PPS23-24: Card Game Engine

Index

The development process artifacts can be found here.

Introduction

The Card Game Engine is a powerful domain-specific language (DSL) designed to empower its users to easily define a given card game using easy to read syntax.

Example

The following is an example of the game marafone defined with the Card Game Engine DSL:

```
game is "Marafone"
game has player called "Player 1"
game has player called "Player 2"
game has player called "Player 3"
game has player called "Player 4"
game gives 10 cards to each player
game starts from player "Player 1"
game suitsAre ("Batons", "Coins", "Cups", "Swords")
game ranksAre ("Ace", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "
```

```
game trumpIs "Batons"
game playing rules are:
  (table, player, playerCard) =>
    player.hand.cards.contains(playerCard) &&
    (table.cardsOnTable.isEmpty ||
    (table.cardsOnTable.head.suit == playerCard.suit || player.hand.cards.filter(_.suit

game hand rules are:
  (hand, cardPlayed, trump, ranks) =>
    val handSuit = hand.head.suit
    val t = trump.getOrElse("")
    val trumpsInHand = hand.filter(_.suit == t)
    val leadingSuitsInHand = hand.filter(_.suit == handSuit)
    val winningCard = if (trumpsInHand.nonEmpty) then
      trumpsInHand.maxBy(c => ranks.indexOf(c.rank))
    else
      leadingSuitsInHand.maxBy(c => ranks.indexOf(c.rank))
    cardPlayed == winningCard
game win conditions are:
  (g, p) =>
    val nordSudCouple = List(g.players(0), g.players(2))
    val eastWestCouple = List(g.players(1), g.players(3))
    val playerCouple = if nordSudCouple.contains(p) then nordSudCouple else eastWestCoup
    val enemyCouple = if nordSudCouple.contains(p) then eastWestCouple else nordSudCoupl
    playerCouple.map(_.points).sum > enemyCouple.map(_.points).sum
GameController(game.build).startGame
```

DSL Syntax

Game Name

The game name can be defined as follows:

```
game is "<gameName>"
```

Where <gameName> is the name of the game.

Players

Player Name A player can be defined as follows:

```
game has player called "<playerName>"
```

Where <playerName> is the name of the player.

The game can have from 1 to 4 players.

Starting Player The starting player is defined as follows:

```
game starts from player "<playerName>"
```

Where <playerName> is the name of the starting player.

The starting player can only be defined after defining the players' names.

Starting Hand The number of cards in the starting hand of every player can be defined as follows:

```
game gives <numberOfCards> cards to each player
```

Where <numberOfCards> is the number of cards in hand for every player.

Also the game can also be defined to give a random number of cards to every player as follows:

```
game gives random cards to each player
```

Cards

Suits And Ranks The cards' ranks and suits can be defined as follows:

```
game suitsAre ("<suit1>", "<suit2>", "<suit3>", "<suit4>")
game ranksAre ("<rank1>", "<rank2>", "<rank3>", "<rank4>")
```

Where:

- <suit[1..4]> is a card suit
- <rank[1..4]> is a card rank

As a good practice ranks' order defines their relative strenght.

This means that the example defines the <rank1> to be the strongest rank and <rank4> the weakest.

Trump The game trump can be defined as follows:

```
game trumpIs "<trumpSuit>"
```

Where <trumpSuit> is the suit selected as trump.

The trump suit must be defined after defining the game suits.

The trump suit definition is not mandatory.

Game Rules

Game rules are defined as follows:

```
game playing rules are:
    <rule1>, <rule2>, <rule3>

game hand rules are:
    <rule1>, <rule2>, <rule3>
```

```
game win conditions are:
    <rule1>, <rule2>, <rule3>
```

Where <rule1><rule2><rule3> are Scala 3 lambdas that define the game rules.

See the game definition example for rules parameters.

Final Line

Every CGE game definition must end with the following line to run the game.

```
GameController(game.build).startGame
```

Back to index | Next Chapter

Development Process

- Scrum
- Test-Driven Development
- Git Flow
- Commit Standardization
- Quality Assurance
  - Scoverage
  - Scalafmt
  - Wartremover
  - Scalafix
- Build Automation
- Continuous Integration

Scrum

The development process choosen for this project is SCRUM(SCRUM Primer), an agile project management framework.

As per agile development, SCRUM defines a loose set of activities that enable the development team to embrace an iterative and incremental process.

The development process will be devided sprints of the lenght of one week each. Every sprint starts with a meeting, called sprint planning, where items from the product backlog will be selected to be implemented and expandend in multiple items, adding up to form the sprint backlog. The team members volontier to work on a given item and progress is tracked in the sprint backlog.

At the end of a given sprint a meeting takes place with 3 goals:

- Product Backlog Refinement;
- Sprint Review;
- Sprint Retrospective.

Members of the development team will also roleplay as:

- Product Owner (Leonardo Randacio) : responsible for maximizing the value brought by the product.
- Client (Filippo Gurioli) : the eventual buyer and founder for the project, responsible for ensuring it meets the requirements.

Dialy SCRUM meetings have been ditched for efficient indipendent work. A meeting will take place once a week to close the previous sprint and to start the next one.

Trello will be used for coordination, while meeting will take place through Google Meet videocalls.

Test-Driven Development

Test-Driven Development(TDD) will be followed during code production throughout the project.

TDD is a software development approach here developers write tests for a feature or function before implementing the actual code. The process follows a simple cycle, often referred to as "Red-Green-Refactor":

- Red (Write a failing test): Write a test for a specific functionality that doesn't exist yet. Since the functionality hasn't been implemented, the test will fail.
- Green (Write the minimal code to pass the test): Write just enough code to make the test pass. The focus here is on passing the test, not on perfect code.
- Refactor (Improve the code): Once the test passes, refactor the code to improve its structure, readability, and maintainability while ensuring the tests continue to pass.

For time constraints reasons the testing will not cover the GUI part of the project.

Git Flow

Git flow branching model will be used.

Two main branches will always exist:

- `main` : for stable releases;
- `develop` : for feature integration.

Feature branches can be created, branching from the `develop` branch, to implement new fetures. Once a new feature is implemented it should be merged into the `develop` branch. To create a new release the `develop` branch is merged into the `main` branch.

Hotfix and fix branches can be branched from the `main` and `develop` branches accordingly for bug fixes.

gitflow command line tool will be used for high-level repository operations.

Commit Standardization

Commit messages must follow a standardized form:

`<type>: <short summary>`

Where `type` is one of the following types: - feat: A new feature is introduced to the codebase; - fix: A bug fix; - docs: Documentation updates only; - style: Changes related to formatting, like spacing or indentation, that don't affect the code's logic; - refactor: Code changes that neither fix a bug nor add a feature, typically done to improve the structure; - test: Adding or updating tests; - chore: Routine tasks like updating dependencies or configuration; - perf: Code changes aimed at improving performance.

and `short summary` is a short summary of the commit contents.

Quality Assurance

The following Quality Assurance tools have been considered:

- Scala formatter
- Wart Remover
- Scala style
- Scala Fix
- Scalatest
- Scoverage
- CPD

Scala style and CPD have been discarded due to compatibility issues.

Scoverage

Default configuration has been used.

Scoverage allows to check for the test coverage percentage. This is also used in the Continuous Integration pipeline.

Scoverage doc

Scalafmt

It works flawlessly with the Metals plugin for VSCode. Using `Alt + Shift + F` will automatically update the file formatting according to the provided

instructions in the scalafmt config file. It's still possible to do it manually using the following commands: - `./gradlew scalafmt` formats your scala and sbt source code (main sourceset only); - `./gradlew checkScalafmt` checks whether all files are correctly formatted, if not, the task fails (main sourceset only); - `./gradlew testScalafmt` formats your test scala code based on the provided configuration; - `./gradlew checkTestScalafmt` checks whether your test scala code is correctly formatted; - `./gradlew scalafmtAll` formats scala code from all source sets; - `./gradlew checkScalafmtAll` checks formatting of all source sets.

Scalafmt doc

Wartremover

It works flawlessly with the Metals plugin for VSCode.

It finds warts in the code defined in the wartremover config file.

WartRemover doc

Scalafix

It is a code linter with configuration defined in the scalafix config file.

It can be run using the comand `gradle scalafix`.

Scalafix doc

Build Automation

Gradle has been chosen as the build automation tool.

Continuous Integration

To ensure code correctness and integrity a Continuous Integration pipeline has been setup, using Github Actions, to execute the test on various operative systems every code update.

CI is configured in the github actions config file

Back to index | Previous Chapter | Next Chapter

Requirements

- Business Requirements
- Domain Requirements
- Functional Requirements
  - User Functional Requirements
  - System Functional Requirements

- Non-Functional Requirements
- Implementation Requirements

During project analysis the following requirements have been identified.

Business Requirements

- The DSL should allow users to express card game rules concisely and intuitively.
- The DSL should be extensible to support various card game genres (e.g., trick-taking games, deck-building games).

Domain Requirements

- The DSL must represent card decks, including standard playing cards (52 cards) and any custom decks.
- The DSL should handle card ranks (e.g., Ace, King, Queen) and suits (e.g., Hearts, Spades).
- Users can define custom card decks by designing custom ranks and suits.
- Users can express game rules (e.g., how cards are played, winning conditions) using a concise syntax.

Functional Requirements

User Functional Requirements

- Users should be able to easily define custom card sets and rules
- Users should be able to define game rules such as:
  - deck composition (seeds and ranks);
  - card value for hand winning;
  - trump selection through player decision;
  - seed based card play restrictions;
  - win condition:
    - number of hands won;
    - score bas- draw decks and draw rules;
  - card play/discard;
  - play cards to the table in other ways than simply playing a hand, such as:
    - placing them on the table;
    - giving them to other players;ed on cards won;

System Functional Requirements

- The system should validate whether a played card follows the game rules.
- The system should manage the game state (e.g., player turns, scores).

Non-Functional Requirements

- Readability: The DSL syntax should be human-readable and expressive.
- Modularity: The DSL should allow composing rules from reusable components (e.g., reusable card effects).
- Performance: The DSL library should execute rules efficiently during gameplay.
- Error Handling: The DSL should provide clear error messages for invalid rule definitions.

Implementation Requirements

- Use of Scala 3.x.
- Use of JDK 11+.
- The DSL should be implemented as an internal DSL in Scala.
- The DSL should use Scala's expressive syntax to create a natural language-like experience.
- The DSL should provide a clear separation between the DSL definition and the game logic.
- The codebase should be well-organized, documented, and maintainable.

Back to index | Previous Chapter | Next Chapter

Architectural Design

- DSL
- Engine
  - GameModel
  - GameView
  - GameController

The project can be divided into two main parts:

- **DSL**: Handles the Domain Specific Language's custom syntax and changes the internal model state. It enables the user to define the rules of the card game.
- **Engine**: Enables the user to play the defined game. It is structured following the Model-View-Controller pattern:
  - **Model**: Manages the data of the Engine. It responds to requests for information and updates the data when instructed.
  - **View**: Manages the GUI and displayes the data from the model to the user. It updates when data in the Model changes and sends user actions to the Controller.
  - **Controller**: Acts as an intermediary, handling input from the View and updating the Model. It manages the flow between the View and Model, ensuring the right data and responses are presented to the user.

The user defined rules for the game are stored entirely in the Model, while the Controller and View are constant for every game iteration. This ensures that the View and Model have no dependency and updates to one of them would only require implementation of new features to them selves or at most the Controller.

TODO add simple uml of the project.

DSL

The DSL provides a Domain Specific Language to enable the user to define a card game using domain terminology.

The game being defined is then created using the Engine and run, allowing the user to play the card game.

The DSL uses a builder pattern to create a model that is then passed to the controller, which starts the game.

High level UML diagram of the dsl architecture.

Figure 1: dsl

Engine

High level UML diagram of the game engine architecture.

GameModel

The Model defines the data rappresentation for every main concept in the game.

It is queried by the Controller when data is needed and it is updated to reflect changes to the data requested by the Controller.

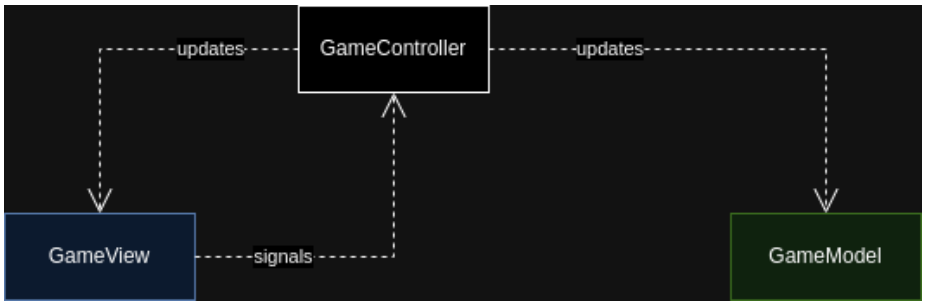

Figure 2: engine

GameView

The View defines the GUI that displays the model contents to the user and allowes the players to interact with the game.

A tecnology must be choosen during detailed design for the GUI implementation.

GameController

The Controller is an intermidiary between the Model and the View.

It defines the flow of the game extracting the rules defined by the user from the Model and using the View to display the current game state.

Back to index | Previous Chapter | Next Chapter

Detailed Design

- DSL
- Engine
  - GameModel
  - GameView
  - GameController
  - GameBuilder

Scala 3 and Java 21 are choosen as programming languages for the project.

DSL

The main goal of the DSL part is to let the user specify parameters and settings, and to prepare a `GameBuilder` with them.

The following UML diagram illustrates the main architecture of the DSL part.

`CardGameEngineDSL` represents the entry point for every DSL sentence: using some pre-defined *words* from `SyntacticSugar` and providing specific builders

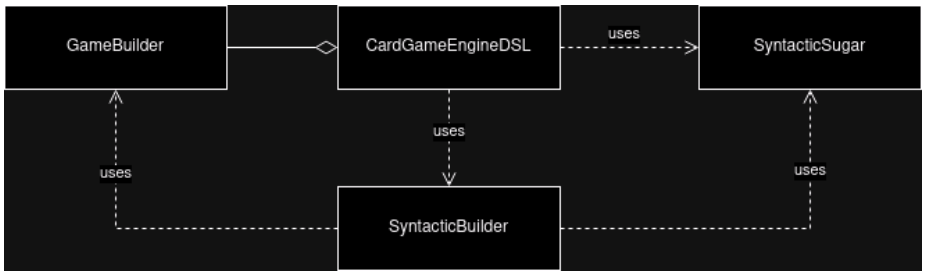

Figure 3: dsl

to lock sentences syntax, it is possible to create a chain of commands changing the builder's internal state. The implementation phase is developed as a direct extension of the design section: each element defined here serves as a clear guide for implementation.

Engine

Java Swing is used as technology for the view implementation.

Below the UML diagram of the engine.

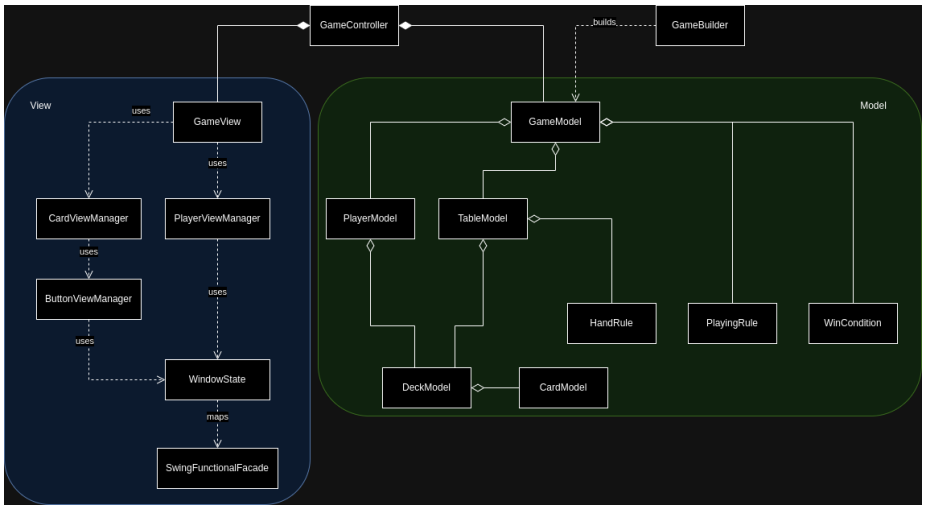

Figure 4: engine

GameModel

The Model defines the following concepts:

- `CardModel` : rappresents a card in the game.

- `DeckModel` : rappresents a collection of `CardModel`.
- `TableModel` : rappresents the table, where cards are placed once played, and the rules for winning a given hand.
- `PlayerModel` : rappresents a single player.
- `HandRule` : defines a rule for winning hand in the game.
- `PlayingRule` : defines a rule for what cards can be played at a given model state.
- `WinCondition` : defines a rule for how the winners of the game are selected.

GameView

The View is made functional using a State Pattern.

The `SwingFunctionalFacade` is mapped by `WindowState` to expose the Swing GUI methods in a functional manner.

The `CardViewManager` and `PlayerViewManager` objects provide functions for adding a player to the GUI and adding and removing cards from the GUI respectively.

The cards are rappresented as buttons in the GUI so the `CardViewManger` uses a `ButtonViewManager`.

GameController

The Controller takes a prebuilt Model of the game and creates an initial State to be run as a sequence of updates that generate the initial View.

The Controller then defines the State updates depending on the events published by the View (aka the user interaction).

As a final step the Controller runs the composed State of the View.

On events being published by the View the Controller updates the model accordingly and then updates the State of the View.

GameBuilder

The Builder takes the various game parameters and then builds the correct initial Model accordingly.

Back to index | Previous Chapter | Next Chapter

Implementation

- Biagini Andrea
- Gurioli Filippo
- Randacio Leonardo
- Pair Programming

– GameController

In the following every member of the team details the parts of the project that they have developed

Biagini Andrea

Gurioli Filippo

Randacio Leonardo

Pair Programming

The following parts have been developed in pair programming.

GameController

The `GameController` has been developed in Pair Programming by Filipo Gurioli and Leonardo Randacio.

The Controller takes a Model and uses it to build the initial View State. It defines the methods to be run on every event published by the View and then runs the initial View State.

The Controller parses the events published by the View playing a given card if a player is trying to play it.

```
val windowEventsHandling: State[Window, Unit] = for
  events <- windowCreation
  _ <- seqN(events.map(e =>
    val parsedEvent = e.split(":")
    val eventName = parsedEvent(0)
    eventName match
      // Checking if eventName is the name of one of the players.
      case s if game.players.map(_.name).contains(eventName) =>
        handleCardPlayed(s, parsedEvent(1))
      case _ =>
        // If the eventName does not correspond to a player name
        // a state that does not update the GUI is returned.
        unitState()
  ))
yield ()
```

The `handleCardPlayed` function checks if the player that played the card is the turn player and can play that card at that moment.

```
private def handleCardPlayed(playerName: String, cardFromEvent: String): State[Window, Uni
  val rank = cardFromEvent.split(" ")(0)
  val suit = cardFromEvent.split(" ")(1)
  val card = CardModel(rank, suit)
```

```
    game.players.find(_.name == playerName) match
      case Some(player) =>
        if game.turn.name != playerName || !game.canPlayCard(game.turn, card) then
          unitState()
        else playCard(player, card)
      case None =>
        throw new NoSuchElementException(s"Player $playerName not found")
```

The `playCard` function updates the Model and returns the correct state update, depending on if the hand has finished or not.

```
  private def playCard(player: PlayerModel, card: CardModel) =
    turnCounter += 1
    game.playCard(player, card)
    if turnCounter == game.players.size then
      turnCounter = 0
      moveCardToTable(player, card).flatMap(_ =>
        endHand())
    else moveCardToTable(player, card)
```

Back to index | Previous Chapter | Next Chapter

Andrea Biagini

In the project implementation, I contributed working on `GameModel` and its related objects inside the Model part.

GameModel

The structure of a GameModel mainy adheres to the object oriented paradigm: it contains all the necessary information related to a card game. A more functional structure can be seen in the implementation of `WinConditions`, `HandRules` and `PlayingRules`.

The following diagram express relations between the elements composing `GameModel`.

Each element of the game model part is implemented hiding different implementations and trying to separate simplet concepts from the more advanced ones. Here is the UML class diagram of `GameModel`, which has a structure analogous to `TableModel`, `PlayerModel`, `DeckModel`, and `CardModel`.

WinCondition, HandRule and PlayingRule

The implementations of `WinCondition`, `HandRule`, and `PlayingRule` are based on the same idea of using lambdas: each of these names is a type alias for a

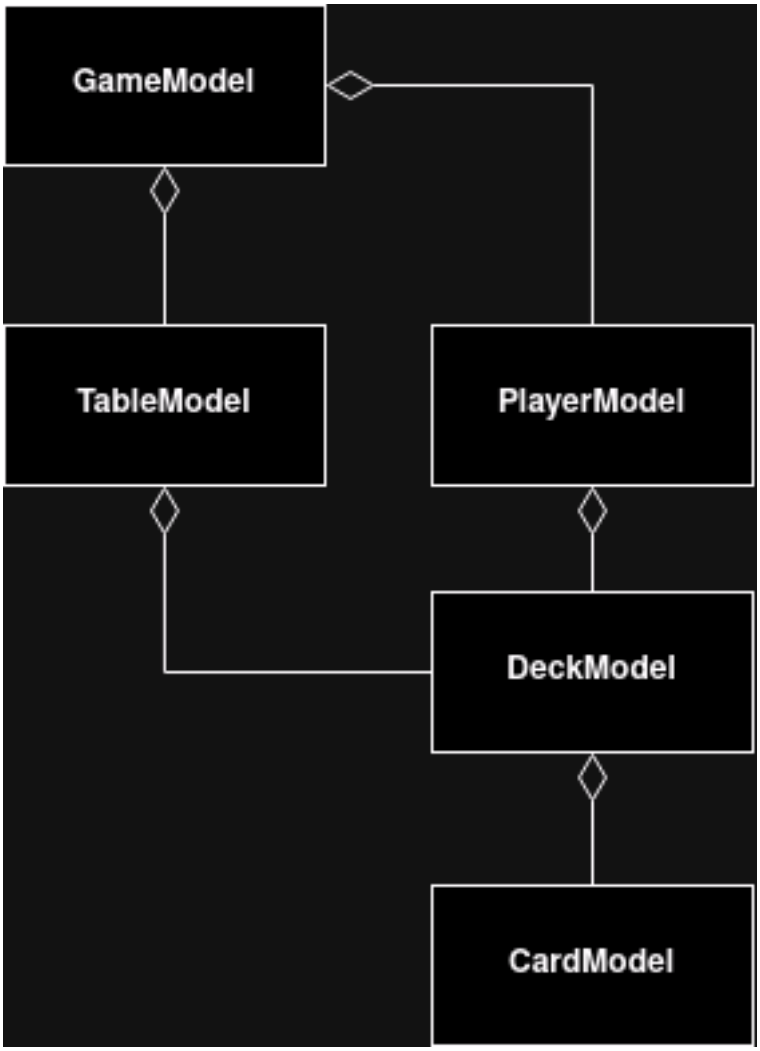

Figure 5: GameModel diagram

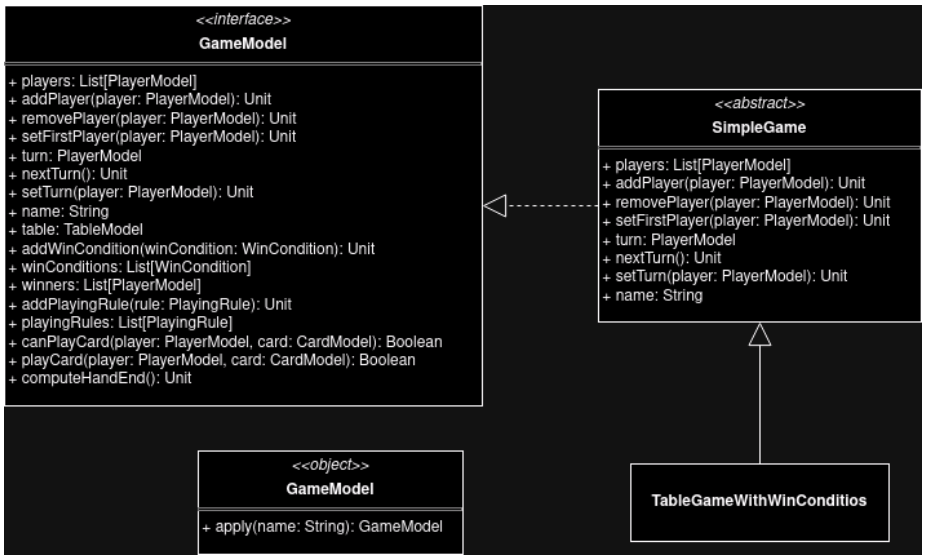

Figure 6: GameModel implementation structure

lambda that takes various objects as arguments, which might be necessary to define the rules.

With `WinCondition`, a win condition for the game can be expressed based on the entire game model and the player for whom we want to check for a win. The lambda is structured as follows:

```
(GameModel, PlayerModel) => Boolean
```

and returns `true` if the specified player is the winner of the game, `false` otherwise.

With `HandRule`, a rule can be expressed to determine if a player wins a hand based on the cards on the table, the card that the specified player has played, the trump suit of the game, and the ordered list of ranks in the deck being used. The lambda is structured as follows:

```
(List[CardModel], CardModel, Option[Suit], List[Rank]) => Boolean
```

and returns `true` if the specified card is the winner of the current hand, `false` otherwise.

With `PlayingRule`, it's possible to specify a rule that determines whether a player is allowed to play a certain card based on the current state of the table, the current player, and the card they intend to play. The lambda is structured as follows:

```
(TableModel, PlayerModel, CardModel) => Boolean
```

and returns `true` if the specified player is permitted to play the specified card, `false` otherwise.

Back to index | Back to implementation

Filippo Gurioli

Overview

My work focuses on implementing a Domain-Specific Language (DSL) that allows users to naturally express the definitions, rules, and behaviors of a card game.

Design

Based on the fact that the model includes a `GameModel`, which is a data structure containing all the necessary information to define a card game, the DSL must essentially act as a builder for this class. A builder is an object capable of incrementally constructing another class, in this case, `GameModel`.

Given the complexity of this task, the work has been divided into two parts:

- `GameBuilder`: The actual builder that encapsulates the logic to construct a `GameModel`.
- `CardGameEngineDSL`: Extension methods for `GameBuilder` that utilize the fluent interface pattern and other techniques to create human-readable sentences.

Below is an initial schematic:

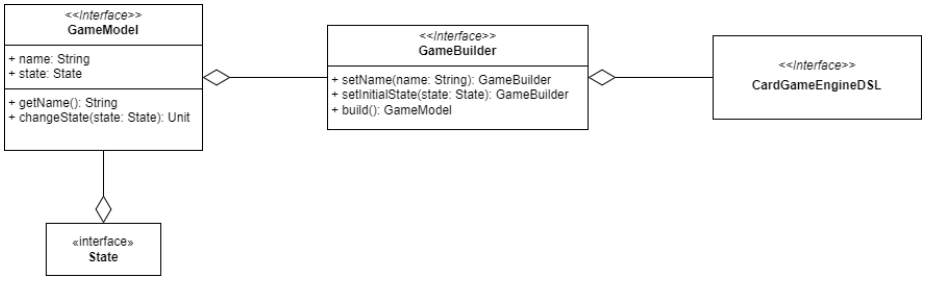

Figure 7: Game Builder Overview

GameBuilder

The `GameBuilder` class is capable of creating a consistent and ready-to-use instance of the `GameModel`. Following the guidelines of the builder pattern, functionalities were designed to be as simple as possible, allowing the `GameModel` to be customized piece by piece.

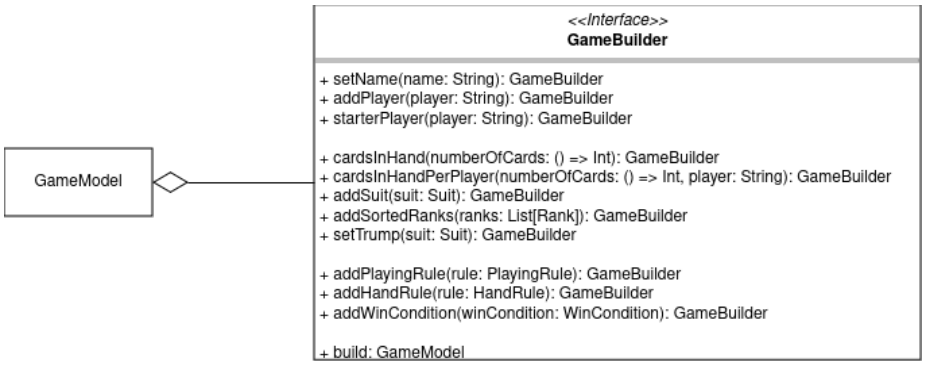

Figure 8: Game Builder

For this reason, the methods shown in the following UML diagram were created.

Important design notes:

- It is common for card games to deal the same number of cards to players, so the `cardsInHand` method was created to do just that.
- To avoid the complexity of defining an order among cards, it was decided to require pre-ordered ranks, explicitly addressed by the `addOrderedRanks` method.

CardGameEngineDSL

The design of the Domain-Specific Language (DSL) differs from the classic design associated with a class diagram. In this case, the design focused more on how to write certain sentences rather than how to organize methods and relationships between classes. The DSL does not have a mutable state; instead, it parses the expressed sentences and translates them into commands for the underlying builder.

Here is an example sentence from which the DSL keywords were derived:

```
game is "Simple Game"
game has player called "Filippo"
game has player called "Andrea"
game has player called "Leonardo"
game gives 10 cards to each player
game starts from player "Filippo"
game suitsAre ("Bastoni", "Denari", "Spade", "Coppe")
game ranksAre ("Asso", "2", "3", "4", "5", "6", "7", "Fante", "Cavallo", "Re")
game trumpIs "Bastoni"
```

Important design notes include:

- To enable the DSL to access the `GameBuilder`, its methods are defined as

extension methods of the builder.

- As extension methods of the builder, the first word of each DSL sentence must return the `GameBuilder`.
- To use infix notation and allow method calls without parentheses, methods must accept exactly one input parameter.
- The fluent interface pattern is used to enforce correct syntax.
- To satisfy the requirement of having exactly one input parameter for each method, objects were created to continue sentences.

Following these analyses, the DSL was designed as follows:

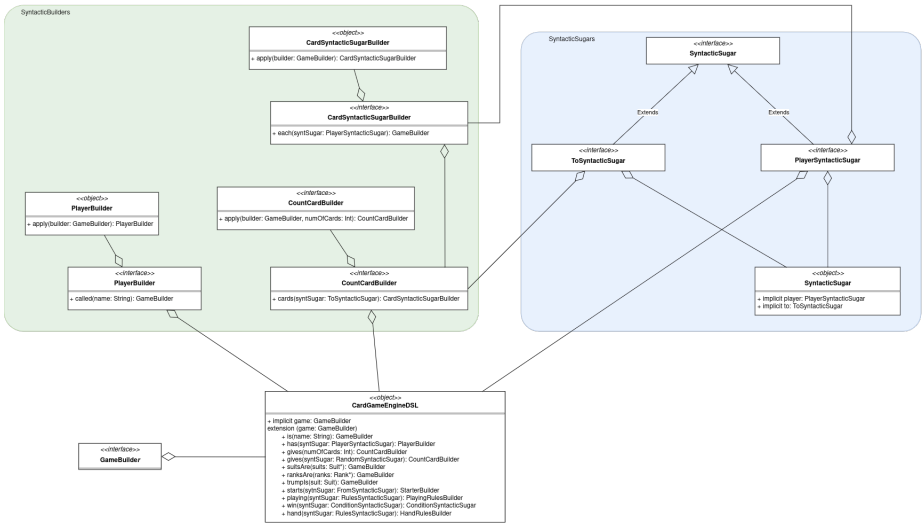

Figure 9: DSL

The diagram represents only the classes that form the sentences `game is "Simple Game"`, `game has player called "Filippo"`, and `game gives 5 cards to each player`. Subsequent sentences follow the same logic and are omitted for brevity and clarity.

For example, to form the sentence `game gives 5 cards to each player`, the user will execute the following calls:

- `game`: An implicit method within the DSL that returns the `GameBuilder`.
- `gives`: An extension method defined by the DSL for `GameBuilder` that returns a `CountCardBuilder`.
- `5`: The parameter for the `gives` method.
- `cards`: The only method of `CountCardBuilder` that returns a `CardSyntacticSugarBuilder` (while also setting the number of cards in the builder).
- `to`: An implicit value defined in the `SyntacticSugar` object and required parameter for the `cards` method.

- `each`: The only method of `CardSyntacticSugarBuilder` that returns the `GameBuilder`.
- `player`: An implicit value defined in the `SyntacticSugar` object and required parameter for the `each` method.

Conclusion

I believe the next sprint will involve refactoring the game builder to break it into smaller parts. Consequently, since the DSL maps to the builder, a similar solution would be applied to it as well.

Back to index | Back to implementation

Leonardo Randacio

I have worked indipendently on the View portion of the Engine.

The following is a detaild uml class diagram of the View:

Classes are colored based on the language used for implementation:

- blue : scala 3
- green : java 21

To implement the View using functional programming the State pattern is used.

State Pattern

The State implementation and it's dependencies are present in the `engine/view/monads` directory.

Facade

To use the Java Swing library a `SwingFunctionalFacade` is implemented.

To limit the volume of Java code necessary the facade implements only basic atomic methods that are then used by the scala view portion of the project.

WindowState

The `WindowState` maps one-to-one all the methods of the java facade into State updates.

It also defines a `Window` type alias that simply replaces the `Frame` type, native to Java Swing.

This object removes the possible dependency from the specific Java GUI library being used.

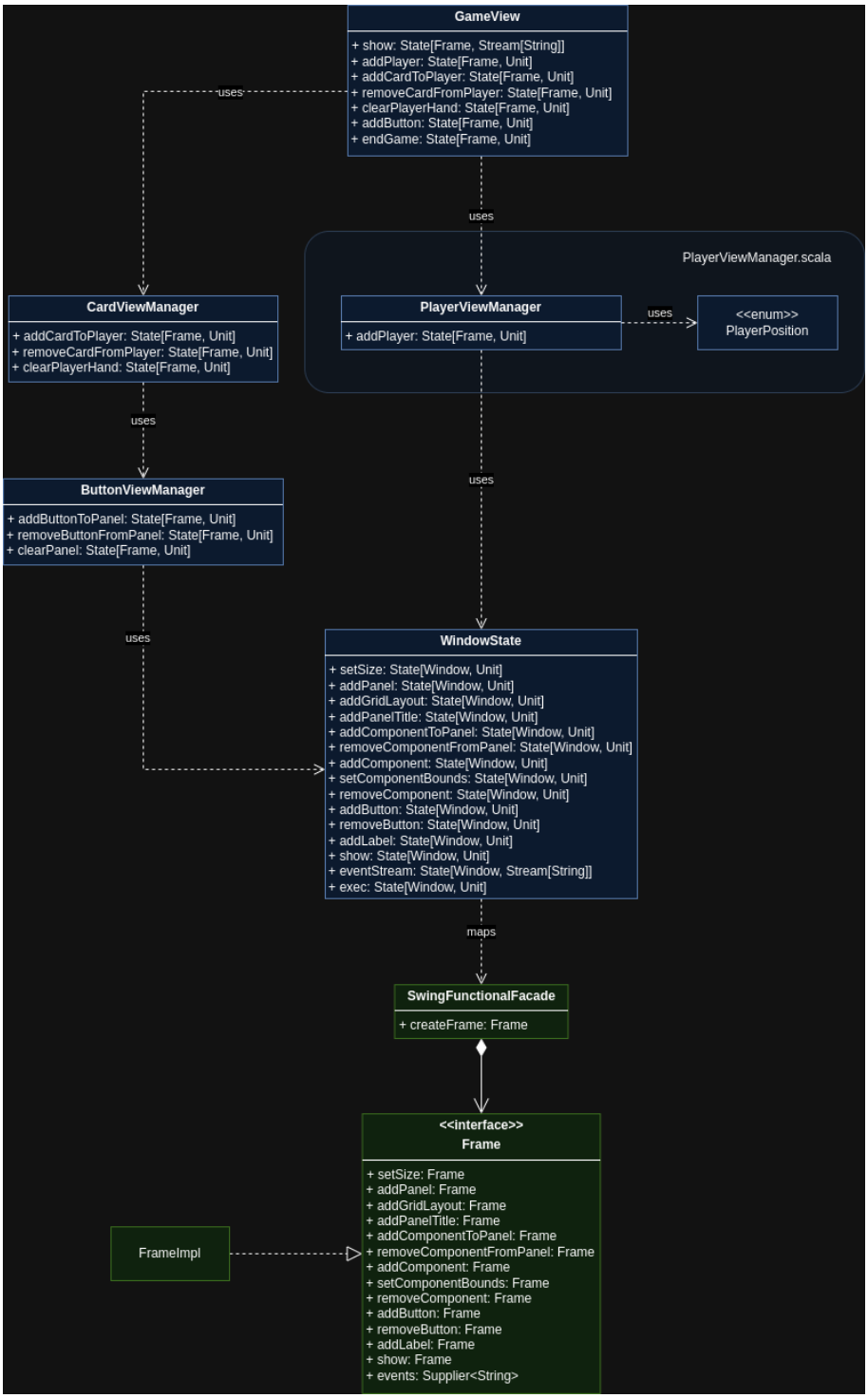

Figure 10: view

GameView

The `GameView` singleton defines high level methods that the Controller can use to generate updates to the View's State.

Every method returns a `State[Window, Unit]` so that it may be called by the Controller as part of a `for-yield` construct to compose the initial state of the View.

CardView

The cards are rappresented in the GUI using buttons.

If a card button is pressed it triggers an event structured as follows:

`"<playerName>:<rank> <suit>"`

Where:

- <playerName> is the name of the player that has that card in it's hand;
- <rank> is the rank of the card;
- <suit> is the suit of the card.

ButtonView

The `clearPanel` method uses the List.foldLeft() method to build a state update composed by states removing one-by-one very button present in the given panel.

A `val unitState(): State[Window, Unit]` is defined that rappresents a state that keeps track of the current window but does not apply any update to it, in other words it is a state that "does nothing".

```
def clearPanel(panelName: String): State[Window, Unit] =
  panelButtons.get(panelName) match
    case Some(buttons) =>
      panelButtons = panelButtons - panelName

      // Using List.foldLeft().
      buttons.foldLeft(unitState()): (acc, button) =>
        // Removing every button, prepending the state obtained by the previous button re
        for
          _ <- acc
          _ <- WindowState.removeComponentFromPanel(panelName, button)
          _ <- WindowState.removeButton(button)
        yield ()

    case None => unitState()
```

Back to index | Back to implementation

Testing

- Technology
- Coverage
- Metodology

The View has not been tested because of time restrains.

Technology

Scala Test has been used for unit testing.

Coverage

Using Scoverege the test coverage has been calculated to be 67.72%.

The team had set the goal to keep the test coverage of the project over 80% but due to time constrains this could not be achieved before the final deadline.

The main culprit of the low test coverage is the `GameController.scala` file, which has not been tested due to time constrains.

The full report can be generated by running the gradle task `checkScoverage`. The report will be located at `build/reports/scoverage`.

Metodology

Most of the source code has been developed using Test-Driven-Development.

Back to index | Previous Chapter | Next Chapter

Conclusion

- Source Code Future Improvments
- Git flow
- Agile
- TDD
- CI

Source Code Future Improvments

The second rule extension was only partially implemented due to time contrains.

The testing could be extended to comprehend the Controller and View modules of the project.

The current form of the game rules definition in the DSL are implemented as plain scala lambdas. This means that the user would have to understand the structure of the game Model in order to define rules. This could be improved

by implementing keywords for the most common rules present in card games enabeling the DSL to be more expressive and ensurng the user does not need to have knowledge of the engine's implementation.

Git flow

The team is happy of how git flow was used during project development although often feature branches where merged with the develop branch before the feature was complete due to misunderstandings on feature completenes between team members.

Agile

The team tried to adopt the SCRUM metodology since the start of the project. During the projects development the team noticed improvements in adhering to agile processes as time went by.

The project proved that understanding Agile development principles is not enough to put them in practice and much experience is necessary to fully adopt a SCRUM development metodology.

The project experience highlighted the need of a SCRUM Master.

Due to misunderstandings between team members some sprints ended with features half implemented, proving the need for frequent meetings between team members even during the sprint or for improved sprint planning.

TDD

The team tried to adhere to TDD during the project, although often it resulted in frustration as class and method structures varied often, especially during the first few sprints.

This highlighted the need for a deeper design phase and experience using all techonologies involved.

CI

Continuous Integration has been apprecieted by all team members as automating keeping track of test coverage and testing every release on every target platform proved very helpful and efficient, letting the team members focus on feature development.

Back to index | Previous Chapter