Introducing automated testing to video game development via Behaviour-Driven Development

Michael Mulder and Petra van den Bos
Formal Methods and Tools, University of Twente, Enschede, the Netherlands

ABSTRACT Game Software Engineering has emerged as a specialized field distinct from traditional software engineering, addressing unique challenges inherent to the creative process of game development, but lagging behind in using new methods of software engineering. This paper introduces the method of Behavior-Driven Development (BDD) to game software engineering. BDD is popular in software engineering for modelling and testing software. In this paper, we first propose a development process for applying BDD in game development. Then, we provide an integration of BDD tooling in Unity 3D, a major platform for game development. Next, we present a framework for identifying and categorizing game behaviours, to cater for modelling game behaviours in BDD scenarios. Finally, we show applicability of these three contributions on a real-world case study.

KEYWORDS Behaviour-Driven Development, video game development, automated testing, software engineering

1. 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 (Ballhaus et al. 2022; Helplama.com 2023). Game Software Engineering is a growing domain that has evolved distinctly from traditional software engineering paradigms. While plenty of research has been done to improve traditional software development methods, game software engineering lags behind (Chueca et al. 2024). However, the widespread occurrence of bugs and crashes continues to cast a shadow over the user experience within this thriving industry (Murphy-Hill et al. 2014). While some bugs may be innocent, such as deformed visuals, bugs that affect the functioning of a game may reduce the game experience significantly, or even prevent a player from playing the game again, e.g. when a level cannot be completed, or when progress cannot be saved.

Nevertheless, game developers often opt to deviate from standard software development techniques, and in particular (structured) testing, because they see game creation as an art, not a science, and testing as a waste of money and time (Politowski et al. 2022). If tests are performed, then this is focussed on testing the fun-factor (Politowski et al. 2021), game rule balance, user experience, and emotional appeal (Kasurinen et al. 2014). To this end, methods such as user research and play-testing are applied (Mirza-Babaei et al. 2020).

In traditional software engineering, Behavior-Driven Development (BDD) has been broadly adopted, and is becoming an established industry practice (Farooq et al. 2023; Binamungu & Maro 2023). BDD is an agile development approach for modelling and testing software, and consists of three phases. In the discovery phase, stakeholders brainstorm about concrete examples of system behaviours, e.g. starting from an abstract feature description such as a user story. Then, in the formulation phase, these examples are consolidated as scenarios, written in natural language, often structured according to a so-called Given-Then-When style as prescribed by the semi-formal Gherkin standard. In this way, the scenarios function as small models for system behaviours, that are easily understandable because of use of natural language. As a last step, the scenarios are transformed to automated test cases. This step is supported by existing tools like Cucumber (Cucumber 2025), and Gauge (Gauge 2025). When these tests are created before writing code, BDD also is a Test-Driven Development approach (Al-Saqqa et al. 2020; North 2006).

In this paper, we investigate how BDD can be integrated in game development. BDD is interesting and relevant for game

JOT reference format:
Michael Mulder and Petra van den Bos. *Introducing automated testing to video game development via Behaviour-Driven Development.* Journal of Object Technology. Vol. 24, No. 2, 2025. Licensed under Attribution 4.0 International (CC BY 4.0) http://dx.doi.org/10.5381/jot.2025.24.2.a5

development for the following reasons. First, BDD's focus on system behaviour, rather than units of code, resonates with the natural approach of testing games, mirroring the manner in which humans assess games. Second, BDD empowers both programmers and non-programmers, e.g. game artists, to be involved in the development process, and to engage in effective communication about game behaviours. This helps in reducing misunderstandings between the different stakeholders, and in discovering relevant details that programmers would otherwise decide themselves about, while the product owner should have been involved. Lastly, since scenarios are automated with tests, BDD is an aid for identifying bugs that may be missed by current manual, exploratory techniques such as play-testing.

The key contributions of this paper are as follows:

1. We provide a comprehensive *development process* that integrates BDD with game development practices. It includes the BDD phases of discovery, formulation, and automation, but also allows for game development practices like prototyping and play-testing. This way we balance artistic freedom from game development with rigorous software development.

2. We implemented the tool *UnitySpec*: an open-source extension for *Unity 3D*, based on the BDD tool *SpecFlow*. *Unity 3D* is a popular game development platform (Chia et al. 2020), a.o. because it can be used to create many different types of games, is multi-platform, and is available for free. Also, it was used in our case study (our fourth contribution). *UnitySpec* enables BDD test automation to be applied directly within *Unity 3D*. Our choice for *SpecFlow* is based on our comparison of available BDD tools.

3. We introduce a framework for categorizing game behaviours into three levels of complexity: basic behaviours, singular interactions, and complete game elements. Complex behaviours need to be described on a higher level of abstraction. Hence, our framework aids in recognizing game behaviours, and in identifying at what abstraction level the corresponding BDD scenario should be written. This helps in obtaining BDD scenarios of good quality.

4. We apply our Behaviour-Driven Game Development process, our tool UnitySpec, and our Game Behaviour Framework on a real-world case study of a game developed in a governmental institute in the security domain. We evaluate the framework by applying it on the game, and we evaluate the process and UnitySpec via interviews with the development team of the game.

We note that this paper is accompanied by an artefact (Artefact 2025), that includes UnitySpec and detailed research results for the framework and case study. UnitySpec is also available as open-source software on GitHub (UnitySpec 2025).

Related work. The state of testing in video games is investigated in (Politowski et al. 2021, 2022): there is a degree of resistance to automated testing for games, because it is seen as "a waste of time and money". Also, game development processes lack well-defined testing strategies, and a focus on the fun-factor of games. A common test strategy in game development is play-testing; in (Mirza-Babaei et al. 2016, 2020) play-testing is shown to be cost-effective. In (Kasurinen & Smolander 2014) it is analyzed how game development organizations approach the testing of their products: exploratory and usability testing are most common, while technical quality is only a second concern. Our approach is able to detect these technical issues, and hence reduce bugs and delays in development.

The tool *Cukunity* (Freitas 2013) provides *Cucumber*-like BDD support for *Unity 3D* games. *Cukunity* is run from the terminal, and does not offer integration with *Unity*. The tool has not been maintained since 2013, and can only be used to test games that run on outdated phone operating version (Android 2.x or iOS 5.x). Also, *Cukunity* provides an API with only limited support for dealing with *Unity 3D* game structures, such that it will surely lack support for current *Unity 3D* games. Moreover, *Cukunity* runs as a standalone tool, and uses Ruby as its language for test implementation. *Unity* developers are used to working with the IDE that *Unity* provides, and the language C# as scripting language. This will make adoption of *Cukunity* by *Unity* game developers unlikely. Our tool runs within *Unity 3D*, so that all *Unity 3D* game structures can be used directly, and uses C# for test implementation.

We note that above works investigate the state of testing in game development, with a specific focus, e.g. on play-testing or on providing a(n outdated) tool. In this paper, we provide a comprehensive approach: a combination of a process, a tool, and a game behaviour classification framework, for supporting and easing the application of BDD in game development from several different angles. We include existing game development methods, such as play-testing, into our automated testing approach. This way, our approach aims at catching more bugs than in standard game development approaches, so that these bugs do not bug the users that pay for playing a game.

2. A process for Behaviour-Driven Game Development

In this section we propose a process that integrates game development with Behaviour-Driven Development (BDD). We first find out, via existing literature, what makes game development unique, and different from standard software engineering. Our process should include these unique characteristics. After that, we outline how BDD fits into agile in general, and in Test-Driven Development (TDD) specifically. We include TDD in our process because of its strong correlation with BDD, and our goal of improving testing in game development. Then, we discuss our new process that integrates all these elements. Finally, we discuss the benefits and drawbacks of our process.

2.1. Unique aspects of game development

In this subsection we investigate via literature what game development entails, and what unique traits for game development should be included in a process.

Experimentation and prototyping Game developers consider their work as a means of expression, and see developing games as a creative process. Many ideas may be generated in the beginning, for which prototypes are developed. Then, only a few prototypes are selected to be implemented in the game (Kasurinen & Smolander 2014). Implementation of prototypes is required, because prototypes are usually unpolished artefacts that should be deleted after the experimentation phase.

Also in (Lewis & Whitehead 2011), the authors conclude that the key to a good game design is the constant experimentation of new features instead of preset requirements and best practices. Additionally, from interviews with practitioners in Massive Multiplayer Online Games it was concluded that "paper-prototyping" is essential for game development (Daneva 2014). We note that experimentation fits well in the iterative way of working in an agile development approach. In practice, agile-like approaches are already used in game development (Kasurinen & Smolander 2014).

Fun-factor and user-experience via play-testing The main characteristics of a good game design are the fun and enjoyment players of the game experience (Draper 1999; Callele et al. 2006; Hooper 2017). Game developers focus on play-testing (Daneva 2014; Mirza-Babaei et al. 2020; Santos et al. 2018), since humans, and especially testers with much experience with and intrinsic knowledge about games, can much better assess this than computers (Politowski et al. 2021). Also, user-feedback from, and collaboration with, players (i.e. users) is critical for development (Daneva 2014).

Lack of attention for functionality While in traditional software engineering there is much focus on delivering features (functionality), there is little attention for functionality in game development practices (Politowski et al. 2021; Lewis & Whitehead 2011; Hooper 2017). In game development, third party engines and platforms may be used to minimize (functionality) bugs, and required effort to implement functionality in software.(Kasurinen & Smolander 2014). However, use of such platforms and other applied approaches like play-testing, are not sufficient to prevent numerous bugs in games (Murphy-Hill et al. 2014; Politowski et al. 2021).

Conclusion From literature we conclude that a process for game development should take into account the importance of the fun-factor and enjoyment of games, and allow for experimentation. The employed methods in game development to achieve this are prototyping, play-testing, and idea generation and selection. Hence, a process for game development should include these methods. An agile approach would fit well with the experimental nature of game development. Automated tests for functionality will catch bugs not found via play-testing.

2.2. Processes for Agile, TDD, and BDD

In this subsection we summarize background on the SCRUM agile approach, test-driven development, and Behaviour-Driven Development, to define a new process for game development in the next subsection.

We use SCRUM as a base for our new process, since SCRUM is the most common agile approach used in practice (Srivastava et al. 2017). SCRUM has the following three phases (Al-Saqqa et al. 2020):

1. **Define**: The general objectives are outlined and the required team, tools, and resources are indexed. Also, a backlog is created with features, often formulated as user stories.
2. **Sprints**: This phase consists of a series of sprints, executed by the team, each adding value for the user to the system. Sprints have a fixed length of 2 to 4 weeks.
3. **Release**: When the backlog features have been achieved, the implemented system is prepared for release.

The sprint phase consists of the following sub-phases:

1. **Planning** The team creates a planning from features from the backlog.
2. **Development** This phase takes the majority of the allocated time for the sprint. The team implements each of the planned features of the system. Also, the team holds daily meetings, called stand-ups, to discuss progress.
3. **Review** The team analyzes and collects feedback on its sprint output, and reviews its process and collaboration.

Since, SCRUM does not prescribe how development should be done, we study Test-Driven Development (TDD), for its method-likeness to BDD (since BDD originates from TDD (North 2006)). TDD takes the following steps for each feature (Al-Saqqa et al. 2020):

1. **Write tests:** Write automated tests for the feature.
2. **Test:** Run all test cases to check failure (because the feature has not been implemented yet).
3. **Implement:** Write the code that implements the feature.
4. **Test:** Run all test cases to check that the feature has been implemented correctly, and that there is no regression.
5. **Refactor:** Refactor to improve code structure for maintainability.
6. **Test:** Run all tests to check that there is no regression.

While TDD starts with creating tests, BDD starts with writing scenarios. A main benefit is that scenarios can be formulated to match closely with user requirements, while tests tend to be much closer to the low-level structures of the code, which may result in tests deviating from the user requirements.

BDD prescribes the following steps (Cucumber documentation 2025; Nagy & Rose 2018, 2021):

1. **Discover**: The team discusses the feature: concrete examples should be generated, and they should reach agreement on the details of what should be implemented.
2. **Formulate**: Document the examples using scenarios.
3. **Automate**: Create automated tests from the scenarios.

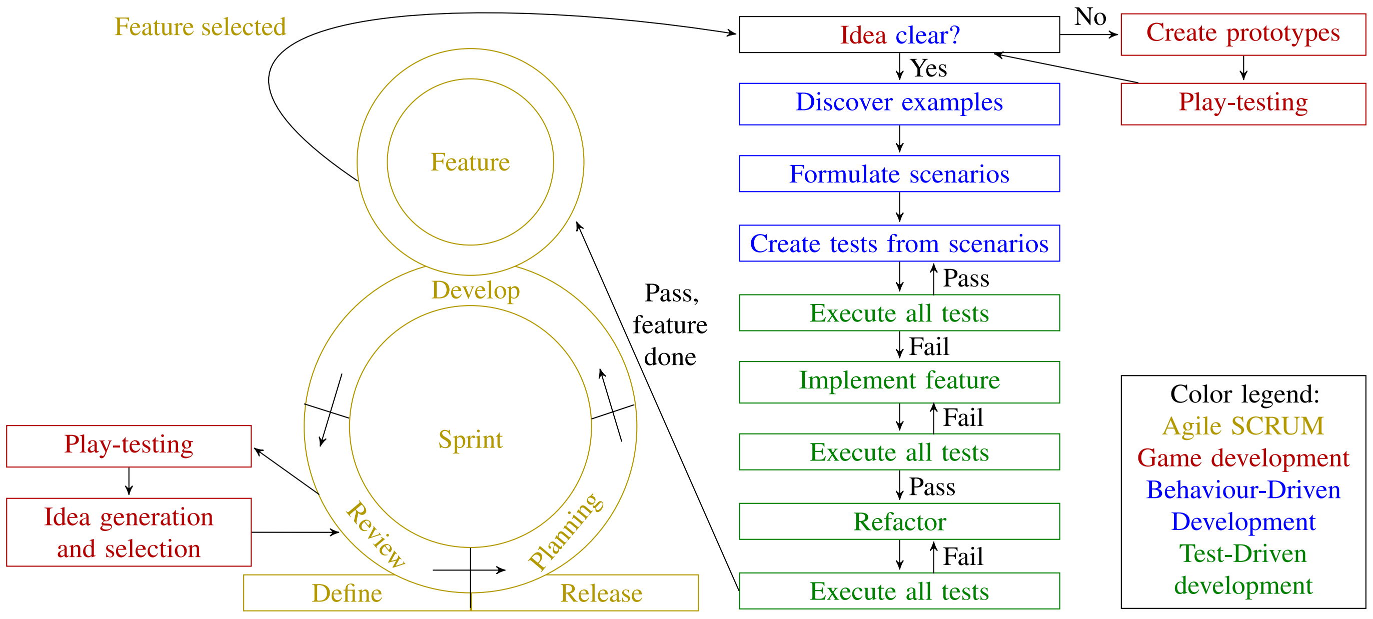

Figure 1 BDD game development process

The results of these three steps are scenarios and tests. TDD can then be continued from step 2. Listing 1 shows an example BDD scenario. We further discuss this example, and details on scenario formats and automation tools in section 3 and section 4.

The BDD philosophy (Nagy & Rose 2018) stresses the importance of collaboration of different roles within the team (Lenka et al. 2018). Specifically, three roles, also called the "Three amigos"(Nagy & Rose 2018; Cucumber documentation 2025). These three roles are (Pyshkin et al. 2012): the product owner, the tester, and the developer. The product owner is concerned with the providing value to the user through the created system, the tester is skilled in finding edge cases, and the developer can see the details of how a requirement can(not) be implemented in the system. They should all three be involved in discovery and formulation.

2.3. Process for behaviour-driven game development

Figure 1 shows the new process that we define for behaviour-driven game development. Below we explain this proces, and explain how it addresses the unique traits of game development, as discussed in subsection 2.1.

The new process includes the phases of SCRUM as a basis (in yellow), to take into account the experimental nature of game development, that requires flexibility and adaptation. The bottom circle depicts that multiple sprints are executed. In the development phase of a sprint, each feature of the sprint is implemented, according to the top circle. In the boxes on the right we elaborated what happens when implementing a feature.

In particular, upon selecting a feature from the sprint, the three amigos determine whether the idea of the feature is clear enough to be worked out into examples and scenarios. We note that this is no standard step in BDD, but we introduce this to decide whether we need game development steps (in red) to clarify the idea, or can continue with BDD (in blue) straightaway.

If the idea is not clear, we apply the game development steps (in red) of prototyping and play-testing. This way, we explicitly cater for the creative nature of game development, and its emphasis on fun and enjoyment, that is already facilitated for with prototyping in current game development processes.

Then, if the idea is clear, the BDD process (in blue) is executed: discover examples, formulate scenarios, and create tests from scenarios. Here we use BDD to ease and facilitate the creation of automated tests. In particular, the examples can be matched with the prototypes that were made (if any), and the idea for the game that this feature is about. After, the game design can be expressed in natural language BDD scenarios, such that all relevant stakeholders, including (non-technical) game designers can stay involved. Because of the shared understanding obtained from formulating the scenarios, the tester (or developer) of the game development team should be able to relatively easily create tests that match this understanding.

After creating the tests, the TDD process follows (in green), such that the feature is implemented and refactored, while tests are run to detect any introduced bugs. This ensures that the tests are kept up to date, and that the tests keep being used until the feature is done. Also, if tests needs updates, the updates are relatively little work. When the refactored implementation is done, and all tests pass, it is decided that the feature is done.

We then return to the development phase of the sprint (in yellow), where a next feature may be selected, or, at the end of the sprint, the review phase is entered. This phase includes game development steps (in red) play-testing, such that fun and enjoyment is tested each sprint. Also, to again cater for experimentation, and to specifically allow creation of ideas, we include a step for this in the review phase. From the generated ideas, a selection of viable ideas is made, so that these can be added as features to the backlog, and can be implemented in some next sprint(s).

2.4. Trade-offs of proposed process

A main argument of game developers against testing is that it costs time that would otherwise be spent on developing the game. Table 1 lists some important benefits and drawbacks of our process, and in particular of applying Behaviour-Driven Development for game development. Below, we use this table to reason about the time argument.

In the drawbacks column all elements cost time: meetings need to be held, and scenarios and test cases need to be created and maintained. However, each of the corresponding benefits also indicated where spent time may be reduced:

– The communication in meetings will help to prevent misunderstandings, that would have lead to work being redone or being unnecessary at all.
– Scenarios provide information that may aid the developer to implement a feature more quickly, than if he would have needed to find out details himself.
– When the automated tests are run, this helps in discovering bugs early, when it is cheaper and less time-consuming to solve them. Moreover, late discoveries of discrepancies between what was expected to be implemented and what has actually been implemented are prevented because of having test cases that test the agreed behaviour as described in a scenario.
– Scenarios provide documentation that captures the knowledge that would otherwise reside only in people's heads. With scenarios, anyone can read about the implemented behaviours by himself, while without any such documentation, it would involve finding the people that know how it works, and discussions with those people.
– BDD will help to correctly implement complex game situations, e.g. by expressing complex conditions or rules in a BDD scenario. For checking simple game details, e.g. the color of an object, BDD should not be used. Such simple details could already be covered by play-testing.

We note that introducing a team to BDD in general, and our new process in particular, will cost a significant time investment at the start. Time for designing and testing behaviours will be spent upfront, while only later in the process, the issues, prevented by this design and testing, would have arisen.

Also, we discuss arguments for using BDD in any development process here, while benefits and drawbacks may be very specific to e.g. the game being developed, or the experience of a team member. We think that our proposed process is especially worthwhile the time investment for complex games.

In section 5 we discuss abstraction levels for game behaviours. We think that for more abstract (and usually more complex) behaviours it is especially useful to apply the full process. For very simple features, e.g. adding a button to the user interface, steps could be skipped. The discussion of development processes in this section should provide helpful insights to make an informed decision.

Benefits	**Drawbacks**
Less misunderstanding between team members and stakeholders	Meetings to discover examples and formulate scenarios for all features
Scenarios provide guidance for implementing features	Scenarios need to be formulated
The test cases of a scenario provide proof that the scenario has been implemented as specified	Test cases need to be created
Scenarios provide documentation	Scenarios and tests need maintenance
BDD helps to deal with the complexity of games	BDD costs too much effort for simple game details

Table 1 The main benefits and drawbacks of the proposed new process of Figure 1.

3. Selecting a BDD tool for integration with Unity 3D

To select a suitable BDD tool for integration with *Unity 3D*, we compared the five tools of Table 2. These tools are open-source, support C#, and were available and maintained at the time of the research (Mulder 2024), so that the tool could be integrated in *Unity 3D*.

For 'BDD tools' we consider tools that offer some specification language (i), a binding method (ii), and test generation (iii). The specification language enables description of tests or scenarios in a format that is easily readable for non-technical experts, e.g. by using natural language. The binding method is a templating system that facilitates the transformation of the specification parts into test code. The tool generates test code automatically by applying the bindings on the specification.

As discussed in related work, *Cukunity* is not suitable for *Unity 3D* game developers, and in particular does not support C#, so that it cannot be integrated in *Unity 3D*. We note one positive point about *Cukunity*: it supports default step implementations. *UnitySpec* does not have this, hence, this is to be considered in future work.

Below, we compare the BDD tools of Table 2 on specification language, their binding method, and usability properties of tools, like documentation and user community. In particular, for specification languages, we evaluate whether game behaviours can be formulated in the language, and whether the language is something *Unity 3D* developers would be familiar with.

Concordion *Concordion* is an open-source tool for automating "Specification by Example", a way of working supported by BDD. *Concordion* enables expressing specifications without a predefined structure. It supports HTML for specification writing for C# code (while it also offers Markdown in Java). Instead of matching structured steps to definitions, *Concordion* uses annotations within the specifications themselves. This approach

Tool name	Description	Specification format	Bindings	URL
Concordion	Write executable specifications in a wiki format	HTML	Inline	https://concordion.org
FitNesse	Fully integrated standalone wiki and acceptance testing framework	Tables	Table headings	https://fitnesse.org/
Gauge	Run specifications written in Markdown	Markdown	Annotations	https://gauge.org
SpecFlow	Use annotation-based bindings from feature files to target language	Gherkin	Annotations	Available at time of research; currently end-of-life
Xunit.Gherkin. Quick	Use annotation-based bindings from feature files to target language	Gherkin	Annotations	https://github.com/ttutisani/Xunit.Gherkin.Quick

Table 2 Compared BDD tools

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 also has the potential to introduce ambiguity. While viewing is very accessible, using HTML to write scenarios may distract or even hinder users from the actual formulation of scenarios.

FitNesse FitNesse is a standalone wiki and acceptance testing framework. Its core is written in Java, but it can run tests in any language via the Fit or SLIM protocol. Its tests are expressed as tables of input data and expected output data. The table title refers to the class under test, while the column headers refer to methods on this class. Each row in the table is an example that is executed. Like Concordion, FitNesse users write 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 we expect it to be challenging to express game behaviours in a table, e.g. actions like moving the player, and picking up an object have to be formulated as data values.

SpecFlow and Xunit.Gherkin.Quick *Cucumber* (Cucumber 2025), a BDD tool for Java, has several extensions in different languages, most notably, *SpecFlow* and *Xunit.Gherkin.Quick* for C#. Like *Cucumber*, both tools support Gherkin as specification language. This language is a plain-text, structured language, which allows specifying a scenario as a sequence of a Given-, When-, and Then-step. It uses annotations with regex-like patterns to bind a method for executing a step to the natural language step itself.

Xunit.Gherkin.Quick is a lightweight framework with usage documentation on their GitHub page.

At the time of research, *SpecFlow* had a large community for support and documentation, and offered plugins for IDE integration. However, Tricentis, the company that developed *SpecFlow*, decided to end the support of tool[1]. We note however that the *Reqnroll* tool offers the *SpecFlow* functionality as a 'reboot' of the *SpecFlow* tool[2].

[1] https://support-hub.tricentis.com/open?number=NEW0001432&id=post
[2] https://reqnroll.net/

Gauge *Gauge* is an open-source test automation framework that supports C#. *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* also allows defining concepts, which are composite steps. *Gauge* does not diferentiate between Given, When, and Then steps, like in Gherkin; all steps are of the same kind. This gives more freedom in the defining steps, but may reduce precision and clarity in specifying BDD scenarios. *Gauge* links specifications to code using annotations. It can run tests in an editor or via the command-line. It shows test results in an HTML-dashboard or the command-line.

Conclusion on selection of BDD tool FitNesse uses table headings for input and output data, which make it difficult to express behaviours, so we deem this unsuitable. Further, Concordion uses HTML for expressing the specification, which may hinder writing of specifications; MarkDown would be much easier to use, because it requires use of less syntax, but is not supported in the C# version of Concordion. Also, we expect that game developers of *Unity 3D* are unfamiliar with HTML. So we also deem Concordion unsuitable.

Then *Gauge*, and *Cucumber*'s C# variants *SpecFlow* and *Xunit.Gherkin.Quick* remain. All these tools offer a specification language with steps, and lightweight annotations as binding method. *SpecFlow* and *Xunit.Gherkin.Quick* use Gherkin, so distinguish between Given, When and Then steps. We think this structure can be especially beneficial for specifying game behaviours, since these can have lots of different game events, and complex conditions and rules. In the Given-step, the condition for a game scenario or event can be expressed, while with the When-step the events or application of a game rule can be specified. By requiring the formulation of a Then-step, also game developers unfamiliar to testing, will express the expected results explicitly. This is important, since a test without any check on expected result will detect few bugs, e.g. it will maybe catch a crash of the game, but no violations of game rules. At the time of research (Mulder 2024), we preferred for *SpecFlow* over *Xunit.Gherkin.Quick*, because the community support and IDE integration of *SpecFlow* was helpful for integration in *Unity 3D*. Because we copied *SpecFlow* into our tool (see subsection 4.4 for details), *SpecFlow*'s end of life has no effect on the functioning of our tool.

4. UnitySpec: Behaviour-Driven Development tooling for Unity3D

This section introduces *UnitySpec*, our open source tool that supports Behaviour-Driven Development in *Unity 3D*. *UnitySpec* builds upon the foundation of *SpecFlow*. Next, we first introduce both *Unity 3D* and *SpecFlow* in more detail. Then we describe the workflow of our tool *UnitySpec*, we discuss its design, and the obstacles we overcame in its implementation. Lastly, we describe a validation of *UnitySpec* with two sample BDD scenarios. We apply the tool in the case study of section 6.

We note that the artefact (Artefact 2025) includes instructions to install and run *UnitySpec*, and contains the two samples discussed in subsection 4.6. Moreover, it includes a video showing the test execution of the scenario from Listing 1. *UnitySpec* is also available as open-source tool on GitHub (UnitySpec 2025).

4.1. Introduction to Unity 3D

This introduction is a summary from online resources (Unity 2025; Conceptartempire.com 2025; Juegostudio.com 2025).

Unity 3D is a versatile platform for creating interactive, real-time 3D content, running on multiple operating systems and platforms. *Unity* is not only used for video games but also for a wide range of applications including simulations, virtual reality, augmented reality, film, architecture, and education.

Unity 3D is a game engine, offering a robust environment for creating and manipulating 3D graphics, physics, sound, and other multimedia elements. *Unity* has a flexible editor that 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.

The scripting in *Unity 3D* is primarily done using C#. *Unity*'s documentation, tutorials, and community further support its user base, providing resources for learning and problem-solving. Developers can exchange assets, plugins and tools for *Unity*, allowing for rapid prototyping and production.

4.2. Introduction to SpecFlow

In *SpecFlow*, BDD scenarios are formulated in natural language, structured according to the Given-Then-When style as used in the Gherkin language. The Given step prescribes a context of, or condition on the system. It may be omitted if the system can show the behaviour of the scenario from any possible context. If multiple conditions need to be satisfied, these can be combined with "And". The When step consist of one or more action that prescribe the behaviour of the scenario. If multiple actions are described, the action sentences are connected with "And", which denotes that the actions happen in sequence. The Then step prescribes the outcome or result after the When actions. If there are multiple outcomes, they can all be stated using link word "And". If actions and/or outcomes comprise of combinations of specific values, these can be given in an accompanying table.

```
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     |
```

Listing 1 Example feature file with a BDD scenario about the behaviour of moving the player

Listing 1 shows an example BDD scenario. Besides the Scenario itself, in Given-When-Then format, it also gives the scenario a name: "Basic movement". The scenario is called "Scenario outline" because it has parameters `<key>` and `<direction>`, which can have any of the value-pairs from the table listed for "Scenarios". Listing 1 also contains the description of the feature the scenario was formulated for, in terms of a feature name, and a user story. User stories are often formulated in the following format: As a [user], I want to [goal], in order to [reason], where a specific user, goal and reason are given. The [reason]-part of the sentence is optional. A feature file consists of a feature, and one or more scenarios.

Given some feature file, *SpecFlow* can generate bindings: place holder code for each step of a BDD scenario of that file. The user of the tool needs to replace the place holder code by code that executes the system in accordance with the step. The bindings are also called step definitions. *SpecFlow* can automatically combine bindings into a test case that executes a particular BDD scenario from the feature file.

4.3. Workflow in UnitySpec

A user can add *UnitySpec* to an existing *Unity* project through the *Unity* Package Manager. A user of *UnitySpec* typically takes the following steps:

1. **Write scenarios:** the user first creates a new feature file via the *Unity* menu. Then the user writes BDD scenarios in this feature file, using the Gherkin language.

2. **Generate test files:** the user presses the generation button in the *UnitySpec* window. This triggers generation of files with test cases. Such a test case consists of calls to step definitions, and has tags for referring back to the steps from the feature file. Also files with method headers for step definitions are generated.

3. **Write step definitions:** Because the generated step definition methods do not have an implementation, the user needs to add the code for executing a step. Also, the user may change the return type of the step definition.

4. **Run tests:** the user invokes the running of tests using *Unity*'s own test runner. *UnitySpec* then outputs which scenario steps have been executed and which corresponding step definition methods has been called.

Note that above steps match with the steps "Formulate scenarios", "Create tests from scenarios" (in two steps), and "Run all tests" from Figure 1.

4.4. Design of UnitySpec

UnitySpec is based on the last alpha-release of *SpecFlow*, at the time of creating *UnitySpec*, namely *SpecFlow* version 3.9.74, which was available on GitHub at that time.

Since *Unity 3D* does not provide testing capabilities by itself, we use the *Unity Test Framework* in *Unity 3D*. The *Unity Test Framework* is a tool within the *Unity 3D* ecosystem that is built upon the *NUnit* library, a well-known open-source unit testing library for .NET languages. The framework facilitates testing in both Play Mode and Edit Mode. 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. With this framework, the user of *UnitySpec* can implement the step definitions.

Currently, UnitySpec has been developed for Windows. In future work we would like to make it multi-platform, like *Unity*.

UnitySpec can be run in *Unity*, because it is a *Unity* package, i.e. a plugin that can be loaded in *Unity 3D*. To put *SpecFlow* in such a package, we adjusted *SpecFlow*'s project structure, dependencies and code. Therefore, we could not design *UnitySpec* as a fork of *SpecFlow*, but instead needed to copy *SpecFlow*'s files into the *Unity* package and adjust them accordingly.

Unity 3D requires a meta file for each of *SpecFlow*'s files that we included. We therefore created C# libraries of the part of *SpecFlow*'s code that did not need to depend on *Unity*. We could then import these libraries in the *Unity* package, and this way avoid creating meta files for those library files.

UnitySpec consists of the following components:

– The *UnitySpecWindow* includes code for supporting *.feature* files, for defining user file settings, and for the custom window with the generation button. The code is included directly in the *Unity* package of *UnitySpec*.
– The *Generator* library implements the generation of the test files. It parses the feature files, and generates all test files. The generator is set up as a library: it is compiled separately from *Unity*, and included as a *.dll* file.
– The *Runner* contains code for running the test cases. The generated test case files call the Runner to match steps to their definitions for executing these definitions. The Runner also generates the output seen in the *Unity* test execution window. Since the Runner depends on *Unity* for logging and asserting, it is included in the package.
– The *General* component contains utility code shared between Runner and Generator. It is included as a library.

We remark that the majority of *UnitySpec*'s code has not been written by ourselves, because most code originates from *SpecFlow*. Our contribution in creating *UnitySpec* lies prominently in organizing, configuring, and adapting *SpecFlow*'s code, such that it functions as a *Unity* package in *Unity 3D*.

4.5. Conquered obstacles in implementing UnitySpec

We solved three main obstacles for creating *UnitySpec*:

1. *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, we adjusted the project discovery and file generation. This also guided the decision to create a new project instead of forking *SpecFlow*.

2. The *Unity Testing Framework* works differently from regular `C#` testing frameworks. *SpecFlow* expects all methods to have a return type of *void* or *Task*. However, the *Unity Testing Framework* 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 be able to have this return type. To allow for this, both the *Generator* and *Runner* had to be adjusted.

3. *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. This change required some minor rewrites.

4.6. Samples for UnitySpec

We tested *UnitySpec* in *Unity 2022.3* for both the .Net Standard 2.1 and .Net Framework API. In particular, we executed *UnitySpec* on two sample feature files: the sample from Listing 2, which is a classic BDD example about addition, and the sample from Listing 1 describing player movement. We took the workflow steps of subsection 4.3.

From the addition sample we learned that we can successfully create feature files, and press the generated button, such that *UnitySpec* parses the feature files and generates test files in readable layout. In particular, there is a test case for the Addition scenario, that compiles successfully. When running the test case, correct test execution output is obtained. This indicates that *UnitySpec* was able to correctly bind the test cases with step definitions, and that the output has the method calls and corresponding steps from the scenario.

With the player movement sample we test two special capabilities of *UnitySpec*: the usage of a scenario outline, and a step definition with return type `IEnumerator`. We learn that this scenario outline is successfully parsed and results in step definitions with the expected parameters. Test execution, with the `IEnumerator` step definitions, is successful, and the output has the expected values from the table inserted in the scenario steps and method calls.

```
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
```

Listing 2 Classic BDD scenario

Some relevant implementation details for the scenario of Listing 1 in *UnitySpec* are as follows. In the `Update` function that executes code for the game every frame (the game loop), we use a custom input control object, such that either the implementation of the When step can provide a key press, or a normal keyboard. Checks for Given or Then steps can be performed by inspecting properties of relevant game objects, e.g. with `GameObject.Find("Player")` the game object of the player is retrieved such that its position, and hence its relative movement can be determined. In Listing 3 we provide an excerpt of the implementation of a step definition for the scenario of Listing 1 in *UnitySpec*.

5. Game Behaviour Framework

To integrate BDD practices into game development, a first step is to find out how BDD scenarios can be used for describing game behaviours. We note that the quality of BDD scenarios, written for systems in any application domain, are always dependent on the practitioner writing the scenarios (Nagy & Rose 2021). However, one generic key element for good quality BDD scenarios is the principle of conservation of proper abstraction (Binamungu et al. 2020; Bezsmertnyi et al. 2021). Here a golden mean between too abstract and too concrete has to be found. Our game behaviour framework aims to support in finding the right abstraction level of an identified game behaviour. Therefore, we created a categorization of behavioural patterns in games.

In the process of Figure 1, this categorization may help to discover relevant examples of game behaviours, and will especially help to formulate scenarios of good quality, which may significantly impact the quality of the implementation and tests developed in the next steps.

We took the following steps:

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

In the first step, we collected game behaviours from projects used in Unity's "Create with Code" course (Unity 3D course: Create with Code 2025). We chose this course 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.

The result of the first step was a database of collected behaviours, which is provided in the artefact (Artefact 2025).

In the second step, we applied an informal form of grounded theory (Strauss & Corbin 1994). This means that, first, we created abstract behaviours (concepts) from our database of behaviours. Then, we identified categories of these abstract behaviours. The result was that we identified three categories, which we call *levels*. We present these levels in subsection 5.1, each with a list of abstract behaviours. We note that these lists are non-exhaustive: analysing more games may lead to more abstract behaviours. We expect that these behaviours can be placed in one of the (generic) levels. We note that it is impossible to identify the behaviours of all existing games. In subsection 5.2 we discuss how to determine what the level is of a (newly identified) game behaviour.

In the third step, we applied our framework on an example game, and formulated one BDD scenario for each of the levels. This way we do a first sanity check that it is possible to formulate BDD scenarios for the identified game behaviours. We discuss the behaviours for the example game in subsection 5.3. Note also that in section 6, we apply our framework on the case study, to evaluate its applicability.

5.1. Levels of game behaviours

We discuss the three identified levels of game behaviours below.

Level 1: Basic controls The first level contains very basic behaviours. These behaviours concern only one object or asset, or span only one frame. We give examples of abstract level 1 behaviours in Table 3. The table shows e.g. *Player movement*, which means that one object is moved. With *Independent behaviour*, it is described that an object, not controlled by the player, performs a level 1 behaviour, e.g. moving. Furthermore, when loading a scene or playing a sound happens in a single frame, it is a level 1 behaviour.

Level 2: Singular interactions The second level contains singular interactions: 2 or more objects, or an object and event interact. This interaction takes place or can be evaluated in a single moment. Table 4 gives examples of abstract level 2 behaviours. The table includes various forms of triggers, where something happens due to some event, e.g. an object moving. An interaction between two objects can be e.g. collision.

Level 3: Game elements Level 3 describes game elements which span multiple frames and concern multiple objects. The behaviours in this level are built upon multiple behaviours from any level, so behaviours from level 1 or 2 and even behaviours from level 3. For example, the spawning of a Non-Player Character (NPC) may lead to more spawning of NPCs, when an NPC reaches some location. Then, spawning may continue this way, unless the player prevents this, or until some game over condition is satisfied. See Table 5 for some more examples of abstract level 3 behaviours.

```
[Given(@"I load the level ""(.*)""")]
public IEnumerator GivenILoadTheLevel(string level) {
  var scene = AssetDatabase.GUIDToAssetPath(AssetDatabase.FindAssets(level).First());
  EditorSceneManager.LoadSceneInPlayMode(scene, new LoadSceneParameters(LoadSceneMode.Single));
```

Listing 3 Excerpt of implementation of step definition in *UnitySpec*

Abstract Behaviour	Explanation
Player Movement	Moving a player object using keys
Camera Movement	Moving the camera using user input
Scene Loading	Successfully load a scene
Assets Exist	Check that assets are in *scene* at boot
Booting	Starting application
In-menu Navigation	Menu interactions that stay in scene
Out-menu Navigation	Menu interactions that lead out of the current scene
Independent Behaviour L1	Object constantly shows a level 1 behaviour without input
Persistent storage	Something is stored and persists in another level
Sound	A sound plays
Spawning	A new object is created

Table 3 Examples of abstract level 1 game behaviours

Level 3 is a very broad category: any behaviours above level 2 fits here. One could conceive more levels to further divide the level 3 behaviours. The highest level would be to play the game from start to finish. However, it is unclear how many levels would be needed, since abstractions can be nested endlessly in games. For a specific game it might be possible to discover additional levels, but with this framework we aimed to keep it generic for all games, so we stop at level 3.

5.2. Categorizing game behaviours

To apply the Game Behaviour Framework on a new game behaviour, we recommend asking two questions on interactions, as shown in Figure 2. We focus on interactions, because the interactions of a behaviour are the key for identifying its level. If that results in doubts, one can also look at the tables with examples, to see if there is a similar abstract behaviour. When still in doubt, one can inspect our database of concrete behaviours (Artefact 2025), to search for similar examples.

5.3. Levels for behaviours of example game

In this section, we identify behaviours and their levels for an example game that has a basic fight mechanic. This imaginary game has a player, a shooting mechanic, and some enemy. Suppose that the player needs to shoot (and hit) the enemy two times to "kill" it. The following BDD scenario formulates this:

Abstract Behaviour	Explanation
Independent Behaviour L2	Object constantly shows a level 2 behaviour without input
Special Control	A user interaction trigger
Colliding	Two or more objects collide
Simple Condition	A level 1 behaviour depends on current state
Throwing/shooting	A user interaction spawns object that moves in a direction
Counter	A counter is updated or read
Collision Trigger	Something happens when objects collide
Movement Trigger	Something happens when an object moves
Location-based Trigger	Something happens when an object reaches a location
Time-based Trigger	Something happens due to an internal timer
View-based Trigger	Something happens due to what is in view

Table 4 Examples of abstract level 2 game behaviours

"Given an enemy in range of the player, When the player shoots the enemy two times, Then the enemy dies". Clearly, there is more than 1 interaction, so it is a level 3 behaviour.

The previous scenario provides no specification of how the shooting mechanic works. Let's imagine our game is a first-person shooter. We could now describe the shooting behaviour with the following BDD scenario: "When the left mouse button is clicked, Then a bullet is spawned from the player, And moves in the view direction". Table 4 has a level 2 abstract shooting behaviour which matches with this concrete scenario.

The shooting scenario abstracts from how the view direction is changed. We specify this as the following BDD scenario: "When the mouse is moved left 1cm Then the view is moved left 10 degrees". This behaviour corresponds with the abstract level 1 behaviour "Camera Movement" from Table 3.

6. Case study

We first introduce the subject of our case study: the serious game *Virtual Brigade*. Then, we apply our game behaviour

Abstract Behaviour	Explanation
Independent Behaviour L3	Object constantly shows a level 3 behaviour without input
Fight	A fight between the player and enemy
NPC Creation	Condition triggers asset creation at a specific location
Powerup	Pickup item which then has effect
Extended Condition	A behaviour depends on factors beyond Simple Condition

Table 5 Examples of abstract level 3 game behaviours

framework on the Virtual Brigade. Lastly, we evaluate our BDD game development process and *UnitySpec*, by having the development team of the game implement a feature, by following the steps of the process, and by using *UnitySpec*. We note that with the case study we performed an evaluation only; we did not adapt our framework, process or tool (obtained from our research) based on evaluation results.

6.1. Virtual Brigade

The serious game Virtual Brigade facilitates training of governmental security professionals. The game includes various scenarios, e.g. on border control, and for securing important events. Virtual Brigade is a configurable game. Educational instructors create trainings in the *Unity* Editor, by selecting predefined locations, objects, non-player characters, events, and dialogues. The resulting serious game is then used as a training for security professionals that need to perform their job in a certain situation, or execute a certain task.

6.2. Applying the game behaviour framework

For applying the framework, we used the roadmap with requirements that was used for developing the Virtual Brigade. We first classified the requirements with our framework, and then reflected on its applicability with these results.

Classifying requirements Table 6 shows a selection of requirements, from various roadmap categories, and their respective levels. With the requirements in the table we illustrate their

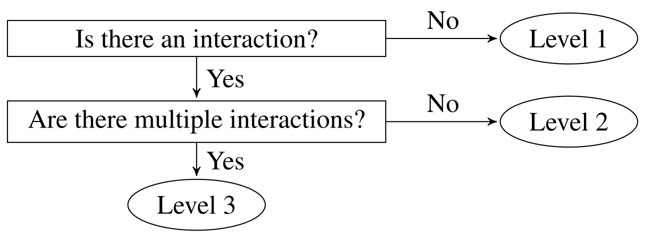

Figure 2 Decision diagram to decide what level a game behaviour belongs in

variety. The classification of the complete list of requirements is included in the artefact (Artefact 2025).

The first requirement describes the behaviour of loading an airport scene, and checking if key objects, such as an immigration desk, are present. Both are Level 1 behaviours, for abstract behaviours Booting, and Asset Exists respectively.

The second requirement is a singular interaction between the player and the immigration desk, which cause something to happen, i.e. taking place. Thus this is a level 2 behaviour for abstract behaviour Special Control.

The third requirement consists of three separate behaviours: (i) the player picks up objects, (ii) the player places picked-up objects into his inventory, and (iii) the player moves items from his inventory to his hands. Each of these behaviours on their own could be classified as a level 2 behaviour (Special Interaction). Behaviour (iii) could be more complicated, but in this case, we believe it to mean moving items from inventory back to the hands of the player.

We classified the fourth requirement as a level 3 behaviour (Independent Behaviour), since 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, and when it is in front of the desk, the person walks towards the player”. This sequence of actions motivates our choice for level 3.

The last requirement is about an appealing look. It is no behaviour, so our framework is not applicable. A human is better at assessing appeal than a computer, so this should not be captured with an automated test (see subsection 2.1).

Reflection We were able to classify all 42 requirements from the roadmap (Artefact 2025), except the mentioned appeal requirement, and another one on ease of use. Based on the diversity of the requirements, we conclude that the framework is rather flexible. While some requirements, on player movement and interactive objects, clearly described game behaviours, this was less obvious for requirements on loading visual worlds, or the user interface.

The relatively simple questions of Figure 2 for classifying behaviours in one of the levels helped to identify the levels, also for the more complex and dynamic behaviours of the requirements on player movement and interactive objects.

Although many behaviours did not fit neatly into the predefined 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. This also shows an area of improvement for the framework: having more abstract behaviours would ease categorization. Examples could be obtained from additional case studies.

6.3. Applying our process and UnitySpec

To evaluate our BDD game development process, and our tool *UnitySpec*, we have employed the help of the development team of Virtual Brigade. We took the following steps:

1. Identify the three amigos among the team members;

Category	Requirement	Level
Visual worlds	1. The visual world for Schiphol exists and contains key objects	Level 1: Asset Exists
Player movement	2. The player can enter an immigration desk and take place.	Level 2: Special Control
Player movement	3. The player can pick up objects, place them in their “bag”, and later use them	Level 3
Interactive objects	4. People can queue and walk towards the player one by one	Level 3
User interface	5. There is a visually appealing menu structure based on task fields	Not applicable

Table 6 Selected requirements and their levels

2. Select a feature for applying our new process;
3. Explain the process and tool to the team members;
4. Support and observe the team in creating the feature according to the process, using *UnitySpec*;
5. Interview the team members to obtain feedback

We identified the three amigos based on the availability and preferences of the team. These three amigos had experience with game development through developing Virtual Brigade. They used agile as a way of working, but did not apply BDD before. For testing, they only applied some play-testing.

Then we selected an available feature from the backlog at the time of the experiment: add a tutorial to the Virtual Brigade. After an introduction meeting (step 3), we followed our process for a feature from Discovery, since the idea for the tutorial was deemed clear enough. We supported the three amigos in formulating scenarios, since they had no prior experience with BDD. Then we held our first set of interviews to collect feedback from the three amigos, see Table 7. Next, we explained *UnitySpec*, and gave the team time to implement the feature, and to use *UnitySpec*. When the team indicated the feature was done, we held our second set of interviews, see Table 8.

Interview results Below we discuss the main conclusions from the interviews. To obtain these, we transcribed the interviews, summarized them, and then extracted below conclusions. The interview summaries are provided in (Artefact 2025).

The three amigos indicated that the new process, despite some initial challenges, offered considerable benefits. All three 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 implementation. The tester also saw potential, especially for larger organizations with external stakeholders, though he noted challenges in their current context where features 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 to systematically check during implementation that each part of

Nr	Aim for feedback
1.1	How they felt about the method thus far
1.2	If they felt they were able to express their expectation of the feature
1.3	If they felt that the discussion was useful
1.4	How they felt about creating the scenarios
1.5	If the method had any impact on their confidence in the successful implementation of the feature
1.6	What their expectation was going forward

Table 7 Aims for feedback to be gathered in interviews after scenario formulation

Nr	Aim for feedback
2.1	How the developer felt about the feature and its alignment with the expectations of the other team members
2.2	What the effect of the method was during development
2.3	How the team members experienced the method
2.4	If a positive/negative impact was felt due to the method
2.5	If they would use the method again
2.6	What the experience was with using *UnitySpec*

Table 8 Aims for feedback to be gathered in interviews after feature implementation

the feature was completed as intended.

Despite recognizing the initial time cost, all participants saw value in the process for its potential to save time and improve accuracy in the long term. The product owner and developer expressed a willingness to use the process again, particularly for new products or in larger teams where its benefits would be more pronounced. However, the tester highlighted challenges in testing: step definitions could not successfully be implemented, because the Virtual Brigade had not been set up with enough support for automated testing. There were no issues encountered with *UnitySpec* itself; the tool fulfils its goal and effectively supports BDD in game development. Overall, the team acknowledges the process' value in enhancing efficiency and reducing errors, suggesting that with further practice and adaptation, it could become an integral part of their development process.

Learned lessons Based on our experience in this case study, we have two recommendations for future case studies:

– Adopt the BDD game development process 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 BDD scenarios can be automated.
– Invest ample time in onboarding team members in the new process. This prevents any initial confusions due to ambiguity, and ensures a clear understanding of expectations and process, such that time is saved later on.

Threats to validity The development team only applied our process for one feature, so we did not evaluate whether the process will fulfill the team's long term expectations that it will save time and improve accuracy. Further, above learned lessons could be different, i.e. more specific to game development and our process, for a team that already has sufficient experience with BDD and testing. Also, the game was a serious game, and no entertainment game, so results for the latter may deviate.

7. Conclusions

In this paper we showed how Behaviour-Driven Development can be integrated into game development. We proposed a new process for behaviour-driven game development, we implemented the tool UnitySpec for BDD support within game development platform Unity, and we provided a framework to identify and classify game behaviours for BDD scenarios. We showed applicability of these contributions on a real-world case study with serious game Virtual Brigade.

For future work, we are interested in doing more case studies. In particular, we like to investigate the maintenance challenges, when BDD is applied over a longer period of time. Also, we are interested in whether our process can be customized to different situations, with respect to e.g. available time, experience of developers, and complexity of the feature. Additionally, we note that a serious game may be different than an entertainment game, and thus a case study on an entertainment game could yield different results. Also, we are interested in larger games, and whether the abstraction levels of our game behaviour framework still apply. Finally, we would also like to validate the levels and corresponding abstract behaviours of our game behaviour framework on a larger set of games.

Acknowledgments

This paper is based on the master thesis of Michael Mulder (Mulder 2024).

References

Al-Saqqa, S., Sawalha, S., & Abdelnabi, H. (2020). Agile software development: Methodologies and trends. *International Journal of Interactive Mobile Technologies*, *14*, 246-270. doi: 10.3991/IJIM.V14I11.13269

Artefact of this paper. (2025). Retrieved from https://doi.org/10.5281/zenodo.15187759

Ballhaus, W., Chow, W., & Rivet, E. (2022). *Perspectives from the global entertainment & media outlook 2022–2026: Fault lines and fractures: Innovation and growth in a new competitive landscape* (Tech. Rep.). PwC.

Bezsmertnyi, O., Golian, N., Golian, V., & Afanasieva, I. (2021, 10). 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*, 217-220. doi: 10.1109/PICST51311.2020.9467891

Binamungu, L. P., Embury, S. M., & Konstantinou, N. (2020). Characterising the quality of behaviour driven development specifications. *Lecture Notes in Business Information Processing*, *383*, 87-102. doi: 10.1007/978-3-030-49392-9_6

Binamungu, L. P., & Maro, S. (2023, 9). Behaviour driven development: A systematic mapping study. *Journal of Systems and Software*, *203*, 111749. doi: 10.1016/J.JSS.2023.111749

Callele, D., Neufeld, E., & Schneider, K. A. (2006). Emotional requirements in video games. In *14th IEEE international conference on requirements engineering (RE 2006), 11-15 september 2006, minneapolis/st.paul, minnesota, USA* (pp. 292–295). IEEE Computer Society. Retrieved from https://doi.org/10.1109/RE.2006.19 doi: 10.1109/RE.2006.19

Chia, A., Keogh, B., Leorke, D., & Nicoll, B. (2020). Platformisation in game development. *Internet Policy Rev.*, *9*(4). Retrieved from https://doi.org/10.14763/2020.4.1515 doi: 10.14763/2020.4.1515

Chueca, J., Verón, J., Font, J., Pérez, F., & Cetina, C. (2024, 1). The consolidation of game software engineering: A systematic literature review of software engineering for industry-scale computer games. *Information and Software Technology*, *165*, 107330. doi: 10.1016/J.INFSOF.2023.107330

Conceptartempire.com. (2025). *What is Unity 3D & what is it used for?* Retrieved from https://conceptartempire.com/what-is-unity/

Cucumber - BDD testing & collaboration tools for teams. (2025). Retrieved from https://cucumber.io/

Cucumber documentation - Behaviour-Driven Development. (2025). Retrieved from https://cucumber.io/docs/bdd/

Daneva, M. (2014, 9). 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*, 3-12. doi: 10.1109/RE.2014.6912242

Draper, S. W. (1999). Analysing fun as a candidate software requirement. *Pers. Ubiquitous Comput.*, *3*(3), 117–122. doi: 10.1007/BF01305336

Farooq, M. S., Omer, U., Ramzan, A., Rasheed, M. A., & Atal, Z. (2023). Behavior driven development: A systematic literature review. *IEEE Access*.

Freitas, M. (2013). *Cukunity : Cucumber for Unity.* Retrieved from https://github.com/imkira/cukunity

Gauge. (2025). Retrieved from https://gauge.org/index.html (https://gauge.org/index.html)

Helplama.com. (2023). *Game industry usage and revenue statistics 2023.* Retrieved from https://helplama.com/game-industry-usage-revenue-statistics/

Hooper, S. (2017). *Automated testing and validation of computer graphics implementations for cross-platform game development* (Unpublished doctoral dissertation). Auckland University of Technology.

Juegostudio.com. (2025). *Unity 3D: A comprehensive guide to Unity's features and uses.* Retrieved from https://www.juegostudio.com/blog/what-is-unity-3d-a-comprehensive-guide-to-unitys-features-and-uses

Kasurinen, J., Maglyas, A., & Smolander, K. (2014). 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. doi: 10.1007/978-3-319-05843-6_1

Kasurinen, J., & Smolander, K. (2014, 9). What do game developers test in their products? *International Symposium on Empirical Software Engineering and Measurement*. doi: 10.1145/2652524.2652525

Lenka, R. K., Kumar, S., & Mamgain, S. (2018, 10). Behavior driven development: Tools and challenges. *Proceedings - IEEE 2018 International Conference on Advances in Computing, Communication Control and Networking, ICACCCN 2018*, 1032-1037. doi: 10.1109/ICACCCN.2018.8748595

Lewis, C., & Whitehead, J. (2011). The whats and the whys of games and software engineering. In *Proceedings of the 1st international workshop on games and software engineering* (pp. 1–4).

Mirza-Babaei, P., Moosajee, N., & Drenikow, B. (2016). Playtesting for indie studios. *Proceedings of the 20th International Academic Mindtrek Conference*. doi: 10.1145/2994310.2994364

Mirza-Babaei, P., Stahlke, S., Wallner, G., & Nova, A. (2020, 4). A postmortem on playtesting: Exploring the impact of playtesting on the critical reception of video games. *Conference on Human Factors in Computing Systems - Proceedings*. doi: 10.1145/3313831.3376831

Mulder, M. (2024, June). *From behaviours to code : Exploring behaviour-driven development in unity 3d game creation.* Retrieved from http://essay.utwente.nl/100096/

Murphy-Hill, E., Zimmermann, T., & Nagappan, N. (2014). 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.* doi: 10.1145/2568225.2568226

Nagy, G., & Rose, S. (2018). The BDD books - Discovery - Explore behaviour using examples. Leanpub. Retrieved from https://leanpub.com/bddbooks-discovery

Nagy, G., & Rose, S. (2021). The BDD books - Formulation - Document examples with Given/When/Then. Leanpub. Retrieved from https://leanpub.com/bddbooks-formulation

North, D. (2006). Introducing BDD. *Better Software Magazine*.

Politowski, C., Guéhéneuc, Y.-G., & Petrillo, F. (2022). 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* (pp. 37–43).

Politowski, C., Petrillo, F., & Guéhéneuc, Y.-G. (2021). A survey of video game testing. In *2021 ieee/acm international conference on automation of software test (ast)* (pp. 90–99).

Pyshkin, E., Mozgovoy, M., & Glukhikh, M. (2012). On requirements for acceptance testing automation tools in behavior driven software development. *Proceedings of the 8th Software Engineering Conference in Russia (CEE-SECR)*.

Santos, R. E., Magalhes, C. V., Capretz, L. F., Correia-Neto, J. S., Silva, F. Q. D., & Saher, A. (2018, 10). 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*. doi: 10.1145/3239235.3268923

Srivastava, A., Bhardwaj, S., & Saraswat, S. (2017, 12). Scrum model for agile methodology. *Proceeding - IEEE International Conference on Computing, Communication and Automation, ICCCA 2017*, *2017-January*, 864-869. doi: 10.1109/CCAA.2017.8229928

Strauss, A., & Corbin, J. (1994). Grounded theory methodology: An overview. In N. K. Denzin & Y. S. Lincoln (Eds.), *Handbook of qualitative research* (p. pp. 273–285). Sage Publications, Inc.

Unity. (2025). *What is Unity? - Unity learn.* Retrieved from https://learn.unity.com/tutorial/what-is-unity

Unity 3D course: Create with code. (2025). Retrieved from https://learn.unity.com/course/create-with-code (https://learn.unity.com/course/create-with-code)

Unityspec github page. (2025). Retrieved from https://github.com/UnitySpec (https://github.com/UnitySpec)

About the authors

Michael Mulder graduated from the master program in Computer Science at the University of Twente (the Netherlands).

Petra van den Bos is assistant professor at the University of Twente (the Netherlands) and performs research on software testing. You can contact the author at p.vandenbos@utwente.nl.