UCLouvain

École polytechnique de Louvain

TDGD
Test-Driven Game Development

Author: Chiara Emanuela ZARRELLA
Supervisors: Prof. Eric PIETTE, Prof. Kim MENS
Academic year: 2024/2025

Contents

1

Introduction

Introduction

Since the earliest days of human civilization, games have played a fundamental role in culture and society. They combine reasoning, strategy, and enjoyment: attributes deeply intrinsic to human nature. With the advent of computing and the rapid evolution of programming languages, games have also become a significant subject in various research domains.

One of the most prominent areas is General Game AI. This field focuses on the analysis, generation, and design of games through artificial agents [1, 2]. These agents, referred to as General Game Playing (GGP) agents [3], are designed to play multiple games by understanding their rules rather than being hardcoded for specific ones.

Over time, these agents have been incorporated into broader frameworks known as General Game Systems (GGS) [4], platforms capable not only of playing but also of supporting game design, teaching, and experimentation with new games. Among the most notable of these systems is Ludii [5], the platform that forms the basis of this thesis.

At present, Ludii includes 1,409 games[1], including reconstructions of ancient and traditional games, modern games, and experimental games. This rich library makes Ludii particularly appealing to historians and scholars of cultural heritage interested in understanding or preserving historical gameplay.

Ludii is distinguished by its unique Game Description Language (GDL), built upon the concept of ludemes [6]: atomic, human-readable units of game-related information. By combining ludemes hierarchically, users can describe a wide variety of games concisely and in a manner close to natural language. Such ludeme-based approach makes Ludii accessible not only to computer scientists, but also to game designers, historians, and researchers in the humanities, many of whom may not possess a background in computer programming but are still able to meaningfully contribute by encoding games.

To support this accessibility, Ludii offers a Graphical User Interface (GUI) that includes an integrated game editor and a visual game board, along with a collection of AI agents [7]. These features allow users to develop, play, and evaluate games directly within the

[1] github.com/Ludeme/Ludii

1

platform, either against artificial opponents or by observing automated matches between different agents.
Despite the accessibility offered by Ludii's ludeme-based language, developing games within the platform remains a non-trivial task, especially for users with no formal background in computer science. While ludemes simplify the writing of game descriptions, users still face challenges: Ludii's GDL has its own syntactic and semantic rules, which must be learned and respected.
Errors in rule encoding, misinterpretation of game mechanics, or subtle inconsistencies in ludeme usage can easily result in flawed implementations. Although the system includes a syntax validator, semantic errors frequently go unnoticed. A game may compile without errors but fail to play correctly. This places a considerable burden on the developer, who must engage in time-consuming manual debugging.
In traditional software development, this problem is mitigated by testing frameworks, which allow developers to verify system behavior through automated tests. Unfortunately, Ludii currently lacks such a framework for its GDL, leaving developers without structured tools to validate their implementations during development.
This thesis explores whether the established methodology of TDD can be adapted to the domain of general game development. We refer to this adaptation as Test-Driven Game Development (TDGD).
TDD [8] is based on writing a test before implementing the corresponding feature or functionality. The process follows a simple cycle:

1. Write a failing test that captures the intended behavior.

2. Write the minimum code necessary to pass the test.

3. Refactor the implementation and the test to improve clarity and maintainability.

This methodology aligns naturally with the incremental and modular nature of game development. In Ludii, a game is generally structured in terms of major components such as Players, Equipment, and Rules, each of which is represented by specific ludemes. A Ludii developer typically starts by defining the players, then proceeds to the equipment, and finally encodes the rules.
A testing tool that supports and encourages this iterative process (validating each component as it is introduced) would represent a significant improvement in Ludii's development ecosystem. Such a tool could not only reduce implementation errors but also serve as a learning companion, helping new users understand Ludii's GDL through well-defined and testable examples.
The goal of this work is to provide a dashboard extension to the existing Ludii GUI, along with an integrated database of tests, to support Ludii developers in adopting a TDGD approach.
The dashboard is designed to allow developers to run tests multiple times across different stages of development, enabling early validation and iterative refinement of their work. Additionally, the system will provide meaningful feedback when tests fail, helping users identify and resolve errors in their game definitions effectively. The supported tests span different categories, including those verifying semantic rules and game-specific behaviors.

The ultimate objective is to offer a user-friendly tool that enables both the reuse of predefined tests and the creation of custom tests. Given that many Ludii users may lack a background in computer science, particular attention is devoted to ensuring the accessibility of the system. For instance, test names and failure messages must be clear and self-explanatory.
This thesis is organized as follows:

- Chapter 2 introduces the concepts of GGP and GGS. It presents the Ludii platform and the TDD methodology.
- Chapter 3 provides a taxonomy of the tests supported by the framework and how they are organized within the project, analysing the types of errors and mistakes that Ludii developers may encounter.
- Chapter 4 defines the proposed solution, offering a detailed explanation of the implemented prototype of the testing framework. It covers both high-level design aspects and technical implementation details.
- Chapter 5 presents a validation of the system, evaluating the graphical user interface and the database of implemented tests.
- Chapter 6 discusses possible enhancements to the current prototype of the testing framework.
- Chapter 7 summarizes the contributions and findings of this thesis.

2

Background

This chapter presents the foundational background for the work presented in this thesis. It begins by introducing the concepts of GGP [9, 10] and GDL, providing the necessary context for understanding the domain. Subsequently, it presents an overview of the General Game System Ludii [5], based on a ludemic representation of games [6, 11]. The chapter concludes with a discussion of TDD and its applicability to the domain of games, highlighting how it can support the development and validation of game implementations.

2.1 General Game AI

GGP is a prominent area of research in Artificial Intelligence (AI) focused on the development of systems capable of playing a wide variety of games based solely on formal descriptions [3]. These systems are not programmed with the specific rules or strategies of any game. Instead, they must interpret the game's rules from a textual description and make decisions in real-time, akin to a human encountering a new game for the first time.
A general game player must operate under uncertainty, manage time constraints, and reason about the potential actions of opponents. As a consequence, it must be able to interpret the rules, analyze the current game state, and select the most promising moves using only the information extracted from the provided game description.
GGP is sometimes considered a stepping stone towards Artificial General Intelligence (AGI) [12], which may be summarized as the development of AI that can accomplish general tasks, not necessary restricted to games. Nevertheless, games provide an ideal domain for such research [13], as they involve strategic reasoning, decision making, and learning capabilities that are essential to AGI.
GGP systems rely on formal game descriptions written in GDLs, which serve as a bridge between a game's abstract representation and the mechanisms required for reasoning and gameplay.
Beyond GGP, General Game AI also includes other aspects such as the analysis, modelling [1, 14], and generation of new games [2]. In this broader field, GGP serves as a supporting tool, reflecting an expanding view of its potential impact and utility. Within this extended context, the focus shifts toward GGS [4].

A GGS is a software environment designed to host and manage a variety of games, often integrating GGP agents for the purpose of analysis, strategic evaluation, or automated game generation. Several GGSs have been developed, each with different goals and levels of expressiveness. Below is an overview of some notable systems:

- **Smart Game Board** [15]: originally designed for the game of Go[1], was later extended to support Othello[2], Chess[3], and Nine Men's Morris[4]. This system features a graphical interface aimed at facilitating the study and teaching of games, with features such as trial recording and testing new strategies.
- **Zillions of Games**[5]: a commercial platform enabling users to define and play a wide variety of abstract board games and puzzles. It supports a proprietary rule language, the Zillions Rules File (ZRF), which functions as a GDL.
- **GGP-Base**[6] [16]: developed at Stanford University, GGP-Base is one of the earliest frameworks for GGP. It introduced the original GDL and provides infrastructure for developing, testing, and evaluating GGP agents. The framework has been foundational to numerous academic projects and international competitions. Such competition led to multiple champions illustrating original techniques or approaches, such as the latest one WoodStock [17] which combined contraint programming and learning techniques [18, 19].
- **Regular Boardgames (RBG)** [20]: both the system and its associated GDL are referred to as RBG. This framework is designed to describe a broad class of deterministic, perfect-information board games, with a focus on expressiveness and computational efficiency.
- **Openspiel**[7] [21]: a research-oriented framework targeting extensive-form games with imperfect information. Unlike other systems, it does not define a standalone GDL; instead, games are implemented directly using Python APIs.
- **Polygames**[8] [22]: developed by Facebook AI Research (FAIR), Polygames is a framework for training deep learning agents through self-play. It does not use a formal GDL; instead, games are implemented in C++ and Python. Its focus is on optimizing performance for specific games, such as Go or Hex [9], rather than supporting generalization.

Among the systems presented, GGP-Base and RBG stand out as they define their own GDL, in addition to providing environments for running general game players. These two will be used as reference points in the following section to introduce and compare different GDL approaches.

[1] en.wikipedia.org/wiki/Go_(game)
[2] en.wikipedia.org/wiki/Reversi
[3] en.wikipedia.org/wiki/Chess
[4] en.wikipedia.org/wiki/Nine_men's_morris
[5] zillions-of-games.com
[6] github.com/ggp-org/ggp-base
[7] github.com/google-deepmind/open_spiel
[8] github.com/facebookarchive/Polygames
[9] en.wikipedia.org/wiki/Hex_(board_game)

2.2 Game Description Languages

As discussed in the previous section, GGP systems rely on formal descriptions of games written in a GDL. The choice of GDL is critical, as it will define the range and complexity of games a GGP system can support.

GDLs can be categorized broadly into two types: *low-level* and *high-level* languages [13]. Low-level GDLs describe games through simple, generic constructs, often but not always based on first-order logic to specify how the game state evolves. While these languages are general, they tend to be verbose and unintuitive. Writing, reading, and debugging game descriptions in low-level GDLs can be challenging and time-consuming.

In contrast, high-level GDLs aim to encapsulate common game concepts using domain-specific terminology. As a result, game descriptions are more concise and readable, allowing authors to express complex mechanics with game-oriented constructs. They also lower the barrier for non-experts, enabling individuals without a technical background to contribute game descriptions that can be integrated into GGP systems.

For many years, the GDL developed at Stanford's GGP testbed has served as the standard for academic GGP research. We refer to this language as **S-GDL**. S-GDL is a logic programming language based on first-order logic, and game rules are expressed through sets of logical clauses. While the S-GDL is capable of describing any deterministic game with perfect information, subsequent versions (S-GDL-II[23] and S-GDL-III[24]) extend the language to support hidden-information and epistemic games, respectively. However, the overall framework still presents several limitations. Fundamental components such as boards or arithmetic operations must be explicitly defined for each game. Its description can be difficult to modify and debug; as a consequence, S-GDL is generally classified as a low-level GDL.

To address some of these issues, the RBG system and language was introduced [20]. Based on regular language theory, RBG supports both a low-level and a high-level syntax, which can be mutually converted. As a result, RBG allows for more human-readable game descriptions and for efficient parsing. It has been demonstrated that RBG can represent all finite deterministic turn-based games with perfect information, and that it is more efficient than S-GDL in practice.

2.3 Ludii: A Complete General Game System

Ludii is a complete General Game System that distinguishes itself from previous systems through its unique GDL [5]. Unlike other GGSs, Ludii's GDL is designed not only for playing games, but also for *designing* and *analysing* them [1, 25, 26], as well as for *teaching* and *learning* through them [27].

Ludii emerged from the Digital Ludeme Project (DLP)[10], a five-year research project launched in 2018 at Maastricht University. The goal of this project was to model the 1000 most influential traditional games in history within a single digital database [28]. This database enables the analysis of relationships between games and their components, supporting the development of a model for the historical evolution of games [29–31]. As part of this project, Ludii is capable of modeling and playing a wide variety of traditional strategy games.

[10] ludeme.eu

Ludii is an evolution of Ludi [4]. While Ludi supported a wide range of combinatorial games, it faced limitations. Some games required rules that extended beyond Ludi's original GDL capabilities, and the system suffered from performance issues, particularly with legal move generation and board evaluation in complex games. In contrast, Ludii addresses these limitations through its class-grammar approach [32] for automated grammar generation (explored in the next section), and through a Monte Carlo-based move-planning system that relies solely on a forward model [7].

2.3.1 The Ludemic Approach

Ludii's GDL is based on the concept of *ludemes*, which are high-level keywords representing game-related information. These elements include all aspects of a game, such as the board, players' pieces, rules of play, and winning conditions. Any aspect that can be described using a common term in games (e.g. Card, Move, or Mode) can have a corresponding ludeme. As a result, Ludii game descriptions are self-explanatory and composed of human-readable terms.

The term *ludeme* was first introduced by Alain Borvo in the 1970s for his analysis of a novel card game [33] and was later refined by Cameron Browne [34]. A ludeme has four key characteristics [34]:

- **Discrete**: represents a distinct unit of game-related information.
- **Transferable**: can be reused across different games or other ludemes, either independently or as part of a composite structure.
- **Compound**: may be composed of other ludemes, though it can also be atomic.
- **Contrastive**: altering a ludeme changes the behavior or structure of the game, meaning it plays a defining role in shaping gameplay.

A Ludii game description is, therefore, a structured composition of ludemes. Just as a house is built from individual bricks, a game is constructed ludeme by ludeme, with each contributing to the game's rules, structure, and dynamics.

This ludemic approach makes game descriptions clear and straightforward; modifying a rule or the board design becomes easy [25]. More importantly, it abstracts away the complex implementation details behind each concept. Unlike other GDLs, such as S-GDL, which require explicitly writing instructions to update the game state, Ludii encapsulates these operations within high-level keywords. As a result, descriptions in Ludii are less verbose, allowing key concepts to be expressed clearly and compactly.

An example of a simple game description using Ludii's GDL is shown below for the game Havannah[11]:

Example 2.1: Havannah description using Ludeme (Source: `github.com/Ludeme/Ludii`)

```
(game "Havannah"
    (players 2)
    (equipment
        {
            (board (hex 8))
```

[11] `en.wikipedia.org/wiki/Havannah_(board_game)`

```
6               (piece "Marker" Each)
7           }
8       )
9       (rules
10          (play (move Add (to (sites Empty))))
11          (end
12              (if
13                  (or
14                      {
15                          (is Loop)
16                          (is Connected 3 SidesNoCorners)
17                          (is Connected 2 Corners)
18                      }
19                  )
20                  (result Mover Win)
21              )
22          )
23      )
24  )
```

The main ludemes are *game*, *players*, *equipment*, and *rules*, which defines the skeleton of a game. The *(game "Havannah")* ludeme assigns a name to the game. *(players 2)* indicates that the game involves two participants; by default, each player takes one move per turn, which is implicit unless specified otherwise. The *equipment* section specifies the physical components of the game: an hexagonal board and a type of piece, "Marker" for each player. Finally, the *rules* section encodes the game's logic: players take turns placing a piece on an empty cell, and the game ends when one of the following three conditions is met:

1. Loop around any site (line 15).
2. Connecting any three edges, excluding corner points (line 16).
3. Make a bridge connection between any two corners (line 17).

2.3.2 Ludii System

The strength of Ludii lies in its class grammar approach [32], which establishes a direct link between each ludeme and the underlying Java code that implements it. In fact, the Ludii GDL is automatically generated from the structure of Ludii's Java source code. As a result, the system interprets ludeme-based game descriptions by reconstructing the corresponding Java objects through Java Reflection. This ensures a strict 1:1 mapping between the grammar used to define games and the internal source code responsible for their execution [26].

This design makes the programming language itself the game description language. It also makes Ludii highly extensible: developers can introduce new game concepts by implementing them as Java classes, provided they adhere to the framework's formatting conventions for ludeme constructors.

At the core of the system is the ludeme library, where each ludeme is represented by a dedicated Java class that implements the *Ludeme* interface. Figure 2.1 illustrates how these ludemes are organized into categories. Referring back to the game description of Havannah in Example 2.1, we can observe clear 1:1 mappings between ludeme keywords and Java classes (e.g., *Game*, *Players*, *Rules*).

As previously noted, a game in Ludii is composed of a hierarchy of ludemes, forming a tree structure where each node represents a ludeme. During gameplay, Ludii traverses this tree

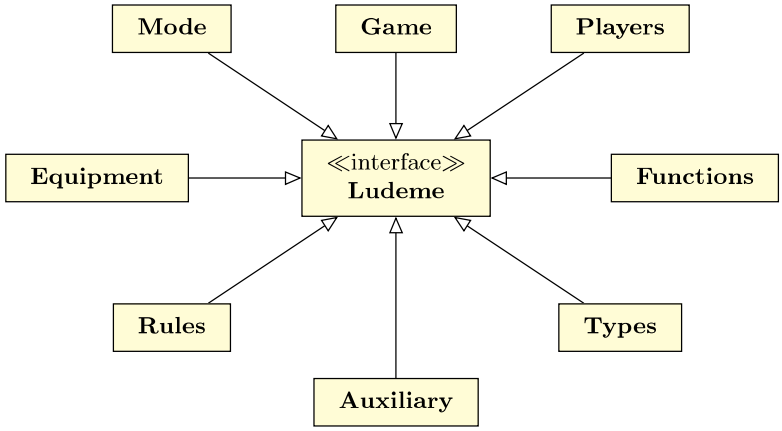

Figure 2.1: Categories of Ludemes [6]

to evaluate the game state and determine legal moves by recursively processing each node's logic. For instance, the *Context* class acts as a central container for the current game session, maintaining the game state and move history via the *State* and *Trial* classes, respectively. By storing all relevant information in *Context* and its associated classes, Ludii efficiently computes which moves are legal, what happens after a move is made and whether a game has ended, and who won.
Ludii class grammar provides notable advantages, making Ludii stand out with respect to other GGS [5][35]:

- **Simplicity**: Ludeme-based descriptions are easy to write and modify. They require a significantly smaller number of keywords compared to other GGS, allowing quick experimentation with rule variants or board configurations.
- **Clarity**: Ludeme-based descriptions are clear and human-readable, thanks to the use of meaningful keywords. This enhances accessibility, particularly for non-specialist users, and enables the automatic generation of Board Game Manuals [36].
- **Generality**: Ludii can support a vast variety of games without requiring structural changes. Thanks to its class grammar architecture, it can theoretically represent any game that can be implemented in Java.
- **Extensibility**: New features or concepts can be easily incorporated into the system due to its modular design. New functionality is introduced by simply adding Java classes to the ludeme library, which are automatically integrated into the class grammar.

- **Efficiency**: Since developers have direct control over ludeme implementations, they can optimize games at the code level. This flexibility enables trade-offs between the length of descriptions and runtime performance.
- **Evolvability**: Ludii's modular structure is well-suited to evolutionary algorithms. Random combinations or mutations of ludeme trees are more likely to result in playable and meaningful games than logic-based formats like GDL.
- **Cultural Application**: Ludii serves as a powerful tool for Digital Archœludology [37–40], enabling the reconstruction, classification, and historical analysis of traditional games using its integration with a cultural-historical database.
- **Universality**: Ludii is provably universal [41] not only for finite deterministic games with perfect information, but also for finite non-deterministic and imperfect-information games. This ensures that all games expressible in both GDL and GDL-II can be described using Ludii's class grammar. Recent research has further extended this direction by proposing general AI models for imperfect-information games, such as the Belief Stochastic Game model [42].

2.3.3 Concepts

The notion of concept is fundamental within Ludii. When referring to concepts, we mean game concepts: features expressed in game-specific terms commonly used by players and designers, which can be associated with a game or an element of play. In Ludii, a concept is characterized by a name, a category, a data type, and a computational type [43]. The chosen name should reflect terminology familiar to human players or game designers as closely as possible. Figure 2.2 illustrates the main categories to which a concept may belong. The data type of a concept may be either numerical, which quantifies the concept,

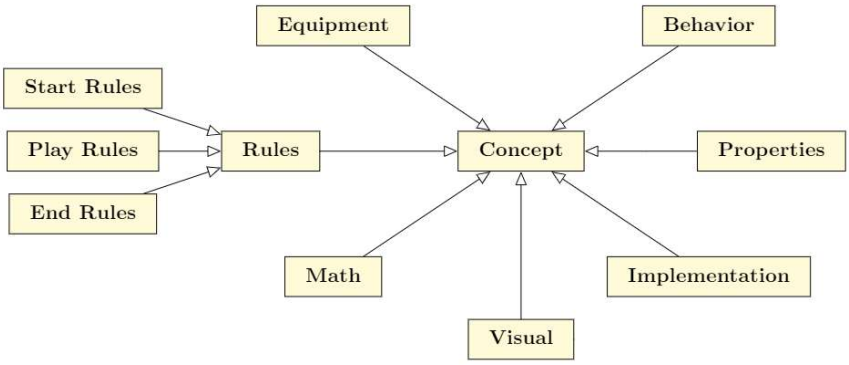

Figure 2.2: Categories of Concepts

or binary, which simply indicates its presence. Typically, a binary concept is determined by the inclusion of one or more ludemes in the game description. A numerical concept, on the

other hand, is associated with a direct numeric value (e.g., Players 2) or can be derived from the frequency of binary concepts (e.g. the average number of times a particular terminal state is reached).
From a computational perspective, concepts are classified based on how their values are obtained. Compilation concepts are computed at compile time and refer to static elements of a game (e.g., the dimensions of the board). Conversely, playout concepts require gameplay and are usually numerical. Their values are derived from statistical analyses over multiple playouts, for instance, the frequency of occurrence of a specific binary concept.

2.4 TDD: Test-Driven Development

TDD [8, 44] is a software development methodology aimed at reducing bugs and encouraging cleaner, more maintainable code. The approach was popularized by software engineer Kent Beck, who rediscovered its origins in an early programming manual. In that manual, the author suggested taking an input tape, manually writing the expected output tape, and then programming until the actual output matched the expected result [45]. Beck adapted and formalized this idea into what is now known as TDD.
The core principle of TDD is to begin the implementation of a feature or function by first writing a test that defines its expected behavior. The development process follows a short, iterative cycle, known as Red-Green-Refactor:

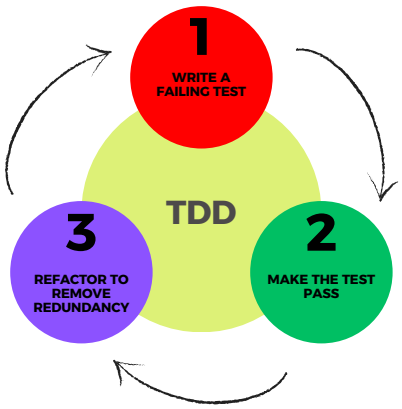

Figure 2.3: Iterative steps of the TDD approach

1. **Red phase**: write a (failing) test that specifies the desired functionality.
2. **Green phase**: write the minimal amount of code necessary to make the test pass.
3. **Refactor**: refactor the code to improve its structure while keeping the test green.

This cycle is repeated continuously, promoting small, incremental changes that are validated at each step.

TDD provides several benefits [46] that often motivate developers to adopt this approach. First, it ensures that all new code is covered by at least one test, leading to more robust and reliable software. As a result, when a change breaks the code, the failure is immediately visible through the test suite, making bugs easier to locate and fix. Additionally, TDD encourages clarity, as developers must fully understand the expected behavior before implementation begins. It promotes modular design by forcing a clear separation of concerns, since components must be independently testable. Moreover, the need for extensive debugging is significantly reduced, as errors are caught early in the development process.

However, TDD is not without drawbacks [46]. It increases the overall volume of code and requires time and effort to adopt the mindset. Writing tests before implementation may also seem unnatural or slow at first.

Nevertheless, TDD is widely adopted in enterprise software development, web services, and backend systems, where the logic is well-defined and modularity is critical. These domains benefit from TDD's emphasis on automated testing and early bug detection. In such contexts, TDD integrates well with continuous integration pipelines, making it easier to catch regressions early and improve developer confidence. TDD is also common in libraries and APIs, especially when stability and long-term maintainability are required. In these cases, writing tests first helps clarify the expected behavior of the interfaces and reduces the likelihood of introducing breaking changes.

2.4.1 TDD in Game Development

Despite its proven effectiveness in many software domains, TDD is not widely adopted in game development. This is largely due to the nature of typical video games, which often involve visual components [47] such as rendering, animation, real-time interactions, and user experience. These aspects are difficult to test through automated unit tests and are inherently subjective and non-deterministic, making it challenging to define precise expected outputs for automated verification. Furthermore, many game engines and frameworks promote tightly coupled design which can hinder modularity and testability.

However, this situation changes significantly when considering the development of board games. In contrast to real-time or graphical games, board games are characterized by well-defined rules, deterministic mechanics, and discrete states. These traits make board games particularly suited for TDD. Board games often operate on structured game states, such as grids or lists of moves, and rely on rule-based logic to determine the legality of actions or the outcome of a game. These aspects are not only testable but also benefit from being validated through automated tests. For example, detecting a checkmate in Chess or verifying the legality of a move in Go are tasks that can be clearly specified, tested, and repeated consistently. In such scenarios, TDD allows developers to encode game rules as tests and iteratively refine their implementation, ensuring correctness from the beginning. Moreover, the deterministic nature of board games allows for consistent test results across runs, simplifying debugging and reducing the risk of non-reproducible bugs. Since most game logic can be developed independently from UI and rendering, board game implementations often achieve better modularity.

In summary, while TDD may pose challenges for general game development, it is particularly well-suited to the domain of board games. Applying TDD in this context enhances

reliability, maintainability, and clarity of the game logic, making it a valuable methodology in the development of structured, rule-based games.

3

Test Identification and Organizational Design

In this chapter, we present the rationale behind the categorization and organization of tests, with the goal of supporting Ludii developers in effectively applying a TDD approach. We begin by outlining the different types of tests considered, explaining the reasoning behind their selection based on the kinds of issues they are intended to detect. We then discuss how each test type is identified and implemented, distinguishing between static and dynamic analysis techniques. Finally, we describe how these tests are structured and integrated into the project, ensuring maintainability, scalability, and alignment with the overall architecture of the testing framework.

3.1 Analysis of Tests

This section presents an analysis of test types identified and evaluated as part of this thesis, with a focus on determining the suitability of and necessity for compile-time tests, runtime tests, or a combination of both for the Ludii system. Compile-time and runtime tests serve different purposes in the software development lifecycle, each with their distinct advantages and limitations.

3.1.1 Compile-time Tests

Compile-time tests are particularly useful for identifying syntax and semantic errors, type mismatches, and other issues that can be detected without executing the program. These tests are typically executed during the compilation process, ensuring that the code adheres to predefined rules and constraints before running. Furthermore, Ludii relies on a custom class-grammar, compile-time tests could also be valuable for verifying how a game is structured using ludemes.

The current Ludii system already enforces some grammar rules through an integrated editor, which provides semantic parsing and uses a colored dot in the top-right corner to indicate whether the code will compile. However, not all semantic rules are covered by this functionality. In some cases, code may compile successfully but still be incorrect due to an unrecognized grammar violation. To illustrate this, consider the following example rule:

> *When a game has N players, the correct way to reference them is using the syntax P1, P2, . . . , PN.*

Figure 3.1a shows the standard Ludii editor with an example Tic-Tac-Toe game. As explained in the previous chapter, a game description consists of three mandatory blocks: Players, Equipment, Rules. The description correctly declares two players using the ludeme **(players 2)**, meaning that **P1** and **P2** should be used to reference individual players. The editor confirms that the code will compile, as indicated by the green dot.

(a) Correct ludeme description

(b) Incorrect ludeme description

Figure 3.1: Example of a semantically valid and invalid ludeme description of Tic-Tac-Toe

To evaluate the robustness of the editor, we deliberately introduce an inconsistency in the game description by referencing **P3**, despite only two players being declared. As shown in the figure 3.1b, the dot remains green, meaning the error is not detected by the current system.

While this specific issue may be obvious to a Ludii developer, similar undetected grammar violations could be harder to identify in larger or more intricate game descriptions. In practice, not all errors are as trivial as using an incorrect player reference. Some issues may involve deeply embedded rules, interactions between multiple ludemes, or subtle misinterpretations of the intended game mechanics. Without explicit validation, these errors might go unnoticed until much later in the development process, making debugging more difficult and time-consuming.

Given these considerations, integrating compile-time tests could provide an additional layer of verification, helping developers catch grammar-related issues before execution. However, as we will discuss in the next section, compile-time validation alone is insufficient for testing gameplay behavior, which is why runtime tests play a crucial role in our framework.

3.1.2 Run-time Tests

Run-time tests are executed during the program's execution, allowing for the validation of dynamic behavior, logic, and interactions that cannot be verified at compile time. Unlike

compile-time tests, run-time tests do not focus on analyzing the ludeme grammar; instead, they investigate winning conditions, common play rules, and board design. These aspects can be particularly challenging to handle, as they often involve complex interactions that can only be observed after the game has been compiled and executed. Additionally, run-time tests enable the evaluation of game trials, allowing conditions that appear only during gameplay to be tested [48]. This makes them particularly useful for ensuring that a game functions as intended beyond just compiling successfully. The table 3.1 highlights the key aspects of these two types of testing.

Aspect	**Compile-time**	**Run-time**
Detection Scope	Syntax errors, incorrect ludeme usage	Rule enforcement, board interactions, win conditions
Performance	No impact on runtime	Adds some execution overhead
Flexibility	Limited to grammar analysis	Can cover broad range of scenarios
Debugging	Identifies issues before execution	Provides insights during gameplay

Table 3.1: Comparison between Run-time and Compile-time tests

From a development perspective, runtime tests align well with TDD because they allow for progressive validation of game behavior as new features are implemented. Since many aspects of a game can only be evaluated during execution, runtime tests provide immediate feedback on whether a feature behaves as expected. Moreover, runtime tests support incremental refinement, a feature of TDD, ensuring that each modification to the game logic maintains correctness. By continuously running tests during development, potential errors are caught early, reducing the need for extensive debugging later.

In conclusion, compile-time tests primarily focus on analyzing how a game is written using ludemes, ensuring compliance with grammar rules. While valuable, they are also highly restrictive, as they limit testing to syntax verification rather than assessing gameplay correctness or overall game design. In contrast, runtime tests offer a broader and more practical approach. They not only cover board design and gameplay behavior but also incorporate some checks that compile-time tests would perform dynamically. This makes them better suited for TDGD, as they provide developers with real-time feedback on game logic and mechanics. Thus, we opted for runtime tests as they align more closely with the goals of the framework.

3.2 Identification of Tests

We identified two major categories into which our tests fall: Static Tests and Dynamic Tests.

3.2.1 Static Tests

As previously discussed, run-time tests can also perform some of the checks traditionally associated with compile-time validation of the ludeme class grammar. These tests are referred to as static tests because they are executed only once, when the game is loaded, and do not rely on any game trials or runtime instances.

Recalling the example provided in Section 3.1.1, where the system failed to notify the developer about an incorrect use of player identifiers. Static tests aim to catch such issues. However, their purpose extends beyond grammar validation. They also verify certain properties and configurations of a game that are determined at load time, right after the game becomes active. To provide a simple example, consider the correct description of the Tic-Tac-Toe game in Figure 3.1a. The game's winning condition is defined using the Line ludeme followed by a number (e.g., 3 in Tic-Tac-Toe). To ensure this rule is valid, that number should be less than or equal to the board's maximum dimension. While it may seem sufficient to check the maximum dimension of the side of the board declared in the equipment section, **(board (square 3))**, that's not always the case. Not all boards are square, and many are constructed with complex structures or ludemic rules that make dimensions non-obvious. Therefore, the maximum board dimension is a value that is usually computed after the game is loaded. A static test, in this case, can validate that the win condition is compatible with the actual structure of the board, even if its dimensions are not directly inferable from the ludeme description.

This illustrates how static tests are essential not just for grammar checks but also for early validation of game logic that emerges at load time.

3.2.2 Dynamic Tests

The other category of tests includes dynamic tests, which are particularly powerful because they provide diverse feedback and quantitative measures, investigating gameplay conditions as the game progresses. To perform such tests, game trials are required, as these tests validate rules, board states, score evolution, and more.

Unlike static tests, which are executed once when the game is loaded, dynamic tests are executed multiple times during the execution of a game simulation. In Ludii, this is made possible thanks to the presence of various AI agents, which can simulate thousands of games automatically [48]. Dynamic tests can begin either from the initial game state or from a specific state snapshot, depending on what the test aims to evaluate.

Another important consideration is the execution time of these tests. Due to the nature of simulations, dynamic tests can be significantly longer than their static counterparts, especially when covering multiple turns or full game trials.

One example of dynamic testing is the verification of score computation for each player. In the simplest use cases of the score, it corresponds to the number of pieces a player has on the board. However, Ludii contains numerous games with different and sometimes complex scoring mechanisms. If not properly handled, these mechanisms may lead to incorrect outcomes. To clarify this, consider the game Reversi (or Othello). The objective of the game is for a player to end with the most pieces on the board. Players can capture opponent pieces, which are "flipped" to the capturing player's color. Each piece is associated with a state value (e.g., 1 for Player 1 and 2 for Player 2), and the score is computed based on the count of each player's pieces on the board, as shown in the following code.

Example 3.1: Computation of score with ludeme description

```
    (set
        Score
        P1
        (count Sites in:(sites State 1))
    )
    (set
        Score
        P2
        (count Sites in:(sites State 2))
    )
```


If a Ludii developer mistakenly defines the score for both players to depend on the same state (e.g., both on State 2), then throughout the game both players will always have identical scores, inevitably ending in a draw. This outcome highlights a logic error, as shown in the figure 3.2, where the correct score of Player 1, with black pieces, should be 3.

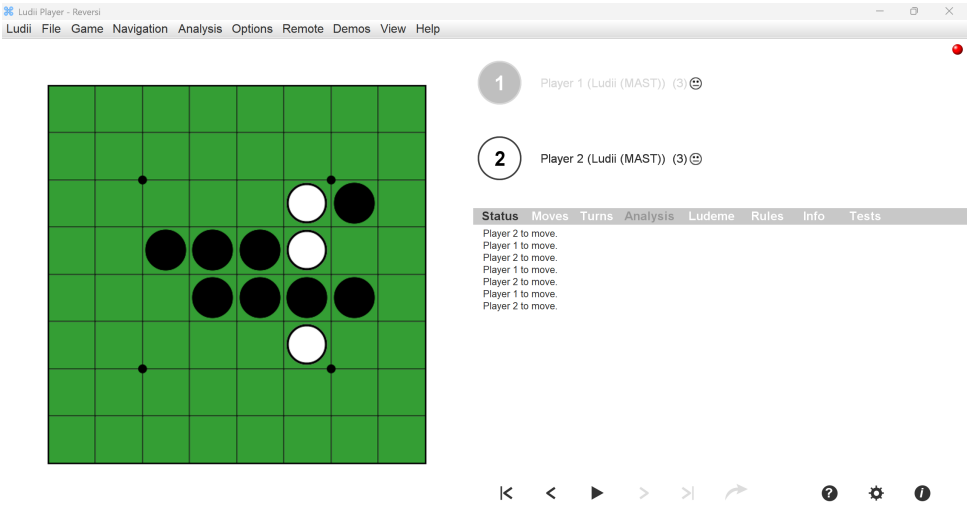

Figure 3.2: Incorrect computation of score in Reversi

Therefore, with the support of a simulation, a dynamic test for this case should count the actual pieces on the board after each move, compare it with the score reported by the system and detect that both players always have the same score.
One of the challenging aspects of dynamic tests lies in interpreting the results and determining the appropriate length of a simulation. In this example, useful data might include the number of moves executed, as well as the score evolution at each step. However, a natural question arises: how long should the simulation be? is it necessary to simulate a full game, or would a limited number of moves (e.g., the first N moves) be sufficient?
In this case, even a single move can be enough to reveal the error. As a matter of fact, according to the game rules, after Player 1 makes the first move and captures some of Player 2's pieces, their score should increase. If both players still have the same score after that, it indicates a scoring issue. Thus, the depth of the simulation may depend heavily on the specific rule being tested. A deep discussion about how to determine the scope of simulations is provided in Chapter 6.

3.3 Organization of Tests

In the context of our testing framework, a well-structured organization of tests is essential to ensure that the system remains functional, scalable, and extensible as it evolves. Ludii currently comprises over 800 distinct concepts, each representing a specific feature within the game system. Ideally, each concept should be tested, which implies a growing number of tests, potentially over 800. Without a modular and clear structure, managing such a large volume of tests would quickly become demanding, leading to difficulties in debugging, extending, and maintaining the framework.

To address this challenge, we adopted a structure in which each concept is assigned its own package. In the current state of the framework, we have only a few tests per concept. As a result, these are grouped into a single test class per package, named by appending the suffix *Test* to the concept name (e.g., *BoardTest* for the *board* concept).

This convention is inspired by early versions of JUnit [49], which recognized test methods using naming patterns such as *testXXX()*. Similarly, our framework uses this naming convention to automatically identify test classes by scanning for those whose names end in *Test*. Consequently, the only requirement imposed by our system is that each test class must conform to the *XXXTest* naming pattern.

We deliberately adopted a naming-based approach for identifying test classes, as JUnit does not provide an explicit mechanism to designate a class as a test class. The available annotations apply only at the method level. Internally, JUnit determines whether a class is a test class by inspecting its contents for annotated test methods. Replicating this behavior would introduce unnecessary complexity and overhead. By enforcing a naming convention we restrict the discovery process to classes whose names conform to the *XXXTest* pattern, streamlining the identification process and improve performance by avoiding the inspection of irrelevant classes.

Additionally, concept classes such as Board, Player, and Piece already exist in the main codebase: using the same names for test classes would create naming conflicts. Our convention avoids this issue while keeping test classes closely aligned with the corresponding concepts.

An example of the current structure is shown below:

Example 3.2: Test package organization

```
src/
  tests/
    board/
      BoardTest.java
    piece/
      PieceTest.java
    player/
      PlayerTest.java
```

This organization aligns naturally with the structure of the Ludii system, which is itself concept-driven. Since concepts are the core units used throughout the platform, assigning each one dedicated and isolated set of tests reinforces the conceptual boundaries and enhances traceability.

This design offers several practical advantages. First, it enables automated discovery and execution of tests through reflection, a key mechanism of the framework that will be discussed in section 4.2.1. The naming and directory conventions simplify the aggregation of tests

for execution in the GUI without requiring hardcoded references, as briefly noted earlier. For instance, the system constructs the fully qualified names of test classes automatically, and thus depends on this predictable structure to locate and run tests reliably.
Second, this modular architecture significantly reduces coupling between tests. Adding a new concept only requires the creation of a corresponding package and one or more test classes that follow the naming convention, making the framework highly extensible with minimal effort.
Moreover, this structure lends itself to future enhancements, such as support for sub-concepts. For example, the **Shape** concept includes variations like **HexShape** and **SquareShape**. These could be tested in subclasses within the same package, where a base class ShapeTest tests shared functionality and shape-specific test classes implement additional validations. An example of how the organization would become follows:

Example 3.3: Test package organization with sub-concepts

```
src/
  tests/
    Shape/
      ShapeTest.java
      HexShapeTest.java
      SquareShapeTest.java
```

In conclusion, the chosen organization ensures a clean separation of concerns and plays a central role in supporting dynamic test discovery and execution through reflection.

This chapter has detailed the principles guiding the classification and implementation of tests within the Ludii framework. By differentiating test types and their corresponding analysis methods, it establishes a clear structure that facilitates effective TDD. The proposed organization promotes maintainability and scalability, laying a solid foundation for the testing framework's integration and future enhancement.

4

Testing framework prototype

This chapter presents the prototype of our testing framework, designed as a supportive tool for Ludii developers. The framework aims to facilitate the testing and validation of Ludii game implementations by providing a structured and automated environment.
The chapter is organized into two main sections: Architecture and Implementation. The first section provides a high-level overview of the framework's design, highlighting its core components and their interactions. The second section delves into the technical details of the implementation, describing the specific tools employed to realize the framework.

4.1 Architecture

This section introduces the architecture of our testing framework. To support scalability, maintainability, and clean separation of concerns, we decided to structure the framework using the Model-View-Controller (MVC) design pattern. A brief overview of the pattern is provided below before discussing how it applies to our specific case.
The MVC is a well-established architectural pattern commonly used in software engineering to separate the concerns of an application [50]. It divides the application into three main components:

- **Model**: handles the core logic and data of the application. It is responsible for maintaining the state and rules of the system.
- **View**: manages the presentation layer. It displays data from the Model and provides a visual interface for the user.
- **Controller**: acts as an intermediary between the View and the Model. It handles user input, updates the Model, and refreshes the View accordingly.

As a reminder, our ultimate goal is to provide a tool for Ludii developers: an interface that facilitates the selection and execution of both generic and game-specific tests, enabling a TDGD approach.

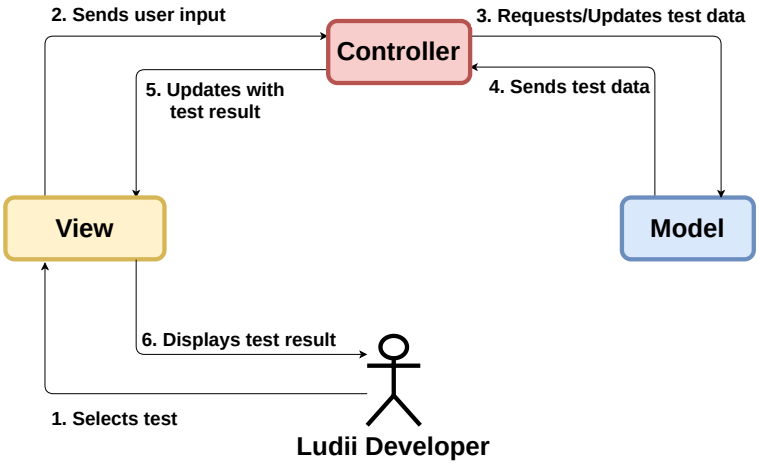

Figure 4.1: Model-View-Controller architecture for our framework

Given this interactive nature, a clear separation between data, logic, and presentation becomes essential. The MVC pattern directly supports this need by decoupling the core functionality from the user interface. For example, when a user selects a test through the graphical interface, the system must dynamically identify and execute the corresponding method, process the results, and display the outcome, while maintaining a clean separation between backend logic and interface components.
Additionally, the Ludii project is under continuous development, with new games, features, and rules introduced regularly. As a result, the testing framework must be adaptable to these changes. By adopting the MVC architecture, we ensure modularity and isolation between components, allowing individual parts of the system to evolve independently. For instance, enhancements to the View can be made without requiring changes to the Model, and new testing capabilities can be integrated without affecting the interface.
The following details explain how the architecture integrates with our framework.

4.1.1 Model

The Model represents the core of our framework, encapsulating the internal representation of test classes, methods, and parameters. It is responsible for managing the data needed throughout the lifecycle of test discovery, execution, and result interpretation.
As introduced in Section 3.3, each testable concept is represented by a dedicated test class. The Model mirrors this structure using three key components, illustrated in Figure 4.2.

- **TestClass** handles the organisational structure and grouping of tests. It tracks all test methods belonging to a concept, along with its package name and fully qualified class name.
- **TestMethod** stores detailed metadata about each individual test, including its name, whether the test passed or failed, the duration of the execution, its classification as

either static or dynamic, and additional runtime information such as failure messages.

- **TestParameter** records both the expected type of each parameter and its associated value.

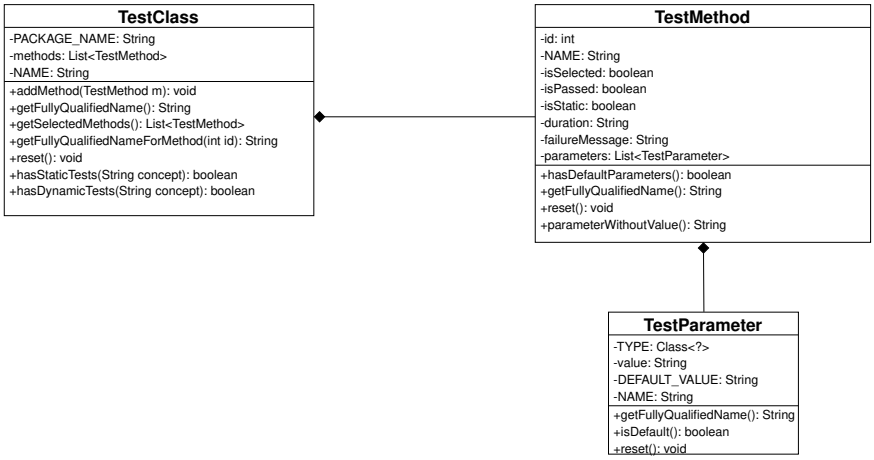

Figure 4.2: Class Diagram of our Model

One interesting design detail is the presence of two fields in the TestMethod: defaultValue and value. While their purpose might seem similar at first glance, they serve fundamentally different roles. One reflects the values annotated directly in the source code, while the other is influenced by user interaction at runtime. This subtle distinction becomes crucial in dynamically supporting the management of default values in our tests, as will be discussed in Section 4.2.2

Another notable feature is the construction of a test's fully qualified name. For those unfamiliar with the term, a fully qualified name is a dot-separated path that uniquely identifies a method within the project. For example:

tests.MyTestClass.testExampleMethod(int, String)

This name, formed by combining the class and method signature including parameter types, may seem like a simple string, but its construction plays a central role throughout the entire framework. As a matter of fact, this approach is a foundational part of how the framework dynamically identifies and executes tests.

It's also worth noting that much of the Model's functionality is built on top of reflection, a powerful Java technique for inspecting classes and methods at runtime. While we only hint at it here, reflection carries many of the framework's dynamic behaviours and will be explored in more depth in Section 4.2.1.

Finally, the Model is also responsible for maintaining consistency across multiple test runs. It provides methods to reset its internal state, ensuring that no stale data persists

between sessions. This is particularly important when the user re-runs tests with different parameters or after modifying game definitions.

4.1.2 View

The View represents the visual interface between the user and the testing framework. It is presented as an integrated dashboard, designed to blend seamlessly into the existing Ludii interface. Its purpose is to allow developers to interact with the framework in a user-friendly and intuitive way.
The View interacts with the Controller to fetch the structure and metadata of test classes, and to update their state based on user interaction or execution feedback.
As shown in Figure **??**, the dashboard provides a structured overview of available tests, clearly distinguishing between static and dynamic tests, and grouping them by concept. Each test method can be selected individually, and when a method requires parameters, a dialog is prompted to gather the necessary input. If default values for the parameters exist, they are pre-filled to improve usability and reduce redundancy. Otherwise, the user is expected to insert the values manually.

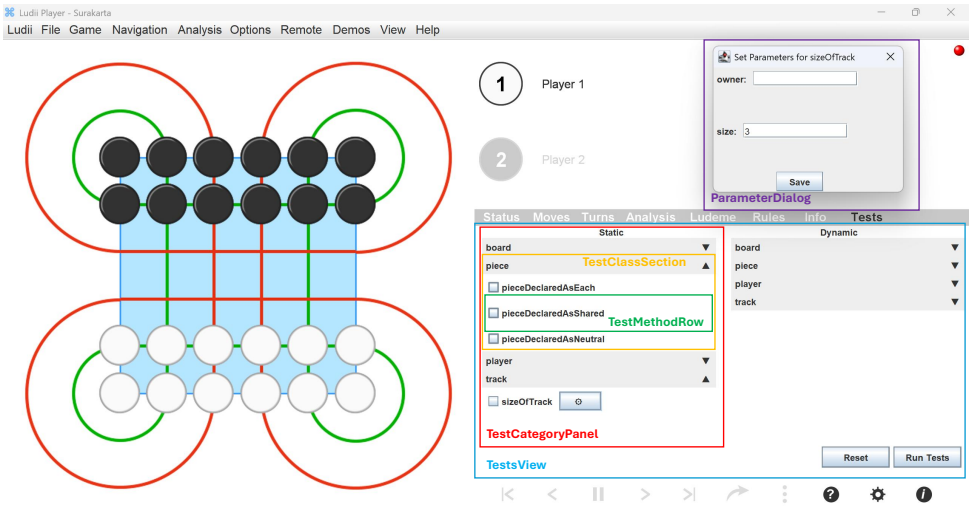

Figure 4.3: Integrated testing dashboard in the Ludii editor

To support maintainability and scalability, the View has been designed in a modular way. The entire visual component is wrapped in the **TestsView** class, which orchestrates the layout and interaction logic. Internally, the View is composed of several components:

- **TestCategoryPanel**: responsible for rendering the Static and Dynamic test sections.
- **TestClassSection**: used to group test methods by concept.
- **TestMethodRow**: represents individual test entries, along with their execution status.

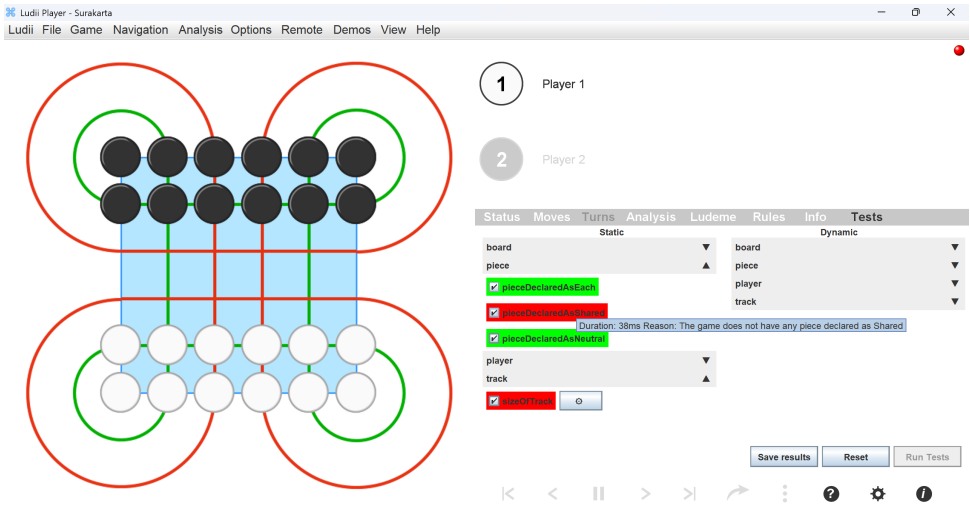

Figure 4.4: Integrated testing dashboard in the Ludii editor with results

- **ParameterDialog**: a dedicated dialog window used to collect user input for test parameters.

This layered organization not only promotes reusability and readability but also enables future extensions to the View without impacting other components of the framework. For example, additional test categories or alternative input methods can be incorporated with minimal structural changes.
Moreover, the View dynamically reflects the outcome of test executions. After running tests, it highlights whether each test has passed or failed, displays execution time, and, in case of failure, reports the reason, as shown in figure 4.4.
This feedback loop is crucial to facilitate TDGD, offering clear and immediate insights into the status of the test suite.
Another key feature offered by the View is the ability to export the results of test executions. Users can generate a *.csv* file containing detailed information about each executed test, including the test name, execution status (pass/fail), duration, and any error messages. The exported file also includes summary statistics such as overall success rate, average execution time and the total number of tests.
This functionality is particularly useful for analyzing the progression of development. In future scenarios, where the test database may become significantly large, the exported file will allow developers and researchers to gain a comprehensive understanding of which aspects of a game behave correctly and which do not, especially in the case of complex games. Furthermore, the .csv file enables external visual analysis through external tools, making it possible to generate graphs or diagrams that can track trends such as the stability of test results across different versions of a game.
The following table 4.1 shows how the information are organized when exporting the file, with the game Tic-Tac-Toe as example.

Game	Test Name	ET (ms)	Passed	Failure Message	Type
Tic-Tac-Toe.lud	lineLessOrEqualThanBoardSide	132	true		Static
Tic-Tac-Toe.lud	pieceDeclaredAsShared	30	false	The game does not have any piece declared as Shared	Static
Tic-Tac-Toe.lud	pieceDeclaredAsEach	200	false	The game does not have any piece declared as Each	Static
Tic-Tac-Toe.lud	pieceDeclaredAsNeutral	16	false	The game does not have any piece declared as Neutral	Static
Tic-Tac-Toe.lud	noUndeclaredPlayerReference	19	true		Static
Tic-Tac-Toe.lud	equalNumberOfPiecesOnTheBoard	24	true		Static
Tic-Tac-Toe.lud	sizeOfTrack	11	false	Track concept is not present	Static
	Total Tests	**7**			
	Success Rate	**42.86%**			
	Mean Duration	**61.7 ms**			

Table 4.1: Test results for Tic-Tac-Toe with summary statistics.

4.1.3 Controller

The Controller is the core orchestrator of our framework, responsible for coordinating the interactions between the interface and the test logic. It is designed to act as the intermediary that manages user input, updates the model accordingly, and triggers visual or logical updates when needed. In our architecture, we distinguish two main responsibilities in the Controller layer: user interaction handling and test execution. To reflect this, the Controller is split into two dedicated components: the Base Controller and the Launcher.

Base Controller: Connecting View and Model

The Base Controller serves as the entry point of the framework's logic, managing the lifecycle of both the Model and the View.
Its primary responsibility is to respond to user interactions, like modifying or inserting test parameters, and to propagate these changes to the Model. Once the model is updated, the Base Controller ensures that the View is synchronized, so that test selections, results, and states are correctly reflected in the user interface.
Additionally, this component acts as a bridge to the Launcher. When a user initiates a test run, it delegates their execution to the Launcher.
This separation of concerns allows for a clean division between user-triggered events and the underlying logic that handles test discovery and execution.

Launcher: Discovery and Execution of Tests

The Launcher is a crucial component of our framework, responsible for executing the tests selected by the user and for providing the results back to both the Model and the View. It has several responsibilities: it must dynamically locate the test in the codebase, resolve and pass the correct parameters, execute the test, and collect its results.
As we can see, these responsibilities are numerous and varied, eventually pushing us to separate this component from the Base Controller. Otherwise, managing all this logic in a single class would quickly become too complex. Nevertheless, the Launcher still acts as a Controller within the MVC architecture: it receives input from the View (i.e., the user

requesting test execution), accesses the Model to retrieve the relevant data, and invokes the necessary components to run the test logic.
To understand how the Launcher accomplishes its tasks, we begin by addressing two fundamental questions:

1. *How can the Launcher dynamically locate a test inside the codebase?*
2. *What information does it need to do this?*

An illustrative analogy can help clarify the process: consider using a map service to locate a specific address. To pinpoint a location, we need to provide complete information, such as the country, region, city, and street number. In our context, the Launcher similarly requires precise information: the package name, the class name, the method name, and any parameters.
All of this information is already gathered and stored within the Model, and it is combined into what is known as the fully qualified name. As detailed in the Model section 4.1.1, each TestClass constructs the fully qualified name for each of its methods. With this information, the Launcher can locate the corresponding test, retrieve and inject the correct parameters (also obtained from the Model), execute the test, and finally collect the results for reporting back to the rest of the application.
At this point, we have addressed the second question: *what type of information is needed to locate the test.*
The remaining tasks (executing the test, passing parameters, and collecting results) are performed by leveraging features provided by JUnit5, which we will explore in the following section.

JUnit5 integration

JUnit is a widely-used testing library for Java that allows developers to write and run automated tests [51].
In a typical testing scenario, a developer uses an Integrated Development Environment (IDE) to run tests separately from the application to ensure code correctness. Such tests verify the logic behind the code and are typically executed during the build process, independent of the application runtime.
Our tests differ from the standard approach because they need to be triggered dynamically while the application is running. In particular, our system requires tests to be executed based on user input at runtime. This demands a solution that varies from the static test execution model.
To achieve this, we turned to JUnit5, which supports dynamic test execution, a core requirement of our framework.
JUnit5 introduces several powerful components that we leverage in our implementation:

- **Launcher**: responsible for triggering the test execution.
- **LauncherDiscoveryRequest**: used to define which tests should be run and to pass this information to the Launcher.
- **DiscoverySelectors**: provide a way to identify which tests to execute, in our case using the fully qualified method names.

- **ParameterResolver**: resolves and injects parameters into test methods at runtime.
- **TestTemplateInvocationContextProvider**: supplies the execution context for parameterized tests; in our setup, it works in tandem with the ParameterResolver.
- **SummaryGeneratingListener**: provides a summary of the results from the executed tests.

Together, these components enable us to dynamically discover test methods within the codebase, inject the correct user-defined parameters, and collect the test results.
In summary, JUnit5 provides the core infrastructure that enables dynamic, user-driven test execution in our framework. The flexibility and extensibility of JUnit5 make it the ideal choice for our non-traditional use case.
A more detailed explanation of how these components are integrated is presented in the following section about implementation details. This will also explain how the Reflection mechanism and custom annotations are employed to further customize the test execution process, helping us manage and execute dynamic tests effectively.

4.2 Implementation details

In this section, we delve into the technical mechanisms underlying the testing framework. We describe how tests are automatically discovered and executed without manual registration, the techniques employed to achieve this, and the constraints and design decisions that arise from these techniques. The focus is on presenting the practical challenges encountered during development and the solutions adopted to ensure a robust and scalable system.

4.2.1 Reflection

A fundamental aspect of the framework is the automatic retrieval of tests without manually listing them, in order to dynamically display the available tests on the dashboard.
To address this, we leveraged Reflection, a powerful technique provided by Java, that allows the inspection and manipulation of class metadata at runtime. With reflection, we can examine classes, interfaces, methods, fields, and annotations dynamically, without needing to know their specifics at compile time [52].
We developed a system in which the Model of the framework is populated by inspecting the tests package and its subpackages. During this inspection, we dynamically detect all the test classes and all the test methods contained within them.
The following code snippet illustrates the use of reflection to retrieve test classes and their public test methods:

Example 4.1: Example of use of reflection to populate TestClass and its methods

```
1  for (TestClass testClass : testClasses) {
2    try {
3        Class<?> clazz = Class.forName(testClass.getFullyQualifiedName());
4        for (Method method : clazz.getDeclaredMethods()) {
5
6            if (method.isAnnotationPresent(TestTemplate.class)) {
7                testClass.addMethod(new TestMethod(method));
```

```
            }
        }
    } catch (ClassNotFoundException e) {
        throw new RuntimeException("Test class not found: " +
          testClass.getFullyQualifiedName(), e);
    }
}
```

In this example, *Class.forName(...)* retrieves a Class object corresponding to a test class, such as BoardTest. To correctly load the class, the system requires the fully qualified name of the class, that is, the complete path including the package structure (e.g., tests.board.BoardTest). Once the class is loaded, reflection allows us to access its testing methods (identified by the @TestTemplate annotation), which are then added to the Model for display and execution.

This mechanism is extremely fast and powerful. However, it also introduces an important constraint: precision in the project's organization is critical. As detailed in Section 3.3, we enforce a strict convention where each concept has its own package and each test class is named consistently by appending Test to the chosen class name. This consistency is not optional since the dynamic discovery system depends on it.
The weakness of reflection mechanisms is that they are sensitive to structural changes. For example, if we decided to rename the class *BoardTest.java* to *TestBoard.java* without updating the system accordingly, the framework would no longer be able to find the test. Similarly, changing the directory structure would break the fully qualified name construction unless handled explicitly. Therefore, careful organizational decisions, such as the package and class naming strategy, were necessary from the outset.

4.2.2 Custom Annotation

Before diving into how our framework dynamically discovers and executes tests using JUnit5 componenets, it's important to explain a key technical feature we leverage: Java annotations.
Annotations allow to attach metadata to code elements, such as classes, methods, or parameters. JUnit5 relies heavily on annotations to define test behavior. In our framework, we also define a custom annotation to support a specific requirement: default parameters for tests.
As described in the View section 4.1.2, Ludii developers can configure test parameters through a dialog in the dashboard. In some cases, tests may offer default values for these parameters. Such values are automatically filled in when the test appears in the dashboard but developers can still modify them. To manage this, our model includes two fields, as reported in Model section 4.1.1: **value**, which stores the eventual value set by the user, and **defaultValue**, which stores the original default value defined by the test itself.
We needed a way to associate these default values directly with method parameters, and for that, we created the **@DefaultParameter** annotation, shown in 4.2.

Example 4.2: Custom Annotation: @DefaultParameter

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.PARAMETER)
public @interface DefaultParameter {

    String value();

```

```
}
```

The **@Retention(RetentionPolicy.RUNTIME)**[1] part is crucial, as it ensures that the annotation is available at runtime, which is necessary for our framework to inspect it via reflection. In fact, we use reflection to scan each test method's parameters. If a parameter is annotated with @DefaultParameter, we simply flag it as such. Here's the relevant code snippet:

Example 4.3: Reflection with Custom Annotation

```
    for(Parameter p: method.getParameters()) {

        String value = null;

        if(p.isAnnotationPresent(DefaultParameter.class)) {

          value = p.getAnnotation(DefaultParameter.class).value();
          parameters.put(p.getName(), new TestParameter(p.getType(), value, true));
          continue;

        }

        parameters.put(p.getName(), new TestParameter(p.getType(), value, false));

    }
```

The defaultValue plays an important role: when the user reruns a test or resets the UI, the original default values can be restored reliably.
For readers familiar with JUnit annotations, a natural question may arise: *why not use existing annotations such as **@ValueSource** or **@CsvSource** to specify default parameters?* While these annotations are available at runtime, they are not suitable for our needs. *@ValueSource* only supports methods with a single parameter, while *@CsvSource* requires specifying values for all parameters of a method. Neither of them allows associating a default value to an individual parameter independently. In contrast, our custom *@DefaultParameter* annotation directly associates a default value with a specific parameter, which is more flexible, cleaner and better suited for our case.

4.2.3 Discovery and Execution with JUnit5 components

To illustrate how JUnit5 components are used for test discovery and execution, we first present an example of how a test class with a test method should be written in our framework. The following example 4.4 shows a BoardTest class and a test that checks whether, if the Line concept is present in the game, its dimension is less than or equal to the maximum side of the board. The method uses the **@DefaultParameter** annotation, explained earlier, and the **@Tag** annotation, which is used to categorize the test. Both of these annotations are straightforward. However, two additional annotations are crucial: **@ExtendWith(ParametersContextProvider.class)** on the test class, and **@TestTemplate** on the test method.

Example 4.4: Example of a test in our framework

```
@ExtendWith(ParametersContextProvider.class)
public class BoardTest {
```

[1] docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/annotation/RetentionPolicy.html

```
3
4       @TestTemplate
5       @Tag("Static")
6       public void lineLessOrEqualThanBoardSide(String gameName,
7               @DefaultParameter("3") int lineLength) {
8
9           Game game = GameLoader.loadGameFromName(gameName);
10          BitSet concepts = game.computeBooleanConcepts();
11
12          if (!concepts.get(Concept.Line.id())) {
13              fail("Line concept is not present");
14          }
15
16          int side = game.board().graph().maxDim();
17          assertTrue(lineLength <= side, "Line should be less than or equal to board
              side");
18      }
19  }
```


A TestTemplate method, according to the JUnit5 specification, is designed to serve as a template for test cases that are invoked multiple times with different parameters. In our framework, every test is a template: what distinguishes each test instance are the runtime parameters.
In this example, the test includes a parameter to identify the game. This parameter is implicitly included in every test and allows the same method to be invoked across multiple games. While a @TestTemplate resembles a regular JUnit test, it requires additional setup to inject parameters. This is where the @ExtendWith(ParametersContextProvider.class) annotation plays a critical role.
As introduced earlier with the @DefaultParameter, parameter values are retrieved at runtime and this annotation serves as a wrapper for default values. This may raise a question: *how are parameters that are not marked as default, such as the String representing the game passed to the test when it is executed?*
The answer lies in the **@ExtendWith(ParametersContextProvider.class)** annotation. The ParametersContextProvider class extends TestTemplateInvocationContextProvider, described in Section 4.1.3, and is responsible for supplying the execution context for parameterized tests.
Before invoking the JUnit launcher, we must deliver this provider with a mapping of parameter values for each test method. Even when a test seems to have no parameters, our system *always* includes at least one implicit parameter: the name of the game. This provider also supplies a ParameterResolver for each test method, as well presented in Section 4.1.3, that parses and injects the correct parameter values at runtime. Since all values are received as strings from the user dashboard, parsing them into the correct types is essential. For this reason we store the expected type of each parameter using a *Class<?>* object, as described in Section 4.1.1.
Once the required parameters have been configured in the ParametersContextProvider, test discovery and execution proceed in a structured sequence. Each test method is identified using its fully qualified name, which is passed to DiscoverySelectors to locate the method in the codebase. These selectors are collected into a LauncherDiscoveryRequest, which is passed to the JUnit Launcher. Finally, execution results are collected via a TestSummaryListener, which extends SummaryGeneratingListener and records durations and failure messages.

Example 4.5: Sequence of steps to discover and execute tests using the JUnit5 Launcher API

```
ParametersContextProvider.setUserInputs(inputs);

LauncherDiscoveryRequest request = LauncherDiscoveryRequestBuilder.request()
      .selectors(selectorsList)
      .build();

TestSummaryListener listener = new TestSummaryListener();

launcher.execute(request, listener);

updateTestResults(selectedMethods, listener);
```


This chapter has illustrated the design and realization of the testing framework prototype, emphasizing its role in supporting Ludii game development through structured and automated testing. By adopting the MVC architecture and leveraging JUnit5, Java reflection, and both standard and custom annotations, the implementation ensures modularity, extensibility, and ease of integration. These design choices collectively contribute to a robust foundation for reliable and maintainable test development within the Ludii platform.

5

Validation

The objective of this thesis has been to develop a testing framework capable of supporting the TDGD methodology, which integrates the principles of TDD into the design and development of games. To achieve this, two core components were required: a graphical dashboard for interaction and a database of tests applicable to the wide range of games supported by the Ludii GGS.
Validating the solution entails confirming two primary aspects:

1. The GUI must be user-friendly, accessible, and capable of integrating and executing tests dynamically during development.
2. The database of tests must be sufficiently generic to support a wide variety of games.

The combination of these two elements enables the adoption of TDD in the context of game development within the Ludii platform.

5.1 GUI validation

The GUI was tested manually, focusing primarily on robustness in edge cases. The objective was to evaluate how the system behaves when interacting with unexpected or incomplete user inputs. Examples of tested scenarios include:

- Attempting to run tests without selecting any.
- Failing to provide a mandatory parameter.
- Interacting with disabled or unavailable options.

This approach allowed us to identify weak points in the interface and improve error handling and user feedback mechanisms. For instance, validation was added to prevent test execution without proper input, and messages were introduced to inform users of missing selections or invalid configurations.
The interface successfully handled all tested edge cases. The improvements introduced after testing have made the dashboard more stable and predictable in real usage scenarios, confirming that the GUI is functionally sound and provides a responsive and intuitive experience.

5.2 Test Generality Validation

To validate the generality of the tests, we used the export functionality (explained in the View section 4.1.2) to analyze test execution results across a large number of games. Specifically:

- Games tested: 1127
- Total tests executed: 7889
- Number of unique test types: 7 (static tests)

The validation focused on determining whether the current set of tests could apply across diverse game types without failure or adaptation. The selection of games included only a subset of the Ludii repository, focusing on board games, dominoes, math games, and puzzles. Despite the limited test suite, the large number of test runs made it possible to draw meaningful insights about test behavior and coverage. The overall test result yields a success rate of 36.68%, with an average execution time of 83 milliseconds per test.

As shown in Figure 5.1, four tests account for the majority of failures. Two of them, **sizeOfTrack** and **lineLessOrEqualThanBoardSide**, are customizable tests, requiring user-supplied parameters. The former checks the length of a track owned by a player, and thus needs both a player identifier and an expected track length. The latter ensures that any Line ludeme defined in the game does not exceed the longest side of the board. When running the tests across all games, we set these parameters to 0 by default to prevent missing-parameter errors. Consequently, many failures were expected.

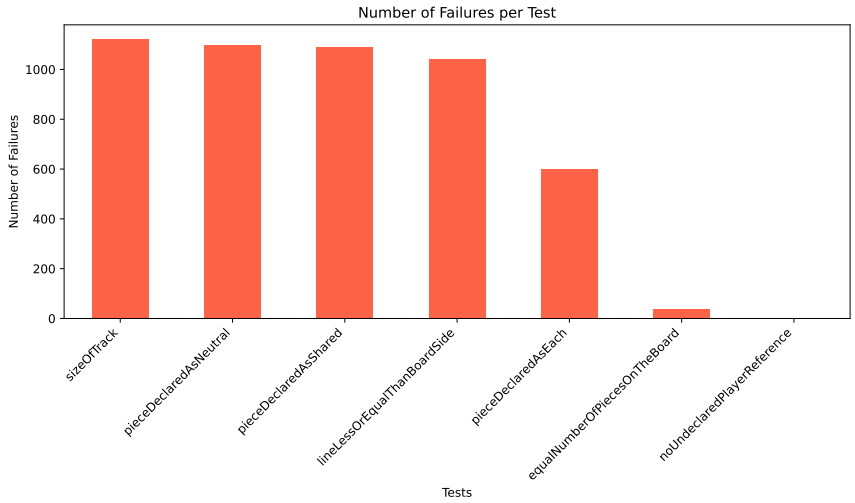

Figure 5.1: Number of failures per test

The other two high-failure tests, **pieceDeclaredAsNeutral** and **pieceDeclaredAsShared**, verify whether specific piece usage conventions are respected. These are general-purpose tests and apply across all games.

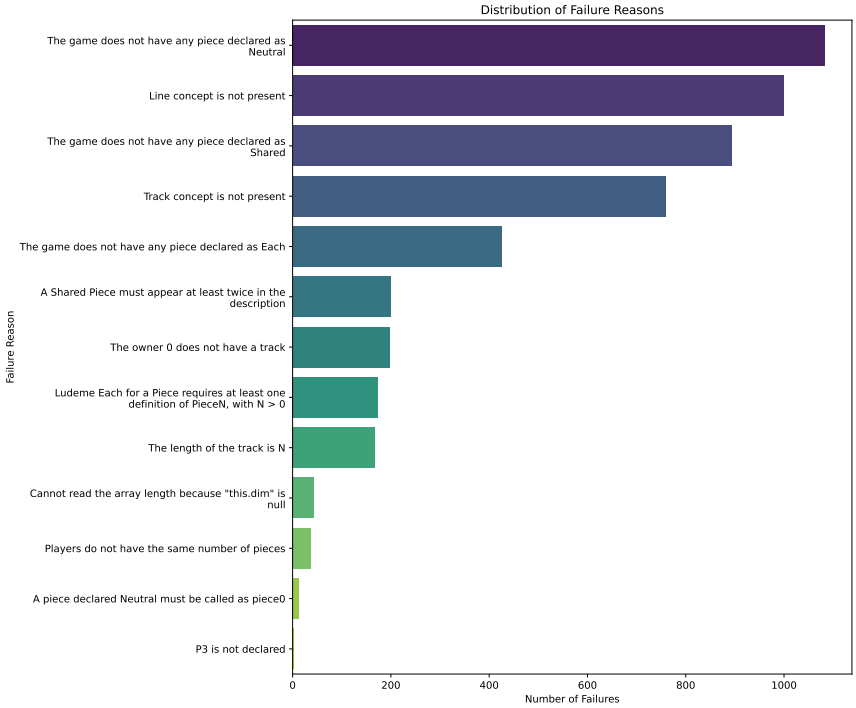

Figure 5.2: Failing reasons of tests

Figure 5.2 provides a breakdown of the primary failure reasons. The most common causes are that the concept targeted by the test (e.g. Track concept) or the specific feature (e.g. Neutral piece) are not present in the game. For instance, **sizeOfTrack** often fails simply because the game does not define a Track, and similarly with Line for **lineLessOrEqualThanBoardSide**. These are not actual errors, as Ludii supports a vast array of mechanics, and it is reasonable for many games to not have such features. Table 5.1 supports this observation. Among all failing tests, 83.29% are due to the targeted concept or feature missing, which implies that the majority of failures are expected and not indicative of incorrect logic.
In a TDGD workflow, such tests become particularly valuable. They are not merely checks for correctness, but instruments that actively drive the development. For example, if a Ludii developer knows that a piece should be declared as Shared, running the corresponding test will initially result in a failure, highlighting the missing concept. After modifying the game description accordingly, the developer can observe the test passing, confirming the correct implementation. Conversely, if the piece should not be declared as Shared, the test is expected to fail, confirming that the piece is indeed not marked as Shared. In this sense, test failures, when properly interpreted, can be used not only to detect errors but to confirm intentional design choices.
Excluding the concept-missing cases, we also observed input-related failures. For example,

in the sizeOfTrack test, the failure messages *The length of the track is N*[1] and *The owner 0 does not have a track*[2] suggest that the test logic works correctly, merely failing due to placeholder parameters. These results are encouraging, as they indicate that the test mechanism itself is sound.
Other failure types are more complex. At this stage, it remains unclear whether these failures are due to faulty test logic or incorrect game descriptions. Fortunately, they represent a minority of the total failures.

Table 5.1: Distribution of Specific Failure Messages

Failure Message	# Failing Tests	% of All Tests	% of Failed Tests
The game does not have any piece declared as Neutral	1084	13.74%	21.70%
Line concept is not present	999	12.66%	20.00%
The game does not have any piece declared as Shared	893	11.32%	17.88%
Track concept is not present	759	9.62%	15.20%
The game does not have any piece declared as Each	425	5.39%	8.51%
Total for these specific failures	4160	52.73%	83.29%

Running all tests enabled us to assess the reliability and robustness of the testing framework. Notably, this process revealed the presence of subgames in certain Ludii games that we did not consider. As discussed in Section 2.3.1, ludemes can be compound, meaning they may be composed of other ludemes, such as the subgame ludeme. Games constructed in this way require special handling, as many of their internal components are initialized only after the game is started. Initially, the tests failed to account for this behavior, relying solely on information available at compile-time, prior to execution. This resulted in unexpected failures and exceptions. We resolved the issue by ensuring such games are explicitly started before the test logic is applied.
This observation highlights a broader challenge: Ludii's GDL supports a wide variety of game representations, each with distinct mechanics and structural properties. As a result, building a generic and reusable test suite is non-trivial, since it must accommodate numerous scenarios and game constructs. Nevertheless, executing tests across a wide selection of games proves extremely beneficial, as it helps construct a more robust, general and comprehensive dataset for test development.
During the analysis of test failures, we discovered that certain valid game constructs were not correctly handled by our initial implementation. One such case involved the correct verification of piece declared as Each. When a piece is declared as Each (e.g. *(piece "Disc" Each)*), it means that each player owns their own instance of that piece. Usually, game descriptions refer to such instances with player-specific identifiers, such as Disc1 for player

[1]The test calculates the actual length of the track (e.g., The length of the track is 16); however, for presentation purposes, such messages were grouped under a single representative category.
[2]When the owner is 0, the track is shared among players.

1's Disc. Our test, pieceDeclaredAsEach, correctly checked for this behavior. However, this approach did not cover all valid uses of a piece declared as Each. For example, in some games, the Each piece is used in combination with the Hand ludeme. Even though the piece is correctly declared as Each, the game description may contain a line like *(place "Disc" "Hand")*, which places each player's own Disc into their hand. In such cases, identifiers like Disc1 do not appear; our initial test did not account for this, and incorrectly marked these descriptions as invalid. After updating the test to support this type of usage, our validation success rate improved by nearly 7%, a significant gain in both coverage and accuracy.
However, from the result summary in Figure 5.2, we observe that a considerable number of tests (almost 200) still fail due to what appears to be an incorrect interpretation of pieces declared as Each. This suggests that there may exist other valid ways to define or refer to such pieces, which remain unknown in the current specification. Further analysis will be necessary to identify these patterns and extend test coverage accordingly.

The validation results confirm that the proposed framework meets the essential requirements to support a TDD-oriented workflow in Ludii. The dashboard has proven to be both responsive and user-friendly, handling various edge cases effectively and providing reliable feedback during interaction.
As for the tests, although the current suite is limited in number, the majority have demonstrated both soundness and generality. Their successful execution across a wide range of games highlights their potential as foundational components of a broader and more comprehensive test database. These initial results are encouraging and establish a solid basis for future expansion and refinement of the testing suite.

6

Future Work

This chapter outlines the possible enhancements and open challenges for the testing framework. Building upon the limitations discussed previously, future work focuses on four main areas: refining the existing static tests, developing robust strategies for dynamic tests, improving the dashboard interface to better support users' needs, enabling users to create their own customizable tests. The following sections detail these priorities and the approaches considered for their resolution.

6.1 Static Tests Refinement

The previous chapter has highlighted certain limitations of the current prototype, which naturally suggest the next steps for improving the system.
First, the existing static tests require further analysis to clarify the causes of the remaining unexplained failures. It is currently unclear whether these failures are due to flaws in the test logic or inaccuracies in the game descriptions. A systematic investigation is necessary to distinguish between these two possibilities and refine both components accordingly.

6.2 Dynamic Tests

The current test database lacks dynamic tests, although their theoretical foundation and intended role were discussed extensively in their dedicated section. One key open problem in this context is determining the appropriate duration for test simulations. Specifically, it remains to be defined whether a test should require full-game simulations or if a limited number of moves (e.g., the first N moves) is sufficient to validate the intended conditions. To address this challenge, we propose the implementation of an accuracy configuration tool. This feature would allow Ludii developers to specify the desired accuracy threshold for a test. For example, a 100% accuracy requirement would result in simulations running until the end of the game or until the maximum number of trials permitted by Ludii is reached. Lower thresholds could result in shorter simulations, balancing performance with confidence in the test outcome.

This approach would provide flexibility in managing the trade-off between execution time and validation accuracy, enabling more practical and scalable use of runtime tests across a large number of games.
However, this solution raises an inherent challenge: how should the system determine the appropriate ratio between the number of simulated moves and the desired level of accuracy? Given the substantial diversity among games in Ludii, ranging from very short to extremely long and complex, there is no universally optimal value for such a ratio. In many games, reaching a conclusive end state may require a large number of moves, making full simulations impractical for routine testing.
For this reason, a more practical approach may be to allow the Ludii developer to explicitly specify either the number of moves to simulate or a concrete in-game condition that terminates the simulation (e.g., the first capture or the end of a phase). This strategy delegates the responsibility to domain experts who are best positioned to judge what constitutes a meaningful simulation for each individual game.

6.3 Dashboard Enhancements

Additional enhancements are planned for the dashboard component. First, we intend to expand the report creation functionality by allowing users to customize which information is included in the generated reports. Furthermore, we aim to augment the report with analytical metrics, such as the average number of tests per concept. This would provide insight into which concepts are thoroughly tested and which remain underrepresented, thereby guiding future test development.
Another significant improvement would be the implementation of test filtering within the dashboard, based on the concepts associated with a specific game. This would enable users to narrow down the set of relevant tests, especially useful in the presence of a large database. However, enabling such functionality requires extending the current model to store the set of concepts associated with each game.

6.4 Customizable Tests Creation

A key future enhancement is to enable users to create customizable tests directly through the GUI. This will involve integrating an AI-based test generation system that leverages a comprehensive test database once it is fully developed. The AI would learn from existing test patterns and understand the code structures required to generate new tests based on the database. Users would provide natural language prompts describing their testing objectives, and the AI would generate tests appropriately aligned with the game logic and user requirements. Among the challenges to address are ensuring the accuracy, reliability, and maintainability of AI-generated tests.
Such a concept is becoming increasingly viable due to recent advances in AI-assisted software engineering. Notable examples include:

- **TestForge** [53]: this system begins by zero-shot prompting a large language model (LLM) with the code under test. It then progressively refines the generated tests over multiple iterations, focusing on undercovered lines or regions with low mutation scores. TestForge integrates detailed execution feedback, including compilation

errors, runtime failures, and uncovered code segments, into an agentic feedback loop, enabling it to improve test quality iteratively.

- **Diffblue Cover**[1]: a reinforcement learning-based AI platform that autonomously generates comprehensive, human-readable unit tests for Java and Kotlin projects. It can automatically create and maintain an entire test suite as the software evolves. Similar to TestForge, Diffblue Cover follows an iterative process: it writes an initial test candidate for each method, evaluates coverage and correctness, and refines the tests until code coverage is maximized.
- **SmartUnit** [54]: originally developed for embedded software, SmartUnit consists of a dynamic symbolic execution engine, a unit test generator, and a cloud-based service. It analyzes control-flow and data-flow paths in the source code to generate effective test inputs, enabling high path coverage and early bug detection.

These represent only a subset of the potential improvements that can benefit our prototype testing framework. Numerous additional enhancements and modifications may be implemented to further advance our initial work, which aims to establish a solid foundation for a robust supporting tool within the Ludii GGS.

[1] www.diffblue.com

7

Conclusion

In the domain of GGSs, Ludii stands out for its expressive ludeme-based language, which enables a wide variety of games to be described in a human-readable yet semantically rich format. This flexibility makes Ludii accessible to a broad audience, including game historians and enthusiasts with limited or no programming experience. However, its current validation mechanisms are limited, relying solely on the execution of a few game trials and without offering sufficient guarantees regarding the correctness and robustness of complex game implementations.

This thesis addressed this limitation by introducing a dedicated testing framework and exploring the feasibility of adopting a TDD approach for game development. The framework features a dashboard-based interface and a structured dataset of tests, divided into two categories: static and dynamic. Static tests focus on compile-time validation, identifying syntactic and semantic issues in game descriptions. Dynamic tests, in contrast, require runtime evaluation over multiple game trials to verify specific game states and behaviors. The effectiveness of this solution was validated through several strategies. The dashboard demonstrated resilience to edge cases, providing responsive feedback and robust error handling. Furthermore, despite the limited number of static tests, the suite was executed across over a thousand Ludii games, achieving a success rate of 36.68%. While this figure may appear modest, it is important to note that 83.29% of the failures are expected, deriving from concepts or features that are deliberately excluded in game descriptions. These results confirm both the soundness of the test logic and its applicability across diverse game categories.

Looking forward, this project lays the groundwork for multiple promising extensions. A key priority is the full integration of dynamic tests, which will enhance the framework's ability to validate behavioral properties through simulation. Additionally, a systematic investigation is required to understand whether unexplained test failures are caused by inaccuracies in test logic or flaws in the underlying game descriptions.

Further development will also focus on improving the dashboard's usability and empowering users to define and run custom tests, thereby increasing the flexibility and adaptability of the framework.

In conclusion, this work establishes a foundation for reliable and automated validation within Ludii, representing a significant step toward bridging the gap between formal software engineering practices and general game design. By enabling structured and repeatable testing, it contributes to the development of more robust, maintainable, and accessible games within the Ludii GGS.

7

References

[1] Cameron Browne, Éric Piette, Matthew Stephenson, and Dennis J. N. J. Soemers. Ludii General Game System for Modeling, Analyzing, and Designing Board Games. In Newton Lee, editor, *Encyclopedia of Computer Graphics and Games*, pages 1082–1095. Springer International Publishing, Cham, 2023.

[2] Graham Todd, Alexander Padula, Matthew Stephenson, Eric Piette, Dennis J. N. J. Soemers, and Julian Togelius. GAVEL: Generating Games Via Evolution and Language Models. In *Proceedings of the Neural Information Processing Systems Conference (NeurIPS)*, 2024.

[3] Michael Genesereth and Yngvi Björnsson. The international general game playing competition. *AI Magazine*, 34(2):107–107, 2013.

[4] Cameron Bolitho Browne. *Automatic generation and evaluation of recombination games*. PhD thesis, Queensland University of Technology, 2008.

[5] Eric Piette, Matthew Stephenson, Dennis J. N. J. Soemers, Chiara Sironi, Mark H. M. Winands, and Cameron Browne. Ludii - the ludemic general game system. In *European Conference on Artificial Intelligence (ECAI)*, 2020.

[6] Eric Piette, Cameron Browne, and Dennis J. N. J. Soemers. Ludii Game Logic Guide. *arXiv:2101.02120*, 2021.

[7] Dennis J. N. J. Soemers, Eric Piette, Matthew Stephenson, and Cameron Browne. Ludii User Guide. `https://ludii.games`, 2019.

[8] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2022.

[9] Jacques Pitrat. Realization of a general game-playing program. In *IFIP Congress (2)*, pages 1570–1574, 1968. (Cited on pages 3, 48, and 275).

[10] Michael Genesereth and Michael Thielscher. *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, San Rafael, CA, 2014. (Cited on pages 49 and 80).

[11] Cameron Browne, Dennis J. N. J. Soemers, Eric Piette, Matthew Stephenson, and Walter Crist. Ludii Language Reference. `https://ludii.games/downloads/LudiiLanguageReference.pdf`, 2020.

[12] Ben Goertzel and Cassio Pennachin. *Artificial general intelligence*, volume 2. Springer, 2007.

[13] Cameron Browne, Julian Togelius, and Nathan Sturtevant. Guest editorial: General games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(04):317–319, 2014.

[14] Matthew Stephenson, Dennis J. N. J. Soemers, Eric Piette, and Cameron Browne. Measuring Board Game Distance. In *Computer Games (CG)*, 2022.

[15] Anders Kierulf, Ken Chen, and Jurg Nievergelt. Smart game board and go explorer: A study in software and knowledge engineering. *Communications of the ACM*, 33(2):152–166, 1990.

[16] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62–62, 2005.

[17] Frédéric Koriche, Sylvain Lagrue, Eric Piette, and Sébastien Tabary. WoodStock: Un programme-joueur générique dirigé par les contraintes stochastiques. *Revue d'Intelligence Artificielle (RIA)*, 2017. Numéro spécial "IA des jeux informatisés".

[18] Frédéric Koriche, Sylvain Lagrue, Eric Piette, and Sébastien Tabary. Constraint-based symmetry detection in general game playing. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[19] Eric Piette. *Une nouvelle approche au General Game Playing dirigée par les contraintes*. Phd thesis, Université d'Artois, France, 2016.

[20] Jakub Kowalski, Maksymilian Mika, Jakub Sutowicz, and Marek Szykuła. Regular boardgames. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1699–1706, 2019.

[21] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, Daniel Hennes, Dustin Morrill, Paul Muller, Timo Ewalds, Ryan Faulkner, János Kramár, Bart De Vylder, Brennan Saeta, James Bradbury, David Ding, Sebastian Borgeaud, Matthew Lai, Julian Schrittwieser, Thomas Anthony, Edward Hughes, Ivo Danihelka, and Jonah Ryan-Davis. Openspiel: A framework for reinforcement learning in games, 2020.

[22] Tristan Cazenave, Yen-Chi Chen, Guan-Wei Chen, Shi-Yu Chen, Xian-Dong Chiu, Julien Dehos, Maria Elsa, Qucheng Gong, Hengyuan Hu, Vasil Khalidov, Cheng-Ling Li, Hsin-I Lin, Yu-Jin Lin, Xavier Martinet, Vegard Mella, Jeremy Rapin, Baptiste Roziere, Gabriel Synnaeve, Fabien Teytaud, Olivier Teytaud, Shi-Cheng Ye, Yi-Jun Ye, Shi-Jim Yen, and Sergey Zagoruyko. Polygames: Improved zero learning, 2020.

[23] Stephan Schiffel and Michael Thielscher. Representing and reasoning about the rules of general games with imperfect information. *Journal of Artificial Intelligence Research*, 49:171–206, 2014.

[24] Michael Thielscher. GDL-III: a description language for epistemic general game playing. *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 2017.

[25] Cameron Browne, Matthew Stephenson, Eric Piette, and Dennis JNJ Soemers. A practical introduction to the ludii general game system. In *Advances in Computer Games*, pages 167–179. Springer, 2019.

[26] Matthew Stephenson, Eric Piette, Dennis JNJ Soemers, and Cameron Browne. An overview of the ludii general game system. In *2019 IEEE Conference on Games (CoG)*, pages 1–2. IEEE, 2019.

[27] Matthew Stephenson, Eric Piette, and Cameron Browne. Teaching and Learning with LUDII. *Board Game Studies (BGS)*, 2019.

[28] Walter Crist, Matthew Stephenson, Eric Piette, and Cameron Browne. The ludii games database: A resource for computational and cultural research on traditional board games. *DHQ: Digital Humanities Quarterly*, 18(4), 2024.

[29] Walter Crist, Eric Piette, Dennis J. N. J. Soemers, Matthew Stephenson, and Cameron Browne. Computational Approaches for Recognising and Reconstructing Ancient Games: The Case of Ludus Latrunculorum. In The Archaeology of Play: Material Approaches to Games and Gaming in the Ancient World, Véronique Dasen and Marco Vespa (Eds.). Oxbow, Oxford, 2024.

[30] Eric Piette, Lisa Rougetet, Walter Crist, Matthew Stephenson, DJNJ Soemers, and Cameron Browne. A ludii analysis of the French Military Game. *Board Game Studies (BGS)*, 2021.

[31] Dennis J. N. J. Soemers, Jakub Kowalski, Walter Crist, Summer Courts, Tim Penn, and Eric Piette. Bridging AI and Cultural Heritage: Outcomes from the GameTable WG1 London Meeting. *International Computer Games Association Journal (ICGA)*, 2025.

[32] Cameron Browne. A class grammar for general games. In Aske Plaat, Walter Kosters, and Jaap van den Herik, editors, *Computers and Games*, pages 167–182, Cham, 2016. Springer International Publishing.

[33] A. Borvo. *Anatomie d'un jeu de cartes: L'Aluette ou le jeu de Vache*. Librairie Nantaise Yves Vachon, Nantes, 1977.

[34] Cameron Browne. Everything's a Ludeme Well, Almost Everything. In *XXIII BOARD GAME STUDIES COLLOQUIUM-The Evolutions of Board Games*, 2021.

[35] Eric Piette, Matthew Stephenson, Dennis JNJ Soemers, and Cameron Browne. An empirical evaluation of two general game systems: Ludii and RGB. In *IEEE Conference on Games (CoG)*, pages 1–4, 2019.

[36] Matthew Stephenson, Eric Piette, Dennis J. N. J. Soemers, and Cameron Browne. Automatic generation of board game manuals. In *Advances in Computer Games (ACG)*, 2021.

[37] Cameron Browne and Eric Piette. Digital Archaeoludology. *Computer Applications and Quantitative Methods in Archaeology (CAA)*, 2019.

[38] Cameron Browne, Dennis J. N. J. Soemers, Éric Piette, Matthew Stephenson, Michael Conrad, Walter Crist, Thierry Depaulis, Eddie Duggan, Fred Horn, Steven Kelk, Simon M. Lucas, João Pedro Neto, David Parlett, Abdallah Saffidine, Ulrich Schädler, Jorge Nuno Silva, Alex de Voogt, and Mark H. M. Winands. Foundations of Digital Archæoludology. Report, Dagstuhl Research Meeting, 2019.

[39] Eric Piette, Walter Crist, Dennis J. N. J. Soemers, Lisa Rougetet, Summer Courts, Tim Penn, and Achille Morenville. GameTable COST Action: kickoff report. *International Computer Games Association (ICGA) Journal*, 2024.

[40] Eric Piette, Achille Morenville, Barbara Care, Dorina Moullou, and Walter Crist. Ai-powered game recognition: A collaborative dataset for traditional games. In *Computer Applications and Quantitative Methods in Archaeology (CAA)*, 2025.

[41] Dennis J.N.J. Soemers, Eric Piette, Matthew Stephenson, and Cameron Browne. The ludii game description language is universal. In *IEEE Conference on Games (CoG)*, 2024.

[42] Achille Morenville and Eric Piette. Belief Stochastic Game: A Model for Imperfect-Information Games with Known Positions. In *Computer and Games (CG)*, 2024.

[43] Eric Piette, Matthew Stephenson, Dennis JNJ Soemers, and Cameron Browne. General board game concepts. In *IEEE Conference on Games (CoG)*, pages 01–08, 2021.

[44] D. Janzen and H. Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005.

[45] Kent Beck. Aim, fire [test-first coding]. *IEEE Software*, 18(5):87–89, 2001.

[46] Dua Agha, Rashida Sohail, Areej Meghji, Ramsha Qaboolio, and Sania Bhatti. Test driven development and its impact on program design and software quality: A systematic literature review. *VAWKUM Transactions on Computer Sciences*, 11:268–280, 06 2023.

[47] Noel Llopis and Sean Houghton. Backwards is forward: Making better games with test-driven development. Game Developers Conference, 2006. `https://www.convexhull.com/articles/tdd_gdc06.pdf`.

[48] Dennis J.N.J. Soemers, Eric Piette, Matthew Stephenson, and Cameron Browne. Optimised Playout Implementations for the Ludii General Game System. *Advances in Computer Games (ACG)*, 2021.

[49] Vincent Massol. *JUnit in action*. Citeseer, 2004.

[50] James Bucanek. Model-view-controller pattern. *Learn Objective-C for Java Developers*, pages 353–402, 2009.

[51] Frank Appel. *Testing with JUnit*. Packt Publishing Ltd, 2015.

[52] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., USA, 2004.

[53] Kush Jain and Claire Le Goues. Testforge: Feedback-driven, agentic test suite generation, 2025.

[54] Chengyu Zhang, Yichen Yan, Hanru Zhou, Yinbo Yao, Ke Wu, Ting Su, Weikai Miao, and Geguang Pu. Smartunit: Empirical evaluations for automated unit testing of embedded software in industry. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 296–305, 2018.

Glossary

AGI Artificial General Intelligence.

AI Artificial Intelligence.

API Application Programming Interface.

GDL Game Description Language.

GGP General Game Playing.

GGS General Game Systems.

GUI Graphical User Interface.

IDE Integrated Development Environment.

MVC Model-View-Controller.

RBG Regular Boardgames.

TDD Test-Driven Development.

TDGD Test-Driven Game Development.

List of Figures

Listings

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**