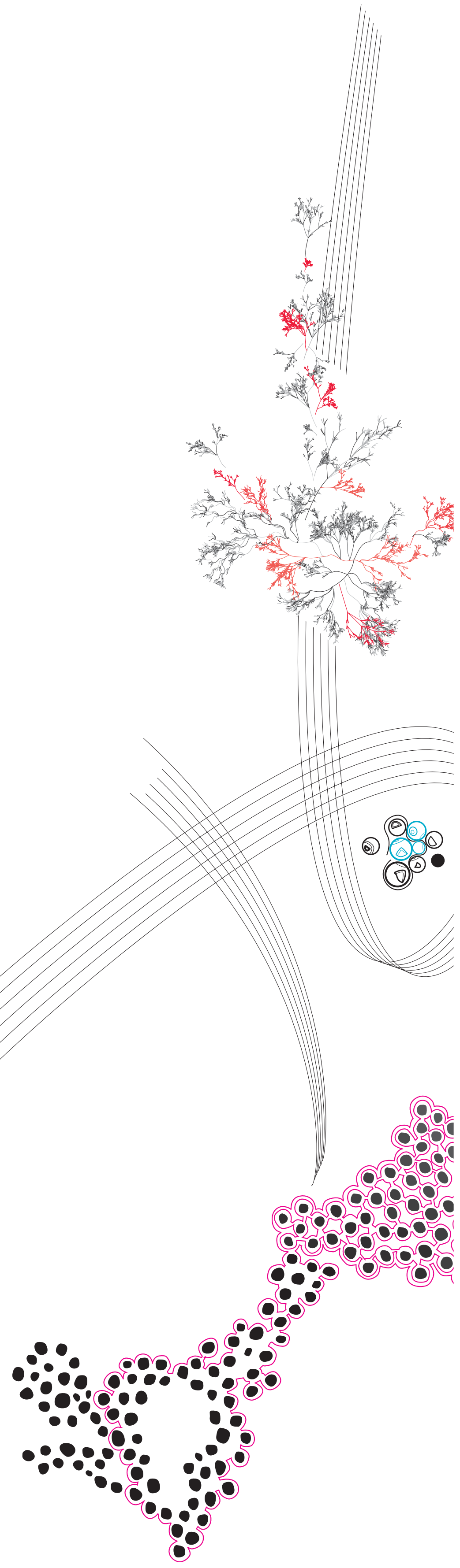
MSc Computer Science
Thesis



From Behaviours to Code: Exploring Behaviour-Driven Development in Unity 3D Game Creation

Michael Mulder

Supervisors:
dr. P. van den Bos
prof.dr. M.I.A. Stoelinga
dr.ir. R.W. van Delden

kap. J. Wouters MSc

June, 2024

Formal Methods & Tools Group
Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Glossary

asset

An asset in Unity is a single item or object. 28

BDD feature

A BDD feature contains multiple BDD scenarios that describe one behaviour or feature. Each BDD feature is placed in its own featurefile. 5

BDGD

Behaviour-Driven Game Development. , 2, 3, 10, 47, 57, 70, 71

Behaviour-Driven Game Development

The method developed in this thesis to apply BDD in the development process of a game. 3, 9

Game Behaviour Framework

The framework to categorize game behaviours developed in this thesis. , 2, 3, 8, 10, 32, 47, 70, 71

NPC

Non-Player Character. 7, 32

NuGet

NuGet is a package manager, primarily used for packaging and distributing software written using .NET and the .NET framework. Unity does not support the use of NuGet. 41

scene

In game development a scene often refers to a distinct, individual level or section of the game world. 28

UnitySpec

The tooling developed in this thesis to support BDD in Unity. , 2, 3, 9, 10, 47, 71

UTF

Unity Test Framework. 40, 43, 45

Abstract

Game Software Engineering (GSE) has emerged as a specialized field distinct from traditional software engineering, addressing unique challenges inherent in game development, but lagging behind in utilizing the new methods of software engineering. This thesis aims to introduce one of these new methods in the realm of game development. It will introduce a method for applying Behavior-Driven Development (BDD) principles to game development in Unity 3D, aiming to enhance the reliability and quality of game software.

BDD is an agile development approach that emphasizes user-centric scenarios and collaborative communication, bridging the gap between technical and non-technical stakeholders. This approach improves upon Test-Driven Development (TDD) by focusing on system behaviours rather than low-level code functionality, ensuring that developed features align with user requirements.

This thesis will present a framework to help express game behaviours as BDD scenarios, a method to apply BDD in game development, and tooling to support the process. The results of this thesis are validated through a case study with the Royal Netherlands Marechaussee, which requires robust and effective training simulations to prepare for evolving security threats.

Keywords: Behaviour-Driven Development (BDD) • Game Software Engineering (GSE) • Game Development • Unity 3D • Game Testing • Game Behaviour Framework • Behaviour-Driven Game Development • UnitySpec

This study has been reviewed by the Computer & Information Sciences Ethics Committee of the University of Twente. It has been approved under application nr. 230711.

Chapter 1

Introduction

1.1 Project Introduction

The gaming industry has emerged as a prominent sector, marked by a compounded annual growth rate of 13% and a projected revenue exceeding $366 billion in 2023 [19, 47]. Game Software Engineering (GSE) represents a growing research domain that has evolved distinctly from traditional software engineering paradigms [32]. While a lot of research has been done to improve traditional software development and methods, game software engineering lags behind [32]. However, the widespread occurrence of software bugs and crashes continues to cast a shadow over the user experience within this thriving industry [64]. As Bethke states "Too often game developers hold themselves apart from formal software development and production methods with the false rationalization that games are an art, not a science." [21] This leads their projects to be delivered behind schedule, have inefficient use of resources, and to published games having many defects [21].

The present thesis endeavours to introduce a method from traditional software engineering to game software engineering. The result of this will be a method that applies specifications and testing within the domain of game software engineering. This approach will be based on the principles of behaviour-driven development (BDD).

Behaviour-driven development is an agile development approach originally developed by Dan North that focuses on feature behaviours [66]. Since its conception, BDD has been broadly adopted and is becoming an established industry practice [27, 41]. One of the biggest toolsets supporting BDD boasts thousands of success stories [1].

North's motivation for devising BDD stemmed from observations regarding the limitations of test-driven development (TDD). TDD emphasizes writing tests for individual units of code before the code itself is written, ensuring that each piece of functionality is independently verifiable. However, TDD often struggles with ensuring that the developed features align with user requirements, as it tends to focus on low-level implementation details rather than user interactions and overall system behaviour. BDD addresses this gap by emphasizing scenarios written in a natural language style that describes the desired behaviour of the system. These scenarios, documented using the Given-When-Then format, bridge the communication gap between developers, testers, and non-technical stakeholders, ensuring that everyone has a shared understanding of the expected system behaviour [66].

Additionally, BDD represents an amalgamation of TDD and Domain-Driven Design (DDD) principles. DDD, aimed at addressing projects characterized by intricate business rules, necessitates collaborative efforts between domain experts and developers to establish a common language and domain model that can subsequently be translated into detailed and specific requirements [22].

The utilization of specifications within BDD empowers both programmers and non-programmers involved in the development process to engage in effective communication concerning system features. Furthermore, BDD's emphasis on system behaviours resonates with the natural approach of testing games, mirroring the manner in which humans assess games, as opposed to concentrating solely on individual methods, which often hold limited significance within the broader context of the game.

This project will be tested using a case study from the Royal Netherlands Marechaussee, which faces increasing challenges in ensuring public safety due to evolving threats. The Royal Netherlands Marechaussee has a growing need for reliable and effective training simulations and software solutions to prepare its personnel for various scenarios. By applying BDD principles this thesis aims to help develop robust and well-tested simulation software that meets the requirements of the Royal Netherlands Marechaussee, ultimately contributing to enhanced training capabilities and operational readiness.

1.2 Research Questions

The goal of this paper is to determine if and how well Behaviour-Driven Development can be applied in the realm of game development. While most of this research can be applied to any game engine, the practical application has been scoped to Unity 3D. To this end, we have formulated the following research question:

How can the principles of Behaviour-Driven Development be applied in game development in Unity 3D?

In support of this research question four secondary research questions have been formulated.

Q1. How can BDD scenarios be used to describe game behaviour?

This question centres on the first aspect of BDD, which involves creating scenarios to describe game behaviour. This question aims to determine how scenarios can be applied to describe different kinds of game behaviour. In answering this question we will create the Game Behaviour Framework, a framework for creating BDD scenarios from game behaviours.

Q2. What method could be employed to apply BDD in the development process of a game?

This question shifts the focus to integrating BDD within the game development process. It aims to formulate a method to apply BDD principles in the game development workflow, henceforth Behaviour-Driven Game Development (BDGD).

Q3. What tooling can be developed to support BDD in Unity 3D game development?

This question dives deeper into the practical application of BDD. The aim is to develop tooling to support behaviour-driven development in Unity. This tooling will be called UnitySpec.

Q4. What are the implications of applying our findings to a real-world case study?

The fourth question focuses on evaluating the results from the previous questions. This evaluation will take place by applying the research to a case study. We have subdivided this question into two subquestions. The first subquestion focuses on evaluating the results from *Q1*, the second on evaluating *Q2* and *Q3*.

Q4.1. How well can the proposed Game Behaviour Framework be applied to a real-world case study?

Q4.2. What are the implications of applying BDGD and UnitySpec to a real-world case study?

1.3 Contributions

This paper makes several significant contributions to the field of Game Software Engineering (GSE) by integrating Behavior-Driven Development (BDD) principles into the Unity 3D game development process. The key contributions are:

Game Behaviour Framework: We introduce a structured framework for categorizing game behaviours into three levels of complexity: basic behaviours, singular interactions, and complete game elements. This framework allows for clear and organized descriptions of game behaviours, facilitating both the development and testing processes. By breaking down behaviours into these distinct levels, developers can address both simple actions and complex interactions in a modular and systematic manner.

Behaviour-Driven Game Development Method: We design a comprehensive methodology that integrates BDD with game development practices. This methodology involves introducing scenarios to describe game behaviour. It incorporates phases of discovery, formulation, and automation. It emphasizes the collaboration of the "Three Amigos"—the product owner, tester, and developer—to ensure diverse perspectives and effective communication throughout the development process. Additionally, it includes prototyping and exploratory testing to balance artistic freedom from game development with structured development.

UnitySpec Tool Development: We introduce UnitySpec, an open-source tool tailored for Unity 3D, to support the application of BDD in game development. UnitySpec facilitates the creation, generation, and execution of Gherkin-based feature files within Unity projects. Key functionalities include scenario creation, test file generation, and step definition binding; all designed to integrate seamlessly with Unity's development environment. UnitySpec addresses the unique requirements of Unity, such as handling frame-based operations, and provides detailed test execution outputs.

Real-World Case Study Validation: We apply the Game Behaviour Framework, Behaviour-Driven Game Development method, and UnitySpec to a real-world case study involving the Royal Netherlands Marechaussee. This case study demonstrates the effectiveness and versatility of our approach in structuring and validating game requirements.

Overall, this paper advances the field of GSE by integrating formal software development practices into the creative process of game development. Demonstrating how BDD principles and tools can enhance the quality, reliability, and efficiency of game development in Unity 3D.

Chapter 2

Background

2.1 Introduction to BDD

Behavior-Driven Development (BDD) is an agile software development process that encourages collaboration among developers, QA, and non-technical or business participants in a software project. It emerged from Test-Driven Development (TDD) expanding its practices to involve more stakeholders and focusing on the behaviour of an application for end-users [66].

BDD was conceptualized by Dan North in 2003 to address the shortcomings and difficulties experienced in TDD, particularly around communication and understanding the requirements. BDD is rooted in the idea that software development should be guided by the desired behaviour of the application as perceived by stakeholders. This approach shifts the focus from testing and verification to specification and validation, ensuring that the development team builds the right software that meets business needs [63, 66].

In BDD, the behaviour of the software under development is given as a collection of example interactions with the system, expressed using natural language sentences. These sentences, called scenarios, are organized within a "Given-When-Then"-structure [63, 66]. The utilization of this structure facilitates the description of how users interact with the system, resulting in a specification that is non-technical and domain-specific in nature. A hallmark of BDD scenarios is their readability and comprehensibility by end-users and non-technical team members [23]. These scenarios thus create a bridge between the technical intricacies of software development and the conceptual understanding of system behaviours.

Listing 2.1 presents an example scenario that adheres to the "Given-When-Then"-structure to articulate expected behaviour. The behaviour expressed here is picking up a passport from a table. "And" can be used to combine sentences in each of the steps, in the example it is only utilized in the "Given"-step.

- **Given:** This section describes the initial state or context necessary for the scenario to unfold. It specifies the conditions that must be met for the scenario to proceed. In the provided example scenario, the conditions are that the player is located behind a table and that there is a passport on the table.
- **When:** This segment describes the user's action or input that triggers the scenario. In the illustrative scenario, the user's action is clicking on the passport to initiate the process of picking it up.
- **Then:** This section outlines the expected result or outcome of the scenario, contingent upon the specified conditions and actions. It defines what should happen when

```
Scenario: Pick up a passport from the table
    Given the player is behind a table
    And there is a passport on the table
    When the player interacts with the passport
    Then the passport should be shown
```

LISTING 2.1: Example Scenario

the scenario is executed. In the example, the expected outcome is that the passport is “picked up”, in this case shown in detail.

Multiple scenarios describing the same feature form a BDD feature. These are placed together in a featurefile.

To operationalize these scenarios, they are linked to the system through glue code, enabling their execution [25]. Glue code, in essence, serves as an intermediary layer of code that translates high-level behavioural scenarios into executable testing steps. This translation is typically facilitated by specialized tools designed explicitly for BDD purposes. Notably, a plethora of such tools exists, tailored to various platforms and programming languages [88].

Following the successful implementation of the feature, the scenario and its corresponding glue code are retained within the codebase. Subsequently, whenever alterations or enhancements are introduced into the code, the scenarios should be systematically executed to verify the integrity of previous features and to identify any potential bugs or unintended consequences [25, 66].

Behavior-Driven Development represents a significant evolution in agile software development, emphasizing collaboration, clear communication, and alignment with business goals. By structuring scenarios around the "Given-When-Then" framework, BDD facilitates the creation of understandable and actionable behavioural specifications that bridge the gap between conceptual understanding and technical implementation. Furthermore, the integration of glue code and scenario automation in BDD ensures the ongoing verification of system behaviours, offering a safeguard against inadvertent disruptions as software evolves [23, 25, 63, 66].

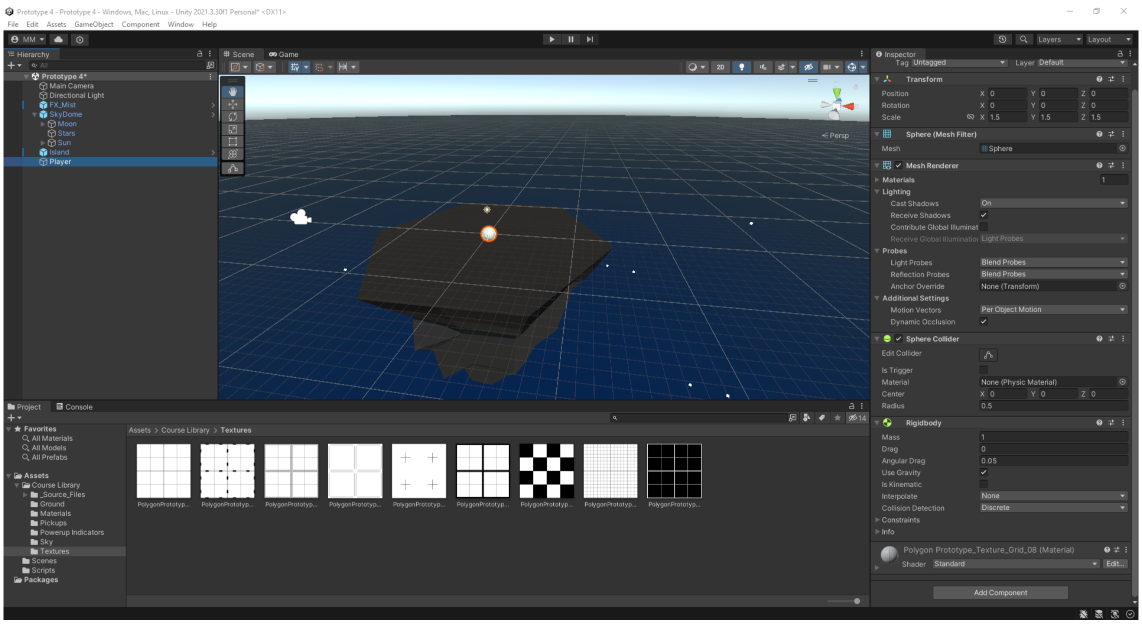
2.2 Introduction to Unity 3D

This introduction is a summary from online resources, anything said here can be found in [34, 50, 90].

Unity is a comprehensive and highly versatile platform for creating interactive, real-time 3D content. Launched by Unity Technologies in 2005, it has evolved into one of the most popular game development engines in the world, used not only for video games but also for a wide range of applications including simulations, virtual reality (VR), augmented reality (AR), film, architecture, and more. Unity provides developers with a powerful set of tools and workflows, enabling them to bring their creative visions to life across multiple platforms.

At its core, Unity is a game engine, offering a robust environment for creating and manipulating 3D graphics, physics, sound, and other multimedia elements essential for building immersive experiences. It features a highly intuitive and flexible editor that

[1]The project shown is from Unit 4 of the programmer pathway from Unity Learn [89]



Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1.5 Y 1.5 Z 1.5

Sphere (Mesh Filter)
Mesh Sphere

Mesh Renderer
Materials 1
Lighting
Cast Shadows On
Receive Shadows ✓
Contribute Global Illuminat
Receive Global Illuminatio Light Probes
Probes
Light Probes Blend Probes
Reflection Probes Blend Probes
Anchor Override None (Transform)
Additional Settings
Motion Vectors Per Object Motion
Dynamic Occlusion ✓

Sphere Collider
Edit Collider
Is Trigger
Material None (Physic Material)
Center X 0 Y 0 Z 0
Radius 0.5

Rigidbody
Mass 1
Drag 0
Angular Drag 0.05
Use Gravity ✓
Is Kinematic
Interpolate None
Collision Detection Discrete
Constraints
Info

Polygon Prototype_Texture_Grid_08 (Material)
Shader Standard

FIGURE 2.1: Standard Unity Editor[1]

allows developers to design, edit, and test their projects in real-time. This real-time aspect is particularly crucial, as it means creators can see the results of their work immediately, making the development process more dynamic and iterative.

One of the key strengths of Unity is its cross-platform capabilities. With Unity, developers can build applications for a wide range of platforms, including PC, consoles, mobile devices, and the web, with a single codebase. This capability significantly reduces the time and resources needed to deploy applications across different platforms. Unity's support for VR and AR has also made it a go-to tool for developers looking to create immersive experiences in these emerging fields.

The scripting in Unity is primarily done using C#, a modern, object-oriented programming language. This makes the engine accessible to a broad audience of developers, from beginners to seasoned professionals. Unity's documentation, tutorials, and community further support its user base, providing resources for learning and problem-solving. Additionally, the Unity Asset Store offers a marketplace where developers can find and share assets, plugins, and tools, enhancing the development process and allowing for rapid prototyping and production.

Unity's rendering engine is another highlight, capable of producing high-quality visuals. It supports both forward and deferred rendering, real-time global illumination, and a range of post-processing effects, which contribute to creating visually stunning environments. Unity's graphics pipeline is highly customizable, allowing developers to fine-tune their rendering processes to achieve the desired aesthetic and performance.

The Unity Editor serves as the primary interface for developers to interact with the Unity platform. The standard editor is shown in Figure 2.1. This tool allows developers to design and shape the virtual worlds in which their games unfold. Within the Unity editor, users can design environments, manage assets, adjust properties, and preview project files.

Unity is not limited to traditional gaming. Its applications extend to diverse industries. In architecture and construction, Unity is used for creating interactive walkthroughs and visualizations of buildings. In the automotive industry, it's used for designing and testing vehicle systems and for creating virtual showrooms. The film industry uses Unity for real-time cinematics, allowing filmmakers to visualize scenes with greater flexibility and speed. Education and training sectors leverage Unity to develop interactive learning modules and simulations, enhancing the engagement and effectiveness of their programs.

2.3 Case Study: the *Virtual Brigade* by the Royal Netherlands Marechaussee

The Royal Netherlands Marechaussee (RNLM) holds the responsibility of safeguarding the safety and security of the Netherlands and the Caribbean territories under the Kingdom of the Netherlands. Their deployment extends globally to locations of strategic importance, ranging from royal palaces to the outer borders of Europe, from Dutch and Caribbean airports to regions of conflict and crisis across the world.

The RNLM carries out three primary tasks, namely border police duties, monitoring and security, and international and military policing. The training of the marechaussees, or military police officers, takes place at the "Opleidings-, Trainings- en Kenniscentrum" (OTCKMar) located in Apeldoorn. To facilitate this training, the OTC employs innovative educational techniques, including serious gaming, virtual and augmented reality, and e-learning [59].

The *Virtual Brigade* operates as a simulation-building and player platform, providing a program with simulation-building capabilities to instructors. In this tool, instructors can construct scenarios tailored for students' educational purposes using the Editor. They have the flexibility to select various simulations of physical locations, such as Amsterdam Airport Schiphol, Binnenhof, or a highway, and subsequently populate the scenario with diverse elements, including objects, Non-Player Character (NPC), events, and dialogues. The decision-making process and the sequence of events within the scenario are determined by a flow diagram, which can be designed by instructors within the editor interface. Once a scenario is created, it is saved as a file that players can load to play the scenario.

This case study of the Royal Netherlands Marechaussee (RNLM) is particularly useful for testing Behaviour-Driven Development (BDD) because it embodies a complex, real-world scenario where the alignment between software development and end-user requirements is crucial. The RNLM's responsibilities span diverse and high-stakes environments, necessitating robust training programs that can adapt to a wide range of situations, from airport security to international conflict zones. The simulation-building capabilities provided by the Virtual Brigade allow for intricate and varied scenarios, making it an ideal candidate for BDD, which emphasizes clear communication and shared understanding between developers and stakeholders.

Applying BDD in this context ensures that the software development process is driven by the actual needs and experiences of the RNLM instructors and trainees. By focusing on creating detailed, behaviour-focused specifications, BDD helps bridge the gap between the technical team and the end-users, ensuring that the simulations are not only technically sound but also highly relevant and effective for training purposes. The use of BDD will facilitate the creation of realistic and reliable training scenarios that can accurately reflect the dynamic and unpredictable nature of the challenges faced by the RNLM.

Furthermore, the evolving nature of threats that the RNLM must prepare for underscores the need for a development approach that can rapidly adapt to new requirements. BDD's iterative process and emphasis on continuous feedback align well with this need for agility. As scenarios and threats change, BDD enables the development team to quickly update and refine the simulation software, ensuring that the RNLM's training tools remain up-to-date and effective.

Chapter 3

Research method

For each secondary research question, we will present the chosen approach. A global overview of each question's expected result and how these relate to the main research questions has been presented in section 1.2.

3.1 Q1: How can BDD scenarios be used to describe game behaviour?

The goal of this research question is to create a Game Behaviour Framework which can be used to categorize game behaviours. To achieve this we will identify common game behaviours and construct scenarios that describe these behaviours. To accomplish this, the following tasks will be executed:

1. Examine example projects and identify game behaviours.
2. Extract a categorization for game behaviours from the found behaviours.
3. Develop scenarios that articulate these identified behaviours.

To identify how scenarios can be used to describe game behaviour, a database of different kinds of game behaviours will be created. These behaviours will be collected from projects available within Unity's instructional resources.

The collection will take place from these projects as these are created to showcase different kinds of game behaviours in Unity. Furthermore, these projects are built by iteratively adding behaviours, thus allowing for easy extraction of the different behaviours.

Once a database of game behaviours has been created, we will attempt to create a categorization. To achieve this an informal variant of grounded theory will be applied. First, we will create abstract behaviours (concepts) from our database of behaviours. Using these abstract behaviours we will then look for possible categories. Once we are satisfied we will attempt to formalize these categories into the Game Behaviour Framework.

This categorization will aid in recognizing behaviours in different games and in choosing an appropriate approach for creating a scenario from the behaviour.

3.2 Q2: What method could be employed to apply BDD in the development process of a game?

In addressing this research question, our primary objective is to formulate a method that incorporates behaviour-driven development into the game development process. To fulfil this aim, we will perform the following steps:

1. Scan academic and grey literature to determine whether to use automated testing;

2. Do a literature search on the needed elements in the development process;

3. Formulate a method that incorporates BDD and the game development process.

To address this question comprehensively, our initial step will be to gather relevant literature. This we will do in two areas.

The first area in which we will gather literature is that of automated testing within the context of game development. BDD is mostly used in conjunction with automated testing. However, it is also possible to apply its principles without automated testing. Before we start formulating our method we will search for academic and grey literature. This will inform our decision on whether our method will use automated testing.

Meanwhile, we will search for literature to inform us of the current best methods in game development and BDD. From this, we will determine a base method, which we will then extend with elements from both BDD and game development.

These results will coalesce into one method that can be used for Behaviour-Driven Game Development. This method will be applied to a case study in subquestion 4.2.

3.3 Q3: What tooling can be developed to support BDD in Unity 3D game development?

To answer this research question, we will develop UnitySpec - a Behaviour-Driven Development (BDD) tool tailored for use within the Unity 3D environment. Our approach involves several stages:

1. Examine the state of the art of BDD tooling.

2. Develop UnitySpec, tooling to support BDD in Unity.

3. Create and apply the tool to samples.

To initiate our endeavour, we will conduct a search for existing tools designed for behaviour-driven development. This survey will encompass an evaluation of the functionalities these tools provide. By examining the offerings of these tools, we aim to identify the functionalities we wish to incorporate into UnitySpec. This process will be informed by the (partial) findings derived from our second research question (RQ2), which will help guide our decisions regarding the desired functionalities, specifically, whether or not the tool should support automated testing.

Next, we will develop tooling to support BDD in Unity3D. The features of this tooling will be guided by the state of the art in BDD tooling examined in step 1. From this examination, it will also be determined whether it is feasible to reuse parts of existing tools for the creation of UnitySpec.

Lastly, we will test the tool by applying it to samples. These samples will then be included in the package of the tool to showcase the tool's functionality. These samples will consist of basic scenarios and game elements to showcase and test the different features of the tooling.

3.4 Q4: What are the implications of applying our findings to a real-world case study?

The goal of this question is to evaluate the result of the previous two questions. We will utilize a case study to achieve this. For this case study a serious game currently in development at the Royal Netherlands Marechaussee (RNLM) has been selected. The game is called the *Virtual Brigade*. An introduction to the game can be found in section 2.3.

3.4.1 Q4.1: How well can the proposed Game Behaviour Framework be applied to a real-world case study?

This question focuses on evaluating the results from the first question. The result of this question will be the Game Behaviour Framework - a framework for transforming game behaviours into BDD scenarios. This framework will be developed using small, clearly defined games. For this question, we will evaluate how this framework holds up in a real-world game.

For this question, we will focus on version 1.0 of the *Virtual Brigade*. A roadmap has been created to guide development up to this point. This will serve as our reference to the behaviours that are present in the game.

Our approach will exist out of the following steps:

1. Examine roadmap for behaviours;
2. Classify behaviours described by requirements in the roadmap using the proposed Game Behaviour Framework;
3. Reflect on the applicability of the Game Behaviour Framework.

We will start by examining the roadmap with all requirements for version 1.0. For each of these requirements, we will classify the accompanying behaviour using the Game Behaviour Framework. Lastly, we will evaluate the Game Behaviour Framework using the teachings from this process.

3.4.2 Q4.2: What are the implications of applying BDGD and UnitySpec to a real-world case study?

This question focuses on applying the results from the second and third questions to a real-world case study - the *Virtual Brigade*. To evaluate the results we have employed the help of the development team.

For this evaluation, we will ask the development team to use our developed method and tooling for one feature. Meanwhile, we will observe the team to evaluate our method. We will also interview the involved team members to get their feedback on both the method and the tooling.

Our approach will contain the following stages:

1. Fill the required roles;
2. Select a feature for which BDD will be applied;
3. Explain the developed method and tooling to the involved team members;
4. Support team in creating the feature using the proposed method;

5. Interview the involved team members.

The case study will start by selecting participants from the development team. This choice will be made based on the availability and preferences of the team.

Before starting a feature should also be selected for which the BDD approach will be applied. This feature selection is based on the features available in the project backlog at the time of implementation.

Subsequently, the proposed BDD approach is introduced to both the development team and the product owner. During this introduction, they will receive the instruction needed to apply the method.

The specifics of the approach are determined based on the findings from the preceding research questions. The following description outlines the expected steps of the BDD approach, which may be subject to adjustments depending on the outcomes of the preceding research questions.

Following the introduction, a meeting will convene in which the development team and the product owner, guided by the researcher, will collaboratively formulate structured scenarios describing the behaviour of the proposed feature. The meeting will be concluded by an interview.

After the creation of the scenarios, the development team will be asked to implement the new feature. Throughout this implementation phase, the researcher remains accessible for guidance and to address any queries or concerns from the team.

Upon completion of the feature, the development team and the product owner are again interviewed. This interview serves to collect feedback from the development team regarding their experiences with the application of the BDD approach to the new feature. Additionally, interviews with team members serve to obtain their perspectives on changes in collaboration and communication resulting from the introduced approach, as well as the impact of structured scenarios on the development process.

Furthermore, the applied process and the feedback obtained are compared with the proposed Behaviour-Driven Game Development method, enabling an analysis of possible points of improvement of the proposed method.

Chapter 4

Related Work On Behaviour Driven Development (BDD)

Behavior-Driven Development (BDD) has been the subject of extensive research, exploring its advancements and potential in the realm of software development.

Farooq et al. [41] conducted a systematic literature review of the latest developments in BDD as of 2023. Their study assesses how BDD can address various challenges in software development and its impact on software development stages. We used their study as an overview of the latest developments in BDD and as a library of recent papers published on the subject.

In the same year, Binamunu et al [27] conducted a systematic mapping study to analyze the existing body of literature on BDD. They analyzed 166 papers published between 2006 and 2021. In these papers, they noted a sizeable use of case studies and experiments to evaluate the contributions of different BDD studies. However, they also noted the limited representation of research conducted in industry settings, exploration of BDD in conjunction with other techniques, philosophical discussions, and a notable scarcity of metrics for quantifying different aspects of BDD.

These works helped to identify four areas of BDD research that are of interest for this thesis. We will discuss the related works per area of interest. These two works together provided us with a sizeable, indexed library of literature on BDD which helped greatly in creating this related work chapter.

4.1 BDD for graphical interfaces

Within the multifaceted landscape of software development, the testing of Graphical User Interfaces (GUIs) stands out as a particularly intriguing facet. While this research has not been done with game UIs in mind, the principles and approaches to test GUIs can be relevant. The graphical nature of GUIs introduces challenges akin to those encountered in programmatic game testing.

Bahaweres et al. [17] employed BDD methodologies in the testing of CURA and Swag Labs web applications. They noted an increase in test response time. Their results are specific to their chosen frameworks and tooling.

Silva et al. [82,83] applied a different approach to introducing BDD in User Interfaces (UIs). They introduced an ontological model for describing interactive behaviours on UIs. It aims to support testing automation of, among others, interactive UI concepts. Subsequently, they harnassed this ontology to develop model-driven tools [81,84,85], which enable the modelling of BDD scenarios. Such a model allows defining a set of interactive behaviours on GUIs which can be reused to allow automated tests without intervention from a developer. The approach requires an initial manual assignment of identifiers but runs in a fully automated process afterwards. Their ontology is specific to GUIs of web and mobile applications. It shows a different, functioning, approach to using BDD in a graphical and interactive context.

Another approach has been taken by Bunder and Kuchen [29, 30]. They pursued a model-driven approach to fully generate automatically executable test cases, guided by BDD-like feature descriptions. Their research focused on GUI prototypes used during the development process.

These endeavours have traversed the terrain of structured GUIs. An unexplored domain lies in the application of BDD principles to graphical games. The present thesis will attempt to be a first foray into this domain.

4.2 Scenario Quality and Best Practices

Research question 1 asks how BDD scenarios can be used to describe game behaviour. Before we attempt to answer this question it is imperative to consider previous work on what makes a high-quality scenario and what the best practices are in creating scenarios.

A series of studies by Oliveira et al. [67, 68, 69] delved into the empirical assessment of scenario quality within the context of Behavior-Driven Development (BDD). The result of their work is a 12-question-based checklist for practitioners to help guide them in evaluating the quality of the written scenarios. These questions can be found in Table 4.1. Each question should be answered for its scope, i.e. questions 1 and 2 should be answered per feature, questions 3 to 8 should be answered for each scenario, et cetera.

In a parallel vein of research, Binamungu et al. [26] embarked on empirical investigations to discern the fundamental principles underpinning scenario quality. They found 14 quality aspects by examining both scientific and grey literature. These aspects are presented in Table 4.2. From this list they selected four pivotal principles, based on their potential to be precisely defined: the principle of conservation of steps, the principle of conservation of domain vocabulary, the principle of elimination of technical vocabulary, and the principle of conservation of proper abstraction. They surveyed the support of 56 practicioners on these four principles and found that they were all accepted by at least 75% of the respondents.

Contributing to the discourse, Bezsmertnyi et al. [22] distilled a set of best practices for BDD scenarios drawn from practical experience. These best practices encompass seven key guidelines, each aimed at enhancing the quality of scenarios. Recommendations include adherence to a standardized scenario naming pattern, limiting scenarios to a single action, adopting a third-person point of view, and managing the length of scenarios effectively. The authors underscore the significance of striking a balance between generic and specific scenario steps, as well as the meticulous use of "Given"-steps to prepare the scenario environment and employing tabulation for "And"-steps.

This research, while not from the domain of game development, informs on what high-quality BDD features, and -scenarios look like. We will test our proposed framework for crafting BDD scenarios from game behaviour against the advice and best practices gathered here in section 6.5.

4.3 Impact of Behaviour-Driven Development

In addressing Research Question 4, which revolves around the real-world application of the proposed method, it is imperative to review prior studies that have investigated the

ID	Question	Scope
1	Can the feature file business value or outcome be identified by its description?	Feature
2	Does the feature file has any missing scenarios?	Feature
3	Does the scenario carry all the information needed to understand it?	Scenario
4	Does the scenario has steps that can be removed without affecting its understanding?	Scenario
5	How different each scenario is from the others?	Scenario
6	Can the scenario single action be identified on its title and match what the scenario is doing?	Scenario
7	Can the scenario outcome or verifications be identified on its title and match what the scenario is doing?	Scenario
8	Does the scenario respect Gherkin keywords meaning and its natural order?	Scenario
9	Does the step correctly employs business terms, including a proper actor?	Step
10	Does the step has details that can be removed without affecting its meaning?	Step
11	Does the step express "what" it is doing by being written in a declarative way?	Step
12	Does the step allow different interpretations by being vague or misleading?	Step

TABLE 4.1: Question-based Checklist for BDD Scenarios [69]

S/n	Quality Aspect
1	A good quality scenario should be concise, testable, understandable, unambiguous, complete, and valuable
2	Reuse of steps across scenarios can improve suite quality
3	Declarative (high level) steps are preferred to imperative (low level) steps
4	Business terminology should be consistently used across the specification
5	Scenarios should focus on the benefit they offer to users, if implemented
6	Scenarios should use the terminology understood by all project stakeholders
7	Each scenario should test one thing
8	Scenario titles should be clear
9	Scenario descriptions should be focused
10	Personal pronoun “I” should be avoided in steps
11	Too obvious and obsolete scenarios should be avoided in the suite
12	Scenario outlines should be used sparingly
13	Scenarios should clearly separate Given, When and Then steps
14	Use past tense for contexts (Given), present tense for events (When), and “should” for outcomes (Then)

TABLE 4.2: BDD quality aspects from scientific and grey literature [26]

impact of Behaviour-Driven Development (BDD). The literature offers valuable insights into the benefits and challenges associated with the adoption of BDD methods, as well as the potential of this approach in real-world development settings.

Pereira et al. [70] conducted an empirical study focused on identifying the perceived benefits and challenges of implementing BDD. Their research revealed that the main advantages derived from adopting BDD practices are improved communication, enhanced collaboration among team members, and the creation of living documentation. In contrast, the key challenges observed included the presence of poorly written scenarios due to limited experience in crafting them, as well as the difficulty in convincing customers and fellow software developers to embrace the principles of BDD.

In a study conducted by Binamungu et al. [24] the focus was on exploring the utilization of BDD, with particular attention to the benefits and challenges of BDD adoption and the subsequent maintenance of specifications. The research revealed that the principal benefits encompassed the employment of domain-specific terms, an improvement in communication among stakeholders, the executable nature of BDD specifications, and the enhanced comprehensibility of code intentions. Counterbalancing these advantages were the findings that the adoption of BDD methodologies often required a significant shift in the way development teams approached software development. Moreover, a notable portion of respondents highlighted that BDD specifications could potentially encounter the same maintenance challenges frequently associated with automated test suites.

Nascimento et al. [65] conducted interviews with Agile development teams, aiming to ascertain the impact of utilizing Behavior-Driven Development in practical development scenarios. Their findings underscored the predominance of positive effects associated with BDD. Among the positive aspects, the research highlighted a more comprehensive understanding of features, the assurance of correct execution, better alignment within development teams, and a reduction in unexpected changes. In contrast, the primary drawback identified was the potential difficulty in implementing BDD practices.

This body of research provides valuable insights into the application of BDD methodologies, shedding light on the multifaceted impacts, benefits, and challenges that may arise. These findings provide a foundation for addressing Research Question 4 and offer a context for evaluating the impact of the proposed BDD approach within real-world game development settings. In section 9.3 we will compare the impact of BDD in our case study with the insights shared in this body of work.

4.4 BDD tooling

To address the research inquiry posited in Research Question 3, it is imperative to gain a comprehensive understanding of the existing BDD tools and their functionalities. First relevant literature will be considered. Following this a state of the art of BDD tooling will be drafted in subsection 4.4.1.

In this context, it is essential to consider the work of Pyshkin et al. [74], whose research in 2012 presented a comprehensive analysis of the state of the art in the domain of behaviour-driven development automation.

Pyshkin and colleagues undertook a meticulous examination that encompassed overviews of the characteristics inherent to various BDD toolkits. They provided a thorough assessment of the landscape at the time, offering insights into the existing functionalities and capabilities of these toolkits. Their findings indicated that the tools designed to support BDD primarily catered to developers, which ran counter to the fundamental concept of BDD, which aims to foster effective communication between

Name	Description	Note
BeanSpec [45]	Define predicates in code that can be run as tests	Abandoned
Concordion [3]	Tool to write executable specifications in a wiki format	
Cucumber Open [1]	BDD-framework that uses annotation-based binding of feature files to target language	
Cucumber Studio [1]	Extension for Cucumber Open, adding cloud-based collaboration platform	Integration
EasyB [4]	BDD framework for Java	Abandoned
FitNesse [5]	Fully integrated standalone wiki and acceptance testing framework	
Gauge [6]	Run specifications written in markdown	
jBehave [8]	BDD-framework for Java that uses annotation-based binding of feature files to Java methods	
JDave [9]	BDD framework for Java based on RSpec	Abandoned
qTest [13]	A Jira app to improve interaction with feature files	Integration
Squish [10]	Seperate IDE with support to create, record, maintain and debug Behavior Driven GUI Tests	
TestLeft [11]	IDE-plugin to support (BDD) development	Integration
TestRigor [12]	Executable specification engine	

Table 4.3: BDD tools
Grey indicates tools that will not be taken into account

technical and non-technical stakeholders. Consequently, they offered recommendations to developers, suggesting a shift in focus towards enhancing BDD tooling in line with this concept.

Moreover, in the realm of comparative studies, Lenka et al. [56] contributed in 2018 by investigating the differences between Test-Driven Development (TDD) and BDD. Their investigation extended to the examination of five BDD tools, namely Cucumber, Concordion, Jbehave, FitNesse, and SpecFlow.

The present thesis endeavours to build upon this foundation, extending the exploration of BDD tooling to the year 2023.

4.4.1 State of the art of BDD tooling

To help identify the functionalities needed to support behaviour-driven development in Unity 3D, we have commenced by examining existing BDD tooling. In Table 4.3 the discovered tools and a short description are presented. Cucumber Open, from here on out referred to as Cucumber, has support for a lot of different languages, and some of these implementations have their own names[1]. As these tools are all based on the same basis, and to keep the amount of tools manageable, these tools are not separately listed.

In this exploration tools that have had no updates for over 5 years are considered out-of-date and will not be considered. Tools whose only contribution is simplifying the interactions with other BDD tooling are marked as integration tools and will also not be considered. These tools have been marked grey in Table 4.3.

[1]Overview of Cucumber based tools: https://cucumber.io/docs/installation/

After applying these criteria, 7 tools remain: TestRigor, jBehave, Concordion, Cucumber, FitNesse, Gauge, and Squish.

In this section, the existing tools will be examined with two goals in mind. The first goal is to take stock of what features they offer to help determine what kind of features would be helpful to support BDD in Unity 3D game development. The second goal is to check the possibility of using one of the existing tools as a base on which a tool for Unity 3D can be built.

Tool Comparison

An overview of the main characteristics of the tools can be found in Table 4.4. Cells marked grey indicate a feature that makes it impossible to use the tool as a base. Each tool will be discussed separately in the order that they are in the table.

The main features that are discussed are specification language(s), mapping rules method, extendability, and programming language support [74].

The first two features are important for the first goal. They help determine what kind of tool it is and how it is used. The last two features are important for the second goal - finding a base on which a tool for Unity 3D can be built.

The specification language is the first feature that deserves our attention. In BDD the specifications are an important part of the process. In the specifications, the requirements and all test cases are documented. The specifications are also used as a means to communicate features between the development team and other less technical teams. Depending on how BDD is chosen to be used specifications could even be written by non-developers. The specifications look different in different tools and impact how they are written and who can write them.

After writing the specifications the tool should facilitate linking these specifications to tests to allow for automatic testing of the specifications. This is handled differently by different tools and can have a significant impact on the development process.

Extendability includes if the code is open source or if it has a clearly documented API. This is important for the second goal, as the tool needs to be extendable to be able to use it as a base for UnitySpec.

The last examined feature is programming language support. Given that the Unity3D API is written in C#, any selected tool must accommodate C# compatibility to serve as a viable foundation.

To be able to apply the tool in our case study it also has to be standalone and on-premise. This means that it should not depend on a central server, and cannot send any data through the internet. All data that the tool has access to, and results of the tool, should stay on the computer it is being run on, or it could be sent to an on-premise server.

After considering each of these features, combined with any other peculiarities, a judgement will be made if any part of the tool will be used for UnitySpec.

Concordion Concordion is an open-source tool for automating Specification by Example. It is distributed under the Apache license, allowing cloning, modification, and publication of the source code. Supporting both Java and C#, Concordion stands out due to its unique approach to specification language and its linkage to tests.

Unlike some other BDD tools, Concordion enables expressing specifications without a predefined structure. It uses HTML or Markdown for specification writing, restricting C# to HTML exclusively. Instead of matching structured steps to definitions, Concordion uses annotations within the specifications themselves.

Tool	Specification language	Mapping Rules	Extendable	Supported Languages
Concordion	Unstructured	in-line	✓	Java, C#
Cucumber	Gherkin	Annotations	✓	Java, C# & more
FitNesse	Tables	Table headings	✓	Extendable for all
Gauge	Lists	Annotations	✓	Java, C# & more
jBehave	Gherkin, own language	Annotations	✓	Java
Squish	Gherkin	Annotations / recording	×	GUI
TestRigor	own language	Code-less	×	GUI

Table 4.4: Main characteristics BDD tools
Grey indicates that a characteristic makes the tool unsuitable to be used as a base for UnityFlow

This approach allows users to view specifications in a wiki-like environment, presenting a more user-friendly experience for non-technical team members who can browse a website with regular text and embedded examples. However, this flexibility removes the structured requirements that BDD aims to enforce, potentially reintroducing ambiguity. Moreover, it makes it harder for non-technical team members to write their own specifications.

From a developer's perspective, managing spans in HTML files might be less straightforward than focusing on implementing a singular step in one method. Consequently, while Concordion's unstructured specifications offer benefits in terms of accessibility, they may compromise the precision and clarity that the BDD process seeks to maintain.

In conclusion, the choice of making specifications unstructured in Concordion may diminish the advantages of applying BDD in mitigating ambiguity, potentially impeding developers' workflows. Consequently, we have opted not to use Concordion as the base for the Unity3D tool.

Cucumber Cucumber has originally been created for Ruby. It has been extended to support many more languages. Interesting for our project, it has two extensions for C#: SpecFlow and Xunit.Gherkin.Quick.

The original Cucumber and Xunit.Gherkin.Quick have been published under the MIT licence, SpecFlow has been published under the 3-clause BDS licence. Both allow for use and redistributing of the source code.

Cucumber has designed and makes use of the Gherkin language. The Gherkin language is a plain-text, structured language. It uses keywords combined with natural language to express specifications in a "Given-Then-When"-structure.

To link the specifications to tests Cucumber uses annotations. Cucumber has defined annotations for each of the step-keywords and uses a regex-like pattern to match steps to the correct annotation.

Cucumber would be suitable to use as a base for our tool. SpecFlow and Xunit.Gherkin.Quick would both be suitable for this goal. Specifically, SpecFlow as it has a larger community supporting it and it offers plugins for IDE integration which simplifies using the tool. However, SpecFlow has closed-source extensions for advanced features like a dedicated runner and execution reports. The dedicated runner is not essential as it also offers NUnit integration to run the tests. In the open-source part only basic reporting in the terminal or IDE is supported.

FitNesse FitNesse is a standalone wiki and acceptance testing framework. Its core is written in Java, but it depends on the Fit or SLIM protocol to run tests in any language.

In FitNesse, tests are expressed as tables of input data and expected output data. The table title references the class under test, while the column headers reference methods on this class. Each row in the table is an example that is executed.

Like Concordion, FitNesse writes its tests in a wiki-based environment. This simplifies the interaction for non-technical users. However, FitNesse enforces that all tests are written in tables. This might work well for data-driven systems, but it is difficult to express game behaviours in a table. Therefore, we conclude that the approach chosen by FitNesse is unsuitable for use in game development.

Gauge Gauge is an open-source test automation framework. It supports C#, Java, JavaScript, Python, and Ruby. It is released under the Apache licence, allowing for the use and publication of its source code.

Gauge specifications are written in a syntax similar to Markdown. A specification exists out of a specification heading, optionally tags and comments, and scenarios. A scenario has a name, optionally tags and comments, and steps. Steps can have parameters.

Gauge links specifications to code using annotations. It can run tests in editor or using the command-line. It shows test results in the command-line and in a HTML-dashboard.

Gauge works similarly to Cucumber. The main difference lies in the lack of step-keywords. Where Cucumber differentiates between given, when, and then-steps, in Gauge all steps are the same. This gives more freedom in the defining of the steps.

Gauge also allows defining concepts, also known as composite steps. This allows for the definition of a custom step by combining substeps.

jBehave jBehave is an open-source framework for BDD in Java. It supports Gherkin-syntax, and an own, similar language for writing stories. It uses annotations to map these steps to Java-code. It does not support C# and can thus not be used as a base for the tool for Unity3D.

jBehave's scenario language differentiates itself from Gherkin by the inclusion of Composite Steps, defining a step using sub-steps. This feature has also been seen in Gauge. Besides this jBehave offers no interesting features different from Cucumber.

Squish Squish is a closed-source testing toolkit with BDD support. It is compatible with the Gherkin language to create GUI tests. It supports applications within the browser and native applications for Windows, macOS, Android and iOS.

Squish allows the user to write tests (BDD and non-BDD). Its main contribution lies in the ability to record tests. In the case of BDD tests, Squish allows the user to record what a step should do by doing it themselves. With proper usage of this feature, users can write tests without ever having to touch code.

When inspecting the code generated by Squish we can see that it uses annotations to match code to steps.

Squish does not support Unity3D, and since it is closed-source, can not be extended to do so.

There is a similar tool for Unity3D, called gamedriver, which also allows recording user input and automatically generates tests. However, this tool does not support BDD and is also closed-source, making more detailed inspections impossible.

TestRigor TestRigor is a closed-source, generative AI-based, test automation tool. Their faq states:

> " You can think of testRigor as Cucumber already implemented for your project. We have eliminated the need to write and maintain any underlying Selenium

> code, as our system uses an NLP-based parser to parse plain English and executes your specifications in place. " [12]

TestRigor specializes in testing form-based UI or functionality with predictable input/output and specifically does not test games. This makes it unsuitable for the goal of this thesis.

Its approach of using a NLP-based parser seems promising. However, as it is closed source it cannot be further investigated. For this project this approach is far out of scope.

Conclusion

Different tools for BDD exist, each with different approaches and strengths. For the goal of supporting BDD in Unity3D two tools stand out. Cucumber for C#, specifically SpecFlow, and Gauge.

The main consideration in choosing between these two lies in their specification language. Where SpecFlow uses the Gherkin-language, enforcing structure, Gauge allows for more freedom using bullet points. For this research, the structure that SpecFlow enforces has the preference, as that is an integral part of BDD.

In this consideration, the development status, extra features, and expected effort to adjust for use in Unity of both tools have been considered. From this consideration, the decision has been made to use SpecFlow as a base for the development of UnitySpec.

Chapter 5

Related Work In Game Development

5.1 Requirements Engineering in Games

As our research encompasses the integration of scenarios as a means of documenting requirements in game development, it is essential to examine the domain of requirements engineering in games. We draw from studies that provide insights into how requirements are perceived and managed in the gaming industry.

Kasurinen et al. [53] investigated the application of requirements engineering practices within game development organizations. Through interviews with software professionals in various game development settings, the study unveiled that practical considerations and the pursuit of an enjoyable gaming experience dominate the landscape of requirements engineering. While game development organizations employ approaches akin to requirements engineering and requirement management, they do not consciously adhere to conventional requirements engineering practices.

Daneva [37] delved into how practitioners in Massive Multiplayer Online Games (MMOGs) perceive and manage gameplay requirements in their projects. This research, based on interviews with practitioners from leading MMOG-producing companies, revealed that gameplay requirements in MMOG projects are collaboratively created with players. These requirements are regarded as sets of choices and consequences, and gameplay in MMOGs is considered an ongoing, endless experience. The study underscored the pivotal role of "paper-prototyping" and play-testing in gameplay validation.

These insights into requirements engineering in the gaming industry provide valuable context for our research into integrating scenarios as a means of documenting requirements in game development. The gaming industry's emphasis on practicality, player engagement, and ongoing gameplay dynamics informs our approach to creating and using scenarios effectively. Our research builds upon these foundational principles as we seek to optimize the role of scenarios within Behavior-Driven Development for game development.

5.2 Testing in Game Development

The application of scenarios for testing games demands a comprehensive understanding of the testing landscape in Game Software Engineering. First a general overview of the landscape will be presented, including an overview of work that applies testing specifically in Unity. Following this a more in depth look will be taken into automated testing in game development. From this work it will be determined whether it is feasible to apply automated testing to support behaviour-driven development.

Politowski et al. undertook a comprehensive exploration of the state of testing in video games, with studies in 2021 [72] and 2022 [71]. Their findings indicated a degree of scepticism and resistance to the notion of automated testing within the gaming industry, providing insights into the industry's stance on this testing approach. The main point of resistance is that automated game testing is seen as a "waste of time and money that can be spent somewhere else" [71]. In their other paper, they note a lack of well-defined testing strategies, a focus on the search for the "fun-factor", and an overall lack of plan [72].

Kasurinen et al. [54] undertook an analysis of how game development organizations approach the testing of their products and their primary test objectives, while also examining the organizations' self-perception within the broader software business context. Their findings revealed distinctions in testing priorities between game developers and traditional software developers. In game development, testing outcomes exert a greater influence on the final product, and the testing process is highly focused on softer aspects, including internal mechanics, game rule balance, and user experience. Game organizations allocate resources for technical testing but often prioritize explorative and usability testing to enhance the user experience. Technical concerns are typically secondary, even among developers and testers responsible for most technical software development tasks. In essence, game development projects aim to deliver the necessary functionality within budget and schedule, emphasizing emotional appeal to customers.

Mirza-Babaei et al. [61] contributed valuable insights into the role of playtesting, particularly within small independent studios, through a series of case studies in commercial indie games. They underscored the significance of cost-effective playtesting and provided methods to incorporate playtesting into the indie development cycle.

Their subsequent study [62] examined the quality of features in commercial games through the analysis of playtesting reports and game reviews. This work emphasized the importance of playtesting in validating game features and introduced guidelines for structuring playtests and selecting methods according to their ability to address specific game features.

Masella [60] and Baker [18] presented a practical, real-life-tested method for incorporating automated tests into game development, specifically focusing on gameplay features. They prioritized automated testing for its speed, precision, and ability to interact with the game at various levels, allowing human testers to focus on visual, audio, and exploratory testing, and overall game experience assessment. Their approach included various test types, with integration tests being particularly noteworthy. These tests target specific game behaviours as predefined scenarios to assess whether they occurred as expected. These tests followed a three-phase process, involving map setup and asset loading, behavior execution through simulated player input, and result verification. To manage test runtime, actor tests were introduced. Actor tests are described as a unit test for game code, one that treats Unreal engine concepts like actors and components as first-class dependencies. This means that Actor tests do not start the game or scene but run on a code-level. Unreal is one of Unity's competitors. They found the best balance between fast execution time and reasonable test coverage by using integration tests for the successful execution of gameplay scenarios and using actor tests for the failures and edge cases. The main benefits Masella found of using this approach are reduced build verification time, decreased manual testing, fewer bugs, and minimized development crunch.

Collectively, these studies offer a nuanced understanding of testing practices in the gaming industry, laying the foundation for our research into leveraging BDD for game testing. Our work builds upon these insights as we design a method to introduce Behaviour-Driven Development in game development.

5.2.1 Testing in Unity 3D

In the realm of Unity 3D game development, the facet of testing has remained in the shadows. Nevertheless, some recent work has been done to explore the nuances and challenges associated with testing in game development and tools tailored to Unity 3D. This subsection will highlight some of these forays.

Jurvanen [51] made notable strides in the realm of automated testing within the Unity 3D framework. Jurvanen addresses the challenges and opportunities associated with automated testing specifically in Unity 3D, emphasizing its significance in streamlining the testing process within this game development environment.

Meanwhile, Ghayyur [44] has delved into the domain of mutation testing within the Unity 3D ecosystem. Mutation testing is a technique used to evaluate the effectiveness of test suites by introducing artificial faults, or mutations, into the source code and assessing whether the tests can detect these changes. While this work provides valuable insights into testing strategies, it primarily focuses on mutation testing's applicability in Unity 3D.

The Unity Test Framework (UTF) stands as a crucial tool in the Unity 3D ecosystem, enabling comprehensive testing capabilities. UTF facilitates testing not only in Play Mode but also in Edit Mode, offering developers a robust suite of testing options for various target platforms. Play Mode tests are executed in the application's runtime environment whereas Edit Mode tests are executed directly in the editor's environment. The advantage of Edit Mode tests is that they are quicker - because they do not have to launch a separate environment - and that the tools and methods of the Unity Editor are available. This framework is built upon the NUnit library, a renowned open-source unit testing library for .NET languages [39]. Its versatility and compatibility with Unity 3D make UTF a valuable asset for ensuring the functionality and reliability of Unity-based projects.

An earlier endeavour to integrate Behavior-Driven Development (BDD) principles into Unity 3D was made through the tool known as Cukunity [43]. However, Cukunity presented certain limitations, such as exclusive compatibility with MacOSX, support restricted to Android 2.X and iOS 5.x applications, limited coverage of Unity features, and a lack of maintenance since 2013. Cukunity's primary contribution was bridging the gap between the Cucumber testing framework and Unity 3D, showcasing an initial attempt to apply BDD concepts in this game development environment.

While these research efforts provide valuable insights into testing practices and tools within the Unity 3D framework, there remains substantial room for further exploration and development in this domain. This thesis aims to contribute to this evolving landscape by proposing a method that combines BDD principles with Unity 3D, offering innovative strategies for enhancing game development through structured scenarios and automated testing.

5.3 In-depth review on automated testing

This review aims to answer the question if it is feasible to use automated testing to test games. In short, the answer to this question is yes, as evidenced by the fact that automated testing is used by major game studios like Activision [91], Riot Games [79], Crytek [31], Rare [18, 60], Ironhide Game Studio [35], and VBlank Entertainment [73]. Their experiences will be considered as valuable insight during this literature review. This will be combined by the experiences from gamedeveloper.com, a self-proclaimed "leading resource and reference for game development and industry knowledge" [38, 48]. Next to these personal insights we will also consider academic literature written on the subject [49, 54, 57, 64, 72, 75, 76, 78].

Despite the evidence that it can be beneficial, there is a lack of test automation in game development, as highlighted by Murphy-Hill et al. [64], Politowski et al. [72], and Hooper [49]. They suggest reluctance in the game industry to adopt automated testing practices. To expand on this a discussion on the benefits of automated testing will follow, followed by a discussion of common arguments against the use of automatic testing. Lastly a compilation of advice for applying automated testing will be presented.

5.3.1 Advantages of automated testing

The review will commence with an exploration of the advantages of automated testing. Initially it will delve into the broader benefits of automated testing, discussing the positive impact on software development irrespective of the application domain. Subsequently, the focus will be narrowed to the distinct advantages that automated testing can bring to game development.

Advantages of automated software testing

Rafi et al. [75] conducted a systematic literature review and practitioner survey to determine the main benefits and limitations of automated software testing (AST). Combining the findings from their literature review and survey we find there to be two main benefits.

Improved product quality Automated software testing enhances software quality by offering improved test coverage and systematic verification, resulting in fewer defects in the software product [16, 28, 52, 58, 87]. AST allows for comprehensive and repeatable testing, ensuring that various aspects of the software's functionality are rigorously examined [14, 52]. This systematic approach to testing helps identify and address issues early in the development process, contributing to a higher-quality end product [33, 42, 77]. Additionally, the repeatability and reliability of automated tests enable developers to catch defects promptly and maintain consistent software quality throughout the development lifecycle [40, 46, 52, 92].

Time and cost savings Automated software testing contributes to time and cost savings in software development [14, 16]. Its efficiency in test execution allows for swift and repeatable tests, accelerating the verification of software functionality [40, 46, 52, 92]. Regression testing automation and the ability to execute tests quickly and on-demand enhances efficiency, reducing the time required for testing cycles [14, 52]. This ensures a faster feedback loop, enabling quicker identification and resolution of issues. By minimizing human effort in repetitive testing tasks, automated testing frees up time to spend on other tasks [14, 20, 40, 55].

In addition, automated software testing is cost-effective by enabling the execution of tests at no additional cost once created. While there is an initial investment in developing automated tests, the long-term savings in manual testing effort and time are said to outweigh this cost [14, 52]. Moreover, automated testing contributes to a reduction in defect-fixing costs by facilitating early detection, preventing the accumulation of defects, and addressing issues at their source [52, 58].

Advantages in game development

The advantages of automated software testing are also applicable in the realm of game development. However, this section will highlight some common characteristics of game development to show that automated testing's benefits transfer to this domain.

The key to a good game design is the constant experimentation of new features [57]. The risk with a piece of software that is constantly changing is that new features are likely to break previously implemented features [60]. Automated testing allows the developer to pin down the behaviour of finished features in tests, ensuring that no bugs are introduced in finished features [60]. This reduces the workload on development teams by reducing the effort of manual testing and allowing bugs to be found as soon as they are introduced [31]. This leads to significantly compressed turn-around time [79]. This, in turn, leaves more time to be spent on polishing the game and reduces the chance of going over deadline or budgets [31, 60]. By pinning down expected feature behaviour, tests are also a good form of extra documentation, giving more clarity on how features are meant to work during development [35]. This also allows for late-term changes to core features while ensuring that all other features remain working as intended [31].

Finding all the bugs and checking if old bugs have not been reintroduced is a repetitive and time-consuming task. This nature makes it challenging for human testers, causing them to potentially overlook defects [72]. Computers excel in consistently rerunning the same scenarios, checking if no known defects have resurfaced [79, 91]. Besides missing steps due to repetition, human testers also often get motion sickness by constantly replaying the games they are testing [72]. Properly used automation reduces the burden on the development team [72]. This allows the human testers to spend more time on tasks that humans excel at, like exploratory testing [91], evaluating the fun-factor [79], and checking if the user interface (UI) is intuitive [79].

This all combines to show that automated testing allows for the creation of higher-quality code while reducing the burden on the whole development team. Yet, there is a reluctance to adopt automated testing in the game industry [49, 64, 72].

5.3.2 Arguments against automated testing in game development

A common argument in and outside the realm of game development against the use of automated testing is that automated tests cannot test everything in a product. This is why we, together with most advocates do not suggest to only use automated testing. Automated testing should be used as a tool to do the kind of tests that it is well suited for, leaving the human testers to do the kind of tests that they are better at [31, 35, 38, 49, 60, 72, 78, 79, 91].

Game developers hold themselves apart from software development, considering their work to be creative in nature and only using software as a means of expression [54]. This is a commonly cited argument against the use of automated testing, namely that automated tests cannot test if a game is fun to play [49, 72]. However, this is not the goal of automated testing. Automated testing should be used to test the quality of the game software. A game that is riddled with bugs is certainly not fun to play. When the quality is assured the fun can be tested by human testers [38]. As stated before, automated testing should complement the human testers; both working on the parts that they do best.

Another argument against automated testing lies within the volatility of the code. The key to a good game design is the constant experimentation of new features instead of preset requirements and best practices [57]. This is recognized by advocates of automated testing sharing how they apply it [31, 60]. These suggest to first try out changes in an untested branch. When these work play-wise they should be integrated into the main branch with tests to pin down the behaviour of the new feature and guarantee that it does not change in the future. This way the flexibility needed to try out new things is not limited, and the strength of automated tests is used to ensure that the feature remains working as intended.

Another argument against automated testing heard inside and outside the realm of game development is that writing tests takes time away from writing code [31,49]. However, this notion ignores the time that is saved by introducing testing in the development process mentioned before. Furthermore, Carucci [31] argues that tests written test-first, as done in TDD and BDD, are part of the design process, a step that has to be taken anyway. Davis [38] writes that working test-first help jettison busy work that does not add value to the product, thus saving time. Davis also argues that “Culling unnecessary scope early and quickly is one of the most effective productivity enhancements you can make” [38]. However, whether this actually holds is highly dependent on the project and the team working on it.

In summary, the integration of automated testing in game development appears to be a powerful tool. There are a number of arguments as to why it might not offer benefits, but we believe that by creating a custom method the benefits will outweigh the drawbacks. In section 7.5 we will reflect whether this remains true. Furthermore, in section 9.5 we will further reflect whether this holds in the real-world.

Chapter 6

Game Behaviour Framework: Behavioural patterns in games

In order to determine how BDD scenarios can be used to describe game behaviour we set out to create a categorization of behavioural patterns in games. To create this categorization we first created a database of game behaviours.

These game behaviours have been derived from projects within Unity's instructional resources, specifically from the "Create with Code" course[1]. The collected behaviours are presented in Appendix A. This course has been selected for three reasons.

- The course aims to be a general introduction, showing different kinds of games with different behaviours;
- The course builds a number of games from scratch, giving insight and access to all components of the games;
- The course is structured in a way that introduces one behaviour at a time to each of the games, this allows us to easily extract behaviours.

From this database of behaviours, we formulated abstract behaviours. For example, the behaviour "Press W to move forward" got abstracted to "Player Movement - use keys to move the player". While doing so we saw a pattern emerging. We have attempted to define this pattern and will present it in this chapter.

We propose to categorize game behaviours using three categories, which we will refer to as levels. Level 1 contains basic behaviours, level 2 singular interactions, and level 3 contains game elements. Below we will further specify the scope of each of these levels.

The goal of creating these levels is to subdivide the behaviours into different levels of complexity and abstraction. Each of these levels can then build upon the behaviours of the previous levels. We will demonstrate this principle using an example.

Example An example of a level 3 behaviour is a basic fight mechanic. In this imaginary game, there is a player, a shooting mechanic, and some form of enemy. We will imagine the player needs to shoot (and hit) the enemy 2 times to "kill" it. In BDD it is recommended to use a declarative style to describe features [36,80]. In line with this recommendation, we believe that it is most intuitive to think about a mechanic like this in an abstract manner. Going back to the example it could be described like "When the enemy is shot 2 times then it dies".

This would be a great start to writing specifications for this behaviour. Yet, this leaves us with no specification of how the shooting mechanic works. The player could be shooting

[1]https://learn.unity.com/course/create-with-code

	Category	Explanation
1	Player Movement	Moving a player object using keys
2	Camera Movement	Moving the camera using user input
3	Scene Loading	Successfully load a scene
4	Assets Exist	Check that assets are in scene at boot
5	Booting	Starting application
6	In-menu Navigation	Menu interactions that stay in scene
7	Out-menu Navigation	Menu interactions that lead out of the current scene
8	Independent Behaviour L1	Object constantly shows a level 1 behaviour without input, e.g. constant movement
9	Persistent storage	Something is stored and persists in another level
10	Sound	A sound plays
11	Spawning	A new object is created

Table 6.1: Overview of level 1 game behaviours

a bow, a gun, or even throwing a rock. To specify this we will look at the feature from a lower abstraction level. We have defined shooting as a level 2 behaviour. The behaviours in this level are slightly less abstract and complex than those in level 3. Let's imagine our game is a first-person shooter. We could now describe the shooting behaviour as "When the left mouse button is clicked then a bullet is spawned and moves in the view direction".

These descriptions almost completely specify how an enemy can be shot and killed. Yet, in the level 2 behaviour, it is still abstract how the view direction can be changed. This we have categorized as a level 1 behaviour. This is the most concrete and simple level of behaviour descriptions. In this level we can specify "When the mouse is moved left 10 degrees then the view is moved left 10 degrees".

These three descriptions together describe all behaviours needed to shoot and kill an enemy. From very simple - moving the camera - to slightly more complex - shooting the gun - to a complete game mechanic - shooting and killing an enemy.

The third level is a very broad category, it could be described as anything above level 2. One could conceive many more levels to further divide the level 3 behaviours, with the highest level possible being playing the game from start to finish. However, for this research, we will limit ourselves to three levels as we believe that to be enough levels to make meaningful observations without getting lost in possibly endless complexity.

For each level, we will present a table with some observed example behaviours in that level. Per level, we will present and explain a feature for one such behaviour. In Appendix A "Observed Game Behaviours" the observed games and their behaviours per category can be found. The presented behaviours should not be taken as an exhaustive list of all possible game behaviours as creating such a list would be a Sisyphean task.

6.1 Level 1 - Basic controls

The first level contains very basic behaviours. These behaviours concern only one object (asset) or span only one frame. These behaviours form the building blocks for more complex behaviours.

Table 6.1 contains a non-exhaustive list of commonly observed behaviours.

The first behaviour, *Player Movement*, can be seen in any first- or third-person game. A basic example of this behaviour, from game 1, is controlling a car using the `wasd`-keys.

```
Feature: WASD moves player
    As a user
    I want to be able to move using the wasd-keys

    Scenario Outline: Basic movement
        Given I load the level "move-test"
        When I press <key>
        Then the player moves <direction>

        Scenarios:
            | key    | direction |
            | w      | forward   |
            | a      | left      |
            | s      | backward  |
            | d      | right     |

    Scenario: Press two keys
        Given I load the level "move-test"
        When I press both 'w' and 'a'
        Then I move both 'forward' and 'left'
```

Listing 6.1: Player Movement Feature

This behaviour could be described as in Listing 6.1. This feature has the title "WASD moves player" and a user-story describing the desired behaviour. Furthermore the feature contains a scenario outline which gets filled by each of the individual keys. Lastly, the feature has another scenario for the expected behaviour when pressing two keys at the same time.

6.2 Level 2 - Singular interactions

The second level contains singular interactions. The behaviours in this level concern 2 or more objects and take place and/or can be evaluated in a single moment. Table 6.2 contains a non-exhaustive list of commonly observed, abstract behaviours.

	Category	**Explanation**
12	Independent Behaviour L2	Object constantly shows a level 2 behaviour without input, e.g. the camera staying behind the player
13	Special Control	A user interaction trigger, e.g. pausing on a keypress or destroying an object on click
14	Colliding	Two or more objects collide
15	Simple Condition	A level 1 behaviour depends on the current state
16	Throwing/shooting	A user interaction spawns an object that moves in a direction
17	Counter	A counter is updated or read
18	Collision Trigger	Something happens when two or more objects collide
19	Movement Trigger	Something happens due to something else moving
20	Location-based Trigger	Something happens due to something reaching a location
21	Time-based Trigger	Something happens due to an internal timer
22	View-based Trigger	Something happens due to what is in view
23	Event-based Trigger	Something happens due to another event

TABLE 6.2: Overview of level 2 game behaviours

Listing 6.2 shows the specification of a feature from game 3. In this game the player throws food at approaching animals in order to feed them. This behaviour belongs to category 16 “Throwing/shooting”. This feature contains a scenario describing what throwing food entails. It also includes a scenario for what happens if two items get thrown immediatly after each other.

6.3 Level 3 - Game elements

Level 3 describes game elements. Game elements span multiple frames and concern multiple objects. The behaviours in this level are built upon multiple level 2 (or 1) behaviours. One of the simplest examples of a behaviour in this level can be observed in game 3, a fight with a single animal. This includes loading a level with the player and an animal (level 1), throwing a piece of food at the animal (level 2), having the food hit the animal (level 2) and having them both disappear (level 2) and your score increased (level 2). This is expressed in Listing 6.3. Notice how this feature has the same description as the feature in Listing 6.2, yet this is a level 3 behaviour whereas Listing 6.2 is a level 2 behaviour. The previous feature only describes the throwing of food, e.i. when the player presses the spacebar an instance of food gets spawned correctly. Whereas this feature describes the complete interaction of throwing food at an animal. It no longer specifies that the spacebar should be pressed and that food should be spawned, instead, it looks from a higher abstraction level and simply states that the food gets thrown.

```
Feature: Throw food
    As a player
    I want to throw food
    In order to feed the animals

    Scenario: Throw food
        Given I load the level "throw-food-test"
        When I press the spacebar
        Then a new instance of food should be spawned
        And it should have a speed of 10
        And it should move forward from my location

    Scenario: Double-tap spacebar
        Given I load the level "throw-food-test"
        When I press the spacebar
        And I press the spacebar
        Then two instances of food should be spawned
```

Listing 6.2: Throw food

```
Feature: Hit animal with food
    As a player
    I want to hit animals with food
    In order to feed them

    Background:
        Given I load the level "feed-animal-test"
        And there is an animal

    Scenario: Throw food and hit
        Given I have a score of 0
        When I throw food in the direction of the animal
        And I hit the animal
        Then the animal should disappear
        And the food should disappear
        And my score should be 1

    Scenario: Throw food and miss
        Given I have a score of 0
        When I throw food
        And I do not hit the animal
        Then there should be an animal
        And my score should be 0
```

Listing 6.3: Feed animal

	Category	Explanation
24	Independent Behaviour L3	Object constantly shows a level 3 behaviour without input, e.g. full npc-behaviour
25	Fight	A fight between the player and enemy
26	NPC Creation	Condition triggers asset creation at a specific location with some behaviour
27	Powerup	Pickup item which then has effect
28	Extended Condition	A behaviour depends on factors beyond Simple Condition (15)

TABLE 6.3: Overview of level 3 game behaviours

6.4 Conclusion

The Game Behaviour Framework provides an approach to describe game behaviours by categorizing them into three levels of complexity: basic behaviours, singular interactions, and game elements. The essence of this framework lies in categorizing behaviours by the amount of interactions that take place.

To apply the Game Behaviour Framework we recommend asking 2 questions, these have been visualized in Figure 6.1.

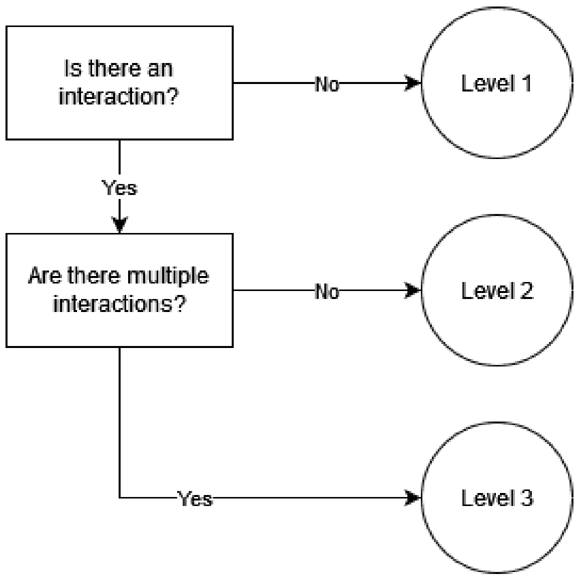


FIGURE 6.1: Decision diagram for determining behaviour level of requirement

In section 9.1 we will apply the Game Behaviour Framework to our case study. There we will reflect on the applicability of the framework using real-world requirements.

6.5 Reflection

In chapter 4 we presented relevant literature on BDD-scenario quality and best practices in scenario creation. In this section, we will reflect on the gathered advice and evaluate the Game Behaviour Framework against this.

Most quality aspects mentioned in the literature are dependent on the practitioner writing the scenarios and are not influenced by the framework. However, one of the key principles presented by Binamungu et al. [26] is the principle of conservation of proper abstraction.

On this principle, Binamungu et al. write “the abstraction levels should be determined by capturing correct requirements and producing scenarios that are readable to customers; lower abstraction levels can be appropriate if scenarios carry data; sometimes, one can use different abstraction levels for Given, When, and Then steps.” [26]. Similarly, Bezsmertnyi et al. recommend finding the golden mean between generic and specific steps [22].

The proposed framework offers guidance in finding this golden balance between generic and specific steps. It does this by defining abstraction levels. This encourages practitioners to think about their abstraction and separate basic keypresses from complex behaviours.

Chapter 7

A method for Behaviour-Driven Game Development

In this chapter, a method for behaviour-driven game development will be introduced. This method aims to answer subquestion two "What method could be employed to apply BDD in the development process of a game?". We will test and evaluate the proposed method in section 9.3 "Applying Behaviour-Driven Game Development and UnitySpec".

7.1 Inspiration

Applying BDD complements agile development practices. In developing this method inspiration has been drawn from SCRUM, as the most common application of agile practices [86], and TDD, for its method-likeness to BDD [66].

SCRUM has three main phases, described below [15]:

1. **The initial phase**: The general objectives are outlined and the required team, tools, and resources are indexed. In this phase, a backlog is created with the user requirements as user stories and features.

2. **Sprint phase**: This phase consists of a series of sprints, each adding value to the system. Sprints have a fixed length of 2 to 4 weeks.

3. **Closure phase**: This phase is entered when the requirements are achieved. The product is now ready for release.

Each sprint starts by creating a planning from items from the backlog. Once the planning is complete the team implements each of these work items. During development, the team should hold daily meetings to discuss progress. At the end of the sprint, a review takes place. In this review the sprint output is inspected, analyzed and assessed, and the team reviews their processes and collaboration.

Exactly how development should be done is not prescribed by SCRUM. For more guidance, the TDD process can be used. TDD prescribes that for each work item, the following steps should be taken [15]:

1. Write automated tests;
2. Run all test cases to check failure;
3. Write code;
4. Run all test cases to check passage;
5. Refactor, while running tests to check that all functionality remains.

Combining these two approaches creates a comprehensive base. From this method, only a few modifications are needed to support BDD in game development.

7.2 Incorporating BDD

The first modification needed is to use behaviour-driven development instead of test-driven development. To achieve this the method should be adapted to use behaviours instead of tests. Practically this means that automated tests get replaced by scenarios describing behaviour.

In order to create these scenarios, literature suggests using three steps [2]:

1. **Discover**: A discussion on the new functionality during which concrete examples should be generated and there should be agreement on the details of what should be done.
2. **Formulate**: Document the examples using scenarios.
3. **Automate**: Create tests from the examples to guide development.

These three steps will generate written scenarios and tests. With these tests, it is then possible to continue with the TDD process starting from step 2.

BDD stresses the importance of collaboration of different roles within the team [56]. Specifically, three roles, also called the "Three amigos" [2]. These three roles are [74]:

- The product owner;
- The tester;
- The developer;

The product owner is concerned with the features of the product, the tester is skilled in finding edge cases, and the developer can see the details of how a requirement fits into the product.

In practice, it is less important what the job descriptions of the participating amigos are, and more that these three viewpoints are represented.

7.3 Adjusting for Game Development

Game development is sometimes said to exist out of constant experimentation of new features instead of following pre-determined requirements [57]. To find a balance between structured development and artistic freedom this method encourages the use of prototyping [64].

The goal of prototyping is to test out ideas, to figure out if they work game-play-wise, without worrying about writing high-quality code or writing tests. The artefacts created in this step should be unpolished and deleted at the end of the prototype. Including prototyping allows for a balance between the need to quickly test out things while looking for the fun factor and the need for well-documented and tested code [38, 60].

Prototyping can be done at any point when a team member has an idea for a feature but is still unsure how it would look in the product. It can be specifically useful when picking up a new work item. Here a prototype can help guide the team members in formulating the examples of the work item.

Exploratory testing is very important in game development [48, 78]. In the proposed model it is recommended to incorporate this into the sprint. During the review-step extra time should be taken to incorporate exploratory testing.

7.4 Complete method

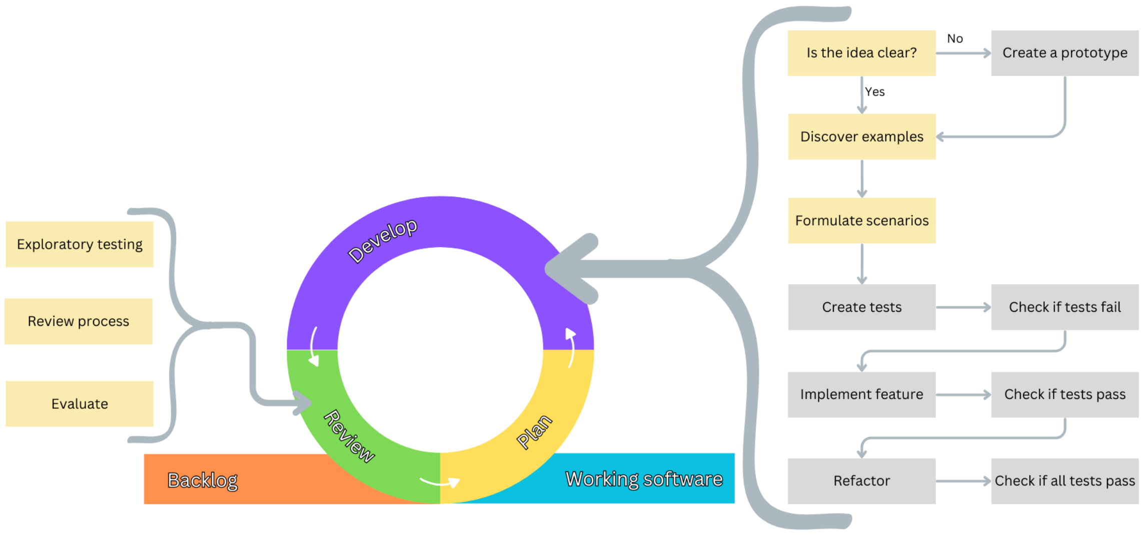


FIGURE 7.1: Method for BDD in Game Development
Illustration of method designed and described in chapter 7

Figure 7.1 illustrates the complete method. The bars illustrate the start-up and closing phases from SCRUM, phases 1 and 3 respectively, while the big circle illustrates a sprint. Inside the sprint, a workflow, based on TDD [15], is pictured for each work item. In this workflow yellow indicates steps that should be taken together by the three amigos while grey indicates steps taken by one.

The project starts with the initial phase as known from SCRUM. The main resulting artefact of this phase is a list of prioritized items that need to be implemented, henceforth called a backlog, and a complete team to do so.

Then the team will go through sprints. Each sprint starts by creating a plan for the sprint, taking work items from the backlog.

For each of these work items the developer, tester, and product owner come together to brainstorm examples of what the behaviour of the feature should be in different circumstances and document these in scenarios. If the product owner is not present during the writing of the scenarios then he should actively check the scenarios. Scenarios are the way to communicate clearly what a feature will look like in a way that can be tested and can be understood by the product owner. To aid this discussion a prototype can be created.

Once the scenarios are formulated the developer can write the code needed to turn the scenarios into automated tests. Once these are written the developer should check that the tests fail to ensure that they accurately describe new functionality.

Next, the developer can implement the work item, using the scenarios as a guideline. The new automated tests should be used to check that item is completed.

Once the new feature is implemented the developer should check if all tests still pass to ensure he has not introduced a bug elsewhere. Some attention should be spent to check if any refactoring is needed to refine the code or remove duplications. If any refactoring is done it should be checked if no unintentional behaviour changes are introduced by running all tests again.

When all work items are implemented and merged a new build with all additions from the current sprint should be created. This build should be used to review the added work.

This build should also be used for exploratory testing, looking for bugs or new ideas in the product. If any are found they should be added to the backlog. Lastly, the team should take a moment to review their processes and discuss any problems or good things that occurred during the past sprint.

Once all items from the backlog have been implemented the game is done and ready to be shipped.

7.5 Reflection

In section 5.3 we have reviewed the arguments in favour and against automated testing of games. We concluded that the benefits outweighed the drawbacks when applied properly. We will now discuss whether BDD and our proposed method properly mitigate these drawbacks. In subsection 5.3.2 we identified three main arguments against automated testing in game development. We will discuss each of these in the order they were presented.

Fun-factor The first argument against automated testing is that it cannot replace the function of human playtesters. These playtesters can test whether games are fun to play, which is impossible to test using automated tests. In our proposed method there is a place for automated testing and a place for playtesting. Automated testing is done during development to check quality and ensure a functioning product. Whereas playtesting is done at the end of the sprint, in the review phase, to test how much fun the game is, find unforeseen bugs, and discover new features. On top of this, our method deviates from the standard BDD practice by introducing prototyping. This allows to test for the fun-factor of a new feature before formally implementing it. As such, we feel that this argument holds no water in light of the proposed method.

Code volatility The second argument against automated testing is that game code is too volatile to warrant the formal specification of its features. This has partly been incorporated into the method by introducing early prototyping. This encourages developers to try out how a feature can and should work before capturing the desired behaviour in BDD specification.

However, there is a second aspect that warrants consideration when talking about code volatility. This is the evolution of features during development. New features can cause the appreciation or functioning of old features to change. This is part of the natural flow of game development.

If the workings of an existing feature change or should be changed due to a new feature being added then some work is required to update the specifications. If features interplay it is useful to add specifications of how the features work in isolation and how they interplay. This should be considered required maintenance of the documentation and test suite during development.

In considering the fun factor of new features the interplay between different features can get complicated. One feature may be no fun without another feature. Properly handling this interplay requires experience from the development team. We feel, however, that this is part of any type of game development and not specific to this method or to the usage of BDD specifications.

Time Starting with the proposed method is a big time investment. It introduces meetings, the writing and checking of scenarios, the writing of tests, and the maintenance of the scenarios. A natural question would be to ask if this time investment is worth it.

Benefits	**Drawbacks**
Less chance of misunderstanding between technical and non-technical members	Meetings to discuss the workings of every feature
Specification of expected behaviour to guide development	Writing scenarios
Executable specifications to easily check if all features still work	Writing of step definitions (tests)
Living documentation of how features work and interact	Maintenance of scenarios

TABLE 7.1: The main benefits and drawbacks of the proposed method

Like any method, the suitability of our proposed method depends on the project, the team behind it, and their priorities. If the project's priority is to get something working as quickly as possible, and the priority is not on quality, then it is unlikely that the proposed method is a good fit. However, for the sake of this evaluation, we will assume that quality and maintainability are priorities.

To make a judgement on the justifiability of the proposed method and its time investment, we will compare the overall benefits and drawbacks.

Extended benefits and drawbacks of BDD We have listed the benefits and drawbacks side by side in Table 7.1.

Whether or not the benefits of the method outweigh the drawbacks is highly dependent on the experience level of the team and the features to which the method gets applied. There is a large difference between assigning an experienced developer to make a text field scrollable or assigning a less-experienced developer to create a tutorial level.

In the first example, there is no ambiguity on how the feature should behave. In this case, there is little added benefit of organizing a meeting to discuss the expected behaviour of the feature. It is also unlikely that the experienced developer will need a guidebook on how to develop such a simple feature. It can, however, be useful to have an automatic test that checks if the behaviour remains as expected with the addition of new features. Having documentation that (some) text fields should be scrollable can also offer benefits.

On the other hand, when we imagine a less-experienced developer who is assigned a complex feature we can clearly see the added benefit of each of the steps. In this case, a discussion of the expected behaviour will help clarify the feature and decrease the chances of miscommunication. For a complex feature and/or a less-experienced developer the specifications can also provide a guidebook on what to implement. Not to mention, the added benefit of having the developer think about how features should interact before starting development. This is an invaluable step in decreasing the development time that less-experienced developers often overlook. Lastly, for a complex feature automatic tests and documentation offer a clear benefit to the quality of the final product.

In conclusion, the proposed method's benefits might not outweigh the costs for every feature. For complex features and/or inexperienced developers the benefits outweigh the initial time investment. However, for a combination of experienced developers and simple features the benefits may not outweigh the time investment. In chapter 10 we will further discuss customization options for the method to be able to minimize the drawbacks. We will also come back to this point in the application of the method to our case study in section 9.3.

Chapter 8

UnitySpec: Behaviour-Driven Development tooling for Unity3D

To support BDD in Unity a tool has been created. This tool is called UnitySpec. It is open-source and can be freely downloaded from GitHub. In this section the main functionality of UnitySpec will be introduced, more detailed documentation is available on GitHub.

In subsection 4.4.1 it has been concluded to build the tool upon the foundation of SpecFlow, which is based on Cucumber. As a result of this the resulting tool functions similarly. UnitySpec has been tested in Unity 2022.3 for both the .Net Standard 2.1 and .Net Framework API. It can be added to an existing Unity project through the Unity Package Manager.

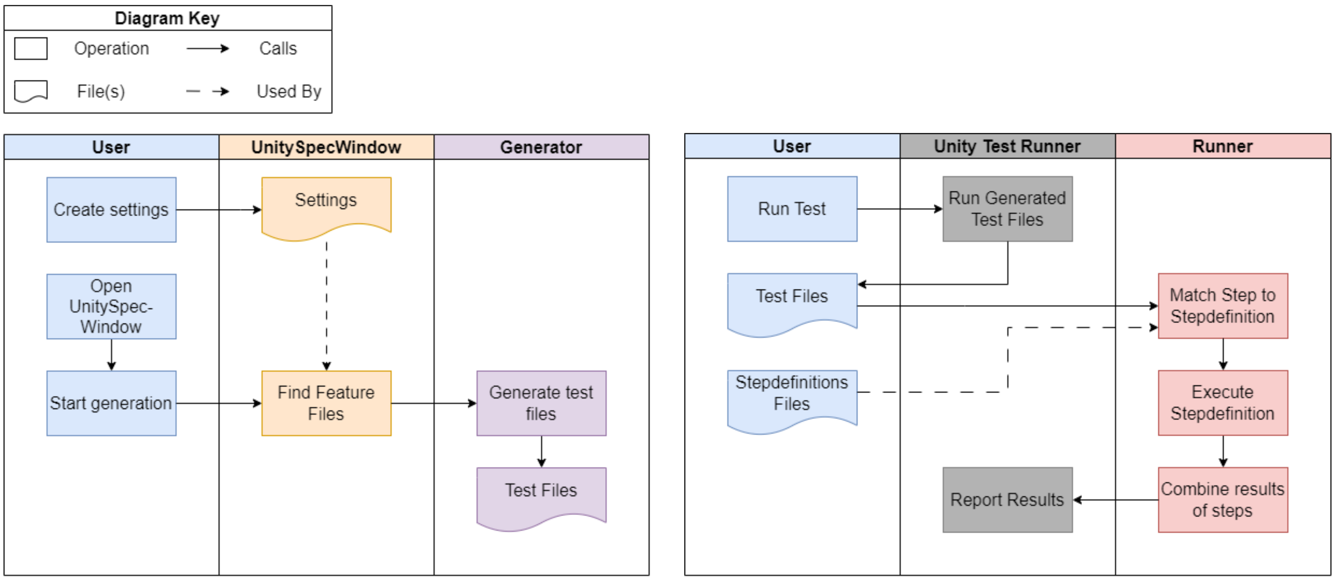
The process of using UnitySpec in Unity can be subdivided into 4 steps. For each of these steps, it is presented what functionality UnitySpec adds to Unity.

Create scenarios UnitySpec adds support for `.feature` files in Unity. This includes the ability to preview them in the inspector. It also introduces a new option in the create menu to create a new feature file. In these files the scenarios should be written using the Gherkin specification language.

Generate test files UnitySpec generates test files from the feature files, this generation gets triggered through a button in a custom editor window. In these files, a test method gets created for each of the scenarios. In these scenarios, a special test runner gets called to execute each of the steps in the scenario. Tags from the scenarios are added to the test methods.

Create step-definitions For the test files to execute tests step-definitions have to be created. A step definition defines what code should be executed for a step. UnitySpec introduces bindings to facilitate coupling between the generated test files and these definitions. Special in UnitySpec is that step definitions can have the *IEnumerator* return type, this allows for operations that need to wait for something in a scene. UnitySpec adds a setting in Unity where it can be configured in which folder the bindings are.

Run tests The generated tests are recognized by Unity's test runner. When a test is run UnitySpec outputs which steps get executed and which methods got called for each of these steps. UnitySpec also includes helpful messages in cases where suited bindings cannot be found.



(A) User flow for generating test files (B) User flow for running tests

FIGURE 8.1: User flow diagrams of UnitySpec

8.1 Development

8.1.1 Overview

UnitySpec consists of 4 blocks. Each has their own responsibilities.

The **UnitySpecWindow** includes code to include support for *.feature* files, code to define settings and code for the custom window.

The **Runner** is required to run the generated files. It gets called from the generated test files to match steps to their definitions and execute these definitions. It also generates the output seen in the test window. Currently, the runner is dependent on Unity for logging and asserting. As such, the code is included directly in the package.

The **Generator**, as the name implies, is responsible for generating code. It parses the feature files and from this, it generates test files. The generator is set up as a library. It is compiled outside of Unity and included as a dll file in the package. It can also be found on `https://github.com/UnitySpec/Generator`.

Lastly, there is some code shared between the runner and generator, we named this **General**. This code is compiled separately and completely independent of Unity. It can be found on `https://github.com/UnitySpec/General` and is included in the package as a compiled library.

Figure 8.1 shows the user flow when using UnitySpec.

Figure 8.1a shows the flow for generating test files. The user opens UnitySpecWindow and starts the generation. UnitySpecWindow checks the settings and looks for the feature files. These files get passed to the Generator, which generates the test code and writes this to files.

Figure 8.1b shows the flow for running the generated tests. The user interacts with the Unity Test Framework (UTF) to run the tests. This runs the code in the generated test files. This code calls the Runner. The Runner matches each of the steps with a step definition. It executes the step definitions and congregates the results. These results get reported back to the UTF, which displays it for the user.

8.1.2 Key choices

In subsection 4.4.1 “State of the art of BDD tooling” it was decided to build UnitySpec upon the foundation of SpecFlow. It is based on the code from the last alpha-release at the time of writing, version 3.9.74. The code from SpecFlow can be found on Github at `https://github.com/SpecFlowOSS/SpecFlow/tree/v3.9.74`.

UnitySpec is supposed to be tooling for Unity, as such it needs to work inside Unity. The way to make this work is to make UnitySpec a Unity Package. This immediately presents a first hurdle in adjusting SpecFlow. The project structure, dependencies and code needed to be changed to work in a Unity Package. Instead of trying to manage to update all files and dependencies at the same time, we decided to take the main class that starts the generation of the test files, copy this to UnitySpec and copy the classes it requires and their dependencies as we move through the workflow. We followed the same approach to import the testrunner in UnitySpec.

This method led to a rapid increase in the number of files within UnitySpec. Unity projects, including Unity packages, require each file to have an accompanying meta file, and all namespaces need assembly definition files for proper recognition and interaction with other namespaces. The resulting management overhead prompted the decision to separate the Unity interface, Generator, and Runner components.

The solution involved generating a C# library, managing its dependencies with NuGet, and incorporating the compiled library into the Unity Package. Consequently, libraries for the Generator and Runner were created, but it became apparent that they had interdependencies. To avoid circular dependencies, shared code was moved to a separate library named "General."

To enable users to trigger the test file generation, a custom Unity Window with a button was developed.

As development progressed, it was discovered that the Runner required specific Unity features, such as logging and assertions. Therefore, the Runner library was integrated back into the Unity Package. This integration became more manageable after separating the Generator and General code, as the Runner now contained only the necessary files for running tests.

8.1.3 Detailed look per package and contributions

In this section we will go into detail on what is included in each of the packages - `UnitySpec.General`, `UnitySpec.Generator`, `UnitySpec.Runner`, and `UnitySpec` - and which parts have been written by us and which parts have been copied from SpecFlow.

Before we do so we feel it is important to note that a big contribution of UnitySpec is the work of finding the needed files and organizing them into these packages. Even though we didn't contribute any code in e.g. `UnitySpec.General`, a major contribution was separating and configuring the package.

UnitySpec.General

Figure C.1 presents the classes in UnitySpec.General and how they relate. It uses the packages `BoDi`, `Gherkin`, and `SpecFlow.Internal.Json`. It mainly consists out of:

- Configuration: In SpecFlow this handles the configuration of the Generator and Runner. SpecFlow supports configuration through `specflow.json` and `app.config`.

UnitySpec currently does not offer this functionality. It does offer some configuration, but that does not use this code.
There is an open issue to refactor the code in this folder to remove artefacts from SpecFlow and make the configuration functional.

- GeneratorInterfaces: These are interfaces that are implemented in the Generator and used in Runner. This namespace also includes the files needed for the interfaces.
- Parser: This includes the internal representation of a feature file. Both the Generator and Runner use this representation.
- Extensions: Extensions on various types that make handling easier.
- Utils: Abstraction and helpers for interacting with the filesystem.
- Various classes: these are either used by one of the classes in General or happen to be used by both the Runner and Generator.

UnitySpec.Generator

Figure C.2 presents the classes in UnitySpec.Generator and how they relate. It requires `UnitySpec.General` for the files we just discussed, `Microsoft.CodeAnalysis` for Roslyn which helps generate code, `Gherkin` for parsing the Gherkin syntax, and `BoDi` a very simple IoC container which was created for SpecFlow.

The figure is very densely packed, so we will highlight the important connections. Colours have been added to the figure, this is to indicate where we have contributed lines of code. Blue classes have been adjusted and green classes have been created or completely rewritten for use in UnitySpec. We have defined completely rewritten as meaningful change is over 95% of lines of code.

UnitySpecWindow calls `RunGenerator` to start the generation (right-top of the figure). This entry point has been created for UnitySpec. `RunGenerator` uses `Project` to create an internal representation of the project, which then gets passed to `Generator`. `Generator` parses each feature file. Once completely parsed and placed in context, `Generator` calls upon `Generation` to actually generate the test classes. Generation starts with the `UnitTestFeatureGenerator` which generates the test file for a feature with the help of `UnitTestMethodGenerator`, `RoslynHelper`, and `UnitTestProvider`.

UnitySpec.Runner

Figure C.3 presents the classes in UnitySpec's Runner and how they relate. It requires `UnitySpec.General` for the previously discussed files, `Gherkin` for some last Gherkin processing, and again `BoDi` a very simple IoC container which was created for SpecFlow.

Note how the base namespace is UnitySpec and not UnitySpec.Runner. This has been chosen for a better user experience. When the user uses UnitySpec and wants to define step definitions they have to import the bindings from this package. Meaning that now they can simply import `UnitySpec`, while if we had named it `UnitySpec.Runner` they would have had to import that, possibly causing confusion.

This figure is even smaller than the previous one, so we will again highlight the important parts. This figure uses the same colour coding as the previous, meaning that blue classes have been meaningfully changed and green classes have been created for UnitySpec or changed over 95% from the original SpecFlow-code.

The classes in the base namespace have been split into two categories to keep the list manageable. The left contains all attributes and the right the other classes. These attributes are used in the step definitions to indicate that something is a step definition.

The `TestRunner` is the main entry point of the Runner. This gets called by the generated test files. `TestRunner` calls upon `TestExecutionEngine` in `Infrastructure` to execute the tests. Once the steps are matched to their definitions `BindingInvoker` in `Binding` gets called to invoke the step definitions. The actual execution happens in `SynchronousBindingDelegateInvoker`. The result of this gets passed back up the chain until it is back in the generated test. Meanwhile, there is `Errorhandling` in case anything goes wrong during this process; `BindingSkeletons` to create the skeleton code that gets returned when a binding does not exist; `Tracing` to show and report the progress; and `Events` to keep everything in sync.

8.1.4 Challenges during development

Starting from SpecFlow there were three main challenges in adjusting the code to work inside Unity.

The first problem encountered is the project structure of Unity projects. SpecFlow is written to be used on `C#` class libraries. A Unity project is an entirely different kind of project, even though both use `C#`. To resolve this problem the project discovery and file generation have been adjusted. This also guided the decision to create a new project instead of forking SpecFlow.

The second problem is that the Unity Test Framework (UTF) works differently from regular `C#` testing frameworks. The runner expects all methods to have a return type of *void* or *Task*. UTF, on the other hand, uses methods with return-type *void* or `IEnumerator`. This is mostly used to allow tests to wait for one or more frames or seconds. This means that step definitions should also be able to have this return type. To allow for this both the generator and runner had to be adjusted.

The last main problem is the .Net-version. UnitySpec has been based on SpecFlow 3.9, which supports .Net Core 2.1 and 3.1. Meanwhile, Unity has the option to use .Net Standard 2.1 and .Net Framework. In order to support both UnitySpec has to target .Net Standard 2.0. This change made it impossible to use the code-generation library from SpecFlow. This means that all code generation had to be rewritten to use Roslyn instead, which is included in .Net Standard. For this change, some features were also no longer available, like the null-coalescing operator, which required some minor rewrites.

In conclusion, UnitySpec required the creation of a way to trigger the generation and discover the needed files, reworking the runner to allow for the `IEnumerator` return type, and rewriting the generation to a different code-model-library.

8.2 Proof of concept by samples

In order to test if UnitySpec work correctly we generated two samples. The first sample is a classic BDD example: checking addition. The second sample [1] checks if a player can walk. It includes a basic scene and player controller.

8.2.1 First sample

The first sample is a classic Behavior-Driven Development (BDD) example that tests the addition operation. This sample has been created to test the basic features of UnitySpec in the simplest scenario possible. The objective is to ensure that various components of UnitySpec function correctly and interact seamlessly. The feature looks as follows:

[1]Based on [7]

```
Feature: Addition

Scenario: Add two numbers
    Given the first number is 50
    And the second number is 70
    When the two numbers are added
    Then the result should be 120
```

To begin with, the sample tests the functionality of the UnitySpecWindow component. It verifies the capability to create feature files within the Unity environment and ensures that the UnitySpec Window can be accessed through the Unity menu. A crucial part of this test is checking if the UnitySpec Window correctly reads and understands settings, specifically the location of feature files. Additionally, it confirms that the UnitySpec Window can locate a feature file within the designated folder and successfully invoke the generator.

Next, the sample tests the generator's functionalities. It ensures that the generator can parse a basic feature file without issues and can create an appropriate layout for the test file. Moreover, it verifies that the generator can generate a test method for the scenario described in the feature file. Creating a valid test file is critical, so the sample checks that the generator produces a test file that is recognized as such, compiles without errors, and functions correctly.

Lastly, the sample examines the runner's capabilities. It confirms that the runner can locate the corresponding step definitions through bindings and can execute step definitions that return void. Furthermore, the sample verifies that the runner can aggregate results from individual steps and report them accurately. It also tests whether values can be added to and retrieved from the ScenarioContext correctly.

This sample serves as a foundational check to validate the core functionalities of UnitySpec, ensuring that all essential components are working together as intended.

8.2.2 Second sample

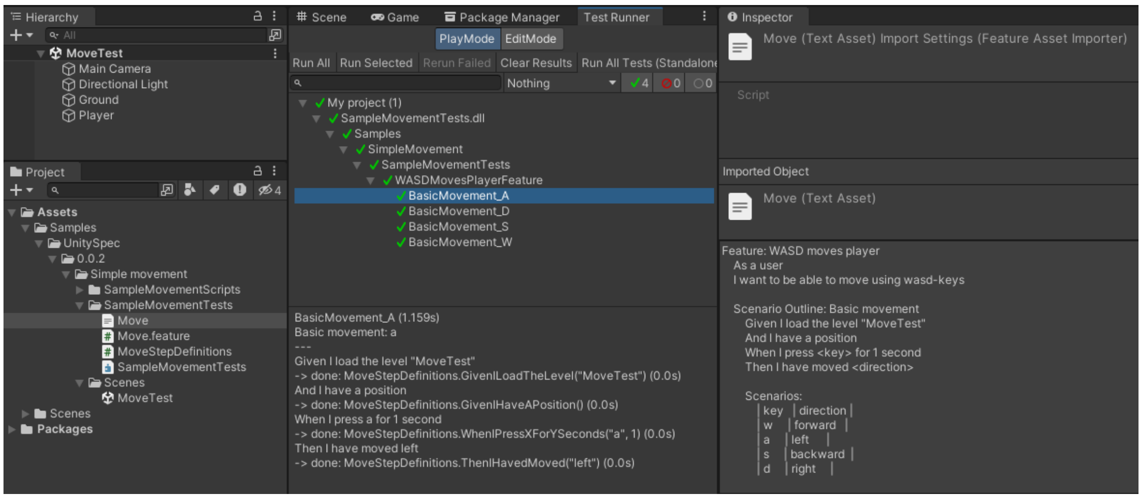
The second sample introduces additional complexities to test the robustness of UnitySpec. The two primary complexities are the usage of a scenario outline and step definitions that return an IEnumerator. These complexities specifically challenge the generator and runner, respectively. Along with these new elements, this sample again tests whether all elements from the first sample still function correctly with the added project complexity of introducing a scene and a player controller. The feature used in this sample is:

```
Feature: WASD moves player
    As a user
    I want to be able to move using wasd-keys

    Scenario Outline: Basic movement
        Given I load the level "MoveTest"
        And I have a position
        When I press <key> for 1 second
        Then I have moved <direction>

        Scenarios:
            | key   | direction |
            | w     | forward   |
            | a     | left      |
            | s       | backward  |
            | d       | right     |
```

For the generator, there is some added complexity in the parser. This feature includes a name with multiple words, a feature description, and most importantly a scenario outline. This sample tests whether the generator can properly parse all of this and generate a test file for it. This includes creating a base method and specific test methods that incorporate the different variables and call the base method.

The runner is tested for its ability to handle methods that return an IEnumerator, ensuring it can properly execute asynchronous code. It also verifies that the runner can load a scene and execute IEnumerators in the correct order. This sample tests the runner's enhanced capabilities to manage more complex step definitions and maintain accurate execution and reporting.

Overall, this second sample serves as a more rigorous test of UnitySpec, ensuring that it can handle the increased complexity of game testing.



```
BasicMovement_A (1.159s)
Basic movement: a
---
Given I load the level "MoveTest"
-> done: MoveStepDefinitions.GivenILoadTheLevel("MoveTest") (0.0s)
And I have a position
-> done: MoveStepDefinitions.GivenIHaveAPosition() (0.0s)
When I press a for 1 second
-> done: MoveStepDefinitions.WhenIPressXForYSeconds("a", 1) (0.0s)
Then I have moved left
-> done: MoveStepDefinitions.ThenIHavedMoved("left") (0.0s)
```

```
Feature: WASD moves player
  As a user
  I want to be able to move using wasd-keys

  Scenario Outline: Basic movement
    Given I load the level "MoveTest"
    And I have a position
    When I press <key> for 1 second
    Then I have moved <direction>

    Scenarios:
      | key  | direction |
      | w    | forward   |
      | a    | left      |
      | s    | backward  |
      | d    | right     |
```

FIGURE 8.2: A run of a test generated by UnitySpec

Running the second sample gives an output as in Figure 8.2. This figure shows the Unity Editor after running the movement test. This can be recreated by importing the package in a Unity project and importing the movement sample.

On the left hand, we see the different files, *Move.feature*, *MoveStepDefinitions.cs*, and the generated file *Move.feature.cs*. The contents of the Move feature can be viewed on the right-hand side. From this feature, the shown tests were generated. A test has been generated for each of the scenarios, using the first column to distinguish the test names. The test run for 'a' has been selected. In the test output, each of the executed steps is shown. Below each step, it is printed what the result of this step was, and which method got executed.

8.3 Reflection

Cross-platform support In chapter 5 we mentioned a previous attempt to create tooling for BDD in Unity, Cukunity. One of the limitations of this tool is that it only supports Android 2.X and iOS 5.x applications and can only be run from MacOSX.

UnitySpec, contrary to Cukunity, creates code that that can be run directly by Unity's own test framework UTF. Because of this, UnitySpec supports any and all platforms that are supported within UTF. UTF support all Unity's plaforms except WSA. These include, among others, Windows, Mac, Linux, iOS, Android, Oculus, PS4, PS5, Xbox One, Nintento Switch, and WebGL[2].

It should be noted that we have only tested UnitySpec on Windows with the target platform also being Windows.

Accesability for non-tech stakeholders In section 4.4 we discussed a paper by Pyshkin et al [74] which observed that most BDD toolkits primarily cater to developers. This runs counter to a base principle of BDD - to foster communication between technical and non-technical stakeholders.

UnitySpec is no different in this aspect. It has been created to support developers during development, and not with non-technical stakeholders in mind.

UnitySpec does do a few things that make the process slightly clearer for stakeholders. The first thing is that UnitySpec uses feature names to generate test class names and scenario names to generate test names. This makes it easier to see which feature is broken without knowledge of the code base.

UnitySpec also reports each step in the scenario together with its status. This means that it is easy to see - in plain English - at which step the test failed.

Industry Engagement One notable reflection on the UnitySpec project is its interaction on GitHub with a company specializing in digital health solutions. This company combines clinically validated medical tests with a gaming experience to enhance patient engagement. Their interest in UnitySpec underscores its potential applicability beyond traditional software development and into innovative fields such as digital health. The company's engagement suggests that UnitySpec can be a valuable tool in environments where user engagement and reliable test automation are critical.

[2]https://support.unity.com/hc/en-us/articles/206336795-What-platforms-are-supported-by-Unity

Chapter 9

Application

In this chapter, we will answer subquestion 4: *What are the implications of applying our findings to a real-world case study?* First, we will apply the results from subquestion 1 to answer subquestion 4.1: *How well can the proposed Game Behaviour Framework be applied to a real-world case study?* Next, we will apply the results from subquestions 2 and 3 in an experiment to answer subquestion 4.2: *What are the implications of applying BDGD and UnitySpec to a real-world case study?*

9.1 Applying the Game Behaviour Framework

In this section, we will apply the behaviour framework developed in chapter 6 "Game Behaviour Framework: Behavioural patterns in games" to a real-world case study. The case study has been introduced in section 2.3 - a serious game called the *Virtual Brigade*.

Before starting development on the *Virtual Brigade* the product owner created a roadmap with requirements for the first release. This release has been done at the start of this year, and the game is currently being developed further to add more features. To test the applicability of Game Behaviour Framework we will focus on the requirements in the roadmap. The roadmap offers properly specified and diverse requirements, allowing us to fully apply the Game Behaviour Framework.

The roadmap has been subdivided into the following 11 categories:

1. Visual worlds (3d-design)
2. Game objects
3. Player movement
4. Interactive objects
5. Events
6. User interface
7. Rule engine & game finishing
8. Data save (player)
9. After action review
10. Technology systems
11. Deployment

From these categories, we will not consider the category Deployment and Data save. Deployment exists out of hardware and build requirements. These requirements do not describe game behaviour and cannot be tested inside the game. Data save describes requirements for the save format of the game. While this is a behaviour of the software, it is not a behaviour of the game.

We go through the other 10 categories in their presented order. For each category, we will present the requirements in them. The first two categories are mostly directed at the design team, they list items and worlds that need to be created. These are not interesting from a behavioural standpoint, so we will go through these categories more quickly. For each of the presented behaviours, we will then apply the framework from chapter 6 "Game Behaviour Framework: Behavioural patterns in games".

9.1.1 Visual worlds (3d-design)

This category exists out of a list of worlds that needed to be designed for the game. In the game each world is a Unity scene.

The only behaviours that we can derive from these worlds are whether or not the scene can load and if the needed elements are present.

Both these behaviours are level 1 behaviours. Using the categories presented in chapter 6, they belong in the categories "Booting" and "Assets Exist".

One of the worlds that should exist is Amsterdam Airport Schiphol. This world can, for example, be used to train people who will man the immigration desk.

For the sake of brevity, we will limit ourselves to only showing a BDD feature to check if the Schiphol scene exists, works, and if there is an immigration desk. This can be done as follows:

```
Feature: Visual World Schiphol exists
The visual world for Schiphol exists and contains key objects.

Scenario Schiphol loads
    When the scene Schiphol_Template gets loaded
    Then the loaded scene is Schiphol_Template

Scenario: Schiphol contains an immigration desk
    Given the scene Schiphol_Template is loaded
    When the scene has finished loading
    Then there is an instance of Immigration_Desk
```

While the existence of visual worlds is not a behaviour, our developed framework allows us to express the expected result in a BDD feature.

9.1.2 Game objects

Like visual worlds, the category Game Objects is aimed toward the designers. The only behaviour we can express is whether objects are present in a scene when it starts.

An example is the immigration desk in the BDD feature from the Visual Worlds category.

9.1.3 Player movement

The category Player Movement contains descriptions of how the player should be able to move. The (translated) requirements are presented in Table 9.1. Next to each behaviour, the behaviour has been classified into a level, and where applicable a category as presented in chapter 6.

The first behaviour is simply allowing the player to walk around and is thus classified as a level 1 behaviour. The following four behaviours are all singular interactions, the player interacts with an object in the environment and something happens.

Requirement	**Level**
The player can move around the visual world.	Level 1: Player Movement
The player can enter a vehicle.	Level 2: Special Control
The player can control the systems in a vehicle. Driving is out of scope.	Level 2: Special Control
The player can enter an immigration desk and take place.	Level 2: Special Control
The player can open and close doors.	Level 2: Special Control
The player can pick up objects, place them in their "bag", and later use them.	Level 3

Table 9.1: Player Movement Requirements and their Level

Requirement	**Level**
Vehicles can drive according to a defined pattern.	Level 1/2: Independent Behaviour
Vehicles obey traffic laws.	Level 3
People can walk to a specific point.	Level 2: Independent Behaviour
People can walk according to a random pattern.	Level 2: Independent Behaviour
People can queue and walk towards the player one by one.	Level 3
RNLM dogs follow an RNLM employee.	Level 2: Independent Behaviour
The player can look at a matrix board and then set the matrix board via the interaction wheel.	Level 2: Special Control
After a certain action, a certain website can be opened in-game with an embedded browser, to simulate operational systems.	Level 2: Event-based Trigger

Table 9.2: Interactive Objects Requirements and their Level

In this category we limit the systems in a vehicle to those that can be controlled with a single interaction, like turning something on or off. In a military police vehicle, there can be more complex systems that require multiple interactions - such as looking up a licence plate number. These systems we will consider at Interactive Objects.

The last behaviour is an outlier in this category. It could be described as three separate behaviours.

1. The player can pick up objects,
2. The player can place picked-up objects into inventory,
3. The player can move items from their inventory to their hands.

Each of these behaviours on their own could be classified as a level 2 behaviour (Special Interaction). The third behaviour could be more complicated, but in this case, we believe it to mean moving items from inventory back to the hands of the player - at which point he can use them.

When we have multiple level 2 behaviours like this who together form a complex behaviour we classify this as a level 3 behaviour. We have not yet defined a fitting subcategory for this behaviour. We suggest classifying it as "Inventory".

9.1.4 Interactive Objects

The category interactive objects describes behaviours of non-players and objects that can be interacted with. The requirements have been translated and are presented in Table 9.2.

The behaviours in this category are significantly more complex than those in the previous category. We will discuss the interesting cases in more detail below.

Vehicles can drive according to a defined pattern Depending on the patterns this behaviour can either be a level 1, 2, or even 3 behaviour. If the pattern is, for example, that the car constantly drives in a circle this behaviour could be classified as a level 1 "Independent Behaviour L1". However, if the pattern is dependent on some condition, e.g. the car follows something or drives straight until there is no building next to it and then turns, then this could be a level 2 behaviour "Independent Behaviour L2".

Vehicles obey traffic laws On the surface, this feature request looks simple. However, when we try to specify it it becomes clear that it actually is very complex and vague. This feature would greatly benefit from a meeting to further discuss the details and how different parties expect this feature to work.

We imagine that what is meant by this feature is that while vehicles are driving their predefined pattern if they cross paths with another vehicle they respect the right of way. Likely, vehicles should also stop for red traffic lights and continue when they turn green. Maybe, there are also stop signs and crossroads. In the future, there might even be trams or military colones which have different traffic laws.

If we consider this behaviour as a whole we would classify it as a level 3 behaviour. However, we would recommend splitting the behaviour into multiple behaviours - each specifying one traffic law that should be obeyed. Each of these could then be a level 2 or level 3 behaviour. For example, stopping for a traffic light would be a level 2 behaviour: Event-based trigger, where the event is the light turning red or green.

People can queue and walk towards the player one by one We would suggest classifying this as a level 3 behaviour. Specifically, in the Independent category, as no player interaction is needed for the behaviour to occur. If we were to describe this feature we would likely say something like "First the person is walking according to its pattern, then it reaches the queue, it joins the queue, moving forward as the queue does, when it is in front and x condition is met, then the person walks toward the player". If a feature is easiest described in a chronologic story like this, it is a good indicator that it concerns a level 3 behaviour.

The player can look at a matrix board and then set the matrix board via the interaction wheel This is likely a level 2 behaviour. Looking at the matrix board is explicitly mentioned in this requirement, but is no different than looking at a door to be able to open it. Setting its value through the interaction wheel is a singular interaction that has an immediate effect.

After a certain action, a certain website can be opened in-game with an embedded browser, to simulate operational systems This would be a level 2 behaviour. The certain action is a trigger-event, which triggers opening a website. The technical complexity of opening an embedded browser is not relevant for the complexity of the behaviour.

9.1.5 Events

The category Events describes different events that can take place in the game. The (translated) requirements are presented in Table 9.3. We will discuss the interesting requirements.

Requirement	**Level**
There is an Event engine. This consists of placing or picking up an object, having a dialogue with a passenger or with a '3rd person'. These actions can be scheduled over time, started as soon as the player clicks on an item or person, triggered when the player walks through a certain area, or pop up randomly.	Multiple level 2
A dialogue can be conducted with a passenger who approaches. The player can respond with multiple-choice answers.	Level 3
A dialogue can be conducted with a passenger/citizen where the player takes the initiative. The player can respond with multiple-choice answers.	Level 3
A dialogue can be conducted without a passenger ('3rd person'). The player can respond with multiple-choice answers.	Level 3
A dialogue can have a different flow depending on the player's response (branching).	Level 3
There is an interaction wheel. When multiple actions are possible on a person or item, the appropriate action can be chosen using the interaction wheel. An event can be triggered depending on the choice made in the interaction wheel.	Level 3
An action in the world can be carried out through a dialogue based on the player's response.	Level 2: Event-based Trigger
The player can use their radio. When using the radio, an interaction wheel appears where they can choose what to report to the control room. An event can be triggered depending on the choice made in the interaction wheel.	Example of interaction wheel
The player can use their mobile phone. This opens a web browser that is relevant at that moment.	Level 2
The player can place caution tape.	Level 3
The player can pick up a passport and place it in the passport scanner.	Level 3
All actions have a label (for example, 'PassportForeigner1Taken') which can be defined by the developer, and a user-friendly name for the After-Action Report (AAR). They also have a desired outcome (a specific response or 'executed') which can be displayed by the developer. Additionally, a description can be provided.	Level 1

Table 9.3: Events Requirements and their Level

There is an Event engine This requirement describes different events that can happen in the game, caused by different triggers. Each of the described events is a singular interaction. We would sugggest to split this requirement into a list of level 2 behaviours, where each behaviour describes one distinct interaction and trigger.

A dialogue can be conducted with a passenger who approaches A complete conversation exists out of multiple interactions, as such it is a level 3 behaviour. The triggering of the conversation and the ability to answer can be described as a level 2 behaviour.

A dialogue can be conducted with a passenger/citizen where the player takes the initiative In isolation this would again be a level 3 behaviour. However, it shows a lot of similarities with the previous behaviour. To avoid duplication it would likely be beneficial to separately specify the dialogue and the dialogue triggers.

There should be one level 3 behaviour that specifies how the dialogue should function, and multiple level 2 behaviours that specify the different ways to trigger a dialogue.

There is an interaction wheel The existence of an interaction wheel is a low-level requirement. However, this requirement is meant to include the entire functionality of the interaction wheel. This functionality includes interacting with an object, showing the interaction wheel, interacting with the interaction wheel, and triggering some event. This includes the following level 2 behaviours:

1. Interacting with an object with multiple possible actions shows the interaction wheel;
2. Interacting with the interaction wheel triggers an event.

Capturing the whole behaviour of the interaction wheel is thus a level 3 behaviour - consisting of two level 2 behaviours.

The player can use their mobile phone. This opens a web browser that is relevant at that moment. For this requirement, we assume that what page is relevant is predefined. This behaviour thus only describes that the player can do something to use their phone which then shows the predefined web page. This means that this is a singular interaction and thus a level 2 behaviour.

The player can place caution tape While we have classified all other interactions like this as a level 2 behaviour, we feel this behaviour is different. Placing a regular item is quite simple, the player does some form of interaction and an object appears somewhere - possibly disappearing from the hands of the player. Caution tape is, however, strung between two objects. This means that when the player is holding the caution tape, he should first interact with one object - connecting the tape to that object, then move to the next object and interact with that - connecting the end of the tape to that object. As this involves two interactions and walking we feel that placing caution tape classifies as a level 3 behaviour.

The player can pick up a passport and place it in the passport scanner This requirement clearly describes two interactions: one interaction with the passport to pick it up, and one interaction with the scanner to place the passport in it. This behaviour as a whole would thus be a level 3 behaviour.

Requirement	Level
Players can easily start the game via a shortcut.	n.a.
Players do not need to log in; the game works through Single Sign-On.	Level 2
There is a visually appealing menu structure based on task fields.	n.a.
Players can exit the game by pressing the escape key.	Level 1
There is a first-time-play tutorial explaining how the game works.	Level 3
Players have access to in-game help via a link to SharePoint in the main menu.	Level 1

TABLE 9.4: User Interface Requirements and their Level

9.1.6 User interface

This category exists out of requirements for the user interface. The requirements are presented in Table 9.4.

Players can easily start the game via a shortcut. This requirement can either be a level 2 behaviour or be left entirely out of the equation. From a purely behavioural point of view, it is a singular interaction that starts the game - thus a level 2 behaviour. However, if the goal of our behaviours is to automatically test the game, then this behaviour cannot be classified. The automatic tests run within the game and can thus not test how the game gets started.

There is a first-time-play tutorial explaining how the game works. Here we assume that this requirement means to also include that the tutorial works. As such, it is a very large level 3 behaviour. This behaviour is formalized and tested in our case study, see section 9.3 for more on this.

9.1.7 Rule engine & game finishing

This category describes the expected behaviour of the rule engine and when the game is finished. The requirements are listed in Table 9.5. We did not find any special behaviours in this category.

9.1.8 After action review

The last category exists out of the requirements of the After Action Review that have not been discussed previously. These are presented in Table 9.6. Both these requirements concern a singular state without any interaction and are thus classified as level 1.

9.2 Reflection Behavioural Patterns

In this section we have taken the requirements of a real-life project, the *Virtual Brigade*, and applied the behavioural framework we designed in chapter 6. We will now evaluate the behavioural framework based on this experience.

The behavioural framework's primary strength lies in its structured approach to categorizing behaviours into three distinct levels of abstraction: Basic Controls, Singular Interactions, and Game Elements. This distinction is easily applied to the real-world requirements of the *Virtual Brigade*, allowing for a clear categorization of behaviours based on their complexity and interactivity.

Requirement	**Level**
There are events with triggers and associated desired behaviour, as well as self-initiated actions (triggerless events). Both can be tested and are referred to as 'test objects'.	Level 2: Trigger
Test objects can be prioritized, first this, then that. Test objects may or may not be optional.	Level 3
The game determines its own ending. It can be set to occur either when all mandatory actions are completed, when the set time has elapsed, or when the player reports back to the control room that they are available again.	Level 2: Trigger
At the end, the game determines whether the scenario has been 'completed' or not. A scenario can either be completed or not completed. Additionally, a percentage is displayed.	Level 1
The developer can specify actions that must always be completed correctly. If these actions are incorrect (or not executed), the player fails the scenario.	Level 1
The percentage is calculated based on the number of correctly executed actions divided by the total number of actions.	Level 1

Table 9.5: Rule Engine & Game Finishing Requirements and their Level

Requirement	**Level**
After playing the scenario, the student should be able to see which actions were taken, how they should have been done, and which ones were missed. Additionally, they see their answers to the questions, including the correct answers.	Level 1
A link to a relevant knowledge base article can be provided.	Level 1

Table 9.6: After Action Review Requirements and their Level

The case study encompassed various categories of game elements, from visual worlds and game objects to player movements, interactive objects, and user interface elements.

The framework effectively captured the essential behaviours of visual worlds and game objects, primarily focusing on the presence and loading of assets within the game scenes. These behaviours were classified at a basic level (Level 1), ensuring that fundamental requirements such as "Booting" and "Assets Exist" were adequately tested. While the existence of visual worlds and game objects does not inherently constitute behaviour, the framework's ability to express these expectations in BDD features underscores its flexibility.

For player movement and interactive objects, the framework showed its capability to handle more dynamic and complex behaviours. Player movements, ranging from basic navigation to intricate interactions with the environment, were appropriately classified into different levels. This categorization will help in structuring test scenarios that reflect real user interactions and game mechanics. Similarly, the framework managed to encapsulate the behaviours of interactive objects, differentiating between simple idle behaviours and more complex, event-driven interactions.

In the realm of events and user interface requirements, the framework demonstrated its comprehensive nature by accommodating a wide range of behaviours. From initiating dialogues to handling complex event engines and interaction wheels, the framework provided a structured approach to defining and testing these features. The categorization into different behavioural levels helped in breaking down complex requirements into manageable and testable components.

We were able to identify and classify all behaviours, highlighting the Game Behaviour Framework's robustness in handling diverse game functionalities. It has demonstrated a high degree of adaptability, allowing for the systematic classification and testing of a wide variety of game behaviours.

Key insights

One of the key insights gained from this case study is the framework's flexibility. Although many behaviours did not fit neatly into the pre-defined example categories, the framework was still adaptable enough to classify these behaviours effectively. This adaptability is crucial in real-world scenarios where game requirements can be highly variable. The ability to extend or modify categories without losing the framework's integrity underscores its practicality.

The straightforward classification process, guided by the two primary questions in the decision diagram, makes the framework accessible for both designers and developers. This ease of application is a significant advantage, especially in collaborative settings where clear communication and shared understanding of behaviours are essential.

The framework effectively manages complex behaviours by breaking them down into their interactions. For instance, the classification of the caution tape placement and dialogue systems as level 3 behaviours demonstrated the framework's capacity to handle multi-step interactions. This granularity is essential for accurately capturing and implementing intricate game mechanics.

While the framework proved effective overall, the case study revealed one main area where it could be enhanced. The example categories, while helpful, did not cover all the specific behaviours encountered in the *Virtual Brigade* project. Future iterations of the framework could benefit from a more extensive and detailed set of categories, possibly informed by a broader range of case studies.

9.3 Applying Behaviour-Driven Game Development and UnitySpec

The results of this section are based on a series of interviews. Transcriptions of these interviews are included in Appendix B.

This section aims to answer the question "*What are the implications of applying the proposed method and developed tooling to a real-world case study?*" As described in chapter 3 "Research method", we will aim to answer this question by asking the development team of the *Virtual Brigade* to use our developed method and tooling for one feature.

9.3.1 Process

First, we needed to find a feature to use for this experiment and gather willing subjects to execute it. Our method prescribes that we need a tester, developer, and product owner. We did not strictly use these roles, however, we did find three people who were able to fill their duties. As we did not have a tester available, we asked the lead developer - who also tests and checks all pull requests - to fulfil this role. We also chose not to use the product owner, but instead use a close colleague who suggested adding this feature to the product.

With the construction of our team we also automatically picked a feature. This was the feature that the developer was about to start on and was suggested by the person we chose to fulfil the role of product owner.

The feature we applied BDD to is the addition of a tutorial. In this tutorial, the basic controls of the game should be explained. It includes looking around, walking, and sprinting.

We gathered the team and explained the methodology and what was expected from them. From this, we immediately went on to the first step in our proposed method - a meeting with the three amigos to discuss what the feature should work like. During this meeting, the researcher remained present to observe and help guide the discussion. Once satisfied, we interviewed each of the team members separately.

From these interviews, we hoped to gather a few things.

1. How they felt about the method thus far,
2. If they felt they were able to express their expectation of the feature,
3. If they felt that the discussion was useful,
4. How they felt about creating the scenarios,
5. If the method had any impact on their confidence in the successful implementation of the feature,
6. What their expectation was going forward.

After these interviews, we met with the developer to go over the workings of the developed tooling, UnitySpec. We also pointed him towards the documentation and samples of UnitySpec.

Afterwards, we gave the developer some time to develop the feature. Once he indicated that he was finished we again interviewed each of the three team members. From these interviews, we aimed to gather the following information in evaluation of the method:

1. How the developer felt about the feature and its alignment with the expectations of the other team members,
2. What the effect of the method was during development,
3. How each of the team members experienced the method,
4. If a positive or negative impact was felt due to the method,

5. If they would use the method again.

Next to this we also asked the developer for an evaluation of the tool UnitySpec.

In the reflection on our proposed method, we noted that the time investment would be a big factor in determining whether the method would be picked up in real-world projects. To gauge how this team felt about that we explicitly asked about this aspect.

9.3.2 Evaluation of the BDGD method

In evaluating the results of the interviews we will first formulate answers based on the interview's responses to each of the objectives we set before the interview. Afterwards, we will draw a conclusion, highlighting the important takeaways. Where applicable we will denote the person fulfilling the role of tester as T, the developer as D, and the product owner as P.

Interview on Scenario Creation

First, we will dive into the results of the first string of interviews. These interviews are included in section B.2.

Feelings about the method after initial meeting The interviews conducted with the tester (T), developer (D), and product owner (P) reveal that the initial meeting for the new methodology started with some ambiguity. Both T and P mentioned that it was unclear at first what was expected of them, but clarity improved as the meeting progressed. This indicates a learning curve associated with the new approach.

Despite the initial confusion, all participants recognized potential benefits in the new methodology. The developer (D) noted that while the meeting required a significant time investment upfront, it would likely save time in the long run by eliminating the need for ongoing discussions and planning. The tester (T) acknowledged that the methodology could be particularly effective in larger organizations where external stakeholders generate tickets. However, they also pointed out a challenge in their context, where many tickets originate internally. The product owner (P) emphasized that the methodology brought about clear definitions of tasks, which is beneficial for organizing and executing work.

The time investment was a common concern, with T attributing it to the novelty of the methodology and D expressing hope that the initial time cost would lead to future time savings. P emphasized the benefit of increased clarity and structure, which helps in organizing and executing tasks more effectively.

Lastly, T raised a concern about the internal generation of tickets, suggesting that while the methodology might work well with external input, it poses some challenges when applied to their situation. They proposed that internal tickets could be managed through collaborative discussions among developers, or with educational teams when they requested a new feature.

Ability to express expectations of the feature In response to the question about their clarity on what the feature entails, all three interviewees indicated that they now have a clear understanding.

The tester mentioned that it is clearly documented what is expected. Initially, there was uncertainty about the format of the feature documentation, but this was clarified during the meeting.

The developer (D) confirmed that there is now more clarity on what the feature must be capable of, particularly regarding its validity. D noted that the meeting allowed for immediate alignment on the concept, which is a departure from their usual practice where each person would independently figure out implementation details. This new approach reduces the risk of having to redo work if it did not align with what the product owner had in mind.

The product owner stated that the scenarios clearly indicate how the feature should function.

Overall, the meeting successfully clarified the expectations and details of the feature for all participants.

Usefulness of discussion The meeting significantly enhanced the participants' understanding and planning of the feature. Initially, while everyone had a general idea, the specifics were unclear. The tester (T) emphasized that although they all had an overall concept of how the feature should look, the detailed workings were not yet clear. It was during the meeting that they collaboratively decided on crucial details, such as having the player look at objects, which will now be implemented. Without this discussion, these decisions would have been left to the developer, potentially leading to inconsistencies.

The developer (D) found the meeting essential in defining the specifics of the feature. He pointed out that before the meeting, there was ambiguity about the exact requirements and validity of the feature. The meeting allowed them to immediately align on the concept, which is different from their usual practice where he would have to figure out the implementation details independently. D noted that they had never thought of using objects to explain the looking mechanism and that this idea emerged only during the meeting. This collaborative approach ensured that all choices were made collectively rather than relying solely on individual interpretations.

The product owner (P) appreciated the detailed documentation process facilitated by the meeting, which reduced the likelihood of errors. They mentioned that writing down and detailing the feature clarified their expectations and provided a clear guideline for implementation. P highlighted that the brainstorming session forced them to think concretely about how the feature should work, something they had not fully considered before. This process gave P better insight into the technical realization of the feature, which might also influence how it is technically implemented. The meeting significantly enhanced the participants' understanding and planning of the feature.

Overall, the meeting successfully aligned everyone's expectations and understanding, fostering a concrete and shared vision of the feature. This alignment helped ensure that everyone had the same understanding, reducing the risk of discrepancies during implementation. The tester and product owner both indicated that the meeting provided greater technical clarity and insight into how the feature would be implemented. The collaborative discussion ensured that everyone's ideas and expectations were aligned, demonstrating the added value of the discussion.

Evaluation on creating scenarios The tester (T) and developer (D) both had positive experiences with the process of writing the documentation and scenarios, though they highlighted different aspects.

The tester (T) acknowledged that there was an initial challenge in figuring out how to phrase things correctly, especially since the researcher had a much deeper understanding of the project. This disparity in knowledge created some difficulties, as new team members needed time to catch up. However, T felt that through collaboration and guidance from the researcher, they were able to work together effectively and achieve clarity in the documentation.

The developer (D) appreciated the process of writing the documentation for a different reason. D found that by creating the scenarios, they were able to think more comprehensively about the feature. This was a departure from their usual approach of starting implementation and solving problems as they arose. Documenting the scenarios helped D anticipate potential issues and make more informed decisions about how to approach the feature's implementation, ultimately providing a clearer vision of the final outcome.

Impact on confidence The tester (T) and developer (D) both reported an increase in their confidence that the feature would turn out as envisioned due to the meeting and the process of writing documentation.

The tester (T) mentioned that they already had confidence that the feature would be successful. However, having everything documented provides a solid foundation for their testing process. This documentation serves as a reference to ensure that the feature functions as intended and looks as expected. T felt reassured that they are less likely to need to return the feature for revisions, as any discrepancies can now be identified and addressed more effectively during testing.

The developer (D) also expressed increased confidence following the meeting. While they initially believed that the feature would turn out well, the process of documenting the scenarios solidified their trust that their vision would be accurately executed. The act of writing down the scenarios helped D align their understanding with the intended outcomes, ensuring that their approach would meet the expectations set forth during the meeting.

Expectation after first meeting When asked about their expectations for the future use of the written scenarios during implementation, the tester (T) and developer (D) provided insightful feedback on how they foresee the process unfolding.

The tester (T) anticipates using the scenarios as acceptance criteria. Once the feature is implemented, T plans to use these scenarios to verify if everything works as agreed. However, T does not expect the developer to refer to the scenarios during implementation but rather to use their own to-do list. T believes the developer will likely revisit the scenarios only once they think the implementation is complete. T also raised the concern that the continued use of this methodology will depend on the time investment required to generate the documentation. While it currently takes a lot of time due to its novelty, T expects that with practice, the team will become more efficient. T suggested that this detailed process might not be necessary for simple features, such as making a text box scrollable, which can be implemented quickly. The true value of this methodology lies in more complex features, especially those assigned to interns, as it forces the team to think through and discuss the implementation in detail beforehand, reducing the need for revisions.

The developer (D) expects to use the written scenarios as a checklist during implementation. This will help them keep track of the order of steps and ensure that they are implementing the feature correctly. By using the scenarios in this way, D can systematically ensure that each part of the feature is completed as intended.

Interview after implemenation

Following the implementation of the feature, the tester (T), developer (D), and product owner (P) were interviewed to gauge their satisfaction with the outcome and the process.

These interviews and their answers are included in section B.3.

Results of development The developer (D) confirmed that the feature was successfully implemented and is working as intended. D expressed complete confidence that the feature functions correctly and attributes this confidence to the testing process. The structured approach of thinking in steps and aligning with the team throughout the process ensured that the implementation met all expectations.

The product owner (P) simply affirmed that the result matched their expectations, indicating satisfaction with the final outcome.

The tester (T) also confirmed that the feature works as expected, highlighting that it operates precisely as specified in the documentation. T expressed satisfaction that the feature meets the predefined criteria, reflecting the success of the detailed planning and documentation process.

Effect of method during implementation The interview with the developer reveals several key effects of the new method during the implementation phase. When asked if he felt he had done things differently due to the scenarios and the pre-discussion meeting, the developer mentioned that he had been more inclined to think about testing in advance. This approach aligned somewhat with test-driven development, indicating a more proactive stance towards ensuring the code's functionality from the outset.

Regarding whether they had questions about the required behaviour during implementation, the developer responded that everything was clear. Although there were multiple ways to achieve the desired outcome, he chose one and stuck with it, as long as it met the requirements. This clarity can be attributed to the structured pre-discussion and scenarios provided.

The developer also noted that without this method, the implementation might have taken longer. They would have proceeded with implementation without seeking clarifications, which could have led to more time-consuming revisions later. The upfront time investment in discussions and scenarios ultimately saved time by reducing uncertainties and rework.

During the implementation, the developer frequently referred to the scenarios, using them as a continuous guide. This practice not only kept his work on track but also helped him improve his ability to write and refine scenarios. He learned that sometimes a scenario could be better written in a different way.

Writing the scenarios also prompted the developer to think more about edge cases, something he might not have considered as thoroughly otherwise. For example, he mentioned realizing that certain actions needed to follow a specific order, such as not being able to proceed without completing a prior step.

However, converting all scenarios into tests proved challenging. The build settings of the Virtual Brigade and a lack of expertise and time prevented them from achieving this goal.

Final evaluation The developer (D) feels that the new methodology has been helpful, particularly appreciating the clarity it provides regarding the necessary steps and how to achieve them. For the feature he worked on, the methodology proved to be very useful. He believes that using this methodology actually saved time compared to the regular approach. However, he has mixed feelings about using it again. While he acknowledges that it can be cumbersome, especially since the current Virtual Brigade (VB) is not equipped for it and is already quite large, he sees significant value in using this methodology when starting a new product and would certainly opt for it in such cases.

The product owner (P) would definitely use the methodology again with a larger team. However, in the current small team with loosely defined roles and a low-threshold collaboration style, he sees less value. He believes the methodology is particularly beneficial in larger companies where direct communication is more challenging. P admits that the methodology takes more time initially, but he believes this investment is worth it as it ensures things are done correctly from the start. He views the initial time investment as valuable.

The tester (T) sees clear value in the methodology, especially for larger features. He finds that it forces the team to think ahead about what they want to achieve with a feature, which is particularly beneficial for interns. Additionally, it provides him with a clear checklist to verify if the feature meets all the pre-defined requirements. However, T mentioned that creating tests was not successful, primarily due to a lack of experienced personnel and because the Virtual Brigade is not built for it. He noted that significant time and potentially hiring someone with experience would be required to implement this in the code, which is unlikely given the current manpower constraints.

Overall, the team members see the value in the new software development methodology, especially for larger projects and new product launches. They recognize the need for initial time investments but believe these are worthwhile for ensuring accurate and efficient execution. However, current limitations within the team, such as a lack of experience and the setup of the Virtual Brigade, present challenges for a broader implementation of the methodology.

9.3.3 Evaluation of UnitySpec

During the interview with the developer, we also asked some questions to evaluate the developed tool UnitySpec.

Initially, we provided the developer (D) with an introduction to UnitySpec, along with its documentation and sample files. D confirmed that the provided information was sufficient for effectively using the tool, noting that the documentation was clear and well-organized. This clarity in the documentation facilitated a smooth onboarding process.

Throughout the evaluation, D explored all major functionalities of UnitySpec. He successfully created a feature file and matching step definitions. D proceeded to generate test classes from his feature file and executed the generated tests. Importantly, D found the process intuitive and encountered no issues. He was also able to understand the reasons behind test failures and could identify the specific code executed by the tool, which underscores UnitySpec's transparency and ease of debugging.

D expressed satisfaction with UnitySpec. He did not experience any difficulties or identify any areas needing improvement, suggesting that the tool met his expectations effectively.

In summary, the evaluation, although concise, provided valuable insights into the usability and functionality of UnitySpec. The developer's positive feedback reinforces the tool's practicality and efficiency, indicating that UnitySpec fulfils its goal and effectively supports BDD in game development.

9.4 Conclusion

The interviews conducted with the tester (T), developer (D), and product owner (P) indicate that the new software development methodology, despite some initial challenges, offers considerable benefits. All three participants acknowledged that the method introduced a learning curve, particularly during the initial meeting, which began with some ambiguity. However, as the meeting progressed, clarity improved significantly. The developer was not concerned with the upfront time investment, believing it would save time in the long run by reducing the need for ongoing discussions and rework. The product owner valued the methodology for its ability to bring clear definitions and structure to tasks, which is crucial for effective execution. The tester also saw potential, especially for larger organizations with external stakeholders, though they noted challenges in their current context where tickets are often generated internally.

A key takeaway from the interviews is the enhanced ability of the team to express and align expectations for the feature. The collaborative discussions during the meeting ensured that all participants had a clear and shared understanding of the feature requirements. This alignment reduced the risk of discrepancies during implementation and provided a solid foundation for the testing process. Both the tester and developer reported increased confidence that the feature would turn out as envisioned, attributing this to the structured approach and detailed documentation process. The developer found the scenarios particularly useful as a checklist during implementation, which helped in systematically ensuring that each part of the feature was completed as intended.

Despite recognizing the initial time cost, all participants saw value in the methodology for its potential to save time and improve accuracy in the long term. The product owner and developer expressed a willingness to use the method again, particularly for new products or in larger teams where its benefits would be more pronounced. However, the tester highlighted challenges in creating tests due to a lack of experienced personnel and technical constraints within the current setup. Overall, the team acknowledges the methodology's value in enhancing efficiency and reducing errors, suggesting that with further practice and adaptation, it could become an integral part of their development process.

Furthermore, from the interview with the developer, we conclude that UnitySpec fulfils its goal and effectively supports BDD in game development.

9.5 Reflection

In section 4.3 we presented relevant literature on the impact of Behaviour-Driven Development. In this section we we look back on the positive and negative aspects reported and investigate how our experience compares. Table 9.7 summarizes the aspects mentioned in the relevant literature.

Table 9.8 summarizes the benefits and drawbacks identified during the case study.

We will first compare the drawbacks mentioned in related work with those found in our case study.

Periera et al. noted that teams often suffer from poorly written scenarios due to limited experience in crafting them [70]. This was echoed in the interviews with the team. However, they did not feel that this should be a major consideration in determining the suitability of the method as it would be solved with experience.

Binamungu et al. remarked that BDD requires a significant shift in the approach to software development [24], similarly Nascimento et al. percieved difficulty in implementing BDD practices [65]. This matches with the comments that the *Virtual Brigade* is currently

Paper	Benefits	Drawbacks
Pereira et al. [70]	Improved communication	Poorly written scenarios due to limited experience in crafting them
	Enhanced collaboration	Difficulty in convincing customers and fellow software developers to embrace the principles of BDD
	Living documentation	
Binamungu et al. [24]	Domain-specific terms	Significant shift in approach to software development
	Improved communication	Maintanance challanges
	Executable specifications	
	Enhanced comprehensibility of code intentions	
Nascimento et al. [65]	More comprehensive understanding of features	Difficulty in implementing BDD practices
	The assurance of correct execution	
	Better alignment within the development team	
	Reduction in unexpected changes	

Table 9.7: Benefits and drawbacks of applying BDD found in related work

Benefits	Drawbacks
Time saved during development	Learning curve in writing scenarios
Enhanced ability to express and align expectations	Initial time investment
Foundation for testing process	Difficulty filling roles for internal tickets
Increased confidence	Difficulty introducing automated tests in existing project

Table 9.8: Benefits and drawbacks of applying BDD found in case study

unable to completely apply our method as it is not set up with it in mind. In our case study, the problem mainly lies in the difficulty of employing automated testing. However, they do indicate that if a project uses this method from the start, or if it is set up with automated tests already in place, that this would likely not be a consideration.

Difficulty in convincing customers and fellow software developers to embrace the principles of BDD [70] was not a consideration in our case study. Similarly, maintenance challenges [24] were not faced in our case study. However, these might arise in a longer case study that uses BDD for multiple or all features.

Our case study also identified the initial time investment as a drawback. This has not been identified as a major drawback in the surveyed literature. This contradiction is likely because the relevant literature recognizes the initial time investment as part of the process. As noted in our case study, this time is later won by streamlining development.

Now we will compare the perceived benefits.

Improved communication [24, 70], enhanced collaboration [70], more comprehensive understanding of features [65], the assurance of correct execution [65], better alignment within the development team [65], and reduction in unexpected changes [65] were all echoed during the interviews.

Furthermore, Periera et al. [70] mention living documentation as a major benefit, and Binamungu et al. [24] mention enhanced comprehensibility of code intentions. While it is clear to see how BDD causes these benefits they were not mentioned in our case study. This is likely due to the limited size of our case study. These benefits would be better presented in projects where BDD is used for multiple features and over a longer period of time.

The benefits perceived during the case study are, although worded differently, covered by the benefits presented in the surveyed literature.

In conclusion, while the wording and focus of the benefits and drawbacks differ between our case study and the related work, they do align and offer the same insight. The case study did highlight the difficulty in introducing BDD with automated testing into an existing project. This perspective was not apparent in the existing literature, as these mostly focus on projects that use BDD from the start.

Chapter 10

Reflection and Future Work

A reflection has been written for each subquestion in its corresponding chapter. In this chapter, we will reflect on the aspects that transcend one subquestion. Here, we will reflect on the concept of using BDD in game development. In this reflection, we will also suggest avenues for future work. Lastly, we will conclude this chapter with advice for a possible future case study bassed on our experience.

Maintanance challenges

In section 4.3 we reviewed relevant literature on the impact of BDD, with key points summarized in Table 9.7. One of the significant challenges highlighted by Binamungu et al. is that "respondents admitted that BDD suffers from the same kinds of maintenance challenges associated with any current form of automated testing" [24]. This section discusses these maintenance challenges in the context of game development.

Properly written BDD statements should theoretically remain stable with the introduction of new features. However, in practice, updates to these statements can be necessary to ensure that they remain accurate and satisfactory. As new features are added and old ones modified or removed, the corresponding feature files may need to be updated.

For instance, consider a game where a new feature like enhanced movement capabilities is introduced. This could potentially disrupt existing behaviour definitions related to movement. To manage this, developers need to revisit and update the relevant feature files, ensuring that the new behaviour is accurately described and that existing scenarios are adjusted to maintain consistency.

This need for updates is an inherent part of the maintenance process, keeping the documentation — in the form of feature files — current. Effective maintenance, therefore, involves revising these feature files to reflect changes in the game's functionality, ensuring that the documentation is always aligned with the actual behaviour of the game. This process ensures that automated tests remain relevant and that the game's behaviour is accurately captured and tested.

Representativeness of case study

Our case study focuses on a serious game developed for the Royal Netherlands Marreschaussee, which introduces specific characteristics that distinguish it from traditional entertainment games. In this section, we will highlight some of the biggest differences and evaluate on how our results translate to the realm of entertainment games.

One significant difference is the stakeholders involved. In entertainment game development, stakeholders are usually limited to game designers, developers, and players. In contrast, serious games for government purposes must align with regulatory requirements, educational objectives, and policy goals, requiring input from a broader and more varied group of stakeholders.

Moreover, serious games require more deterministic behaviour compared to entertainment games. While entertainment games often thrive on unpredictability and player-driven narratives, serious games must deliver consistent and reliable experiences to achieve their educational or training objectives. This deterministic nature ensures that users can reliably practice and learn specific skills or concepts. For challenges in this domain, we will continue in the next section.

An interesting avenue of future work would be to test whether the results from this case study translate directly to entertainment games. In this future work, one or more case studies could be undertaken that apply the results of this research to an entertainment game.

Randomness in games

Games often incorporate a degree of randomness to enhance replayability and provide a dynamic user experience. This randomness poses a unique challenge when using Behavior-Driven Development (BDD), as specifying the exact expected behaviour of random elements can be problematic. In traditional BDD, scenarios are written to define specific outcomes, but randomness makes it difficult to predict a single, definitive outcome that will always hold true. For instance, in a game where a dice roll determines a player's next move, the behaviour cannot be precisely predicted each time the dice is rolled.

To apply BDD effectively to every behaviour in games, including those with inherent randomness, it is crucial to develop a method to account for this unpredictability. One approach is to focus on specifying the range of acceptable behaviours rather than a single expected outcome. For example, instead of expecting a dice roll to result in a specific number, the scenario could define that the roll should result in a value between 1 and 6. Additionally, leveraging probabilistic assertions, where tests confirm that outcomes fall within expected distributions over multiple iterations, can help manage randomness in BDD scenarios.

Further research into how randomness is handled in automatic software testing can provide valuable insights for integrating it into BDD for games. Techniques such as using seed values for pseudo-random number generators to reproduce test conditions, or employing statistical methods to validate random behaviours, could be adapted for BDD. By incorporating these strategies, BDD can be extended to account for randomness in game behaviours, ensuring that even dynamic and unpredictable elements are thoroughly tested and verified. This approach not only maintains the integrity of BDD but also enhances its applicability to the complex and varied nature of game development.

Addressing State Space Explosion

In game development, the vast number of possible states a game can be in presents a significant challenge for Behavior-Driven Development (BDD). Games often allow players a high degree of freedom, enabling them to move anywhere within the game world, interact with numerous objects, and trigger a multitude of events. This freedom creates a combinatorial explosion of potential states, making it impractical to define scenarios for every possible situation. To manage this complexity, abstraction and a deep understanding of the codebase are crucial.

Abstraction plays a vital role in BDD by allowing developers to generalize scenarios without needing to account for every possible player location or object position. For instance, instead of writing separate scenarios for each specific location a player might stand, scenarios can be designed to test general behaviours or outcomes regardless of exact positioning. This approach reduces the number of scenarios needed, focusing on core functionalities and interactions rather than exhaustive state coverage. By abstracting the key behaviours, developers can ensure that critical aspects of the game are tested without becoming overwhelmed by the sheer number of possible states.

Moreover, a thorough knowledge of the codebase enables the design of effective and efficient tests. One effective technique is boundary testing, where tests are designed to focus on the edges of possible state ranges rather than every individual state. For example, instead of testing every single position a player might occupy, tests can be concentrated on boundary conditions such as the edges of the game world or extreme positions where bugs are more likely to occur. This method ensures that the most critical and error-prone areas are covered, providing robust test coverage without necessitating an impractical number of scenarios.

By combining abstraction with strategic testing techniques like boundary testing, BDD can be effectively applied to game development despite the challenge of state space explosion. These approaches allow developers to maintain manageable and meaningful test suites, ensuring that key game behaviors are thoroughly validated while avoiding the impracticality of testing every possible state.

Balancing Execution Time and Test Coverage

Balancing the execution time of tests with the coverage they provide is a significant challenge in games. Each test typically starts a new scene, and as the size and complexity of the game increase, the time required to initiate these scenes grows. Consequently, if BDD is applied extensively throughout the entire game, the number of scenarios will expand rapidly, making it impractical to run all scenarios after every code change due to the excessive time it would take.

Masella and Baker [18, 60] identified this issue in their work on game development. They proposed a solution that differentiates between actor tests and play mode tests to manage this problem effectively. Actor tests are designed to test specific pieces of code without starting the entire game, making them much faster to execute. These tests focus on the functionality of individual components or systems in isolation. On the other hand, play mode tests start the game and test how various components interact within a running game environment. While playmode tests provide a more comprehensive assessment of the game's behaviour, they are significantly more time-consuming due to the need to initialize the entire game scene.

Incorporating this differentiation into the BDD process for games can greatly enhance the efficiency and practicality of testing. Actor tests can be run frequently and rapidly to catch issues at the code level, ensuring that individual functionalities work correctly. Playmode tests can be reserved for less frequent execution, such as during nightly builds or specific integration testing phases, to verify that all components work together as expected in the full game environment. This approach allows for a balanced test suite that maximizes coverage while managing execution time effectively.

Adopting the distinction between actor tests and play mode tests would significantly improve the methodology presented in this thesis. It provides a way to maintain comprehensive test coverage without overwhelming resources, ensuring that game development remains agile and responsive to changes. This balance between detailed code-level testing and broader integration testing helps maintain high-quality game development practices without sacrificing efficiency.

Customization of the method

In section 7.5 "Time" we evaluated the benefits and drawbacks of integrating Behaviour-Driven Development (BDD) into the game development process. Our findings indicated that the advantages of BDD are variable, heavily influenced by the complexity of the feature and the developer's level of experience. Specifically, our case study highlighted that for small, straightforward features, the overhead associated with applying BDD might not be justified.

To address this variability, future work could focus on developing a customizable, hybrid version of the BDD method. This hybrid approach would introduce a flexible decision-making process for when and how to apply BDD, ensuring that the method's benefits are maximized while minimizing unnecessary overhead.

The core of this hybrid approach would likely involve incorporating an additional step during the ticket assignment phase of the development process. In this step, an experienced developer or project manager evaluates the feature in question and determines the suitability of applying BDD. We identified two variables that are likely to offer insight into the effectiveness of BDD on a feature.

The first variable to consider would be the complexity of the feature. For complex features involving intricate interactions and behaviours, BDD can provide significant value by offering clear, testable requirements and fostering collaboration among team members. Conversely, for simple features, the overhead of writing and maintaining BDD scenarios might outweigh the benefits.

Another variable to consider is the experience level of the developer. Less experienced developers might benefit more from the structured guidance provided by BDD, whereas seasoned developers might efficiently implement simple features without the additional framework.

This hybrid method could provide a tailored application of BDD, ensuring that its advantages are leveraged where most beneficial, while minimizing unnecessary overhead for simpler tasks. Future work in this direction could to enhance the overall efficiency and effectiveness of the development process.

Introducing AI

In the evolving landscape of software development, the integration of Artificial Intelligence (AI) offers promising advancements. In the evaluation of the state of the art of BDD tooling in subsection 4.4.1, one of the examined tools is TestRigor. TestRigor utilizes Natural Language Processing (NLP) to generate tests from natural language feature descriptions.

However, TestRigor is focused on simple web pages. To extend this approach to the more complex domain of game development would be no easy feat.

Future research could take steps to apply this approach to increasingly more difficult domains. This would include enhancing the AI algorithms to better understand the context and semantics of feature descriptions. Additionally, exploring AI's role in other aspects of BDD, such as automating the maintenance and updating of feature files, could further streamline the development process.

10.1 Advice for next case study

Based on our experience, we recommend to keep the following things in mind for future case studies.

Integrate the Approach Early and/or Ensure Automated Testing Compatibility We would advise to adopt the approach from the very beginning of the project. This ensures that team members grow comfortable in their roles as the project is growing. This also ensures that the project gets built with automated testing in mind, thus ensuring that it is possible to automate the BDD scenarios.

If it is not possible to adopt the approach from the start of the project, we would advise to ensure that it is possible to run automated tests on the project. This will prevent the need for major rewrites and ensure a smooth implementation process.

Invest Time in Comprehensive Onboarding At the start of the project, allocate sufficient time to thoroughly explain the method and the associated frameworks to all team members. This step is essential to avoid any ambiguity and ensure everyone has a clear understanding of the expectations and processes.

This clarity will reduce confusion, and ensure that the team can effectively start implementing the method throughout the project.

Chapter 11

Conclusion

At the start of this thesis, we asked **How can the principles of Behaviour-Driven Development be applied in game development in Unity 3D?** To help answer this question we asked 4 subquestions. We will answer these before coming back to the main question.

Question 1: How can BDD scenarios be used to describe game behaviour? To answer this question we created the Game Behaviour Framework. This framework categorizes game behaviours into three levels of complexity: Basic Behaviours, Singular Interactions, and Game Elements. By breaking down game behaviours into these distinct levels, BDD scenarios in this framework can address both simple actions and complex interactions. Simple behaviours can be tested independently before integrating them into more complex scenarios, ensuring modularity and clarity.

Question 2: What method could be employed to apply BDD in the development process of a game? In this thesis, we created a Behaviour-Driven Game Development (BDGD) method to apply BDD in the development process of a game. This method consists of several key phases and steps designed to enhance collaboration, maintain a clear focus on behaviour, and allow for iterative development.

The method is based on the method for Test-Driven Development (TDD) and Scrum. Incorporating BDD into this framework involves replacing traditional tests with behaviour scenarios. The process follows three main steps: discovery, formulation, and automation. This process emphasizes the collaboration of the "Three Amigos"—the product owner, tester, and developer—to ensure diverse perspectives are considered.

Given the experimental nature of game development, BDGD also incorporates prototyping to allow quick testing of new ideas without the constraints of full implementation. Prototyping helps balance artistic freedom of game creation and structured development, ensuring that innovative gameplay ideas can be explored effectively.

Furthermore, exploratory testing is integrated into the sprint review phase to identify bugs and new ideas, ensuring a high-quality product. The method concludes with a closure phase once all backlog items are implemented, signalling that the game is ready for release.

In conclusion, BDGD is a method to apply BDD in the development process of a game. It is structured around collaboration, iterative development, and flexible prototyping. It ensures that game development remains both creative and disciplined, aiming to produce well-documented, tested, and high-quality games.

Question 3: What tooling can be developed to support BDD in Unity 3D game development? In answering this question we have created UnitySpec, an open-source tool designed specifically to integrate Behavior-Driven Development (BDD) practices within the Unity 3D game development environment.

UnitySpec supports the creation, generation, and execution of Gherkin-based feature files within Unity projects. By doing so UnitySpec enables developers to define and automate tests in a manner that aligns closely with BDD principles.

The key functionalities provided by UnitySpec include:

1. **Creating Scenarios:** UnitySpec integrates with the Unity editor to support the creation and inspection of `.feature` files, facilitating the definition of scenarios using the Gherkin specification language.
2. **Generating Test Files:** UnitySpec generates corresponding test files from these feature files. This process is initiated through a custom editor window within Unity, simplifying the workflow for developers.
3. **Creating Step Definitions:** To execute the generated tests, step definitions must be created. UnitySpec provides bindings that link these definitions with the generated test files. It also uniquely supports step definitions with the `IEnumerator` return type, essential for handling operations requiring frame-based waiting in Unity scenes.
4. **Running Tests:** The generated tests are executed using Unity's test runner. UnitySpec provides detailed output, indicating which steps were executed and which methods were called, along with helpful messages when bindings are not found.

With these functionalities, UnitySpec enables developers to apply BDD in Unity. Each step in the BDD process is enabled or supported by UnitySpec.

Question 4.1: How well can the proposed Game Behaviour Framework be applied to a real-world case study? The application of the Game Behaviour Framework to the *Virtual Brigade* project has validated its utility and effectiveness in real-world scenarios. Its structured yet flexible approach to classifying game behaviours has proven to be a robust tool. While there is room for refinement, particularly in the area of category specificity, the framework's core strengths make it a valuable asset in the field of game development.

Question 4.2: What are the implications of applying BDGD and UnitySpec to a real-world case study? The application of Behaviour-Driven Game Development (BDGD) and UnitySpec to a real-world case study reveals several implications. Despite an initial learning curve and some challenges, the methodology brings significant benefits, including enhanced clarity, structure, and alignment of feature requirements among team members. This leads to reduced discrepancies during implementation and a stronger foundation for the testing process. The structured approach and detailed documentation foster confidence in achieving envisioned outcomes and aid in systematic feature completion. While initial time investment is required, long-term gains in efficiency and accuracy are foreseen. Both the product owner and developer are inclined to continue using the method, especially for larger teams or new projects. The tester notes some challenges due to technical constraints and lack of experience, but acknowledges the potential for the methodology to improve with practice. Overall, BDGD and UnitySpec contribute to more efficient and error-free development processes, supporting their adoption in game development.

How can the principles of Behaviour-Driven Development be applied in game development in Unity 3D? To apply the principles of Behaviour-Driven Development (BDD) in game development using Unity 3D, the key is to integrate the structured, collaborative, and iterative nature of BDD into the unique aspects of game creation.

The Game Behaviour Framework (GBF) breaks down game behaviours into basic behaviours, singular interactions, and game elements, facilitating the creation of clear and modular BDD scenarios that address both simple and complex interactions.

In practice, Behaviour-Driven Game Development (BDGD) can be implemented, providing guidance for applying BDD in game development. The iterative approach allows for prototyping, ensuring that creative ideas can be tested quickly without full implementation, thus balancing artistic freedom and structured development. Exploratory testing during sprint reviews ensures a high-quality product by identifying bugs and new ideas.

Supporting this approach, UnitySpec is an open-source tool that integrates BDD practices within Unity 3D. UnitySpec allows developers to define, generate, and execute Gherkin-based feature files directly within Unity. This tool supports the creation of scenarios, generation of test files, creation of step definitions, and execution of tests using Unity's test runner, providing detailed output to guide the development process.

When applied to real-world projects like the *Virtual Brigade*, these methods and tools demonstrate their effectiveness in enhancing clarity, structure, and alignment of feature requirements among team members, reducing discrepancies during implementation, and fostering a systematic and error-free development process.

While there is an initial learning curve, the long-term benefits of efficiency, accuracy, and robust documentation may make BDD the next big thing in game development.

Bibliography

[1] BDD Testing & Collaboration Tools for Teams | Cucumber. URL: https://cucumber.io/.

[2] Behaviour-Driven Development - Cucumber Documentation. URL: https://cucumber.io/docs/bdd/.

[3] Concordion. URL: https://concordion.org/index.html.

[4] EasyB. URL: https://code.google.com/archive/p/easyb/.

[5] FitNesse. URL: https://fitnesse.org/.

[6] Guage. URL: https://gauge.org/index.html.

[7] How to run automated tests for your games with the Unity Test Framework. URL: https://unity.com/how-to/automated-tests-unity-test-framework.

[8] jBehave. URL: https://jbehave.org/.

[9] jDave. URL: https://github.com/jdave/JDave.

[10] Squish. URL: https://www.qt.io/product/quality-assurance/squish.

[11] TestLeft. URL: https://smartbear.com/product/testleft/overview/.

[12] TestRigor. URL: https://testrigor.com/.

[13] Tricentis qTest. URL: https://www.tricentis.com/software-testing-tool-trial-demo/qtest-trial?utm_source=google&utm_medium=paidsearch&utm_campaign=qTest_Search_Brand_High_EMEA_EN&utm_term=qtest&gad_source=1&gclid=CjwKCAjw88yxBhBWEiwA7cm6pfKb1XIoh9FU3ronJQIER-7aBg1VJoG9woh5i0nqEpDr5qzwbCPfnBoCyfAQAvD_BwE.

[14] Jehad Al Dallal. Automation of object-oriented framework application testing. pages 1–5, 03 2009. doi:10.1109/IEEEGCC.2009.5734312.

[15] Samar Al-Saqqa, Samer Sawalha, and Hiba Abdelnabi. Agile Software Development: Methodologies and Trends. *International Journal of Interactive Mobile Technologies*, 14:246–270, 2020. doi:10.3991/IJIM.V14I11.13269.

[16] Mohammad Alshraideh. A Complete Automation of Unit Testing for JavaScript Programs. *Journal of Computer Science*, 4, 12 2008. doi:10.3844/jcssp.2008.1012.1019.

[17] Rizal Broer Bahaweres, Elda Oktaviani, Luh Kesuma Wardhani, Irman Hermadi, Arif Imam Suroso, Indra Permana Solihin, and Yandra Arkeman. Behavior-driven development (BDD) Cucumber Katalon for Automation GUI testing case CURA and Swag Labs. *Proceedings - 2nd International Conference on Informatics, Multimedia, Cyber, and Information System, ICIMCIS 2020*, pages 87–92, 11 2020. doi:10.1109/ICIMCIS51567.2020.9354325.

[18] Jessica Baker. Automated Testing at Scale in Sea of Thieves, 2019. Unreal Fest Europe 2019. URL: https://www.unrealengine.com/en-US/events/unreal-fest-europe-2019/automated-testing-at-scale-in-sea-of-thieves.

[19] Werner Ballhaus, Wilson Chow, and Emmanuelle Rivet. Perspectives from the Global Entertainment & Media Outlook 2022–2026: Fault lines and fractures: Innovation and growth in a new competitive landscape. Technical report, PwC, 2022.

[20] Stefan Berner, Roland Weber, and Rudolf Keller. Observations and lessons learned from automated testing. pages 571– 579, 06 2005. doi:10.1109/ICSE.2005.1553603.

[21] Erik Bethke. *Game development and production*. Wordware Publishing, Inc., 2003.

[22] Oleksandr Bezsmertnyi, Nataliia Golian, Vira Golian, and Iryna Afanasieva. Behavior Driven Development Approach in the Modern Quality Control Process. *2020 IEEE International Conference on Problems of Infocommunications Science and Technology, PIC S and T 2020 - Proceedings*, pages 217–220, 10 2021. doi:10.1109/PICST51311.2020.9467891.

[23] Leonard Peter Binamungu, Suzanne M Embury, and Nikolaos Konstantinou. Maintaining behaviour driven development specifications: Challenges and opportunities. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 175–184. IEEE, 2018.

[24] Leonard Peter Binamungu, Suzanne M. Embury, and Nikolaos Konstantinou. Maintaining behaviour driven development specifications: Challenges and opportunities. *25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings*, 2018-March:175–184, 4 2018. doi:10.1109/SANER.2018.8330207.

[25] Leonard Peter Binamungu, Suzanne M. Embury, and Nikolaos Konstantinou. Characterising the Quality of Behaviour Driven Development Specifications. In *Agile Processes in Software Engineering and Extreme Programming*, pages 87–102. Springer International Publishing, 2020.

[26] Leonard Peter Binamungu, Suzanne M. Embury, and Nikolaos Konstantinou. Characterising the Quality of Behaviour Driven Development Specifications. *Lecture Notes in Business Information Processing*, 383 LNBIP:87–102, 2020. doi:10.1007/978-3-030-49392-9_6.

[27] Leonard Peter Binamungu and Salome Maro. Behaviour driven development: A systematic mapping study. *Journal of Systems and Software*, 203:111749, 9 2023. doi:10.1016/J.JSS.2023.111749.

[28] Jacob Burnim and Koushik Sen. Heuristics for Scalable Dynamic Test Generation. pages 443–446, 09 2008. doi:10.1109/ASE.2008.69.

[29] H Bünder and H Kuchen. A model-driven approach for behavior-driven GUI testing. *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, 2019•dl.acm.org*, Part F147772:1742–1751, 2019. doi:10.1145/3297280.3297450.

[30] Hendrik Bünder and Herbert Kuchen. Towards behavior-driven graphical user interface testing. *ACM SIGAPP Applied Computing Review*, 19:5–17, 8 2019. doi:10.1145/3357385.3357386.

[31] Fransesco Carucci. AAA Automated Testing... For AAA Games. Crytek, 2009.

[32] Jorge Chueca, Javier Verón, Jaime Font, Francisca Pérez, and Carlos Cetina. The consolidation of game software engineering: A systematic literature review of software engineering for industry-scale computer games. *Information and Software Technology*, 165:107330, 1 2024. doi:10.1016/J.INFSOF.2023.107330.

[33] Roberta Coelho, Elder Cirilo, Uira Kulesza, Arndt von Staa, Awais Rashid, and Carlos Lucena. JAT: A Test Automation Framework for Multi-Agent Systems. In *2007 IEEE International Conference on Software Maintenance*, pages 425–434, 2007. doi:10.1109/ICSM.2007.4362655.

[34] Conceptartempire.com. What is Unity 3D & What is it Used For? URL: https://conceptartempire.com/what-is-unity/.

[35] Ariel Coppes. How I use Test Driven Development to make games, 2023. URL: https://arielcoppes.dev/2023/10/29/tdd-to-make-games.html.

[36] Cucumber. URL: https://cucumber.io/docs/bdd/better-gherkin/.

[37] Maya Daneva. How practitioners approach gameplay requirements? An exploration into the context of massive multiplayer online role-playing games. *2014 IEEE 22nd International Requirements Engineering Conference, RE 2014 - Proceedings*, pages 3–12, 9 2014. doi:10.1109/RE.2014.6912242.

[38] Ashley Davis and Adam SIngle. Testing for Game Development, 2016. URL: https://www.gamedeveloper.com/programming/testing-for-game-development#close-modal.

[39] Unity3D Docs. About Unity Test Framework | 1.1.13. URL: https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/index.html.

[40] Lydie du Bousquet and Nicolas Zuanon. An Overview of Lutess: A Specification-based Tool for Testing Synchronous Software. pages 208–215, 01 1999. doi:10.1109/ASE.1999.802255.

[41] Muhammad Shoaib Farooq, Uzma Omer, Amna Ramzan, Mansoor Ahmad Rasheed, and Zabihullah Atal. Behavior Driven Development: A Systematic Literature Review. *IEEE Access*, 2023.

[42] Mariusz Fecko and Christopher Lott. Lessons learned from automating tests for an operations support system. *Softw., Pract. Exper.*, 32:1485–1506, 12 2002. doi:10.1002/spe.491.

[43] Mário Freitas. Cukunity : Cucumber for Unity. URL: https://github.com/imkira/cukunity.

[44] Omaid Ghayyur. *Mutation Testing for Unity 3D*. PhD thesis, CAPITAL UNIVERSITY, 2018.

[45] Mario Gleichmann. Behaviour driven development with beanSpec, Nov 2007. URL: https://gleichmann.wordpress.com/2007/11/20/behaviour-driven-development-with-beanspec/.

[46] Dorothy Graham, Mark Fewster, and Addison Wesley. Software test automation: effective use of test execution tools. 1999. URL: https://api.semanticscholar.org/CorpusID:60799996.

[47] Helplama.com. Game Industry Usage and Revenue Statistics 2023, 2023. URL: https://helplama.com/game-industry-usage-revenue-statistics/.

[48] Johan Hoberg. Differences between Software Testing and Game Testing, 2014. URL: https://www.gamedeveloper.com/programming/differences-between-software-testing-and-game-testing.

[49] Steffan Hooper. *Automated Testing and Validation of Computer Graphics Implementations for Cross-platform Game Development*. PhD thesis, Auckland University of Technology, 2017.

[50] Juegostudio.com. Unity 3D: A Comprehensive Guide to Unity's Features and Uses. URL: https://www.juegostudio.com/blog/what-is-unity-3d-a-comprehensive-guide-to-unitys-features-and-uses.

[51] Asmo Jurvanen. Automated testing with Unity. *Theseus*, 2023.

[52] Katja Karhu, Tiina Repo, Ossi Taipale, and Kari Smolander. Empirical Observations on Software Testing Automation. pages 201 – 209, 05 2009. doi:10.1109/ICST.2009.16.

[53] Jussi Kasurinen, Andrey Maglyas, and Kari Smolander. Is Requirements Engineering Useless in Game Development? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8396 LNCS:1–16, 2014. URL: https://link.springer.com/chapter/10.1007/978-3-319-05843-6_1, doi:10.1007/978-3-319-05843-6_1.

[54] Jussi Kasurinen and Kari Smolander. What do game developers test in their products? *International Symposium on Empirical Software Engineering and Measurement*, 9 2014. URL: https://dl.acm.org/doi/10.1145/2652524.2652525, doi:10.1145/2652524.2652525.

[55] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling Manual and Automated Testing: The AutoTest Experience. page 261, 01 2007. doi:10.1109/HICSS.2007.462.

[56] Rakesh Kumar Lenka, Srikant Kumar, and Sunakshi Mamgain. Behavior Driven Development: Tools and Challenges. *Proceedings - IEEE 2018 International Conference on Advances in Computing, Communication Control and Networking, ICACCCN 2018*, pages 1032–1037, 10 2018. doi:10.1109/ICACCCN.2018.8748595.

[57] Chris Lewis and Jim Whitehead. The whats and the whys of games and software engineering. In *Proceedings of the 1st international workshop on games and software engineering*, pages 1–4, 2011.

[58] Mehdi Malekzadeh and Raja Ainon. An automatic test case generator for testing safety-critical software systems. 1, 02 2010. doi:10.1109/ICCAE.2010.5451975.

[59] Koninklijke Marechaussee. Als het erop aankomt, 2022.

[60] Robert Masella. Automated Testing of Gameplay Features in 'Sea of Thieves', 2019. Game Developers Conference. URL: https://www.gdcvault.com/play/1026366/Automated-Testing-of-Gameplay-Features.

[61] Pejman Mirza-Babaei, Naeem Moosajee, and Brandon Drenikow. Playtesting for Indie Studios. *Proceedings of the 20th International Academic Mindtrek Conference*, 2016. URL: http://dx.doi.org/10.1145/2994310.2994364, doi:10.1145/2994310.2994364.

[62] Pejman Mirza-Babaei, Samantha Stahlke, Günter Wallner, and Atiya Nova. A Postmortem on Playtesting: Exploring the Impact of Playtesting on the Critical Reception of Video Games. *Conference on Human Factors in Computing Systems - Proceedings*, 4 2020. URL: https://dl.acm.org/doi/10.1145/3313831.3376831, doi:10.1145/3313831.3376831.

[63] Abhishek Mishra. Introduction to behavior-driven development. *iOS Code Testing: Test-Driven Development and Behavior-Driven Development with Swift*, pages 317–327, 2017.

[64] Emerson Murphy-Hill, Thomas Zimmermann, and Nachiappan Nagappan. Cowboys, Ankle Sprains, and Keepers of Quality: How Is Video Game Development Different from Software Development? In *Proceedings of the 36th International Conference on Software Engineering*, 2014. doi:10.1145/2568225.2568226.

[65] Nicolas Nascimento, Alan R. Santos, Afonso Sales, and Rafael Chanin. Behavior-Driven Development: A case study on its impacts on agile development teams. *Proceedings - 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW 2020*, 20:109–116, 6 2020. doi:10.1145/3387940.3391480.

[66] Dan North. Introducing BDD. *Better Software Magazine*, 2006.

[67] Gabriel Oliveira and Sabrina Marczak. On the empirical evaluation of BDD scenarios quality: Preliminary findings of an empirical study. *Proceedings - 2017 IEEE 25th International Requirements Engineering Conference Workshops, REW 2017*, pages 299–302, 9 2017. doi:10.1109/REW.2017.62.

[68] Gabriel Oliveira and Sabrina Marczak. On the understanding of BDD scenarios' quality: Preliminary practitioners' opinions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10753 LNCS:290–296, 2018. doi:10.1007/978-3-319-77243-1_18.

[69] Gabriel Oliveira, Sabrina Marczak, and Cassiano Moralles. How to evaluate BDD scenarios' quality? *ACM International Conference Proceeding Series*, pages 481–490, 9 2019. doi:10.1145/3350768.3351301.

[70] Lauriane Pereira, Helen Sharp, Cleidson De Souza, Gabriel Oliveira, Sabrina Marczak, and Ricardo Bastos. Behavior-driven development benefits and challenges: Reports from an industrial study. *ACM International Conference Proceeding Series*, Part F147763, 2018. URL: https://dl.acm.org/doi/10.1145/3234152.3234167, doi:10.1145/3234152.3234167.

[71] Cristiano Politowski, Yann-Gaël Guéhéneuc, and Fabio Petrillo. Towards automated video game testing: still a long way to go. In *Proceedings of the 6th International ICSE Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation*, pages 37–43, 2022.

[72] Cristiano Politowski, Fabio Petrillo, and Yann-Gaël Guéhéneuc. A survey of video game testing. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 90–99. IEEE, 2021.

[73] Brian Provinciano. Automated Testing and Instant Replays in Retro City Rampage. Game Developers Conference, 2015. URL: https://www.gdcvault.com/play/1021825/Automated-Testing-and-Instant-Replays.

[74] Evgeny Pyshkin, Maxim Mozgovoy, and Mikhail Glukhikh. On Requirements for Acceptance Testing Automation Tools in Behavior Driven Software Development. *Proceedings of the 8th Software Engineering Conference in Russia (CEE-SECR)*, 2012.

[75] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. *2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings*, pages 36–42, 2012. doi:10.1109/IWAST.2012.6228988.

[76] Mattia Riola. Test automation in video game development: Literature review and Sound testing implementation. *POLITECNICO DI TORINO*, 2023.

[77] F. Saglietti and F. Pinte. Automated unit and integration testing for component-based software systems. *ACM International Conference Proceeding Series*, 09 2010. doi:10.1145/1868433.1868440.

[78] Ronnie E.S. Santos, Cleyton V.C. Magalhes, Luiz Fernando Capretz, Jorge S. Correia-Neto, Fabio Q.B. Da Silva, and Abdelrahman Saher. Computer games are serious business and so is their quality: Particularities of software testing in game development from the perspective of practitioners. *International Symposium on Empirical Software Engineering and Measurement*, 10 2018. doi:10.1145/3239235.3268923.

[79] Ben Seifert. It's Raining New Content: Successful Rapid Test Iterations. 2013. URL: https://www.gdcvault.com/play/1018011/It-s-Raining-New-Content.

[80] Liraz Shay. BDD cucumber features best practices, Feb 2021. URL: https://www.linkedin.com/pulse/bdd-cucumber-features-best-practices-liraz-shay/.

[81] Thiago Rocha Silva, Jean-Luc Hak, and Marco Winckler. Testing prototypes and final user interfaces through an ontological perspective for behavior-driven development. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9856 LNCS:86–107, 2016. doi:10.1007/978-3-319-44902-9_7.

[82] Thiago Rocha Silva, Jean-Luc Hak, and Marco Winckler. A Behavior-Based Ontology for Supporting Automated Assessment of Interactive Systems. In *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*, pages 250–257. IEEE, 2017.

[83] Thiago Rocha Silva, Jean Luc Hak, and Marco Winckler. A Formal Ontology for Describing Interactive Behaviors and Supporting Automated Testing on User Interfaces. *International Journal of Semantic Computing*, 11:513–539, 12 2017. doi:10.1142/S1793351X17400219.

[84] Thiago Rocha Silva, Marco Winckler, and Hallvard Trætteberg. Ensuring the Consistency Between User Requirements and Graphical User Interfaces: A Behavior-Based Automated Approach. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11619 LNCS:616–632, 2019. doi:10.1007/978-3-030-24289-3_46.

[85] Thiago Rocha Silva, Marco Winckler, and Hallvard Trætteberg. Ensuring the Consistency Between User Requirements and GUI Prototypes: A Behavior-Based Automated Approach. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11746 LNCS:644–665, 2019. doi:10.1007/978-3-030-29381-9_39.

[86] Apoorva Srivastava, Sukriti Bhardwaj, and Shipra Saraswat. SCRUM model for agile methodology. *Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2017*, 2017-January:864–869, 12 2017. doi:10.1109/CCAA.2017.8229928.

[87] Roy Tan and Stephen Edwards. Evaluating Automated Unit Testing in Sulu. pages 62–71, 05 2008. doi:10.1109/ICST.2008.59.

[88] Tutorialspoint.com. Behavior Driven Development - Tools. URL: https://www.tutorialspoint.com/behavior_driven_development/behavior_driven_development_tools.htm.

[89] Unity. Unit 4 - Gameplay Mechanics - Unity Learn. URL: https://learn.unity.com/project/unit-4-gameplay-mechanics.

[90] Unity. What is Unity? - Unity Learn. URL: https://learn.unity.com/tutorial/what-is-unity.

[91] Jan van Valburg. Automated Testing in Call of Duty, 2018. URL: https://research.activision.com/publications/archives/automated-testing-in-call-of-duty.

[92] Tom Wissink and Carlos Amaro. Successful Test Automation for Software Maintenance. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, ICSM '06, page 265–266, USA, 2006. IEEE Computer Society. doi:10.1109/ICSM.2006.63.

Appendix A

Observed Game Behaviours

This appendix presents each of the games found in the Unity course "Create with Code"[1]. Each game has a short description and a table containing behaviours found in that game and their corresponding category as defined in chapter 6 "Game Behaviour Framework: Behavioural patterns in games". The categories are indicated by their number and name. The behaviours in the table are all of the behaviours described in the Unity course.

Game 1: A car driving on a floating road, trying to avoid (or hit) obstacles in the way.

	Category	Behaviour
Level 1		
1	Player Movement	Arrow left/right turns player
1	Player Movement	Arrow forward/backward moves player
2	Camera Movement	Switch camera perspective on keypress
Level 2		
14	Colliding	Hitting obstacle moves obstacle
19	Movement Trigger	The camera follows the player

Game 2: Side-view game flying a plane around obstacles

	Category	Behaviour
Level 1		
1	Player Movement	Plane tilts when the player presses arrow up/down
1	Player Movement	Plane does not tilt when no key is pressed
8	Independent Behaviour L1	Plane constantly moves forward
8	Independent Behaviour L1	The plane's propeller spins
Level 2		
14	Colliding	The plane stops when hitting a wall
19	Movement Trigger	The camera follows the plane

Game 3: A top-down game with the objective of throwing food to hungry animals - who are stampeding towards you - before they can run past you.

[1] Create with Code - Unity Learn

	Category	Behaviour
Level 1		
1	Player Movement	Left/right moves player left/right
11	Spawning	Spawn animal at a random point on the line
8	Independent Behaviour L1	Animals move forward (towards player)
Level 2		
16	Throwing/shooting	Spacebar throws food (instance of prefab), starting at the player's position and moving strictly forward
16	Throwing/shooting	Spawn (random - from list) animal (instance of one of the prefabs) on button-pres
18	Collision Trigger	When an instance of food 'hits' an animal, both get destroyed
18	Collision Trigger	Player stays in bounds
20	Location-based Trigger	When an animal reaches the bottom, show the game over and stop the game
20	Location-based Trigger	Destroy food that left screen
20	Location-based Trigger	Destroy animals off-screen
21	Time-based Trigger	Spawn the animals in intervals starting after delay
	Extensions on base game:	
Level 1		
1	Player Movement	Vertical movement for player
Level 2		
21	Time-based Trigger	Animals coming from side that can kill player
18	Collision Trigger	Animal passing or getting hit reduces lives, starting with 3 lives
18	Collision Trigger	Every successful feeding gives 1 point, score starts at 0
Level 3		
25	Fight	Animal have hunger bar, food fills the bar x% (different per animal), animal stays alive until bar is full

Game 4: Fetch: balls are randomly falling from the sky and you have to send your dog out to catch them before they hit the ground.

	Category	Behaviour
Level 1		
11	Spawning	A random ball (of 3) is generated at a random x position above the screen
Level 2		
16	Throwing/shooting	When the user presses the spacebar, a dog is spawned and runs to catch the ball (ie. moves left at constant speed)
18	Collision Trigger	If the dog collides with the ball, the ball is destroyed
18	Collision Trigger	If the ball hits the ground, a "Game Over" debug message is displayed
21	Time-based Trigger	Spawn internal is random value between 3 and 5 second, changing
21	Time-based Trigger	Dog can only be spawned every X seconds to avoid spamming
22	View-based Trigger	The dogs and balls are removed from the scene when they leave the screen

Game 5: An endless side-scrolling runner game where the player needs to time jumps over oncoming obstacles to avoid crashing.

	Category	Behaviour
Level 1		
1	Player Movement	Player jumps when the button is pressed
8	Independent Behaviour L1	Background moves past at the same speed as obstacles
8	Independent Behaviour L1	Background repeats
8	Independent Behaviour L1	Music plays during game
Level 2		
15	Simple Condition	Player cannot double-jump, only jump when on the ground
15	Simple Condition	On game over display message and stop background and obstacle movement
15	Simple Condition	On game over stop spawning of objects
15	Simple Condition	After game over stop player from jumping
18	Collision Trigger	When player collides with obstacle the game is over
18	Collision Trigger	Boom is played on crash, followed by cloud of smoke
19	Movement Trigger	Sound effect is played on jump
19	Movement Trigger	Dust is shown after running feet
21	Time-based Trigger	Obstacles spawn at interval
22	View-based Trigger	Objects that leave the screen get destroyed
Level 3		
26	NPC Creation	Obstacles (instance of prefab) spawn outside of the screen, on the floor, and move left
28	Extended Condition	The player runs during game when on the ground, shows jump animation during jump, fall animation during fall, and death animation on game over

Game 6: An endless side-scroling game with a balloon floating through town, picking up tokens while avoiding explosives.

	Category	Behaviour
Level 1		
1	Player Movement	Balloon floats upwards as player holds spacebar
8	Independent Behaviour L1	Balloon falls without spacebar
8	Independent Behaviour L1	Background seamlessly repeats, simulating the balloon's movement
8	Independent Behaviour L1	Bomb and Money tokens move left at constant speed - same as background
Level 2		
15	Simple Condition	Balloon stays within screen - can only float upwards to top of screen, bounce of ground (with sound effect)
18	Collision Trigger	Colliding with money gives a particle (at place of money/player) and sound effect
18	Collision Trigger	Colliding with Bomb gives particle (at place of bomb/player) and sound effect and stops background (game over)
21	Time-based Trigger	Bombs and Money tokens are spanwed randomly on a timer

Game 7: An arcade-style Sumo battle with the objective of knocking increasingly difficult waves of enemies off of a floating island, using power ups to help defeat them.

	Category	Behaviour
Level 1		
1	Player Movement	Player rolls in direction of camera based on vertical input
2	Camera Movement	Camera rotates around the island based on horizontal input
Level 2		
14	Colliding	Spheres bounce off of each other
18	Collision Trigger	When the player collects a powerup, a visual indicator appears
20	Location-based Trigger	Enemy gets destroyed after it falls of platform
21	Time-based Trigger	After a 7 seconds, the powerup ability and indicator disappear
23	Event-based Trigger	A new powerup gets spawned with each wave;
Level 3		
24	Independent Behaviour L3	Enemy follows the player around
25	Fight	When the player collides with an enemy while they have the powerup, the enemy goes flying
25	Fight	Create a new powerup that gives the player the ability to launch projectiles at enemies to knock them off (or something that automatically fires projectiles in all directions when the powerup is enabled)
26	NPC Creation	Enemy spawns at random location on the island
26	NPC Creation	Enemies get spawned in waves, starting with 1 enemy and adding an enemy each wave
26	NPC Creation	Add a new more difficult type of enemy and randomly select which is spawned.
27	Powerup	when player collides with powerup, the powerup disappears, a powerup indicator appears, and the player has the powerup

Game 8: An arcade-style game where the player controls a soccer ball, pushing enemy balls into their goal while preventing the enemy balls from getting in the player's goal, also includes powerups.

	Category	Behaviour
Level 1		
8	Independent Behaviour L1	The enemy balls are moving toward the player's goal
Level 2		
13	Special Control	The player gets a speedboost whener he presses spacebar - and a particle effect should appear when they use it
14	Colliding	On collision enemy should be pushed away from player
21	Time-based Trigger	The powerup gets deactivated after some time
Level 3		
26	NPC Creation	One enemy should be spawned in wave 1, two in wave 2, three in wave 3, etc
28	Extended Condition	a new wave spawns when all enemy balls have been removed
28	Extended Condition	The enemies' speed should increase a small amount which each wave so that it gets more difficult

Game 9: Click and destroy objects randomly tossed in the air before they can fall off the screen

	Category	Behaviour
Level 1		
4	Assets Exist	There is a UI element for score on the screen
5	Booting	At the start of the game a title screen is shown, with a title and difficulty buttons
7	Out-menu Navigation	When the restart button is clicked a new game is started
8	Independent Behaviour L1	Objects are given random speed, position, and torque
9	Persistent Storage	When the easy mode button gets clicked the game starts in easy mode
9	Persistent Storage	Add a UI Slider element to adjust the volume
10	Sound	Add background music
Level 2		
13	Special Control	If you click on an object, it is destroyed
13	Special Control	During gameplay, allow the user to press a key to toggle between pausing and resuming the game, where a pause screen comes up while the game is paused.
13	Special Control	Program click-and-swipe functionality instead of clicking, generating a trail where the mouse has been dragged.
15	Simple Condition	When the game is over "Game Over" appears in the middle of the screen
15	Simple Condition	When the game is over a restart button appears
15	Simple Condition	While playing the restart button is inactive
15	Simple Condition	During the game the title screen is not visible
17	Counter	Create a "Lives" UI element that counts down by 1 when an object leaves the bottom of the screen and triggers Game Over when Lives reaches 0
17	Counter	The player's score is tracked and displayed by the score text when hit a target
18	Collision Trigger	If a good object falls below screen the game is over
21	Time-based Trigger	One of 4 random objects is tossed into the air on intervals
22	View-based Trigger	If an object drops below screen it is destroyed
23	Event-based Trigger	There are particle explosions when the player gets an object
Level 3		
28	Extended Condition	If the game is over no more objects are spawned
28	Extended Condition	If the game is over no more objects can be clicked

Game 10: Whack-a-mole-like challenge in which you have to get all the food that pops up on a grid while avoiding the skulls.

	Category	**Behaviour**
Level 1		
4	Assets Exist	The score should always say, "Score: __" with the value displayed after "Score:"
9	Persistent Storage	When you click Easy, the spawnRate should be slower - if you click Hard, the spawnRate should be faster
Level 2		
13	Special Control	The food should be destroyed when the player clicks on it
15	Simple Condition	When you lose the restart button appears
21	Time-based Trigger	Add a "Time: __" display that counts down from 60 in whole numbers (i.e. 59, 58, 57, etc) and triggers the game over sequence when it reaches 0

Appendix B

Interviews

B.1 Baseline interview

Below is an introductory interview, undertaken with the lead developer to gain an understanding of the development practices currently in use at the *Virtual Brigade*.

Q1. Wanneer ben je bij het project gekomen?

Ik ben aangenomen toen development begon. Op dat moment bestond de roadmap al.

Q2. Hoe zijn de requirements bepaald?

Er zijn geen requirements opgesteld, de roadmap enige documentatie van doel. Er is wel een algemeen idee van waar het project heen moet maar dit is niet ergens gespecifierd.

Q3. Hoe en waar zijn de requirements opgeschreven?

Er zijn niet echt requirements, er is alleen een roadmap.

Q4. Word er nog regelmatig gekeken naar de requirements?

Tijdens sprintreviews (maandelijks) word de progressie gecontroleerd met de roadmap.

Q5. Welke tools of processen worden gebruikt voor het bijhouden van de requirements?

Er zijn geen requirements, dus niet van toepassing.

Q6. Is er werk gedaan om te zorgen dat alle requirements duidelijk en volledig zijn? Zo ja, hoe?

Nvt.

Q7. Zijn er misverstanden geweest over de requirements? Zo ja, hoe is hier mee omgegaan?

Nvt.

Q8. Hoe word bijgehouden of de requirements vervuld zijn?

Nvt.

Q9. Kan je beschrijven hoe een sprint er ongeveer uit ziet?

De leads hebben wekelijks een meeting, hier word bepaald wat er die week gedaan word. Voor elk van deze features word een ticket aangemaakt in Azure. Deze items worden vervolgens samen met de stagiairs verdeeld. Zodra de feature is toegewezen start de developer een nieuwe branch gelinkt aan de ticket. Op deze branch word de feature geimplementeerd. Zodra de developer vind dat de feature klaar is verplaatst hij de ticket naar Ready To Merge. Een van de leads (afhankelijk van het soort ticket) pullt de branch en controleert de feature. Dit word gedaan door middel van playtesting, en in het geval van programeertickets door vluchtig door de code te kijken om te controleren of het er, naar inzicht van de lead, logisch uitziet. Zodra de ticket naar Ready To Merge gaat word automatisch een pull request aangemaakt vanaf de gelinkte branch naar dev. Als de lead tevreden is keurt hij deze goed en word de code gemerged en de ticket gemarkeerd als Done. Op het einde van elke sprint, dus elke maand, word de Dev branch naar Master gemerged.

Q10. Hoe worden nieuwe taken aangemaakt/bepaald? Hoe worden ze vervolgens toegewezen?

Taken worden wekelijks door de leads bepaald en aangemaakt, hierbij word rekening gehouden met de roadmap. Ze worden in een algemene teamvergadering verdeeld.

Q11. Hoe ziet het process er uit als een developer denkt klaar te zijn met een item?

Zie beschrijving van sprint.

Q12. Kan je beschrijven hoe het test process er uit ziet?

Op het moment is er geen test process.

Q13. Wat voor soort testen worden er uitgevoerd? Unit testing, integration testing, etc.

We zijn op het moment bezig met playtesting. Dan pakken we de laatste build en krijgt iedere developer om en om een half uur om het te spelen en te testen. Hierbij word vooral gefocused op controleren of bugs van de laatste sprint goed opgelost zijn.

Q14. Er komen ook wel eens eindgebruikers of geintresseerden langs, hoe gaat dat process? Word daar testdata uit gehaald?

Dit zijn alleen demo's, hier word niet getest.

Q15. Hoe word bepaald wat er getest word? Worden alleen nieuwe features getest of word het hele product ook getest op zoek naar nieuwe bugs?

Er is geen process voor testen, mensen bepalen zelf wat ze testen.

Q16. Kan je beschrijven hoe bugs gemiddeld gevonden worden?

Hier zit geen process achter, gebeurt eigenlijk bij toeval tijdens het spelen.

Q17. Hoe worden deze bugs vervolgens opgelost?

Als een bug gevonden word word er een ticket voor aangemaakt en word die in een komende sprint toegewezen.

Q18. Worden gevonden bugs genoteerd zodat later getest kan worden of ze niet terug komen?

Dit gebeurt niet.

Q19. Hoe worden bekende bugs bijgehouden?

Dit gebeurt niet.

Q20. Houden jullie bij wat er getest is en wat het resultaat van de tests was? Zo ja, hoe?

Nee

Q21. Hebben jullie metrics om inzicht te krijgen in hoe goed er getest word?

Nee

Q22. Hoe word er voor gezorgd dat alles getest is?

Dit gebeurt niet.

Q23. Word er gecommuniceerd over de gedane tests, bijvoorbeeld naar de productowner?

Dit gebeurt niet.

Q24. Hoe word er gecommuniceerd tussen developers, testers, etc.?

Dit gebeurt niet.

Q25. Wat gaat er volgens jou mis in het test process?

Er is geen process.

Q26. Wat zou je graag beter zien aan het test process?

Er moeten vaste testmomenten komen en er moet een plan voor het testen komen.

Q27. Zijn jullie bezig met plannen om de documentatie, testing, of bug management te verbeteren? Zo ja, hoe zien die plannen er nu uit?

Er is nog geen plan voor na d eoplevering van versie 1.0.

Q28. Wat zou je zelf graag veranderen aan het process?

Er word nu eigenlijk niet gebruik gemaakt van een duidelijk process voor development. Er zijn wel sprints, maar eingelijk worden deze niet echt uitgevoerd zoals bedoeld. Er is te weinig mankracht om de tijd te besteden om een goed process op te zetten.

Het testprocess moet beter, maar er is geen tijd of mankracht om dit beter te doen.

Ik wil wel graag het process verbeteren, maar ik zou niet weten hoe. Daarnaast zitten we zo dicht bij de deadline dat het nu ook geen zin meer heeft.

Voor na de oplevering zou het mooi zijn om een beter process toe te passen maar ik weet niet waar we daarmee zouden moeten beginnen.

B.2 Interview on scenario creation

These interview are done after the meeting between the "Three Amigos" and after the scenarios have been created. Three interviews were taken, one with the team lead and tester, one with the developer, and one with the product owner. The questions asked and their answer are included below (in Dutch).

Team lead / Tester Questions 1 and 2 are to gauge the sentiment towards the method up to now. Questions 3 and 4 aim to discern the success of the meeting. Question 5 asks if they succeeded in executing the latest step - creating scenarios. Followed by questions 6 and 7 which look towards the future and ask about the expectation during development and the trust in the final product.

Q1. Wat vond je van de meeting?

Prima. Aan het begin was het nog erg abstract wat er precies van ons verwacht werd, maar gedurende meeting werd dit duidelijk.

Q2. Wat vind je van de methode dusver?

Het idee is goed. Het duurt allemaal wel langer dan ik had verwacht, maar dit komt natuurlijk ook doordat het allemaal nieuw is. Het werkt waarschijnlijk echt goed bij een groot bedrijf waar de tickets uit echte stakeholders komen. Het is hier toch wat lastig aangezien veel tickets intern vanuit onszelf komen. Na de trial moeten we kijken hoe dat gaat. Ongeveer de helft van de tickets komt vanuit onszelf, dan zouden we misschien onderling tussen de developers kunnen sparren. De andere helft komt vanuit de opleidingen, dan kunnen we met hun sparren over de features.

Q3. Is het duidelijk wat de feature inhoud?

Ja. Het is ook duidelijk opgeschreven wat er verwacht word. In het begin was het nog erg onduidelijk hoe het featurebestand er uit moest gaan zien maar tijdens de meeting werd dit duidelijk.

Q4. Is het beeld van de feature anders door het process, bij jou of de anderen?

Ja, dat zit hem dan vooral in de specifieke dingen. We hadden allemaal wel een algemeen idee van hoe de feature er uit zou zien maar hoe het allemaal zou werken was nog niet duidelijk. We hadden allemaal wel in ons hoofd dat er een stap zou zijn om het kijken uit te leggen, maar of je dan ergens naar moet kijken of een bepaald aantal graden hadden we nog niet over na gedacht. Het is tijdens deze meeting dat we samen het idee hebben gekregen om de speler naar objecten te laten kijken, wat we nu gaan implementeren. Anders waren al deze keuzes aan de developer geweest, nu hebben we van te voren bedacht hoe het er uit moet zien en wat er geimplementeerd moet worden.

Q5. Hoe ging het opstellen van de scenarios?

Goed, in het begin was het even zoeken naar hoe dingen verwoord moeten worden, maar uiteindelijk wel uitgekomen. Jij (red. de researcher) hebt duidelijk in je hoofd hoe alles werkt doordat je er al zo lang mee bezig bent en dan komen er nieuwe mensen die geen idee hebben waar het over gaat. Dat is lastig. Maar uiteindelijk zijn we er samen (red. team onder leiding van researcher) uitgekomen en was het wel duidelijk.

Q6. Wat verwacht je dat er met de geschreven scenarios gebeurt tijdens de implementatie?

Ik denk dat ik ze als acceptatiecriteria ga gebruiken. Als de feature klaar is dan kan ik aan de hand van de scenarios kijken of alles werkt zoals afgesproken. Ik verwacht niet dat de developer tijdens het implementeren naar de scenarios zal kijken, maar een eigen lijstje van todos zal gebruiken. Waarschijnlijk kijkt hij er ook pas weer naar als hij denkt klaar te zijn.

Q7. Heeft dit iets gedaan met je vertrouwen in dat de functie zal werken en er uit zien zoals bedoeld?

Ik had wel vertrouwen in dat het goed ging komen, maar het is toch wel fijn dat het nu zo is vastgelegd. Dit kan ik nu als onderbouwing gebruiken tijdens het testen als dingen dan anders zijn of niet werken zoals verwacht. Waarschijnlijk hoef ik de feature nu niet terug te sturen.

Q8. Heb je nog overige opmerkingen of vragen over hoe het nu verder gaat?

Of we deze methode blijven gebruiken gaat vooral afhangen van hoeveel tijd er in het genereren van al die bestanden gaat zitten. Het is nu natuurlijk nieuw dus gaat er veel tijd in zitten, maar verwacht dat we hier wel sneller in gaan worden. Als het meer tijd kost dan het oplevert dan zullen we het niet blijven gebruiken.

We zullen ook moeten gaan kijken voor welke features we het daadwerkelijk willen uitvoeren. Een feature als "Maak een tekstvak scrolbaar" is in een halfuur geimplementeerd en niet complex, dan gaan we niet dit hele process doorlopen. De waarde zal vooral liggen in grotere features. Vooral als de feature word toegewezen aan een stagiair. Dit process forceert ons dan om van te voren na te denken over hoe het er uit moet komen te zien en dit met de stagiair te bespreken. Ik denk dat dat hun heel erg helpt en zorgt dat we features minder vaak terug hoeven te sturen.

Developer

Q1. Wat vond je van de meeting?

Normaliter is er geen plan voor hoe je een opdracht moet aanpakken. Deze meeting kost tijd aan het begin, maar bespaart later waarschijnlijk tijd doordat je niet meer over het plan hoeft na te denken of te discussieren.

Q2. Wat vind je van de methode dusver?

Aan het begin kost het wel veel tijd, maar hopelijk bespaart het uiteindelijk.

Q3. Is nu duidelijk wat de feature inhoud?

Ja. Er was wat onduidelijkheid over wanneer iets geldig was. Nu is er meer duidelijkheid over wat de feature moet kunnen.

Q4. En wat PO verwacht van deze feature?

Door deze meeting hebben we nu gelijk afgestemd wat het idee is. Normaal deden we dat nooit. Dan moest je zelf bedenken hoe je dingen ging implementeren of hoe dingen er uit moeten komen te zien. Dat gaat meestal wel goed maar je loopt wel het gevaar dat je het opnieuw moet doen als je het anders deed dan [PO] in gedachten had.

Q5. Hoe ging het opstellen van de scenarios?

Goed, door het opstellen heb ik wat uitgebreider over de feature na gedacht. Normaal begin ik gewoon en dan kom je onderweg de problemen wel tegen en lost je ze dan op. Hierdoor kun je eerder voorspellen hoe dingen er uiteindelijk uit gaan zien en keuzes maken hoe je iets aan gaat pakken.

Q6. Is je beeld van de feature anders door dit process?

Ja, we hebben tijdens de meeting eigenlijk pas bedacht hoe de feature er precies uit moest komen te zien. We hebben toen onder andere bedacht om objecten te gebruiken om het kijken uit te leggen. Anders had ik dit nooit zo bedacht.

Q7. Wat verwacht je met de geschreven scenarios te doen tijdens development?

Ik denk dat ik ze ga gebruiken als een soort checklist, dan kan ik gaandeweg in de gaten houden in welke volgorde de stappen moeten en ze zo implementeren.

Q8. Wil je nog iets anders kwijt?

Nee.

Product Owner

Q1. Wat vond je van de meeting?

Prima. In het begin was het erg onduidelijk wat de bedoeling was, maar uiteindelijk was het wel duidelijk wat er gedaan moest worden.

Q2. Wat vind je van de methode dusver?

Duidelijk. Hierdoor kunnen we duidelijk opstellen wat er gedaan moet worden.

Q3. Heb je het gevoel dat je duidelijk op heb kunnen schrijven wat je verwacht van deze feature?

Ja, de scenarios geven duidelijk aan hoe de feature zou moeten werken.

Q4. Ben je dingen tegengekomen in het process waar de verwachtingen anders waren?

Nee, niet echt. We hebben wel invulling kunnen geven aan de feature. Bijvoorbeeld hoe je het rondkijken uitlegt en controleerd dat het gedaan is. Daar hadden we andere ideeen over. Alle opties hadden gewerkt dus dat was sowieso wel goed gekomen.

Q5. Heeft dit je geholpen duidelijk te maken wat je bedoelde met deze feature?

Ja, door het op te schrijven en zo uit te schrijven staat het er gewoon duidelijk. Hierdoor is er minder kans op fouten.

Q6. Heb je meer of anders nagedacht over de feature door dit process?

Ja, tijdens zo'n brainstorm moet je heel concreet nadenken over hoe de feature werkt. Bijvoorbeeld met de voorbeeld van rondkijken had ik alleen in mijn hoofd dat dat in de tutorial moest, maar hoe precies het er uit moest zien had ik nog niet over nagedacht.

Q7. Heeft dit je beeld van wat de feature technisch inhoud veranderd, en zo ja hoe?

Ja en nee, hierdoor heb ik wel meer inzicht van hoe de feature technisch gerealiseerd gaat worden, misschien heeft het ook wel veranderd hoe de feature technisch geimplenteerd gaat worden.

Q8. Heeft de meeting invloed gehad op je vertrouwen dat de feature gaat worden zoals verwacht?

Ja, ik had wel vertrouwen dat het goed zou komen, maar door het uitschrijven van de scenarios heb ik nu ook vertrouwen dat mijn visie word uitgevoerd.

B.3 Interview after implementation

Developer

Introductie

Q1. Is het gelukt om de feature te implementeren?

Ja, hij werkt.

Vertrouwen

Q2. Heb je er vertrouwen in dat de feature werkt zoals zou moeten?

Ja, geen twijfel.

Q3. Hebben de testen een effect gehad op dit vertrouwen?

Ja, denk daardoor in stappen en darrdoor kom je tijdens het process in een lijn met je team. Het werkt en voldoet ook gelijk aan de verwachtingen van Jesse.

Effect tijdens implemetatie

Q4. Heb je het gevoel iets anders gedaan te hebben tijdens implementatiefase door de scenarios en de voorbespreekmeeting?

Denkt van te voren naar hoe je zou kunnen testen. Het ging een beetje de kant op van test-driven development.

Q5. Heb je tijdens de implementatie nog vragen gehad over wat het gevraagde gedrag was is een bepaalde situatie?

Nee, was allemaal duidelijk. Waren misschien verschillende manieren maar gewoon een gekozen want als het doet wat gevraagd word is het goed.

Q6. Denk je dat je dit zonder deze methode wel gehad zou hebben?

Waarschijnlijk had het langer geduurd. Was dan niet teruggekomen met vragen maar gewoon wat gedaan. Vooraf kost het een hoop tijd maar uiteindelijk bespaar je tijd.

Q7. Heb je tijdens implementatie nog naar de scenarios gekeken?

Ja, de hele tijd er naast gehouden en gebruikt en dan heb je ook dat je leert dat jeen scenario soms beter in een andere manier kan schrijven en dat je daar vaardiger in word

Q8. Heb je door de scenarios te schrijven meer nagedacht over edgecases?

Ja, eigenlijk wel. Bijvoorbeeld dat je niet kan lopen voordat je het kijkdeel hebt afgerond

Testbaarheid scenarios

Q9. Heb je alle scenarios omgezet in tests?

Geprobeerd, maar niet gelukt. De buildsettings van de *Virtual Brigade* maakte testen lastig en niet genoeg expertise en tijd om op te lossen

Q10. Heb je dingen veranderd aan de scenarios om ze te kunnen testen?

Ja, dingen gewijzigd om ze duidelijker en testbaar te maken. Het taaltje en die manier moet je leren, als je er dan weer naar kijkt zie je dingen die beter kunnen.

Q11. Heb je bij het testen fouten gevonden in je implementatie?

NVT. Viel wel error in ander deel van de code op.

Evaluatie tool

Q12. Was het duidelijk hoe de tool werkt?

Ja. Documentatie was duidelijk, je had het al een keer laten zien. Met beetje experience moet het lukken

Q13. Deed de tool wat je verwachte?

JA, deed wat ik verwachte. Hij genereerde gewoon tests van die scenarios.

Q14. Zijn er dingen die je miste in de tool?

Nee, hij deed gewoon wat ik verwachte.

Evaluatie methode algemeen

Q15. Heb je het gevoel dat dit process je geholpen heeft of heeft het meer gehinderd?

Het heeft wel geholpen. Je bent al gelijk afgestemd over die stappen en wat ie moet doen om die stappen te halen. Voor deze feature was het super handing

Q16. Hoeveel tijd denk je bezig te zijn geweest met deze methode ten opzichte van hoe je her regulier zou doen?

Ik denk juist tijd gewonnen.

Q17. Zou je deze methode vaker willen gebruiken?

Ik denk dat het weleen beetje te veel gedoe is aangezien de VB er niet op uitgerust is en al zo groot is. Als je een nieuw product zou starten zou ik het zeker wel weer gebruiken.

Q18. Wil je nog iets kwijt?

Nee.

PO

Q1. Is het resultaat wat je verwacht had?

Ja.

Q2. Als je mag kiezen zou je deze method dan nogmaals gebruiken of niet, en waarom?

Met een groter team zeker. In dit team waarschijnlijk niet. We zijn een erg klein team met niet strak gedefinieerde rollen. WE zijn heel laagdrempelig. Als de developer tijdens het implementeren ergens over twijfelt of tegen aan loopt dan kan hij gewoon binnenstappen en kunnen we het bespreken. Bij een groter bedrijf heb je dat niet, ik denk dat het dan wel veel waarde zou toevoegen.

Q3. Denk je dat deze method uiteindelijk meer tijd heeft gekost of heeft opgeleverd?

Het kost aan het begin wel wat meer tijd maar dan kan het wel in 1 keer goed gedaan worden. Het is zeker de initiele tijd waard.

Team Lead / Tester

Q1. Is het resultaat wat je verwacht had?

Ja, de feature werkt en zoals ik verwacht had. Het werkt precies zoals we gespecifieerd hadden.

Q2. Als je mag kiezen zou je deze method dan nogmaals gebruiken of niet, en waarom?

Ik zie zeker waarde in de methode voor grotere features. De method forceert ons dan om van te voren na te denken over wat we willen bereiken met de feature, dit zal vooral voor de stagiair het implementeren veel makkelijker maken. Het geeft mij ook een mooie checklist met of de feature alles doet wat we van te voren bedacht hadden.

Het maken van de tests is nu niet gelukt. Dit komt vooral doordat we gewoon niemand met die ervaring hebben en de *Virtual Brigade* er niet op gebouwd is. Als we dat echt willen gaan doen dan zouden we daar veel tijd in moeten steken of iemand met ervaring moeten inhuren om dat in de code in te bouwen. Met het gebrek aan mankracht dat we nu al hebben zie ik dit niet snel gebeuren.

Appendix C

Class Diagrams

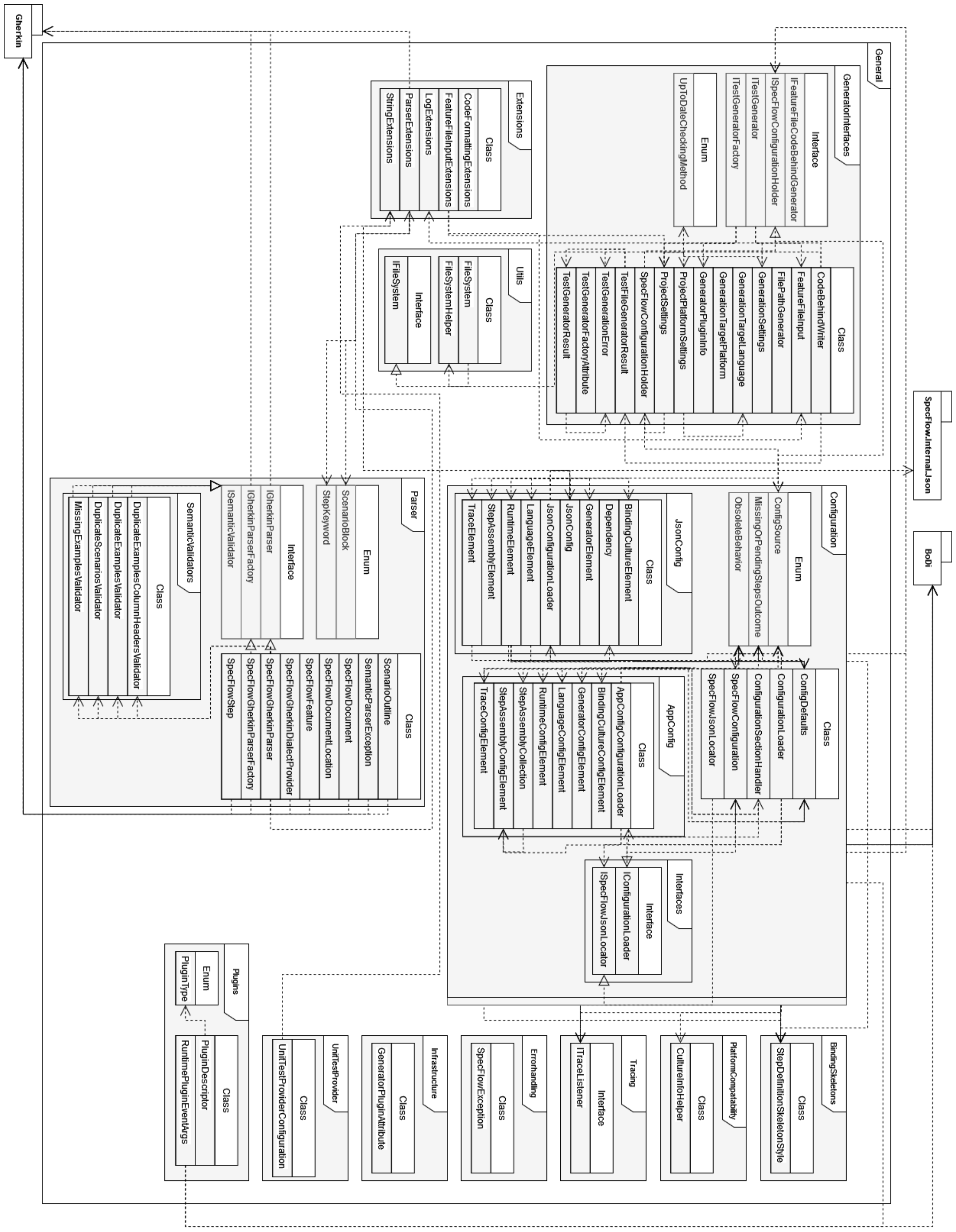


FIGURE C.1: Class Diagram UnitySpec.General

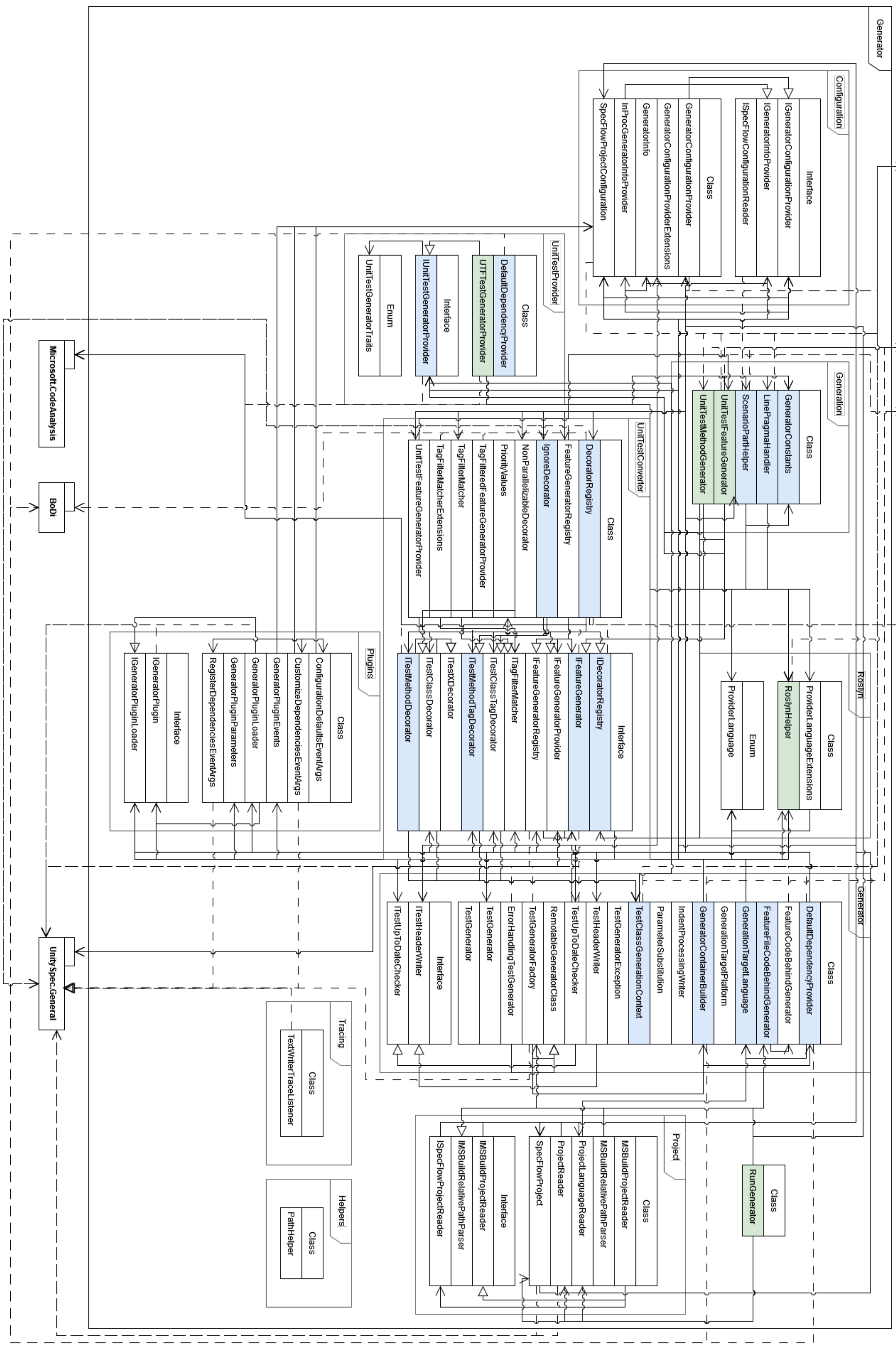


FIGURE C.2: Class Diagram UnitySpec.Generator
Blue indicates meaningfully changed, green indicates own addition or rewritten more than 95%

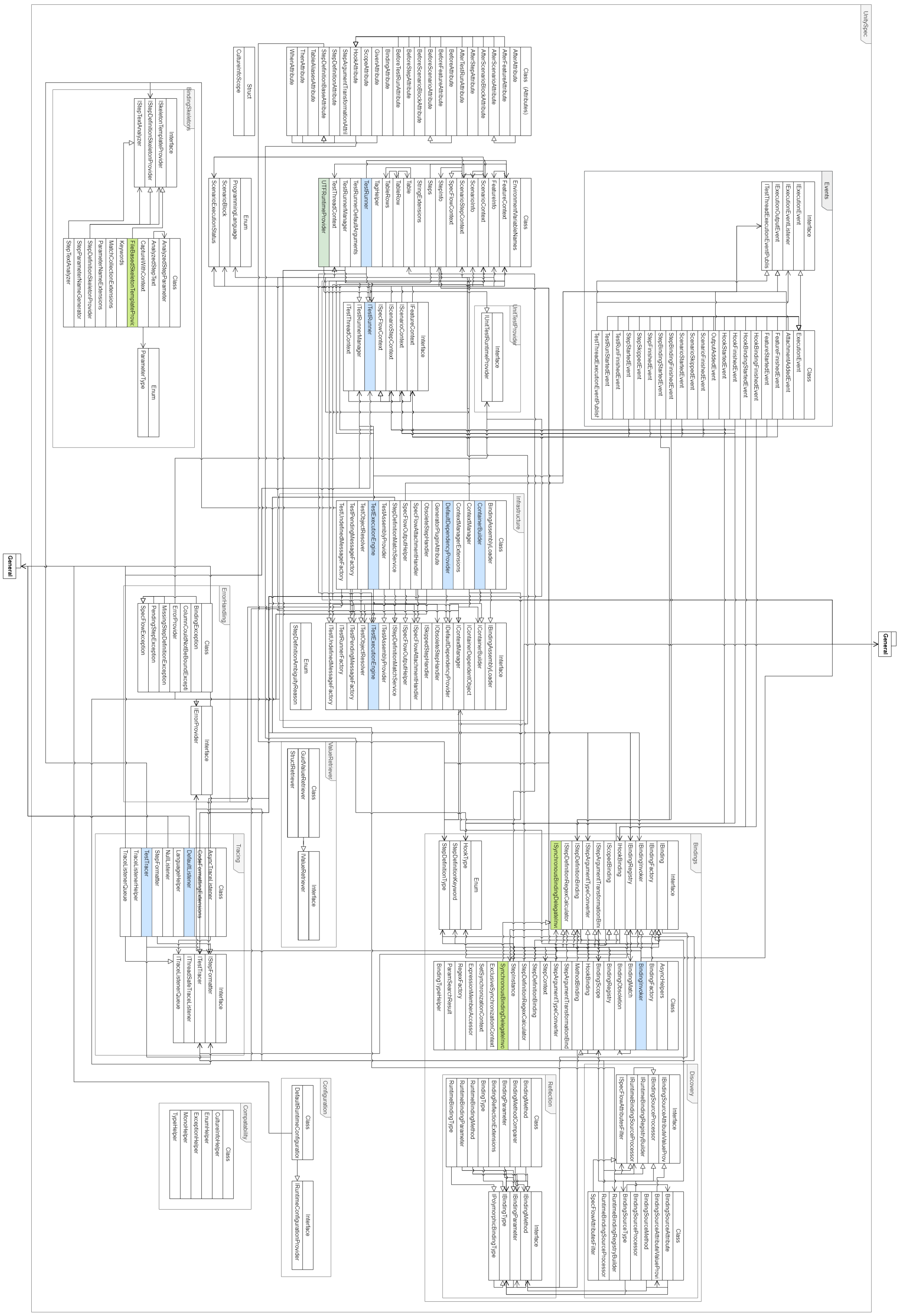


FIGURE C.3: Class Diagram UnitySpec Runner
Blue indicates meaningfully changed, green indicates own addition or rewritten more than 95%