Tampere University of Applied Sciences

# Automated Testing with Unity

Asmo Jurvanen

BACHELOR'S THESIS
April 2023

Degree Programme in ICT Engineering
Software Engineering

**ABSTRACT**

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

JURVANEN, ASMO:
Automated Testing with Unity

Bachelor's thesis 41 pages
April 2023

---

This thesis focuses on the benefits of using automated testing in game development with Unity. It also provides practical guidance how to include automated testing in development workflow. The thesis starts by exploring game development in general. It then delves into the different types of testing that are available, including unit testing, integration testing and usability testing.

The thesis covers the concept of test-driven development (TDD), explaining how it works and why it can be a valuable approach to developing games. A practical example is provided to demonstrate how to create a simple game in Unity using TDD. GitHub was used for version control.

The thesis also displays a practical implementation of continuous integration/continuous delivery (CI/CD) pipelines for Unity games using GitHub Actions. The game example is used to demonstrate the implementation. It shows the possible benefits of this approach, such as reducing the risk of errors during deployment and improving the efficiency of the development process.

Overall, this thesis provides an overview of automated testing in Unity and practical guidance for developers looking to improve the quality and efficiency of their game development processes.

---


Key words: Unity, automated testing, game development

**TIIVISTELMÄ**

Tampereen ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma
Ohjelmistotekniikka

JURVANEN, ASMO:
Automaattinen testaus Unity-pelimoottorissa

Opinnäytetyö 41 sivua
Huhtikuu 2023

---

Opinnäytetyössä tutkitaan automaattista testausta Unity-pelimoottorissa ja miksi se on kannattavaa pelinkehityksessä. Työssä myös tarjotaan käytännön ohjeita automaattisen testauksen liittämiseen osaksi pelinkehitys prosessia. Aluksi esitellään pelinkehityksen osa alueita. Sen jälkeen työssä käsitellään erilaisia testityyppejä kuten yksikkötestit, integraatiotestit ja käytettävyystestit.

Opinnäytetyö kattaa testiohjatun ohjelmistokehityksen käsitteen (TDD), selittää miten se toimii ja miksi se voisi olla arvokas lähestymistapa pelinkehityksessä. Lisäksi työssä esitellään käytännön esimerkki yksinkertaisen pelin luomiseen Unity-pelimoottorissa testiohjatun kehityksen avulla. Versionhallinta toteutettiin GitHubin avulla.

Opinnäytetyössä esitellään myös jatkuvan integroinnin/julkaisun (CI/CD) putkiston luomista Unity-pelille käyttäen GitHubin tarjoamaa GitHub Actions -työkalua. Esimerkkinä käytettävä CI/CD putkisto on luotu käyttäen peliesimerkkiä. Esimerkki esittelee tämän lähestymistavan mahdolliset hyödyt käyttöönoton virheriskin vähentämisessä ja kehitysprosessin tehokkuuden parantamisessa.

Kaiken kaikkiaan tämä opinnäytetyö tarjoaa yleiskatsauksen automaattisesta testauksesta Unity-pelimoottorin kanssa ja tarjoaa käytännön ohjeita kehittäjille, jotka haluavat parantaa pelinkehitysprosessien laatua ja tehokkuutta.

---

Avainsanat: Unity, automaattinen testaus, pelikehitys

**CONTENTS**


1 INTRODUCTION ........................................................................ 7
2 BACKGROUNDS AND GOALS ........................................................ 8
3 UNITY ...................................................................................... 10
3.1 Overview ............................................................................ 10
3.2 Unity Editor ........................................................................ 10
3.3 Tools ................................................................................... 11
4 TESTING .................................................................................. 13
4.1 Test types ........................................................................... 13
4.1.1 Business-facing tests that support the development process ........................................................................ 14
4.1.2 Business-facing tests that critique the project .................... 14
4.1.3 Technology-facing tests that support the development process ........................................................................ 14
4.1.4 Technology-facing tests that critique the project ................ 15
4.2 Test-driven development methods ........................................ 15
5 TEST DRIVEN DEVELOPMENT (TDD) ........................................ 16
5.1 Overview ............................................................................ 16
5.2 Process steps ...................................................................... 16
5.3 Advantages ......................................................................... 16
6 TESTING IN UNITY .................................................................. 17
6.1 Unity Test Framework .......................................................... 17
6.2 Edit Mode tests .................................................................... 17
6.3 Play Mode tests ................................................................... 18
6.4 How to use .......................................................................... 18
7 TDD AND TESTING IN EXAMPLE GAME PROJECT ........................ 20
7.1 Overview ............................................................................ 20
7.2 Game mechanics ................................................................. 20
7.3 Splitting project to features with requirements ........................ 21
7.3.1 Hero character ................................................................ 21
7.3.2 Enemy character ............................................................. 22
7.3.3 Battle system .................................................................. 22
7.3.4 User interface .................................................................. 22
7.4 Script development using TDD ............................................... 23
7.4.1 Character ........................................................................ 24
7.4.2 Hero ............................................................................... 26
7.4.3 Enemy ............................................................................ 27

7.4.4 BattleSystem ........................................................................ 28
8 AUTOMATED TESTING WITH GITHUB ACTIONS ........................... 30
8.1 GitHub Actions ........................................................................ 30
8.2 Tools needed ........................................................................... 30
8.3 Setting up ................................................................................ 30
8.4 Create workflow in GitHub repository ........................................ 31
8.4.1 Testing job ........................................................................... 31
8.4.2 Building job .......................................................................... 33
8.4.3 Deployment job ..................................................................... 35
8.4.4 View workflow status in GitHub ............................................ 35
9 DISCUSSION ............................................................................... 39
REFERENCES .................................................................................. 40

## ABBREVIATIONS AND TERMS

| | |
|---|---|
| Unity | game engine |
| TDD | test-driven development |
| Git | version control tool |
| GitHub | version control service |
| Cross-platform | available for multiple platforms like IOS and android |
| UTF | Unity Test Framework |
| CI/CD | Continuous integration/continuous delivery |
| LFS | Large File Storage |
| GitLab | Version control service |
| CircleCI | Version control service |

# 1 INTRODUCTION

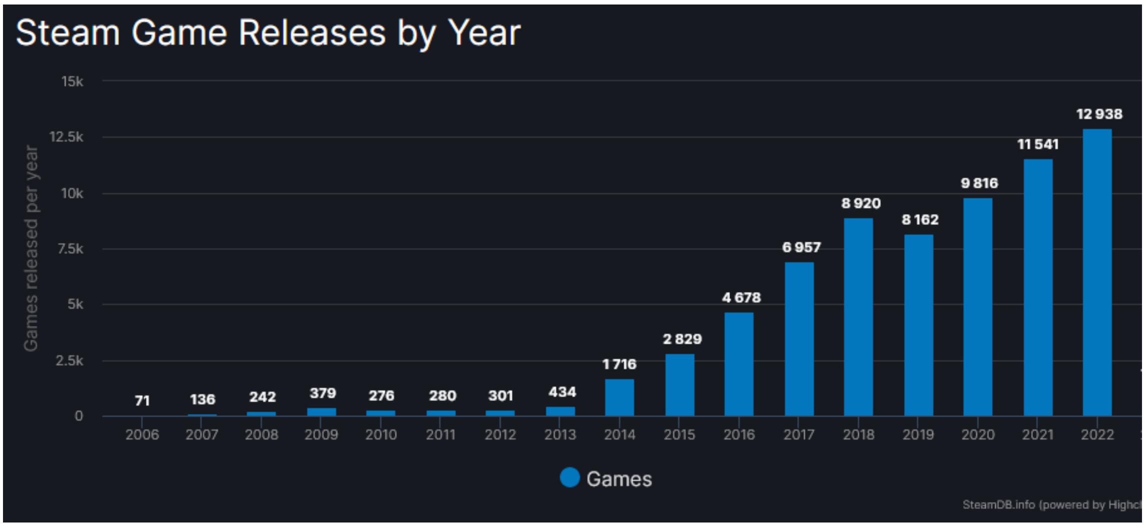
Like in all software development also game development automated testing is coming a more essential way to save man hours and add reliability to software development. As projects and development teams grow larger, automated testing ensures a solid foundation to games development. It also plays an essential role for setting up pipelines for continuous integration/deployment as it can verify code functionality before deployment. (Pêcheux: Unit testing automation with Unity/C# and Codemagic.)

The motivation for this thesis is to research and learn to optimize game development workflow using automated testing in Unity. Also dive into topics such as test-driven development (TDD), testing in general, continuous integration and continuous delivery (CI/CD) with GitHub version control. By providing practical examples, this thesis aims to demonstrate how TDD and automated workflows can be beneficial to the development workflow and how to start applying these concepts.

The thesis is structured as follows: The first chapter provides a brief introduction of the thesis and topics it covers. The second chapter covers the topics of game development in general, testing in game development, game engines, version control, key questions for this thesis. The third chapter introduces Unity game engine. The fourth chapter dives deeper into different kinds of testing options. The fifth chapter explains the concept of Test-Driven Development (TDD) and its benefits in the development process. The sixth chapter explains test tools in Unity and how to use them. The seventh chapter demonstrates the TDD concept in a simple example game project. The eight chapter focuses on the implementation of automated workflows with GitHub Actions and provides practical examples. Finally, the ninth chapter discusses the benefits and possible drawbacks of including automated testing to game development workflow and examines the examples used in this thesis.

## 2 BACKGROUNDS AND GOALS

Game industry as a business market is growing at fast pace (Parreno: Gaming Is Five Times Bigger Than Movies Now), which lures many to game development. Game development is coming more and more available to beginners with free and easy to use game engines like Unity. Also learning material is easy to find from the internet.
The number of new releases made by indie developers are growing in popularity but also competition is getting fiercer as Steam game data indicates. Over 12 thousand games were released on Steam game store in 2022.




GRAPH 1 Steam game releases by year (https://steamdb.info/stats/releases/)

Differentiation from other games is key to succeed, so projects might need to be larger and more complicated which requires a process to get project through. Game development shares a lot with software development when it comes to development process and testing is one part of that process. Although games can have elements that needs more thought process what approach should be taken when testing, for example, fun-factor, AI-behaviour, or level-testing. (Hoberg: Differences between Software Testing and Game Testing)

Game development process differs from style of the game, what game engine is used and what tools are available. Unity is popular game engine among beginners, but there are also Unreal Engine, CryEngine, Godot and more different game engines available, each with their own benefits.

All serious projects should use version control in its development workflow, such as Git. Version control brings safety to development and helps with collaboration when multiple people are working on the same project. There are many services for version control with Git as GitHub, GitLab, Bitbucket, Azure DevOps etc. GitHub was used in this project.

Test-driven development as a technique is a great tool to write simplified, clear, and uncluttered code. In a practical matter TDD is a technique that can be seen as a skill that can be learned. (Siddiqui: Learning Test-Driven Development)

The purpose of this project is to create a testing workflow in Unity to unveil questions related to testing in game development:

- How to implement automated testing to game development workflow?
- How easy it is to integrate Unity tests to GitHub Actions CI/CD pipeline?
- How much extra work writing and updating tests generates?
- How to write good tests?

# 3 UNITY

## 3.1 Overview

Unity is a cross-platform game engine used to develop games in 2D, 3D, VR and more. It was first released in 2005 and has since become one of the most popular game engines in the industry, used by a wide range of developers, from hobbyists to large game studios. Many popular titles have been developed using Unity such as Pokémon Go, Cuphead and Call of Duty: Mobile.

PICTURE 1: Unity Logo (https://brandguide.brandfolder.com/unity/unitylogo)

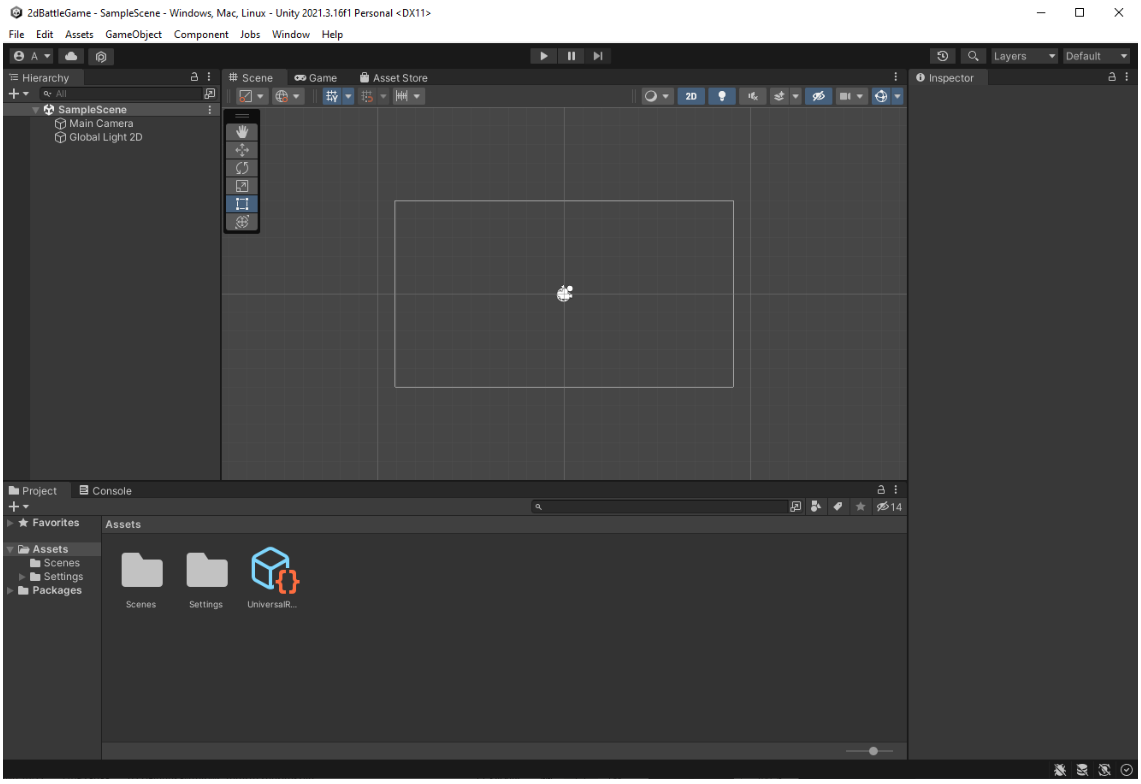
Unity supports multiple platforms, including Windows, MacOS, iOS, Android, and many more, allowing developers to easily deploy their games and experiences to a wide range of devices. Unity supports C# coding language.

The Unity Editor provides a visual development environment, allowing developers to create game assets, scenes, animations, and more using a drag-and-drop interface. The engine also includes a useful physics engine, which allows to create realistic simulation of rigid bodies and collisions.

Unity also has a large community of developers and an asset store, where developers can sell or purchase and download pre-made assets and tools to help them in their development. This includes a large and diverse range of assets, from 3D models and animations to plugins and scripts.

## 3.2 Unity Editor

The Unity Editor is a visual development environment that provides a range of tools for creating and editing game assets, scenes, animations and more. It includes a Scene View, which provides an interactable visual representation of the game world, and a Game View, which shows how the game will look when played. Also, the game is testable in the editor, even if the game is meant to be played on mobile devices.




PICTURE 2: Unity Editor user interface with default layout

Unity recommends Visual Studio for code editing tool, but it support other editors like Visual Studio Code or JetBrains Rider. (Unity: Unity Documentation)

## 3.3 Tools

Unity provides a lot of useful, free, and ready to use tools for its developers. Unity has built-in tools and systems for animations, lighting, physics, inputs, testing, localisation and more. This thesis focuses on testing related tools, such as Unity Test Framework.

Unity also has a lot of external tools developed by the community via Unity Asset Store. External tools might cost some money, but usually prices are quite affordable even for smaller developers. (Unity: Unity Asset Store)

# 4 TESTING

Software testing is essential to ensure software is functioning correctly and meets its requirements. Testing can be approached and implemented in multiple ways, whether it is automatic or manually done by developers or externally. In an ideal project developers, tester and users collaborate to write automated test from the start of project. Effective testing reduces the risk of bugs and delays in the project. Automated testing combined with CI system provides a way to caught bugs early when the cost of fixing them is lower. (David Farley and Jez Humble, 2010)

## 4.1 Test types

There are many kinds of tests when it comes to software testing. Brian Marwick's figure is widely used to model various types of tests. (David Farley and Jez Humble, 2010)

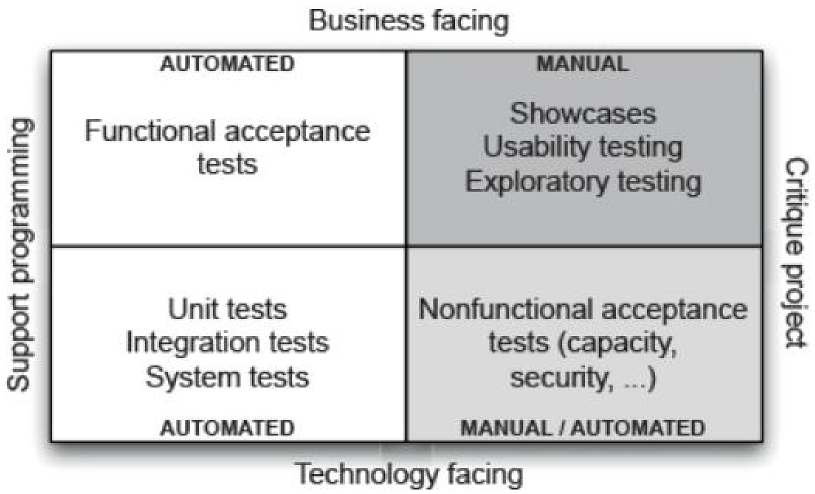



FIGURE 1: Brian Marwick's model of different kinds of tests (David Farley and Jez Humble, 2010)

Tests are divided in four compartments:

- Technology-Facing Tests That Critique the Project
- Business-Facing Tests That Support the Development Process
- Business-Facing Tests That Critique the Project

- Technology-Facing Tests That Support the Development Process

### 4.1.1 Business-facing tests that support the development process

Tests in this quadrant functions like acceptance criteria for features to develop. These are formed with a customer and developers, and they can be used to define how implementation should work to be acceptable for the customer. Commonly these are known as functional or acceptance tests.

### 4.1.2 Business-facing tests that critique the project

These tests are used to verify applications expected value for the users, and for checking that specifications are correct and up to date. These are manually done because these are very much related to users' experience interacting with the software.

### 4.1.3 Technology-facing tests that support the development process

Tests in this compartment are created and maintained by developers themselves. Unit tests, integration tests and system tests are found in this category. All of these should be automated tests.

Unit tests test very specific pieces of code in isolation and therefore helps to point out if something is not working properly. These are very fast to run and quickly show if changes have broken any existing functionality, but they might miss some problems, for example, problems with lifecycles of objects or data.

Integration tests are for testing components in a bit wider picture than unit tests. These are bit slower to run and they require more setup, but they can catch problems which are not detected in unit tests.

### 4.1.4 Technology-facing tests that critique the project

Tests in this quadrant are non-functional tests that tests qualities of the whole system, such as security, capacity, availability and so forth. These tests can be both automatic and manual, but automatic is preferrable if possible.

## 4.2 Test-driven development methods

There are multiple ways to approach testing-driven development. (Unadkat: TDD vs BDD vs ATDD: Key Differences.) Commonly used are:

- TDD: In test-driven development tests are defined and written before coding any implementation. This encourages developers to think about the feature or problem in a more structured way and may reveal problems before any time has been spent in the development.
- BDD: Behaviour-driven development emphasizes defining the expected behaviour of the program before development. Test cases are described in a natural language in which can be fully understood without any developer background.
- ATDD: Acceptance test-driven development focuses defining acceptance criteria for each requirement, user story or feature before development. Automated tests will be created from these criteria that will verify that the software meets the requirements and behaves correctly.

# 5 TEST DRIVEN DEVELOPMENT (TDD)

## 5.1 Overview

Test-driven development, often referred as TDD, is a software development process where tests for new code are written before the code itself is written. It is a technique for designing and structuring simplistic code that increases confidence when project gets bigger. (Siddiqui: Learning Test-Driven Development) This thesis will focus on using TDD as a development technique in game development.

## 5.2 Process steps

1. Write a failing test.
2. Run test to verify it fails, which confirms that the expected functionality is not yet implemented.
3. Write the minimum amount of code that is needed to make the test pass.
4. Run test again to verify it passes the test.
5. If any tests fail, refactor the code to correct the issue and return to step 3.
6. Process is done after all tests are successful.

## 5.3 Advantages

This process produces more testable code as the developer must think about the problem in a more structured way, before thinking the implementation itself. Code will be easier to maintain, and tests helps catching bugs or regressions as the codebase grows larger. (Siddiqui: Learning Test-Driven Development)

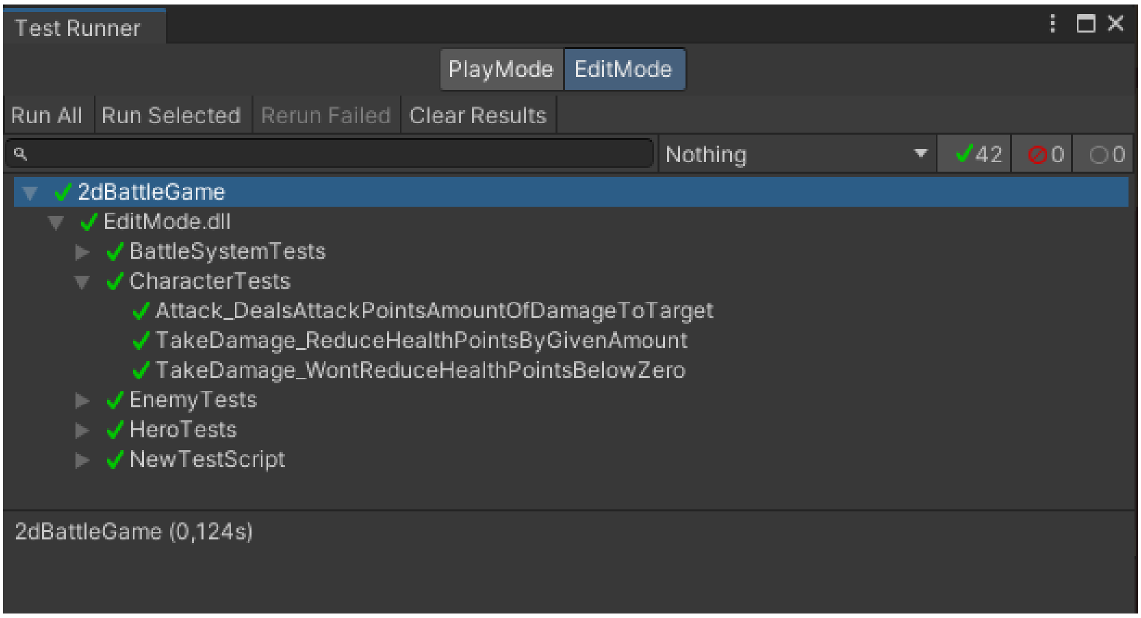
This technique encourages writing more simplified, obvious, and uncluttered code, which is a key of being successful. (Siddiqui: Learning Test-Driven Development)

# 6 TESTING IN UNITY

## 6.1 Unity Test Framework

Unity Test Framework (UTF) provides Unity users to test their code in both Edit Mode and Play Mode, also on different target platforms such as Android, iOS etc. (Unity: About Unity Test Framework)

UTF utilizes the open-source NUnit unit testing library that is specifically designed for .Net languages. The tests are written in C#, and they can be conveniently executed using the Unity Test Runner that is integrated into the Unity editor.




PICTURE 3: Unity Test Runner with multiple successful test suites

## 6.2 Edit Mode tests

Edit Mode tests are only run in the Unity Editor and have access to code in the Editor. (Unity: About Unity Test Framework)

Edit mode test are run in the editor so only non-play related functionality is testable. Tests in edit mode are great for testing features that do not use physics engine or animations. Unit tests are more suitable as Edit Mode tests.

### 6.3 Play Mode tests

Play mode tests runs in a standalone player, which simulates the game (Unity: About Unity Test Framework). These tests are better for testing features that relies on physics engine or time-based delays and coroutines. For example, testing a feature that deletes a game object one second later after collision. Tests in play mode are heavier to run and a bit heavier to maintain

### 6.4 How to use

Unity has documentation on their website how to setup Unity Test Framework and how it works (Unity: Getting started with Unity Test Framework). Here is a brief explanation of the setup process:

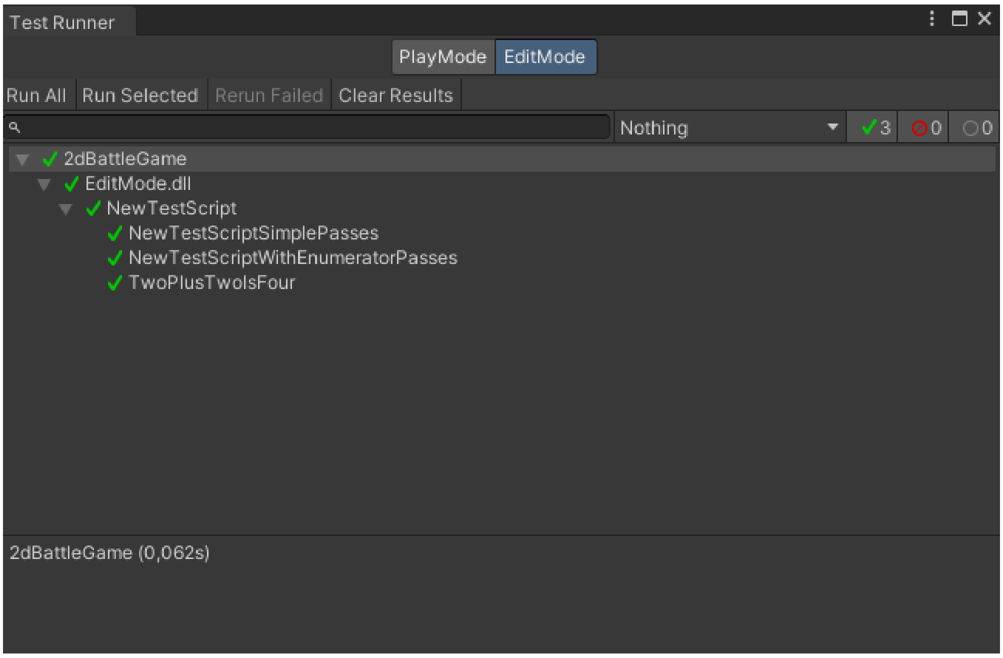
1. Create Test Assembly folder for PlayMode or EditMode tests.
2. Link project scripts Assembly Definition file to Assembly Definition References list in newly created Test Assembly
3. Create test scripts
4. Run tests from Unity Test Runner

Here is an illustrative example of a test that can be used to verify the Test Runners functionality after setup process. The test involves a basic calculation that adds two and two, and it includes an assertion to verify that the result is indeed four.

```
[Test]
public void TwoPlusTwoIsFour()
{
    // ARRANGE
    int firstNumber = 2;
    int secondNumber = 2;
    int result;

    // ACT
    result = firstNumber + secondNumber;

    // ASSERT
    Assert.AreEqual(4, result);

}
```

FIGURE 2: Calculation test in NewTestScript

The calculation test that has been created should be visible in the Test Runner, and it is expected to execute successfully when executed. Picture 4 shows that the calculation test is included in the Test Runner and it passed.




PICTURE 4: Unity Test Runner with successfully ran tests from NewTestScript

# 7 TDD AND TESTING IN EXAMPLE GAME PROJECT

## 7.1 Overview

Small example project to develop for this thesis. Project is a turn-based battle RPG in 2D. It has simple game mechanics, minimalistic animations, and a simple user interface. Graphical assets are free community-made assets from Unity Asset Store. Game is built for WebGL platform and is deployed to GitHub Pages where game is publicly available to play.

PICTURE 5: Example game scene in Unity

## 7.2 Game mechanics

Game will have simple turn-based battle mechanics.

Player hero and enemy will alternate turns. Player can make one action per turn. Enemy will always strike and deal damage to player. This will repeat until either player or enemy is defeated.

Enemies' health and attack damage increase after every battle. Enemies get stronger a bit faster than player, so they will eventually defeat the player. The goal is to defeat as many enemies as possible before player gets defeated. The player has two options for action which are attack or heal.

After every three adversaries defeated, player is rewarded with a new weapon and a new armour, which of player can only select and keep one. Weapons increase players damage and armour increases players health.

## 7.3 Splitting project to features with requirements

It is good to divide the project to smaller pieces or features. It can help with planning and figuring out what kind of functionality is needed.

### 7.3.1 Hero character

Hero should have variables for tracking health and attack points and the Hero needs a function to deal damage to enemy. Damage value is determined by hero's attack points. This can also handle playing hero's attack animation.

Hero could have a function to receive damage points and check that health points don't go below zero. This function also handles proper animations when taking damage or getting defeated.

Hero needs a function to calculate healthPoints and maxHealthPoints when armour is upgraded. Current health points should adjust correctly to be at same percentual value compared to maxHealth as it was before armour upgrade.

Also, a heal function is needed where 25% of maxHealthPoints will be added to healthPoints. Health points should not exceed maximum health points when healed.

### 7.3.2 Enemy character

Enemy is a bit simpler feature-wise than hero. It should have variables to track health points and attack points. A function is needed to deal attack points amount of damage to players hero character and play attack animation.

Enemy needs a function to receive damage, check that health points don’t go below zero, and play proper animations for taking damage and getting defeated. Also, a function to summon a new enemy is needed. New enemy should have stronger health and attack depending on how many enemies have been defeated.

### 7.3.3 Battle system

Battle system controls the flow of actions in this turn-based battle. Player and enemy alternate turns until either is defeated. Enemy attacks player character on its turn and player can choose an action.

Battle system checks if player or enemy is defeated when turn changes and reacts accordingly. Game will end when hero is defeated. Every time enemy is defeated, a new and stronger enemy will come to its place and battle starts again. Player always gets the first turn when new battle starts.

### 7.3.4 User interface

Simple user interface is needed to show important info and get input from the player. Health points for hero and enemy characters should be always visible for the player.

Player should have a way to select action from attack or heal but only when its players turn. Also, player needs to select from new weapon or new armour whenever three enemies have been defeated in a row.

Game over screen with a button to restart game. Game over screen should show how many enemies player defeated.

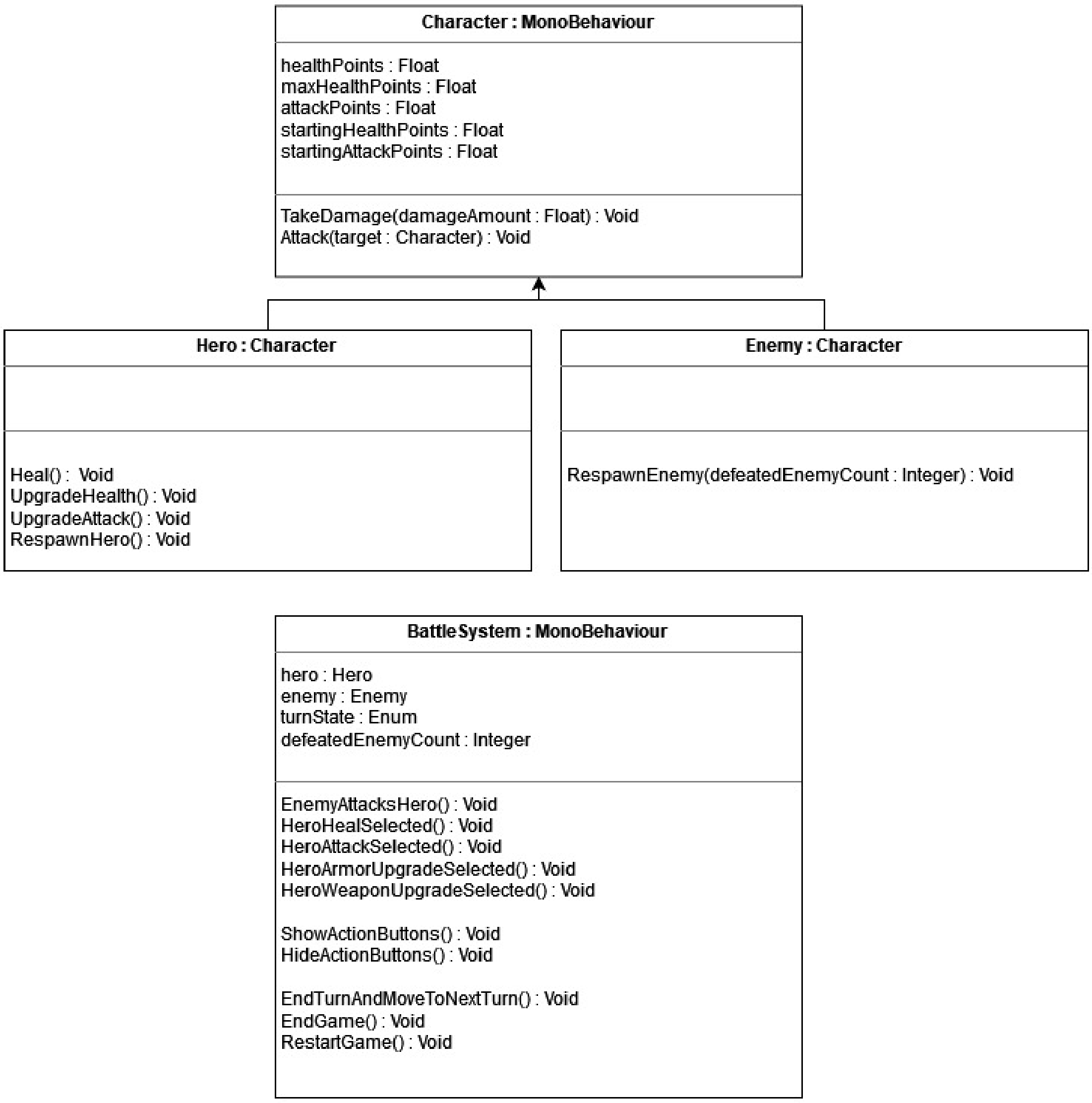



FIGURE 3: Description of custom classes and their methods

### 7.4 Script development using TDD

Once the desired features and functionalities have been defined, the development process can commence with the creation of tests, which is the initial step in test-driven development.

### 7.4.1 Character

For Character we will start by writing tests for all the methods defined earlier.

Function “TakeDamage” will need tests to check that correct number of points are deducted from characters health points, and that health points are never reduced below zero.

```
[Test]
public void TakeDamage_ReduceHealthPointsByGivenAmount()
{
    // ARRANGE
    GameObject gameObject = new GameObject("Character", typeof(Character));
    Character character = gameObject.GetComponent<Character>();
    character.healthPoints = 50f;

    // ACT
    character.TakeDamage(20f);

    // ASSERT
    Assert.AreEqual(30f, character.healthPoints);
}

[Test]
public void TakeDamage_WontReduceHealthPointsBelowZero()
{
    // ARRANGE
    GameObject gameObject = new GameObject("Character", typeof(Character));
    Character character = gameObject.GetComponent<Character>();
    character.healthPoints = 50f;

    // ACT
    character.TakeDamage(100f);

    // ASSERT
    Assert.AreEqual(0f, character.healthPoints);
}
```

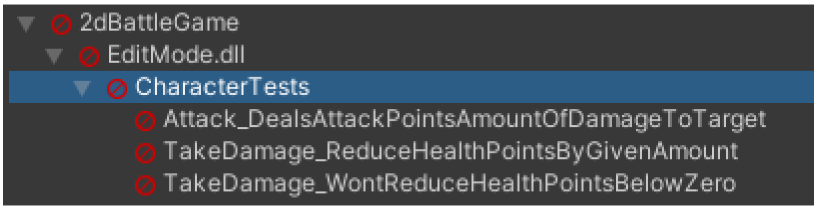
FIGURE 4: Tests for function “TakeDamage”

Function “Attack” is simple and can be covered with one test that ensures correct deduction of targets health points.

```
[Test]
public void Attack_DealsAttackPointsAmountOfDamageToTarget()
{
    // ARRANGE
    GameObject gameObject = new GameObject("Character", typeof(Character));
    GameObject gameObject2 = new GameObject("Character", typeof(Character));
    Character character = gameObject.GetComponent<Character>();
    Character targetCharacter = gameObject2.GetComponent<Character>();
    targetCharacter.healthPoints = 50f;
    character.attackPoints = 10f;

    // ACT
    character.Attack(targetCharacter);

    // ASSERT
    Assert.AreEqual(40f, targetCharacter.healthPoints);
}
```

FIGURE 5: Test for function “Attack”

The next step is to fix compilation errors that occurs if variables or methods are yet to be implemented. Create Character class with defined variables and empty placeholder methods. Unity Test Runner should allow to run all tests after compilation errors are fixed by creating the “Character” class with variables and methods used in the tests.



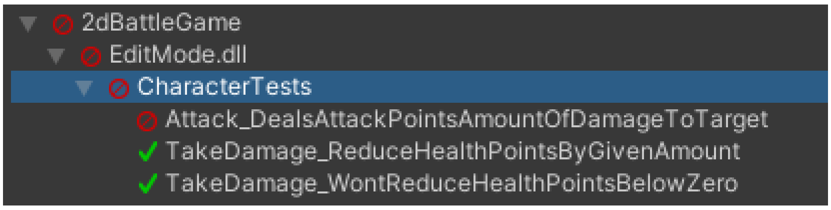

PICTURE 6: Failing test run after creating CharacterTests

The next step includes developing the solution to failed tests with minimal amount of code. In this case the function deducts characters health points for the amount received as a parameter. After deduction the script checks and sets health points to zero if health points have reduced to negative value.

```
public void TakeDamage(float damageAmount)
{
    // Deduct damageAmount from healthPoints
    healthPoints -= damageAmount;

    // Set health to zero if it reduced below zero
    if (healthPoints < 0)
    {
        healthPoints = 0;
    }
}
```

FIGURE 6: TakeDamage function in Character script

The next step is to check if tests are passing now. Tests for TakeDamage function are passing as test statuses are successful as shown in picture 7.



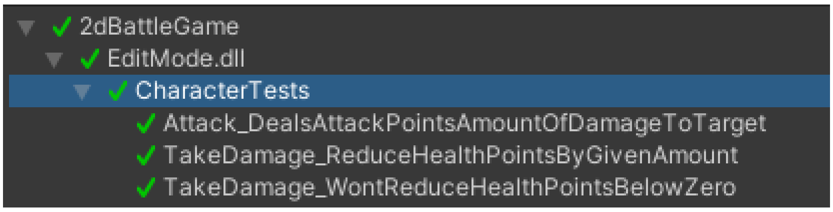

PICTURE 7: Unity Test Runner status with successful “TakeDamage” tests

Test for attack function is still failing and missing an implementation. Attack function receives a target character as a parameter and invokes target characters “TakeDamage” function with value from attack points.

```
public void Attack(Character target)
{
    // Deal attackPoints amount of damage to target
    target.TakeDamage(attackPoints);
}
```

FIGURE 7: Attack function in Character script

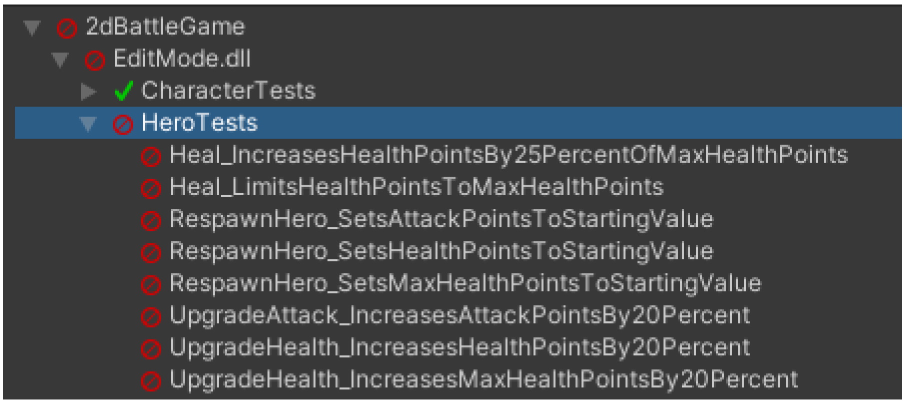
The final step is to run test and verify that all tests are successful. Character functionality can be considered as done after successful test run.




PICTURE 8: Unity Test Runner status with successful character tests

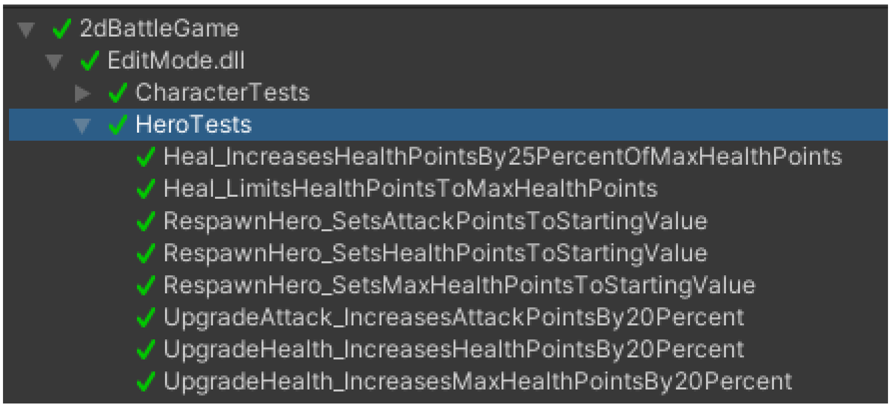
### 7.4.2 Hero

For Hero we will start by writing tests for the methods defined in the planning phase. That includes functions for healing the hero, upgrading health points, upgrading attack points, and respawning the hero.




PICTURE 9: Failed HeroTests

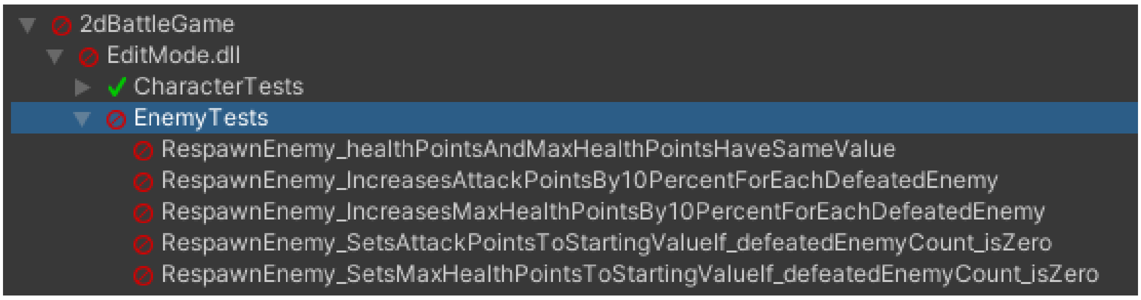
Implementation is the next step to make these tests pass. After all tests are successful, hero can be considered as done and we can progress to next piece of the game.




PICTURE 10: Successful run of HeroTests

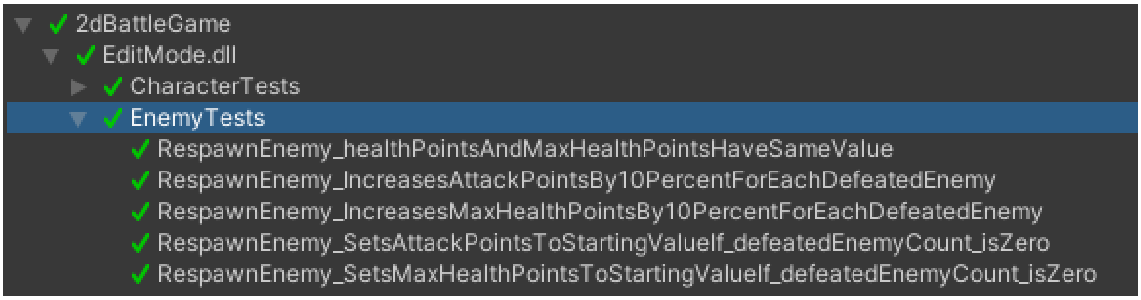
### 7.4.3 Enemy

Enemy is a simpler case, as it involves the testing of a single function. This function handles the enemy's respawn mechanics, ensuring that attack and health attributes scale properly as the game progresses.




PICTURE 11: Failed tests of EnemyTests

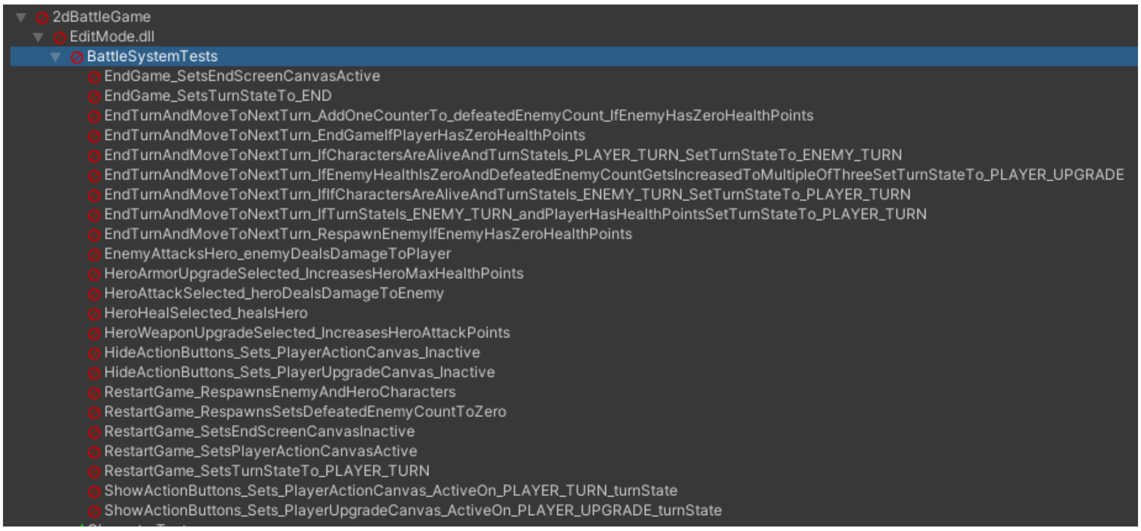
Next step of the process is to develop needed functionality to pass these tests created for the enemy character. Implementation is created in similar fashion as with character, minimalistic solutions to make tests pass.




PICTURE 12: Successful run of EnemyTests

### 7.4.4 BattleSystem

Last component to implement is the battle system which handles battles flow in the game. Battle system is responsible for showing correct possible actions for the player, enemy actions and respawn, handle turn transitions between enemy and hero characters, checking which character has been defeated and either respawn a new enemy or end game with an end screen, and more. Process starts by writing tests for battle system.




PICTURE 13: Failed run of battle systems tests

Functionality is then being added to pass these tests created in the previous step. Battle system is considered as done when all test cases are passing as shown in picture 14.


PICTURE 14: Successful run of battle systems tests

# 8 AUTOMATED TESTING WITH GITHUB ACTIONS

## 8.1 GitHub Actions

GitHub Actions is a platform for automated software workflows within a GitHub repository. GitHub Actions workflows can be triggered by various events, such as when a commit is pushed or a new pull request is created.

GitHub Actions uses GitHub hosted runners by default, but users can also setup and use self-hosted runners for running workflows. There is a limitation how much users can use GitHub-provided runners with free and private repositories, but most of limitations are removed with repositories that are set as public repository. (Github: Understanding GitHub Actions)

## 8.2 Tools needed

GameCI provides open-source tools to create automated workflows with GitHub, GitLab or CircleCI. This thesis focuses on setting up workflows for GitHub using GitHub Actions. These actions can be found from GitHub Marketplace from Actions section. Also a GitHub repository is needed.

## 8.3 Setting up

Setting up automated testing workflow with Github Actions is well documented at GameCI website. (GameCI: Getting started for GitHub) Briefly steps to setup are:

1. Configure a license for Unity and add it to repository secrets
2. Add workflow configuration file to your projects GitHub repository

Workflow is easily configurable to fit specific needs. For example, it is possible to have different workflows for different branches where a specific branch will test, build and deploy, but other branches only run tests.

### 8.4 Create workflow in GitHub repository

In this thesis project we will setup a workflow that will run when a commit is pushed to repository. Commits pushed to any branch will trigger testing workflow but commits to main branch will trigger additional jobs that automatically build and deploy to GitHub Pages after running the tests.

Workflow for this project is configured in “main.yml” file.

First lines specify workflows name and what actions will trigger this workflow. This workflow will run every time a new commit is pushed to repository.

.github > workflows > ! main.yml

```
name: Workflow Actions

on: push

```

FIGURE 8: First lines of main.yml

In “jobs:” will be all jobs the workflow contains. All jobs can have different run conditions and functionalities. In this project this workflow contains three jobs: “testRunner” for running tests, “buildWebGL” for building the game, and “deployPages” for deploying WebGL build to GitHub Pages.

#### 8.4.1 Testing job

Job for running Unity tests is named “testRunner” and it runs on the latest version of Ubuntu. It automates a process of testing a Unity game. The job consists of the following steps:

1. Check out the repository code using the GitHub-provided checkout action.
2. Create a list of large file storage (LFS) asset files used by the repository.
3. Restore the LFS cache using the GitHub-provided cache action to speed up the process if the LFS cache is already stored.

4. Pull the LFS assets using the Git LFS command, using the "add" and "reset" commands to ensure that the files are correctly added.
5. Restore the library cache, if it has been stored, using the GitHub-provided cache action. The cache has been created with a key that is generated from the project files. Using cache can speed up the process significantly.
6. Use the GameCI-provided action to run the Unity tests. This step requires a Unity license, which is passed as an environmental variable from the repository secrets. The GitHub token is also specified as a variable to authenticate the GitHub calls made by the action.
7. Upload the test results using the GitHub-provided upload-artifact action. The "with" field specifies the name of the artifact and the path to the directory containing the test results. The "outputs" field is used to get the path to the artifacts generated by the previous step.

```yaml
jobs:
  testRunner:
    name: Test all modes 📝
    runs-on: ubuntu-latest
    steps:
      # Checkout (without LFS)
      - name: Checkout code
        uses: actions/checkout@v2

      # Git LFS
      - name: Create LFS file list
        run: git lfs ls-files -l | cut -d' ' -f1 | sort > .lfs-assets-id

      - name: Restore LFS cache
        uses: actions/cache@v2
        id: lfs-cache
        with:
          path: .git/lfs
          key: ${{ runner.os }}-lfs-${{ hashFiles('.lfs-assets-id') }}

      - name: Git LFS Pull
        run: |
          git lfs pull
          git add .
          git reset --hard

      # Cache
      - name: Restore Library cache
        uses: actions/cache@v2
        with:
          path: Library
          key: Library-${{ hashFiles('Assets/**', 'Packages/**', 'ProjectSettings/**') }}
          restore-keys: |
            Library-
      # Test
      - name: Run tests
        uses: game-ci/unity-test-runner@v2
        id: testRunner
        env:
          UNITY_LICENSE: ${{ secrets.UNITY_LICENSE }}
        with:
          githubToken: ${{ secrets.GITHUB_TOKEN }}

      - uses: actions/upload-artifact@v2
        with:
          name: Test results (all modes)
          path: ${{ steps.testRunner.outputs.artifactsPath }}
```

FIGURE 9: Workflow job for running tests

### 8.4.2 Building job

The next job includes automated building process of building a WebGL version of the Unity game. The job for creating a WebGL build for the Unity game runs only if workflow is triggered on “main” branch of the repository. The job also needs a successful completion of the “testRunner” job that contains all the testing.

This job has same beginning steps as the testing job by checking repository code, downloading needed assets and libraries from cache, and then building the game using the unity-builder action from GameCI. Final step uploads the built files as an artifact that can be accessed later in the workflow.

```yaml
  buildWebGL:
    if: github.ref == 'refs/heads/main'
    needs: testRunner
    name: Build for WebGL 💻
    runs-on: ubuntu-latest
    steps:
      # Checkout (without LFS)
      - name: Checkout repository
        uses: actions/checkout@v2

      # Git LFS
      - name: Create LFS file list
        run: git lfs ls-files -l | cut -d' ' -f1 | sort > .lfs-assets-id

      - name: Restore LFS cache
        uses: actions/cache@v2
        id: lfs-cache
        with:
          path: .git/lfs
          key: ${{ runner.os }}-lfs-${{ hashFiles('.lfs-assets-id') }}

      - name: Git LFS Pull
        run: |
          git lfs pull
          git add .
          git reset --hard

      # Cache
      - uses: actions/cache@v2
        with:
          path: Library
          key: Library-${{ hashFiles('Assets/**', 'Packages/**', 'ProjectSettings/**') }}
          restore-keys: |
            Library-

      # Build
      - name: Build project
        uses: game-ci/unity-builder@v2
        env:
          UNITY_LICENSE: ${{ secrets.UNITY_LICENSE }}
          UNITY_EMAIL: ${{ secrets.UNITY_EMAIL }}
          UNITY_PASSWORD: ${{ secrets.UNITY_PASSWORD }}
        with:
          targetPlatform: WebGL

      # Output
      - uses: actions/upload-artifact@v2
        with:
          name: build-WebGL
          path: build/WebGL
```

FIGURE 10: WebGL build job

### 8.4.3 Deployment job

Deployment job handles deployment if workflow is triggered on “main” branch of the repository. The job also requires a successful completion of the WebGL build job.

Deployment is automatically done with “deployPages” job by checking out the code, downloading an artifact that contains the recently built WebGL version of the Unity game, and then deploying build to separate branch named “github-pages-build” using an action from JamesIves/github-pages-deploy-action (GitHub Actions: Deploy to GitHub Pages).

This new branch is linked to GitHub Pages website associated with the repository from GitHub settings as part of setting up the project.

```
deployPages:
  if: github.ref == 'refs/heads/main'
  needs: buildWebGL
  name: Deploy to GitHub Pages 🚀
  runs-on: ubuntu-latest
  steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - uses: actions/download-artifact@v2
      with:
        name: build-WebGL
        path: build

    - name: Display structure of root files
      run: ls -R
      working-directory: build/WebGL

    - name: Deploy 🚀
      uses: JamesIves/github-pages-deploy-action@v4
      with:
        branch: github-pages-build
        folder: build/WebGL
```

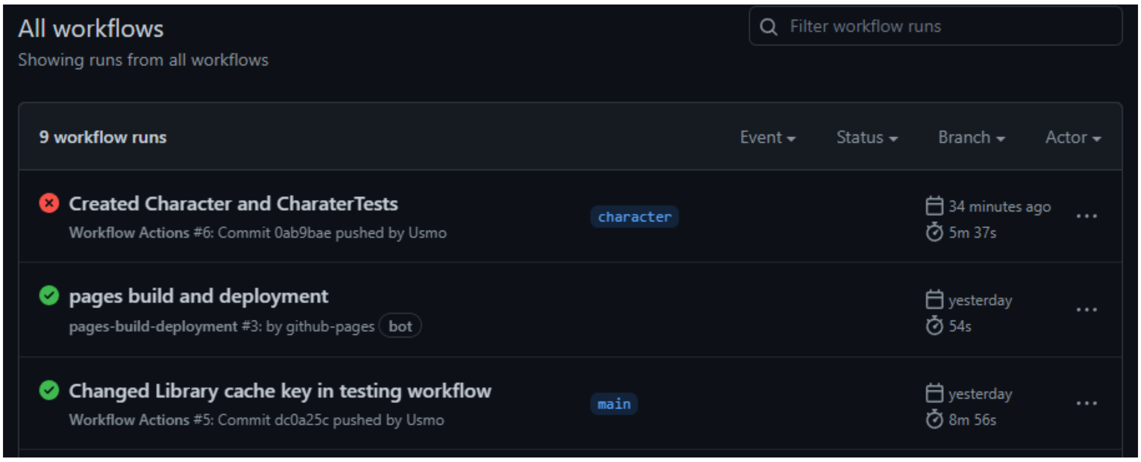
FIGURE 11: Deployment job

### 8.4.4 View workflow status in GitHub

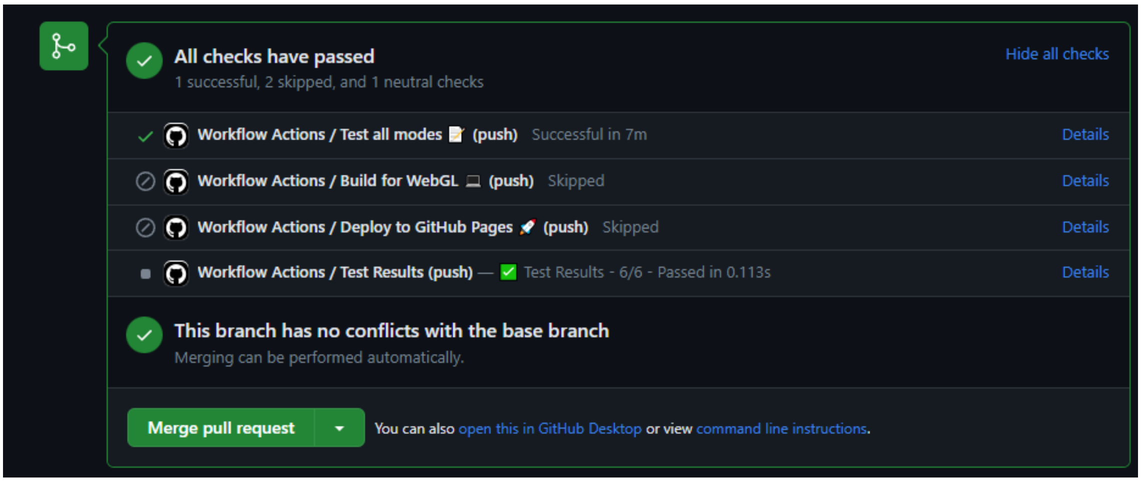
Seeing workflow statuses quickly on GitHub can help developers to easily identify failures and issues in the workflow. It also brings increased visibility and awareness between collaborators, which helps the team stay on the same page and work efficiently.

GitHub Action workflow status is shown in several places on GitHub, including:

- Next to commit or pull request. It shows workflows status icon whether it is still in progress, has succeeded, or has failed. Clicking the icon reveals more information about the workflow.
- The Actions tab shows all the workflows that have been run for a repository. You can see the status of each workflow, as well as its name, when it was run, and which branch or pull request it was run on. The Actions tab is shown in picture 16.
- In the notifications: If you have notifications turned on, you will receive a notification when a workflow succeeds or fails.
- In the pull request review: When a pull request has a failing check or workflow, GitHub displays a warning banner in the pull request, making it clear to reviewers that the pull request is not yet ready for merging. In the case of workflow and checks are successful, GitHub displays clearly that pull request is good to merge as shown in picture 17.

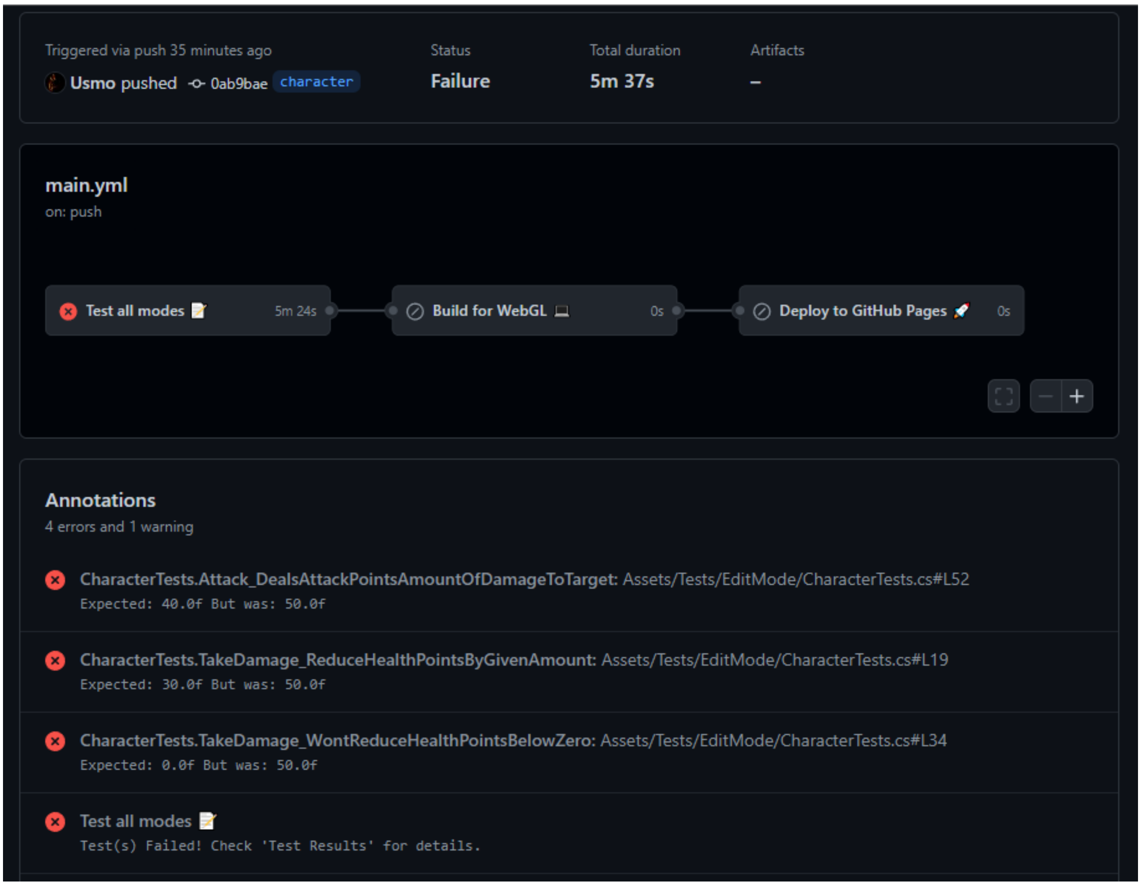



PICTURE 16: Repository workflows in GitHub Actions



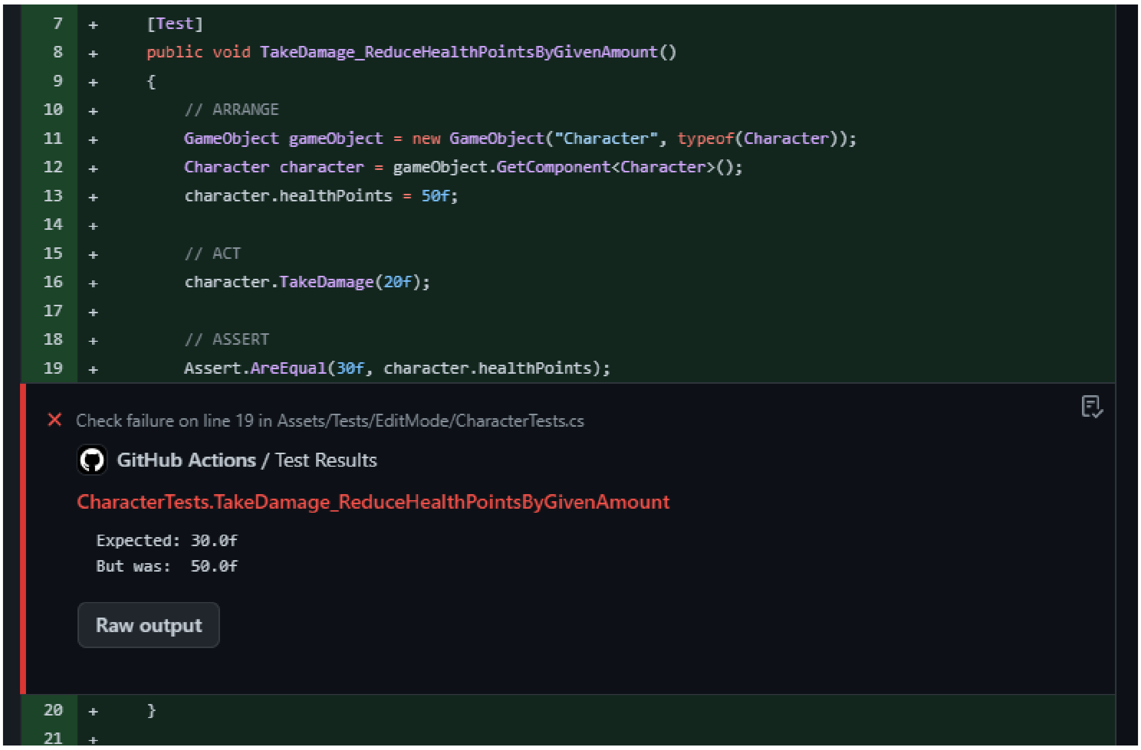

PICTURE 17: Successful workflow status in pull request

All workflows can be inspected more closely to obtain more detailed information about them. An overview is provided that displays all the jobs that the workflow includes, along with their current status. Annotations show possible errors or log messages from the workflow process.




PICTURE 18: Workflow details in GitHub

Additionally, developers can see specific test error banners within the code review page on GitHub, as shown in picture 19. These banners display a concise overview of the anticipated outcome for a particular test, along with the actual outcome that was obtained.




PICTURE 19: Failed test case shown at related line of code in GitHub code review

# 9 DISCUSSION

Based on this thesis examples and research, it can be concluded that the use of automated testing in game development with Unity can significantly benefit the development process. However, it is worth noting that preparing the testing environment and writing tests can take some time away from development, and creating comprehensive test cases during the prototyping phase may not be the best use of time if the project is ultimately scrapped or significantly altered based on the prototype. Therefore, projects in prototyping phase might benefit in having fast and minimalistic unit tests to support developing and debugging during prototyping phase.

In established projects, TDD can help create useful tests during development and be a significant aid when the project expands and becomes more complex. Creating workflows using GitHub Actions was a relatively quick and straightforward process, and implementing CI/CD workflows can be a useful addition to TDD development, particularly when the development team consists of multiple people.

The example game used in this thesis was very minimalistic with simple tests and did not showcase more complex testing scenarios that can arise in game development. However, the examples provided a simplistic guide and a good foundation for expanding on.

Overall, testing is a crucial aspect of game development and automating as much of it can improve the quality of the game and speed up development. Although preparing the testing environment, creating test cases, and writing tests can be time-consuming, it ultimately pays off in the long run by reducing errors and improving the quality of the development process. Additionally, implementing CI/CD workflows can help reduce errors and improve the efficiency of the development and deployment processes.

## REFERENCES

GitHub repository of the example project referenced in this thesis:
https://github.com/Usmo/2d-battlegame

Pêcheux, Mina. Unit testing automation with Unity/C# and Codemagic. Blog. Updated 7.2022. Read 29.3.2023.
https://blog.codemagic.io/unit-testing-automation-unity/

Parreno, Ryan. Gaming Is Five Times Bigger Than Movies Now. Online article. Updated 13.12.2023. Read 1.2023.
https://gameranx.com/updates/id/416500/article/gaming-is-five-times-bigger-than-movies-now/

Hoberg, Johan. Differences between Software Testing and Game Testing. Blog. Updated 21.7.2014. Read 1.2023.
https://www.gamedeveloper.com/programming/differences-between-software-testing-and-game-testing

Davis, Ash. Testing for Game Development. Blog. Updated 26.7.2016. Read 1.2023.
https://www.gamedeveloper.com/programming/testing-for-game-development

Siddiqui, Saleem. Learning Test-Driven Development. Online book. Published 10.2021 by O'Reilly Media. Read 1.2023.
https://learning.oreilly.com/library/view/learning-test-driven-development/9781098106461/preface01.html#idm44839567619456

Unity: Unity Documentation. Read 2.2023.
https://docs.unity.com/

Unity: Unity Asset Store. Read 2.2023.
https://assetstore.unity.com/

David Farley and Jez Humble. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Online book. Published 7.2010 by Addison-Wesley-Professional. Read 2.2023.
https://learning.oreilly.com/library/view/continuous-delivery-reliable/9780321670250/ch04.xhtml

Unadkat, Jash. TDD vs BDD vs ATDD: Key Differences. Online guide. Updated 9.3.2023. Read 3.2023.
https://www.browserstack.com/guide/tdd-vs-bdd-vs-atdd

Unity: About Unity Test Framework. Version 1.3.4. Read 2.2023
https://docs.unity3d.com/Packages/com.unity.test-framework@1.3/manual/index.html

Unity: Getting started with Unity Test Framework. Version 1.3.4. Read 2.2023 https://docs.unity3d.com/Packages/com.unity.test-framework@1.3/manual/getting-started.html

Github: Understanding GitHub Actions. Read 3.2023.
https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

GameCI: Getting started for GitHub. Version v2. Read 3.2023.
https://game.ci/docs/github/getting-started

LP Games: How to build your Unity game with Game CI and Github. Online video guide. Published 9.7.2021 on video-sharing platform Youtube. Viewed 3.2023.
https://www.youtube.com/watch?v=M2BZr02uai0

GitHub Actions: Checkout. Read 3.2023.
https://github.com/marketplace/actions/checkout

GitHub Actions: Cache. Read 3.2023.
https://github.com/marketplace/actions/cache

GitHub Actions: Upload a Build Artifact. Read 3.2023.
https://github.com/marketplace/actions/upload-a-build-artifact

GitHub Actions: Download a Build Artifact. Read 3.2023.
https://github.com/marketplace/actions/download-a-build-artifact

GitHub Actions: Deploy to GitHub Pages. Read 3.2023.
https://github.com/marketplace/actions/deploy-to-github-pages