoja. Pelin tasot muistuttavat toisiaan rakenteellisesti, mutta eroavat toisistaan muun muassa sprite-grafiikan, tehtäväviestien ja vaikeustason osalta.

Tämän vuoksi järjestelmässä päätettiin käyttää ScriptableObjecteja, sillä ne ovat itsenäisiä että Unityn asset-pohjaisia muuttujia, jotka säilyttävät datan py-

syvästi projektin asset-kansiossa ja joita voidaan hyödyntää useissa eri paikoissa ilman riippuvuuksia muihin komponentteihin [4]. Uusia Quest-tyyppisiä ScriptableObjecteja voidaan luoda suoraan Unityn editorissa ilman erillistä koodia, mikä tekee järjestelmästä helposti laajennettavan ja muokattavan.

QuestManager-luokka hakee kaikki quest-tyyppiset ScriptableObjectit projektin Resources/Quests-kansiosta ja lisää ne listaan, jolloin järjestelmä toimii riippumatta tehtävien määrästä. Tämä mahdollistaa myös tehtävien poistamisen ja lisäämisen ilman, että ohjelmiston logiikkaa tarvitsee muuttaa.

Esimerkkikoodista 11 voidaan nähdä, miten Unityyn tehtiin oma editorilaajennus, joka jakaa kaikille QuestScriptableObjecteille oman id-arvon, sekä nimeää tiedoston nimen siinä järjestyksessä, jossa objektit ovat kansiossa. Esimerkiksi jos listan keskeltä poistaa yhden quest objektin, pitää kaikille generoida uusi tunniste(id) ja tiedosto sekä päivitettävän olion nimi, koska järjestelmä ei enää tunnista järjestystä. Tästä syystä luotiin tämä laajennus, jotta tiedoston nimeäminen ja tunnusgeneraatio olisi vaivatonta.

```
#if UNITY_EDITOR

using UnityEngine;
using UnityEditor;

public class QuestIDAssigner : MonoBehaviour
{
      [MenuItem("Tools/Assign Quest IDs")]
      public static void AssignQuestIDs()
      {
            var quests = Resources.LoadAll<QuestScriptableObject>("Quests");

            for (int i = 0; i < quests.Length; i++)
            {
                  quests[i].questID = i;
                  quests[i].name = $"{i}_Quest";

                  string path = AssetDatabase.GetAssetPath(quests[i]);
                  AssetDatabase.RenameAsset(path, $"{i}_Quest");

                  EditorUtility.SetDirty(quests[i]);
            }

            Debug.Log("Quest IDs assigned successfully!");
      }
}
#endif
```

Esimerkkikoodi 11. Ohjelmistoon kehitetty editorilaajennus, joka määrittää kaikille Quest ScriptableObject -olioille tiedostonimen, olion nimen ja ID-arvon.

Kun ScriptableObjectien määrä kasvaa suureksi (esimerkiksi yli 90 tiedostoa), alkaa virheiden etsiminen ja hallinta olla haastavaa, jos kaikki tiedostot sijaitsevat samassa kansiossa. Tämän ongelman ratkaisemiseksi peliin toteutettiin uusi ScriptableObject-olio, joka toimii tietokantana kaikille pelin Quest-olioille. Tietokanta pitää sisällään kaikki tehtävät ja tarjoaa hakutoiminnon, jolla tehtävätyyppejä voidaan etsiä joko nimen tai indeksin perusteella, mikä helpottaa ylläpitoa ja kehitystyötä. Tietokantaa ei ole vielä integroitu järjestelmään, koska se on vielä kehitysvaiheessa.

```
using UnityEngine;

namespace Data
{
     [CreateAssetMenu]
     public class QuestDatabase : ScriptableObject
     {

          public List<QuestScriptableObject> quests = new();

          public QuestScriptableObject GetQuestByName(string name)
          {
              return quests.Find(q => q.name == name);
          }
          public QuestScriptableObject GetQuestByID(int id)
          {
              return quests.Find(q => q.questID == id);
          }
     }
}
```

Esimerkkikoodi 12. Yksinkertainen esimerkki peliin toteutettavasta tietokannasta.

ScriptableObject-tietokanta sisältää quests-nimisen olion, joka varastoi kaikki QuestScriptableObject-tyyppiset objektit. Tietokantaan voidaan lisäksi toteuttaa erilaisia hakuun perustuvia funktioita.

Tietokanta sisältää tällä hetkellä kaikki pelin quest -oliot, joita voidaan hakea joko nimen tai ID-tunnuksen perusteella. Pelin skaalautuessa suuremmaksi tietokantaan tulisi kuitenkin lisätä enemmän hakulogiikkaa. Esimerkiksi kun järjes-

telmää tulevaisuudessa pilkotaan pienempiin kokonaisuuksiin, olisi hyödyllistä, että hakutoiminto olisi tarkempi ja kohdistuisi vain haluttuun tietotyyppiin.

Tällöin tietokannan hakuparametreihin voitaisiin lisätä ehto, joka määrittää, liittyykö haku QuestUI- vai QuestGameplay-olioiden dataan. Jatkokehityksen myötä peli tullaan jakamaan tehtäväryhmiin (Districts), jolloin tietokantaan tulee lisätä myös näihin alueisiin perustuva hakuominaisuus, jotta tehtävien hallinta ja tiedonhaku pysyvät tehokkaina pelin kasvaessa.

5 ScriptableObjectit ja käyttöliittymä

5.1 Datan siirto ScriptableObjectista QuestManageriin

Pelin ydin pohjautuu Service Locator -arkkitehtuuriin, johon myös QuestManager-luokka on integroitu. Service Locator -suunnittelumallin ideana on ylläpitää keskitettyä rekisteriä, josta voidaan hakea ohjelmiston eri riippuvuuksia ja palveluita. [18.] Peliin toteutetut hallintaluokat (managerit) oli rekisteröity tähän palveluun, koska niiden tarjoamia toiminnallisuuksia tarvitaan useissa eri osissa ohjelmistoa. Tämän ratkaisun ansiosta peli tarvitsee vain yhden Singleton-luokan, joka vastaa palveluiden välittämisestä eteenpäin. [18.] Service Locator -mallin etuna on, että luokkien väliset riippuvuudet vähenevät merkittävästi, sillä niiden ei tarvitse sisältää suoria viittauksia muihin ohjelmiston osiin. [18.]

Ohjelmisto on jaettu kolmeen eri nimiavaruuteen: dataan, logiikkaan ja visuaalisiin komponentteihin. Datakerros vastaa kaiken pelissä tarvittavan tiedon varastoimisesta. Esimerkiksi Player-data-nimiavaruudessa oleva olio voi sisältää pelaajan elämäpisteet. Logiikkakerroksessa sijaitsevat kaikki hallintaluokat. Esimerkiksi PlayerManager-luokka hallinnoi pelaajan dataa ja sisältää siihen liittyvät toiminnot ja metodit.

Visuaalinen kerros puolestaan vastaa käyttöliittymän ja pelin graafisten elementtien päivittämisestä. Esimerkiksi pelaajan health bar -komponentti päivittyy sen mukaan, mikä pelaajan elämäarvo on datakerroksessa. Yleensä visuaali-

nen kerros hakee tarvitsemansa tiedon, tai metodin esimerkiksi PlayerManager-luokan kautta ServiceLocatorin avulla.

Pelin logiikkakerroksessa olevat manager-luokat toteuttavat rajapinnan (interface) eivätkä peri Unityn MonoBehaviour-luokkaa, koska peliin toteutettu ServiceLocator-järjestelmä hallitsee kaikki palvelut erillisinä komponentteina pelin modulaarisuuden säilyttämiseksi. Tästä syystä QuestScriptableObject-oliot alustetaan esimerkkikoodin 13 mukaisesti QuestManager-luokan konstruktorissa Unityn Awake- tai Start-metodien sijaan.

```
public QuestManager()
{
     LoadQuests();

     activeQuestID = PlayerPrefs.GetInt(PP_ACTIVEQUEST_INDEX, 0);
     LoadQuestData();

     if(activeQuestID == 0)
     {
          StartQuest(0);
     }
}
```

Esimerkkikoodi 13. QuestManager-luokan konstruktori, jossa ladataan kaikki listassa olevat QuestScriptable-oliot.

QuestManager lataa konstruktorissaan esimerkkikoodin 14 mukaisesti kaikki datakerroksessa olevat QuestScriptableObjectit listaan käyttämällä Unityn Resources.LoadAll -komentoa. Kaikki QuestScriptableObjectit alustetaan käymällä listan oliot läpi ja kutsumalla niiden Reset()-metodia. Tämä metodi nollaa kaikkien quest-elementtien arvot ja asettaa kaikki muut paitsi ensimmäisen tason lukittuun tilaan.

```
public void LoadQuests()
{
     questList = Resources.LoadAll<QuestScriptableObject>(QUESTS_PATH);
     foreach (QuestScriptableObject quest in questList)
     {
          quest.Reset();
     }

     if (questList == null || questList.Length == 0)
     {
          Debug.LogWarning("No quests found at the specified path!");
     }

}
```

Esimerkkikoodi 14. LoadQuests-metodi, jossa ladataan ScriptableObjektit QuestManagerissa sijaitsevaan listaan.

Tämän jälkeen kutsutaan esimerkkikoodin 15 mukaisesti LoadQuestData()-metodia, joka tarkistaa, onko levylle tallennettu tallennustiedosto olemassa. Jos tiedostoa ei löydy, peli alustetaan alkuasetuksiinsa. Jos tallennustiedosto on olemassa, kutsutaan SaveManager-luokan LoadQuestData()-metodia ja sille välitetään aiemmin ladattu QuestScriptableObject-lista viiteparametrina.

```
public void LoadQuestData()
{
     if(!SaveManager.IsSaveDataAvailable(
          SaveManager.QUEST_SAVE_FILE_PATH)
     {
          DeleteActiveQuestIndex(PP_ACTIVEQUEST_INDEX);
          activeQuestID = 0;
          InitializeQuests();
     }
     else
     {
          SaveManager.LoadQuestData(ref questList);
     }
     Debug.Log(
          "If path exists " +
          SaveManager.IsSaveDataAvailable(
               SaveManager.QUEST_SAVE_FILE_PATH));
}
```

Esimerkkikoodi 15. LoadQuestData-metodi, jossa Quest-tyyppisiin ScriptableObjecteihin ladataan tallennustiedoista saatu data.

SaveManager-luokan LoadQuestData-metodi on varsin yksinkertainen toteutus, joka vastaa kovalevylle tallennetun JSON-muotoisen datan lataamisesta.

Luokassa datan käsittely on toteutettu kahden sisäkkäisen serialisoitavan C#-apuolion avulla, nimeltään QuestData ja LevelData, jotka edustavat tehtävä- ja tasokohtaista dataa esimerkkikoodin 16 mukaisesti.

Kolmantena rakenteena toimii lista, joka sisältää kaikki QuestData-oliot, ja jokainen niistä sisältää vastaavan LevelData-listan. Tämä lista on nimeltään QuestDataList, ja sitä käytetään ainoastaan SaveManager-luokan sisällä.

```
[System.Serializable]
private class QuestDataList
{
     public List<QuestData> quests;
}

[System.Serializable]
public class QuestData
{
     public int questID;
     public int totalExp;
     public bool newMessageIndicator;
     public QuestState questState;
     public List<LevelData> levels;

     [System.Serializable]
     public class LevelData
     {
          public int levelIndex;
          public int levelNumber;
          public int bonesCount;
          public bool isLevelCompleted;
          public bool isLevelActive;
          public int levelHeat;
          public int levelExp;
     }

}
```

Esimerkkikoodi 16. SaveManager-luokassa käytetyt apuluokat tallennusdatan tallentamiseen ja lataamiseen.

Kun aktiivisten tehtävien tilaa päivitetään QuestManager-komponentissa, SaveManager päivittää tallennustiedoston mukaisen JSON-datan apuluokkia hyödyntäen omaan sisäiseen listaansa.

Tiedot päivitetään ensin SaveManagerin sisäiseen listaan, jotta QuestManager voi päivittää QuestScriptableObject-olioiden datan oikein.

Tämän jälkeen SaveManager-luokan LoadQuestData()-metodi synkronoi tallennustiedoston sisällön esimerkkikoodin 17 mukaisesti QuestManagerin ylläpitämään QuestScriptableObject-listaan. Metodi vastaanottaa parametrina viittauksen QuestManager-komponentissa sijaitsevaan QuestScriptableObject-listaan. QuestManagerista käyttöliittymän elementteihin päivitettävä data haetaan questList-nimisestä listasta.

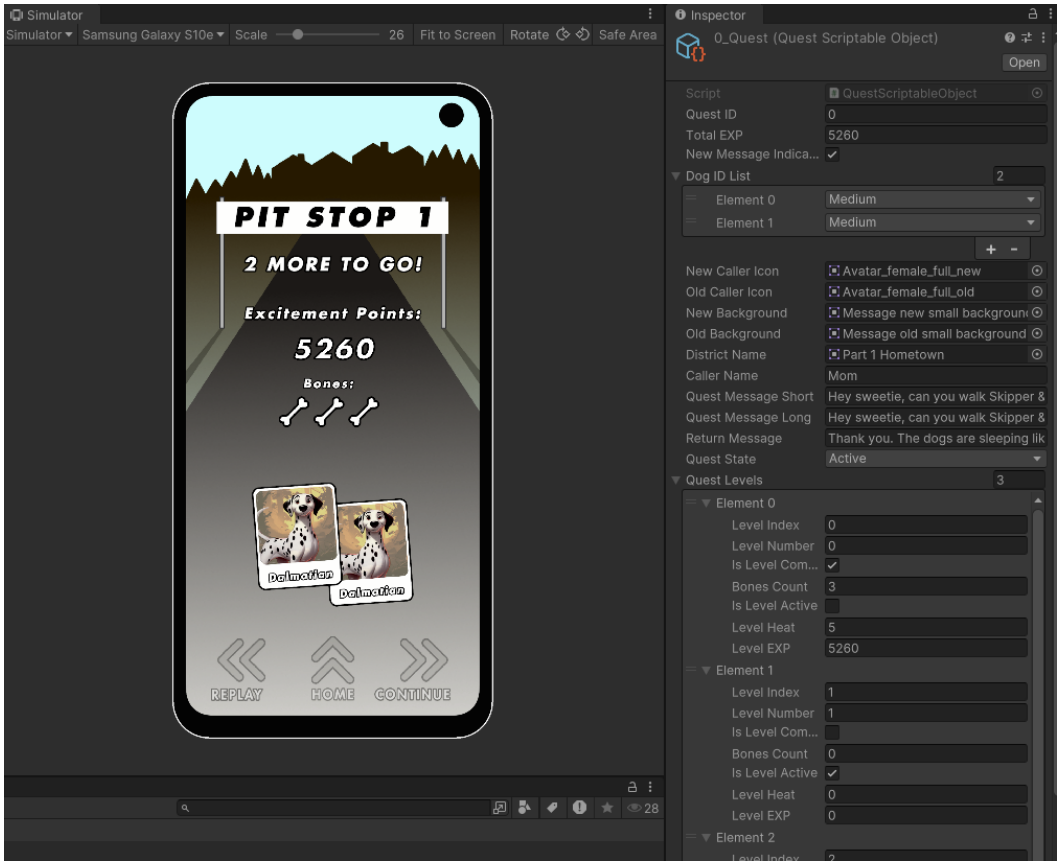
QuestManagerin QuestScriptableObject-lista toimii pelin aktiivisena datarakenteena, joka on suoraan kytköksissä käyttöliittymän ja pelilogiikan komponentteihin.

```
public static void LoadQuestData(
      ref QuestScriptableObject[] questList)
{
      string path =
            Application.persistentDataPath + "/" +
            QUEST_SAVE_FILE_PATH;

      if (File.Exists(path))
      {
            string jsonData =
                  File.ReadAllText(path);

            QuestDataList questDatalist =
                  JsonUtility.FromJson<QuestDataList>(
                        jsonData);

            foreach (QuestData data in questDatalist.quests)
            {
                  var quest =
                        questList.FirstOrDefault(
                              q => q.questID == data.questID);

                  if (quest != null)
                  {
                        quest.questState = data.questState;
                        quest.totalEXP = data.totalExp;
                        quest.newMessageIndicator =
                              data.newMessageIndicator;

                        for (int i = 0; i < data.levels.Count; i++)
                        {
                              QuestData.LevelData levelData =
                                    data.levels[i];

                              QuestLevel level =
                                    quest.GetLevel(
                                          levelData.levelNumber);

                              if (level != null)
                              {
                                    level.levelIndex =
                                          levelData.levelIndex;

                                    level.isLevelCompleted =
                                          levelData.isLevelCompleted;

                                    level.isLevelActive =
                                          levelData.isLevelActive;

                                    level.bonesCount =
                                          levelData.bonesCount;

                                    level.levelHeat =
                                          levelData.levelHeat;

                                    level.levelEXP =
                                          levelData.levelExp;
                              }
                        }
                  }
            }

            Debug.Log("Quest data loaded successfully.");
      }
      else
      {
            Debug.Log("No saved quest data found.");
      }
}
```

Esimerkkikoodi 17. SaveManager-luokassa käytetty LoadQuestData-metodi.

5.2 Esimerkki: Questin tilan päivitys käyttöliittymässä

Kuten kuva 8 havainnollistaa peli on rakennettu siten, että jokaisessa quest-objektissa on kolme tasoa. Tasot on suunniteltu siten, että kun tehtävän aloittaa, tulee kaikki kolme tasoa suorittaa yhtäjaksoisesti. Jokaisen tason läpäisyn jälkeen tulee kuvan 6 mukainen varikkotila, jossa tason voi pelata uudestaan, mennä kotiruututilaan (homescreen) tai jatkaa suoraan seuraavaan tasoon.

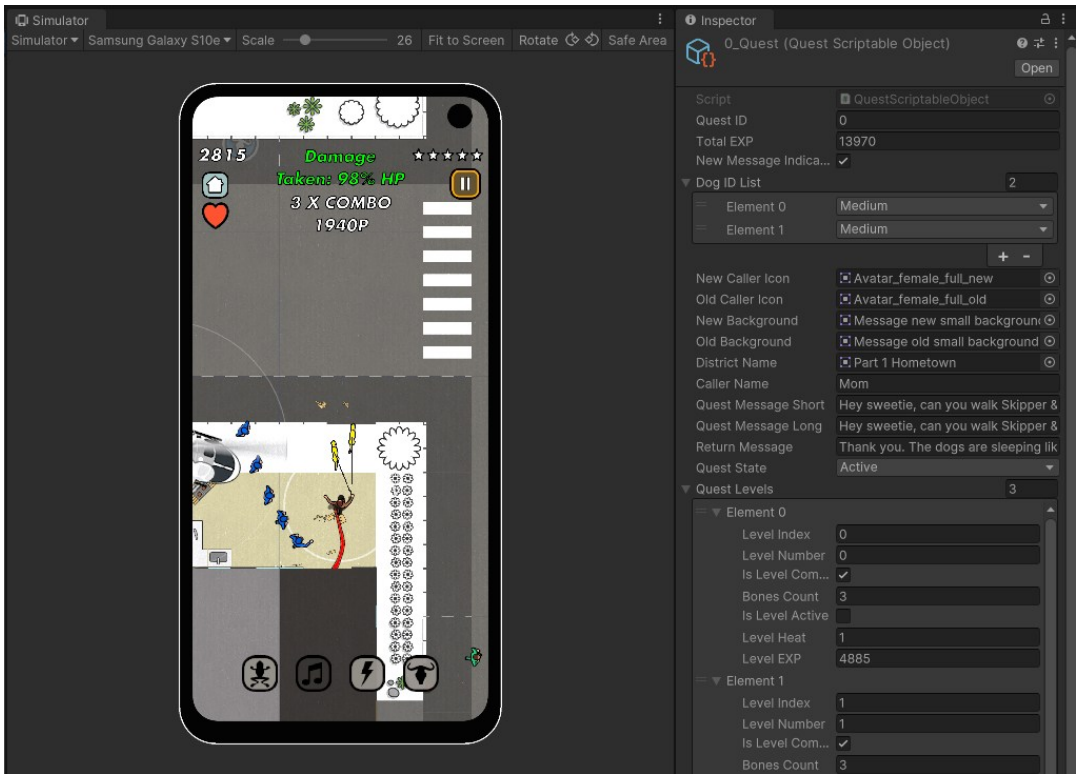


Kuva 6. ScriptableObjectin käyttöliittymäpäivitys ensimmäisen varikon jälkeen.

Pelinäkymässä koirien määrä sekä heat-arvo (eli arvo, joka määrittää, kuinka monta poliisi-NPC-hahmoa eli tietokoneen ohjaamaa pelihahmoa luodaan pelikenttään tason aikana) periytyvät aina tason vaihtuessa, kunnes uusi tehtävä alkaa. Myös kokemuspisteet lasketaan yhteen tason vaihtuessa.

Quest-tyyppisessä ScriptableObjectissa näkyy myös DogIdList-niminen lista, joka sisältää enum-arvoja. Nämä enum-arvot määrittelevät, kuinka monta koiraa

kyseiseen tasoon kuuluu ja mitä koiratyyppejä tasolla käytetään. Tämän listan perusteella pelin visuaaliset komponentit osaavat päivittää käyttöliittymän elementit oikein ja näyttää juuri ne koirat, jotka kuuluvat kyseiseen tasoon kuvan 7 mukaisesti.



Kuva 7. Koirien määrä ja tyyppi ovat päivittyneet pelinäkymään ScriptableObjectien asetusten mukaisesti.

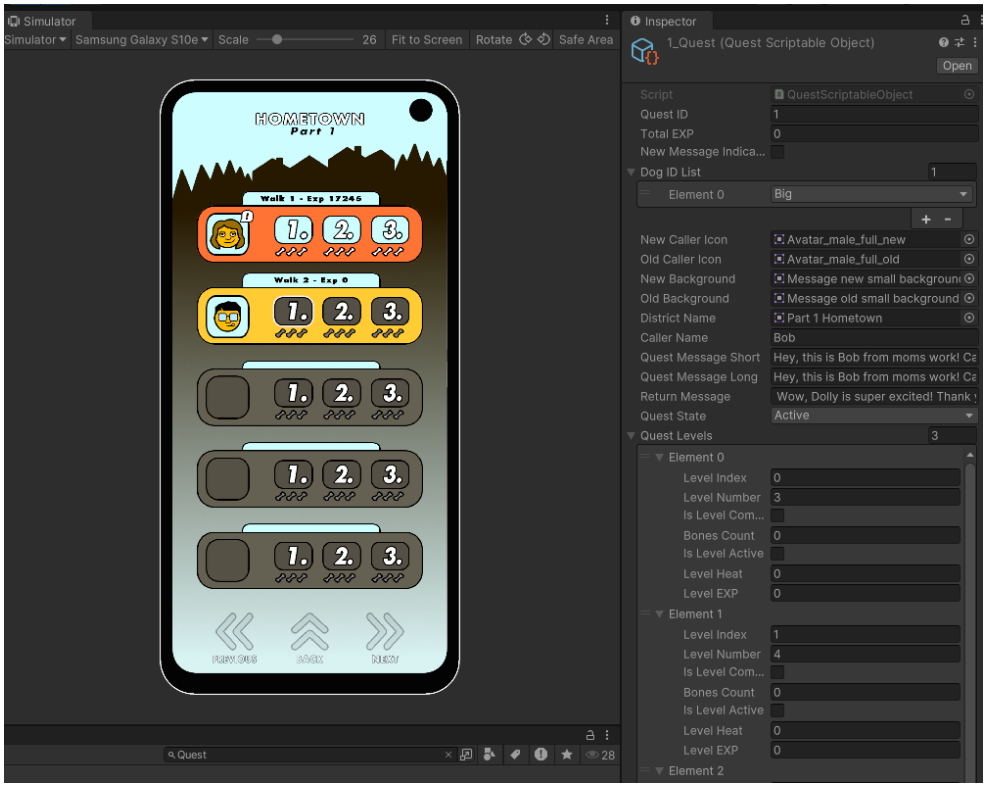
Pelin yläkulmassa oleva kotipainike mahdollistaa siirtymisen takaisin päävalikkonäkymään. Kun painiketta painetaan, peli asetetaan taukotilaan muuttamalla Unityn Time.timeScale-arvoa. [19]. Tämä pysäyttää pelin ajan kulun ja keskeyttää kaikki siihen sidotut toiminnot, mutta käyttöliittymän (UI) elementit pysyvät edelleen toiminnassa.

Esimerkkikoodin 18 mukaisesti ensimmäinen funktio pysäyttää pelin ajan kulun asettamalla timeScale-arvon nollaan. Toinen funktio palauttaa pelin normaalin toiminnan asettamalla timeScale-arvon takaisin arvoon 1.

```
public void PauseGame()
{
    Time.timeScale = 0f; // Pysäyttää pelin
}

public void ResumeGame()
{
    Time.timeScale = 1f; // Jatkaa peliä normaalisti
}
```

Esimerkkikoodi 18. Esimerkki UnityEngine-kirjaston Time.timeScale -muuttujan käytöstä.

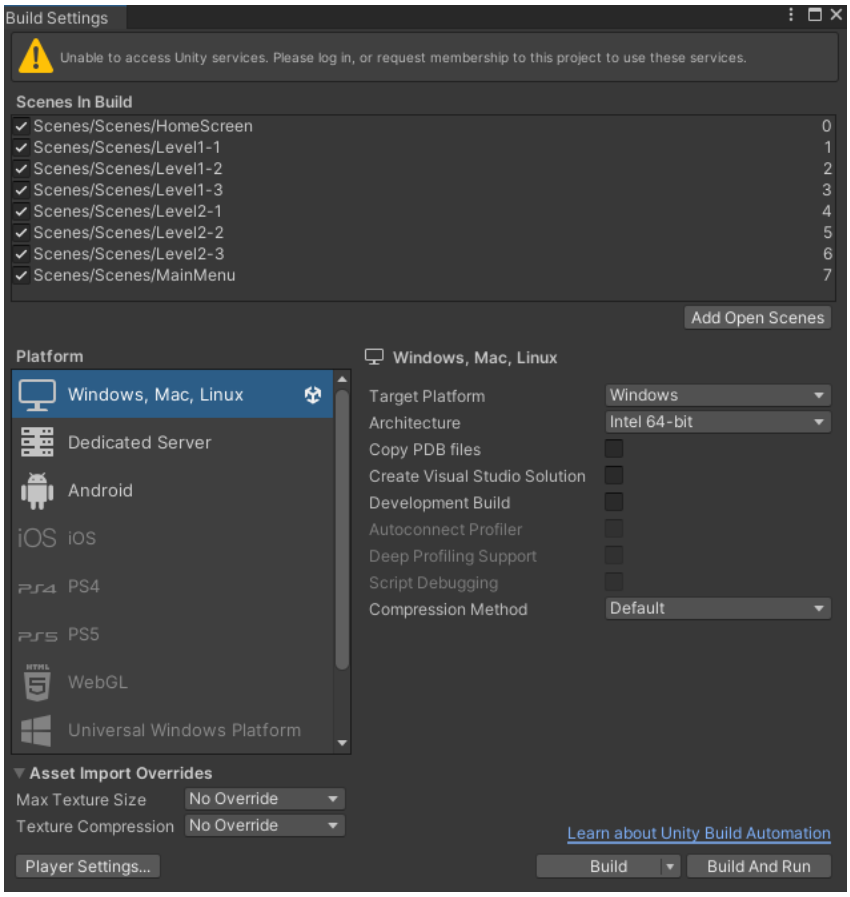
Kun kaikki kolme varikkotilaa on suoritettu, siirrytään takaisin päävalikkonäkymään, jossa pelaaja voi valita seuraavan tason, jossa ulkoilutetaan uutta koiraa. Seuraavan tason valitseminen siirtää järjestelmän käyttämään seuraavaa Quest-tyyppistä ScriptableObjectia. Järjestelmä tarkistaa, onko kaikki tason tehtävät läpäisty, ja jos näin on, aktivoidaan seuraava Quest-tyyppinen ScriptableObject.



Kuva 8. Näkymä, jossa seuraava Quest-tyyppinen Scriptableobject on aktivoitu, ja voidaan siirtyä seuraavaan tasoon.

ScriptableObjectien päivitys tapahtuu aina, kun taso läpäistään. Pelissä käytetään PitStopManager-nimistä komponenttia, joka vastaa nykyisen tason tietojen päivittämisestä ScriptableObject-olioihin. Manageri hyödyntää Unityn SceneManager-luokkaa, jonka avulla se selvittää aktiivisen kohtauksen indeksin. Tätä indeksiä verrataan aktiiviseen Quest-tyyppiseen ScriptableObject-olioon, jolloin voidaan määrittää, mikä taso on suoritettu ja mikä aktivoidaan seuraavaksi.

Unityssä kohtausten järjestys määritetään kuvan 9 mukaisessa Build Settings -näkymässä. Jokaisella kohtauksella on oma numerotunnisteensa, joka kertoo sen sijainnin kohtauslistassa. Koska jokainen pelattava taso on toteutettu omana kohtauksenaan, voidaan kohtauksen indeksiä helposti verrata ScriptableObject-listan järjestykseen. Tämän rakenteen ansiosta ScriptableObjectien päivittäminen ja tason määrittäminen on selkeää ja luotettavaa. [20.]

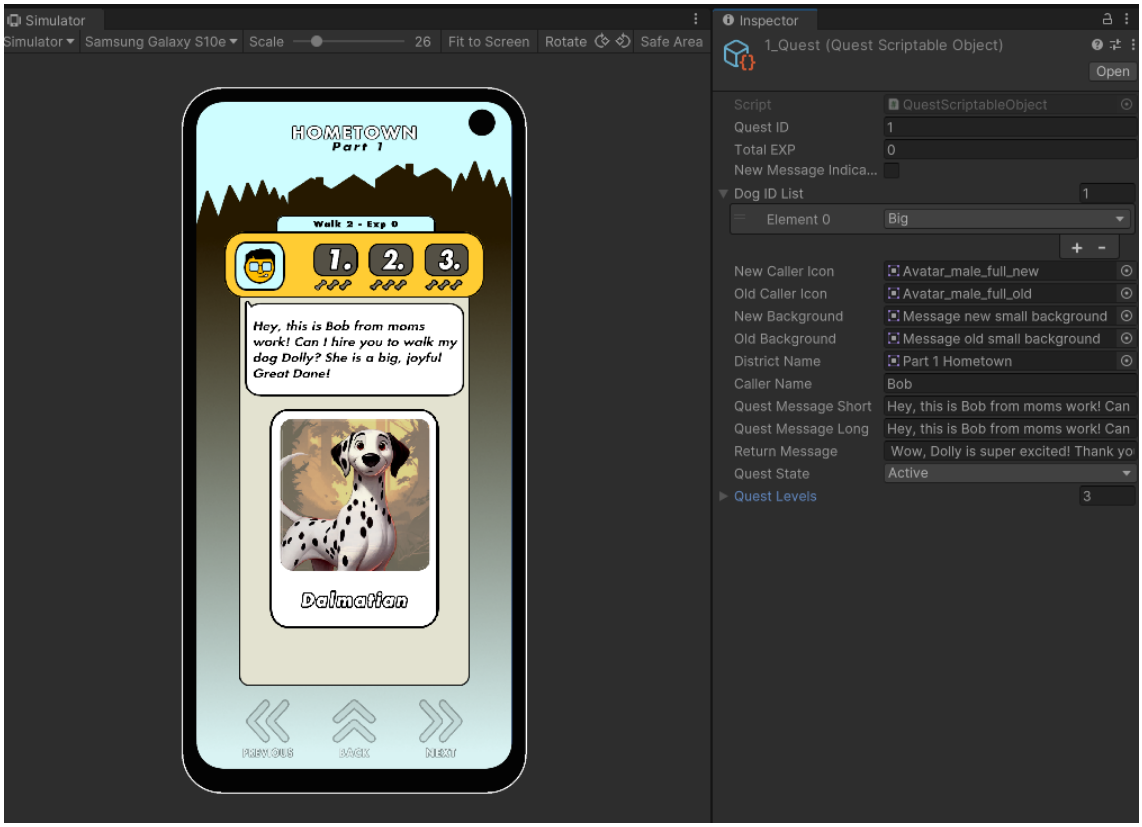


Kuva 9. Unity-pelimoottorin Build Settings -näkymä. Kuvassa näkyy projektin tasorakenne ja käytetty nimeämisprotokolla, joka määrittelee pelikohtausten järjestyksen ja tunnisteet.

PitStopManagerissa on ajastin, joka määrittää kunkin tason keston kohtauskohtaisesti. PitStopManager luo (instansioi) näkymättömän Finish Line -nimisen peliobjektin, ja kun pelaaja osuu siihen, taso päättyy. Tason päättymisen jälkeen kaikki tarvittavat tiedot päivitetään sekä ScriptableObjecteihin että JSON-tallennusdataan. Tämän jälkeen siirrytään näkymään, joka esitettiin kuvassa 6.

Pelissä pelaajalle esitetään useita asiakkaita avatar-kuvakkeina, joista kukin edustaa yksittäistä tehtävää tarjoavaa hahmoa. Pelaaja valitsee tehtävän painamalla haluamaansa avatar-kuvaketta, jolloin kyseiseen asiakkaaseen liittyvä tehtävä avautuu.

Pelissä on tarkoitus simuloida puhelinnäkymää, jossa pelaaja saa viestin asiakkaalta, joka tarvitsee koiranulkoiluttajaa. Jokainen pelissä näkyvä avatar-kuva vastaa yhtä asiakasta. Kuvassa 10 esitetään näkymä, jossa avatar-kuvaketta painettaessa avautuu kyseisen tehtävän viesti-ikkuna. Viestin sisältö haetaan suoraan ScriptableObjecteista, jolloin tieto pysyy ajan tasalla ilman manuaalista päivitystä. Avatar-kuvake on suunniteltu simuloimaan puhelimen viestinäyttöä, jossa pelaaja vastaanottaa uuden työtehtävän eli koiran ulkoilutuksen. Tällä ratkaisulla pyritään vahvistamaan pelin tarinallisuutta ja immersiota. Viesti-ikkunassa näkyy myös kuva ulkoilutettavasta koirasta, ja tämä tieto haetaan niin ikään suoraan ScriptableObjectien datasta.



Kuva 10. Laajennettu näkymä avatar-kuvakkeen painamisen jälkeen. Näytölle avautuu puhekupla, jossa näkyy ScriptableObjectista haettu tehtäväviesti sekä ulkoilutettavan koiran kuva.

Angry Birds -etenemismallin mukaisesti aikaisempia tasoja voidaan pelata uudelleen ja yrittää parantaa aiempaa tulosta. Uudet tasot avautuvat kuitenkin pelattavaksi vasta, kun viimeisin avattu taso on läpäisty. Kun kaikki kolme tasoa

on suoritettu, QuestManager päivittää kyseisen tehtävän QuestState-enumeroinnin tilaan Completed ja asettaa seuraavan Quest-tyyppisen ScriptableObjectin tilaan Active.

6 ScriptableObjectit ja tallennusjärjestelmät

Toteutettuun peliin valittiin JSON-pohjainen tallennusratkaisu, koska se on nopea ja yksinkertainen tapa tallentaa pelin tilaan liittyvää dataa. UnityEngine-kirjasto tarjoaa sisäänrakennetun JsonUtility-komponentin, jonka avulla data voidaan serialisoida ja tallentaa paikallisesti JSON-muodossa. Tämän ansiosta peliin voitiin toteuttaa kevyt tallennusjärjestelmä, joka lukee tallennetut tiedot paikallisesta JSON-tiedostosta sovelluksen uudelleenkäynnistyksen yhteydessä.

Projektin nopeatempoisen luonteen vuoksi tavoitteena oli toteuttaa aluksi toimiva demonstraatio, jota voidaan esitellä varhaisessa vaiheessa. Tästä syystä paikallinen tallennus JSON-formaatissa oli perusteltu ja tehokas ratkaisu.

Paikallisen tallennuksen merkittävin rajoite on se, että peli toimii ainoastaan yksinpelinä. Tallennettu data sijaitsee paikallisena JSON-tiedostona laitteen tallennustilassa eikä ole yhteydessä verkkoon. Pelin jatkokehityksessä on tarkoitus toteuttaa monetisointi, eli pelin kaupallinen ansaintamalli, kosmeettisten ostojen sekä kertamaksullisen latauksen muodossa. Lisäksi peliin suunnitellaan kaikkien pelaajien välinen tulostaulukko. Näiden ominaisuuksien toteuttaminen edellyttää tulevaisuudessa palvelinpohjaista tietokantaratkaisua.

Kovakoodattua tallennusratkaisua ei valittu, koska peli noudattaa Angry Birds -tyyppistä etenemismallia, jossa pelaajan eteneminen ja saavutukset on säilytettävä pelisessioiden välillä. Kovakoodatussa ratkaisussa kaikki pelin arvot palautuvat oletustilaan sovelluksen sulkemisen yhteydessä, mikä ei vastaa projektin tavoitteita.

6.1 Kovakoodattu

Kovakoodaus tarkoittaa sitä, että kaikki ohjelmiston data on ohjelmoitu suoraan lähdekoodiin. Tämä ei varsinaisesti ole varsinaista tallentamista, vaan pelin tila palautuu aina lähtöasetuksiinsa, kun peli käynnistetään uudelleen. Kovakoodattuja arvoja voidaan muuttaa vain muokkaamalla ohjelmakoodia manuaalisesti ja kääntämällä ohjelma uudelleen. [21.]

Kovakoodausta voidaan joissain tapauksissa pitää hyväksyttävänä ratkaisuna, erityisesti silloin, kun data pysyy varmasti muuttumattomana. Tällaisissa tilanteissa datan käsittely on myös tehokkaampaa, koska tietoa ei tarvitse hakea erillisestä tallennustiedostosta tai tietokannasta. [21.]

Kovakoodaus on lähes aina huono ohjelmistokäytäntö. Hyvin suunnitellussa ohjelmistossa data ja logiikka tulisi kapseloida omiin erillisiin komponentteihinsa, jotta ohjelmiston kasvaessa ja skaalautuessa tiedon hallinta ja ylläpito pysyisivät selkeinä. Tällä tavoin voidaan myös välttää niin sanottu spagettikoodi, jossa ohjelman rakenne muuttuu vaikeaselkoiseksi ja virheherkäksi. [21.]

Esimerkkikoodi 19 esittää miten Unity-kirjaston Start-metodissa pelaajan elämä ja pisteet tulostetaan pelimoottorin konsoliin. Update-metodin sisällä on ehtolause, jossa välilyöntiä painamalla pelaajan pisteet kasvavat kymmenellä. Kun peli suljetaan ja käynnistetään uudelleen, pelaajan pisteet palautuvat kovakoodattuun arvoon tässä tapauksessa nollaan.

```
using UnityEngine;

Public class GameManager : Monobehaviour

{
     private int playerHealt = 100;
     private int playerScore = 0;

     void Start()
     {
          Debug.Log(”Pelaajan elämä: ” + playerHealth);
          Debug.Log(”Pelaajan pisteet” + playerScore);
     }

     void Update()

     {
          if(Input.GetKeyDown(KeyCode.Space))
          {
               playerScore += 10;
               Debug.Log(“Uudet pisteet: ”+ playerScore);
          }
     }

}
```

Esimerkkikoodi 19. Kovakoodattu esimerkki playerHealth- ja playerScore-muuttujista.

6.2 JSON- ja XML-serialisointiformaatit

XML ja JSON ovat molemmat datan serialisointiformaatteja, joita käytetään tiedon siirtämiseen sovellusten välillä. Tyypillisiä käyttökohteita ovat datan vaihto päätelaitteen ja palvelimen välillä sekä konfiguraatiotiedostojen tallentaminen. [22.]

Kun data halutaan esittää muodossa, jota sekä ihmiset että koneet voivat lukea, käytetään rakenteista tekstiformaattia, kuten JSONia tai XML:ää. Tallentamisesta vastaava ohjelmisto kirjoittaa datan jompaankumpaan formaattiin paikalliseksi tekstitiedostoksi tai palvelimelle. Vastaanottava sovellus voi myöhemmin lukea ja jäsentää tiedon suoraan tallennetusta JSON- tai XML-tiedostosta.

JSON tulee sanoista JavaScript Object Notation. JSON käyttää datan rakentamiseen avain–arvopareja. [22]. Esimerkiksi peliin toteutettu tallennusjärjestelmä, joka hyödyntää JSON-formaattia, tallentaa quests-nimisen taulukon.

Tämä taulukko sisältää kaikki tiettyyn tehtävään liittyvät tiedot avain-arvoparein.

Esimerkkikoodin 20 mukaisesti JSON-rakenteessa avain on aina merkkijono ja arvo on siihen liitetty tallennettava data. Esimerkiksi rakenteessa "questID": 0 merkkijono questID on avain ja arvo 0 on kyseiseen avaimeen liitetty arvo

```
{
"quests":
     [
          {
          "questID":0
          }
     ]
}
```

Esimerkkikoodi 20. Pelkistetty esimerkki JSON-kielen avain-arvoparirakenteesta.

XML tulee sanoista Extensible Markup Language. Toisin kuin JSON, XML käyttää puurakennetta, joka muistuttaa HTML-ohjelmointia. [22]. Jos esimerkkikoodi 20 olisi XML-kieltä, niin se näyttäisi esimerkkikoodi 21:ltä.

```
<quests>
     <quest>
     <questID>0</questID>
     </quest>
</quests>
```

Esimerkkikoodi 21. Pelkistetty esimerkki XML-kielen puurakenteesta.

Projektissa päädyttiin käyttämään JSON-formaattia paikalliseen tallennukseen, koska Unity-pelimoottorin UnityEngine-kirjasto tarjoaa JsonUtility-moduulin. Vaikka JSON ei tue olioita suoraan, JsonUtility pystyy muuntamaan oliot taulukoiksi ja tallentamaan ne JSON-muodossa paikallisesti. Päinvastainen muunnos tehdään tiedostoa luettaessa.

UnityEnginen JsonUtility on lisäksi optimoitu datan siirtoon, mikä tarkoittaa nopeaa lukemista ja kirjoittamista levyltä. [23]. Tämä on tärkeää, koska peli tallentaa pelin tilaa automaattisesti jokaisen tason läpäisyn jälkeen pelaajan huomaa-

matta. Scene-vaihdosten on oltava mahdollisimman nopeita ja suoraviivaisia pelin nopeatempoisen luonteen vuoksi.

XML-formaatti serialisoi dataa huomattavasti hitaammin verrattuna JSON-muotoon. Sitä ei ole alun perin suunniteltu nopeaan datan käsittelyyn, kuten esimerkiksi API-funktiokutsuihin tai palvelimelle ladattavaan pelidataan. API-funktiokutsu tarkoittaa tilannetta, jossa yksi ohjelmisto pyytää toiselta ohjelmistolta tiettyä toimintoa tai dataa ennalta määritellyn rajapinnan kautta, ja saa vastauksena pyydetyn tiedon. XML soveltuu paremmin monimutkaisiin dokumenttirakenteisiin, joissa datan luettavuus ja rakenteellisuus ovat tärkeämpiä kuin suorituskyky ja nopeus. [22.]

Kuva 11 havainnollistaa, miten QuestManager komponentin ylläpitämä lista kaikkien tehtävien tilasta on tallennettu JSON-formaatissa levylle. JSON-tallennus paikallisesti on hyvin yksinkertainen ja tehokas tapa tallentaa pelin tilaa, ja sopii hyvin tähän asiakasprojektiin toteutetun pelin vaatimuksiin.

Kuva 11. Pelin Quest-lista tallennettuna JSON-formaatissa paikallisesti levylle.

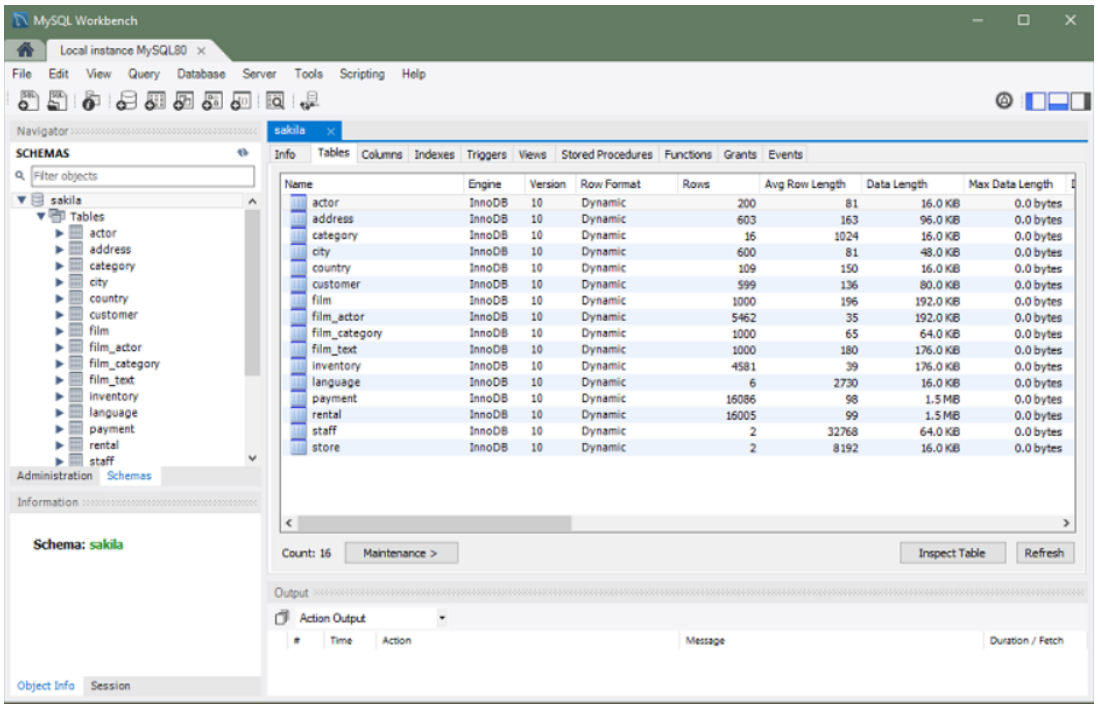
Tällainen toteutus on helppo toteuttaa, mutta rajoituksena on se, että tallennusjärjestelmä on rakennettu täysin yksinpelattavaan muotoon. Se sopii yksinkertaisiin pienen mittakaavan peleihin hyvin, mutta järjestelmän kasvaessa ja monimutkaistuessa tarvitaan hienostuneempia datan varastointitekniikoita. Tulevaisuudessa, mikäli toimeksiantajalle ilmenee tarve toteuttaa esimerkiksi kaikkien

pelaajien yhteinen tulostaulukko, tätä ei voida toteuttaa paikallisella JSON-tyyppisellä tallennusjärjestelmällä.

6.3 Tietokannat ja palvelimet

Tähän asti käsitellyissä tallennusratkaisuissa on ilmennyt merkittäviä rajoitteita erityisesti skaalautuvuuden näkökulmasta. Kuvitellaan esimerkiksi, että peliin haluttaisiin lisätä moninpelio minaisuus, maailmanlaajuinen tulostaulukko tai kosmeettisten päivitysten kauppa. Vaikka peliä ei edes suunniteltaisi verkkoyhteyttä varten, datan määrä kasvaa luonnollisesti pelin laajentuessa. Tällöin kaikki tieto jouduttaisiin tallentamaan yhä suurempiin JSON-tiedostoihin, mikä kuluttaisi resursseja ja tekisi järjestelmästä hitaamman ja vaikeammin hallittavan. Lopputuloksena pelin kehitys muuttuisi epäjohdonmukaiseksi ja monimutkaiseksi, koska tallennetun datan määrä kasvaisi liian suureksi paikallista JSON-pohjaista järjestelmää varten.

Tätä ongelmaa varten voidaan käyttää tietokantoja ja niiden hallintajärjestelmiä (engl. Database Management System, DBMS). Relaatiotietokannat järjestävät datan tauluihin, joissa tiedot ovat sarakkeissa ja riveissä. Tämän järjestelymallin ansiosta tiedon haku, muokkaus, päivitys ja hallinta on helppoa ja suoraviivaista. Tietokantojen hallinnassa käytettävä ohjelmointikieli on yleensä SQL (Structured Query Language), jonka avulla data voidaan organisoida ja käsitellä suoraan koodista, yleensä palvelimen kautta. [24.]



Kuva 12. Esimerkki MySQL-tietokannan taulukkonäkymästä [25].

Ei ole suositeltavaa siirtää dataa suoraan Unity-pelimoottorista tietokantaan, vaan tiedonsiirron väliin tarvitaan palvelinohjelmisto. Keskeisin tällainen on Node.js-palvelin. Node.js on avoimen lähdekoodin monialustainen ajoympäristö, joka suorittaa JavaScript-koodia. [26]. Node.js käyttää asynkronista, tapahtuma-pohjaista mallia, joka mahdollistaa useiden samanaikaisten yhteyksien käsittelyn. [26]. Palvelin muodostaa yhteyden pelin ja tietokannan välillä. WebSocket on yksi monista tiedonsiirtoprotokollista, joka mahdollistaa kahden eri tietokoneen välisen kommunikoinnin reaaliajassa. Tämän tekniikan ansiosta edellä mainitut rajoitteet voidaan kiertää. Esimerkkikoodi 22 havainnollistaa Node.js palvelimen luontia.

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 }, () => {
     console.log('server started');
});

wss.on('connection', function connection(ws) {

     ws.on('message', (data) => {
          console.log('data received \n %o', data);
          ws.send(data);
     });

});

wss.on('listening', () => {
     console.log('listening on 8080');
});
```

Esimerkkikoodi 22. Esimerkki Node.js-pohjaisesta WebSocket-palvelimesta. [27].

Kun palvelin on luotu, pelin tulee muodostaa siihen yhteys. Unity tarjoaa tähän useita valmiita kirjastoja, kuten WebSocketSharp-kirjaston. [27]. Kirjaston avulla peli ja palvelin voivat kommunikoida keskenään ja siirtää dataa, mukaan lukien kokonaisia JSON-tiedostoja. Esimerkkikoodi 23 havainnollistaa, miten Unityn peliobjekti luo yhteyden WebSocket-palvelimeen

```
using UnityEngine;
using WebSocketSharp;

public class WsClient : MonoBehaviour
{
     WebSocket ws;

     private void Start()
     {
          ws = new WebSocket("ws://localhost:8080");
          ws.Connect();

          ws.OnMessage += (sender, e) =>
          {
               Debug.Log(
               "Message Received from " +
               ((WebSocket)sender).Url + ", Data : " + e.Data
               );
          };
     }

     private void Update()
     {
          if (ws == null)
          {
               return;
          }

          if (Input.GetKeyDown(KeyCode.Space))
          {
               ws.Send("Hello");
          }
     }
}
```

Esimerkkikoodi 23. Esimerkki Unity-komponentista, joka muodostaa yhteyden WebSocketin avulla. Välilyönnin painaminen lähettää palvelimelle viestin "Hello" [27].

WebSocket-palvelin välittää ja hakee dataa tietokannasta luomalla siihen yhteyden. Esimerkkikoodin 24 mukaan saveData lisää SQL-kyselyn avulla pelaajan pisteet tietokantaan, ja getData hakee tiedot palauttaen listan kaikista pisteistä asiakkaalle (client).

```
async function saveData(player, score) {
     const sql = "INSERT INTO scores (player, score) VALUES (?, ?)";
     await db.query(sql, [player, score]);
     console.log("Data saved:", player, score);
}

async function getData() {
     const [rows] = await db.query("SELECT * FROM scores");
     return rows; // returns an array of objects
}
```

Esimerkkikoodi 24. Node.js-esimerkki, jossa palvelin tallentaa ja hakee pelaajan pisteitä tietokannasta.

Asiakasprojektiin tarkoitetussa pelissä ei yleensä ole tarpeen rakentaa omaa palvelin- ja tietokanta-arkkitehtuuria, koska se sisältää paljon tietoturvariskejä ja vaatii työlästä ylläpitoa. Tällaiset ratkaisut ovat yleensä tarpeen vain laajoissa projekteissa, joissa on spesifisiä vaatimuksia, esimerkiksi suurissa peliyrityksissä. Pienissä itsenäisissä peleissä on usein järkevämpää hyödyntää ns. BaaS (Backend as a Service) -malleja.

BaaS on palvelumalli, jossa pelin taustajärjestelmät ovat valmiiksi toteutettuja ja palveluntarjoajan ylläpitämiä. Se on tietoturvallinen ja käytännöllinen vaihtoehto erityisesti pienille indie-pelifirmoille. Jos peli sisältää mikromaksuja, joiden kautta pelaajat maksavat pelin sisällöstä, tällaisen palvelun käyttö on usein järkevin ratkaisu.

Palvelussa peli yhdistetään taustajärjestelmäpalveluntarjoajan palvelimiin ohjelmointirajapinnan (API, Application Programming Interface) kautta tehtävien kutsujen avulla. API-kutsut mahdollistavat ohjelmien välisen kommunikoinnin. Palvelut tarjoavat työkaluja tietokantojen hallintaan sekä pilvipalveluita. Ne sisältävät ajantasaisen tietoturvan ja ovat luotettavia, mikäli ne on hankittu luotettavalta toimittajalta, kuten Googlelta tai Unityn omista pilvipalveluista. [28.]

Tulevaisuudessa, kun tallennusjärjestelmä siirretään paikallisesta tallennuksesta palvelimeen, todennäköisimmät vaihtoehdot ovat Googlen Firebase tai Unity Cloud Code.

7 Yhteenveto

Tässä opinnäytetyössä suunniteltiin ja toteutettiin modulaarinen ja skaalautuva tehtäväjärjestelmä Unity-pelimoottoria hyödyntävään mobiilipeliin. Työ toteutettiin osana asiakasprojektia, jonka tavoitteena oli kehittää nopeatempoinen rampage-tyylinen peli, jossa pelin eteneminen perustuu useisiin lyhytkestoisiin ja toisiaan muistuttaviin tasoihin.

Tehtäväjärjestelmän toteutustavaksi valittiin Unityn ScriptableObject-pohjainen arkkitehtuuri, jonka avulla tehtäviin liittyvä data voitiin kapseloida omiin komponentteihinsa ja hallinnoida suoraan Unityn Inspector-näkymässä. Tämä ratkaisu mahdollisti uusien tehtävien lisäämisen ja poistamisen ilman merkittäviä muutoksia koodipohjaan, mikä tukee pelikehityksen aikaista joustavuutta, skaalautuvuutta ja modularisuutta. ScriptableObjectit toimivat keskitettyinä datavarastoina, joita hyödynnetään sekä pelin logiikassa että käyttöliittymässä.

Järjestelmän hallintaa varten toteutettiin QuestManager-niminen komponentti, joka liitettiin osaksi ohjelmiston Service Locator -arkkitehtuuria. Tämän rakenteen ansiosta pelin eri osat pystyvät kommunikoimaan keskenään ilman suoria riippuvuuksia, mikä parantaa koodin ylläpidettävyyttä ja testattavuutta. Pelin tilan tallennus toteutettiin paikallisella JSON-pohjaisella tallennusjärjestelmällä, joka hyödyntää Unityn JsonUtility-kirjastoa. Tallennus tapahtuu automaattisesti pelaajan huomaamatta, ja se soveltuu hyvin yksinpelattavaan demoversioon pelistä.

Työssä tarkasteltiin myös vaihtoehtoisia tallennusratkaisuja, kuten tietokantapohjaisia järjestelmiä ja palvelinarkkitehtuureja, sekä niiden soveltuvuutta laajempaan ja monimutkaisempaan pelikehitykseen. Tämän tarkastelun perusteella todettiin, että nykyinen toteutus vastaa projektin tarpeita, mutta tulevaisuudessa järjestelmää voidaan laajentaa esimerkiksi Backend as a Service -ratkaisujen avulla.

Työn lopputuloksena syntyi toimiva, helposti laajennettava ja kehitystä tukeva tehtäväjärjestelmä, joka täyttää asiakasprojektin vaatimukset ja toimii hyvänä pohjana pelin jatkokehitykselle.

Lähteet

1 Martin, Robert. 2017. Clean Architecture: A Craftman’s Guide to Software Structure and Design. Prentice Hall.

2 ScriptableObject. Verkkoaineisto. Unity documentation. <https://docs.unity3d.com/6000.2/Documentation/Manual/class-ScriptableObject.html>. Luettu 20.9.2025.

3 JsonUtility. Verkkoaineisto. Unity documentation. <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/JsonUtility.html>. Luettu 21.9.2025.

4 Hipple, Ryan. 2020. Three ways to architect your game with ScriptableObjects. Verkkoaineisto. Unity. <https://unity.com/how-to/architect-game-code-scriptable-objects>. Luettu 21.9.2025.

5 What Is An ’Instance’ Anyway? 2023. Verkkoaineisto. Archios. <https://www.youtube.com/watch?v=SiE1ADHqMXM>. Katsottu 28.10.2025.

6 C# Classes and Objects. Verkkoaineisto. W3Schools <https://www.w3schools.com/cs/cs_classes.php>. Luettu 30.10.2025.

7 Monobehaviour. Verkkoaineisto. Unity documentation. <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/MonoBehaviour.html>. Luettu 30.10.2025.

8 Static Classes and Static Class Members (C# Programming Guide). Verkkoainesto. Microsoft. <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members>. Luettu 23.9.2025.

9 Understanding the Singleton Pattern in C# and Unity. 2024. Verkkoaineisto. Medium. <https://medium.com/%40Code_With_K/understanding-the-singleton-pattern-in-c-and-unity-f5abd1ab80bb>. Luettu 23.9.2025.

10 Game Design Pattern- Using Singletons in Unity. 2022. Verkkoaineisto. Medium. <https://medium.com/codex/game-design-pattern-using-singletons-in-unity-acbd05d8ac9d>. Luettu 25.9.2025.

11 Gokdemir, Taha. 2023. Design Patterns for Unity Developers: Service Locator. Verkkoaineisto. Medium. <https://medium.com/@taha.m.gokdemir/design-patterns-for-unity-developers-service-locator-124cd4628c43>. Luettu 20.11.2025.

12 Introduction to Scriptable Objects – February 2016 Live Training. 2016. Verkkoaineisto. Unity learn. <https://learn.unity.com/tutorial/introduction-to-scriptable-objects>. Luettu 29.9.2025.

13 Korpinen, Mikael. 2020. Yksikkö- ja integraatiotestaus Unity-pelimoottorilla. Insinöörityö. Metropolia Ammattikorkeakoulu. Theseus-tietokanta.

14 Unit Tests in Unity. 2020. Verkkoaineisto. Infallible Code. <https://www.youtube.com/watch?v=PDYB32qAsLU>. Katsottu 4.10.2025.

15 Unity documentation Write and run coroutines. Verkkoaineisto. Unity documentation. <https://docs.unity3d.com/6000.2/Documentation/Manual/Coroutines.html>. Luettu 4.10.2025.

16 Alkhaldi, Nadejda. 2023. What is software scalability, and why should your company take it seriously? Verkkoaineisto. ITRex Group. <https://itrexgroup.com/blog/what-is-software-scalability/>. Luettu 4.10.2025.

17 Why the God Class Anti-Pattern Destroys Maintainability. 2024. Verkkoaineisto. Geekific. <https://www.youtube.com/watch?v=J8e3tZcOhGU&t=6s>. Katsottu 13.10.2025.

18 Mikoski, Bruno. 2020. Functional Unity Architecture: A Developer's Guide. Verkkoaineisto. Medium. <https://medium.com/@bada/functional-unity-architecture-a-developers-guide-359e5111c5c2>. Luettu 17.10.2025.

19 Time.timeScale. Verkkoaineisto. Unity documentation. <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Time-timeScale.html>. Luettu 6.11.2025.

20 Build Settings. Verkkoaineisto. Unity documentation. <https://docs.unity3d.com/540/Documentation/Manual/BuildSettings.html>. Luettu 10.11.2025.

21 Hard code explained, and why it is so bad? Verkkoaineisto. appleute. <https://www.appleute.de/en/app-developer-library/hardcoding-meaning/>. Luettu 13.11.2025.

22 What's the Difference Between JSON and XML? Verkkoaineisto. AWS. <https://aws.amazon.com/compare/the-difference-between-json-xml/>. Luettu 23.11.2025.

23 JSON Serialization. Verkkoaineisto. Unity documentation. <https://docs.unity3d.com/2020.1/Documentation/Manual/JSONSerialization.html>. Luettu 23.11.2025.

24 What Is a Database? Verkkoaineisto. Oracle. <https://www.oracle.com/database/what-is-database/#database-defined>. Luettu 26.11.2025.

25 8.2.3 Schema and Table Inspector. Verkkoaineisto. MySQL. <https://dev.mysql.com/doc/workbench/en/wb-develop-object-management-inspector.html>. Luettu 26.11.2025.

26 About Node.js. Verkkoaineisto. nodejs. <https://nodejs.org/en/about>. Luettu 26.11.2025.

27 Schrute, Ahmed. 2020. Unity & NodeJS. Verkkoaineisto. Medium. <https://medium.com/unity-nodejs/websocket-client-server-unity-nodejs-e33604c6a006>. Luettu 26.11.2025.

28 What is BaaS? Backend as a Service vs serverless. Verkkoaineisto. Cloudflare. <https://www.cloudflare.com/learning/serverless/glossary/backend-as-a-service-baas/>. Luettu 26.11.2025.